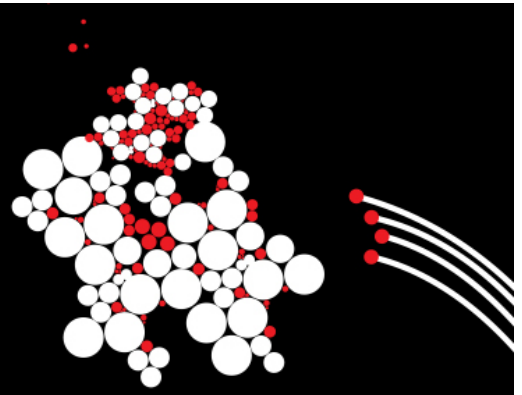
UNIVERSITY OF TWENTE.

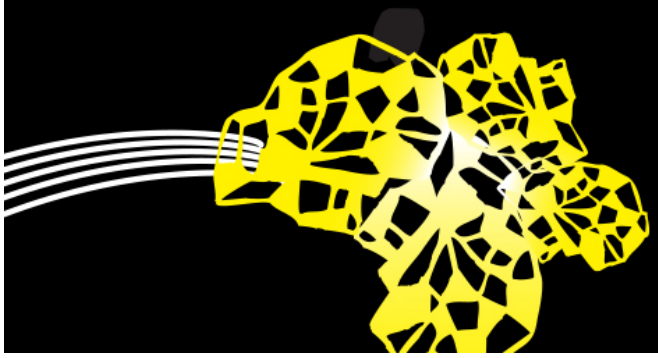


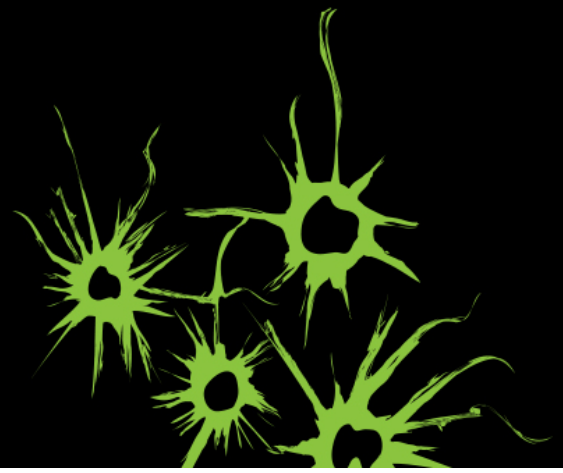
P7.2: NETWORKING AND MULTITHREADING

L. FERREIRA PIRES

201700117-1B MODULE 2: SOFTWARE SYSTEMS

7 JANUARY 2020





PROGRAMMING LINE OVERVIEW

Week 1 Values and variables Control flow	**Week 2** Classes and objects Testing	**Week 3** Interfaces and Inheritance Subtyping Security 1
Week 4 Arrays and Lists List implementations Collections	**Week 5** Exceptions I/O Streams and MVC Security 2	**Week 6** Concurrency Project kick-off IDE Tips & Tricks
Week 7 Basic Networking Networking and Multithreading GUIs	**Week 8/9** Advanced Java facilities Test	**Week 10** Project Test resit

TRANSPORT LAYER: SOCKET

- Socket is a common abstraction of the transport layer → TCP or UDP
  - Combine IP address (machine) and port number (application)
- Provides communication between program parts
  - Server waits on a port at an Internet address
  - Client tries to connect with port at an Internet address
- Java classes
  - `Socket`, `ServerSocket` (TCP)

JAVA.NET.SERVERSOCKET

SOCKET FOR ACCEPTING A CONNECTION

- Listens on fixed port for incoming connections
  - Creates a connection on a certain port for each incoming request
  - Provides a `Socket` object for each created connection
- `ServerSocket(int port)`: constructor; if port is `0` chooses a free port
- `Socket accept()`
  - Blocks waiting for a client's attempt to establish a connection
  - Returns a `Socket` if the attempt is successful

JAVA.NET.SOCKET

SOCKET FOR COMMUNICATION

- Provides access to TCP/IP streams
  - Bi-directