

The Architecture of Open Source Applications

The Architecture of Open Source Applications

Elegance, Evolution, and a Few Fearless Hacks

Edited by Amy Brown & Greg Wilson

The Architecture of Open Source Applications

Edited by Amy Brown and Greg Wilson

This work is licensed under the Creative Commons Attribution 3.0 Unported license (CC BY 3.0). You are free:

- to Share—to copy, distribute and transmit the work
- to Remix—to adapt the work

under the following conditions:

- Attribution—you must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

with the understanding that:

- Waiver—Any of the above conditions can be waived if you get permission from the copyright holder.
- Public Domain—Where the work or any of its elements is in the public domain under applicable law, that status is in no way affected by the license.
- Other Rights—In no way are any of the following rights affected by the license:
 - Your fair dealing or fair use rights, or other applicable copyright exceptions and limitations;
 - The author's moral rights;
 - Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights.
- Notice—For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to <http://creativecommons.org/licenses/by/3.0/>.

To view a copy of this license, visit <http://creativecommons.org/licenses/by/3.0/> or send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

The full text of this book is available online at <http://www.aosabook.org/>.

All royalties from its sale will be donated to Amnesty International.

Product and company names mentioned herein may be the trademarks of their respective owners.

While every precaution has been taken in the preparation of this book, the editors and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

The cover image is a photograph by Peter Dutton. The photograph is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 2.0 Generic license. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/2.0/> or send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

Revision Date: September 20, 2022

ISBN: 978-1-257-63801-7

Dedicated to Brian Kernighan,
who has taught us all so much;
and to prisoners of conscience everywhere.

Contents

Introduction	ix
<i>by Amy Brown and Greg Wilson</i>	
1 Asterisk	1
<i>by Russell Bryant</i>	
2 Audacity	15
<i>by James Crook</i>	
3 The Bourne-Again Shell	29
<i>by Chet Ramey</i>	
4 Berkeley DB	45
<i>by Margo Seltzer and Keith Bostic</i>	
5 CMake	67
<i>by Bill Hoffman and Kenneth Martin</i>	
6 Continuous Integration	77
<i>by C. Titus Brown and Rosangela Canino-Koning</i>	
7 Eclipse	89
<i>by Kim Moir</i>	
8 Graphite	111
<i>by Chris Davis</i>	
9 The Hadoop Distributed File System	121
<i>by Robert Chansler, Hairong Kuang, Sanjay Radia, Konstantin Shvachko, and Suresh Srinivas</i>	
10 Jitsi	135
<i>by Emil Ivov</i>	
11 LLVM	151
<i>by Chris Lattner</i>	

12	Mercurial <i>by Dirkjan Ochtman</i>	167
13	The NoSQL Ecosystem <i>by Adam Marcus</i>	181
14	Python Packaging <i>by Tarek Ziadé</i>	201
15	Riak and Erlang/OTP <i>by Francesco Cesarini, Andy Gross, and Justin Sheehy</i>	223
16	Selenium WebDriver <i>by Simon Stewart</i>	239
17	Sendmail <i>by Eric Allman</i>	265
18	SnowFlock <i>by Roy Bryant and Andrés Lagar-Cavilla</i>	285
19	SocialCalc <i>by Audrey Tang</i>	297
20	Telepathy <i>by Danielle Madeley</i>	317
21	Thousand Parsec <i>by Alan Laudicina and Aaron Mavrinac</i>	337
22	Violet <i>by Cay Horstmann</i>	353
23	VisTrails <i>by Juliana Freire, David Koop, Emanuele Santos, Carlos Scheidegger, Claudio Silva, and Huy T. Vo</i>	367
24	VTK <i>by Berk Geveci and Will Schroeder</i>	387
25	Battle for Wesnoth <i>by Richard Shimooka and David White</i>	403

Introduction

Amy Brown and Greg Wilson

Carpentry is an exacting craft, and people can spend their entire lives learning how to do it well. But carpentry is not architecture: if we step back from pitch boards and miter joints, buildings as a whole must be designed, and doing that is as much an art as it is a craft or science.

Programming is also an exacting craft, and people can spend their entire lives learning how to do it well. But programming is not software architecture. Many programmers spend years thinking about (or wrestling with) larger design issues: Should this application be extensible? If so, should that be done by providing a scripting interface, through some sort of plugin mechanism, or in some other way entirely? What should be done by the client, what should be left to the server, and is “client-server” even a useful way to think about this application? These are not programming questions, any more than where to put the stairs is a question of carpentry.

Building architecture and software architecture have a lot in common, but there is one crucial difference. While architects study thousands of buildings in their training and during their careers, most software developers only ever get to know a handful of large programs well. And more often than not, those are programs they wrote themselves. They never get to see the great programs of history, or read critiques of those programs’ designs written by experienced practitioners. As a result, they repeat one another’s mistakes rather than building on one another’s successes.

This book is our attempt to change that. Each chapter describes the architecture of an open source application: how it is structured, how its parts interact, why it’s built that way, and what lessons have been learned that can be applied to other big design problems. The descriptions are written by the people who know the software best, people with years or decades of experience designing and re-designing complex applications. The applications themselves range in scale from simple drawing programs and web-based spreadsheets to compiler toolkits and multi-million line visualization packages. Some are only a few years old, while others are approaching their thirtieth anniversary. What they have in common is that their creators have thought long and hard about their design, and are willing to share those thoughts with you. We hope you enjoy what they have written.

Contributors

Eric P. Allman (Sendmail): Eric Allman is the original author of sendmail, syslog, and trek, and the co-founder of Sendmail, Inc. He has been writing open source software since before it had a name, much less became a “movement”. He is a member of the *ACM Queue* Editorial Review Board and the Cal Performances Board of Trustees. His personal web site is <http://www.neophilic.com/~eric>.

Keith Bostic (Berkeley DB): Keith was a member of the University of California Berkeley Computer Systems Research Group, where he was the architect of the 2.10BSD release and a principal developer of 4.4BSD and related releases. He received the USENIX Lifetime Achievement Award (“The Flame”), which recognizes singular contributions to the Unix community, as well as a Distinguished Achievement Award from the University of California, Berkeley, for making the 4BSD release open source. Keith was the architect and one of the original developers of Berkeley DB, the open source embedded database system.

Amy Brown (editorial): Amy has a bachelor’s degree in Mathematics from the University of Waterloo, and worked in the software industry for ten years. She now writes and edits books, sometimes about software. She also sings and organizes other people’s lives—professionally and recreationally.

C. Titus Brown (Continuous Integration): Titus has worked in evolutionary modeling, physical meteorology, developmental biology, genomics, and bioinformatics. He is now an Assistant Professor at Michigan State University, where he has expanded his interests into several new areas, including reproducibility and maintainability of scientific software. He is also a member of the Python Software Foundation, and blogs at <http://ivory.idyll.org>.

Roy Bryant (Snowflock): In 20 years as a software architect and CTO, Roy designed systems including Electronics Workbench (now National Instruments’ Multisim) and the Linkwalker Data Pipeline, which won Microsoft’s worldwide Winning Customer Award for High-Performance Computing in 2006. After selling his latest startup, he returned to the University of Toronto to do graduate studies in Computer Science with a research focus on virtualization and cloud computing. Most recently, he published his Kaleidoscope extensions to Snowflock at ACM’s Eurosys Conference in 2011. His personal web site is <http://www.roybryant.net/>.

Russell Bryant (Asterisk): Russell is the Engineering Manager for the Open Source Software team at Digium, Inc. He has been a core member of the Asterisk development team since the Fall of 2004. He has since contributed to almost all areas of Asterisk development, from project management to core architectural design and development. He blogs at <http://www.russellbryant.net>.

Rosangela Canino-Koning (Continuous Integration): After 13 years of slogging in the software industry trenches, Rosangela returned to university to pursue a Ph.D. in Computer Science and Evolutionary Biology at Michigan State University. In her copious spare time, she likes to read, hike, travel, and hack on open source bioinformatics software. She blogs at <http://www.voidptr.net>.

Francesco Cesarini (Riak): Francesco Cesarini has used Erlang on a daily basis since 1995, having worked in various turnkey projects at Ericsson, including the OTP R1 release. He is the founder of Erlang Solutions and co-author of O’Reilly’s *Erlang Programming*. He currently works as Technical Director at Erlang Solutions, but still finds the time to teach graduates and undergraduates alike at Oxford University in the UK and the IT University of Gotheburg in Sweden.

Robert Chansler (HDFS): Robert is a Senior Manager for Software Development at Yahoo! After graduate studies in distributed systems at Carnegie-Mellon University, he worked on compilers (Tartan Labs), printing and imaging systems (Adobe Systems), electronic commerce (Adobe Systems, Impresse), and storage area network management (SanNavigator, McDATA). Returning to distributed systems and HDFS, Rob found many familiar problems, but all of the numbers had two or three more zeros.

James Crook (Audacity): James is a contract software developer based in Dublin, Ireland. Currently he is working on tools for electronics design, though in a previous life he developed bioinformatics software. He has many audacious plans for Audacity, and he hopes some, at least, will see the light of day.

Chris Davis (Graphite): Chris is a software consultant and Google engineer who has been designing and building scalable monitoring and automation tools for over 12 years. Chris originally wrote Graphite in 2006 and has lead the open source project ever since. When he's not writing code he enjoys cooking, making music, and doing research. His research interests include knowledge modeling, group theory, information theory, chaos theory, and complex systems.

Juliana Freire (VisTrails): Juliana is an Associate Professor of Computer Science at the University of Utah. Before that, she was member of technical staff at the Database Systems Research Department at Bell Laboratories (Lucent Technologies) and an Assistant Professor at OGI/OHSU. Her research interests include provenance, scientific data management, information integration, and Web mining. She is a recipient of an NSF CAREER and an IBM Faculty award. Her research has been funded by the National Science Foundation, Department of Energy, National Institutes of Health, IBM, Microsoft and Yahoo!

Berk Geveci (VTK): Berk is the Director of Scientific Computing at Kitware. He is responsible for leading the development effort of ParaView, an award-winning visualization application based on VTK. His research interests include large scale parallel computing, computational dynamics, finite elements and visualization algorithms.

Andy Gross (Riak): Andy Gross is Principal Architect at Basho Technologies, managing the design and development of Basho's Open Source and Enterprise data storage systems. Andy started at Basho in December of 2007 with 10 years of software and distributed systems engineering experience. Prior to Basho, Andy held senior distributed systems engineering positions at Mochi Media, Apple, Inc., and Akamai Technologies.

Bill Hoffman (CMake): Bill is CTO and co-Founder of Kitware, Inc. He is a key developer of the CMake project, and has been working with large C++ systems for over 20 years.

Cay Horstmann (Violet): Cay is a professor of computer science at San Jose State University, but every so often he takes a leave of absence to work in industry or teach in a foreign country. He is the author of many books on programming languages and software design, and the original author of the Violet and GridWorld open-source programs.

Emil Ivov (Jitsi): Emil is the founder and project lead of the Jitsi project (previously SIP Communicator). He is also involved with other initiatives like the ice4j.org and JAIN SIP projects. Emil obtained his Ph.D. from the University of Strasbourg in early 2008, and has been focusing primarily on Jitsi related activities ever since.

David Koop (VisTrails): David is a Ph.D. candidate in computer science at the University of Utah (finishing in the summer of 2011). His research interests include visualization, provenance, and scientific data management. He is a lead developer of the VisTrails system, and a senior software architect at VisTrails, Inc.

Hairong Kuang (HDFS) is a long time contributor and committer to the Hadoop project, which she has worked on passionately, currently at Facebook and previously at Yahoo! Prior to working in industry, she was an Assistant Professor at California State Polytechnic University, Pomona. She received a Ph.D. in Computer Science from the University of California at Irvine. Her interests include cloud computing, mobile agents, parallel computing, and distributed systems.

H. Andrés Lagar-Cavilla (Snowflock): Andrés is a software systems researcher who does experimental work on virtualization, operating systems, security, cluster computing, and mobile computing. He has a B.A.Sc. from Argentina, and an M.Sc. and Ph.D. in Computer Science from University of Toronto, and can be found online at <http://lagarcavilla.org>.

Chris Lattner (LLVM): Chris is a software developer with a diverse range of interests and experiences, particularly in the area of compiler tool chains, operating systems, graphics and image rendering. He is the designer and lead architect of the Open Source LLVM Project. See <http://>

[//nondot.org/~sabre/](http://nondot.org/~sabre/) for more about Chris and his projects.

Alan Laudicina (Thousand Parsec): Alan is an M.Sc. student in computer science at Wayne State University, where he studies distributed computing. In his spare time he codes, learns programming languages, and plays poker. You can find more about him at <http://alanp.ca/>.

Danielle Madeley (Telepathy): Danielle is an Australian software engineer working on Telepathy and other magic for Collabora Ltd. She has bachelor's degrees in electronic engineering and computer science. She also has an extensive collection of plush penguins. She blogs at <http://blogs.gnome.org/danni/>.

Adam Marcus (NoSQL): Adam is a Ph.D. student focused on the intersection of database systems and social computing at MIT's Computer Science and Artificial Intelligence Lab. His recent work ties traditional database systems to social streams such as Twitter and human computation platforms such as Mechanical Turk. He likes to build usable open source systems from his research prototypes, and prefers tracking open source storage systems to long walks on the beach. He blogs at <http://blog.marcua.net>.

Kenneth Martin (CMake): Ken is currently Chairman and CFO of Kitware, Inc., a research and development company based in the US. He co-founded Kitware in 1998 and since then has helped grow the company to its current position as a leading R&D provider with clients across many government and commercial sectors.

Aaron Mavrinac (Thousand Parsec): Aaron is a Ph.D. candidate in electrical and computer engineering at the University of Windsor, researching camera networks, computer vision, and robotics. When there is free time, he fills some of it working on Thousand Parsec and other free software, coding in Python and C, and doing too many other things to get good at any of them. His web site is <http://www.mavrinac.com>.

Kim Moir (Eclipse): Kim works at the IBM Rational Software lab in Ottawa as the Release Engineering lead for the Eclipse and Runtime Equinox projects and is a member of the Eclipse Architecture Council. Her interests lie in build optimization, Equinox and building component based software. Outside of work she can be found hitting the pavement with her running mates, preparing for the next road race. She blogs at <http://relengofthenerds.blogspot.com/>.

Dirkjan Ochtman (Mercurial): Dirkjan graduated as a Master in CS in 2010, and has been working at a financial startup for 3 years. When not procrastinating in his free time, he hacks on Mercurial, Python, Gentoo Linux and a Python CouchDB library. He lives in the beautiful city of Amsterdam. His personal web site is <http://dirkjan.ochtman.nl/>.

Sanjay Radia (HDFS): Sanjay is the architect of the Hadoop project at Yahoo!, and a Hadoop committer and Project Management Committee member at the Apache Software Foundation. Previously he held senior engineering positions at Cassatt, Sun Microsystems and INRIA where he developed software for distributed systems and grid/utility computing infrastructures. Sanjay has a Ph.D. in Computer Science from University of Waterloo, Canada.

Chet Ramey (Bash): Chet has been involved with bash for more than twenty years, the past seventeen as primary developer. He is a longtime employee of Case Western Reserve University in Cleveland, Ohio, from which he received his B.Sc. and M.Sc. degrees. He lives near Cleveland with his family and pets, and can be found online at <http://tiswww.cwru.edu/~chet>.

Emanuele Santos (VisTrails): Emanuele is a research scientist at the University of Utah. Her research interests include scientific data management, visualization, and provenance. She received her Ph.D. in Computing from the University of Utah in 2010. She is also a lead developer of the VisTrails system.

Carlos Scheidegger (VisTrails): Carlos has a Ph.D. in Computing from the University of Utah, and is now a researcher at AT&T Labs–Research. Carlos has won best paper awards at IEEE Visualization

in 2007, and Shape Modeling International in 2008. His research interests include data visualization and analysis, geometry processing and computer graphics.

Will Schroeder (VTK): Will is President and co-Founder of Kitware, Inc. He is a computational scientist by training and has been one of the key developers of VTK. He enjoys writing beautiful code, especially when it involves computational geometry or graphics.

Margo Seltzer (Berkeley DB): Margo is the Herchel Smith Professor of Computer Science at Harvard's School of Engineering and Applied Sciences and an Architect at Oracle Corporation. She was one of the principal designers of Berkeley DB and a co-founder of Sleepycat Software. Her research interests are in filesystems, database systems, transactional systems, and medical data mining. Her professional life is online at <http://www.eecs.harvard.edu/~margo>, and she blogs at <http://mis-misinformation.blogspot.com/>.

Justin Sheehy (Riak): Justin is the CTO of Basho Technologies, the company behind the creation of Webmachine and Riak. Most recently before Basho, he was a principal scientist at the MITRE Corporation and a senior architect for systems infrastructure at Akamai. At both of those companies he focused on multiple aspects of robust distributed systems, including scheduling algorithms, language-based formal models, and resilience.

Richard Shimooka (Battle for Wesnoth): Richard is a Research Associate at Queen's University's Defence Management Studies Program in Kingston, Ontario. He is also a Deputy Administrator and Secretary for the Battle for Wesnoth. Richard has written several works examining the organizational cultures of social groups, ranging from governments to open source projects.

Konstantin V. Shvachko (HDFS), a veteran HDFS developer, is a principal Hadoop architect at eBay. Konstantin specializes in efficient data structures and algorithms for large-scale distributed storage systems. He discovered a new type of balanced trees, S-trees, for optimal indexing of unstructured data, and was a primary developer of an S-tree-based Linux filesystem, treeFS, a prototype of reiserFS. Konstantin holds a Ph.D. in computer science from Moscow State University, Russia. He is also a member of the Project Management Committee for Apache Hadoop.

Claudio Silva (VisTrails): Claudio is a full professor of computer science at the University of Utah. His research interests are in visualization, geometric computing, computer graphics, and scientific data management. He received his Ph.D. in computer science from the State University of New York at Stony Brook in 1996. Later in 2011, he will be joining the Polytechnic Institute of New York University as a full professor of computer science and engineering.

Suresh Srinivas (HDFS): Suresh works on HDFS as a software architect at Yahoo! He is a Hadoop committer and PMC member at Apache Software Foundation. Prior to Yahoo!, he worked at Sylanro Systems, developing scalable infrastructure for hosted communication services. Suresh has a bachelor's degree in Electronics and Communication from National Institute of Technology Karnataka, India.

Simon Stewart (Selenium): Simon lives in London and works as a Software Engineer in Test at Google. He is a core contributor to the Selenium project, was the creator of WebDriver and is enthusiastic about open source. Simon enjoys beer and writing better software, sometimes at the same time. His personal home page is <http://www.pubbitch.org/>.

Audrey Tang (SocialCalc): Audrey is a self-educated programmer and translator based in Taiwan. She currently works at Socialtext, where her job title is "Untitled Page", as well as at Apple as contractor for localization and release engineering. She previously designed and led the Pugs project, the first working Perl 6 implementation; she has also served in language design committees for Haskell, Perl 5, and Perl 6, and has made numerous contributions to CPAN and Hackage. She blogs at <http://pugs.blogs.com/audreyt/>.

Huy T. Vo (VisTrails): Huy is receiving his Ph.D. from the University of Utah in May 2011. His research interests include visualization, dataflow architecture and scientific data management. He is a senior developer at VisTrails, Inc. He also holds a Research Assistant Professor appointment with the Polytechnic Institute of New York University.

David White (Battle for Wesnoth): David is the founder and lead developer of Battle for Wesnoth. David has been involved with several Open Source video game projects, including Frogatto which he also co-founded. David is a performance engineer at Sabre Holdings, a leader in travel technology.

Greg Wilson (editorial): Greg has worked over the past 25 years in high-performance scientific computing, data visualization, and computer security, and is the author or editor of several computing books (including the 2008 Jolt Award winner *Beautiful Code*) and two books for children. Greg received a Ph.D. in Computer Science from the University of Edinburgh in 1993.

Tarek Ziadé (Python Packaging): Tarek lives in Burgundy, France. He's a Senior Software Engineer at Mozilla, building servers in Python. In his spare time, he leads the packaging effort in Python.

Acknowledgments

We would like to thank our reviewers:

Eric Aderhold	Muhammad Ali	Lillian Angel
Robert Beghian	Taavi Burns	Luis Pedro Coelho
David Cooper	Mauricio de Simone	Jonathan Deber
Patrick Dubroy	Igor Foox	Alecia Fowler
Marcus Hanwell	Johan Harjono	Vivek Lakshmanan
Greg Lapouchnian	Laurie MacDougall Sookraj	Josh McCarthy
Jason Montojo	Colin Morris	Christian Muise
Victor Ng	Nikita Pchelin	Andrew Petersen
Andrey Petrov	Tom Plaskon	Pascal Rapiçault
Todd Ritchie	Samar Sabie	Misa Sakamoto
David Scannell	Clara Severino	Tim Smith
Kyle Spaans	Sana Tapal	Tony Targonski
Miles Thibault	David Wright	Tina Yee

We would also like to thank Jackie Carter, who helped with the early stages of editing.

Contributing

Dozens of volunteers worked hard to create this book, but there is still lots to do. You can help by reporting errors, by helping to translate the content into other languages, or by describing the architecture of other open source projects. Please contact us at aosa@aosabook.org if you would like to get involved.

Asterisk

Russell Bryant

Asterisk¹ is an open source telephony applications platform distributed under the GPLv2. In short, it is a server application for making, receiving, and performing custom processing of phone calls.

The project was started by Mark Spencer in 1999. Mark had a company called Linux Support Services and he needed a phone system to help operate his business. He did not have a lot of money to spend on buying one, so he just made his own. As the popularity of Asterisk grew, Linux Support Services shifted focus to Asterisk and changed its name to Digium, Inc.

The name Asterisk comes from the Unix wildcard character, *. The goal for the Asterisk project is to do everything telephony. Through pursuing this goal, Asterisk now supports a long list of technologies for making and receiving phone calls. This includes many VoIP (Voice over IP) protocols, as well as both analog and digital connectivity to the traditional telephone network, or the PSTN (Public Switched Telephone Network). This ability to get many different types of phone calls into and out of the system is one of Asterisk's main strengths.

Once phone calls are made to and from an Asterisk system, there are many additional features that can be used to customize the processing of the phone call. Some features are larger pre-built common applications, such as voicemail. There are other smaller features that can be combined together to create custom voice applications, such as playing back a sound file, reading digits, or speech recognition.

1.1 Critical Architectural Concepts

This section discusses some architectural concepts that are critical to all parts of Asterisk. These ideas are at the foundation of the Asterisk architecture.

Channels

A channel in Asterisk represents a connection between the Asterisk system and some telephony endpoint (Figure 1.1). The most common example is when a phone makes a call into an Asterisk system. This connection is represented by a single channel. In the Asterisk code, a channel exists as

¹<http://www.asterisk.org/>

an instance of the `ast_channel` data structure. This call scenario could be a caller interacting with voicemail, for example.

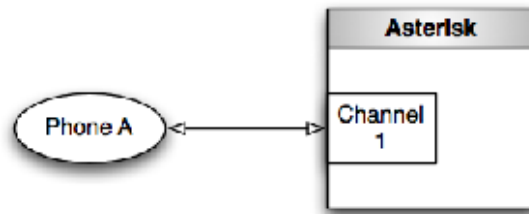


Figure 1.1: A Single Call Leg, Represented by a Single Channel

Channel Bridging

Perhaps a more familiar call scenario would be a connection between two phones, where a person using phone A has called a person on phone B. In this call scenario, there are two telephony endpoints connected to the Asterisk system, so two channels exist for this call (Figure 1.2).

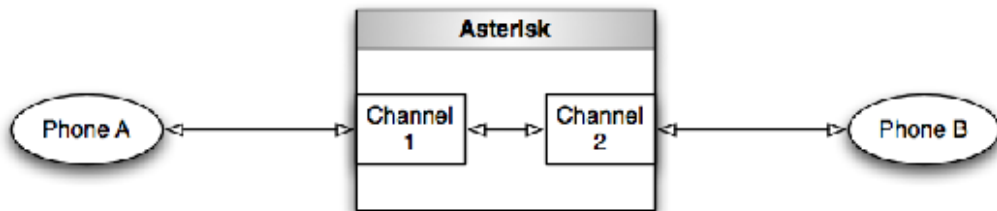


Figure 1.2: Two Call Legs Represented by Two Channels

When Asterisk channels are connected like this, it is referred to as a channel bridge. Channel bridging is the act of connecting channels together for the purpose of passing media between them. The media stream is most commonly an audio stream. However, there may also be a video or a text stream in the call. Even in the case where there is more than one media stream (such as both audio and video), it is still handled by a single channel for each end of the call in Asterisk. In Figure 1.2, where there are two channels for phones A and B, the bridge is responsible for passing the media coming from phone A to phone B, and similarly, for passing the media coming from phone B to phone A. All media streams are negotiated through Asterisk. Anything that Asterisk does not understand and have full control over is not allowed. This means that Asterisk can do recording, audio manipulation, and translation between different technologies.

When two channels are bridged together, there are two methods that may be used to accomplish this: generic bridging and native bridging. A generic bridge is one that works regardless of what channel technologies are in use. It passes all audio and signalling through the Asterisk abstract

channel interfaces. While this is the most flexible bridging method, it is also the least efficient due to the levels of abstraction necessary to accomplish the task. Figure 1.2 illustrates a generic bridge.

A native bridge is a technology specific method of connecting channels together. If two channels are connected to Asterisk using the same media transport technology, there may be a way to connect them that is more efficient than going through the abstraction layers in Asterisk that exist for connecting different technologies together. For example, if specialized hardware is being used for connecting to the telephone network, it may be possible to bridge the channels on the hardware so that the media does not have to flow up through the application at all. In the case of some VoIP protocols, it is possible to have endpoints send their media streams to each other directly, such that only the call signalling information continues to flow through the server.

The decision between generic bridging and native bridging is done by comparing the two channels when it is time to bridge them. If both channels indicate that they support the same native bridging method, then that will be used. Otherwise, the generic bridging method will be used. To determine whether or not two channels support the same native bridging method, a simple C function pointer comparison is used. It's certainly not the most elegant method, but we have not yet hit any cases where this was not sufficient for our needs. Providing a native bridge function for a channel is discussed in more detail in Section 1.2. Figure 1.3 illustrates an example of a native bridge.

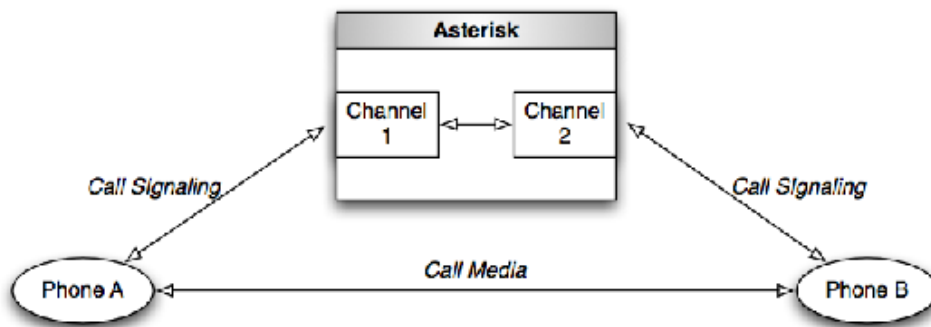


Figure 1.3: Example of a Native Bridge

Frames

Communication within the Asterisk code during a call is done by using frames, which are instances of the `ast_frame` data structure. Frames can either be media frames or signalling frames. During a basic phone call, a stream of media frames containing audio would be passing through the system. Signalling frames are used to send messages about call signalling events, such as a digit being pressed, a call being put on hold, or a call being hung up.

The list of available frame types is statically defined. Frames are marked with a numerically encoded type and subtype. A full list can be found in the source code in `include/asterisk/frame.h`; some examples are:

- VOICE: These frames carry a portion of an audio stream.
- VIDEO: These frames carry a portion of a video stream.

- **MODEM:** The encoding used for the data in this frame, such as T.38 for sending a FAX over IP. The primary usage of this frame type is for handling a FAX. It is important that frames of data be left completely undisturbed so that the signal can be successfully decoded at the other end. This is different than **AUDIO** frames, because in that case, it is acceptable to transcode into other audio codecs to save bandwidth at the cost of audio quality.
- **CONTROL:** The call signalling message that this frame indicates. These frames are used to indicate call signalling events. These events include a phone being answered, hung up, put on hold, etc.
- **DTMF_BEGIN:** Which digit just started. This frame is sent when a caller presses a DTMF key² on their phone.
- **DTMF_END:** Which digit just ended. This frame is sent when a caller stops pressing a DTMF key on their phone.

1.2 Asterisk Component Abstractions

Asterisk is a highly modularized application. There is a core application that is built from the source in the `main/` directory of the source tree. However, it is not very useful by itself. The core application acts primarily as a module registry. It also has code that knows how to connect all of the abstract interfaces together to make phone calls work. The concrete implementations of these interfaces are registered by loadable modules at runtime.

By default, all modules found in a predefined Asterisk modules directory on the filesystem will be loaded when the main application is started. This approach was chosen for its simplicity. However, there is a configuration file that can be updated to specify exactly which modules to load and in what order to load them. This makes the configuration a bit more complex, but provides the ability to specify that modules that are not needed should not be loaded. The primary benefit is reducing the memory footprint of the application. However, there are some security benefits, as well. It is best not to load a module that accepts connections over a network if it is not actually needed.

When the module loads, it registers all of its implementations of component abstractions with the Asterisk core application. There are many types of interfaces that modules can implement and register with the Asterisk core. A module is allowed to register as many of these different interfaces as it would like. Generally, related functionality is grouped into a single module.

Channel Drivers

The Asterisk channel driver interface is the most complex and most important interface available. The Asterisk channel API provides the telephony protocol abstraction which allows all other Asterisk features to work independently of the telephony protocol in use. This component is responsible for translating between the Asterisk channel abstraction and the details of the telephony technology that it implements.

The definition of the Asterisk channel driver interface is called the `ast_channel_tech` interface. It defines a set of methods that must be implemented by a channel driver. The first method that a channel driver must implement is an `ast_channel` factory method, which is the `requester` method in `ast_channel_tech`. When an Asterisk channel is created, either for an incoming or outgoing

²DTMF stands for Dual-Tone Multi-Frequency. This is the tone that is sent in the audio of a phone call when someone presses a key on their telephone.

phone call, the implementation of `ast_channel_tech` associated with the type of channel needed is responsible for instantiation and initialization of the `ast_channel` for that call.

Once an `ast_channel` has been created, it has a reference to the `ast_channel_tech` that created it. There are many other operations that must be handled in a technology-specific way. When those operations must be performed on an `ast_channel`, the handling of the operation is deferred to the appropriate method from `ast_channel_tech`. Figure 1.2 shows two channels in Asterisk. Figure 1.4 expands on this to show two bridged channels and how the channel technology implementations fit into the picture.

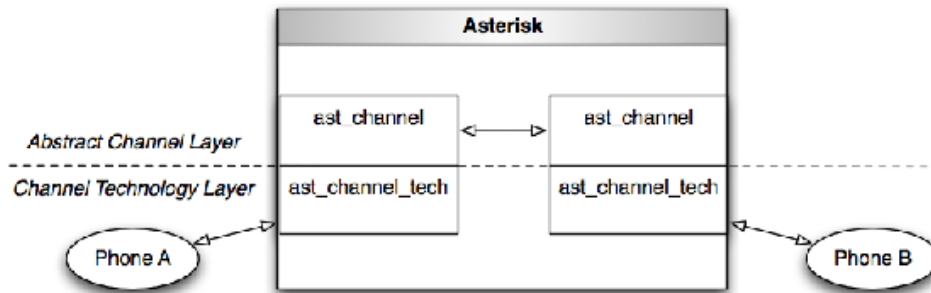


Figure 1.4: Channel Technology and Abstract Channel Layers

The most important methods in `ast_channel_tech` are:

- **requester:** This callback is used to request a channel driver to instantiate an `ast_channel` object and initialize it as appropriate for this channel type.
- **call:** This callback is used to initiate an outbound call to the endpoint represented by an `ast_channel`.
- **answer:** This is called when Asterisk decides that it should answer the inbound call associated with this `ast_channel`.
- **hangup:** This is called when the system has determined that the call should be hung up. The channel driver will then communicate to the endpoint that the call is over in a protocol specific manner.
- **indicate:** Once a call is up, there are a number of other events that may occur that need to be signalled to an endpoint. For example, if the device is put on hold, this callback is called to indicate that condition. There may be a protocol specific method of indicating that the call has been on hold, or the channel driver may simply initiate the playback of music on hold to the device.
- **send_digit_begin:** This function is called to indicate the beginning of a digit (DTMF) being sent to this device.
- **send_digit_end:** This function is called to indicate the end of a digit (DTMF) being sent to this device.
- **read:** This function is called by the Asterisk core to read back an `ast_frame` from this endpoint. An `ast_frame` is an abstraction in Asterisk that is used to encapsulate media (such as audio or video), as well as to signal events.

- **write:** This function is used to send an `ast_frame` to this device. The channel driver will take the data and packetize it as appropriate for the telephony protocol that it implements and pass it along to the endpoint.
- **bridge:** This is the native bridge callback for this channel type. As discussed before, native bridging is when a channel driver is able to implement a more efficient bridging method for two channels of the same type instead of having all signalling and media flow through additional unnecessary abstraction layers. This is incredibly important for performance reasons.

Once a call is over, the abstract channel handling code that lives in the Asterisk core will invoke the `ast_channel_tech_hangup` callback and then destroy the `ast_channel` object.

Dialplan Applications

Asterisk administrators set up call routing using the Asterisk dialplan, which resides in the `/etc/asterisk/extensions.conf` file. The dialplan is made up of a series of call rules called extensions. When a phone call comes in to the system, the dialed number is used to find the extension in the dialplan that should be used for processing the call. The extension includes a list of dialplan applications which will be executed on the channel. The applications available for execution in the dialplan are maintained in an application registry. This registry is populated at runtime as modules are loaded.

Asterisk has nearly two hundred included applications. The definition of an application is very loose. Applications can use any of the Asterisk internal APIs to interact with the channel. Some applications do a single task, such as `Playback`, which plays back a sound file to the caller. Other applications are much more involved and perform a large number of operations, such as the `Voicemail` application.

Using the Asterisk dialplan, multiple applications can be used together to customize call handling. If more extensive customization is needed beyond what is possible in the provided dialplan language, there are scripting interfaces available that allow call handling to be customized using any programming language. Even when using these scripting interfaces with another programming language, dialplan applications are still invoked to interact with the channel.

Before we get into an example, let's have a look at the syntax of an Asterisk dialplan that handles calls to the number 1234. Note that the choice of 1234 here is arbitrary. It invokes three dialplan applications. First, it answers the call. Next, it plays back a sound file. Finally, it hangs up the call.

```
; Define the rules for what happens when someone dials 1234.
;
exten => 1234,1,Answer()
    same => n,Playback(demo-congrats)
    same => n,Hangup()
```

The `exten` keyword is used to define the extension. On the right side of the `exten` line, the 1234 means that we are defining the rules for when someone calls 1234. The next 1 means this is the first step that is taken when that number is dialed. Finally, `Answer` instructs the system to answer the call. The next two lines that begin with the `same` keyword are rules for the last extension that was specified, which in this case is 1234. The `n` is short for saying that this is the next step to take. The last item on those lines specifies what action to take.

Here is another example of using the Asterisk dialplan. In this case, an incoming call is answered. The caller is played a beep, and then up to 4 digits are read from the caller and stored into the `DIGITS` variable. Then, the digits are read back to the caller. Finally, the call is ended.

```

exten => 5678,1,Answer()
      same => n,Read(DIGITS,beep,4)
      same => n,SayDigits(${DIGITS})
      same => n,Hangup()

```

As previously mentioned, the definition of an application is very loose—the function prototype registered is very simple:

```
int (*execute)(struct ast_channel *chan, const char *args);
```

However, the application implementations use virtually all of the APIs found in `include/asterisk/`.

Dialplan Functions

Most dialplan applications take a string of arguments. While some values may be hard coded, variables are used in places where behavior needs to be more dynamic. The following example shows a dialplan snippet that sets a variable and then prints out its value to the Asterisk command line interface using the Verbose application.

```

exten => 1234,1,Set(MY_VARIABLE=foo)
      same => n,Verbose(MY_VARIABLE is ${MY_VARIABLE})

```

Dialplan functions are invoked by using the same syntax as the previous example. Asterisk modules are able to register dialplan functions that can retrieve some information and return it to the dialplan. Alternatively, these dialplan functions can receive data from the dialplan and act on it. As a general rule, while dialplan functions may set or retrieve channel meta data, they do not do any signalling or media processing. That is left as the job of dialplan applications.

The following example demonstrates usage of a dialplan function. First, it prints out the CallerID of the current channel to the Asterisk command line interface. Then, it changes the CallerID by using the Set application. In this example, Verbose and Set are applications, and CALLERID is a function.

```

exten => 1234,1,Verbose(The current CallerID is ${CALLERID(num)})
      same => n,Set(CALLERID(num)=<256>555-1212)

```

A dialplan function is needed here instead of just a simple variable since the CallerID information is stored in data structures on the instance of `ast_channel`. The dialplan function code knows how to set and retrieve the values from these data structures.

Another example of using a dialplan function is for adding custom information into the call logs, which are referred to as CDRs (Call Detail Records). The CDR function allows the retrieval of call detail record information, as well as adding custom information.

```

exten => 555,1,Verbose(Time this call started: ${CDR(start)})
      same => n,Set(CDR(mycustomfield)=snickerdoodle)

```

Codec Translators

In the world of VOIP, many different codecs are used for encoding media to be sent across networks. The variety of choices offers tradeoffs in media quality, CPU consumption, and bandwidth requirements. Asterisk supports many different codecs and knows how to translate between them when necessary.

When a call is set up, Asterisk will attempt to get two endpoints to use a common media codec so that transcoding is not required. However, that is not always possible. Even if a common codec is being used, transcoding may still be required. For example, if Asterisk is configured to do some signal processing on the audio as it passes through the system (such as to increase or decrease the volume level), Asterisk will need to transcode the audio back to an uncompressed form before it can perform the signal processing. Asterisk can also be configured to do call recording. If the configured format for the recording is different than that of the call, transcoding will be required.

Codec Negotiation

The method used to negotiate which codec will be used for a media stream is specific to the technology used to connect the call to Asterisk. In some cases, such as a call on the traditional telephone network (the PSTN), there may not be any negotiation to do. However, in other cases, especially using IP protocols, there is a negotiation mechanism used where capabilities and preferences are expressed and a common codec is agreed upon.

For example, in the case of SIP (the most commonly used VOIP protocol) this is a high level view of how codec negotiation is performed when a call is sent to Asterisk.

1. An endpoint sends a new call request to Asterisk which includes the list of codecs it is willing to use.
2. Asterisk consults its configuration provided by the administrator which includes a list of allowed codecs in preferred order. Asterisk will respond by choosing the most preferred codec (based on its own configured preferences) that is listed as allowed in the Asterisk configuration and was also listed as supported in the incoming request.

One area that Asterisk does not handle very well is that of more complex codecs, especially video. Codec negotiation demands have gotten more complicated over the last ten years. We have more work to do to be able to better deal with the newest audio codecs and to be able to support video much better than we do today. This is one of the top priorities for new development for the next major release of Asterisk.

Codec translator modules provide one or more implementations of the `ast_translator` interface. A translator has source and destination format attributes. It also provides a callback that will be used to convert a chunk of media from the source to the destination format. It knows nothing about the concept of a phone call. It only knows how to convert media from one format to another.

For more detailed information about the translator API, see `include/asterisk/translate.h` and `main/translate.c`. Implementations of the translator abstraction can be found in the codecs directory.

1.3 Threads

Asterisk is a very heavily multithreaded application. It uses the POSIX threads API to manage threads and related services such as locking. All of the Asterisk code that interacts with threads does so by going through a set of wrappers used for debugging purposes. Most threads in Asterisk can be classified as either a Network Monitor Thread, or a Channel Thread (sometimes also referred to as a PBX thread, because its primary purpose is to run the PBX for a channel).

Network Monitor Threads

Network monitor threads exist in every major channel driver in Asterisk. They are responsible for monitoring whatever network they are connected to (whether that is an IP network, the PSTN, etc.) and monitor for incoming calls or other types of incoming requests. They handle the initial connection setup steps such as authentication and dialed number validation. Once the call setup has been completed, the monitor threads will create an instance of an Asterisk channel (`ast_channel`), and start a channel thread to handle the call for the rest of its lifetime.

Channel Threads

As discussed earlier, a channel is a fundamental concept in Asterisk. Channels are either inbound or outbound. An inbound channel is created when a call comes in to the Asterisk system. These channels are the ones that execute the Asterisk dialplan. A thread is created for every inbound channel that executes the dialplan. These threads are referred to as channel threads.

Dialplan applications always execute in the context of a channel thread. Dialplan functions *almost* always do, as well. It is possible to read and write dialplan functions from an asynchronous interface such as the Asterisk CLI. However, it is still always the channel thread that is the owner of the `ast_channel` data structure and controls the object lifetime.

1.4 Call Scenarios

The previous two sections introduced important interfaces for Asterisk components, as well as the thread execution model. In this section, some common call scenarios are broken down to demonstrate how Asterisk components operate together to process phone calls.

Checking Voicemail

One example call scenario is when someone calls into the phone system to check their Voicemail. The first major component involved in this scenario is the channel driver. The channel driver will be responsible for handling the incoming call request from the phone, which will occur in the channel driver's monitor thread. Depending on the telephony technology being used to deliver the call to the system, there may be some sort of negotiation required to set up the call. Another step of setting up the call is determining the intended destination for the call. This is usually specified by the number that was dialed by the caller. However, in some cases there is no specific number available since the technology used to deliver the call does not support specifying the dialed number. An example of this would be an incoming call on an analog phone line.

If the channel driver verifies that the Asterisk configuration has extensions defined in the dialplan (the call routing configuration) for the dialed number, it will then allocate an Asterisk channel object

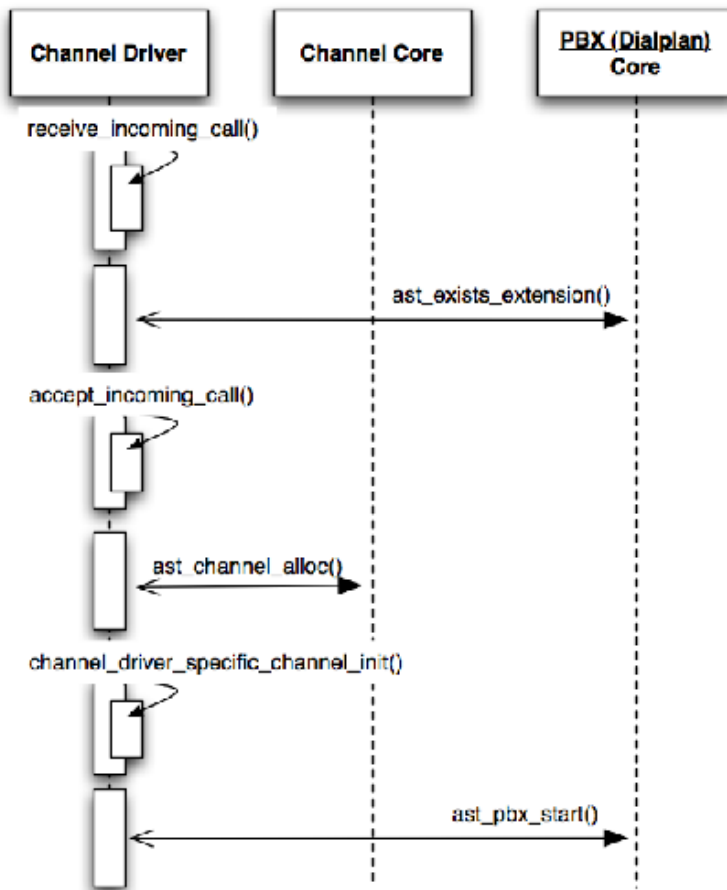


Figure 1.5: Call Setup Sequence Diagram

(`ast_channel`) and create a channel thread. The channel thread has the primary responsibility for handling the rest of the call (Figure 1.5).

The main loop of the channel thread handles dialplan execution. It goes to the rules defined for the dialed extension and executes the steps that have been defined. The following is an example extension expressed in the `extensions.conf` dialplan syntax. This extension answers the call and executes the VoicemailMain application when someone dials *123. This application is what a user would call to be able to check messages left in their mailbox.

```

exten => *123,1,Answer()
    same => n,VoiceMailMain()

```

When the channel thread executes the Answer application, Asterisk will answer the incoming call. Answering a call requires technology specific processing, so in addition to some generic answer handling, the answer callback in the associated `ast_channel_tech` structure is called to handle

answering the call. This may involve sending a special packet over an IP network, taking an analog line off hook, etc.

The next step is for the channel thread to execute `VoicemailMain` (Figure 1.6). This application is provided by the `app_voicemail` module. One important thing to note is that while the Voicemail code handles a lot of call interaction, it knows nothing about the technology that is being used to deliver the call into the Asterisk system. The Asterisk channel abstraction hides these details from the Voicemail implementation.

There are many features involved in providing a caller access to their Voicemail. However, all of them are primarily implemented as reading and writing sound files in response to input from the caller, primarily in the form of digit presses. DTMF digits can be delivered to Asterisk in many different ways. Again, these details are handled by the channel drivers. Once a key press has arrived in Asterisk, it is converted into a generic key press event and passed along to the Voicemail code.

One of the important interfaces in Asterisk that has been discussed is that of a codec translator. These codec implementations are very important to this call scenario. When the Voicemail code would like to play back a sound file to the caller, the format of the audio in the sound file may not be the same format as the audio being used in the communication between the Asterisk system and the caller. If it must transcode the audio, it will build a translation path of one or more codec translators to get from the source to the destination format.

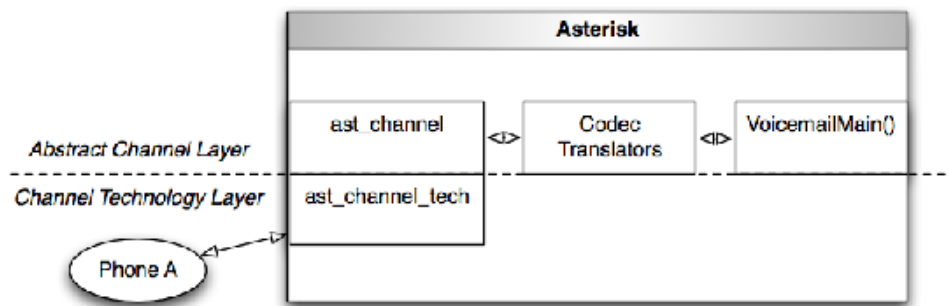


Figure 1.6: A Call to `VoicemailMain`

At some point, the caller will be done interacting with the Voicemail system and hang up. The channel driver will detect that this has occurred and convert this into a generic Asterisk channel signalling event. The Voicemail code will receive this signalling event and will exit, since there is nothing left to do once the caller hangs up. Control will return back to the main loop in the channel thread to continue dialplan execution. Since in this example there is no further dialplan processing to be done, the channel driver will be given an opportunity to handle technology specific hangup processing and then the `ast_channel` object will be destroyed.

Bridged Call

Another very common call scenario in Asterisk is a bridged call between two channels. This is the scenario when one phone calls another through the system. The initial call setup process is identical to the previous example. The difference in handling begins when the call has been set up and the channel thread begins executing the dialplan.

The following dialplan is a simple example that results in a bridged call. Using this extension, when a phone dials 1234, the dialplan will execute the `Dial` application, which is the main application used to initiate an outbound call.

```
exten => 1234,1,Dial(SIP/bob)
```

The argument specified to the `Dial` application says that the system should make an outbound call to the device referred to as `SIP/bob`. The `SIP` portion of this argument specifies that the `SIP` protocol should be used to deliver the call. `bob` will be interpreted by the channel driver that implements the `SIP` protocol, `chan_sip`. Assuming the channel driver has been properly configured with an account called `bob`, it will know how to reach Bob's phone.

The `Dial` application will ask the Asterisk core to allocate a new Asterisk channel using the `SIP/bob` identifier. The core will request that the `SIP` channel driver perform technology specific initialization. The channel driver will also initiate the process of making a call out to the phone. As the request proceeds, it will pass events back into the Asterisk core, which will be received by the `Dial` application. These events may include a response that the call has been answered, the destination is busy, the network is congested, the call was rejected for some reason, or a number of other possible responses. In the ideal case, the call will be answered. The fact that the call has been answered is propagated back to the inbound channel. Asterisk will not answer the part of the call that came into the system until the outbound call was answered. Once both channels are answered, the bridging of the channels begins (Figure 1.7).

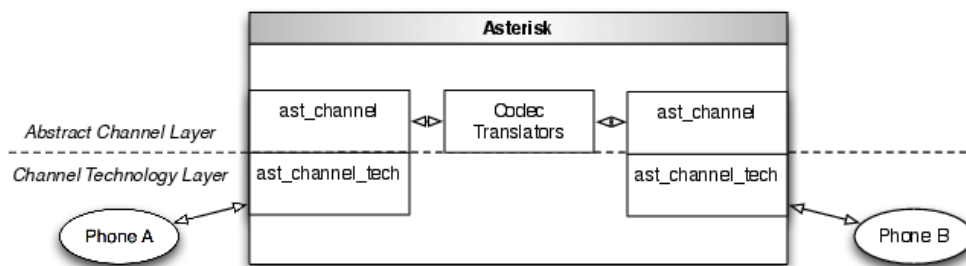


Figure 1.7: Block Diagram of a Bridged Call in a Generic Bridge

During a channel bridge, audio and signalling events from one channel are passed to the other until some event occurs that causes the bridge to end, such as one side of the call hanging up. The sequence diagram in Figure 1.8 demonstrates the key operations that are performed for an audio frame during a bridged call.

Once the call is done, the hangup process is very similar to the previous example. The major difference here is that there are two channels involved. The channel technology specific hangup processing will be executed for both channels before the channel thread stops running.

1.5 Final Comments

The architecture of Asterisk is now more than ten years old. However, the fundamental concepts of channels and flexible call handling using the Asterisk dialplan still support the development of complex telephony systems in an industry that is continuously evolving. One area that the architecture

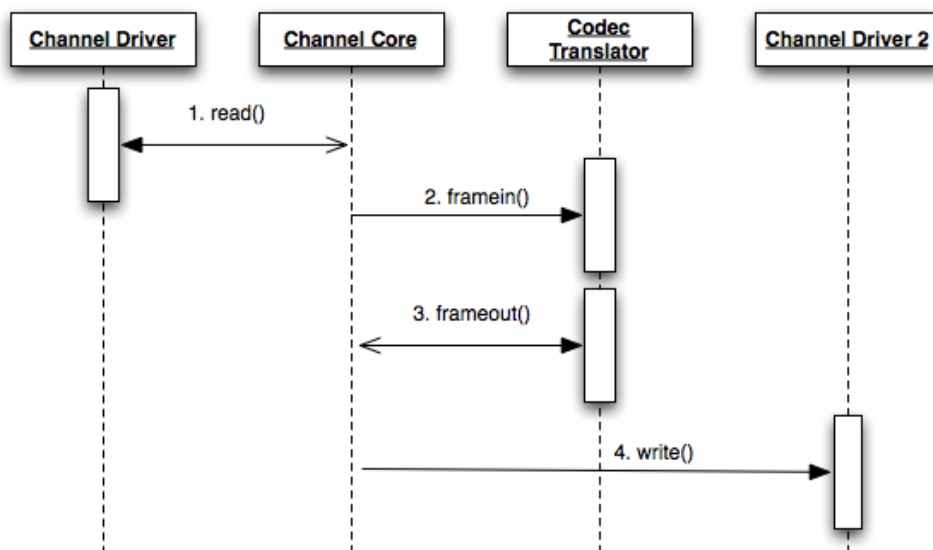


Figure 1.8: Sequence Diagram for Audio Frame Processing During a Bridge

of Asterisk does not address very well is scaling a system across multiple servers. The Asterisk development community is currently developing a companion project called Asterisk SCF (Scalable Communications Framework) which is intended to address these scalability concerns. In the next few years, we expect to see Asterisk, along with Asterisk SCF, continue to take over significant portions of the telephony market, including much larger installations.

Audacity

James Crook

Audacity is a popular sound recorder and audio editor. It is a capable program while still being easy to use. The majority of users are on Windows but the same Audacity source code compiles to run on Linux and Mac too.

Dominic Mazzoni wrote the original version of Audacity in 1999 while he was a research student at Carnegie Mellon University. Dominic wanted to create a platform on which to develop and debug audio processing algorithms. The software grew to become useful in its own right in many other ways. Once Audacity was released as open source software, it attracted other developers. A small, gradually-changing team of enthusiasts have modified, maintained, tested, updated, written documentation for, helped users with, and translated Audacity's interface into other languages over the years.

One goal is that its user interface should be discoverable: people should be able to sit down without a manual and start using it right away, gradually discovering its features. This principle has been crucial in giving Audacity greater consistency to the user interface than there otherwise would be. For a project in which many people have a hand this kind of unifying principle is more important than it might seem at first.

It would be good if the architecture of Audacity had a similar guiding principle, a similar kind of discoverability. The closest we have to that is “try and be consistent”. When adding new code, developers try to follow the style and conventions of code nearby. In practice, though, the Audacity code base is a mix of well-structured and less well-structured code. Rather than an overall architecture the analogy of a small city is better: there are some impressive buildings but you will also find run-down neighborhoods that are more like a shanty town.

2.1 Structure in Audacity

Audacity is layered upon several libraries. While most new programming in Audacity code doesn't require a detailed knowledge of exactly what is going on in these libraries, familiarity with their APIs and what they do is important. The two most important libraries are PortAudio which provides a low-level audio interface in a cross-platform way, and wxWidgets which provides GUI components in a cross-platform way.

When reading Audacity's code, it helps to realize that only a fraction of the code is essential. Libraries contribute a lot of optional features—though people who use those features might not

consider them optional. For example, as well as having its own built-in audio effects, Audacity supports LADSPA (Linux Audio Developer's Simple Plugin API) for dynamically loadable plugin audio effects. The VAMP API in Audacity does the same thing for plugins that analyze audio. Without these APIs, Audacity would be less feature-rich, but it does not absolutely depend on these features.

Other optional libraries used by Audacity are libFLAC, libogg, and libvorbis. These provide various audio compression formats. MP3 format is catered for by dynamically loading the LAME or FFmpeg library. Licensing restrictions prevent these very popular compression libraries from being built-in.

Licensing is behind some other decisions about Audacity libraries and structure. For example, support for VST plugins is not built in because of licensing restrictions. We would also like to use the very efficient FFTW fast Fourier transform code in some of our code. However, we only provide that as an option for people who compile Audacity themselves, and instead fall back to a slightly slower version in our normal builds. As long as Audacity accepts plugins, it can be and has been argued that Audacity cannot use FFTW. FFTW's authors do not want their code to be available as a general service to arbitrary other code. So, the architectural decision to support plugins leads to a trade-off in what we can offer. It makes LADSPA plugins possible but bars us from using FFTW in our pre-built executables.

Architecture is also shaped by considerations of how best to use our scarce developer time. With a small team of developers, we do not have the resources to do, for example, the in-depth analysis of security loopholes that teams working on Firefox and Thunderbird do. However, we do not want Audacity to provide a route to bypass a firewall, so we have a rule not to have TCP/IP connections to or from Audacity at all. Avoiding TCP/IP cuts out many security concerns. The awareness of our limited resources leads us to better design. It helps us cut features that would cost us too much in developer time and focus on what is essential.

A similar concern for developers' time applies to scripting languages. We want scripting, but the code implementing the languages does not need to be in Audacity. It does not make sense to compile copies of each scripting language into Audacity to give users all the choices they could want.¹ We have instead implemented scripting with a single plugin module and a pipe, which we will cover later.

Figure 2.1 shows some layers and modules in Audacity. The diagram highlights three important classes within wxWidgets, each of which has a reflection in Audacity. We're building higher-level abstractions from related lower-level ones. For example, the BlockFile system is a reflection of and is built on wxWidgets' wxFiles. It might, at some stage, make sense to split out BlockFiles, ShuttleGUI, and command handling into an intermediate library in their own right. This would encourage us to make them more general.

Lower down in the diagram is a narrow strip for "Platform Specific Implementation Layers." Both wxWidgets and PortAudio are OS abstraction layers. Both contain conditional code that chooses between different implementations depending on the target platform.

The "Other Supporting Libraries" category includes a wide collection of libraries. Interestingly quite a few of these rely on dynamically loaded modules. Those dynamic modules know nothing of wxWidgets.

On the Windows platform we used to compile Audacity as a single monolithic executable with wxWidgets and Audacity application code in the same executable. In 2008 we changed over to using a modular structure with wxWidgets as a separate DLL. This is to allow additional optional DLLs to

¹The one exception to this is the Lisp-based Nyquist language which has been built into Audacity from very early days. We would like to make it a separate module, bundled with Audacity, but we have not yet had the time to make that change.

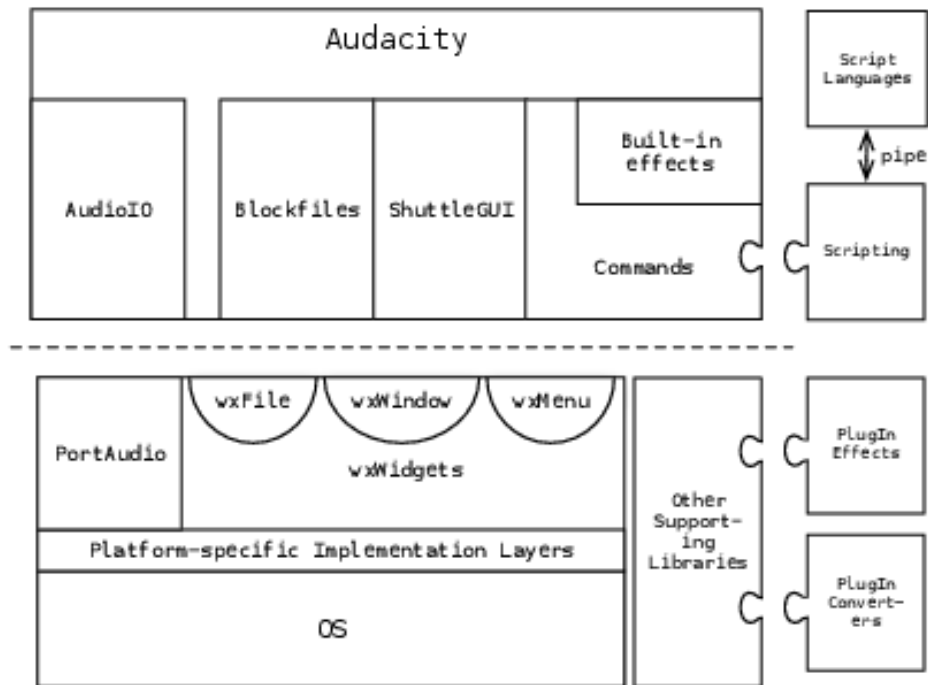


Figure 2.1: Layers in Audacity

be loaded at run time where those DLLs directly use features of wxWidgets. Plugins that plug in above the dotted line in the diagram can use wxWidgets.

The decision to use DLLs for wxWidgets has its downsides. The distribution is now larger, partly because many unused functions are provided in the DLLs that would previously have been optimized away. Audacity also takes longer to start up because each DLL is loaded separately. The advantages are considerable. We expect modules to have similar advantages for us as they do for Apache. As we see it, modules allow the core of Apache to be very stable while facilitating experimentation, special features and new ideas in the modules. Modules go a very long way to counteracting the temptation to fork a project to take it in a new direction. We think it's been a very important architectural change for us. We're expecting these advantages but have not seen them yet. Exposing the wxWidgets functions is only a first step and we have more to do to have a flexible modular system.

The structure of a program like Audacity clearly is not designed up front. It is something that develops over time. By and large the architecture we now have works well for us. We find ourselves fighting the architecture when we try to add features that affect many of the source files. For example, Audacity currently handles stereo and mono tracks in a special cased way. If you wanted to modify Audacity to handle surround sound you'd need to make changes in many classes in Audacity.

Going Beyond Stereo: The GetLink Story

Audacity has never had an abstraction for number of channels. Instead the abstraction it uses is to link audio channels. There is a function `GetLink` that returns the other audio channel in a pair if there are two and that returns `NULL` if the track is mono. Code that uses `GetLink` typically looks exactly as if it were originally written for mono and later a test of `(GetLink() != NULL)` used to extend that code to handle stereo. I'm not sure it was actually written that way, but I suspect it. There's no looping using `GetLink` to iterate through all channels in a linked list. Drawing, mixing, reading and writing all contain a test for the stereo case rather than general code that can work for n channels where n is most likely to be one or two. To go for the more general code you'd need to make changes at around 100 of these calls to the `GetLink` function modifying at least 26 files.

It's easy to search the code to find `GetLink` calls and the changes needed are not that complex, so it is not as big a deal to fix this "problem" as it might sound at first. The `GetLink` story is not about a structural defect that is hard to fix. Rather it's illustrative of how a relatively small defect can travel into a lot of code, if allowed to.

With hindsight it would have been good to make the `GetLink` function private and instead provide an iterator to iterate through all channels in a track. This would have avoided much special case code for stereo, and at the same time made code that uses the list of audio channels agnostic with respect to the list implementation.

The more modular design is likely to drive us towards better hiding of internal structure. As we define and extend an external API we'll need to look more closely at the functions we're providing. This will draw our attention to abstractions that we don't want to lock in to an external API.

2.2 wxWidgets GUI Library

The most significant single library for Audacity user interface programmers is the wxWidgets GUI library, which provides such things as buttons, sliders, check boxes, windows and dialogs. It provides the most visible cross-platform behavior. The wxWidgets library has its own string class `wxString`, it has cross-platform abstractions for threads, filesystems, and fonts, and a mechanism for localization to other languages, all of which we use. We advise people new to Audacity development to first download wxWidgets and compile and experiment with some of the samples that come with that library. wxWidgets is a relatively thin layer on the underlying GUI objects provided by the operating system.

To build up complex dialogs wxWidgets provides not only individual widget elements but also sizers that control the elements' sizes and positions. This is a lot nicer than giving absolute fixed positions to graphical elements. If the widgets are resized either directly by the user or, say, by using a different font size, the positioning of the elements in the dialogs updates in a very natural way. Sizers are important for a cross-platform application. Without them we might have to have custom layouts of dialogs for each platform.

Often the design for these dialogs is in a resource file that is read by the program. However in Audacity we exclusively compile dialog designs into the program as a series of calls to wxWidgets functions. This provides maximum flexibility: that is, dialogs whose exact contents and behavior will be determined by application level code.

You could at one time find places in Audacity where the initial code for creating a GUI had clearly been code-generated using a graphical dialog building tool. Those tools helped us get a basic design. Over time the basic code was hacked around to add new features, resulting in many places where new dialogs were created by copying and modifying existing, already hacked-around dialog code.

After a number of years of such development we found that large sections of the Audacity source code, particularly the dialogs for configuring user preferences, consisted of tangled repetitive code. That code, though simple in what it did, was surprisingly hard to follow. Part of the problem was that the sequence in which dialogs were built up was quite arbitrary: smaller elements were combined into larger ones and eventually into complete dialogs, but the order in which elements were created by the code did not (and did not need to) resemble the order elements were laid out on screen. The code was also verbose and repetitive. There was GUI-related code to transfer data from preferences stored on disk to intermediate variables, code to transfer from intermediate variables to the displayed GUI, code to transfer from the displayed GUI to intermediate variables, and code to transfer from intermediate variables to the stored preferences. There were comments in the code along the lines of `//this is a mess`, but it was quite some time before anything was done about it.

2.3 ShuttleGui Layer

The solution to untangling all this code was a new class, ShuttleGui, that much reduced the number of lines of code needed to specify a dialog, making the code more readable. ShuttleGui is an extra layer between the wxWidgets library and Audacity. Its job is to transfer information between the two. Here's an example which results in the GUI elements pictured in Figure 2.2.

```
ShuttleGui S;  
// GUI Structure  
S.StartStatic("Some Title",...);  
{  
    S.AddButton("Some Button",...);  
    S.TieCheckbox("Some Checkbox",...);  
}  
S.EndStatic();
```

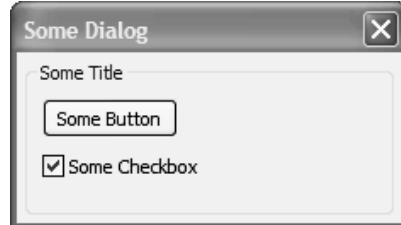


Figure 2.2: Example Dialog

This code defines a static box in a dialog and that box contains a button and a checkbox. The correspondence between the code and the dialog should be clear. The `StartStatic` and `EndStatic` are paired calls. Other similar `StartSomething/EndSomething` pairs, which must match, are used for controlling other aspects of layout of the dialog. The curly brackets and the indenting that goes with them aren't needed for this to be correct code. We adopted the convention of adding them in to make the structure and particularly the matching of the paired calls obvious. It really helps readability in larger examples.

The source code shown does not just create the dialog. The code after the comment `“//GUI Structure”` can also be used to shuttle data from the dialog out to where the user preferences are stored, and to shuttle data back in. Previously a lot of the repetitive code came from the need to do this. Nowadays that code is only written once and is buried within the ShuttleGui class.

There are other extensions to the basic wxWidgets in Audacity. Audacity has its own class for managing toolbars. Why doesn't it use wxWidget's built in toolbar class? The reason is historic: Audacity's toolbars were written before wxWidgets provided a toolbar class.

2.4 The TrackPanel

The main panel in Audacity which displays audio waveforms is the TrackPanel. This is a custom control drawn by Audacity. It's made up of components such as smaller panels with track information, a ruler for the timebase, rulers for amplitude, and tracks which may show waveforms, spectra or textual labels. The tracks can be resized and moved around by dragging. The tracks containing textual labels make use of our own re-implementation of an editable text box rather than using the built-in text box. You might think these panels tracks and rulers should each be a wxWidgets component, but they are not.

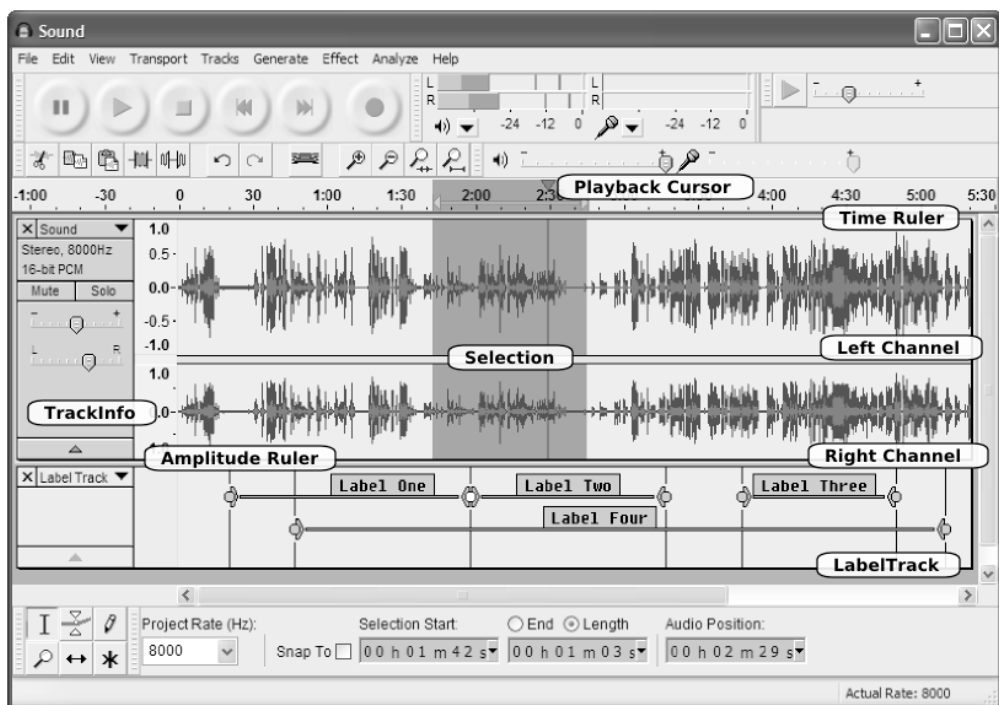


Figure 2.3: Audacity Interface with Track Panel Elements Labelled

The screenshot shown in Figure 2.3 shows the Audacity user interface. All the components that have been labelled are custom for Audacity. As far as wxWidgets is concerned there is one wxWidget component for the TrackPanel. Audacity code, not wxWidgets, takes care of the positioning and repainting within that.

The way all these components fit together to make the TrackPanel is truly horrible. (It's the code that's horrible; the end result the user sees looks just fine.) The GUI and application-specific code is all mixed together, not separated cleanly. In a good design only our application-specific code should

know about left and right audio channels, decibels, muting and soloing. GUI elements should be application agnostic elements that are reusable in a non-audio application. Even the purely GUI parts of TrackPanel are a patchwork of special case code with absolute positions and sizes and not enough abstraction. It would be so much nicer, cleaner and more consistent if these special components were self-contained GUI elements and if they used sizers with the same kinds of interface as wxWidgets uses.

To get to such a TrackPanel we'd need a new sizer for wxWidgets that can move and resize tracks or, indeed, any other widget. wxWidgets sizers aren't yet that flexible. As a spin off benefit we could use that sizer elsewhere. We could use it in the toolbars that hold the buttons, making it easy to customize the order of buttons within a toolbar by dragging.

Some exploratory work has been done in creating and using such sizers, but not enough. Some experiments with making the GUI components fully fledged wxWidgets ran into a problem: doing so reduces our control over repainting of the widgets, resulting in flicker when resizing and moving components. We would need to extensively modify wxWidgets to achieve flicker-free repainting, and better separate the resizing steps from the repainting steps.

A second reason to be wary of this approach for the TrackPanel is that we already know wxWidgets start running very slowly when there are large numbers of widgets. This is mostly outside of wxWidget's control. Each wxWidget, button, and text entry box uses a resource from the windowing system. Each has a handle to access it. Processing large numbers of these takes time. Processing is slow even when the majority of widgets are hidden or off screen. We want to be able to use many small widgets on our tracks.

The best solution is to use a flyweight pattern, lightweight widgets that we draw ourselves, which do not have corresponding objects that consume windowing system resources or handles. We would use a structure like wxWidgets's sizers and component widgets, and give the components a similar API but not actually derive from wxWidgets classes. We'd be refactoring our existing TrackPanel code so that its structure became a lot clearer. If this were an easy solution it would already have been done, but diverging opinions about exactly what we want to end up with derailed an earlier attempt. Generalizing our current ad hoc approach would take significant design work and coding. There is a great temptation to leave complex code that already works well enough alone.

2.5 PortAudio Library: Recording and Playback

PortAudio is the audio library that gives Audacity the ability to play and record audio in a cross-platform way. Without it Audacity would not be able to use the sound card of the device it's running on. PortAudio provides the ring buffers, sample rate conversion when playing/recording and, crucially, provides an API that hides the differences between audio on Mac, Linux and Windows. Within PortAudio there are alternative implementation files to support this API for each platform.

I've never needed to dig into PortAudio to follow what happens inside. It is, however, useful to know how we interface with PortAudio. Audacity accepts data packets from PortAudio (recording) and sends packets to PortAudio (playback). It's worth looking at exactly how the sending and receiving happens, and how it fits in with reading and writing to disk and updates to the screen.

Several different processes are going on at the same time. Some happen frequently, transfer small amounts of data, and must be responded to quickly. Others happen less frequently, transfer larger quantities of data, and the exact timing of when they happen is less critical. This is an impedance mismatch between the processes, and buffers are used to accommodate it. A second part of the picture is that we are dealing with audio devices, hard drives, and the screen. We don't go down to the wire

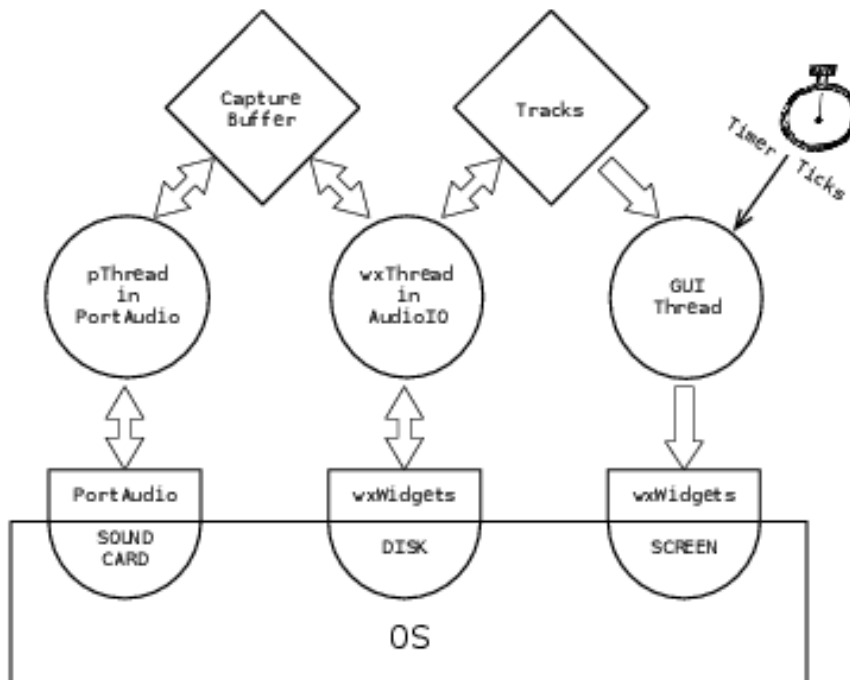


Figure 2.4: Threads and Buffers in Playback and Recording

and so have to work with the APIs we’re given. Whilst we would like each of our processes to look similar, for example to have each running from a wxThread, we don’t have that luxury (Figure 2.4).

One audio thread is started by PortAudio code and interacts directly with the audio device. This is what drives recording or playback. This thread has to be responsive or packets will get lost. The thread, under the control of PortAudio code, calls `audacityAudioCallback` which, when recording, adds newly arrived small packets to a larger (five second) capture buffer. When playing back it takes small chunks off a five second playback buffer. The PortAudio library knows nothing about wxWidgets and so this thread created by PortAudio is a pthread.

A second thread is started by code in Audacity’s class `AudioIO`. When recording, `AudioIO` takes the data from the capture buffer and appends it to Audacity’s tracks so that it will eventually get displayed. Additionally, when enough data has been added, `AudioIO` writes the data to disk. This same thread also does the disk reads for audio playback. The function `AudioIO::FillBuffers` is the key function here and depending on the settings of some Boolean variables, handles both recording and playback in the one function. It’s important that the one function handle both directions. Both the recording and playback parts are used at the same time when doing “software play through,” where you overdub what was previously recorded. In the `AudioIO` thread we are totally at the mercy of the operating system’s disk IO. We may stall for an unknown length of time reading or writing to a disk. We could not do those reads or writes in `audacityAudioCallback` because of the need to be responsive there.

Communication between these two threads happens via shared variables. Because we control which threads are writing to these variables and when, we avoid the need for more expensive mutexes.

In both playback and recording, there is an additional requirement: Audacity also needs to update the GUI. This is the least time critical operation. The update happens in the main GUI thread and is due to a periodic timer that ticks twenty times a second. This timer's tick causes `TrackPanel::OnTimer` to be called, and if updates to the GUI are found to be needed, they are applied. This main GUI thread is created within `wxWidgets` rather than by our own code. It is special in that other threads cannot directly update the GUI. Using a timer to get the GUI thread to check if it needs to update the screen allows us to reduce the number of repaints to a level that is acceptable for a responsive display, and not make too heavy demands on processor time for displaying.

Is it good design to have an audio device thread, a buffer/disk thread and a GUI thread with periodic timer to handle these audio data transfers? It is somewhat ad hoc to have these three different threads that are not based on a single abstract base class. However, the ad-hockery is largely dictated by the libraries we use. `PortAudio` expects to create a thread itself. The `wxWidgets` framework automatically has a GUI thread. Our need for a buffer filling thread is dictated by our need to fix the impedance mismatch between the frequent small packets of the audio device thread and the less frequent larger packets of the disk drive. There is very clear benefit in using these libraries. The cost in using the libraries is that we end up using the abstractions they provide. As a result we copy data in memory from one place to another more than is strictly necessary. In fast data switches I've worked on, I've seen extremely efficient code for handling these kinds of impedance mismatches that is interrupt driven and does not use threads at all. Pointers to buffers are passed around rather than copying data. You can only do that if the libraries you are using are designed with a richer buffer abstraction. Using the existing interfaces, we're forced to use threads and we're forced to copy data.

2.6 BlockFiles

One of the challenges faced by Audacity is supporting insertions and deletions into audio recordings that may be hours long. Recordings can easily be too long to fit in available RAM. If an audio recording is in a single disk file, inserting audio somewhere near the start of that file could mean moving a lot of data to make way. Copying that data on disk would be time consuming and mean that Audacity could then not respond rapidly to simple edits.

Audacity's solution to this is to divide audio files into many `BlockFiles`, each of which could be around 1 MB. This is the main reason Audacity has its own audio file format, a master file with the extension `.aup`. It is an XML file which coordinates the various blocks. Changes near the start of a long audio recording might affect just one block and the master `.aup` file.

`BlockFiles` balance two conflicting forces. We can insert and delete audio without excessive copying, and during playback we are guaranteed to get reasonably large chunks of audio with each request to the disk. The smaller the blocks, the more potential disk requests to fetch the same amount of audio data; the larger the blocks, the more copying on insertions and deletions.

Audacity's `BlockFiles` never have internal free space and they never grow beyond the maximum block size. To keep this true when we insert or delete we may end up copying up to one block's worth of data. When we don't need a `BlockFile` anymore we delete it. The `BlockFiles` are reference counted so if we delete some audio, the relevant `BlockFiles` will still hang around to support the undo mechanism until we save. There is never a need to garbage collect free space within Audacity `BlockFiles`, which we would need to do with an all-in-one-file approach.

Merging and splitting larger chunks of data is the bread and butter of data management systems, from B-trees to Google's `BigTable` tablets to the management of unrolled linked lists. Figure 2.5 shows what happens in Audacity when removing a span of audio near the start.

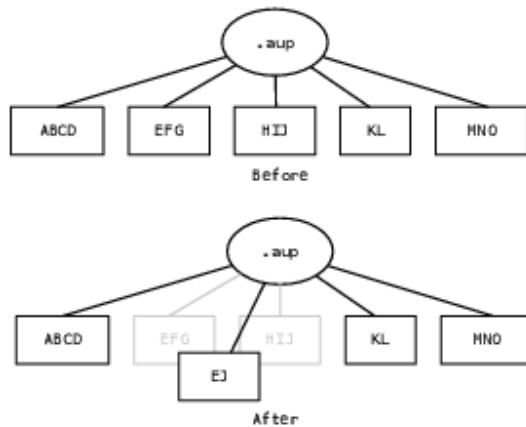


Figure 2.5: Before deletion, .aup file and BlockFiles hold the sequence ABCDEFGHIJKLMNO. After deletion of FGHI, two BlockFiles are merged.

BlockFiles aren't just used for the audio itself. There are also BlockFiles that cache summary information. If Audacity is asked to display a four hour long recording on screen it is not acceptable for it to process the entire audio each time it redraws the screen. Instead it uses summary information which gives the maximum and minimum audio amplitude over ranges of time. When zoomed in, Audacity is drawing using actual samples. When zoomed out, Audacity is drawing using summary information.

A refinement in the BlockFile system is that the blocks needn't be files created by Audacity. They can be references to subsections of audio files such as a timespan from audio stored in the .wav format. A user can create an Audacity project, import audio from a .wav file and mix a number of tracks whilst only creating BlockFiles for the summary information. This saves disk space and saves time in copying audio. All told it is, however, a rather bad idea. Far too many of our users have removed the original audio .wav file thinking there will be a complete copy in the Audacity project folder. That's not so and without the original .wav file the audio project can no longer be played. The default in Audacity nowadays is to always copy imported audio, creating new BlockFiles in the process.

The BlockFile solution ran into problems on Windows systems where having a large number of BlockFiles performed very poorly. This appeared to be because Windows was much slower handling files when there were many in the same directory, a similar problem to the slowdown with large numbers of widgets. A later addition was made to use a hierarchy of subdirectories, never with more than a hundred files in each subdirectory.

The main problem with the BlockFile structure is that it is exposed to end users. We often hear from users who move the .aup file and don't realize they also need to move the folder containing all the BlockFiles too. It would be better if Audacity projects were a single file with Audacity taking responsibility for how the space inside the file is used. If anything this would increase performance rather than reduce it. The main additional code needed would be for garbage collection. A simple approach to that would be to copy the blocks to a new file when saving if more than a set percentage of the file were unused.

2.7 Scripting

Audacity has an experimental plugin that supports multiple scripting languages. It provides a scripting interface over a named pipe. The commands exposed via scripting are in a textual format, as are the responses. As long as the user's scripting language can write text to and read text from a named pipe, the scripting language can drive Audacity. Audio and other high-volume data does not need to travel on the pipe (Figure 2.6).

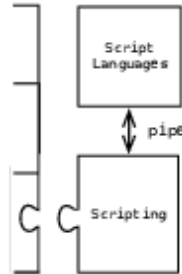


Figure 2.6: Scripting Plugin Provides Scripting Over a Named Pipe

The plugin itself knows nothing about the content of the text traffic that it carries. It is only responsible for conveying it. The plugin interface (or rudimentary extension point) used by the scripting plugin to plug in to Audacity already exposes Audacity commands in textual format. So, the scripting plugin is small, its main content being code for the pipe.

Unfortunately a pipe introduces similar security risks to having a TCP/IP connection—and we've ruled out TCP/IP connections for Audacity on security grounds. To reduce that risk the plugin is an optional DLL. You have to make a deliberate decision to obtain and use it and it comes with a health/security warning.

After the scripting feature had already been started, a suggestion surfaced in the feature requests page of our wiki that we should consider using KDE's D-Bus standard to provide an inter-process call mechanism using TCP/IP. We'd already started going down a different route but it still might make sense to adapt the interface we've ended up with to support D-Bus.

2.8 Real-Time Effects

Audacity does not have real-time effects, that is, audio effects that are calculated on demand as the audio plays. Instead in Audacity you apply an effect and must wait for it to complete. Real-time effects and rendering of audio effects in the background whilst the user interface stays responsive are among the most frequently made feature requests for Audacity.

A problem we have is that what may be a real-time effect on one machine may not run fast enough to be real-time on a much slower machine. Audacity runs on a wide range of machines. We'd like a graceful fallback. On a slower machine we'd still want to be able to request an effect be applied to an entire track and to then listen to the processed audio near the middle of the track, after a small wait, with Audacity knowing to process that part first. On a machine too slow to render the effect in real time we'd be able to listen to the audio until playback caught up with the rendering. To do this we'd need to remove the restrictions that audio effects hold up the user interface and that the order of processing the audio blocks is strictly left to right.

Origins of Scripting Code

The scripting feature grew from an enthusiast's adaptation of Audacity for a particular need that was heading in the direction of being a fork. These features, together called CleanSpeech, provide for mp3 conversion of sermons. CleanSpeech adds new effects such as truncate silence—the effect finds and cuts out long silences in audio—and the ability to apply a fixed sequence of existing noise removal effects, normalization and mp3 conversion to a whole batch of audio recordings. We wanted some of the excellent functionality in this, but the way it was written was too special case for Audacity. Bringing it into mainstream Audacity led us to code for a flexible sequence rather than a fixed sequence. The flexible sequence could use any of the effects via a look-up table for command names and a Shuttle class to persist the command parameters to a textual format in user preferences. This feature is called *batch chains*. Very deliberately we stopped short of adding conditionals or calculation to avoid inventing an ad hoc scripting language.

In retrospect the effort to avoid a fork has been well worthwhile. There is still a CleanSpeech mode buried in Audacity that can be set by modifying a preference. It also cuts down the user interface, removing advanced features. A simplified version of Audacity has been requested for other uses, most notably in schools. The problem is that each person's view of which are the advanced features and which are the essential ones is different. We've subsequently implemented a simple hack that leverages the translation mechanism. When the translation of a menu item starts with a “#” it is no longer shown in the menus. That way people who want to reduce the menus can make choices themselves without recompiling—more general and less invasive than the mCleanspeech flag in Audacity, which in time we may be able to remove entirely.

The CleanSpeech work gave us batch chains and the ability to truncate silence. Both have attracted additional improvement from outside the core team. Batch chains directly led on to the scripting feature. That in turn has begun the process of supporting more general purpose plugins to adapt Audacity.

A relatively recent addition in Audacity called *on demand loading* has many of the elements we need for real time effects, though it doesn't involve audio effects at all. When you import an audio file into Audacity, it can now make the summary BlockFiles in a background task. Audacity will show a placeholder of diagonal blue and gray stripes for audio that it has not yet processed and respond to many user commands whilst the audio is still being loaded. The blocks do not have to be processed in left-to-right order. The intention has always been that the same code will in due course be used for real-time effects.

On demand loading gives us an evolutionary approach to adding real time effects. It's a step that avoids some of the complexities of making the effects themselves real-time. Real-time effects will additionally need overlap between the blocks, otherwise effects like echo will not join up correctly. We'll also need to allow parameters to vary as the audio is playing. By doing on demand loading first, the code gets used at an earlier stage than it otherwise would. It will get feedback and refinement from actual use.

2.9 Summary

The earlier sections of this chapter illustrate how good structure contribute to a program's growth, or how the absence of good structure hinders it.

- Third party APIs such as PortAudio and wxWidgets have been of huge benefit. They've given us code that works to build on, and abstracted away many platform differences. One price we pay for using them is that we don't get the flexibility to choose the abstractions. We have less than pretty code for playback and recording because we have to handle threading in three different ways. The code also does more copying of data than it could do if we controlled the abstractions.
- The API given to us by wxWidgets tempted us into writing some verbose, hard to follow application code. Our solution to that was to add a facade in front of wxWidgets to give us the abstractions we wanted and cleaner application code.
- In the TrackPanel of Audacity we needed to go outside the features that could easily be got from existing widgets. As a result we rolled our own ad hoc system. There is a cleaner system with widgets and sizers and logically distinct application level objects struggling to come out of the TrackPanel.
- Structural decisions are wider ranging than deciding how to structure new features. A decision about what not to include in a program can be as important. It can lead to cleaner, safer code. It's a pleasure to get the benefits of scripting languages like Perl without having to do the work of maintaining our own copy. Structural decisions also are driven by plans for future growth. Our embryonic modular system is expected to lead to more experimentation by making experiments safer. On demand loading is expected to be an evolutionary step towards on demand processing of real time effects.

The more you look, the more obvious it is that Audacity is a community effort. The community is larger than just those contributing directly because it depends on libraries, each of which has its own community with its own domain experts. Having read about the mix of structure in Audacity it probably comes as no surprise that the community developing it welcomes new developers and is well able to handle a wide range of skill levels.

For me there is no question that the nature of the community behind Audacity is reflected in the strengths and weaknesses of the code. A more closed group could write high quality code more consistently than we have, but it would be harder to match the range of capabilities Audacity has with fewer people contributing.

The Bourne-Again Shell

Chet Ramey

3.1 Introduction

A Unix shell provides an interface that lets the user interact with the operating system by running commands. But a shell is also a fairly rich programming language: there are constructs for flow control, alternation, looping, conditionals, basic mathematical operations, named functions, string variables, and two-way communication between the shell and the commands it invokes.

Shells can be used interactively, from a terminal or terminal emulator such as `xterm`, and non-interactively, reading commands from a file. Most modern shells, including `bash`, provide command-line editing, in which the command line can be manipulated using `emacs`- or `vi`-like commands while it's being entered, and various forms of a saved history of commands.

`Bash` processing is much like a shell pipeline: after being read from the terminal or a script, data is passed through a number of stages, transformed at each step, until the shell finally executes a command and collects its return status.

This chapter will explore `bash`'s major components: input processing, parsing, the various word expansions and other command processing, and command execution, from the pipeline perspective. These components act as a pipeline for data read from the keyboard or from a file, turning it into an executed command.

Bash

`Bash` is the shell that appears in the GNU operating system, commonly implemented atop the Linux kernel, and several other common operating systems, most notably Mac OS X. It offers functional improvements over historical versions of `sh` for both interactive and programming use.

The name is an acronym for Bourne-Again SHell, a pun combining the name of Stephen Bourne (the author of the direct ancestor of the current Unix shell `/bin/sh`, which appeared in the Bell Labs Seventh Edition Research version of Unix) with the notion of rebirth through reimplementation. The original author of `bash` was Brian Fox, an employee of the Free Software Foundation. I am the current developer and maintainer, a volunteer who works at Case Western Reserve University in Cleveland, Ohio.

Like other GNU software, `bash` is quite portable. It currently runs on nearly every version of Unix and a few other operating systems—independently-supported ports exist for hosted Windows

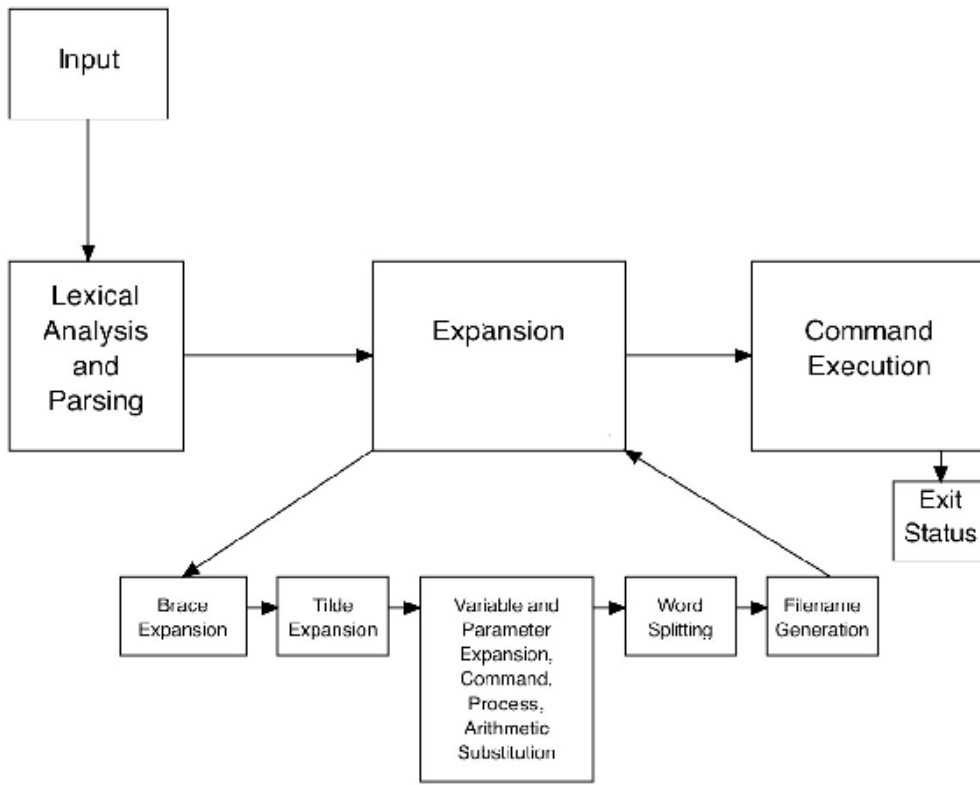


Figure 3.1: Bash Component Architecture

environments such as Cygwin and MinGW, and ports to Unix-like systems such as QNX and Minix are part of the distribution. It only requires a Posix environment to build and run, such as one provided by Microsoft's Services for Unix (SFU).

3.2 Syntactic Units and Primitives

Primitives

To bash, there are basically three kinds of tokens: reserved words, words, and operators. Reserved words are those that have meaning to the shell and its programming language; usually these words introduce flow control constructs, like `if` and `while`. Operators are composed of one or more metacharacters: characters that have special meaning to the shell on their own, such as `|` and `>`. The rest of the shell's input consists of ordinary words, some of which have special meaning—assignment statements or numbers, for instance—depending on where they appear on the command line.

Variables and Parameters

As in any programming language, shells provide variables: names to refer to stored data and operate on it. The shell provides basic user-settable variables and some built-in variables referred to as parameters. Shell parameters generally reflect some aspect of the shell's internal state, and are set automatically or as a side effect of another operation.

Variable values are strings. Some values are treated specially depending on context; these will be explained later. Variables are assigned using statements of the form `name=value`. The value is optional; omitting it assigns the empty string to name. If the value is supplied, the shell expands the value and assigns it to name. The shell can perform different operations based on whether or not a variable is set, but assigning a value is the only way to set a variable. Variables that have not been assigned a value, even if they have been declared and given attributes, are referred to as *unset*.

A word beginning with a dollar sign introduces a variable or parameter reference. The word, including the dollar sign, is replaced with the value of the named variable. The shell provides a rich set of expansion operators, from simple value replacement to changing or removing portions of a variable's value that match a pattern.

There are provisions for local and global variables. By default, all variables are global. Any simple command (the most familiar type of command—a command name and optional set of arguments and redirections) may be prefixed by a set of assignment statements to cause those variables to exist only for that command. The shell implements stored procedures, or shell functions, which can have function-local variables.

Variables can be minimally typed: in addition to simple string-valued variables, there are integers and arrays. Integer-typed variables are treated as numbers: any string assigned to them is expanded as an arithmetic expression and the result is assigned as the variable's value. Arrays may be indexed or associative; indexed arrays use numbers as subscripts, while associative arrays use arbitrary strings. Array elements are strings, which can be treated as integers if desired. Array elements may not be other arrays.

Bash uses hash tables to store and retrieve shell variables, and linked lists of these hash tables to implement variable scoping. There are different variable scopes for shell function calls and temporary scopes for variables set by assignment statements preceding a command. When those assignment statements precede a command that is built into the shell, for instance, the shell has to keep track of the correct order in which to resolve variable references, and the linked scopes allow bash to do that. There can be a surprising number of scopes to traverse depending on the execution nesting level.

The Shell Programming Language

A *simple* shell command, one with which most readers are most familiar, consists of a command name, such as `echo` or `cd`, and a list of zero or more arguments and redirections. Redirections allow the shell user to control the input to and output from invoked commands. As noted above, users can define variables local to simple commands.

Reserved words introduce more complex shell commands. There are constructs common to any high-level programming language, such as `if-then-else`, `while`, a `for` loop that iterates over a list of values, and a C-like arithmetic `for` loop. These more complex commands allow the shell to execute a command or otherwise test a condition and perform different operations based on the result, or execute commands multiple times.

One of the gifts Unix brought the computing world is the pipeline: a linear list of commands, in which the output of one command in the list becomes the input of the next. Any shell construct can

be used in a pipeline, and it's not uncommon to see pipelines in which a command feeds data to a loop.

Bash implements a facility that allows the standard input, standard output, and standard error streams for a command to be redirected to another file or process when the command is invoked. Shell programmers can also use redirection to open and close files in the current shell environment.

Bash allows shell programs to be stored and used more than once. Shell functions and shell scripts are both ways to name a group of commands and execute the group, just like executing any other command. Shell functions are declared using a special syntax and stored and executed in the same shell's context; shell scripts are created by putting commands into a file and executing a new instance of the shell to interpret them. Shell functions share most of the execution context with the shell that calls them, but shell scripts, since they are interpreted by a new shell invocation, share only what is passed between processes in the environment.

A Further Note

As you read further, keep in mind that the shell implements its features using only a few data structures: arrays, trees, singly-linked and doubly-linked lists, and hash tables. Nearly all of the shell constructs are implemented using these primitives.

The basic data structure the shell uses to pass information from one stage to the next, and to operate on data units within each processing stage, is the `WORD_DESC`:

```
typedef struct word_desc {
    char *word;          /* Zero terminated string. */
    int flags;           /* Flags associated with this word. */
} WORD_DESC;
```

Words are combined into, for example, argument lists, using simple linked lists:

```
typedef struct word_list {
    struct word_list *next;
    WORD_DESC *word;
} WORD_LIST;
```

`WORD_LIST`s are pervasive throughout the shell. A simple command is a word list, the result of expansion is a word list, and the built-in commands each take a word list of arguments.

3.3 Input Processing

The first stage of the bash processing pipeline is input processing: taking characters from the terminal or a file, breaking them into lines, and passing the lines to the shell parser to transform into commands. As you would expect, the lines are sequences of characters terminated by newlines.

Readline and Command Line Editing

Bash reads input from the terminal when interactive, and from the script file specified as an argument otherwise. When interactive, bash allows the user to edit command lines as they are typed in, using familiar key sequences and editing commands similar to the Unix `emacs` and `vi` editors.

Bash uses the readline library to implement command line editing. This provides a set of functions allowing users to edit command lines, functions to save command lines as they are entered, to recall previous commands, and to perform csh-like history expansion. Bash is readline's primary client, and they are developed together, but there is no bash-specific code in readline. Many other projects have adopted readline to provide a terminal-based line editing interface.

Readline also allows users to bind key sequences of unlimited length to any of a large number of readline commands. Readline has commands to move the cursor around the line, insert and remove text, retrieve previous lines, and complete partially-typed words. On top of this, users may define macros, which are strings of characters that are inserted into the line in response to a key sequence, using the same syntax as key bindings. Macros afford readline users a simple string substitution and shorthand facility.

Readline Structure

Readline is structured as a basic read/dispatch/execute/redisplay loop. It reads characters from the keyboard using `read` or equivalent, or obtains input from a macro. Each character is used as an index into a keymap, or dispatch table. Though indexed by a single eight-bit character, the contents of each element of the keymap can be several things. The characters can resolve to additional keymaps, which is how multiple-character key sequences are possible. Resolving to a readline command, such as `beginning-of-line`, causes that command to be executed. A character bound to the `self-insert` command is stored into the editing buffer. It's also possible to bind a key sequence to a command while simultaneously binding subsequences to different commands (a relatively recently-added feature); there is a special index into a keymap to indicate that this is done. Binding a key sequence to a macro provides a great deal of flexibility, from inserting arbitrary strings into a command line to creating keyboard shortcuts for complex editing sequences. Readline stores each character bound to `self-insert` in the editing buffer, which when displayed may occupy one or more lines on the screen.

Readline manages only character buffers and strings using C chars, and builds multibyte characters out of them if necessary. It does not use `wchar_t` internally for both speed and storage reasons, and because the editing code existed before multibyte character support became widespread. When in a locale that supports multibyte characters, readline automatically reads an entire multibyte character and inserts it into the editing buffer. It's possible to bind multibyte characters to editing commands, but one has to bind such a character as a key sequence; this is possible, but difficult and usually not wanted. The existing emacs and vi command sets do not use multibyte characters, for instance.

Once a key sequence finally resolves to an editing command, readline updates the terminal display to reflect the results. This happens regardless of whether the command results in characters being inserted into the buffer, the editing position being moved, or the line being partially or completely replaced. Some bindable editing commands, such as those that modify the history file, do not cause any change to the contents of the editing buffer.

Updating the terminal display, while seemingly simple, is quite involved. Readline has to keep track of three things: the current contents of the buffer of characters displayed on the screen, the updated contents of that display buffer, and the actual characters displayed. In the presence of multibyte characters, the characters displayed do not exactly match the buffer, and the redisplay engine must take that into account. When redisplaying, readline must compare the current display buffer's contents with the updated buffer, figure out the differences, and decide how to most efficiently modify the display to reflect the updated buffer. This problem has been the subject of considerable research through the years (the *string-to-string correction problem*). Readline's approach is to

identify the beginning and end of the portion of the buffer that differs, compute the cost of updating just that portion, including moving the cursor backward and forward (e.g., will it take more effort to issue terminal commands to delete characters and then insert new ones than to simply overwrite the current screen contents?), perform the lowest-cost update, then clean up by removing any characters remaining at the end of the line if necessary and position the cursor in the correct spot.

The redisplay engine is without question the one piece of readline that has been modified most heavily. Most of the changes have been to add functionality—most significantly, the ability to have non-displaying characters in the prompt (to change colors, for instance) and to cope with characters that take up more than a single byte.

Readline returns the contents of the editing buffer to the calling application, which is then responsible for saving the possibly-modified results in the history list.

Applications Extending Readline

Just as readline offers users a variety of ways to customize and extend readline's default behavior, it provides a number of mechanisms for applications to extend its default feature set. First, bindable readline functions accept a standard set of arguments and return a specified set of results, making it easy for applications to extend readline with application-specific functions. Bash, for instance, adds more than thirty bindable commands, from bash-specific word completions to interfaces to shell built-in commands.

The second way readline allows applications to modify its behavior is through the pervasive use of pointers to hook functions with well-known names and calling interfaces. Applications can replace some portions of readline's internals, interpose functionality in front of readline, and perform application-specific transformations.

Non-interactive Input Processing

When the shell is not using readline, it uses either `stdio` or its own buffered input routines to obtain input. The bash buffered input package is preferable to `stdio` when the shell is not interactive because of the somewhat peculiar restrictions Posix imposes on input consumption: the shell must consume only the input necessary to parse a command and leave the rest for executed programs. This is particularly important when the shell is reading a script from the standard input. The shell is allowed to buffer input as much as it wants, as long as it is able to roll the file offset back to just after the last character the parser consumes. As a practical matter, this means that the shell must read scripts a character at a time when reading from non-seekable devices such as pipes, but may buffer as many characters as it likes when reading from files.

These idiosyncrasies aside, the output of the non-interactive input portion of shell processing is the same as readline: a buffer of characters terminated by a newline.

Multibyte Characters

Multibyte character processing was added to the shell a long time after its initial implementation, and it was done in a way designed to minimize its impact on the existing code. When in a locale that supports multibyte characters, the shell stores its input in a buffer of bytes (C chars), but treats these bytes as potentially multibyte characters. Readline understands how to display multibyte characters (the key is knowing how many screen positions a multibyte character occupies, and how many bytes to consume from a buffer when displaying a character on the screen), how to move forward and

backward in the line a character at a time, as opposed to a byte at a time, and so on. Other than that, multibyte characters don't have much effect on shell input processing. Other parts of the shell, described later, need to be aware of multibyte characters and take them into account when processing their input.

3.4 Parsing

The initial job of the parsing engine is lexical analysis: to separate the stream of characters into words and apply meaning to the result. The word is the basic unit on which the parser operates. Words are sequences of characters separated by metacharacters, which include simple separators like spaces and tabs, or characters that are special to the shell language, like semicolons and ampersands.

One historical problem with the shell, as Tom Duff said in his paper about `rc`, the Plan 9 shell, is that nobody really knows what the Bourne shell grammar is. The Posix shell committee deserves significant credit for finally publishing a definitive grammar for a Unix shell, albeit one that has plenty of context dependencies. That grammar isn't without its problems—it disallows some constructs that historical Bourne shell parsers have accepted without error—but it's the best we have.

The bash parser is derived from an early version of the Posix grammar, and is, as far as I know, the only Bourne-style shell parser implemented using Yacc or Bison. This has presented its own set of difficulties—the shell grammar isn't really well-suited to yacc-style parsing and requires some complicated lexical analysis and a lot of cooperation between the parser and the lexical analyzer.

In any event, the lexical analyzer takes lines of input from `readline` or another source, breaks them into tokens at metacharacters, identifies the tokens based on context, and passes them on to the parser to be assembled into statements and commands. There is a lot of context involved—for instance, the word `for` can be a reserved word, an identifier, part of an assignment statement, or other word, and the following is a perfectly valid command:

```
for for in for; do for=for; done; echo $for
```

that displays `for`.

At this point, a short digression about aliasing is in order. Bash allows the first word of a simple command to be replaced with arbitrary text using aliases. Since they're completely lexical, aliases can even be used (or abused) to change the shell grammar: it's possible to write an alias that implements a compound command that bash doesn't provide. The bash parser implements aliasing completely in the lexical phase, though the parser has to inform the analyzer when alias expansion is permitted.

Like many programming languages, the shell allows characters to be escaped to remove their special meaning, so that metacharacters such as `&` can appear in commands. There are three types of quoting, each of which is slightly different and permits slightly different interpretations of the quoted text: the backslash, which escapes the next character; single quotes, which prevent interpretation of all enclosed characters; and double quotes, which prevent some interpretation but allow certain word expansions (and treats backslashes differently). The lexical analyzer interprets quoted characters and strings and prevents them from being recognized by the parser as reserved words or metacharacters. There are also two special cases, `$'...'` and `"..."`, that expand backslash-escaped characters in the same fashion as ANSI C strings and allow characters to be translated using standard internationalization functions, respectively. The former is widely used; the latter, perhaps because there are few good examples or use cases, less so.

The rest of the interface between the parser and lexical analyzer is straightforward. The parser encodes a certain amount of state and shares it with the analyzer to allow the sort of context-dependent

analysis the grammar requires. For example, the lexical analyzer categorizes words according to the token type: reserved word (in the appropriate context), word, assignment statement, and so on. In order to do this, the parser has to tell it something about how far it has progressed parsing a command, whether it is processing a multiline string (sometimes called a “here-document”), whether it’s in a case statement or a conditional command, or whether it is processing an extended shell pattern or compound assignment statement.

Much of the work to recognize the end of the command substitution during the parsing stage is encapsulated into a single function (`parse_comsub`), which knows an uncomfortable amount of shell syntax and duplicates rather more of the token-reading code than is optimal. This function has to know about here documents, shell comments, metacharacters and word boundaries, quoting, and when reserved words are acceptable (so it knows when it’s in a case statement); it took a while to get that right.

When expanding a command substitution during word expansion, `bash` uses the parser to find the correct end of the construct. This is similar to turning a string into a command for `eval`, but in this case the command isn’t terminated by the end of the string. In order to make this work, the parser must recognize a right parenthesis as a valid command terminator, which leads to special cases in a number of grammar productions and requires the lexical analyzer to flag a right parenthesis (in the appropriate context) as denoting EOF. The parser also has to save and restore parser state before recursively invoking `yyparse`, since a command substitution can be parsed and executed as part of expanding a prompt string in the middle of reading a command. Since the input functions implement read-ahead, this function must finally take care of rewinding the `bash` input pointer to the right spot, whether `bash` is reading input from a string, a file, or the terminal using `readline`. This is important not only so that input is not lost, but so the command substitution expansion functions construct the correct string for execution.

Similar problems are posed by programmable word completion, which allows arbitrary commands to be executed while parsing another command, and solved by saving and restoring parser state around invocations.

Quoting is also a source of incompatibility and debate. Twenty years after the publication of the first Posix shell standard, members of the standards working group are still debating the proper behavior of obscure quoting. As before, the Bourne shell is no help other than as a reference implementation to observe behavior.

The parser returns a single C structure representing a command (which, in the case of compound commands like loops, may include other commands in turn) and passes it to the next stage of the shell’s operation: word expansion. The command structure is composed of command objects and lists of words. Most of the word lists are subject to various transformations, depending on their context, as explained in the following sections.

3.5 Word Expansions

After parsing, but before execution, many of the words produced by the parsing stage are subjected to one or more word expansions, so that (for example) `$OSTYPE` is replaced with the string `"linux-gnu"`.

Parameter and Variable Expansions

Variable expansions are the ones users find most familiar. Shell variables are barely typed, and, with few exceptions, are treated as strings. The expansions expand and transform these strings into new

words and word lists.

There are expansions that act on the variable's value itself. Programmers can use these to produce substrings of a variable's value, the value's length, remove portions that match a specified pattern from the beginning or end, replace portions of the value matching a specified pattern with a new string, or modify the case of alphabetic characters in a variable's value.

In addition, there are expansions that depend on the state of a variable: different expansions or assignments happen based on whether or not the variable is set. For instance, `${parameter:-word}` will expand to `parameter` if it's set, and `word` if it's not set or set to the empty string.

And Many More

Bash does many other kinds of expansion, each of which has its own quirky rules. The first in processing order is brace expansion, which turns:

```
pre{one,two,three}post
```

into:

```
preonepost pretwopost prethreepost
```

There is also command substitution, which is a nice marriage of the shell's ability to run commands and manipulate variables. The shell runs a command, collects the output, and uses that output as the value of the expansion.

One of the problems with command substitution is that it runs the enclosed command immediately and waits for it to complete: there's no easy way for the shell to send input to it. Bash uses a feature named process substitution, a sort of combination of command substitution and shell pipelines, to compensate for these shortcomings. Like command substitution, bash runs a command, but lets it run in the background and doesn't wait for it to complete. The key is that bash opens a pipe to the command for reading or writing and exposes it as a filename, which becomes the result of the expansion.

Next is tilde expansion. Originally intended to turn `~alan` into a reference to Alan's home directory, it has grown over the years into a way to refer to a large number of different directories.

Finally, there is arithmetic expansion. `$(expression)` causes `expression` to be evaluated according to the same rules as C language expressions. The result of the expression becomes the result of the expansion.

Variable expansion is where the difference between single and double quotes becomes most apparent. Single quotes inhibit all expansions—the characters enclosed by the quotes pass through the expansions unscathed—whereas double quotes permit some expansions and inhibit others. The word expansions and command, arithmetic, and process substitution take place—the double quotes only affect how the result is handled—but brace and tilde expansion do not.

Word Splitting

The results of the word expansions are split using the characters in the value of the shell variable `IFS` as delimiters. This is how the shell transforms a single word into more than one. Each time one of the characters in `$IFS`¹ appears in the result, bash splits the word into two. Single and double quotes both inhibit word splitting.

¹In most cases, a sequence of one of the characters.

Globbering

After the results are split, the shell interprets each word resulting from the previous expansions as a potential pattern and tries to match it against an existing filename, including any leading directory path.

Implementation

If the basic architecture of the shell parallels a pipeline, the word expansions are a small pipeline unto themselves. Each stage of word expansion takes a word and, after possibly transforming it, passes it to the next expansion stage. After all the word expansions have been performed, the command is executed.

The bash implementation of word expansions builds on the basic data structures already described. The words output by the parser are expanded individually, resulting in one or more words for each input word. The `WORD_DESC` data structure has proved versatile enough to hold all the information required to encapsulate the expansion of a single word. The flags are used to encode information for use within the word expansion stage and to pass information from one stage to the next. For instance, the parser uses a flag to tell the expansion and command execution stages that a particular word is a shell assignment statement, and the word expansion code uses flags internally to inhibit word splitting or note the presence of a quoted null string ("x", where x is unset or has a null value). Using a single character string for each word being expanded, with some kind of character encoding to represent additional information, would have proved much more difficult.

As with the parser, the word expansion code handles characters whose representation requires more than a single byte. For example, the variable length expansion (`$(variable)`) counts the length in characters, rather than bytes, and the code can correctly identify the end of expansions or characters special to expansions in the presence of multibyte characters.

3.6 Command Execution

The command execution stage of the internal bash pipeline is where the real action happens. Most of the time, the set of expanded words is decomposed into a command name and set of arguments, and passed to the operating system as a file to be read and executed with the remaining words passed as the rest of the elements of `argv`.

The description thus far has deliberately concentrated on what Posix calls simple commands—those with a command name and a set of arguments. This is the most common type of command, but bash provides much more.

The input to the command execution stage is the command structure built by the parser and a set of possibly-expanded words. This is where the real bash programming language comes into play. The programming language uses the variables and expansions discussed previously, and implements the constructs one would expect in a high-level language: looping, conditionals, alternation, grouping, selection, conditional execution based on pattern matching, expression evaluation, and several higher-level constructs specific to the shell.

Redirection

One reflection of the shell's role as an interface to the operating system is the ability to redirect input and output to and from the commands it invokes. The redirection syntax is one of the things that

reveals the sophistication of the shell's early users: until very recently, it required users to keep track of the file descriptors they were using, and explicitly specify by number any other than standard input, output, and error.

A recent addition to the redirection syntax allows users to direct the shell to choose a suitable file descriptor and assign it to a specified variable, instead of having the user choose one. This reduces the programmer's burden of keeping track of file descriptors, but adds extra processing: the shell has to duplicate file descriptors in the right place, and make sure they are assigned to the specified variable. This is another example of how information is passed from the lexical analyzer to the parser through to command execution: the analyzer classifies the word as a redirection containing a variable assignment; the parser, in the appropriate grammar production, creates the redirection object with a flag indicating assignment is required; and the redirection code interprets the flag and ensures that the file descriptor number is assigned to the correct variable.

The hardest part of implementing redirection is remembering how to undo redirections. The shell deliberately blurs the distinction between commands executed from the filesystem that cause the creation of a new process and commands the shell executes itself (builtins), but, no matter how the command is implemented, the effects of redirections should not persist beyond the command's completion². The shell therefore has to keep track of how to undo the effects of each redirection, otherwise redirecting the output of a shell builtin would change the shell's standard output. Bash knows how to undo each type of redirection, either by closing a file descriptor that it allocated, or by saving file descriptor being duplicated to and restoring it later using `dup2`. These use the same redirection objects as those created by the parser and are processed using the same functions.

Since multiple redirections are implemented as simple lists of objects, the redirections used to undo are kept in a separate list. That list is processed when a command completes, but the shell has to take care when it does so, since redirections attached to a shell function or the `."` builtin must stay in effect until that function or builtin completes. When it doesn't invoke a command, the `exec` builtin causes the undo list to simply be discarded, because redirections associated with `exec` persist in the shell environment.

The other complication is one bash brought on itself. Historical versions of the Bourne shell allowed the user to manipulate only file descriptors 0-9, reserving descriptors 10 and above for the shell's internal use. Bash relaxed this restriction, allowing a user to manipulate any descriptor up to the process's open file limit. This means that bash has to keep track of its own internal file descriptors, including those opened by external libraries and not directly by the shell, and be prepared to move them around on demand. This requires a lot of bookkeeping, some heuristics involving the close-on-exec flag, and yet another list of redirections to be maintained for the duration of a command and then either processed or discarded.

Builtin Commands

Bash makes a number of commands part of the shell itself. These commands are executed by the shell, without creating a new process.

The most common reason to make a command a builtin is to maintain or modify the shell's internal state. `cd` is a good example; one of the classic exercises for introduction to Unix classes is to explain why `cd` can't be implemented as an external command.

Bash builtins use the same internal primitives as the rest of the shell. Each builtin is implemented using a C language function that takes a list of words as arguments. The words are those output by the

²The `exec` builtin is an exception to this rule.

word expansion stage; the builtins treat them as command names and arguments. For the most part, the builtins use the same standard expansion rules as any other command, with a couple of exceptions: the bash builtins that accept assignment statements as arguments (e.g., `declare` and `export`) use the same expansion rules for the assignment arguments as those the shell uses for variable assignments. This is one place where the `flags` member of the `WORD_DESC` structure is used to pass information between one stage of the shell's internal pipeline and another.

Simple Command Execution

Simple commands are the ones most commonly encountered. The search for and execution of commands read from the filesystem, and collection of their exit status, covers many of the shell's remaining features.

Shell variable assignments (i.e., words of the form `var=value`) are a kind of simple command themselves. Assignment statements can either precede a command name or stand alone on a command line. If they precede a command, the variables are passed to the executed command in its environment (if they precede a built-in command or shell function, they persist, with a few exceptions, only as long as the builtin or function executes). If they're not followed by a command name, the assignment statements modify the shell's state.

When presented a command name that is not the name of a shell function or builtin, bash searches the filesystem for an executable file with that name. The value of the `PATH` variable is used as a colon-separated list of directories in which to search. Command names containing slashes (or other directory separators) are not looked up, but are executed directly.

When a command is found using a `PATH` search, bash saves the command name and the corresponding full pathname in a hash table, which it consults before conducting subsequent `PATH` searches. If the command is not found, bash executes a specially-named function, if it's defined, with the command name and arguments as arguments to the function. Some Linux distributions use this facility to offer to install missing commands.

If bash finds a file to execute, it forks and creates a new execution environment, and executes the program in this new environment. The execution environment is an exact duplicate of the shell environment, with minor modifications to things like signal disposition and files opened and closed by redirections.

Job Control

The shell can execute commands in the foreground, in which it waits for the command to finish and collects its exit status, or the background, where the shell immediately reads the next command. Job control is the ability to move processes (commands being executed) between the foreground and background, and to suspend and resume their execution. To implement this, bash introduces the concept of a job, which is essentially a command being executed by one or more processes. A pipeline, for instance, uses one process for each of its elements. The process group is a way to join separate processes together into a single job. The terminal has a process group ID associated with it, so the foreground process group is the one whose process group ID is the same as the terminal's.

The shell uses a few simple data structures in its job control implementation. There is a structure to represent a child process, including its process ID, its state, and the status it returned when it terminated. A pipeline is just a simple linked list of these process structures. A job is quite similar: there is a list of processes, some job state (running, suspended, exited, etc.), and the job's process group ID. The process list usually consists of a single process; only pipelines result in more than

one process being associated with a job. Each job has a unique process group ID, and the process in the job whose process ID is the same as the job's process group ID is called the process group leader. The current set of jobs is kept in an array, conceptually very similar to how it's presented to the user. The job's state and exit status are assembled by aggregating the state and exit statuses of the constituent processes.

Like several other things in the shell, the complex part about implementing job control is bookkeeping. The shell must take care to assign processes to the correct process groups, make sure that child process creation and process group assignment are synchronized, and that the terminal's process group is set appropriately, since the terminal's process group determines the foreground job (and, if it's not set back to the shell's process group, the shell itself won't be able to read terminal input). Since it's so process-oriented, it's not straightforward to implement compound commands such as `while` and `for` loops so an entire loop can be stopped and started as a unit, and few shells have done so.

Compound Commands

Compound commands consist of lists of one or more simple commands and are introduced by a keyword such as `if` or `while`. This is where the programming power of the shell is most visible and effective.

The implementation is fairly unsurprising. The parser constructs objects corresponding to the various compound commands, and interprets them by traversing the object. Each compound command is implemented by a corresponding C function that is responsible for performing the appropriate expansions, executing commands as specified, and altering the execution flow based on the command's return status. The function that implements the `for` command is illustrative. It must first expand the list of words following the `in` reserved word. The function must then iterate through the expanded words, assigning each word to the appropriate variable, then executing the list of commands in the `for` command's body. The `for` command doesn't have to alter execution based on the return status of the command, but it does have to pay attention to the effects of the `break` and `continue` builtins. Once all the words in the list have been used, the `for` command returns. As this shows, for the most part, the implementation follows the description very closely.

3.7 Lessons Learned

What I Have Found Is Important

I have spent over twenty years working on `bash`, and I'd like to think I have discovered a few things. The most important—one that I can't stress enough—is that it's vital to have detailed change logs. It's good when you can go back to your change logs and remind yourself about why a particular change was made. It's even better when you can tie that change to a particular bug report, complete with a reproducible test case, or a suggestion.

If it's appropriate, extensive regression testing is something I would recommend building into a project from the beginning. `Bash` has thousands of test cases covering virtually all of its non-interactive features. I have considered building tests for interactive features—`Posix` has them in its conformance test suite—but did not want to have to distribute the framework I judged it would need.

Standards are important. `Bash` has benefited from being an implementation of a standard. It's important to participate in the standardization of the software you're implementing. In addition to

discussions about features and their behavior, having a standard to refer to as the arbiter can work well. Of course, it can also work poorly—it depends on the standard.

External standards are important, but it's good to have internal standards as well. I was lucky enough to fall into the GNU Project's set of standards, which provide plenty of good, practical advice about design and implementation.

Good documentation is another essential. If you expect a program to be used by others, it's worth having comprehensive, clear documentation. If software is successful, there will end up being lots of documentation for it, and it's important that the developer writes the authoritative version.

There's a lot of good software out there. Use what you can: for instance, gnuilib has a lot of convenient library functions (once you can unravel them from the gnuilib framework). So do the BSDs and Mac OS X. Picasso said "Great artists steal" for a reason.

Engage the user community, but be prepared for occasional criticism, some that will be head-scratching. An active user community can be a tremendous benefit, but one consequence is that people will become very passionate. Don't take it personally.

What I Would Have Done Differently

Bash has millions of users. I've been educated about the importance of backwards compatibility. In some sense, backwards compatibility means never having to say you're sorry. The world, however, isn't quite that simple. I've had to make incompatible changes from time to time, nearly all of which generated some number of user complaints, though I always had what I considered to be a valid reason, whether that was to correct a bad decision, to fix a design misfeature, or to correct incompatibilities between parts of the shell. I would have introduced something like formal bash compatibility levels earlier.

Bash's development has never been particularly open. I have become comfortable with the idea of milestone releases (e.g., bash-4.2) and individually-released patches. There are reasons for doing this: I accommodate vendors with longer release timelines than the free software and open source worlds, and I've had trouble in the past with beta software becoming more widespread than I'd like. If I had to start over again, though, I would have considered more frequent releases, using some kind of public repository.

No such list would be complete without an implementation consideration. One thing I've considered multiple times, but never done, is rewriting the bash parser using straight recursive-descent rather than using bison. I once thought I'd have to do this in order to make command substitution conform to Posix, but I was able to resolve that issue without changes that extensive. Were I starting bash from scratch, I probably would have written a parser by hand. It certainly would have made some things easier.

3.8 Conclusions

Bash is a good example of a large, complex piece of free software. It has had the benefit of more than twenty years of development, and is mature and powerful. It runs nearly everywhere, and is used by millions of people every day, many of whom don't realize it.

Bash has been influenced by many sources, dating back to the original 7th Edition Unix shell, written by Stephen Bourne. The most significant influence is the Posix standard, which dictates a significant portion of its behavior. This combination of backwards compatibility and standards compliance has brought its own challenges.

Bash has profited by being part of the GNU Project, which has provided a movement and a framework in which bash exists. Without GNU, there would be no bash. Bash has also benefited from its active, vibrant user community. Their feedback has helped to make bash what it is today—a testament to the benefits of free software.

Berkeley DB

Margo Seltzer and Keith Bostic

Conway’s Law states that a design reflects the structure of the organization that produced it. Stretching that a bit, we might anticipate that a software artifact designed and initially produced by two people might somehow reflect, not merely the structure of the organization, but the internal biases and philosophies each brings to the table. One of us (Seltzer) has spent her career between the worlds of filesystems and database management systems. If questioned, she’ll argue the two are fundamentally the same thing, and furthermore, operating systems and database management systems are essentially both resource managers and providers of convenient abstractions. The differences are “merely” implementation details. The other (Bostic) believes in the tool-based approach to software engineering and in the construction of components based on simpler building blocks, because such systems are invariably superior to monolithic architectures in the important “-ilities”: understandability, extensibility, maintainability, testability, and flexibility.

When you combine those two perspectives, it’s not surprising to learn that together we spent much of the last two decades working on Berkeley DB—a software library that provides fast, flexible, reliable and scalable data management. Berkeley DB provides much of the same functionality that people expect from more conventional systems, such as relational databases, but packages it differently. For example, Berkeley DB provides fast data access, both keyed and sequential, as well as transaction support and recovery from failure. However, it provides those features in a library that links directly with the application that needs those services, rather than being made available by a standalone server application.

In this chapter, we’ll take a deeper look at Berkeley DB and see that it is composed of a collection of modules, each of which embodies the Unix “do one thing well” philosophy. Applications that embed Berkeley DB can use those components directly or they can simply use them implicitly via the more familiar operations to get, put, and delete data items. We’ll focus on architecture—how we got started, what we were designing, and where we’ve ended up and why. Designs can (and certainly will!) be forced to adapt and change—what’s vital is maintaining principles and a consistent vision over time. We will also briefly consider the code evolution of long-term software projects. Berkeley DB has over two decades of on-going development, and that inevitably takes its toll on good design.

4.1 In the Beginning

Berkeley DB dates back to an era when the Unix operating system was proprietary to AT&T and there were hundreds of utilities and libraries whose lineage had strict licensing constraints. Margo Seltzer was a graduate student at the University of California, Berkeley, and Keith Bostic was a member of Berkeley's Computer Systems Research Group. At the time, Keith was working on removing AT&T's proprietary software from the Berkeley Software Distribution.

The Berkeley DB project began with the modest goal of replacing the in-memory `hsearch` hash package and the on-disk `dbm/ndbm` hash packages with a new and improved hash implementation able to operate both in-memory and on disk, as well as be freely redistributed without a proprietary license. The hash library that Margo Seltzer wrote [SY91] was based on Litwin's Extensible Linear Hashing research. It boasted a clever scheme allowing a constant time mapping between hash values and page addresses, as well as the ability to handle large data—items larger than the underlying hash bucket or filesystem page size, typically four to eight kilobytes.

If hash tables were good, then Btrees and hash tables would be better. Mike Olson, also a graduate student at the University of California, Berkeley, had written a number of Btree implementations, and agreed to write one more. The three of us transformed Margo's hash software and Mike's Btree software into an access-method-agnostic API, where applications reference hash tables or Btrees via database handles that had handle methods to read and modify data.

Building on these two access methods, Mike Olson and Margo Seltzer wrote a research paper ([SO92]) describing LIBTP, a programmatic transactional library that ran in an application's address space.

The hash and Btree libraries were incorporated into the final 4BSD releases, under the name Berkeley DB 1.85. Technically, the Btree access method implements a B+link tree, however, we will use the term Btree for the rest of this chapter, as that is what the access method is called. Berkeley DB 1.85's structure and APIs will likely be familiar to anyone who has used any Linux or BSD-based system.

The Berkeley DB 1.85 library was quiescent for a few years, until 1996 when Netscape contracted with Margo Seltzer and Keith Bostic to build out the full transactional design described in the LIBTP paper and create a production-quality version of the software. This effort produced the first transactional version of Berkeley DB, version 2.0.

The subsequent history of Berkeley DB is a simpler and more traditional timeline: Berkeley DB 2.0 (1997) introduced transactions to Berkeley DB; Berkeley DB 3.0 (1999) was a re-designed version, adding further levels of abstraction and indirection to accommodate growing functionality. Berkeley DB 4.0 (2001) introduced replication and high availability, and Oracle Berkeley DB 5.0 (2010) added SQL support.

At the time of writing, Berkeley DB is the most widely used database toolkit in the world, with hundreds of millions of deployed copies running in everything from routers and browsers to mailers and operating systems. Although more than twenty years old, the Berkeley DB tool-based and object-oriented approach has allowed it to incrementally improve and re-invent itself to match the requirements of the software using it.

4.2 Architectural Overview

In this section, we'll review the Berkeley DB library's architecture, beginning with LIBTP, and highlight key aspects of its evolution.

Design Lesson 1

It is vital for any complex software package's testing and maintenance that the software be designed and built as a cooperating set of modules with well-defined API boundaries. The boundaries can (and should!) shift as needs dictate, but they always need to be there. The existence of those boundaries prevents the software from becoming an unmaintainable pile of spaghetti. Butler Lampson once said that all problems in computer science can be solved by another level of indirection. More to the point, when asked what it meant for something to be object-oriented, Lampson said it meant being able to have multiple implementations behind an API. The Berkeley DB design and implementation embody this approach of permitting multiple implementations behind a common interface, providing an object-oriented look and feel, even though the library is written in C.

Figure 4.1, which is taken from Seltzer and Olson's original paper, illustrates the original LIBTP architecture, while Figure 4.2 presents the Berkeley DB 2.0 designed architecture.

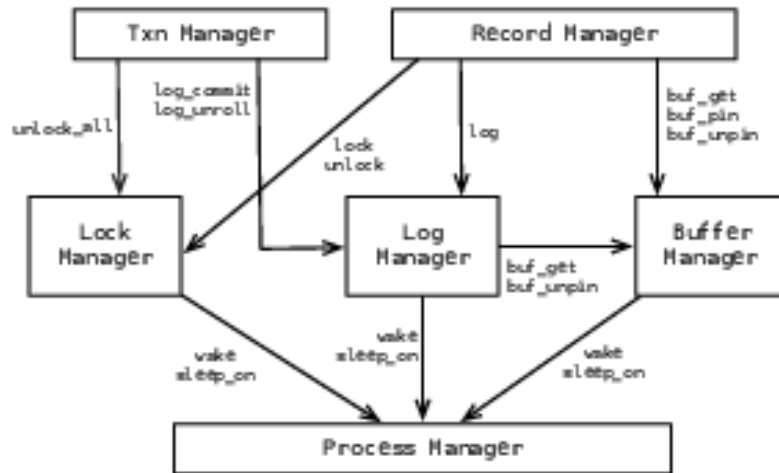


Figure 4.1: Architecture of the LIBTP Prototype System

The only significant difference between the LIBTP implementation and the Berkeley DB 2.0 design was the removal of the process manager. LIBTP required that each thread of control register itself with the library and then synchronized the individual threads/processes rather than providing subsystem level synchronization. As is discussed in Section 4.4, that original design might have served us better.

The difference between the design and the actual released db-2.0.6 architecture, shown in Figure 4.3, illustrates the reality of implementing a robust recovery manager. The recovery subsystem is shown in gray. Recovery includes both the driver infrastructure, depicted in the recovery box, as well as a set of recovery redo and undo routines that recover the operations performed by the access methods. These are represented by the circle labelled "access method recovery routines." There is a consistent design to how recovery is handled in Berkeley DB 2.0 as opposed to hand-coded logging

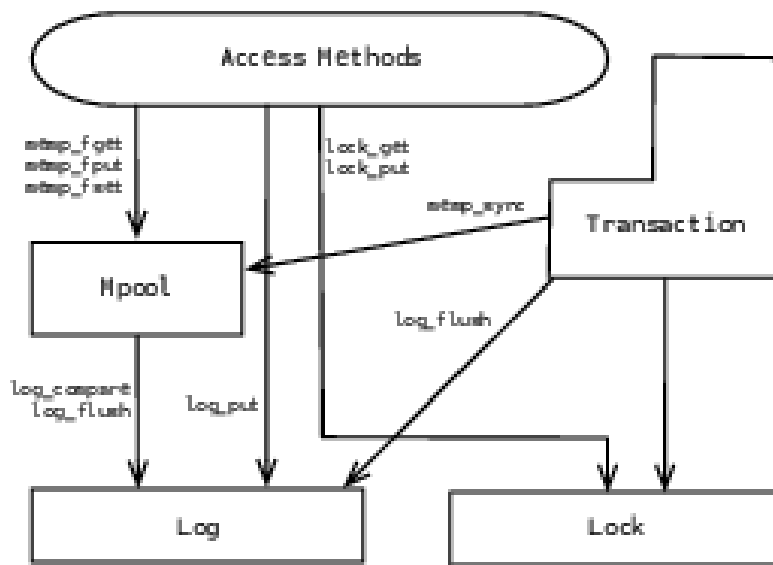


Figure 4.2: Intended Architecture for Berkeley DB-2.0.

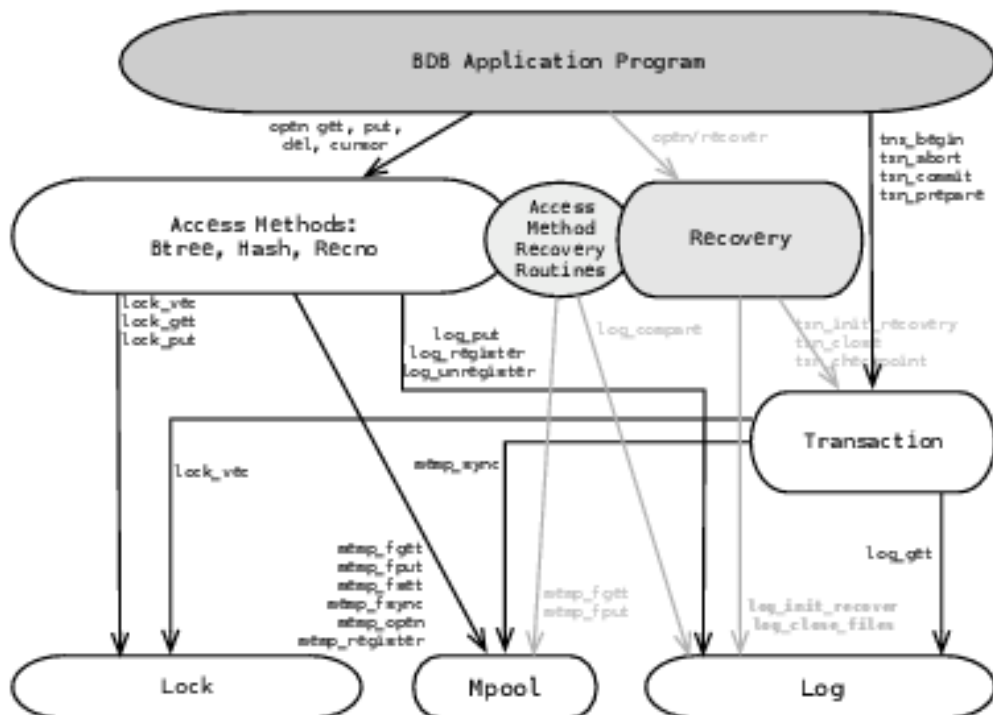


Figure 4.3: Actual Berkeley DB 2.0.6 Architecture.

and recovery routines in LIBTP particular to specific access methods. This general purpose design also produces a much richer interface between the various modules.

Figure 4.4 illustrates the Berkeley DB-5.0.21 architecture. The numbers in the diagram reference the APIs listed in the table in Table 4.1. Although the original architecture is still visible, the current architecture shows its age with the addition of new modules, the decomposition of old modules (e.g., log has become log and dbreg), and a significant increase in the number of intermodule APIs.

Over a decade of evolution, dozens of commercial releases, and hundreds of new features later, we see that the architecture is significantly more complex than its ancestors. The key things to note are: First, replication adds an entirely new layer to the system, but it does so cleanly, interacting with the rest of the system via the same APIs as does the historical code. Second, the log module is split into log and dbreg (database registration). This is discussed in more detail in Section 4.8. Third, we have placed all inter-module calls into a namespace identified with leading underscores, so that applications won't collide with our function names. We discuss this further in Design Lesson 6.

Fourth, the logging subsystem's API is now cursor based (there is no log_get API; it is replaced by the log_cursor API). Historically, Berkeley DB never had more than one thread of control reading or writing the log at any instant in time, so the library had a single notion of the current seek pointer in the log. This was never a good abstraction, but with replication it became unworkable. Just as the application API supports iteration using cursors, the log now supports iteration using cursors. Fifth, the fileop module inside of the access methods provides support for transactionally protected database create, delete, and rename operations. It took us multiple attempts to make the implementation palatable (it is still not as clean as we would like), and after reworking it numerous times, we pulled it out into its own module.

Design Lesson 2

A software design is simply one of several ways to force yourself to think through the entire problem before attempting to solve it. Skilled programmers use different techniques to this end: some write a first version and throw it away, some write extensive manual pages or design documents, others fill out a code template where every requirement is identified and assigned to a specific function or comment. For example, in Berkeley DB, we created a complete set of Unix-style manual pages for the access methods and underlying components before writing any code. Regardless of the technique used, it's difficult to think clearly about program architecture after code debugging begins, not to mention that large architectural changes often waste previous debugging effort. Software architecture requires a different mind set from debugging code, and the architecture you have when you begin debugging is usually the architecture you'll deliver in that release.

Why architect the transactional library out of components rather than tune it to a single anticipated use? There are three answers to this question. First, it forces a more disciplined design. Second, without strong boundaries in the code, complex software packages inevitably degenerate into unmaintainable piles of glop. Third, you can never anticipate all the ways customers will use your software; if you empower users by giving them access to software components, they will use them in ways you never considered.

In subsequent sections we'll consider each component of Berkeley DB, understand what it does and how it fits into the larger picture.

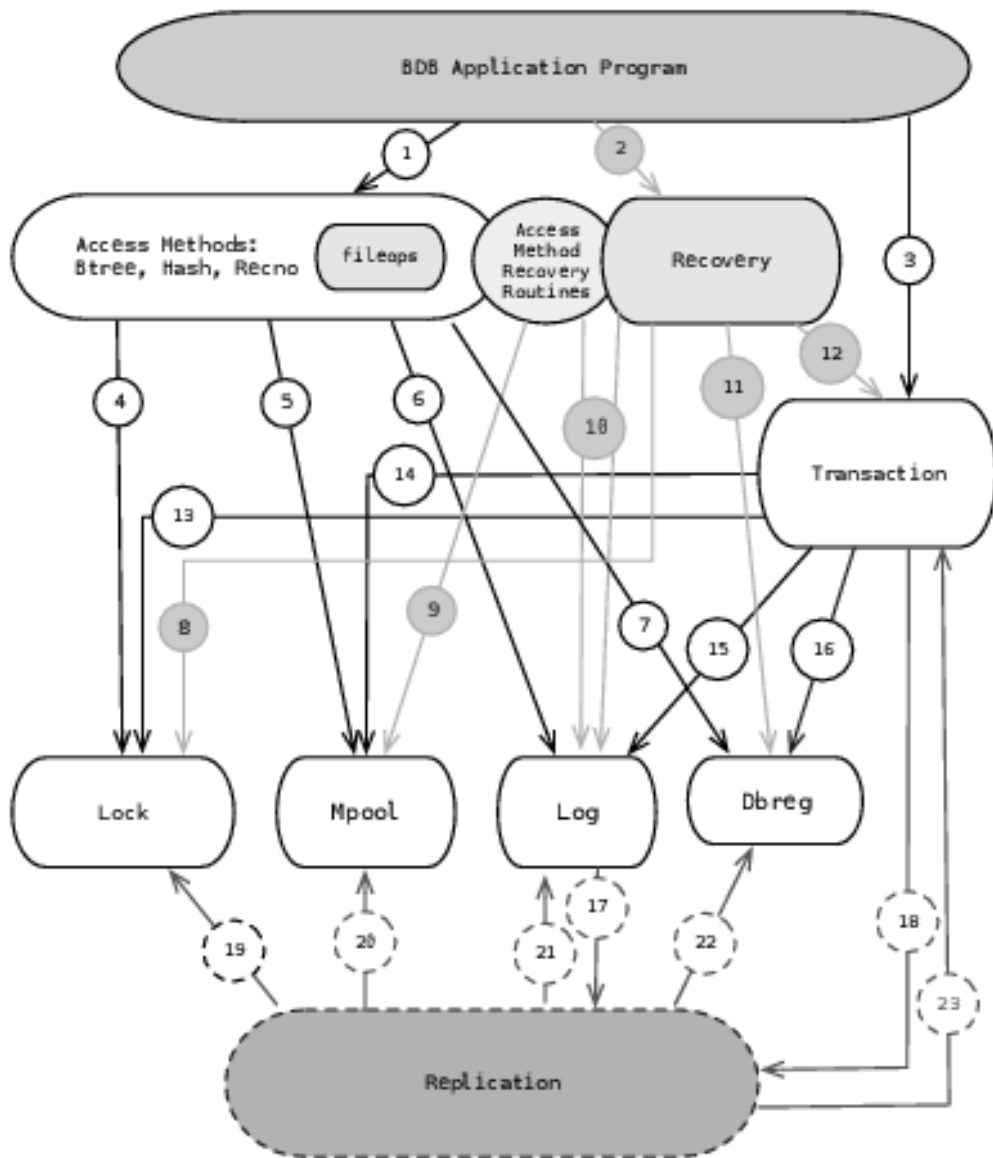


Figure 4.4: Berkeley DB-5.0.21 Architecture

4.3 The Access Methods: Btree, Hash, Recno, Queue

The Berkeley DB access methods provide both keyed lookup of, and iteration over, variable and fixed-length byte strings. Btree and Hash support variable-length key/value pairs. Recno and Queue support record-number/value pairs (where Recno supports variable-length values and Queue supports only fixed-length values).

Application APIs				
1. DBP handle operations open get put del cursor		2. DB_ENV Recovery open(... DB_RECOVER ...)		3. Transaction APIs DB_ENV->txn_begin DB_TXN->abort DB_TXN->commit DB_TXN->prepare
APIs Used by the Access Methods				
4. Into Lock __lock_downgrade __lock_vec __lock_get __lock_put	5. Into Mpool __memp_nameop __memp_fget __memp_fput __memp_fset __memp_fsync __memp_fopen __memp_fclose __memp_ftruncate __memp_extend_freelist	6. Into Log __log_print_record	7. Into Dbreg __dbreg_setup __dbreg_net_id __dbreg_revoke __dbreg_teardown __dbreg_close_id __dbreg_log_id	
Recovery APIs				
8. Into Lock __lock_getlocker __lock_get_list	9. Into Mpool __memp_fget __memp_fput __memp_fset __memp_nameop	10. Into Log __log_compare __log_open __log_earliest __log_backup __log_cursor __log_vtruncate	11. Into Dbreg __dbreg_close_files __dbreg_mark_restored __dbreg_init_recover	12. Into Txn __txn_getckp __txn_checkpoint __txn_reset __txn_recycle_id __txn_findlastckp __txn_ckp_read
APIs Used by the Transaction Module				
13. Into Lock __lock_vec __lock_downgrade	14. Into Mpool __memp_sync __memp_nameop	15. Into Log __log_cursor __log_current_lsn	16. Into Dbreg __dbreg_invalidate_files __dbreg_close_files __dbreg_log_files	
API Into the Replication System				
		17. From Log __rep_send_message __rep_bulk_message		18. From Txn __rep_lease_check __rep_txn_applied __rep_send_message
API From the Replication System				
19. Into Lock __lock_vec __lock_get __lock_id	20. Into Mpool __memp_fclose __memp_fget __memp_fput __memp_fsync	21. Into Log __log_get_stable_lsn __log_cursor __log_newfile __log_flush __log_rep_put __log_zero __log_vtruncate	22. Into Dbreg __dbreg_mark_restored __dbreg_invalidate_files __dbreg_close_files	23. Into Txn __txn_recycle_id __txn_begin __txn_recover __txn_getckp __txn_updateckp

Table 4.1: Berkeley DB 5.0.21 APIs

The main difference between Btree and Hash access methods is that Btree offers locality of reference for keys, while Hash does not. This implies that Btree is the right access method for almost all data sets; however, the Hash access method is appropriate for data sets so large that not even the Btree indexing structures fit into memory. At that point, it's better to use the memory for data than for indexing structures. This trade-off made a lot more sense in 1990 when main memory was typically much smaller than today.

The difference between Recno and Queue is that Queue supports record-level locking, at the cost of requiring fixed-length values. Recno supports variable-length objects, but like Btree and Hash, supports only page-level locking.

We originally designed Berkeley DB such that the CRUD functionality (create, read, update and delete) was key-based and the primary interface for applications. We subsequently added cursors to support iteration. That ordering led to the confusing and wasteful case of largely duplicated code paths inside the library. Over time, this became unmaintainable and we converted all keyed operations

to cursor operations (keyed operations now allocate a cached cursor, perform the operation, and return the cursor to the cursor pool). This is an application of one of the endlessly-repeated rules of software development: don't optimize a code path in any way that detracts from clarity and simplicity until you know that it's necessary to do so.

Design Lesson 3

Software architecture does not age gracefully. Software architecture degrades in direct proportion to the number of changes made to the software: bug fixes corrode the layering and new features stress design. Deciding when the software architecture has degraded sufficiently that you should re-design or re-write a module is a hard decision. On one hand, as the architecture degrades, maintenance and development become more difficult and at the end of that path is a legacy piece of software maintainable only by having an army of brute-force testers for every release, because nobody understands how the software works inside. On the other hand, users will bitterly complain over the instability and incompatibilities that result from fundamental changes. As a software architect, your only guarantee is that someone will be angry with you no matter which path you choose.

We omit detailed discussions of the Berkeley DB access method internals; they implement fairly well-known Btree and hashing algorithms (Recno is a layer on top of the Btree code, and Queue is a file block lookup function, albeit complicated by the addition of record-level locking).

4.4 The Library Interface Layer

Over time, as we added additional functionality, we discovered that both applications and internal code needed the same top-level functionality (for example, a table join operation uses multiple cursors to iterate over the rows, just as an application might use a cursor to iterate over those same rows).

Design Lesson 4

It doesn't matter how you name your variables, methods, functions, or what comments or code style you use; that is, there are a large number of formats and styles that are "good enough." What does matter, and matters very much, is that naming and style be consistent. Skilled programmers derive a tremendous amount of information from code format and object naming. You should view naming and style inconsistencies as some programmers investing time and effort to lie to the other programmers, and vice versa. Failing to follow house coding conventions is a firing offense.

For this reason, we decomposed the access method APIs into precisely defined layers. These layers of interface routines perform all of the necessary generic error checking, function-specific error checking, interface tracking, and other tasks such as automatic transaction management. When applications call into Berkeley DB, they call the first level of interface routines based on methods in the object handles. (For example, `__dbc_put_pp`, is the interface call for the Berkeley DB cursor "put" method, to update a data item. The `"_pp"` is the suffix we use to identify all functions that an application can call.)

One of the Berkeley DB tasks performed in the interface layer is tracking what threads are running inside the Berkeley DB library. This is necessary because some internal Berkeley DB operations may be performed only when no threads are running inside the library. Berkeley DB tracks threads in the library by marking that a thread is executing inside the library at the beginning of every library API and clearing that flag when the API call returns. This entry/exit checking is always performed in the interface layer, as is a similar check to determine if the call is being performed in a replicated environment.

The obvious question is “why not pass a thread identifier into the library, wouldn’t that be easier?” The answer is yes, it would be a great deal easier, and we surely wish we’d done just that. But, that change would have modified every single Berkeley DB application, most of every application’s calls into Berkeley DB, and in many cases would have required application re-structuring.

Design Lesson 5

Software architects must choose their upgrade battles carefully: users will accept minor changes to upgrade to new releases (if you guarantee compile-time errors, that is, obvious failures until the upgrade is complete; upgrade changes should never fail in subtle ways). But to make truly fundamental changes, you must admit it’s a new code base and requires a port of your user base. Obviously, new code bases and application ports are not cheap in time or resources, but neither is angering your user base by telling them a huge overhaul is really a minor upgrade.

Another task performed in the interface layer is transaction generation. The Berkeley DB library supports a mode where every operation takes place in an automatically generated transaction (this saves the application having to create and commit its own explicit transactions). Supporting this mode requires that every time an application calls through the API without specifying its own transaction, a transaction is automatically created.

Finally, all Berkeley DB APIs require argument checking. In Berkeley DB there are two flavors of error checking—generic checks to determine if our database has been corrupted during a previous operation or if we are in the midst of a replication state change (for example, changing which replica allows writes). There are also checks specific to an API: correct flag usage, correct parameter usage, correct option combinations, and any other type of error we can check before actually performing the requested operation.

This API-specific checking is all encapsulated in functions suffixed with `_arg`. Thus, the error checking specific to the cursor put method is located in the function `__dbc_put_arg`, which is called by the `__dbc_put_pp` function.

Finally, when all the argument verification and transaction generation is complete, we call the worker method that actually performs the operation (in our example, it would be `__dbc_put`), which is the same function we use when calling the cursor put functionality internally.

This decomposition evolved during a period of intense activity, when we were determining precisely what actions we needed to take when working in replicated environments. After iterating over the code base some non-trivial number of times, we pulled apart all this preamble checking to make it easier to change the next time we identified a problem with it.

4.5 The Underlying Components

There are four components underlying the access methods: a buffer manager, a lock manager, a log manager and a transaction manager. We'll discuss each of them separately, but they all have some common architectural features.

First, all of the subsystems have their own APIs, and initially each subsystem had its own object handle with all methods for that subsystem based on the handle. For example, you could use Berkeley DB's lock manager to handle your own locks or to write your own remote lock manager, or you could use Berkeley DB's buffer manager to handle your own file pages in shared memory. Over time, the subsystem-specific handles were removed from the API in order to simplify Berkeley DB applications. Although the subsystems are still individual components that can be used independently of the other subsystems, they now share a common object handle, the DB_ENV "environment" handle. This architectural feature enforces layering and generalization. Even though the layer moves from time-to-time, and there are still a few places where one subsystem reaches across into another subsystem, it is good discipline for programmers to think about the parts of the system as separate software products in their own right.

Second, all of the subsystems (in fact, all Berkeley DB functions) return error codes up the call stack. As a library, Berkeley DB cannot step on the application's name space by declaring global variables, not to mention that forcing errors to return in a single path through the call stack enforces good programmer discipline.

Design Lesson 6

In library design, respect for the namespace is vital. Programmers who use your library should not need to memorize dozens of reserved names for functions, constants, structures, and global variables to avoid naming collisions between an application and the library.

Finally, all of the subsystems support shared memory. Because Berkeley DB supports sharing databases between multiple running processes, all interesting data structures have to live in shared memory. The most significant implication of this choice is that in-memory data structures must use base address and offset pairs instead of pointers in order for pointer-based data structures to work in the context of multiple processes. In other words, instead of indirecting through a pointer, the Berkeley DB library must create a pointer from a base address (the address at which the shared memory segment is mapped into memory) plus an offset (the offset of a particular data structure in that mapped-in segment). To support this feature, we wrote a version of the Berkeley Software Distribution queue package that implemented a wide variety of linked lists.

4.6 The Buffer Manager: Mpool

The Berkeley DB Mpool subsystem is an in-memory buffer pool of file pages, which hides the fact that main memory is a limited resource, requiring the library to move database pages to and from disk when handling databases larger than memory. Caching database pages in memory was what enabled the original hash library to significantly out-perform the historic hsearch and ndbm implementations.

Design Lesson 7

Before we wrote a shared-memory linked-list package, Berkeley DB engineers hand-coded a variety of different data structures in shared memory, and these implementations were fragile and difficult to debug. The shared-memory list package, modeled after the BSD list package (`queue.h`), replaced all of those efforts. Once it was debugged, we never had to debug another shared memory linked-list problem. This illustrates three important design principles: First, if you have functionality that appears more than once, write the shared functions and use them, because the mere existence of two copies of any specific functionality in your code guarantees that one of them is incorrectly implemented. Second, when you develop a set of general purpose routines, write a test suite for the set of routines, so you can debug them in isolation. Third, the harder code is to write, the more important for it to be separately written and maintained; it's almost impossible to keep surrounding code from infecting and corroding a piece of code.

Although the Berkeley DB Btree access method is a fairly traditional B+tree implementation, pointers between tree nodes are represented as page numbers, not actual in-memory pointers, because the library's implementation uses the on-disk format as its in-memory format as well. The advantage of this representation is that a page can be flushed from the cache without format conversion; the disadvantage is that traversing an index structures requires (costlier) repeated buffer pool lookups rather than (cheaper) memory indirections.

There are other performance implications that result from the underlying assumption that the in-memory representation of Berkeley DB indices is really a cache for on-disk persistent data. For example, whenever Berkeley DB accesses a cached page, it first pins the page in memory. This pin prevents any other threads or processes from evicting it from the buffer pool. Even if an index structure fits entirely in the cache and need never be flushed to disk, Berkeley DB still acquires and releases these pins on every access, because the underlying model provided by Mpool is that of a cache, not persistent storage.

The Mpool File Abstraction

Mpool assumes it sits atop a filesystem, exporting the file abstraction through the API. For example, `DB_MPOOLFILE` handles represent an on-disk file, providing methods to get/put pages to/from the file. While Berkeley DB supports temporary and purely in-memory databases, these too are referenced by `DB_MPOOLFILE` handles because of the underlying Mpool abstractions. The get and put methods are the primary Mpool APIs: get ensures a page is present in the cache, acquires a pin on the page and returns a pointer to the page. When the library is done with the page, the put call unpins the page, releasing it for eviction. Early versions of Berkeley DB did not differentiate between pinning a page for read access versus pinning a page for write access. However, in order to increase concurrency, we extended the Mpool API to allow callers to indicate their intention to update a page. This ability to distinguish read access from write access was essential to implement multi-version concurrency control. A page pinned for reading that happens to be dirty can be written to disk, while a page pinned for writing cannot, since it may be in an inconsistent state at any instant.

Write-ahead Logging

Berkeley DB uses write-ahead-logging (WAL) as its transaction mechanism to make recovery after failure possible. The term write-ahead-logging defines a policy requiring log records describing any change be propagated to disk *before* the actual data updates they describe. Berkeley DB's use of WAL as its transaction mechanism has important implications for Mpool, and Mpool must balance its design point as a generic caching mechanism with its need to support the WAL protocol.

Berkeley DB writes log sequence numbers (LSNs) on all data pages to document the log record corresponding to the most recent update to a particular page. Enforcing WAL requires that before Mpool writes any page to disk, it must verify that the log record corresponding to the LSN on the page is safely on disk. The design challenge is how to provide this functionality without requiring that all clients of Mpool use a page format identical to that used by Berkeley DB. Mpool addresses this challenge by providing a collection of set (and get) methods to direct its behavior. The DB_MPOOLFILE method `set_lsn_offset` provides a byte offset into a page, indicating where Mpool should look for an LSN to enforce WAL. If the method is never called, Mpool does not enforce the WAL protocol. Similarly, the `set_clearlen` method tells Mpool how many bytes of a page represent metadata that should be explicitly cleared when a page is created in the cache. These APIs allow Mpool to provide the functionality necessary to support Berkeley DB's transactional requirements, without forcing all users of Mpool to do so.

Design Lesson 8

Write-ahead logging is another example of providing encapsulation and layering, even when the functionality is never going to be useful to another piece of software: after all, how many programs care about LSNs in the cache? Regardless, the discipline is useful and makes the software easier to maintain, test, debug and extend.

4.7 The Lock Manager: Lock

Like Mpool, the lock manager was designed as a general-purpose component: a hierarchical lock manager (see [GLPT76]), designed to support a hierarchy of objects that can be locked (such as individual data items), the page on which a data item lives, the file in which a data item lives, or even a collection of files. As we describe the features of the lock manager, we'll also explain how Berkeley DB uses them. However, as with Mpool, it's important to remember that other applications can use the lock manager in completely different ways, and that's OK—it was designed to be flexible and support many different uses.

The lock manager has three key abstractions: a “locker” that identifies on whose behalf a lock is being acquired, a “lock_object” that identifies the item being locked, and a “conflict matrix”.

Lockers are 32-bit unsigned integers. Berkeley DB divides this 32-bit name space into transactional and non-transactional lockers (although that distinction is transparent to the lock manager). When Berkeley DB uses the lock manager, it assigns locker IDs in the range 0 to 0x7ffffff to non-transactional lockers and the range 0x80000000 to 0xffffffff to transactions. For example, when an application opens a database, Berkeley DB acquires a long-term read lock on that database to ensure no other thread of control removes or renames it while it is in-use. As this is a long-term lock, it does not belong to any transaction and the locker holding this lock is non-transactional.

Any application using the lock manager needs to assign locker ids, so the lock manager API provides both `DB_ENV->lock_id` and `DB_ENV->lock_id_free` calls to allocate and deallocate lockers. So applications need not implement their own locker ID allocator, although they certainly can.

Lock Objects

Lock objects are arbitrarily long opaque byte-strings that represent the objects being locked. When two different lockers want to lock a particular object, they use the same opaque byte string to reference that object. That is, it is the application's responsibility to agree on conventions for describing objects in terms of opaque byte strings.

For example, Berkeley DB uses a `DB_LOCK_ILOCK` structure to describe its database locks. This structure contains three fields: a file identifier, a page number, and a type.

In almost all cases, Berkeley DB needs to describe only the particular file and page it wants to lock. Berkeley DB assigns a unique 32-bit number to each database at create time, writes it into the database's metadata page, and then uses it as the database's unique identifier in the Mpool, locking, and logging subsystems. This is the `fileid` to which we refer in the `DB_LOCK_ILOCK` structure. Not surprisingly, the page number indicates which page of the particular database we wish to lock. When we reference page locks, we set the type field of the structure to `DB_PAGE_LOCK`. However, we can also lock other types of objects as necessary. As mentioned earlier, we sometimes lock a database handle, which requires a `DB_HANDLE_LOCK` type. The `DB_RECORD_LOCK` type lets us perform record level locking in the queue access method, and the `DB_DATABASE_LOCK` type lets us lock an entire database.

Design Lesson 9

Berkeley DB's choice to use page-level locking was made for good reasons, but we've found that choice to be problematic at times. Page-level locking limits the concurrency of the application as one thread of control modifying a record on a database page will prevent other threads of control from modifying other records on the same page, while record-level locks permit such concurrency as long as the two threads of control are not modifying the same record. Page-level locking enhances stability as it limits the number of recovery paths that are possible (a page is always in one of a couple of states during recovery, as opposed to the infinite number of possible states a page might be in if multiple records are being added and deleted to a page). As Berkeley DB was intended for use as an embedded system where no database administrator would be available to fix things should there be corruption, we chose stability over increased concurrency.

The Conflict Matrix

The last abstraction of the locking subsystem we'll discuss is the conflict matrix. A conflict matrix defines the different types of locks present in the system and how they interact. Let's call the entity holding a lock, the holder and the entity requesting a lock the requester, and let's also assume that the holder and requester have different locker ids. The conflict matrix is an array indexed by `[requester][holder]`, where each entry contains a zero if there is no conflict, indicating that the requested lock can be granted, and a one if there is a conflict, indicating that the request cannot be granted.

The lock manager contains a default conflict matrix, which happens to be exactly what Berkeley DB needs, however, an application is free to design its own lock modes and conflict matrix to suit its own purposes. The only requirement on the conflict matrix is that it is square (it has the same number of rows and columns) and that the application use 0-based sequential integers to describe its lock modes (e.g., read, write, etc.). Table 4.2 shows the Berkeley DB conflict matrix.

Requester	Holder No-Lock	Read	Write	Wait	iWrite	iRead	iRW	uRead	wasWrite
No-Lock									
Read			✓		✓		✓		✓
Write		✓	✓	✓	✓	✓	✓	✓	✓
Wait									
iWrite		✓	✓					✓	✓
iRead			✓						✓
iRW		✓	✓					✓	✓
uRead			✓		✓		✓		
iwasWrite		✓	✓		✓	✓	✓		✓

Table 4.2: Read-Writer Conflict Matrix.

Supporting Hierarchical Locking

Before explaining the different lock modes in the Berkeley DB conflict matrix, let's talk about how the locking subsystem supports hierarchical locking. Hierarchical locking is the ability to lock different items within a containment hierarchy. For example, files contain pages, while pages contain individual elements. When modifying a single page element in a hierarchical locking system, we want to lock just that element; if we were modifying every element on the page, it would be more efficient to simply lock the page, and if we were modifying every page in a file, it would be best to lock the entire file. Additionally, hierarchical locking must understand the hierarchy of the containers because locking a page also says something about locking the file: you cannot modify the file that contains a page at the same time that pages in the file are being modified.

The question then is how to allow different lockers to lock at different hierarchical levels without chaos resulting. The answer lies in a construct called an intention lock. A locker acquires an intention lock on a container to indicate the intention to lock things within that container. So, obtaining a read-lock on a page implies obtaining an intention-to-read lock on the file. Similarly, to write a single page element, you must acquire an intention-to-write lock on both the page and the file. In the conflict matrix above, the iRead, iWrite, and iWR locks are all intention locks that indicate an intention to read, write or do both, respectively.

Therefore, when performing hierarchical locking, rather than requesting a single lock on something, it is necessary to request potentially many locks: the lock on the actual entity as well as intention locks on any containing entities. This need leads to the Berkeley DB `DB_ENV->lock_vec` interface, which takes an array of lock requests and grants them (or rejects them), atomically.

Although Berkeley DB doesn't use hierarchical locking internally, it takes advantage of the ability to specify different conflict matrices, and the ability to specify multiple lock requests at once. We use the default conflict matrix when providing transactional support, but a different conflict matrix to provide simple concurrent access without transaction and recovery support. We use `DB_ENV->lock_vec` to perform lock coupling, a technique that enhances the concurrency of Btree

traversals [Com79]. In lock coupling, you hold one lock only long enough to acquire the next lock. That is, you lock an internal Btree page only long enough to read the information that allows you to select and lock a page at the next level.

Design Lesson 10

Berkeley DB's general-purpose design was well rewarded when we added concurrent data store functionality. Initially Berkeley DB provided only two modes of operation: either you ran without any write concurrency or with full transaction support. Transaction support carries a certain degree of complexity for the developer and we found some applications wanted improved concurrency without the overhead of full transactional support. To provide this feature, we added support for API-level locking that allows concurrency, while guaranteeing no deadlocks. This required a new and different lock mode to work in the presence of cursors. Rather than adding special purpose code to the lock manager, we were able to create an alternate lock matrix that supported only the lock modes necessary for the API-level locking. Thus, simply by configuring the lock manager differently, we were able provide the locking support we needed. (Sadly, it was not as easy to change the access methods; there are still significant parts of the access method code to handle this special mode of concurrent access.)

4.8 The Log Manager: Log

The log manager provides the abstraction of a structured, append-only file. As with the other modules, we intended to design a general-purpose logging facility, however the logging subsystem is probably the module where we were least successful.

Design Lesson 11

When you find an architectural problem you don't want to fix "right now" and that you're inclined to just let go, remember that being nibbled to death by ducks will kill you just as surely as being trampled by elephants. Don't be too hesitant to change entire frameworks to improve software structure, and when you make the changes, don't make a partial change with the idea that you'll clean up later—do it all and then move forward. As has been often repeated, "If you don't have the time to do it right now, you won't find the time to do it later." And while you're changing the framework, write the test structure as well.

A log is conceptually quite simple: it takes opaque byte strings and writes them sequentially to a file, assigning each a unique identifier, called a log sequence number (LSN). Additionally, the log must provide efficient forward and backward traversal and retrieval by LSN. There are two tricky parts: first, the log must guarantee it is in a consistent state after any possible failure (where consistent means it contains a contiguous sequence of uncorrupted log records); second, because log records must be written to stable storage for transactions to commit, the performance of the log is usually what bounds the performance of any transactional application.

As the log is an append-only data structure, it can grow without bound. We implement the log as a collection of sequentially numbered files, so log space may be reclaimed by simply removing old log files. Given the multi-file architecture of the log, we form LSNs as pairs specifying a file number and offset within the file. Thus, given an LSN, it is trivial for the log manager to locate the record: it seeks to the given offset of the given log file and returns the record written at that location. But how does the log manager know how many bytes to return from that location?

Log Record Formatting

The log must persist per-record metadata so that, given an LSN, the log manager can determine the size of the record to return. At a minimum, it needs to know the length of the record. We prepend every log record with a log record header containing the record's length, the offset of the previous record (to facilitate backward traversal), and a checksum for the log record (to identify log corruption and the end of the log file). This metadata is sufficient for the log manager to maintain the sequence of log records, but it is not sufficient to actually implement recovery; that functionality is encoded in the contents of log records and in how Berkeley DB uses those log records.

Berkeley DB uses the log manager to write before- and after-images of data before updating items in the database [HR83]. These log records contain enough information to either redo or undo operations on the database. Berkeley DB then uses the log both for transaction abort (that is, undoing any effects of a transaction when the transaction is discarded) and recovery after application or system failure.

In addition to APIs to read and write log records, the log manager provides an API to force log records to disk (`DB_ENV->log_flush`). This allows Berkeley DB to implement write-ahead logging—before evicting a page from Mpool, Berkeley DB examines the LSN on the page and asks the log manager to guarantee that the specified LSN is on stable storage. Only then does Mpool write the page to disk.

Design Lesson 12

Mpool and Log use internal handle methods to facilitate write-ahead logging, and in some cases, the method declaration is longer than the code it runs, since the code is often comparing two integral values and nothing more. Why bother with such insignificant methods, just to maintain consistent layering? Because if your code is not so object-oriented as to make your teeth hurt, it is not object-oriented enough. Every piece of code should do a small number of things and there should be a high-level design encouraging programmers to build functionality out of smaller chunks of functionality, and so on. If there's anything we have learned about software development in the past few decades, it is that our ability to build and maintain significant pieces of software is fragile. Building and maintaining significant pieces of software is difficult and error-prone, and as the software architect, you must do everything that you can, as early as you can, as often as you can, to maximize the information conveyed in the structure of your software.

Berkeley DB imposes structure on the log records to facilitate recovery. Most Berkeley DB log records describe transactional updates. Thus, most log records correspond to page modifications to a database, performed on behalf of a transaction. This description provides the basis for identifying what metadata Berkeley DB must attach to each log record: a database, a transaction, and a record type. The transaction identifier and record type fields are present in every record at the same location.

This allows the recovery system to extract a record type and dispatch the record to an appropriate handler that can interpret the record and perform appropriate actions. The transaction identifier lets the recovery process identify the transaction to which a log record belongs, so that during the various stages of recovery, it knows whether the record can be ignored or must be processed.

Breaking the Abstraction

There are also a few “special” log records. Checkpoint records are, perhaps, the most familiar of those special records. Checkpointing is the process of making the on-disk state of the database consistent as of some point in time. In other words, Berkeley DB aggressively caches database pages in Mpool for performance. However, those pages must eventually get written to disk and the sooner we do so, the more quickly we will be able to recover in the case of application or system failure. This implies a trade-off between the frequency of checkpointing and the length of recovery: the more frequently a system takes checkpoints, the more quickly it will be able to recover. Checkpointing is a transaction function, so we’ll describe the details of checkpointing in the next section. For the purposes of this section, we’ll talk about checkpoint records and how the log manager struggles between being a stand-alone module and a special-purpose Berkeley DB component.

In general, the log manager, itself, has no notion of record types, so in theory, it should not distinguish between checkpoint records and other records—they are simply opaque byte strings that the log manager writes to disk. In practice, the log maintains metadata revealing that it does understand the contents of some records. For example, during log startup, the log manager examines all the log files it can find to identify the most recently written log file. It assumes that all log files prior to that one are complete and intact, and then sets out to examine the most recent log file and determine how much of it contains valid log records. It reads from the beginning of a log file, stopping if/when it encounters a log record header that does not checksum properly, which indicates either the end of the log or the beginning of log file corruption. In either case, it determines the logical end of log.

During this process of reading the log to find the current end, the log manager extracts the Berkeley DB record type, looking for checkpoint records. It retains the position of the last checkpoint record it finds in log manager metadata as a “favor” to the transaction system. That is, the transaction system needs to find the last checkpoint, but rather than having both the log manager and transaction manager read the entire log file to do so, the transaction manager delegates that task to the log manager. This is a classic example of violating abstraction boundaries in exchange for performance.

What are the implications of this tradeoff? Imagine that a system other than Berkeley DB is using the log manager. If it happens to write the value corresponding to the checkpoint record type in the same position that Berkeley DB places its record type, then the log manager will identify that record as a checkpoint record. However, unless the application asks the log manager for that information (by directly accessing `cached_ckp_lsn` field in the log metadata), this information never affects anything. In short, this is either a harmful layering violation or a savvy performance optimization.

File management is another place where the separation between the log manager and Berkeley DB is fuzzy. As mentioned earlier, most Berkeley DB log records have to identify a database. Each log record could contain the full filename of the database, but that would be expensive in terms of log space, and clumsy, because recovery would have to map that name to some sort of handle it could use to access the database (either a file descriptor or a database handle). Instead, Berkeley DB identifies databases in the log by an integer identifier, called a log file id, and implements a set of functions, called `dbreg` (for “database registration”), to maintain mappings between filenames and log file ids. The persistent version of this mapping (with the record type `DBREG_REGISTER`) is written

to log records when the database is opened. However, we also need in-memory representations of this mapping to facilitate transaction abort and recovery. What subsystem should be responsible for maintaining this mapping?

In theory, the file to log-file-id mapping is a high-level Berkeley DB function; it does not belong to any of the subsystems, which were intended to be ignorant of the larger picture. In the original design, this information was left in the logging subsystems data structures because the logging system seemed like the best choice. However, after repeatedly finding and fixing bugs in the implementation, the mapping support was pulled out of the logging subsystem code and into its own small subsystem with its own object-oriented interfaces and private data structures. (In retrospect, this information should logically have been placed with the Berkeley DB environment information itself, outside of any subsystem.)

Design Lesson 13

There is rarely such thing as an unimportant bug. Sure, there's a typo now and then, but usually a bug implies somebody didn't fully understand what they were doing and implemented the wrong thing. When you fix a bug, don't look for the symptom: look for the underlying cause, the misunderstanding, if you will, because that leads to a better understanding of the program's architecture as well as revealing fundamental underlying flaws in the design itself.

4.9 The Transaction Manager: Txn

Our last module is the transaction manager, which ties together the individual components to provide the transactional ACID properties of atomicity, consistency, isolation, and durability. The transaction manager is responsible for beginning and completing (either committing or aborting) transactions, coordinating the log and buffer managers to take transaction checkpoints, and orchestrating recovery. We'll visit each of these areas in order.

Jim Gray invented the ACID acronym to describe the key properties that transactions provide [Gra81]. Atomicity means that all the operations performed within a transaction appear in the database in a single unit—they either are all present in the database or all absent. Consistency means that a transaction moves the database from one logically consistent state to another. For example, if the application specifies that all employees must be assigned to a department that is described in the database, then the consistency property enforces that (with properly written transactions). Isolation means that from the perspective of a transaction, it appears that the transaction is running sequentially without any concurrent transactions running. Finally, durability means that once a transaction is committed, it stays committed—no failure can cause a committed transaction to disappear.

The transaction subsystem enforces the ACID properties, with the assistance of the other subsystems. It uses traditional transaction begin, commit, and abort operations to delimit the beginning and ending points of a transaction. It also provides a prepare call, which facilitates two phase commit, a technique for providing transactional properties across distributed transactions, which are not discussed in this chapter. Transaction begin allocates a new transaction identifier and returns a transaction handle, DB_TXN, to the application. Transaction commit writes a commit log record and then forces the log to disk (unless the application indicates that it is willing to forego durability in exchange for faster commit processing), ensuring that even in the presence of failure, the transaction

will be committed. Transaction abort reads backwards through the log records belonging to the designated transaction, undoing each operation that the transaction had done, returning the database to its pre-transaction state.

Checkpoint Processing

The transaction manager is also responsible for taking checkpoints. There are a number of different techniques in the literature for taking checkpoints [HR83]. Berkeley DB uses a variant of fuzzy checkpointing. Fundamentally, checkpointing involves writing buffers from Mpool to disk. This is a potentially expensive operation, and it's important that the system continues to process new transactions while doing so, to avoid long service disruptions. At the beginning of a checkpoint, Berkeley DB examines the set of currently active transactions to find the lowest LSN written by any of them. This LSN becomes the checkpoint LSN. The transaction manager then asks Mpool to flush its dirty buffers to disk; writing those buffers might trigger log flush operations. After all the buffers are safely on disk, the transaction manager then writes a checkpoint record containing the checkpoint LSN. This record states that all the operations described by log records before the checkpoint LSN are now safely on disk. Therefore, log records prior to the checkpoint LSN are no longer necessary for recovery. This has two implications: First, the system can reclaim any log files prior to the checkpoint LSN. Second, recovery need only process records after the checkpoint LSN, because the updates described by records prior to the checkpoint LSN are reflected in the on-disk state.

Note that there may be many log records between the checkpoint LSN and the actual checkpoint record. That's fine, since those records describe operations that logically happened after the checkpoint and that may need to be recovered if the system fails.

Recovery

The last piece of the transactional puzzle is recovery. The goal of recovery is to move the on-disk database from a potentially inconsistent state to a consistent state. Berkeley DB uses a fairly conventional two-pass scheme that corresponds loosely to "relative to the last checkpoint LSN, undo any transactions that never committed and redo any transactions that did commit." The details are a bit more involved.

Berkeley DB needs to reconstruct its mapping between log file ids and actual databases so that it can redo and undo operations on the databases. The log contains a full history of DBREG_REGISTER log records, but since databases stay open for a long time and we do not want to require that log files persist for the entire duration a database is open, we'd like a more efficient way to access this mapping. Prior to writing a checkpoint record, the transaction manager writes a collection of DBREG_REGISTER records describing the current mapping from log file ids to databases. During recovery, Berkeley DB uses these log records to reconstruct the file mapping.

When recovery begins, the transaction manager probes the log manager's `cached_ckp_lsn` value to determine the location of the last checkpoint record in the log. This record contains the checkpoint LSN. Berkeley DB needs to recover from that checkpoint LSN, but in order to do so, it needs to reconstruct the log file id mapping that existed at the checkpoint LSN; this information appears in the checkpoint *prior* to the checkpoint LSN. Therefore, Berkeley DB must look for the last checkpoint record that occurs before the checkpoint LSN. Checkpoint records contain, not only the checkpoint LSN, but the LSN of the previous checkpoint to facilitate this process. Recovery begins at the most recent checkpoint and using the `prev_lsn` field in each checkpoint record, traverses checkpoint

records backwards through the log until it finds a checkpoint record appearing before the checkpoint LSN. Algorithmically:

```
ckp_record = read (cached_ckp_lsn)
ckp_lsn = ckp_record.checkpoint_lsn
cur_lsn = ckp_record.my_lsn
while (cur_lsn > ckp_lsn) {
    ckp_record = read (ckp_record.prev_ckp)
    cur_lsn = ckp_record.my_lsn
}
```

Starting with the checkpoint selected by the previous algorithm, recovery reads sequentially until the end of the log to reconstruct the log file id mappings. When it reaches the end of the log, its mappings should correspond exactly to the mappings that existed when the system stopped. Also during this pass, recovery keeps track of any transaction commit records encountered, recording their transaction identifiers. Any transaction for which log records appear, but whose transaction identifier does not appear in a transaction commit record, was either aborted or never completed and should be treated as aborted. When recovery reaches the end of the log, it reverses direction and begins reading backwards through the log. For each transactional log record encountered, it extracts the transaction identifier and consults the list of transactions that have committed, to determine if this record should be undone. If it finds that the transaction identifier does not belong to a committed transaction, it extracts the record type and calls a recovery routine for that log record, directing it to undo the operation described. If the record belongs to a committed transaction, recovery ignores it on the backwards pass. This backward pass continues all the way back to the checkpoint LSN¹. Finally, recovery reads the log one last time in the forward direction, this time redoing any log records belonging to committed transactions. When this final pass completes, recovery takes a checkpoint. At this point, the database is fully consistent and ready to begin running the application.

Thus, recovery can be summarized as:

1. Find the checkpoint prior to the checkpoint LSN in the most recent checkpoint
2. Read forward to restore log file id mappings and construct a list of committed transactions
3. Read backward to the checkpoint LSN, undoing all operations for uncommitted transactions
4. Read forward, redoing all operations for committed transactions
5. Checkpoint

In theory, the final checkpoint is unnecessary. In practice, it bounds the time for future recoveries and leaves the database in a consistent state.

Design Lesson 14

Database recovery is a complex topic, difficult to write and harder to debug because recovery simply shouldn't happen all that often. In his Turing Award Lecture, Edsger Dijkstra argued that programming was inherently difficult and the beginning of wisdom is to admit we are unequal to the task. Our goal as architects and programmers is to use the tools at our disposal: design, problem decomposition, review, testing, naming and style conventions, and other good habits, to constrain programming problems to problems we *can* solve.

¹Note that we only need to go backwards to the checkpoint LSN, not the checkpoint record preceding it.

4.10 Wrapping Up

Berkeley DB is now over twenty years old. It was arguably the first general-purpose transactional key/value store and is the grandfather of the NoSQL movement. Berkeley DB continues as the underlying storage system for hundreds of commercial products and thousands of Open Source applications (including SQL, XML and NoSQL engines) and has millions of deployments across the globe. The lessons we've learned over the course of its development and maintenance are encapsulated in the code and summarized in the design tips outlined above. We offer them in the hope that other software designers and architects will find them useful.

CMake

Bill Hoffman and Kenneth Martin

In 1999 the National Library of Medicine engaged a small company called Kitware to develop a better way to configure, build, and deploy complex software across many different platforms. This work was part of the Insight Segmentation and Registration Toolkit, or ITK¹. Kitware, the engineering lead on the project, was tasked with developing a build system that the ITK researchers and developers could use. The system had to be easy to use, and allow for the most productive use of the researchers' programming time. Out of this directive emerged CMake as a replacement for the aging autoconf/libtool approach to building software. It was designed to address the weaknesses of existing tools while maintaining their strengths.

In addition to a build system, over the years CMake has evolved into a family of development tools: CMake, CTest, CPack, and CDash. CMake is the build tool responsible for building software. CTest is a test driver tool, used to run regression tests. CPack is a packaging tool used to create platform-specific installers for software built with CMake. CDash is a web application for displaying testing results and performing continuous integration testing.

5.1 CMake History and Requirements

When CMake was being developed, the normal practice for a project was to have a configure script and Makefiles for Unix platforms, and Visual Studio project files for Windows. This duality of build systems made cross-platform development very tedious for many projects: the simple act of adding a new source file to a project was painful. The obvious goal for developers was to have a single unified build system. The developers of CMake had experience with two approaches of solving the unified build system problem.

One approach was the VTK build system of 1999. That system consisted of a configure script for Unix and an executable called pcmaker for Windows. pcmaker was a C program that read in Unix Makefiles and created NMake files for Windows. The binary executable for pcmaker was checked into the VTK CVS system repository. Several common cases, like adding a new library, required changing that source and checking in a new binary. Although this was a unified system in some sense, it had many shortcomings.

¹<http://www.itk.org/>

The other approach the developers had experience with was a gmake based build system for TargetJr. TargetJr was a C++ computer vision environment originally developed on Sun workstations. Originally TargetJr used the imake system to create Makefiles. However, at some point, when a Windows port was needed, the gmake system was created. Both Unix compilers and Windows compilers could be used with this gmake-based system. The system required several environment variables to be set prior to running gmake. Failure to have the correct environment caused the system to fail in ways that were difficult to debug, especially for end users.

Both of these systems suffered from a serious flaw: they forced Windows developers to use the command line. Experienced Windows developers prefer to use integrated development environments (IDEs). This would encourage Windows developers to create IDE files by hand and contribute them to the project, creating the dual build system again. In addition to the lack of IDE support, both of the systems described above made it extremely difficult to combine software projects. For example, VTK (Chapter 24) had very few modules for reading images mostly because the build system made it very difficult to use libraries like libtiff and libjpeg.

It was decided that a new build system would be developed for ITK and C++ in general. The basic constraints of the new build system would be as follows:

- Depend only on a C++ compiler being installed on the system.
- It must be able to generate Visual Studio IDE input files.
- It must be easy to create the basic build system targets, including static libraries, shared libraries, executables, and plugins.
- It must be able to run build time code generators.
- It must support separate build trees from the source tree.
- It must be able to perform system introspection, i.e., be able to determine automatically what the target system could and could not do.
- It must do dependency scanning of C/C++ header files automatically.
- All features would need to work consistently and equally well on all supported platforms.

In order to avoid depending on any additional libraries and parsers, CMake was designed with only one major dependency, the C++ compiler (which we can safely assume we have if we're building C++ code). At the time, building and installing scripting languages like Tcl was difficult on many popular UNIX and Windows systems. It can still be an issue today on modern supercomputers and secured computers with no Internet connection, so it can still be difficult to build third-party libraries. Since the build system is such a basic requirement for a package, it was decided that no additional dependencies would be introduced into CMake. This did limit CMake to creating its own simple language, which is a choice that still causes some people to dislike CMake. However, at the time the most popular embedded language was Tcl. If CMake had been a Tcl-based build system, it is unlikely that it would have gained the popularity that it enjoys today.

The ability to generate IDE project files is a strong selling point for CMake, but it also limits CMake to providing only the features that the IDE can support natively. However, the benefits of providing native IDE build files outweigh the limitations. Although this decision made the development of CMake more difficult, it made the development of ITK and other projects using CMake much easier. Developers are happier and more productive when using the tools they are most familiar with. By allowing developers to use their preferred tools, projects can take best advantage of their most important resource: the developer.

All C/C++ programs require one or more of the following fundamental building blocks of software: executables, static libraries, shared libraries, and plugins. CMake had to provide the ability

to create these products on all supported platforms. Although all platforms support the creation of those products, the compiler flags used to create them vary greatly from compiler to compiler and platform to platform. By hiding the complexity and platform differences behind a simple command in CMake, developers are able to create them on Windows, Unix and Mac. This ability allows developers to focus on the project rather than on the details of how to build a shared library.

Code generators provide added complexity to a build system. From the start, VTK provided a system that automatically wrapped the C++ code into Tcl, Python, and Java by parsing the C++ header files, and automatically generating a wrapping layer. This requires a build system that can build a C/C++ executable (the wrapper generator), then run that executable at build time to create more C/C++ source code (the wrappers for the particular modules). That generated source code must then be compiled into executables or shared libraries. All of this has to happen within the IDE environments and the generated Makefiles.

When developing flexible cross-platform C/C++ software, it is important to program to the features of the system, and not to the specific system. Autotools has a model for doing system introspection which involves compiling small snippets of code, inspecting and storing the results of that compile. Since CMake was meant to be cross-platform it adopted a similar system introspection technique. This allows developers to program to the canonical system instead of to specific systems. This is important to make future portability possible, as compilers and operating systems change over time. For example, code like this:

```
#ifdef linux
// do some linux stuff
#endif
```

Is more brittle than code like this:

```
#ifdef HAS_FEATURE
// do something with a feature
#endif
```

Another early CMake requirement also came from autotools: the ability to create build trees that are separate from the source tree. This allows for multiple build types to be performed on the same source tree. It also prevents the source tree from being cluttered with build files, which often confuses version control systems.

One of the most important features of a build system is the ability to manage dependencies. If a source file is changed, then all products using that source file must be rebuilt. For C/C++ code, the header files included by a .c or .cpp file must also be checked as part of the dependencies. Tracking down issues where only some of the code that should be compiled actually gets compiled as a result of incorrect dependency information can be time consuming.

All of the requirements and features of the new build system had to work equally well on all supported platforms. CMake needed to provide a simple API for developers to create complicated software systems without having to understand platform details. In effect, software using CMake is outsourcing the build complications to the CMake team. Once the vision for the build tool was created with the basic set of requirements, implementation needed to proceed in an agile way. ITK needed a build system almost from day one. The first versions of CMake did not meet all of the requirements set out in the vision, but they were able to build on Windows and Unix.

5.2 How CMake Is Implemented

As mentioned, CMake's development languages are C and C++. To explain its internals this section will first describe the CMake process from a user's point of view, then examine its structures.

The CMake Process

CMake has two main phases. The first is the “configure” step, in which CMake processes all the input given to it and creates an internal representation of the build to be performed. Then next phase is the “generate” step. In this phase the actual build files are created.

Environment Variables (or Not)

In many build systems in 1999, and even today, shell level environment variables are used during the build of a project. It is typical that a project has a `PROJECT_ROOT` environment variable that points to the location of the root of the source tree. Environment variables are also used to point to optional or external packages. The trouble with this approach is that for the build to work, all of these external variables need to be set each time a build is performed. To solve this problem CMake has a cache file that stores all of the variables required for a build in one place. These are not shell or environment variables, but CMake variables. The first time CMake is run for a particular build tree, it creates a `CMakeCache.txt` file which stores all the persistent variables for that build. Since the file is part of the build tree, the variables will always be available to CMake during each run.

The Configure Step

During the configure step, CMake first reads the `CMakeCache.txt` if it exists from a prior run. It then reads `CMakeLists.txt`, found in the root of the source tree given to CMake. During the configure step, the `CMakeLists.txt` files are parsed by the CMake language parser. Each of the CMake commands found in the file is executed by a command pattern object. Additional `CMakeLists.txt` files can be parsed during this step by the `include` and `add_subdirectory` CMake commands. CMake has a C++ object for each of the commands that can be used in the CMake language. Some examples of commands are `add_library`, `if`, `add_executable`, `add_subdirectory`, and `include`. In effect, the entire language of CMake is implemented as calls to commands. The parser simply converts the CMake input files into command calls and lists of strings that are arguments to commands.

The configure step essentially “runs” the user-provided CMake code. After all of the code is executed, and all cache variable values have been computed, CMake has an in-memory representation of the project to be built. This will include all of the libraries, executables, custom commands, and all other information required to create the final build files for the selected generator. At this point, the `CMakeCache.txt` file is saved to disk for use in future runs of CMake.

The in-memory representation of the project is a collection of targets, which are simply things that may be built, such as libraries and executables. CMake also supports custom targets: users can define their inputs and outputs, and provide custom executables or scripts to be run at build time. CMake stores each target in a `cmTarget` object. These objects are stored in turn in the `cmMakefile` object, which is basically a storage place for all of the targets found in a given directory of the source tree. The end result is a tree of `cmMakefile` objects containing maps of `cmTarget` objects.

The Generate Step

Once the configure step has been completed, the generate step can take place. The generate step is when CMake creates the build files for the target build tool selected by the user. At this point the internal representation of targets (libraries, executables, custom targets) is converted to either an input to an IDE build tool like Visual Studio, or a set of Makefiles to be executed by make. CMake's internal representation after the configure step is as generic as possible so that as much code and data structures as possible can be shared between different built tools.

An overview of the process can be seen in Figure 5.1.

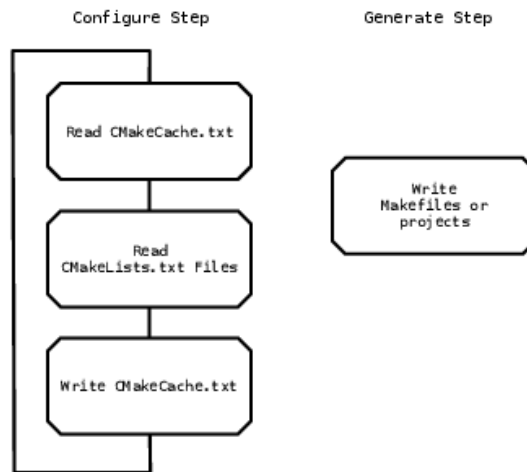


Figure 5.1: Overview of the CMake Process

CMake: The Code

CMake Objects

CMake is an object-oriented system using inheritance, design patterns and encapsulation. The major C++ objects and their relationships can be seen in Figure 5.2.

The results of parsing each CMakeLists.txt file are stored in the cmMakefile object. In addition to storing the information about a directory, the cmMakefile object controls the parsing of the CMakeLists.txt file. The parsing function calls an object that uses a lex/yacc-based parser for the CMake language. Since the CMake language syntax changes very infrequently, and lex and yacc are not always available on systems where CMake is being built, the lex and yacc output files are processed and stored in the Source directory under version control with all of the other handwritten files.

Another important class in CMake is cmCommand. This is the base class for the implementation of all commands in the CMake language. Each subclass not only provides the implementation for the command, but also its documentation. As an example, see the documentation methods on the cmUnsetCommand class:

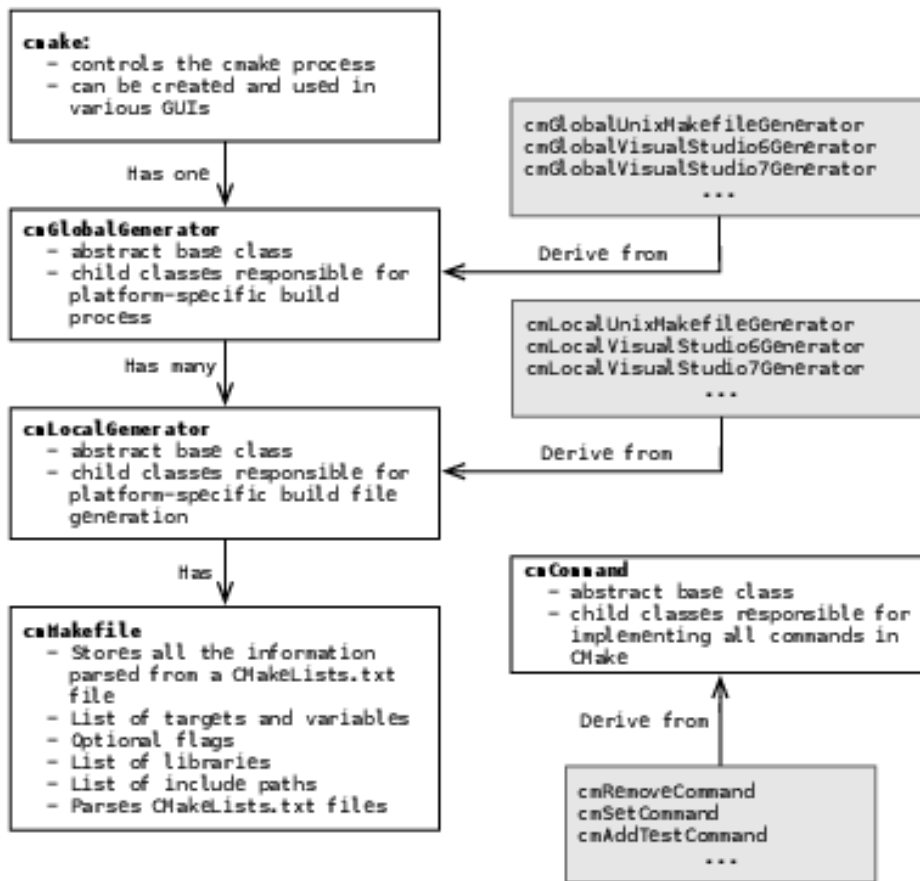


Figure 5.2: CMake Objects

```

virtual const char* GetTerseDocumentation()
{
    return "Unset a variable, cache variable, or environment variable.";
}

/**
 * More documentation.
 */

virtual const char* GetFullDocumentation()
{
    return
        " unset(<variable> [CACHE])\n"
        "Removes the specified variable causing it to become undefined. "
        "If CACHE is present then the variable is removed from the cache "
        "instead of the current scope.\n"
        "<variable> can be an environment variable such as:\n"

```

```

    " unset(ENV{LD_LIBRARY_PATH})\n"
    "in which case the variable will be removed from the current "
    "environment.";
}

```

Dependency Analysis

CMake has powerful built-in dependency analysis capabilities for individual Fortran, C and C++ source code files. Since Integrated Development Environments (IDEs) support and maintain file dependency information, CMake skips this step for those build systems. For IDE builds, CMake creates a native IDE input file, and lets the IDE handle the file level dependency information. The target level dependency information is translated to the IDE's format for specifying dependency information.

With Makefile-based builds, native make programs do not know how to automatically compute and keep dependency information up-to-date. For these builds, CMake automatically computes dependency information for C, C++ and Fortran files. Both the generation and maintenance of these dependencies are automatically done by CMake. Once a project is initially configured by CMake, users only need to run make and CMake does the rest of the work.

Although users do not need to know how CMake does this work, it may be useful to look at the dependency information files for a project. This information for each target is stored in four files called `depend.make`, `flags.make`, `build.make`, and `DependInfo.cmake`. `depend.make` stores the dependency information for all the object files in the directory. `flags.make` contains the compile flags used for the source files of this target. If they change then the files will be recompiled. `DependInfo.cmake` is used to keep the dependency information up-to-date and contains information about what files are part of the project and what languages they are in. Finally, the rules for building the dependencies are stored in `build.make`. If a dependency for a target is out of date then the depend information for that target will be recomputed, keeping the dependency information current. This is done because a change to a `.h` file could add a new dependency.

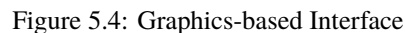
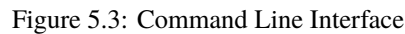
CTest and CPack

Along the way, CMake grew from a build system into a family of tools for building, testing, and packaging software. In addition to command line `cmake`, and the CMake GUI programs, CMake ships with a testing tool CTest, and a packaging tool CPack. CTest and CPack shared the same code base as CMake, but are separate tools not required for a basic build.

The `ctest` executable is used to run regression tests. A project can easily create tests for CTest to run with the `add_test` command. The tests can be run with CTest, which can also be used to send testing results to the CDash application for viewing on the web. CTest and CDash together are similar to the Hudson testing tool. They do differ in one major area: CTest is designed to allow a much more distributed testing environment. Clients can be setup to pull source from version control system, run tests, and send the results to CDash. With Hudson, client machines must give Hudson ssh access to the machine so tests can be run.

The `cpack` executable is used to create installers for projects. CPack works much like the build part of CMake: it interfaces with other packaging tools. For example, on Windows the NSIS packaging tool is used to create executable installers from a project. CPack runs the install rules of a project to create the install tree, which is then given to an installer program like NSIS. CPack also supports creating RPM, Debian `.deb` files, `.tar`, `.tar.gz` and self-extracting tar files.

The first place many users first see CMake is one of CMake’s user interface programs. CMake has two main user interface programs: a windowed Qt-based application, and a command line curses graphics-based application. These GUIs are graphical editors for the `CMakeCache.txt` file. They are relatively simple interfaces with two buttons, `configure` and `generate`, used to trigger the main phases of the CMake process. The curses-based GUI is available on Unix TTY-type platforms and Cygwin. The Qt GUI is available on all platforms. The GUIs can be seen in Figure 5.3 and Figure 5.4.



Since cache values can be modified as the commands are executed, the process of converging on a final build can be iterative. For example, turning on an option may reveal additional options. For

this reason, the GUI disables the “generate” button until the user has had a chance to see all options at least once. Each time the configure button is pressed, new cache variables that have not yet been presented to the user are displayed in red. Once there are no new cache variables created during a configure run, the generate button is enabled.

Testing CMake

Any new CMake developer is first introduced to the testing process used in CMake development. The process makes use of the CMake family of tools (CMake, CTest, CPack, and CDash). As the code is developed and checked into the version control system, continuous integration testing machines automatically build and test the new CMake code using CTest. The results are sent to a CDash server which notifies developers via email if there are any build errors, compiler warnings, or test failures.

The process is a classic continuous integration testing system. As new code is checked into the CMake repository, it is automatically tested on the platforms supported by CMake. Given the large number of compilers and platforms that CMake supports, this type of testing system is essential to the development of a stable build system.

For example, if a new developer wants to add support for a new platform, the first question he or she is asked is whether they can provide a nightly dashboard client for that system. Without constant testing, it is inevitable that new systems will stop working after some period of time.

5.3 Lessons Learned

CMake was successfully building ITK from day one, and that was the most important part of the project. If we could redo the development of CMake, not much would change. However, there are always things that could have been done better.

Backwards Compatibility

Maintaining backwards compatibility is important to the CMake development team. The main goal of the project is to make building software easier. When a project or developer chooses CMake for a build tool, it is important to honor that choice and try very hard to not break that build with future releases of CMake. CMake 2.6 implemented a policy system where changes to CMake that would break existing behavior will warn but still perform the old behavior. Each `CMakeLists.txt` file is required to specify which version of CMake they are expecting to use. Newer versions of CMake might warn, but will still build the project as older versions did.

Language, Language, Language

The CMake language is meant to be very simple. However, it is one of the major obstacles to adoption when a new project is considering CMake. Given its organic growth, the CMake language does have a few quirks. The first parser for the language was not even `lex/yacc` based but rather just a simple string parser. Given the chance to do the language over, we would have spent some time looking for a nice embedded language that already existed. Lua is the best fit that might have worked. It is very small and clean. Even if an external language like Lua was not used, I would have given more consideration to the existing language from the start.

Plugins Did Not Work

To provide the ability for extension of the CMake language by projects, CMake has a plugin class. This allows a project to create new CMake commands in C. This sounded like a good idea at the time, and the interface was defined for C so that different compilers could be used. However, with the advent of multiple API systems like 32/64 bit Windows and Linux, the compatibility of plugins became hard to maintain. While extending CMake with the CMake language is not as powerful, it avoids CMake crashing or not being able to build a project because a plugin failed to build or load.

Reduce Exposed APIs

A big lesson learned during the development of the CMake project is that you don't have to maintain backward compatibility with something that users don't have access to. Several times during the development of CMake, users and customers requested that CMake be made into a library so that other languages could be bound to the CMake functionality. Not only would this have fractured the CMake user community with many different ways to use CMake, but it would have been a huge maintenance cost for the CMake project.

Continuous Integration

C. Titus Brown and Rosangela Canino-Koning

Continuous Integration (CI) systems are systems that build and test software automatically and regularly. Though their primary benefit lies in avoiding long periods between build and test runs, CI systems can also simplify and automate the execution of many otherwise tedious tasks. These include cross-platform testing, the regular running of slow, data-intensive, or difficult-to-configure tests, verification of proper performance on legacy platforms, detection of infrequently failing tests, and the regular production of up-to-date release products. And, because build and test automation is necessary for implementing continuous integration, CI is often a first step towards a *continuous deployment* framework wherein software updates can be deployed quickly to live systems after testing.

Continuous integration is a timely subject, not least because of its prominence in the Agile software methodology. There has been an explosion of open source CI tools in recent years, in and for a variety of languages, implementing a huge range of features in the context of a diverse set of architectural models. The purpose of this chapter is to describe common sets of features implemented in continuous integration systems, discuss the architectural options available, and examine which features may or may not be easy to implement given the choice of architecture.

Below, we will briefly describe a set of systems that exemplify the extremes of architectural choices available when designing a CI system. The first, Buildbot, is a master/slave system; the second, CDash is a reporting server model; the third Jenkins, uses a hybrid model; and the fourth, Pony-Build, is a Python-based decentralized reporting server that we will use as a foil for further discussion.

6.1 The Landscape

The space of architectures for continuous integration systems seems to be dominated by two extremes: master/slave architectures, in which a central server directs and controls remote builds; and reporting architectures, in which a central server aggregates build reports contributed by clients. All of the continuous integration systems of which we are aware have chosen some combination of features from these two architectures.

Our example of a centralized architecture, Buildbot, is composed of two parts: the central server, or *buildmaster*, which schedules and coordinates builds between one or more connected clients; and the clients, or *buildslaves*, which execute builds. The buildmaster provides a central

location to which to connect, along with configuration information about which clients should execute which commands in what order. Buildslaves connect to the buildmaster and receive detailed instructions. Buildslave configuration consists of installing the software, identifying the master server, and providing connection credentials for the client to connect to the master. Builds are scheduled by the buildmaster, and output is streamed from the buildslaves to the buildmaster and kept on the master server for presentation via the Web and other reporting and notification systems.

On the opposite side of the architecture spectrum lies CDash, which is used for the Visualization Toolkit (VTK)/Insight Toolkit (ITK) projects by Kitware, Inc. CDash is essentially a reporting server, designed to store and present information received from client computers running CMake and CTest. With CDash, the clients initiate the build and test suite, record build and test results, and then connect to the CDash server to deposit the information for central reporting.

Finally, a third system, Jenkins (known as Hudson before a name change in 2011), provides both modes of operation. With Jenkins, builds can either be executed independently with the results sent to the master server; or nodes can be slaved to the Jenkins master server, which then schedules and directs the execution of builds.

Both the centralized and decentralized models have some features in common, and, as Jenkins shows, both models can co-exist in a single implementation. However, Buildbot and CDash exist in stark contrast to each other: apart from the commonalities of building software and reporting on the builds, essentially every other aspect of the architecture is different. Why?

Further, to what extent does the choice of architecture seem to make certain features easier or harder to implement? Do some features emerge naturally from a centralized model? And how extensible are the existing implementations—can they easily be modified to provide new reporting mechanisms, or scale to many packages, or execute builds and tests in a cloud environment?

What Does Continuous Integration Software Do?

The core functionality of a continuous integration system is simple: build software, run tests, and report the results. The build, test, and reporting can be performed by a script running from a scheduled task or cron job: such a script would just check out a new copy of the source code from the VCS, do a build, and then run the tests. Output would be logged to a file, and either stored in a canonical location or sent out via e-mail in case of a build failure. This is simple to implement: in UNIX, for example, this entire process can be implemented for most Python packages in a seven line script:

```
cd /tmp && \  
svn checkout http://some.project.url && \  
cd project_directory && \  
python setup.py build && \  
python setup.py test || \  
echo build failed | sendmail notification@project.domain  
cd /tmp && rm -fr project_directory
```

In Figure 6.1, the unshaded rectangles represent discrete subsystems and functionality within the system. Arrows show information flow between the various components. The cloud represents potential remote execution of build processes. The shaded rectangles represent potential coupling between the subsystems; for example, build monitoring may include monitoring of the build process itself and aspects of system health (CPU load, I/O load, memory usage, etc.)

But this simplicity is deceptive. Real-world CI systems usually do much more. In addition to initiating or receiving the results of remote build processes, continuous integration software may support any of the following additional features:

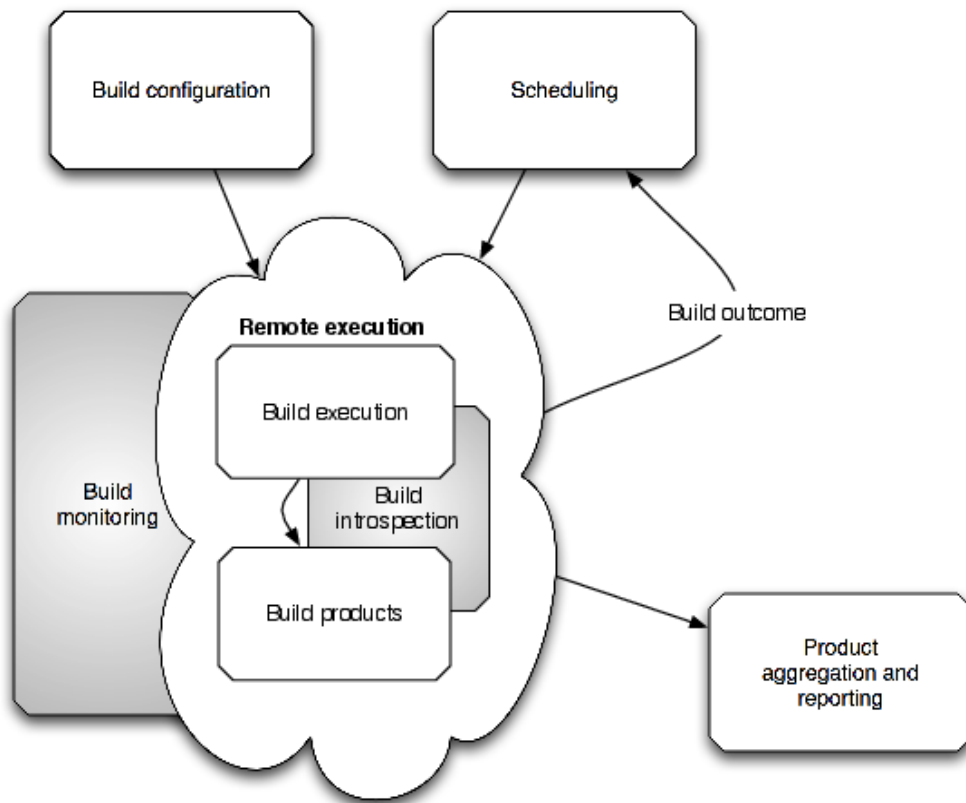


Figure 6.1: Internals of a Continuous Integration System

Checkout and update: For large projects, checking out a new copy of the source code can be costly in terms of bandwidth and time. Usually, CI systems update an existing working copy in place, which communicates only the differences from the previous update. In exchange for this savings, the system must keep track of the working copy and know how to update it, which usually means at least minimal integration with a VCS.

Abstract build recipes: A configure/build/test recipe must be written for the software in question. The underlying commands will often be different for different operating systems, e.g. Mac OS X vs. Windows vs. UNIX, which means either specialized recipes need to be written (introducing potential bugs or disconnection from the actual build environment) or some suitable level of abstraction for recipes must be provided by the CI configuration system.

Storing checkout/build/test status: It may be desirable for details of the checkout (files updated, code version), build (warnings or errors) and test (code coverage, performance, memory usage) to be stored and used for later analysis. These results can be used to answer questions across the build architectures (did the latest check-in significantly affect performance on any particular architecture?) or over history (has code coverage changed dramatically in the last month?) As with the build recipe, the mechanisms and data types for this kind of introspection are usually specific to the platform or build system.

Release packages: Builds may produce binary packages or other products that need to be made externally available. For example, developers who don't have direct access to the build machine may want to test the latest build in a specific architecture; to support this, the CI system needs to be able to transfer build products to a central repository.

Multiple architecture builds: Since one goal of continuous integration is to build on multiple architectures to test cross-platform functionality, the CI software may need to track the architecture for each build machine and link builds and build outcomes to each client.

Resource management: If a build step is resource intensive on a single machine, the CI system may want to run conditionally. For example, builds may wait for the absence of other builds or users, or delay until a particular CPU or memory load is reached.

External resource coordination: Integration tests may depend on non-local resources such as a staging database or a remote web service. The CI system may therefore need to coordinate builds between multiple machines to organize access to these resources.

Progress reports: For long build processes, regular reporting from the build may also be important. If a user is primarily interested in the results of the first 30 minutes of a 5 hour build and test, then it would be a waste of time to make them wait until the end of a run to see any results.

A high-level view of all of these potential components of a CI system is shown in Figure 6.1. CI software usually implements some subset of these components.

External Interactions

Continuous integration systems also need to interact with other systems. There are several types of potential interactions:

Build notification: The outcomes of builds generally need to be communicated to interested clients, either via pull (Web, RSS, RPC, etc.) or push notification (e-mail, Twitter, PubSubHubbub, etc.) This can include notification of all builds, or only failed builds, or builds that haven't been executed within a certain period.

Build information: Build details and products may need to be retrieved, usually via RPC or a bulk download system. For example, it may be desirable to have an external analysis system do an in-depth or more targeted analysis, or report on code coverage or performance results. In addition, an external test result repository may be employed to keep track of failing and successful tests separately from the CI system.

Build requests: External build requests from users or a code repository may need to be handled. Most VCSs have post-commit hooks that can execute an RPC call to initiate a build, for example. Or, users may request builds manually through a Web interface or other user-initiated RPC.

Remote control of the CI system: More generally, the entire runtime may be modifiable through a more-or-less well-defined RPC interface. Either ad hoc extensions or a more formally specified interface may need to be able to drive builds on specific platforms, specify alternate source branches in order to build with various patches, and execute additional builds conditionally. This is useful in support of more general workflow systems, e.g. to permit commits only after they have passed the full set of CI tests, or to test patches across a wide variety of systems before final integration. Because of the variety of bug tracker, patch systems, and other external systems in use, it may not make sense to include this logic within the CI system itself.

6.2 Architectures

Buildbot and CDash have chosen opposite architectures, and implement overlapping but distinct sets of features. Below we examine these feature sets and discuss how features are easier or harder to implement given the choice of architecture.

Implementation Model: Buildbot

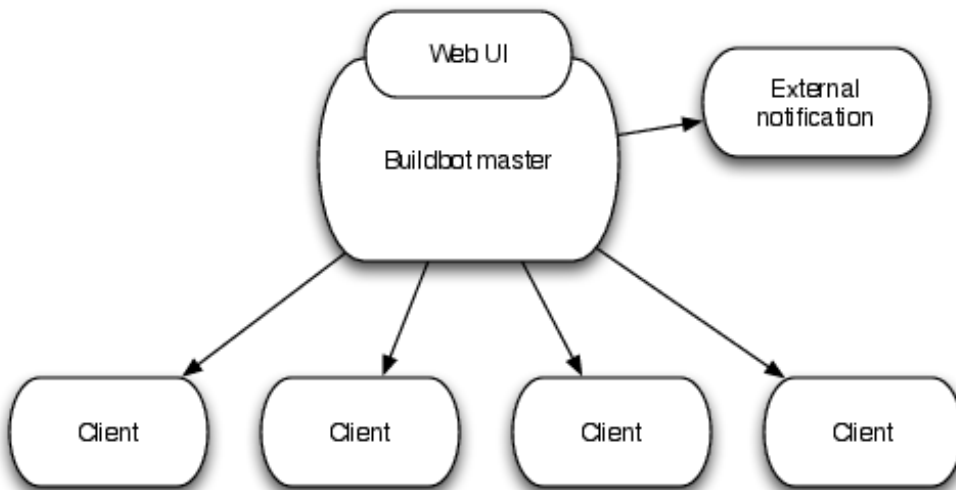


Figure 6.2: Buildbot Architecture

Buildbot uses a master/slave architecture, with a single central server and multiple build slaves. Remote execution is entirely scripted by the master server in real time: the master configuration specifies the command to be executed on each remote system, and runs them when each previous command is finished. Scheduling and build requests are not only coordinated through the master but directed entirely *by* the master. No built-in recipe abstraction exists, except for basic version control system integration (“our code is in this repository”) and a distinction between commands that operate on the build directory vs. within the build directory. OS-specific commands are typically specified directly in the configuration.

Buildbot maintains a constant connection with each builds slave, and manages and coordinates job execution between them. Managing remote machines through a persistent connection adds significant practical complexity to the implementation, and has been a long-standing source of bugs. Keeping robust long-term network connections running is not simple, and testing applications which interact with the local GUI is challenging through a network connection. OS alert windows are particularly difficult to deal with. However, this constant connection makes resource coordination and scheduling straightforward, because slaves are entirely at the disposal of the master for execution of jobs.

The kind of tight control designed into the Buildbot model makes centralized build coordination between resources very easy. Buildbot implements both master and slave locks on the buildmaster, so that builds can coordinate system-global and machine-local resources. This makes Buildbot

particularly suitable for large installations that run system integration tests, e.g. tests that interact with databases or other expensive resources.

The centralized configuration causes problems for a distributed use model, however. Each new buildslave must be explicitly allowed for in the master configuration, which makes it impossible for new buildslaves to dynamically attach to the central server and offer build services or build results. Moreover, because each build slave is entirely driven by the build master, build clients are vulnerable to malicious or accidental misconfigurations: the master literally controls the client entirely, within the client OS security restrictions.

One limiting feature of Buildbot is that there is no simple way to return build products to the central server. For example, code coverage statistics and binary builds are kept on the remote buildslave, and there is no API to transmit them to the central buildmaster for aggregation and distribution. It is not clear why this feature is absent. It may be a consequence of the limited set of command abstractions distributed with Buildbot, which are focused on executing remote commands on the build slaves. Or, it may be due to the decision to use the connection between the buildmaster and buildslave as a control system, rather than as an RPC mechanism.

Another consequence of the master/slave model and this limited communications channel is that buildslaves do not report system utilization and the master cannot be configured to be aware of high slave load.

External CPU notification of build results is handled entirely by the buildmaster, and new notification services need to be implemented within the buildmaster itself. Likewise, new build requests must be communicated directly to the buildmaster.

Implementation Model: CDash

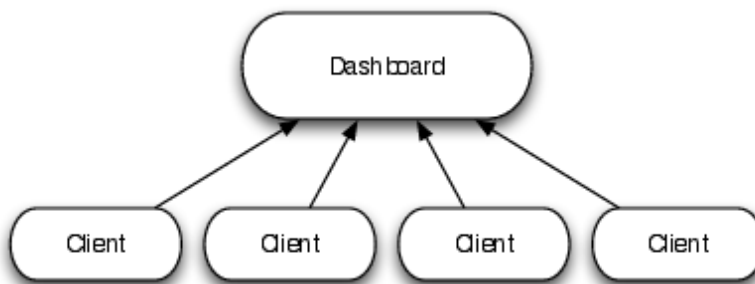


Figure 6.3: CDash Architecture

In contrast to Buildbot, CDash implements a reporting server model. In this model, the CDash server acts as a central repository for information on remotely executed builds, with associated reporting on build and test failures, code coverage analysis, and memory usage. Builds run on remote clients on their own schedule, and submit build reports in an XML format. Builds can be submitted both by “official” build clients and by non-core developers or users running the published build process on their own machines.

This simple model is made possible because of the tight conceptual integration between CDash and other elements of the Kitware build infrastructure: CMake, a build configuration system, CTest, a test runner, and CPack, a packaging system. This software provides a mechanism by which build,

test, and packaging recipes can be implemented at a fairly high level of abstraction in an OS-agnostic manner.

CDash's client-driven process simplifies many aspects of the client-side CI process. The decision to run a build is made by build clients, so client-side conditions (time of day, high load, etc.) can be taken into account by the client before starting a build. Clients can appear and disappear as they wish, easily enabling volunteer builds and builds "in the cloud". Build products can be sent to the central server via a straightforward upload mechanism.

However, in exchange for this reporting model, CDash lacks many convenient features of Buildbot. There is no centralized coordination of resources, nor can this be implemented simply in a distributed environment with untrusted or unreliable clients. Progress reports are also not implemented: to do so, the server would have to allow incremental updating of build status. And, of course, there is no way to both globally request a build, and guarantee that anonymous clients *perform* the build in response to a check-in—clients must be considered unreliable.

Recently, CDash added functionality to enable an "@Home" cloud build system, in which clients offer build services to a CDash server. Clients poll the server for build requests, execute them upon request, and return the results to the server. In the current implementation (October 2010), builds must be manually requested on the server side, and clients must be connected for the server to offer their services. However, it is straightforward to extend this to a more generic scheduled-build model in which builds are requested automatically by the server whenever a relevant client is available. The "@Home" system is very similar in concept to the Pony-Build system described later.

Implementation Model: Jenkins

Jenkins is a widely used continuous integration system implemented in Java; until early 2011, it was known as Hudson. It is capable of acting either as a standalone CI system with execution on a local system, or as a coordinator of remote builds, or even as a passive receiver of remote build information. It takes advantage of the JUnit XML standard for unit test and code coverage reporting to integrate reports from a variety of test tools. Jenkins originated with Sun, but is very widely used and has a robust open-source community associated with it.

Jenkins operates in a hybrid mode, defaulting to master-server build execution but allowing a variety of methods for executing remote builds, including both server- and client-initiated builds. Like Buildbot, however, it is primarily designed for central server control, but has been adapted to support a wide variety of distributed job initiation mechanisms, including virtual machine management.

Jenkins can manage multiple remote machines through a connection initiated by the master via an SSH connection, or from the client via JNLP (Java Web Start). This connection is two-way, and supports the communication of objects and data via serial transport.

Jenkins has a robust plugin architecture that abstracts the details of this connection, which has allowed the development of many third-party plugins to support the return of binary builds and more significant result data.

For jobs that are controlled by a central server, Jenkins has a "locks" plugin to discourage jobs from running in parallel, although as of January 2011 it is not yet fully developed.

Implementation Model: Pony-Build

Pony-Build is a proof-of-concept decentralized CI system written in Python. It is composed of three core components, which are illustrated in Figure 6.4. The results server acts as a centralized database containing build results received from individual clients. The clients independently contain

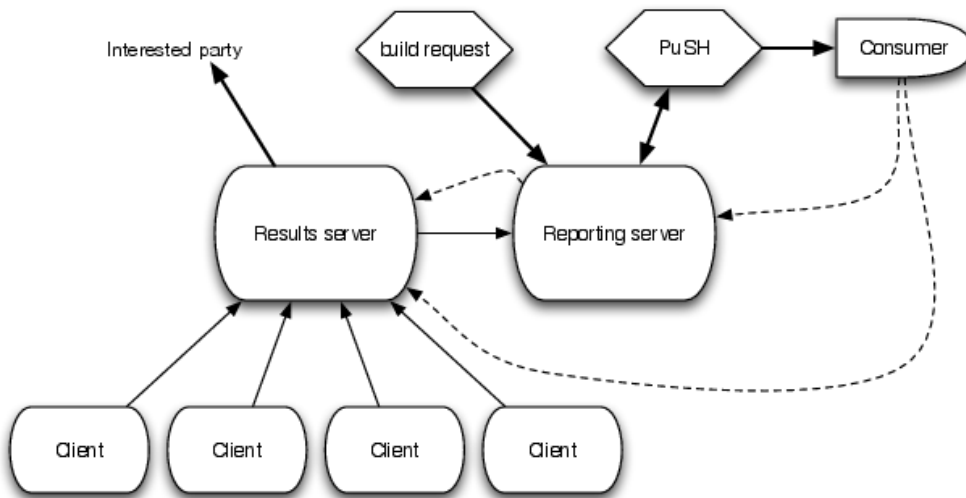


Figure 6.4: Pony-Build Architecture

all configuration information and build context, coupled with a lightweight client-side library to help with VCS repository access, build process management, and the communication of results to the server. The reporting server is optional, and contains a simple Web interface, both for reporting on the results of builds and potentially for requesting new builds. In our implementation, the reporting server and results server run in a single multithreaded process but are loosely coupled at the API level and could easily be altered to run independently.

This basic model is decorated with a variety of webhooks and RPC mechanisms to facilitate build and change notification and build introspection. For example, rather than tying VCS change notification from the code repository directly into the build system, remote build requests are directed to the reporting system, which communicates them to the results server. Likewise, rather than building push notification of new builds out to e-mail, instant messaging, and other services directly into the reporting server, notification is controlled using the PubSubHubbub (PuSH) active notification protocol. This allows a wide variety of consuming applications to receive notification of “interesting” events (currently limited to new builds and failed builds) via a PuSH webhook.

The advantages of this very decoupled model are substantial:

Ease of communication: The basic architectural components and webhook protocols are extremely easy to implement, requiring only a basic knowledge of Web programming.

Easy modification: The implementation of new notification methods, or a new reporting server interface, is extremely simple.

Multiple language support: Since the various components call each other via webhooks, which are supported by most programming languages, different components can be implemented in different languages.

Testability: Each component can be completely isolated and mocked, so the system is very testable.

Ease of configuration: The client-side requirements are minimal, with only a single library file required beyond Python itself.

Minimal server load: Since the central server has virtually no control responsibilities over the clients, isolated clients can run in parallel without contacting the server and placing any corresponding load it, other than at reporting time.

VCS integration: Build configuration is entirely client side, allowing it to be included within the VCS.

Ease of results access: Applications that wish to consume build results can be written in any language capable of an XML-RPC request. Users of the build system can be granted access to the results and reporting servers at the network level, or via a customized interface at the reporting server. The build clients only need access to post results to the results server.

Unfortunately, there are also many serious *disadvantages*, as with the CDash model:

Difficulty requesting builds: This difficulty is introduced by having the build clients be entirely independent of the results server. Clients may poll the results server, to see if a build request is operative, but this introduces high load and significant latency. Alternatively, command and control connections need to be established to allow the server to notify clients of build requests directly. This introduces more complexity into the system and eliminates the advantages of decoupled build clients.

Poor support for resource locking: It is easy to provide an RPC mechanism for holding and releasing resource locks, but much more difficult to enforce client policies. While CI systems like CDash assume good faith on the client side, clients may fail unintentionally and badly, e.g. without releasing locks. Implementing a robust distributed locking system is hard and adds significant undesired complexity. For example, in order to provide master resource locks with unreliable clients, the master lock controller must have a policy in place for clients that take a lock and never release it, either because they crash or because of a deadlock situation.

Poor support for real-time monitoring: Real-time monitoring of the build, and control of the build process itself, is challenging to implement in a system without a constant connection. One significant advantage of Buildbot over the client-driven model is that intermediate inspection of long builds is easy, because the results of the build are communicated to the master interface incrementally. Moreover, because Buildbot retains a control connection, if a long build goes bad in the middle due to misconfiguration or a bad check-in, it can be interrupted and aborted. Adding such a feature into Pony-Build, in which the results server has no guaranteed ability to contact the clients, would require either constant polling by the clients, or the addition of a standing connection to the clients.

Two other aspects of CIs that were raised by Pony-Build were how best to implement *recipes*, and how to manage *trust*. These are intertwined issues, because recipes execute arbitrary code on build clients.

Build Recipes

Build recipes add a useful level of abstraction, especially for software built in a cross-platform language or using a multi-platform build system. For example, CDash relies on a strict kind of recipe; most, or perhaps all, software that uses CDash is built with CMake, CTest, and CPack, and these tools are built to handle multi-platform issues. This is the ideal situation from the viewpoint of a continuous integration system, because the CI system can simply delegate all issues to the build tool chain.

However, this is not true for all languages and build environments. In the Python ecosystem, there has been increasing standardization around `distutils` and `distutils2` for building and packaging software, but as yet no standard has emerged for discovering and running tests, and collating the results. Moreover, many of the more complex Python packages add specialized build logic into their system, through a `distutils` extension mechanism that allows the execution of arbitrary code. This is typical of most build tool chains: while there may be a fairly standard set of commands to be run, there are always exceptions and extensions.

Recipes for building, testing, and packaging are therefore problematic, because they must solve two problems: first, they should be specified in a platform independent way, so that a single recipe can be used to build software on multiple systems; and second, they must be customizable to the software being built.

Trust

This raises a third problem. Widespread use of recipes by a CI system introduces a second party that must be trusted by the system: not only must the software itself be trustworthy (because the CI clients are executing arbitrary code), but the recipes must also be trustworthy (because they, too, must be able to execute arbitrary code).

These trust issues are easy to handle in a tightly controlled environment, e.g. a company where the build clients and CI system are part of an internal process. In other development environments, however, interested third parties may want to offer build services, for example to open source projects. The ideal solution would be to support the inclusion of standard build recipes in software on a community level, a direction that the Python community is taking with `distutils2`. An alternative solution would be to allow for the use of digitally signed recipes, so that trusted individuals could write and distribute signed recipes, and CI clients could check to see if they should trust the recipes.

Choosing a Model

In our experience, a loosely coupled RPC or webhook callback-based model for continuous integration is extremely easy to implement, as long as one ignores any requirements for tight coordination that would involve complex coupling. Basic execution of remote checkouts and builds has similar design constraints whether the build is being driven locally or remotely; collection of information about the build (success/failure, etc.) is primarily driven by client-side requirements; and tracking information by architecture and result involves the same basic requirements. Thus a basic CI system can be implemented quite easily using the reporting model.

We found the loosely coupled model to be very flexible and expandable, as well. Adding new results reporting, notification mechanisms, and build recipes is easy because the components are clearly separated and quite independent. Separated components have clearly delegated tasks to perform, and are also easy to test and easy to modify.

The only challenging aspect of remote builds in a CDash-like loosely-coupled model is build coordination: starting and stopping builds, reporting on ongoing builds, and coordinating resource locks between different clients is technically demanding compared to the rest of the implementation.

It is easy to reach the conclusion that the loosely coupled model is “better” all around, but obviously this is only true if build coordination is not needed. This decision should be made based on the needs of projects using the CI system.

6.3 The Future

While thinking about Pony-Build, we came up with a few features that we would like to see in future continuous integration systems.

A language-agnostic set of build recipes: Currently, each continuous integration system reinvents the wheel by providing its own build configuration language, which is manifestly ridiculous; there are fewer than a dozen commonly used build systems, and probably only a few dozen test runners. Nonetheless, each CI system has a new and different way of specifying the build and test commands to be run. In fact, this seems to be one of the reasons why so many basically identical CI systems exist: each language and community implements their own configuration system, tailored to their own build and test systems, and then layers on the same set of features above that system. Therefore, building a domain-specific language (DSL) capable of representing the options used by the few dozen commonly used build and test tool chains would go a long way toward simplifying the CI landscape.

Common formats for build and test reporting: There is little agreement on exactly what information, in what format, a build and test system needs to provide. If a common format or standard could be developed it would make it much easier for continuous integration systems to offer both detailed and summary views across builds. The Test Anywhere Protocol, TAP (from the Perl community) and the JUnit XML test output format (from the Java community) are two interesting options that are capable of encoding information about number of tests run, successes and failures, and per-file code coverage details.

Increased granularity and introspection in reporting: Alternatively, it would be convenient if different build platforms provided a well-documented set of hooks into their configuration, compilation, and test systems. This would provide an API (rather than a common format) that CI systems could use to extract more detailed information about builds.

Concluding Thoughts

The continuous integration systems described above implemented features that fit their architecture, while the hybrid Jenkins system started with a master/slave model but added features from the more loosely coupled reporting architecture.

It is tempting to conclude that architecture dictates function. This is nonsense, of course. Rather, the choice of architecture seems to canalize or direct development towards a particular set of features. For Pony-Build, we were surprised at the extent to which our initial choice of a CDash-style reporting architecture drove later design and implementation decisions. Some implementation choices, such as the avoidance of a centralized configuration and scheduling system in Pony-Build were driven by our use cases: we needed to allow dynamic attachment of remote build clients, which is difficult to support with Buildbot. Other features we didn't implement, such as progress reports and centralized resource locking in Pony-Build, were desirable but simply too complicated to add without a compelling requirement.

Similar logic may apply to Buildbot, CDash, and Jenkins. In each case there are useful features that are absent, perhaps due to architectural incompatibility. However, from discussions with members of the Buildbot and CDash communities, and from reading the Jenkins website, it seems likely that the desired features were chosen first, and the system was then developed using an architecture that permitted those features to be easily implemented. For example, CDash serves a community with a relatively small set of core developers, who develop software using a centralized model. Their

primary consideration is to keep the software working on a core set of machines, and secondarily to receive bug reports from tech-savvy users. Meanwhile, Buildbot is increasingly used in complex build environments with many clients that require coordination to access shared resources. Buildbot's more flexible configuration file format with its many options for scheduling, change notification, and resource locks fits that need better than the other options. Finally, Jenkins seems aimed at ease of use and simple continuous integration, with a full GUI for configuring it and configuration options for running on the local server.

The sociology of open source development is another confounding factor in correlating architecture with features: suppose developers choose open source projects based on how well the project architecture and features fit their use case? If so, then their contributions will generally reflect an extension of a use case that already fits the project well. Thus projects may get locked into a certain feature set, since contributors are self-selected and may avoid projects with architectures that don't fit their own desired features. This was certainly true for us in choosing to implement a new system, Pony-Build, rather than contributing to Buildbot: the Buildbot architecture was simply not appropriate for building hundreds or thousands of packages.

Existing continuous integration systems are generally built around one of two disparate architectures, and generally implement only a subset of desirable features. As CI systems mature and their user populations grow, we would expect them to grow additional features; however, implementation of these features may be constrained by the base choice of architecture. It will be interesting to see how the field evolves.

Acknowledgments

We thank Greg Wilson, Brett Cannon, Eric Holscher, Jesse Noller, and Victoria Laidler for interesting discussions on CI systems in general, and Pony-Build in particular. Several students contributed to Pony-Build development, including Jack Carlson, Fatima Cherkaoui, Max Laite, and Khushboo Shakya.

Eclipse

Kim Moir

Implementing software modularity is a notoriously difficult task. Interoperability with a large code base written by a diverse community is also difficult to manage. At Eclipse, we have managed to succeed on both counts. In June 2010, the Eclipse Foundation made available its Helios coordinated release, with over 39 projects and 490 committers from over 40 companies working together to build upon the functionality of the base platform. What was the original architectural vision for Eclipse? How did it evolve? How does the architecture of an application serve to encourage community engagement and growth? Let's go back to the beginning.

On November 7, 2001, an open source project called Eclipse 1.0 was released. At the time, Eclipse was described as “an integrated development environment (IDE) for anything and nothing in particular.” This description was purposely generic because the architectural vision was not just another set of tools, but a framework; a framework that was modular and scalable. Eclipse provided a component-based platform that could serve as the foundation for building tools for developers. This extensible architecture encouraged the community to build upon a core platform and extend it beyond the limits of the original vision. Eclipse started as a platform and the Eclipse SDK was the proof-of-concept product. The Eclipse SDK allowed the developers to self-host and use the Eclipse SDK itself to build newer versions of Eclipse.

The stereotypical image of an open source developer is that of an altruistic person toiling late into night fixing bugs and implementing fantastic new features to address their own personal interests. In contrast, if you look back at the early history of the Eclipse project, some of the initial code that was donated was based on VisualAge for Java, developed by IBM. The first committers who worked on this open source project were employees of an IBM subsidiary called Object Technology International (OTI). These committers were paid to work full time on the open source project, to answer questions on newsgroups, address bugs, and implement new features. A consortium of interested software vendors was formed to expand this open tooling effort. The initial members of the Eclipse consortium were Borland, IBM, Merant, QNX Software Systems, Rational Software, RedHat, SuSE, and TogetherSoft.

By investing in this effort, these companies would have the expertise to ship commercial products based on Eclipse. This is similar to investments that corporations make in contributing to the Linux kernel because it is in their self-interest to have employees improving the open source software that underlies their commercial offerings. In early 2004, the Eclipse Foundation was formed to manage and expand the growing Eclipse community. This not-for-profit foundation was funded

by corporate membership dues and is governed by a board of directors. Today, the diversity of the Eclipse community has expanded to include over 170 member companies and almost 1000 committers.

Originally, people knew “Eclipse” as the SDK only but today it is much more. In July 2010, there were 250 diverse projects under development at eclipse.org. There’s tooling to support developing with C/C++, PHP, web services, model driven development, build tooling and many more. Each of these projects is included in a top-level project (TLP) which is managed by a project management committee (PMC) consisting of senior members of the project nominated for the responsibility of setting technical direction and release goals. In the interests of brevity, the scope of this chapter will be limited to the evolution of the architecture of the Eclipse SDK within Eclipse¹ and Runtime Equinox² projects. Since Eclipse has long history, I’ll be focusing on early Eclipse, as well as the 3.0, 3.4 and 4.0 releases.

7.1 Early Eclipse

At the beginning of the 21st century, there were many tools for software developers, but few of them worked together. Eclipse sought to provide an open source platform for the creation of interoperable tools for application developers. This would allow developers to focus on writing new tools, instead of writing code to deal with infrastructure issues like interacting with the filesystem, providing software updates, and connecting to source code repositories. Eclipse is perhaps most famous for the Java Development Tools (JDT). The intent was that these exemplary Java development tools would serve as an example for people interested in providing tooling for other languages.

Before we delve into the architecture of Eclipse, let’s look at what the Eclipse SDK looks like to a developer. Upon starting Eclipse and selecting the workbench, you’ll be presented with the Java perspective. A perspective organizes the views and editors that are specific to the tooling that is currently in use.

Early versions of the Eclipse SDK architecture had three major elements, which corresponded to three major sub-projects: the Platform, the JDT (Java Development Tools) and the PDE (Plug-in Development Environment).

Platform

The Eclipse platform is written using Java and a Java VM is required to run it. It is built from small units of functionality called plugins. Plugins are the basis of the Eclipse component model. A plugin is essentially a JAR file with a manifest which describes itself, its dependencies, and how it can be utilized, or extended. This manifest information was initially stored in a `plugin.xml` file which resides in the root of the plugin directory. The Java development tools provided plugins for developing in Java. The Plug-in Development Environment (PDE) provides tooling for developing plugins to extend Eclipse. Eclipse plugins are written in Java but could also contain non-code contributions such as HTML files for online documentation. Each plugin has its own class loader. Plugins can express dependencies on other plugins by the use of `requires` statements in the `plugin.xml`. Looking at the `plugin.xml` for the `org.eclipse.ui` plugin you can see its name and version specified, as well as the dependencies it needs to import from other plugins.

¹<http://www.eclipse.org>

²<http://www.eclipse.org/equinox>

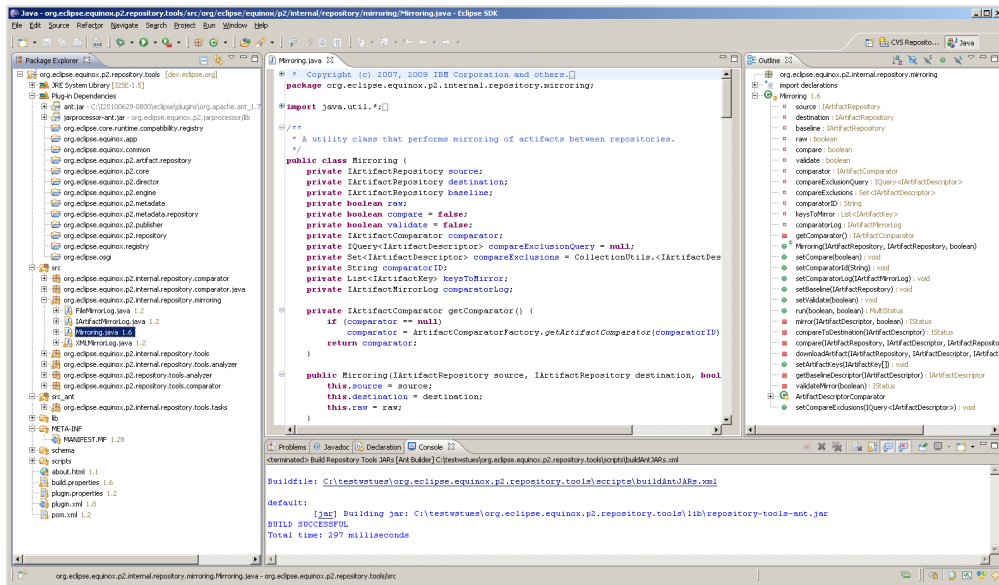


Figure 7.1: Java Perspective

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
  id="org.eclipse.ui"
  name="%Plugin.name"
  version="2.1.1"
  provider-name="%Plugin.providerName"
  class="org.eclipse.ui.internal.UIPlugin">

  <runtime>
    <library name="ui.jar">
      <export name="*" />
      <packages prefixes="org.eclipse.ui" />
    </library>
  </runtime>
  <requires>
    <import plugin="org.apache.xerces" />
    <import plugin="org.eclipse.core.resources" />
    <import plugin="org.eclipse.update.core" />
    :
    :
    <import plugin="org.eclipse.text" export="true" />
    <import plugin="org.eclipse.ui.workbench.texteditor" export="true" />
    <import plugin="org.eclipse.ui.editors" export="true" />
  </requires>
</plugin>
```

In order to encourage people to build upon the Eclipse platform, there needs to be a mechanism to make a contribution to the platform, and for the platform to accept this contribution. This is achieved

through the use of extensions and extension points, another element of the Eclipse component model. The export identifies the interfaces that you expect others to use when writing their extensions, which limits the classes that are available outside your plugin to the ones that are exported. It also provides additional limitations on the resources that are available outside the plugin, as opposed to making all public methods or classes available to consumers. Exported plugins are considered public API. All others are considered private implementation details. To write a plugin that would contribute a menu item to the Eclipse toolbar, you can use the `actionSets` extension point in the `org.eclipse.ui` plugin.

```
<extension-point id="actionSets" name="%ExtPoint.actionSets"
                schema="schema/actionSets.exsd"/>
<extension-point id="commands" name="%ExtPoint.commands"
                schema="schema/commands.exsd"/>
<extension-point id="contexts" name="%ExtPoint.contexts"
                schema="schema/contexts.exsd"/>
<extension-point id="decorators" name="%ExtPoint.decorators"
                schema="schema/decorators.exsd"/>
<extension-point id="dropActions" name="%ExtPoint.dropActions"
                schema="schema/dropActions.exsd"/> =
```

Your plugin's extension to contribute a menu item to the `org.eclipse.ui.actionSet` extension point would look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
  id="com.example.helloworld"
  name="com.example.helloworld"
  version="1.0.0">
  <runtime>
    <library name="helloworld.jar"/>
  </runtime>
  <requires>
    <import plugin="org.eclipse.ui"/>
  </requires>
  <extension
    point="org.eclipse.ui.actionSets">
    <actionSet
      label="Example Action Set"
      visible="true"
      id="org.eclipse.helloworld.actionSet">
      <menu
        label="Example &Menu"
        id="exampleMenu">
        <separator
          name="exampleGroup">
        </separator>
      </menu>
      <action
        label="&Example Action"
        icon="icons/example.gif"
        tooltip="Hello, Eclipse world"
```

```

        class="com.example.helloworld.actions.ExampleAction"
        menubarPath="exampleMenu/exampleGroup"
        toolbarPath="exampleGroup"
        id="org.eclipse.helloworld.actions.ExampleAction">
    </action>
</actionSet>
</extension>
</plugin>

```

When Eclipse is started, the runtime platform scans the manifests of the plugins in your install, and builds a plugin registry that is stored in memory. Extension points and the corresponding extensions are mapped by name. The resulting plugin registry can be referenced from the API provided by the Eclipse platform. The registry is cached to disk so that this information can be reloaded the next time Eclipse is restarted. All plugins are discovered upon startup to populate the registry but they are not activated (classes loaded) until the code is actually used. This approach is called lazy activation. The performance impact of adding additional bundles into your install is reduced by not actually loading the classes associated with the plugins until they are needed. For instance, the plugin that contributes to the `org.eclipse.ui.actionSet` extension point wouldn't be activated until the user selected the new menu item in the toolbar.

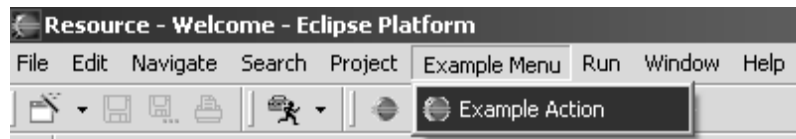


Figure 7.2: Example Menu

The code that generates this menu item looks like this:

```

package com.example.helloworld.actions;

import org.eclipse.jface.action.IAction;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.ui.IWorkbenchWindow;
import org.eclipse.ui.IWorkbenchWindowActionDelegate;
import org.eclipse.jface.dialogs.MessageDialog;

public class ExampleAction implements IWorkbenchWindowActionDelegate {
    private IWorkbenchWindow window;

    public ExampleAction() {
    }

    public void run(IAction action) {
        MessageDialog.openInformation(
            window.getShell(),
            "org.eclipse.helloworld",
            "Hello, Eclipse architecture world");
    }
}

```

```

public void selectionChanged(IAction action, ISelection selection) {
}

public void dispose() {
}

public void init(IWorkbenchWindow window) {
    this.window = window;
}
}

```

Once the user selects the new item in the toolbar, the extension registry is queried by the plugin implementing the extension point. The plugin supplying the extension instantiates the contribution, and loads the plugin. Once the plugin is activated, the `ExampleAction` constructor in our example is run, and then initializes a `Workbench` action delegate. Since the selection in the workbench has changed and the delegate has been created, the action can change. The message dialog opens with the message “Hello, Eclipse architecture world”.

This extensible architecture was one of the keys to the successful growth of the Eclipse ecosystem. Companies or individuals could develop new plugins, and either release them as open source or sell them commercially.

One of the most important concepts about Eclipse is that *everything is a plugin*. Whether the plugin is included in the Eclipse platform, or you write it yourself, plugins are all first class components of the assembled application. Figure 7.3 shows clusters of related functionality contributed by plugins in early versions of Eclipse.

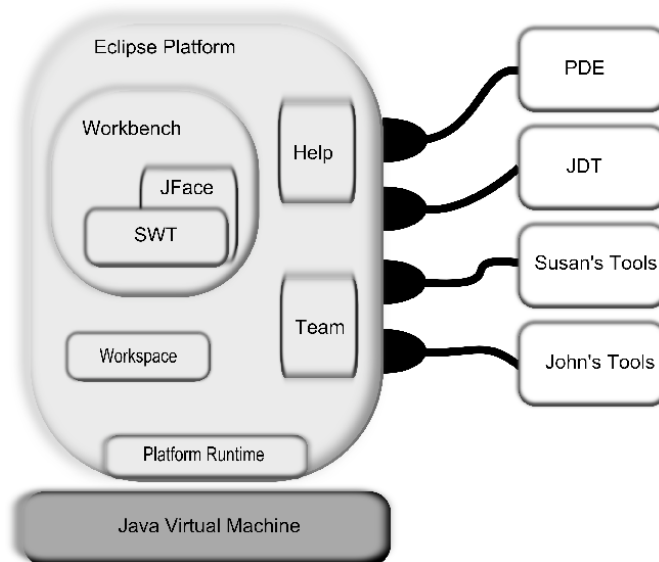


Figure 7.3: Early Eclipse Architecture

The workbench is the most familiar UI element to users of the Eclipse platform, as it provides the structures that organize how Eclipse appears to the user on the desktop. The workbench consists

of perspectives, views, and editors. Editors are associated with file types so the correct editor is launched when a file is opened. An example of a view is the “problems” view that indicates errors or warnings in your Java code. Together, editors and views form a perspective which presents the tooling to the user in an organized fashion.

The Eclipse workbench is built on the Standard Widget Toolkit (SWT) and JFace, and SWT deserves a bit of exploration. Widget toolkits are generally classified as either native or emulated. A native widget toolkit uses operating system calls to build user interface components such as lists and push buttons. Interaction with components is handled by the operating system. An emulated widget toolkit implements components outside of the operating system, handling mouse and keyboard, drawing, focus and other widget functionality itself, rather than deferring to the operating system. Both designs have different strengths and weaknesses.

Native widget toolkits are “pixel perfect.” Their widgets look and feel like their counterparts in other applications on the desktop. Operating system vendors constantly change the look and feel of their widgets and add new features. Native widget toolkits get these updates for free. Unfortunately, native toolkits are difficult to implement because their underlying operating system widget implementations are vastly different, leading to inconsistencies and programs that are not portable.

Emulated widget toolkits either provide their own look and feel, or try to draw and behave like the operating system. Their great strength over native toolkits is flexibility (although modern native widget toolkits such as Windows Presentation Framework (WPF) are equally as flexible). Because the code to implement a widget is part of the toolkit rather than embedded in the operating system, a widget can be made to draw and behave in any manner. Programs that use emulated widget toolkits are highly portable. Early emulated widget toolkits had a bad reputation. They were often slow and did a poor job of emulating the operating system, making them look out of place on the desktop. In particular, Smalltalk-80 programs at the time were easy to recognize due to their use of emulated widgets. Users were aware that they were running a “Smalltalk program” and this hurt acceptance of applications written in Smalltalk.

Unlike other computer languages such as C and C++, the first versions of Java came with a native widget toolkit library called the Abstract Window Toolkit (AWT). AWT was considered to be limited, buggy and inconsistent and was widely decried. At Sun and elsewhere, in part because of experience with AWT, a native widget toolkit that was portable and performant was considered to be unworkable. The solution was Swing, a full-featured emulated widget toolkit.

Around 1999, OTI was using Java to implement a product called VisualAge Micro Edition. The first version of VisualAge Micro Edition used Swing and OTI’s experience with Swing was not positive. Early versions of Swing were buggy, had timing and memory issues and the hardware at the time was not powerful enough to give acceptable performance. OTI had successfully built a native widget toolkit for Smalltalk-80 and other Smalltalk implementations to gain acceptance of Smalltalk. This experience was used to build the first version of SWT. VisualAge Micro Edition and SWT were a success and SWT was the natural choice when work began on Eclipse. The use of SWT over Swing in Eclipse split the Java community. Some saw conspiracies, but Eclipse was a success and the use of SWT differentiated it from other Java programs. Eclipse was performant, pixel perfect and the general sentiment was, “I can’t believe it’s a Java program.”

Early Eclipse SDKs ran on Linux and Windows. In 2010, there is support for over a dozen platforms. A developer can write an application for one platform, and deploy it to multiple platforms. Developing a new widget toolkit for Java was a contentious issue within the Java community at the time, but the Eclipse committers felt that it was worth the effort to provide the best native experience on the desktop. This assertion applies today, and there are millions of lines of code that depend on SWT.

JFace is a layer on top of SWT that provides tools for common UI programming tasks, such as frameworks for preferences and wizards. Like SWT, it was designed to work with many windowing systems. However, it is pure Java code and doesn't contain any native platform code.

The platform also provided an integrated help system based upon small units of information called topics. A topic consists of a label and a reference to its location. The location can be an HTML documentation file, or an XML document describing additional links. Topics are grouped together in table of contents (TOCs). Consider the topics as the leaves, and TOCs as the branches of organization. To add help content to your application, you can contribute to the `org.eclipse.help.toc` extension point, as the `org.eclipse.platform.doc.isv.plugin.xml` does below.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin>

<!-- ===== -->
<!-- Define primary TOC -->
<!-- ===== -->
    <extension
        point="org.eclipse.help.toc">
        <toc
            file="toc.xml"
            primary="true">
        </toc>
        <index path="index"/>
    </extension>

<!-- ===== -->
<!-- Define TOCs -->
<!-- ===== -->
    <extension
        point="org.eclipse.help.toc">
        <toc
            file="topics_Guide.xml">
        </toc>
        <toc
            file="topics_Reference.xml">
        </toc>
        <toc
            file="topics_Porting.xml">
        </toc>
        <toc
            file="topics_Questions.xml">
        </toc>
        <toc
            file="topics_Samples.xml">
        </toc>
    </extension>
```

Apache Lucene is used to index and search the online help content. In early versions of Eclipse, online help was served as a Tomcat web application. Additionally, by providing help within Eclipse itself, you can also use the subset of help plugins to provide a standalone help server.³

³For example: <http://help.eclipse.org>.

Eclipse also provides team support to interact with a source code repository, create patches and other common tasks. The workspace provided collection of files and metadata that stored your work on the filesystem. There was also a debugger to trace problems in the Java code, as well as a framework for building language specific debuggers.

One of the goals of the Eclipse project was to encourage open source and commercial consumers of this technology to extend the platform to meet their needs, and one way to encourage this adoption is to provide a stable API. An API can be thought of as a technical contract specifying the behavior of your application. It also can be thought of as a social contract. On the Eclipse project, the mantra is, “API is forever”. Thus careful consideration must be given when writing an API given that it is meant to be used indefinitely. A stable API is a contract between the client or API consumer and the provider. This contract ensures that the client can depend on the Eclipse platform to provide the API for the long term without the need for painful refactoring on the part of the client. A good API is also flexible enough to allow the implementation to evolve.

Java Development Tools (JDT)

The JDT provides Java editors, wizards, refactoring support, debugger, compiler and an incremental builder. The compiler is also used for content assist, navigation and other editing features. A Java SDK isn't shipped with Eclipse so it's up to the user to choose which SDK to install on their desktop. Why did the JDT team write a separate compiler to compile your Java code within Eclipse? They had an initial compiler code contribution from VisualAge Micro Edition. They planned to build tooling on top of the compiler, so writing the compiler itself was a logical decision. This approach also allowed the JDT committers to provide extension points for extending the compiler. This would be difficult if the compiler was a command line application provided by a third party.

Writing their own compiler provided a mechanism to provide support for an incremental builder within the IDE. An incremental builder provides better performance because it only recompiles files that have changed or their dependencies. How does the incremental builder work? When you create a Java project within Eclipse, you are creating resources in the workspace to store your files. A builder within Eclipse takes the inputs within your workspace (.java files), and creates an output (.class files). Through the build state, the builder knows about the types (classes or interfaces) in the workspace, and how they reference each other. The build state is provided to the builder by the compiler each time a source file is compiled. When an incremental build is invoked, the builder is supplied with a resource delta, which describes any new, modified or deleted files. Deleted source files have their corresponding class files deleted. New or modified types are added to a queue. The files in the queue are compiled in sequence and compared with the old class file to determine if there are structural changes. Structural changes are modifications to the class that can impact another type that references it. For example, changing a method signature, or adding or removing a method. If there are structural changes, all the types that reference it are also added to the queue. If the type has changed at all, the new class file is written to the build output folder. The build state is updated with reference information for the compiled type. This process is repeated for all the types in the queue until empty. If there are compilation errors, the Java editor will create problem markers. Over the years, the tooling that JDT provides has expanded tremendously in concert with new versions of the Java runtime itself.

Plug-in Development Environment (PDE)

The Plug-in Development Environment (PDE) provided the tooling to develop, build, deploy and test plugins and other artifacts that are used to extend the functionality of Eclipse. Since Eclipse plugins were a new type of artifact in the Java world there wasn't a build system that could transform the source into plugins. Thus the PDE team wrote a component called PDE Build which examined the dependencies of the plugins and generated Ant scripts to construct the build artifacts.

7.2 Eclipse 3.0: Runtime, RCP and Robots

Runtime

Eclipse 3.0 was probably one of the most important Eclipse releases due to the number of significant changes that occurred during this release cycle. In the pre-3.0 Eclipse architecture, the Eclipse component model consisted of plugins that could interact with each other in two ways. First, they could express their dependencies by the use of the `requires` statement in their `plugin.xml`. If plugin A requires plugin B, plugin A can see all the Java classes and resources from B, respecting Java class visibility conventions. Each plugin had a version, and they could also specify the versions of their dependencies. Secondly, the component model provided *extensions* and *extension points*. Historically, Eclipse committers wrote their own runtime for the Eclipse SDK to manage classloading, plugin dependencies and extensions and extension points.

The Equinox project was created as a new incubator project at Eclipse. The goal of the Equinox project was to replace the Eclipse component model with one that already existed, as well as provide support for dynamic plugins. The solutions under consideration included JMX, Jakarta Avalon and OSGi. JMX was not a fully developed component model so it was not deemed appropriate. Jakarta Avalon wasn't chosen because it seemed to be losing momentum as a project. In addition to the technical requirements, it was also important to consider the community that supported these technologies. Would they be willing to incorporate Eclipse-specific changes? Was it actively developed and gaining new adopters? The Equinox team felt that the community around their final choice of technology was just as important as the technical considerations.

After researching and evaluating the available alternatives, the committers selected OSGi. Why OSGi? It had a semantic versioning scheme for managing dependencies. It provided a framework for modularity that the JDK itself lacked. Packages that were available to other bundles must be explicitly exported, and all others were hidden. OSGi provided its own classloader so the Equinox team didn't have to continue to maintain their own. By standardizing on a component model that had wider adoption outside the Eclipse ecosystem, they felt they could appeal to a broader community and further drive the adoption of Eclipse.

The Equinox team felt comfortable that since OSGi already had an existing and vibrant community, they could work with that community to help include the functionality that Eclipse required in a component model. For instance, at the time, OSGi only supported listing requirements at a package level, not a plugin level as Eclipse required. In addition, OSGi did not yet include the concept of fragments, which were Eclipse's preferred mechanism for supplying platform or environment specific code to an existing plugin. For example, fragments provide code for working with Linux and Windows filesystems as well as fragments which contribute language translations. Once the decision was made to proceed with OSGi as the new runtime, the committers needed an open source framework implementation. They evaluated Oscar, the precursor to Apache Felix, and the Service Management Framework (SMF) developed by IBM. At the time, Oscar was a research

project with limited deployment. SMF was ultimately chosen since it was already used in shipping products and thus was deemed enterprise-ready. The Equinox implementation serves as the reference implementation of the OSGi specification.

A compatibility layer was also provided so that existing plugins would still work in a 3.0 install. Asking developers to rewrite their plugins to accommodate changes in the underlying infrastructure of Eclipse 3.0 would have stalled the momentum on Eclipse as a tooling platform. The expectation from Eclipse consumers was that the platform should just continue to work.

With the switch to OSGi, Eclipse plugins became known as bundles. A plugin and a bundle are the same thing: They both provide a modular subset of functionality that describes itself with metadata in a manifest. Previously, dependencies, exported packages and the extensions and extension points were described in `plugin.xml`. With the move to OSGi bundles, the extensions and extension points continued to be described in `plugin.xml` since they are Eclipse concepts. The remaining information was described in the `META-INF/MANIFEST.MF`, OSGi's version of the bundle manifest. To support this change, PDE provided a new manifest editor within Eclipse. Each bundle has a name and version. The manifest for the `org.eclipse.ui` bundle looks like this:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: %Plugin.name
Bundle-SymbolicName: org.eclipse.ui; singleton:=true
Bundle-Version: 3.3.0.qualifier
Bundle-ClassPath: .
Bundle-Activator: org.eclipse.ui.internal.UIPlugin
Bundle-Vendor: %Plugin.providerName
Bundle-Localization: plugin
Export-Package: org.eclipse.ui.internal;x-internal:=true
Require-Bundle: org.eclipse.core.runtime;bundle-version="[3.2.0,4.0.0)",
    org.eclipse.swt;bundle-version="[3.3.0,4.0.0)";visibility=reexport,
    org.eclipse.jface;bundle-version="[3.3.0,4.0.0)";visibility=reexport,
    org.eclipse.ui.workbench;bundle-version="[3.3.0,4.0.0)";visibility=reexport,
    org.eclipse.core.expressions;bundle-version="[3.3.0,4.0.0)"
Eclipse-LazyStart: true
Bundle-RequiredExecutionEnvironment: CDC-1.0/Foundation-1.0, J2SE-1.3
```

As of Eclipse 3.1, the manifest can also specify a bundle required execution environment (BREE). Execution environments specify the minimum Java environment required for the bundle to run. The Java compiler does not understand bundles and OSGi manifests. PDE provides tooling for developing OSGi bundles. Thus, PDE parses the bundle's manifest, and generates the classpath for that bundle. If you specified an execution environment of J2SE-1.4 in your manifest, and then wrote some code that included generics, you would be advised of compile errors in your code. This ensures that your code adheres to the contract you have specified in the manifest.

OSGi provides a modularity framework for Java. The OSGi framework manages collections of self-describing bundles and manages their classloading. Each bundle has its own classloader. The classpath available to a bundle is constructed by examining the dependencies of the manifest and generating a classpath available to the bundle. OSGi applications are collections of bundles. In order to fully embrace modularity, you must be able to express your dependencies in a reliable format for consumers. Thus the manifest describes exported packages that are available to clients of this bundle which corresponds to the public API that was available for consumption. The bundle that is consuming that API must have a corresponding import of the package they are consuming.

The manifest also allows you to express version ranges for your dependencies. Looking at the `Require-Bundle` heading in the above manifest, you will note that the `org.eclipse.core.runtime` bundle that `org.eclipse.ui` depends on must be at least 3.2.0 and less than 4.0.0.

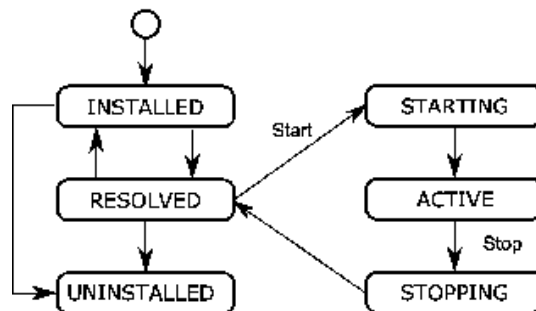


Figure 7.4: OSGi Bundle Lifecycle

OSGi is a dynamic framework which supports the installation, starting, stopping, or uninstallation of bundles. As mentioned before, lazy activation was a core advantage to Eclipse because plugin classes were not loaded until they were needed. The OSGi bundle lifecycle also enables this approach. When you start an OSGi application, the bundles are in the installed state. If its dependencies are met, the bundle changes to the resolved state. Once resolved, the classes within that bundle can be loaded and run. The starting state means that the bundle is being activated according to its activation policy. Once activated, the bundle is in the active state, it can acquire required resources and interact with other bundles. A bundle is in the stopping state when it is executing its activator stop method to clean up any resources that were opened when it was active. Finally, a bundle may be uninstalled, which means that it's not available for use.

As the API evolves, there needs to be a way to signal changes to your consumers. One approach is to use semantic versioning of your bundles and version ranges in your manifests to specify the version ranges for your dependencies. OSGi uses a four-part versioning naming scheme as shown in Figure 7.5.

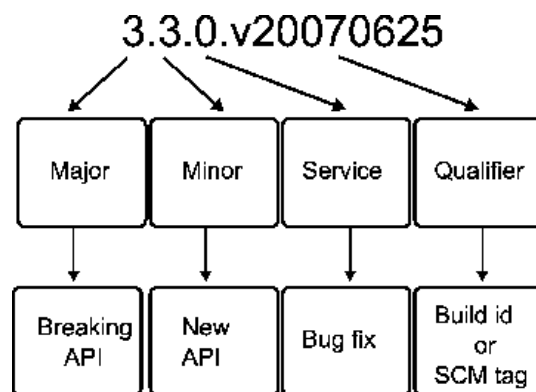


Figure 7.5: Versioning Naming Scheme

With the OSGi version numbering scheme, each bundle has a unique identifier consisting of a name and a four part version number. An id and version together denote a unique set of bytes to the consumer. By Eclipse convention, if you're making changes to a bundle, each segment of the version signifies to the consumer the type of change being made. Thus, if you want to indicate that you intend to break API, you increment the first (major) segment. If you have just added API, you increment the second (minor) segment. If you fix a small bug that doesn't impact API, the third (service) segment is incremented. Finally, the fourth or qualifier segment is incremented to indicate a build id source control repository tag.

In addition to expressing the fixed dependencies between bundles, there is also a mechanism within OSGi called services which provides further decoupling between bundles. Services are objects with a set of properties that are registered with the OSGi service registry. Unlike extensions, which are registered in the extension registry when Eclipse scans bundles during startup, services are registered dynamically. A bundle that is consuming a service needs to import the package defining the service contract, and the framework determines the service implementation from the service registry.

Like a main method in a Java class file, there is a specific application defined to start Eclipse. Eclipse applications are defined using extensions. For instance, the application to start the Eclipse IDE itself is `org.eclipse.ui.ide.workbench` which is defined in the `org.eclipse.ui.ide.application` bundle.

```
<plugin>
  <extension
    id="org.eclipse.ui.ide.workbench"
    point="org.eclipse.core.runtime.applications">
    <application>
      <run
        class="org.eclipse.ui.internal.ide.application.IDEApplication">
      </run>
    </application>
  </extension>
</plugin>
```

There are many applications provided by Eclipse such as those to run standalone help servers, Ant tasks, and JUnit tests.

Rich Client Platform (RCP)

One of the most interesting things about working in an open source community is that people use the software in totally unexpected ways. The original intent of Eclipse was to provide a platform and tooling to create and extend IDEs. However, in the time leading up to the 3.0 release, bug reports revealed that the community was taking a subset of the platform bundles and using them to build Rich Client Platform (RCP) applications, which many people would recognize as Java applications. Since Eclipse was initially constructed with an IDE-centric focus, there had to be some refactoring of the bundles to allow this use case to be more easily adopted by the user community. RCP applications didn't require all the functionality in the IDE, so several bundles were split into smaller ones that could be consumed by the community for building RCP applications.

Examples of RCP applications in the wild include the use of RCP to monitor the Mars Rover robots developed by NASA at the Jet Propulsion Laboratory, Bioclipse for data visualization of bioinformatics and Dutch Railway for monitoring train performance. The common thread that ran

through many of these applications was that these teams decided that they could take the utility provided by the RCP platform and concentrate on building their specialized tools on top of it. They could save development time and money by focusing on building their tools on a platform with a stable API that guaranteed that their technology choice would have long term support.

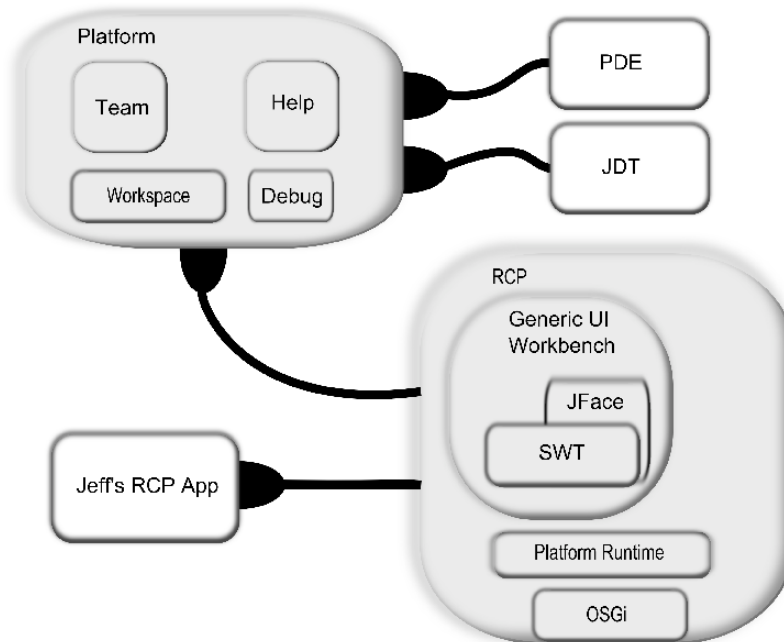


Figure 7.6: Eclipse 3.0 Architecture

Looking at the 3.0 architecture in Figure 7.6, you will note that the Eclipse Runtime still exists to provide the application model and extension registry. Managing the dependencies between components, the plugin model is now managed by OSGi. In addition to continuing to be able to extend Eclipse for their own IDEs, consumers can also build upon the RCP application framework for more generic applications.

7.3 Eclipse 3.4

The ability to easily update an application to a new version and add new content is taken for granted. In Firefox it happens seamlessly. For Eclipse it hasn't been so easy. Update Manager was the original mechanism that was used to add new content to the Eclipse install or update to a new version.

To understand what changes during an update or install operation, it's necessary to understand what Eclipse means by "features". A feature is a PDE artifact that defines a set of bundles that are packaged together in a format that can be built or installed. Features can also include other features. (See Figure 7.7.)

If you wished to update your Eclipse install to a new build that only incorporated one new bundle, the entire feature had to be updated since this was the coarse grained mechanism that was used by update manager. Updating a feature to fix a single bundle is inefficient.

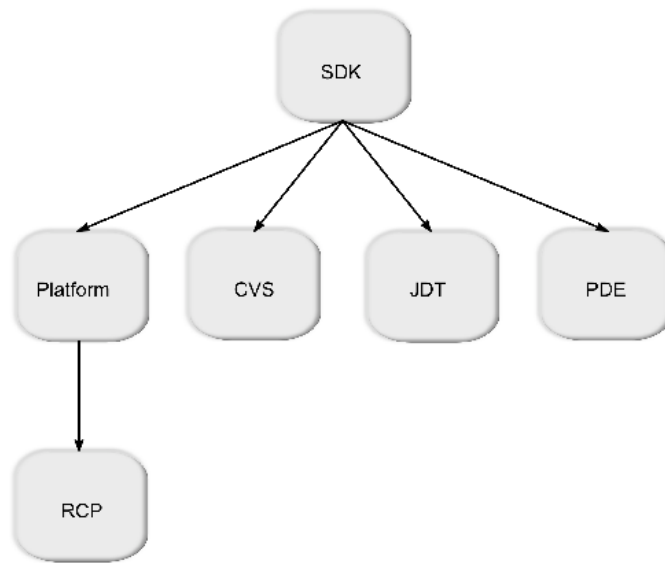


Figure 7.7: Eclipse 3.3 SDK Feature Hierarchy

There are PDE wizards to create features, and build them in your workspace. The `feature.xml` file defines the bundles included in the feature, and some simple properties of the bundles. A feature, like a bundle, has a name and a version. Features can include other features, and specify version ranges for the features they include. The bundles that are included in a feature are listed, along with specific properties. For instance, you can see that the `org.eclipse.launcher.gtk.linux.x86_64` fragment specifies the operating system (os), windowing system (ws) and architecture (arch) where it should be used. Thus upgrading to a new release, this fragment would only be installed on this platform. These platform filters are included in the OSGi manifest of this bundle.

```
<?xml version="1.0" encoding="UTF-8"?>
<feature
  id="org.eclipse.rcp"
  label="%featureName"
  version="3.7.0.qualifier"
  provider-name="%providerName"
  plugin="org.eclipse.rcp"
  image="eclipse_update_120.jpg">

  <description>
    %description
  </description>

  <copyright>
    %copyright
  </copyright>

  <license url="%licenseURL">
    %license
```

```

</license>

<plugin
    id="org.eclipse.equinox.launcher"
    download-size="0"
    install-size="0"
    version="0.0.0"
    unpack="false"/>

<plugin
    id="org.eclipse.equinox.launcher.gtk.linux.x86_64"
    os="linux"
    ws="gtk"
    arch="x86_64"
    download-size="0"
    install-size="0"
    version="0.0.0"
    fragment="true"/>

```

An Eclipse application consists of more than just features and bundles. There are platform specific executables to start Eclipse itself, license files, and platform specific libraries, as shown in this list of files included in the Eclipse application.

```

com.ibm.icu
org.eclipse.core.commands
org.eclipse.core.contenttype
org.eclipse.core.databinding
org.eclipse.core.databinding.beans
org.eclipse.core.expressions
org.eclipse.core.jobs
org.eclipse.core.runtime
org.eclipse.core.runtime.compatibility.auth
org.eclipse.equinox.common
org.eclipse.equinox.launcher
org.eclipse.equinox.launcher.carbon.macosx
org.eclipse.equinox.launcher.gtk.linux.ppc
org.eclipse.equinox.launcher.gtk.linux.s390
org.eclipse.equinox.launcher.gtk.linux.s390x
org.eclipse.equinox.launcher.gtk.linux.x86
org.eclipse.equinox.launcher.gtk.linux.x86_64

```

These files couldn't be updated via update manager, because again, it only dealt with features. Since many of these files were updated every major release, this meant that users had to download a new zip each time there was a new release instead of updating their existing install. This wasn't acceptable to the Eclipse community. PDE provided support for product files, which specified all the files needed to build an Eclipse RCP application. However, update manager didn't have a mechanism to provision these files into your install which was very frustrating for users and product developers alike. In March 2008, p2 was released into the SDK as the new provisioning solution. In the interest of backward compatibility, Update Manager was still available for use, but p2 was enabled by default.

p2 Concepts

Equinox p2 is all about installation units (IU). An IU is a description of the name and id of the artifact you are installing. This metadata also describes the capabilities of the artifact (what is provided) and its requirements (its dependencies). Metadata can also express applicability filters if an artifact is only applicable to a certain environment. For instance, the `org.eclipse.swt.gtk.linux.x86` fragment is only applicable if you're installing on a Linux gtk x86 machine. Fundamentally, metadata is an expression of the information in the bundle's manifest. Artifacts are simply the binary bits being installed. A separation of concerns is achieved by separating the metadata and the artifacts that they describe. A p2 repository consists of both metadata and artifact repositories.

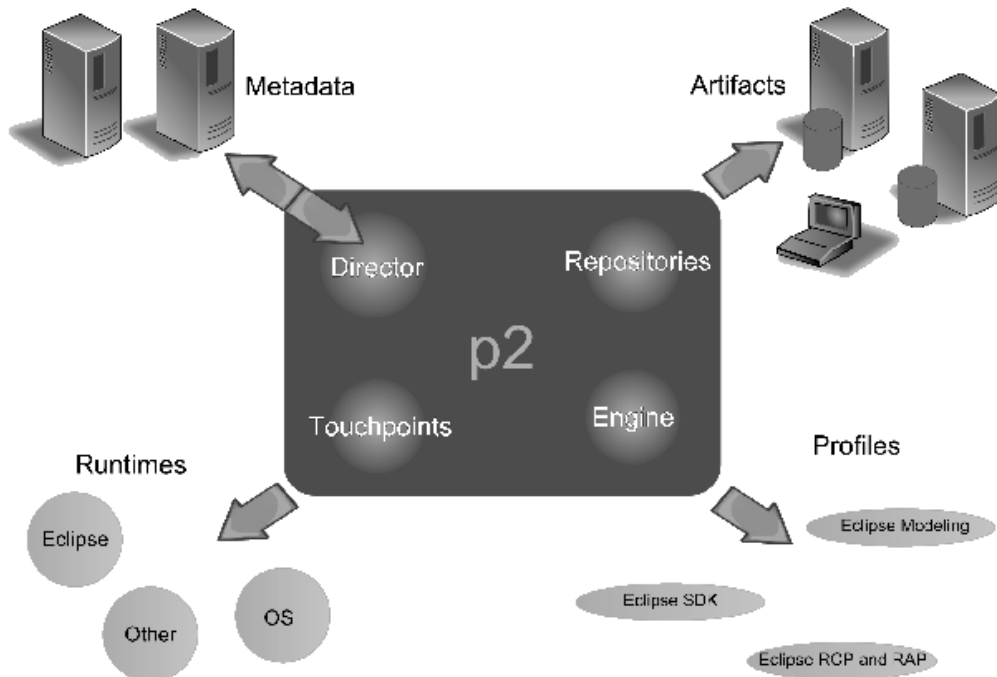


Figure 7.8: P2 Concepts

A profile is a list of IUs in your install. For instance, your Eclipse SDK has a profile that describes your current install. From within Eclipse, you can request an update to a newer version of the build which will create a new profile with a different set of IUs. A profile also provides a list of properties associated with the installation, such as the operating system, windowing system, and architecture parameters. Profiles also store the installation directory and the location. Profiles are held by a profile registry, which can store multiple profiles. The director is responsible for invoking provisioning operations. It works with the planner and the engine. The planner examines the existing profile, and determines the operations that must occur to transform the install into its new state. The engine is responsible for carrying out the actual provisioning operations and installing the new artifacts on disk. Touchpoints are part of the engine that work with the runtime implementation of the system being installed. For instance, for the Eclipse SDK, there is an Eclipse touchpoint which knows how to install bundles. For a Linux system where Eclipse is installed from RPM binaries, the engine

would deal with an RPM touchpoint. Also, p2 can perform installs in-process or outside in a separate process, such as a build.

There were many benefits to the new p2 provisioning system. Eclipse install artifacts could be updated from release to release. Since previous profiles were stored on disk, there was also a way to revert to a previous Eclipse install. Additionally, given a profile and a repository, you could recreate the Eclipse install of a user that was reporting a bug to try to reproduce the problem on your own desktop. Provisioning with p2 provided a way to update and install more than just the Eclipse SDK, it was a platform that applied to RCP and OSGi use cases as well. The Equinox team also worked with the members of another Eclipse project, the Eclipse Communication Framework (ECF) to provide reliable transport for consuming artifacts and metadata in p2 repositories.

There were many spirited discussions within the Eclipse community when p2 was released into the SDK. Since update manager was a less than optimal solution for provisioning your Eclipse install, Eclipse consumers had the habit of unzipping bundles into their install and restarting Eclipse. This approach resolves your bundles on a best effort basis. It also meant that any conflicts in your install were being resolved at runtime, not install time. Constraints should be resolved at install time, not run time. However, users were often oblivious to these issues and assumed since the bundles existed on disk, they were working. Previously, the update sites that Eclipse provided were a simple directory consisting of JARred bundles and features. A simple `site.xml` file provided the names of the features that were available to be consumed in the site. With the advent of p2, the metadata that was provided in the p2 repositories was much more complex. To create metadata, the build process needed to be tweaked to either generate metadata at build time or run a generator task over the existing bundles. Initially, there was a lack of documentation available describing how to make these changes. As well, as is always the case, exposing new technology to a wider audience exposed unexpected bugs that had to be addressed. However, by writing more documentation and working long hours to address these bugs, the Equinox team was able to address these concerns and now p2 is the underlying provision engine behind many commercial offerings. As well, the Eclipse Foundation ships its coordinated release every year using a p2 aggregate repository of all the contributing projects.

7.4 Eclipse 4.0

Architecture must continually be examined to evaluate if it is still appropriate. Is it able to incorporate new technology? Does it encourage growth of the community? Is it easy to attract new contributors? In late 2007, the Eclipse project committers decided that the answers to these questions were no and they embarked on designing a new vision for Eclipse. At the same time, they realized that there were thousands of Eclipse applications that depended on the existing API. An incubator technology project was created in late 2008 with three specific goals: simplify the Eclipse programming model, attract new committers and enable the platform to take advantage of new web-based technologies while providing an open architecture.

Eclipse 4.0 was first released in July 2010 for early adopters to provide feedback. It consisted of a combination of SDK bundles that were part of the 3.6 release, and new bundles that graduated from the technology project. Like 3.0, there was a compatibility layer so that existing bundles could work with the new release. As always, there was the caveat that consumers needed to be using the public API in order to be assured of that compatibility. There was no such guarantee if your bundle used internal code. The 4.0 release provided the Eclipse 4 Application Platform which provided the following features.

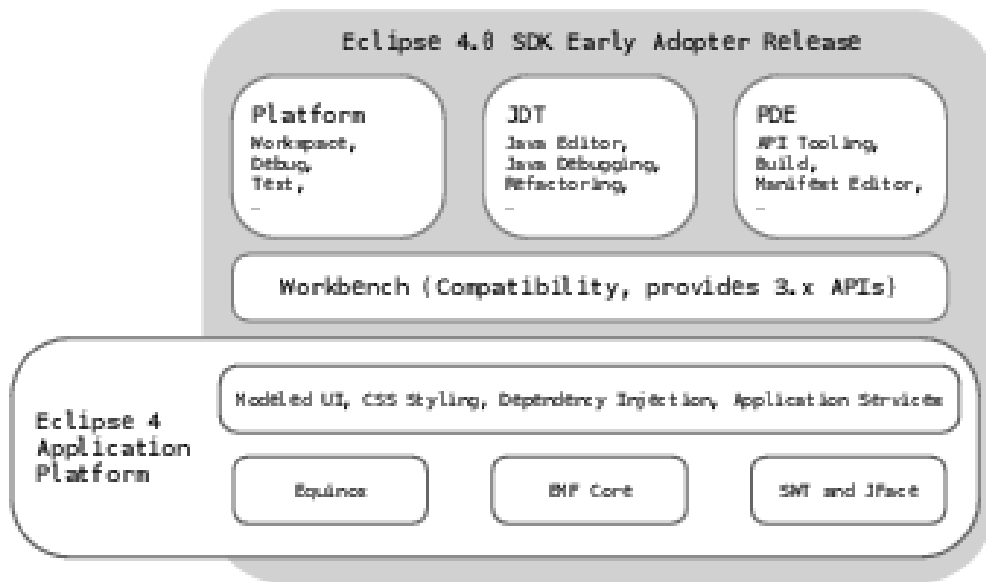


Figure 7.9: Eclipse 4.0 SDK Early Adopter Release

Model Workbench

In 4.0, a model workbench is generated using the Eclipse Modeling Framework (EMFgc). There is a separation of concerns between the model and the rendering of the view, since the renderer talks to the model and then generates the SWT code. The default is to use the SWT renderers, but other solutions are possible. If you create an example 4.x application, an XMI file will be created for the default workbench model. The model can be modified and the workbench will be instantly updated to reflect the changes in the model. Figure 7.10 is an example of a model generated for an example 4.x application.

Cascading Style Sheets Styling

Eclipse was released in 2001, before the era of rich Internet applications that could be skinned via CSS to provide a different look and feel. Eclipse 4.0 provides the ability to use stylesheets to easily change the look and feel of the Eclipse application. The default CSS stylesheets can be found in the `css` folder of the `org.eclipse.platform` bundle.

Dependency Injection

Both the Eclipse extensions registry and OSGi services are examples of service programming models. By convention, a service programming model contains service producers and consumers. The broker is responsible for managing the relationship between producers and consumers.

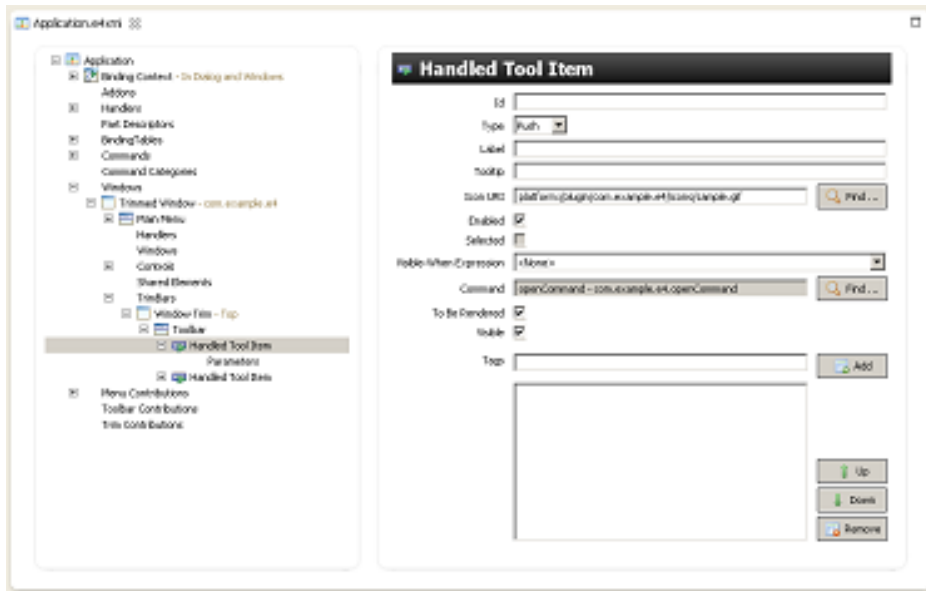


Figure 7.10: Model Generated for Example 4.x Application

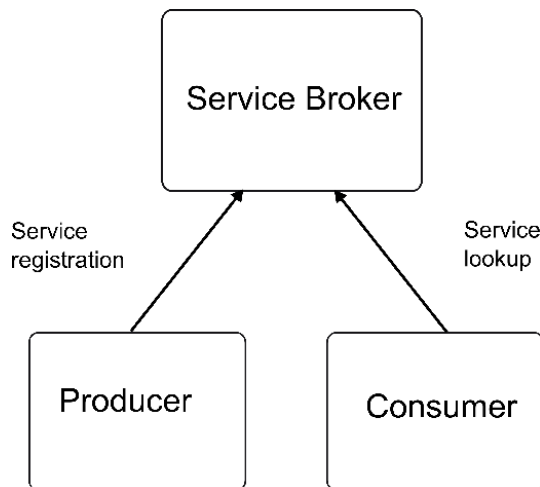


Figure 7.11: Relationship Between Producers and Consumers

Traditionally, in Eclipse 3.4.x applications, the consumer needed to know the location of the implementation, and to understand inheritance within the framework to consume services. The consumer code was therefore less reusable because people couldn't override which implementation the consumer receives. For example, if you wanted to update the message on the status line in Eclipse 3.x, the code would look like:

```
getViewSite().getActionBars().getStatusLineManager().setMessage(msg);
```

Eclipse 3.6 is built from components, but many of these components are too tightly coupled. To assemble applications of more loosely coupled components, Eclipse 4.0 uses dependency injection to provide services to clients. Dependency injection in Eclipse 4.x is through the use of a custom framework that uses the concept of a context that serves as a generic mechanism to locate services for consumers. The context exists between the application and the framework. Contexts are hierarchical. If a context has a request that cannot be satisfied, it will delegate the request to the parent context. The Eclipse context, called `IEclipseContext`, stores the available services and provides OSGi services lookup. Basically, the context is similar to a Java map in that it provides a mapping of a name or class to an object. The context handles model elements and services. Every element of the model, will have a context. Services are published in 4.x by means of the OSGi service mechanism.

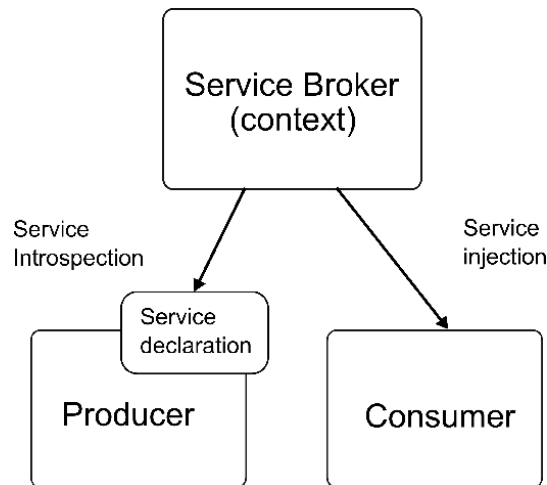


Figure 7.12: Service Broker Context

Producers add services and objects to the context which stores them. Services are injected into consumer objects by the context. The consumer declares what it wants, and the context determines how to satisfy this request. This approach has made consuming dynamic service easier. In Eclipse 3.x, a consumer had to attach listeners to be notified when services were available or unavailable. With Eclipse 4.x, once a context has been injected into a consumer object, any change is automatically delivered to that object again. In other words, dependency injection occurs again. The consumer indicates that it will use the context by the use of Java 5 annotations which adhere to the JSR 330 standard, such as `@inject`, as well as some custom Eclipse annotations. Constructor, method, and field injection are supported. The 4.x runtime scans the objects for these annotations. The action that is performed depends on the annotation that's found.

This separation of concerns between context and application allows for better reuse of components, and absolves the consumer from understanding the implementation. In 4.x, the code to update the status line would look like this:

```

@Inject
IStatusLineManager statusLine;
...
statusLine.setMessage(msg);
  
```

Application Services

One of the main goals in Eclipse 4.0 was to simplify the API for consumers so that it was easy to implement common services. The list of simple services came to be known as “the twenty things” and are known as the Eclipse Application services. The goal is to offer standalone APIs that clients can use without having to have a deep understanding of all the APIs available. They are structured as individual services so that they can also be used in other languages other than Java, such as Javascript. For example, there is an API to access the application model, to read and modify preferences and report errors and warnings.

7.5 Conclusion

The component-based architecture of Eclipse has evolved to incorporate new technology while maintaining backward compatibility. This has been costly, but the reward is the growth of the Eclipse community because of the trust established that consumers can continue to ship products based on a stable API.

Eclipse has so many consumers with diverse use cases and our expansive API became difficult for new consumers to adopt and understand. In retrospect, we should have kept our API simpler. If 80% of consumers only use 20% of the API, there is a need for simplification which was one of the reasons that the Eclipse 4.x stream was created.

The wisdom of crowds does reveal interesting use cases, such as disaggregating the IDE into bundles that could be used to construct RCP applications. Conversely, crowds often generate a lot of noise with requests for edge case scenarios that take a significant amount of time to implement.

In the early days of the Eclipse project, committers had the luxury of dedicating significant amounts of time to documentation, examples and answering community questions. Over time, this responsibility has shifted to the Eclipse community as a whole. We could have been better at providing documentation and use cases to help out the community, but this has been difficult given the large number of items planned for every release. Contrary to the expectation that software release dates slip, at Eclipse we consistently deliver our releases on time which allows our consumers to trust that they will be able to do the same.

By adopting new technology, and reinventing how Eclipse looks and works, we continue the conversation with our consumers and keep them engaged in the community. If you’re interested in becoming involved with Eclipse, please visit <http://www.eclipse.org>.

Graphite

Chris Davis

Graphite¹ performs two pretty simple tasks: storing numbers that change over time and graphing them. There has been a lot of software written over the years to do these same tasks. What makes Graphite unique is that it provides this functionality as a network service that is both easy to use and highly scalable. The protocol for feeding data into Graphite is simple enough that you could learn to do it by hand in a few minutes (not that you'd actually want to, but it's a decent litmus test for simplicity). Rendering graphs and retrieving data points are as easy as fetching a URL. This makes it very natural to integrate Graphite with other software and enables users to build powerful applications on top of Graphite. One of the most common uses of Graphite is building web-based dashboards for monitoring and analysis. Graphite was born in a high-volume e-commerce environment and its design reflects this. Scalability and real-time access to data are key goals.

The components that allow Graphite to achieve these goals include a specialized database library and its storage format, a caching mechanism for optimizing I/O operations, and a simple yet effective method of clustering Graphite servers. Rather than simply describing how Graphite works today, I will explain how Graphite was initially implemented (quite naively), what problems I ran into, and how I devised solutions to them.

8.1 The Database Library: Storing Time-Series Data

Graphite is written entirely in Python and consists of three major components: a database library named *whisper*, a back-end daemon named *carbon*, and a front-end webapp that renders graphs and provides a basic UI. While *whisper* was written specifically for Graphite, it can also be used independently. It is very similar in design to the round-robin-database used by *RRDtool*, and only stores time-series numeric data. Usually we think of databases as server processes that client applications talk to over sockets. However, *whisper*, much like *RRDtool*, is a database library used by applications to manipulate and retrieve data stored in specially formatted files. The most basic *whisper* operations are *create* to make a new *whisper* file, *update* to write new data points into a file, and *fetch* to retrieve data points.

As shown in Figure 8.1, *whisper* files consist of a header section containing various metadata, followed by one or more archive sections. Each archive is a sequence of consecutive data points

¹<http://launchpad.net/graphite>

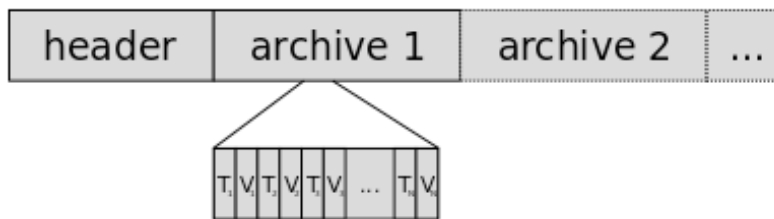


Figure 8.1: Basic Anatomy of a whisper File

which are (timestamp, value) pairs. When an update or fetch operation is performed, whisper determines the offset in the file where data should be written to or read from, based on the timestamp and the archive configuration.

8.2 The Back End: A Simple Storage Service

Graphite's back end is a daemon process called carbon-cache, usually simply referred to as carbon. It is built on Twisted, a highly scalable event-driven I/O framework for Python. Twisted enables carbon to efficiently talk to a large number of clients and handle a large amount of traffic with low overhead. Figure 8.2 shows the data flow among carbon, whisper and the webapp: Client applications collect data and send it to the Graphite back end, carbon, which stores the data using whisper. This data can then be used by the Graphite webapp to generate graphs.

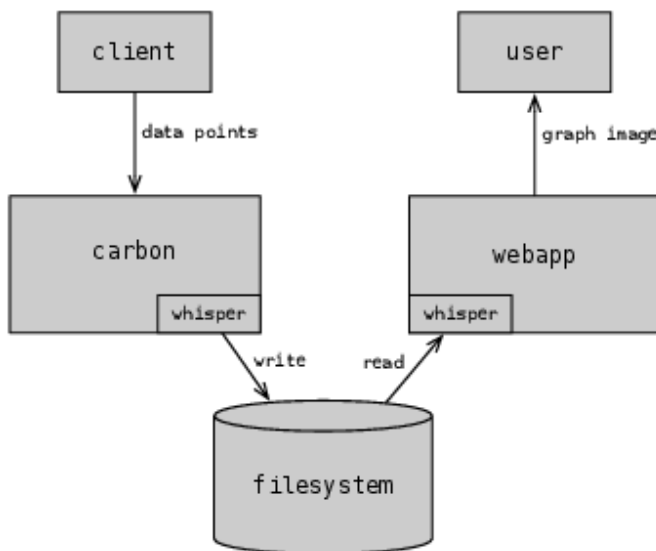


Figure 8.2: Data Flow

The primary function of carbon is to store data points for metrics provided by clients. In Graphite terminology, a metric is any measurable quantity that can vary over time (like the CPU

utilization of a server or the number of sales of a product). A data point is simply a (timestamp, value) pair corresponding to the measured value of a particular metric at a point in time. Metrics are uniquely identified by their name, and the name of each metric as well as its data points are provided by client applications. A common type of client application is a monitoring agent that collects system or application metrics, and sends its collected values to carbon for easy storage and visualization. Metrics in Graphite have simple hierarchical names, similar to filesystem paths except that a dot is used to delimit the hierarchy rather than a slash or backslash. carbon will respect any legal name and creates a whisper file for each metric to store its data points. The whisper files are stored within carbon's data directory in a filesystem hierarchy that mirrors the dot-delimited hierarchy in each metric's name, so that (for example) `servers.www01.cpuUsage` maps to `.../servers/www01/cpuUsage.wsp`.

When a client application wishes to send data points to Graphite it must establish a TCP connection to carbon, usually on port 2003². The client does all the talking; carbon does not send anything over the connection. The client sends data points in a simple plain-text format while the connection may be left open and re-used as needed. The format is one line of text per data point where each line contains the dotted metric name, value, and a Unix epoch timestamp separated by spaces. For example, a client might send:

```
servers.www01.cpuUsage 42 1286269200
products.snake-oil.salesPerMinute 123 1286269200
[one minute passes]
servers.www01.cpuUsageUser 44 1286269260
products.snake-oil.salesPerMinute 119 1286269260
```

On a high level, all carbon does is listen for data in this format and try to store it on disk as quickly as possible using whisper. Later on we will discuss the details of some tricks used to ensure scalability and get the best performance we can out of a typical hard drive.

8.3 The Front End: Graphs On-Demand

The Graphite webapp allows users to request custom graphs with a simple URL-based API. Graphing parameters are specified in the query-string of an HTTP GET request, and a PNG image is returned in response. For example, the URL:

```
http://graphite.example.com/render?target=servers.www01.cpuUsage&
width=500&height=300&from=-24h
```

requests a 500×300 graph for the metric `servers.www01.cpuUsage` and the past 24 hours of data. Actually, only the `target` parameter is required; all the others are optional and use your default values if omitted.

Graphite supports a wide variety of display options as well as data manipulation functions that follow a simple functional syntax. For example, we could graph a 10-point moving average of the metric in our previous example like this:

```
target=movingAverage(servers.www01.cpuUsage,10)
```

Functions can be nested, allowing for complex expressions and calculations.

²There is another port over which serialized objects can be sent, which is more efficient than the plain-text format. This is only needed for very high levels of traffic.

Here is another example that gives the running total of sales for the day using per-product metrics of sales-per-minute:

```
target=integral(sumSeries(products.*.salesPerMinute))\&from=midnight
```

The `sumSeries` function computes a time-series that is the sum of each metric matching the pattern `products.*.salesPerMinute`. Then `integral` computes a running total rather than a per-minute count. From here it isn't too hard to imagine how one might build a web UI for viewing and manipulating graphs. Graphite comes with its own Composer UI, shown in Figure 8.3, that does this using Javascript to modify the graph's URL parameters as the user clicks through menus of the available features.

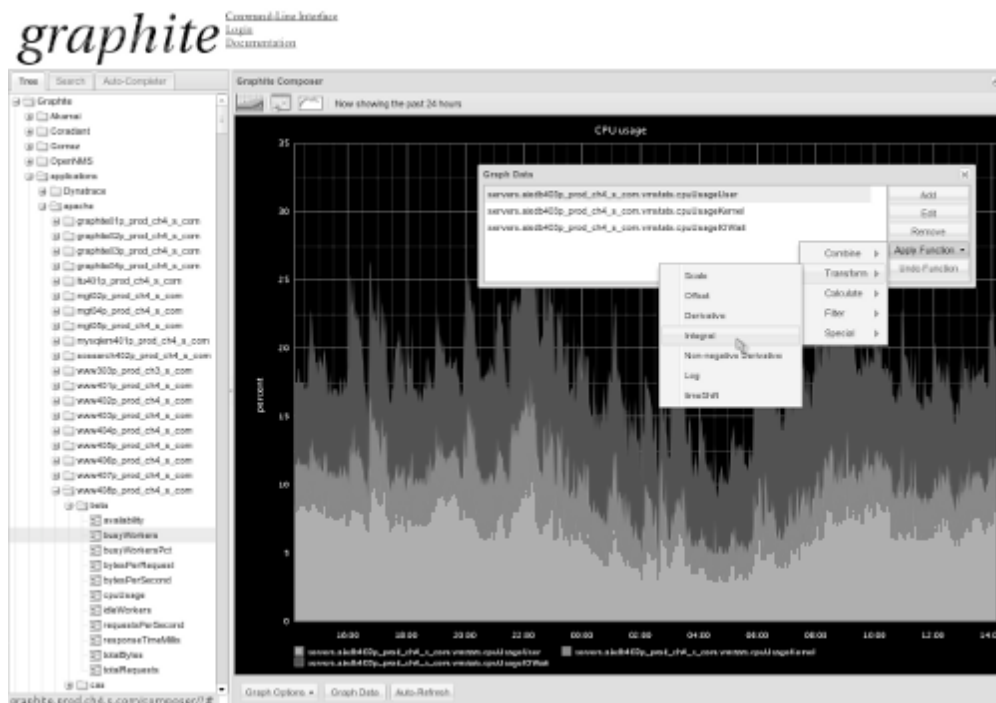


Figure 8.3: Graphite's Composer Interface

8.4 Dashboards

Since its inception Graphite has been used as a tool for creating web-based dashboards. The URL API makes this a natural use case. Making a dashboard is as simple as making an HTML page full of tags like this:

```

```

However, not everyone likes crafting URLs by hand, so Graphite's Composer UI provides a point-and-click method to create a graph from which you can simply copy and paste the URL. When

coupled with another tool that allows rapid creation of web pages (like a wiki) this becomes easy enough that non-technical users can build their own dashboards pretty easily.

8.5 An Obvious Bottleneck

Once my users started building dashboards, Graphite quickly began to have performance issues. I investigated the web server logs to see what requests were bogging it down. It was pretty obvious that the problem was the sheer number of graphing requests. The webapp was CPU-bound, rendering graphs constantly. I noticed that there were a lot of identical requests, and the dashboards were to blame.

Imagine you have a dashboard with 10 graphs in it and the page refreshes once a minute. Each time a user opens the dashboard in their browser, Graphite has to handle 10 more requests per minute. This quickly becomes expensive.

A simple solution is to render each graph only once and then serve a copy of it to each user. The Django web framework (which Graphite is built on) provides an excellent caching mechanism that can use various back ends such as memcached. Memcached³ is essentially a hash table provided as a network service. Client applications can get and set key-value pairs just like an ordinary hash table. The main benefit of using memcached is that the result of an expensive request (like rendering a graph) can be stored very quickly and retrieved later to handle subsequent requests. To avoid returning the same stale graphs forever, memcached can be configured to expire the cached graphs after a short period. Even if this is only a few seconds, the burden it takes off Graphite is tremendous because duplicate requests are so common.

Another common case that creates lots of rendering requests is when a user is tweaking the display options and applying functions in the Composer UI. Each time the user changes something, Graphite must redraw the graph. The same data is involved in each request so it makes sense to put the underlying data in the memcache as well. This keeps the UI responsive to the user because the step of retrieving data is skipped.

8.6 Optimizing I/O

Imagine that you have 60,000 metrics that you send to your Graphite server, and each of these metrics has one data point per minute. Remember that each metric has its own *whisper* file on the filesystem. This means carbon must do one write operation to 60,000 different files each minute. As long as carbon can write to one file each millisecond, it should be able to keep up. This isn't too far fetched, but let's say you have 600,000 metrics updating each minute, or your metrics are updating every second, or perhaps you simply cannot afford fast enough storage. Whatever the case, assume the rate of incoming data points exceeds the rate of write operations that your storage can keep up with. How should this situation be handled?

Most hard drives these days have slow seek time⁴, that is, the delay between doing I/O operations at two different locations, compared to writing a contiguous sequence of data. This means the more contiguous writing we do, the more throughput we get. But if we have thousands of files that need to be written to frequently, and each write is very small (one *whisper* data point is only 12 bytes) then our disks are definitely going to spend most of their time seeking.

³<http://memcached.org>

⁴Solid-state drives generally have extremely fast seek times compared to conventional hard drives.

Working under the assumption that the rate of write operations has a relatively low ceiling, the only way to increase our data point throughput beyond that rate is to write multiple data points in a single write operation. This is feasible because whisper arranges consecutive data points contiguously on disk. So I added an `update_many` function to whisper, which takes a list of data points for a single metric and compacts contiguous data points into a single write operation. Even though this made each write larger, the difference in time it takes to write ten data points (120 bytes) versus one data point (12 bytes) is negligible. It takes quite a few more data points before the size of each write starts to noticeably affect the latency.

Next I implemented a buffering mechanism in carbon. Each incoming data point gets mapped to a queue based on its metric name and is then appended to that queue. Another thread repeatedly iterates through all of the queues and for each one it pulls all of the data points out and writes them to the appropriate whisper file with `update_many`. Going back to our example, if we have 600,000 metrics updating every minute and our storage can only keep up with 1 write per millisecond, then the queues will end up holding about 10 data points each on average. The only resource this costs us is memory, which is relatively plentiful since each data point is only a few bytes.

This strategy dynamically buffers as many datapoints as necessary to sustain a rate of incoming datapoints that may exceed the rate of I/O operations your storage can keep up with. A nice advantage of this approach is that it adds a degree of resiliency to handle temporary I/O slowdowns. If the system needs to do other I/O work outside of Graphite then it is likely that the rate of write operations will decrease, in which case carbon's queues will simply grow. The larger the queues, the larger the writes. Since the overall throughput of data points is equal to the rate of write operations times the average size of each write, carbon is able to keep up as long as there is enough memory for the queues. carbon's queueing mechanism is depicted in Figure 8.4.

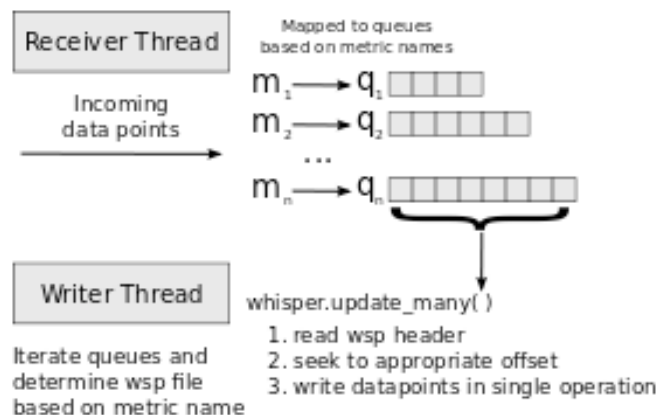


Figure 8.4: Carbon's Queueing Mechanism

8.7 Keeping It Real-Time

Buffering data points was a nice way to optimize carbon's I/O but it didn't take long for my users to notice a rather troubling side effect. Revisiting our example again, we've got 600,000 metrics that update every minute and we're assuming our storage can only keep up with 60,000 write operations

per minute. This means we will have approximately 10 minutes worth of data sitting in carbon's queues at any given time. To a user this means that the graphs they request from the Graphite webapp will be missing the most recent 10 minutes of data: Not good!

Fortunately the solution is pretty straight-forward. I simply added a socket listener to carbon that provides a query interface for accessing the buffered data points and then modifies the Graphite webapp to use this interface each time it needs to retrieve data. The webapp then combines the data points it retrieves from carbon with the data points it retrieved from disk and voila, the graphs are real-time. Granted, in our example the data points are updated to the minute and thus not exactly "real-time", but the fact that each data point is instantly accessible in a graph once it is received by carbon is real-time.

8.8 Kernels, Caches, and Catastrophic Failures

As is probably obvious by now, a key characteristic of system performance that Graphite's own performance depends on is I/O latency. So far we've assumed our system has consistently low I/O latency averaging around 1 millisecond per write, but this is a big assumption that requires a little deeper analysis. Most hard drives simply aren't that fast; even with dozens of disks in a RAID array there is very likely to be more than 1 millisecond latency for random access. Yet if you were to try and test how quickly even an old laptop could write a whole kilobyte to disk you would find that the write system call returns in far less than 1 millisecond. Why?

Whenever software has inconsistent or unexpected performance characteristics, usually either buffering or caching is to blame. In this case, we're dealing with both. The write system call doesn't technically write your data to disk, it simply puts it in a buffer which the kernel then writes to disk later on. This is why the write call usually returns so quickly. Even after the buffer has been written to disk, it often remains cached for subsequent reads. Both of these behaviors, buffering and caching, require memory of course.

Kernel developers, being the smart folks that they are, decided it would be a good idea to use whatever user-space memory is currently free instead of allocating memory outright. This turns out to be a tremendously useful performance booster and it also explains why no matter how much memory you add to a system it will usually end up having almost zero "free" memory after doing a modest amount of I/O. If your user-space applications aren't using that memory then your kernel probably is. The downside of this approach is that this "free" memory can be taken away from the kernel the moment a user-space application decides it needs to allocate more memory for itself. The kernel has no choice but to relinquish it, losing whatever buffers may have been there.

So what does all of this mean for Graphite? We just highlighted carbon's reliance on consistently low I/O latency and we also know that the write system call only returns quickly because the data is merely being copied into a buffer. What happens when there is not enough memory for the kernel to continue buffering writes? The writes become synchronous and thus terribly slow! This causes a dramatic drop in the rate of carbon's write operations, which causes carbon's queues to grow, which eats up even more memory, starving the kernel even further. In the end, this kind of situation usually results in carbon running out of memory or being killed by an angry sysadmin.

To avoid this kind of catastrophe, I added several features to carbon including configurable limits on how many data points can be queued and rate-limits on how quickly various whisper operations can be performed. These features can protect carbon from spiraling out of control and instead impose less harsh effects like dropping some data points or refusing to accept more data points. However, proper values for those settings are system-specific and require a fair amount of

testing to tune. They are useful but they do not fundamentally solve the problem. For that, we'll need more hardware.

8.9 Clustering

Making multiple Graphite servers appear to be a single system from a user perspective isn't terribly difficult, at least for a naïve implementation. The webapp's user interaction primarily consists of two operations: finding metrics and fetching data points (usually in the form of a graph). The find and fetch operations of the webapp are tucked away in a library that abstracts their implementation from the rest of the codebase, and they are also exposed through HTTP request handlers for easy remote calls.

The find operation searches the local filesystem of whisper data for things matching a user-specified pattern, just as a filesystem glob like *.txt matches files with that extension. Being a tree structure, the result returned by find is a collection of Node objects, each deriving from either the Branch or Leaf sub-classes of Node. Directories correspond to branch nodes and whisper files correspond to leaf nodes. This layer of abstraction makes it easy to support different types of underlying storage including RRD files⁵ and gzipped whisper files.

The Leaf interface defines a fetch method whose implementation depends on the type of leaf node. In the case of whisper files it is simply a thin wrapper around the whisper library's own fetch function. When clustering support was added, the find function was extended to be able to make remote find calls via HTTP to other Graphite servers specified in the webapp's configuration. The node data contained in the results of these HTTP calls gets wrapped as RemoteNode objects which conform to the usual Node, Branch, and Leaf interfaces. This makes the clustering transparent to the rest of the webapp's codebase. The fetch method for a remote leaf node is implemented as another HTTP call to retrieve the data points from the node's Graphite server.

All of these calls are made between the webapps the same way a client would call them, except with one additional parameter specifying that the operation should only be performed locally and not be redistributed throughout the cluster. When the webapp is asked to render a graph, it performs the find operation to locate the requested metrics and calls fetch on each to retrieve their data points. This works whether the data is on the local server, remote servers, or both. If a server goes down, the remote calls timeout fairly quickly and the server is marked as being out of service for a short period during which no further calls to it will be made. From a user standpoint, whatever data was on the lost server will be missing from their graphs unless that data is duplicated on another server in the cluster.

A Brief Analysis of Clustering Efficiency

The most expensive part of a graphing request is rendering the graph. Each rendering is performed by a single server so adding more servers does effectively increase capacity for rendering graphs. However, the fact that many requests end up distributing find calls to every other server in the cluster means that our clustering scheme is sharing much of the front-end load rather than dispersing it. What we have achieved at this point, however, is an effective way to distribute back-end load, as each carbon instance operates independently. This is a good first step since most of the time the back end is a bottleneck far before the front end is, but clearly the front end will not scale horizontally with this approach.

⁵RRD files are actually branch nodes because they can contain multiple data sources; an RRD data source is a leaf node.

In order to make the front end scale more effectively, the number of remote find calls made by the webapp must be reduced. Again, the easiest solution is caching. Just as memcached is already used to cache data points and rendered graphs, it can also be used to cache the results of find requests. Since the location of metrics is much less likely to change frequently, this should typically be cached for longer. The trade-off of setting the cache timeout for find results too long, though, is that new metrics that have been added to the hierarchy may not appear as quickly to the user.

Distributing Metrics in a Cluster

The Graphite webapp is rather homogeneous throughout a cluster, in that it performs the exact same job on each server. carbon's role, however, can vary from server to server depending on what data you choose to send to each instance. Often there are many different clients sending data to carbon, so it would be quite annoying to couple each client's configuration with your Graphite cluster's layout. Application metrics may go to one carbon server, while business metrics may get sent to multiple carbon servers for redundancy.

To simplify the management of scenarios like this, Graphite comes with an additional tool called carbon-relay. Its job is quite simple; it receives metric data from clients exactly like the standard carbon daemon (which is actually named carbon-cache) but instead of storing the data, it applies a set of rules to the metric names to determine which carbon-cache servers to relay the data to. Each rule consists of a regular expression and a list of destination servers. For each data point received, the rules are evaluated in order and the first rule whose regular expression matches the metric name is used. This way all the clients need to do is send their data to the carbon-relay and it will end up on the right servers.

In a sense carbon-relay provides replication functionality, though it would more accurately be called input duplication since it does not deal with synchronization issues. If a server goes down temporarily, it will be missing the data points for the time period in which it was down but otherwise function normally. There are administrative scripts that leave control of the re-synchronization process in the hands of the system administrator.

8.10 Design Reflections

My experience in working on Graphite has reaffirmed a belief of mine that scalability has very little to do with low-level performance but instead is a product of overall design. I have run into many bottlenecks along the way but each time I look for improvements in design rather than speed-ups in performance. I have been asked many times why I wrote Graphite in Python rather than Java or C++, and my response is always that I have yet to come across a true need for the performance that another language could offer. In [Knu74], Donald Knuth famously said that premature optimization is the root of all evil. As long as we assume that our code will continue to evolve in non-trivial ways then all optimization⁶ is in some sense premature.

One of Graphite's greatest strengths and greatest weaknesses is the fact that very little of it was actually "designed" in the traditional sense. By and large Graphite evolved gradually, hurdle by hurdle, as problems arose. Many times the hurdles were foreseeable and various pre-emptive solutions seemed natural. However it can be useful to avoid solving problems you do not actually have yet, even if it seems likely that you soon will. The reason is that you can learn much more from closely studying actual failures than from theorizing about superior strategies. Problem solving

⁶Knuth specifically meant low-level code optimization, not macroscopic optimization such as design improvements.

is driven by both the empirical data we have at hand and our own knowledge and intuition. I've found that doubting your own wisdom sufficiently can force you to look at your empirical data more thoroughly.

For example, when I first wrote `whisper` I was convinced that it would have to be rewritten in C for speed and that my Python implementation would only serve as a prototype. If I weren't under a time-crunch I very well may have skipped the Python implementation entirely. It turns out however that I/O is a bottleneck so much earlier than CPU that the lesser efficiency of Python hardly matters at all in practice.

As I said, though, the evolutionary approach is also a great weakness of Graphite. Interfaces, it turns out, do not lend themselves well to gradual evolution. A good interface is consistent and employs conventions to maximize predictability. By this measure, Graphite's URL API is currently a sub-par interface in my opinion. Options and functions have been tacked on over time, sometimes forming small islands of consistency, but overall lacking a global sense of consistency. The only way to solve such a problem is through versioning of interfaces, but this too has drawbacks. Once a new interface is designed, the old one is still hard to get rid of, lingering around as evolutionary baggage like the human appendix. It may seem harmless enough until one day your code gets appendicitis (i.e. a bug tied to the old interface) and you're forced to operate. If I were to change one thing about Graphite early on, it would have been to take much greater care in designing the external APIs, thinking ahead instead of evolving them bit by bit.

Another aspect of Graphite that causes some frustration is the limited flexibility of the hierarchical metric naming model. While it is quite simple and very convenient for most use cases, it makes some sophisticated queries very difficult, even impossible, to express. When I first thought of creating Graphite I knew from the very beginning that I wanted a human-editable URL API for creating graphs⁷. While I'm still glad that Graphite provides this today, I'm afraid this requirement has burdened the API with excessively simple syntax that makes complex expressions unwieldy. A hierarchy makes the problem of determining the "primary key" for a metric quite simple because a path is essentially a primary key for a node in the tree. The downside is that all of the descriptive data (i.e. column data) must be embedded directly in the path. A potential solution is to maintain the hierarchical model and add a separate metadata database to enable more advanced selection of metrics with a special syntax.

8.11 Becoming Open Source

Looking back at the evolution of Graphite, I am still surprised both by how far it has come as a project and by how far it has taken me as a programmer. It started as a pet project that was only a few hundred lines of code. The rendering engine started as an experiment, simply to see if I could write one. `whisper` was written over the course of a weekend out of desperation to solve a show-stopper problem before a critical launch date. `carbon` has been rewritten more times than I care to remember. Once I was allowed to release Graphite under an open source license in 2008 I never really expected much response. After a few months it was mentioned in a CNET article that got picked up by Slashdot and the project suddenly took off and has been active ever since. Today there are dozens of large and mid-sized companies using Graphite. The community is quite active and continues to grow. Far from being a finished product, there is a lot of cool experimental work being done, which keeps it fun to work on and full of potential.

⁷This forces the graphs themselves to be open source. Anyone can simply look at a graph's URL to understand it or modify it.

The Hadoop Distributed File System

Robert Chansler, Hairong Kuang, Sanjay Radia,
Konstantin Shvachko, and Suresh Srinivas

The Hadoop Distributed File System (HDFS) is designed to store very large data sets reliably, and to stream those data sets at high bandwidth to user applications. In a large cluster, thousands of servers both host directly attached storage and execute user application tasks. By distributing storage and computation across many servers, the resource can grow with demand while remaining economical at every size. We describe the architecture of HDFS and report on experience using HDFS to manage 40 petabytes of enterprise data at Yahoo!

9.1 Introduction

Hadoop¹ provides a distributed filesystem and a framework for the analysis and transformation of very large data sets using the MapReduce [DG04] paradigm. While the interface to HDFS is patterned after the Unix filesystem, faithfulness to standards was sacrificed in favor of improved performance for the applications at hand.

An important characteristic of Hadoop is the partitioning of data and computation across many (thousands) of hosts, and the execution of application computations in parallel close to their data. A Hadoop cluster scales computation capacity, storage capacity and I/O bandwidth by simply adding commodity servers. Hadoop clusters at Yahoo! span 40,000 servers, and store 40 petabytes of application data, with the largest cluster being 4000 servers. One hundred other organizations worldwide report using Hadoop.

HDFS stores filesystem metadata and application data separately. As in other distributed filesystems, like PVFS [CIRT00], Lustre², and GFS [GGL03, MQ09], HDFS stores metadata on a dedicated server, called the NameNode. Application data are stored on other servers called DataNodes. All servers are fully connected and communicate with each other using TCP-based protocols. Unlike Lustre and PVFS, the DataNodes in HDFS do not rely on data protection mechanisms such as RAID to make the data durable. Instead, like GFS, the file content is replicated on multiple DataNodes for reliability. While ensuring data durability, this strategy has the added advantage that data transfer

¹<http://hadoop.apache.org>

²<http://www.lustre.org>

bandwidth is multiplied, and there are more opportunities for locating computation near the needed data.

9.2 Architecture

NameNode

The HDFS namespace is a hierarchy of files and directories. Files and directories are represented on the NameNode by inodes. Inodes record attributes like permissions, modification and access times, namespace and disk space quotas. The file content is split into large blocks (typically 128 megabytes, but user selectable file-by-file), and each block of the file is independently replicated at multiple DataNodes (typically three, but user selectable file-by-file). The NameNode maintains the namespace tree and the mapping of blocks to DataNodes. The current design has a single NameNode for each cluster. The cluster can have thousands of DataNodes and tens of thousands of HDFS clients per cluster, as each DataNode may execute multiple application tasks concurrently.

Image and Journal

The inodes and the list of blocks that define the metadata of the name system are called the *image*. NameNode keeps the entire namespace image in RAM. The persistent record of the image stored in the NameNode's local native filesystem is called a checkpoint. The NameNode records changes to HDFS in a write-ahead log called the journal in its local native filesystem. The location of block replicas are not part of the persistent checkpoint.

Each client-initiated transaction is recorded in the journal, and the journal file is flushed and synced before the acknowledgment is sent to the client. The checkpoint file is never changed by the NameNode; a new file is written when a checkpoint is created during restart, when requested by the administrator, or by the CheckpointNode described in the next section. During startup the NameNode initializes the namespace image from the checkpoint, and then replays changes from the journal. A new checkpoint and an empty journal are written back to the storage directories before the NameNode starts serving clients.

For improved durability, redundant copies of the checkpoint and journal are typically stored on multiple independent local volumes and at remote NFS servers. The first choice prevents loss from a single volume failure, and the second choice protects against failure of the entire node. If the NameNode encounters an error writing the journal to one of the storage directories it automatically excludes that directory from the list of storage directories. The NameNode automatically shuts itself down if no storage directory is available.

The NameNode is a multithreaded system and processes requests simultaneously from multiple clients. Saving a transaction to disk becomes a bottleneck since all other threads need to wait until the synchronous flush-and-sync procedure initiated by one of them is complete. In order to optimize this process, the NameNode batches multiple transactions. When one of the NameNode's threads initiates a flush-and-sync operation, all the transactions batched at that time are committed together. Remaining threads only need to check that their transactions have been saved and do not need to initiate a flush-and-sync operation.

DataNodes

Each block replica on a DataNode is represented by two files in the local native filesystem. The first file contains the data itself and the second file records the block's metadata including checksums for the data and the generation stamp. The size of the data file equals the actual length of the block and does not require extra space to round it up to the nominal block size as in traditional filesystems. Thus, if a block is half full it needs only half of the space of the full block on the local drive.

During startup each DataNode connects to the NameNode and performs a handshake. The purpose of the handshake is to verify the namespace ID and the software version of the DataNode. If either does not match that of the NameNode, the DataNode automatically shuts down.

The namespace ID is assigned to the filesystem instance when it is formatted. The namespace ID is persistently stored on all nodes of the cluster. Nodes with a different namespace ID will not be able to join the cluster, thus protecting the integrity of the filesystem. A DataNode that is newly initialized and without any namespace ID is permitted to join the cluster and receive the cluster's namespace ID.

After the handshake the DataNode registers with the NameNode. DataNodes persistently store their unique storage IDs. The storage ID is an internal identifier of the DataNode, which makes it recognizable even if it is restarted with a different IP address or port. The storage ID is assigned to the DataNode when it registers with the NameNode for the first time and never changes after that.

A DataNode identifies block replicas in its possession to the NameNode by sending a block report. A block report contains the block ID, the generation stamp and the length for each block replica the server hosts. The first block report is sent immediately after the DataNode registration. Subsequent block reports are sent every hour and provide the NameNode with an up-to-date view of where block replicas are located on the cluster.

During normal operation DataNodes send heartbeats to the NameNode to confirm that the DataNode is operating and the block replicas it hosts are available. The default heartbeat interval is three seconds. If the NameNode does not receive a heartbeat from a DataNode in ten minutes the NameNode considers the DataNode to be out of service and the block replicas hosted by that DataNode to be unavailable. The NameNode then schedules creation of new replicas of those blocks on other DataNodes.

Heartbeats from a DataNode also carry information about total storage capacity, fraction of storage in use, and the number of data transfers currently in progress. These statistics are used for the NameNode's block allocation and load balancing decisions.

The NameNode does not directly send requests to DataNodes. It uses replies to heartbeats to send instructions to the DataNodes. The instructions include commands to replicate blocks to other nodes, remove local block replicas, re-register and send an immediate block report, and shut down the node.

These commands are important for maintaining the overall system integrity and therefore it is critical to keep heartbeats frequent even on big clusters. The NameNode can process thousands of heartbeats per second without affecting other NameNode operations.

HDFS Client

User applications access the filesystem using the HDFS client, a library that exports the HDFS filesystem interface.

Like most conventional filesystems, HDFS supports operations to read, write and delete files, and operations to create and delete directories. The user references files and directories by paths in

the namespace. The user application does not need to know that filesystem metadata and storage are on different servers, or that blocks have multiple replicas.

When an application reads a file, the HDFS client first asks the NameNode for the list of DataNodes that host replicas of the blocks of the file. The list is sorted by the network topology distance from the client. The client contacts a DataNode directly and requests the transfer of the desired block. When a client writes, it first asks the NameNode to choose DataNodes to host replicas of the first block of the file. The client organizes a pipeline from node-to-node and sends the data. When the first block is filled, the client requests new DataNodes to be chosen to host replicas of the next block. A new pipeline is organized, and the client sends the further bytes of the file. Choice of DataNodes for each block is likely to be different. The interactions among the client, the NameNode and the DataNodes are illustrated in Figure 9.1.

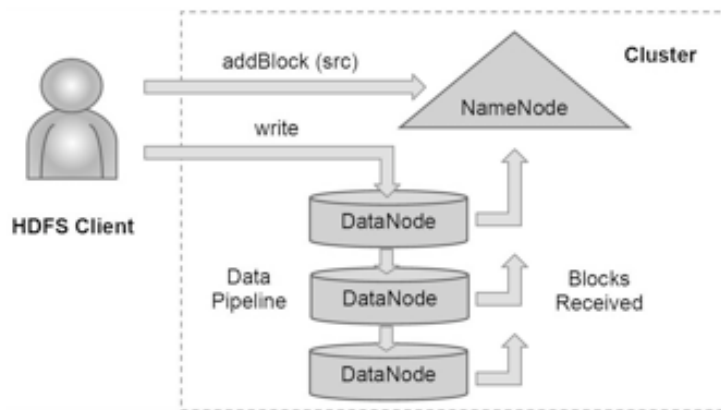


Figure 9.1: HDFS Client Creates a New File

Unlike conventional filesystems, HDFS provides an API that exposes the locations of a file blocks. This allows applications like the MapReduce framework to schedule a task to where the data are located, thus improving the read performance. It also allows an application to set the replication factor of a file. By default a file's replication factor is three. For critical files or files which are accessed very often, having a higher replication factor improves tolerance against faults and increases read bandwidth.

CheckpointNode

The NameNode in HDFS, in addition to its primary role serving client requests, can alternatively execute either of two other roles, either a CheckpointNode or a BackupNode. The role is specified at the node startup.

The CheckpointNode periodically combines the existing checkpoint and journal to create a new checkpoint and an empty journal. The CheckpointNode usually runs on a different host from the NameNode since it has the same memory requirements as the NameNode. It downloads the current checkpoint and journal files from the NameNode, merges them locally, and returns the new checkpoint back to the NameNode.

Creating periodic checkpoints is one way to protect the filesystem metadata. The system can start from the most recent checkpoint if all other persistent copies of the namespace image or journal

are unavailable. Creating a checkpoint also lets the NameNode truncate the journal when the new checkpoint is uploaded to the NameNode. HDFS clusters run for prolonged periods of time without restarts during which the journal constantly grows. If the journal grows very large, the probability of loss or corruption of the journal file increases. Also, a very large journal extends the time required to restart the NameNode. For a large cluster, it takes an hour to process a week-long journal. Good practice is to create a daily checkpoint.

BackupNode

A recently introduced feature of HDFS is the BackupNode. Like a CheckpointNode, the BackupNode is capable of creating periodic checkpoints, but in addition it maintains an in-memory, up-to-date image of the filesystem namespace that is always synchronized with the state of the NameNode.

The BackupNode accepts the journal stream of namespace transactions from the active NameNode, saves them in journal on its own storage directories, and applies these transactions to its own namespace image in memory. The NameNode treats the BackupNode as a journal store the same way as it treats journal files in its storage directories. If the NameNode fails, the BackupNode's image in memory and the checkpoint on disk is a record of the latest namespace state.

The BackupNode can create a checkpoint without downloading checkpoint and journal files from the active NameNode, since it already has an up-to-date namespace image in its memory. This makes the checkpoint process on the BackupNode more efficient as it only needs to save the namespace into its local storage directories.

The BackupNode can be viewed as a read-only NameNode. It contains all filesystem metadata information except for block locations. It can perform all operations of the regular NameNode that do not involve modification of the namespace or knowledge of block locations. Use of a BackupNode provides the option of running the NameNode without persistent storage, delegating responsibility of persisting the namespace state to the BackupNode.

Upgrades and Filesystem Snapshots

During software upgrades the possibility of corrupting the filesystem due to software bugs or human mistakes increases. The purpose of creating snapshots in HDFS is to minimize potential damage to the data stored in the system during upgrades.

The snapshot mechanism lets administrators persistently save the current state of the filesystem, so that if the upgrade results in data loss or corruption it is possible to rollback the upgrade and return HDFS to the namespace and storage state as they were at the time of the snapshot.

The snapshot (only one can exist) is created at the cluster administrator's option whenever the system is started. If a snapshot is requested, the NameNode first reads the checkpoint and journal files and merges them in memory. Then it writes the new checkpoint and the empty journal to a new location, so that the old checkpoint and journal remain unchanged.

During handshake the NameNode instructs DataNodes whether to create a local snapshot. The local snapshot on the DataNode cannot be created by replicating the directories containing the data files as this would require doubling the storage capacity of every DataNode on the cluster. Instead each DataNode creates a copy of the storage directory and hard links existing block files into it. When the DataNode removes a block it removes only the hard link, and block modifications during appends use the copy-on-write technique. Thus old block replicas remain untouched in their old directories.

The cluster administrator can choose to roll back HDFS to the snapshot state when restarting the system. The NameNode recovers the checkpoint saved when the snapshot was created. DataNodes restore the previously renamed directories and initiate a background process to delete block replicas created after the snapshot was made. Having chosen to roll back, there is no provision to roll forward. The cluster administrator can recover the storage occupied by the snapshot by commanding the system to abandon the snapshot; for snapshots created during upgrade, this finalizes the software upgrade.

System evolution may lead to a change in the format of the NameNode's checkpoint and journal files, or in the data representation of block replica files on DataNodes. The layout version identifies the data representation formats, and is persistently stored in the NameNode's and the DataNodes' storage directories. During startup each node compares the layout version of the current software with the version stored in its storage directories and automatically converts data from older formats to the newer ones. The conversion requires the mandatory creation of a snapshot when the system restarts with the new software layout version.

9.3 File I/O Operations and Replica Management

Of course, the whole point of a filesystem is to store data in files. To understand how HDFS does this, we must look at how reading and writing works, and how blocks are managed.

File Read and Write

An application adds data to HDFS by creating a new file and writing the data to it. After the file is closed, the bytes written cannot be altered or removed except that new data can be added to the file by reopening the file for append. HDFS implements a single-writer, multiple-reader model.

The HDFS client that opens a file for writing is granted a lease for the file; no other client can write to the file. The writing client periodically renews the lease by sending a heartbeat to the NameNode. When the file is closed, the lease is revoked. The lease duration is bound by a soft limit and a hard limit. Until the soft limit expires, the writer is certain of exclusive access to the file. If the soft limit expires and the client fails to close the file or renew the lease, another client can preempt the lease. If after the hard limit expires (one hour) and the client has failed to renew the lease, HDFS assumes that the client has quit and will automatically close the file on behalf of the writer, and recover the lease. The writer's lease does not prevent other clients from reading the file; a file may have many concurrent readers.

An HDFS file consists of blocks. When there is a need for a new block, the NameNode allocates a block with a unique block ID and determines a list of DataNodes to host replicas of the block. The DataNodes form a pipeline, the order of which minimizes the total network distance from the client to the last DataNode. Bytes are pushed to the pipeline as a sequence of packets. The bytes that an application writes first buffer at the client side. After a packet buffer is filled (typically 64 KB), the data are pushed to the pipeline. The next packet can be pushed to the pipeline before receiving the acknowledgment for the previous packets. The number of outstanding packets is limited by the outstanding packets window size of the client.

After data are written to an HDFS file, HDFS does not provide any guarantee that data are visible to a new reader until the file is closed. If a user application needs the visibility guarantee, it can explicitly call the `hflush` operation. Then the current packet is immediately pushed to the pipeline, and the `hflush` operation will wait until all DataNodes in the pipeline acknowledge the successful

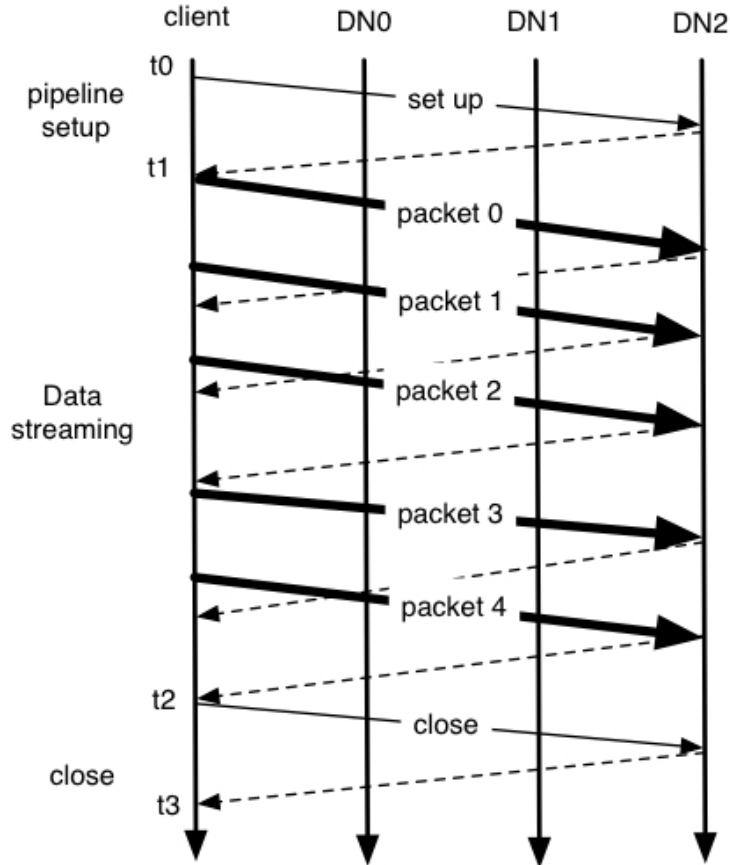


Figure 9.2: Data Pipeline While Writing a Block

transmission of the packet. All data written before the hflush operation are then certain to be visible to readers.

If no error occurs, block construction goes through three stages as shown in Figure 9.2 illustrating a pipeline of three DataNodes (DN) and a block of five packets. In the picture, bold lines represent data packets, dashed lines represent acknowledgment messages, and thin lines represent control messages to setup and close the pipeline. Vertical lines represent activity at the client and the three DataNodes where time proceeds from top to bottom. From t_0 to t_1 is the pipeline setup stage. The interval t_1 to t_2 is the data streaming stage, where t_1 is the time when the first data packet gets sent and t_2 is the time that the acknowledgment to the last packet gets received. Here an hflush operation transmits packet 2. The hflush indication travels with the packet data and is not a separate operation. The final interval t_2 to t_3 is the pipeline close stage for this block.

In a cluster of thousands of nodes, failures of a node (most commonly storage faults) are daily occurrences. A replica stored on a DataNode may become corrupted because of faults in memory,

disk, or network. HDFS generates and stores checksums for each data block of an HDFS file. Checksums are verified by the HDFS client while reading to help detect any corruption caused either by client, DataNodes, or network. When a client creates an HDFS file, it computes the checksum sequence for each block and sends it to a DataNode along with the data. A DataNode stores checksums in a metadata file separate from the block's data file. When HDFS reads a file, each block's data and checksums are shipped to the client. The client computes the checksum for the received data and verifies that the newly computed checksums matches the checksums it received. If not, the client notifies the NameNode of the corrupt replica and then fetches a different replica of the block from another DataNode.

When a client opens a file to read, it fetches the list of blocks and the locations of each block replica from the NameNode. The locations of each block are ordered by their distance from the reader. When reading the content of a block, the client tries the closest replica first. If the read attempt fails, the client tries the next replica in sequence. A read may fail if the target DataNode is unavailable, the node no longer hosts a replica of the block, or the replica is found to be corrupt when checksums are tested.

HDFS permits a client to read a file that is open for writing. When reading a file open for writing, the length of the last block still being written is unknown to the NameNode. In this case, the client asks one of the replicas for the latest length before starting to read its content.

The design of HDFS I/O is particularly optimized for batch processing systems, like MapReduce, which require high throughput for sequential reads and writes. Ongoing efforts will improve read/write response time for applications that require real-time data streaming or random access.

Block Placement

For a large cluster, it may not be practical to connect all nodes in a flat topology. A common practice is to spread the nodes across multiple racks. Nodes of a rack share a switch, and rack switches are connected by one or more core switches. Communication between two nodes in different racks has to go through multiple switches. In most cases, network bandwidth between nodes in the same rack is greater than network bandwidth between nodes in different racks. Figure 9.3 describes a cluster with two racks, each of which contains three nodes.

HDFS estimates the network bandwidth between two nodes by their distance. The distance from a node to its parent node is assumed to be one. A distance between two nodes can be calculated by summing the distances to their closest common ancestor. A shorter distance between two nodes means greater bandwidth they can use to transfer data.

HDFS allows an administrator to configure a script that returns a node's rack identification given a node's address. The NameNode is the central place that resolves the rack location of each DataNode. When a DataNode registers with the NameNode, the NameNode runs the configured script to decide which rack the node belongs to. If no such a script is configured, the NameNode assumes that all the nodes belong to a default single rack.

The placement of replicas is critical to HDFS data reliability and read/write performance. A good replica placement policy should improve data reliability, availability, and network bandwidth utilization. Currently HDFS provides a configurable block placement policy interface so that the users and researchers can experiment and test alternate policies that are optimal for their applications.

The default HDFS block placement policy provides a tradeoff between minimizing the write cost, and maximizing data reliability, availability and aggregate read bandwidth. When a new block is created, HDFS places the first replica on the node where the writer is located. The second and the third replicas are placed on two different nodes in a different rack. The rest are placed on random

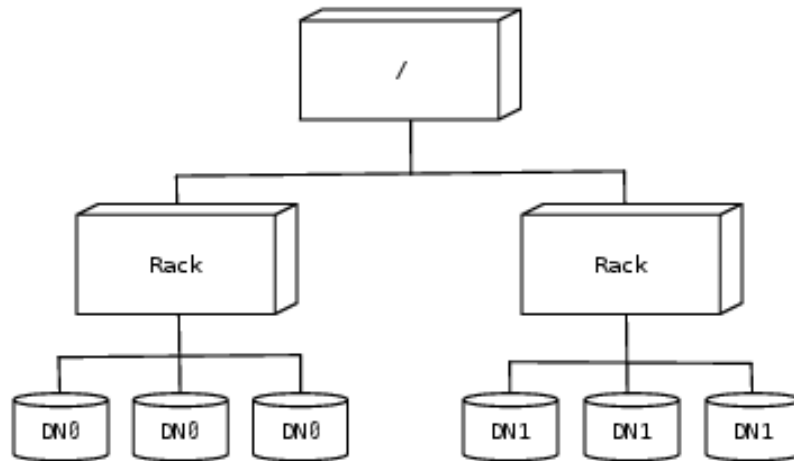


Figure 9.3: Cluster Topology

nodes with restrictions that no more than one replica is placed at any one node and no more than two replicas are placed in the same rack, if possible. The choice to place the second and third replicas on a different rack better distributes the block replicas for a single file across the cluster. If the first two replicas were placed on the same rack, for any file, two-thirds of its block replicas would be on the same rack.

After all target nodes are selected, nodes are organized as a pipeline in the order of their proximity to the first replica. Data are pushed to nodes in this order. For reading, the NameNode first checks if the client's host is located in the cluster. If yes, block locations are returned to the client in the order of its closeness to the reader. The block is read from DataNodes in this preference order.

This policy reduces the inter-rack and inter-node write traffic and generally improves write performance. Because the chance of a rack failure is far less than that of a node failure, this policy does not impact data reliability and availability guarantees. In the usual case of three replicas, it can reduce the aggregate network bandwidth used when reading data since a block is placed in only two unique racks rather than three.

Replication Management

The NameNode endeavors to ensure that each block always has the intended number of replicas. The NameNode detects that a block has become under- or over-replicated when a block report from a DataNode arrives. When a block becomes over replicated, the NameNode chooses a replica to remove. The NameNode will prefer not to reduce the number of racks that host replicas, and secondly prefer to remove a replica from the DataNode with the least amount of available disk space. The goal is to balance storage utilization across DataNodes without reducing the block's availability.

When a block becomes under-replicated, it is put in the replication priority queue. A block with only one replica has the highest priority, while a block with a number of replicas that is greater than two thirds of its replication factor has the lowest priority. A background thread periodically scans the head of the replication queue to decide where to place new replicas. Block replication follows a similar policy as that of new block placement. If the number of existing replicas is one, HDFS places the next replica on a different rack. In case that the block has two existing replicas, if the two

existing replicas are on the same rack, the third replica is placed on a different rack; otherwise, the third replica is placed on a different node in the same rack as an existing replica. Here the goal is to reduce the cost of creating new replicas.

The NameNode also makes sure that not all replicas of a block are located on one rack. If the NameNode detects that a block's replicas end up at one rack, the NameNode treats the block as mis-replicated and replicates the block to a different rack using the same block placement policy described above. After the NameNode receives the notification that the replica is created, the block becomes over-replicated. The NameNode then will decide to remove an old replica because the over-replication policy prefers not to reduce the number of racks.

Balancer

HDFS block placement strategy does not take into account DataNode disk space utilization. This is to avoid placing new—more likely to be referenced—data at a small subset of the DataNodes with a lot of free storage. Therefore data might not always be placed uniformly across DataNodes. Imbalance also occurs when new nodes are added to the cluster.

The balancer is a tool that balances disk space usage on an HDFS cluster. It takes a threshold value as an input parameter, which is a fraction between 0 and 1. A cluster is balanced if, for each DataNode, the utilization of the node³ differs from the utilization of the whole cluster⁴ by no more than the threshold value.

The tool is deployed as an application program that can be run by the cluster administrator. It iteratively moves replicas from DataNodes with higher utilization to DataNodes with lower utilization. One key requirement for the balancer is to maintain data availability. When choosing a replica to move and deciding its destination, the balancer guarantees that the decision does not reduce either the number of replicas or the number of racks.

The balancer optimizes the balancing process by minimizing the inter-rack data copying. If the balancer decides that a replica A needs to be moved to a different rack and the destination rack happens to have a replica B of the same block, the data will be copied from replica B instead of replica A.

A configuration parameter limits the bandwidth consumed by rebalancing operations. The higher the allowed bandwidth, the faster a cluster can reach the balanced state, but with greater competition with application processes.

Block Scanner

Each DataNode runs a block scanner that periodically scans its block replicas and verifies that stored checksums match the block data. In each scan period, the block scanner adjusts the read bandwidth in order to complete the verification in a configurable period. If a client reads a complete block and checksum verification succeeds, it informs the DataNode. The DataNode treats it as a verification of the replica.

The verification time of each block is stored in a human-readable log file. At any time there are up to two files in the top-level DataNode directory, the current and previous logs. New verification times are appended to the current file. Correspondingly, each DataNode has an in-memory scanning list ordered by the replica's verification time.

³Defined as the ratio of used space at the node to total capacity of the node.

⁴Defined as the ratio of used space in the cluster to total capacity of the cluster.

Whenever a read client or a block scanner detects a corrupt block, it notifies the NameNode. The NameNode marks the replica as corrupt, but does not schedule deletion of the replica immediately. Instead, it starts to replicate a good copy of the block. Only when the good replica count reaches the replication factor of the block the corrupt replica is scheduled to be removed. This policy aims to preserve data as long as possible. So even if all replicas of a block are corrupt, the policy allows the user to retrieve its data from the corrupt replicas.

Decommissioning

The cluster administrator specifies list of nodes to be decommissioned. Once a DataNode is marked for decommissioning, it will not be selected as the target of replica placement, but it will continue to serve read requests. The NameNode starts to schedule replication of its blocks to other DataNodes. Once the NameNode detects that all blocks on the decommissioning DataNode are replicated, the node enters the decommissioned state. Then it can be safely removed from the cluster without jeopardizing any data availability.

Inter-Cluster Data Copy

When working with large datasets, copying data into and out of a HDFS cluster is daunting. HDFS provides a tool called DistCp for large inter/intra-cluster parallel copying. It is a MapReduce job; each of the map tasks copies a portion of the source data into the destination filesystem. The MapReduce framework automatically handles parallel task scheduling, error detection and recovery.

9.4 Practice at Yahoo!

Large HDFS clusters at Yahoo! include about 4000 nodes. A typical cluster node has two quad core Xeon processors running at 2.5 GHz, 4–12 directly attached SATA drives (holding two terabytes each), 24 Gbyte of RAM, and a 1-gigabit Ethernet connection. Seventy percent of the disk space is allocated to HDFS. The remainder is reserved for the operating system (Red Hat Linux), logs, and space to spill the output of map tasks (MapReduce intermediate data are not stored in HDFS).

Forty nodes in a single rack share an IP switch. The rack switches are connected to each of eight core switches. The core switches provide connectivity between racks and to out-of-cluster resources. For each cluster, the NameNode and the BackupNode hosts are specially provisioned with up to 64 GB RAM; application tasks are never assigned to those hosts. In total, a cluster of 4000 nodes has 11 PB (petabytes; 1000 terabytes) of storage available as blocks that are replicated three times yielding a net 3.7 PB of storage for user applications. Over the years that HDFS has been in use, the hosts selected as cluster nodes have benefited from improved technologies. New cluster nodes always have faster processors, bigger disks and larger RAM. Slower, smaller nodes are retired or relegated to clusters reserved for development and testing of Hadoop.

On an example large cluster (4000 nodes), there are about 65 million files and 80 million blocks. As each block typically is replicated three times, every data node hosts 60 000 block replicas. Each day, user applications will create two million new files on the cluster. The 40 000 nodes in Hadoop clusters at Yahoo! provide 40 PB of on-line data storage.

Becoming a key component of Yahoo!’s technology suite meant tackling technical problems that are the difference between being a research project and being the custodian of many petabytes of corporate data. Foremost are issues of robustness and durability of data. But also important are

economical performance, provisions for resource sharing among members of the user community, and ease of administration by the system operators.

Durability of Data

Replication of data three times is a robust guard against loss of data due to uncorrelated node failures. It is unlikely Yahoo! has ever lost a block in this way; for a large cluster, the probability of losing a block during one year is less than 0.005. The key understanding is that about 0.8 percent of nodes fail each month. (Even if the node is eventually recovered, no effort is taken to recover data it may have hosted.) So for the sample large cluster as described above, a node or two is lost each day. That same cluster will re-create the 60 000 block replicas hosted on a failed node in about two minutes: re-replication is fast because it is a parallel problem that scales with the size of the cluster. The probability of several nodes failing within two minutes such that all replicas of some block are lost is indeed small.

Correlated failure of nodes is a different threat. The most commonly observed fault in this regard is the failure of a rack or core switch. HDFS can tolerate losing a rack switch (each block has a replica on some other rack). Some failures of a core switch can effectively disconnect a slice of the cluster from multiple racks, in which case it is probable that some blocks will become unavailable. In either case, repairing the switch restores unavailable replicas to the cluster. Another kind of correlated failure is the accidental or deliberate loss of electrical power to the cluster. If the loss of power spans racks, it is likely that some blocks will become unavailable. But restoring power may not be a remedy because one-half to one percent of the nodes will not survive a full power-on restart. Statistically, and in practice, a large cluster will lose a handful of blocks during a power-on restart.

In addition to total failures of nodes, stored data can be corrupted or lost. The block scanner scans all blocks in a large cluster each fortnight and finds about 20 bad replicas in the process. Bad replicas are replaced as they are discovered.

Features for Sharing HDFS

As the use of HDFS has grown, the filesystem itself has had to introduce means to share the resource among a large number of diverse users. The first such feature was a permissions framework closely modeled on the Unix permissions scheme for file and directories. In this framework, files and directories have separate access permissions for the owner, for other members of the user group associated with the file or directory, and for all other users. The principle differences between Unix (POSIX) and HDFS are that ordinary files in HDFS have neither execute permissions nor sticky bits.

In the earlier version of HDFS, user identity was weak: you were who your host said you are. When accessing HDFS, the application client simply queries the local operating system for user identity and group membership. In the new framework, the application client must present to the name system credentials obtained from a trusted source. Different credential administrations are possible; the initial implementation uses Kerberos. The user application can use the same framework to confirm that the name system also has a trustworthy identity. And the name system also can demand credentials from each of the data nodes participating in the cluster.

The total space available for data storage is set by the number of data nodes and the storage provisioned for each node. Early experience with HDFS demonstrated a need for some means to enforce the resource allocation policy across user communities. Not only must fairness of sharing be enforced, but when a user application might involve thousands of hosts writing data, protection against applications inadvertently exhausting resources is also important. For HDFS, because the

system metadata are always in RAM, the size of the namespace (number of files and directories) is also a finite resource. To manage storage and namespace resources, each directory may be assigned a quota for the total space occupied by files in the sub-tree of the namespace beginning at that directory. A separate quota may also be set for the total number of files and directories in the sub-tree.

While the architecture of HDFS presumes most applications will stream large data sets as input, the MapReduce programming framework can have a tendency to generate many small output files (one from each reduce task) further stressing the namespace resource. As a convenience, a directory sub-tree can be collapsed into a single Hadoop Archive file. A HAR file is similar to a familiar tar, JAR, or Zip file, but filesystem operations can address the individual files within the archive, and a HAR file can be used transparently as the input to a MapReduce job.

Scaling and HDFS Federation

Scalability of the NameNode has been a key struggle [Shv10]. Because the NameNode keeps all the namespace and block locations in memory, the size of the NameNode heap limits the number of files and also the number of blocks addressable. This also limits the total cluster storage that can be supported by the NameNode. Users are encouraged to create larger files, but this has not happened since it would require changes in application behavior. Furthermore, we are seeing new classes of applications for HDFS that need to store a large number of small files. Quotas were added to manage the usage, and an archive tool has been provided, but these do not fundamentally address the scalability problem.

A new feature allows multiple independent namespaces (and NameNodes) to share the physical storage within a cluster. Namespaces use blocks grouped under a Block Pool. Block pools are analogous to logical units (LUNs) in a SAN storage system and a namespace with its pool of blocks is analogous to a filesystem volume.

This approach offers a number of advantages besides scalability: it can isolate namespaces of different applications improving the overall availability of the cluster. Block pool abstraction allows other services to use the block storage with perhaps a different namespace structure. We plan to explore other approaches to scaling such as storing only partial namespace in memory, and truly distributed implementation of the NameNode.

Applications prefer to continue using a single namespace. Namespaces can be mounted to create such a unified view. A client-side mount table provide an efficient way to do that, compared to a server-side mount table: it avoids an RPC to the central mount table and is also tolerant of its failure. The simplest approach is to have shared cluster-wide namespace; this can be achieved by giving the same client-side mount table to each client of the cluster. Client-side mount tables also allow applications to create a private namespace view. This is analogous to the per-process namespaces that are used to deal with remote execution in distributed systems [PPT⁺93, Rad94, RP93].

9.5 Lessons Learned

A very small team was able to build the Hadoop filesystem and make it stable and robust enough to use it in production. A large part of the success was due to the very simple architecture: replicated blocks, periodic block reports and central metadata server. Avoiding the full POSIX semantics also helped. Although keeping the entire metadata in memory limited the scalability of the namespace, it made the NameNode very simple: it avoids the complex locking of typical filesystems. The other reason for Hadoop's success was to quickly use the system for production at Yahoo!, as it was rapidly

and incrementally improved. The filesystem is very robust and the NameNode rarely fails; indeed most of the down time is due to software upgrades. Only recently have failover solutions (albeit manual) emerged

Many have been surprised by the choice of Java in building a scalable filesystem. While Java posed challenges for scaling the NameNode due to its object memory overhead and garbage collection, Java has been responsible to the robustness of the system; it has avoided corruption due to pointer or memory management bugs.

9.6 Acknowledgment

We thank Yahoo! for investing in Hadoop and continuing to make it available as open source; 80% of the HDFS and MapReduce code was developed at Yahoo! We thank all Hadoop committers and collaborators for their valuable contributions.

Jitsi

Emil Ivov

Jitsi is an application that allows people to make video and voice calls, share their desktops, and exchange files and messages. More importantly it allows people to do this over a number of different protocols, ranging from the standardized XMPP (Extensible Messaging and Presence Protocol) and SIP (Session Initiation Protocol) to proprietary ones like Yahoo! and Windows Live Messenger (MSN). It runs on Microsoft Windows, Apple Mac OS X, Linux, and FreeBSD. It is written mostly in Java but it also contains parts written in native code. In this chapter, we'll look at Jitsi's OSGi-based architecture, see how it implements and manages protocols, and look back on what we've learned from building it.¹

10.1 Designing Jitsi

The three most important constraints that we had to keep in mind when designing Jitsi (at the time called SIP Communicator) were multi-protocol support, cross-platform operation, and developer-friendliness.

From a developer's perspective, being multi-protocol comes down to having a common interface for all protocols. In other words, when a user sends a message, our graphical user interface needs to always call the same `sendMessage` method regardless of whether the currently selected protocol actually uses a method called `sendXmppMessage` or `sendSipMsg`.

The fact that most of our code is written in Java satisfies, to a large degree, our second constraint: cross-platform operation. Still, there are things that the Java Runtime Environment (JRE) does not support or does not do the way we'd like it to, such as capturing video from your webcam. Therefore, we need to use DirectShow on Windows, QTKit on Mac OS X, and Video for Linux 2 on Linux. Just as with protocols, the parts of the code that control video calls cannot be bothered with these details (they are complicated enough as it is).

Finally, being developer-friendly means that it should be easy for people to add new features. There are millions of people using VoIP today in thousands of different ways; various service providers and server vendors come up with different use cases and ideas about new features. We have to make sure that it is easy for them to use Jitsi the way they want. Someone who needs to add

¹To refer directly to the source as you read, download it from <http://jitsi.org/source>. If you are using Eclipse or NetBeans, you can go to <http://jitsi.org/eclipse> or <http://jitsi.org/netbeans> for instructions on how configure them.

something new should have to read and understand only those parts of the project they are modifying or extending. Similarly, one person's changes should have as little impact as possible on everyone else's work.

To sum up, we needed an environment where different parts of the code are relatively independent from each other. It had to be possible to easily replace some parts depending on the operating system; have others, like protocols, run in parallel and yet act the same; and it had to be possible to completely rewrite any one of those parts and have the rest of the code work without any changes. Finally, we wanted the ability to easily switch parts on and off, as well as the ability to download plugins over the Internet to our list.

We briefly considered writing our own framework, but soon dropped the idea. We were itching to start writing VoIP and IM code as soon as possible, and spending a couple of months on a plugin framework didn't seem that exciting. Someone suggested OSGi, and it seemed to be the perfect fit.

10.2 Jitsi and the OSGi Framework

People have written entire books about OSGi, so we're not going to go over everything the framework stands for. Instead we will only explain what it gives us and the way we use it in Jitsi.

Above everything else, OSGi is about modules. Features in OSGi applications are separated into bundles. An OSGi bundle is little more than a regular JAR file like the ones used to distribute Java libraries and applications. Jitsi is a collection of such bundles. There is one responsible for connecting to Windows Live Messenger, another one that does XMPP, yet another one that handles the GUI, and so on. All these bundles run together in an environment provided, in our case, by Apache Felix, an open source OSGi implementation.

All these modules need to work together. The GUI bundle needs to send messages via the protocol bundles, which in turn need to store them via the bundles handling message history. This is what OSGi services are for: they represent the part of a bundle that is visible to everyone else. An OSGi service is most often a group of Java interfaces that allow use of a specific functionality like logging, sending messages over the network, or retrieving the list of recent calls. The classes that actually implement the functionality are known as a service implementation. Most of them carry the name of the service interface they implement, with an "Impl" suffix at the end (e.g., `ConfigurationServiceImpl`). The OSGi framework allows developers to hide service implementations and make sure that they are never visible outside the bundle they are in. This way, other bundles can only use them through the service interfaces.

Most bundles also have activators. Activators are simple interfaces that define a `start` and a `stop` method. Every time Felix loads or removes a bundle in Jitsi, it calls these methods so that the bundle can prepare to run or shut down. When calling these methods Felix passes them a parameter called `BundleContext`. The `BundleContext` gives bundles a way to connect to the OSGi environment. This way they can discover whatever OSGi service they need to use, or register one themselves (Figure 10.1).

So let's see how this actually works. Imagine a service that persistently stores and retrieves properties. In Jitsi this is what we call the `ConfigurationService` and it looks like this:

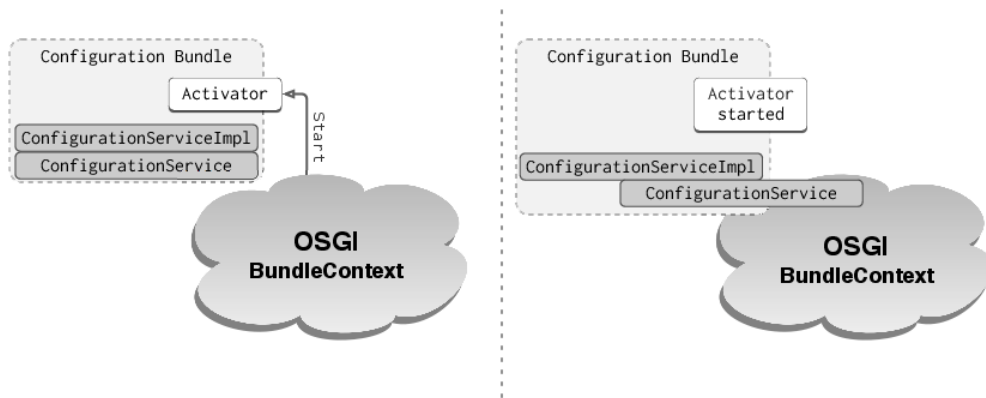


Figure 10.1: OSGi Bundle Activation

```
package net.java.sip.communicator.service.configuration;

public interface ConfigurationService
{
    public void setProperty(String propertyName, Object property);
    public Object getProperty(String propertyName);
}
```

A very simple implementation of the ConfigurationService looks like this:

```
package net.java.sip.communicator.impl.configuration;

import java.util.*;
import net.java.sip.communicator.service.configuration.*;

public class ConfigurationServiceImpl implements ConfigurationService
{
    private final Properties properties = new Properties();

    public Object getProperty(String name)
    {
        return properties.get(name);
    }

    public void setProperty(String name, Object value)
    {
        properties.setProperty(name, value.toString());
    }
}
```

Notice how the service is defined in the `net.java.sip.communicator.service` package, while the implementation is in `net.java.sip.communicator.impl`. All services and implementations in Jitsi are separated under these two packages. OSGi allows bundles to only make some packages

visible outside their own JAR, so the separation makes it easier for bundles to only *export* their service packages and keep their implementations hidden.

The last thing we need to do so that people can start using our implementation is to register it in the `BundleContext` and indicate that it provides an implementation of the `ConfigurationService`. Here's how this happens:

```
package net.java.sip.communicator.impl.configuration;

import org.osgi.framework.*;
import net.java.sip.communicator.service.configuration;

public class ConfigActivator implements BundleActivator
{
    public void start(BundleContext bc) throws Exception
    {
        bc.registerService(ConfigurationService.class.getName(), // service name
            new ConfigurationServiceImpl(), // service implementation
            null);
    }
}
```

Once the `ConfigurationServiceImpl` class is registered in the `BundleContext`, other bundles can start using it. Here's an example showing how some random bundle can use our configuration service:

```
package net.java.sip.communicator.plugin.randombundle;

import org.osgi.framework.*;
import net.java.sip.communicator.service.configuration.*;

public class RandomBundleActivator implements BundleActivator
{
    public void start(BundleContext bc) throws Exception
    {
        ServiceReference cRef = bc.getServiceReference(
            ConfigurationService.class.getName());
        configService = (ConfigurationService) bc.getService(cRef);

        // And that's all! We have a reference to the service implementation
        // and we are ready to start saving properties:
        configService.setProperty("propertyName", "propertyValue");
    }
}
```

Once again, notice the package. In `net.java.sip.communicator.plugin` we keep bundles that use services defined by others but that neither export nor implement any themselves. Configuration forms are a good example of such plugins: They are additions to the Jitsi user interface that allow users to configure certain aspects of the application. When users change preferences, configuration forms interact with the `ConfigurationService` or directly with the bundles responsible for a feature. However, none of the other bundles ever need to interact with them in any way (Figure 10.2).

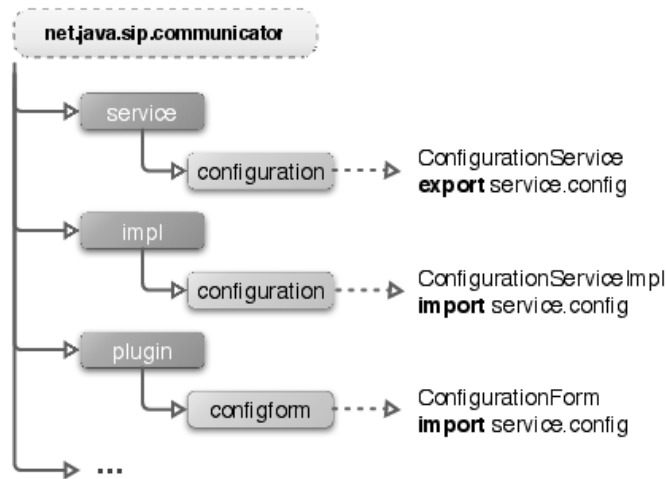


Figure 10.2: Service Structure

10.3 Building and Running a Bundle

Now that we've seen how to write the code in a bundle, it's time to talk about packaging. When running, all bundles need to indicate three different things to the OSGi environment: the Java packages they make available to others (i.e. exported packages), the ones that they would like to use from others (i.e. imported packages), and the name of their BundleActivator class. Bundles do this through the manifest of the JAR file that they will be deployed in.

For the ConfigurationService that we defined above, the manifest file could look like this:

```

Bundle-Activator: net.java.sip.communicator.impl.configuration.ConfigActivator
Bundle-Name: Configuration Service Implementation
Bundle-Description: A bundle that offers configuration utilities
Bundle-Vendor: jitsi.org
Bundle-Version: 0.0.1
System-Bundle: yes
Import-Package: org.osgi.framework,
Export-Package: net.java.sip.communicator.service.configuration
  
```

After creating the JAR manifest, we are ready to create the bundle itself. In Jitsi we use Apache Ant to handle all build-related tasks. In order to add a bundle to the Jitsi build process, you need to edit the build.xml file in the root directory of the project. Bundle JARs are created at the bottom of the build.xml file, with bundle-xxx targets. In order to build our configuration service we need the following:

```

<target name="bundle-configuration">
  <jar destfile="${bundles.dest}/configuration.jar" manifest=
    "${src}/net/java/sip/communicator/impl/configuration/conf.manifest.mf" >

    <zipfileset dir="${dest}/net/java/sip/communicator/service/configuration"
      prefix="net/java/sip/communicator/service/configuration"/>
  </jar>
</target>
  
```

```

    <zipfileset dir="${dest}/net/java/sip/communicator/impl/configuration"
      prefix="net/java/sip/communicator/impl/configuration" />
  </jar>
</target>

```

As you can see, the Ant target simply creates a JAR file using our configuration manifest, and adds to it the configuration packages from the service and impl hierarchies. Now the only thing that we need to do is to make Felix load it.

We already mentioned that Jitsi is merely a collection of OSGi bundles. When a user executes the application, they actually start Felix with a list of bundles that it needs to load. You can find that list in our lib directory, inside a file called `felix.client.run.properties`. Felix starts bundles in the order defined by start levels: All those within a particular level are guaranteed to complete before bundles in subsequent levels start loading. Although you can't see this in the example code above, our configuration service stores properties in files so it needs to use our *FileAccessService*, shipped within the `fileaccess.jar` file. We'll therefore make sure that the *ConfigurationService* starts after the *FileAccessService*:

```

...
felix.auto.start.30= \
  reference:file:sc-bundles/fileaccess.jar

felix.auto.start.40= \
  reference:file:sc-bundles/configuration.jar \
  reference:file:sc-bundles/jmdnslib.jar \
  reference:file:sc-bundles/provdisc.jar \
...

```

If you look at the `felix.client.run.properties` file, you'll see a list of packages at the beginning:

```

org.osgi.framework.system.packages.extra= \
  apple.awt; \
  com.apple.cocoa.application; \
  com.apple.cocoa.foundation; \
  com.apple.eawt; \
...

```

The list tells Felix what packages it needs to make available to bundles from the system classpath. This means that packages that are on this list can be imported by bundles (i.e. added to their *Import-Package* manifest header) without any being exported by any other bundle. The list mostly contains packages that come from OS-specific JRE parts, and Jitsi developers rarely need to add new ones to it; in most cases packages are made available by bundles.

10.4 Protocol Provider Service

The *ProtocolProviderService* in Jitsi defines the way all protocol implementations behave. It is the interface that other bundles (like the user interface) use when they need to send and receive messages, make calls, and share files through the networks that Jitsi connects to.

The protocol service interfaces can all be found under the `net.java.sip.communicator.service.protocol` package. There are multiple implementations of the service, one per supported protocol, and all are stored in `net.java.sip.communicator.impl.protocol.protocol_name`.

Let's start with the `service.protocol` directory. The most prominent piece is the *ProtocolProviderService* interface. Whenever someone needs to perform a protocol-related task, they have to look up an implementation of that service in the `BundleContext`. The service and its implementations allow Jitsi to connect to any of the supported networks, to retrieve the connection status and details, and most importantly to obtain references to the classes that implement the actual communications tasks like chatting and making calls.

Operation Sets

As we mentioned earlier, the `ProtocolProviderService` needs to leverage the various communication protocols and their differences. While this is particularly simple for features that all protocols share, like sending a message, things get trickier for tasks that only some protocols support. Sometimes these differences come from the service itself: For example, most of the SIP services out there do not support server-stored contact lists, while this is a relatively well-supported feature with all other protocols. MSN and AIM are another good example: at one time neither of them offered the ability to send messages to offline users, while everyone else did. (This has since changed.)

The bottom line is our `ProtocolProviderService` needs to have a way of handling these differences so that other bundles, like the GUI, act accordingly; there's no point in adding a call button to an AIM contact if there's no way to actually make a call.

`OperationSets` to the rescue (Figure 10.3). Unsurprisingly, they are sets of operations, and provide the interface that Jitsi bundles use to control the protocol implementations. The methods that you find in an operation set interface are all related to a particular feature. *OperationSetBasicInstantMessaging*, for instance, contains methods for creating and sending instant messages, and registering listeners that allow Jitsi to retrieve messages it receives. Another example, *OperationSetPresence*, has methods for querying the status of the contacts on your list and setting a status for yourself. So when the GUI updates the status it shows for a contact, or sends a message to a contact, it is first able to ask the corresponding provider whether they support presence and messaging. The methods that `ProtocolProviderService` defines for that purpose are:

```
public Map<String, OperationSet> getSupportedOperationSets();
public <T extends OperationSet> T getOperationSet(Class<T> opsetClass);
```

`OperationSets` have to be designed so that it is unlikely that a new protocol we add has support for only some of the operations defined in an `OperationSet`. For example, some protocols do not support server-stored contact lists even though they allow users to query each other's status. Therefore, rather than combining the presence management and buddy list retrieval features in `OperationSetPresence`, we also defined an `OperationSetPersistentPresence` which is only used with protocols that can store contacts online. On the other hand, we have yet to come across a protocol that only allows sending messages without receiving any, which is why things like sending and receiving messages can be safely combined.

Accounts, Factories and Provider Instances

An important characteristic of the `ProtocolProviderService` is that one instance corresponds to one protocol account. Therefore, at any given time you have as many service implementations in the

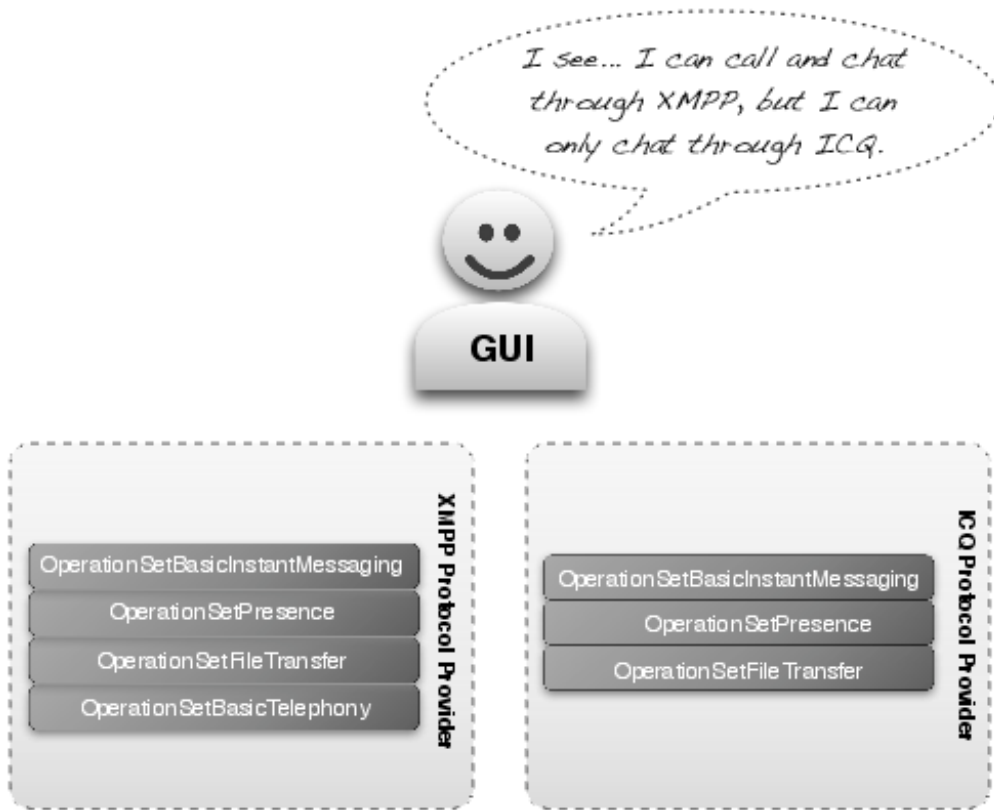


Figure 10.3: Operation Sets

BundleContext as you have accounts registered by the user.

At this point you may be wondering who creates and registers the protocol providers. There are two different entities involved. First, there is `ProtocolProviderFactory`. This is the service that allows other bundles to instantiate providers and then registers them as services. There is one factory per protocol and every factory is responsible for creating providers for that particular protocol. Factory implementations are stored with the rest of the protocol internals. For SIP, for example we have `net.java.sip.communicator.impl.protocol.sip.ProtocolProviderFactorySipImpl`.

The second entity involved in account creation is the protocol wizard. Unlike factories, wizards are separated from the rest of the protocol implementation because they involve the graphical user interface. The wizard that allows users to create SIP accounts, for example, can be found in `net.java.sip.communicator.plugin.sipaccregwizz`.

10.5 Media Service

When working with real-time communication over IP, there is one important thing to understand: protocols like SIP and XMPP, while recognized by many as the most common VoIP protocols, are not the ones that actually move voice and video over the Internet. This task is handled by the Real-time

Transport Protocol (RTP). SIP and XMPP are only responsible for preparing everything that RTP needs, like determining the address where RTP packets need to be sent and negotiating the format that audio and video need to be encoded in (i.e. codec), etc. They also take care of things like locating users, maintaining their presence, making the phones ring, and many others. This is why protocols like SIP and XMPP are often referred to as signalling protocols.

What does this mean in the context of Jitsi? Well, first of all it means that you are not going to find any code manipulating audio or video flows in either the *sip* or *jabber* jitsi packages. This kind of code lives in our MediaService. The MediaService and its implementation are located in `net.java.sip.communicator.service.neomedia` and `net.java.sip.communicator.impl.neomedia`.

Why “neomedia”?

The “neo” in the neomedia package name indicates that it replaces a similar package that we used originally and that we then had to completely rewrite. This is actually how we came up with one of our rules of thumb: It is hardly ever worth it to spend a lot of time designing an application to be 100% future-proof. There is simply no way of taking everything into account, so you are bound to have to make changes later anyway. Besides, it is quite likely that a painstaking design phase will introduce complexities that you will never need because the scenarios you prepared for never happen.

In addition to the MediaService itself, there are two other interfaces that are particularly important: MediaDevice and MediaStream.

Capture, Streaming, and Playback

MediaDevices represent the capture and playback devices that we use during a call (Figure 10.4). Your microphone and speakers, your headset and your webcam are all examples of such MediaDevices, but they are not the only ones. Desktop streaming and sharing calls in Jitsi capture video from your desktop, while a conference call uses an AudioMixer device in order to mix the audio we receive from the active participants. In all cases, MediaDevices represent only a single MediaType. That is, they can only be either audio or video but never both. This means that if, for example, you have a webcam with an integrated microphone, Jitsi sees it as two devices: one that can only capture video, and another one that can only capture sound.

Devices alone, however, are not enough to make a phone or a video call. In addition to playing and capturing media, one has to also be able to send it over the network. This is where MediaStreams come in. A MediaStream interface is what connects a MediaDevice to your interlocutor. It represents incoming and outgoing packets that you exchange with them within a call.

Just as with devices, one stream can be responsible for only one MediaType. This means that in the case of an audio/video call Jitsi has to create two separate media streams and then connect each to the corresponding audio or video MediaDevice.

Codecs

Another important concept in media streaming is that of MediaFormats, also known as codecs. By default most operating systems let you capture audio in 48KHz PCM or something similar. This is what we often refer to as “raw audio” and it’s the kind of audio you get in WAV files: great quality

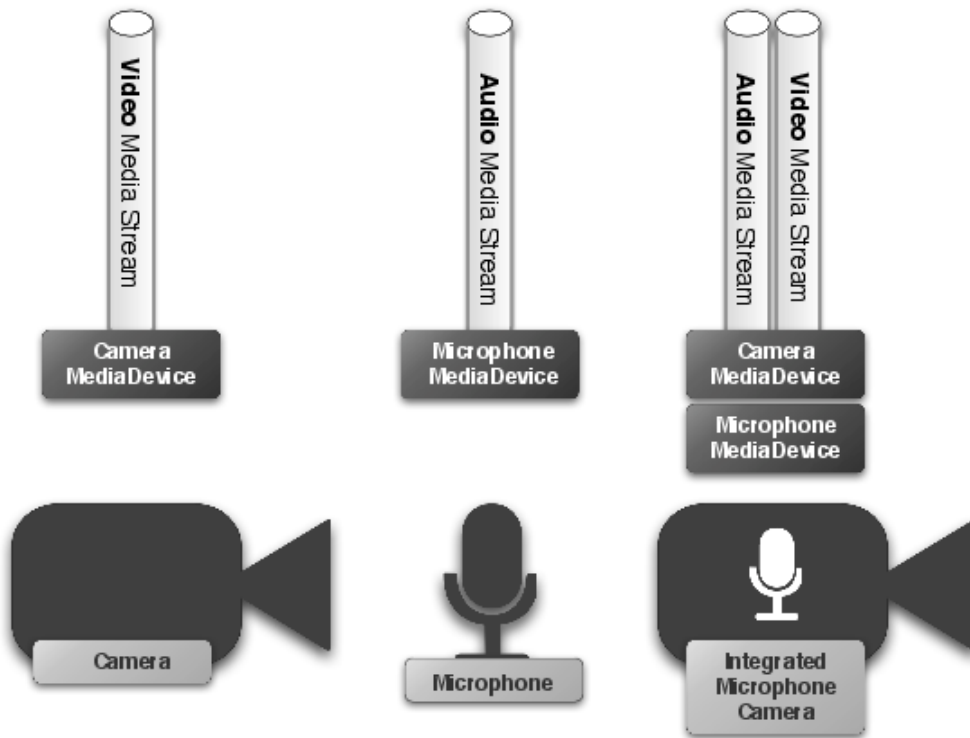


Figure 10.4: Media Streams For Different Devices

and enormous size. It is quite impractical to try and transport audio over the Internet in the PCM format.

This is what codecs are for: they let you present and transport audio or video in a variety of different ways. Some audio codecs like iLBC, 8KHz Speex, or G.729, have low bandwidth requirements but sound somewhat muffled. Others like wideband Speex and G.722 give you great audio quality but also require more bandwidth. There are codecs that try to deliver good quality while keeping bandwidth requirements at a reasonable level. H.264, the popular video codec, is a good example of that. The trade-off here is the amount of calculation required during conversion. If you use Jitsi for an H.264 video call you see a good quality image and your bandwidth requirements are quite reasonable, but your CPU runs at maximum.

All this is an oversimplification, but the idea is that codec choice is all about compromises. You either sacrifice bandwidth, quality, CPU intensity, or some combination of those. People working with VoIP rarely need to know more about codecs.

Connecting with the Protocol Providers

Protocols in Jitsi that currently have audio/video support all use our MediaServices exactly the same way. First they ask the MediaService about the devices that are available on the system:

```
public List<MediaDevice> getDevices(MediaType mediaType, MediaUseCase useCase);
```

The `MediaType` indicates whether we are interested in audio or video devices. The `MediaUseCase` parameter is currently only considered in the case of video devices. It tells the media service whether we'd like to get devices that could be used in a regular call (`MediaUseCase.CALL`), in which case it returns a list of available webcams, or a desktop sharing session (`MediaUseCase.DESKTOP`), in which case it returns references to the user desktops.

The next step is to obtain the list of formats that are available for a specific device. We do this through the `MediaDevice.getSupportedFormats` method:

```
public List<MediaFormat> getSupportedFormats();
```

Once it has this list, the protocol implementation sends it to the remote party, which responds with a subset of them to indicate which ones it supports. This exchange is also known as the Offer/Answer Model and it often uses the Session Description Protocol or some form of it.

After exchanging formats and some port numbers and IP addresses, VoIP protocols create, configure and start the `MediaStreams`. Roughly speaking, this initialization is along the following lines:

```
// first create a stream connector telling the media service what sockets
// to use when transport media with RTP and flow control and statistics
// messages with RTCP
StreamConnector connector = new DefaultStreamConnector(rtpSocket, rtcpSocket);
MediaStream stream = mediaService.createMediaStream(connector, device, control);

// A MediaStreamTarget indicates the address and ports where our
// interlocutor is expecting media. Different VoIP protocols have their
// own ways of exchanging this information
stream.setTarget(target);

// The MediaDirection parameter tells the stream whether it is going to be
// incoming, outgoing or both
stream.setDirection(direction);

// Then we set the stream format. We use the one that came
// first in the list returned in the session negotiation answer.
stream.setFormat(format);

// Finally, we are ready to actually start grabbing media from our
// media device and streaming it over the Internet
stream.start();
```

Now you can wave at your webcam, grab the mic and say, “Hello world!”

10.6 UI Service

So far we have covered parts of Jitsi that deal with protocols, sending and receiving messages and making calls. Above all, however, Jitsi is an application used by actual people and as such, one of its most important aspects is its user interface. Most of the time the user interface uses the services that all the other bundles in Jitsi expose. There are some cases, however, where things happen the other way around.

Plugins are the first example that comes to mind. Plugins in Jitsi often need to be able to interact with the user. This means they have to open, close, move or add components to existing windows and panels in the user interface. This is where our `UIService` comes into play. It allows for basic control over the main window in Jitsi and this is how our icons in the Mac OS X dock and the Windows notification area let users control the application.

In addition to simply playing with the contact list, plugins can also extend it. The plugin that implements support for chat encryption (OTR) in Jitsi is a good example for this. Our OTR bundle needs to register several GUI components in various parts of the user interface. It adds a padlock button in the chat window and a sub-section in the right-click menu of all contacts.

The good news is that it can do all this with just a few method calls. The OSGi activator for the OTR bundle, `OtrActivator`, contains the following lines:

```
Hashtable<String, String> filter = new Hashtable<String, String>();

// Register the right-click menu item.
filter(Container.CONTAINER_ID,
        Container.CONTAINER_CONTACT_RIGHT_BUTTON_MENU.getID());

bundleContext.registerService(PluginComponent.class.getName(),
    new OtrMetaContactMenu(Container.CONTAINER_CONTACT_RIGHT_BUTTON_MENU),
    filter);

// Register the chat window menu bar item.
filter.put(Container.CONTAINER_ID,
            Container.CONTAINER_CHAT_MENU_BAR.getID());

bundleContext.registerService(PluginComponent.class.getName(),
    new OtrMetaContactMenu(Container.CONTAINER_CHAT_MENU_BAR),
    filter);
```

As you can see, adding components to our graphical user interface simply comes down to registering OSGi services. On the other side of the fence, our `UIService` implementation is looking for implementations of its `PluginComponent` interface. Whenever it detects that a new implementation has been registered, it obtains a reference to it and adds it to the container indicated in the OSGi service filter.

Here's how this happens in the case of the right-click menu item. Within the UI bundle, the class that represents the right click menu, `MetaContactRightButtonMenu`, contains the following lines:

```
// Search for plugin components registered through the OSGi bundle context.
ServiceReference[] serRefs = null;

String osgiFilter = "("
    + Container.CONTAINER_ID
    + "=" + Container.CONTAINER_CONTACT_RIGHT_BUTTON_MENU.getID() + ")";

serRefs = GuiActivator.bundleContext.getServiceReferences(
    PluginComponent.class.getName(),
    osgiFilter);
// Go through all the plugins we found and add them to the menu.
for (int i = 0; i < serRefs.length; i ++)
```



```

{
    PluginComponent component = (PluginComponent) GuiActivator
        .bundleContext.getService(serRefs[i]);

    component.setCurrentContact(metaContact);

    if (component.getComponent() == null)
        continue;

    this.add((Component)component.getComponent());
}

```

And that's all there is to it. Most of the windows that you see within Jitsi do exactly the same thing: They look through the bundle context for services implementing the `PluginComponent` interface that have a filter indicating that they want to be added to the corresponding container. Plugins are like hitch-hikers holding up signs with the names of their destinations, making Jitsi windows the drivers who pick them up.

10.7 Lessons Learned

When we started work on SIP Communicator, one of the most common criticisms or questions we heard was: “Why are you using Java? Don't you know it's slow? You'd never be able to get decent quality for audio/video calls!” The “Java is slow” myth has even been repeated by potential users as a reason they stick with Skype instead of trying Jitsi. But the first lesson we've learned from our work on the project is that efficiency is no more of a concern with Java than it would have been with C++ or other native alternatives.

We won't pretend that the decision to choose Java was the result of rigorous analysis of all possible options. We simply wanted an easy way to build something that ran on Windows and Linux, and Java and the Java Media Framework seemed to offer one relatively easy way of doing so.

Throughout the years we haven't had many reasons to regret this decision. Quite the contrary: even though it doesn't make it completely transparent, Java does help portability and 90% of the code in SIP Communicator doesn't change from one OS to the next. This includes all the protocol stack implementations (e.g., SIP, XMPP, RTP, etc.) that are complex enough as they are. Not having to worry about OS specifics in such parts of the code has proven immensely useful.

Furthermore, Java's popularity has turned out to be very important when building our community. Contributors are a scarce resource as it is. People need to like the nature of the application, they need to find time and motivation—all of this is hard to muster. Not requiring them to learn a new language is, therefore, an advantage.

Contrary to most expectations, Java's presumed lack of speed has rarely been a reason to go native. Most of the time decisions to use native languages were driven by OS integration and how much access Java was giving us to OS-specific utilities. Below we discuss the three most important areas where Java fell short.

Java Sound vs. PortAudio

Java Sound is Java's default API for capturing and playing audio. It is part of the runtime environment and therefore runs on all the platforms the Java Virtual Machine comes for. During its first years

as SIP Communicator, Jitsi used JavaSound exclusively and this presented us with quite a few inconveniences.

First of all, the API did not give us the option of choosing which audio device to use. This is a big problem. When using their computer for audio and video calls, users often use advanced USB headsets or other audio devices to get the best possible quality. When multiple devices are present on a computer, JavaSound routes all audio through whichever device the OS considers default, and this is not good enough in many cases. Many users like to keep all other applications running on their default sound card so that, for example, they could keep hearing music through their speakers. What's even more important is that in many cases it is best for SIP Communicator to send audio notifications to one device and the actual call audio to another, allowing a user to hear an incoming call alert on their speakers even if they are not in front of the computer and then, after picking up the call, to start using a headset.

None of this is possible with Java Sound. What's more, the Linux implementation uses OSS which is deprecated on most of today's Linux distributions.

We decided to use an alternative audio system. We didn't want to compromise our multi-platform nature and, if possible, we wanted to avoid having to handle it all by ourselves. This is where PortAudio² came in extremely handy.

When Java doesn't let you do something itself, cross-platform open source projects are the next best thing. Switching to PortAudio has allowed us to implement support for fine-grained configurable audio rendering and capture just as we described it above. It also runs on Windows, Linux, Mac OS X, FreeBSD and others that we haven't had the time to provide packages for.

Video Capture and Rendering

Video is just as important to us as audio. However, this didn't seem to be the case for the creators of Java, because there is no default API in the JRE that allows capturing or rendering video. For a while the Java Media Framework seemed to be destined to become such an API until Sun stopped maintaining it.

Naturally we started looking for a PortAudio-style video alternative, but this time we weren't so lucky. At first we decided to go with the LTI-CIVIL framework from Ken Larson³. This is a wonderful project and we used it for quite a while⁴. However it turned out to be suboptimal when used in a real-time communications context.

So we came to the conclusion that the only way to provide impeccable video communication for Jitsi would be for us to implement native grabbers and renderers all by ourselves. This was not an easy decision since it implied adding a lot of complexity and a substantial maintenance load to the project but we simply had no choice: we really wanted to have quality video calls. And now we do!

Our native grabbers and renderers directly use Video4Linux 2, QTKit and DirectShow/Direct3D on Linux, Mac OS X, and Windows respectively.

Video Encoding and Decoding

SIP Communicator, and hence Jitsi, supported video calls from its first days. That's because the Java Media Framework allowed encoding video using the H.263 codec and a 176x144 (CIF) format.

²<http://portaudio.com/>

³<http://lti-civil.org/>

⁴Actually we still have it as a non-default option.

Those of you who know what H.263 CIF looks like are probably smiling right now; few of us would use a video chat application today if that's all it had to offer.

In order to offer decent quality we've had to use other libraries like FFmpeg. Video encoding is actually one of the few places where Java shows its limits performance-wise. So do other languages, as evidenced by the fact that FFmpeg developers actually use Assembler in a number of places in order to handle video in the most efficient way possible.

Others

There are a number of other places where we've decided that we needed to go native for better results. Systray notifications with Growl on Mac OS X and libnotify on Linux are one such example. Others include querying contact databases from Microsoft Outlook and Apple Address Book, determining source IP address depending on a destination, using existing codec implementations for Speex and G.722, capturing desktop screenshots, and translating chars into key codes.

The important thing is that whenever we needed to choose a native solution, we could, and we did. This brings us to our point: Ever since we've started Jitsi we've fixed, added, or even entirely rewritten various parts of it because we wanted them to look, feel or perform better. However, we've never ever regretted any of the things we didn't get right the first time. When in doubt, we simply picked one of the available options and went with it. We could have waited until we knew better what we were doing, but if we had, there would be no Jitsi today.

10.8 Acknowledgments

Many thanks to Yana Stamcheva for creating all the diagrams in this chapter.

LLVM

Chris Lattner

This chapter discusses some of the design decisions that shaped LLVM¹, an umbrella project that hosts and develops a set of close-knit low-level toolchain components (e.g., assemblers, compilers, debuggers, etc.), which are designed to be compatible with existing tools typically used on Unix systems. The name “LLVM” was once an acronym, but is now just a brand for the umbrella project. While LLVM provides some unique capabilities, and is known for some of its great tools (e.g., the Clang compiler², a C/C++/Objective-C compiler which provides a number of benefits over the GCC compiler), the main thing that sets LLVM apart from other compilers is its internal architecture.

From its beginning in December 2000, LLVM was designed as a set of reusable libraries with well-defined interfaces [LA04]. At the time, open source programming language implementations were designed as special-purpose tools which usually had monolithic executables. For example, it was very difficult to reuse the parser from a static compiler (e.g., GCC) for doing static analysis or refactoring. While scripting languages often provided a way to embed their runtime and interpreter into larger applications, this runtime was a single monolithic lump of code that was included or excluded. There was no way to reuse pieces, and very little sharing across language implementation projects.

Beyond the composition of the compiler itself, the communities surrounding popular language implementations were usually strongly polarized: an implementation usually provided *either* a traditional static compiler like GCC, Free Pascal, and FreeBASIC, *or* it provided a runtime compiler in the form of an interpreter or Just-In-Time (JIT) compiler. It was very uncommon to see language implementation that supported both, and if they did, there was usually very little sharing of code.

Over the last ten years, LLVM has substantially altered this landscape. LLVM is now used as a common infrastructure to implement a broad variety of statically and runtime compiled languages (e.g., the family of languages supported by GCC, Java, .NET, Python, Ruby, Scheme, Haskell, D, as well as countless lesser known languages). It has also replaced a broad variety of special purpose compilers, such as the runtime specialization engine in Apple’s OpenGL stack and the image processing library in Adobe’s After Effects product. Finally LLVM has also been used to create a broad variety of new products, perhaps the best known of which is the OpenCL GPU programming language and runtime.

¹<http://llvm.org>

²<http://clang.llvm.org>

11.1 A Quick Introduction to Classical Compiler Design

The most popular design for a traditional static compiler (like most C compilers) is the three phase design whose major components are the front end, the optimizer and the back end (Figure 11.1). The front end parses source code, checking it for errors, and builds a language-specific Abstract Syntax Tree (AST) to represent the input code. The AST is optionally converted to a new representation for optimization, and the optimizer and back end are run on the code.

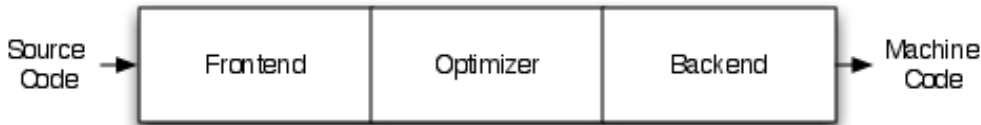


Figure 11.1: Three Major Components of a Three-Phase Compiler

The optimizer is responsible for doing a broad variety of transformations to try to improve the code's running time, such as eliminating redundant computations, and is usually more or less independent of language and target. The back end (also known as the code generator) then maps the code onto the target instruction set. In addition to making *correct* code, it is responsible for generating *good* code that takes advantage of unusual features of the supported architecture. Common parts of a compiler back end include instruction selection, register allocation, and instruction scheduling.

This model applies equally well to interpreters and JIT compilers. The Java Virtual Machine (JVM) is also an implementation of this model, which uses Java bytecode as the interface between the front end and optimizer.

Implications of this Design

The most important win of this classical design comes when a compiler decides to support multiple source languages or target architectures. If the compiler uses a common code representation in its optimizer, then a front end can be written for any language that can compile to it, and a back end can be written for any target that can compile from it, as shown in Figure 11.2.

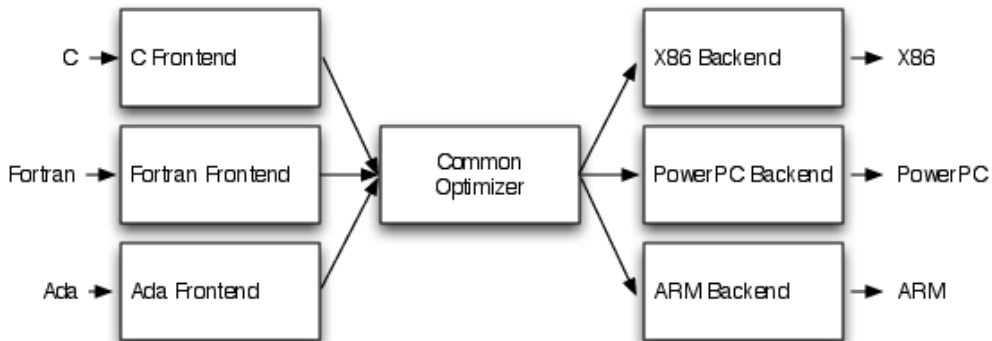


Figure 11.2: Retargetability

With this design, porting the compiler to support a new source language (e.g., Algol or BASIC) requires implementing a new front end, but the existing optimizer and back end can be reused. If these parts weren't separated, implementing a new source language would require starting over from scratch, so supporting N targets and M source languages would need $N \cdot M$ compilers.

Another advantage of the three-phase design (which follows directly from retargetability) is that the compiler serves a broader set of programmers than it would if it only supported one source language and one target. For an open source project, this means that there is a larger community of potential contributors to draw from, which naturally leads to more enhancements and improvements to the compiler. This is the reason why open source compilers that serve many communities (like GCC) tend to generate better optimized machine code than narrower compilers like FreePASCAL. This isn't the case for proprietary compilers, whose quality is directly related to the project's budget. For example, the Intel ICC Compiler is widely known for the quality of code it generates, even though it serves a narrow audience.

A final major win of the three-phase design is that the skills required to implement a front end are different than those required for the optimizer and back end. Separating these makes it easier for a "front-end person" to enhance and maintain their part of the compiler. While this is a social issue, not a technical one, it matters a lot in practice, particularly for open source projects that want to reduce the barrier to contributing as much as possible.

11.2 Existing Language Implementations

While the benefits of a three-phase design are compelling and well-documented in compiler textbooks, in practice it is almost never fully realized. Looking across open source language implementations (back when LLVM was started), you'd find that the implementations of Perl, Python, Ruby and Java share no code. Further, projects like the Glasgow Haskell Compiler (GHC) and FreeBASIC are retargetable to multiple different CPUs, but their implementations are very specific to the one source language they support. There is also a broad variety of special purpose compiler technology deployed to implement JIT compilers for image processing, regular expressions, graphics card drivers, and other subdomains that require CPU intensive work.

That said, there are three major success stories for this model, the first of which are the Java and .NET virtual machines. These systems provide a JIT compiler, runtime support, and a very well defined bytecode format. This means that any language that can compile to the bytecode format (and there are dozens of them³) can take advantage of the effort put into the optimizer and JIT as well as the runtime. The tradeoff is that these implementations provide little flexibility in the choice of runtime: they both effectively force JIT compilation, garbage collection, and the use of a very particular object model. This leads to suboptimal performance when compiling languages that don't match this model closely, such as C (e.g., with the LLJVM project).

A second success story is perhaps the most unfortunate, but also most popular way to reuse compiler technology: translate the input source to C code (or some other language) and send it through existing C compilers. This allows reuse of the optimizer and code generator, gives good flexibility, control over the runtime, and is really easy for front-end implementers to understand, implement, and maintain. Unfortunately, doing this prevents efficient implementation of exception handling, provides a poor debugging experience, slows down compilation, and can be problematic for languages that require guaranteed tail calls (or other features not supported by C).

³http://en.wikipedia.org/wiki/List_of_JVM_languages

A final successful implementation of this model is GCC⁴. GCC supports many front ends and back ends, and has an active and broad community of contributors. GCC has a long history of being a C compiler that supports multiple targets with hacky support for a few other languages bolted onto it. As the years go by, the GCC community is slowly evolving a cleaner design. As of GCC 4.4, it has a new representation for the optimizer (known as “GIMPLE Tuples”) which is closer to being separate from the front-end representation than before. Also, its Fortran and Ada front ends use a clean AST.

While very successful, these three approaches have strong limitations to what they can be used for, because they are designed as monolithic applications. As one example, it is not realistically possible to embed GCC into other applications, to use GCC as a runtime/JIT compiler, or extract and reuse pieces of GCC without pulling in most of the compiler. People who have wanted to use GCC’s C++ front end for documentation generation, code indexing, refactoring, and static analysis tools have had to use GCC as a monolithic application that emits interesting information as XML, or write plugins to inject foreign code into the GCC process.

There are multiple reasons why pieces of GCC cannot be reused as libraries, including rampant use of global variables, weakly enforced invariants, poorly-designed data structures, sprawling code base, and the use of macros that prevent the codebase from being compiled to support more than one front-end/target pair at a time. The hardest problems to fix, though, are the inherent architectural problems that stem from its early design and age. Specifically, GCC suffers from layering problems and leaky abstractions: the back end walks front-end ASTs to generate debug info, the front ends generate back-end data structures, and the entire compiler depends on global data structures set up by the command line interface.

11.3 LLVM’s Code Representation: LLVM IR

With the historical background and context out of the way, let’s dive into LLVM: The most important aspect of its design is the LLVM Intermediate Representation (IR), which is the form it uses to represent code in the compiler. LLVM IR is designed to host mid-level analyses and transformations that you find in the optimizer section of a compiler. It was designed with many specific goals in mind, including supporting lightweight runtime optimizations, cross-function/interprocedural optimizations, whole program analysis, and aggressive restructuring transformations, etc. The most important aspect of it, though, is that it is itself defined as a first class language with well-defined semantics. To make this concrete, here is a simple example of a .ll file:

```
define i32 @add1(i32 %a, i32 %b) {
entry:
    %tmp1 = add i32 %a, %b
    ret i32 %tmp1
}

define i32 @add2(i32 %a, i32 %b) {
entry:
    %tmp1 = icmp eq i32 %a, 0
    br i1 %tmp1, label %done, label %recurse

recurse:
```

⁴A backronym that now stands for “GNU Compiler Collection”.


```

    %tmp2 = sub i32 %a, 1
    %tmp3 = add i32 %b, 1
    %tmp4 = call i32 @add2(i32 %tmp2, i32 %tmp3)
    ret i32 %tmp4

done:
    ret i32 %b
}

```

This LLVM IR corresponds to this C code, which provides two different ways to add integers:

```

unsigned add1(unsigned a, unsigned b) {
    return a+b;
}

// Perhaps not the most efficient way to add two numbers.
unsigned add2(unsigned a, unsigned b) {
    if (a == 0) return b;
    return add2(a-1, b+1);
}

```

As you can see from this example, LLVM IR is a low-level RISC-like virtual instruction set. Like a real RISC instruction set, it supports linear sequences of simple instructions like add, subtract, compare, and branch. These instructions are in three address form, which means that they take some number of inputs and produce a result in a different register.⁵ LLVM IR supports labels and generally looks like a weird form of assembly language.

Unlike most RISC instruction sets, LLVM is strongly typed with a simple type system (e.g., i32 is a 32-bit integer, i32** is a pointer to pointer to 32-bit integer) and some details of the machine are abstracted away. For example, the calling convention is abstracted through call and ret instructions and explicit arguments. Another significant difference from machine code is that the LLVM IR doesn't use a fixed set of named registers, it uses an infinite set of temporaries named with a % character.

Beyond being implemented as a language, LLVM IR is actually defined in three isomorphic forms: the textual format above, an in-memory data structure inspected and modified by optimizations themselves, and an efficient and dense on-disk binary “bitcode” format. The LLVM Project also provides tools to convert the on-disk format from text to binary: llvm-as assembles the textual .ll file into a .bc file containing the bitcode goop and llvm-dis turns a .bc file into a .ll file.

The intermediate representation of a compiler is interesting because it can be a “perfect world” for the compiler optimizer: unlike the front end and back end of the compiler, the optimizer isn't constrained by either a specific source language or a specific target machine. On the other hand, it has to serve both well: it has to be designed to be easy for a front end to generate and be expressive enough to allow important optimizations to be performed for real targets.

Writing an LLVM IR Optimization

To give some intuition for how optimizations work, it is useful to walk through some examples. There are lots of different kinds of compiler optimizations, so it is hard to provide a recipe for how to solve an arbitrary problem. That said, most optimizations follow a simple three-part structure:

⁵This is in contrast to a two-address instruction set, like X86, which destructively updates an input register, or one-address machines which take one explicit operand and operate on an accumulator or the top of the stack on a stack machine.

- Look for a pattern to be transformed.
- Verify that the transformation is safe/correct for the matched instance.
- Do the transformation, updating the code.

The most trivial optimization is pattern matching on arithmetic identities, such as: for any integer X , $X-X$ is 0, $X-0$ is X , $(X*2)-X$ is X . The first question is what these look like in LLVM IR. Some examples are:

```
...
%example1 = sub i32 %a, %a
...
%example2 = sub i32 %b, 0
...
%tmp = mul i32 %c, 2
%example3 = sub i32 %tmp, %c
...
```

For these sorts of “peephole” transformations, LLVM provides an instruction simplification interface that is used as utilities by various other higher level transformations. These particular transformations are in the `SimplifySubInst` function and look like this:

```
// X - 0 -> X
if (match(Op1, m_Zero()))
    return Op0;

// X - X -> 0
if (Op0 == Op1)
    return Constant::getNullValue(Op0-&gtgetType());

// (X*2) - X -> X
if (match(Op0, m_Mul(m_Specific(Op1), m_ConstantInt<2>())))
    return Op1;

...

return 0; // Nothing matched, return null to indicate no transformation.
```

In this code, `Op0` and `Op1` are bound to the left and right operands of an integer subtract instruction (importantly, these identities don’t necessarily hold for IEEE floating point!). LLVM is implemented in C++, which isn’t well known for its pattern matching capabilities (compared to functional languages like Objective Caml), but it does offer a very general template system that allows us to implement something similar. The `match` function and the `m_` functions allow us to perform declarative pattern matching operations on LLVM IR code. For example, the `m_Specific` predicate only matches if the left hand side of the multiplication is the same as `Op1`.

Together, these three cases are all pattern matched and the function returns the replacement if it can, or a null pointer if no replacement is possible. The caller of this function (`SimplifyInstruction`) is a dispatcher that does a switch on the instruction opcode, dispatching to the per-opcode helper functions. It is called from various optimizations. A simple driver looks like this:

```
for (BasicBlock::iterator I = BB->begin(), E = BB->end(); I != E; ++I)
    if (Value *V = SimplifyInstruction(I))
        I->replaceAllUsesWith(V);
```

This code simply loops over each instruction in a block, checking to see if any of them simplify. If so (because `SimplifyInstruction` returns non-null), it uses the `replaceAllUsesWith` method to update anything in the code using the simplifiable operation with the simpler form.

11.4 LLVM's Implementation of Three-Phase Design

In an LLVM-based compiler, a front end is responsible for parsing, validating and diagnosing errors in the input code, then translating the parsed code into LLVM IR (usually, but not always, by building an AST and then converting the AST to LLVM IR). This IR is optionally fed through a series of analysis and optimization passes which improve the code, then is sent into a code generator to produce native machine code, as shown in Figure 11.3. This is a very straightforward implementation of the three-phase design, but this simple description glosses over some of the power and flexibility that the LLVM architecture derives from LLVM IR.

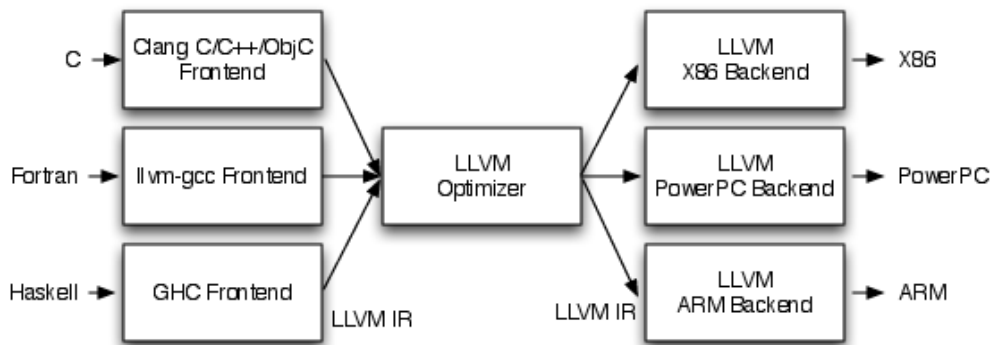


Figure 11.3: LLVM's Implementation of the Three-Phase Design

LLVM IR is a Complete Code Representation

In particular, LLVM IR is both well specified and the *only* interface to the optimizer. This property means that all you need to know to write a front end for LLVM is what LLVM IR is, how it works, and the invariants it expects. Since LLVM IR has a first-class textual form, it is both possible and reasonable to build a front end that outputs LLVM IR as text, then uses Unix pipes to send it through the optimizer sequence and code generator of your choice.

It might be surprising, but this is actually a pretty novel property to LLVM and one of the major reasons for its success in a broad range of different applications. Even the widely successful and relatively well-architected GCC compiler does not have this property: its GIMPLE mid-level representation is not a self-contained representation. As a simple example, when the GCC code generator goes to emit DWARF debug information, it reaches back and walks the source level “tree” form. GIMPLE itself uses a “tuple” representation for the operations in the code, but (at least as of GCC 4.5) still represents operands as references back to the source level tree form.

The implications of this are that front-end authors need to know and produce GCC's tree data structures as well as GIMPLE to write a GCC front end. The GCC back end has similar problems, so they also need to know bits and pieces of how the RTL back end works as well. Finally, GCC doesn't

have a way to dump out “everything representing my code”, or a way to read and write GIMPLE (and the related data structures that form the representation of the code) in text form. The result is that it is relatively hard to experiment with GCC, and therefore it has relatively few front ends.

LLVM is a Collection of Libraries

After the design of LLVM IR, the next most important aspect of LLVM is that it is designed as a set of libraries, rather than as a monolithic command line compiler like GCC or an opaque virtual machine like the JVM or .NET virtual machines. LLVM is an infrastructure, a collection of useful compiler technology that can be brought to bear on specific problems (like building a C compiler, or an optimizer in a special effects pipeline). While one of its most powerful features, it is also one of its least understood design points.

Let’s look at the design of the optimizer as an example: it reads LLVM IR in, chews on it a bit, then emits LLVM IR which hopefully will execute faster. In LLVM (as in many other compilers) the optimizer is organized as a pipeline of distinct optimization passes each of which is run on the input and has a chance to do something. Common examples of passes are the inliner (which substitutes the body of a function into call sites), expression reassociation, loop invariant code motion, etc. Depending on the optimization level, different passes are run: for example at -O0 (no optimization) the Clang compiler runs no passes, at -O3 it runs a series of 67 passes in its optimizer (as of LLVM 2.8).

Each LLVM pass is written as a C++ class that derives (indirectly) from the Pass class. Most passes are written in a single .cpp file, and their subclass of the Pass class is defined in an anonymous namespace (which makes it completely private to the defining file). In order for the pass to be useful, code outside the file has to be able to get it, so a single function (to create the pass) is exported from the file. Here is a slightly simplified example of a pass to make things concrete.⁶

```
namespace {
  class Hello : public FunctionPass {
  public:
    // Print out the names of functions in the LLVM IR being optimized.
    virtual bool runOnFunction(Function &F) {
      cerr << "Hello: " << F.getName() << "\n";
      return false;
    }
  };
}

FunctionPass *createHelloPass() { return new Hello(); }
```

As mentioned, the LLVM optimizer provides dozens of different passes, each of which are written in a similar style. These passes are compiled into one or more .o files, which are then built into a series of archive libraries (.a files on Unix systems). These libraries provide all sorts of analysis and transformation capabilities, and the passes are as loosely coupled as possible: they are expected to stand on their own, or explicitly declare their dependencies among other passes if they depend on some other analysis to do their job. When given a series of passes to run, the LLVM PassManager uses the explicit dependency information to satisfy these dependencies and optimize the execution of passes.

⁶For all the details, please see *Writing an LLVM Pass manual* at <http://llvm.org/docs/WritingAnLLVMPass.html>.

Libraries and abstract capabilities are great, but they don't actually solve problems. The interesting bit comes when someone wants to build a new tool that can benefit from compiler technology, perhaps a JIT compiler for an image processing language. The implementer of this JIT compiler has a set of constraints in mind: for example, perhaps the image processing language is highly sensitive to compile-time latency and has some idiomatic language properties that are important to optimize away for performance reasons.

The library-based design of the LLVM optimizer allows our implementer to pick and choose both the order in which passes execute, and which ones make sense for the image processing domain: if everything is defined as a single big function, it doesn't make sense to waste time on inlining. If there are few pointers, alias analysis and memory optimization aren't worth bothering about. However, despite our best efforts, LLVM doesn't magically solve all optimization problems! Since the pass subsystem is modularized and the PassManager itself doesn't know anything about the internals of the passes, the implementer is free to implement their own language-specific passes to cover for deficiencies in the LLVM optimizer or to exploit language-specific optimization opportunities. Figure 11.4 shows a simple example for our hypothetical XYZ image processing system:

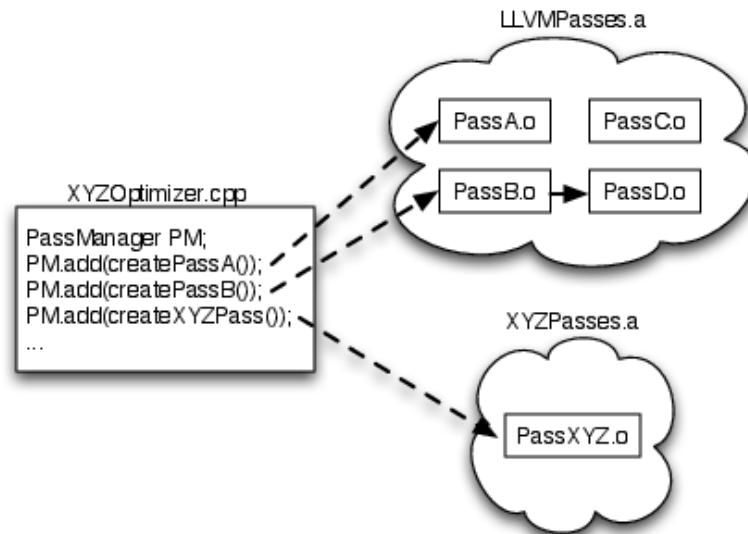


Figure 11.4: Hypothetical XYZ System using LLVM

Once the set of optimizations is chosen (and similar decisions are made for the code generator) the image processing compiler is built into an executable or dynamic library. Since the only reference to the LLVM optimization passes is the simple create function defined in each .o file, and since the optimizers live in .a archive libraries, only the optimization passes *that are actually used* are linked into the end application, not the entire LLVM optimizer. In our example above, since there is a reference to PassA and PassB, they will get linked in. Since PassB uses PassD to do some analysis, PassD gets linked in. However, since PassC (and dozens of other optimizations) aren't used, its code isn't linked into the image processing application.

This is where the power of the library-based design of LLVM comes into play. This straightforward design approach allows LLVM to provide a vast amount of capability, some of which may only be useful to specific audiences, without punishing clients of the libraries that just want to do simple

things. In contrast, traditional compiler optimizers are built as a tightly interconnected mass of code, which is much more difficult to subset, reason about, and come up to speed on. With LLVM you can understand individual optimizers without knowing how the whole system fits together.

This library-based design is also the reason why so many people misunderstand what LLVM is all about: the LLVM libraries have many capabilities, but they don't actually *do* anything by themselves. It is up to the designer of the client of the libraries (e.g., the Clang C compiler) to decide how to put the pieces to best use. This careful layering, factoring, and focus on subset-ability is also why the LLVM optimizer can be used for such a broad range of different applications in different contexts. Also, just because LLVM provides JIT compilation capabilities, it doesn't mean that every client uses it.

11.5 Design of the Retargetable LLVM Code Generator

The LLVM code generator is responsible for transforming LLVM IR into target specific machine code. On the one hand, it is the code generator's job to produce the best possible machine code for any given target. Ideally, each code generator should be completely custom code for the target, but on the other hand, the code generators for each target need to solve very similar problems. For example, each target needs to assign values to registers, and though each target has different register files, the algorithms used should be shared wherever possible.

Similar to the approach in the optimizer, LLVM's code generator splits the code generation problem into individual passes—instruction selection, register allocation, scheduling, code layout optimization, and assembly emission—and provides many builtin passes that are run by default. The target author is then given the opportunity to choose among the default passes, override the defaults and implement completely custom target-specific passes as required. For example, the x86 back end uses a register-pressure-reducing scheduler since it has very few registers, but the PowerPC back end uses a latency optimizing scheduler since it has many of them. The x86 back end uses a custom pass to handle the x87 floating point stack, and the ARM back end uses a custom pass to place constant pool islands inside functions where needed. This flexibility allows target authors to produce great code without having to write an entire code generator from scratch for their target.

LLVM Target Description Files

The “mix and match” approach allows target authors to choose what makes sense for their architecture and permits a large amount of code reuse across different targets. This brings up another challenge: each shared component needs to be able to reason about target specific properties in a generic way. For example, a shared register allocator needs to know the register file of each target and the constraints that exist between instructions and their register operands. LLVM's solution to this is for each target to provide a target description in a declarative domain-specific language (a set of .td files) processed by the tblgen tool. The (simplified) build process for the x86 target is shown in Figure 11.5.

The different subsystems supported by the .td files allow target authors to build up the different pieces of their target. For example, the x86 back end defines a register class that holds all of its 32-bit registers named “GR32” (in the .td files, target specific definitions are all caps) like this:

```
def GR32 : RegisterClass<[i32], 32,  
    [EAX, ECX, EDX, ESI, EDI, EBX, EBP, ESP,  
    R8D, R9D, R10D, R11D, R14D, R15D, R12D, R13D]> { ... }
```

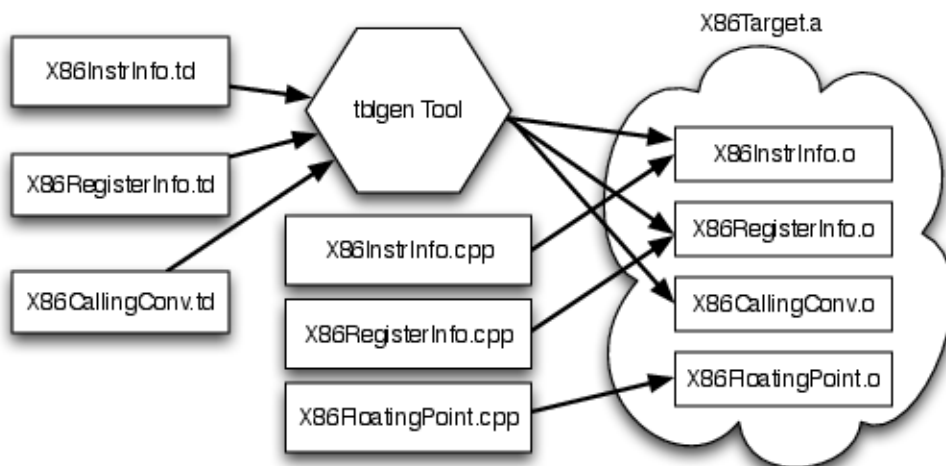


Figure 11.5: Simplified x86 Target Definition

This definition says that registers in this class can hold 32-bit integer values (“i32”), prefer to be 32-bit aligned, have the specified 16 registers (which are defined elsewhere in the .td files) and have some more information to specify preferred allocation order and other things. Given this definition, specific instructions can refer to this, using it as an operand. For example, the “complement a 32-bit register” instruction is defined as:

```

let Constraints = "$src = $dst" in
def NOT32r : I<0xF7, MRM2r,
    (outs GR32:$dst), (ins GR32:$src),
    "not{1}\t$dst",
    [(set GR32:$dst, (not GR32:$src))]>;

```

This definition says that NOT32r is an instruction (it uses the I tblgen class), specifies encoding information (0xF7, MRM2r), specifies that it defines an “output” 32-bit register \$dst and has a 32-bit register “input” named \$src (the GR32 register class defined above defines which registers are valid for the operand), specifies the assembly syntax for the instruction (using the {} syntax to handle both AT&T and Intel syntax), specifies the effect of the instruction and provides the pattern that it should match on the last line. The “let” constraint on the first line tells the register allocator that the input and output register must be allocated to the same physical register.

This definition is a very dense description of the instruction, and the common LLVM code can do a lot with information derived from it (by the tblgen tool). This one definition is enough for instruction selection to form this instruction by pattern matching on the input IR code for the compiler. It also tells the register allocator how to process it, is enough to encode and decode the instruction to machine code bytes, and is enough to parse and print the instruction in a textual form. These capabilities allow the x86 target to support generating a stand-alone x86 assembler (which is a drop-in replacement for the “gas” GNU assembler) and disassemblers from the target description as well as handle encoding the instruction for the JIT.

In addition to providing useful functionality, having multiple pieces of information generated from the same “truth” is good for other reasons. This approach makes it almost infeasible for the

assembler and disassembler to disagree with each other in either assembly syntax or in the binary encoding. It also makes the target description easily testable: instruction encodings can be unit tested without having to involve the entire code generator.

While we aim to get as much target information as possible into the .td files in a nice declarative form, we still don't have everything. Instead, we require target authors to write some C++ code for various support routines and to implement any target specific passes they might need (like `X86FloatingPoint.cpp`, which handles the x87 floating point stack). As LLVM continues to grow new targets, it becomes more and more important to increase the amount of the target that can be expressed in the .td file, and we continue to increase the expressiveness of the .td files to handle this. A great benefit is that it gets easier and easier write targets in LLVM as time goes on.

11.6 Interesting Capabilities Provided by a Modular Design

Besides being a generally elegant design, modularity provides clients of the LLVM libraries with several interesting capabilities. These capabilities stem from the fact that LLVM provides functionality, but lets the client decide most of the *policies* on how to use it.

Choosing When and Where Each Phase Runs

As mentioned earlier, LLVM IR can be efficiently (de)serialized to/from a binary format known as LLVM bitcode. Since LLVM IR is self-contained, and serialization is a lossless process, we can do part of compilation, save our progress to disk, then continue work at some point in the future. This feature provides a number of interesting capabilities including support for link-time and install-time optimization, both of which delay code generation from “compile time”.

Link-Time Optimization (LTO) addresses the problem where the compiler traditionally only sees one translation unit (e.g., a .c file with all its headers) at a time and therefore cannot do optimizations (like inlining) across file boundaries. LLVM compilers like Clang support this with the `-f1` to or `-O4` command line option. This option instructs the compiler to emit LLVM bitcode to the .o file instead of writing out a native object file, and delays code generation to link time, shown in Figure 11.6.

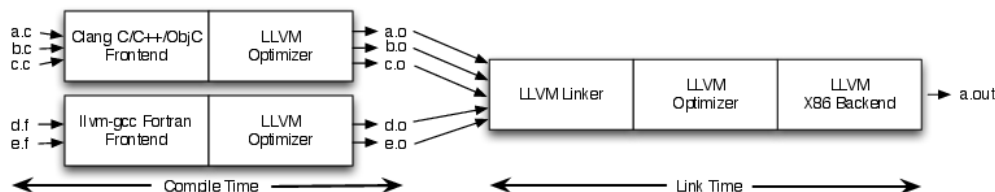


Figure 11.6: Link-Time Optimization

Details differ depending on which operating system you're on, but the important bit is that the linker detects that it has LLVM bitcode in the .o files instead of native object files. When it sees this, it reads all the bitcode files into memory, links them together, then runs the LLVM optimizer over the aggregate. Since the optimizer can now see across a much larger portion of the code, it can inline, propagate constants, do more aggressive dead code elimination, and more across file boundaries. While many modern compilers support LTO, most of them (e.g., GCC, Open64, the Intel compiler, etc.) do so by having an expensive and slow serialization process. In LLVM, LTO

falls out naturally from the design of the system, and works across different source languages (unlike many other compilers) because the IR is truly source language neutral.

Install-time optimization is the idea of delaying code generation even later than link time, all the way to install time, as shown in Figure 11.7. Install time is a very interesting time (in cases when software is shipped in a box, downloaded, uploaded to a mobile device, etc.), because this is when you find out the specifics of the device you're targeting. In the x86 family for example, there are broad variety of chips and characteristics. By delaying instruction choice, scheduling, and other aspects of code generation, you can pick the best answers for the specific hardware an application ends up running on.

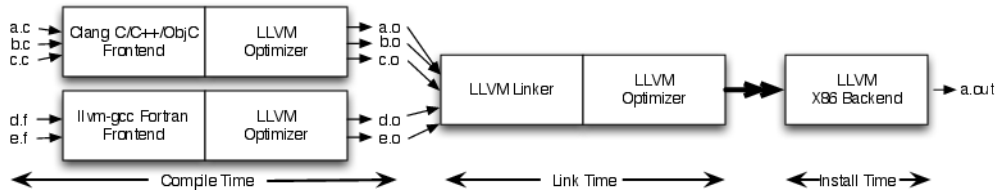


Figure 11.7: Install-Time Optimization

Unit Testing the Optimizer

Compilers are very complicated, and quality is important, therefore testing is critical. For example, after fixing a bug that caused a crash in an optimizer, a regression test should be added to make sure it doesn't happen again. The traditional approach to testing this is to write a .c file (for example) that is run through the compiler, and to have a test harness that verifies that the compiler doesn't crash. This is the approach used by the GCC test suite, for example.

The problem with this approach is that the compiler consists of many different subsystems and even many different passes in the optimizer, all of which have the opportunity to change what the input code looks like by the time it gets to the previously buggy code in question. If something changes in the front end or an earlier optimizer, a test case can easily fail to test what it is supposed to be testing.

By using the textual form of LLVM IR with the modular optimizer, the LLVM test suite has highly focused regression tests that can load LLVM IR from disk, run it through exactly one optimization pass, and verify the expected behavior. Beyond crashing, a more complicated behavioral test wants to verify that an optimization is actually performed. Here is a simple test case that checks to see that the constant propagation pass is working with add instructions:

```

; RUN: opt < %s -constprop -S | FileCheck %s
define i32 @test() {
    %A = add i32 4, 5
    ret i32 %A
    ; CHECK: @test()
    ; CHECK: ret i32 9
}

```

The RUN line specifies the command to execute: in this case, the opt and FileCheck command line tools. The opt program is a simple wrapper around the LLVM pass manager, which links in all the standard passes (and can dynamically load plugins containing other passes) and exposes them

through to the command line. The FileCheck tool verifies that its standard input matches a series of CHECK directives. In this case, this simple test is verifying that the constprop pass is folding the add of 4 and 5 into 9.

While this might seem like a really trivial example, this is very difficult to test by writing .c files: front ends often do constant folding as they parse, so it is very difficult and fragile to write code that makes its way downstream to a constant folding optimization pass. Because we can load LLVM IR as text and send it through the specific optimization pass we’re interested in, then dump out the result as another text file, it is really straightforward to test exactly what we want, both for regression and feature tests.

Automatic Test Case Reduction with BugPoint

When a bug is found in a compiler or other client of the LLVM libraries, the first step to fixing it is to get a test case that reproduces the problem. Once you have a test case, it is best to minimize it to the smallest example that reproduces the problem, and also narrow it down to the part of LLVM where the problem happens, such as the optimization pass at fault. While you eventually learn how to do this, the process is tedious, manual, and particularly painful for cases where the compiler generates incorrect code but does not crash.

The LLVM BugPoint tool⁷ uses the IR serialization and modular design of LLVM to automate this process. For example, given an input .ll or .bc file along with a list of optimization passes that causes an optimizer crash, BugPoint reduces the input to a small test case and determines which optimizer is at fault. It then outputs the reduced test case and the opt command used to reproduce the failure. It finds this by using techniques similar to “delta debugging” to reduce the input and the optimizer pass list. Because it knows the structure of LLVM IR, BugPoint does not waste time generating invalid IR to input to the optimizer, unlike the standard “delta” command line tool.

In the more complex case of a miscompilation, you can specify the input, code generator information, the command line to pass to the executable, and a reference output. BugPoint will first determine if the problem is due to an optimizer or a code generator, and will then repeatedly partition the test case into two pieces: one that is sent into the “known good” component and one that is sent into the “known buggy” component. By iteratively moving more and more code out of the partition that is sent into the known buggy code generator, it reduces the test case.

BugPoint is a very simple tool and has saved countless hours of test case reduction throughout the life of LLVM. No other open source compiler has a similarly powerful tool, because it relies on a well-defined intermediate representation. That said, BugPoint isn’t perfect, and would benefit from a rewrite. It dates back to 2002, and is typically only improved when someone has a really tricky bug to track down that the existing tool doesn’t handle well. It has grown over time, accreting new features (such as JIT debugging) without a consistent design or owner.

11.7 Retrospective and Future Directions

LLVM’s modularity wasn’t originally designed to directly achieve any of the goals described here. It was a self-defense mechanism: it was obvious that we wouldn’t get everything right on the first try. The modular pass pipeline, for example, exists to make it easier to isolate passes so that they can be discarded after being replaced by better implementations⁸.

⁷<http://llvm.org/docs/Bugpoint.html>

⁸I often say that none of the subsystems in LLVM are really good until they have been rewritten at least once.

Another major aspect of LLVM remaining nimble (and a controversial topic with clients of the libraries) is our willingness to reconsider previous decisions and make widespread changes to APIs without worrying about backwards compatibility. Invasive changes to LLVM IR itself, for example, require updating all of the optimization passes and cause substantial churn to the C++ APIs. We've done this on several occasions, and though it causes pain for clients, it is the right thing to do to maintain rapid forward progress. To make life easier for external clients (and to support bindings for other languages), we provide C wrappers for many popular APIs (which are intended to be extremely stable) and new versions of LLVM aim to continue reading old `.ll` and `.bc` files.

Looking forward, we would like to continue making LLVM more modular and easier to subset. For example, the code generator is still too monolithic: it isn't currently possible to subset LLVM based on features. For example, if you'd like to use the JIT, but have no need for inline assembly, exception handling, or debug information generation, it should be possible to build the code generator without linking in support for these features. We are also continuously improving the quality of code generated by the optimizer and code generator, adding IR features to better support new language and target constructs, and adding better support for performing high-level language-specific optimizations in LLVM.

The LLVM project continues to grow and improve in numerous ways. It is really exciting to see the number of different ways that LLVM is being used in other projects and how it keeps turning up in surprising new contexts that its designers never even thought about. The new LLDB debugger is a great example of this: it uses the C/C++/Objective-C parsers from Clang to parse expressions, uses the LLVM JIT to translate these into target code, uses the LLVM disassemblers, and uses LLVM targets to handle calling conventions among other things. Being able to reuse this existing code allows people developing debuggers to focus on writing the debugger logic, instead of reimplementing yet another (marginally correct) C++ parser.

Despite its success so far, there is still a lot left to be done, as well as the ever-present risk that LLVM will become less nimble and more calcified as it ages. While there is no magic answer to this problem, I hope that the continued exposure to new problem domains, a willingness to reevaluate previous decisions, and to redesign and throw away code will help. After all, the goal isn't to be perfect, it is to keep getting better over time.

Mercurial

Dirkjan Ochtman

Mercurial is a modern distributed version control system (VCS), written mostly in Python with bits and pieces in C for performance. In this chapter, I will discuss some of the decisions involved in designing Mercurial's algorithms and data structures. First, allow me to go into a short history of version control systems, to add necessary context.

12.1 A Short History of Version Control

While this chapter is primarily about Mercurial's software architecture, many of the concepts are shared with other version control systems. In order to fruitfully discuss Mercurial, I'd like to start off by naming some of the concepts and actions in different version control systems. To put all of this in perspective, I will also provide a short history of the field.

Version control systems were invented to help developers work on software systems simultaneously, without passing around full copies and keeping track of file changes themselves. Let's generalize from software source code to any tree of files. One of the primary functions of version control is to pass around changes to the tree. The basic cycle is something like this:

1. Get the latest tree of files from someone else
2. Work on a set of changes to this version of the tree
3. Publish the changes so that others can retrieve them

The first action, to get a local tree of files, is called a *checkout*. The store where we retrieve and publish our changes is called a *repository*, while the result of the checkout is called a *working directory*, *working tree*, or *working copy*. Updating a working copy with the latest files from the repository is simply called *update*; sometimes this requires *merging*, i.e., combining changes from different users in a single file. A *diff* command allows us to review changes between two revisions of a tree or file, where the most common mode is to check the local (unpublished) changes in your working copy. Changes are published by issuing a *commit* command, which will save the changes from the working directory to the repository.

Centralized Version Control

The first version control system was the Source Code Control System, SCCS, first described in 1975. It was mostly a way of saving deltas to single files that was more efficient than just keeping around copies, and didn't help with publishing these changes to others. It was followed in 1982 by the Revision Control System, RCS, which was a more evolved and free alternative to SCCS (and which is still being maintained by the GNU project).

After RCS came CVS, the Concurrent Versioning System, first released in 1986 as a set of scripts to manipulate RCS revision files in groups. The big innovation in CVS is the notion that multiple users can edit simultaneously, with merges being done after the fact (concurrent edits). This also required the notion of edit conflicts. Developers may only commit a new version of some file if it's based on the latest version available in the repository. If there are changes in the repository and in my working directory, I have to resolve any conflicts resulting from those changes (edits changing the same lines).

CVS also pioneered the notions of *branches*, which allow developers to work on different things in parallel, and *tags*, which enable naming a consistent snapshot for easy reference. While CVS deltas were initially communicated via the repository on a shared filesystem, at some point CVS also implemented a client-server architecture for use across large networks (such as the Internet).

In 2000, three developers got together to build a new VCS, christened Subversion, with the intention of fixing some of the larger warts in CVS. Most importantly, Subversion works on whole trees at a time, meaning changes in a revisions should be atomic, consistent, isolated, and durable. Subversion working copies also retain a pristine version of the checked out revision in the working directory, so that the common diff operation (comparing the local tree against a checked-out changeset) is local and thus fast.

One of the interesting concepts in Subversion is that tags and branches are part of a project tree. A Subversion project is usually divided into three areas: tags, branches, and trunk. This design has proved very intuitive to users who were unfamiliar with version control systems, although the flexibility inherent in this design has caused numerous problems for conversion tools, mostly because tags and branches have more structural representation in other systems.

All of the aforementioned systems are said to be *centralized*; to the extent that they even know how to exchange changes (starting with CVS), they rely on some other computer to keep track of the history of the repository. *Distributed* version control systems instead keep a copy of all or most of the repository history on each computer that has a working directory of that repository.

Distributed Version Control

While Subversion was a clear improvement over CVS, there are still a number of shortcomings. For one thing, in all centralized systems, committing a changeset and publishing it are effectively the same thing, since repository history is centralized in one place. This means that committing changes without network access is impossible. Secondly, repository access in centralized systems always needs one or more network round trips, making it relatively slow compared to the local accesses needed with distributed systems. Third, the systems discussed above were not very good at tracking merges (some have since grown better at it). In large groups working concurrently, it's important that the version control system records what changes have been included in some new revision, so that nothing gets lost and subsequent merges can make use of this information. Fourth, the centralization required by traditional VCSes sometimes seems artificial, and promotes a single place for integration. Advocates of distributed VCSes argue that a more distributed system allows for a more organic

organization, where developers can push around and integrate changes as the project requires at each point in time.

A number of new tools have been developed to address these needs. From where I sit (the open source world), the most notable three of these in 2011 are Git, Mercurial and Bazaar. Both Git and Mercurial were started in 2005 when the Linux kernel developers decided to no longer use the proprietary BitKeeper system. Both were started by Linux kernel developers (Linus Torvalds and Matt Mackall, respectively) to address the need for a version control system that could handle hundreds of thousands of changesets in tens of thousands of files (for example, the kernel). Both Matt and Linus were also heavily influenced by the Monotone VCS. Bazaar was developed separately but gained widespread usage around the same time, when it was adopted by Canonical for use with all of their projects.

Building a distributed version control system obviously comes with some challenges, many of which are inherent in any distributed system. For one thing, while the source control server in centralized systems always provided a canonical view of history, there is no such thing in a distributed VCS. Changesets can be committed in parallel, making it impossible to temporally order revisions in any given repository.

The solution that has been almost universally adopted is to use a directed acyclic graph (DAG) of changesets instead of a linear ordering (Figure 12.1). That is, a newly committed changeset is the child revision of the revision it was based on, and no revision can depend on itself or its descendant revisions. In this scheme, we have three special types of revisions: *root revisions* which have no parents (a repository can have multiple roots), *merge revisions* which have more than one parent, and *head revisions* which have no children. Each repository starts from an empty root revision and proceeds from there along a line of changesets, ending up in one or more heads. When two users have committed independently and one of them wants to pull in the changes from the other, he or she will have to explicitly merge the other's changes into a new revision, which he subsequently commits as a merge revision.

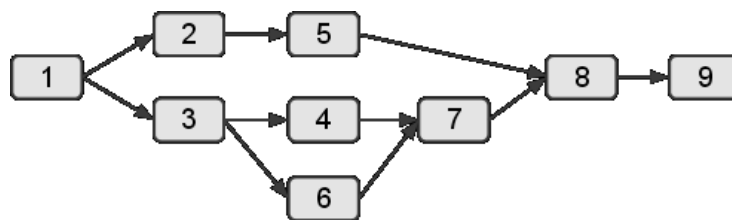


Figure 12.1: Directed Acyclic Graph of Revisions

Note that the DAG model helps solve some of the problems that are hard to solve in centralized version control systems: merge revisions are used to record information about newly merged branches of the DAG. The resulting graph can also usefully represent a large group of parallel branches, merging into smaller groups, finally merging into one special branch that's considered canonical.

This approach requires that the system keep track of the ancestry relations between changesets; to facilitate exchange of changeset data, this is usually done by having changesets keep track of their parents. To do this, changesets obviously also need some kind of identifier. While some systems use a UUID or a similar kind of scheme, both Git and Mercurial have opted to use SHA1 hashes of the contents of the changesets. This has the additional useful property that the changeset ID can be used to verify the changeset contents. In fact, because the parents are included in the hashed data, all history leading up to any revision can be verified using its hash. Author names, commit messages,

timestamps and other changeset metadata is hashed just like the actual file contents of a new revision, so that they can also be verified. And since timestamps are recorded at commit time, they too do not necessarily progress linearly in any given repository.

All of this can be hard for people who have previously only used centralized VCSes to get used to: there is no nice integer to globally name a revision, just a 40-character hexadecimal string. Moreover, there's no longer any global ordering, just a local ordering; the only global "ordering" is a DAG instead of a line. Accidentally starting a new head of development by committing against a parent revision that already had another child changeset can be confusing when you're used to a warning from the VCS when this kind of thing happens.

Luckily, there are tools to help visualize the tree ordering, and Mercurial provides an unambiguous short version of the changeset hash *and* a local-only linear number to aid identification. The latter is a monotonically climbing integer that indicates the order in which changesets have entered the clone. Since this order can be different from clone to clone, it cannot be relied on for non-local operations.

12.2 Data Structures

Now that the concept of a DAG should be somewhat clear, let's try and see how DAGs are stored in Mercurial. The DAG model is central to the inner workings of Mercurial, and we actually use several different DAGs in the repository storage on disk (as well as the in-memory structure of the code). This section explains what they are and how they fit together.

Challenges

Before we dive into actual data structures, I'd like to provide some context about the environment in which Mercurial evolved. The first notion of Mercurial can be found in an email Matt Mackall sent to the Linux Kernel Mailing List on April 20, 2005. This happened shortly after it was decided that BitKeeper could no longer be used for the development of the kernel. Matt started his mail by outlining some goals: to be simple, scalable, and efficient.

In [Mac06], Matt claimed that a modern VCS must deal with trees containing millions of files, handle millions of changesets, and scale across many thousands of users creating new revisions in parallel over a span of decades. Having set the goals, he reviewed the limiting technology factors:

- speed: CPU
- capacity: disk and memory
- bandwidth: memory, LAN, disk, and WAN
- disk seek rate

Disk seek rate and WAN bandwidth are the limiting factors today, and should thus be optimized for. The paper goes on to review common scenarios or criteria for evaluating the performance of such a system at the file level:

- Storage compression: what kind of compression is best suited to save the file history on disk? Effectively, what algorithm makes the most out of the I/O performance while preventing CPU time from becoming a bottleneck?
- Retrieving arbitrary file revisions: a number of version control systems will store a given revision in such a way that a large number of older revisions must be read to reconstruct the newer one (using deltas). We want to control this to make sure that retrieving old revisions is still fast.

- Adding file revisions: we regularly add new revisions. We don't want to rewrite old revisions every time we add a new one, because that would become too slow when there are many revisions.
- Showing file history: we want to be able to review a history of all changesets that touched a certain file. This also allows us to do annotations (which used to be called `blame` in CVS but was renamed to `annotate` in some later systems to remove the negative connotation): reviewing the originating changeset for each line currently in a file.

The paper goes on to review similar scenarios at the project level. Basic operations at this level are checking out a revision, committing a new revision, and finding differences in the working directory. The latter, in particular, can be slow for large trees (like those of the Mozilla or NetBeans projects, both of which use Mercurial for their version control needs).

Fast Revision Storage: Revlogs

The solution Matt came up with for Mercurial is called the *revlog* (short for revision log). The revlog is a way of efficiently storing revisions of file contents (each with some amount of changes compared to the previous version). It needs to be efficient in both access time (thus optimizing for disk seeks) and storage space, guided by the common scenarios outlined in the previous section. To do this, a revlog is really two files on disk: an index and the data file.

6 bytes	hunk offset
2 bytes	flags
4 bytes	hunk length
4 bytes	uncompressed length
4 bytes	base revision
4 bytes	link revision
4 bytes	parent 1 revision
4 bytes	parent 2 revision
32 bytes	hash

Table 12.1: Mercurial Record Format

The index consists of fixed-length records, whose contents are detailed in Table 12.1. Having fixed-length records is nice, because it means that having the local revision number allows direct (i.e., constant-time) access to the revision: we can simply read to the position ($\text{index-length} \times \text{revision}$) in the index file, to locate the data. Separating the index from the data also means we can quickly read the index data without having to seek the disk through all the file data.

The hunk offset and hunk length specify a chunk of the data file to read in order to get the compressed data for that revision. To get the original data we have to start by reading the base revision, and apply deltas through to this revision. The trick here is the decision on when to store a new base revision. This decision is based on the cumulative size of the deltas compared to the uncompressed length of the revision (data is compressed using `zlib` to use even less space on disk). By limiting the length of the delta chain in this way, we make sure that reconstruction of the data in a given revision does not require reading and applying lots of deltas.

Link revisions are used to have dependent revlogs point back to the highest-level revlog (we'll talk more about this in a little bit), and the parent revisions are stored using the local integer revision number. Again, this makes it easy to look up their data in the relevant revlog. The hash is used to

save the unique identifier for this changeset. We have 32 bytes instead of the 20 bytes required for SHA1 in order to allow future expansion.

The Three Revlogs

With the revlog providing a generic structure for historic data, we can layer the data model for our file tree on top of that. It consists of three types of revlogs: the *changelog*, *manifests*, and *filelogs*. The changelog contains metadata for each revision, with a pointer into the manifest revlog (that is, a node id for one revision in the manifest revlog). In turn, the manifest is a file that has a list of filenames plus the node id for each file, pointing to a revision in that file's filelog. In the code, we have classes for changelog, manifest, and filelog that are subclasses of the generic revlog class, providing a clean layering of both concepts.

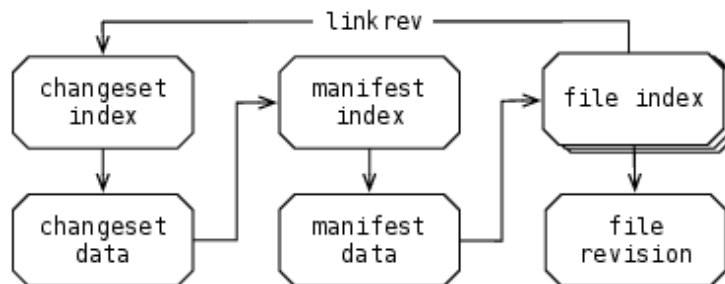


Figure 12.2: Log Structure

A changelog revision looks like this:

```
0a773e3480fe58d62dcc67bd9f7380d6403e26fa
Dirkjan Ochtman <dirkjan@ochtman.nl>
1276097267 -7200
mercurial/discovery.py
discovery: fix description line
```

This is the value you get from the revlog layer; the changelog layer turns it into a simple list of values. The initial line provides the manifest hash, then we get author name, date and time (in the form of a Unix timestamp and a timezone offset), a list of affected files, and the description message. One thing is hidden here: we allow arbitrary metadata in the changelog, and to stay backwards compatible we added those bits to go after the timestamp.

Next comes the manifest:

```
.hgignore\x006d2dc16e96ab48b2fcca44f7e9f4b8c3289cb701
.hgsigs\x00de81f258b33189c609d299fd605e6c72182d7359
.hgtags\x00b174a4a4813ddd89c1d2f88878e05acc58263efa
CONTRIBUTORS\x007c8afb9501740a450c549b4b1f002c803c45193a
COPYING\x005ac863e17c7035f1d11828d848fb2ca450d89794
...
```

This is the manifest revision that changeset 0a773e points to (Mercurial's UI allows us to shorten the identifier to any unambiguous prefix). It is a simple list of all files in the tree, one per line, where the

filename is followed by a NULL byte, followed by the hex-encoded node id that points into the file's filelog. Directories in the tree are not represented separately, but simply inferred from including slashes in the file paths. Remember that the manifest is diffed in storage just like every revlog, so this structure should make it easy for the revlog layer to store only changed files and their new hashes in any given revision. The manifest is usually represented as a hashtable-like structure in Mercurial's Python code, with filenames as keys and nodes as values.

The third type of revlog is the filelog. Filelogs are stored in Mercurial's internal store directory, where they're named almost exactly like the file they're tracking. The names are encoded a little bit to make sure things work across all major operating systems. For example, we have to deal with casefolding filesystems on Windows and Mac OS X, specific disallowed filenames on Windows, and different character encodings as used by the various filesystems. As you can imagine, doing this reliably across operating systems can be fairly painful. The contents of a filelog revision, on the other hand, aren't nearly as interesting: just the file contents, except with some optional metadata prefix (which we use for tracking file copies and renames, among other minor things).

This data model gives us complete access to the data store in a Mercurial repository, but it's not always very convenient. While the actual underlying model is vertically oriented (one filelog per file), Mercurial developers often found themselves wanting to deal with all details from a single revision, where they start from a changeset from the changelog and want easy access to the manifest and filelogs from that revision. They later invented another set of classes, layered cleanly on top of the revlogs, which do exactly that. These are called contexts.

One nice thing about the way the separate revlogs are set up is the ordering. By ordering appends so that filelogs get appended to first, then the manifest, and finally the changelog, the repository is always in a consistent state. Any process that starts reading the changelog can be sure all pointers into the other revlogs are valid, which takes care of a number of issues in this department. Nevertheless, Mercurial also has some explicit locks to make sure there are no two processes appending to the revlogs in parallel.

The Working Directory

A final important data structure is what we call the *dirstate*. The *dirstate* is a representation of what's in the working directory at any given point. Most importantly, it keeps track of what revision has been checked out: this is the baseline for all comparisons from the `status` or `diff` commands, and also determines the parent(s) for the next changeset to be committed. The *dirstate* will have two parents set whenever the `merge` command has been issued, trying to merge one set of changes into the other.

Because `status` and `diff` are very common operations (they help you check the progress of what you've currently got against the last changeset), the *dirstate* also contains a cache of the state of the working directory the last time it was traversed by Mercurial. Keeping track of last modified timestamps and file sizes makes it possible to speed up tree traversal. We also need to keep track of the state of the file: whether it's been added, removed, or merged in the working directory. This will again help speed up traversing the working directory, and makes it easy to get this information at commit time.

12.3 Versioning Mechanics

Now that you are familiar with the underlying data model and the structure of the code at the lower levels of Mercurial, let's move up a little bit and consider how Mercurial implements version control concepts on top of the foundation described in the previous section.

Branches

Branches are commonly used to separate different lines of development that will be integrated later. This might be because someone is experimenting with a new approach, just to be able to always keep the main line of development in a shippable state (feature branches), or to be able to quickly release fixes for an old release (maintenance branches). Both approaches are commonly used, and are supported by all modern version control systems. While implicit branches are common in DAG-based version control named branches (where the branch name is saved in the changeset metadata) are not as common.

Originally, Mercurial had no way to explicitly name branches. Branches were instead handled by making different clones and publishing them separately. This is effective, easy to understand, and especially useful for feature branches, because there is little overhead. However, in large projects, clones can still be quite expensive: while the repository store will be hardlinked on most filesystems, creating a separate working tree is slow and may require a lot of disk space.

Because of these downsides, Mercurial added a second way to do branches: including a branch name in the changeset metadata. A branch command was added that can set the branch name for the current working directory, such that that branch name will be used for the next commit. The normal update command can be used to update to a branch name, and a changeset committed on a branch will always be related to that branch. This approach is called *named branches*. However, it took a few more Mercurial releases before Mercurial started including a way to close these branches up again (closing a branch will hide the branch from view in a list of branches). Branch closing is implemented by adding an extra field in the changeset metadata, stating that this changeset closes the branch. If the branch has more than one head, all of them have to be closed before the branch disappears from the list of branches in the repository.

Of course, there's more than one way to do it. Git has a different way of naming branches, using references. References are names pointing to another object in the Git history, usually a changeset. This means that Git's branches are ephemeral: once you remove the reference, there is no trace of the branch ever having existed, similar to what you would get when using a separate Mercurial clone and merging it back into another clone. This makes it very easy and lightweight to manipulate branches locally, and prevents cluttering of the list of branches.

This way of branching turned out to be very popular, much more popular than either named branches or branch clones in Mercurial. This has resulted in the `bookmarksq` extension, which will probably be folded into Mercurial in the future. It uses a simple unversioned file to keep track of references. The wire protocol used to exchange Mercurial data has been extended to enable communicating about bookmarks, making it possible to push them around.

Tags

At first sight, the way Mercurial implements tags can be a bit confusing. The first time you add a tag (using the `tag` command), a file called `.hgtags` gets added to the repository and committed. Each

line in that file will contain a changeset node id and the tag name for that changeset node. Thus, the tags file is treated the same way as any other file in the repository.

There are three important reasons for this. First, it must be possible to change tags; mistakes do happen, and it should be possible to fix them or delete the mistake. Second, tags should be part of changeset history: it's valuable to see when a tag was made, by whom, and for what reason, or even if a tag was changed. Third, it should be possible to tag a changeset retroactively. For example, some projects extensively test drive a release artifact exported from the version control system before releasing it.

These properties all fall easily out of the .hgtags design. While some users are confused by the presence of the .hgtags file in their working directories, it makes integration of the tagging mechanism with other parts of Mercurial (for example, synchronization with other repository clones) very simple. If tags existed outside the source tree (as they do in Git, for example), separate mechanisms would have to exist to audit the origin of tags and to deal with conflicts from (parallel) duplicate tags. Even if the latter is rare, it's nice to have a design where these things are not even an issue.

To get all of this right, Mercurial only ever appends new lines to the .hgtags file. This also facilitates merging the file if the tags were created in parallel in different clones. The newest node id for any given tag always takes precedence, and adding the null node id (representing the empty root revision all repositories have in common) will have the effect of deleting the tag. Mercurial will also consider tags from all branches in the repository, using recency calculations to determine precedence among them.

12.4 General Structure

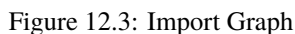
Mercurial is almost completely written in Python, with only a few bits and pieces in C because they are critical to the performance of the whole application. Python was deemed a more suitable choice for most of the code because it is much easier to express high-level concepts in a dynamic language like Python. Since much of the code is not really critical to performance, we don't mind taking the hit in exchange for making the coding easier for ourselves in most parts.

A Python module corresponds to a single file of code. Modules can contain as much code as needed, and are thus an important way to organize code. Modules may use types or call functions from other modules by explicitly importing the other modules. A directory containing an `__init__.py` module is said to be a package, and will expose all contained modules and packages to the Python importer.

Mercurial by default installs two packages into the Python path: `mercurial` and `hgext`. The `mercurial` package contains the core code required to run Mercurial, while `hgext` contains a number of extensions that were deemed useful enough to be delivered alongside the core. However, they must still be enabled by hand in a configuration file if desired (which we will discuss later.)

To be clear, Mercurial is a command-line application. This means that we have a simple interface: the user calls the `hg` script with a command. This command (like `log`, `diff` or `commit`) may take a number of options and arguments; there are also some options that are valid for all commands. Next, there are three different things that can happen to the interface.

- `hg` will often output something the user asked for or show status messages
- `hg` can ask for further input through command-line prompts
- `hg` may launch an external program (such as an editor for the commit message or a program to help merging code conflicts)



After this is done, we have to decide whether to create a repository object. While most commands require a local repository (represented by the `localrepo` class from the `localrepo` module), some commands may work on remote repositories (either HTTP, SSH, or some other registered form), while some commands can do their work without referring to any repository. The latter category includes the `init` command, for example, which is used to initialize a new repository.

176 Mercurial

12.5 Extensibility

One of the things that makes Mercurial powerful is the ability to write extensions for it. Since Python is a relatively easy language to get started with, and Mercurial's API is mostly quite well-designed (although certainly under-documented in places), a number of people actually first learned Python because they wanted to extend Mercurial.

Writing Extensions

Extensions must be enabled by adding a line to one of the configuration files read by Mercurial on startup; a key is provided along with the path to any Python module. There are several ways to add functionality:

- adding new commands;
- wrapping existing commands;
- wrapping the repository used;
- wrap any function in Mercurial; and
- add new repository types.

Adding new commands can be done simply by adding a hashtable called `cmdtable` to the extension module. This will get picked up by the extension loader, which will add it to the commands table considered when a command is dispatched. Similarly, extensions can define functions called `uisetup` and `reposesetup` which are called by the dispatching code after the UI and repository have been instantiated. One common behavior is to use a `reposesetup` function to wrap the repository in a repository subclass provided by the extension. This allows the extension to modify all kinds of basic behavior. For example, one extension I have written hooks into the `uisetup` and sets the `ui.username` configuration property based on the SSH authentication details available from the environment.

More extreme extensions can be written to add repository types. For example, the `hgsubversion` project (not included as part of Mercurial) registers a repository type for Subversion repositories. This makes it possible to clone from a Subversion repository almost as if it were a Mercurial repository. It's even possible to push back to a Subversion repository, although there are a number of edge cases because of the impedance mismatch between the two systems. The user interface, on the other hand, is completely transparent.

For those who want to fundamentally change Mercurial, there is something commonly called “monkeypatching” in the world of dynamic languages. Because extension code runs in the same address space as Mercurial, and Python is a fairly flexible language with extensive reflection capabilities, it's possible (and even quite easy) to modify any function or class defined in Mercurial. While this can result in kind of ugly hacks, it's also a very powerful mechanism. For example, the `highlight` extension that lives in `hgext` modifies the built-in webserver to add syntax highlighting to pages in the repository browser that allow you to inspect file contents.

There's one more way to extend Mercurial, which is much simpler: *aliases*. Any configuration file can define an alias as a new name for an existing command with a specific group of options already set. This also makes it possible to give shorter names to any commands. Recent versions of Mercurial also include the ability to call a shell command as an alias, so that you can design complicated commands using nothing but shell scripting.

Hooks

Version control systems have long provided hooks as a way for VCS events to interact with the outside world. Common usage includes sending off a notification to a continuous integration system¹ or updating the working directory on a web server so that changes become world-visible. Of course, Mercurial also includes a subsystem to invoke hooks like this.

In fact, it again contains two variants. One is more like traditional hooks in other version control systems, in that it invokes scripts in the shell. The other is more interesting, because it allows users to invoke Python hooks by specifying a Python module and a function name to call from that module. Not only is this faster because it runs in the same process, but it also hands off `repo` and `ui` objects, meaning you can easily initiate more complex interactions inside the VCS.

Hooks in Mercurial can be divided into pre-command, post-command, controlling, and miscellaneous hooks. The first two are trivially defined for any command by specifying a *pre-command* or *post-command* key in the hooks section of a configuration file. For the other two types, there's a predefined set of events. The difference in controlling hooks is that they are run right before something happens, and may not allow that event to progress further. This is commonly used to validate changesets in some way on a central server; because of Mercurial's distributed nature, no such checks can be enforced at commit time. For example, the Python project uses a hook to make sure some aspects of coding style are enforced throughout the code base—if a changeset adds code in a style that is not allowed, it will be rejected by the central repository.

Another interesting use of hooks is a pushlog, which is used by Mozilla and a number of corporate organizations. A pushlog records each push (since a push may contain any number of changesets) and records who initiated that push and when, providing a kind of audit trail for the repository.

12.6 Lessons Learned

One of the first decisions Matt made when he started to develop Mercurial was to develop it in Python. Python has been great for the extensibility (through extensions and hooks) and is very easy to code in. It also takes a lot of the work out of being compatible across different platforms, making it relatively easy for Mercurial to work well across the three major OSes. On the other hand, Python is slow compared to many other (compiled) languages; in particular, interpreter startup is relatively slow, which is particularly bad for tools that have many shorter invocations (such as a VCS) rather than longer running processes.

An early choice was made to make it hard to modify changesets after committing. Because it's impossible to change a revision without modifying its identity hash, “recalling” changesets after having published them on the public Internet is a pain, and Mercurial makes it hard to do so. However, changing unpublished revisions should usually be fine, and the community has been trying to make this easier since soon after the release. There are extensions that try to solve the problem, but they require learning steps that are not very intuitive to users who have previously used basic Mercurial.

Revlogs are good at reducing disk seeks, and the layered architecture of changelog, manifest and filelogs has worked very well. Committing is fast and relatively little disk space is used for revisions. However, some cases like file renames aren't very efficient due to the separate storage of revisions for each file; this will eventually be fixed, but it will require a somewhat hacky layering violation. Similarly, the per-file DAG used to help guide filelog storage isn't used a lot in practice, such that some code used to administrate that data could be considered to be overhead.

¹ See Chapter 6.

Another core focus of Mercurial has been to make it easy to learn. We try to provide most of the required functionality in a small set of core commands, with options consistent across commands. The intention is that Mercurial can mostly be learned progressively, especially for those users who have used another VCS before; this philosophy extends to the idea that extensions can be used to customize Mercurial even more for a particular use case. For this reason, the developers also tried to keep the UI in line with other VCSs, Subversion in particular. Similarly, the team has tried to provide good documentation, available from the application itself, with cross-references to other help topics and commands. We try hard to provide useful error messages, including hints of what to try instead of the operation that failed.

Some smaller choices made can be surprising to new users. For example, handling tags (as discussed in a previous section) by putting them in a separate file inside the working directory is something many users dislike at first, but the mechanism has some very desirable properties (though it certainly has its shortcomings as well). Similarly, other VCSs have opted to send only the checked out changeset and any ancestors to a remote host by default, whereas Mercurial sends every committed changeset the remote doesn't have. Both approaches make some amount of sense, and it depends on the style of development which one is the best for you.

As in any software project, there are a lot of trade-offs to be made. I think Mercurial made good choices, though of course with the benefit of 20/20 hindsight some other choices might have been more appropriate. Historically, Mercurial seems to be part of a first generation of distributed version control systems mature enough to be ready for general use. I, for one, am looking forward to seeing what the next generation will look like.

The NoSQL Ecosystem

Adam Marcus

Unlike most of the other projects in this book, NoSQL is not a tool, but an ecosystem composed of several complimentary and competing tools. The tools branded with the NoSQL monicker provide an alternative to SQL-based relational database systems for storing data. To understand NoSQL, we have to understand the space of available tools, and see how the design of each one explores the space of data storage possibilities.

If you are considering using a NoSQL storage system, you should first understand the wide space of options that NoSQL systems span. NoSQL systems do away with many of the traditional comforts of relational database systems, and operations which were typically encapsulated behind the system boundary of a database are now left to application designers. This requires you to take on the hat of a systems architect, which requires a more in-depth understanding of how such systems are built.

13.1 What's in a Name?

In defining the space of NoSQL, let's first take a stab at defining the name. Taken literally, a NoSQL system presents a query interface to the user that is not SQL. The NoSQL community generally takes a more inclusive view, suggesting that NoSQL systems provide alternatives to traditional relational databases, and allow developers to design projects which use *Not Only* a SQL interface. In some cases, you might replace a relational database with a NoSQL alternative, and in others you will employ a mix-and-match approach to different problems you encounter in application development.

Before diving into the world of NoSQL, let's explore the cases where SQL and the relational model suit your needs, and others where a NoSQL system might be a better fit.

SQL and the Relational Model

SQL is a declarative language for querying data. A declarative language is one in which a programmer specifies *what* they want the system to do, rather than procedurally defining *how* the system should do it. A few examples include: find the record for employee 39, project out only the employee name and phone number from their entire record, filter employee records to those that work in accounting, count the employees in each department, or join the data from the employees table with the managers table.

To a first approximation, SQL allows you to ask these questions without thinking about how the data is laid out on disk, which indices to use to access the data, or what algorithms to use to process the data. A significant architectural component of most relational databases is a *query optimizer*, which decides which of the many logically equivalent query plans to execute to most quickly answer a query. These optimizers are often better than the average database user, but sometimes they do not have enough information or have too simple a model of the system in order to generate the most efficient execution.

Relational databases, which are the most common databases used in practice, follow the *relational data model*. In this model, different real-world entities are stored in different tables. For example, all employees might be stored in an Employees table, and all departments might be stored in a Departments table. Each row of a table has various properties stored in columns. For example, employees might have an employee id, salary, birth date, and first/last names. Each of these properties will be stored in a column of the Employees table.

The relational model goes hand-in-hand with SQL. Simple SQL queries, such as filters, retrieve all records whose field matches some test (e.g., `employeeid = 3`, or `salary > $20000`). More complex constructs cause the database to do some extra work, such as joining data from multiple tables (e.g., what is the name of the department in which employee 3 works?). Other complex constructs such as aggregates (e.g., what is the average salary of my employees?) can lead to full-table scans.

The relational data model defines highly structured entities with strict relationships between them. Querying this model with SQL allows complex data traversals without too much custom development. The complexity of such modeling and querying has its limits, though:

- Complexity leads to unpredictability. SQL's expressiveness makes it challenging to reason about the cost of each query, and thus the cost of a workload. While simpler query languages might complicate application logic, they make it easier to provision data storage systems, which only respond to simple requests.
- There are many ways to model a problem. The relational data model is strict: the schema assigned to each table specifies the data in each row. If we are storing less structured data, or rows with more variance in the columns they store, the relational model may be needlessly restrictive. Similarly, application developers might not find the relational model perfect for modeling every kind of data. For example, a lot of application logic is written in object-oriented languages and includes high-level concepts such as lists, queues, and sets, and some programmers would like their persistence layer to model this.
- If the data grows past the capacity of one server, then the tables in the database will have to be partitioned across computers. To avoid JOINS having to cross the network in order to get data in different tables, we will have to denormalize it. Denormalization stores all of the data from different tables that one might want to look up at once in a single place. This makes our database look like a key-lookup storage system, leaving us wondering what other data models might better suit the data.

It's generally not wise to discard many years of design considerations arbitrarily. When you consider storing your data in a database, consider SQL and the relational model, which are backed by decades of research and development, offer rich modeling capabilities, and provide easy-to-understand guarantees about complex operations. NoSQL is a good option when you have a specific problem, such as large amounts of data, a massive workload, or a difficult data modeling decision for which SQL and relational databases might not have been optimized.

NoSQL Inspirations

The NoSQL movement finds much of its inspiration in papers from the research community. While many papers are at the core of design decisions in NoSQL systems, two stand out in particular.

Google's BigTable [CDG⁺06] presents an interesting data model, which facilitates sorted storage of multi-column historical data. Data is distributed to multiple servers using a hierarchical range-based partitioning scheme, and data is updated with strict consistency (a concept that we will eventually define in Section 13.5).

Amazon's Dynamo [DHJ⁺07] uses a different key-oriented distributed datastore. Dynamo's data model is simpler, mapping keys to application-specific blobs of data. The partitioning model is more resilient to failure, but accomplishes that goal through a looser data consistency approach called eventual consistency.

We will dig into each of these concepts in more detail, but it is important to understand that many of them can be mixed and matched. Some NoSQL systems such as HBase¹ sticks closely to the BigTable design. Another NoSQL system named Voldemort² replicates many of Dynamo's features. Still other NoSQL projects such as Cassandra³ have taken some features from BigTable (its data model) and others from Dynamo (its partitioning and consistency schemes).

Characteristics and Considerations

NoSQL systems part ways with the hefty SQL standard and offer simpler but piecemeal solutions for architecting storage solutions. These systems were built with the belief that in simplifying how a database operates over data, an architect can better predict the performance of a query. In many NoSQL systems, complex query logic is left to the application, resulting in a data store with more predictable query performance because of the lack of variability in queries

NoSQL systems part with more than just declarative queries over the relational data. Transactional semantics, consistency, and durability are guarantees that organizations such as banks demand of databases. *Transactions* provide an all-or-nothing guarantee when combining several potentially complex operations into one, such as deducting money from one account and adding the money to another. *Consistency* ensures that when a value is updated, subsequent queries will see the updated value. *Durability* guarantees that once a value is updated, it will be written to stable storage (such as a hard drive) and recoverable if the database crashes.

NoSQL systems relax some of these guarantees, a decision which, for many non-banking applications, can provide acceptable and predictable behavior in exchange for improved performance. These relaxations, combined with data model and query language changes, often make it easier to safely partition a database across multiple machines when the data grows beyond a single machine's capability.

NoSQL systems are still very much in their infancy. The architectural decisions that go into the systems described in this chapter are a testament to the requirements of various users. The biggest challenge in summarizing the architectural features of several open source projects is that each one is a moving target. Keep in mind that the details of individual systems will change. When you pick between NoSQL systems, you can use this chapter to guide your thought process, but not your feature-by-feature product selection.

¹<http://hbase.apache.org/>

²<http://project-voldemort.com/>

³<http://cassandra.apache.org/>

As you think about NoSQL systems, here is a roadmap of considerations:

- Data and query model: Is your data represented as rows, objects, data structures, or documents?
Can you ask the database to calculate aggregates over multiple records?
- Durability: When you change a value, does it immediately go to stable storage? Does it get stored on multiple machines in case one crashes?
- Scalability: Does your data fit on a single server? Do the amount of reads and writes require multiple disks to handle the workload?
- Partitioning: For scalability, availability, or durability reasons, does the data need to live on multiple servers? How do you know which record is on which server?
- Consistency: If you've partitioned and replicated your records across multiple servers, how do the servers coordinate when a record changes?
- Transactional semantics: When you run a series of operations, some databases allow you to wrap them in a transaction, which provides some subset of ACID (Atomicity, Consistency, Isolation, and Durability) guarantees on the transaction and all others currently running. Does your business logic require these guarantees, which often come with performance tradeoffs?
- Single-server performance: If you want to safely store data on disk, what on-disk data structures are best-gearred toward read-heavy or write-heavy workloads? Is writing to disk your bottleneck?
- Analytical workloads: We're going to pay a lot of attention to lookup-heavy workloads of the kind you need to run a responsive user-focused web application. In many cases, you will want to build dataset-sized reports, aggregating statistics across multiple users for example. Does your use-case and toolchain require such functionality?

While we will touch on all of these consideration, the last three, while equally important, see the least attention in this chapter.

13.2 NoSQL Data and Query Models

The *data model* of a database specifies how data is logically organized. Its *query model* dictates how the data can be retrieved and updated. Common data models are the relational model, key-oriented storage model, or various graph models. Query languages you might have heard of include SQL, key lookups, and MapReduce. NoSQL systems combine different data and query models, resulting in different architectural considerations.

Key-based NoSQL Data Models

NoSQL systems often part with the relational model and the full expressivity of SQL by restricting lookups on a dataset to a single field. For example, even if an employee has many properties, you might only be able to retrieve an employee by her ID. As a result, most queries in NoSQL systems are key lookup-based. The programmer selects a key to identify each data item, and can, for the most part, only retrieve items by performing a lookup for their key in the database.

In key lookup-based systems, complex join operations or multiple-key retrieval of the same data might require creative uses of key names. A programmer wishing to look up an employee by his employee ID and to look up all employees in a department might create two key types. For example, the key `employee:30` would point to an employee record for employee ID 30, and `employee_departments:20` might contain a list of all employees in department 20. A join operation

gets pushed into application logic: to retrieve employees in department 20, an application first retrieves a list of employee IDs from key `employee_departments:20`, and then loops over key lookups for each `employee:ID` in the employee list.

The key lookup model is beneficial because it means that the database has a consistent query pattern—the entire workload consists of key lookups whose performance is relatively uniform and predictable. Profiling to find the slow parts of an application is simpler, since all complex operations reside in the application code. On the flip side, the data model logic and business logic are now more closely intertwined, which muddles abstraction.

Let's quickly touch on the data associated with each key. Various NoSQL systems offer different solutions in this space.

Key-Value Stores

The simplest form of NoSQL store is a *key-value* store. Each key is mapped to a value containing arbitrary data. The NoSQL store has no knowledge of the contents of its payload, and simply delivers the data to the application. In our Employee database example, one might map the key `employee:30` to a blob containing JSON or a binary format such as Protocol Buffers⁴, Thrift⁵, or Avro⁶ in order to encapsulate the information about employee 30.

If a developer uses structured formats to store complex data for a key, she must operate against the data in application space: a key-value data store generally offers no mechanisms for querying for keys based on some property of their values. Key-value stores shine in the simplicity of their query model, usually consisting of `set`, `get`, and `delete` primitives, but discard the ability to add simple in-database filtering capabilities due to the opacity of their values. Voldemort, which is based on Amazon's Dynamo, provides a distributed key-value store. BDB⁷ offers a persistence library that has a key-value interface.

Key-Data Structure Stores

Key-data structure stores, made popular by Redis⁸, assign each value a type. In Redis, the available types a value can take on are integer, string, list, set, and sorted set. In addition to `set/get/delete`, type-specific commands, such as `increment/decrement` for integers, or `push/pop` for lists, add functionality to the query model without drastically affecting performance characteristics of requests. By providing simple type-specific functionality while avoiding multi-key operations such as aggregation or joins, Redis balances functionality and performance.

Key-Document Stores

Key-document stores, such as CouchDB⁹, MongoDB¹⁰, and Riak¹¹, map a key to some document that contains structured information. These systems store documents in a JSON or JSON-like format. They store lists and dictionaries, which can be embedded recursively inside one-another.

⁴<http://code.google.com/p/protobuf/>

⁵<http://thrift.apache.org/>

⁶<http://avro.apache.org/>

⁷<http://www.oracle.com/technetwork/database/berkeleydb/overview/index.html>

⁸<http://redis.io/>

⁹<http://couchdb.apache.org/>

¹⁰<http://www.mongodb.org/>

¹¹http://www.basho.com/products_riak_overview.php

MongoDB separates the keyspace into collections, so that keys for Employees and Department, for example, do not collide. CouchDB and Riak leave type-tracking to the developer. The freedom and complexity of document stores is a double-edged sword: application developers have a lot of freedom in modeling their documents, but application-based query logic can become exceedingly complex.

BigTable Column Family Stores

HBase and Cassandra base their data model on the one used by Google's BigTable. In this model, a key identifies a row, which contains data stored in one or more Column Families (CFs). Within a CF, each row can contain multiple columns. The values within each column are timestamped, so that several versions of a row-column mapping can live within a CF.

Conceptually, one can think of Column Families as storing complex keys of the form (row ID, CF, column, timestamp), mapping to values which are sorted by their keys. This design results in data modeling decisions which push a lot of functionality into the keyspace. It is particularly good at modeling historical data with timestamps. The model naturally supports sparse column placement since row IDs that do not have certain columns do not need an explicit NULL value for those columns. On the flip side, columns which have few or no NULL values must still store the column identifier with each row, which leads to greater space consumption.

Each project data model differs from the original BigTable model in various ways, but Cassandra's changes are most notable. Cassandra introduces the notion of a supercolumn within each CF to allow for another level of mapping, modeling, and indexing. It also does away with a notion of locality groups, which can physically store multiple column families together for performance reasons.

Graph Storage

One class of NoSQL stores are graph stores. Not all data is created equal, and the relational and key-oriented data models of storing and querying data are not the best for all data. Graphs are a fundamental data structure in computer science, and systems such as HyperGraphDB¹² and Neo4J¹³ are two popular NoSQL storage systems for storing graph-structured data. Graph stores differ from the other stores we have discussed thus far in almost every way: data models, data traversal and querying patterns, physical layout of data on disk, distribution to multiple machines, and the transactional semantics of queries. We can not do these stark differences justice given space limitations, but you should be aware that certain classes of data may be better stored and queried as a graph.

Complex Queries

There are notable exceptions to key-only lookups in NoSQL systems. MongoDB allows you to index your data based on any number of properties and has a relatively high-level language for specifying which data you want to retrieve. BigTable-based systems support scanners to iterate over a column family and select particular items by a filter on a column. CouchDB allows you to create different views of the data, and to run MapReduce tasks across your table to facilitate more complex lookups and updates. Most of the systems have bindings to Hadoop or another MapReduce framework to perform dataset-scale analytical queries.

¹²<http://www.hypergraphdb.org/index>

¹³<http://neo4j.org/>

Transactions

NoSQL systems generally prioritize performance over *transactional semantics*. Other SQL-based systems allow any set of statements—from a simple primary key row retrieval, to a complicated join between several tables which is then subsequently averaged across several fields—to be placed in a transaction.

These SQL databases will offer ACID guarantees between transactions. Running multiple operations in a transaction is Atomic (the A in ACID), meaning all or none of the operations happen. Consistency (the C) ensures that the transaction leaves the database in a consistent, uncorrupted state. Isolation (the I) makes sure that if two transactions touch the same record, they will do without stepping on each other's feet. Durability (the D, covered extensively in the next section), ensures that once a transaction is committed, it's stored in a safe place.

ACID-compliant transactions keep developers sane by making it easy to reason about the state of their data. Imagine multiple transactions, each of which has multiple steps (e.g., first check the value of a bank account, then subtract \$60, then update the value). ACID-compliant databases often are limited in how they can interleave these steps while still providing a correct result across all transactions. This push for correctness results in often-unexpected performance characteristics, where a slow transaction might cause an otherwise quick one to wait in line.

Most NoSQL systems pick performance over full ACID guarantees, but do provide guarantees at the key level: two operations on the same key will be serialized, avoiding serious corruption to key-value pairs. For many applications, this decision will not pose noticeable correctness issues, and will allow quick operations to execute with more regularity. It does, however, leave more considerations for application design and correctness in the hands of the developer.

Redis is the notable exception to the no-transaction trend. On a single server, it provides a MULTI command to combine multiple operations atomically and consistently, and a WATCH command to allow isolation. Other systems provide lower-level *test-and-set* functionality which provides some isolation guarantees.

Schema-free Storage

A cross-cutting property of many NoSQL systems is the lack of schema enforcement in the database. Even in document stores and column family-oriented stores, properties across similar entities are not required to be the same. This has the benefit of supporting less structured data requirements and requiring less performance expense when modifying schemas on-the-fly. The decision leaves more responsibility to the application developer, who now has to program more defensively. For example, is the lack of a `lastname` property on an employee record an error to be rectified, or a schema update which is currently propagating through the system? Data and schema versioning is common in application-level code after a few iterations of a project which relies on *sloppy-schema* NoSQL systems.

13.3 Data Durability

Ideally, all data modifications on a storage system would immediately be safely persisted and replicated to multiple locations to avoid data loss. However, ensuring data safety is in tension with performance, and different NoSQL systems make different *data durability* guarantees in order to improve performance. Failure scenarios are varied and numerous, and not all NoSQL systems protect you against these issues.

A simple and common failure scenario is a server restart or power loss. Data durability in this case involves having moved the data from memory to a hard disk, which does not require power to store data. Hard disk failure is handled by copying the data to secondary devices, be they other hard drives in the same machine (RAID mirroring) or other machines on the network. However, a data center might not survive an event which causes correlated failure (a tornado, for example), and some organizations go so far as to copy data to backups in data centers several hurricane widths apart. Writing to hard drives and copying data to multiple servers or data centers is expensive, so different NoSQL systems trade off durability guarantees for performance.

Single-server Durability

The simplest form of durability is a *single-server durability*, which ensures that any data modification will survive a server restart or power loss. This usually means writing the changed data to disk, which often bottlenecks your workload. Even if you order your operating system to write data to an on-disk file, the operating system may buffer the write, avoiding an immediate modification on disk so that it can group several writes together into a single operation. Only when the `fsync` system call is issued does the operating system make a best-effort attempt to ensure that buffered updates are persisted to disk.

Typical hard drives can perform 100-200 random accesses (seeks) per second, and are limited to 30-100 MB/sec of sequential writes. Memory can be orders of magnitudes faster in both scenarios. Ensuring efficient single-server durability means limiting the number of random writes your system incurs, and increasing the number of sequential writes per hard drive. Ideally, you want a system to minimize the number of writes between `fsync` calls, maximizing the number of those writes that are sequential, all the while never telling the user their data has been successfully written to disk until that write has been `fsync`d. Let's cover a few techniques for improving performance of single-server durability guarantees.

Control `fsync` Frequency

Memcached¹⁴ is an example of a system which offers no on-disk durability in exchange for extremely fast in-memory operations. When a server restarts, the data on that server is gone: this makes for a good cache and a poor durable data store.

Redis offers developers several options for when to call `fsync`. Developers can force an `fsync` call after every update, which is the slow and safe choice. For better performance, Redis can `fsync` its writes every `N` seconds. In a worst-case scenario, the you will lose last `N` seconds worth of operations, which may be acceptable for certain uses. Finally, for use cases where durability is not important (maintaining coarse-grained statistics, or using Redis as a cache), the developer can turn off `fsync` calls entirely: the operating system will eventually flush the data to disk, but without guarantees of when this will happen.

Increase Sequential Writes by Logging

Several data structures, such as B+Trees, help NoSQL systems quickly retrieve data from disk. Updates to those structures result in updates in random locations in the data structures' files, resulting in several random writes per update if you `fsync` after each update. To reduce random writes, systems such as Cassandra, HBase, Redis, and Riak append update operations to a sequentially-written file

¹⁴<http://memcached.org/>

called a *log*. While other data structures used by the system are only periodically fsynced, the log is frequently fsynced. By treating the log as the ground-truth state of the database after a crash, these storage engines are able to turn random updates into sequential ones.

While NoSQL systems such as MongoDB perform writes in-place in their data structures, others take logging even further. Cassandra and HBase use a technique borrowed from BigTable of combining their logs and lookup data structures into one *log-structured merge tree*. Riak provides similar functionality with a *log-structured hash table*. CouchDB has modified the traditional B+Tree so that all changes to the data structure are appended to the structure on physical storage. These techniques result in improved write throughput, but require a periodic log compaction to keep the log from growing unbounded.

Increase Throughput by Grouping Writes

Cassandra groups multiple concurrent updates within a short window into a single fsync call. This design, called *group commit*, results in higher latency per update, as users have to wait on several concurrent updates to have their own update be acknowledged. The latency bump comes at an increase in throughput, as multiple log appends can happen with a single fsync. As of this writing, every HBase update is persisted to the underlying storage provided by the Hadoop Distributed File System (HDFS)¹⁵, which has recently seen patches to allow support of appends that respect fsync and group commit.

Multi-server Durability

Because hard drives and machines often irreparably fail, copying important data across machines is necessary. Many NoSQL systems offer multi-server durability for data.

Redis takes a traditional master-slave approach to replicating data. All operations executed against a master are communicated in a log-like fashion to slave machines, which replicate the operations on their own hardware. If a master fails, a slave can step in and serve the data from the state of the operation log that it received from the master. This configuration might result in some data loss, as the master does not confirm that the slave has persisted an operation in its log before acknowledging the operation to the user. CouchDB facilitates a similar form of directional replication, where servers can be configured to replicate changes to documents on other stores.

MongoDB provides the notion of replica sets, where some number of servers are responsible for storing each document. MongoDB gives developers the option of ensuring that all replicas have received updates, or to proceed without ensuring that replicas have the most recent data. Many of the other distributed NoSQL storage systems support multi-server replication of data. HBase, which is built on top of HDFS, receives multi-server durability through HDFS. All writes are replicated to two or more HDFS nodes before returning control to the user, ensuring multi-server durability.

Riak, Cassandra, and Voldemort support more configurable forms of replication. With subtle differences, all three systems allow the user to specify N , the number of machines which should ultimately have a copy of the data, and $W < N$, the number of machines that should confirm the data has been written before returning control to the user.

To handle cases where an entire data center goes out of service, multi-server replication across data centers is required. Cassandra, HBase, and Voldemort have *rack-aware* configurations, which specify the rack or data center in which various machines are located. In general, blocking the user's

¹⁵<http://hadoop.apache.org/hdfs/>

request until a remote server has acknowledged an update incurs too much latency. Updates are streamed without confirmation when performed across wide area networks to backup data centers.

13.4 Scaling for Performance

Having just spoken about handling failure, let's imagine a rosier situation: success! If the system you build reaches success, your data store will be one of the components to feel stress under load. A cheap and dirty solution to such problems is to *scale up* your existing machinery: invest in more RAM and disks to handle the workload on one machine. With more success, pouring money into more expensive hardware will become infeasible. At this point, you will have to replicate data and spread requests across multiple machines to distribute load. This approach is called *scale out*, and is measured by the *horizontal scalability* of your system.

The ideal horizontal scalability goal is *linear scalability*, in which doubling the number of machines in your storage system doubles the query capacity of the system. The key to such scalability is in how the data is spread across machines. Sharding is the act of splitting your read and write workload across multiple machines to scale out your storage system. Sharding is fundamental to the design of many systems, namely Cassandra, HBase, Voldemort, and Riak, and more recently MongoDB and Redis. Some projects such as CouchDB focus on single-server performance and do not provide an in-system solution to sharding, but secondary projects provide coordinators to partition the workload across independent installations on multiple machines.

Let's cover a few interchangeable terms you might encounter. We will use the terms *sharding* and *partitioning* interchangeably. The terms *machine*, *server*, or *node* refer to some physical computer which stores part of the partitioned data. Finally, a *cluster* or *ring* refers to the set of machines which participate in your storage system.

Sharding means that no one machine has to handle the write workload on the entire dataset, but no one machine can answer queries about the entire dataset. Most NoSQL systems are key-oriented in both their data and query models, and few queries touch the entire dataset anyway. Because the primary access method for data in these systems is key-based, sharding is typically key-based as well: some function of the key determines the machine on which a key-value pair is stored. We'll cover two methods of defining the key-machine mapping: hash partitioning and range partitioning.

Do Not Shard Until You Have To

Sharding adds system complexity, and where possible, you should avoid it. Let's cover two ways to scale without sharding: read replicas and caching.

Read Replicas

Many storage systems see more read requests than write requests. A simple solution in these cases is to make copies of the data on multiple machines. All write requests still go to a master node. Read requests go to machines which replicate the data, and are often slightly stale with respect to the data on the write master.

If you are already replicating your data for multi-server durability in a master-slave configuration, as is common in Redis, CouchDB, or MongoDB, the read slaves can shed some load from the write master. Some queries, such as aggregate summaries of your dataset, which might be expensive and often do not require up-to-the-second freshness, can be executed against the slave replicas. Generally,

the less stringent your demands for freshness of content, the more you can lean on read slaves to improve read-only query performance.

Caching

Caching the most popular content in your system often works surprisingly well. Memcached dedicates blocks of memory on multiple servers to cache data from your data store. Memcached clients take advantage of several horizontal scalability tricks to distribute load across Memcached installations on different servers. To add memory to the cache pool, just add another Memcached host.

Because Memcached is designed for caching, it does not have as much architectural complexity as the persistent solutions for scaling workloads. Before considering more complicated solutions, think about whether caching can solve your scalability woes. Caching is not solely a temporary band-aid: Facebook has Memcached installations in the range of tens of terabytes of memory!

Read replicas and caching allow you to scale up your read-heavy workloads. When you start to increase the frequency of writes and updates to your data, however, you will also increase the load on the master server that contains all of your up-to-date data. For the rest of this section, we will cover techniques for sharding your write workload across multiple servers.

Sharding Through Coordinators

The CouchDB project focuses on the single-server experience. Two projects, Lounge and BigCouch, facilitate sharding CouchDB workloads through an external proxy, which acts as a front end to standalone CouchDB instances. In this design, the standalone installations are not aware of each other. The coordinator distributes requests to individual CouchDB instances based on the key of the document being requested.

Twitter has built the notions of sharding and replication into a coordinating framework called Gizzard¹⁶. Gizzard takes standalone data stores of any type—you can build wrappers for SQL or NoSQL storage systems—and arranges them in trees of any depth to partition keys by key range. For fault tolerance, Gizzard can be configured to replicate data to multiple physical machines for the same key range.

Consistent Hash Rings

Good hash functions distribute a set of keys in a uniform manner. This makes them a powerful tool for distributing key-value pairs among multiple servers. The academic literature on a technique called *consistent hashing* is extensive, and the first applications of the technique to data stores was in systems called *distributed hash tables (DHTs)*. NoSQL systems built around the principles of Amazon's Dynamo adopted this distribution technique, and it appears in Cassandra, Voldemort, and Riak.

Hash Rings by Example

Consistent hash rings work as follows. Say we have a hash function H that maps keys to uniformly distributed large integer values. We can form a ring of numbers in the range $[1, L]$ that wraps around itself with these values by taking $H(\text{key}) \bmod L$ for some relatively large integer L . This will map each key into the range $[1, L]$. A consistent hash ring of servers is formed by taking each server's

¹⁶<http://github.com/twitter/gizzard>

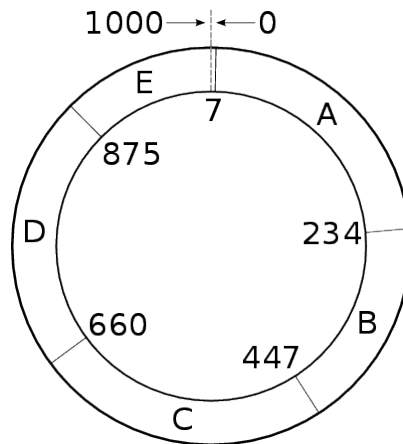


Figure 13.1: A Distributed Hash Table Ring

unique identifier (say its IP address), and applying H to it. You can get an intuition for how this works by looking at the hash ring formed by five servers (A-E) in Figure 13.1.

There, we picked $L = 1000$. Let's say that $H(A) \bmod L = 7$, $H(B) \bmod L = 234$, $H(C) \bmod L = 447$, $H(D) \bmod L = 660$, and $H(E) \bmod L = 875$. We can now tell which server a key should live on. To do this, we map all keys to a server by seeing if it falls in the range between that server and the next one in the ring. For example, A is responsible for keys whose hash value falls in the range $[7, 233]$, and E is responsible for keys in the range $[875, 6]$ (this range wraps around on itself at 1000). So if $H(\text{'employee30'}) \bmod L = 899$, it will be stored by server E, and if $H(\text{'employee31'}) \bmod L = 234$, it will be stored on server B.

Replicating Data

Replication for multi-server durability is achieved by passing the keys and values in one server's assigned range to the servers following it in the ring. For example, with a replication factor of 3, keys mapped to the range $[7, 233]$ will be stored on servers A, B, and C. If A were to fail, its neighbors B and C would take over its workload. In some designs, E would replicate and take over A's workload temporarily, since its range would expand to include A's.

Achieving Better Distribution

While hashing is statistically effective at uniformly distributing a keyspace, it usually requires many servers before it distributes evenly. Unfortunately, we often start with a small number of servers that are not perfectly spaced apart from one-another by the hash function. In our example, A's key range is of length 227, whereas E's range is 132. This leads to uneven load on different servers. It also makes it difficult for servers to take over for one-another when they fail, since a neighbor suddenly has to take control of the entire range of the failed server.

To solve the problem of uneven large key ranges, many DHTs including Riak create several 'virtual' nodes per physical machine. For example, with 4 virtual nodes, server A will act as server A_1, A_2, A_3, and A_4. Each virtual node hashes to a different value, giving it more opportunity to manage keys distributed to different parts of the keyspace. Voldemort takes a similar approach, in

which the number of partitions is manually configured and usually larger than the number of servers, resulting in each server receiving a number of smaller partitions.

Cassandra does not assign multiple small partitions to each server, resulting in sometimes uneven key range distributions. For load-balancing, Cassandra has an asynchronous process which adjusts the location of servers on the ring depending on their historic load.

Range Partitioning

In the range partitioning approach to sharding, some machines in your system keep metadata about which servers contain which key ranges. This metadata is consulted to route key and range lookups to the appropriate servers. Like the consistent hash ring approach, this range partitioning splits the keyspace into ranges, with each key range being managed by one machine and potentially replicated to others. Unlike the consistent hashing approach, two keys that are next to each other in the key's sort order are likely to appear in the same partition. This reduces the size of the routing metadata, as large ranges are compressed to [start, end] markers.

In adding active record-keeping of the *range-to-server* mapping, the range partitioning approach allows for more fine-grained control of load-shedding from heavily loaded servers. If a specific key range sees higher traffic than other ranges, a load manager can reduce the size of the range on that server, or reduce the number of shards that this server serves. The added freedom to actively manage load comes at the expense of extra architectural components which monitor and route shards.

The BigTable Way

Google's BigTable paper describes a range-partitioning hierarchical technique for sharding data into tablets. A tablet stores a range of row keys and values within a column family. It maintains all of the necessary logs and data structures to answer queries about the keys in its assigned range. Tablet servers serve multiple tablets depending on the load each tablet is experiencing.

Each tablet is kept at a size of 100-200 MB. As tablets change in size, two small tablets with adjoining key ranges might be combined, or a large tablet might be split in two. A master server analyzes tablet size, load, and tablet server availability. The master adjusts which tablet server serves which tablets at any time.

The master server maintains the tablet assignment in a metadata table. Because this metadata can get large, the metadata table is also sharded into tablets that map key ranges to tablets and tablet servers responsible for those ranges. This results in a three-layer hierarchy traversal for clients to find a key on its hosting tablet server, as depicted in Figure 13.2.

Let's look at an example. A client searching for key 900 will query server A, which stores the tablet for metadata level 0. This tablet identifies the metadata level 1 tablet on server 6 containing key ranges 500-1500. The client sends a request to server B with this key, which responds that the tablet containing keys 850-950 is found on a tablet on server C. Finally, the client sends the key request to server C, and gets the row data back for its query. Metadata tablets at level 0 and 1 may be cached by the client, which avoids putting undue load on their tablet servers from repeat queries. The BigTable paper explains that this 3-level hierarchy can accommodate 2^{61} bytes worth of storage using 128MB tablets.

Handling Failures

The master is a single point of failure in the BigTable design, but can go down temporarily without affecting requests to tablet servers. If a tablet server fails while serving tablet requests, it is up to the master to recognize this and re-assign its tablets while requests temporarily fail.

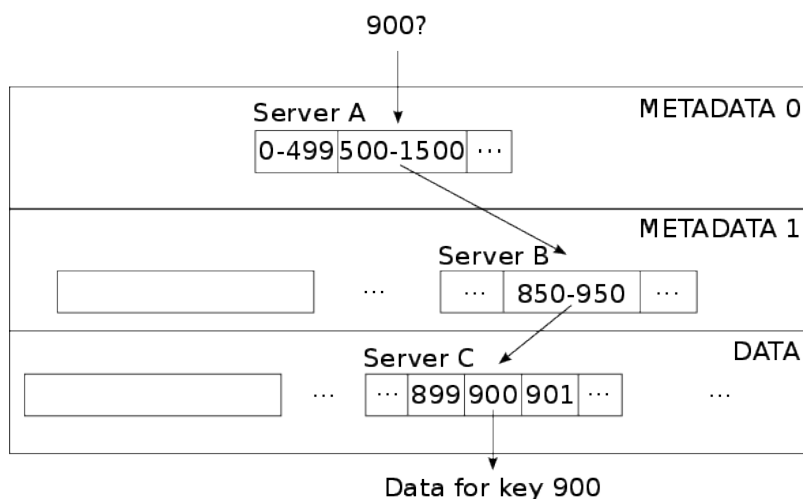


Figure 13.2: BigTable-based Range Partitioning

In order to recognize and handle machine failures, the BigTable paper describes the use of Chubby, a distributed locking system for managing server membership and liveness. ZooKeeper¹⁷ is the open source implementation of Chubby, and several Hadoop-based projects utilize it to manage secondary master servers and tablet server reassignment.

Range Partitioning-based NoSQL Projects

HBase employs BigTable's hierarchical approach to range-partitioning. Underlying tablet data is stored in Hadoop's distributed filesystem (HDFS). HDFS handles data replication and consistency among replicas, leaving tablet servers to handle requests, update storage structures, and initiate tablet splits and compactions.

MongoDB handles range partitioning in a manner similar to that of BigTable. Several configuration nodes store and manage the routing tables that specify which storage node is responsible for which key ranges. These configuration nodes stay in sync through a protocol called *two-phase commit*, and serve as a hybrid of BigTable's master for specifying ranges and Chubby for highly available configuration management. Separate routing processes, which are stateless, keep track of the most recent routing configuration and route key requests to the appropriate storage nodes. Storage nodes are arranged in replica sets to handle replication.

Cassandra provides an order-preserving partitioner if you wish to allow fast range scans over your data. Cassandra nodes are still arranged in a ring using consistent hashing, but rather than hashing a key-value pair onto the ring to determine the server to which it should be assigned, the key is simply mapped onto the server which controls the range in which the key naturally fits. For example, keys 20 and 21 would both be mapped to server A in our consistent hash ring in Figure 13.1, rather than being hashed and randomly distributed in the ring.

Twitter's Gizzard framework for managing partitioned and replicated data across many back ends uses range partitioning to shard data. Routing servers form hierarchies of any depth, assigning ranges of keys to servers below them in the hierarchy. These servers either store data for keys in

¹⁷<http://hadoop.apache.org/zookeeper/>

their assigned range, or route to yet another layer of routing servers. Replication in this model is achieved by sending updates to multiple machines for a key range. Gizzard routing nodes manage failed writes in different manner than other NoSQL systems. Gizzard requires that system designers make all updates idempotent (they can be run twice). When a storage node fails, routing nodes cache and repeatedly send updates to the node until the update is confirmed.

Which Partitioning Scheme to Use

Given the hash- and range-based approaches to sharding, which is preferable? It depends. Range partitioning is the obvious choice to use when you will frequently be performing range scans over the keys of your data. As you read values in order by key, you will not jump to random nodes in the network, which would incur heavy network overhead. But if you do not require range scans, which sharding scheme should you use?

Hash partitioning gives reasonable distribution of data across nodes, and random skew can be reduced with virtual nodes. Routing is simple in the hash partitioning scheme: for the most part, the hash function can be executed by clients to find the appropriate server. With more complicated rebalancing schemes, finding the right node for a key becomes more difficult.

Range partitioning requires the upfront cost of maintaining routing and configuration nodes, which can see heavy load and become central points of failure in the absence of relatively complex fault tolerance schemes. Done well, however, range-partitioned data can be load-balanced in small chunks which can be reassigned in high-load situations. If a server goes down, its assigned ranges can be distributed to many servers, rather than loading the server's immediate neighbors during downtime.

13.5 Consistency

Having spoken about the virtues of replicating data to multiple machines for durability and spreading load, it's time to let you in on a secret: keeping replicas of your data on multiple machines consistent with one-another is hard. In practice, replicas will crash and get out of sync, replicas will crash and never come back, networks will partition two sets of replicas, and messages between machines will get delayed or lost. There are two major approaches to data consistency in the NoSQL ecosystem. The first is strong consistency, where all replicas remain in sync. The second is eventual consistency, where replicas are allowed to get out of sync, but eventually catch up with one-another. Let's first get into why the second option is an appropriate consideration by understanding a fundamental property of distributed computing. After that, we'll jump into the details of each approach.

A Little Bit About CAP

Why are we considering anything short of strong consistency guarantees over our data? It all comes down to a property of distributed systems architected for modern networking equipment. The idea was first proposed by Eric Brewer as the *CAP Theorem*, and later proved by Gilbert and Lynch [GL02]. The theorem first presents three properties of distributed systems which make up the acronym CAP:

Consistency: do all replicas of a piece of data always logically agree on the same version of that data by the time you read it? (This concept of consistency is different than the C in ACID.)

Availability: Do replicas respond to read and write requests regardless of how many replicas are inaccessible?

Partition tolerance: Can the system continue to operate even if some replicas temporarily lose the ability to communicate with each other over the network?

The theorem then goes on to say that a storage system which operates on multiple computers can only achieve two of these properties at the expense of a third. Also, we are forced to implement partition-tolerant systems. On current networking hardware using current messaging protocols, packets can be lost, switches can fail, and there is no way to know whether the network is down or the server you are trying to send a message to is unavailable. All NoSQL systems should be partition-tolerant. The remaining choice is between consistency and availability. No NoSQL system can provide both at the same time.

Opting for consistency means that your replicated data will not be out of sync across replicas. An easy way to achieve consistency is to require that all replicas acknowledge updates. If a replica goes down and you can not confirm data updates on it, then you degrade availability on its keys. This means that until all replicas recover and respond, the user can not receive successful acknowledgment of their update operation. Thus, opting for consistency is opting for a lack of round-the-clock availability for each data item.

Opting for availability means that when a user issues an operation, replicas should act on the data they have, regardless of the state of other replicas. This may lead to diverging consistency of data across replicas, since they weren't required to acknowledge all updates, and some replicas may have not noted all updates.

The implications of the CAP theorem lead to the strong consistency and eventual consistency approaches to building NoSQL data stores. Other approaches exist, such as the relaxed consistency and relaxed availability approach presented in Yahoo!'s PNUTS [CRS⁺08] system. None of the open source NoSQL systems we discuss has adopted this technique yet, so we will not discuss it further.

Strong Consistency

Systems which promote strong consistency ensure that the replicas of a data item will always be able to come to consensus on the value of a key. Some replicas may be out of sync with one-another, but when the user asks for the value of `employee30:salary`, the machines have a way to consistently agree on the value the user sees. How this works is best explained with numbers.

Say we replicate a key on N machines. Some machine, perhaps one of the N , serves as a coordinator for each user request. The coordinator ensures that a certain number of the N machines has received and acknowledged each request. When a write or update occurs to a key, the coordinator does not confirm with the user that the write occurred until W replicas confirm that they have received the update. When a user wants to read the value for some key, the coordinator responds when at least R have responded with the same value. We say that the system exemplifies strong consistency if $R+W>N$.

Putting some numbers to this idea, let's say that we're replicating each key across $N=3$ machines (call them A, B, and C). Say that the key `employee30:salary` is initially set to the value \$20,000, but we want to give `employee30` a raise to \$30,000. Let's require that at least $W=2$ of A, B, or C acknowledge each write request for a key. When A and B confirm the write request for (`employee30:salary`, \$30,000), the coordinator lets the user know that `employee30:salary` is safely updated. Let's assume that machine C never received the write request for `employee30:salary`, so it still has the value \$20,000. When a coordinator gets a read request for key `employee30:salary`, it will send that request to all 3 machines:

- If we set $R=1$, and machine C responds first with \$20,000, our employee will not be very happy.
- However, if we set $R=2$, the coordinator will see the value from C, wait for a second response from A or B, which will conflict with C's outdated value, and finally receive a response from the third machine, which will confirm that \$30,000 is the majority opinion.

So in order to achieve strong consistency in this case, we need to set $R \geq 2$ so that $R+W > 3$.

What happens when W replicas do not respond to a write request, or R replicas do not respond to a read request with a consistent response? The coordinator can timeout eventually and send the user an error, or wait until the situation corrects itself. Either way, the system is considered unavailable for that request for at least some time.

Your choice of R and W affect how many machines can act strangely before your system becomes unavailable for different actions on a key. If you force all of your replicas to acknowledge writes, for example, then $W=N$, and write operations will hang or fail on any replica failure. A common choice is $R + W = N + 1$, the minimum required for strong consistency while still allowing for temporary disagreement between replicas. Many strong consistency systems opt for $W=N$ and $R=1$, since they then do not have to design for nodes going out of sync.

HBase bases its replicated storage on HDFS, a distributed storage layer. HDFS provides strong consistency guarantees. In HDFS, a write cannot succeed until it has been replicated to all N (usually 2 or 3) replicas, so $W = N$. A read will be satisfied by a single replica, so $R = 1$. To avoid bogging down write-intensive workloads, data is transferred from the user to the replicas asynchronously in parallel. Once all replicas acknowledge that they have received copies of the data, the final step of swapping the new data in to the system is performed atomically and consistently across all replicas.

Eventual Consistency

Dynamo-based systems, which include Voldemort, Cassandra, and Riak, allow the user to specify N , R , and W to their needs, even if $R + W \leq N$. This means that the user can achieve either strong or eventual consistency. When a user picks eventual consistency, and even when the programmer opts for strong consistency but W is less than N , there are periods in which replicas might not see eye-to-eye. To provide eventual consistency among replicas, these systems employ various tools to catch stale replicas up to speed. Let's first cover how various systems determine that data has gotten out of sync, then discuss how they synchronize replicas, and finally bring in a few dynamo-inspired methods for speeding up the synchronization process.

Versioning and Conflicts

Because two replicas might see two different versions of a value for some key, data versioning and conflict detection is important. The dynamo-based systems use a type of versioning called *vector clocks*. A vector clock is a vector assigned to each key which contains a counter for each replica. For example, if servers A, B, and C are the three replicas of some key, the vector clock will have three entries, (N_A, N_B, N_C) , initialized to $(0, 0, 0)$.

Each time a replica modifies a key, it increments its counter in the vector. If B modifies a key that previously had version $(39, 1, 5)$, it will change the vector clock to $(39, 2, 5)$. When another replica, say C, receives an update from B about the key's data, it will compare the vector clock from B to its own. As long as its own vector clock counters are all less than the ones delivered from B, then it has a stale version and can overwrite its own copy with B's. If B and C have clocks in which some counters are greater than others in both clocks, say $(39, 2, 5)$ and $(39, 1, 6)$, then the servers

recognize that they received different, potentially unreconcilable updates over time, and identify a conflict.

Conflict Resolution

Conflict resolution varies across the different systems. The Dynamo paper leaves conflict resolution to the application using the storage system. Two versions of a shopping cart can be merged into one without significant loss of data, but two versions of a collaboratively edited document might require human reviewer to resolve conflict. Voldemort follows this model, returning multiple copies of a key to the requesting client application upon conflict.

Cassandra, which stores a timestamp on each key, uses the most recently timestamped version of a key when two versions are in conflict. This removes the need for a round-trip to the client and simplifies the API. This design makes it difficult to handle situations where conflicted data can be intelligently merged, as in our shopping cart example, or when implementing distributed counters. Riak allows both of the approaches offered by Voldemort and Cassandra. CouchDB provides a hybrid: it identifies a conflict and allows users to query for conflicted keys for manual repair, but deterministically picks a version to return to users until conflicts are repaired.

Read Repair

If *R* replicas return non-conflicting data to a coordinator, the coordinator can safely return the non-conflicting data to the application. The coordinator may still notice that some of the replicas are out of sync. The Dynamo paper suggests, and Cassandra, Riak, and Voldemort implement, a technique called *read repair* for handling such situations. When a coordinator identifies a conflict on read, even if a consistent value has been returned to the user, the coordinator starts conflict-resolution protocols between conflicted replicas. This proactively fixes conflicts with little additional work. Replicas have already sent their version of the data to the coordinator, and faster conflict resolution will result in less divergence in the system.

Hinted Handoff

Cassandra, Riak, and Voldemort all employ a technique called *hinted handoff* to improve write performance for situations where a node temporarily becomes unavailable. If one of the replicas for a key does not respond to a write request, another node is selected to temporarily take over its write workload. Writes for the unavailable node are kept separately, and when the backup node notices the previously unavailable node become available, it forwards all of the writes to the newly available replica. The Dynamo paper utilizes a 'sloppy quorum' approach and allows the writes accomplished through hinted handoff to count toward the *W* required write acknowledgments. Cassandra and Voldemort will not count a hinted handoff against *W*, and will fail a write which does not have *W* confirmations from the originally assigned replicas. Hinted handoff is still useful in these systems, as it speeds up recovery when an unavailable node returns.

Anti-Entropy

When a replica is down for an extended period of time, or the machine storing hinted handoffs for an unavailable replica goes down as well, replicas must synchronize from one-another. In this case, Cassandra and Riak implement a Dynamo-inspired process called *anti-entropy*. In anti-entropy, replicas exchange *Merkle Trees* to identify parts of their replicated key ranges which are out of sync.

A Merkle tree is a hierarchical hash verification: if the hash over the entire keyspace is not the same between two replicas, they will exchange hashes of smaller and smaller portions of the replicated keyspace until the out-of-sync keys are identified. This approach reduces unnecessary data transfer between replicas which contain mostly similar data.

Gossip

Finally, as distributed systems grow, it is hard to keep track of how each node in the system is doing. The three Dynamo-based systems employ an age-old high school technique known as *gossip* to keep track of other nodes. Periodically (every second or so), a node will pick a random node it once communicated with to exchange knowledge of the health of the other nodes in the system. In providing this exchange, nodes learn which other nodes are down, and know where to route clients in search of a key.

13.6 A Final Word

The NoSQL ecosystem is still in its infancy, and many of the systems we've discussed will change architectures, designs, and interfaces. The important takeaways in this chapter are not what each NoSQL system currently does, but rather the design decisions that led to a combination of features that make up these systems. NoSQL leaves a lot of design work in the hands of the application designer. Understanding the architectural components of these systems will not only help you build the next great NoSQL amalgamation, but also allow you to use current versions responsibly.

13.7 Acknowledgments

I am grateful to Jackie Carter, Mihir Kedia, and the anonymous reviewers for their comments and suggestions to improve the chapter. This chapter would also not be possible without the years of dedicated work of the NoSQL community. Keep building!

Python Packaging

Tarek Ziadé

14.1 Introduction

There are two schools of thought when it comes to installing applications. The first, common to Windows and Mac OS X, is that applications should be self-contained, and their installation should not depend on anything else. This philosophy simplifies the management of applications: each application is its own standalone “appliance”, and installing and removing them should not disturb the rest of the OS. If the application needs an uncommon library, that library is included in the application’s distribution.

The second school, which is the norm for Linux-based systems, treats software as a collection of small self-contained units called *packages*. Libraries are bundled into packages, any given library package might depend on other packages. Installing an application might involve finding and installing particular versions of dozens of other libraries. These dependencies are usually fetched from a central repository that contains thousands of packages. This philosophy is why Linux distributions use complex package management systems like dpkg and RPM to track dependencies and prevent installation of two applications that use incompatible versions of the same library.

There are pros and cons to each approach. Having a highly modular system where every piece can be updated or replaced makes management easier, because each library is present in a single place, and all applications that use it benefit when it is updated. For instance, a security fix in a particular library will reach all applications that use it at once, whereas if an application ships with its own library, that security fix will be more complex to deploy, especially if different applications use different versions of the library.

But that modularity is seen as a drawback by some developers, because they’re not in control of their applications and dependencies. It is easier for them to provide a standalone software appliance to be sure that the application environment is stable and not subject to “dependency hell” during system upgrades.

Self-contained applications also make the developer’s life easier when she needs to support several operating systems. Some projects go so far as to release portable applications that remove *any* interaction with the hosting system by working in a self-contained directory, even for log files.

Python’s packaging system was intended to make the second philosophy—multiple dependencies for each install—as developer-, admin-, packager-, and user-friendly as possible. Unfortunately it had (and has) a variety of flaws which caused or allowed all kinds of problems: unintuitive version

schemes, mishandled data files, difficulty re-packaging, and more. Three years ago I and a group of other Pythoners decided to reinvent it to address these problems. We call ourselves the Fellowship of the Packaging, and this chapter describes the problems we have been trying to fix, and what our solution looks like.

Terminology

In Python a *package* is a directory containing Python files. Python files are called *modules*. That definition makes the usage of the word “package” a bit vague since it is also used by many systems to refer to a *release* of a project.

Python developers themselves are sometimes vague about this. One way to remove this ambiguity is to use the term “Python packages” when we talk about a directory containing Python modules. The term “release” is used to define one version of a project, and the term “distribution” defines a source or a binary distribution of a release as something like a tarball or zip file.

14.2 The Burden of the Python Developer

Most Python programmers want their programs to be usable in any environment. They also usually want to use a mix of standard Python libraries and system-dependent libraries. But unless you package your application separately for every existing packaging system, you are doomed to provide Python-specific releases—a Python-specific release is a release aimed to be installed within a Python installation no matter what the underlying Operating System is—and hope that:

- packagers for every target system will be able to repackage your work,
- the dependencies you have will themselves be repackaged in every target system, and
- system dependencies will be clearly described.

Sometimes, this is simply impossible. For example, Plone (a full-fledged Python-powered CMS) uses hundreds of small pure Python libraries that are not always available as packages in every packaging system out there. This means that Plone *must* ship everything that it needs in a portable application. To do this, it uses `zc.buildout`, which collects all its dependencies and creates a portable application that will run on any system within a single directory. It is effectively a binary release, since any piece of C code will be compiled in place.

This is a big win for developers: they just have to describe their dependencies using the Python standards described below and use `zc.buildout` to release their application. But as discussed earlier, this type of release sets up a fortress within the system, which most Linux sysadmins will hate. Windows admins won’t mind, but those managing CentOS or Debian will, because those systems base their management on the assumption that every file in the system is registered, classified, and known to admin tools.

Those admins will want to repackage your application according to their own standards. The question we need to answer is, “Can Python have a packaging system that can be automatically translated into other packaging systems?” If so, one application or library can be installed on any system without requiring extra packaging work. Here, “automatically” doesn’t necessarily mean that the work should be fully done by a script: RPM or dpkg packagers will tell you that’s impossible—they

always need to add some specifics in the projects they repackage. They'll also tell you that they often have a hard time re-packaging a piece of code because its developers were not aware of a few basic packaging rules.

Here's one example of what you can do to annoy packagers using the existing Python packaging system: release a library called "MathUtils" with the version name "Fumanchu". The brilliant mathematician who wrote the library have found it amusing to use his cats' names for his project versions. But how can a packager know that "Fumanchu" is his second cat's name, and that the first one was called "Phil", so that the "Fumanchu" version comes after the "Phil" one?

This may sound extreme, but it can happen with today's tools and standards. The worst thing is that tools like `easy_install` or `pip` use their own non-standard registry to keep track of installed files, and will sort the "Fumanchu" and "Phil" versions alphanumerically.

Another problem is how to handle data files. For example, what if your application uses an SQLite database? If you put it inside your package directory, your application might fail because the system forbids you to write in that part of the tree. Doing this will also compromise the assumptions Linux systems make about where application data is for backups (`/var`).

In the real world, system administrators need to be able to place your files where they want without breaking your application, and you need to tell them what those files are. So let's rephrase the question: is it possible to have a packaging system in Python that can provide all the information needed to repackage an application with any third-party packaging system out there without having to read the code, and make everyone happy?

14.3 The Current Architecture of Packaging

The `Distutils` package that comes with the Python standard library is riddled with the problems described above. Since it's the standard, people either live with it and its flaws, or use more advanced tools like `Setuptools`, which add features on the top of it, or `Distribute`, a fork of `Setuptools`. There's also `Pip`, a more advanced installer, that relies on `Setuptools`.

However, these newer tools are all based on `Distutils` and inherit its problems. Attempts were made to fix `Distutils` in place, but the code is so deeply used by other tools that any change to it, even its internals, is a potential regression in the whole Python packaging ecosystem.

We therefore decided to freeze `Distutils` and start the development of `Distutils2` from the same code base, without worrying too much about backward compatibility. To understand what changed and why, let's have a closer look at `Distutils`.

Distutils Basics and Design Flaws

`Distutils` contains commands, each of which is a class with a `run` method that can be called with some options. `Distutils` also provides a `Distribution` class that contains global values every command can look at.

To use `Distutils`, a developer adds a single Python module to a project, conventionally called `setup.py`. This module contains a call to `Distutils`' main entry point: the `setup` function. This function can take many options, which are held by a `Distribution` instance and used by commands. Here's an example that defines a few standard options like the name and version of the project, and a list of modules it contains:

```
from distutils.core import setup
```

```
setup(name='MyProject', version='1.0', py_modules=['mycode.py'])
```

This module can then be used to run Distutils commands like `sdist`, which creates a source distribution in an archive and places it in a `dist` directory:

```
$ python setup.py sdist
```

Using the same script, you can install the project using the `install` command:

```
$ python setup.py install
```

Distutils provides other commands such as:

- `upload` to upload a distribution into an online repository.
- `register` to register the metadata of a project in an online repository without necessarily uploading a distribution,
- `bdist` to create a binary distribution, and
- `bdist_msi` to create a `.msi` file for Windows.

It will also let you get information about the project via other command line options.

So installing a project or getting information about it is always done by invoking Distutils through this file. For example, to find out the name of the project:

```
$ python setup.py --name  
MyProject
```

`setup.py` is therefore how everyone interacts with the project, whether to build, package, publish, or install it. The developer describes the content of his project through options passed to a function, and uses that file for all his packaging tasks. The file is also used by installers to install the project on a target system.

Having a single Python module used for packaging, releasing, *and* installing a project is one of Distutils' main flaws. For example, if you want to get the name from the `lxml` project, `setup.py` will do a lot of things besides returning a simple string as expected:

```
$ python setup.py --name  
Building lxml version 2.2.  
NOTE: Trying to build without Cython, pre-generated 'src/lxml/lxml.etree.c'  
needs to be available.  
Using build configuration of libxslt 1.1.26  
Building against libxml2/libxslt in the following directory: /usr/lib/lxml
```

It might even fail to work on some projects, since developers make the assumption that `setup.py` is used only to install, and that other Distutils features are only used by them during development. The multiple roles of the `setup.py` script can easily cause confusion.

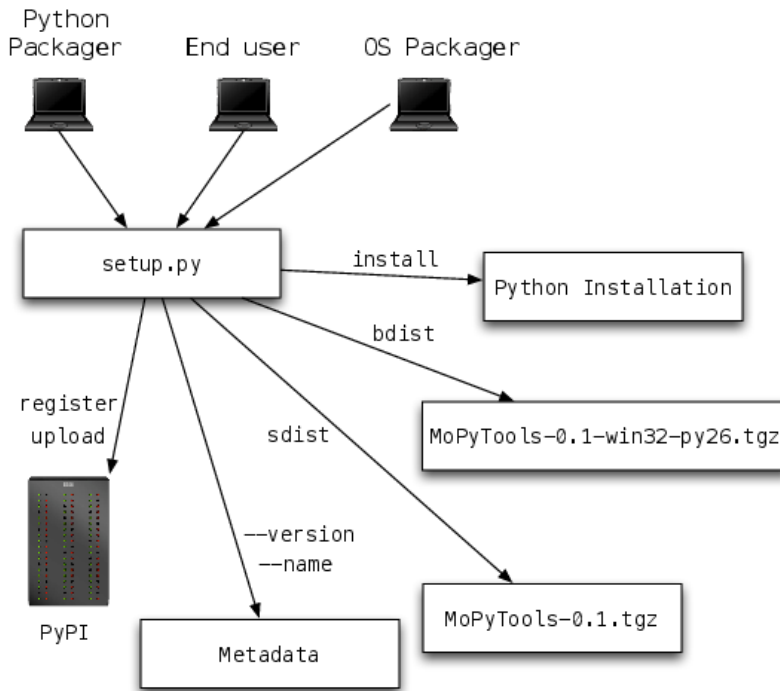


Figure 14.1: Setup

Metadata and PyPI

When Distutils builds a distribution, it creates a Metadata file that follows the standard described in PEP 314¹. It contains a static version of all the usual metadata, like the name of the project or the version of the release. The main metadata fields are:

- Name: The name of the project.
- Version: The version of the release.
- Summary: A one-line description.
- Description: A detailed description.
- Home-Page: The URL of the project.
- Author: The author name.
- Classifiers: Classifiers for the project. Python provides a list of classifiers for the license, the maturity of the release (beta, alpha, final), etc.
- Requires, Provides, and Obsoletes: Used to define dependencies with modules.

These fields are for the most part easy to map to equivalents in other packaging systems.

The Python Package Index (PyPI)², a central repository of packages like CPAN, is able to register projects and publish releases via Distutils' register and upload commands. register builds the Metadata file and sends it to PyPI, allowing people and tools—like installers—to browse them via web pages or via web services.

¹The Python Enhancement Proposals, or PEPs, that we refer to are summarized at the end of this chapter

²Formerly known as the CheeseShop.

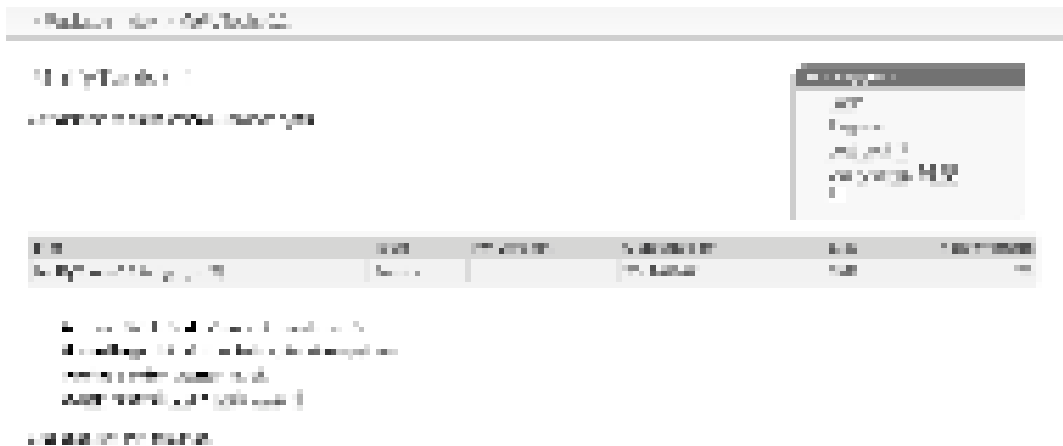


Figure 14.2: The PyPI Repository

You can browse projects by Classifiers, and get the author name and project URL. Meanwhile, Requires can be used to define dependencies on Python modules. The requires option can be used to add a Requires metadata element to the project:

```
from distutils.core import setup

setup(name='foo', version='1.0', requires=['ldap'])
```

Defining a dependency on the ldap module is purely declarative: no tools or installers ensure that such a module exists. This would be satisfactory if Python defined requirements at the module level through a require keyword like Perl does. Then it would just be a matter of the installers browsing the dependencies at PyPI and installing them; that's basically what CPAN does. But that's not possible in Python since a module named ldap can exist in any Python project. Since Distutils allows people to release projects that can contain several packages and modules, this metadata field is not useful at all.

Another flaw of Metadata files is that they are created by a Python script, so they are specific to the platform they are executed in. For example, a project that provides features specific to Windows could define its setup.py as:

```
from distutils.core import setup

setup(name='foo', version='1.0', requires=['win32com'])
```

But this assumes that the project only works under Windows, even if it provides portable features. One way to solve this is to make the requires option specific to Windows:

```
from distutils.core import setup
import sys
```

```

if sys.platform == 'win32':
    setup(name='foo', version='1.0', requires=['win32com'])
else:
    setup(name='foo', version='1.0')

```

This actually makes the issue worse. Remember, the script is used to build source archives that are then released to the world via PyPI. This means that the static Metadata file sent to PyPI is dependent on the platform that was used to compile it. In other words, there is no way to indicate statically in the metadata field that it is platform-specific.

Architecture of PyPI

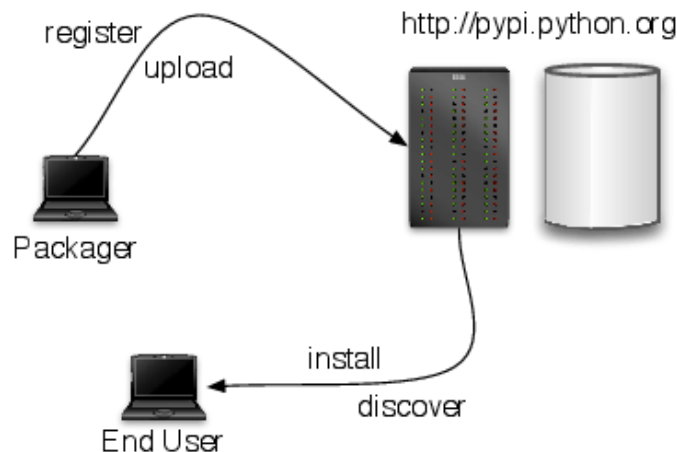


Figure 14.3: PyPI Workflow

As indicated earlier, PyPI is a central index of Python projects where people can browse existing projects by category or register their own work. Source or binary distributions can be uploaded and added to an existing project, and then downloaded for installation or study. PyPI also offers web services that can be used by tools like installers.

Registering Projects and Uploading Distributions

Registering a project to PyPI is done with the `Distutils` `register` command. It builds a POST request containing the metadata of the project, whatever its version is. The request requires an Authorization header, as PyPI uses Basic Authentication to make sure every registered project is associated with a user that has first registered with PyPI. Credentials are kept in the local `Distutils` configuration or typed in the prompt every time a `register` command is invoked. An example of its use is:

```

$ python setup.py register
running register
Registering MPTools to http://pypi.python.org/pypi
Server response (200): OK

```

Each registered project gets a web page with an HTML version of the metadata, and packagers can upload distributions to PyPI using upload:

```
$ python setup.py sdist upload
running sdist
...
running upload
Submitting dist/mopytools-0.1.tar.gz to http://pypi.python.org/pypi
Server response (200): OK
```

It's also possible to point users to another location via the Download-URL metadata field rather than uploading files directly to PyPI.

Querying PyPI

Besides the HTML pages PyPI publishes for web users, it provides two services that tools can use to browse the content: the Simple Index protocol and the XML-RPC APIs.

The Simple Index protocol starts at <http://pypi.python.org/simple/>, a plain HTML page that contains relative links to every registered project:

```
<html><head><title>Simple Index</title></head><body>
...
<a href='MontyLingua/'>MontyLingua</a><br/>
<a href='mootiro_web/'>mootiro_web</a><br/>
<a href='Mopidy/'>Mopidy</a><br/>
<a href='mopowg/'>mopowg</a><br/>
<a href='MOPPY/'>MOPPY</a><br/>
<a href='MPTools/'>MPTools</a><br/>
<a href='morbid/'>morbid</a><br/>
<a href='Morelia/'>Morelia</a><br/>
<a href='morse/'>morse</a><br/>
...
</body></html>
```

For example, the MPTools project has a MPTools/ link, which means that the project exists in the index. The site it points at contains a list of all the links related to the project:

- links for every distribution stored at PyPI
- links for every Home URL defined in the Metadata, for each version of the project registered
- links for every Download-URL defined in the Metadata, for each version as well.

The page for MPTools contains:

```
<html><head><title>Links for MPTools</title></head>
<body><h1>Links for MPTools</h1>
<a href=".../packages/source/M/MPTools/MPTools-0.1.tar.gz">MPTools-0.1.tar.gz</a><br/>
<a href="http://bitbucket.org/tarek/mopytools" rel="homepage">0.1 home_page</a><br/>
</body></html>
```

Tools like installers that want to find distributions of a project can look for it in the index page, or simply check if `http://pypi.python.org/simple/PROJECT_NAME/` exists.

This protocol has two main limitations. First, PyPI is a single server right now, and while people usually have local copies of its content, we have experienced several downtimes in the past two years that have paralyzed developers that are constantly working with installers that browse PyPI to get all the dependencies a project requires when it is built. For instance, building a Plone application will generate several hundreds queries at PyPI to get all the required bits, so PyPI may act as a single point of failure.

Second, when the distributions are not stored at PyPI and a Download-URL link is provided in the Simple Index page, installers have to follow that link and hope that the location will be up and will really contain the release. These indirections weakens any Simple Index-based process.

The Simple Index protocol's goal is to give to installers a list of links they can use to install a project. The project metadata is not published there; instead, there are XML-RPC methods to get extra information about registered projects:

```
>>> import xmlrpclib
>>> import pprint
>>> client = xmlrpclib.ServerProxy('http://pypi.python.org/pypi')
>>> client.package_releases('MPTools')
['0.1']
>>> pprint.pprint(client.release_urls('MPTools', '0.1'))
[{'comment_text': '',
  'downloads': 28,
  'filename': 'MPTools-0.1.tar.gz',
  'has_sig': False,
  'md5_digest': '6b06752d62c4bffe1fb65cd5c9b7111a',
  'packagetype': 'sdist',
  'python_version': 'source',
  'size': 3684,
  'upload_time': <DateTime '20110204T09:37:12' at f4da28>,
  'url': 'http://pypi.python.org/packages/source/M/MPTools/MPTools-0.1.tar.gz'}]
>>> pprint.pprint(client.release_data('MPTools', '0.1'))
{'author': 'Tarek Ziade',
 'author_email': 'tarek@mozilla.com',
 'classifiers': [],
 'description': 'UNKNOWN',
 'download_url': 'UNKNOWN',
 'home_page': 'http://bitbucket.org/tarek/mopytools',
 'keywords': None,
 'license': 'UNKNOWN',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'MPTools',
 'package_url': 'http://pypi.python.org/pypi/MPTools',
 'platform': 'UNKNOWN',
 'release_url': 'http://pypi.python.org/pypi/MPTools/0.1',
 'requires_python': None,
 'stable_version': None,
 'summary': 'Set of tools to build Mozilla Services apps',
 'version': '0.1'}
```

The issue with this approach is that some of the data that the XML-RPC APIs are publishing could have been stored as static files and published in the Simple Index page to simplify the work of client tools. That would also avoid the extra work PyPI has to do to handle those queries. It's fine to have non-static data like the number of downloads per distribution published in a specialized web service, but it does not make sense to have to use two different services to get all static data about a project.

Architecture of a Python Installation

If you install a Python project using `python setup.py install`, `Distutils`—which is included in the standard library—will copy the files onto your system.

- *Python packages* and modules will land in the Python directory that is loaded when the interpreter starts: under the latest Ubuntu they will wind up in `/usr/local/lib/python2.6/dist-packages/` and under Fedora in `/usr/local/lib/python2.6/sites-packages/`.
- *Data files* defined in a project can land anywhere on the system.
- The *executable script* will land in a `bin` directory on the system. Depending on the platform, this could be `/usr/local/bin` or in a `bin` directory specific to the Python installation.

Ever since Python 2.5, the metadata file is copied alongside the modules and packages as `project-version.egg-info`. For example, the `virtualenv` project could have a `virtualenv-1.4.9.egg-info` file. These metadata files can be considered a database of installed projects, since it's possible to iterate over them and build a list of projects with their versions. However, the `Distutils` installer does not record the list of files it installs on the system. In other words, there is no way to remove all files that were copied in the system. This is a shame since the `install` command has a `-record` option that can be used to record all installed files in a text file. However, this option is not used by default and `Distutils`' documentation barely mentions it.

Setuptools, Pip and the Like

As mentioned in the introduction, some projects tried to fix some of the problems with `Distutils`, with varying degrees of success.

The Dependencies Issue

PyPI allowed developers to publish Python projects that could include several modules organized into Python packages. But at the same time, projects could define module-level dependencies via `Require`. Both ideas are reasonable, but their combination is not.

The right thing to do was to have project-level dependencies, which is exactly what `Setuptools` added as a feature on the top of `Distutils`. It also provided a script called `easy_install` to automatically fetch and install dependencies by looking for them on PyPI. In practice, module-level dependency was never really used, and people jumped on `Setuptools`' extensions. But since these features were added in options specific to `Setuptools`, and ignored by `Distutils` or PyPI, `Setuptools` effectively created its own standard and became a hack on a top of a bad design.

`easy_install` therefore needs to download the archive of the project and run its `setup.py` script again to get the metadata it needs, and it has to do this again for every dependency. The dependency graph is built bit by bit after each download.

Even if the new metadata was accepted by PyPI and browsable online, `easy_install` would still need to download all archives because, as said earlier, metadata published at PyPI is specific to

the platform that was used to upload it, which can differ from the target platform. But this ability to install a project and its dependencies was good enough in 90% of the cases and was a great feature to have. So Setuptools became widely used, although it still suffers from other problems:

- If a dependency install fails, there is no rollback and the system can end up in a broken state.
- The dependency graph is built on the fly during installation, so if a dependency conflict is encountered the system can end up in a broken state as well.

The Uninstall Issue

Setuptools did not provide an uninstaller, even though its custom metadata could have contained a file listing the installed files. Pip, on the other hand, extended Setuptools' metadata to record installed files, and is therefore able to uninstall. But that's yet another custom set of metadata, which means that a single Python installation may contain up to four different flavours of metadata for each installed project:

- Distutils' egg-info, which is a single metadata file.
- Setuptools' egg-info, which is a directory containing the metadata and extra Setuptools specific options.
- Pip's egg-info, which is an extended version of the previous.
- Whatever the hosting packaging system creates.

What About Data Files?

In Distutils, data files can be installed anywhere on the system. If you define some package data files in `setup.py` script like this:

```
setup(...,
    packages=['mypkg'],
    package_dir={'mypkg': 'src/mypkg'},
    package_data={'mypkg': ['data/*.dat']},
)
```

then all files with the `.dat` extension in the `mypkg` project will be included in the distribution and eventually installed along with the Python modules in the Python installation.

For data files that need to be installed outside the Python distribution, there's another option that stores files in the archive but puts them in defined locations:

```
setup(...,
    data_files=[('bitmaps', ['bm/b1.gif', 'bm/b2.gif']),
                ('config', ['cfg/data.cfg']),
                ('etc/init.d', ['init-script'])]
)
```

This is terrible news for OS packagers for several reasons:

- Data files are not part of the metadata, so packagers need to read `setup.py` and sometimes dive into the project's code.
- The developer should not be the one deciding where data files should land on a target system.

- There are no categories for these data files: images, man pages, and everything else are all treated the same way.

A packager who needs to repackage a project with such a file has no choice but to patch the `setup.py` file so that it works as expected for her platform. To do that, she must review the code and change every line that uses those files, since the developer made an assumption about their location. `Setuptools` and `Pip` did not improve this.

14.4 Improved Standards

So we ended up with a mixed up and confused packaging environment, where everything is driven by a single Python module, with incomplete metadata and no way to describe everything a project contains. Here's what we're doing to make things better.

Metadata

The first step is to fix our Metadata standard. PEP 345 defines a new version that includes:

- a saner way to define versions
- project-level dependencies
- a static way to define platform-specific values

Version

One goal of the metadata standard is to make sure that all tools that operate on Python projects are able to classify them the same way. For versions, it means that every tool should be able to know that “1.1” comes after “1.0”. But if project have custom versioning schemes, this becomes much harder.

The only way to ensure consistent versioning is to publish a standard that projects will have to follow. The scheme we chose is a classical sequence-based scheme. As defined in PEP 386, its format is:

`N.N[.N]+[{a|b|c|rc}N[.N]+][.postN][.devN]`

where:

- *N* is an integer. You can use as many *N*s as you want and separate them by dots, as long as there are at least two (MAJOR.MINOR).
- *a*, *b*, *c* and *rc* are *alpha*, *beta* and *release candidate* markers. They are followed by an integer. Release candidates have two markers because we wanted the scheme to be compatible with Python, which uses *rc*. But we find *c* simpler.
- *dev* followed by a number is a dev marker.
- *post* followed by a number is a post-release marker.

Depending on the project release process, dev or post markers can be used for all intermediate versions between two final releases. Most process use dev markers.

Following this scheme, PEP 386 defines a strict ordering:

- alpha < beta < rc < final
- dev < non-dev < post, where non-dev can be a alpha, beta, rc or final

Here's a full ordering example:

```
1.0a1 < 1.0a2.dev456 < 1.0a2 < 1.0a2.1.dev456
  < 1.0a2.1 < 1.0b1.dev456 < 1.0b2 < 1.0b2.post345
    < 1.0c1.dev456 < 1.0c1 < 1.0.dev456 < 1.0
      < 1.0.post456.dev34 < 1.0.post456
```

The goal of this scheme is to make it easy for other packaging systems to translate Python projects' versions into their own schemes. PyPI now rejects any projects that upload PEP 345 metadata with version numbers that don't follow PEP 386.

Dependencies

PEP 345 defines three new fields that replace PEP 314 Requires, Provides, and Obsoletes. Those fields are Requires-Dist, Provides-Dist, and Obsoletes-Dist, and can be used multiple times in the metadata.

For Requires-Dist, each entry contains a string naming some other Distutils project required by this distribution. The format of a requirement string is identical to that of a Distutils project name (e.g., as found in the Name field) optionally followed by a version declaration within parentheses. These Distutils project names should correspond to names as found at PyPI, and version declarations must follow the rules described in PEP 386. Some example are:

```
Requires-Dist: pkginfo
Requires-Dist: PasteDeploy
Requires-Dist: zope.interface (>3.5.0)
```

Provides-Dist is used to define extra names contained in the project. It's useful when a project wants to merge with another project. For example the ZODB project can include the transaction project and state:

```
Provides-Dist: transaction
```

Obsoletes-Dist is useful to mark another project as an obsolete version:

```
Obsoletes-Dist: OldName
```

Environment Markers

An environment marker is a marker that can be added at the end of a field after a semicolon to add a condition about the execution environment. Some examples are:

```
Requires-Dist: pywin32 (>1.0); sys.platform == 'win32'
Obsoletes-Dist: pywin31; sys.platform == 'win32'
Requires-Dist: foo (1,!=1.3); platform.machine == 'i386'
Requires-Dist: bar; python_version == '2.4' or python_version == '2.5'
Requires-External: libxslt; 'linux' in sys.platform
```

The micro-language for environment markers is deliberately kept simple enough for non-Python programmers to understand: it compares strings with the `==` and `in` operators (and their opposites), and allows the usual Boolean combinations. The fields in PEP 345 that can use this marker are:

- Requires-Python
- Requires-External
- Requires-Dist
- Provides-Dist
- Obsoletes-Dist
- Classifier

What's Installed?

Having a single installation format shared among all Python tools is mandatory for interoperability. If we want Installer A to detect that Installer B has previously installed project Foo, they both need to share and update the same database of installed projects.

Of course, users should ideally use a single installer in their system, but they may want to switch to a newer installer that has specific features. For instance, Mac OS X ships Setuptools, so users automatically have the `easy_install` script. If they want to switch to a newer tool, they will need it to be backward compatible with the previous one.

Another problem when using a Python installer on a platform that has a packaging system like RPM is that there is no way to inform the system that a project is being installed. What's worse, even if the Python installer could somehow ping the central packaging system, we would need to have a mapping between the Python metadata and the system metadata. The name of the project, for instance, may be different for each. That can occur for several reasons. The most common one is a conflict name: another project outside the Python land already uses the same name for the RPM. Another cause is that the name used include a python prefix that breaks the convention of the platform. For example, if you name your project `foo-python`, there are high chances that the Fedora RPM will be called `python-foo`.

One way to avoid this problem is to leave the global Python installation alone, managed by the central packaging system, and work in an isolated environment. Tools like `Virtualenv` allows this.

In any case, we do need to have a single installation format in Python because interoperability is also a concern for other packaging systems when they install themselves Python projects. Once a third-party packaging system has registered a newly installed project in its own database on the system, it needs to generate the right metadata for the Python installation itself, so projects appear to be installed to Python installers or any APIs that query the Python installation.

The metadata mapping issue can be addressed in that case: since an RPM knows which Python projects it wraps, it can generate the proper Python-level metadata. For instance, it knows that `python26-webob` is called `WebOb` in the PyPI ecosystem.

Back to our standard: PEP 376 defines a standard for installed packages whose format is quite similar to those used by Setuptools and Pip. This structure is a directory with a `dist-info` extension that contains:

- METADATA: the metadata, as described in PEP 345, PEP 314 and PEP 241.
- RECORD: the list of installed files in a csv-like format.
- INSTALLER: the name of the tool used to install the project.

- REQUESTED: the presence of this file indicates that the project installation was explicitly requested (i.e., not installed as a dependency).

Once all tools out there understand this format, we'll be able to manage projects in Python without depending on a particular installer and its features. Also, since PEP 376 defines the metadata as a directory, it will be easy to add new files to extend it. As a matter of fact, a new metadata file called RESOURCES, described in the next section, might be added in a near future without modifying PEP 376. Eventually, if this new file turns out to be useful for all tools, it will be added to the PEP.

Architecture of Data Files

As described earlier, we need to let the packager decide where to put data files during installation without breaking the developer's code. At the same time, the developer must be able to work with data files without having to worry about their location. Our solution is the usual one: indirection.

Using Data Files

Suppose your MPTools application needs to work with a configuration file. The developer will put that file in a Python package and use `__file__` to reach it:

```
import os

here = os.path.dirname(__file__)
cfg = open(os.path.join(here, 'config', 'mopy.cfg'))
```

This implies that configuration files are installed like code, and that the developer *must* place it alongside her code: in this example, in a subdirectory called `config`.

The new architecture of data files we have designed uses the project tree as the root of all files, and allows access to any file in the tree, whether it is located in a Python package or a simple directory. This allowed developers to create a dedicated directory for data files and access them using `pkgutil.open`:

```
import os
import pkgutil

# Open the file located in config/mopy.cfg in the MPTools project
cfg = pkgutil.open('MPTools', 'config/mopy.cfg')
```

`pkgutil.open` looks for the project metadata and see if it contains a RESOURCES file. This is a simple map of files to locations that the system may contain:

```
config/mopy.cfg {confdir}/{distribution.name}
```

Here the `confdir` variable points to the system's configuration directory, and `distribution.name` contains the name of the Python project as found in the metadata.

As long as this RESOURCES metadata file is created at installation time, the API will find the location of `mopy.cfg` for the developer. And since `config/mopy.cfg` is the path relative to the project tree, it means that we can also offer a development mode where the metadata for the project are generated in-place and added in the lookup paths for `pkgutil`.

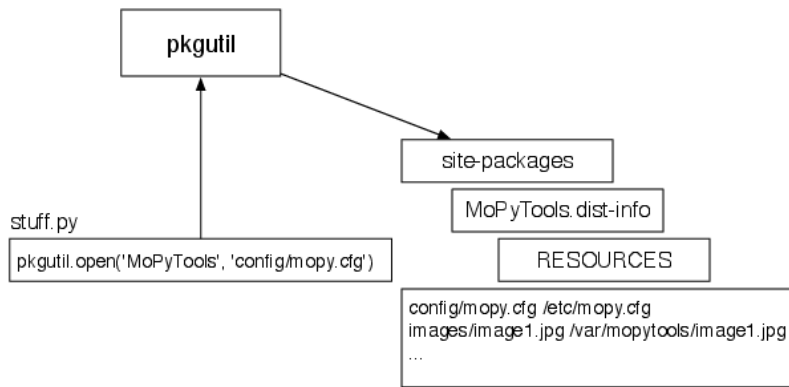


Figure 14.4: Finding a File

Declaring Data Files

In practice, a project can define where data files should land by defining a mapper in their `setup.cfg` file. A mapper is a list of (glob-style pattern, target) tuples. Each pattern points to one of several files in the project tree, while the target is an installation path that may contain variables in brackets. For example, MPTools's `setup.cfg` could look like this:

```
[files]
resources =
    config/mopy.cfg {confdir}/{application.name}/
    images/*.jpg    {datadir}/{application.name}/
```

The `sysconfig` module will provide and document a specific list of variables that can be used, and default values for each platform. For example `confdir` is `/etc` on Linux. Installers can therefore use this mapper in conjunction with `sysconfig` at installation time to know where the files should be placed. Eventually, they will generate the `RESOURCES` file mentioned earlier in the installed metadata so `pkgutil` can find back the files.

PyPI Improvements

I said earlier that PyPI was effectively a single point of failure. PEP 380 addresses this problem by defining a mirroring protocol so that users can fall back to alternative servers when PyPI is down. The goal is to allow members of the community to run mirrors around the world.

The mirror list is provided as a list of host names of the form `X.py.pi.python.org`, where `X` is in the sequence `a, b, c, ..., aa, ab, ...`. `a.py.pi.python.org` is the master server and mirrors start with `b`. A CNAME record `last.py.pi.python.org` points to the last host name so clients that are using PyPI can get the list of the mirrors by looking at the CNAME.

For example, this call tells use that the last mirror is `h.py.pi.python.org`, meaning that PyPI currently has 6 mirrors (b through h):

```
>>> import socket
>>> socket.gethostbyname_ex('last.py.pi.python.org')[0]
'h.py.pi.python.org'
```

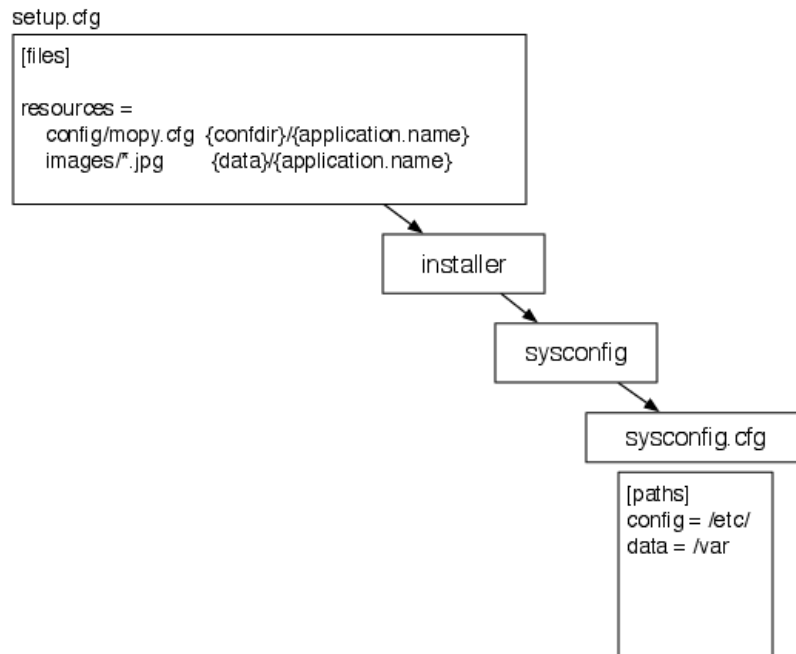


Figure 14.5: Installer

Potentially, this protocol allows clients to redirect requests to the nearest mirror by localizing the mirrors by their IPs, and also fall back to the next mirror if a mirror or the master server is down. The mirroring protocol itself is more complex than a simple rsync because we wanted to keep downloads statistics accurate and provide minimal security.

Synchronization

Mirrors must reduce the amount of data transferred between the central server and the mirror. To achieve that, they *must* use the `changelog` PyPI XML-RPC call, and only refetch the packages that have been changed since the last time. For each package `P`, they *must* copy documents `/simple/P/` and `/serversig/P`.

If a package is deleted on the central server, they *must* delete the package and all associated files. To detect modification of package files, they may cache the file's ETag, and may request skipping it using the `If-None-Match` header. Once the synchronization is over, the mirror changes its `/last-modified` to the current date.

Statistics Propagation

When you download a release from any of the mirrors, the protocol ensures that the download hit is transmitted to the master PyPI server, then to other mirrors. Doing this ensures that people or tools browsing PyPI to find out how many times a release was downloaded will get a value summed across all mirrors.

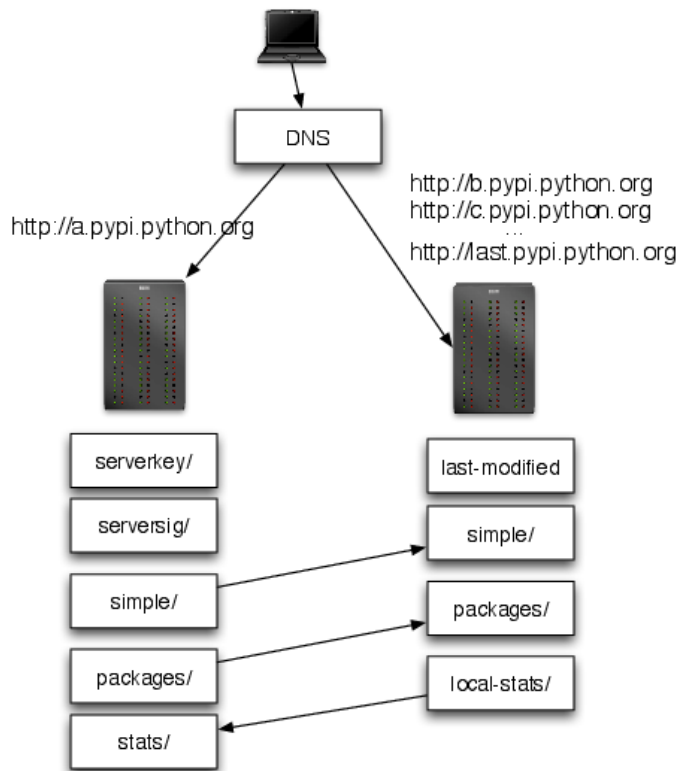


Figure 14.6: Mirroring

Statistics are grouped into daily and weekly CSV files in the stats directory at the central PyPI itself. Each mirror needs to provide a local-stats directory that contains its own statistics. Each file provides the number of downloads for each archive, grouped by use agents. The central server visits mirrors daily to collect those statistics, and merge them back into the global stats directory, so each mirror must keep /local-stats up-to-date at least once a day.

Mirror Authenticity

With any distributed mirroring system, clients may want to verify that the mirrored copies are authentic. Some of the possible threats include:

- the central index may be compromised
- the mirrors might be tampered with
- a man-in-the-middle attack between the central index and the end user, or between a mirror and the end user

To detect the first attack, package authors need to sign their packages using PGP keys, so that users can verify that the package comes from the author they trust. The mirroring protocol itself only addresses the second threat, though some attempt is made to detect man-in-the-middle attacks.

The central index provides a DSA key at the URL `/serverkey`, in the PEM format as generated by `openssl dsa -pubout`³. This URL must not be mirrored, and clients must fetch the official `serverkey` from PyPI directly, or use the copy that came with the PyPI client software. Mirrors should still download the key so that they can detect a key rollover.

For each package, a mirrored signature is provided at `/serversig/package`. This is the DSA signature of the parallel URL `/simple/package`, in DER form, using SHA-1 with DSA⁴.

Clients using a mirror need to perform the following steps to verify a package:

1. Download the `/simple` page, and compute its SHA-1 hash.
2. Compute the DSA signature of that hash.
3. Download the corresponding `/serversig`, and compare it byte for byte with the value computed in step 2.
4. Compute and verify (against the `/simple` page) the MD5 hashes of all files they download from the mirror.

Verification is not needed when downloading from central index, and clients should not do it to reduce the computation overhead.

About once a year, the key will be replaced with a new one. Mirrors will have to re-fetch all `/serversig` pages. Clients using mirrors need to find a trusted copy of the new server key. One way to obtain one is to download it from `https://pypi.python.org/serverkey`. To detect man-in-the-middle attacks, clients need to verify the SSL server certificate, which will be signed by the CACert authority.

14.5 Implementation Details

The implementation of most of the improvements described in the previous section are taking place in `Distutils2`. The `setup.py` file is not used anymore, and a project is completely described in `setup.cfg`, a static `.ini`-like file. By doing this, we make it easier for packagers to change the behavior of a project installation without having to deal with Python code. Here's an example of such a file:

```
[metadata]
name = MPTools
version = 0.1
author = Tarek Ziade
author-email = tarek@mozilla.com
summary = Set of tools to build Mozilla Services apps
description-file = README
home-page = http://bitbucket.org/tarek/pypi2rpm
project-url: Repository, http://hg.mozilla.org/services/server-devtools
classifier = Development Status :: 3 - Alpha
License :: OSI Approved :: Mozilla Public License 1.1 (MPL 1.1)
```

³I.e., RFC 3280 SubjectPublicKeyInfo, with the algorithm 1.3.14.3.2.12.

⁴I.e., as a RFC 3279 Dsa-Sig-Value, created by algorithm 1.2.840.10040.4.3.

```
[files]
packages =
    mopytools
    mopytools.tests

extra_files =
    setup.py
    README
    build.py
    _build.py

resources =
    etc/mopytools.cfg {confdir}/mopytools
```

Distutils2 use this configuration file to:

- generate META-1.2 metadata files that can be used for various actions, like registering at PyPI.
- run any package management command, like `sdist`.
- install a Distutils2-based project.

Distutils2 also implements VERSION via its version module.

The INSTALL-DB implementation will find its way to the standard library in Python 3.3 and will be in the `pkgutil` module. In the interim, a version of this module exists in Distutils2 for immediate use. The provided APIs will let us browse an installation and know exactly what's installed.

These APIs are the basis for some neat Distutils2 features:

- installer/uninstaller
- dependency graph view of installed projects

14.6 Lessons learned

It's All About PEPs

Changing an architecture as wide and complex as Python packaging needs to be carefully done by changing standards through a PEP process. And changing or adding a new PEP takes in my experience around a year.

One mistake the community made along the way was to deliver tools that solved some issues by extending the Metadata and the way Python applications were installed without trying to change the impacted PEPs.

In other words, depending on the tool you used, the standard library Distutils or Setuptools, applications were installed differently. The problems were solved for one part of the community that used these new tools, but added more problems for the rest of the world. OS Packagers for instance, had to face several Python standards: the official documented standard and the de-facto standard imposed by Setuptools.

But in the meantime, Setuptools had the opportunity to experiment in a realistic scale (the whole community) some innovations in a very fast pace, and the feedback was invaluable. We were able to write down new PEPs with more confidence in what worked and what did not, and maybe it would have been impossible to do so differently. So it's all about detecting when some third-party tools are contributing innovations that are solving problems and that should ignite a PEP change.

A Package that Enters the Standard Library Has One Foot in the Grave

I am paraphrasing Guido van Rossum in the section title, but that's one aspect of the batteries-included philosophy of Python that impacts a lot our efforts.

Distutils is part of the standard library and Distutils2 will soon be. A package that's in the standard library is very hard to make evolve. There are of course deprecation processes, where you can kill or change an API after 2 minor versions of Python. But once an API is published, it's going to stay there for years.

So any change you make in a package in the standard library that is not a bug fix, is a potential disturbance for the eco-system. So when you're doing important changes, you have to create a new package.

I've learned it the hard way with Distutils since I had to eventually revert all the changes I had done in it for more than a year and create Distutils2. In the future, if our standards change again in a drastic way, there are high chances that we will start a standalone Distutils3 project first, unless the standard library is released on its own at some point.

Backward Compatibility

Changing the way packaging works in Python is a very long process: the Python ecosystem contains so many projects based on older packaging tools that there is and will be a lot of resistance to change. (Reaching consensus on some of the topics discussed in this chapter took several years, rather than the few months I originally expected.) As with Python 3, it will take years before all projects switch to the new standard.

That's why everything we are doing has to be backward-compatible with all previous tools, installations and standards, which makes the implementation of Distutils2 a wicked problem.

For example, if a project that uses the new standards depends on another project that doesn't use them yet, we can't stop the installation process by telling the end-user that the dependency is in an unknown format!

For example, the INSTALL-DB implementation contains compatibility code to browse projects installed by the original Distutils, Pip, Distribute, or Setuptools. Distutils2 is also able to install projects created by the original Distutils by converting their metadata on the fly.

14.7 References and Contributions

Some sections in this paper were directly taken from the various PEP documents we wrote for packaging. You can find the original documents at <http://python.org>:

- PEP 241: Metadata for Python Software Packages 1.0: <http://python.org/peps/pep-0214.html>
- PEP 314: Metadata for Python Software Packages 1.1: <http://python.org/peps/pep-0314.html>
- PEP 345: Metadata for Python Software Packages 1.2: <http://python.org/peps/pep-0345.html>
- PEP 376: Database of Installed Python Distributions: <http://python.org/peps/pep-0376.html>
- PEP 381: Mirroring infrastructure for PyPI: <http://python.org/peps/pep-0381.html>

- PEP 386: Changing the version comparison module in Distutils: <http://python.org/peps/pep-0386.html>

I would like to thank all the people that are working on packaging; you will find their name in every PEP I've mentioned. I would also like to give a special thank to all members of The Fellowship of the Packaging. Also, thanks to Alexis Metaireau, Toshio Kuratomi, Holger Krekel and Stefane Fermigier for their feedback on this chapter.

The projects that were discussed in this chapter are:

- Distutils: <http://docs.python.org/distutils>
- Distutils2: <http://packages.python.org/Distutils2>
- Distribute: <http://packages.python.org/distribute>
- Setuptools: <http://pypi.python.org/pypi/setuptools>
- Pip: <http://pypi.python.org/pypi/pip>
- Virtualenv: <http://pypi.python.org/pypi/virtualenv>

Riak and Erlang/OTP

Francesco Cesarini, Andy Gross, and Justin Sheehy

Riak is a distributed, fault tolerant, open source database that illustrates how to build large scale systems using Erlang/OTP. Thanks in large part to Erlang's support for massively scalable distributed systems, Riak offers features that are uncommon in databases, such as high-availability and linear scalability of both capacity and throughput.

Erlang/OTP provides an ideal platform for developing systems like Riak because it provides inter-node communication, message queues, failure detectors, and client-server abstractions out of the box. What's more, most frequently-used patterns in Erlang have been implemented in library modules, commonly referred to as OTP behaviors. They contain the generic code framework for concurrency and error handling, simplifying concurrent programming and protecting the developer from many common pitfalls. Behaviors are monitored by supervisors, themselves a behavior, and grouped together in supervision trees. A supervision tree is packaged in an application, creating a building block of an Erlang program.

A complete Erlang system such as Riak is a set of loosely coupled applications that interact with each other. Some of these applications have been written by the developer, some are part of the standard Erlang/OTP distribution, and some may be other open source components. They are sequentially loaded and started by a boot script generated from a list of applications and versions.

What *differs* among systems are the applications that are part of the release which is started. In the standard Erlang distribution, the boot files will start the *Kernel* and *StdLib* (Standard Library) applications. In some installations, the *SASL* (Systems Architecture Support Library) application is also started. SASL contains release and software upgrade tools together with logging capabilities. Riak is no different, other than starting the Riak specific applications as well as their runtime dependencies, which include *Kernel*, *StdLib* and *SASL*. A complete and ready-to-run build of Riak actually embeds these standard elements of the Erlang/OTP distribution and starts them all in unison when `riak start` is invoked on the command line. Riak consists of many complex applications, so this chapter should not be interpreted as a complete guide. It should be seen as an introduction to OTP where examples from the Riak source code are used. The figures and examples have been abbreviated and shortened for demonstration purposes.

15.1 An Abridged Introduction to Erlang

Erlang is a concurrent functional programming language that compiles to byte code and runs in a virtual machine. Programs consist of functions that call each other, often resulting in side effects such as inter-process message passing, I/O and database operations. Erlang variables are single assignment, i.e., once they have been given values, they cannot be updated. The language makes extensive use of pattern matching, as shown in the factorial example below:

```
-module(factorial).  
-export([fac/1]).  
fac(0) -> 1;  
fac(N) when N>0 ->  
    Prev = fac(N-1),  
    N*Prev.
```

Here, the first clause gives the factorial of zero, the second factorials of positive numbers. The body of each clause is a sequence of expressions, and the final expression in the body is the result of that clause. Calling the function with a negative number will result in a run time error, as none of the clauses match. Not handling this case is an example of non-defensive programming, a practice encouraged in Erlang.

Within the module, functions are called in the usual way; outside, the name of the module is prepended, as in `factorial:fac(3)`. It is possible to define functions with the same name but different numbers of arguments—this is called their *arity*. In the export directive in the `factorial` module the `fac` function of arity one is denoted by `fac/1`.

Erlang supports tuples (also called product types) and lists. Tuples are enclosed in curly brackets, as in `{ok, 37}`. In tuples, we access elements by position. Records are another data type; they allow us to store a fixed number of elements which are then accessed and manipulated by name. We define a record using the `-record(state, {id, msg_list=[]})`. To create an instance, we use the expression `Var = #state{id=1}`, and we examine its contents using `Var#state.id`. For a variable number of elements, we use lists defined in square brackets such as in `[23, 34]`. The notation `[X|Xs]` matches a non-empty list with head `X` and tail `Xs`. Identifiers beginning with a lower case letter denote atoms, which simply stand for themselves; the `ok` in the tuple `{ok, 37}` is an example of an atom. Atoms used in this way are often used to distinguish between different kinds of function result: as well as `ok` results, there might be results of the form `{error, "Error String"}`.

Processes in Erlang systems run concurrently in separate memory, and communicate with each other by message passing. Processes can be used for a wealth of applications, including gateways to databases, as handlers for protocol stacks, and to manage the logging of trace messages from other processes. Although these processes handle different requests, there will be similarities in how these requests are handled.

As processes exist only within the virtual machine, a single VM can simultaneously run millions of processes, a feature Riak exploits extensively. For example, each request to the database—reads, writes, and deletes—is modeled as a separate process, an approach that would not be possible with most OS-level threading implementations.

Processes are identified by process identifiers, called PIDs, but they can also be registered under an alias; this should only be used for long-lived “static” processes. Registering a process with its alias allows other processes to send it messages without knowing its PID. Processes are created using the `spawn(Module, Function, Arguments)` built-in function (BIF). BIFs are functions integrated in the VM and used to do what is impossible or slow to execute in pure Erlang. The `spawn/3` BIF

takes a Module, a Function and a list of Arguments as parameters. The call returns the PID of the newly spawned process and as a side effect, creates a new process that starts executing the function in the module with the arguments mentioned earlier.

A message `Msg` is sent to a process with process id `Pid` using `Pid ! Msg`. A process can find out its PID by calling the BIF `self`, and this can then be sent to other processes for them to use to communicate with the original process. Suppose that a process expects to receive messages of the form `{ok, N}` and `{error, Reason}`. To process these it uses a receive statement:

```
receive
  {ok, N} ->
    N+1;
  {error, _} ->
    0
end
```

The result of this is a number determined by the pattern-matched clause. When the value of a variable is not needed in the pattern match, the underscore wild-card can be used as shown above.

Message passing between processes is asynchronous, and the messages received by a process are placed in the process's mailbox in the order in which they arrive. Suppose that now the receive expression above is to be executed: if the first element in the mailbox is either `{ok, N}` or `{error, Reason}` the corresponding result will be returned. If the first message in the mailbox is not of this form, it is retained in the mailbox and the second is processed in a similar way. If no message matches, the receive will wait for a matching message to be received.

Processes terminate for two reasons. If there is no more code to execute, they are said to terminate with reason *normal*. If a process encounters a run-time error, it is said to terminate with a *non-normal* reason. A process terminating will not affect other processes unless they are linked to it. Processes can link to each other through the `link(Pid)` BIF or when calling the `spawn_link(Module, Function, Arguments)`. If a process terminates, it sends an EXIT signal to processes in its link set. If the termination reason is non-normal, the process terminates itself, propagating the EXIT signal further. By calling the `process_flag(trap_exit, true)` BIF, processes can receive the EXIT signals as Erlang messages in their mailbox instead of terminating.

Riak uses EXIT signals to monitor the well-being of helper processes performing non-critical work initiated by the request-driving finite state machines. When these helper processes terminate abnormally, the EXIT signal allows the parent to either ignore the error or restart the process.

15.2 Process Skeletons

We previously introduced the notion that processes follow a common pattern regardless of the particular purpose for which the process was created. To start off, a process has to be spawned and then, optionally, have its alias registered. The first action of the newly spawned process is to initialize the process loop data. The loop data is often the result of arguments passed to the `spawn` built-in function at the initialization of the process. Its loop data is stored in a variable we refer to as the process state. The state, often stored in a record, is passed to a receive-evaluate function, running a loop which receives a message, handles it, updates the state, and passes it back as an argument to a tail-recursive call. If one of the messages it handles is a 'stop' message, the receiving process will clean up after itself and then terminate.

This is a recurring theme among processes that will occur regardless of the task the process has been assigned to perform. With this in mind, let's look at the differences between the processes that conform to this pattern:

- The arguments passed to the spawn BIF calls will differ from one process to another.
- You have to decide whether you should register a process under an alias, and if you do, what alias should be used.
- In the function that initializes the process state, the actions taken will differ based on the tasks the process will perform.
- The state of the system is represented by the loop data in every case, but the contents of the loop data will vary among processes.
- When in the body of the receive-evaluate loop, processes will receive different messages and handle them in different ways.
- Finally, on termination, the cleanup will vary from process to process.

So, even if a skeleton of generic actions exists, these actions are complemented by specific ones that are directly related to the tasks assigned to the process. Using this skeleton as a template, programmers can create Erlang processes that act as servers, finite state machines, event handlers and supervisors. But instead of re-implementing these patterns every time, they have been placed in library modules referred to as behaviors. They come as part as the OTP middleware.

15.3 OTP Behaviors

The core team of developers committing to Riak is spread across nearly a dozen geographical locations. Without very tight coordination and templates to work from, the result would consist of different client/server implementations not handling special borderline cases and concurrency-related errors. There would probably be no uniform way to handle client and server crashes or guaranteeing that a response from a request is indeed the response, and not just any message that conforms to the internal message protocol.

OTP is a set of Erlang libraries and design principles providing ready-made tools with which to develop robust systems. Many of these patterns and libraries are provided in the form of “behaviors.”

OTP behaviors address these issues by providing library modules that implement the most common concurrent design patterns. Behind the scenes, without the programmer having to be aware of it, the library modules ensure that errors and special cases are handled in a consistent way. As a result, OTP behaviors provide a set of standardized building blocks used in designing and building industrial-grade systems.

Introduction

OTP behaviors are provided as library modules in the `stdlib` application which comes as part of the Erlang/OTP distribution. The specific code, written by the programmer, is placed in a separate module and called through a set of predefined callback functions standardized for each behavior. This callback module will contain all of the specific code required to deliver the desired functionality.

OTP behaviors include worker processes, which do the actual processing, and supervisors, whose task is to monitor workers and other supervisors. Worker behaviors, often denoted in diagrams as circles, include servers, event handlers, and finite state machines. Supervisors, denoted in illustrations

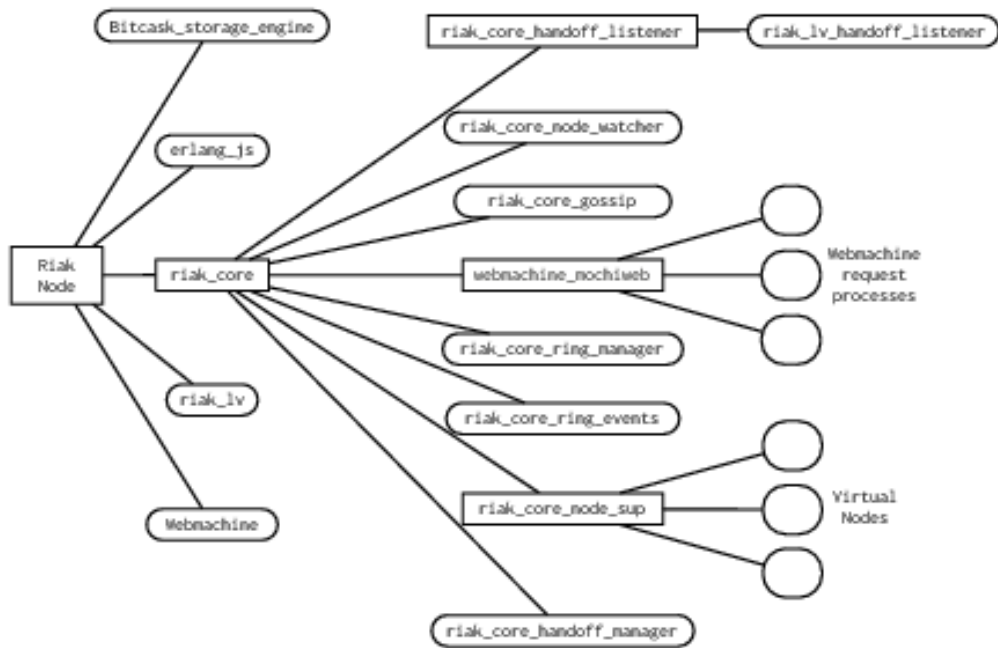


Figure 15.1: OTP Riak Supervision Tree

as squares, monitor their children, both workers and other supervisors, creating what is called a supervision tree.

Supervision trees are packaged into a behavior called an application. OTP applications are not only the building blocks of Erlang systems, but are also a way to package reusable components. Industrial-grade systems like Riak consist of a set of loosely coupled, possibly distributed applications. Some of these applications are part of the standard Erlang distribution and some are the pieces that make up the specific functionality of Riak.

Examples of OTP applications include the Corba ORB or the Simple Network Management Protocol (SNMP) agent. An OTP application is a reusable component that packages library modules together with supervisor and worker processes. From now on, when we refer to an application, we will mean an OTP application.

The behavior modules contain all of the generic code for each given behavior type. Although it is possible to implement your own behavior module, doing so is rare because the ones that come with the Erlang/OTP distribution will cater to most of the design patterns you would use in your code. The generic functionality provided in a behavior module includes operations such as:

- spawning and possibly registering the process;
- sending and receiving client messages as synchronous or asynchronous calls, including defining the internal message protocol;
- storing the loop data and managing the process loop; and
- stopping the process.

The loop data is a variable that will contain the data the behavior needs to store in between calls. After the call, an updated variant of the loop data is returned. This updated loop data, often referred

to as the new loop data, is passed as an argument in the next call. Loop data is also often referred to as the behavior state.

The functionality to be included in the callback module for the generic server application to deliver the specific required behavior includes the following:

- Initializing the process loop data, and, if the process is registered, the process name.
- Handling the specific client requests, and, if synchronous, the replies sent back to the client.
- Handling and updating the process loop data in between the process requests.
- Cleaning up the process loop data upon termination.

Generic Servers

Generic servers that implement client/server behaviors are defined in the `gen_server` behavior that comes as part of the standard library application. In explaining generic servers, we will use the `riak_core_node_watcher.erl` module from the `riak_core` application. It is a server that tracks and reports on which sub-services and nodes in a Riak cluster are available. The module headers and directives are as follows:

```
-module(riak_core_node_watcher).
-behavior(gen_server).
%% API
-export([start_link/0,service_up/2,service_down/1,node_up/0,node_down/0,services/0,
        services/1,nodes/1,avsn/0]).
%% gen_server callbacks
-export([init/1,handle_call/3,handle_cast/2,handle_info/2,terminate/2,code_change/3]).

-record(state, {status=up, services=[], peers=[], avsn=0, bcast_tref,
               bcast_mod={gen_server, abcast}}).
```

We can easily recognize generic servers through the `-behavior(gen_server).` directive. This directive is used by the compiler to ensure all callback functions are properly exported. The record state is used in the server loop data.

Starting Your Server

With the `gen_server` behavior, instead of using the `spawn` and `spawn_link` BIFs, you will use the `gen_server:start` and `gen_server:start_link` functions. The main difference between `spawn` and `start` is the synchronous nature of the call. Using `start` instead of `spawn` makes starting the worker process more deterministic and prevents unforeseen race conditions, as the call will not return the PID of the worker until it has been initialized. You call the functions with either of:

```
gen_server:start_link(ServerName, CallbackModule, Arguments, Options)
gen_server:start_link(CallbackModule, Arguments, Options)
```

`ServerName` is a tuple of the format `{local, Name}` or `{global, Name}`, denoting a local or global Name for the process alias if it is to be registered. Global names allow servers to be transparently accessed across a cluster of distributed Erlang nodes. If you do not want to register the process and instead reference it using its PID, you omit the argument and use a `start_link/3` or `start/3` function call instead. `CallbackModule` is the name of the module in which the specific callback

functions are placed, `Arguments` is a valid Erlang term that is passed to the `init/1` callback function, while `Options` is a list that allows you to set the memory management flags `fullsweep_after` and `heapsize`, as well as other tracing and debugging flags.

In our example, we call `start_link/4`, registering the process with the same name as the callback module, using the `?MODULE` macro call. This macro is expanded to the name of the module it is defined in by the preprocessor when compiling the code. It is always good practice to name your behavior with an alias that is the same as the callback module it is implemented in. We don't pass any arguments, and as a result, just send the empty list. The options list is kept empty:

```
start_link() ->
    gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).
```

The obvious difference between the `start_link` and `start` functions is that `start_link` links to its parent, most often a supervisor, while `start` doesn't. This needs a special mention as it is an OTP behavior's responsibility to link itself to the supervisor. The `start` functions are often used when testing behaviors from the shell, as a typing error causing the shell process to crash would not affect the behavior. All variants of the `start` and `start_link` functions return `{ok, Pid}`.

The `start` and `start_link` functions will spawn a new process that calls the `init(Arguments)` callback function in the `CallbackModule`, with the `Arguments` supplied. The `init` function must initialize the `LoopData` of the server and has to return a tuple of the format `{ok, LoopData}`. `LoopData` contains the first instance of the loop data that will be passed between the callback functions. If you want to store some of the arguments you passed to the `init` function, you would do so in the `LoopData` variable. The `LoopData` in the Riak node watcher server is the result of the `schedule_broadcast/1` called with a record of type `state` where the fields are set to the default values:

```
init([]) ->

    %% Watch for node up/down events
    net_kernel:monitor_nodes(true),

    %% Setup ETS table to track node status
    ets:new(?MODULE, [protected, named_table]),

    {ok, schedule_broadcast(#state{})}.
```

Although the supervisor process might call the `start_link/4` function, a different process calls the `init/1` callback: the one that was just spawned. As the purpose of this server is to notice, record, and broadcast the availability of sub-services within Riak, the initialization asks the Erlang runtime to notify it of such events, and sets up a table to store this information in. This needs to be done during initialization, as any calls to the server would fail if that structure did not yet exist. Do only what is necessary and minimize the operations in your `init` function, as the call to `init` is a synchronous call that prevents all of the other serialized processes from starting until it returns.

Passing Messages

If you want to send a synchronous message to your server, you use the `gen_server:call/2` function. Asynchronous calls are made using the `gen_server:cast/2` function. Let's start by taking two functions from Riak's service API; we will provide the rest of the code later. They are called by the

client process and result in a synchronous message being sent to the server process registered with the same name as the callback module. Note that validating the data sent to the server should occur on the client side. If the client sends incorrect information, the server should terminate.

```
service_up(Id, Pid) ->
    gen_server:call(?MODULE, {service_up, Id, Pid}).
```

```
service_down(Id) ->
    gen_server:call(?MODULE, {service_down, Id}).
```

Upon receiving the messages, the `gen_server` process calls the `handle_call/3` callback function dealing with the messages in the same order in which they were sent:

```
handle_call({service_up, Id, Pid}, _From, State) ->
    %% Update the set of active services locally
    Services = ordsets:add_element(Id, State#state.services),
    S2 = State#state { services = Services },

    %% Remove any existing mrefs for this service
    delete_service_mref(Id),

    %% Setup a monitor for the Pid representing this service
    Mref = erlang:monitor(process, Pid),
    erlang:put(Mref, Id),
    erlang:put(Id, Mref),

    %% Update our local ETS table and broadcast
    S3 = local_update(S2),
    {reply, ok, update_avsn(S3)};

handle_call({service_down, Id}, _From, State) ->
    %% Update the set of active services locally
    Services = ordsets:del_element(Id, State#state.services),
    S2 = State#state { services = Services },

    %% Remove any existing mrefs for this service
    delete_service_mref(Id),

    %% Update local ETS table and broadcast
    S3 = local_update(S2),
    {reply, ok, update_avsn(S3)};
```

Note the return value of the callback function. The tuple contains the control atom `reply`, telling the `gen_server` generic code that the second element of the tuple (which in both of these cases is the atom `ok`) is the reply sent back to the client. The third element of the tuple is the new `State`, which, in a new iteration of the server, is passed as the third argument to the `handle_call/3` function; in both cases here it is updated to reflect the new set of available services. The argument `_From` is a tuple containing a unique message reference and the client process identifier. The tuple as a whole is used in library functions that we will not be discussing in this chapter. In the majority of cases, you will not need it.

The `gen_server` library module has a number of mechanisms and safeguards built in that operate behind the scenes. If your client sends a synchronous message to your server and you do not get a response within five seconds, the process executing the `call/2` function is terminated. You can override this by using `gen_server:call(Name, Message, Timeout)` where `Timeout` is a value in milliseconds or the atom `infinity`.

The timeout mechanism was originally put in place for deadlock prevention purposes, ensuring that servers that accidentally call each other are terminated after the default timeout. The crash report would be logged, and hopefully would result in the error being debugged and fixed. Most applications will function appropriately with a timeout of five seconds, but under very heavy loads, you might have to fine-tune the value and possibly even use `infinity`; this choice is application-dependent. All of the critical code in Erlang/OTP uses `infinity`. Various places in Riak use different values for the timeout: `infinity` is common between coupled pieces of the internals, while `Timeout` is set based on a user-passed parameter in cases where the client code talking to Riak has specified that an operation should be allowed to time out.

Other safeguards when using the `gen_server:call/2` function include the case of sending a message to a nonexistent server and the case of a server crashing before sending its reply. In both cases, the calling process will terminate. In raw Erlang, sending a message that is never pattern-matched in a receive clause is a bug that can cause a memory leak. Two different strategies are used in Riak to mitigate this, both of which involve “catchall” matching clauses. In places where the message might be user-initiated, an unmatched message might be silently discarded. In places where such a message could only come from Riak’s internals, it represents a bug and so will be used to trigger an error-alerting internal crash report, restarting the worker process that received it.

Sending asynchronous messages works in a similar way. Messages are sent asynchronously to the generic server and handled in the `handle_cast/2` callback function. The function has to return a tuple of the format `{reply, NewState}`. Asynchronous calls are used when we are not interested in the request of the server and are not worried about producing more messages than the server can consume. In cases where we are not interested in a response but want to wait until the message has been handled before sending the next request, we would use a `gen_server:call/2`, returning the atom `ok` in the reply. Picture a process generating database entries at a faster rate than Riak can consume. By using asynchronous calls, we risk filling up the process mailbox and make the node run out of memory. Riak uses the message-serializing properties of synchronous `gen_server` calls to regulate load, processing the next request only when the previous one has been handled. This approach eliminates the need for more complex throttling code: in addition to enabling concurrency, `gen_server` processes can also be used to introduce serialization points.

Stopping the Server

How do you stop the server? In your `handle_call/3` and `handle_cast/2` callback functions, instead of returning `{reply, Reply, NewState}` or `{noreply, NewState}`, you can return `{stop, Reason, Reply, NewState}` or `{stop, Reason, NewState}`, respectively. Something has to trigger this return value, often a stop message sent to the server. Upon receiving the stop tuple containing the Reason and State, the generic code executes the `terminate(Reason, State)` callback.

The `terminate` function is the natural place to insert the code needed to clean up the State of the server and any other persistent data used by the system. In our example, we send out one last message to our peers so that they know that this node watcher is no longer up and watching. In this example, the variable `State` contains a record with the fields `status` and `peers`:

```

terminate(_Reason, State) ->
    %% Let our peers know that we are shutting down
    broadcast(State#state.peers, State#state { status = down }).

```

Use of the behavior callbacks as library functions and invoking them from other parts of your program is an extremely bad practice. For example, you should never call `riak_core_node_watcher:init(Args)` from another module to retrieve the initial loop data. Such retrievals should be done through a synchronous call to the server. Calls to behavior callback functions should originate only from the behavior library modules as a result of an event occurring in the system, and never directly by the user.

15.4 Other Worker Behaviors

A large number of other worker behaviors can and have been implemented using these same ideas.

Finite State Machines

Finite state machines (FSMs), implemented in the `gen_fsm` behavior module, are a crucial component when implementing protocol stacks in telecom systems (the problem domain Erlang was originally invented for). States are defined as callback functions named after the state that return a tuple containing the next State and the updated loop data. You can send events to these states synchronously and asynchronously. The finite state machine callback module should also export the standard callback functions such as `init`, `terminate`, and `handle_info`.

Of course, finite state machines are not telecom specific. In Riak, they are used in the request handlers. When a client issues a request such as `get`, `put`, or `delete`, the process listening to that request will spawn a process implementing the corresponding `gen_fsm` behavior. For instance, the `riak_kv_get_fsm` is responsible for handling a `get` request, retrieving data and sending it out to the client process. The FSM process will pass through various states as it determines which nodes to ask for the data, as it sends out messages to those nodes, and as it receives data, errors, or timeouts in response.

Event Handlers

Event handlers and managers are another behavior implemented in the `gen_event` library module. The idea is to create a centralized point that receives events of a specific kind. Events can be sent synchronously and asynchronously with a predefined set of actions being applied when they are received. Possible responses to events include logging them to file, sending off an alarm in the form of an SMS, or collecting statistics. Each of these actions is defined in a separate callback module with its own loop data, preserved between calls. Handlers can be added, removed, or updated for every specific event manager. So, in practice, for every event manager there could be many callback modules, and different instances of these callback modules could exist in different managers. Event handlers include processes receiving alarms, live trace data, equipment related events or simple logs.

One of the uses for the `gen_event` behavior in Riak is for managing subscriptions to “ring events”, i.e., changes to the membership or partition assignment of a Riak cluster. Processes on a Riak node can register a function in an instance of `riak_core_ring_events`, which implements the `gen_event` behavior. Whenever the central process managing the ring for that node changes the membership record for the overall cluster, it fires off an event that causes each of those callback

modules to call the registered function. In this fashion, it is easy for various parts of Riak to respond to changes in one of Riak’s most central data structures without having to add complexity to the central management of that structure.

Most common concurrency and communication patterns are handled with the three primary behaviors we’ve just discussed: `gen_server`, `gen_fsm`, and `gen_event`. However, in large systems, some application-specific patterns emerge over time that warrant the creation of new behaviors. Riak includes one such behavior, `riak_core_vnode`, which formalizes how virtual nodes are implemented. Virtual nodes are the primary storage abstraction in Riak, exposing a uniform interface for key-value storage to the request-driving FSMs. The interface for callback modules is specified using the `behavior_info/1` function, as follows:

```
behavior_info(callbacks) ->
  [{init,1},
   {handle_command,3},
   {handoff_starting,2},
   {handoff_cancelled,1},
   {handoff_finished,2},
   {handle_handoff_command,3},
   {handle_handoff_data,2},
   {encode_handoff_item,2},
   {is_empty,1},
   {terminate,2},
   {delete,1}];
```

The above example shows the `behavior_info/1` function from `riak_core_vnode`. The list of `{CallbackFunction, Arity}` tuples defines the contract that callback modules must follow. Concrete virtual node implementations must export these functions, or the compiler will emit a warning. Implementing your own OTP behaviors is relatively straightforward. Alongside defining your callback functions, using the `proc_lib` and `sys` modules, you need to start them with particular functions, handle system messages and monitor the parent in case it terminates.

15.5 Supervisors

The supervisor behavior’s task is to monitor its children and, based on some preconfigured rules, take action when they terminate. Children consist of both supervisors and worker processes. This allows the Riak codebase to focus on the correct case, which enables the supervisor to handle software bugs, corrupt data or system errors in a consistent way across the whole system. In the Erlang world, this non-defensive programming approach is often referred to the “let it crash” strategy. The children that make up the supervision tree can include both supervisors and worker processes. Worker processes are OTP behaviors including the `gen_fsm`, `gen_server`, and `gen_event`. The Riak team, not having to handle borderline error cases, get to work with a smaller code base. This code base, because of its use of behaviors, is smaller to start off with, as it only deals with specific code. Riak has a top-level supervisor like most Erlang applications, and also has sub-supervisors for groups of processes with related responsibilities. Examples include Riak’s virtual nodes, TCP socket listeners, and query-response managers.

Supervisor Callback Functions

To demonstrate how the supervisor behavior is implemented, we will use the `riak_core_sup.erl` module. The Riak core supervisor is the top level supervisor of the Riak core application. It starts a set of static workers and supervisors, together with a dynamic number of workers handling the HTTP and HTTPS bindings of the node's RESTful API defined in application specific configuration files. In a similar way to `gen_servers`, all supervisor callback modules must include the `-behavior(supervisor).` directive. They are started using the `start` or `start_link` functions which take the optional `ServerName`, the `CallBackModule`, and an `Argument` which is passed to the `init/1` callback function.

Looking at the first few lines of code in the `riak_core_sup.erl` module, alongside the `behavior` directive and a macro we will describe later, we notice the `start_link/3` function:

```
-module(riak_core_sup).
-behavior(supervisor).
%% API
-export([start_link/0]).
%% Supervisor callbacks
-export([init/1]).
-define(CHILD(I, Type), {I, {I, start_link, []}, permanent, 5000, Type, [I]}).
start_link() ->
    supervisor:start_link({local, ?MODULE}, ?MODULE, []).
```

Starting a supervisor will result in a new process being spawned, and the `init/1` callback function being called in the callback module `riak_core_sup.erl`. The `ServerName` is a tuple of the format `{local, Name}` or `{global, Name}`, where `Name` is the supervisor's registered name. In our example, both the registered name and the callback module are the atom `riak_core_sup`, originating from the `?MODULE` macro. We pass the empty list as an argument to `init/1`, treating it as a null value. The `init` function is the only supervisor callback function. It has to return a tuple with format:

```
{ok, {SupervisorSpecification, ChildSpecificationList}}
```

where `SupervisorSpecification` is a 3-tuple `{RestartStrategy, AllowedRestarts, MaxSeconds}` containing information on how to handle process crashes and restarts. `RestartStrategy` is one of three configuration parameters determining how the behavior's siblings are affected upon abnormal termination:

- `one_for_one`: other processes in the supervision tree are not affected.
- `rest_for_one`: processes started after the terminating process are terminated and restarted.
- `one_for_all`: all processes are terminated and restarted.

`AllowedRestarts` states how many times any of the supervisor children may terminate in `MaxSeconds` before the supervisor terminates itself (and its children). When one terminates, it sends an `EXIT` signal to its supervisor which, based on its restart strategy, handles the termination accordingly. The supervisor terminating after reaching the maximum allowed restarts ensures that cyclic restarts and other issues that cannot be resolved at this level are escalated. Chances are that the issue is in a process located in a different sub-tree, allowing the supervisor receiving the escalation to terminate the affected sub-tree and restart it.

Examining the last line of the `init/1` callback function in the `riak_core_sup.erl` module, we notice that this particular supervisor has a one-for-one strategy, meaning that the processes are

independent of each other. The supervisor will allow a maximum of ten restarts before restarting itself.

ChildSpecificationList specifies which children the supervisor has to start and monitor, together with information on how to terminate and restart them. It consists of a list of tuples of the following format:

```
{Id, {Module, Function, Arguments}, Restart, Shutdown, Type, ModuleList}
```

Id is a unique identifier for that particular supervisor. Module, Function, and Arguments is an exported function which results in the behavior `start_link` function being called, returning the tuple of the format `{ok, Pid}`. The Restart strategy dictates what happens depending on the termination type of the process, which can be:

- transient processes, which are never restarted;
- temporary processes, are restarted only if they terminate abnormally; and
- permanent processes, which are always restarted, regardless of the termination being normal or abnormal.

Shutdown is a value in milliseconds referring to the time the behavior is allowed to execute in the `terminate` function when terminating as the result of a restart or shutdown. The atom `infinity` can also be used, but for behaviors other than supervisors, it is highly discouraged. Type is either the atom `worker`, referring to the generic servers, event handlers and finite state machines, or the atom `supervisor`. Together with `ModuleList`, a list of modules implementing the behavior, they are used to control and suspend processes during the runtime software upgrade procedures. Only existing or user implemented behaviors may be part of the child specification list and hence included in a supervision tree.

With this knowledge at hand, we should now be able to formulate a restart strategy defining inter-process dependencies, fault tolerance thresholds and escalation procedures based on a common architecture. We should also be able to understand what is going on in the `init/1` example of the `riak_core_sup.erl` module. First of all, study the `CHILD` macro. It creates the child specification for one child, using the callback module name as `Id`, making it permanent and giving it a shut down time of 5 seconds. Different child types can be workers or supervisors. Have a look at the example, and see what you can make out of it:

```
-define(CHILD(I, Type), {I, {I, start_link, []}, permanent, 5000, Type, [I]}).
```

```
init([]) ->
    RiakWebs = case lists:flatten(riak_core_web:bindings(http),
                                riak_core_web:bindings(https)) of
        [] ->
            %% check for old settings, in case app.config
            %% was not updated
            riak_core_web:old_binding();
        Binding ->
            Binding
    end,

    Children =
        [?CHILD(riak_core_vnode_sup, supervisor),
```

```

        ?CHILD(riak_core_handoff_manager, worker),
        ?CHILD(riak_core_handoff_listener, worker),
        ?CHILD(riak_core_ring_events, worker),
        ?CHILD(riak_core_ring_manager, worker),
        ?CHILD(riak_core_node_watcher_events, worker),
        ?CHILD(riak_core_node_watcher, worker),
        ?CHILD(riak_core_gossip, worker) |
        RiakWebs
    ],
    {ok, {{one_for_one, 10, 10}, Children}}.

```

Most of the Children started by this supervisor are statically defined workers (or in the case of the `vnode_sup`, a supervisor). The exception is the `RiakWebs` portion, which is dynamically defined depending on the HTTP portion of Riak's configuration file.

With the exception of library applications, every OTP application, including those in Riak, will have their own supervision tree. In Riak, various top-level applications are running in the Erlang node, such as `riak_core` for distributed systems algorithms, `riak_kv` for key/value storage semantics, `webmachine` for HTTP, and more. We have shown the expanded tree under `riak_core` to demonstrate the multi-level supervision going on. One of the many benefits of this structure is that a given subsystem can be crashed (due to bug, environmental problem, or intentional action) and only that subtree will in a first instance be terminated.

The supervisor will restart the needed processes and the overall system will not be affected. In practice we have seen this work well for Riak. A user might figure out how to crash a virtual node, but it will just be restarted by `riak_core_vnode_sup`. If they manage to crash that, the `riak_core` supervisor will restart it, propagating the termination to the top-level supervisor. This failure isolation and recovery mechanism allows Riak (and Erlang) developers to straightforwardly build resilient systems.

The value of the supervisory model was shown when one large industrial user created a very abusive environment in order to find out where each of several database systems would fall apart. This environment created random huge bursts of both traffic and failure conditions. They were confused when Riak simply wouldn't stop running, even under the worst such arrangement. Under the covers, of course, they were able to make individual processes or subsystems crash in multiple ways—but the supervisors would clean up and restart things to put the whole system back into working order every time.

Applications

The application behavior we previously introduced is used to package Erlang modules and resources into reusable components. In OTP, there are two kinds of applications. The most common form, called normal applications, will start a supervision tree and all of the relevant static workers. Library applications such as the Standard Library, which come as part of the Erlang distribution, contain library modules but do not start a supervision tree. This is not to say that the code may not contain processes or supervision trees. It just means they are started as part of a supervision tree belonging to another application.

An Erlang system will consist of a set of loosely coupled applications. Some are written by the developers, some are available as open source, and others are be part of the Erlang/OTP distribution. The Erlang runtime system and its tools treat all applications equally, regardless of whether they are part of the Erlang distribution or not.

15.6 Replication and Communication in Riak

Riak was designed for extreme reliability and availability at a massive scale, and was inspired by Amazon's Dynamo storage system [DHJ⁺07]. Dynamo and Riak's architectures combine aspects of both Distributed Hash Tables (DHTs) and traditional databases. Two key techniques that both Riak and Dynamo use are *consistent hashing* for replica placement and a *gossip protocol* for sharing common state.

Consistent hashing requires that all nodes in the system know about each other, and know what partitions each node owns. This assignment data could be maintained in a centrally managed configuration file, but in large configurations, this becomes extremely difficult. Another alternative is to use a central configuration server, but this introduces a single point of failure in the system. Instead, Riak uses a gossip protocol to propagate cluster membership and partition ownership data throughout the system.

Gossip protocols, also called epidemic protocols, work exactly as they sound. When a node in the system wishes to change a piece of shared data, it makes the change to its local copy of the data and gossips the updated data to a random peer. Upon receiving an update, a node merges the received changes with its local state and gossips again to another random peer.

When a Riak cluster is started, all nodes must be configured with the same partition count. The consistent hashing ring is then divided by the partition count and each interval is stored locally as a {HashRange, Owner} pair. The first node in a cluster simply claims all the partitions. When a new node joins the cluster, it contacts an existing node for its list of {HashRange, Owner} pairs. It then claims (partition count)/(number of nodes) pairs, updating its local state to reflect its new ownership. The updated ownership information is then gossiped to a peer. This updated state then spread throughout the entire cluster using the above algorithm.

By using a gossip protocol, Riak avoids introducing a single point of failure in the form of a centralized configuration server, relieving system operators from having to maintain critical cluster configuration data. Any node can then use the gossiped partition assignment data in the system to route requests. When used together, the gossip protocol and consistent hashing enable Riak to function as a truly decentralized system, which has important consequences for deploying and operating large-scale systems.

15.7 Conclusions and Lessons Learned

Most programmers believe that smaller and simpler codebases are not only easier to maintain, they often have fewer bugs. By using Erlang's basic distribution primitives for communication in a cluster, Riak can start out with a fundamentally sound asynchronous messaging layer and build its own protocols without having to worry about that underlying implementation. As Riak grew into a mature system, some aspects of its networked communication moved away from use of Erlang's built-in distribution (and toward direct manipulation of TCP sockets) while others remained a good fit for the included primitives. By starting out with Erlang's native message passing for everything, the Riak team was able to build out the whole system very quickly. These primitives are clean and clear enough that it was still easy later to replace the few places where they turned out to not be the best fit in production.

Also, due to the nature of Erlang messaging and the lightweight core of the Erlang VM, a user can just as easily run 12 nodes on 1 machine or 12 nodes on 12 machines. This makes development and testing much easier when compared to more heavyweight messaging and clustering mechanisms.

This has been especially valuable due to Riak’s fundamentally distributed nature. Historically, most distributed systems are very difficult to operate in a “development mode” on a single developer’s laptop. As a result, developers often end up testing their code in an environment that is a subset of their full system, with very different behavior. Since a many-node Riak cluster can be trivially run on a single laptop without excessive resource consumption or tricky configuration, the development process can more easily produce code that is ready for production deployment.

The use of Erlang/OTP supervisors makes Riak much more resilient in the face of subcomponent crashes. Riak takes this further; inspired by such behaviors, a Riak cluster is also able to easily keep functioning even when whole nodes crash and disappear from the system. This can lead to a sometimes-surprising level of resilience. One example of this was when a large enterprise was stress-testing various databases and intentionally crashing them to observe their edge conditions. When they got to Riak, they became confused. Each time they would find a way (through OS-level manipulation, bad IPC, etc) to crash a subsystem of Riak, they would see a very brief dip in performance and then the system returned to normal behavior. This is a direct result of a thoughtful “let it crash” approach. Riak was cleanly restarting each of these subsystems on demand, and the overall system simply continued to function. That experience shows exactly the sort of resilience enabled by Erlang/OTP’s approach to building programs.

Acknowledgments

This chapter is based on Francesco Cesarini and Simon Thompson’s 2009 lecture notes from the central European Functional Programming School held in Budapest and Komárno. Major contributions were made by Simon Thompson of the University of Kent in Canterbury, UK. A special thank you goes to all of the reviewers, who at different stages in the writing of this chapter provided valuable feedback.

Selenium WebDriver

Simon Stewart

Selenium is a browser automation tool, commonly used for writing end-to-end tests of web applications. A browser automation tool does exactly what you would expect: automate the control of a browser so that repetitive tasks can be automated. It sounds like a simple problem to solve, but as we will see, a lot has to happen behind the scenes to make it work.

Before describing the architecture of Selenium it helps to understand how the various related pieces of the project fit together. At a very high level, Selenium is a suite of three tools. The first of these tools, Selenium IDE, is an extension for Firefox that allows users to record and playback tests. The record/playback paradigm can be limiting and isn't suitable for many users, so the second tool in the suite, Selenium WebDriver, provides APIs in a variety of languages to allow for more control and the application of standard software development practices. The final tool, Selenium Grid, makes it possible to use the Selenium APIs to control browser instances distributed over a grid of machines, allowing more tests to run in parallel. Within the project, they are referred to as "IDE", "WebDriver" and "Grid". This chapter explores the architecture of Selenium WebDriver.

This chapter was written during the betas of Selenium 2.0 in late 2010. If you're reading the book after then, then things will have moved forward, and you'll be able to see how the architectural choices described here have unfolded. If you're reading before that date: Congratulations! You have a time machine. Can I have some winning lottery numbers?

16.1 History

Jason Huggins started the Selenium project in 2004 while working at ThoughtWorks on their in-house Time and Expenses (T&E) system, which made extensive use of Javascript. Although Internet Explorer was the dominant browser at the time, ThoughtWorks used a number of alternative browsers (in particular Mozilla variants) and would file bug reports when the T&E app wouldn't work on their browser of choice. Open Source testing tools at the time were either focused on a single browser (typically IE) or were simulations of a browser (like HttpUnit). The cost of a license for a commercial tool would have exhausted the limited budget for a small in-house project, so they weren't even considered as viable testing choices.

Where automation is difficult, it's common to rely on manual testing. This approach doesn't scale when the team is very small or when releases are extremely frequent. It's also a waste of humanity to

ask people to step through a script that could be automated. More prosaically, people are slower and more error prone than a machine for dull repetitive tasks. Manual testing wasn't an option.

Fortunately, all the browsers being tested supported Javascript. It made sense to Jason and the team he was working with to write a testing tool in that language which could be used to verify the behavior of the application. Inspired by work being done on FIT¹, a table-based syntax was placed over the raw Javascript and this allowed tests to be written by people with limited programming experience using a keyword-driven approach in HTML files. This tool, originally called "Selenium" but later referred to as "Selenium Core", was released under the Apache 2 license in 2004.

The table format of Selenium is structured similarly to the ActionFixture from FIT. Each row of the table is split into three columns. The first column gives the name of the command to execute, the second column typically contains an element identifier and the third column contains an optional value. For example, this is how to type the string "Selenium WebDriver" into an element identified with the name "q":

type	name=q	Selenium WebDriver

Because Selenium was written in pure Javascript, its initial design required developers to host Core and their tests on the same server as the application under test (AUT) in order to avoid falling foul of the browser's security policies and the Javascript sandbox. This was not always practical or possible. Worse, although a developer's IDE gives them the ability to swiftly manipulate code and navigate a large codebase, there is no such tool for HTML. It rapidly became clear that maintaining even a medium-sized suite of tests was an unwieldy and painful proposition.²

To resolve this and other issues, an HTTP proxy was written so that every HTTP request could be intercepted by Selenium. Using this proxy made it possible to side-step many of the constraints of the "same host origin" policy, where a browser won't allow Javascript to make calls to anything other than the server from which the current page has been served, allowing the first weakness to be mitigated. The design opened up the possibility of writing Selenium bindings in multiple languages: they just needed to be able to send HTTP requests to a particular URL. The wire format was closely modeled on the table-based syntax of Selenium Core and it, along with the table-based syntax, became known as "Selenese". Because the language bindings were controlling the browser at a distance, the tool was called "Selenium Remote Control", or "Selenium RC".

While Selenium was being developed, another browser automation framework was brewing at ThoughtWorks: WebDriver. The initial code for this was released early in 2007. WebDriver was derived from work on projects which wanted to isolate their end-to-end tests from the underlying test tool. Typically, the way that this isolation is done is via the Adapter pattern. WebDriver grew out of insight developed by applying this approach consistently over numerous projects, and initially was a wrapper around HtmlUnit. Internet Explorer and Firefox support followed rapidly after release.

When WebDriver was released there were significant differences between it and Selenium RC, though they sat in the same software niche of an API for browser automation. The most obvious difference to a user was that Selenium RC had a dictionary-based API, with all methods exposed on a single class, whereas WebDriver had a more object-oriented API. In addition, WebDriver only supported Java, whereas Selenium RC offered support for a wide-range of languages. There were also strong technical differences: Selenium Core (on which RC was based) was essentially a Javascript application, running inside the browser's security sandbox. WebDriver attempted to

¹<http://fit.c2.com>

²This is very similar to FIT, and James Shore, one of that project's coordinators, helps explain some of the drawbacks at <http://jamesshore.com/Blog/The-Problems-With-Acceptance-Testing.html>.

bind natively to the browser, side-stepping the browser's security model at the cost of significantly increased development effort for the framework itself.

In August, 2009, it was announced that the two projects would merge, and Selenium WebDriver is the result of those merged projects. As I write this, WebDriver supports language bindings for Java, C#, Python and Ruby. It offers support for Chrome, Firefox, Internet Explorer, Opera, and the Android and iPhone browsers. There are sister projects, not kept in the same source code repository but working closely with the main project, that provide Perl bindings, an implementation for the BlackBerry browser, and for “headless” WebKit—useful for those times where tests need to run on a continuous integration server without a proper display. The original Selenium RC mechanism is still maintained and allows WebDriver to provide support for browsers that would otherwise be unsupported.

16.2 A Digression About Jargon

Unfortunately, the Selenium project uses a lot of jargon. To recap what we've already come across:

- *Selenium Core* is the heart of the original Selenium implementation, and is a set of Javascript scripts that control the browser. This is sometimes referred to as “Selenium” and sometimes as “Core”.
- *Selenium RC* was the name given to the language bindings for Selenium Core, and is commonly, and confusingly, referred to as just “Selenium” or “RC”. It has now been replaced by Selenium WebDriver, where RC's API is referred to as the “Selenium 1.x API”.
- *Selenium WebDriver* fits in the same niche as RC did, and has subsumed the original 1.x bindings. It refers to both the language bindings and the implementations of the individual browser controlling code. This is commonly referred to as just “WebDriver” or sometimes as Selenium 2. Doubtless, this will be contracted to “Selenium” over time.

The astute reader will have noticed that “Selenium” is used in a fairly general sense. Fortunately, context normally makes it clear which particular Selenium people are referring to.

Finally, there's one more phrase which I'll be using, and there's no graceful way of introducing it: “driver” is the name given to a particular implementation of the WebDriver API. For example, there is a Firefox driver, and an Internet Explorer driver.

16.3 Architectural Themes

Before we start looking at the individual pieces to understand how they're wired together, it's useful to understand the the overarching themes of the architecture and development of the project. Succinctly put, these are:

- Keep the costs down.
- Emulate the user.
- Prove the drivers work...
- ...but you shouldn't need to understand how everything works.
- Lower the bus factor.
- Have sympathy for a Javascript implementation.
- Every method call is an RPC call.
- We are an Open Source project.

Keep the Costs Down

Supporting X browsers on Y platforms is inherently an expensive proposition, both in terms of initial development and maintenance. If we can find some way to keep the quality of the product high without violating too many of the other principles, then that's the route we favor. This is most clearly seen in our adoption of Javascript where possible, as you'll read about shortly.

Emulate the User

WebDriver is designed to accurately simulate the way that a user will interact with a web application. A common approach for simulating user input is to make use of Javascript to synthesize and fire the series of events that an app would see if a real user were to perform the same interaction. This "synthesized events" approach is fraught with difficulties as each browser, and sometimes different versions of the same browser, fire slightly different events with slightly different values. To complicate matters, most browsers won't allow a user to interact in this way with form elements such as file input elements for security reasons.

Where possible WebDriver uses the alternative approach of firing events at the OS level. As these "native events" aren't generated by the browser this approach circumvents the security restrictions placed on synthesized events and, because they are OS specific, once they are working for one browser on a particular platform reusing the code in another browser is relatively easy. Sadly, this approach is only possible where WebDriver can bind closely with the browser and where the development team have determined how best to send native events without requiring the browser window to be focused (as Selenium tests take a long time to run, and it's useful to be able to use the machine for other tasks as they run). At the time of writing, this means that native events can be used on Linux and Windows, but not Mac OS X.

No matter how WebDriver is emulating user input, we try hard to mimic user behavior as closely as possible. This in contrast to RC, which provided APIs that operated at a level far lower than that which a user works at.

Prove the Drivers Work

It may be an idealistic, "motherhood and apple pie" thing, but I believe there's no point in writing code if it doesn't work. The way we prove the drivers work on the Selenium project is to have an extensive set of automated test cases. These are typically "integration tests", requiring the code to be compiled and making use of a browser interacting with a web server, but where possible we write "unit tests", which, unlike an integration test can be run without a full recompilation. At the time of writing, there are about 500 integration tests and about 250 unit tests that could be run across each and every browser. We add more as we fix issues and write new code, and our focus is shifting to writing more unit tests.

Not every test is run against every browser. Some test specific capabilities that some browsers don't support, or which are handled in different ways on different browsers. Examples would include the tests for new HTML5 features which aren't supported on all browsers. Despite this, each of the major desktop browsers have a significant subset of tests run against them. Understandably, finding a way to run 500+ tests per browser on multiple platforms is a significant challenge, and it's one that the project continues to wrestle with.

You Shouldn't Need to Understand How Everything Works

Very few developers are proficient and comfortable in every language and technology we use. Consequently, our architecture needs to allow developers to focus their talents where they can do the most good, without needing them to work on pieces of the codebase where they are uncomfortable.

Lower the Bus Factor

There's a (not entirely serious) concept in software development called the "bus factor". It refers to the number of key developers who would need to meet some grisly end—presumably by being hit by a bus—to leave the project in a state where it couldn't continue. Something as complex as browser automation could be especially prone to this, so a lot of our architectural decisions are made to raise this number as high as possible.

Have Sympathy for a Javascript Implementation

WebDriver falls back to using pure Javascript to drive the browser if there is no other way of controlling it. This means that any API we add should be "sympathetic" to a Javascript implementation. As a concrete example, HTML5 introduces `LocalStorage`, an API for storing structured data on the client-side. This is typically implemented in the browser using `SQLite`. A natural implementation would have been to provide a database connection to the underlying data store, using something like `JDBC`. Eventually, we settled on an API that closely models the underlying Javascript implementation because something that modeled typical database access APIs wasn't sympathetic to a Javascript implementation.

Every Call Is an RPC Call

WebDriver controls browsers that are running in other processes. Although it's easy to overlook it, this means that every call that is made through its API is an RPC call and therefore the performance of the framework is at the mercy of network latency. In normal operation, this may not be terribly noticeable—most OSes optimize routing to `localhost`—but as the network latency between the browser and the test code increases, what may have seemed efficient becomes less so to both API designers and users of that API.

This introduces some tension into the design of APIs. A larger API, with coarser functions would help reduce latency by collapsing multiple calls, but this must be balanced by keeping the API expressive and easy to use. For example, there are several checks that need to be made to determine whether an element is visible to an end-user. Not only do we need to take into account various CSS properties, which may need to be inferred by looking at parent elements, but we should probably also check the dimensions of the element. A minimalist API would require each of these checks to be made individually. WebDriver collapses all of them into a single `isDisplayed` method.

Final Thought: This Is Open Source

Although it's not strictly an architectural point, Selenium is an Open Source project. The theme that ties all the above points together is that we'd like to make it as easy as possible for a new developer to contribute. By keeping the depth of knowledge required as shallow as possible, using as few languages as necessary and by relying on automated tests to verify that nothing has broken, we hopefully enable this ease of contribution.

Originally the project was split into a series of modules, with each module representing a particular browser with additional modules for common code and for support and utility code. Source trees for each binding were stored under these modules. This approach made a lot of sense for languages such as Java and C#, but was painful to work with for Rubyists and Pythonistas. This translated almost directly into relative contributor numbers, with only a handful of people able and interested to work on the Python and Ruby bindings. To address this, in October and November of 2010 the source code was reorganized with the Ruby and Python code stored under a single top-level directory per language. This more closely matched the expectations of Open Source developers in those languages, and the effect on contributions from the community was noticeable almost immediately.

16.4 Coping with Complexity

Software is a lumpy construct. The lumps are complexity, and as designers of an API we have a choice as where to push that complexity. At one extreme we could spread the complexity as evenly as possible, meaning that every consumer of the API needs to be party to it. The other extreme suggests taking as much of the complexity as possible and isolating it in a single place. That single place would be a place of darkness and terror for many if they have to venture there, but the trade-off is that users of the API, who need not delve into the implementation, have that cost of complexity paid up-front for them.

The WebDriver developers lean more towards finding and isolating the complexity in a few places rather than spreading it out. One reason for this is our users. They're exceptionally good at finding problems and issues, as a glance at our bug list shows, but because many of them are not developers a complex API isn't going to work well. We sought to provide an API that guides people in the right direction. As an example, consider the following methods from the original Selenium API, each of which can be used to set the value of an input element:

- type
- typeKeys
- typeKeysNative
- keydown
- keypress
- keyup
- keydownNative
- keypressNative
- keyupNative
- attachFile

Here's the equivalent in the WebDriver API:

- sendKeys

As discussed earlier, this highlights one of the major philosophical differences between RC and WebDriver in that WebDriver is striving to emulate the user, whereas RC offers APIs that deal at a lower level that a user would find hard or impossible to reach. The distinction between typeKeys and typeKeysNative is that the former always uses synthetic events, whereas the latter attempts to use the AWT Robot to type the keys. Disappointingly, the AWT Robot sends the key presses to whichever window has focus, which may not necessarily be the browser. WebDriver's native events, by contrast, are sent directly to the window handle, avoiding the requirement that the browser window have focus.

The WebDriver Design

The team refers to WebDriver’s API as being “object-based”. The interfaces are clearly defined and try to adhere to having only a single role or responsibility, but rather than modeling every single possible HTML tag as its own class we only have a single `WebElement` interface. By following this approach developers who are using an IDE which supports auto-completion can be led towards the next step to take. The result is that coding sessions may look like this (in Java):

```
WebDriver driver = new FirefoxDriver();  
driver.<user hits space>
```

At this point, a relatively short list of 13 methods to pick from appears. The user selects one:

```
driver.findElement(<user hits space>)
```

Most IDEs will now drop a hint about the type of the argument expected, in this case a “By”. There are a number of preconfigured factory methods for “By” objects declared as static methods on the `By` itself. Our user will quickly end up with a line of code that looks like:

```
driver.findElement(By.id("some_id"));
```

I find `UnsupportedOperationException`s and their ilk deeply displeasing, but there needs to be something that allows functionality to be exposed for the subset of users who might need it without cluttering the rest of the APIs for the majority of users. To this end, WebDriver makes extensive use of role-based interfaces. For example, there is a `JavascriptExecutor` interface that provides the ability to execute arbitrary chunks of Javascript in the context of the current page. A successful cast of a `WebDriver` instance to that interface indicates that you can expect the methods on it to work.

Dealing with the Combinatorial Explosion

One of the first things that is apparent from a moment’s thought about the wide range of browsers and languages that WebDriver supports is that unless care is taken it would quickly face an escalating cost of maintenance. With X browsers and Y languages, it would be very easy to fall into the trap of maintaining $X \times Y$ implementations.

Reducing the number of languages that WebDriver supports would be one way to reduce this cost, but we don’t want to go down this route for two reasons. Firstly, there’s a cognitive load to be paid when switching from one language to another, so it’s advantageous to users of the framework to be able to write their tests in the same language that they do the majority of their development work in. Secondly, mixing several languages on a single project is something that teams may not be comfortable with, and corporate coding standards and requirements often seem to demand a technology monoculture (although, pleasingly, I think that this second point is becoming less true over time), therefore reducing the number of supported languages isn’t an available option.

Reducing the number of supported browsers also isn’t an option—there were vociferous arguments when we phased out support for Firefox 2 in WebDriver, despite the fact that when we made this choice it represented less than 1% of the browser market.

The only choice we have left is to try and make all the browsers look identical to the language bindings: they should offer a uniform interface that can be addressed easily in a wide variety of languages. What is more, we want the language bindings themselves to be as easy to write as possible, which suggests that we want to keep them as slim as possible. We push as much logic as we can

Role-based Interfaces

Think of a simplified Shop class. Every day, it needs to be restocked, and it collaborates with a Stockist to deliver this new stock. Every month, it needs to pay staff and taxes. For the sake of argument, let's assume that it does this using an Accountant. One way of modeling this looks like:

```
public interface Shop {  
    void addStock(StockItem item, int quantity);  
    Money getSalesTotal(Date startDate, Date endDate);  
}
```

We have two choices about where to draw the boundaries when defining the interface between the Shop, the Accountant and the Stockist. We could draw a theoretical line as shown in Figure 16.1.

This would mean that both Accountant and Stockist would accept a Shop as an argument to their respective methods. The drawback here, though, is that it's unlikely that the Accountant really wants to stack shelves, and it's probably not a great idea for the Stockist to realize the vast mark-up on prices that the Shop is adding. So, a better place to draw the line is shown in Figure 16.2.

We'll need two interfaces that the Shop needs to implement, but these interfaces clearly define the role that the Shop fulfills for both the Accountant and the Stockist. They are role-based interfaces:

```
public interface HasBalance {  
    Money getSalesTotal(Date startDate, Date endDate);  
}  
  
public interface Stockable {  
    void addStock(StockItem item, int quantity);  
}  
  
public interface Shop extends HasBalance, Stockable {  
}
```

into the underlying driver in order to support this: every piece of functionality we fail to push into the driver is something that needs to be implemented in every language we support, and this can represent a significant amount of work.

As an example, the IE driver has successfully pushed the responsibility for locating and starting IE into the main driver logic. Although this has resulted in a surprising number of lines of code being in the driver, the language binding for creating a new instance boils down to a single method call into that driver. For comparison, the Firefox driver has failed to make this change. In the Java world alone, this means that we have three major classes that handle configuring and starting Firefox weighing in at around 1300 lines of code. These classes are duplicated in every language binding that wants to support the FirefoxDriver without relying on starting a Java server. That's a lot of additional code to maintain.

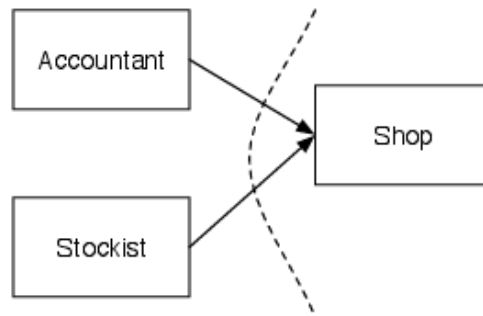


Figure 16.1: Accountant and Stockist Depend on Shop

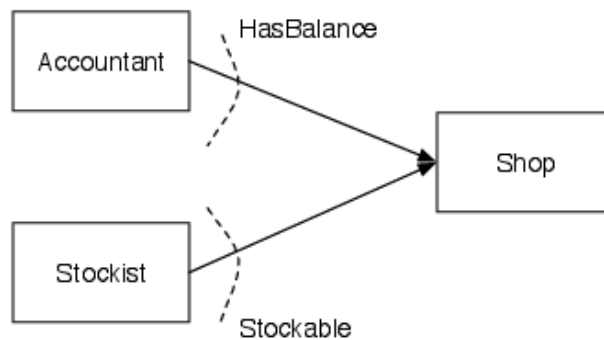


Figure 16.2: Shop Implements HasBalance and Stockable

Flaws in the WebDriver Design

The downside of the decision to expose capabilities in this way is that until someone knows that a particular interface exists they may not realize that WebDriver supports that type of functionality; there's a loss of explorability in the API. Certainly when WebDriver was new we seemed to spend a lot of time just pointing people towards particular interfaces. We've now put a lot more effort into our documentation and as the API gets more widely used it becomes easier and easier for users to find the information they need.

There is one place where I think our API is particularly poor. We have an interface called `RenderedWebElement` which has a strange mish-mash of methods to do with querying the rendered state of the element (`isDisplayed`, `getSize` and `getLocation`), performing operations on it (hover and drag and drop methods), and a handy method for getting the value of a particular CSS property. It was created because the `HtmlUnit` driver didn't expose the required information, but the `Firefox` and `IE` drivers did. It originally only had the first set of methods but we added the other methods before I'd done hard thinking about how I wanted the API to evolve. The interface is well known now, and the tough choice is whether we keep this unsightly corner of the API given that it's widely used, or whether we attempt to delete it. My preference is not to leave a "broken window" behind, so fixing this before we release Selenium 2.0 is important. As a result, by the time you read this chapter, `RenderedWebElement` may well be gone.

From an implementor's point of view, binding tightly to a browser is also a design flaw, albeit an inescapable one. It takes significant effort to support a new browser, and often several attempts need to be made in order to get it right. As a concrete example, the Chrome driver has gone through four complete rewrites, and the IE driver has had three major rewrites too. The advantage of binding tightly to a browser is that it offers more control.

16.5 Layers and Javascript

A browser automation tool is essentially built of three moving parts:

- A way of interrogating the DOM.
- A mechanism for executing Javascript.
- Some means of emulating user input.

This section focuses on the first part: providing a mechanism to interrogate the DOM. The lingua franca of the browser is Javascript, and this seems like the ideal language to use when interrogating the DOM. Although this choice seems obvious, making it leads to some interesting challenges and competing requirements that need balancing when thinking about Javascript.

Like most large projects, Selenium makes use of a layered set of libraries. The bottom layer is Google's Closure Library, which supplies primitives and a modularization mechanism allowing source files to be kept focused and as small as possible. Above this, there is a utility library providing functions that range from simple tasks such as getting the value of an attribute, through determining whether an element would be visible to an end user, to far more complex actions such as simulating a click using synthesized events. Within the project, these are viewed as offering the smallest units of browser automation, and so are called Browser Automation Atoms or atoms. Finally, there are adapter layers that compose atoms in order to meet the API contracts of both WebDriver and Core.

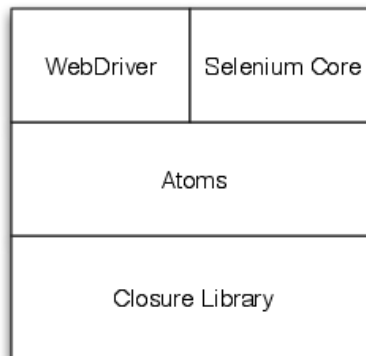


Figure 16.3: Layers of Selenium Javascript Library

The Closure Library was chosen for several reasons. The main one was that the Closure Compiler understands the modularization technique the Library uses. The Closure Compiler is a compiler targeting Javascript as the output language. “Compilation” can be as simple as ordering input files in dependency order, concatenating and pretty printing them, or as complex as doing advanced

minification and dead code removal. Another undeniable advantage was that several members of the team doing the work on the Javascript code were very familiar with Closure Library.

This “atomic” library of code is used pervasively throughout the project when there is a requirement to interrogate the DOM. For RC and those drivers largely composed of Javascript, the library is used directly, typically compiled as a monolithic script. For drivers written in Java, individual functions from the WebDriver adapter layer are compiled with full optimization enabled, and the generated Javascript included as resources in the JARs. For drivers written in C variants, such as the iPhone and IE drivers, not only are the individual functions compiled with full optimization, but the generated output is converted to a constant defined in a header which is executed via the driver’s normal Javascript execution mechanism on demand. Although this seems like a strange thing to do, it allows the Javascript to be pushed into the underlying driver without needing to expose the raw source in multiple places.

Because the atoms are used pervasively it’s possible to ensure consistent behavior between the different browsers, and because the library is written in Javascript and doesn’t require elevated privileges to execute the development cycle, is easy and fast. The Closure Library can load dependencies dynamically, so the Selenium developer need only write a test and load it in a browser, modifying code and hitting the refresh button as required. Once the test is passing in one browser, it’s easy to load it in another browser and confirm that it passes there. Because the Closure Library does a good job of abstracting away the differences between browsers, this is often enough, though it’s reassuring to know that there are continuous builds that will run the test suite in every supported browser.

Originally Core and WebDriver had many areas of congruent code—code that performed the same function in slightly different ways. When we started work on the atoms, this code was combed through to try and find the “best of breed” functionality. After all, both projects had been used extensively and their code was very robust so throwing away everything and starting from scratch would not only have been wasteful but foolish. As each atom was extracted, the sites at which it would be used were identified and switched to using the atom. For example, the Firefox driver’s `getAttribute` method shrunk from approximately 50 lines of code to 6 lines long, including blank lines:

```
FirefoxDriver.prototype.getElementAttribute =
function(respond, parameters) {
    var element = Utils.getElementAt(parameters.id,
                                    respond.session.getDocument());
    var attributeName = parameters.name;

    respond.value = webdriver.element.getAttribute(element, attributeName);
    respond.send();
};
```

That second-to-last line, where `respond.value` is assigned to, is using the atomic WebDriver library.

The atoms are a practical demonstration of several of the architectural themes of the project. Naturally they enforce the requirement that an implementation of an API be sympathetic to a Javascript implementation. What’s even better is that the same library is shared throughout the codebase; where once a bug had to be verified and fixed across multiple implementations, it is now enough to fix the bug in one place, which reduces the cost of change while improving stability and effectiveness. The atoms also make the bus factor of the project more favorable. Since a normal Javascript unit test can be used to check that a fix works the barrier to joining the Open Source project is considerably lower than it was when knowledge of how each driver was implemented was required.

There is another benefit to using the atoms. A layer emulating the existing RC implementation but backed by WebDriver is an important tool for teams looking to migrate in a controlled fashion to the newer WebDriver APIs. As Selenium Core is atomized it becomes possible to compile each function from it individually, making the task of writing this emulating layer both easier to implement and more accurate.

It goes without saying that there are downsides to the approach taken. Most importantly, compiling Javascript to a C const is a very strange thing to do, and it always baffles new contributors to the project who want to work on the C code. It is also a rare developer who has every version of every browser and is dedicated enough to run every test in all of those browsers—it is possible for someone to inadvertently cause a regression in an unexpected place, and it can take some time to identify the problem, particularly if the continuous builds are being flaky.

Because the atoms normalize return values between browsers, there can also be unexpected return values. For example, consider this HTML:

```
<input name="example" checked>
```

The value of the checked attribute will depend on the browser being used. The atoms normalize this, and other Boolean attributes defined in the HTML5 spec, to be “true” or “false”. When this atom was introduced to the code base, we discovered many places where people were making browser-dependent assumptions about what the return value should be. While the value was now consistent there was an extended period where we explained to the community what had happened and why.

16.6 The Remote Driver, and the Firefox Driver in Particular

The remote WebDriver was originally a glorified RPC mechanism. It has since evolved into one of the key mechanisms we use to reduce the cost of maintaining WebDriver by providing a uniform interface that language bindings can code against. Even though we’ve pushed as much of the logic as we can out of the language bindings and into the driver, if each driver needed to communicate via a unique protocol we would still have an enormous amount of code to repeat across all the language bindings.

The remote WebDriver protocol is used wherever we need to communicate with a browser instance that’s running out of process. Designing this protocol meant taking into consideration a number of concerns. Most of these were technical, but, this being open source, there was also the social aspect to consider.

Any RPC mechanism is split into two pieces: the transport and the encoding. We knew that however we implemented the remote WebDriver protocol, we would need support for both pieces in the languages we wanted to use as clients. The first iteration of the design was developed as part of the Firefox driver.

Mozilla, and therefore Firefox, was always seen as being a multi-platform application by its developers. In order to facilitate the development, Mozilla created a framework inspired by Microsoft’s COM that allowed components to be built and bolted together called XPCOM (cross-platform COM). An XPCOM interface is declared using IDL, and there are language bindings for C and Javascript as well as other languages. Because XPCOM is used to construct Firefox, and because XPCOM has Javascript bindings, it’s possible to make use of XPCOM objects in Firefox extensions.

Normal Win32 COM allows interfaces to be accessed remotely. There were plans to add the same ability to XPCOM too, and Darin Fisher added an XPCOM ServerSocket implementation to facilitate this. Although the plans for D-XPCOM never came to fruition, like an appendix, the

vestigial infrastructure is still there. We took advantage of this to create a very basic server within a custom Firefox extension containing all the logic for controlling Firefox. The protocol used was originally text-based and line-oriented, encoding all strings as UTF-2. Each request or response began with a number, indicating how many newlines to count before concluding that the request or reply had been sent. Crucially, this scheme was easy to implement in Javascript as SeaMonkey (Firefox's Javascript engine at the time) stores Javascript strings internally as 16 bit unsigned integers.

Although futzing with custom encoding protocols over raw sockets is a fun way to pass the time, it has several drawbacks. There were no widely available libraries for the custom protocol, so it needed to be implemented from the ground up for every language that we wanted to support. This requirement to implement more code would make it less likely that generous Open Source contributors would participate in the development of new language bindings. Also, although a line-oriented protocol was fine when we were only sending text-based data around, it brought problems when we wanted to send images (such as screenshots) around.

It became very obvious, very quickly that this original RPC mechanism wasn't practical. Fortunately, there was a well-known transport that has widespread adoption and support in almost every language that would allow us to do what we wanted: HTTP.

Once we had decided to use HTTP for a transport mechanism, the next choice that needed to be made was whether to use a single end-point (à la SOAP) or multiple end points (in the style of REST). The original Selenese protocol used a single end-point and had encoded commands and arguments in the query string. While this approach worked well, it didn't "feel" right: we had visions of being able to connect to a remote WebDriver instance in a browser to view the state of the server. We ended up choosing an approach we call "REST-ish": multiple end-point URLs using the verbs of HTTP to help provide meaning, but breaking a number of the constraints required for a truly RESTful system, notably around the location of state and cacheability, largely because there is only one location for the application state to meaningfully exist.

Although HTTP makes it easy to support multiple ways of encoding data based on content type negotiation, we decided that we needed a canonical form that all implementations of the remote WebDriver protocol could work with. There were a handful of obvious choices: HTML, XML or JSON. We quickly ruled out XML: although it's a perfectly reasonable data format and there are libraries that support it for almost every language, my perception of how well-liked it is in the Open Source community was that people don't enjoy working with it. In addition, it was entirely possible that although the returned data would share a common "shape" it would be easy for additional fields to be added³. Although these extensions could be modeled using XML namespaces this would start to introduce Yet More Complexity into the client code: something I was keen to avoid. XML was discarded as an option. HTML wasn't really a good choice, as we needed to be able to define our own data format, and though an embedded micro-format could have been devised and used that seems like using a hammer to crack an egg.

The final possibility considered was Javascript Object Notation (JSON). Browsers can transform a string into an object using either a straight call to `eval` or, on more recent browsers, with primitives designed to transform a Javascript object to and from a string securely and without side-effects. From a practical perspective, JSON is a popular data format with libraries for handling it available for almost every language and all the cool kids like it. An easy choice.

The second iteration of the remote WebDriver protocol therefore used HTTP as the transport mechanism and UTF-8 encoded JSON as the default encoding scheme. UTF-8 was picked as the default encoding so that clients could easily be written in languages with limited support for Unicode,

³For example, the remote server returns a base64-encoded screen grab with every exception as a debugging aid but the Firefox driver doesn't.

as UTF-8 is backwardly compatible with ASCII. Commands sent to the server used the URL to determine which command was being sent, and encoded the parameters for the command in an array.

For example a call to `WebDriver.get("http://www.example.com")` mapped to a POST request to a URL encoding the session ID and ending with `"/url"`, with the array of parameters looking like `['http://www.example.com']`. The returned result was a little more structured, and had place-holders for a returned value and an error code. It wasn't long until the third iteration of remote protocol, which replaced the request's array of parameters with a dictionary of named parameters. This had the benefit of making debugging requests significantly easier, and removed the possibility of clients mistakenly mis-ordering parameters, making the system as a whole more robust. Naturally, it was decided to use normal HTTP error codes to indicate certain return values and responses where they were the most appropriate way to do so; for example, if a user attempts to call a URL with nothing mapped to it, or when we want to indicate the "empty response".

The remote WebDriver protocol has two levels of error handling, one for invalid requests, and one for failed commands. An example of an invalid request is for a resource that doesn't exist on the server, or perhaps for a verb that the resource doesn't understand (such as sending a DELETE command to the the resource used for dealing with the URL of the current page) In those cases, a normal HTTP 4xx response is sent. For a failed command, the responses error code is set to 500 ("Internal Server Error") and the returned data contains a more detailed breakdown of what went wrong.

When a response containing data is sent from the server, it takes the form of a JSON object:

Key	Description
sessionId	An opaque handle used by the server to determine where to route session-specific commands.
status	A numeric status code summarizing the result of the command. A non-zero value indicates that the command failed.
value	The response JSON value.

An example response would be:

```
{
  sessionId: 'BD204170-1A52-49C2-A6F8-872D127E7AE8',
  status: 7,
  value: 'Unable to locate element with id: foo'
}
```

As can be seen, we encode status codes in the response, with a non-zero value indicating that something has gone horribly awry. The IE driver was the first to use status codes, and the values used in the wire protocol mirror these. Because all error codes are consistent between drivers, it is possible to share error handling code between all the drivers written in a particular language, making the job of the client-side implementors easier.

The Remote WebDriver Server is simply a Java servlet that acts as a multiplexer, routing any commands it receives to an appropriate WebDriver instance. It's the sort of thing that a second year graduate student could write. The Firefox driver also implements the remote WebDriver protocol, and its architecture is far more interesting, so let's follow a request through from the call in the language bindings to that back-end until it returns to the user.

Assuming that we're using Java, and that "element" is an instance of `WebElement`, it all starts here:

```
element.getAttribute("row");
```

Internally, the element has an opaque "id" that the server-side uses to identify which element we're talking about. For the sake of this discussion, we'll imagine it has the value "some_opaque_id". This is encoded into a Java Command object with a Map holding the (now named) parameters id for the element ID and name for the name of the attribute being queried.

A quick look up in a table indicates that the correct URL is:

```
/session/:sessionId/element/:id/attribute/:name
```

Any section of the URL that begins with a colon is assumed to be a variable that requires substitution. We've been given the id and name parameters already, and the sessionId is another opaque handle that is used for routing when a server can handle more than one session at a time (which the Firefox driver cannot). This URL therefore typically expands to something like:

```
http://localhost:7055/hub/session/XXX/element/some_opaque_id/attribute/row
```

As an aside, WebDriver's remote wire protocol was originally developed at the same time as URL Templates were proposed as a draft RFC. Both our scheme for specifying URLs and URL Templates allow variables to be expanded (and therefore derived) within a URL. Sadly, although URL Templates were proposed at the same time, we only became aware of them relatively late in the day, and therefore they are not used to describe the wire protocol.

Because the method we're executing is idempotent⁴, the correct HTTP method to use is a GET. We delegate down to a Java library that can handle HTTP (the Apache HTTP Client) to call the server.

The Firefox driver is implemented as a Firefox extension, the basic design of which is shown in Figure 16.4. Somewhat unusually, it has an embedded HTTP server. Although originally we used one that we had built ourselves, writing HTTP servers in XPCOM wasn't one of our core competencies, so when the opportunity arose we replaced it with a basic HTTPD written by Mozilla themselves. Requests are received by the HTTPD and almost straight away passed to a dispatcher object.

The dispatcher takes the request and iterates over a known list of supported URLs, attempting to find one that matches the request. This matching is done with knowledge of the variable interpolation that went on in the client side. Once an exact match is found, including the verb being used, a JSON object, representing the command to execute, is constructed. In our case it looks like:

```
{
  'name': 'getElementAttribute',
  'sessionId': { 'value': 'XXX' },
  'parameters': {
    'id': 'some_opaque_key',
    'name': 'rows'
  }
}
```

⁴I.e., always returns the same result.

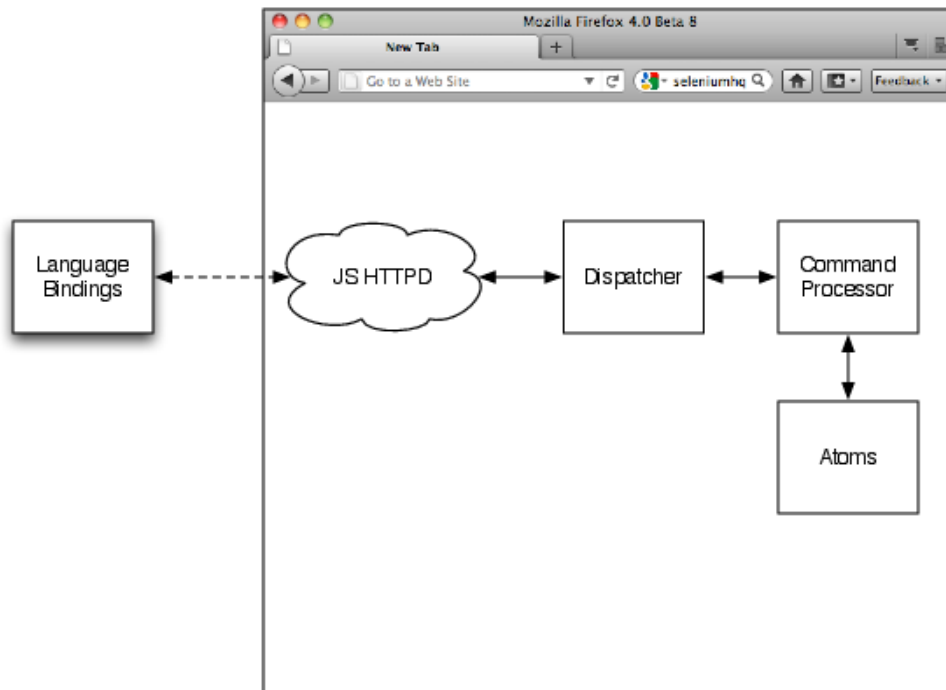


Figure 16.4: Overview of the Firefox Driver Architecture

This is then passed as a JSON string to a custom XPCOM component we've written called the `CommandProcessor`. Here's the code:

```

var jsonResponseString = JSON.stringify(json);
var callback = function(jsonResponseString) {
    var jsonResponse = JSON.parse(jsonResponseString);

    if (jsonResponse.status != ErrorCode.SUCCESS) {
        response.setStatus(Response.INTERNAL_ERROR);
    }

    response.setContentType('application/json');
    response.setBody(jsonResponseString);
    response.commit();
};

// Dispatch the command.
Components.classes['@googlecode.com/webdriver/command-processor;1'].
    getService(Components.interfaces.nsICommandProcessor).
    execute(jsonString, callback);

```

There's quite a lot of code here, but there are two key points. First, we converted the object above to a JSON string. Secondly, we pass a callback to the execute method that causes the HTTP response to be sent.

The execute method of the command processor looks up the “name” to determine which function to call, which it then does. The first parameter given to this implementing function is a respond” object (so called because it was originally just the function used to send the response back to the user), which encapsulates not only the possible values that might be sent, but also has a method that allows the response to be dispatched back to the user and mechanisms to find out information about the DOM. The second parameter is the value of the parameters object seen above (in this case, id and name). The advantage of this scheme is that each function has a uniform interface that mirrors the structure used on the client side. This means that the mental models used for thinking about the code on each side are similar. Here's the underlying implementation of getAttribute, which you've seen before in Section 16.5:

```
FirefoxDriver.prototype.getElementAttribute = function(respond, parameters) {
    var element = Utils.getElementAt(parameters.id,
                                    respond.session.getDocument());
    var attributeName = parameters.name;

    respond.value = webdriver.element.getAttribute(element, attributeName);
    respond.send();
};
```

In order to make element references consistent, the first line simply looks up the element referred to by the opaque ID in a cache. In the Firefox driver, that opaque ID is a UUID and the “cache” is simply a map. The getElementAt method also checks to see if the referred to element is both known and attached to the DOM. If either check fails, the ID is removed from the cache (if necessary) and an exception is thrown and returned to the user.

The second line from the end makes use of the browser automation atoms discussed earlier, this time compiled as a monolithic script and loaded as part of the extension.

In the final line, the send method is called. This does a simple check to ensure that we only send a response once before it calls the callback given to the execute method. The response is sent back to the user in the form of a JSON string, which is decanted into an object that looks like (assuming that getAttribute returned “7”, meaning the element wasn't found):

```
{
  'value': '7',
  'status': 0,
  'sessionId': 'XXX'
}
```

The Java client then checks the value of the status field. If that value is non-zero, it converts the numeric status code into an exception of the correct type and throws that, using the “value” field to help set the message sent to the user. If the status is zero the value of the “value” field is returned to the user.

Most of this makes a certain amount of sense, but there was one piece that an astute reader will raise questions about: why did the dispatcher convert the object it had into a string before calling the execute method?

The reason for this is that the Firefox Driver also supports running tests written in pure Javascript. Normally, this would be an extremely difficult thing to support: the tests are running in the context of

the browser's Javascript security sandbox, and so may not do a range of things that are useful in tests, such as traveling between domains or uploading files. The WebDriver Firefox extension, however, provides an escape hatch from the sandbox. It announces its presence by adding a `webdriver` property to the document element. The WebDriver Javascript API uses this as an indicator that it can add JSON serialized command objects as the value of a `command` property on the document element, fire a custom `webdriverCommand` event and then listen for a `webdriverResponse` event on the same element to be notified that the response property has been set.

This suggests that browsing the web in a copy of Firefox with the WebDriver extension installed is a seriously bad idea as it makes it trivially easy for someone to remotely control the browser.

Behind the scenes, there is a DOM messenger, waiting for the `webdriverCommand` this reads the serialized JSON object and calls the `execute` method on the command processor. This time, the callback is one that simply sets the `response` attribute on the document element and then fires the expected `webdriverResponse` event.

16.7 The IE Driver

Internet Explorer is an interesting browser. It's constructed of a number of COM interfaces working in concert. This extends all the way into the Javascript engine, where the familiar Javascript variables actually refer to underlying COM instances. That Javascript window is an `IHTMLWindow`. `document` is an instance of the COM interface `IHTMLDocument`. Microsoft have done an excellent job in maintaining existing behavior as they enhanced their browser. This means that if an application worked with the COM classes exposed by IE6 it will still continue to work with IE9.

The Internet Explorer driver has an architecture that's evolved over time. One of the major forces upon its design has been a requirement to avoid an installer. This is a slightly unusual requirement, so perhaps needs some explanation. The first reason not to require an installer is that it makes it harder for WebDriver to pass the "5 minute test", where a developer downloads a package and tries it out for a brief period of time. More importantly, it is relatively common for users of WebDriver to not be able to install software on their own machines. It also means that no-one needs to remember to log on to the continuous integration servers to run an installer when a project wants to start testing with IE. Finally, running installers just isn't in the culture of some languages. The common Java idiom is to simply drop JAR files on to the `CLASSPATH`, and, in my experience, those libraries that require installers tend not to be as well-liked or used.

So, no installer. There are consequences to this choice.

The natural language to use for programming on Windows would be something that ran on .Net, probably C#. The IE driver integrates tightly with IE by making use of the IE COM Automation interfaces that ship with every version of Windows. In particular, we use COM interfaces from the native `MSHTML` and `ShDocVw` DLLs, which form part of IE. Prior to C# 4, CLR/COM interoperability was achieved via the use of separate Primary Interop Assemblies (PIAs) A PIA is essentially a generated bridge between the managed world of the CLR and that of COM.

Sadly, using C# 4 would mean using a very modern version of the .Net runtime, and many companies avoid living on the leading edge, preferring the stability and known issues of older releases. By using C# 4 we would automatically exclude a reasonable percentage of our user-base. There are also other disadvantages to using a PIA. Consider licensing restrictions. After consultation with Microsoft, it became clear that the Selenium project would not have the rights to distribute the PIAs of either the `MSHTML` or `ShDocVw` libraries. Even if those rights had been granted, each installation of Windows and IE has a unique combination of these libraries, which means that we

would have needed to ship a vast number of these things. Building the PIAs on the client machine on demand is also a non-starter, as they require developer tools that may not exist on a normal user's machine.

So, although C# would have been an attractive language to do the bulk of the coding in, it wasn't an option. We needed to use something native, at least for the communication with IE. The next natural choice for this is C++, and this is the language that we chose in the end. Using C++ has the advantage that we don't need to use PIAs, but it does mean that we need to redistribute the Visual Studio C++ runtime DLL unless we statically link against them. Since we'd need to run an installer in order to make that DLL available, we statically link our library for communicating with IE.

That's a fairly high cost to pay for a requirement not to use an installer. However, going back to the theme of where complexity should live, it is worth the investment as it makes our users' lives considerably easier. It is a decision we re-evaluate on an ongoing basis, as the benefit to the user is a trade-off with the fact that the pool of people able to contribute to an advanced C++ Open Source project seems significantly smaller than those able to contribute to an equivalent C# project.

The initial design of the IE driver is shown in Figure 16.5.

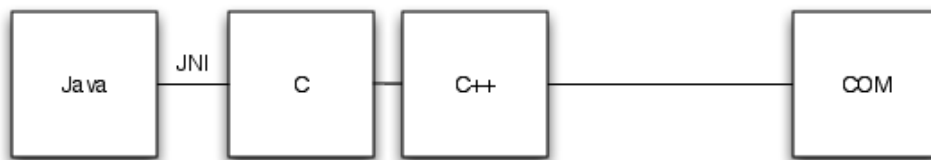


Figure 16.5: Original IE Driver

Starting from the bottom of that stack, you can see that we're using IE's COM Automation interfaces. In order to make these easier to deal with on a conceptual level, we wrapped those raw interfaces with a set of C++ classes that closely mirrored the main WebDriver API. In order to get the Java classes communicating with the C++ we made use of JNI, with the implementations of the JNI methods using the C++ abstractions of the COM interfaces.

This approach worked reasonably well while Java was the only client language, but it would have been a source of pain and complexity if each language we supported needed us to alter the underlying library. Thus, although JNI worked, it didn't provide the correct level of abstraction.

What was the correct level of abstraction? Every language that we wanted to support had a mechanism for calling down to straight C code. In C#, this takes the form of PInvoke. In Ruby there is FFI, and Python has ctypes. In the Java world, there is an excellent library called JNA (Java Native Architecture). We needed to expose our API using this lowest common denominator. This was done by taking our object model and flattening it, using a simple two or three letter prefix to indicate the "home interface" of the method: "wd" for "WebDriver" and "wde" for WebDriver Element. Thus `WebDriver.get` became `wdGet`, and `WebElement.getText` became `wdeGetText`. Each method returns an integer representing a status code, with "out" parameters being used to allow functions to return more meaningful data. Thus we ended up with method signatures such as:

```
int wdeGetAttribute(WebDriver*, WebElement*, const wchar_t*, StringWrapper**)
```

To calling code, the `WebDriver`, `WebElement` and `StringWrapper` are opaque types: we expressed the difference in the API to make it clear what value should be used as that parameter, though could

just as easily have been “void *”. You can also see that we were using wide characters for text, since we wanted to deal with internationalized text properly.

On the Java side, we exposed this library of functions via an interface, which we then adapted to make it look like the normal object-oriented interface presented by WebDriver. For example, the Java definition of the `getAttribute` method looks like:

```
public String getAttribute(String name) {
    PointerByReference wrapper = new PointerByReference();
    int result = lib.wdeGetAttribute(
        parent.getDriverPointer(), element, new WString(name), wrapper);

    errors.verifyErrorCode(result, "get attribute of");

    return wrapper.getValue() == null ? null : new StringWrapper(lib, wrapper).toString();
}
```

This lead to the design shown in Figure 16.6.

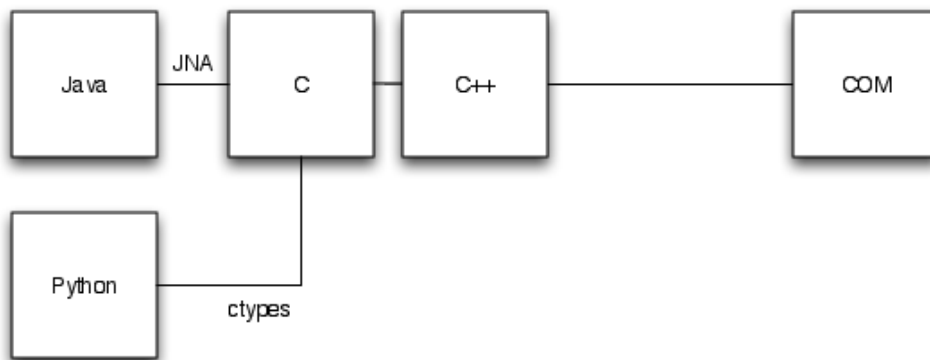


Figure 16.6: Modified IE Driver

While all the tests were running on the local machine, this worked out well, but once we started using the IE driver in the remote WebDriver we started running into random lock ups. We traced this problem back to a constraint on the IE COM Automation interfaces. They are designed to be used in a “Single Thread Apartment” model. Essentially, this boils down to a requirement that we call the interface from the same thread every time. While running locally, this happens by default. Java app servers, however, spin up multiple threads to handle the expected load. The end result? We had no way of being sure that the same thread would be used to access the IE driver in all cases.

One solution to this problem would have been to run the IE driver in a single-threaded executor and serialize all access via Futures in the app server, and for a while this was the design we chose. However, it seemed unfair to push this complexity up to the calling code, and it’s all too easy to imagine instances where people accidentally make use of the IE driver from multiple threads. We decided to sink the complexity down into the driver itself. We did this by holding the IE instance in a separate thread and using the `PostThreadMessage` Win32 API to communicate across the thread boundary. Thus, at the time of writing, the design of the IE driver looks like Figure 16.7.

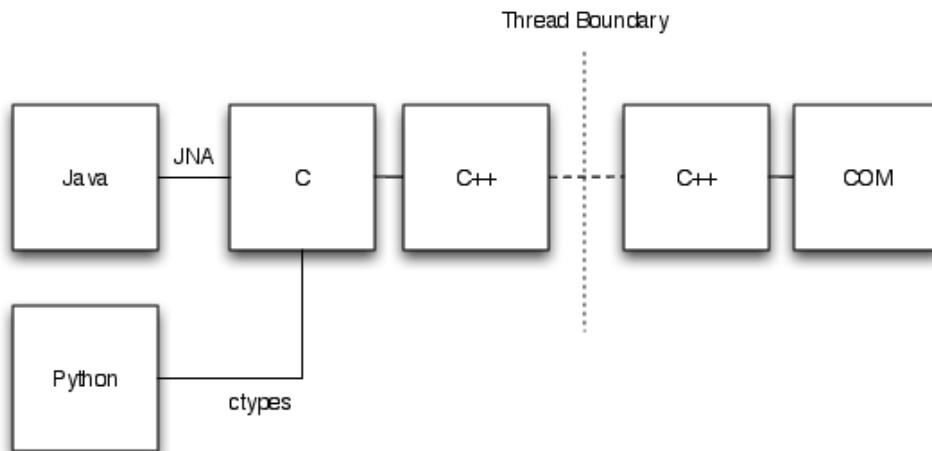


Figure 16.7: IE Driver as of Selenium 2.0 alpha 7

This isn't the sort of design that I would have chosen voluntarily, but it has the advantage of working and surviving the horrors that our users may chose to inflict upon it.

One drawback to this design is that it can be hard to determine whether the IE instance has locked itself solid. This may happen if a modal dialog opens while we're interacting with the DOM, or it may happen if there's a catastrophic failure on the far side of the thread boundary. We therefore have a timeout associated with every thread message we post, and this is set to what we thought was a relatively generous 2 minutes. From user feedback on the mailing lists, this assumption, while generally true, isn't always correct, and later versions of the IE driver may well make the timeout configurable.

Another drawback is that debugging the internals can be deeply problematic, requiring a combination of speed (after all, you've got two minutes to trace the code through as far as possible), the judicious use of break points and an understanding of the expected code path that will be followed across the thread boundary. Needless to say, in an Open Source project with so many other interesting problems to solve, there is little appetite for this sort of grungy work. This significantly reduces the bus factor of the system, and as a project maintainer, this worries me.

To address this, more and more of the IE driver is being moved to sit upon the same Automation Atoms as the Firefox driver and Selenium Core. We do this by compiling each of the atoms we plan to use and preparing it as a C++ header file, exposing each function as a constant. At runtime, we prepare the Javascript to execute from these constants. This approach means that we can develop and test a reasonable percentage of code for the IE driver without needing a C compiler involved, allowing far more people to contribute to finding and resolving bugs. In the end, the goal is to leave only the interaction APIs in native code, and rely on the atoms as much as possible.

Another approach we're exploring is to rewrite the IE driver to make use of a lightweight HTTP server, allowing us to treat it as a remote WebDriver. If this occurs, we can remove a lot of the complexity introduced by the thread boundary, reducing the total amount of code required and making the flow of control significantly easier to follow.

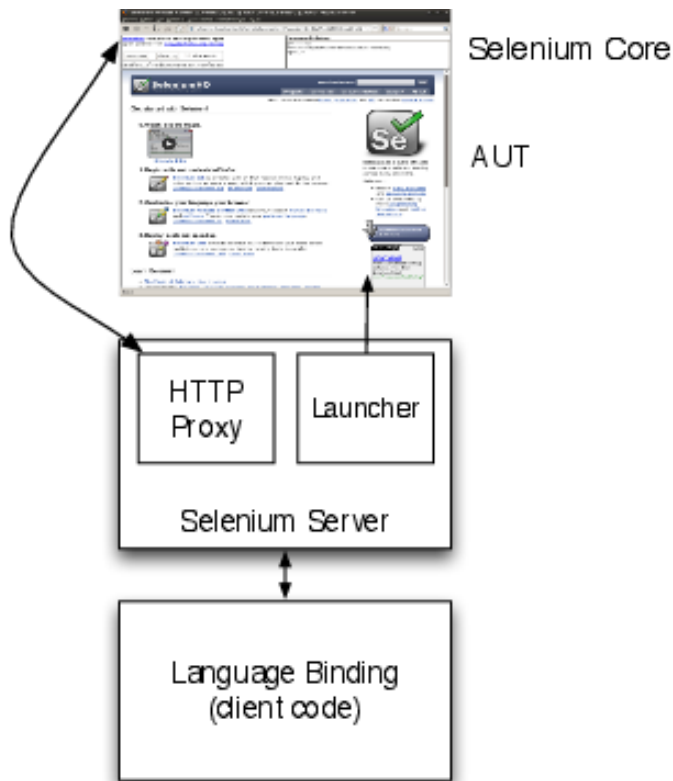


Figure 16.8: Outline of Selenium RC's Architecture

16.8 Selenium RC

It's not always possible to bind tightly to a particular browser. In those cases, WebDriver falls back to the original mechanism used by Selenium. This means using Selenium Core, a pure Javascript framework, which introduces a number of drawbacks as it executes firmly in the context of the Javascript sandbox. From a user of WebDriver's APIs this means that the list of supported browsers falls into tiers, with some being tightly integrated with and offering exceptional control, and others being driven via Javascript and offering the same level of control as the original Selenium RC.

Conceptually, the design used is pretty simple, as you can see in Figure 16.8.

As you can see, there are three moving pieces here: the client code, the intermediate server and the Javascript code of Selenium Core running in the browser. The client side is just an HTTP client that serializes commands to the server-side piece. Unlike the remote WebDriver, there is just a single end-point, and the HTTP verb used is largely irrelevant. This is partly because the Selenium RC protocol is derived from the table-based API offered by Selenium Core, and this means that the entire API can be described using three URL query parameters.

When the client starts a new session, the Selenium server looks up the requested "browser string" to identify a matching browser launcher. The launcher is responsible for configuring and starting an instance of the requested browser. In the case of Firefox, this is as simple as expanding a pre-built

profile with a handful of extensions pre-installed (one for handling a “quit” command, and another for modeling “document.readyState” which wasn’t present on older Firefox releases that we still support). The key piece of configuration that’s done is that the server configures itself as a proxy for the browser, meaning that at least some requests (those for “/selenium-server”) are routed through it. Selenium RC can operate in one of three modes: controlling a frame in a single window (“singlewindow” mode), in a separate window controlling the AUT in a second window (“multiwindow” mode) or by injecting itself into the page via a proxy (“proxyinjection” mode). Depending on the mode of operation, all requests may be proxied.

Once the browser is configured, it is started, with an initial URL pointing to a page hosted on the Selenium server—RemoteRunner.html. This page is responsible for bootstrapping the process by loading all the required Javascript files for Selenium Core. Once complete, the “runSeleniumTest” function is called. This uses reflection of the Selenium object to initialize the list of available commands that are available before kicking off the main command processing loop.

The Javascript executing in the browser opens an XMLHttpRequest to a URL on the waiting server (/selenium-server/driver), relying on the fact that the server is proxying all requests to ensure that the request actually goes somewhere valid. Rather than making a request, the first thing that this does is send the response from the previously executed command, or “OK” in the case where the browser is just starting up. The server then keeps the request open until a new command is received from the user’s test via the client, which is then sent as the response to the waiting Javascript. This mechanism was originally dubbed “Response/Request”, but would now be more likely to be called “Comet with AJAX long polling”.

Why does RC work this way? The server needs to be configured as a proxy so that it can intercept any requests that are made to it without causing the calling Javascript to fall foul of the “Single Host Origin” policy, which states that only resources from the same server that the script was served from can be requested via Javascript. This is in place as a security measure, but from the point of view of a browser automation framework developer, it’s pretty frustrating and requires a hack such as this.

The reason for making an XMLHttpRequest call to the server is two-fold. Firstly, and most importantly, until WebSockets, a part of HTML5, become available in the majority of browsers there is no way to start up a server process reliably within a browser. That means that the server had to live elsewhere. Secondly, an XMLHttpRequest calls the response callback asynchronously, which means that while we’re waiting for the next command the normal execution of the browser is unaffected. The other two ways to wait for the next command would have been to poll the server on a regular basis to see if there was another command to execute, which would have introduced latency to the users tests, or to put the Javascript into a busy loop which would have pushed CPU usage through the roof and would have prevented other Javascript from executing in the browser (since there is only ever one Javascript thread executing in the context of a single window).

Inside Selenium Core there are two major moving pieces. These are the main selenium object, which acts as the host for all available commands and mirrors the API offered to users. The second piece is the browserbot. This is used by the Selenium object to abstract away the differences present in each browser and to present an idealized view of commonly used browser functionality. This means that the functions in selenium are clearer and easier to maintain, whilst the browserbot is tightly focused.

Increasingly, Core is being converted to make use of the Automation Atoms. Both selenium and browserbot will probably need to remain as there is an extensive amount of code that relies on using the APIs it exposes, but it is expected that they will ultimately be shell classes, delegating to the atoms as quickly as possible.

16.9 Looking Back

Building a browser automation framework is a lot like painting a room; at first glance, it looks like something that should be pretty easy to do. All it takes is a few coats of paint, and the job's done. The problem is, the closer you get, the more tasks and details emerge, and the longer the task becomes. With a room, it's things like working around light fittings, radiators and the skirting boards that start to consume time. For a browser automation framework, it's the quirks and differing capabilities of browsers that make the situation more complex. The extreme case of this was expressed by Daniel Wagner-Hall as he sat next to me working on the Chrome driver; he banged his hands on the desk and in frustration muttered, "It's all edge cases!" It would be nice to be able to go back and tell myself that, and that the project is going to take a lot longer than I expected.

I also can't help but wonder where the project would be if we'd identified and acted upon the need for a layer like the automation atoms sooner than we did. It would certainly have made some of the challenges the project faced, internal and external, technically and socially, easier to deal with. Core and RC were implemented in a focused set of languages—essentially just Javascript and Java. Jason Huggins used to refer to this as providing Selenium with a level of “hackability”, which made it easy for people to get involved with the project. It's only with the atoms that this level of hackability has become widely available in WebDriver. Balanced against this, the reason why the atoms can be so widely applied is because of the Closure compiler, which we adopted almost as soon as it was released as Open Source.

It's also interesting to reflect on the things that we got right. The decision to write the framework from the viewpoint of the user is something that I still feel is correct. Initially, this paid off as early adopters highlighted areas for improvement, allowing the utility of the tool to increase rapidly. Later, as WebDriver gets asked to do more and harder things and the number of developers using it increases, it means that new APIs are added with care and attention, keeping the focus of the project tight. Given the scope of what we're trying to do, this focus is vital.

Binding tightly to the browser is something that is both right and wrong. It's right, as it has allowed us to emulate the user with extreme fidelity, and to control the browser extremely well. It's wrong because this approach is extremely technically demanding, particularly when finding the necessary hook point into the browser. The constant evolution of the IE driver is a demonstration of this in action, and, although it's not covered here, the same is true of the Chrome driver, which has a long and storied history. At some point, we'll need to find a way to deal with this complexity.

16.10 Looking to the Future

There will always be browsers that WebDriver can't integrate tightly to, so there will always be a need for Selenium Core. Migrating this from its current traditional design to a more modular design based on the same Closure Library that the atoms are using is underway. We also expect to embed the atoms more deeply within the existing WebDriver implementations.

One of the initial goals of WebDriver was to act as a building block for other APIs and tools. Of course, Selenium doesn't live in a vacuum: there are plenty of other Open Source browser automation tools. One of these is Watir (Web Application Testing In Ruby), and work has begun, as a joint effort by the Selenium and Watir developers, to place the Watir API over the WebDriver core. We're keen to work with other projects too, as successfully driving all the browsers out there is hard work. It would be nice to have a solid kernel that others could build on. Our hope is that the kernel is WebDriver.

A glimpse of this future is offered by Opera Software, who have independently implemented the WebDriver API, using the WebDriver test suites to verify the behavior of their code, and who will be releasing their own OperaDriver. Members of the Selenium team are also working with members of the Chromium team to add better hooks and support for WebDriver to that browser, and by extension to Chrome too. We have a friendly relationship with Mozilla, who have contributed code for the FirefoxDriver, and with the developers of the popular HtmlUnit Java browser emulator.

One view of the future sees this trend continue, with automation hooks being exposed in a uniform way across many different browsers. The advantages for people keen to write tests for web applications are clear, and the advantages for browser manufacturers are also obvious. For example, given the relative expense of manual testing, many large projects rely heavily on automated testing. If it's not possible, or even if it's "only" extremely taxing, to test with a particular browser, then tests just aren't run for it, with knock-on effects for how well complex applications work with that browser. Whether those automation hooks are going to be based on WebDriver is an open question, but we can hope!

The next few years are going to be very interesting. As we're an open source project, you'd be welcome to join us for the journey at <http://selenium.googlecode.com/>.

Sendmail

Eric Allman

Most people think of electronic mail as the program that they interact with—their mail client, technically known as a Mail User Agent (MUA). But another important part of electronic mail is the software that actually transfers the mail from the sender to the recipient—the Mail Transfer Agent (MTA). The first MTA on the Internet, and still the most prevalent, was sendmail.

Sendmail was first created before the Internet officially existed. It has been extraordinarily successful, having grown from 1981, when it wasn't at all obvious that the Internet was going to be more than an academic experiment with only a few hundred hosts, to today, with over 800 million Internet hosts as of January 2011¹. Sendmail remains among the most used implementations of SMTP on the Internet.

17.1 Once Upon a Time...

The first versions of the program that would become known as sendmail were written in 1980. It started as a quick hack to forward messages between different networks. The Internet was being developed but was not functional at that time. In fact, many different networks had been proposed with no obvious consensus emerging. The Arpanet was in use in the United States and the Internet was being designed as an upgrade, but Europe had thrown its weight behind the OSI (Open Systems Interconnect) effort, and for a while it appeared that OSI might triumph. Both of these used leased lines from the phone companies; in the US that speed was 56 Kbps.

Probably the most successful network of the time, in terms of numbers of computers and people connected, was the UUCP network, which was unusual in that it had absolutely no central authority. It was, in some sense, the original peer-to-peer network, which ran over dialup phone lines: 9600 bps was about the fastest available for some time. The fastest network (at 3 Mbps) was based on the Ethernet from Xerox, which ran a protocol called XNS (Xerox Network Systems)—but it didn't work outside of a local installation.

The environment of the time was rather different than what exists today. Computers were highly heterogeneous, to the extent that there wasn't even complete agreement to use 8-bit bytes. For example, other machines included the PDP-10 (36 bit words, 9 bit bytes), the PDP-11 (16 bit words, 8 bit bytes), the CDC 6000 series (60 bit words, 6 bit characters), the IBM 360 (32 bit words, 8 bit

¹<http://ftp.isc.org/www/survey/reports/2011/01/>

bytes), the XDS 940, the ICL 470, and the Sigma 7. One of the up-and-coming platforms was Unix, which at that time came from Bell Laboratories. Most Unix-based machines had 16-bit addresses spaces: at that time the PDP-11 was the major Unix machine, with the Data General 8/32 and the VAX-11/780 just appearing. Threads didn't exist—in fact, the concept of dynamic processes was still fairly new (Unix had them, but “serious” systems such as IBM's OS/360 did not). File locking was not supported in the Unix kernel (but tricks were possible using filesystem links).

To the extent they existed at all, networks were generally low speed (many based on 9600-baud TTY lines; the truly rich might have had Ethernet available, but for local use only). The venerable socket interface wasn't going to be invented for many years. Public key encryption hadn't been invented either, so most network security as we know it today wasn't feasible.

Network email already existed on Unix, but it was created using hacks. The primary user agent at the time was the `/bin/mail` command (today sometimes referred to as `binmail` or `v7mail`), but some sites had other user agents such as Mail from Berkeley, which actually understood how to treat messages as individual items rather than being a glorified `cat` program. Every user agent read (and usually wrote!) `/usr/spool/mail` directly; there was no abstraction for how the messages were actually stored.

The logic to route a message to the network versus local e-mail was nothing more than seeing if the address contained an exclamation point (UUCP) or a colon (BerkNET). People with Arpanet access had to use a completely separate mail program, which would not interoperate with other networks, and which even stored local mail in a different place and in a different format.

To make things even more interesting, there was virtually no standardization on the format of the messages themselves. There was general agreement that there would be a block of header fields at the top of the message, that each header field would be on a new line, and that header field names and values would be separated by a colon. Beyond that, there was very little standardization in either the selection of header field names or the syntaxes of individual fields. For example, some systems used `Subj:` instead of `Subject:`, `Date:` fields were different syntaxes, and some systems didn't understand full names in a `From:` field. On top of all of this, what was documented was often ambiguous or not quite what was actually in use. In particular, RFC 733 (which purported to describe the format of Arpanet messages) was different from what was actually used in subtle but sometimes important ways, and the method of actually transmitting messages was not officially documented at all (although several RFCs made reference to the mechanism, none defined it). The result was that there was somewhat of a priesthood around messaging systems.

In 1979, the INGRES Relational Database Management Project (a.k.a. my day job) got a DARPA grant, and with it a 9600bps Arpanet connection to our PDP-11. At the time it was the only Arpanet connection available in the Computer Science Division, so everyone wanted access to our machine so they could get to the Arpanet. However, that machine was already maxed out, and so we could only make two login ports available for everyone in the department to share. This caused substantial contention and frequent conflicts. However, I noticed that what people wanted most of all was not remote login or file transfer, but e-mail.

Into this, `sendmail` (initially called `delivermail`) emerged as an attempt to unify the chaos into one place. Every MUA (mail user agent, or mail client) would just call `delivermail` to deliver email rather than figuring out how to do it on an ad hoc (and often incompatible) basis. `Delivermail/sendmail` made no attempt to dictate how local mail should be stored or delivered; it did absolutely nothing except shuffle mail between other programs. (This changed when SMTP was added, as we'll see shortly.) In some sense it was just glue to hold the various mail systems together rather than being a mail system in its own right.

During the development of sendmail the Arpanet was transformed into the Internet. The changes were extensive, from the low level packets on the wire up through application protocols, and did not happen instantly. Sendmail was literally developed concurrently with the standards, and in some cases influenced them. It's also notable that sendmail has survived and even thrived as "the network" (as we think of it today) scaled from a few hundred hosts to hundreds of millions of hosts.

Another Network

It's worth mentioning that another completely separate mail standard was proposed at the time called X.400, which was a part of ISO/OSI (International Standards Organization/Open Systems Interconnect). X.400 was a binary protocol, with the message encoded using ASN.1 (Abstract Syntax Notation 1), which is still in use in some Internet protocols today such as LDAP. LDAP was in turn a simplification of X.500, which was the directory service used by X.400. Sendmail made no attempt whatsoever to be directly compatible with X.400, although there were some gateway services extant at the time. Although X.400 was initially adopted by many of the commercial vendors at the time, Internet mail and SMTP ended up winning in the marketplace.

17.2 Design Principles

While developing sendmail, I adhered to several design principles. All of these in some sense came down to one thing: do as little as possible. This is in sharp contrast to some of the other efforts of the time that had much broader goals and required much larger implementations.

Accept that One Programmer Is Finite

I wrote sendmail as a part-time, unpaid project. It was intended to be a quick way of making Arpanet mail more accessible to people at U.C. Berkeley. The key was to forward mail between existing networks, all of which were implemented as standalone programs that were unaware that more than one network even existed. Modifying more than a tiny amount of the existing software was infeasible with only one part-time programmer. The design had to minimize the amount of existing code that needed to be modified as well as the amount of new code that needed to be written. This constraint drove most of the rest of the design principles. As it turned out, in most cases they would have been the right thing to do even if there had been a larger team available.

Don't Redesign User Agents

A Mail User Agent (MUA) is what most end users think of as the "mail system"—it's the program that they use to read, write, and answer mail. It is quite distinct from the Mail Transfer Agent (MTA), which routes email from the sender to the receiver. At the time sendmail was written, many implementations at least partly combined these two functions, so they were often developed in tandem. Trying to work on both at the same time would have been too much, so Sendmail completely punted on the user interface problem: the only changes to MUAs were to have them invoke sendmail instead of doing their own routing. In particular, there were already several user agents, and people were often quite emotional about how they interacted with mail. Trying to work on both at the same

time would have been too much. This separation of the MUA from the MTA is accepted wisdom now, but was far from standard practice at the time.

Don't Redesign the Local Mail Store

The local mail store (where messages would be saved until the recipient came along to read them) was not formally standardized. Some sites liked to store them in a centralized place, such as `/usr/mail`, `/var/mail`, or `/var/spool/mail`. Other sites liked to store them in the recipient's home directory (e.g., as a file called `.mail`). Most sites started each message with a line beginning "From" followed by a space character (an extraordinarily bad decision, but that was the convention at the time), but sites that were Arpanet-focused usually stored messages separated by a line containing four control-A characters. Some sites attempted to lock the mailbox to prevent collisions, but they used different locking conventions (file locking primitives were not yet available). In short, the only reasonable thing to do was treat local mail storage as a black box.

On nearly all sites, the actual mechanism for doing local mailbox storage was embodied in the `/bin/mail` program. This had a (quite primitive) user interface, routing, and storage built into one program. To incorporate sendmail, the routing portion was pulled out and replaced with a call to sendmail. A `-d` flag was added to force final delivery, i.e., it prevented `/bin/mail` from calling sendmail to do the routing. In later years the code used to deliver a message to a physical mailbox was extracted into another program called `mail.local`. The `/bin/mail` program exists today only to include a lowest common denominator for scripts to send mail.

Make Sendmail Adapt to the World, Not the Other Way Around

Protocols such as UUCP and BerkNET were already implemented as separate programs that had their own, sometimes quirky, command line structure. In some cases they were being actively developed at the same time as sendmail. It was clear that reimplementing them (for example, to convert them to standard calling conventions) was going to be painful. This led directly to the principle that sendmail should adapt to the rest of the world rather than trying to make the rest of the world adapt to sendmail.

Change as Little as Possible

To the fullest extent possible, during the development of sendmail I didn't touch anything I didn't absolutely have to touch. Besides just not having enough time to do it, there was a culture at Berkeley at the time that eschewed most formal code ownership in favor of a policy of "the last person who touched the code is the go-to person for that program" (or more simply, "you touch it, you own it"). Although that sounds chaotic by most modern-day standards, it worked quite well in a world where no one at Berkeley was assigned full time to work on Unix; individuals worked on parts of the system that they were interested in and committed to and didn't touch the rest of the code base except in dire circumstances.

Think About Reliability Early

The mail system prior to sendmail (including most of the transport systems) wasn't terribly concerned about reliability. For example, versions of Unix prior to 4.2BSD did not have native file locking, although it could be simulated by creating a temporary file and then linking it to a lock file (if the lock file already existed the link call would fail). However, sometimes different programs writing

the same data file wouldn't agree on how the locking should be done (for example, they might use a different lock file name or even make no attempt to do locking at all), and so it wasn't that uncommon to lose mail. Sendmail took the approach that losing mail wasn't an option (possibly a result of my background as a database guy, where losing data is a mortal sin).

What Was Left Out

There were many things that were not done in the early versions. I did not try to re-architect the mail system or build a completely general solution: functionality could be added as the need arose. Very early versions were not even intended to be completely configurable without access to the source code and a compiler (although this changed fairly early on). In general, the *modus operandi* for sendmail was to get something working quickly and then enhance working code as needed and as the problem was better understood.

17.3 Development Phases

Like most long-lived software, sendmail was developed in phases, each with its own basic theme and feeling.

Wave 1: delivermail

The first instantiation of sendmail was known as delivermail. It was extremely simple, if not simplistic. Its sole job was to forward mail from one program to another; in particular, it had no SMTP support, and so never made any direct network connections. No queuing was necessary because each network already had its own queue, so the program was really just a crossbar switch. Since delivermail had no direct network protocol support, there was no reason for it to run as a daemon—it would be invoked to route each message as it was submitted, pass it to the appropriate program that would implement the next hop, and terminate. Also, there was no attempt to rewrite headers to match the network to which a message was being delivered. This commonly resulted in messages being forwarded that could not be replied to. The situation was so bad that an entire book was written about addressing mail (called, fittingly, *!%:: A Directory of Electronic Mail Addressing & Networks* [AF94]).

All configuration in delivermail was compiled in and was based only on special characters in each address. The characters had precedence. For example, a host configuration might search for an “@” sign and, if one was found, send the entire address to a designated Arpanet relay host. Otherwise, it might search for a colon, and send the message to BerkNET with the designated host and user if it found one, then could check for an exclamation point (“!”) signalling that the message should be forwarded to a designated UUCP relay. Otherwise it would attempt local delivery. This configuration might result in the following:

Input	Sent To net, host, user
foo@bar	{ Arpanet, bar, foo }
foo:bar	{ Berknet, foo, bar }
foo!bar!baz	{ Uucp, foo, bar!baz }
foo!bar@baz	{ Arpanet, baz, foo!bar }

Note that address delimiters differed in their associativity, resulting in ambiguities that could only be resolved using heuristics. For example, the last example might reasonably be parsed as {Uucp, foo, bar@baz} at another site.

The configuration was compiled in for several reasons: first, with a 16 bit address space and limited memory, parsing a runtime configuration was too expensive. Second, the systems of the time had been so highly customized that recompiling was a good idea, just to make sure you had the local versions of the libraries (shared libraries did not exist with Unix 6th Edition).

Delivermail was distributed with 4.0 and 4.1 BSD and was more successful than expected; Berkeley was far from the only site with hybrid network architectures. It became clear that more work was required.

Wave 2: sendmail 3, 4, and 5

Versions 1 and 2 were distributed under the delivermail name. In March 1981 work began on version 3, which would be distributed under the sendmail name. At this point the 16-bit PDP-11 was still in common use but the 32-bit VAX-11 was becoming popular, so many of the original constraints associated with small address spaces were starting to be relaxed.

The initial goals of sendmail were to convert to runtime configuration, allow message modification to provide compatibility across networks for forwarded mail, and have a richer language on which to make routing decisions. The technique used was essentially textual rewriting of addresses (based on tokens rather than character strings), a mechanism used in some expert systems at the time. There was ad hoc code to extract and save any comment strings (in parentheses) as well as to re-insert them after the programmatic rewriting completed. It was also important to be able to add or augment header fields (e.g., adding a Date header field or including the full name of the sender in the From header if it was known).

SMTP development started in November 1981. The Computer Science Research Group (CSRG) at U.C. Berkeley had gotten the DARPA contract to produce a Unix-based platform to support DARPA funded research, with the intent of making sharing between projects easier. The initial work on the TCP/IP stack was done by that time, although the details of the socket interface were still changing. Basic application protocols such as Telnet and FTP were done, but SMTP had yet to be implemented. In fact, the SMTP protocol wasn't even finalized at that point; there had been a huge debate about how mail should be sent using a protocol to be creatively named Mail Transfer Protocol (MTP). As the debate raged, MTP got more and more complex until in frustration SMTP (Simple Mail Transfer Protocol) was drafted more-or-less by fiat (but not officially published until August 1982). Officially, I was working on the INGRES Relational Database Management System, but since I knew more about the mail system than anyone else around Berkeley at the time, I got talked into implementing SMTP.

My initial thought was to create a separate SMTP mailer that would have its own queueing and daemon; that subsystem would attach to sendmail to do the routing. However, several features of SMTP made this problematic. For example, the EXPN and VRFY commands required access to the parsing, aliasing, and local address verification modules. Also, at the time I thought it was important that the RCPT command return immediately if the address was unknown, rather than accepting the message and then having to send a delivery failure message later. This turns out to have been a prescient decision. Ironically, later MTAs often got this wrong, exacerbating the spam backscatter problem. These issues drove the decision to include SMTP as part of sendmail itself.

Sendmail 3 was distributed with 4.1a and 4.1c BSD (beta versions), sendmail 4 was distributed with 4.2 BSD, and sendmail 5 was distributed with 4.3 BSD.

Wave 3: The Chaos Years

After I left Berkeley and went to a startup company, my time available to work on sendmail rapidly decreased. But the Internet was starting to seriously explode and sendmail was being used in a variety of new (and larger) environments. Most of the Unix system vendors (Sun, DEC, and IBM in particular) created their own versions of sendmail, all of which were mutually incompatible. There were also attempts to build open source versions, notably IDA sendmail and KJS.

IDA sendmail came from Linköping University. IDA included extensions to make it easier to install and manage in larger environments and a completely new configuration system. One of the major new features was the inclusion of dbm(3) database maps to support highly dynamic sites. These were available using a new syntax in the configuration file and were used for many functions including mapping of addresses to and from external syntax (for example, sending out mail as john_doe@example.com instead of johnd@example.com) and routing.

King James Sendmail (KJS, produced by Paul Vixie) was an attempt to unify all the various versions of sendmail that had sprung up. Unfortunately, it never really got enough traction to have the desired effect. This era was also driven by a plethora of new technologies that were reflected in the mail system. For example, Sun's creation of diskless clusters added the YP (later NIS) directory services and NFS, the Network File System. In particular, YP had to be visible to sendmail, since aliases were stored in YP rather than in local files.

Wave 4: sendmail 8

After several years, I returned to Berkeley as a staff member. My job was to manage a group installing and supporting shared infrastructure for research around the Computer Science department. For that to succeed, the largely ad hoc environments of individual research groups had to be unified in some rational way. Much like the early days of the Internet, different research groups were running on radically different platforms, some of which were quite old. In general, every research group ran its own systems, and although some of them were well managed, most of them suffered from “deferred maintenance.”

In most cases email was similarly fractured. Each person's email address was “person@host.berkeley.edu”, where host was the name of the workstation in their office or the shared server they used (the campus didn't even have internal subdomains) with the exception of a few special people who had @berkeley.edu addresses. The goal was to switch to internal subdomains (so all individual hosts would be in the cs.berkeley.edu subdomain) and have a unified mail system (so each person would have an @cs.berkeley.edu address). This goal was most easily realized by creating a new version of sendmail that could be used throughout the department.

I began by studying many of the variants of sendmail that had become popular. My intent was not to start from a different code base but rather to understand the functionality that others had found useful. Many of those ideas found their way into sendmail 8, often with modifications to merge related ideas or make them more generic. For example, several versions of sendmail had the ability to access external databases such as dbm(3) or NIS; sendmail 8 merged these into one “map” mechanism that could handle multiple types of databases (and even arbitrary non-database transformations). Similarly, the “generics” database (internal to external name mapping) from IDA sendmail was incorporated.

Sendmail 8 also included a new configuration package using the m4(1) macro processor. This was intended to be more declarative than the sendmail 5 configuration package, which had been largely procedural. That is, the sendmail 5 configuration package required the administrator to

essentially lay out the entire configuration file by hand, really only using the “include” facility from m4 as shorthand. The sendmail 8 configuration file allowed the administrator to just declare what features, mailers, and so on were required, and m4 laid out the final configuration file.

Much of Section 17.7 discusses the enhancements in sendmail 8.

Wave 5: The Commercial Years

As the Internet grew and the number of sendmail sites expanded, support for the ever larger user base became more problematic. For a while I was able to continue support by setting up a group of volunteers (informally called the “Sendmail Consortium”, a.k.a. sendmail.org) who provided free support via e-mail and newsgroup. But by the late 1990s, the installed base had grown to such an extent that it was nearly impossible to support it on a volunteer basis. Together with a more business-savvy friend I founded Sendmail, Inc.², with the expectation of getting new resources to bear on the code.

Although the commercial product was originally based largely on configuration and management tools, many new features were added to the open-source MTA to support the needs of the commercial world. Notably, the company added support for TLS (connection encryption), SMTP Authentication, site security enhancements such as Denial of Service protection, and most importantly mail filtering plugins (the Milter interface discussed below).

At of this writing the commercial product has expanded to include a large suite of e-mail based applications, nearly all of which are constructed on the extensions added to sendmail during the first few years of the company.

Whatever Happened to sendmail 6 and 7?

Sendmail 6 was essentially the beta for sendmail 8. It was never officially released, but was distributed fairly widely. Sendmail 7 never existed at all; sendmail jumped directly to version 8 because all the other source files for the BSD distribution were bumped to version 8 when 4.4 BSD was released in June 1993.

17.4 Design Decisions

Some design decisions were right. Some started out right and became wrong as the world changed. Some were dubious and haven’t become any less so.

The Syntax of the Configuration File

The syntax of the configuration file was driven by a couple of issues. First, the entire application had to fit into a 16-bit address space, so the parser had to be small. Second, early configurations were quite short (under one page), so while the syntax was obscure, the file was still comprehensible. However, as time passed, more operational decisions moved out of the C code into the configuration file, and the file started to grow. The configuration file acquired a reputation for being arcane. One particular frustration for many people was the choice of the tab character as an active syntax item. This was a mistake that was copied from other systems of the time, notably make. That particular

²<http://www.sendmail.com>

problem became more acute as window systems (and hence cut-and-paste, which usually did not preserve the tabs) became available.

In retrospect, as the file got larger and 32-bit machines took over, it would have made sense to reconsider the syntax. There was a time when I thought about doing this but decided against it because I didn't want to break the "large" installed base (which at that point was probably a few hundred machines). In retrospect this was a mistake; I had simply not appreciated how large the install base would grow and how many hours it would save me had I changed the syntax early. Also, when the standards stabilized a fair amount of the generality could have been pushed back into the C code base, thus simplifying the configurations.

Of particular interest was how more functionality got moved into the configuration file. I was developing sendmail at the same time as the SMTP standard was evolving. By moving operational decisions into the configuration file I was able to respond rapidly to design changes—usually in under 24 hours. I believe that this improved the SMTP standard, since it was possible to get operational experience with a proposed design change quite quickly, but only at the cost of making the configuration file difficult to understand.

Rewriting Rules

One of the difficult decisions when writing sendmail was how to do the necessary rewriting to allow forwarding between networks without violating the standards of the receiving network. The transformations required changing metacharacters (for example, BerkNET used colon as a separator, which was not legal in SMTP addresses), rearranging address components, adding or deleting components, etc. For example, the following rewrites would be needed under certain circumstances:

From	To
a:foo	a.foo@berkeley.edu
a!b!c	b!c@a.uucp
<@a.net,@b.org:user@c.com>	<@b.org:user@c.com>

Regular expressions were not a good choice because they didn't have good support for word boundaries, quoting, etc. It quickly became obvious that it would be nearly impossible to write regular expressions that were accurate, much less intelligible. In particular, regular expressions reserve a number of metacharacters, including ".", "*", "+", "[", and "]", all of which can appear in e-mail addresses. These could have been escaped in configuration files, but I deemed that to be complicated, confusing, and a bit ugly. (This was tried by UPAS from Bell Laboratories, the mailer for Unix Eighth Edition, but it never caught on³.) Instead, a scanning phase was necessary to produce tokens that could then be manipulated much like characters in regular expressions. A single parameter describing "operator characters", which were themselves both tokens and token separators, was sufficient. Blank spaces separated tokens but were not tokens themselves. The rewriting rules were just pattern match/replace pairs organized into what were essentially subroutines.

Instead of a large number of metacharacters that had to be escaped to lose their "magic" properties (as used in regular expressions), I used a single "escape" character that combined with ordinary characters to represent wildcard patterns (to match an arbitrary word, for example). The traditional Unix approach would be to use backslash, but backslash was already used as a quote character in some address syntaxes. As it turned out, "\$" was one of the few characters that had not already been used as a punctuation character in some email syntax.

³http://doc.cat-v.org/bell_labs/upas_mail_system/upas.pdf

One of the original bad decisions was, ironically, just a matter of how white space was used. A space character was a separator, just as in most scanned input, and so could have been used freely between tokens in patterns. However, the original configuration files distributed did not include spaces, resulting in patterns that were far harder to understand than necessary. Consider the difference between the following two (semantically identical) patterns:

```
$+ + $* @ $+ . $={mydomain}  
$++$*@$+. $={mydomain}
```

Using Rewriting for Parsing

Some have suggested that sendmail should have used conventional grammar-based parsing techniques to parse addresses rather than rewriting rules and leave the rewriting rules for address modification. On the surface this would seem to make sense, given that the standards define addresses using a grammar. The main reason for reusing rewriting rules is that in some cases it was necessary to parse header field addresses (e.g., in order to extract the sender envelope from a header when receiving mail from a network that didn't have a formal envelope). Such addresses aren't easy to parse using (say) an LALR(1) parser such as YACC and a traditional scanner because of the amount of lookahead required. For example, parsing the address: `allman@foo.bar.baz.com <eric@example.com>` requires lookahead by either the scanner or the parser; you can't know that the initial "allman@..." is not an address until you see the "<". Since LALR(1) parsers only have one token of lookahead this would have had to be done in the scanner, which would have complicated it substantially. Since the rewriting rules already had arbitrary backtracking (i.e., they could look ahead arbitrarily far), they were sufficient.

A secondary reason was that it was relatively easy to make the patterns recognize and fix broken input. Finally, rewriting was more than powerful enough to do the job, and reusing any code was wise.

One unusual point about the rewriting rules: when doing the pattern matching, it is useful for both the input and the pattern to be tokenized. Hence, the same scanner is used for both the input addresses and the patterns themselves. This requires that the scanner be called with different character type tables for differing input.

Embedding SMTP and Queueing in sendmail

An "obvious" way to implement outgoing (client) SMTP would have been to build it as an external mailer, similarly to UUCP. But this would raise a number of other questions. For example, would queueing be done in sendmail or in the SMTP client module? If it was done in sendmail then either separate copies of messages would have to be sent to each recipient (i.e., no "piggybacking", wherein a single connection can be opened and then multiple RCPT commands can be sent) or a much richer communication back-path would be necessary to convey the necessary per-recipient status than was possible using simple Unix exit codes. If queueing was done in the client module then there was a potential for large amounts of replication; in particular, at the time other networks such as XNS were still possible contenders. Additionally, including the queue into sendmail itself provided a more elegant way of dealing with certain kinds of failures, notably transient problems such as resource exhaustion.

Incoming (server) SMTP involved a different set of decisions. At the time, I felt it was important to implement the VRFY and EXPN SMTP commands faithfully, which required access to the alias mechanism. This would once again require a much richer protocol exchange between the server

SMTP module and sendmail than was possible using command lines and exit codes—in fact, a protocol akin to SMTP itself.

I would be much more inclined today to leave queueing in the core sendmail but move both sides of the SMTP implementation into other processes. One reason is to gain security: once the server side has an open instance of port 25 it no longer needs access to root permissions. Modern extensions such as TLS and DKIM signing complicate the client side (since the private keys should not be accessible to unprivileged users), but strictly speaking root access is still not necessary. Although the security issue is still an issue here, if the client SMTP is running as a non-root user who can read the private keys, that user by definition has special privileges, and hence should not be communicating directly with other sites. All of these issues can be finessed with a bit of work.

The Implementation of the Queue

Sendmail followed the conventions of the time for storing queue files. In fact, the format used is extremely similar to the lpr subsystem of the time. Each job had two files, one with the control information and one with the data. The control file was a flat text file with the first character of each line representing the meaning of that line.

When sendmail wanted to process the queue it had to read all of the control files, storing the relevant information in memory, and then sort the list. That worked fine with a relatively small number of messages in the queue, but started to break down at around 10,000 queued messages. Specifically, when the directory got large enough to require indirect blocks in the filesystem, there was a serious performance knee that could reduce performance by as much as an order of magnitude. It was possible to ameliorate this problem by having sendmail understand multiple queue directories, but that was at best a hack.

An alternative implementation might be to store all the control files in one database file. This wasn't done because when sendmail coding began there was no generally available database package, and when dbm(3) became available it had several flaws, including the inability to reclaim space, a requirement that all keys that hashed together fit on one (512 byte) page, and a lack of locking. Robust database packages didn't appear for many years.

Another alternative implementation would have been to have a separate daemon that would keep the state of the queue in memory, probably writing a log to allow recovery. Given the relatively low email traffic volumes of the time, the lack of memory on most machines, the relatively high cost of background processes, and the complexity of implementing such a process, this didn't seem like a good tradeoff at the time.

Another design decision was to store the message header in the queue control file rather than the data file. The rationale was that most headers needed considerable rewriting that varied from destination to destination (and since messages could have more than one destination, they would have to be customized multiple times), and the cost of parsing the headers seemed high, so storing them in a pre-parsed format seemed like a savings. In retrospect this was not a good decision, as was storing the message body in Unix-standard format (with newline endings) rather than in the format in which it was received (which could use newlines, carriage-return/line-feed, bare carriage-return, or line-feed/carriage-return). As the e-mail world evolved and standards were adopted, the need for rewriting diminished, and even seemingly innocuous rewriting has the risk of error.

Accepting and Fixing Bogus Input

Since sendmail was created in a world of multiple protocols and disturbingly few written standards, I decided to clean up malformed messages wherever possible. This matches the “Robustness Principle” (a.k.a. Postel’s Law) articulated in RFC 793⁴. Some of these changes were obvious and even required: when sending a UUCP message to the Arpanet, the UUCP addresses needed to be converted to Arpanet addresses, if only to allow “reply” commands to work correctly, line terminations needed to be converted between the conventions used by various platforms, and so on. Some were less obvious: if a message was received that did not include a From: header field required in the Internet specifications, should you add a From: header field, pass the message on without the From: header field, or reject the message? At the time, my prime consideration was interoperability, so sendmail patched the message, e.g., by adding the From: header field. However, this is claimed to have allowed other broken mail systems to be perpetuated long past the time when they should have been fixed or killed off.

I believe my decision was correct for the time, but is problematic today. A high degree of interoperability was important to let mail flow unimpeded. Had I rejected malformed messages, most messages at the time would have been rejected. Had I passed them through unfixed, recipients would have received messages that they couldn’t reply to and in some cases couldn’t even determine who sent the message—that or the message would have been rejected by another mailer.

Today the standards are written, and for the most part those standards are accurate and complete. It is no longer the case that most messages would be rejected, and yet there is still mail software out there that send out mangled messages. This unnecessarily creates numerous problems for other software on the Internet.

Configuration and the Use of M4

For a period I was both making regular changes to the sendmail configuration files and personally supporting many machines. Since a large amount of the configuration file was the same between different machines, the use of a tool to build the configuration files was desirable. The m4 macro processor was included with Unix. It was designed as a front end for programming languages (notably ratfor). Most importantly, it had “include” capabilities, like “#include” in the C language. The original configuration files used little more than this capability and some minor macro expansions.

IDA sendmail also used m4, but in a dramatically different way. In retrospect I should have probably studied these prototypes in more detail. They contained many clever ideas, in particular the way they handled quoting.

Starting with sendmail 6, the m4 configuration files were completely rewritten to be in a more declarative style and much smaller. This used considerably more of the power of the m4 processor, which was problematic when the introduction of GNU m4 changed some of the semantics in subtle ways.

The original plan was that the m4 configurations would follow the 80/20 rule: they would be simple (hence 20% of the work), and would cover 80% of the cases. This broke down fairly quickly, for two reasons. The minor reason was that it turned out to be relatively easy to handle the vast majority of the cases, at least in the beginning. It became much harder as sendmail and the world evolved, notably with the inclusion of features such as TLS encryption and SMTP Authentication, but those didn’t come until quite a bit later.

⁴“Be conservative in what you do, be liberal in what you accept from others”

The important reason was that it was becoming clear that the raw configuration file was just too difficult for most people to manage. In essence, the `.cf` (raw) format had become assembly code—editable in principle, but in reality quite opaque. The “source code” was an m4 script stored in the `.mc` file.

Another important distinction is that the raw format configuration file was really a programming language. It had procedural code (rulesets), subroutine calls, parameter expansion, and loops (but no `gotos`). The syntax was obscure, but in many ways resembled the `sed` and `awk` commands, at least conceptually. The m4 format was declarative: although it was possible to drop into the low-level raw language, in practice these details were hidden from the user.

It isn’t clear that this decision was correct or incorrect. I felt at the time (and still feel) that with complex systems it can be useful to implement what amounts to a Domain Specific Language (DSL) for building certain portions of that system. However, exposing that DSL to end users as a configuration methodology essentially converts all attempts to configure a system into a programming problem. Great power results from this, but at a non-trivial cost.

17.5 Other Considerations

Several other architectural and development points deserve to be mentioned.

A Word About Optimizing Internet Scale Systems

In most network-based systems there is a tension between the client and the server. A good strategy for the client may be the wrong thing for the server and vice versa. For example, when possible the server would like to minimize its processing costs by pushing as much as possible back to the client, and of course the client feels the same way but in the opposite direction. For example, a server might want to keep a connection open while doing spam processing since that lowers the cost of rejecting a message (which these days is the common case), but the client wants to move on as quickly as possible. Looking at the entire system, that is, the Internet as a whole, the optimum solution may be to balance these two needs.

There have been cases of MTAs that have used strategies that explicitly favor either the client or the server. They can do this only because they have a relatively small installed base. When your system is used on a significant portion of the Internet you have to design it in order to balance the load between both sides in an attempt to optimize the Internet as a whole. This is complicated by the fact that there will always be MTAs completely skewed in one direction or the other—for example, mass mailing systems only care about optimizing the outgoing side.

When designing a system that incorporates both sides of the connection, it is important to avoid playing favorites. Note that this is in stark contrast to the usual asymmetry of clients and services—for example, web servers and web clients are generally not developed by the same groups.

Milter

One of the most important additions to `sendmail` was the milter (*mail filter*) interface. Milter allows for the use of offboard plugins (i.e., they run in a separate process) for mail processing. These were originally designed for anti-spam processing. The milter protocol runs synchronously with the server SMTP protocol. As each new SMTP command is received from the client, `sendmail` calls the milters with the information from that command. The milter has the opportunity to accept the command or

send a rejection, which rejects the phase of the protocol appropriate for the SMTP command. Milters are modeled as callbacks, so as an SMTP command comes in, the appropriate milter subroutine is called. Milters are threaded, with a per-connection context pointer handed in to each routine to allow passing state.

In theory milters could work as loadable modules in the sendmail address space. We declined to do this for three reasons. First, the security issues were too significant: even if sendmail were running as a unique non-root user id, that user would have access to all of the state of other messages. Similarly, it was inevitable that some milter authors would try to access internal sendmail state.

Second, we wanted to create a firewall between sendmail and the milters: if a milter crashed, we wanted it to be clear who was at fault, and for mail to (potentially) continue to flow. Third, it was much easier for a milter author to debug a standalone process than sendmail as a whole.

It quickly became clear that the milter was useful for more than anti-spam processing. In fact, the milter.org⁵ web site lists milters for anti-spam, anti-virus, archiving, content monitoring, logging, traffic shaping, and many other categories, produced by commercial companies and open source projects. The postfix mailer⁶ has added support for milters using the same interface. Milters have proven to be one of sendmail's great successes.

Release Schedules

There is a popular debate between “release early and often” and “release stable systems” schools of thought. Sendmail has used both of these at various times. During times of considerable change I was sometimes doing more than one release a day. My general philosophy was to make a release after each change. This is similar to providing public access to the source management system tree. I personally prefer doing releases over providing public source trees, at least in part because I use source management in what is now considered an unapproved way: for large changes, I will check in non-functioning snapshots while I am writing the code. If the tree is shared I will use branches for these snapshots, but in any case they are available for the world to see and can create considerable confusion. Also, creating a release means putting a number on it, which makes it easier to track the changes when going through a bug report. Of course, this requires that releases be easy to generate, which is not always true.

As sendmail became used in ever more critical production environments this started to become problematic. It wasn't always easy for others to tell the difference between changes that I wanted out there for people to test versus changes that were really intended to be used in the wild. Labeling releases as “alpha” or “beta” alleviates but does not fix the problem. The result was that as sendmail matured it moved toward less frequent but larger releases. This became especially acute when sendmail got folded into a commercial company which had customers who wanted both the latest and greatest but also only stable versions, and wouldn't accept that the two are incompatible.

This tension between open source developer needs and commercial product needs will never go away. There are many advantages to releasing early and often, notably the potentially huge audience of brave (and sometimes foolish) testers who stress the system in ways that you could almost never expect to reproduce in a standard development system. But as a project becomes successful it tends to turn into a product (even if that product is open source and free), and products have different needs than projects.

⁵<http://milter.org>

⁶<http://postfix.org>

17.6 Security

Sendmail has had a tumultuous life, security-wise. Some of this is well deserved, but some not, as our concept of “security” changed beneath us. The Internet started out with a user base of a few thousand people, mostly in academic and research settings. It was, in many ways, a kinder, gentler Internet than we know today. The network was designed to encourage sharing, not to build firewalls (another concept that did not exist in the early days). The net is now a dangerous, hostile place, filled with spammers and crackers. Increasingly it is being described as a war zone, and in war zones there are civilian casualties.

It’s hard to write network servers securely, especially when the protocol is anything beyond the most simple. Nearly all programs have had at least minor problems; even common TCP/IP implementations have been successfully attacked. Higher-level implementation languages have proved no panacea, and have even created vulnerabilities of their own. The necessary watch phrase is “distrust all input,” no matter where it comes from. Distrusting input includes secondary input, for example, from DNS servers and milters. Like most early network software, sendmail was far too trusting in its early versions.

But the biggest problem with sendmail was that early versions ran with root permissions. Root permission is needed in order to open the SMTP listening socket, to read individual users’ forwarding information, and to deliver to individual users’ mailboxes and home directories. However, on most systems today the concept of a mailbox name has been divorced from the concept of a system user, which effectively eliminates the need for root access except to open the SMTP listening socket. Today sendmail has the ability to give up root permissions before it processes a connection, eliminating this concern for environments that can support it. It’s worth noting that on those systems that do not deliver directly to users’ mailboxes, sendmail can also run in a chrooted environment, allowing further permission isolation.

Unfortunately, as sendmail gained a reputation for poor security, it started to be blamed for problems that had nothing to do with sendmail. For example, one system administrator made his /etc directory world writable and then blamed sendmail when someone replaced the /etc/passwd file. It was incidents like this that caused us to tighten security substantially, including explicitly checking the ownerships and modes on files and directories that sendmail accesses. These were so draconian that we were obliged to include the DontBlameSendmail option to (selectively) turn off these checks.

There are other aspects of security that are not related to protecting the address space of the program itself. For example, the rise of spam also caused a rise in address harvesting. The VRFY and EXPN commands in SMTP were designed specifically to validate individual addresses and expand the contents of mailing lists respectively. These have been so badly abused by spammers that most sites now turn them off entirely. This is unfortunate, at least with VRFY, as this command was sometimes used by some anti-spam agents to validate the purported sending address.

Similarly, anti-virus protection was once seen as a desktop problem, but rose in importance to the point where any commercial-grade MTA had to have anti-virus checking available. Other security-related requirements in modern settings include mandatory encryption of sensitive data, data loss protection, and enforcement of regulatory requirements, for example, for HIPPA.

One of the principles that sendmail took to heart early on was reliability—every message should either be delivered or reported back to the sender. But the problem of joe-jobs (attackers forging the return address on a message, viewed by many as a security issue) has caused many sites to turn off the creation of bounce messages. If a failure can be determined while the SMTP connection is still open, the server can report the problem by failing the command, but after the SMTP connection

is closed an incorrectly addressed message will silently disappear. To be fair, most legitimate mail today is single hop, so problems will be reported, but at least in principle the world has decided that security wins over reliability.

17.7 Evolution of Sendmail

Software doesn't survive in a rapidly changing environment without evolving to fit the changing environment. New hardware technologies appear, which push changes in the operating system, which push changes in libraries and frameworks, which push changes in applications. If an application succeeds, it gets used in ever more problematic environments. Change is inevitable; to succeed you have to accept and embrace change. This section describes some of the more important changes that have occurred as sendmail evolved.

Configuration Became More Verbose

The original configuration of sendmail was quite terse. For example, the names of options and macros were all single characters. There were three reasons for this. First, it made parsing very simple (important in a 16-bit environment). Second, there weren't very many options, so it wasn't hard to come up with mnemonic names. Third, the single character convention was already established with command-line flags.

Similarly, rewriting rulesets were originally numbered instead of named. This was perhaps tolerable with a small number of rulesets, but as their number grew it became important that they have more mnemonic names.

As the environment in which sendmail operated became more complex, and as the 16-bit environment faded away, the need for a richer configuration language became evident. Fortunately, it was possible to make these changes in a backward compatible way. These changes dramatically improved the understandability of the configuration file.

More Connections with Other Subsystems: Greater Integration

When sendmail was written the mail system was largely isolated from the rest of the operating system. There were a few services that required integration, e.g., the `/etc/passwd` and `/etc/hosts` files. Service switches had not been invented, directory services were nonexistent, and configuration was small and hand-maintained.

That quickly changed. One of the first additions was DNS. Although the system host lookup abstraction (`gethostbyname`) worked for looking up IP addresses, email had to use other queries such as MX. Later, IDA sendmail included an external database lookup functionality using `dbm(3)` files. Sendmail 8 updated that to a general mapping service that allowed other database types, including external databases and internal transformations that could not be done using rewriting (e.g., dequoting an address).

Today, the email system relies on many external services that are, in general, not designed specifically for the exclusive use of email. This has moved sendmail toward more abstractions in the code. It has also made maintaining the mail system more difficult as more "moving parts" are added.

Adaptation to a Hostile World

Sendmail was developed in a world that seems completely foreign by today's standards. The user population on the early network were mostly researchers who were relatively benign, despite the sometimes vicious academic politics. Sendmail reflected the world in which it was created, putting a lot of emphasis on getting the mail through as reliably as possible, even in the face of user errors.

Today's world is much more hostile. The vast majority of email is malicious. The goal of an MTA has transitioned from getting the mail through to keeping the bad mail out. Filtering is probably the first priority for any MTA today. This required a number of changes in sendmail.

For example, many rulesets have been added to allow checking of parameters on incoming SMTP commands in order to catch problems as early as possible. It is much cheaper to reject a message when reading the envelope than after you have committed to reading the entire message, and even more expensive after you have accepted the message for delivery. In the early days filtering was generally done by accepting the message, passing it to a filter program, and then sending it to another instance of sendmail if the message passed (the so-called "sandwich" configuration). This is just far too expensive in today's world.

Similarly, sendmail has gone from being a quite vanilla consumer of TCP/IP connections to being much more sophisticated, doing things like "peeking" at network input to see if the sender is transmitting commands before the previous command has been acknowledged. This breaks down some of the previous abstractions that were designed to make sendmail adaptable to multiple network types. Today, it would involve considerable work to connect sendmail to an XNS or DECnet network, for example, since the knowledge of TCP/IP has been built into so much of the code.

Many configuration features were added to address the hostile world, such as support for access tables, Realtime Blackhole Lists, address harvesting mitigation, denial-of-service protection, and spam filtering. This has dramatically complicated the task of configuring a mail system, but was absolutely necessary to adapt to today's world.

Incorporation of New Technologies

Many new standards have come along over the years that required significant changes to sendmail. For example, the addition of TLS (encryption) required significant changes through much of the code. SMTP pipelining required peering into the low-level TCP/IP stream to avoid deadlocks. The addition of the submission port (587) required the ability to listen to multiple incoming ports, including having different behaviors depending on the arrival port.

Other pressures were forced by circumstances rather than standards. For example, the addition of the milter interface was a direct response to spam. Although milter was not a published standard, it was a major new technology.

In all cases, these changes enhanced the mail system in some way, be it increased security, better performance, or new functionality. However, they all came with costs, in nearly all cases complicating both the code base and the configuration file.

17.8 What If I Did It Today?

Hindsight is 20/20. There are many things I would do differently today. Some were unforeseeable at the time (e.g., how spam would change our perception of e-mail, what modern toolsets would look like, etc.), and some were eminently predictable. Some were just that in the process of writing

sendmail I learned a lot about e-mail, about TCP/IP, and about programming itself—everyone grows as they code.

But there are also many things I would do the same, some in contradiction to the standard wisdom.

Things I Would Do Differently

Perhaps my biggest mistake with sendmail was to not recognize early enough how important it was going to be. I had several opportunities to nudge the world in the correct direction but didn't take them; in fact, in some cases I did damage, e.g., by not making sendmail stricter about bad input when it became appropriate to do so. Similarly, I recognized that the configuration file syntax needed to be improved fairly early on, when there were perhaps a few hundred sendmail instances deployed, but decided not to change things because I didn't want to cause the installed user base undue pain. In retrospect it would have been better to improve things early and cause temporary pain in order to produce a better long-term result.

Version 7 Mailbox Syntax

One example of this was the way version 7 mailboxes separated messages. They used a line beginning "From_" (where "_" represents the ASCII space character, 0x20) to separate messages. If a message came in containing the word "From_" at the beginning of the line, local mailbox software converted it to ">From_". One refinement on some but not all systems was to require a preceding blank line, but this could not be relied upon. To this day, ">From" appears in extremely unexpected places that aren't obviously related to email (but clearly were processed by email at one time or another). In retrospect I probably could have converted the BSD mail system to use a new syntax. I would have been roundly cursed at the time, but I would have saved the world a heap of trouble.

Syntax and Contents of Configuration File

Perhaps my biggest mistake in the syntax of the configuration file was the use of tab (HT, 0x09) in rewriting rules to separate the pattern from the replacement. At the time I was emulating make, only to learn years later that Stuart Feldman, the author of make, thought that was one of his biggest mistakes. Besides being non-obvious when looking at the configuration on a screen, the tab character doesn't survive cut-and-paste in most window systems.

Although I believe that rewriting rules were the correct idea (see below), I would change the general structure of the configuration file. For example, I did not anticipate the need for hierarchies in the configuration (e.g., options that would be set differently for different SMTP listener ports). At the time the configuration file was designed there were no "standard" formats. Today, I would be inclined to use an Apache-style configuration—it's clean, neat, and has adequate expressive power—or perhaps even embed a language such as Lua.

When sendmail was developed the address spaces were small and the protocols were still in flux. Putting as much as possible into the configuration file seemed like a good idea. Today, that looks like a mistake: we have plenty of address space (for an MTA) and the standards are fairly static. Furthermore, part of the "configuration file" is really code that needs to be updated in new releases. The .mc configuration file fixes that, but having to rebuild your configuration every time you update the software is a pain. A simple solution to this would simply be to have two configuration files that sendmail would read, one hidden and installed with each new software release and the other exposed and used for local configuration.

Use of Tools

There are many new tools available today—for example, for configuring and building the software. Tools can be good leverage if you need them, but they can also be overkill, making it harder than necessary to understand the system. For example, you should never use a yacc(1) grammar when all you need is strtok(3). But reinventing the wheel isn't a good idea either. In particular, despite some reservations I would almost certainly use autoconf today.

Backward Compatibility

With the benefit of hindsight, and knowing how ubiquitous sendmail became, I would not worry so much about breaking existing installations in the early days of development. When existing practice is seriously broken it should be fixed, not accommodated for. That said, I would still not do strict checking of all message formats; some problems can be easily and safely ignored or patched. For example, I would probably still insert a Message-Id: header field into messages that did not have one, but I would be more inclined to reject messages without a From: header field rather than try to create one from the information in the envelope.

Internal Abstractions

There are certain internal abstractions that I would not attempt again, and others that I would add. For example, I would not use null-terminated strings, opting instead for a length/value pair, despite the fact that this means that much of the Standard C Library becomes difficult to use. The security implications of this alone make it worthwhile. Conversely, I would not attempt to build exception handling in C, but I would create a consistent status code system that would be used throughout the code rather than having routines return null, false, or negative numbers to represent errors.

I would certainly abstract the concept of mailbox names from Unix user ids. At the time I wrote sendmail the model was that you only sent messages to Unix users. Today, that is almost never the case; even on systems that do use that model, there are system accounts that should never receive e-mail.

Things I Would Do The Same

Of course, some things *did* work well...

Syslog

One of the successful side projects from sendmail was syslog. At the time sendmail was written, programs that needed to log had a specific file that they would write. These were scattered around the filesystem. Syslog was difficult to write at the time (UDP didn't exist yet, so I used something called mpx files), but well worth it. However, I would make one specific change: I would pay more attention to making the syntax of logged messages machine parseable—essentially, I failed to predict the existence of log monitoring.

Rewriting Rules

Rewriting rules have been much maligned, but I would use them again (although probably not for as many things as they are used for now). Using the tab character was a clear mistake, but given the

limitations of ASCII and the syntax of e-mail addresses, some escape character is probably required⁷. In general, the concept of using a pattern-replace paradigm worked well and was very flexible.

Avoid Unnecessary Tools

Despite my comment above that I would use more existing tools, I am reluctant to use many of the run-time libraries available today. In my opinion far too many of them are so bloated as to be dangerous. Libraries should be chosen with care, balancing the merits of reuse against the problems of using an overly powerful tool to solve a simple problem. One particular tool I would avoid is XML, at least as a configuration language. I believe that the syntax is too baroque for much of what it is used for. XML has its place, but it is overused today.

Code in C

Some people have suggested that a more natural implementation language would be Java or C++. Despite the well-known problems with C, I would still use it as my implementation language. In part this is personal: I know C much better than I know Java or C++. But I'm also disappointed by the cavalier attitude that most object-oriented languages take toward memory allocation. Allocating memory has many performance concerns that can be difficult to characterize. Sendmail uses object-oriented concepts internally where appropriate (for example, the implementation of map classes), but in my opinion going completely object-oriented is wasteful and overly restrictive.

17.9 Conclusions

The sendmail MTA was born into a world of immense upheaval, a sort of “wild west” that existed when e-mail was ad hoc and the current mail standards were not yet formulated. In the intervening 31 years the “e-mail problem” has changed from just working reliably to working with large messages and heavy load to protecting sites from spam and viruses and finally today to being used as a platform for a plethora of e-mail-based applications. Sendmail has evolved into a work-horse that is embraced by even the most risk-averse corporations, even as e-mail has evolved from pure text person-to-person communications into a multimedia-based mission-critical part of the infrastructure.

The reasons for this success are not always obvious. Building a program that survives and even thrives in a rapidly changing world with only a handful of part-time developers can't be done using conventional software development methodologies. I hope I've provided some insights into how sendmail succeeded.

⁷Somehow I suspect that using Unicode for configuration would not prove popular.

SnowFlock

Roy Bryant and Andrés Lagar-Cavilla

Cloud computing provides an attractively affordable computing platform. Instead of buying and configuring a physical server, with all the associated time, effort and up front costs, users can rent “servers” in the cloud with a few mouse clicks for less than 10 cents per hour. Cloud providers keep their costs low by providing virtual machines (VMs) instead of physical computers. The key enabler is the virtualization software, called a virtual machine monitor (VMM), that emulates a physical machine. Users are safely isolated in their “guest” VMs, and are blissfully unaware that they typically share the physical machine (“host”) with many others.

18.1 Introducing SnowFlock

Clouds are a boon to agile organizations. With physical servers, users are relegated to waiting impatiently while others (slowly) approve the server purchase, place the order, ship the server, and install and configure the Operating System (OS) and application stacks. Instead of waiting weeks for others to deliver, the cloud user retains control of the process and can create a new, standalone server in minutes.

Unfortunately, few cloud servers stand alone. Driven by the quick instantiation and pay-per-use model, cloud servers are typically members of a variable pool of similarly configured servers performing dynamic and scalable tasks related to parallel computing, data mining, or serving web pages. Because they repeatedly boot new instances from the same, static template, commercial clouds fail to fully deliver on the promise of true on-demand computation. After instantiating the server, the cloud user must still manage cluster membership and broker the addition of new servers.

SnowFlock addresses these issues with VM Cloning, our proposed cloud API call. In the same way that application code routinely invokes OS services through a syscall interface, it could now also invoke cloud services through a similar interface. With SnowFlock’s VM Cloning, resource allocation, cluster management, and application logic can be interwoven programmatically and dealt with as a single logical operation.

The VM Cloning call instantiates multiple cloud servers that are identical copies of the originating parent VM up to the point of cloning. Logically, clones inherit all the state of their parent, including OS- and application-level caches. Further, clones are automatically added to an internal private network, thus effectively joining a dynamically scalable cluster. New computation resources, encapsulated as identical VMs, can be created on-the-fly and can be dynamically leveraged as needed.

To be of practical use, VM cloning has to be applicable, efficient, and fast. In this chapter we will describe how SnowFlock’s implementation of VM Cloning can be effectively interwoven in several different programming models and frameworks, how it can be implemented to keep application runtime and provider overhead to a minimum, and how it can be used to create dozens of new VMs in five seconds or less.

With an API for the programmatic control of VM Cloning with bindings in C, C++, Python and Java, SnowFlock is extremely flexible and versatile. We’ve successfully used SnowFlock in prototype implementations of several, quite different, systems. In parallel computation scenarios, we’ve achieved excellent results by explicitly cloning worker VMs that cooperatively distribute the load across many physical hosts. For parallel applications that use the Message Passing Interface (MPI) and typically run on a cluster of dedicated servers, we modified the MPI startup manager to provide unmodified applications with good performance and much less overhead by provisioning a fresh cluster of clones on demand for each run. Finally, in a quite different use case, we used SnowFlock to improve the efficiency and performance of elastic servers. Today’s cloud-based elastic servers boot new, cold workers as needed to service spikes in demand. By cloning a running VM instead, SnowFlock brings new workers on line 20 times faster, and because clones inherit the warm buffers of their parent, they reach their peak performance sooner.

18.2 VM Cloning

As the name suggests, VM clones are (nearly) identical to their parent VM. There are actually some minor but necessary differences to avoid issues such as MAC address collisions, but we’ll come back to that later. To create a clone, the entire local disk and memory state must be made available, which brings us to the first major design tradeoff: should we copy that state up-front or on demand?

The simplest way to achieve VM cloning is to adapt the standard VM “migration” capability. Typically, migration is used when a running VM needs to be moved to a different host, such as when the host becomes overloaded or must be brought down for maintenance. Because the VM is purely software, it can be encapsulated in a data file that can then be copied to a new, more appropriate host, where it picks up execution after a brief interruption. To accomplish this, off-the-shelf VMMs create a file containing a “checkpoint” of the VM, including its local filesystem, memory image, virtual CPU (VCPU) registers, etc. In migration, the newly booted copy replaces the original, but the process can be altered to produce a clone while leaving the original running. In this “eager” process, the entire VM state is transferred up front, which provides the best initial performance, because the entire state of the VM is in place when execution begins. The disadvantage of eager replication is that the laborious process of copying the entire VM must happen before execution can begin, which significantly slows instantiation.

The other extreme, adopted by SnowFlock, is “lazy” state replication. Instead of copying everything the VM might ever need, SnowFlock transfers only the vital bits needed to begin execution, and transfers state later, only when the clone needs it. This has two advantages. First, it minimizes the instantiation latency by doing as little work as possible up front. Second, it increases the efficiency by copying only the state that is actually used by the clone. The yield of this benefit, of course, depends on the clone’s behavior, but few applications access every page of memory and every file in the local filesystem.

However, the benefits of lazy replication aren’t free. Because the state transfer is postponed until the last moment, the clone is left waiting for state to arrive before it can continue execution. This situation parallels swapping of memory to disk in time-shared workstation: applications are

blocked waiting for state to be fetched from a high latency source. In the case of SnowFlock, the blocking somewhat degrades the clone's performance; the severity of the slowdown depends on the application. For high performance computing applications we've found this degradation has little impact, but a cloned database server may perform poorly at first. It should be noted that this is a transient effect: within a few minutes, most of the necessary state has been transferred and the clone's performance matches that of the parent.

As an aside, if you're well versed in VMs, you're likely wondering if the optimizations used by "live" migration are useful here. Live migration is optimized to shorten the interval between the original VM's suspension and the resumption of execution by the new copy. To accomplish this, the Virtual Machine Monitor (VMM) pre-copies the VM's state while the original is still running, so that after suspending it, only the recently changed pages need to be transferred. This technique does not affect the interval between the migration request and the time the copy begins execution, and so would not reduce the instantiation latency of eager VM cloning.

18.3 SnowFlock's Approach

SnowFlock implements VM cloning with a primitive called "VM Fork", which is like a standard Unix fork, but with a few important differences. First, rather than duplicating a single process, VM Fork duplicates an entire VM, including all of memory, all processes and virtual devices, and the local filesystem. Second, instead of producing a single copy running on the same physical host, VM Fork can simultaneously spawn many copies in parallel. Finally, VMs can be forked to distinct physical servers, letting you quickly increase your cloud footprint as needed.

The following concepts are key to SnowFlock:

- **Virtualization:** The VM encapsulates the computation environment, making clouds and machine cloning possible.
- **Lazy Propagation:** The VM state isn't copied until it's needed, so clones come alive in a few seconds.
- **Multicast:** Clone siblings have similar needs in terms of VM state. With multicast, dozens of clones start running as quickly as one.
- **Page Faults:** When a clone tries to use missing memory, it faults and triggers a request to the parent. The clone's execution is blocked until the needed page arrives.
- **Copy on Write (CoW):** By taking a copy of its memory and disk pages before overwriting them, the parent VM can continue to run while preserving a frozen copy of its state for use by the clones.

We've implemented SnowFlock using the Xen virtualization system, so it's useful to introduce some Xen-specific terminology for clarity. In a Xen environment, the VMM is called the hypervisor, and VMs are called domains. On each physical machine (host), there is a privileged domain, called "domain 0" (dom0), that has full access to the host and its physical devices, and can be used to control additional guest, or "user", VMs that are called "domain U" (domU).

In broad strokes, SnowFlock consists of a set of modifications to the Xen hypervisor that enable it to smoothly recover when missing resources are accessed, and a set of supporting processes and systems that run in dom0 and cooperatively transfer the missing VM state, and some optional modifications to the OS executing inside clone VMs. There are six main components.

- **VM Descriptor:** This small object is used to seed the clone, and holds the bare-bones skeleton of the VM as needed to begin execution. It lacks the guts and muscle needed to perform any useful work.
- **Multicast Distribution System (mcdist):** This parent-side system efficiently distributes the VM state information simultaneously to all clones.
- **Memory Server Process:** This parent-side process maintains a frozen copy of the parent's state, and makes it available to all clones on demand through mcdist.
- **Memtap Process:** This clone-side process acts on the clone's behalf, and communicates with the memory server to request pages that are needed but missing.
- **Clone Enlightenment:** The guest kernel running inside the clones can alleviate the on-demand transfer of VM state by providing hints to the VMM. This is optional but highly desirable for efficiency.
- **Control Stack:** Daemons run on each physical host to orchestrate the other components and manage the SnowFlock parent and clone VMs.

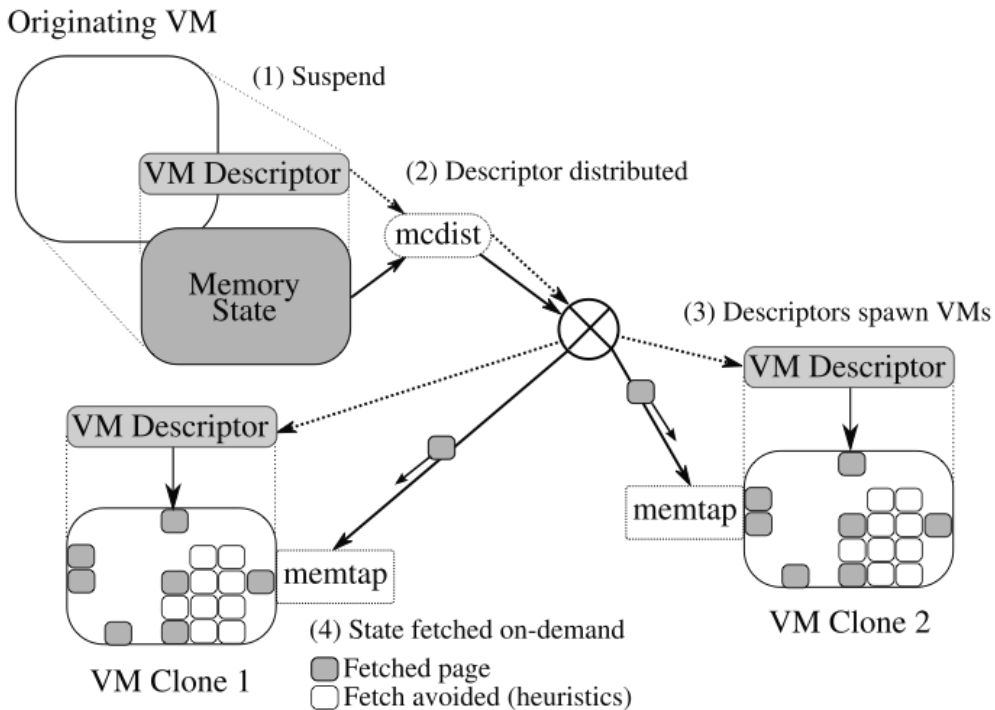


Figure 18.1: SnowFlock VM Replication Architecture

Pictorially speaking, Figure 18.1 depicts the process of cloning a VM, showing the the four main steps: (1) suspending the parent VM to produce an architectural descriptor; (2) distributing this descriptor to all target hosts; (3) initiating clones that are mostly empty state-wise; and (4) propagating state on-demand. The figure also depicts the use of multicast distribution with mcdist, and fetch avoidance via guest enlightenment.

If you're interested in trying SnowFlock, it's available in two flavors. The documentation and open source code for the original University of Toronto SnowFlock research project are available¹. If you'd prefer to take the industrial-strength version for a spin, a free, non-commercial license is available from GridCentric Inc.² Because SnowFlock includes changes to the hypervisor and requires access to dom0, installing SnowFlock requires privileged access on the host machines. For that reason, you'll need to use your own hardware, and won't be able to try it out as a user in a commercial cloud environment such as Amazon's EC2.

Throughout the next few sections we'll describe the different pieces that cooperate to achieve instantaneous and efficient cloning. All the pieces we will describe fit together as shown in Figure 18.2.

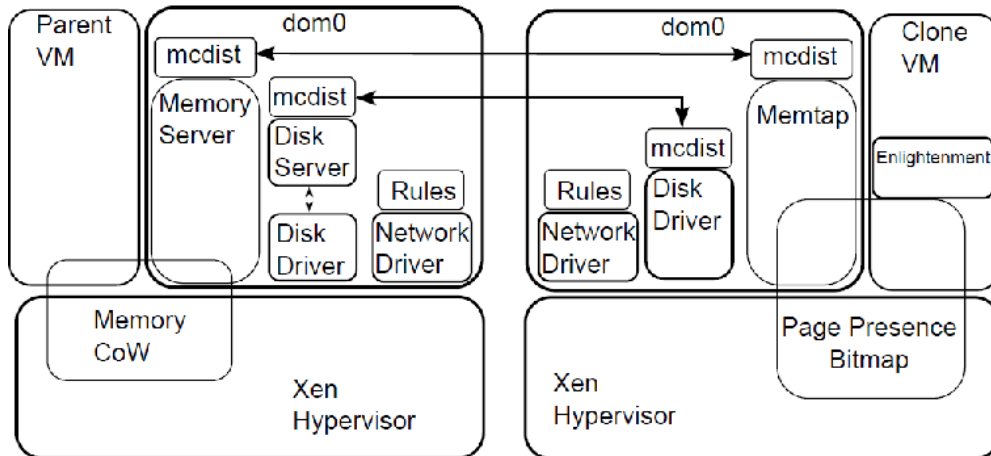


Figure 18.2: Software Components of SnowFlock

18.4 Architectural VM Descriptor

The key design decision for SnowFlock is to postpone the replication of VM state to a lazy runtime operation. In other words, copying the memory of a VM is a late binding operation, allowing for many opportunities for optimization.

The first step to carry out this design decision is the generation of an architectural descriptor of the VM state. This is the seed that will be used to create clone VMs. It contains the bare minimum necessary to create a VM and make it schedulable. As the name implies, this bare minimum consists of data structures needed by the underlying architectural specification. In the case of SnowFlock, the architecture is a combination of Intel x86 processor requirements and Xen requirements. The architectural descriptor thus contains data structures such as page tables, virtual registers, device metadata, wallclock timestamps, etc. We refer the interested reader to [LCWB⁺11] for an in-depth description of the contents of the architectural descriptor.

An architectural descriptor has three important properties: First, it can be created in little time; 200 milliseconds is not uncommon. Second, it is small, typically three orders of magnitude smaller

¹<http://sysweb.cs.toronto.edu/projects/1>

²<http://www.gridcentriclabs.com/architecture-of-open-source-applications>

than the memory allocation of the originating VM (1 MB for a 1 GB VM). And third, a clone VM can be created from a descriptor in less than a second (typically 800 milliseconds).

The catch, of course, is that the cloned VMs are missing most of their memory state by the time they are created from the descriptor. The following sections explain how we solve this problem—and how we take advantage of the opportunities for optimization it presents.

18.5 Parent-Side Components

Once a VM is cloned it becomes a parent for its children or clones. Like all responsible parents, it needs to look out for the well-being of its descendants. It does so by setting up a set of services that provision memory and disk state to cloned VMs on demand.

Memserver Process

When the architectural descriptor is created, the VM is stopped in its tracks throughout the process. This is so the VM memory state settles; before actually pausing a VM and descheduling from execution, internal OS drivers quiesce into a state from which clones can reconnect to the external world in their new enclosing VMs. We take advantage of this quiescent state to create a “memory server”, or memserver.

The memory server will provide all clones with the bits of memory they need from the parent. Memory is propagated at the granularity of an x86 memory page (4 kbytes). In its simplest form, the memory server sits waiting for page requests from clones, and serves one page at a time, one clone at a time.

However, this is the very same memory the parent VM needs to use to keep executing. If we would allow the parent to just go ahead and modify this memory, we would serve corrupted memory contents to clone VMs: the memory served would be different from that at the point of cloning, and clones would be mightily confused. In kernel hacking terms, this is a sure recipe for stack traces.

To circumvent this problem, a classical OS notion comes to the rescue: Copy-on-Write, or CoW memory. By enlisting the aid of the Xen hypervisor, we can remove writing privileges from all pages of memory in the parent VM. When the parent actually tries to modify one page, a hardware page fault is triggered. Xen knows why this happened, and makes a copy of the page. The parent VM is allowed to write to the original page and continue execution, while the memory server is told to use the copy, which is kept read-only. In this manner, the memory state at the point of cloning remains frozen so that clones are not confused, while the parent is able to proceed with execution. The overhead of CoW is minimal: similar mechanisms are used by Linux, for example, when creating new processes.

Multicasting with Mcdist

Clones are typically afflicted with an existential syndrome known as “fate determinism.” We expect clones to be created for a single purpose: for example, to align X chains of DNA against a segment Y of a database. Further, we expect a set of clones to be created so that all siblings do the same, perhaps aligning the same X chains against different segments of the database, or aligning different chains against the same segment Y. Clones will thus clearly exhibit a great amount of temporal locality in their memory accesses: they will use the same code and large portions of common data.

We exploit the opportunities for temporal locality through `mcdist`, our own multicast distribution system tailored to SnowFlock. `Mcdist` uses IP multicast to simultaneously distribute the same packet to a set of receivers. It takes advantage of network hardware parallelism to decrease the load on the memory server. By sending a reply to all clones on the first request for a page, each clone's requests act as a prefetch for its siblings, because of their similar memory access patterns.

Unlike other multicast systems, `mcdist` does not have to be reliable, does not have to deliver packets in an ordered manner, and does not have to atomically deliver a reply to all intended receivers. Multicast is strictly an optimization, and delivery need only be ensured to the clone explicitly requesting a page. The design is thus elegantly simple: the server simply multicasts responses, while clients time-out if they have not received a reply for a request, and retry the request.

Three optimizations specific to SnowFlock are included in `mcdist`:

- **Lockstep Detection:** When temporal locality does happen, multiple clones request the same page in very close succession. The `mcdist` server ignores all but the first of such requests.
- **Flow Control:** Receivers piggyback their receive rate on requests. The server throttles its sending rate to a weighted average of the clients' receive rate. Otherwise, receivers will be drowned by too many pages sent by an eager server.
- **End Game:** When the server has sent most pages, it falls back to unicast responses. Most requests at this point are retries, and thus blasting a page through the wire to all clones is unnecessary.

Virtual Disk

SnowFlock clones, due to their short life span and fate determinism, rarely use their disk. The virtual disk for a SnowFlock VM houses the root partition with binaries, libraries and configuration files. Heavy data processing is done through suitable filesystems such as HDFS (Chapter 9) or PVFS. Thus, when SnowFlock clones decide to read from their root disk, they typically have their requests satisfied by the kernel filesystem page cache.

Having said that, we still need to provide access to the virtual disk for clones, in the rare instance that such access is needed. We adopted the path of least resistance here, and implemented the disk by closely following the memory replication design. First, the state of the disk is frozen at the time of cloning. The parent VM keeps using its disk in a CoW manner: writes are sent to a separate location in backing storage, while the view of the disk clones expect remains immutable. Second, disk state is multicast to all clones, using `mcdist`, with the same 4 KB page granularity, and under the same expectations of temporal locality. Third, replicated disk state for a clone VM is strictly transient: it is stored in a sparse flat file which is deleted once the clone is destroyed.

18.6 Clone-Side Components

Clones are hollow shells when they are created from an architectural descriptor, so like everybody else, they need a lot of help from their parents to grow up: the children VMs move out and immediately call home whenever they notice something they need is missing, asking their parent to send it over right away.

Memtap Process

Attached to each clone after creation, the memtap process is the lifeline of a clone. It maps all of the memory of the clone and fills it on demand as needed. It enlists some crucial bits of help from the Xen hypervisor: access permission to the memory pages of the clones is turned off, and hardware faults caused by first access to a page are routed by the hypervisor into the memtap process.

In its simplest incarnation, the memtap process simply asks the memory server for the faulting page, but there are more complicated scenarios as well. First, memtap helpers use `mcdist`. This means that at any point in time, any page could arrive by virtue of having been requested by another clone—the beauty of asynchronous prefetching. Second, we allow SnowFlock VMs to be multi-processor VMs. There wouldn't be much fun otherwise. This means that multiple faults need to be handled in parallel, perhaps even for the same page. Third, in later versions memtap helpers can explicitly prefetch a batch of pages, which can arrive in any order given the lack of guarantees from the `mcdist` server. Any of these factors could have led to a concurrency nightmare, and we have all of them.

The entire memtap design centers on a page presence bitmap. The bitmap is created and initialized when the architectural descriptor is processed to create the clone VM. The bitmap is a flat bit array sized by the number of pages the VM's memory can hold. Intel processors have handy atomic bit mutation instructions: setting a bit, or doing a test and set, can happen with the guarantee of atomicity with respect to other processors in the same box. This allows us to avoid locks in most cases, and thus to provide access to the bitmap by different entities in different protection domains: the Xen hypervisor, the memtap process, and the cloned guest kernel itself.

When Xen handles a hardware page fault due to a first access to a page, it uses the bitmap to decide whether it needs to alert memtap. It also uses the bitmap to enqueue multiple faulting virtual processors as dependent on the same absent page. Memtap buffers pages as they arrive. When its buffer is full or an explicitly requested page arrives, the VM is paused, and the bitmap is used to discard any duplicate pages that have arrived but are already present. Any remaining pages that are needed are then copied into the VM memory, and the appropriate bitmap bits are set.

Clever Clones Avoid Unnecessary Fetches

We just mentioned that the page presence bitmap is visible to the kernel running inside the clone, and that no locks are needed to modify it. This gives clones a powerful “enlightenment” tool: they can prevent the fetching of pages by modifying the bitmap and pretending they are present. This is extremely useful performance-wise, and safe to do when pages will be completely overwritten before they are used.

There happens to be a very common situation when this happens and fetches can be avoided. All memory allocations in the kernel (using `vmalloc`, `kzalloc`, `get_free_page`, user-space `brk`, and the like) are ultimately handled by the kernel page allocator. Typically pages are requested by intermediate allocators which manage finer-grained chunks: the slab allocator, the glibc malloc allocator for a user-space process, etc. However, whether allocation is explicit or implicit, one key semantic implication always holds true: no one cares about what the page contained, because its contents will be arbitrarily overwritten. Why fetch such a page, then? There is no reason to do so, and empirical experience shows that avoiding such fetches is tremendously advantageous.

<code>sf_request_ticket(n)</code>	Requests an allocation for n clones. Returns a ticket describing an allocation for $m \leq n$ clones.
<code>sf_clone(ticket)</code>	Clones, using the ticket allocation. Returns the clone ID, $0 \leq ID < m$.
<code>sf_checkpoint_parent()</code>	Prepares an immutable checkpoint C of the parent VM to be used for creating clones at an arbitrarily later time.
<code>sf_create_clones(C, ticket)</code>	Same as <code>sf_clone</code> , uses the checkpoint C . Clones will begin execution at the point at which the corresponding <code>sf_checkpoint_parent()</code> was invoked.
<code>sf_exit()</code>	For children ($1 \leq ID < m$), terminates the child.
<code>sf_join(ticket)</code>	For the parent ($ID = 0$), blocks until all children in the ticket reach their <code>sf_exit</code> call. At that point all children are terminated and the ticket is discarded.
<code>sf_kill(ticket)</code>	Parent only, discards ticket and immediately kills all associated children.

Table 18.1: The SnowFlock VM Cloning API

18.7 VM Cloning Application Interface

So far we have focused on the internals of cloning a VM efficiently. As much fun as solipsistic systems can be, we need to turn our attention to those who will use the system: applications.

API Implementation

VM Cloning is offered to the application via the simple SnowFlock API, depicted in Figure 18.1. Cloning is basically a two-stage process. You first request an allocation for the clone instances, although due to the system policies that are in effect, that allocation may be smaller than requested. Second, you use the allocation to clone your VM. A key assumption is that your VM focuses on a single operation. VM Cloning is appropriate for single-application VMs such as a web server or a render farm component. If you have a hundred-process desktop environment in which multiple applications concurrently call VM cloning, you're headed for chaos.

The API simply marshals messages and posts them to the XenStore, a shared-memory low-throughput interface used by Xen for control plane transactions. A SnowFlock Local Daemon (SFLD) executes on the hypervisor and listens for such requests. Messages are unmarshalled, executed, and requests posted back.

Programs can control VM Cloning directly through the API, which is available for C, C++, Python and Java. Shell scripts that harness the execution of a program can use the provided command-line scripts instead. Parallel frameworks such as MPI can embed the API: MPI programs can then use SnowFlock without even knowing, and with no modification to their source. Load balancers sitting in front of web or application servers can use the API to clone the servers they manage.

SFLDs orchestrate the execution of VM Cloning requests. They create and transmit architectural descriptors, create cloned VMs, launch disk and memory servers, and launch memtap helper processes. They are a miniature distributed system in charge of managing the VMs in a physical cluster.

SFLDs defer allocation decisions to a central SnowFlock Master Daemon (SFMD). SFMD simply interfaces with appropriate cluster management software. We did not see any need to reinvent the wheel here, and deferred decisions on resource allocation, quotas, policies, etc. to suitable software such as Sun Grid Engine or Platform EGO.

Necessary Mutations

After cloning, most of the cloned VM's processes have no idea that they are no longer the parent, and that they are now running in a copy. In most aspects, this just works fine and causes no issues. After all, the primary task of the OS is to isolate applications from low-level details, such as the network identity. Nonetheless, making the transition smooth requires a set of mechanisms to be put in place. The meat of the problem is in managing the clone's network identity; to avoid conflicts and confusion, we must introduce slight mutations during the cloning process. Also, because these tweaks may necessitate higher-level accommodations, a hook is inserted to allow the user to configure any necessary tasks, such as (re)mounting network filesystems that rely on the clone's identity.

Clones are born to a world that is mostly not expecting them. The parent VM is part of a network managed most likely by a DHCP server, or by any other of the myriad ways sysadmins find to do their job. Rather than assume a necessarily inflexible scenario, we place the parent and all clones in their own private virtual network. Clones from the same parent are all assigned a unique ID, and their IP address in this private network is automatically set up upon cloning as a function of the ID. This guarantees that no intervention from a sysadmin is necessary, and that no IP address collisions will ever happen.

IP reconfiguration is performed directly by a hook we place on the virtual network driver. However, we also rig the driver to automatically generate synthetic DHCP responses. Thus, regardless of your choice of distribution, your virtual network interface will ensure that the proper IP coordinates are propagated to the guest OS, even when you are restarting from scratch.

To prevent clones from different parents colliding with each others' virtual private networks—and to prevent mutual DDoS attacks—clone virtual network are isolated at the Ethernet (or layer 2) level. We hijack a range of Ethernet MAC OUIs³ and dedicate them to clones. The OUI will be a function of the parent VM. Much like the ID of a VM determines its IP address, it also determines its non-OUI Ethernet MAC address. The virtual network driver translates the MAC address the VM believes it has to the one assigned as a function of its ID, and filters out all traffic to and from virtual private network with different OUIs. This isolation is equivalent to that achievable via ebtables, although much simpler to implement.

Having clones talk only to each other may be fun, but not fun enough. Sometimes we will want our clones to reply to HTTP requests from the Internet, or mount public data repositories. We equip any set of parent and clones with a dedicated router VM. This tiny VM performs firewalling, throttling and NATing of traffic from the clones to the Internet. It also limits inbound connections to the parent VM and well-known ports. The router VM is lightweight but represents a single point of centralization for network traffic, which can seriously limit scalability. The same network rules could be applied in a distributed fashion to each host upon which a clone VM runs. We have not released that experimental patch.

SFLDs assign IDs, and teach the virtual network drivers how they should configure themselves: internal MAC and IP addresses, DHCP directives, router VM coordinates, filtering rules, etc.

³OUI, or Organizational Unique ID, is a range of MAC addresses assigned to a vendor.

18.8 Conclusion

By tweaking the Xen hypervisor and lazily transferring the VM's state, SnowFlock can produce dozens of clones of a running VM in a few seconds. Cloning VMs with SnowFlock is thus instantaneous and live—it improves cloud usability by automating cluster management and giving applications greater programmatic control over the cloud resources. SnowFlock also improves cloud agility by speeding up VM instantiation by a factor of 20, and by improving the performance of most newly created VMs by leveraging their parent's warm, in-memory OS and application caches. The keys to SnowFlock's efficient performance are heuristics that avoid unnecessary page fetches, and the multicast system that lets clone siblings cooperatively prefetch their state. All it took was the clever application of a few tried-and-true techniques, some sleight of hand, and a generous helping of industrial-strength debugging.

We learned two important lessons throughout the SnowFlock experience. The first is the often-underestimated value of the KISS theorem. We were expecting to implement complicated prefetching techniques to alleviate the spate of requests for memory pages a clone would issue upon startup. This was, perhaps surprisingly, not necessary. The system performs very well for many workloads based on one single principle: bring the memory over as needed. Another example of the value of simplicity is the page presence bitmap. A simple data structure with clear atomic access semantics greatly simplifies what could have been a gruesome concurrency problem, with multiple virtual CPUs competing for page updates with the asynchronous arrival of pages via multicast.

The second lesson is that scale does not lie. In other words, be prepared to have your system shattered and new bottlenecks uncovered every time you bump your scale by a power of two. This is intimately tied with the previous lesson: simple and elegant solutions scale well and do not hide unwelcome surprises as load increases. A prime example of this principle is our `mcdist` system. In large-scale tests, a TCP/IP-based page distribution mechanism fails miserably for hundreds of clones. `Mcdist` succeeds by virtue of its extremely constrained and well-defined roles: clients only care about their own pages; the server only cares about maintaining a global flow control. By keeping `mcdist` humble and simple, SnowFlock is able to scale extremely well.

If you are interested in knowing more, you can visit the University of Toronto site⁴ for the academic papers and open-source code licensed under GPLv2, and GridCentric⁵ for an industrial strength implementation.

⁴<http://sysweb.cs.toronto.edu/projects/1>

⁵<http://www.gridcentriclabs.com/>

SocialCalc

Audrey Tang

The history of spreadsheets spans more than 30 years. The first spreadsheet program, VisiCalc, was conceived by Dan Bricklin in 1978 and shipped in 1979. The original concept was quite straightforward: a table that spans infinitely in two dimensions, its cells populated with text, numbers, and formulas. Formulas are composed of normal arithmetic operators and various built-in functions, and each formula can use the current contents of other cells as values.

Although the metaphor was simple, it had many applications: accounting, inventory, and list management are just a few. The possibilities were practically limitless. All these uses made VisiCalc into the first “killer app” of the personal computer era.

In the decades that followed successors like Lotus 1-2-3 and Excel made incremental improvements, but the core metaphor stayed the same. Most spreadsheets were stored as on-disk files, and loaded into memory when opened for editing. Collaboration was particularly hard under the file-based model:

- Each user needed to install a version of the spreadsheet editor.
- E-mail ping-pong, shared folders, or setting up a dedicated version-control system all added bookkeeping overhead.
- Change tracking was limited; for example, Excel does not preserve history for formatting changes and cell comments.
- Updating formatting or formulas in templates required painstaking changes to existing spreadsheet files that used that template.

Fortunately, a new collaboration model emerged to address these issues with elegant simplicity. It is the wiki model, invented by Ward Cunningham in 1994, and popularized by Wikipedia in the early 2000s.

Instead of files, the wiki model features server-hosted pages, editable in the browser without requiring special software. Those hypertext pages can easily link to each other, and even include portions of other pages to form a larger page. All participants view and edit the latest version by default, with revision history automatically managed by the server.

Inspired by the wiki model, Dan Bricklin started working on WikiCalc in 2005. It aims to combine the authoring ease and multi-person editing of wikis with the familiar visual formatting and calculating metaphor of spreadsheets.

19.1 WikiCalc

The first version of WikiCalc (Figure 19.1) had several features that set it apart from other spreadsheets at the time:

- Plain text, HTML, and wiki-style markup rendering for text data.
- Wiki-style text that includes commands to insert links, images, and values from cell references.
- Formula cells may reference values of other WikiCalc pages hosted on other websites.
- Ability to create output to be embedded in other web pages, both static and live data.
- Cell formatting with access to CSS style attributes and CSS classes.
- Logging of all edit operations as an audit trail.
- Wiki-like retention of each new version of a page with roll-back capability.

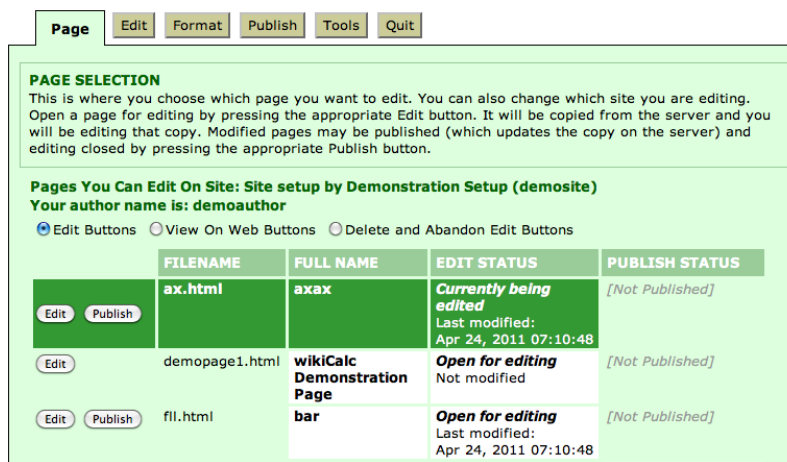


Figure 19.1: WikiCalc 1.0 Interface

WikiCalc 1.0's internal architecture (Figure 19.2) and information flow (Figure 19.3) were deliberately simple, but nevertheless powerful. The ability to compose a master spreadsheet from several smaller spreadsheets proved particularly handy. For example, imagine a scenario where each salesperson keeps numbers in a spreadsheet page. Each sales manager then rolls up their reps' numbers into a regional spreadsheet, and the VP of sales then rolls up the regional numbers into a top-level spreadsheet.

Each time one of the individual spreadsheets is updated, all the roll-up spreadsheets can reflect the update. If someone wants further detail, they simply click through to view the spreadsheet behind the spreadsheet. This roll-up capability eliminates the redundant and error-prone effort of updating numbers in multiple places, and ensures all views of the information stay fresh.

To ensure the recalculations are up-to-date, WikiCalc adopted a thin-client design, keeping all the state information on the server side. Each spreadsheet is represented on the browser as a `<table>` element; editing a cell will send an `ajaxsetcell` call to the server, and the server then tells the browser which cells need updating.

Unsurprisingly, this design depends on a fast connection between the browser and the server. When the latency is high, users will start to notice the frequent appearance of "Loading..." messages

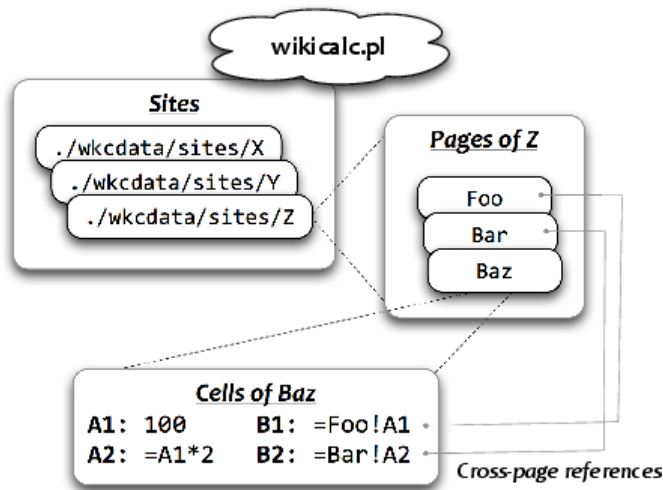


Figure 19.2: WikiCalc Components

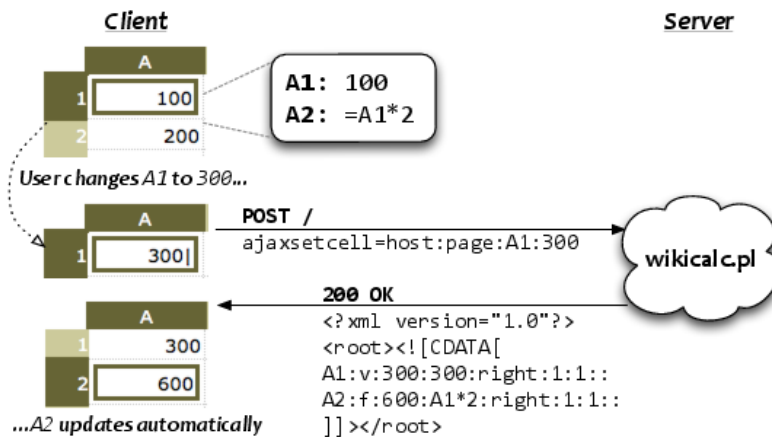


Figure 19.3: WikiCalc Flow

between updating a cell and seeing its new contents as shown in Figure 19.4. This is especially a problem for users interactively editing formulas by tweaking the input and expecting to see results in real time.

Moreover, because the `<table>` element had the same dimensions as the spreadsheet, a 100×100 grid would create 10,000 `<td>` DOM objects, which strains the memory resource of browsers, further limiting the size of pages.

Due to these shortcomings, while WikiCalc was useful as a stand-alone server running on localhost, it was not very practical to embed as part of web-based content management systems.

In 2006, Dan Bricklin teamed up with Socialtext to start developing SocialCalc, a ground-up rewrite of WikiCalc in Javascript based on some of the original Perl code.

	A	B	C	D
1	Loading...			
2				
3	Sample financial calculation in a table with borders	Year	2006	2007
4		Sales	Loading...	170.5
5		Cost	124.0	136.4
6		Profit	31.0	34.1

Figure 19.4: Loading Message

This rewrite was aimed at large, distributed collaborations, and sought to deliver a look and feel more like that of a desktop app. Other design goals included:

- Capable of handling hundreds of thousands of cells.
- Fast turnaround time for edit operations.
- Client-side audit trail and undo/redo stack.
- Better use of Javascript and CSS to provide full-fledged layout functionality.
- Cross-browser support, despite the more extensive use of responsive Javascript.

After three years of development and various beta releases, Socialtext released SocialCalc 1.0 in 2009, successfully meeting the design goals. Let's now take a look at the architecture of the SocialCalc system.

19.2 SocialCalc

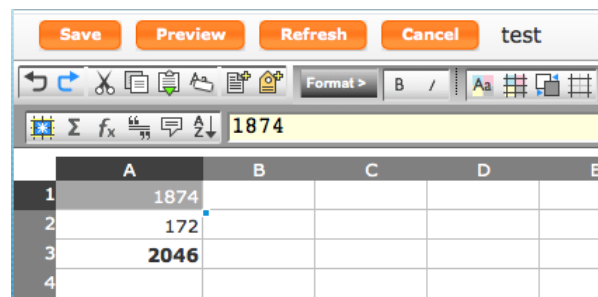


Figure 19.5: SocialCalc Interface

Figure 19.5 and Figure 19.6 show SocialCalc's interface and classes respectively. Compared to WikiCalc, the server's role has been greatly reduced. Its only responsibility is responding to HTTP GETs by serving entire spreadsheets serialized in the save format; once the browser receives the data, all calculations, change tracking and user interaction are now implemented in Javascript.

The Javascript components were designed with a layered MVC (Model/View/Controller) style, with each class focusing on a single aspect:

SocialCalc Class Diagram

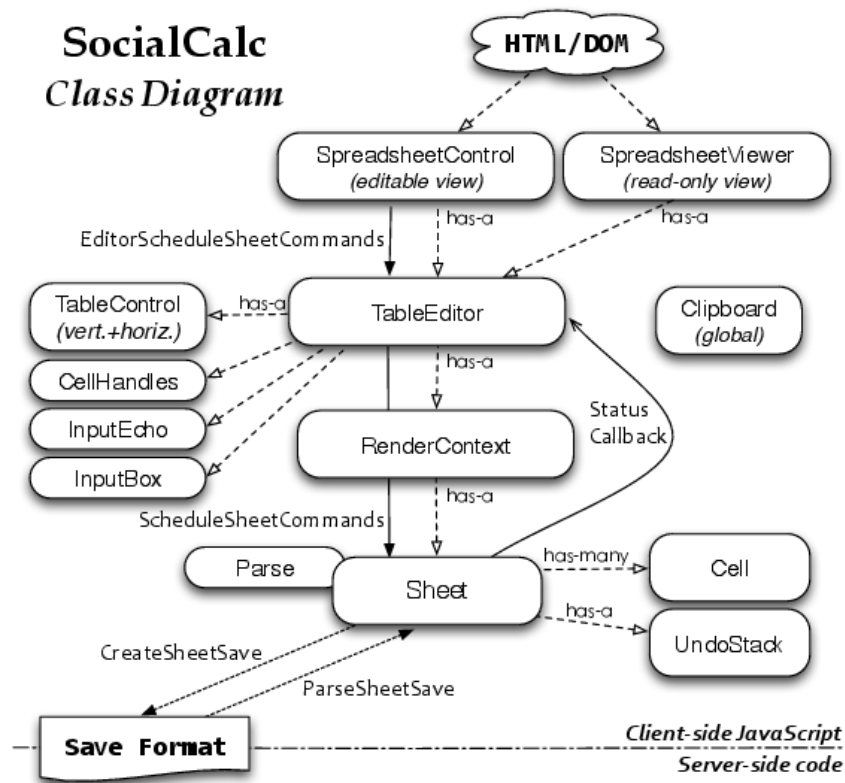


Figure 19.6: SocialCalc Class Diagram

Sheet is the data model, representing an in-memory structure of a spreadsheet. It contains a dictionary from coordinates to *Cell* objects, each representing a single cell. Empty cells need no entries, and hence consume no memory at all.

Cell represents a cell's content and formats. Some common properties are shown in Table 19.1.

RenderContext implements the view; it is responsible for rendering a sheet into DOM objects.

TableControl is the main controller, accepting mouse and keyboard events. As it receives view events such as scrolling and resizing, it updates its associated *RenderContext* object. As it receives update events that affects the sheet's content, it schedules new commands to the sheet's command queue.

SpreadSheetControl is the top-level UI with toolbars, status bars, dialog boxes and color pickers.

SpreadSheetViewer is an alternate top-level UI that provides a read-only interactive view.

We adopted a minimal class-based object system with simple composition/delegation, and make no use of inheritance or object prototypes. All symbols are placed under the `SocialCalc.*` namespace to avoid naming conflicts.

Each update on the sheet goes through the `ScheduleSheetCommands` method, which takes a command string representing the edit. (Some common commands are show in Table 19.2.) The

datatype	t
datavalue	1Q84
color	black
bgcolor	white
font	italic bold 12pt Ubuntu
comment	Ichi-Kyu-Hachi-Yon

Table 19.1: Cell Contents and Formats

application embedding SocialCalc may define extra commands on their own, by adding named callbacks into the `SocialCalc.SheetCommandInfo.CmdExtensionCallbacks` object, and use the `startcmdextension` command to invoke them.

set	sheet defaultcolor blue	erase	A2
set	A width 100	cut	A3
set	A1 value n 42	paste	A4
set	A2 text t Hello	copy	A5
set	A3 formula A1*2	sort	A1:B9 A up B down
set	A4 empty	name	define Foo A1:A5
set	A5 bgcolor green	name	desc Foo Used in formulas like SUM(Foo)
merge	A1:B2	name	delete Foo
unmerge	A1	startcmdextension	UserDefined args

Table 19.2: SocialCalc Commands

19.3 Command Run-loop

To improve responsiveness, SocialCalc performs all recalculation and DOM updates in the background, so the user can keep making changes to several cells while the engine catches up on earlier changes in the command queue.

When a command is running, the `TableEditor` object sets its busy flag to true; subsequent commands are then pushed into the `deferredCommands` queue, ensuring a sequential order of execution. As the event loop diagram in Figure 19.7 shows, the `Sheet` object keeps sending `StatusCallback` events to notify the user of the current state of command execution, through each of the four steps:

ExecuteCommand: Sends `cmdstart` upon start, and `cmdend` when the command finishes execution. If the command changed a cell’s value indirectly, enter the *Recalc* step. Otherwise, if the command changed the visual appearance of one or more on-screen cells, enter the *Render* step. If neither of the above applies (for example with the copy command), skip to the *PositionCalculations* step.

SocialCalc

Command Run-loop

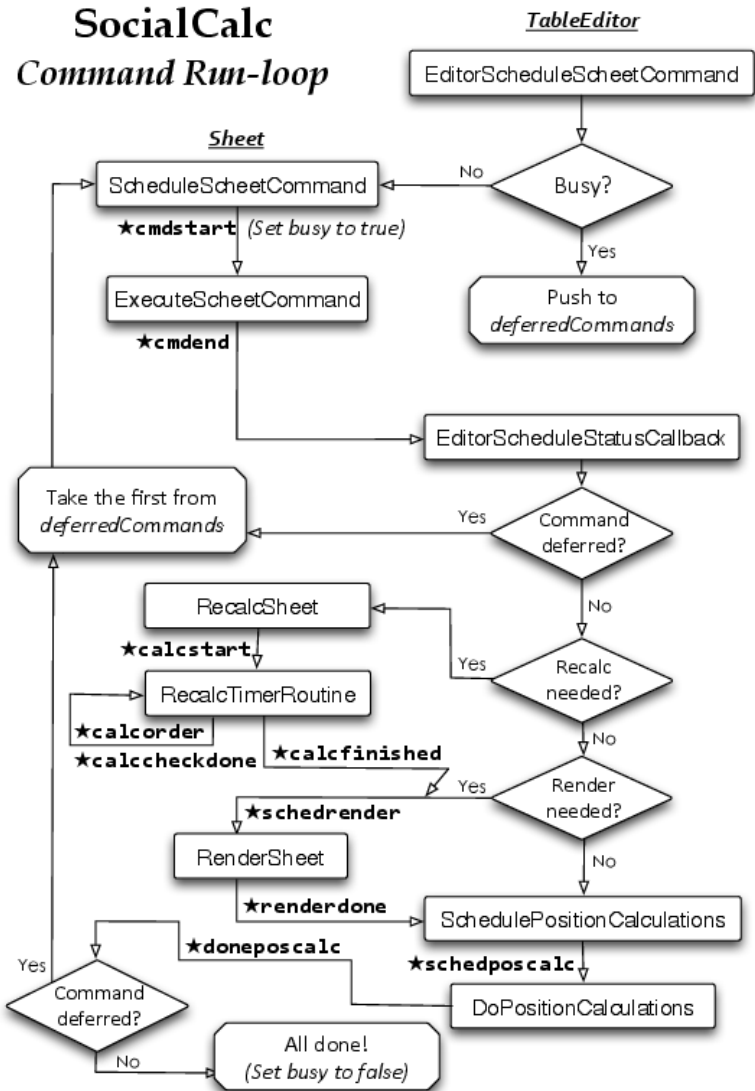


Figure 19.7: SocialCalc Command Run-loop

Recalc (as needed): Sends *calcstart* upon start, *calccorder* every 100ms when checking the dependency chain of cells, *calccheckdone* when the check finishes, and *calcfinished* when all affected cells received their re-calculated values. This step is always followed by the *Render* step.

Render (as needed): Sends schedrender upon start, and renderdone when the <table> element is updated with formatted cells. This step is always followed by *PositionCalculations*.

PositionCalculations: Sends schedposcalc upon start, and doneposcalc after updating the scrollbars, the current editable cell cursor, and other visual components of the TableEditor.

Because all commands are saved as they are executed, we naturally get an audit log of all operations. The `Sheet.CreateAuditString` method provides a newline-delimited string as the audit trail, with each command in a single line.

`ExecuteSheetCommand` also creates an undo command for each command it executes. For example, if the cell A1 contains “Foo” and the user executes `set A1 text Bar`, then an undo-command `set A1 text Foo` is pushed to the undo stack. If the user clicks Undo, then the undo-command is executed to restore A1 to its original value.

19.4 Table Editor

Now let’s look at the `TableEditor` layer. It calculates the on-screen coordinates of its `RenderContext`, and manages horizontal/vertical scroll bars through two `TableControl` instances.

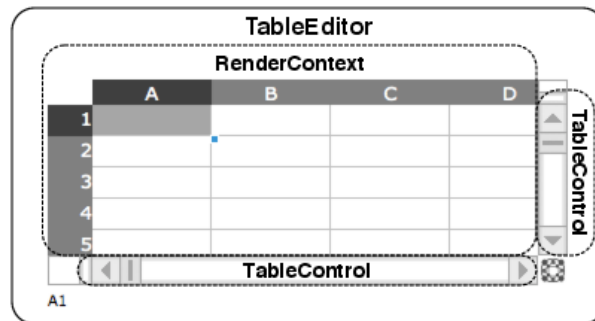


Figure 19.8: `TableControl` Instances Manage Scroll Bars

The view layer, handled by the `RenderContext` class, also differs from WikiCalc’s design. Instead of mapping each cell to a `<td>` element, we now simply create a fixed-size `<table>` that fits the browser’s visible area, and pre-populate it with `<td>` elements.

As the user scrolls the spreadsheet through our custom-drawn scroll bars, we dynamically update the innerHTML of the pre-drawn `<td>` elements. This means we don’t need to create or destroy any `<tr>` or `<td>` elements in many common cases, which greatly speeds up response time.

Because `RenderContext` only renders the visible region, the size of `Sheet` object can be arbitrarily large without affecting its performance.

`TableEditor` also contains a `CellHandles` object, which implements the radial fill/move/slide menu attached to the bottom-right corner to the current editable cell, known as the `ECell`, shown in Figure 19.9.

The input box is managed by two classes: `InputBox` and `InputEcho`. The former manages the above-the-grid edit row, while the latter shows an updated-as-you-type preview layer, overlaying the `ECell`’s content (Figure 19.10).

Usually, the SocialCalc engine only needs to communicate to the server when opening a spreadsheet for edit, and when saving it back to server. For this purpose, the `Sheet.ParseSheetSave` method parses a save format string into a `Sheet` object, and the `Sheet.CreateSheetSave` method serializes a `Sheet` object back into the save format.

Formulas may refer to values from any remote spreadsheet with a URL. The `recalc` command re-fetches the externally referenced spreadsheets, parses them again with `Sheet.ParseSheetSave`,

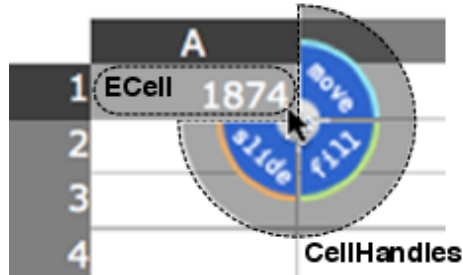


Figure 19.9: Current Editable Cell, Known as the ECell

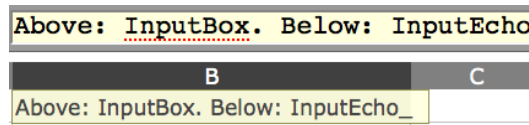


Figure 19.10: The Input Box is Managed by Two Classes

and stores them in a cache so the user can refer to other cells in the same remote spreadsheets without re-fetching its content.

19.5 Save Format

The save format is in standard MIME multipart/mixed format, consisting of four text/plain; charset=UTF-8 parts, each part containing newline-delimited text with colon-delimited data fields. The parts are:

- The meta part lists the types of the other parts.
- The sheet part lists each cell's format and content, each column's width (if not default), the sheet's default format, followed by a list of fonts, colors and borders used in the sheet.
- The optional edit part saves the TableEditor's edit state, including ECell's last position, as well as the fixed sizes of row/column panes.
- The optional audit part contains the history of commands executed in the previous editing session.

For example, Figure 19.11 shows a spreadsheet with three cells, with 1874 in A1 as the ECell, the formula $2^2 \times 43$ in A2, and the formula SUM(Foo) in A3 rendered in bold, referring to the named range Foo over A1:A2.

The serialized save format for the spreadsheet looks like this:

```
socialcalc:version:1.0
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary=SocialCalcSpreadsheetControlSave
--SocialCalcSpreadsheetControlSave
Content-type: text/plain; charset=UTF-8
```

	A	
1	1874	
2	"Foo"	=2^2*43
3	2046	=SUM(Foo)

Figure 19.11: A Spreadsheet with Three Cells

```
# SocialCalc Spreadsheet Control Save
version:1.0
part:sheet
part:edit
part:audit
--SocialCalcSpreadsheetControlSave
Content-type: text/plain; charset=UTF-8

version:1.5
cell:A1:v:1874
cell:A2:vtf:n:172:2^2*43
cell:A3:vtf:n:2046:SUM(Foo):f:1
sheet:c:1:r:3
font:1:normal bold * *
name:F00::A1\cA2
--SocialCalcSpreadsheetControlSave
Content-type: text/plain; charset=UTF-8

version:1.0
rowpane:0:1:14
colpane:0:1:16
ecell:A1
--SocialCalcSpreadsheetControlSave
Content-type: text/plain; charset=UTF-8

set A1 value n 1874
set A2 formula 2^2*43
name define Foo A1:A2
set A3 formula SUM(Foo)
--SocialCalcSpreadsheetControlSave--
```

This format is designed to be human-readable, as well as being relatively easy to generate programmatically. This makes it possible for Drupal's Sheetnode plugin to use PHP to convert between this format and other popular spreadsheet formats, such as Excel (.xls) and OpenDocument (.ods).

Now that we have a good idea about how the pieces in SocialCalc fit together, let's look at two real-world examples of extending SocialCalc.

19.6 Rich-text Editing

The first example we'll look at is enhancing SocialCalc's text cells with wiki markup to display its rich-text rendering right in the table editor (Figure 19.12).

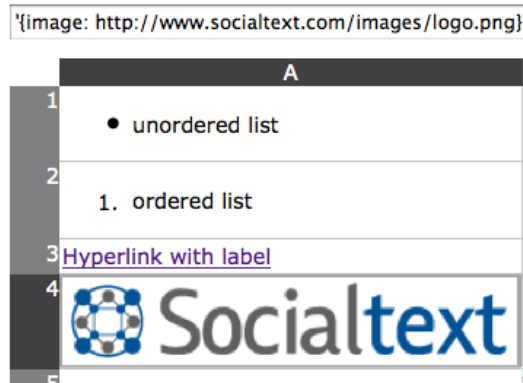


Figure 19.12: Rich Text Rendering in the Table Editor

We added this feature to SocialCalc right after its 1.0 release, to address the popular request of inserting images, links and text markups using a unified syntax. Since Socialtext already has an open-source wiki platform, it was natural to re-use the syntax for SocialCalc as well.

To implement this, we need a custom renderer for the `textvalueformat` of `text-wiki`, and to change the default format for text cells to use it.

What is this `textvalueformat`, you ask? Read on.

Types and Formats

In SocialCalc, each cell has a `datatype` and a `valuetype`. Data cells with text or numbers correspond to `text/numeric` value types, and formula cells with `datatype="f"` may generate either numeric or text values.

Recall that on the Render step, the Sheet object generates HTML from each of its cells. It does so by inspecting each cell's `valuetype`: If it begins with `t`, then the cell's `textvalueformat` attribute determines how generation is done. If it begins with `n`, then the `nontextvalueformat` attribute is used instead.

However, if the cell's `textvalueformat` or `nontextvalueformat` attribute is not defined explicitly, then a default format is looked up from its `valuetype`, as shown in Figure 19.13.

Support for the `text-wiki` value format is coded in `SocialCalc.format_text_for_display`:

```
if (SocialCalc.Callbacks.expand_wiki && /^text-wiki/.test(valueformat)) {  
  // do general wiki markup  
  displayvalue = SocialCalc.Callbacks.expand_wiki(  
    displayvalue, sheetobj, linkstyle, valueformat  
  );  
}
```

Instead of inlining the wiki-to-HTML expander in `format_text_for_display`, we will define a new hook in `SocialCalc.Callbacks`. This is the recommended style throughout the SocialCalc

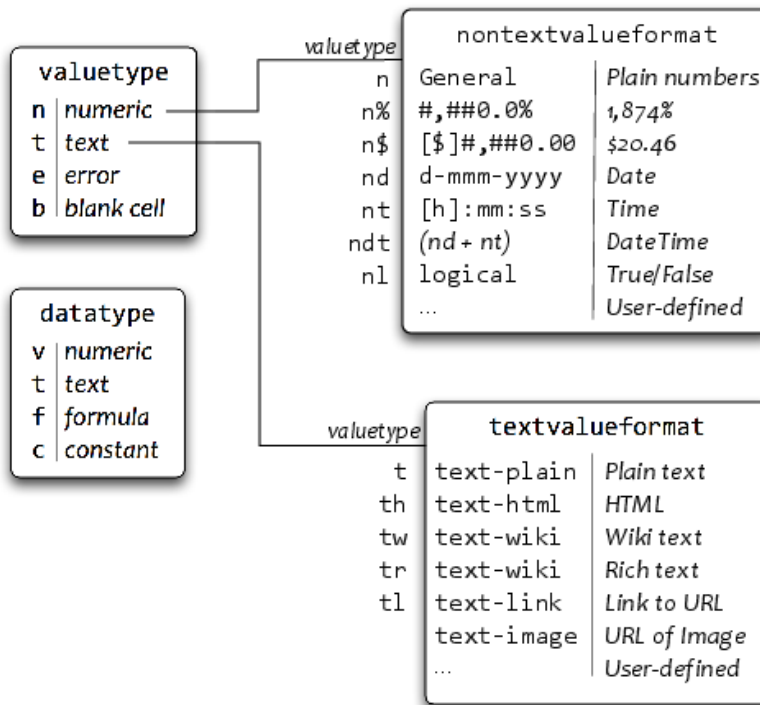


Figure 19.13: Value Types

codebase; it improves modularity by making it possible to plug in different ways of expanding wikitext, as well as keeping compatibility with embedders that do not desire this feature.

Rendering Wikitext

Next, we'll make use of Wikiwyg¹, a Javascript library offering two-way conversions between wikitext and HTML.

We define the `expand_wiki` function by taking the cell's text, running it through Wikiwyg's wikitext parser and its HTML emitter:

```
var parser = new Document.Parser.Wikitext();
var emitter = new Document.Emitter.HTML();
SocialCalc.Callbacks.expand_wiki = function(val) {
    // Convert val from Wikitext to HTML
    return parser.parse(val, emitter);
}
```

¹<https://github.com/audreyt/wikiwyg-js>

The final step involves scheduling the `set sheet defaulttextvalueformat text-wiki` command right after the spreadsheet initializes:

```
// We assume there's a <div id="tableeditor"/> in the DOM already
var spreadsheet = new SocialCalc.SpreadsheetControl();
spreadsheet.InitializeSpreadsheetControl("tableeditor", 0, 0, 0);
spreadsheet.ExecuteCommand('set sheet defaulttextvalueformat text-wiki');
```

Taken together, the Render step now works as shown in Figure 19.14.

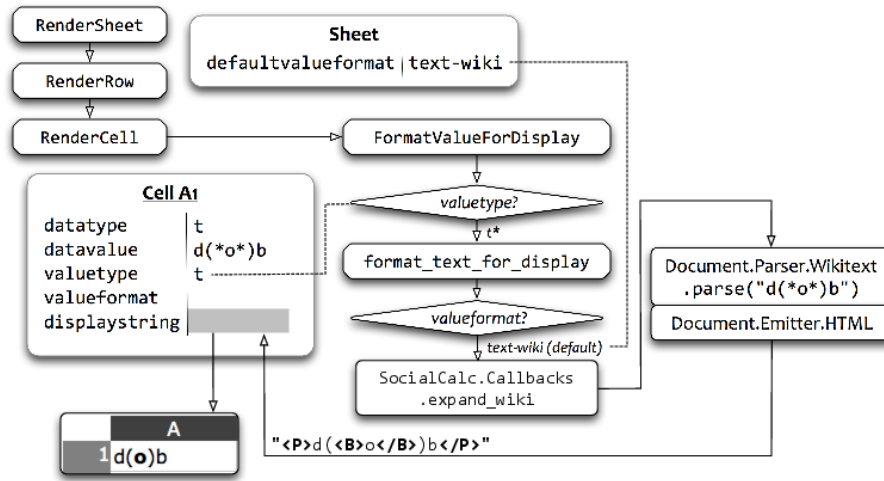


Figure 19.14: Render Step

That's all! The enhanced SocialCalc now supports a rich set of wiki markup syntax:

```
*bold* _italic_ `monospace` {{unformatted}}
> indented text
* unordered list
# ordered list
"Hyperlink with label"<http://softwaregarden.com/>
{image: http://www.socialtext.com/images/logo.png}
```

Try entering `*bold* _italic_ 'monospace'` in A1, and you'll see it rendered as rich text (Figure 19.15).

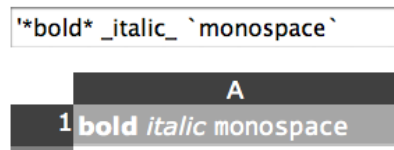


Figure 19.15: Wikywyg Example

19.7 Real-time Collaboration

The next example we'll explore is multi-user, real-time editing on a shared spreadsheet. This may seem complicated at first, but thanks to SocialCalc's modular design all it takes is for each on-line user to broadcast their commands to other participants.

To distinguish between locally-issued commands and remote commands, we add an `isRemote` parameter to the `ScheduleSheetCommands` method:

```
SocialCalc.ScheduleSheetCommands = function(sheet, cmdstr, saveundo, isRemote) {  
    if (SocialCalc.Callbacks.broadcast && !isRemote) {  
        SocialCalc.Callbacks.broadcast('execute', {  
            cmdstr: cmdstr, saveundo: saveundo  
        });  
    }  
    // ...original ScheduleSheetCommands code here...  
}
```

Now all we need to do is to define a suitable `SocialCalc.Callbacks.broadcast` callback function. Once it's in place, the same commands will be executed on all users connected to the same spreadsheet.

When this feature was first implemented for OLPC (One Laptop Per Child²) by SEETA's Sugar Labs³ in 2009, the broadcast function was built with XPCOM calls into D-Bus/Telepathy, the standard transport for OLPC/Sugar networks (see Figure 19.16).

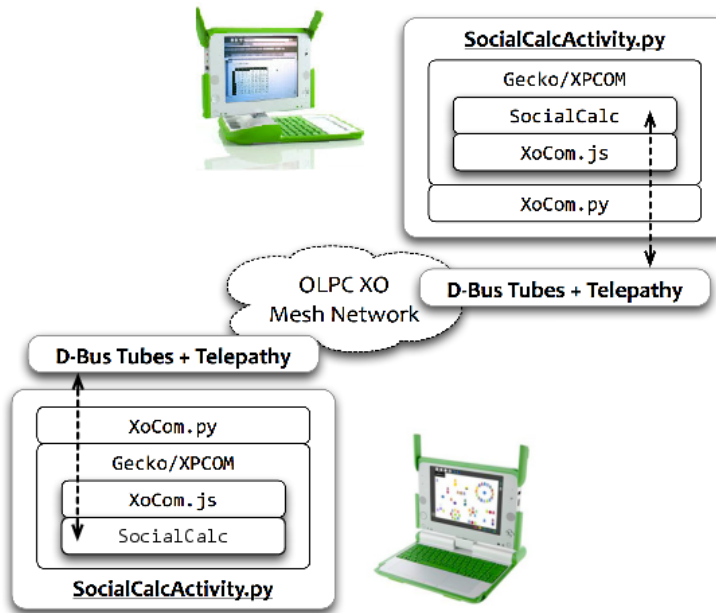


Figure 19.16: OLPC Implementation

²<http://one.laptop.org/>

³http://seeta.in/wiki/index.php?title=Collaboration_in_SocialCalc

That worked reasonably well, enabling XO instances in the same Sugar network to collaborate on a common SocialCalc spreadsheet. However, it is both specific to the Mozilla/XPCOM browser platform, as well as to the D-Bus/Telepathy messaging platform.

Cross-browser Transport

To make this work across browsers and operating systems, we use the `Web::Hippie`⁴ framework, a high-level abstraction of JSON-over-WebSocket with convenient jQuery bindings, with MXHR (Multipart XML HTTP Request⁵) as the fallback transport mechanism if WebSocket is not available.

For browsers with Adobe Flash plugin installed but without native WebSocket support, we use the `web_socket.js` project's Flash emulation of WebSocket, which is often faster and more reliable than MXHR. The operation flow is shown in Figure 19.17.

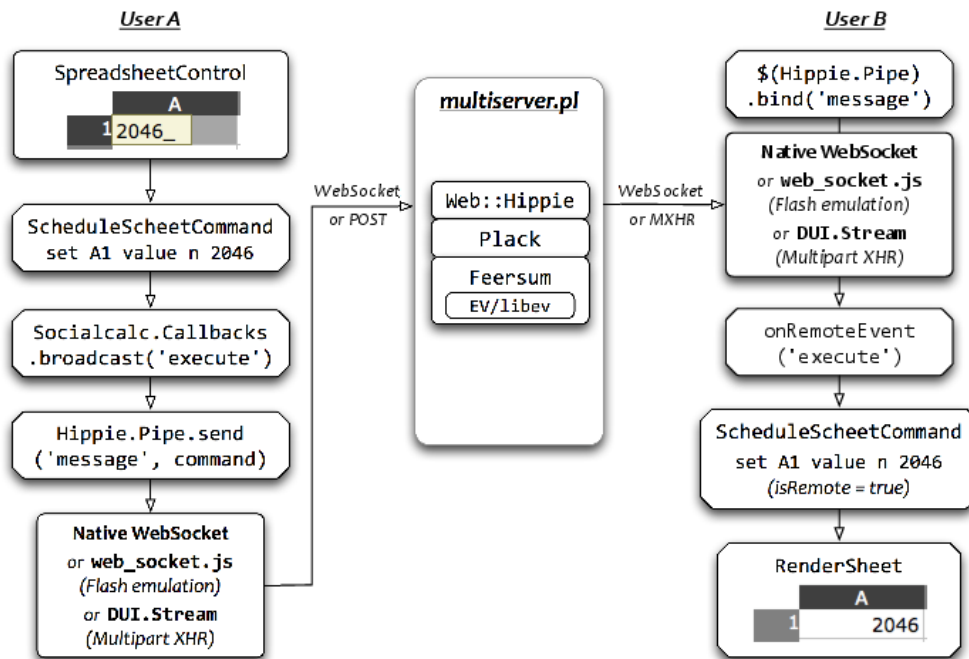


Figure 19.17: Cross-Browser Flow

⁴<http://search.cpan.org/dist/Web-Hippie/>

⁵<http://about.digg.com/blog/duistream-and-mxhr>

⁶<https://github.com/gimite/web-socket-js>

The client-side `SocialCalc.Callbacks.broadcast` function is defined as:

```
var hpipe = new Hippie.Pipe();

SocialCalc.Callbacks.broadcast = function(type, data) {
  hpipe.send({ type: type, data: data });
};

$(hpipe).bind("message.execute", function (e, d) {
  var sheet = SocialCalc.CurrentSpreadsheetControlObject.context.sheetobj;
  sheet.ScheduleSheetCommands(
    d.data.cmdstr, d.data.saveundo, true // isRemote = true
  );
  break;
});
```

Although this works quite well, there are still two remaining issues to resolve.

Conflict Resolution

The first one is a race-condition in the order of commands executed: If users A and B simultaneously perform an operation affecting the same cells, then receive and execute commands broadcast from the other user, they will end up in different states, as shown in Figure 19.18.

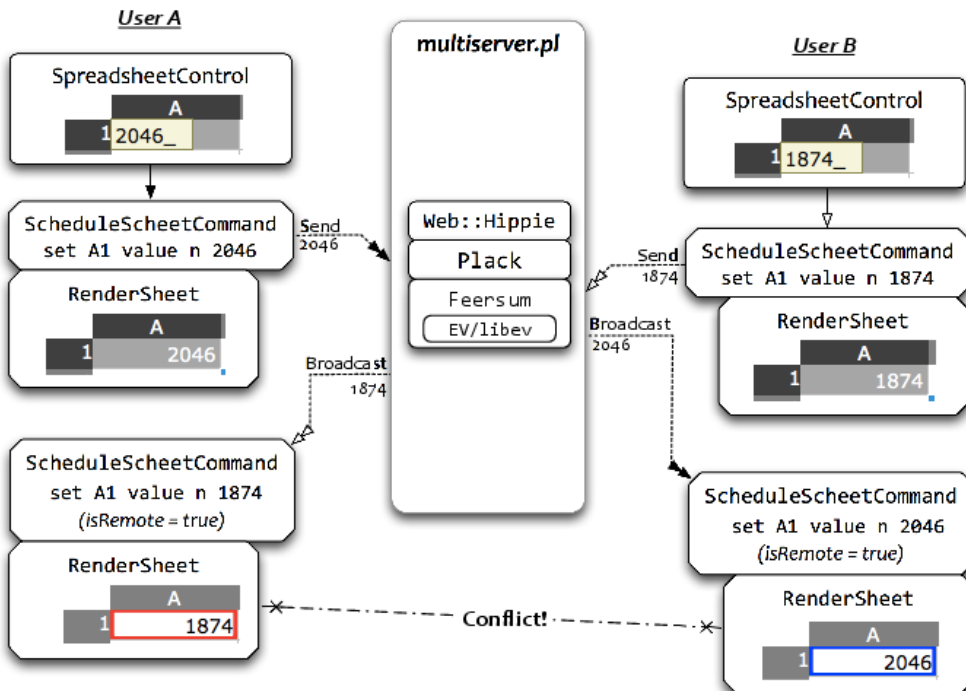


Figure 19.18: Race Condition Conflict

We can resolve this with SocialCalc's built-in undo/redo mechanism, as shown in Figure 19.19.

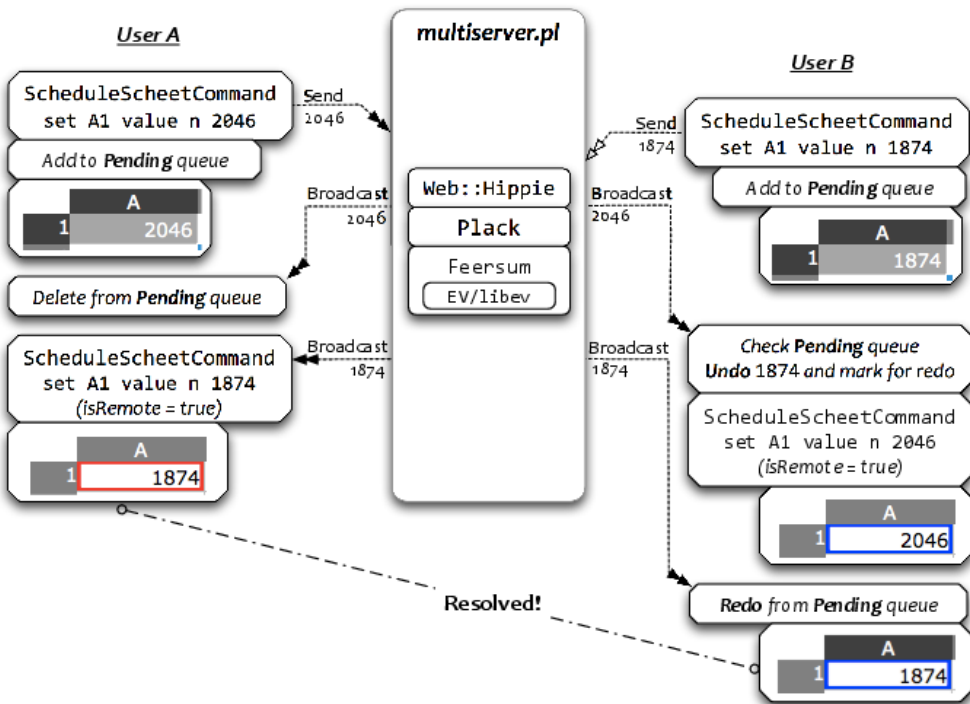


Figure 19.19: Race Condition Conflict Resolution

The process used to resolve the conflict is as follows. When a client broadcasts a command, it adds the command to a Pending queue. When a client receives a command, it checks the remote command against the Pending queue.

If the Pending queue is empty, then the command is simply executed as a remote action. If the remote command matches a command in the Pending queue, then the local command is removed from the queue.

Otherwise, the client checks if there are any queued commands that conflict with the received command. If there are conflicting commands, the client first Undoes those commands and marks them for later Redo. After undoing the conflicting commands (if any), the remote command is executed as usual.

When a marked-for-redo command is received from the server, the client will execute it again, then remove it from the queue.

Remote Cursors

Even with race conditions resolved, it is still suboptimal to accidentally overwrite the cell another user is currently editing. A simple improvement is for each client to broadcast its cursor position to other users, so everyone can see which cells are being worked on.

To implement this idea, we add another broadcast handler to the **MoveCellCallback** event:

```

editor.MoveCellCallback.broadcast = function(e) {
    hpipe.send({
        type: 'ecell',
        data: e.ecell.coord
    });
};

$(hpipe).bind("message.ecell", function (e, d) {
    var cr = SocialCalc.coordToCr(d.data);
    var cell = SocialCalc.GetEditorCellElement(editor, cr.row, cr.col);
    // ...decorate cell with styles specific to the remote user(s) on it...
});

```

To mark cell focus in spreadsheets, it's common to use colored borders. However, a cell may already define its own border property, and since border is mono-colored, it can only represent one cursor on the same cell.

Therefore, on browsers with support for CSS3, we use the box-shadow property to represent multiple peer cursors in the same cell:

```

/* Two cursors on the same cell */
box-shadow: inset 0 0 0 4px red, inset 0 0 0 2px green;

```

Figure 19.20 shows how the screen would look with four people editing on the same spreadsheet.

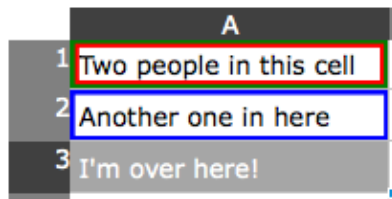


Figure 19.20: Four Users Editing One Spreadsheet

19.8 Lessons Learned

We delivered SocialCalc 1.0 on October 19th, 2009, the 30th anniversary of the initial release of VisiCalc. The experience of collaborating with my colleagues at Socialtext under Dan Bricklin's guidance was very valuable to me, and I'd like to share some lessons I learned during that time.

Chief Designer with a Clear Vision

In [Bro10], Fred Brooks argues that when building complex systems, the conversation is much more direct if we focus on a coherent *design concept*, rather than derivative representations. According to Brooks, the formulation of such a coherent design concept is best kept in a single person's mind:

Since conceptual integrity is the most important attribute of a great design, and since that comes from one or a few minds working *uno animo*, the wise manager boldly entrusts each design task to a gifted chief designer.

In the case of SocialCalc, having Tracy Ruggles as our chief user-experience designer was the key for the project to converge toward a shared vision. Since the underlying SocialCalc engine was so malleable, the temptation of feature creep was very real. Tracy's ability to communicate using design sketches really helped us present features in a way that feels intuitive to users.

Wikis for Project Continuity

Before I joined the SocialCalc project, there was already over two years' worth of ongoing design and development, but I was able to catch up and start contributing in less than a week, simply due to the fact that *everything is in the wiki*. From the earliest design notes to the most up-to-date browser support matrix, the entire process was chronicled in wiki pages and SocialCalc spreadsheets.

Reading through the project's workspace brought me quickly to the same page as others, without the usual hand-holding overhead typically associated with orienting a new team member.

This would not be possible in traditional open source projects, where most conversation takes place on IRC and mailing lists and the wiki (if present) is only used for documentations and links to development resources. For a newcomer, it's much more difficult to reconstruct context from unstructured IRC logs and mail archives.

Embrace Time Zone Differences

David Heinemeier Hansson, creator of Ruby on Rails, once remarked on the benefit of distributed teams when he first joined 37signals. "The seven time zones between Copenhagen and Chicago actually meant that we got a lot done with few interruptions." With nine time zones between Taipei and Palo Alto, that was true for us during SocialCalc's development as well.

We often completed an entire Design-Development-QA feedback cycle within a 24-hour day, with each aspect taking one person's 8-hour work day in their local daytime. This asynchronous style of collaboration compelled us to produce self-descriptive artifacts (design sketch, code and tests), which in turn greatly improved our trust in each other.

Optimize for Fun

In my 2006 keynote for the CONISLI conference [Tan06], I summarized my experience leading a distributed team implementing the Perl 6 language into a few observations. Among them, *Always have a Roadmap*, *Forgiveness > Permission*, *Remove deadlocks*, *Seek ideas, not consensus*, and *Sketch ideas with code* are particularly relevant for small distributed teams.

When developing SocialCalc, we took great care in distributing knowledge among team members with collaborative code ownership, so nobody would become a critical bottleneck.

Furthermore, we pre-emptively resolved disputes by actually coding up alternatives to explore the design space, and were not afraid of replacing fully-working prototypes when a better design arrived.

These cultural traits helped us foster a sense of anticipation and camaraderie despite the absence of face-to-face interaction, kept politics to a minimum, and made working on SocialCalc a lot of fun.

Drive Development with Story Tests

Prior to joining Socialtext, I've advocated the "interleave tests with the specification" approach, as can be seen in the Perl 6 specification⁷, where we annotate the language specification with the official test suite. However, it was Ken Pier and Matt Heusser, the QA team for SocialCalc, who really opened my eyes to how this can be taken to the next level, bringing tests to the place of *executable specification*.

In Chapter 16 of [GR09], Matt explained our story-test driven development process as follows:

The basic unit of work is a "story," which is an extremely lightweight requirements document. A story contains a brief description of a feature along with examples of what needs to happen to consider the story completed; we call these examples "acceptance tests" and describe them in plain English.

During the initial cut of the story, the product owner makes a good-faith first attempt to create acceptance tests, which are augmented by developers and testers before any developer writes a line of code.

These story tests are then translated into wikiprojects, a table-based specification language inspired by Ward Cunningham's FIT framework⁸, which drives automated testing frameworks such as Test::WWW::Mechanize⁹ and Test::WWW::Selenium¹⁰.

It's hard to overstate the benefit of having story tests as a common language to express and validate requirements. It was instrumental in reducing misunderstanding, and has all but eliminated regressions from our monthly releases.

Open Source With CPAL

Last but not least, the open source model we chose for SocialCalc makes an interesting lesson in itself.

Socialtext created the Common Public Attribution License¹¹ for SocialCalc. Based on the Mozilla Public License, CPAL is designed to allow the original author to require an attribution to be displayed on the software's user interface, and has a network-use clause that triggers share-alike provisions when derived work is hosted by a service over the network.

After its approval by both the Open Source Initiative¹² and the Free Software Foundation¹³, we've seen prominent sites such as Facebook¹⁴ and Reddit¹⁵ opting to release their platform's source code under the CPAL, which is very encouraging.

Because CPAL is a "weak copyleft" license, developers can freely combine it with either free or proprietary software, and only need to release modifications to SocialCalc itself. This enabled various communities to adopt SocialCalc and made it more awesome.

There are many interesting possibilities with this open-source spreadsheet engine, and if you can find a way to embed SocialCalc into your favorite project, we'd definitely love to hear about it.

⁷<http://perlcal.org/syn/S02.html>

⁸<http://fit.c2.com/>

⁹<http://search.cpan.org/dist/Test-WWW-Mechanize/>

¹⁰<http://search.cpan.org/dist/Test-WWW-Selenium/>

¹¹<https://www.socialtext.net/open?cpal>

¹²<http://opensource.org/>

¹³<http://www.fsf.org>

¹⁴<https://github.com/facebook/platform>

¹⁵<https://github.com/reddit/reddit>

Telepathy

Danielle Madeley

Telepathy¹ is a modular framework for real-time communications that handles voice, video, text, file transfer, and so on. What's unique about Telepathy is not that it abstracts the details of various instant messaging protocols, but that it provides the idea of communications as a service, in much the same way that printing is a service, available to many applications at once. To achieve this Telepathy makes extensive use of the D-Bus messaging bus and a modular design.

Communications as a service is incredibly useful, because it allows us to break communications out of a single application. This enables lots of interesting use cases: being able to see a contact's presence in your email application; start communicating with her; launching a file transfer to a contact straight from your file browser; or providing contact-to-contact collaboration within applications, known in Telepathy as *Tubes*.

Telepathy was created by Robert McQueen in 2005 and since that time has been developed and maintained by several companies and individual contributors including Collabora, the company co-founded by McQueen.

20.1 Components of the Telepathy Framework

Telepathy is modular, with each module communicating with the others via a D-Bus messaging bus. Most usually via the user's session bus. This communication is detailed in the Telepathy specification². The components of the Telepathy framework are as shown in Figure 20.1:

- A Connection Manager provides the interface between Telepathy and the individual communication services. For instance, there is a Connection Manager for XMPP, one for SIP, one for IRC, and so on. Adding support for a new protocol to Telepathy is simply a matter of writing a new Connection Manager.
- The Account Manager service is responsible for storing the user's communications accounts and establishing a connection to each account via the appropriate Connection Manager when requested.

¹<http://telepathy.freedesktop.org/>, or see the developers' manual at <http://telepathy.freedesktop.org/doc/book/>

²<http://telepathy.freedesktop.org/spec/>

The D-Bus Message Bus

D-Bus is an asynchronous message bus for interprocess communication that forms the backbone of most GNU/Linux systems including the GNOME and KDE desktop environments. D-Bus is primarily a shared bus architecture: applications connect to a bus (identified by a socket address) and can either transmit a targeted message to another application on the bus, or broadcast a signal to all bus members. Applications on the bus have a bus address, similar to an IP address, and can claim a number of well-known names, like DNS names, for example `org.freedesktop.Telepathy.AccountManager`. All processes communicate via the D-Bus daemon, which handles message passing, and name registration.

From the user's perspective, there are two buses available on every system. The system bus is a bus that allows the user to communicate with system-wide components (printers, bluetooth, hardware management, etc.) and is shared by all users on the system. The session bus is unique to that user—i.e., there is a session bus per logged-in user—and is used for the user's applications to communicate with each other. When a lot of traffic is to be transmitted over the bus, it's also possible for applications to create their own private bus, or to create a peer-to-peer, unarbitrated bus with no `dbus-daemon`.

Several libraries implement the D-Bus protocol and can communicate with the D-Bus daemon, including `libdbus`, `GDBus`, `QtDBus`, and `python-dbus`. These libraries are responsible for sending and receiving D-Bus messages, marshalling types from the language's type system into D-Bus' type format and publishing objects on the bus. Usually, the libraries also provide convenience APIs for listing connected applications and activatable applications, and requesting well-known names on the bus. At the D-Bus level, all of these are done by making method calls on an object published by `dbus-daemon` itself.

For more information on D-Bus, see <http://www.freedesktop.org/wiki/Software/dbus>.

- The Channel Dispatcher's role is to listen for incoming channels signalled by each Connection Manager and dispatch them to clients that indicate their ability to handle that type of channel, such as text, voice, video, file transfer, tubes. The Channel Dispatcher also provides a service so that applications, most importantly applications that are not Telepathy clients, can request outgoing channels and have them handled locally by the appropriate client. This allows an application, such as an email application, to request a text chat with a contact, and have your IM client show a chat window.
- Telepathy clients handle or observe communications channels. They include both user interfaces like IM and VoIP clients and services such the chat logger. Clients register themselves with the Channel Dispatcher, giving a list of channel types they wish to handle or observe.

Within the current implementation of Telepathy, the Account Manager and the Channel Dispatcher are both provided by a single process known as Mission Control.

This modular design was based on Doug McIlroy's philosophy, "Write programs that do one thing and do it well," and has several important advantages:

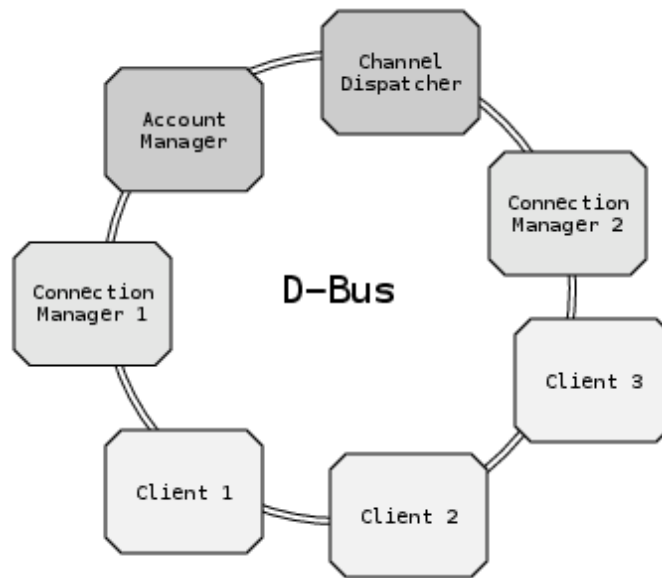


Figure 20.1: Example Telepathy Components

Robustness: a fault in one component won't crash the entire service.

Ease of development: components can be replaced within a running system without affecting others.

It's possible to test a development version of one module against another known to be good.

Language independence: components can be written in any language that has a D-Bus binding.

If the best implementation of a given communications protocol is in a certain language, you are able to write your Connection Manager in that language, and still have it available to all Telepathy clients. Similarly, if you wish to develop your user interface in a certain language, you have access to all available protocols.

License independence: components can be under different software licenses that would be incompatible if everything was running as one process.

Interface independence: multiple user interfaces can be developed on top of the same Telepathy components. This allows native interfaces for desktop environments and hardware devices (e.g., GNOME, KDE, Meego, Sugar).

Security: Components run in separate address spaces and with very limited privileges. For example, a typical Connection Manager only needs access to the network and the D-Bus session bus, making it possible to use something like SELinux to limit what a component can access.

The Connection Manager manages a number of Connections, where each Connection represents a logical connection to a communications service. There is one Connection per configured account. A Connection will contain multiple Channels. Channels are the mechanism through which communications are carried out. A channel might be an IM conversation, voice or video call, file transfer or some other stateful operation. Connections and channels are discussed in detail in Section 20.3.

20.2 How Telepathy uses D-Bus

Telepathy components communicate via a D-Bus messaging bus, which is usually the user's session bus. D-Bus provides features common to many IPC systems: each service publishes objects which have a strictly namespaced object path, like `/org/freedesktop/Telepathy/AccountManager`³. Each object implements a number of interfaces. Again strictly namespaced, these have forms like `org.freedesktop.DBus.Properties` and `ofdT.Connection`. Each interface provides methods, signals and properties that you can call, listen to, or request.

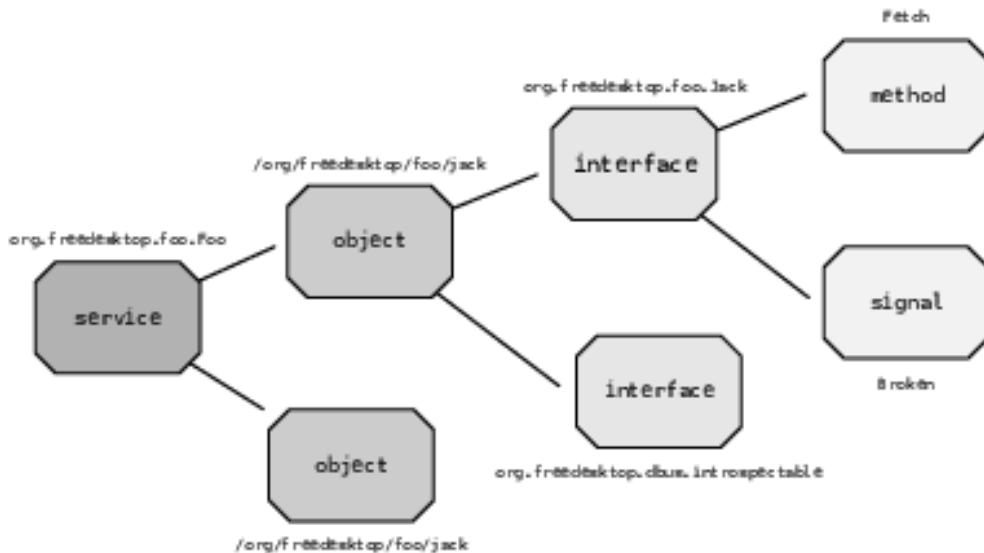


Figure 20.2: Conceptual Representation of Objects Published by a D-Bus Service

Publishing D-Bus Objects

Publishing D-Bus objects is handled entirely by the D-Bus library being used. In effect it is a mapping from a D-Bus object path to the software object implementing those interfaces. The paths of objects being published by a service are exposed by the optional `org.freedesktop.DBus.Introspectable` interface.

When a service receives an incoming method call with a given destination path (e.g., `/ofdT/AccountManager`), the D-Bus library is responsible for locating the software object providing that D-Bus object and then making the appropriate method call on that object.

The interfaces, methods, signal and properties provided by Telepathy are detailed in an XML-based D-Bus IDL that has been expanded to include more information. The specification can be parsed to generate documentation and language bindings.

³From here on, `/org/freedesktop/Telepathy/` and `org.freedesktop.Telepathy` will be abbreviated to `ofdT` to save space.

Telepathy services publish a number of objects onto the bus. Mission Control publishes objects for the Account Manager and Channel Dispatcher so that their services can be accessed. Clients publish a Client object that can be accessed by the Channel Dispatcher. Finally, Connection Managers publish a number of objects: a service object that can be used by the Account Manager to request new connections, an object per open connection, and an object per open channel.

Although D-Bus objects do not have a type (only interfaces), Telepathy simulates types several ways. The object's path tells us whether the object is a connection, channel, client, and so on, though generally you already know this when you request a proxy to it. Each object implements the base interface for that type, e.g., `ofdT.Connection` or `ofdT.Channel`. For channels this is sort of like an abstract base class. Channel objects then have a concrete class defining their channel type. Again, this is represented by a D-Bus interface. The channel type can be learned by reading the `ChannelType` property on the Channel interface.

Finally, each object implements a number of optional interfaces (unsurprisingly also represented as D-Bus interfaces), which depend on the capabilities of the protocol and the Connection Manager. The interfaces available on a given object are available via the `Interfaces` property on the object's base class.

For Connection objects of type `ofdT.Connection`, the optional interfaces have names like `ofdT.Connection.Interface.Avatars` (if the protocol has a concept of avatars), `ofdT.Connection.Interface.ContactList` (if the protocol provides a contact roster—not all do) and `ofdT.Connection.Interface.Location` (if a protocol provides geolocation information). For Channel objects, of type `ofdT.Channel`, the concrete classes have interface names of the form `ofdT.Channel.Type.Text`, `ofdT.Channel.Type.Call` and `ofdT.Channel.Type.FileTransfer`. Like Connections, optional interface have names like `ofdT.Channel.Interface.Messages` (if this channel can send and receive text messages) and `ofdT.Channel.Interface.Group` (if this channel is to a group containing multiple contacts, e.g., a multi-user chat). So, for example, a text channel implements at least the `ofdT.Channel`, `ofdT.Channel.Type.Text` and `Channel.Interface.Messages` interfaces. If it's a multi-user chat, it will also implement `ofdT.Channel.Interface.Group`.

Why an Interfaces Property and not D-Bus Introspection?

You might wonder why each base class implements an `Interfaces` property, instead of relying on D-Bus' introspection capabilities to tell us what interfaces are available. The answer is that different channel and connection objects may offer different interfaces to each other, depending on the capabilities of the channel or connection, but that most of the implementations of D-Bus introspection assume that all objects of the same object class will have the same interfaces. For example, in `telepathy-glib`, the D-Bus interfaces listed by D-Bus introspection are retrieved from the object interfaces a class implements, which is statically defined at compile time. We work around this by having D-Bus introspection provide data for all the interfaces that could exist on an object, and use the `Interfaces` property to indicate which ones actually do.

Although D-Bus itself provides no sanity checking that connection objects only have connection-related interfaces and so forth (since D-Bus has no concept of types, only arbitrarily named interfaces), we can use the information contained within the Telepathy specification to provide sanity checking within the Telepathy language bindings.

Why and How the Specification IDL was Expanded

The existing D-Bus specification IDL defines the names, arguments, access restrictions and D-Bus type signatures of methods, properties and signals. It provides no support for documentation, binding hints or named types.

To resolve these limitations, a new XML namespace was added to provide the required information. This namespace was designed to be generic so that it could be used by other D-Bus APIs. New elements were added to include inline documentation, rationales, introduction and deprecation versions and potential exceptions from methods.

D-Bus type signatures are the low-level type notation of what is serialized over the bus. A D-Bus type signature may look like (ii) (which is a structure containing two int32s), or it may be more complex. For example, a{sa(usuu)}, is a map from string to an array of structures containing uint32, string, uint32, uint32 (Figure 20.3). These types, while descriptive of the data format, provide no semantic meaning to the information contained in the type.

In an effort to provide semantic clarity for programmers and strengthen the typing for language bindings, new elements were added to name simple types, structs, maps, enums, and flags, providing their type signature, as well as documentation. Elements were also added in order to simulate object inheritance for D-Bus objects.

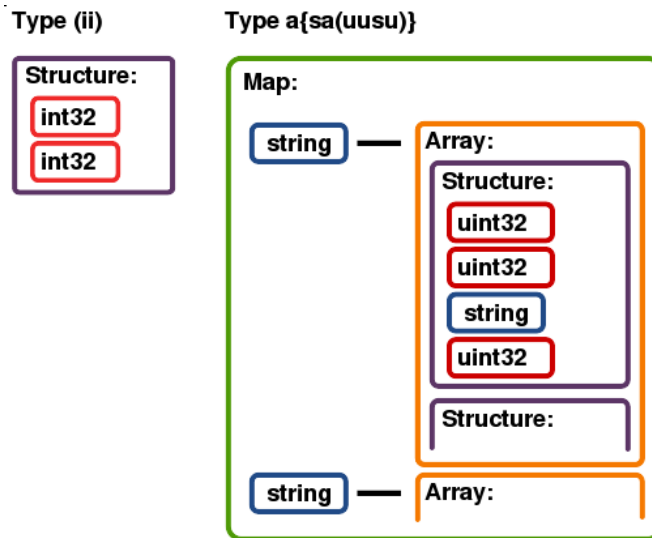


Figure 20.3: D-Bus Types (ii) and a{sa(usuu)}

Handles

Handles are used in Telepathy to represent identifiers (e.g., contacts and room names). They are an unsigned integer value assigned by the connection manager, such that the tuple (connection, handle type, handle) uniquely refers to a given contact or room.

Because different communications protocols normalize identifiers in different ways (e.g., case sensitivity, resources), handles provide a way for clients to determine if two identifiers are the same. They can request the handle for two different identifiers, and if the handle numbers match, then the identifiers refer to the same contact or room.

Identifier normalization rules are different for each protocol, so it is a mistake for clients to compare identifier strings to compare identifiers. For example, `escher@tuxedo.cat/bed` and `escher@tuxedo.cat/litterbox` are two instances of the same contact (`escher@tuxedo.cat`) in the XMPP protocol, and therefore have the same handle. It is possible for clients to request channels by either identifier or handle, but they should only ever use handles for comparison.

Discovering Telepathy Services

Some services, such as the Account Manager and the Channel Dispatcher, which always exist, have well known names that are defined in the Telepathy specification. However, the names of Connection Managers and clients are not well-known, and must be discovered.

There's no service in Telepathy responsible for the registration of running Connection Managers and Clients. Instead, interested parties listen on the D-Bus for the announcement of a new service. The D-Bus bus daemon will emit a signal whenever a new named D-Bus service appears on the bus. The names of Clients and Connection Managers begin with known prefixes, defined by the specification, and new names can be matched against these.

The advantage of this design is that it's completely stateless. When a Telepathy component is starting up, it can ask the bus daemon (which has a canonical list, based on its open connections) what services are currently running. For instance, if the Account Manager crashes, it can look to see what connections are running, and reassociate those with its account objects.

Connections are Services Too

As well as the Connection Managers themselves, the connections are also advertised as D-Bus services. This hypothetically allows for the Connection Manager to fork each connection off as a separate process, but to date no Connection Manager like this has been implemented. More practically, it allows all running connections to be discovered by querying the D-Bus bus daemon for all services beginning with `ofdT.Connection`.

The Channel Dispatcher also uses this method to discover Telepathy clients. These begin with the name `ofdT.Client`, e.g., `ofdT.Client.Logger`.

Reducing D-Bus Traffic

Original versions of the Telepathy specification created an excessive amount of D-Bus traffic in the form of method calls requesting information desired by lots of consumers on the bus. Later versions of the Telepathy have addressed this through a number of optimizations.

Individual method calls were replaced by D-Bus properties. The original specification included separate method calls for object properties: `GetInterfaces`, `GetChannelType`, etc. Requesting all the properties of an object required several method calls, each with its own calling overhead. By using D-Bus properties, everything can be requested at once using the standard `GetAll` method.

Furthermore, quite a number of properties on a channel are immutable for the lifetime of the channel. These include things like the channel's type, interfaces, who it's connected to and the

requestor. For a file transfer channel, for example, it also includes things like the file size and its content type.

A new signal was added to herald the creation of channels (both incoming and in response to outgoing requests) that includes a hash table of the immutable properties. This can be passed directly to the channel proxy constructor (see Section 20.4), which saves interested clients from having to request this information individually.

User avatars are transmitted across the bus as byte arrays. Although Telepathy already used tokens to refer to avatars, allowing clients to know when they needed a new avatar and to save downloading unrequired avatars, each client had to individually request the avatar via a `RequestAvatar` method that returned the avatar as its reply. Thus, when the Connection Manager signalled that a contact had updated its avatar, several individual requests for the avatar would be made, requiring the avatar to be transmitted over the message bus several times.

This was resolved by adding a new method which did not return the avatar (it returns nothing). Instead, it placed the avatar in a request queue. Retrieving the avatar from the network would result in a signal, `AvatarRetrieved`, that all interested clients could listen to. This means the avatar data only needs to be transmitted over the bus once, and will be available to all the interested clients. Once the client's request was in the queue, all further client requests can be ignored until the emission of the `AvatarRetrieved`.

Whenever a large number of contacts need to be loaded (i.e., when loading the contact roster), a significant amount of information needs to be requested: their aliases, avatars, capabilities, and group memberships, and possibly their location, address, and telephone numbers. Previously in Telepathy this would require one method call per information group (most API calls, such as `GetAliases` already took a list of contacts), resulting in half a dozen or more method calls.

To solve this, the `Contacts` interface was introduced. It allowed information from multiple interfaces to be returned via a single method call. The Telepathy specification was expanded to include `Contact Attributes`: namespaced properties returned by the `GetContactAttributes` method that shadowed method calls used to retrieve contact information. A client calls `GetContactAttributes` with a list of contacts and interfaces it is interested in, and gets back a map from contacts to a map of contact attributes to values.

A bit of code will make this clearer. The request looks like this:

```
connection[CONNECTION_INTERFACE_CONTACTS].GetContactAttributes(  
    [ 1, 2, 3 ], # contact handles  
    [ "ofdT.Connection.Interface.Aliasing",  
      "ofdT.Connection.Interface.Avatars",  
      "ofdT.Connection.Interface.ContactGroups",  
      "ofdT.Connection.Interface.Location"  
    ],  
    False # don't hold a reference to these contacts  
)
```

and the reply might look like this:

```
{ 1: { 'ofdT.Connection.Interface.Aliasing/alias': 'Harvey Cat',  
      'ofdT.Connection.Interface.Avatars/token': hex string,  
      'ofdT.Connection.Interface.Location/location': location,  
      'ofdT.Connection.Interface.ContactGroups/groups': [ 'Squid House' ],  
      'ofdT.Connection/contact-id': 'harvey@nom.cat'  
    },  
    ...  
}
```

```

2: { 'ofdT.Connection.Interface.Aliasing/alias': 'Escher Cat',
    'ofdT.Connection.Interface.Avatars/token': hex string,
    'ofdT.Connection.Interface.Location/location': location,
    'ofdT.Connection.Interface.ContactGroups/groups': [],
    'ofdT.Connection/contact-id': 'escher@tuxedo.cat'
},
3: { 'ofdT.Connection.Interface.Aliasing/alias': 'Cami Cat',
    ...
}
}

```

20.3 Connections, Channels and Clients

Connections

A Connection is created by the Connection Manager to establish a connection to a single protocol/account. For example, connecting to the XMPP accounts `escher@tuxedo.cat` and `cami@egg.cat` would result in two Connections, each represented by a D-Bus object. Connections are typically set up by the Account Manager, for the currently enabled accounts.

The Connection provides some mandatory functionality for managing and monitoring the connection status and for requesting channels. It can then also provide a number of optional features, depending on the features of the protocol. These are provided as optional D-Bus interfaces (as discussed in the previous section) and listed by the Connection's Interfaces property.

Typically Connections are managed by the Account Manager, created using the properties of the respective accounts. The Account Manager will also synchronize the user's presence for each account to its respective connection and can be asked to provide the connection path for a given account.

Channels

Channels are the mechanism through which communications are carried out. A channel is typically an IM conversation, voice or video call or file transfer, but channels are also used to provide some stateful communication with the server itself, (e.g., to search for chat rooms or contacts). Each channel is represented by a D-Bus object.

Channels are typically between two or more users, one of whom is yourself. They typically have a target identifier, which is either another contact, in the case of one-to-one communication; or a room identifier, in the case of multi-user communication (e.g., a chat room). Multi-user channels expose the Group interface, which lets you track the contacts who are currently in the channel.

Channels belong to a Connection, and are requested from the Connection Manager, usually via the Channel Dispatcher; or they are created by the Connection in response to a network event (e.g., incoming chat), and handed to the Channel Dispatcher for dispatching.

The type of channel is defined by the channel's ChannelType property. The core features, methods, properties, and signals that are needed for this channel type (e.g., sending and receiving text messages) are defined in the appropriate Channel.Type D-Bus interface, for instance Channel.Type.Text. Some channel types may implement optional additional features (e.g., encryption) which appear as additional interfaces listed by the channel's Interfaces property. An

example text channel that connects the user to a multi-user chatroom might have the interfaces shown in Table 20.1.

odfT.Channel	Features common to all channels
odfT.Channel.Type.Text	The Channel Type, includes features common to text channels
odfT.Channel.Interface.Messages	Rich-text messaging
odfT.Channel.Interface.Group	List, track, invite and approve members in this channel
odfT.Channel.Interface.Room	Read and set properties such as the chatroom's subject

Table 20.1: Example Text Channel

Contact List Channels: A Mistake

In the first versions of the Telepathy specification, contact lists were considered a type of channel. There were several server-defined contact lists (subscribed users, publish-to users, blocked users), that could be requested from each Connection. The members of the list were then discovered using the Group interface, like for a multi-user chat.

Originally this would allow for channel creation to occur only once the contact list had been retrieved, which takes time on some protocols. A client could request the channel whenever it liked, and it would be delivered once ready, but for users with lots of contacts this meant the request would occasionally time out. Determining the subscription/publish/blocked status of a client required checking three channels.

Contact Groups (e.g., Friends) were also exposed as channels, one channel per group. This proved extremely difficult for client developers to work with. Operations like getting the list of groups a contact was in required a significant amount of code in the client. Further, with the information only available via channels, properties such as a contact's groups or subscription state could not be published via the Contacts interface.

Both channel types have since been replaced by interfaces on the Connection itself which expose contact roster information in ways more useful to client authors, including subscription state of a contact (an enum), groups a contact is in, and contacts in a group. A signal indicates when the contact list has been prepared.

Requesting Channels, Channel Properties and Dispatching

Channels are requested using a map of properties you wish the desired channel to possess. Typically, the channel request will include the channel type, target handle type (contact or room) and target. However, a channel request may also include properties such as the filename and filesize for file transfers, whether to initially include audio and video for calls, what existing channels to combine into a conference call, or which contact server to conduct a contact search on.

The properties in the channel request are properties defined by interfaces of the Telepathy spec, such as the ChannelType property (Table 20.2). They are qualified with the namespace of the interface they come from Properties which can be included in channel requests are marked as *requestable* in the Telepathy spec.

Property	Value
ofdT.Channel.ChannelType	ofdT.Channel.Type.Text
ofdT.Channel.TargetHandleType	Handle_Type_Contact (1)
ofdT.Channel.TargetID	escher@tuxedo.cat

Table 20.2: Example Channel Requests

The more complicated example in Table 20.3 requests a file transfer channel. Notice how the requested properties are qualified by the interface from which they come. (For brevity, not all required properties are shown.)

Property	Value
ofdT.Channel.ChannelType	ofdT.Channel.Type.FileTransfer
ofdT.Channel.TargetHandleType	Handle_Type_Contact (1)
ofdT.Channel.TargetID	escher@tuxedo.cat
ofdT.Channel.Type.FileTransfer.Filename	meow.jpg
ofdT.Channel.Type.FileTransfer.ContentType	image/jpeg

Table 20.3: File Transfer Channel Request

Channels can either be *created* or *ensured*. Ensuring a channel means creating it only if it does not already exist. Asking to create a channel will either result in a completely new and separate channel being created, or in an error being generated if multiple copies of such a channel cannot exist. Typically you wish to ensure text channels and calls (i.e., you only need one conversation open with a person, and in fact many protocols do not support multiple separate conversations with the same contact), and wish to create file transfers and stateful channels.

Newly created channels (requested or otherwise) are announced by a signal from the Connection. This signal includes a map of the channel's *immutable* properties. These are the properties which are guaranteed not to change throughout the channel's lifetime. Properties which are considered immutable are marked as such in the Telepathy spec, but typically include the channel's type, target handle type, target, initiator (who created the channel) and interfaces. Properties such as the channel's state are obviously not included.

Old-School Channel Requesting

Channels were originally requested simply by type, handle type and target handle. This wasn't sufficiently flexible because not all channels have a target (e.g., contact search channels), and some channels require additional information included in the initial channel request (e.g., file transfers, requesting voicemails and channels for sending SMSes).

It was also discovered that two different behaviors might be desired when a channel was requested (either to create a guaranteed unique channel, or simply ensure a channel existed), and until this time the Connection had been responsible for deciding which behavior would occur. Hence, the old method was replaced by the newer, more flexible, more explicit ones.

Returning a channel's immutable properties when you create or ensure the channel makes it much faster to create a proxy object for the channel. This is information we now don't have to request. The map in Table 20.4 shows the immutable properties that might be included when we request a text channel (i.e., using the channel request in Table 20.3). Some properties (including TargetHandle and InitiatorHandle) have been excluded for brevity.

Property	Value
ofdT.Channel.ChannelType	Channel.Type.Text
ofdT.Channel.Interfaces	[Channel.Interface.Messages, Channel.Interface.Destroyable, Channel.Interface.ChatState]
ofdT.Channel.TargetHandleType	Handle_Type_Contact (1)
ofdT.Channel.TargetID	escher@tuxedo.cat
ofdT.Channel.InitiatorID	danielle.madeley@collabora.co.uk
ofdT.Channel.Requested	True
ofdT.Channel.Interface.Messages. SupportedContentTypes	[text/html, text/plain]

Table 20.4: Example Immutable Properties Returned by a New Channel

The requesting program typically makes a request for a channel to the Channel Dispatcher, providing the account the request is for, the channel request, and optionally the name of a the desired handler (useful if the program wishes to handle the channel itself). Passing the name of an account instead of a connection means that the Channel Dispatcher can ask the Account Manager to bring an account online if required.

Once the request is complete, the Channel Dispatcher will either pass the channel to the named Handler, or locate an appropriate Handler (see below for discussion on Handlers and other clients). Making the name of the desired Handler optional makes it possible for programs that have no interest in communication channels beyond the initial request to request channels and have them handled by the best program available (e.g., launching a text chat from your email client).

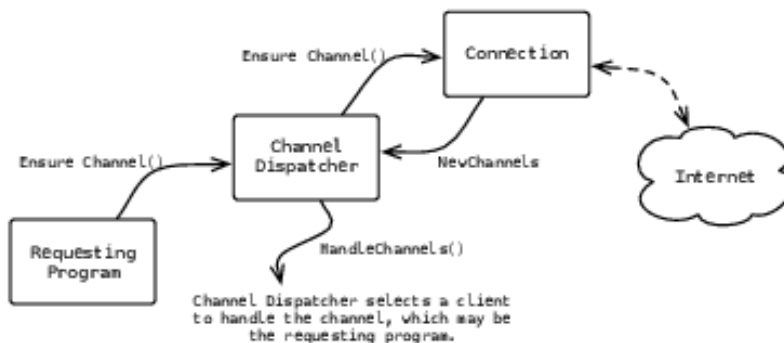


Figure 20.4: Channel Request and Dispatching

The requesting program makes a channel request to the Channel Dispatcher, which in turn forwards the request to the appropriate Connection. The Connection emits the NewChannels signal which is picked up by the Channel Dispatcher, which then finds the appropriate client to handle the

Property	Value
<code>ofdT.Channel.ChannelType</code>	<code>Channel.Type.Text</code>
<code>ofdT.Channel.TargetHandleType</code>	<code>Handle_Type_Contact (1)</code>

Table 20.5: Example Channel Filter

channel. Incoming, unrequested channels are dispatched in much the same way, with a signal from the Connection that is picked up by the Channel Dispatcher, but obviously without the initial request from a program.

Clients

Clients handle or observe incoming and outgoing communications channels. A client is anything that is registered with the Channel Dispatcher. There are three types of clients (though a single client may be two, or all three, types if the developer wishes):

Observers: Observe channels without interacting with them. Observers tend to be used for chat and activity logging (e.g., incoming and outgoing VoIP calls).

Approvers: Responsible for giving users an opportunity to accept or reject an incoming channel.

Handlers: Actually interact with the channel. That might be acknowledging and sending text messages, sending or receiving a file, etc. A Handler tends to be associated with a user interface.

Clients offer D-Bus services with up to three interfaces: `Client.Observer`, `Client.Approver`, and `Client.Handler`. Each interface provides a method that the Channel Dispatcher can call to inform the client about a channel to observe, approve or handle.

The Channel Dispatcher dispatches the channel to each group of clients in turn. First, the channel is dispatched to all appropriate Observers. Once they have all returned, the channel is dispatched to all the appropriate Approvers. Once the first Approver has approved or rejected the channel, all other Approvers are informed and the channel is finally dispatched to the Handler. Channel dispatching is done in stages because Observers might need time to get set up before the Handler begins altering the channel.

Clients expose a channel filter property which is a list of filters read by the Channel Dispatcher so that it knows what sorts of channels a client is interested in. A filter must include at least the channel type, and target handle type (e.g., contact or room) that the client is interested in, but it can contain more properties. Matching is done against the channel's immutable properties, using simple equality for comparison. The filter in Table 20.5 matches all one-to-one text channels.

Clients are discoverable via D-Bus because they publish services beginning with the well-known name `ofdT.Client` (for example `ofdT.Client.Empathy.Chat`). They can also optionally install a file which the Channel Dispatcher will read specifying the channel filters. This allows the Channel Dispatcher to start a client if it is not already running. Having clients be discoverable in this way makes the choice of user interface configurable and changeable at any time without having to replace any other part of Telepathy.

All or Nothing

It is possible to provide a filter indicating you are interested in all channels, but in practice this is only useful as an example of observing channels. Real clients contain code that is specific to channel types.

An empty filter indicates a Handler is not interested in any channel types. However it is still possible to dispatch a channel to this handler if you do so by name. Temporary Handlers which are created on demand to handle a specific channel use such a filter.

20.4 The Role of Language Bindings

As Telepathy is a D-Bus API, and thus can be driven by any programming language that supports D-Bus. Language bindings are not required for Telepathy, but they can be used to provide a convenient way to use it.

Language bindings can be split into two groups: low-level bindings that include code generated from the specification, constants, method names, etc.; and high-level bindings, which are hand-written code that makes it easier for programmers to do things using Telepathy. Examples of high-level bindings are the GLib and Qt4 bindings. Examples of low-level bindings are the Python bindings and the original libtelepathy C bindings, though the GLib and Qt4 bindings include a low-level binding.

Asynchronous Programming

Within the language bindings, all method calls that make requests over D-Bus are asynchronous: the request is made, and the reply is given in a callback. This is required because D-Bus itself is asynchronous.

Like most network and user interface programming, D-Bus requires the use of an event loop to dispatch callbacks for incoming signals and method returns. D-Bus integrates well with the GLib mainloop used by the GTK+ and Qt toolkits.

Some D-Bus language bindings (such as dbus-glib) provide a pseudo-synchronous API, where the main loop is blocked until the method reply is returned. Once upon a time this was exposed via the telepathy-glib API bindings. Unfortunately using pseudo-synchronous API turns out to be fraught with problems, and was eventually removed from telepathy-glib.

Method calls in the first Telepathy bindings, generated in C, simply used typedef callback functions. Your callback function simply had to implement the same type signature.

```
typedef void (*tp_conn_get_self_handle_reply) (  
    DBusGProxy *proxy,  
    guint handle,  
    GError *error,  
    gpointer userdata  
);
```

This idea is simple, and works for C, so was continued into the next generation of bindings.

In recent years, people have developed a way to use scripting languages such as Javascript and Python, as well as a C#-like language called Vala, that use GLib/GObject-based APIs via a tool called GObject-Introspection. Unfortunately, it's extremely difficult to rebind these types of callbacks into other languages, so newer bindings are designed to take advantage of the asynchronous callback features provided by the languages and GLib.

Why Pseudo-Synchronous D-Bus Calls Don't Work

The pseudo-synchronous interface offered by dbus-glib and other D-Bus bindings is implemented using a request-and-block technique. While blocking, only the D-Bus socket is polled for new I/O and any D-Bus messages that are not the response to the request are queued for later processing.

This causes several major and inescapable problems:

- The caller is blocked while waiting for the request to be answered. It (and its user interface, if any) will be completely unresponsive. If the request requires accessing the network, that takes time; if the callee has locked up, the caller will be unresponsive until the call times out.

Threading is not a solution here because threading is just another way of making your calling asynchronous. Instead you may as well make asynchronous calls where the responses come in via the existing event loop.

- Messages may be reordered. Any messages received before the watched-for reply will be placed on a queue and delivered to the client after the reply.

This causes problems in situations where a signal indicating a change of state (i.e., the object has been destroyed) is now received after the method call on that object fails (i.e., with the exception `UnknownMethod`). In this situation, it is hard to know what error to display to the user. Whereas if we receive a signal first, we can cancel pending D-Bus method calls, or ignore their responses.

- Two processes making pseudo-blocking calls on each other can deadlock, with each waiting for the other to respond to its query. This scenario can occur with processes that are both a D-Bus service and call other D-Bus services (for example, Telepathy clients). The Channel Dispatcher calls methods on clients to dispatch channels, but clients also call methods on the Channel Dispatcher to request the opening of new channels (or equally they call the Account Manager, which is part of the same process).

Object Readiness

In a simple D-Bus API, such as the low-level Telepathy bindings, you can start making method calls or receive signals on a D-Bus object simply by creating a proxy object for it. It's as simple as giving an object path and interface name and getting started.

However, in Telepathy's high-level API, we want our object proxies to know what interface are available, we want common properties for the object type to be retrieved (e.g., the channel type, target, initiator), and we want to determine and track the object's state or status (e.g., the connection status).

Thus, the concept of *readiness* exists for all proxy objects. By making a method call on a proxy object, you are able to asynchronously retrieve the state for that object and be notified when state is retrieved and the object is ready for use.

Since not all clients implement, or are interested in, all the features of a given object, readiness for an object type is separated into a number of possible features. Each object implements a *core* feature, which will prepare crucial information about the object (i.e., its Interfaces property and basic

state), plus a number of optional features for additional state, which might include extra properties or state-tracking. Specific examples of additional features you can ready on various proxies are contact info, capabilities, geolocation information, chat states (such as “Escher is typing...”) and user avatars.

For example, connection object proxies have:

- a core feature which retrieves the interface and connection status;
- features to retrieve the requestable channel classes and support contact info; and
- a feature to establish a connection and return ready when connected.

The programmer requests that the object is readied, providing a list of features in which they are interested and a callback to call when all of those features are ready. If all the features are already ready, the callback can be called immediately, else the callback is called once all the information for those features is retrieved.

20.5 Robustness

One of the key advantages of Telepathy is its robustness. The components are modular, so a crash in one component should not bring down the whole system. Here are some of the features that make Telepathy robust:

- The Account Manager and Channel Dispatcher can recover their state. When Mission Control (the single process that includes the Account Manager and Channel Dispatcher) starts, it looks at the names of services currently registered on the user’s session bus. Any Connections it finds that are associated with a known account are reassociated with that account (rather than a new connection being established), and running clients are queried for the list of channels they’re handling.
- If a client disappears while a channel it’s handling is open, the Channel Dispatcher will respawn it and reissue the channel.

If a client repeatedly crashes the Channel Dispatcher can attempt to launch a different client, if available, or else it will close the channel (to prevent the client repeatedly crashing on data it can’t handle).

Text messages require acknowledgment before they will disappear from the list of pending messages. A client is only meant to acknowledge a message once it is sure the user has seen it (that is, displayed the message in a focused window). This way if the client crashes trying to render the message, the channel will still have the previously undisplayed message in the pending message queue.

- If a Connection crashes, the Account Manager will respawn it. Obviously the content of any stateful channels will be lost, but it will only affect the Connections running in that process and no others. Clients can monitor the state of the connections and simply re-request information like the contact roster and any stateless channels.

20.6 Extending Telepathy: Sidecars

Although the Telepathy specification tries to cover a wide range of features exported by communication protocols, some protocols are themselves extensible⁴. Telepathy’s developers wanted to make it

⁴E.g., the Extensible Messaging and Presence Protocol (XMPP).

possible extend your Telepathy connections to make use of such extensions without having to extend the Telepathy specification itself. This is done through the use of *sidecars*.

Sidecars are typically implemented by plugins in a Connection Manager. Clients call a method requesting a sidecar that implements a given D-Bus interface. For example, someone's implementation of XEP-0016 privacy lists might implement an interface named `com.example.PrivacyLists`. The method then returns a D-Bus object provided by the plugin, which should implement that interface (and possibly others). The object exists alongside the main Connection object (hence the name sidecar, like on a motorcycle).

The History of Sidecars

In the early days of Telepathy, the One Laptop Per Child project needed to support custom XMPP extensions (XEPs) to share information between devices. These were added directly to Telepathy-Gabble (the XMPP Connection Manager), and exposed via undocumented interfaces on the Connection object. Eventually, with more developers wanting support for specific XEPs which have no analogue in other communications protocols, it was agreed that a more generic interface for plugins was needed.

20.7 A Brief Look Inside a Connection Manager

Most Connection Managers are written using the C/GLib language binding, and a number of high-level base classes have been developed to make writing a Connection Manager easier. As discussed previously, D-Bus objects are published from software objects that implement a number of software interfaces that map to D-Bus interfaces. Telepathy-GLib provides base objects to implement the Connection Manager, Connection and Channel objects. It also provides an interface to implement a Channel Manager. Channel Managers are factories that can be used by the BaseConnection to instantiate and manage channel objects for publishing on the bus.

The bindings also provide what are known as *mixins*. These can be added to a class to provide additional functionality, abstract the specification API and provide backwards compatibility for new and deprecated versions of an API through one mechanism. The most commonly used mixin is one that adds the D-Bus properties interface to an object. There are also mixins to implement the `ofdT.Connection.Interface.Contacts` and `ofdT.Channel.Interface.Group` interfaces and mixins making it possible to implement the old and new presence interfaces, and old and new text message interfaces via one set of methods.

20.8 Lessons Learned

Telepathy is an excellent example of how to build a modular, flexible API on top of D-Bus. It shows how you can develop an extensible, decoupled framework on top of D-Bus. One which requires no central management daemon and allows components to be restartable, without loss of data in any other component. Telepathy also shows how you can use D-Bus efficiently and effectively, minimizing the amount of traffic you transmit on the bus.

Telepathy's development has been iterative, improving its use of D-Bus as time goes on. Mistakes were made, and lessons have been learned. Here are some of the important things we learned in

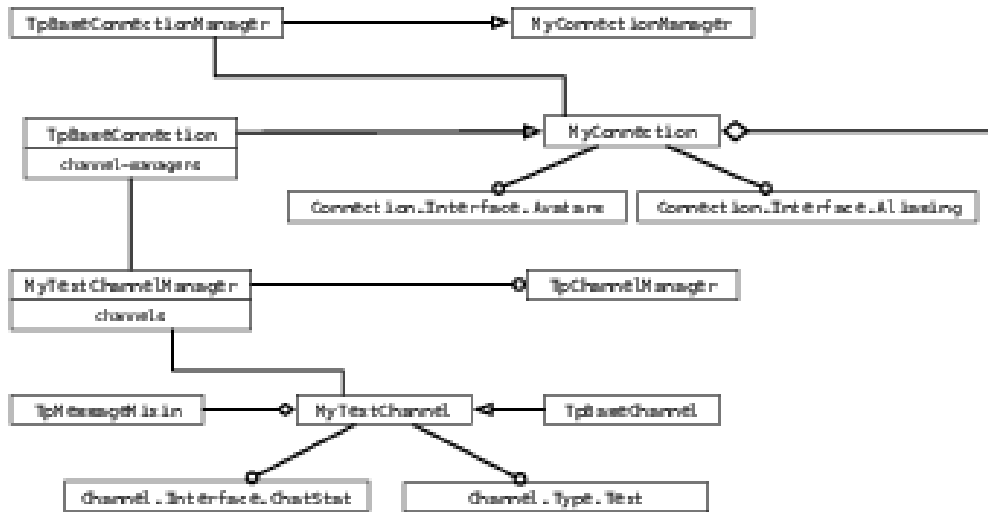


Figure 20.5: Example Connection Manager Architecture

Using Mixins to Solve API Mistakes

One place where mixins have been used to solve a mistake in the Telepathy specification is the `TpPresenceMixin`. The original interface exposed by Telepathy (`odfT.Connection.Interface.Presence`) was incredibly complicated, hard to implement for both Connections and Clients, and exposed functionality that was both nonexistent in most communications protocols, and very rarely used in others. The interface was replaced by a much simpler interface (`odfT.Connection.Interface.SimplePresence`), which exposed all the functionality that users cared about and had ever actually been implemented in the connection managers.

The presence mixin implements both interfaces on the Connection so that legacy clients continue to work, but only at the functionality level of the simpler interface.

designing the architecture of Telepathy:

Use D-Bus properties; don't require dozens of small D-Bus method calls to look up information. Every method call has a round-trip time. Rather than making lots of individual calls (e.g., `GetHandle`, `GetChannelType`, `GetInterfaces`) use D-Bus properties and return all the information via a single call to `GetAll`.

Provide as much information as you can when announcing new objects. The first thing clients used to do when they learned about a new object was to request all of its properties to learn whether they were even interested in the object. By including the immutable properties of an object in the signal announcing the object, most clients can determine their interest in the object without making any method calls. Furthermore, if they are interested in the object, they do not have to bother requesting any of its immutable properties.

The Contacts interface allows requesting information from multiple interfaces at once. Rather than making numerous GetAll calls to retrieve all the information for a contact, the Contacts interface lets us request all the information at once, saving a number of D-Bus round trips.

Don't use abstractions that don't quite fit. Exposing the contact roster and contact groups as channels implementing the Group interface seemed like a good idea because it used existing abstractions rather than requiring additional interfaces. However, it made implementing clients difficult and was ultimately not suitable.

Ensure your API will meet your future needs. The original channel requesting API was very rigid, only permitting very basic channel requests. This did not meet our needs when needing to request channels that required more information. This API had to be replaced with one that had significantly more flexibility.

Thousand Parsec

Alan Laudicina and Aaron Mavrinnac

A vast star empire encompasses a hundred worlds, stretching a thousand parsecs across space. Unlike some other areas of the galaxy, few warriors live here; this is an intellectual people, with a rich cultural and academic tradition. Their magnificent planets, built turn by turn around great universities of science and technology, are a beacon of light to all in this age of peace and prosperity. Starships arrive from the vast reaches of the quadrant and beyond, bearing the foremost researchers from far and wide. They come to contribute their skills to the most ambitious project ever attempted by sentient beings: the development of a decentralized computer network to connect the entire galaxy, with all its various languages, cultures, and systems of law.

Thousand Parsec is more than a video game: it is a framework, with a complete toolkit for building multiplayer, turn-based space empire strategy games. Its generic game protocol allows diverse implementations of client, server, and AI software, as well as a vast array of possible games. Though its size has made planning and execution challenging, forcing contributors to walk a thin line between excessively vertical and excessively horizontal development, it also makes it a rather interesting specimen when discussing the architecture of open source applications.

The journalist's label for the genre Thousand Parsec games inhabit is "4X"—shorthand for "explore, expand, exploit, and exterminate," the modus operandi of the player controlling an empire¹. Typically in the 4X genre of games, players will scout to reveal the map (explore), create new settlements or extend the influence of existing ones (expand), gather and use resources in areas they control (exploit), and attack and eliminate rival players (exterminate). The emphasis on economic and technological development, micromanagement, and variety of routes to supremacy yield a depth and complexity of gameplay unparalleled within the greater strategy genre.

From a player's perspective, three main components are involved in a game of Thousand Parsec. First, there is the client: this is the application through which the player interacts with the universe. This connects to a server over the network—communicating using the all-important protocol—to which other players' (or, in some cases, artificial intelligence) clients are also connected. The server stores the entire game state, updating clients at the start of each turn. Players can then perform various actions and communicate them back to the server, which computes the resulting state for the next turn. The nature of the actions a player may perform is dictated by a ruleset: this in essence

¹Some excellent commercial examples of Thousand Parsec's inspiration include *VGA Planets* and *Stars!*, as well as the *Master of Orion*, *Galactic Civilizations*, and *Space Empires* series. For readers unfamiliar with these titles, the *Civilization* series is a popular example of the same gameplay style, albeit in a different setting. A number of real-time 4X games also exist, such as *Imperium Galactica* and *Sins of a Solar Empire*.

defines the game being played, implemented and enforced on the server side, and actualized for the player by any supporting client.

Because of the diversity of possible games, and the complexity of the architecture required to support this diversity, Thousand Parsec is an exciting project both for gamers and for developers. We hope that even the serious coder with little interest in the anatomy of game frameworks might find value in the underlying mechanics of client-server communication, dynamic configuration, metadata handling, and layered implementation, all of which have grown rather organically toward good design over the years in quintessential open source style.

At its core, Thousand Parsec is primarily a set of standard specifications for a game protocol and other related functionality. This chapter discusses the framework mostly from this abstract viewpoint, but in many cases it is much more enlightening to refer to actual implementations. To this end, the authors have chosen the “flagship” implementations of each major component for concrete discussion.

The case model client is `tpclient-pywx`, a relatively mature wxPython-based client which at present supports the largest set of features and the latest game protocol version. This is supported by `libtpclient-py`, a Python client helper library providing caching and other functionality, and `libtpproto-py`, a Python library which implements the latest version of the Thousand Parsec protocol. For the server, `tpserver-cpp`, the mature C++ implementation supporting the latest features and protocol version, is the specimen. This server sports numerous rulesets, among which the *Missile and Torpedo Wars* milestone ruleset is exemplary for making the most extensive use of features and for being a “traditional” 4X space game.

21.1 Anatomy of a Star Empire

In order to properly introduce the things that make up a Thousand Parsec universe, it makes sense first to give a quick overview of a game. For this, we’ll examine the *Missile and Torpedo Wars* ruleset, the project’s second milestone ruleset, which makes use of most of the major features in the current mainline version of the Thousand Parsec protocol. Some terminology will be used here which will not yet be familiar; the remainder of this section will elucidate it so that the pieces all fall into place.

Missile and Torpedo Wars is an advanced ruleset in that it implements all of the methods available in the Thousand Parsec framework. At the time of writing, it is the only ruleset to do so, and it is being quickly expanded to become a more complete and entertaining game.

Upon establishing a connection to a Thousand Parsec server, the client probes the server for a list of game entities and proceeds to download the entire catalog. This cataloger includes all of the objects, boards, messages, categories, designs, components, properties, players, and resources that make up the state of the game, all of which are covered in detail in this section. While this may seem like a lot for the client to digest at the beginning of the game—and also at the end of each turn—this information is absolutely vital for the game. Once this information has been downloaded, which generally takes on the order of a few seconds, the client now has everything it needs to plot the information onto its representation of the game universe.

When first connected to the server, a random planet is generated and assigned as the new player’s “home planet”, and two fleets are automatically created there. Each fleet consists of two default Scout designs, consisting of a Scout Hull with an Alpha Missile Tube. Since there is no Explosive component added, this default fleet is not yet capable of fleet-to-fleet or fleet-to-planet combat; it is, in fact, a sitting duck.

At this point, it is important for a player to begin equipping fleets with weaponry. This is achieved by creating a weapon design using a Build Weapon order, and then loading the finished product onto the target fleet through a Load Armament order. The Build Weapon order converts a planet's resources—of which each planet has amounts and proportions assigned by a random distribution—into a finished product: an explosive warhead which is planted on the creating planet's surface. The Load Armament order then transfers this completed weapon onto a waiting fleet.

Once the easily accessible surface resources of a planet are used up, it is important to obtain more through mining. Resources come in two other states: mineable and inaccessible. Using a Mine order on a planet, mineable resources may be converted over time into surface resources, which can then be used for building.

Objects

In a Thousand Parsec universe, every physical thing is an object. In fact, the universe itself is also an object. This design allows for a virtually unlimited set of elements in a game, while remaining simple for rulesets which require only a few types of objects. On top of the addition of new object types, each object can store some of its own specific information that can be sent and used via the Thousand Parsec protocol. Five basic built-in object types are currently provided by default: Universe, Galaxy, Star System, Planet, and Fleet.

The Universe is the top-level object in a Thousand Parsec game, and it is always accessible to all players. While the Universe object does not actually exert much control over the game, it does store one vastly important piece of information: the current turn number. Also known as the “year” in Thousand Parsec parlance, the turn number, naturally, increments after the completion of each turn. It is stored in an unsigned 32-bit integer, allowing for games to run until year 4,294,967,295. While not impossible in theory, the authors have not, to date, seen a game progress this far.

A Galaxy is a container for a number of proximate objects—Star Systems, Planets and Fleets—and provides no additional information. A large number of Galaxies may exist in a game, each hosting a subsection of the Universe.

Like the previous two objects, a Star System is primarily a container for lower-level objects. However, the Star System object is the first tier of object which is represented graphically by the client. These objects may contain Planets and Fleets (at least temporarily).

A Planet is a large celestial body which may be inhabited and provide resource mines, production facilities, ground-based armaments, and more. The Planet is the first tier of object which can be owned by a player; ownership of a Planet is an accomplishment not to be taken lightly, and not owning any planets is a typical condition for rulesets to proclaim a player's defeat. The Planet object has a relatively large amount of stored data, accounting for the following:

- The player ID of the Planet's owner (or -1 if not owned by any player).
- A list of the Planet's resources, containing the resource ID (type), and the amount of surface, mineable, and inaccessible resources of this type on the Planet.

The built-in objects described above provide a good basis for many rulesets following the traditional 4X space game formula. Naturally, in keeping with good software engineering principles, object classes can be extended within rulesets. A ruleset designer thus has the ability to create new object types or store additional information in the existing object types as required by the ruleset, allowing for virtually unlimited extensibility in terms of the available physical objects in the game.

Orders

Defined by each ruleset, orders can be attached to both Fleet and Planet objects. While the core server does not ship with any default order types, these are an essential part of even the most basic game. Depending on the nature of the ruleset, orders may be used to accomplish almost any task. In the spirit of the 4X genre, there are a few standard orders which are implemented in most rulesets: these are the Move, Intercept, Build, Colonize, Mine, and Attack orders.

In order to fulfill the first imperative (explore) of 4X, one needs to be able to move about the map of the universe. This is typically achieved via a Move order appended to a Fleet object. In the flexible and extensible spirit of the Thousand Parsec framework, Move orders can be implemented differently depending on the nature of the ruleset. In *Minisec* and *Missile and Torpedo Wars*, a Move order typically takes a point in 3D space as a parameter. On the server side, the estimated time of arrival is calculated and the number of required turns is sent back to the client. The Move order also acts as a pseudo-Attack order in rulesets where teamwork is not implemented. For example, moving to a point occupied by an enemy fleet in both *Minisec* and *Missile and Torpedo Wars* is almost certain to be followed by a period of intense combat. Some rulesets supporting a Move order parameterize it differently (i.e. not using 3D points). For example, the *Risk* ruleset only allows single-turn moves to planets which are directly connected by a “wormhole”.

Typically appended to Fleet objects, the Intercept order allows an object to meet another (commonly an enemy fleet) within space. This order is similar to Move, but since two objects might be moving in different directions during the execution of a turn, it is impossible to land directly on another fleet simply using spatial coordinates, so a distinct order type is necessary. The Intercept order addresses this issue, and can be used to wipe out an enemy fleet in deep space or fend off an oncoming attack in a moment of crisis.

The Build order helps to fulfill two of the 4X imperatives—expand and exploit. The obvious means of expansion throughout the universe is to build many fleets of ships and move them far and wide. The Build order is typically appended to Planet objects and is often bound to the amount of resources that a planet contains—and how they are exploited. If a player is lucky enough to have a home planet rich in resources, that player could gain an early advantage in the game through building.

Like the Build order, the Colonize order helps fulfill the expand and exploit imperatives. Almost always appended to Fleet objects, the Colonize order allows the player to take over an unclaimed planet. This helps to expand control over planets throughout the universe.

The Mine order embodies the exploit imperative. This order, typically appended to Planet objects and other celestial bodies, allows the player to mine for unused resources not immediately available on the surface. Doing so brings these resources to the surface, allowing them to be used subsequently to build and ultimately expand the player’s grip on the universe.

Implemented in some rulesets, the Attack order allows a player to explicitly initiate combat with an enemy Fleet or Planet, fulfilling the final 4X imperative (exterminate). In team-based rulesets, the inclusion of a distinct Attack order (as opposed to simply using Move and Intercept to implicitly attack targets) is important to avoid friendly fire and to coordinate attacks.

Since the Thousand Parsec framework requires ruleset developers to define their own order types, it is possible—even encouraged—for them to think outside the box and create custom orders not found elsewhere. The ability to pack extra data into any object allows developers to do very interesting things with custom order types.

Resources

Resources are extra pieces of data that are packed into Objects in the game. Extensively used—particularly by Planet objects—resources allow for easy extension of rulesets. As with many of the design decisions in Thousand Parsec, extensibility was the driving factor in the inclusion of resources.

While resources are typically implemented by the ruleset designer, there is one resource that is in consistent use throughout the framework: the Home Planet resource, which is used to identify a player's home planet.

According to Thousand Parsec best practices, resources are typically used to represent something that can be converted into some type of object. For example, Minisec implements a Ship Parts resource, which is assigned in random quantities to each planet object in the universe. When one of these planets is colonized, you can then convert this Ship Parts resource into actual Fleets using a Build order.

Missile and Torpedo Wars makes perhaps the most extensive use of resources of any ruleset to date. It is the first ruleset where the weapons are of a dynamic nature, meaning that they can be added to a ship from a planet and also removed from a ship and added back to a planet. To account for this, the game creates a resource type for each weapon that is created in the game. This allows ships to identify a weapon type by a resource, and move them freely throughout the universe. *Missile and Torpedo Wars* also keeps track of factories (the production capability of planets) using a Factories resource tied to each planet.

Designs

In Thousand Parsec, both weapons and ships may be composed of various components. These components are combined to form the basis of a Design—a prototype for something which can be built and used within the game. When creating a ruleset, the designer has to make an almost immediate decision: should the ruleset allow dynamic creation of weapon and ship designs, or simply use a predetermined list of designs? On the one hand, a game using pre-packaged designs will be easier to develop and balance, but on the other hand, dynamic creation of designs adds an entirely new level of complexity, challenge, and fun to the game.

User-created designs allow a game to become far more advanced. Since users must strategically design their own ships and their armaments, a stratum of variance is added to the game which can help to mitigate otherwise great advantages that might be conferred on a player based on luck (e.g., of placement) and other aspects of game strategy. These designs are governed by the rules of each component, outlined in the Thousand Parsec Component Language (TPCL, covered later in this chapter), and specific to each ruleset. The upshot is that no additional programming of functionality is necessary on the part of the developer to implement the design of weapons and ships; configuring some simple rules for each component available in the ruleset is sufficient.

Without careful planning and proper balance, the great advantage of using custom designs can become its downfall. In the later stages of a game, an inordinate amount of time can be spent designing new types of weapons and ships to build. The creation of a good user experience on the client side for design manipulation is also a challenge. Since design manipulation can be an integral part of one game, while completely irrelevant to another, the integration of a design window into clients is a significant obstacle. Thousand Parsec's most complete client, `tpclient-pywx`, currently houses the launcher for this window in a relatively out-of-the-way place, in a sub-menu of the menu bar (which is rarely used in-game otherwise).

The Design functionality is designed to be easily accessible to ruleset developers, while allowing games to expand to virtually unlimited levels of complexity. Many of the existing rulesets allow for only predetermined designs. *Missile and Torpedo Wars*, however, allows for full weapon and ship design from a variety of components.

21.2 The Thousand Parsec Protocol

One might say that the Thousand Parsec protocol is the basis upon which everything else in the project is built. It defines the features available to ruleset writers, how servers should work, and what clients should be able to handle. Most importantly, like an interstellar communications standard, it allows the various software components to understand one another.

The server manages the actual state and dynamics of a game according to the instructions provided by the ruleset. Each turn, a player's client receives some of the information about the state of the game: objects and their ownership and current state, orders in progress, resource stockpiles, technological progress, messages, and everything else visible to that particular player. The player can then perform certain actions given the current state, such as issuing orders or creating designs, and send these back to the server to be processed into the computation of the next turn. All of this communication is framed in the Thousand Parsec protocol. An interesting and quite deliberate effect of this architecture is that AI clients—which are external to the server/ruleset and are the only means of providing computer players in a game—are bound by the same rules as the clients human players use, and thus cannot “cheat” by having unfair access to information or by being able to bend the rules.

The protocol specification describes a series of frames, which are hierarchical in the sense that each frame (except the Header frame) has a base frame type to which it adds its own data. There are a variety of abstract frame types which are never explicitly used, but simply exist to describe bases for concrete frames. Frames may also have a specified direction, with the intent that such frames need only be supported for sending by one side (server or client) and receiving by the other.

The Thousand Parsec protocol is designed to function either standalone over TCP/IP, or tunnelled through another protocol such as HTTP. It also supports SSL encryption.

Basics

The protocol provides a few generic frames which are ubiquitous in communication between client and server. The previously mentioned Header frame simply provides a basis for all other frames via its two direct descendants, the Request and Response frames. The former is the basis for frames which initiate communication (in either direction), and the latter for frames which are prompted by these. The OK and Fail frames (both Response frames) provide the two values for Boolean logic in the exchange. A Sequence frame (also a Response) indicates to the recipient that multiple frames are to follow in response to its request.

Thousand Parsec uses numerical IDs to address things. Accordingly, a vocabulary of frames exists to push around data via these IDs. The Get With ID frame is the basic request for things with such an ID; there is also a Get With ID and Slot frame for things which are in a “slot” on a parent thing which has an ID (e.g., an order on an object). Of course, it is often necessary to obtain sequences of IDs, such as when initially populating the client's state; this is handled using Get ID Sequence type requests and ID Sequence type responses. A common structure for requesting multiple items is a Get ID Sequence request and ID Sequence response, followed by a series of Get With ID requests and appropriate responses describing the item requested.

Players and Games

Before a client can begin interacting with a game, some formalities need to be addressed. The client must first issue a `Connect` frame to the server, to which the server might respond with `OK` or `Fail`—since the `Connect` frame includes the client’s protocol version, one reason for failure might be a version mismatch. The server can also respond with the `Redirect` frame, for moves or server pools. Next, the client must issue a `Login` frame, which identifies and possibly authenticates the player; players new to a server can first use the `Create Account` frame if the server allows it.

Because of the vast variability of Thousand Parsec, the client needs some way to ascertain which protocol features are supported by the server; this is accomplished via the `Get Features` request and `Features` response. Some of the features the server might respond with include:

- Availability of SSL and HTTP tunnelling (on this port or another port).
- Support for server-side component property calculation.
- Ordering of ID sequences in responses (ascending vs. descending).

Similarly, the `Get Games` request and sequence of `Game` responses informs the client about the nature of the active games on the server. A single `Game` frame contains the following information about a game:

- The long (descriptive) name of the game.
- A list of supported protocol versions.
- The type and version of the server.
- The name and version of the ruleset.
- A list of possible network connection configurations.
- A few optional items (number of players, number of objects, administrator details, comment, current turn number, etc.).
- The base URL for media used by the game.

It is, of course, important for a player to know who he or she is up against (or working with, as the case may be), and there is a set of frames for that. The exchange follows the common item sequence pattern with a `Get Player IDs` request, a `List of Player IDs` response, and a series of `Get Player Data` requests and `Player Data` responses. The `Player Data` frame contains the player’s name and race.

Turns in the game are also controlled via the protocol. When a player has finished performing actions, he or she may signal readiness for the next turn via the `Finished Turn` request; the next turn is computed when all players have done so. Turns also have a time limit imposed by the server, so that slow or unresponsive players cannot hold up a game; the client normally issues a `Get Time Remaining` request, and tracks the turn with a local timer set to the value in the server’s `Time Remaining` response.

Finally, Thousand Parsec supports messages for a variety of purposes: game broadcasts to all players, game notifications to a single player, player-to-player communications. These are organized into “board” containers which manage ordering and visibility; following the item sequence pattern, the exchange consists of a `Get Board IDs` request, a `List of Board IDs` response, and a series of `Get Board` requests and `Board` responses.

Once the client has information on a message board, it can issue `Get Message` requests to obtain messages on the board by slot (hence, `Get Message` uses the `Get With ID and Slot` base frame); the server responds with `Message` frames containing the message subject and body, the turn on

which the message was generated, and references to any other entities mentioned in the message. In addition to the normal set of items encountered in Thousand Parsec (players, objects, and the like), there are also some special references including message priority, player actions, and order status. Naturally, the client can also add messages using the Post Message frame—a vehicle for a Message frame—and delete them using the Remove Message frame (based on the GetMessage frame).

Objects, Orders, and Resources

The bulk of the process of interacting with the universe is accomplished through a series of frames comprising the functionality for objects, orders, and resources.

The physical state of the universe—or at least that part of it that the player controls or has the ability to see—must be obtained upon connecting, and every turn thereafter, by the client. The client generally issues a Get Object IDs request (a Get ID Sequence), to which the server replies with a List of Object IDs response. The client can then request details about individual objects using Get Object by ID requests, which are answered with Object frames containing such details—again subject to visibility by the player—as their type, name, size, position, velocity, contained objects, applicable order types, and current orders. The protocol also provides the Get Object IDs by Position request, which allows the client to find all objects within a specified sphere of space.

The client obtains the set of possible orders following the usual item sequence pattern by issuing a Get Order Description IDs request and, for each ID in the List of Order Description IDs response, issuing a Get Order Description request and receiving a Order Description response. The implementation of the orders and order queues themselves has evolved markedly over the history of the protocol. Originally, each object had a single order queue. The client would issue an Order request (containing the order type, target object, and other information), receive an Outcome response detailing the expected result of the order, and, after completion of the order, receive a Result frame containing the actual result.

In the second version, the Order frame incorporated the contents of the Outcome frame (since, based on the order description, this did not require the server's input), and the Result frame was removed entirely. The latest version of the protocol refactored the order queue out of objects, and added the Get Order Queue IDs, List of Order Queue IDs, Get Order Queue, and Order Queue frames, which work similarly to the message and board functionality². The Get Order and Remove Order frames (both GetWithIDSlot requests) allow the client to access and remove orders on a queue, respectively. The Insert Order frame now acts as a vehicle for the Order payload; this was done to allow for another frame, Probe Order, which is used by the client in some cases to obtain information for local use.

Resource descriptions also follow the item sequence pattern: a Get Resource Description IDs request, a List of Resource Description IDs response, and a series of Get Resource Description requests and Resource Description responses.

Design Manipulation

The handling of designs in the Thousand Parsec Protocol is broken down into the manipulation of four separate sub-categories: categories, components, properties, and designs.

Categories differentiate the different design types. Two of the most commonly used design types are ships and weapons. Creating a category is simple, as it consists only of a name and description;

²Actually, it's the other way around: messages and boards were derived from orders in the second version of the protocol.

the Category frame itself contains only these two strings. Each category is added by the ruleset to the Design Store using an `Add Category` request, a vehicle for the Category frame. The remainder of the management of categories is handled in the usual item sequence pattern with the `Get Category IDs` request and `List of Category IDs` response.

Components consist of the different parts and modules which comprise a design. This can be anything from the hull of a ship or missile to the tube that a missile is housed in. Components are a bit more involved than categories. A Component frame contains the following information:

- The name and description of the component.
- A list of categories to which the component belongs.
- A Requirements function, in Thousand Parsec Component Language (TPCL).
- A list of properties and their corresponding values.

Of particular note is the Requirements function associated with the component. Since components are the parts that make up a ship, weapon, or other constructed object, it is necessary to ensure that they are valid when adding them to a design. The Requirements function verifies that each component added to the design conforms to the rules of other previously added components. For example, in *Missile and Torpedo Wars*, it is impossible to hold an Alpha Missile in a ship without an Alpha Missile Tube. This verification occurs on both the client side and the server side, which is why the entire function must appear in a protocol frame, and why a concise language (TPCL, covered later in the chapter) was chosen for it.

All of a design's properties are communicated via Property frames. Each ruleset exposes a set of properties used within the game. These typically include things like the number of missile tubes of a certain type allowed on a ship, or the amount of armor included with a certain hull type. Like Component frames, Property frames make use of TPCL. A Property frame contains the following information:

- The (display) name and description of the property.
- A list of categories to which the property belongs.
- The name (valid TPCL identifier) of the property.
- The rank of the property.
- Calculate and Requirements functions, in Thousand Parsec Component Language (TPCL).

The rank of a property is used to distinguish a hierarchy of dependencies. In TPCL, a function may not depend on any property which has a rank less than or equal to this property. This means that if one had an Armor property of rank 1 and an Invisibility property of rank 0, then the Invisibility property could not directly depend on the Armor property. This ranking was implemented as a method of curtailing circular dependencies. The Calculate function is used to define how a property is displayed, differentiating the methods of measurement. *Missile and Torpedo Wars* uses XML to import game properties from a game data file. Figure 21.1 shows an example property from that game data.

In this example, we have a property belonging to the Ships category, of rank 0. This property is called Colonise, and relates to the ability of a ship to colonize planets. A quick look at the TPCL Calculate function (listed here as `tpclDisplayFunction`) reveals that this property outputs either "Yes" or "No" depending on whether the ship in question has said capability. Adding properties in this fashion gives the ruleset designer granular control over metrics of the game and the ability to easily compare them and output them in a player-friendly format.

```

<prop>
<CategoryIDName>Ships</CategoryIDName>
<rank value="0"/>
<name>Colonise</name>
<displayName>Can Colonise Planets</displayName>
<description>Can the ship colonise planets</description>
<tpclDisplayFunction>
  (lambda (design bits) (let ((n (apply + bits))) (cons n (if (= n 1) "Yes" "No")) ) )
</tpclDisplayFunction>
<tpclRequirementsFunction>
  (lambda (design) (cons #t ""))
</tpclRequirementsFunction>
</prop>

```

Figure 21.1: Example Property

The actual design of ships, weapons, and other game artifacts are created and manipulated using the Design frame and related frames. In all current rulesets, these are used for building ships and weaponry using the existing pool of components and properties. Since the rules for designs are already handled in TPCL Requirements functions in both properties and components, the creation of a design is a bit simpler. A Design frame contains the following information:

- The name and description of the design.
- A list of categories to which the design belongs.
- A count of the number of instances of the design.
- The owner of the design.
- A list of component IDs and their corresponding counts.
- A list of properties and their corresponding display string.
- The feedback on the design.

This frame is a bit different from the others. Most notably, since a design is an owned item in the game, there is a relation to the owner of each design. A design also tracks the number of its instantiations with a counter.

Server Administration

A server administration protocol extension is also available, allowing for remote live control of supporting servers. The standard use case is to connect to the server via an administration client—perhaps a shell-like command interface or a GUI configuration panel—to change settings or perform other maintenance tasks. However, other, more specialized uses are possible, such as behind-the-scenes management for single-player games.

As with the game protocol described in the preceding sections, the administration client first negotiates a connection (on a port separate from the normal game port) and authenticates using Connect and Login requests. Once connected, the client can receive log messages from and issue commands to the server.

Log messages are pushed to the client via Log Message frames. These contain a severity level and text; as appropriate to the context, the client can choose to display all, some, or none of the log messages it receives.

The server may also issue a Command Update frame instructing the client to populate or update its local command set; supported commands are exposed to the client in the server's response to a Get Command Description IDs frame. Individual command descriptions must then be obtained by issuing a Get Command Description frame for each, to which the server responds with a Command Description frame.

This exchange is functionally quite similar to (and, in fact, was originally based on) that of the order frames used in the main game protocol. It allows commands to be described to the user and vetted locally to some degree, minimizing network usage. The administration protocol was conceived at a time when the game protocol was already mature; rather than starting from scratch, the developers found existing functionality in the game protocol which did almost what was needed, and added the code to the same protocol libraries.

21.3 Supporting Functionality

Server Persistence

Thousand Parsec games, like many in the turn-based strategy genre, have the potential to last for quite some time. Besides often running far longer than the circadian rhythms of the players' species, during this extended period the server process might be prematurely terminated for any number of reasons. To allow players to pick up a game where they left off, Thousand Parsec servers provide persistence by storing the entire state of the universe (or even multiple universes) in a database. This functionality is also used in a related way for saving single-player games, which will be covered in more detail later in this section.

The flagship server, `tpserver-cpp`, provides an abstract persistence interface and a modular plugin system to allow for various database back ends. At the time of writing, `tpserver-cpp` ships with modules for MySQL and SQLite.

The abstract Persistence class describes the functionality allowing the server to save, update, and retrieve the various elements of a game (as described in the Anatomy of a Star Empire section). The database is updated continuously from various places in the server code where the game state changes, and no matter the point at which the server is terminated or crashes, all information to that point should be recovered when the server starts again from the saved data.

Thousand Parsec Component Language

The Thousand Parsec Component Language (TPCL) exists to allow clients to create designs locally without server interaction—allowing for instant feedback about the properties, makeup, and validity of the designs. This allows the player to interactively create, for example, new classes of starship, by customizing structure, propulsion, instrumentation, defenses, armaments, and more according to available technology.

TPCL is a subset of Scheme, with a few minor changes, though close enough to the Scheme R5RS standard that any compatible interpreter can be used. Scheme was originally chosen because of its simplicity, a host of precedents for using it as an embedded language, the availability of interpreters implemented in many other languages, and, most importantly to an open source project, vast documentation both on using it and on developing interpreters for it.

Consider the following example of a Requirements function in TPCL, used by components and properties, which would be included with a ruleset on the server side and communicated to the client over the game protocol:

```

(lambda (design)
  (if (> (designType.MaxSize design) (designType.Size design))
      (if (= (designType.num-hulls design) 1)
          (cons #t "")
          (cons #f "Ship can only have one hull"))
      )
    (cons #f "This many components can't fit into this Hull"))
  )
)

```

Readers familiar with Scheme will no doubt find this code easy to understand. The game (both client and server) uses it to check other component properties (MaxSize, Size, and Num-Hulls) to verify that this component can be added to a design. It first verifies that the Size of the component is within the maximum size of the design, then ensures that there are no other hulls in the design (the latter test tips us off that this is the Requirements function from a ship hull).

BattleXML

In war, every battle counts, from the short skirmish in deep space between squadrons of small lightly-armed scout craft, to the massive final clash of two flagship fleets in the sky above a capital world. On the Thousand Parsec framework, the details of combat are handled within the ruleset, and there is no explicit client-side functionality regarding combat details—typically, the player will be informed of the initiation and results of combat via messages, and the appropriate changes to the objects will take place (e.g., removal of destroyed ships). Though the player’s focus will normally be on a higher level, under rulesets with complex combat mechanics, it may prove advantageous (or, at least, entertaining) to examine the battle in more detail.

This is where BattleXML comes in. Battle data is split into two major parts: the media definition, which provides details about the graphics to be used, and the battle definition, which specifies what actually occurred during a battle. These are intended to be read by a battle viewer, of which Thousand Parsec currently has two: one in 2D and the other in 3D. Of course, since the nature of battles are entirely a feature of a ruleset, the ruleset code is responsible for actually producing BattleXML data.

The media definition is tied to the nature of the viewer, and is stored in a directory or an archive containing the XML data and any graphics or model files it references. The data itself describes what media should be used for each ship (or other object) type, its animations for actions such as firing and death, and the media and details of its weapons. File locations are assumed to be relative to the XML file itself, and cannot reference parent directories.

The battle definition is independent of the viewer and media. First, it describes a series of entities on each side at the start of the battle, with unique identifiers and information such as name, description, and type. Then, each round of the battle is described: object movement, weapons fire (with source and target), damage to objects, death of objects, and a log message. How much detail is used to describe each round of battle is dictated by the ruleset.

Metaserver

Finding a public Thousand Parsec server to play on is much like locating a lone stealth scout in deep space—a daunting prospect if one doesn’t know where to look. Fortunately, public servers can announce themselves to a metaserver, whose location, as a central hub, should ideally be well-known to players.

The current implementation is `metaserver-lite`, a PHP script, which lives at some central place like the Thousand Parsec website. Supporting servers send an HTTP request specifying the update action and containing the type, location (protocol, host, and port), ruleset, number of players, object count, administrator, and other optional information. Server listings expire after a specified timeout (by default, 10 minutes), so servers are expected to update the metaserver periodically.

The script can then, when called with no specified action, be used to embed the list of servers with details into a web site, presenting clickable URLs (typically with the `tp://` scheme name). Alternatively, the badge action presents server listings in a compact “badge” format.

Clients may issue a request to a metaserver using the `get` action to obtain a list of available servers. In this case, the metaserver returns one or more Game frames for each server in the list to the client. In `tpclient-pywx`, the resulting list is presented through a server browser in the initial connection window.

Single-Player Mode

Thousand Parsec is designed from the ground up to support networked multiplayer games. However, there is nothing preventing a player from firing up a local server, connecting a few AI clients, and hyperjumping into a custom single-player universe ready to be conquered. The project defines some standard metadata and functionality to support streamlining this process, making setup as easy as running a GUI wizard or double-clicking a scenario file.

At the core of this functionality is an XML DTD specifying the format for metadata regarding the capabilities and properties of each component (e.g., server, AI client, ruleset). Component packages ship with one or more such XML files, and eventually all of this metadata is aggregated into an associative array divided into two major portions: servers and AI clients. Within a server’s metadata will typically be found metadata for one or more rulesets—they are found here because even though a ruleset may be implemented for more than one server, some configuration details may differ, so separate metadata is needed in general for each implementation. Each entry for one of these components contains the following information:

- Descriptive data, including a short (binary) name, a long (descriptive) name, and a description.
- The installed version of the component, and the earliest version whose save data is compatible with the installed version.
- The command string (if applicable) and any forced parameters passed to it.
- A set of parameters which can be specified by the player.

Forced parameters are not player-configurable and are typically options which allow the components to function appropriately for a local, single-player context. The player parameters have their own format indicating such details as the name and description, the data type, default, and range of the value, and the format string to append to the main command string.

While specialized cases are possible (e.g., preset game configurations for ruleset-specific clients), the typical process for constructing a single-player game involves selecting a set of compatible components. Selection of the client is implicit, as the player will have already launched one in order to play a game; a well-designed client follows a user-centric workflow to set up the remainder. The next natural choice to make is the ruleset, so the player is presented with a list—at this point, there is no need to bother with server details. In the event that the chosen ruleset is implemented by multiple installed servers (probably a rare condition), the player is prompted to select one; otherwise, the appropriate server is selected automatically. Next, the player is prompted to configure options for the

ruleset and server, with sane defaults pulled from the metadata. Finally, if any compatible AI clients are installed, the player is prompted to configure one or more of them to play against.

With the game so configured, the client launches the local server with appropriate configuration parameters (including the ruleset, its parameters, and any parameters it adds to the server's configuration), using the command string information from the metadata. Once it has verified that the server is running and accepting connections, perhaps using the administration protocol extension discussed previously, it launches each of the specified AI clients similarly, and verifies that they have successfully connected to the game. If all goes well, the client will then connect to the server—just as if it were connecting to an online game—and the player can begin exploring, trading, conquering, and any of a universe of other possibilities.

An alternate—and very important—use for the single-player functionality is the saving and loading of games, and, more or less equivalently, the loading of ready-to-play scenarios. In this case, the save data (probably, though not necessarily, a single file) stores the single-player game configuration data alongside the persistence data for the game itself. Provided all appropriate components in compatible versions are installed on the player's system, launching a saved game or scenario is completely automatic. Scenarios in particular thus provide an attractive one-click entry into a game. Although Thousand Parsec does not currently have a dedicated scenario editor or a client with an edit mode, the concept is to provide some means of crafting the persistence data outside of the normal functioning of the ruleset, and verifying its consistency and compatibility.

So far, the description of this functionality has been rather abstract. On a more concrete level, the Python client helper library, `libtpclient-py`, is currently home to the only full realization of single-player mechanics in the Thousand Parsec project. The library provides the `SinglePlayerGame` class, which upon instantiation automatically aggregates all available single-player metadata on the system (naturally, there are certain guidelines as to where the XML files should be installed on a given platform). The object can then be queried by the client for various information on the available components; servers, rulesets, AI clients, and parameters are stored as dictionaries (Python's associative arrays). Following the general game building process outlined above, a typical client might perform the following:

1. Query a list of available rulesets via `SinglePlayerGame.rulesets`, and configure the object with the chosen ruleset by setting `SinglePlayerGame.rname`.
2. Query a list of servers implementing the ruleset via `SinglePlayerGame.list_servers_with_ruleset`, prompt the user to select one if necessary, and configure the object with the chosen (or only) server by setting `SinglePlayerGame.sname`.
3. Obtain the set of parameters for the server and ruleset via `SinglePlayerGame.list_rparams` and `SinglePlayerGame.list_sparams`, respectively, and prompt the player to configure them.
4. Find available AI clients supporting the ruleset via `SinglePlayerGame.list_aiclients_with_ruleset`, and prompt the player to configure one or more of them using the parameters obtained via `SinglePlayerGame.list_aiparams`.
5. Launch the game by calling `SinglePlayerGame.start`, which will return a TCP/IP port to connect on if successful.
6. Eventually, end the game (and kill any launched server and AI client processes) by calling `SinglePlayerGame.stop`.

Thousand Parsec's flagship client, `tpclient-pywx`, presents a user-friendly wizard which follows such a procedure, initially prompting instead for a saved game or scenario file to load. The user-centric workflow developed for this wizard is an example of good design arising from the open

source development process of the project: the developer initially proposed a very different process more closely aligned with how things were working under the hood, but community discussion and some collaborative development produced a result much more usable for the player.

Finally, saved games and scenarios are currently implemented in practice in `tpserver-cpp`, with supporting functionality in `libtpclient-py` and an interface in `tpclient-pywx`. This is achieved through a persistence module using SQLite, a public domain open source RDBMS which requires no external process and stores databases in a single file. The server is configured, via a forced parameter, to use the SQLite persistence module if it is available, and as usual, the database file (living in a temporary location) is constantly updated throughout the game. When the player opts to save the game, the database file is copied to the specified location, and a special table is added to it containing the single player configuration data. It should be fairly obvious to the reader how this is subsequently loaded.

21.4 Lessons Learned

The creation and growth of the extensive Thousand Parsec framework has allowed the developers plenty of opportunity to look back and assess the design decisions that were made along the way. The original core developers (Tim Ansell and Lee Begg) built the original framework from scratch and have shared with us some suggestions on starting a similar project.

What Worked

A major key to the development of Thousand Parsec was the decision to define and build a subset of the framework, followed by the implementation. This iterative and incremental design process allowed the framework to grow organically, with new features added seamlessly. This led directly to the decision to version the Thousand Parsec protocol, which is credited with a number of major successes of the framework. Versioning the protocol allowed the framework to grow over time, enabling new methods of gameplay along the way.

When developing such an expansive framework, it is important to have a very short-term approach for goals and iterations. Short iterations, on the order of weeks for a minor release, allowed the project to move forward quickly with immediate returns along the way. Another success of the implementation was the client-server model, which allowed for the clients to be developed away from any game logic. The separation of game logic from client software was important to the overall success of Thousand Parsec.

What Didn't Work

A major downfall of the Thousand Parsec framework was the decision to use a binary protocol. As you can imagine, debugging a binary protocol is not a fun task and this has led to many prolonged debugging sessions. We would highly recommend that nobody take this path in the future. The protocol has also grown to have too much flexibility; when creating a protocol, it is important to implement only the basic features that are required.

Our iterations have at times grown too large. When managing such a large framework on an open source development schedule, it is important to have a small subset of added features in each iteration to keep development flowing.

Conclusion

Like a construction skiff inspecting the skeletal hull of a massive prototype battleship in an orbital construction yard, we have passed over the various details of the architecture of Thousand Parsec. While the general design criteria of flexibility and extensibility have been in the minds of the developers from the very beginning, it is evident to us, looking at the history of the framework, that only an open source ecosystem, teeming with fresh ideas and points of view, could have produced the sheer volume of possibilities while remaining functional and cohesive. It is a singularly ambitious project, and as with many of its peers on the open source landscape, much remains to be done; it is our hope and expectation that over time, Thousand Parsec will continue to evolve and expand its capabilities while new and ever more complex games are developed upon it. After all, a journey of a thousand parsecs begins with a single step.

Violet

Cay Horstmann

In 2002, I wrote an undergraduate textbook on object-oriented design and patterns [Hor05]. As with so many books, this one was motivated by frustration with the canonical curriculum. Frequently, computer science students learn how to design a single class in their first programming course, and then have no further training in object-oriented design until their senior level software engineering course. In that course, students rush through a couple of weeks of UML and design patterns, which gives no more than an illusion of knowledge. My book supports a semester-long course for students with a background in Java programming and basic data structures (typically from a Java-based CS1/CS2 sequence). The book covers object-oriented design principles and design patterns in the context of familiar situations. For example, the Decorator design pattern is introduced with a Swing JScrollPane, in the hope that this example is more memorable than the canonical Java streams example.

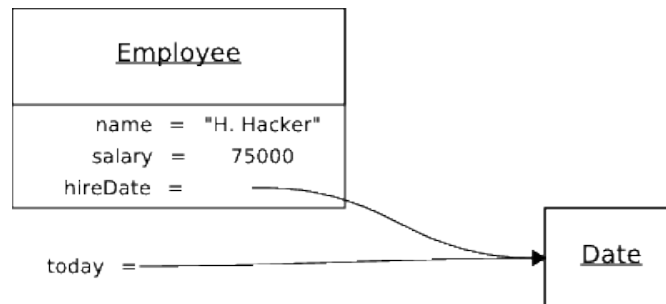


Figure 22.1: A Violet Object Diagram

I needed a light subset of UML for the book: class diagrams, sequence diagrams, and a variant of object diagrams that shows Java object references (Figure 22.1). I also wanted students to draw their own diagrams. However, commercial offerings such as Rational Rose were not only expensive but also cumbersome to learn and use [Shu05], and the open source alternatives available at the time were too limited or buggy to be useful¹. In particular, sequence diagrams in ArgoUML were

¹At the time, I was not aware of Diomidis Spinellis' admirable UMLGraph program [Spi03], in which diagrams are specified by textual declarations rather than the more common point-and-click interface.

seriously broken.

I decided to try my hand at implementing the simplest editor that is (a) useful to students and (b) an example of an extensible framework that students can understand and modify. Thus, Violet was born.

22.1 Introducing Violet

Violet is a lightweight UML editor, intended for students, teachers, and authors who need to produce simple UML diagrams quickly. It is very easy to learn and use. It draws class, sequence, state, object and use-case diagrams. (Other diagram types have since been contributed.) It is open-source and cross-platform software. In its core, Violet uses a simple but flexible graph framework that takes full advantage of the Java 2D graphics API.

The Violet user interface is purposefully simple. You don't have to go through a tedious sequence of dialogs to enter attributes and methods. Instead, you just type them into a text field. With a few mouse clicks, you can quickly create attractive and useful diagrams.

Violet does not try to be an industrial-strength UML program. Here are some features that Violet does *not* have:

- Violet does not generate source code from UML diagrams or UML diagrams from source code.
- Violet does not carry out any semantic checking of models; you can use Violet to draw contradictory diagrams.
- Violet does not generate files that can be imported into other UML tools, nor can it read model files from other tools.
- Violet does not attempt to lay out diagrams automatically, except for a simple “snap to grid” facility.

(Attempting to address some of these limitations makes good student projects.)

When Violet developed a cult following of designers who wanted something more than a cocktail napkin but less than an industrial-strength UML tool, I published the code on SourceForge under the GNU General Public License. Starting in 2005, Alexandre de Pellegrin joined the project by providing an Eclipse plugin and a prettier user interface. He has since made numerous architectural changes and is now the primary maintainer of the project.

In this article, I discuss some of the original architectural choices in Violet as well as its evolution. A part of the article is focused on graph editing, but other parts—such as the use of JavaBeans properties and persistence, Java WebStart and plugin architecture—should be of general interest.

22.2 The Graph Framework

Violet is based on a general graph editing framework that can render and edit nodes and edges of arbitrary shapes. The Violet UML editor has nodes for classes, objects, activation bars (in sequence diagrams), and so on, and edges for the various edge shapes in UML diagrams. Another instance of the graph framework might display entity-relationship diagrams or railroad diagrams.

In order to illustrate the framework, let us consider an editor for very simple graphs, with black and white circular nodes and straight edges (Figure 22.2). The `SimpleGraph` class specifies prototype objects for the node and edge types, illustrating the prototype pattern:

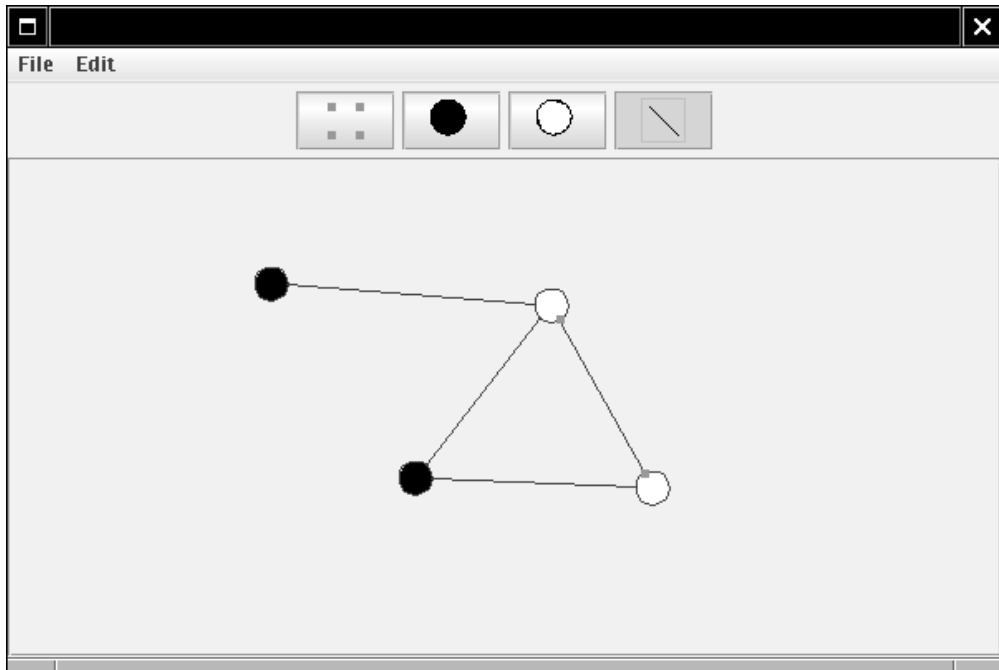


Figure 22.2: A Simple Instance of the Editor Framework

```
public class SimpleGraph extends AbstractGraph
{
    public Node[] getNodePrototypes()
    {
        return new Node[]
        {
            new CircleNode(Color.BLACK),
            new CircleNode(Color.WHITE)
        };
    }
    public Edge[] getEdgePrototypes()
    {
        return new Edge[]
        {
            new LineEdge()
        };
    }
}
```

Prototype objects are used to draw the node and edge buttons at the top of Figure 22.2. They are cloned whenever the user adds a new node or edge instance to the graph. Node and Edge are interfaces with the following key methods:

- Both interfaces have a `getShape` method that returns a Java2D Shape object of the node or edge shape.

- The Edge interface has methods that yield the nodes at the start and end of the edge.
- The getConnectionPoint method in the Node interface type computes an optimal attachment point on the boundary of a node (see Figure 22.3).
- The getConnectionPoints method of the Edge interface yields the two end points of the edge. This method is needed to draw the “grabbers” that mark the currently selected edge.
- A node can have children that move together with the parent. A number of methods are provided for enumerating and managing children.

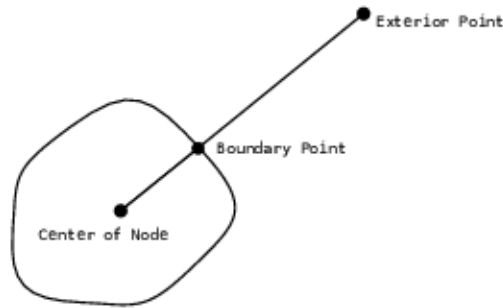


Figure 22.3: Finding a Connection Point on the Boundary of the Node Shape

Convenience classes `AbstractNode` and `AbstractEdge` implement a number of these methods, and classes `RectangularNode` and `SegmentedLineEdge` provide complete implementations of rectangular nodes with a title string and edges that are made up of line segments.

In the case of our simple graph editor, we would need to supply subclasses `CircleNode` and `LineEdge` that provide a draw method, a contains method, and the `getConnectionPoint` method that describes the shape of the node boundary. The code is given below, and Figure 22.4 shows a class diagram of these classes (drawn, of course, with Violet).

```
public class CircleNode extends AbstractNode
{
    public CircleNode(Color aColor)
    {
        size = DEFAULT_SIZE;
        x = 0;
        y = 0;
        color = aColor;
    }

    public void draw(Graphics2D g2)
    {
        Ellipse2D circle = new Ellipse2D.Double(x, y, size, size);
        Color oldColor = g2.getColor();
        g2.setColor(color);
        g2.fill(circle);
        g2.setColor(oldColor);
        g2.draw(circle);
    }
}
```

```

public boolean contains(Point2D p)
{
    Ellipse2D circle = new Ellipse2D.Double(x, y, size, size);
    return circle.contains(p);
}

public Point2D getConnectionPoint(Point2D other)
{
    double centerX = x + size / 2;
    double centerY = y + size / 2;
    double dx = other.getX() - centerX;
    double dy = other.getY() - centerY;
    double distance = Math.sqrt(dx * dx + dy * dy);
    if (distance == 0) return other;
    else return new Point2D.Double(
        centerX + dx * (size / 2) / distance,
        centerY + dy * (size / 2) / distance);
}

private double x, y, size, color;
private static final int DEFAULT_SIZE = 20;
}

public class LineEdge extends AbstractEdge
{
    public void draw(Graphics2D g2)
    { g2.draw(getConnectionPoints()); }

    public boolean contains(Point2D aPoint)
    {
        final double MAX_DIST = 2;
        return getConnectionPoints().ptSegDist(aPoint) < MAX_DIST;
    }
}

```

In summary, Violet provides a simple framework for producing graph editors. To obtain an editor instance, define node and edge classes and provide methods in a graph class that yield prototype node and edge objects.

Of course, there are other graph frameworks available, such as JGraph [Ald02] and JUNG². However, those frameworks are considerably more complex, and they provide frameworks for drawing graphs, not for applications that draw graphs.

22.3 Use of JavaBeans Properties

In the golden days of client-side Java, the JavaBeans specification was developed in order to provide portable mechanisms for editing GUI components in visual GUI builder environments. The vision was that a third-party GUI component could be dropped into any GUI builder, where its properties could be configured in the same way as the standard buttons, text components, and so on.

²<http://jung.sourceforge.net>

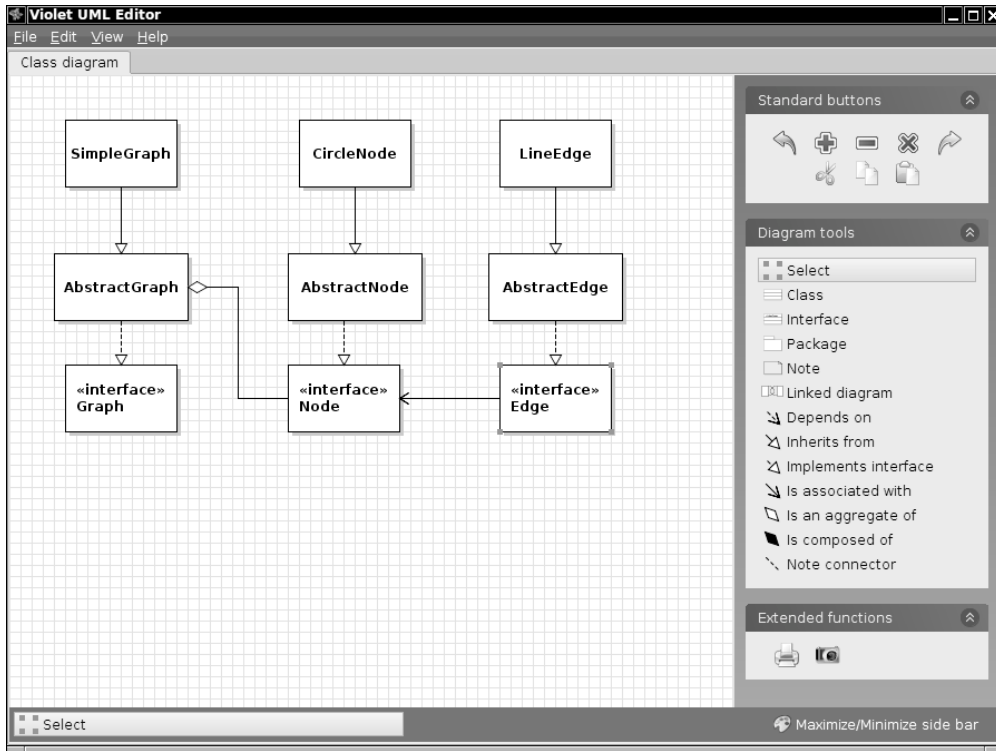


Figure 22.4: Class Diagram for a Simple Graph

Java does not have native properties. Instead, JavaBeans properties can be discovered as pairs of getter and setter methods, or specified with companion BeanInfo classes. Moreover, *property editors* can be specified for visually editing property values. The JDK even contains a few basic property editors, for example for the type `java.awt.Color`.

The Violet framework makes full use of the JavaBeans specification. For example, the `CircleNode` class can expose a color property simply by providing two methods:

```
public void setColor(Color newValue)
public Color getColor()
```

No further work is necessary. The graph editor can now edit node colors of circle nodes (Figure 22.5).

22.4 Long-Term Persistence

Just like any editor program, Violet must save the user's creations in a file and reload them later. I had a look at the XMI specification³ which was designed as a common interchange format for UML models. I found it cumbersome, confusing, and hard to consume. I don't think I was the only one—XMI had a reputation for poor interoperability even with the simplest models [PGL⁺05].

³<http://www.omg.org/technology/documents/formal/xmi.htm>

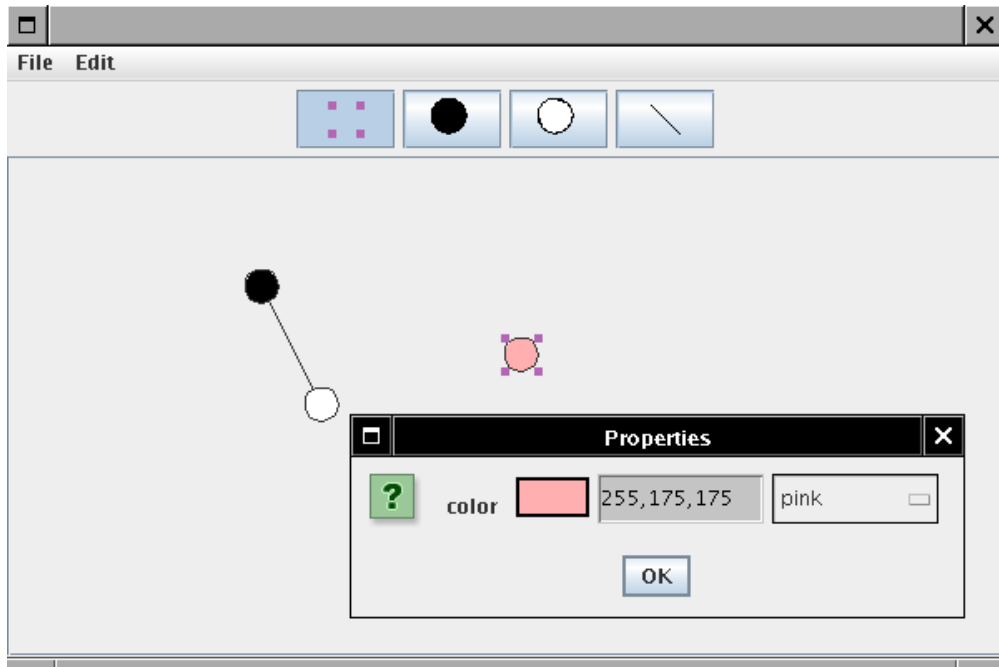


Figure 22.5: Editing Circle Colors with the default JavaBeans Color Editor

I considered simply using Java serialization, but it is difficult to read old versions of a serialized object whose implementation has changed over time. This problem was also anticipated by the JavaBeans architects, who developed a standard XML format for long-term persistence⁴. A Java object—in the case of Violet, the UML diagram—is serialized as a sequence of statements for constructing and modifying it. Here is an example:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.0" class="java.beans.XMLDecoder">
  <object class="com.horstmann.violet.ClassDiagramGraph">
    <void method="addNode">
      <object id="ClassNode0" class="com.horstmann.violet.ClassNode">
        <void property="name">...</void>
      </object>
      <object class="java.awt.geom.Point2D$Double">
        <double>200.0</double>
        <double>60.0</double>
      </object>
    </void>
    <void method="addNode">
      <object id="ClassNode1" class="com.horstmann.violet.ClassNode">
        <void property="name">...</void>
      </object>
      <object class="java.awt.geom.Point2D$Double">
```

⁴<http://jcp.org/en/jsr/detail?id=57>

```

        <double>200.0</double>
        <double>210.0</double>
    </object>
</void>
<void method="connect">
    <object class="com.horstmann.violet.ClassRelationshipEdge">
        <void property="endArrowHead">
            <object class="com.horstmann.violet.ArrowHead" field="TRIANGLE"/>
        </void>
    </object>
    <object idref="ClassNode0"/>
    <object idref="ClassNode1"/>
</void>
</object>
</java>

```

When the XMLDecoder class reads this file, it executes these statements (package names are omitted for simplicity).

```

ClassDiagramGraph obj1 = new ClassDiagramGraph();
ClassNode ClassNode0 = new ClassNode();
ClassNode0.setName(...);
obj1.addNode(ClassNode0, new Point2D.Double(200, 60));
ClassNode ClassNode1 = new ClassNode();
ClassNode1.setName(...);
obj1.addNode(ClassNode1, new Point2D.Double(200, 60));
ClassRelationshipEdge obj2 = new ClassRelationshipEdge();
obj2.setEndArrowHead(ArrowHead.TRIANGLE);
obj1.connect(obj2, ClassNode0, ClassNode1);

```

As long as the semantics of the constructors, properties, and methods has not changed, a newer version of the program can read a file that has been produced by an older version.

Producing such files is quite straightforward. The encoder automatically enumerates the properties of each object and writes setter statements for those property values that differ from the default. Most basic datatypes are handled by the Java platform; however, I had to supply special handlers for Point2D, Line2D, and Rectangle2D. Most importantly, the encoder must know that a graph can be serialized as a sequence of addNode and connect method calls:

```

encoder.setPersistenceDelegate(Graph.class, new DefaultPersistenceDelegate()
{
    protected void initialize(Class<?> type, Object oldInstance,
        Object newInstance, Encoder out)
    {
        super.initialize(type, oldInstance, newInstance, out);
        AbstractGraph g = (AbstractGraph) oldInstance;
        for (Node n : g.getNodes())
            out.writeStatement(new Statement(oldInstance, "addNode", new Object[]
            {
                n,
                n.getLocation()
            }));
    }
});

```

```

    for (Edge e : g.getEdges())
        out.writeStatement(new Statement(oldInstance, "connect", new Object[]
        {
            e, e.getStart(), e.getEnd()
        }));
    }
});

```

Once the encoder has been configured, saving a graph is as simple as:

```
encoder.writeObject(graph);
```

Since the decoder simply executes statements, it requires no configuration. Graphs are simply read with:

```
Graph graph = (Graph) decoder.readObject();
```

This approach has worked exceedingly well over numerous versions of Violet, with one exception. A recent refactoring changed some package names and thereby broke backwards compatibility. One option would have been to keep the classes in the original packages, even though they no longer matched the new package structure. Instead, the maintainer provided an XML transformer for rewriting the package names when reading a legacy file.

22.5 Java WebStart

Java WebStart is a technology for launching an application from a web browser. The deployer posts a JNLP file that triggers a helper application in the browser which downloads and runs the Java program. The application can be digitally signed, in which case the user must accept the certificate, or it can be unsigned, in which case the program runs in a sandbox that is slightly more permissive than the applet sandbox.

I do not think that end users can or should be trusted to judge the validity of a digital certificate and its security implications. One of the strengths of the Java platform is its security, and I feel it is important to play to that strength.

The Java WebStart sandbox is sufficiently powerful to enable users to carry out useful work, including loading and saving files and printing. These operations are handled securely and conveniently from the user perspective. The user is alerted that the application wants to access the local filesystem and then chooses the file to be read or written. The application merely receives a stream object, without having an opportunity to peek at the filesystem during the file selection process.

It is annoying that the developer must write custom code to interact with a `FileOpenService` and a `FileSaveService` when the application is running under WebStart, and it is even more annoying that there is no WebStart API call to find out whether the application was launched by WebStart.

Similarly, saving user preferences must be implemented in two ways: using the Java preferences API when the application runs normally, or using the WebStart preferences service when the application is under WebStart. Printing, on the other hand, is entirely transparent to the application programmer.

Violet provides simple abstraction layers over these services to simplify the lot of the application programmer. For example, here is how to open a file:

```

FileService service = FileService.getInstance(initialDirectory);
// detects whether we run under WebStart
FileService.Open open = fileService.open(defaultDirectory, defaultName,
    extensionFilter);
InputStream in = open.getInputStream();
String title = open.getName();

```

The `FileService.Open` interface is implemented by two classes: a wrapper over `JFileChooser` or the JNLP `FileOpenService`.

No such convenience is a part of the JNLP API itself, but that API has received little love over its lifetime and has been widely ignored. Most projects simply use a self-signed certificate for their WebStart application, which gives users no security. This is a shame—open source developers should embrace the JNLP sandbox as a risk-free way to try out a project.

22.6 Java 2D

Violet makes intensive use of the Java2D library, one of the lesser known gems in the Java API. Every node and edge has a method `getShape` that yields a `java.awt.Shape`, the common interface of all Java2D shapes. This interface is implemented by rectangles, circles, paths, and their unions, intersections, and differences. The `GeneralPath` class is useful for making shapes that are composed of arbitrary line and quadratic/cubic curve segments, such as straight and curved arrows.

To appreciate the flexibility of the Java2D API, consider the following code for drawing a shadow in the `AbstractNode.draw` method:

```

Shape shape = getShape();
if (shape == null) return;
g2.translate(SHADOW_GAP, SHADOW_GAP);
g2.setColor(SHADOW_COLOR);
g2.fill(shape);
g2.translate(-SHADOW_GAP, -SHADOW_GAP);
g2.setColor(BACKGROUND_COLOR);
g2.fill(shape);

```

A few lines of code produce a shadow for any shape, even shapes that a developer may add at a later point.

Of course, Violet saves bitmap images in any format that the `javax.imageio` package supports; that is, GIF, PNG, JPEG, and so on. When my publisher asked me for vector images, I noted another advantage of the Java 2D library. When you print to a PostScript printer, the Java2D operations are translated into PostScript vector drawing operations. If you print to a file, the result can be consumed by a program such as `ps2eps` and then imported into Adobe Illustrator or Inkscape. Here is the code, where `comp` is the Swing component whose `paintComponent` method paints the graph:

```

DocFlavor flavor = DocFlavor.SERVICE_FORMATTED.PRINTABLE;
String mimeType = "application/postscript";
StreamPrintServiceFactory[] factories;
StreamPrintServiceFactory.lookupStreamPrintServiceFactories(flavor, mimeType);
FileOutputStream out = new FileOutputStream(fileName);
PrintService service = factories[0].getPrintService(out);
SimpleDoc doc = new SimpleDoc(new Printable() {

```



```

public int print(Graphics g, PageFormat pf, int page) {
    if (page >= 1) return Printable.NO_SUCH_PAGE;
    else {
        double sf1 = pf.getImageableWidth() / (comp.getWidth() + 1);
        double sf2 = pf.getImageableHeight() / (comp.getHeight() + 1);
        double s = Math.min(sf1, sf2);
        Graphics2D g2 = (Graphics2D) g;
        g2.translate((pf.getWidth() - pf.getImageableWidth()) / 2,
            (pf.getHeight() - pf.getImageableHeight()) / 2);
        g2.scale(s, s);

        comp.paint(g);
        return Printable.PAGE_EXISTS;
    }
}
}, flavor, null);
DocPrintJob job = service.createPrintJob();
PrintRequestAttributeSet attributes = new HashPrintRequestAttributeSet();
job.print(doc, attributes);

```

At the beginning, I was concerned that there might be a performance penalty when using general shapes, but that has proven not to be the case. Clipping works well enough that only those shape operations that are required for updating the current viewport are actually executed.

22.7 No Swing Application Framework

Most GUI frameworks have some notion of an application that manages a set of documents that deals with menus, toolbars, status bars, etc. However, this was never a part of the Java API. JSR 296⁵ was supposed to supply a basic framework for Swing applications, but it is currently inactive. Thus, a Swing application author has two choices: reinvent a good number of wheels or base itself on a third party framework. At the time that Violet was written, the primary choices for an application framework were the Eclipse and NetBeans platform, both of which seemed too heavyweight at the time. (Nowadays, there are more choices, among them JSR 296 forks such as GUTS⁶.) Thus, Violet had to reinvent mechanisms for handling menus and internal frames.

In Violet, you specify menu items in property files, like this:

```

file.save.text=Save
file.save.mnemonic=S
file.save.accelerator=ctrl S
file.save.icon=/icons/16x16/save.png

```

A utility method creates the menu item from the prefix (here file.save). The suffixes .text, .mnemonic, and so on, are what nowadays would be called “convention over configuration”. Using resource files for describing these settings is obviously far superior to setting up menus with API calls because it allows for easy localization. I reused that mechanism in another open source project, the GridWorld environment for high school computer science education⁷.

⁵<http://jcp.org/en/jsr/detail?id=296>

⁶<http://kenai.com/projects/guts>

⁷<http://horstmann.com/gridworld>

An application such as Violet allows users to open multiple “documents”, each containing a graph. When Violet was first written, the multiple document interface (MDI) was still commonly used. With MDI, the main frame has a menu bar, and each view of a document is displayed in an internal frame with a title but no menu bar. Each internal frame is contained in the main frame and can be resized or minimized by the user. There are operations for cascading and tiling windows.

Many developers disliked MDI, and so this style user interface has gone out of fashion. For a while, a single document interface (SDI), in which an application displays multiple top level frames, was considered superior, presumably because those frames can be manipulated with the standard window management tools of the host operating system. When it became clear that having lots of top level windows isn’t so great after all, tabbed interfaces started to appear, in which multiple documents are again contained in a single frame, but now all displayed at full size and selectable with tabs. This does not allow users to compare two documents side by side, but seems to have won out.

The original version of Violet used an MDI interface. The Java API has an internal frames feature, but I had to add support for tiling and cascading. Alexandre switched to a tabbed interface, which is somewhat better-supported by the Java API. It would be desirable to have an application framework where the document display policy was transparent to the developer and perhaps selectable by the user.

Alexandre also added support for sidebars, a status bar, a welcome panel, and a splash screen. All this should ideally be a part of a Swing application framework.

22.8 Undo/Redo

Implementing multiple undo/redo seems like a daunting task, but the Swing undo package ([Top00], Chapter 9) gives good architectural guidance. An `UndoManager` manages a stack of `UndoableEdit` objects. Each of them has an `undo` method that undoes the effect of the edit operation, and a `redo` method that undoes the undo (that is, carries out the original edit operation). A `CompoundEdit` is a sequence of `UndoableEdit` operations that should be undone or redone in their entirety. You are encouraged to define small, atomic edit operations (such as adding or removing a single edge or node in the case of a graph) that are grouped into compound edits as necessary.

A challenge is to define a small set of atomic operations, each of which can be undone easily. In Violet, they are:

- adding or removing a node or edge
- attaching or detaching a node’s child
- moving a node
- changing a property of a node or edge

Each of these operations has an obvious undo. For example the undo of adding a node is the node’s removal. The undo of moving a node is to move it by the opposite vector.

Note that these atomic operations are *not* the same as the actions in the user interface or the methods of the `Graph` interface that the user interface actions invoke. For example, consider the sequence diagram in Figure 22.6, and suppose the user drags the mouse from the activation bar to the lifeline on the right. When the mouse button is released, the method:

```
public boolean addEdgeAtPoints(Edge e, Point2D p1, Point2D p2)
```

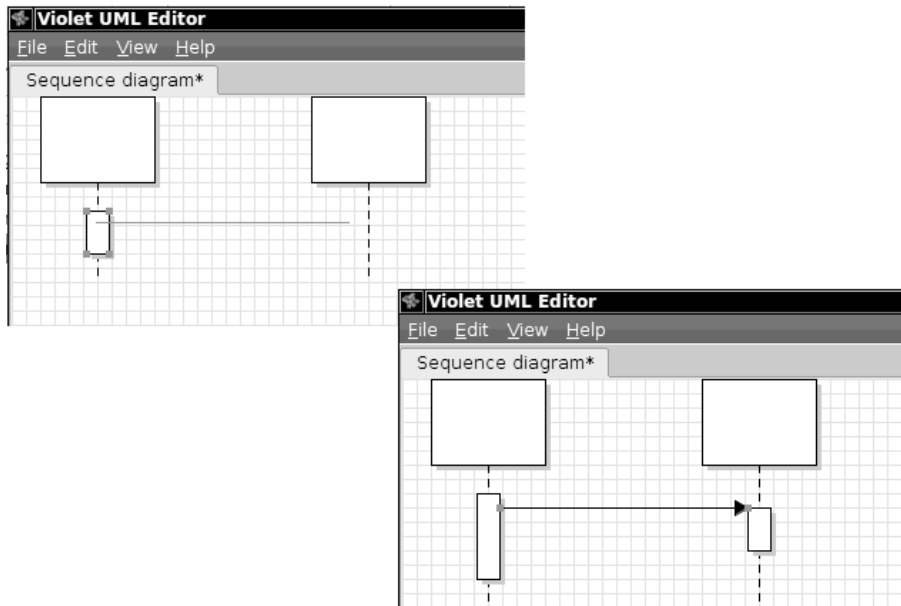


Figure 22.6: An Undo Operation Must Undo Structural Changes in the Model

is invoked. That method adds an edge, but it may also carry out other operations, as specified by the participating Edge and Node subclasses. In this case, an activation bar will be added to the lifeline on the right. Undoing the operation needs to remove that activation bar as well. Thus, the *model* (in our case, the graph) needs to record the structural changes that need to be undone. It is not enough to collect controller operations.

As envisioned by the Swing undo package, the graph, node, and edge classes should send `UndoableEditEvent` notifications to an `UndoManager` whenever a structural edit occurs. Violet has a more general design where the graph itself manages listeners for the following interface:

```
public interface GraphModificationListener
{
    void nodeAdded(Graph g, Node n);
    void nodeRemoved(Graph g, Node n);
    void nodeMoved(Graph g, Node n, double dx, double dy);
    void childAttached(Graph g, int index, Node p, Node c);
    void childDetached(Graph g, int index, Node p, Node c);
    void edgeAdded(Graph g, Edge e);
    void edgeRemoved(Graph g, Edge e);
    void propertyChangedOnNodeOrEdge(Graph g, PropertyChangeEvent event);
}
```

The framework installs a listener into each graph that is a bridge to the undo manager. For supporting undo, adding generic listener support to the model is overdesigned—the graph operations could directly interact with the undo manager. However, I also wanted to support an experimental collaborative editing feature.

If you want to support undo/redo in your application, think carefully about the atomic operations in your model (and not your user interface). In the model, fire events when a structural change happens, and allow the Swing undo manager to collect and group these events.

22.9 Plugin Architecture

For a programmer familiar with 2D graphics, it is not difficult to add a new diagram type to Violet. For example, the activity diagrams were contributed by a third party. When I needed to create railroad diagrams and ER diagrams, I found it faster to write Violet extensions instead of fussing with Visio or Dia. (Each diagram type took a day to implement.)

These implementations do not require knowledge of the full Violet framework. Only the graph, node, and edge interfaces and convenience implementations are needed. In order to make it easier for contributors to decouple themselves from the evolution of the framework, I designed a simple plugin architecture.

Of course, many programs have a plugin architecture, many quite elaborate. When someone suggested that Violet should support OSGi, I shuddered and instead implemented the simplest thing that works.

Contributors simply produce a JAR file with their graph, node, and edge implementations and drop it into a `plugins` directory. When Violet starts, it loads those plugins, using the Java `ServiceLoader` class. That class was designed to load services such as JDBC drivers. A `ServiceLoader` loads JAR files that promise to provide a class implementing a given interface (in our case, the `Graph` interface.)

Each JAR file must have a subdirectory `META-INF/services` containing a file whose name is the fully qualified classname of the interface (such as `com.horstmann.violet.Graph`), and that contains the names of all implementing classes, one per line. The `ServiceLoader` constructs a class loader for the plugin directory, and loads all plugins:

```
ServiceLoader<Graph> graphLoader = ServiceLoader.load(Graph.class, classLoader);
for (Graph g : graphLoader) // ServiceLoader<Graph> implements Iterable<Graph>
    registerGraph(g);
```

This is a simple but useful facility of standard Java that you might find valuable for your own projects.

22.10 Conclusion

Like so many open source projects, Violet was born of an unmet need—to draw simple UML diagrams with a minimum of fuss. Violet was made possible by the amazing breadth of the Java SE platform, and it draws from a diverse set of technologies that are a part of that platform. In this article, I described how Violet makes use of Java Beans, Long-Term Persistence, Java Web Start, Java 2D, Swing Undo/Redo, and the service loader facility. These technologies are not always as well understood as the basics of Java and Swing, but they can greatly simplify the architecture of a desktop application. They allowed me, as the initial sole developer, to produce a successful application in a few months of part-time work. Relying on these standard mechanisms also made it easier for others to improve on Violet and to extract pieces of it into their own projects.

VisTrails

Juliana Freire, David Koop, Emanuele Santos,
Carlos Scheidegger, Claudio Silva, and Huy T. Vo

VisTrails¹ is an open-source system that supports data exploration and visualization. It includes and substantially extends useful features of scientific workflow and visualization systems. Like scientific workflow systems such as Kepler and Taverna, VisTrails allows the specification of computational processes which integrate existing applications, loosely-coupled resources, and libraries according to a set of rules. Like visualization systems such as AVS and ParaView, VisTrails makes advanced scientific and information visualization techniques available to users, allowing them to explore and compare different visual representations of their data. As a result, users can create complex workflows that encompass important steps of scientific discovery, from data gathering and manipulation to complex analyses and visualizations, all integrated in one system.

A distinguishing feature of VisTrails is its provenance infrastructure [FSC⁺06]. VisTrails captures and maintains a detailed history of the steps followed and data derived in the course of an exploratory task. Workflows have traditionally been used to automate repetitive tasks, but in applications that are exploratory in nature, such as data analysis and visualization, very little is repeated—change is the norm. As a user generates and evaluates hypotheses about their data, a series of different, but related, workflows are created as they are adjusted iteratively.

VisTrails was designed to manage these rapidly-evolving workflows: it maintains provenance of data products (e.g., visualizations, plots), of the workflows that derive these products, and their executions. The system also provides annotation capabilities so users can enrich the automatically-captured provenance.

Besides enabling reproducible results, VisTrails leverages provenance information through a series of operations and intuitive user interfaces that help users to collaboratively analyze data. Notably, the system supports reflective reasoning by storing temporary results, allowing users to examine the actions that led to a result and to follow chains of reasoning backward and forward. Users can navigate workflow versions in an intuitive way, undo changes without losing results, visually compare multiple workflows and show their results side-by-side in a visualization spreadsheet.

VisTrails addresses important usability issues that have hampered a wider adoption of workflow and visualization systems. To cater to a broader set of users, including many who do not have programming expertise, it provides a series of operations and user interfaces that simplify workflow design and use [FSC⁺06], including the ability to create and refine workflows by analogy, to query

¹<http://www.vistrails.org>

workflows by example, and to suggest workflow completions as users interactively construct their workflows using a recommendation system [SVK⁺07]. We have also developed a new framework that allows the creation of custom applications that can be more easily deployed to (non-expert) end users.

The extensibility of VisTrails comes from an infrastructure that makes it simple for users to integrate tools and libraries, as well as to quickly prototype new functions. This has been instrumental in enabling the use of the system in a wide range of application areas, including environmental sciences, psychiatry, astronomy, cosmology, high-energy physics, quantum physics, and molecular modeling.

To keep the system open-source and free for all, we have built VisTrails using only free, open-source packages. VisTrails is written in Python and uses Qt as its GUI toolkit (through PyQt Python bindings). Because of the broad range of users and applications, we have designed the system from the ground up with portability in mind. VisTrails runs on Windows, Mac and Linux.

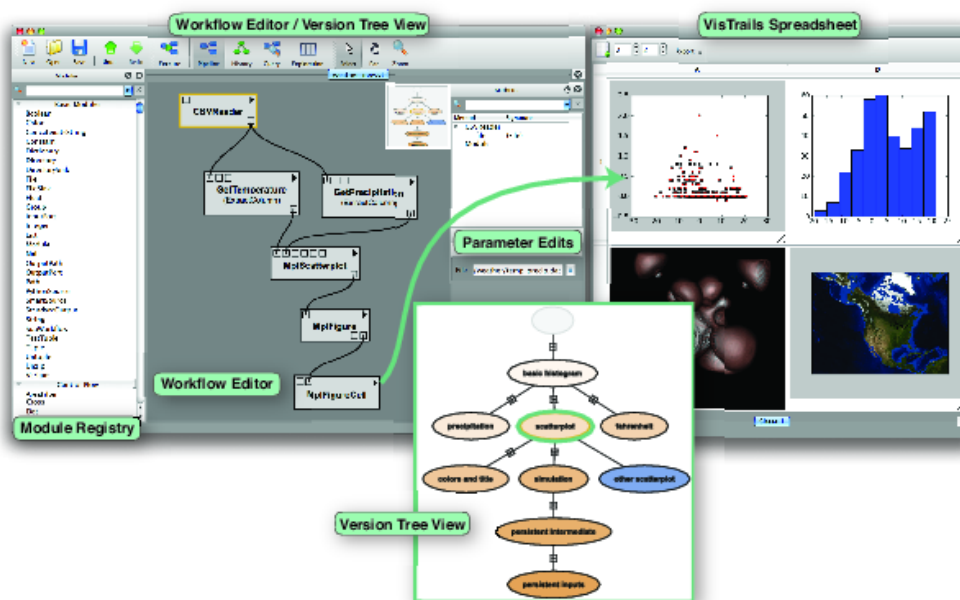


Figure 23.1: Components of the VisTrails User Interface

23.1 System Overview

Data exploration is an inherently creative process that requires users to locate relevant data, to integrate and visualize this data, to collaborate with peers while exploring different solutions, and to disseminate results. Given the size of data and complexity of analyses that are common in scientific exploration, tools are needed that better support creativity.

There are two basic requirements for these tools that go hand in hand. First, it is important to be able to specify the exploration processes using formal descriptions, which ideally, are executable.

Second, to reproduce the results of these processes as well as reason about the different steps followed to solve a problem, these tools must have the ability to systematically capture provenance. VisTrails was designed with these requirements in mind.

Workflows and Workflow-Based Systems

Workflow systems support the creation of pipelines (workflows) that combine multiple tools. As such, they enable the automation of repetitive tasks and result reproducibility. Workflows are rapidly replacing primitive shell scripts in a wide range of tasks, as evidenced by a number of workflow-based applications, both commercial (e.g., Apple's Mac OS X Automator and Yahoo! Pipes) and academic (e.g., NiPype, Kepler, and Taverna).

Workflows have a number of advantages compared to scripts and programs written in high-level languages. They provide a simple programming model whereby a sequence of tasks is composed by connecting the outputs of one task to the inputs of another. Figure 23.1 shows a workflow which reads a CSV file that contains weather observations and creates a scatter plot of the values.

This simpler programming model allows workflow systems to provide intuitive visual programming interfaces, which make them more *suitable for users who do not have substantial programming expertise*. Workflows also have an *explicit structure*: they can be viewed as graphs, where nodes represent processes (or modules) along with their parameters and edges capture the flow of data between the processes. In the example of Figure 23.1, the module CSVReader takes as a parameter a filename (/weather/temp_precip.dat), reads the file, and feeds its contents into the modules GetTemperature and GetPrecipitation, which in turn send the temperature and precipitation values to a matplotlib function that generates a scatter plot.

Most workflow systems are designed for a specific application area. For example, Taverna targets bioinformatics workflows, and NiPype allows the creation of neuroimaging workflows. While VisTrails supports much of the functionality provided by other workflow systems, it was designed to support general exploratory tasks in a broad range of areas, integrating multiple tools, libraries, and services.

Data and Workflow Provenance

The importance of keeping provenance information for results (and data products) is well recognized in the scientific community. The provenance (also referred to as the audit trail, lineage, and pedigree) of a data product contains information about the process and data used to derive the data product. Provenance provides important documentation that is key to preserving the data, to determining the data's quality and authorship, and to reproducing as well as validating the results [FKSS08].

An important component of provenance is information about *causality*, i.e., a description of a process (sequence of steps) which, together with input data and parameters, caused the creation of a data product. Thus, the structure of provenance mirrors the structure of the workflow (or set of workflows) used to derive a given result set.

In fact, a catalyst for the widespread use of workflow systems in science has been that they can be easily used to automatically capture provenance. While early workflow systems have been *extended* to capture provenance, VisTrails was *designed* to support provenance.

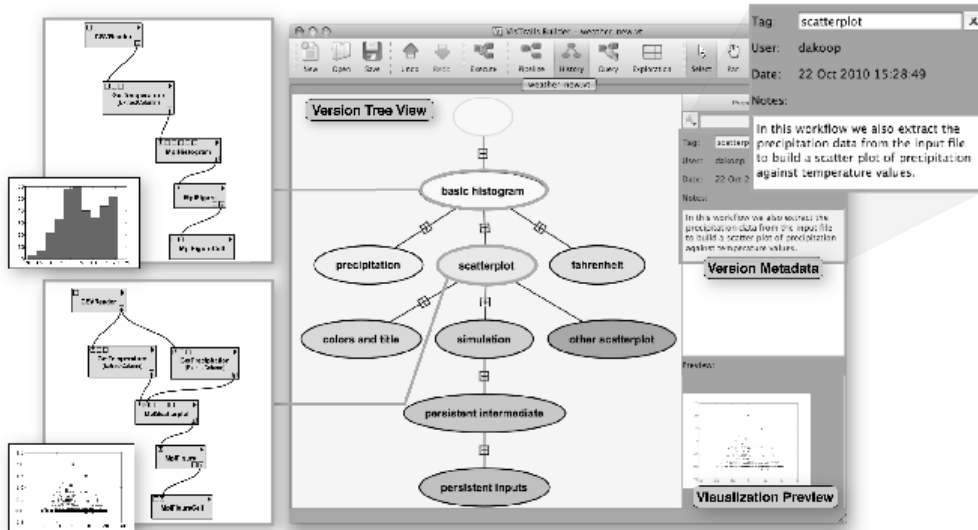


Figure 23.2: Provenance of Exploration Enhanced by Annotations

User Interface and Basic Functionality

The different user interface components of the system are illustrated in Figure 23.1 and Figure 23.2. Users create and edit workflows using the Workflow Editor. To build the workflow graphs, users can drag modules from the Module Registry and drop them into the Workflow Editor canvas. VisTrails provides a series of built-in modules, and users can also add their own (see Section 23.3 for details). When a module is selected, VisTrails displays its parameters (in the Parameter Edits area) where the user can set and modify their values.

As a workflow specification is refined, the system captures the changes and presents them to the user in the Version Tree View described below. Users may interact with the workflows and their results in the VisTrails Spreadsheet. Each cell in the spreadsheet represents a view that corresponds to a workflow instance. In Figure 23.1, the results of the workflow shown in the Workflow Editor are displayed on the top-left cell of the spreadsheet. Users can directly modify the parameters of a workflow as well as synchronize parameters across different cells in the spreadsheet.

The Version Tree View helps users to navigate through the different workflow versions. As shown in Figure 23.2, by clicking on a node in the version tree, users can view a workflow, its associated result (Visualization Preview), and metadata. Some of the metadata is automatically captured, e.g., the id of the user who created a particular workflow and the creation date, but users may also provide additional metadata, including a tag to identify the workflow and a written description.

23.2 Project History

Initial versions of versions of VisTrails were written in Java and C++ [BCC⁺05]. The C++ version was distributed to a few early adopters, whose feedback was instrumental in shaping our requirements for the system.

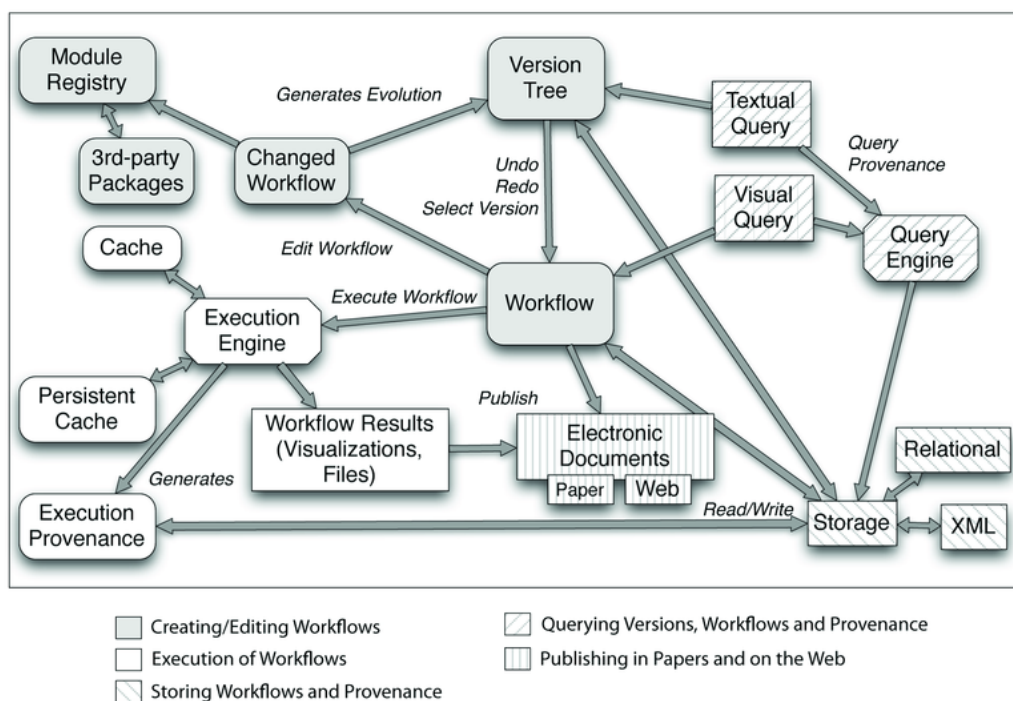


Figure 23.3: VisTrails Architecture

Having observed a trend in the increase of the number of Python-based libraries and tools in multiple scientific communities, we opted to use Python as the basis for VisTrails. Python is quickly becoming a universal modern glue language for scientific software. Many libraries written in different languages such as Fortran, C, and C++ use Python bindings as a way to provide scripting capabilities. Since VisTrails aims to facilitate the orchestration of many different software libraries in workflows, a pure Python implementation makes this much easier. In particular, Python has dynamic code loading features similar to the ones seen in LISP environments, while having a much bigger developer community, and an extremely rich standard library. Late in 2005, we started the development of the current system using Python/PyQt/Qt. This choice has greatly simplified extensions to the system, in particular, the addition of new modules and packages.

A beta version of the VisTrails system was first released in January 2007. Since then, the system has been downloaded over twenty-five thousand times.

23.3 Inside VisTrails

The internal components that support the user-interface functionality described above are depicted in the high-level architecture of VisTrails, shown in Figure 23.3. Workflow execution is controlled by the Execution Engine, which keeps track of invoked operations and their respective parameters and captures the provenance of workflow execution (Execution Provenance). As part of the execution, VisTrails also allows the caching of intermediate results both in memory and on disk. As we discuss

in Section 23.3, only new combinations of modules and parameters are re-run, and these are executed by invoking the appropriate functions from the underlying libraries (e.g., matplotlib). Workflow results, connected to their provenance, can then be included in electronic documents (Section 23.4).

Information about changes to workflows is captured in a Version Tree, which can be persisted using different storage back ends, including an XML file store in a local directory and a relational database. VisTrails also provides a query engine that allows users to explore the provenance information.

We note that, although VisTrails was designed as an interactive tool, it can also be used in server mode. Once workflows are created, they can be executed by a VisTrails server. This feature is useful in a number of scenarios, including the creation of Web-based interfaces that allows users to interact with workflows and the ability to run workflows in high-performance computing environments.

The Version Tree: Change-Based Provenance

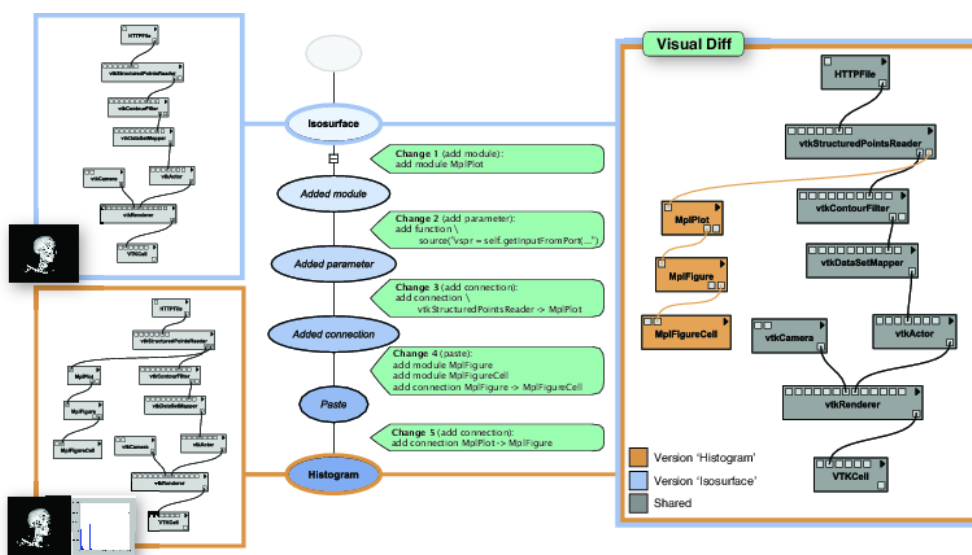


Figure 23.4: Change-Based Provenance Model

A new concept we introduced with VisTrails is the notion of provenance of workflow evolution [FSC⁺06]. In contrast to previous workflow and workflow-based visualization systems, which maintain provenance only for derived data products, VisTrails treats the workflows as first-class data items and also captures their provenance. The availability of workflow-evolution provenance supports reflective reasoning. Users can explore multiple chains of reasoning without losing any results, and because the system stores intermediate results, users can reason about and make inferences from this information. It also enables a series of operations which simplify exploratory processes. For example, users can easily navigate through the space of workflows created for a given task, visually compare the workflows and their results (see Figure 23.4), and explore (large) parameter spaces. In addition, users can query the provenance information and learn by example.

The workflow evolution is captured using the change-based provenance model. As illustrated in Figure 23.4, VisTrails stores the operations or changes that are applied to workflows (e.g., the

addition of a module, the modification of a parameter, etc.), akin to a database transaction log. This information is modeled as a tree, where each node corresponds to a workflow version, and an edge between a parent and a child node represents the change applied to the parent to obtain the child. We use the terms version tree and *vistail* (short for *visual trail*) interchangeably to refer to this tree. Note that the change-based model uniformly captures both changes to parameter values and to workflow definitions. This sequence of changes is sufficient to determine the provenance of data products and it also captures information about how a workflow evolves over time. The model is both simple and compact—it uses substantially less space than the alternative of storing multiple *versions* of a workflow.

There are a number of benefits that come from the use of this model. Figure 23.4 shows the visual difference functionality that VisTrails provides for comparing two workflows. Although the workflows are represented as graphs, using the change-based model, comparing two workflows becomes very simple: it suffices to navigate the version tree and identify the series of actions required to transform one workflow into the other.

Another important benefit of the change-based provenance model is that the underlying version tree can serve as a mechanism to support collaboration. Because designing workflows is a notoriously difficult task, it often requires multiple users to collaborate. Not only does the version tree provide an intuitive way to visualize the contribution of different users (e.g., by coloring nodes according to the user who created the corresponding workflow), but the monotonicity of the model allows for simple algorithms for synchronizing changes performed by multiple users.

Provenance information can be easily captured while a workflow is being executed. Once the execution completes, it is also important to maintain *strong* links between a data product and its provenance, i.e., the workflow, parameters and input files used to derive the data product. When data files or provenance are moved or modified, it can be difficult to find the data associated with the provenance or to find the provenance associated with the data. VisTrails provides a persistent storage mechanism that manages input, intermediate, and output data files, strengthening the links between provenance and data. This mechanism provides better support for reproducibility because it ensures the data referenced in provenance information can be readily (and correctly) located. Another important benefit of such management is that it allows caching of intermediate data which can then be shared with other users.

Workflow Execution and Caching

The execution engine in VisTrails was designed to allow the integration of new and existing tools and libraries. We tried to accommodate different styles commonly used for wrapping third-party scientific visualization and computation software. In particular, VisTrails can be integrated with application libraries that exist either as pre-compiled binaries that are executed on a shell and use files as input/outputs, or as C++/Java/Python class libraries that pass internal objects as input/output.

VisTrails adopts a dataflow execution model, where each module performs a computation and the data produced by a module flows through the connections that exist between modules. Modules are executed in a bottom-up fashion; each input is generated on-demand by recursively executing upstream modules (we say module A is *upstream* of B when there is a sequence of connections that goes from A to B). The intermediate data is temporarily stored either in memory (as a Python object) or on disk (wrapped by a Python object that contains information on accessing the data).

To allow users to add their own functionality to VisTrails, we built an extensible package system (see Section 23.3). Packages allow users to include their own or third-party modules in VisTrails workflows. A package developer must identify a set of computational modules and for each, identify

the input and output ports as well as define the computation. For existing libraries, a compute method needs to specify the translation from input ports to parameters for the existing function and the mapping from result values to output ports.

In exploratory tasks, similar workflows, which share common sub-structures, are often executed in close succession. To improve the efficiency of workflow execution, VisTrails caches intermediate results to minimize recomputation. Because we reuse previous execution results, we implicitly assume that cacheable modules are functional: given the same inputs, modules will produce the same outputs. This requirement imposes definite behavior restrictions on classes, but we believe they are reasonable.

There are, however, obvious situations where this behavior is unattainable. For example, a module that uploads a file to a remote server or saves a file to disk has a significant side effect while its output is relatively unimportant. Other modules might use randomization, and their non-determinism might be desirable; such modules can be flagged as non-cacheable. However, some modules that are not naturally functional can be converted; a function that writes data to two files might be wrapped to output the contents of the files.

Data Serialization and Storage

One of the key components of any system supporting provenance is the serialization and storage of data. VisTrails originally stored data in XML via simple `fromXML` and `toXML` methods embedded in its internal objects (e.g., the version tree, each module). To support the evolution of the schema of these objects, these functions encoded any translation between schema versions as well. As the project progressed, our user base grew, and we decided to support different serializations, including relational stores. In addition, as schema objects evolved, we needed to maintain better infrastructure for common data management concerns like versioning schemas, translating between versions, and supporting entity relationships. To do so, we added a new database (db) layer.

The db layer is composed of three core components: the domain objects, the service logic, and the persistence methods. The domain and persistence components are versioned so that each schema version has its own set of classes. This way, we maintain code to read each version of the schema. There are also classes that define translations for objects from one schema version to those of another. The service classes provide methods to interface with data and deal with detection and translation of schema versions.

Because writing much of this code is tedious and repetitive, we use templates and a meta-schema to define both the object layout (and any in-memory indices) and the serialization code. The meta-schema is written in XML, and is extensible in that serializations other than the default XML and relational mappings VisTrails defines can be added. This is similar to object-relational mappings and frameworks like Hibernate² and SQLAlchemy³, but adds some special routines to automate tasks like re-mapping identifiers and translating objects from one schema version to the next. In addition, we can also use the same meta-schema to generate serialization code for many languages. After originally writing meta-Python, where the domain and persistence code was generated by running Python code with variables obtained from the meta-schema, we have recently migrated to Mako templates⁴.

Automatic translation is key for users that need to migrate their data to newer versions of the system. Our design adds hooks to make this translation slightly less painful for developers. Because

²<http://www.hibernate.org>

³<http://www.sqlalchemy.org>

⁴<http://www.makotemplates.org>

we maintain a copy of code for each version, the translation code just needs to map one version to another. At the root level, we define a map to identify how any version can be transformed to any other. For distant versions, this usually involves a chain through multiple intermediate versions. Initially, this was a forward-only map, meaning new versions could not be translated to old versions, but reverse mappings have been added for more-recent schema mappings.

Each object has an `update_version` method that takes a different version of an object and returns the current version. By default, it does a recursive translation where each object is upgraded by mapping fields of the old object to those in a new version. This mapping defaults to copying each field to one with the same name, but it is possible to define a method to "override" the default behavior for any field. An override is a method that takes the old object and returns a new version. Because most changes to the schema only affect a small number of fields, the default mappings cover most cases, but the overrides provide a flexible means for defining local changes.

Extensibility Through Packages and Python

The first prototype of VisTrails had a fixed set of modules. It was an ideal environment to develop basic ideas about the VisTrails version tree and the caching of multiple execution runs, but it severely limited long-term utility.

We see VisTrails as infrastructure for computational science, and that means, literally, that the system should provide scaffolding for other tools and processes to be developed. An essential requirement of this scenario is extensibility. A typical way to achieve this involves defining a target language and writing an appropriate interpreter. This is appealing because of the intimate control it offers over execution. This appeal is amplified in light of our caching requirements. However, implementing a full-fledged programming language is a large endeavor that has never been our primary goal. More importantly, forcing users who are just trying to use VisTrails to learn an entirely new language was out of the question.

We wanted a system which made it easy for a user to add custom functionality. At the same time, we needed the system to be powerful enough to express fairly complicated pieces of software. As an example, VisTrails supports the VTK visualization library⁵. VTK contains about 1000 classes, which change depending on compilation, configuration, and operating system. Since it seems counterproductive and ultimately hopeless to write different code paths for all these cases, we decided it was necessary to dynamically determine the set of VisTrails modules provided by any given package, and VTK naturally became our model target for a complex package.

Computational science was one of the areas we originally targeted, and at the time we designed the system, Python was becoming popular as "glue code" among these scientists. By specifying the behavior of user-defined VisTrails modules using Python itself, we would all but eliminate a large barrier for adoption. As it turns out, Python offers a nice infrastructure for dynamically-defined classes and reflection. Almost every definition in Python has an equivalent form as a first-class expression. The two important reflection features of Python for our package system are:

- Python classes can be defined dynamically via function calls to the `type` callable. The return value is a representation of a class that can be used in exactly the same way that a typically-defined Python class can.
- Python modules can be imported via function calls to `__import__`, and the resulting value behaves in the same way as the identifier in a standard `import` statement. The path from which these modules come from can also be specified at runtime.

⁵<http://www.vtk.org>

Using Python as our target has a few disadvantages, of course. First of all, this dynamic nature of Python means that while we would like to ensure some things like type safety of VisTrails packages, this is in general not possible. More importantly, some of the requirements for VisTrails modules, notably the ones regarding referential transparency (more on that later) cannot be enforced in Python. Still, we believe that it is worthwhile to restrict the allowed constructs in Python via cultural mechanisms, and with this caveat, Python is an extremely attractive language for software extensibility.

VisTrails Packages and Bundles

A VisTrails package encapsulates a set of modules. Its most common representation in disk is the same representation as a Python package (in a possibly unfortunate naming clash). A Python package consists of a set of Python files which define Python values such as functions and classes. A VisTrails package is a Python package that respects a particular interface. It has files that define specific functions and variables. In its simplest form, a VisTrails package should be a directory containing two files: `__init__.py` and `init.py`.

The first file `__init__.py` is a requirement of Python packages, and should only contain a few definitions which should be constant. Although there is no way to guarantee that this is the case, VisTrails packages failing to obey this are considered buggy. The values defined in the file include a globally unique identifier for the package which is used to distinguish modules when workflows are serialized, and package versions (package versions become important when handling workflow and package upgrades, see Section 23.4). This file can also include functions called `package_dependencies` and `package_requirements`. Since we allow VisTrails modules to subclass from other VisTrails modules beside the root `Module` class, it is conceivable for one VisTrails package to extend the behavior of another, and so one package needs to be initialized before another. These inter-package dependencies are specified by `package_dependencies`. The `package_requirements` function, on the other hand, specifies system-level library requirements which VisTrails, in some cases, can try to automatically satisfy, through its bundle abstraction.

A bundle is a system-level package that VisTrails manages via system-specific tools such as RedHat's RPM or Ubuntu's APT. When these properties are satisfied, VisTrails can determine the package properties by directly importing the Python module and accessing the appropriate variables.

The second file, `init.py`, contains the entry points for all the actual VisTrails module definitions. The most important feature of this file is the definition of two functions, `initialize` and `finalize`. The `initialize` function is called when a package is enabled, after all the dependent packages have themselves been enabled. It performs setup tasks for all of the modules in a package. The `finalize` function, on the other hand, is usually used to release runtime resources (for example, temporary files created by the package can be cleaned up).

Each VisTrails module is represented in a package by one Python class. To register this class in VisTrails, a package developer calls the `add_module` function once for each VisTrails module. These VisTrails modules can be arbitrary Python classes, but they must respect a few requirements. The first of these is that each must be a subclass of a basic Python class defined by VisTrails called, perhaps boringly, `Module`. VisTrails modules can use multiple inheritance, but only one of the classes should be a VisTrails module—no diamond hierarchies in the VisTrails module tree are allowed. Multiple inheritance becomes useful in particular to define class mix-ins: simple behaviors encoded by parent classes which can be composed together to create more complicated behaviors.

The set of available ports determine the interface of a VisTrails module, and so impact not only the display of these modules but also their connectivity to other modules. These ports, then, must be

explicitly described to the VisTrails infrastructure. This can be done either by making appropriate calls to `add_input_port` and `add_output_port` during the call to `initialize`, or by specifying the per-class lists `_input_ports` and `_output_ports` for each VisTrails module.

Each module specifies the computation to be performed by overriding the `compute` method. Data is passed between modules through ports, and accessed through the `get_input_from_port` and `set_result` methods. In traditional dataflow environments, execution order is specified on-demand by the data requests. In our case, the execution order is specified by the topological sorting of the workflow modules. Since the caching algorithm requires an acyclic graph, we schedule the execution in reverse topological sorted order, so the calls to these functions do not trigger executions of upstream modules. We made this decision deliberately: it makes it simpler to consider the behavior of each module separately from all the others, which makes our caching strategy simpler and more robust.

As a general guideline, VisTrails modules should refrain from using functions with side-effects during the evaluation of the `compute` method. As discussed in Section 23.3, this requirement makes caching of partial workflow runs possible: if a module respects this property, then its behavior is a function of the outputs of upstream modules. Every acyclic subgraph then only needs to be computed once, and the results can be reused.

Passing Data as Modules

One peculiar feature of VisTrails modules and their communication is that the data that is passed between VisTrails modules are themselves VisTrails modules. In VisTrails, there is a single hierarchy for module and data classes. For example, a module can provide *itself* as an output of a computation (and, in fact, every module provides a default "self" output port). The main disadvantage is the loss of conceptual separation between computation and data that is sometimes seen in dataflow-based architectures. There are, however, two big advantages. The first is that this closely mimics the object type systems of Java and C++, and the choice was not accidental: it was very important for us to support automatic wrapping of large class libraries such as VTK. These libraries allow objects to produce other objects as computational results, making a wrapping that distinguishes between computation and data more complicated.

The second advantage this decision brings is that defining constant values and user-settable parameters in workflows becomes easier and more uniformly integrated with the rest of the system. Consider, for example, a workflow that loads a file from a location on the Web specified by a constant. This is currently specified by a GUI in which the URL can be specified as a parameter (see the Parameter Edits area in Figure 23.1). A natural modification of this workflow is to use it to fetch a URL that is *computed* somewhere upstream. We would like the rest of the workflow to change as little as possible. By assuming modules can output themselves, we can simply connect a string with the right value to the port corresponding to the parameter. Since the output of a constant evaluates to itself, the behavior is exactly the same as if the value had actually been specified as a constant.

There are other considerations involved in designing constants. Each constant type has a different ideal GUI interface for specifying values. For example, in VisTrails, a file constant module provides a file chooser dialog; a Boolean value is specified by a checkbox; a color value has a color picker native to each operating system. To achieve this generality, a developer must subclass a custom constant from the Constant base class and provide overrides which define an appropriate GUI widget and a string representation (so that arbitrary constants can be serialized to disk).

We note that, for simple prototyping tasks, VisTrails provides a built-in PythonSource module. A PythonSource module can be used to directly insert scripts into a workflow. The configuration

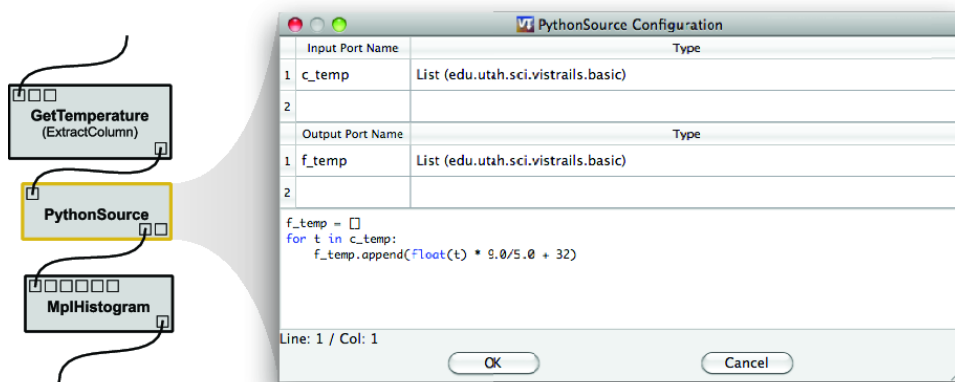


Figure 23.5: Prototyping New Functionality with the PythonSource Module

window for PythonSource (see Figure 23.5) allows multiple input and output ports to be specified along with the Python code that is to be executed.

23.4 Components and Features

As discussed above, VisTrails provides a set of functionalities and user interfaces that simplify the creation and execution of exploratory computational tasks. Below, we describe some of these. We also briefly discuss how VisTrails is being used as the basis for an infrastructure that supports the creation of provenance-rich publications. For a more comprehensive description of VisTrails and its features, see VisTrails' online documentation⁶.

Visual Spreadsheet

VisTrails allows users to explore and compare results from multiple workflows using the Visual Spreadsheet (see Figure 23.6). The spreadsheet is a VisTrails package with its own interface composed of sheets and cells. Each sheet contains a set of cells and has a customizable layout. A cell contains the visual representation of a result produced by a workflow, and can be customized to display diverse types of data.

To display a cell on the spreadsheet, a workflow must contain a module that is derived from the base SpreadsheetCell module. Each SpreadsheetCell module corresponds to a cell in the spreadsheet, so one workflow can generate multiple cells. The compute method of the SpreadsheetCell module handles the communication between the Execution Engine (Figure 23.3) and the spreadsheet. During execution, the spreadsheet creates a cell according to its type on-demand by taking advantage of Python's dynamic class instantiation. Thus, custom visual representations can be achieved by creating a subclass of SpreadsheetCell and having its compute method send a custom cell type to the spreadsheet. For example, the workflow in Figure 23.1, MplFigureCell is a SpreadsheetCell module designed to display images created by matplotlib.

⁶<http://www.vistrails.org/usersguide>

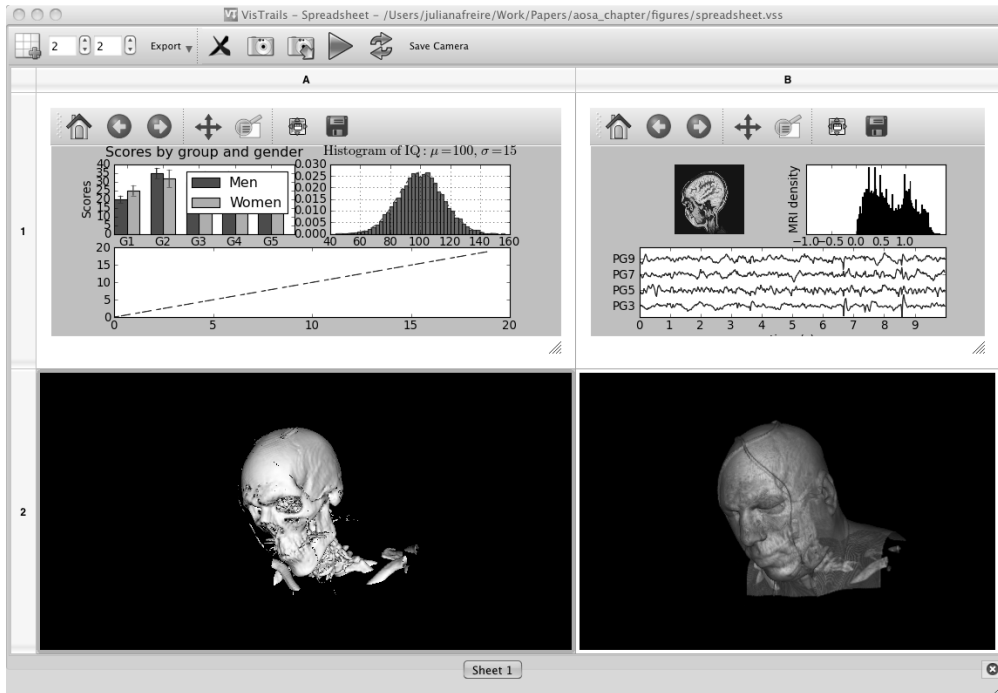


Figure 23.6: The Visual Spreadsheet

Since the spreadsheet uses PyQt as its GUI back end, custom cell widgets must be subclassed from PyQt’s `QWidget`. They must also define the `updateContents` method, which is invoked by the spreadsheet to update the widget when new data arrives. Each cell widget may optionally define a custom toolbar by implementing the `toolbar` method; it will be displayed in the spreadsheet toolbar area when the cell is selected.

Figure 23.6 shows the spreadsheet when a VTK cell is selected, in this case, the toolbar provides specific widgets to export PDF images, save camera positions back to the workflow, and create animations. The spreadsheet package defines a customizable `QCellWidget`, which provides common features such as history replay (animation) and multi-touch events forwarding. This can be used in place of `QWidget` for faster development of new cell types.

Even though the spreadsheet only accepts PyQt widgets as cell types, it is possible to integrate widgets written with other GUI toolkits. To do so, the widget must export its elements to the native platform, and PyQt can then be used to grab it. We use this approach for the `VTKCell` widget because the actual widget is written in C++. At run-time, the `VTKCell` grabs the window id, a `Win32`, `X11`, or `Cocoa/Carbon` handle depending on the system, and maps it to the spreadsheet canvas.

Like cells, sheets may also be customized. By default, each sheet lives in a tabbed view and has a tabular layout. However, any sheet can be undocked from the spreadsheet window, allowing multiple sheets to be visible at once. It is also possible to create a different sheet layout by subclassing the `StandardWidgetSheet`, also a PyQt widget. The `StandardWidgetSheet` manages cell layouts as well as interactions with the spreadsheet in editing mode. In editing mode, users can manipulate the cell layout and perform advanced actions on the cells, rather than interacting with cell contents.

Such actions include applying analogies (see Section 23.4) and creating new workflow versions from parameter explorations.

Visual Differences and Analogies

As we designed VisTrails, we wanted to enable the *use* of provenance information in addition to its capture. First, we wanted users to see the exact differences between versions, but we then realized that a more helpful feature was being able to apply these differences to other workflows. Both of these tasks are possible because VisTrails tracks the evolution of workflows.

Because the version tree captures all of the changes and we can invert each action, we can find a complete sequence of actions that transform one version to another. Note that some changes will cancel each other out, making it possible to compress this sequence. For example, the addition of a module that was later deleted need not be examined when computing the difference. Finally, we have some heuristics to further simplify the sequence: when the same module occurs in both workflows but was added through separate actions, we cancel the adds and deletes.

From the set of changes, we can create a visual representation that shows similar and different modules, connections, and parameters. This is illustrated in Figure 23.4. Modules and connections that appear in both workflows are colored gray, and those appearing in only one are colored according to the workflow they appear in. Matching modules with different parameters are shaded a lighter gray and a user can inspect the parameter differences for a specific module in a table that shows the values in each workflow.

The analogy operation allows users to take these differences and apply them to other workflows. If a user has made a set of changes to an existing workflow (e.g., changing the resolution and file format of an output image), he can apply the same changes to other workflows via an analogy. To do so, the user selects a source and a target workflow, which delimits the set of desired changes, as well as the workflow they wish to apply the analogy to. VisTrails computes the difference between the first two workflows as a template, and then determines how to remap this difference in order to apply it to the third workflow. Because it is possible to apply differences to workflows that do not exactly match the starting workflow, we need a soft matching that allows correspondences between similar modules. With this matching, we can remap the difference so the sequence of changes can be applied to the selected workflow [SVK⁺07]. The method is not foolproof and may generate new workflows that are not exactly what was desired. In such cases, a user may try to fix any introduced mistakes, or go back to the previous version and apply the changes manually.

To compute the soft matching used in analogies, we want to balance local matches (identical or very similar modules) with the overall workflow structure. Note that the computation of even the identical matching is inefficient due to the hardness of subgraph isomorphism, so we need to employ a heuristic. In short, if two somewhat-similar modules in the two workflows share similar neighbors, we might conclude that these two modules function similarly and should be matched as well. More formally, we construct a product graph where each node is a possible pairing of modules in the original workflows and an edge denotes shared connections. Then, we run steps diffusing the scores at each node across the edges to neighboring nodes. This is a Markov process similar to Google's PageRank, and will eventually converge leaving a set of scores that now includes some global information. From these scores, we can determine the best matching, using a threshold to leave very dissimilar modules unpaired.

Querying Provenance

The provenance captured by VisTrails includes a set of workflows, each with its own structure, metadata, and execution logs. It is important that users can access and explore these data. VisTrails provides both text-based and visual (WYSIWYG) query interfaces. For information like tags, annotations, and dates, a user can use keyword search with optional markup. For example, look for all workflows with the keyword `plot` that were created by user `:~dakoop`. However, queries for specific subgraphs of a workflow are more easily represented through a visual, query-by-example interface, where users can either build the query from scratch or copy and modify an existing piece of a pipeline.

In designing this query-by-example interface, we kept most of the code from the existing Workflow Editor, with a few changes to parameter construction. For parameters, it is often useful to search for ranges or keywords rather than exact values. Thus, we added modifiers to the parameter value fields; when a user adds or edits a parameter value, they may choose to select one of these modifiers which default to exact matches. In addition to visual query construction, query results are shown visually. Matching versions are highlighted in the version tree, and any selected workflow is displayed with the matching portion highlighted. The user can exit query results mode by initiating another query or clicking a reset button.

Persistent Data

VisTrails saves the provenance of how results were derived and the specification of each step. However, reproducing a workflow run can be difficult if the data needed by the workflow is no longer available. In addition, for long-running workflows, it may be useful to store intermediate data as a persistent cache across sessions in order to avoid recomputation.

Many workflow systems store filesystem paths to data as provenance, but this approach is problematic. A user might rename a file, move the workflow to another system without copying the data, or change the data contents. In any of these cases, storing the path as provenance is not sufficient. Hashing the data and storing the hash as provenance helps to determine whether the data might have changed, but does not help one locate the data if it exists. To solve this problem, we created the Persistence Package, a VisTrails package that uses version control infrastructure to store data that can be referenced from provenance. Currently we use Git to manage the data, although other systems could easily be employed.

We use universally unique identifiers (UUIDs) to identify data, and commit hashes from git to reference versions. If the data changes from one execution to another, a new version is checked in to the repository. Thus, the `(uuid, version)` tuple is a compound identifier to retrieve the data in any state. In addition, we store the hash of the data as well as the signature of the upstream portion of the workflow that generated it (if it is not an input). This allows one to link data that might be identified differently as well as reuse data when the same computation is run again.

The main concern when designing this package was the way users were able to select and retrieve their data. Also, we wished to keep all data in the same repository, regardless of whether it is used as input, output, or intermediate data (an output of one workflow might be used as the input of another). There are two main modes a user might employ to identify data: choosing to create a new reference or using an existing one. Note that after the first execution, a new reference will become an existing one as it has been persisted during execution; a user may later choose to create another reference if they wish but this is a rare case. Because a user often wishes to always use the latest version of data, a reference identified without a specific version will default to the latest version.

Recall that before executing a module, we recursively update all of its inputs. A persistent data module will not update its inputs if the upstream computations have already been run. To determine this, we check the signature of the upstream subworkflow against the persistent repository and retrieve the precomputed data if the signature exists. In addition, we record the data identifiers and versions as provenance so that a specific execution can be reproduced.

Upgrades

With provenance at the core of VisTrails, the ability to upgrade old workflows so they will run with new versions of packages is a key concern. Because packages can be created by third-parties, we need both the infrastructure for upgrading workflows as well as the hooks for package developers to specify the upgrade paths. The core action involved in workflow upgrades is the replacement of one module with a new version. Note that this action is complicated because we must replace all of the connections and parameters from the old module. In addition, upgrades may need to reconfigure, reassign, or rename these parameters or connections for a module, e.g., when the module interface changes.

Each package (together with its associated modules) is tagged by a version, and if that version changes, we assume that the modules in that package may have changed. Note that some, or even most, may not have changed, but without doing our own code analysis, we cannot check this. We, however, attempt to automatically upgrade any module whose interface has not changed. To do this, we try replacing the module with the new version and throw an exception if it does not work. When developers have changed the interface of a module or renamed a module, we allow them to specify these changes explicitly. To make this more manageable, we have created a `remap_module` method that allows developers to define only the places where the default upgrade behavior needs to be modified. For example, a developer that renamed an input port `'file'` to `'value'` can specify that specific remapping so when the new module is created, any connections to `'file'` in the old module will now connect to `'value'`. Here is an example of an upgrade path for a built-in VisTrails module:

```
def handle_module_upgrade_request(controller, module_id, pipeline):
    module_remap = {'GetItemsFromDirectory':
                    [(None, '1.6', 'Directory',
                      {'dst_port_remap':
                       {'dir': 'value'},
                       'src_port_remap':
                       {'itemlist': 'itemlist'}},
                      )],
                    }
    return UpgradeWorkflowHandler.remap_module(controller, module_id, pipeline,
                                                module_remap)
```

This piece of code upgrades workflows that use the old `GetItemsFromDirectory` (any version up to 1.6) module to use the `Directory` module instead. It maps the `dir` port from the old module to `value` and the `itemlist` port to `itemlist`.

Any upgrade creates a new version in the version tree so that executions before and after upgrades can be differentiated and compared. It is possible that the upgrades change the execution of the workflow (e.g., if a bug is fixed by a package developer), and we need to track this as provenance information. Note that in older `vistrails`, it may be necessary to upgrade every version in the tree. In order to reduce clutter, we only upgrade versions that a user has navigated to. In addition, we provide

a preference that allows a user to delay the persistence of any upgrade until the workflow is modified or executed; if a user just views that version, there is no need to persist the upgrade.

Sharing and Publishing Provenance-Rich Results

While reproducibility is the cornerstone of the scientific method, current publications that describe computational experiments often fail to provide enough information to enable the results to be repeated or generalized. Recently, there has been a renewed interest in the publication of reproducible results. A major roadblock to the more widespread adoption of this practice is the fact that it is hard to create a bundle that includes all of the components (e.g., data, code, parameter settings) needed to reproduce a result as well as verify that result.

By capturing detailed provenance, and through many of the features described above, VisTrails simplifies this process for computational experiments that are carried out within the system. However, mechanisms are needed to both link documents to and share the provenance information.

We have developed VisTrails packages that enable results present in papers to be linked to their provenance, like a deep caption. Using the LaTeX package we developed, users can include figures that link to VisTrails workflows. The following LaTeX code will generate a figure that contains a workflow result:

```
\begin{figure}[t]
\centering{
\vistrail[wfid=119,buildalways=false]{width=0.9\linewidth}
}
\caption{Visualizing a binary star system simulation. This is an image
that was generated by embedding a workflow directly in the text.}
\label{fig:astrophysics}
\end{figure}
```

When the document is compiled using pdf_latex, the `\vistrail` command will invoke a Python script with the parameters received, which sends an XML-RPC message to a VisTrails server to execute the workflow with id 119. This same Python script downloads the results of the workflow from the server and includes them in the resulting PDF document by generating hyperlinked LaTeX `\includegraphics` commands using the specified layout options (`width=0.9\linewidth`).

It is also possible to include VisTrails results into Web pages, wikis, Word documents and PowerPoint presentations. The linking between Microsoft PowerPoint and VisTrails was done through the Component Object Model (COM) and Object Linking and Embedding (OLE) interface. In order for an object to interact with PowerPoint, at least the `IOleObject`, `IDataObject` and `IPersistStorage` interface of COM must be implemented. As we use the `QAxAggregated` class of Qt, which is an abstraction for implementing COM interfaces, to build our OLE object, both `IDataObject` and `IPersistStorage` are automatically handled by Qt. Thus, we only need to implement the `IOleObject` interface. The most important call in this interface is `DoVerb`. It lets VisTrails react to certain actions from PowerPoint, such as object activation. In our implementation, when the VisTrails object is activated, we load the VisTrails application and allow users to open, interact with and select a pipeline that they want to insert. After they close VisTrails, the pipeline result will be shown in PowerPoint. Pipeline information is also stored with the OLE object.

To enable users to freely share their results together with the associated provenance, we have created crowdLabs.⁷ crowdLabs is a social Web site that integrates a set of usable tools and a scalable

⁷<http://www.crowdlabs.org>

infrastructure to provide an environment for scientists to collaboratively analyze and visualize data. crowdLabs is tightly integrated with VisTrails. If a user wants to share any results derived in VisTrails, she can connect to the crowdLabs server directly from VisTrails to upload the information. Once the information is uploaded, users can interact with and execute the workflows through a Web browser—these workflows are executed by a VisTrails server that powers crowdLabs. For more details on how VisTrails is used to create reproducible publications, see <http://www.vistrails.org>.

23.5 Lessons Learned

Luckily, back in 2004 when we started thinking about building a data exploration and visualization system that supported provenance, we never envisioned how challenging it would be, or how long it would take to get to the point we are at now. If we had, we probably would never have started.

Early on, one strategy that worked well was quickly prototyping new features and showing them to a select set of users. The initial feedback and the encouragement we received from these users was instrumental in driving the project forward. It would have been impossible to design VisTrails without user feedback. If there is one aspect of the project that we would like to highlight is that most features in the system were designed as direct response to user feedback. However, it is worthy to note that many times what a user asks for is not the best solution for his/her need—being responsive to users does not necessarily mean doing exactly what they ask for. Time and again, we have had to design and re-design features to make sure they would be useful and properly integrated in the system.

Given our user-centric approach, one might expect that every feature we have developed would be heavily used. Unfortunately this has not been the case. Sometimes the reason for this is that the feature is highly "unusual", since it is not found in other tools. For instance, analogies and even the version tree are not concepts that most users are familiar with, and it takes a while for them to get comfortable with them. Another important issue is documentation, or lack thereof. As with many other open source projects, we have been much better at developing new features than at documenting the existing ones. This lag in documentation leads not only to the underutilization of useful features, but also to many questions on our mailing lists.

One of the challenges of using a system like VisTrails is that it is very general. Despite our best efforts to improve usability, VisTrails is a complex tool and requires a steep learning curve for some users. We believe that over time, with improved documentation, further refinements to the system, and more application- and domain-specific examples, the adoption bar for any given field will get lower. Also, as the concept of provenance becomes more widespread, it will be easier for users to understand the philosophy that we have adopted in developing VisTrails.

Acknowledgments

We would like to thank all the talented developers that contributed to VisTrails: Erik Anderson, Louis Bavoil, Clifton Brooks, Jason Callahan, Steve Callahan, Lorena Carlo, Lauro Lins, Tommy Ellkvist, Phillip Mates, Daniel Rees, and Nathan Smith. Special thanks to Antonio Baptista who was instrumental in helping us develop the vision for the project; and Matthias Troyer, whose collaboration has helped us to improve the system, and in particular has provided much of the impetus for the development and release of the provenance-rich publication functionality. The research and development of the VisTrails system has been funded by the National Science Foundation under grants IIS 1050422, IIS-0905385, IIS 0844572, ATM-0835821, IIS-0844546, IIS-0746500, CNS-

0751152, IIS-0713637, OCE-0424602, IIS-0534628, CNS-0514485, IIS-0513692, CNS-0524096, CCF-0401498, OISE-0405402, CCF-0528201, CNS-0551724, the Department of Energy SciDAC (VACET and SDM centers), and IBM Faculty Awards.

VTK

Berk Geveci and Will Schroeder

The Visualization Toolkit (VTK) is a widely used software system for data processing and visualization. It is used in scientific computing, medical image analysis, computational geometry, rendering, image processing and informatics. In this chapter we provide a brief overview of VTK, including some of the basic design patterns that make it a successful system.

To really understand a software system it is essential to not only understand what problem it solves, but also the particular culture in which it emerged. In the case of VTK, the software was ostensibly developed as a 3D visualization system for scientific data. But the cultural context in which it emerged adds a significant back story to the endeavor, and helps explain why the software was designed and deployed as it was.

At the time VTK was conceived and written, its initial authors (Will Schroeder, Ken Martin, Bill Lorensen) were researchers at GE Corporate R&D. We were heavily invested in a precursor system known as LYMB which was a Smalltalk-like environment implemented in the C programming language. While this was a great system for its time, as researchers we were consistently frustrated by two major barriers when trying to promote our work: 1) IP issues and 2) non-standard, proprietary software. IP issues were a problem because trying to distribute the software outside of GE was nearly impossible once the corporate lawyers became involved. Second, even if we were deploying the software inside of GE, many of our customers balked at learning a proprietary, non-standard system since the effort to master it did not transition with an employee once she left the company, and it did not have the widespread support of a standard tool set. Thus in the end the primary motivation for VTK was to develop an open standard, or *collaboration platform* through which we could easily transition technology to our customers. Thus choosing an open source license for VTK was probably the most important design decision that we made.

The final choice of a non-reciprocal, permissive license (i.e., BSD not GPL) in hindsight was an exemplary decision made by the authors because it ultimately enabled the service and consulting based business that became Kitware. At the time we made the decision we were mostly interested in reduced barriers to collaborating with academics, research labs, and commercial entities. We have since discovered that reciprocal licenses are avoided by many organizations because of the potential havoc they can wreak. In fact we would argue that reciprocal licenses do much to slow the acceptance of open source software, but that is an argument for another time. The point here is: one of the major design decisions to make relative to any software system is the choice of copyright license. It's important to review the goals of the project and then address IP issues appropriately.

24.1 What Is VTK?

VTK was initially conceived as a scientific data visualization system. Many people outside of the field naively consider visualization a particular type of geometric rendering: examining virtual objects and interacting with them. While this is indeed part of visualization, in general data visualization includes the whole process of transforming data into sensory input, typically images, but also includes tactile, auditory, and other forms. The data forms not only consist of geometric and topological constructs, including such abstractions as meshes or complex spatial decompositions, but attributes to the core structure such as scalars (e.g., temperature or pressure), vectors (e.g., velocity), tensors (e.g., stress and strain) plus rendering attributes such as surface normals and texture coordinate.

Note that data representing spatial-temporal information is generally considered part of scientific visualization. However there are more abstract data forms such as marketing demographics, web pages, documents and other information that can only be represented through abstract (i.e., non-spatial temporal) relationships such as unstructured documents, tables, graphs, and trees. These abstract data are typically addressed by methods from information visualization. With the help of the community, VTK is now capable of both scientific and information visualization.

As a visualization system, the role of VTK is to take data in these forms and ultimately transform them into forms comprehensible by the human sensory apparatus. Thus one of the core requirements of VTK is its ability to create data flow pipelines that are capable of ingesting, processing, representing and ultimately rendering data. Hence the toolkit is necessarily architected as a flexible system and its design reflects this on many levels. For example, we purposely designed VTK as a toolkit with many interchangeable components that can be combined to process a wide variety of data.

24.2 Architectural Features

Before getting too far into the specific architectural features of VTK, there are high-level concepts that have significant impact on developing and using the system. One of these is VTK's hybrid wrapper facility. This facility automatically generates language bindings to Python, Java, and Tcl from VTK's C++ implementation (additional languages could be and have been added). Most high-powered developers will work in C++. User and application developers may use C++ but often the interpreted languages mentioned above are preferred. This hybrid compiled/interpreted environment combines the best of both worlds: high performance compute-intensive algorithms and flexibility when prototyping or developing applications. In fact this approach to multi-language computing has found favor with many in the scientific computing community and they often use VTK as a template for developing their own software.

In terms of software process, VTK has adopted CMake to control the build; CDash/CTest for testing; and CPack for cross-platform deployment. Indeed VTK can be compiled on almost any computer including supercomputers which are often notoriously primitive development environments. In addition, web pages, wiki, mailing lists (user and developer), documentation generation facilities (i.e., Doxygen) and a bug tracker (Mantis) round out the development tools.

Core Features

As VTK is an object-oriented system, the access of class and instance data members is carefully controlled in VTK. In general, all data members are either protected or private. Access to them is through Set and Get methods, with special variations for Boolean data, modal data, strings and

vectors. Many of these methods are actually created by inserting macros into the class header files. So for example:

```
vtkSetMacro(Tolerance,double);  
vtkGetMacro(Tolerance,double);
```

become on expansion:

```
virtual void SetTolerance(double);  
virtual double GetTolerance();
```

There are many reasons for using these macros beyond simply code clarity. In VTK there are important data members controlling debugging, updating an object's modified time (MTime), and properly managing reference counting. These macros correctly manipulate these data and their use is highly recommended. For example, a particularly pernicious bug in VTK occurs when the object's MTime is not managed properly. In this case code may not execute when it should, or may execute too often.

One of the strengths of VTK is its relatively simplistic means of representing and managing data. Typically various data arrays of particular types (e.g., `vtkFloatArray`) are used to represent contiguous pieces of information. For example, a list of three XYZ points would be represented with a `vtkFloatArray` of nine entries (x,y,z, x,y,z, etc.) There is the notion of a tuple in these arrays, so a 3D point is a 3-tuple, whereas a symmetric 3×3 tensor matrix is represented by a 6-tuple (where symmetry space savings are possible). This design was adopted purposely because in scientific computing it is common to interface with systems manipulating arrays (e.g., Fortran) and it is much more efficient to allocate and deallocate memory in large contiguous chunks. Further, communication, serializing and performing IO is generally much more efficient with contiguous data. These core data arrays (of various types) represent much of the data in VTK and have a variety of convenience methods for inserting and accessing information, including methods for fast access, and methods that automatically allocate memory as needed when adding more data. Data arrays are subclasses of the `vtkDataArray` abstract class meaning that generic virtual methods can be used to simplify coding. However, for higher performance static, templated functions are used which switch based on type, with subsequent, direct access into the contiguous data arrays.

In general C++ templates are not visible in the public class API; although templates are used widely for performance reasons. This goes for STL as well: we typically employ the PIMPL¹ design pattern to hide the complexities of a template implementation from the user or application developer. This has served us particularly well when it comes to wrapping the code into interpreted code as described previously. Avoiding the complexity of the templates in the public API means that the VTK implementation, from the application developer point of view, is mostly free of the complexities of data type selection. Of course under the hood the code execution is driven by the data type which is typically determined at run time when the data is accessed.

Some users wonder why VTK uses reference counting for memory management versus a more user-friendly approach such as garbage collection. The basic answer is that VTK needs complete control over when data is deleted, because the data sizes can be huge. For example, a volume of byte data 1000×1000×1000 in size is a gigabyte in size. It is not a good idea to leave such data lying around while the garbage collector decides whether or not it is time to release it. In VTK most classes (subclasses of `vtkObject`) have the built-in capability for reference counting. Every object contains a reference count that is initialized to one when the object is instantiated. Every time a

¹http://en.wikipedia.org/wiki/Opaque_pointer.

use of the object is registered, the reference count is increased by one. Similarly, when a use of the object is unregistered (or equivalently the object is deleted) the reference count is reduced by one. Eventually the object's reference count is reduced to zero, at which point it self destructs. A typical example looks like the following:

```
vtkCamera *camera = vtkCamera::New();    //reference count is 1
camera->Register(this);                  //reference count is 2
camera->Unregister(this);                 //reference count is 1
renderer->SetActiveCamera(camera);        //reference count is 2
renderer->Delete();                       //ref count is 1 when renderer is deleted
camera->Delete();                         //camera self destructs
```

There is another important reason why reference counting is important to VTK—it provides the ability to efficiently copy data. For example, imagine a data object D1 that consists of a number of data arrays: points, polygons, colors, scalars and texture coordinates. Now imagine processing this data to generate a new data object D2 which is the same as the first plus the addition of vector data (located on the points). One wasteful approach is to completely (deep) copy D1 to create D2, and then add the new vector data array to D2. Alternatively, we create an empty D2 and then pass the arrays from D1 to D2 (shallow copy), using reference counting to keep track of data ownership, finally adding the new vector array to D2. The latter approach avoids copying data which, as we have argued previously, is essential to a good visualization system. As we will see later in this chapter, the data processing pipeline performs this type of operation routinely, i.e., copying data from the input of an algorithm to the output, hence reference counting is essential to VTK.

Of course there are some notorious problems with reference counting. Occasionally reference cycles can exist, with objects in the cycle referring to each other in a mutually supportive configuration. In this case, intelligent intervention is required, or in the case of VTK, the special facility implemented in `vtkGarbageCollector` is used to manage objects which are involved in cycles. When such a class is identified (this is anticipated during development), the class registers itself with the garbage collector and overloads its own `Register` and `UnRegister` methods. Then a subsequent object deletion (or unregister) method performs a topological analysis on the local reference counting network, searching for detached islands of mutually referencing objects. These are then deleted by the garbage collector.

Most instantiation in VTK is performed through an object factory implemented as a static class member. The typical syntax appears as follows:

```
vtkLight *a = vtkLight::New();
```

What is important to recognize here is what is actually instantiated may not be a `vtkLight`, it could be a subclass of `vtkLight` (e.g., `vtkOpenGLLight`). There are a variety of motivations for the object factory, the most important being application portability and device independence. For example, in the above we are creating a light in a rendered scene. In a particular application on a particular platform, `vtkLight::New` may result in an OpenGL light, however on different platforms there is potential for other rendering libraries or methods for creating a light in the graphics system. Exactly what derived class to instantiate is a function of run-time system information. In the early days of VTK there were a myriad of options including `gl`, `PHIGS`, `Starbase`, `XGL`, and `OpenGL`. While most of these have now vanished, new approaches have appeared including `DirectX` and GPU-based approaches. Over time, an application written with VTK has not had to change as developers have derived new device specific subclasses to `vtkLight` and other rendering classes to support evolving technology. Another important use of the object factory is to enable the run-time

replacement of performance-enhanced variations. For example, a `vtkImageFFT` may be replaced with a class that accesses special-purpose hardware or a numerics library.

Representing Data

One of the strengths of VTK is its ability to represent complex forms of data. These data forms range from simple tables to complex structures such as finite element meshes. All of these data forms are subclasses of `vtkDataObject` as shown in Figure 24.1 (note this is a partial inheritance diagram of the many data object classes).

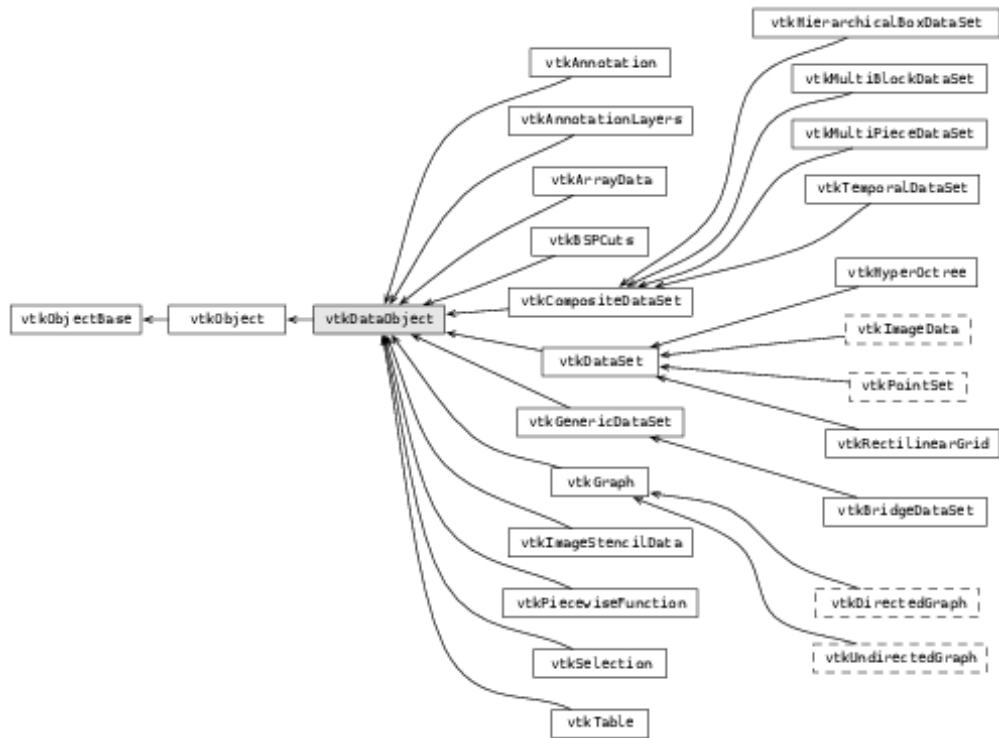


Figure 24.1: Data Object Classes

One of the most important characteristics of `vtkDataObject` is that it can be processed in a visualization pipeline (next subsection). Of the many classes shown, there are just a handful that are typically used in most real world applications. `vtkDataSet` and derived classes are used for scientific visualization (Figure 24.2). For example, `vtkPolyData` is used to represent polygonal meshes; `vtkUnstructuredGrid` to represent meshes, and `vtkImageData` represents 2D and 3D pixel and voxel data.

Pipeline Architecture

VTK consists of several major subsystems. Probably the subsystem most associated with visualization packages is the data flow/pipeline architecture. In concept, the pipeline architecture consists of three

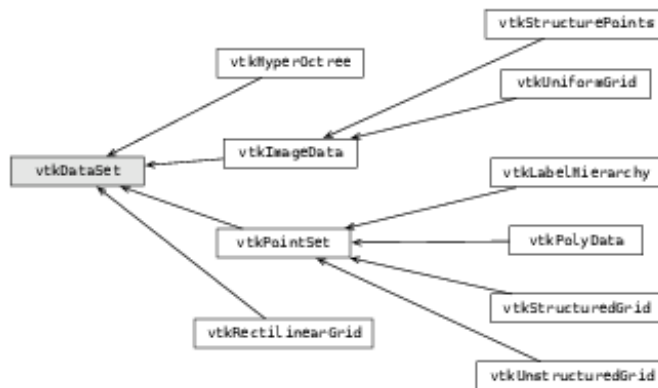


Figure 24.2: Data Set Classes

basic classes of objects: objects to represent data (the `vtkDataObject`s discussed above), objects to process, transform, filter or map data objects from one form into another (`vtkAlgorithm`); and objects to execute a pipeline (`vtkExecutive`) which controls a connected graph of interleaved data and process objects (i.e., the pipeline). Figure 24.3 depicts a typical pipeline.

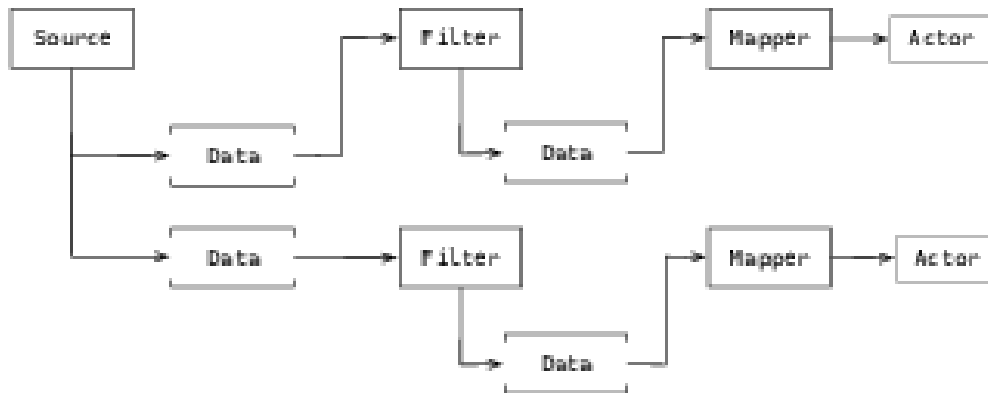


Figure 24.3: Typical Pipeline

While conceptually simple, actually implementing the pipeline architecture is challenging. One reason is that the representation of data can be complex. For example, some datasets consist of hierarchies or grouping of data, so executing across the data requires non-trivial iteration or recursion. To compound matters, parallel processing (whether using shared-memory or scalable, distributed approaches) require partitioning data into pieces, where pieces may be required to overlap in order to consistently compute boundary information such as derivatives.

The algorithm objects also introduce their own special complexity. Some algorithms may take multiple inputs and/or produce multiple outputs of different types. Some can operate locally on data (e.g., compute the center of a cell) while others require global information, for example to compute a histogram. In all cases, the algorithms treat their inputs as immutable, algorithms only read their input in order to produce their output. This is because data may be available as input to multiple

algorithms, and it is not a good idea for one algorithm to trample on the input of another.

Finally the executive can be complicated depending on the particulars of the execution strategy. In some cases we may wish to cache intermediate results between filters. This minimizes the amount of recomputation that must be performed if something in the pipeline changes. On the other hand, visualization data sets can be huge, in which case we may wish to release data when it is no longer needed for computation. Finally, there are complex execution strategies, such as multi-resolution processing of data, which require the pipeline to operate in iterative fashion.

To demonstrate some of these concepts and further explain the pipeline architecture, consider the following C++ example:

```
vtkPExodusIIReader *reader = vtkPExodusIIReader::New();
reader->SetFileName("exampleFile.exo");

vtkContourFilter *cont = vtkContourFilter::New();
cont->SetInputConnection(reader->GetOutputPort());
cont->SetNumberOfContours(1);
cont->SetValue(0, 200);

vtkQuadricDecimation *deci = vtkQuadricDecimation::New();
deci->SetInputConnection(cont->GetOutputPort());
deci->SetTargetReduction( 0.75 );

vtkXMLPolyDataWriter *writer = vtkXMLPolyDataWriter::New();
writer->SetInputConnection(deci->GetOutputPort());
writer->SetFileName("outputFile.vtp");
writer->Write();
```

In this example, a reader object reads a large unstructured grid (or mesh) data file. The next filter generates an isosurface from the mesh. The `vtkQuadricDecimation` filter reduces the size of the isosurface, which is a polygonal dataset, by decimating it (i.e., reducing the number of triangles representing the isocontour). Finally after decimation the new, reduced data file is written back to disk. The actual pipeline execution occurs when the `Write` method is invoked by the writer (i.e., upon demand for the data).

As this example demonstrates, VTK's pipeline execution mechanism is demand driven. When a sink such as a writer or a mapper (a data rendering object) needs data, it asks its input. If the input filter already has the appropriate data, it simply returns the execution control to the sink. However, if the input does not have the appropriate data, it needs to compute it. Consequently, it must first ask its input for data. This process will continue upstream along the pipeline until a filter or source that has "appropriate data" or the beginning of the pipeline is reached, at which point the filters will execute in correct order and the data will flow to the point in the pipeline at which it was requested.

Here we should expand on what "appropriate data" means. By default, after a VTK source or filter executes, its output is cached by the pipeline in order to avoid unnecessary executions in the future. This is done to minimize computation and/or I/O at the cost of memory, and is configurable behavior. The pipeline caches not only the data objects but also the metadata about the conditions under which these data objects were generated. This metadata includes a time stamp (i.e., `ComputeTime`) that captures when the data object was computed. So in the simplest case, the "appropriate data" is one that was computed after all of the pipeline objects upstream from it were modified. It is easier to demonstrate this behavior by considering the following examples. Let's add the following to the end of the previous VTK program:

```

vtkXMLPolyDataWriter *writer2 = vtkXMLPolyDataWriter::New();
writer2->SetInputConnection(deci->GetOutputPort());
writer2->SetFileName("outputFile2.vtp");
writer2->Write();

```

As explained previously, the first `writer->Write` call causes the execution of the entire pipeline. When `writer2->Write()` is called, the pipeline will realize that the cached output of the decimation filter is up to date when it compares the time stamp of the cache with the modification time of the decimation filter, the contour filter and the reader. Therefore, the data request does not have to propagate past `writer2`. Now, let's consider the following change.

```

cont->SetValue(0, 400);

vtkXMLPolyDataWriter *writer2 = vtkXMLPolyDataWriter::New();
writer2->SetInputConnection(deci->GetOutputPort());
writer2->SetFileName("outputFile2.vtp");
writer2->Write();

```

Now the pipeline executive will realize that the contour filter was modified after the outputs of the contour and decimation filters were last executed. Thus, the cache for these two filters are stale and they have to be re-executed. However, since the reader was not modified prior to the contour filter its cache is valid and hence the reader does not have to re-execute.

The scenario described here is the simplest example of a demand-driven pipeline. VTK's pipeline is much more sophisticated. When a filter or a sink requires data, it can provide additional information to request specific data subsets. For example, a filter can perform out-of-core analysis by streaming pieces of data. Let's change our previous example to demonstrate.

```

vtkXMLPolyDataWriter *writer = vtkXMLPolyDataWriter::New();
writer->SetInputConnection(deci->GetOutputPort());
writer->SetNumberOfPieces(2);

writer->SetWritePiece(0);
writer->SetFileName("outputFile0.vtp");
writer->Write();

writer->SetWritePiece(1);
writer->SetFileName("outputFile1.vtp");
writer->Write();

```

Here the writer asks the upstream pipeline to load and process data in two pieces each of which are streamed independently. You may have noticed that the simple execution logic described previously will not work here. By this logic when the `Write` function is called for the second time, the pipeline should not re-execute because nothing upstream changed. Thus to address this more complex case, the executives have additional logic to handle piece requests such as this. VTK's pipeline execution actually consists of multiple passes. The computation of the data objects is actually the last pass. The pass before then is a request pass. This is where sinks and filters can tell upstream what they want from the forthcoming computation. In the example above, the writer will notify its input that it wants piece 0 of 2. This request will actually propagate all the way to the reader. When the pipeline executes, the reader will then know that it needs to read a subset of the data. Furthermore, information about which piece the cached data corresponds to is stored in the metadata for the object.

The next time a filter asks for data from its input, this metadata will be compared with the current request. Thus in this example the pipeline will re-execute in order to process a different piece request.

There are several more types of request that a filter can make. These include requests for a particular time step, a particular structured extent or the number of ghost layers (i.e., boundary layers for computing neighborhood information). Furthermore, during the request pass, each filter is allowed to modify requests from downstream. For example, a filter that is not able to stream (e.g., the streamline filter) can ignore the piece request and ask for the whole data.

Rendering Subsystem

At first glance VTK has a simple object-oriented rendering model with classes corresponding to the components that make up a 3D scene. For example, `vtkActors` are objects that are rendered by a `vtkRenderer` in conjunction with a `vtkCamera`, with possibly multiple `vtkRenderers` existing in a `vtkRenderWindow`. The scene is illuminated by one or more `vtkLights`. The position of each `vtkActor` is controlled by a `vtkTransform`, and the appearance of an actor is specified through a `vtkProperty`. Finally, the geometric representation of an actor is defined by a `vtkMapper`. Mappers play an important role in VTK, they serve to terminate the data processing pipeline, as well as interface to the rendering system. Consider this example where we decimate data and write the result to a file, and then visualize and interact with the result by using a mapper:

```
vtkOBJReader *reader = vtkOBJReader::New();
reader->SetFileName("exampleFile.obj");

vtkTriangleFilter *tri = vtkTriangleFilter::New();
tri->SetInputConnection(reader->GetOutputPort());

vtkQuadricDecimation *deci = vtkQuadricDecimation::New();
deci->SetInputConnection(tri->GetOutputPort());
deci->SetTargetReduction( 0.75 );

vtkPolyDataMapper *mapper = vtkPolyDataMapper::New();
mapper->SetInputConnection(deci->GetOutputPort());

vtkActor *actor = vtkActor::New();
actor->SetMapper(mapper);

vtkRenderer *renderer = vtkRenderer::New();
renderer->AddActor(actor);

vtkRenderWindow *renWin = vtkRenderWindow::New();
renWin->AddRenderer(renderer);

vtkRenderWindowInteractor *interactor = vtkRenderWindowInteractor::New();
interactor->SetRenderWindow(renWin);

renWin->Render();
```

Here a single actor, renderer and render window are created with the addition of a mapper that connects the pipeline to the rendering system. Also note the addition of a `vtkRenderWindowInteractor`, instances of which capture mouse and keyboard events and translate them into camera manipulations

or other actions. This translation process is defined via a `vtkInteractorStyle` (more on this below). By default many instances and data values are set behind the scenes. For example, an identity transform is constructed, as well as a single default (head) light and property.

Over time this object model has become more sophisticated. Much of the complexity has come from developing derived classes that specialize on an aspect of the rendering process. `vtkActors` are now specializations of `vtkProp` (like a prop found on stage), and there are a whole slew of these props for rendering 2D overlay graphics and text, specialized 3D objects, and even for supporting advanced rendering techniques such as volume rendering or GPU implementations (see Figure 24.4).

Similarly, as the data model supported by VTK has grown, so have the various mappers that interface the data to the rendering system. Another area of significant extension is the transformation hierarchy. What was originally a simple linear 4×4 transformation matrix, has become a powerful hierarchy that supports non-linear transformations including thin-plate spline transformation. For example, the original `vtkPolyDataMapper` had device-specific subclasses (e.g., `vtkOpenGLPolyDataMapper`). In recent years it has been replaced with a sophisticated graphics pipeline referred to as the “painter” pipeline illustrated in Figure 24.4.



Figure 24.4: Display Classes

The painter design supports a variety of techniques for rendering data that can be combined to provide special rendering effects. This capability greatly surpasses the simple `vtkPolyDataMapper` that was initially implemented in 1994.

Another important aspect of a visualization system is the selection subsystem. In VTK there is a hierarchy of “pickers”, roughly categorized into objects that select `vtkProps` based on hardware-based methods versus software methods (e.g., ray-casting); as well as objects that provide different levels of information after a pick operations. For example, some pickers provide only a location in XYZ world space without indicating which `vtkProp` they have selected; others provide not only the

selected `vtkProp` but a particular point or cell that make up the mesh defining the prop geometry.

Events and Interaction

Interacting with data is an essential part of visualization. In VTK this occurs in a variety of ways. At its simplest level, users can observe events and respond appropriately through commands (the command/observer design pattern). All subclasses of `vtkObject` maintain a list of observers which register themselves with the object. During registration, the observers indicate which particular event(s) they are interested in, with the addition of an associated command that is invoked if and when the event occurs. To see how this works, consider the following example in which a filter (here a polygon decimation filter) has an observer which watches for the three events `StartEvent`, `ProgressEvent`, and `EndEvent`. These events are invoked when the filter begins to execute, periodically during execution, and then on completion of execution. In the following the `vtkCommand` class has an `Execute` method that prints out the appropriate information relative to the time it takes to execute the algorithm:

```
class vtkProgressCommand : public vtkCommand
{
public:
    static vtkProgressCommand *New() { return new vtkProgressCommand; }
    virtual void Execute(vtkObject *caller, unsigned long, void *callData)
    {
        double progress = *(static_cast<double*>(callData));
        std::cout << "Progress at " << progress<< std::endl;
    }
};

vtkCommand* pobserver = vtkProgressCommand::New();

vtkDecimatePro *deci = vtkDecimatePro::New();
deci->SetInputConnection( byu->GetOutputPort() );
deci->SetTargetReduction( 0.75 );
deci->AddObserver( vtkCommand::ProgressEvent, pobserver );
```

While this is a primitive form of interaction, it is a foundational element to many applications that use VTK. For example, the simple code above can be easily converted to display and manage a GUI progress bar. This Command/Observer subsystem is also central to the 3D widgets in VTK, which are sophisticated interaction objects for querying, manipulating and editing data and are described below.

Referring to the example above, it is important to note that events in VTK are predefined, but there is a back door for user-defined events. The class `vtkCommand` defines the set of enumerated events (e.g., `vtkCommand::ProgressEvent` in the above example) as well as a user event. The `UserEvent`, which is simply an integral value, is typically used as a starting offset value into a set of application user-defined events. So for example `vtkCommand::UserEvent+100` may refer to a specific event outside the set of VTK defined events.

From the user's perspective, a VTK widget appears as an actor in a scene except that the user can interact with it by manipulating handles or other geometric features (the handle manipulation and geometric feature manipulation is based on the picking functionality described earlier.) The interaction with this widget is fairly intuitive: a user grabs the spherical handles and moves them, or

Common	core VTK classes
Filtering	classes used to manage pipeline dataflow
Rendering	rendering, picking, image viewing, and interaction
VolumeRendering	volume rendering techniques
Graphics	3D geometry processing
GenericFiltering	non-linear 3D geometry processing
Imaging	imaging pipeline
Hybrid	classes requiring both graphics and imaging functionality
Widgets	sophisticated interaction
IO	VTK input and output
Infovis	information visualization
Parallel	parallel processing (controllers and communicators)
Wrapping	support for Tcl, Python, and Java wrapping
Examples	extensive, well-documented examples

Table 24.1: VTK Subdirectories

grabs the line and moves it. Behind the scenes, however, events are emitted (e.g., `InteractionEvent`) and a properly programmed application can observe these events, and then take the appropriate action. For example they often trigger on the `vtkCommand::InteractionEvent` as follows:

```

vtkLW2Callback *myCallback = vtkLW2Callback::New();
myCallback->PolyData = seeds;    // streamlines seed points, updated on interaction
myCallback->Actor = streamline;  // streamline actor, made visible on interaction

vtkLineWidget2 *lineWidget = vtkLineWidget2::New();
lineWidget->SetInteractor(iren);
lineWidget->SetRepresentation(rep);
lineWidget->AddObserver(vtkCommand::InteractionEvent, myCallback);

```

VTK widgets are actually constructed using two objects: a subclass of `vtkInteractorObserver` and a subclass of `vtkProp`. The `vtkInteractorObserver` simply observes user interaction in the render window (i.e., mouse and keyboard events) and processes them. The subclasses of `vtkProp` (i.e., actors) are simply manipulated by the `vtkInteractorObserver`. Typically such manipulation consists of modifying the `vtkProp`'s geometry including highlighting handles, changing cursor appearance, and/or transforming data. Of course, the particulars of the widgets require that subclasses are written to control the nuances of widget behavior, and there are more than 50 different widgets currently in the system.

Summary of Libraries

VTK is a large software toolkit. Currently the system consists of approximately 1.5 million lines of code (including comments but not including automatically generated wrapper software), and approximately 1000 C++ classes. To manage the complexity of the system and reduce build and link times the system has been partitioned into dozens of subdirectories. Table 24.1 lists these subdirectories, with a brief summary describing what capabilities the library provides.

24.3 Looking Back/Looking Forward

VTK has been an enormously successful system. While the first line of code was written in 1993, at the time of this writing VTK is still growing strong and if anything the pace of development is increasing.² In this section we talk about some lessons learned and future challenges.

Managing Growth

One of the most surprising aspects to the VTK adventure has been the project's longevity. The pace of development is due to several major reasons:

- New algorithms and capabilities continue to be added. For example, the informatics subsystem (Titan, primarily developed by Sandia National Labs and Kitware) is a recent significant addition. Additional charting and rendering classes are also being added, as well as capabilities for new scientific dataset types. Another important addition were the 3D interaction widgets. Finally, the on-going evolution of GPU-based rendering and data processing is driving new capabilities in VTK.
- The growing exposure and use of VTK is a self-perpetuating process that adds even more users and developers to the community. For example, ParaView is the most popular scientific visualization application built on VTK and is highly regarded in the high-performance computing community. 3D Slicer is a major biomedical computing platform that is largely built on VTK and received millions of dollars per year in funding.
- VTK's development process continues to evolve. In recent years the software process tools CMake, CDash, CTest, and CPack have been integrated into the VTK build environment. More recently, the VTK code repository has moved to Git and a more sophisticated work flow. These improvements ensure that VTK remains on the leading edge of software development in the scientific computing community.

While growth is exciting, validates the creation of the software system, and bodes well for the future of VTK, it can be extremely difficult to manage well. As a result, the near term future of VTK focuses more on managing the growth of the community as well as the software. Several steps have been taken in this regard.

First, formalized management structures are being created. An Architecture Review Board has been created to guide the development of the community and technology, focusing on high-level, strategic issues. The VTK community is also establishing a recognized team of Topic Leads to guide the technical development of particular VTK subsystems.

Next, there are plans to modularize the toolkit further, partially in response to workflow capabilities introduced by git, but also to recognize that users and developers typically want to work with small subsystems of the toolkit, and do not want to build and link against the entire package. Further, to support the growing community, it's important that contributions of new functionality and subsystems are supported, even if they are not necessarily part of the core of the toolkit. By creating a loose, modularized collection of modules it is possible to accommodate the large number of contributions on the periphery while maintaining core stability.

²See the latest VTK code analysis at <http://www.ohloh.net/p/vtk/analyses/latest>.

Technology Additions

Besides the software process, there are many technological innovations in the development pipeline.

- Co-processing is a capability where the visualization engine is integrated into the simulation code, and periodically generates data extracts for visualization. This technology greatly reduces the need to output large amounts of complete solution data.
- The data processing pipeline in VTK is still too complex. Methods are under way to simplify and refactor this subsystem.
- The ability to directly interact with data is increasingly popular with users. While VTK has a large suite of widgets, many more interaction techniques are emerging including touch-screen-based and 3D methods. Interaction will continue its development at a rapid pace.
- Computational chemistry is increasing in importance to materials designers and engineers. The ability to visualize and interact with chemistry data is being added to VTK.
- The rendering system in VTK has been criticized for being too complex, making it difficult to derive new classes or support new rendering technology. In addition, VTK does not directly support the notion of a scene graph, again something that many users have requested.
- Finally new forms of data are constantly emerging. For example, in the medical field hierarchical volumetric datasets of varying resolution (e.g., confocal microscopy with local magnification).

Open Science

Finally Kitware and more generally the VTK community are committed to Open Science. Pragmatically this is a way of saying we will promulgate open data, open publication, and open source—the features necessary to ensure that we are creating reproducible scientific systems. While VTK has long been distributed as an open source and open data system, the documentation process has been lacking. While there are decent books [Kit10, SML06] there have been a variety of ad hoc ways to collect technical publications including new source code contributions. We are improving the situation by developing new publishing mechanisms like the *VTK Journal*³ that enable articles consisting of documentation, source code, data, and valid test images. The journal also enables automated reviews of the code (using VTK’s quality software testing process) as well as human reviews of the submission.

Lessons Learned

While VTK has been successful there are many things we didn’t do right:

Design Modularity: We did a good job choosing the modularity of our classes. For example, we didn’t do something as silly as creating an object per pixel, rather we created the higher-level `vtkImageClass` that under the hood treats data arrays of pixel data. However in some cases we made our classes too high level and too complex, in many instances we’ve had to refactor them into smaller pieces, and are continuing this process. One prime example is the data processing pipeline. Initially, the pipeline was implemented implicitly through interaction of the data and algorithm objects. We eventually realized that we had to create an explicit pipeline executive object to coordinate the interaction between data and algorithms, and to implement different data processing strategies.

³<http://www.midasjournal.org/?journal=35>

Missed Key Concepts: One of our biggest regrets is not making widespread use of C++ iterators. In many cases the traversal of data in VTK is akin to the scientific programming language Fortran. The additional flexibility of iterators would have been a significant benefit to the system. For example, it is very advantageous to process a local region of data, or only data satisfying some iteration criterion.

Design Issues: Of course there is a long list of design decisions that are not optimal. We have struggled with the data execution pipeline, having gone through multiple generations each time making the design better. The rendering system too is complex and hard to derive from. Another challenge resulted from the initial conception of VTK: we saw it as a read-only visualization system for viewing data. However, current customers often want it to be capable of editing data, which requires significantly different data structures.

One of the great things about an open source system like VTK is that many of these mistakes can and will be rectified over time. We have an active, capable development community that is improving the system every day and we expect this to continue into the foreseeable future.

Battle for Wesnoth

Richard Shimooka and David White

Programming tends to be considered a straightforward problem solving activity; a developer has a requirement and codes a solution. Beauty is often judged on the technical implementation's elegance or effectiveness; this book is replete with excellent examples. Yet beyond its immediate computing functions, code can have a profound effect on people's lives. It can inspire people to participate and create new content. Unfortunately, serious barriers exist that prevent individuals from participating in a project.

Most programming languages require significant technical expertise to utilize, which is out of reach for many. In addition, enhancing the accessibility of code is technically difficult and is not necessary for many programs. It rarely translates into neat coding scripts or clever programming solutions. Achieving accessibility requires considerable forethought in project and program design, which often runs counter-intuitive to normal programming standards. Moreover most projects rely upon an established staff of skilled professionals that are expected to operate at a reasonably high level. They do not require additional programming resources. Thus, code accessibility becomes an afterthought, if considered at all.

Our project, the Battle for Wesnoth, attempted to address this issue from its origins. The program is a turn-based fantasy strategy game, produced in an open source model based on a GPL2 license. It has been a moderate success, with over four million downloads at the time of this writing. While this is an impressive metric, we believe the real beauty of our project is the development model that allowed a band of volunteers from widely different skill levels to interact in a productive way.

Enhancing accessibility was not a vague objective set by developers, it was viewed as essential for the project's survival. Wesnoth's open source approach meant that the project could not immediately expect large numbers of highly skilled developers. Making the project accessible to a wide a number of contributors, with varying skill levels, would ensure its long-term viability.

Our developers attempted to lay the foundations for broadening accessibility right from its earliest iteration. This would have undeniable consequences for all aspect of the programming architecture. Major decisions were made largely with this objective in mind. This chapter will provide an in-depth examination of our program with a focus on the efforts to increase accessibility.

The first part of this chapter offers a general overview of the project's programming, covering its language, dependencies and architecture. The second part will focus on Wesnoth's unique data storage language, known as Wesnoth Markup Language (WML). It will explain the specific functions of WML, with a particular emphasis on its effects on in-game units. The next section covers multiplayer

implementation and external programs. The chapter will end with some concluding observations on our structure and the challenges of broadening participation.

25.1 Project Overview

Wesnoth's core engine is written in C++, totalling around 200,000 lines at the time of this publication. This represents the core game engine, approximately half of the code base without any content. The program also allows in game content to be defined in a unique data language known as Wesnoth Markup Language (WML). The game ships with another 250,000 lines of WML code. The proportion has shifted over the project's existence. As the program matured, game content that was hardcoded in C++ has increasingly been rewritten so that WML can be used to define its operation. Figure 25.1 gives a rough picture of the program's architecture; green areas are maintained by Wesnoth developers, while white areas are external dependencies.

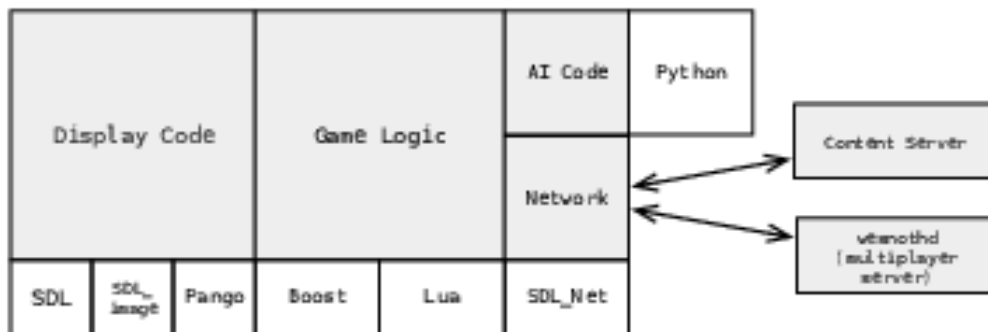


Figure 25.1: Program Architecture

Overall, the project attempts to minimize dependencies in most cases so as to maximize the portability of the application. This has the added benefit of reducing the program's complexity, and decreases the need for developers to learn the nuances of a large number of third party APIs. At the same time, the prudent use of some dependencies can actually achieve the same effect. For example, Wesnoth uses the Simple Directmedia Layer (SDL) for video, I/O and event handling. It was chosen because it is easy to use and provides a common I/O interface across many platforms. This allows it to be portable to a wide array of platforms, rather than the alternative of coding to specific APIs on different platforms. This comes at a price however; it is harder to take advantage of some platform specific features. SDL also has an accompanying family of libraries that are used by Wesnoth for various purposes:

- SDL_Mixer for audio and sound
- SDL_Image for loading PNG and other image formats
- SDL_Net for network I/O

Additionally, Wesnoth uses several other libraries:

- Boost for a variety of advanced C++ features
- Pango with Cairo for internationalized fonts

- zlib for compression
- Python and Lua for scripting support
- GNU gettext for internationalization

Throughout Wesnoth’s engine, the use of WML objects—that is, string dictionaries with child nodes—is fairly ubiquitous. Many objects can be constructed from a WML node, and also serialize themselves to a WML node. Some parts of the engine keep data in this WML dictionary based format, interpreting it directly rather than parsing it into a C++ data structure.

Wesnoth utilizes several important subsystems, most of which are as self-contained as possible. This segmented structure has advantages for accessibility. An interested party can easily work a code in a specific area and introduce changes without damaging the rest of the program. The major subdivisions include:

- A WML parser with preprocessor
- Basic I/O modules that abstract underlying libraries and system calls—a video module, a sound module, a network module
- A GUI module containing widget implementations for buttons, lists, menus, etc.
- A display module for rendering the game board, units, animations, and so forth
- An AI module
- A pathfinding module that includes many utility functions for dealing with a hexagonal gaming board
- A map generation module for generating different kinds of random maps

There are also different modules for controlling different parts of the game flow:

- The titlescreen module, for controlling display of the title screen.
- The storyline module, for showing cut-scene sequences.
- The lobby module, for displaying and allowing setup of games on the multiplayer server.
- The “play game” module that controls the main gameplay.

The “play game” module and the main display module are the largest within Wesnoth. Their purpose is the least well defined, as their function is ever-changing and thus difficult to have a clear specification for. Consequently, the modules has often been in danger of suffering from the Blob anti-pattern over the program’s history—i.e., becoming huge dominant segments without well-defined behaviors. The code in the display and play game modules are regularly reviewed to see if any of it can be separated into a module of its own.

There are also ancillary features that are part of the overall project, but are separate from the main program. This includes a multiplayer server that facilitates multiplayer network games, as well as a content server that allows users to upload their content to a common server and share it with others. Both are written in C++.

25.2 Wesnoth Markup Language

As an extensible game engine, Wesnoth uses a simple data language to store and load all game data. Although XML was considered initially, we decided that we wanted something a little more friendly to non-technical users, and a little more relaxed with regard to use of visual data. We therefore developed our own data language, called Wesnoth Markup Language (WML). It was designed with the least technical of users in mind: the hope was that even users who find Python or HTML intimidating would be able to make sense of a WML file. All Wesnoth game data is stored in WML, including unit definitions, campaigns, scenarios, GUI definitions, and other game logic configuration.

WML shares the same basic attributes as XML: elements and attributes, though it doesn't support text within elements. WML attributes are represented simply as a dictionary mapping strings to strings, with the program logic responsible for interpretation of attributes. A simple example of WML is a trimmed definition for the Elvish Fighter unit within the game:

```
[unit_type]
  id=Elvish Fighter
  name= _ "Elvish Fighter"
  race=elf
  image="units/elves-wood/fighter.png"
  profile="portraits/elves/fighter.png"
  hitpoints=33
  movement_type=woodland
  movement=5
  experience=40
  level=1
  alignment=neutral
  advances_to=Elvish Captain,Elvish Hero
  cost=14
  usage=fighter
  {LESS_NIMBLE_ELF}
  [attack]
    name=sword
    description=_ "sword"
    icon=attacks/sword-elven.png
    type=blade
    range=melee
    damage=5
    number=4
  [/attack]
[/unit_type]
```

Since internationalization is important in Wesnoth, WML does have direct support for it: attribute values which have an underscore prefix are translatable. Any translatable string is converted using GNU gettext to the translated version of the string when the WML is parsed.

Rather than have many different WML documents, Wesnoth opts for the approach of all main game data being presented to the game engine in just a single document. This allows for a single global variable to hold the document, and when the game is loaded all unit definitions, for instance, are loaded by looking for elements with the name `unit_type` within a `units` element.

Though all data is stored in a single conceptual WML document, it would be unwieldy to have it all in a single file. Wesnoth therefore supports a preprocessor that is run over all WML before parsing.

This preprocessor allows one file to include the contents of another file, or an entire directory. For instance:

```
{gui/default/window/}
```

will include all the .cfg files within gui/default/window/.

Since WML can become very verbose, the preprocessor also allows macros to be defined to condense things. For instance, the {LESS_NIMBLE_ELF} invocation in the definition of the Elvish Fighter is a call to a macro that makes certain elf units less nimble under certain conditions, such as when they are stationed in a forest:

```
#define LESS_NIMBLE_ELF
    [defense]
        forest=40
    [/defense]
#endif
```

This design has the advantage of making the engine agnostic to how the WML document is broken up into files. It is the responsibility of WML authors to decide how to structure and divide all game data into different files and directories.

When the game engine loads the WML document, it also defines some preprocessor symbols according to various game settings. For instance, a Wesnoth campaign can define different difficulty settings, with each difficulty setting resulting in a different preprocessor symbol being defined. As an example, a common way to vary difficulty is by varying the amount of resources given to an opponent (represented by gold). To facilitate this, there is a WML macro defined like this:

```
#define GOLD EASY_AMOUNT NORMAL_AMOUNT HARD_AMOUNT
    #ifdef EASY
        gold={EASY_AMOUNT}
    #endif
    #ifdef NORMAL
        gold={NORMAL_AMOUNT}
    #endif
    #ifdef HARD
        gold={HARD_AMOUNT}
    #endif
#endif
```

This macro can be invoked using, for instance, {GOLD 50 100 200} within the definition of an opponent to define how much gold the opponent has based on the difficulty level.

Since the WML is processed conditionally, if any of the symbols provided to the WML document change during execution of the Wesnoth engine, the entire WML document must be re-loaded and processed. For instance, when the user starts the game, the WML document is loaded and available campaigns among other things are loaded. But then, if the user chooses to start a campaign and chooses a certain difficulty level—easy for instance—then the entire document will have to be re-loaded with EASY defined.

This design is convenient in that a single document contains all game data, and that symbols can allow easy configuration of the WML document. However, as a successful project, more and more content is available for Wesnoth, including much downloadable content—all of which ends up inserted into the core document tree—which means the WML document is many megabytes in size.

This has become a performance issue for Wesnoth: Loading the document may take up to a minute on some computers, causing delays in-game any time the document needs to be reloaded. Additionally, it uses a substantial amount of memory. Some measures are used to counter this: when a campaign is loaded, it has a symbol unique to that campaign defined in the preprocessor. This means that any content specific to that campaign can be `#ifdef`ed to only be used when that campaign is needed.

Additionally, Wesnoth uses a caching system to cache the fully preprocessed version of the WML document for a given set of key definitions. Naturally this caching system must inspect the timestamp of all WML files so that if any have changed, the cached document is regenerated.

25.3 Units in Wesnoth

The protagonists of Wesnoth are its units. An Elvish Fighter and an Elvish Shaman might battle against a Troll Warrior and an Orcish Grunt. All units share the same basic behavior, but many have special abilities that alter the normal flow of gameplay. For example, a troll regenerates some of its health every turn, an Elvish shaman slows its opponents with an entangling root, and a Wose is invisible in a forest.

What is the best way to represent this in an engine? It is tempting to make a base unit class in C++, with different types of units derived from it. For instance, a `wose_unit` class could derive from `unit`, and `unit` could have a virtual function, `bool is_invisible() const`, which returns false, which the `wose_unit` overrides, returning true if the unit happens to be in forest.

Such an approach would work reasonably well for a game with a limited set of rules. Unfortunately Wesnoth is quite a large game and such an approach is not easily extendable. If a person wanted to add a new type of unit under this approach, it would require the addition of a new C++ class to the game. Additionally, it does not allow different characteristics to be combined well: what if you had a unit that regenerated, could slow enemies with a net, and was invisible in a forest? You would have to write an entirely new class that duplicates code in the other classes.

Wesnoth's unit system doesn't use inheritance at all to accomplish this task. Instead, it uses a `unit` class to represent instances of units, and a `unit_type` class, which represents the immutable characteristics that all units of a certain type share. The `unit` class has a reference to the type of object that it is. All the possible `unit_type` objects are stored in a globally held dictionary that is loaded when the main WML document is loaded.

A unit type has a list of all the abilities that that unit has. For instance, a Troll has the "regeneration" ability that makes it heal life every turn. A Saurian Skirmisher has the "skirmisher" ability that allows it to move through enemy lines. Recognition of these abilities is built into the engine—for instance, the pathfinding algorithms will check if a unit has the "skirmisher" flag set to see if it can move freely past enemy lines. This approach allows an individual to add new units, which have any combination of abilities made by the engine, by only editing WML. Of course, it doesn't allow adding completely new abilities and unit behavior without modifying the engine.

Additionally, each unit in Wesnoth may have any number of ways to attack. For instance, an Elvish Archer has a long-range bow attack and also a short-range sword attack. Each deals different damage amounts and characteristics. To represent an attack, there is an `attack_type` class, with every `unit_type` instance having a list of possible `attack_types`.

To give each unit more character, Wesnoth has a feature known as traits. Upon recruitment, most units are assigned two traits at random from a predefined list. For instance, a strong unit does more damage with its melee attacks, while an intelligent unit needs less experience before it "levels up." Also, it is possible for units to acquire equipment during the game that make them more powerful.

For instance, there might be a sword a unit can pick up that makes their attacks do more damage. To implement traits and equipment Wesnoth allows modifications on units, which are WML-defined alterations to a unit's statistics. The modification can even be applied to certain types of attacks. For instance, the strong trait gives strong units more damage when attacking in melee, but not when using a ranged strike.

Allowing completely configurable unit behavior with WML would be an admirable goal, so it is instructional to consider why Wesnoth has never achieved such a goal. WML would need to be much more flexible than it is if it were to allow arbitrary unit behavior. Rather than being a data-oriented language, WML would have to be extended into a full-fledged programming language and that would be intimidating for many aspiring contributors.

Additionally, the Wesnoth AI, which is developed in C++, recognizes the abilities present in the game. It takes into account regeneration, invisibility, and so forth, and attempts to maneuver its units to take best advantage of these different abilities. Even if a unit ability could be created using WML, it would be difficult to make the AI sophisticated enough to recognize this ability to take advantage of it. Implementing an ability but not having it accounted for by the AI would not be a very satisfying implementation. Similarly, implementing an ability in WML and then having to modify the AI in C++ to account for the ability would be awkward. Thus, having units definable in WML, but having abilities hard-wired into the engine is considered a reasonable compromise that works best for Wesnoth's specific requirements.

25.4 Wesnoth's Multiplayer Implementation

The Wesnoth multiplayer implementation uses a simple-as-possible approach to implementing multiplayer in Wesnoth. It attempts to mitigate the possibility of malicious attacks on the server, but doesn't make a serious attempt to prevent cheating. Any movement that is made in a Wesnoth game—moving of a unit, attacking an enemy, recruiting a unit, and so forth—can be saved as a WML node. For instance, a command to move a unit might be saved into WML like this:

```
[move]
  x="11,11,10,9,8,7"
  y="6,7,7,8,8,9"
[/move]
```

This shows the path that a unit follows as a result of a player's commands. The game then has a facility to execute any such WML command given to it. This is very useful because it means that a complete replay can be saved, by storing the initial state of the game and then all subsequent commands. Being able to replay games is useful both for players to observe each other playing, as well as to help in certain kinds of bug reports.

We decided that the community would try to focus on friendly, casual games for the network multiplayer implementation of Wesnoth. Rather than fight a technical battle against anti-social crackers trying to compromise cheat prevention systems, the project would simply not try hard to prevent cheating. An analysis of other multiplayer games indicated that competitive ranking systems were a key source of anti-social behavior. Deliberately preventing such functions on the server greatly reduced the motivation for individuals to cheat. Moreover the moderators try to encourage a positive gaming community where individuals develop personal rapport with other players and play with them. This placed a greater emphasis on relationships rather than competition. The outcome of these efforts has been deemed successful, as thus far efforts to maliciously hack the game have been largely isolated.

Wesnoth's multiplayer implementation consists of a typical client-server infrastructure. A server, known as `wesnothd`, accepts connections from the Wesnoth client, and sends the client a summary of available games. Wesnoth will display a 'lobby' to the player who can choose to join a game or create a new game for others to join. Once players are in a game and the game starts, each instance of Wesnoth will generate WML commands describing the actions the player makes. These commands are sent to the server, and then the server relays them on to all the other clients in the game. The server will thus act as a very thin, simple relay. The replay system is used on the other clients to execute the WML commands. Since Wesnoth is a turn-based game, TCP/IP is used for all network communication.

This system also allows observers to easily watch a game. An observer can join a game in-progress, in which case the server will send the WML representing the initial state of the game, followed by a history of all commands that have been carried out since the start of the game. This allows new observers to get up to speed on the state of the game. They can see a history of the game, although it does take time for the observer to get to the game's current position—the history of commands can be fast forwarded but it still takes time. The alternative would be to have one of the clients generate a snapshot of the game's current state as WML and send it to the new observer; however this approach would burden clients with overhead based on observers, and could facilitate denial-of-service attacks by having many observers join a game.

Of course, since Wesnoth clients do not share any kind of game state with each other, only sending commands, it is important that they agree on the rules of the game. The server is segmented by version, with only players using the same version of the game able to interact. Players are immediately alerted if their client's game becomes out of sync with others. This also is a useful system to prevent cheating. Although it is rather easy for a player to cheat by modifying their client, any difference between versions will immediately be identified to players where it can be dealt with.

25.5 Conclusion

We believe that the beauty of the Battle for Wesnoth as a program is how it made coding accessible to a wide variety of individuals. To achieve this aim, the project often made compromises that do not look elegant whatsoever in the code. It should be noted that many of the project's more talented programmers frowned upon WML for its inefficient syntax. Yet this compromise enabled one of the project's greatest successes. Today Wesnoth can boast of hundreds of user-made campaigns and eras, created mostly by users with little or no programming experience. Furthermore it has inspired a number of people to take up programming as a profession, using the project as a learning tool. Those are tangible accomplishments that few programs can equal.

One of the key lessons a reader should take away from Wesnoth's efforts is to consider the challenges faced by lesser skilled programmers. It requires an awareness of what blocks contributors from actually performing coding and developing their skills. For example an individual might want to contribute to the program but does not possess any programming skills. Dedicated technological editors like `emacs` or `vim` possess a significant learning curve that might prove daunting for such an individual. Thus WML was designed to allow a simple text editor to open up its files, giving anybody the tools to contribute.

However, increasing a code base's accessibility is not a simple objective to achieve. There are no hard and fast rules for increasing code's accessibility. Rather it requires a balance between different considerations, which can have negative consequences that the community must be aware of. This is apparent in how the program dealt with dependencies. In some cases, dependencies can actually

increase barriers to participation, while in others they can allow people to contribute more easily. Every issue must be considered on a case-by-case basis.

We should also be careful not to overstate some of Wesnoth's successes. The project enjoyed some advantages that are not easily replicated by other programs. Making code accessible to a wider public is partly a result of the program's setting. As an open source program, Wesnoth had several advantages in this regard. Legally the GNU license allows someone to open up an existing file, understand how it works and makes changes. Individuals are encouraged to experiment, learn and share within this culture, which might not be appropriate for other programs. Nevertheless we hope that there are certain elements that might prove useful for all developers and help them in their effort to find beauty in coding.

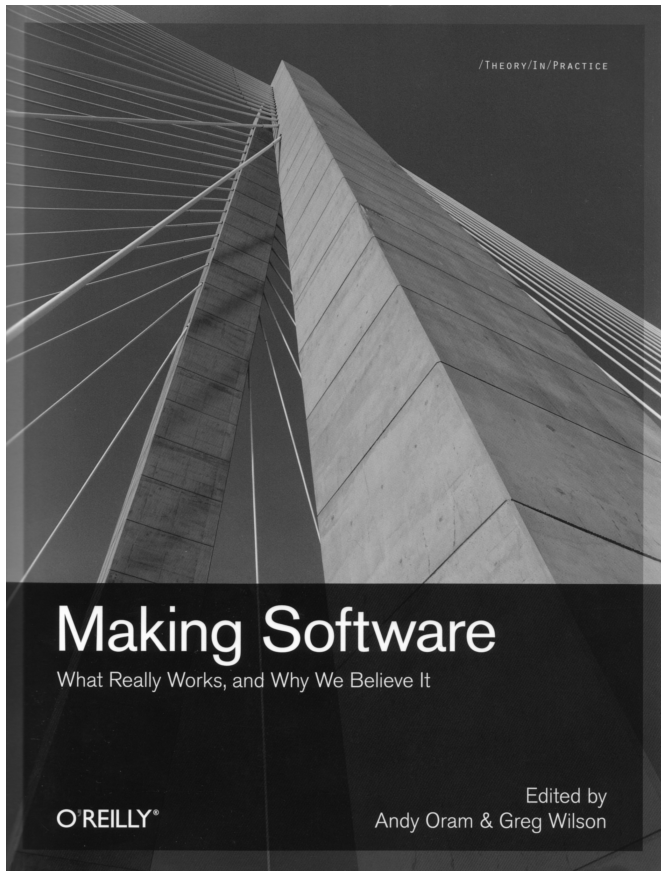
Bibliography

- [AF94] Rick Adams and Donnalyn Frey. *!%:: A Directory of Electronic Mail Addressing & Networks*. O'Reilly Media, Sebastopol, CA, fourth edition, 1994.
- [Ald02] Gaudenz Alder. *The JGraph Swing Component*. PhD thesis, ETH Zurich, 2002.
- [BCC⁺05] Louis Bavoil, Steve Callahan, Patricia Crossno, Juliana Freire, Carlos E. Scheidegger, Cláudio T. Silva, and Huy T. Vo. VisTrails: Enabling Interactive Multiple-View Visualizations. In *Proceedings of IEEE Visualization*, pages 135–142, 2005.
- [Bro10] Frederick P. Brooks, Jr. *The Design of Design: Essays from a Computer Scientist*. Pearson Education, 2010.
- [CDG⁺06] Fay Chang, Jeffrey Dean, Sanjary Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *Proc. 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*. USENIX Association, 2006.
- [CIRT00] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: A Parallel File System for Linux Clusters. *Proc. 4th Annual Linux Showcase and Conference*, pages 317—327, 2000.
- [Com79] Douglas Comer. Ubiquitous B-Tree. *ACM Computing Surveys*, 11:121–137, June 1979.
- [CRS⁺08] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.
- [DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Proc. Sixth Symposium on Operating System Design and Implementation*, 2004.
- [DHJ⁺07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Voshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *SOSP'07: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, pages 205–220, 2007.

- [FKSS08] Juliana Freire, David Koop, Emanuele Santos, and Cláudio T. Silva. Provenance for Computational Tasks: A Survey. *Computing in Science and Engineering*, 10(3):11–21, 2008.
- [FSC⁺06] Juliana Freire, Cláudio T. Silva, Steve Callahan, Emanuele Santos, Carlos E. Scheidegger, and Huy T. Vo. Managing Rapidly-Evolving Scientific Workflows. In *International Provenance and Annotation Workshop (IPAW)*, LNCS 4145, pages 10–18. Springer Verlag, 2006.
- [GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. *Proc. ACM Symposium on Operating Systems Principles*, pages 29—43, 2003.
- [GL02] Seth Gilbert and Nancy Lynch. Brewer’s Conjecture and the Feasibility of Consistent Available Partition-Tolerant Web Services. *ACM SIGACT News*, 33(2), 2002.
- [GLPT76] Jim Gray, Raymond Lorie, Gianfranco Putzolu, and Irving Traiger. Granularity of Locks and Degrees of Consistency in a Shared Data Base. *Proc. 1st International Conference on Very Large Data Bases*, pages 365–394, 1976.
- [GR09] Adam Goucher and Tim Riley, editors. *Beautiful Testing*. O’Reilly, 2009.
- [Gra81] Jim Gray. The Transaction Concept: Virtues and Limitations. *Proc. Seventh International Conference on Very Large Data Bases*, pages 144–154, 1981.
- [Hor05] Cay Horstmann. *Object-Oriented Design and Patterns*. Wiley, 2 edition, 2005.
- [HR83] Theo Haerder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15, December 1983.
- [Kit10] Kitware. *VTK User’s Guide*. Kitware, Inc., 11 edition, 2010.
- [Knu74] Donald E. Knuth. Structured Programming with go to Statements. *ACM Computing Surveys*, 6(4), 1974.
- [LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. 2004 International Symposium on Code Generation and Optimization (CGO’04)*, Mar 2004.
- [LCWB⁺11] H. Andrés Lagar-Cavilla, Joseph A. Whitney, Roy Bryant, Philip Patchin, Michael Brudno, Eyal de Lara, Stephen M. Rumble, M. Satyanarayanan, and Adin Scannell. SnowFlock: Virtual Machine Cloning as a First-Class Cloud Primitive. *ACM Transactions on Computer Systems*, 19(1), 2011.
- [Mac06] Matt Mackall. Towards a Better SCM: Revlog and Mercurial. 2006 Ottawa Linux Symposium, 2006.
- [MQ09] Marshall Kirk McKusick and Sean Quinlan. GFS: Evolution on Fast-forward. *ACM Queue*, 7(7), 2009.
- [PGL⁺05] Anna Persson, Henrik Gustavsson, Brian Lings, Björn Lundell, Anders Mattson, and Ulf Ärlig. OSS Tools in a Heterogeneous Environment for Embedded Systems Modelling: an Analysis of Adoptions of XMI. *SIGSOFT Software Engineering Notes*, 30(4), 2005.

- [PPT⁺93] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The Use of Name Spaces in Plan 9. *Operating Systems Review*, 27(2):72–76, 1993.
- [Rad94] Sanjay Radia. Naming Policies in the Spring System. *Proc. 1st IEEE Workshop on Services in Distributed and Networked Environments*, pages 164–171, 1994.
- [RP93] Sanjay Radia and Jan Pachl. The Per-Process View of Naming and Remote Execution. *IEEE Parallel and Distributed Technology*, 1(3):71–80, 1993.
- [Shu05] Rose Shumba. Usability of Rational Rose and Visio in a Software Engineering Course. *SIGCSE Bulletin*, 37(2), 2005.
- [Shv10] Konstantin V. Shvachko. HDFS Scalability: The limits to growth. *;login.*, 35(2), 2010.
- [SML06] Will Schroeder, Ken Martin, and Bill Lorensen. *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*. Kitware, Inc., 4 edition, 2006.
- [SO92] Margo Seltzer and Michael Olson. LIBTP: Portable, Modular Transactions for UNIX. *Proc. 1992 Winter USENIX Conference*, pages 9–26, January 1992.
- [Spi03] Diomidis Spinellis. On the Declarative Specification of Models. *IEEE Software*, 20(2), 2003.
- [SVK⁺07] Carlos E. Scheidegger, Huy T. Vo, David Koop, Juliana Freire, and Cláudio T. Silva. Querying and Creating Visualizations by Analogy. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1560–1567, 2007.
- [SY91] Margo Seltzer and Ozan Yigit. A New Hashing Package for UNIX. *Proc. 1991 Winter USENIX Conference*, pages 173–184, January 1991.
- [Tan06] Audrey Tang. *-Ofun: Optimizing for Fun*. <http://www.slideshare.net/autang/ofun-optimizing-for-fun>, 2006.
- [Top00] Kim Topley. *Core Swing: Advanced Programming*. Prentice-Hall, 2000.

You May Also Enjoy...



Many claims are made about how certain tools, technologies, and practices improve software development. But which are true, and which are merely wishful thinking? In *Making Software*, leading researchers and practitioners present chapter-length summaries of key empirical findings in software engineering, and answer questions like:

- Are some programmers really ten times more productive than others?
- Does writing tests first help you develop better code faster?
- Can code metrics predict the number of bugs in a piece of software?
- Does using design patterns actually make software better?
- What effect does personality have on pair programming?
- What matters more: how far apart people are geographically, or how far apart they are in the org chart?

As with *The Architecture of Open Source Applications*, royalties from *Making Software* will be donated to Amnesty International.

Making Software: What Really Works, and Why We Believe It

edited by Andy Oram and Greg Wilson

O'Reilly Media, 2010, 978-0596808327

<http://oreilly.com/catalog/9780596808303>

Colophon

The image on the cover is a photograph of Chris Denison's *48 Free Street Mural* in Portland, Maine, taken by Peter Dutton.

The cover font is Junction by Caroline Hadilaksono. The text font is T_EXGyre Termes and the heading font is T_EXGyre Heros, both by Bogusław Jackowski and Janusz M. Nowacki. The code font is Inconsolata by Raph Levien.

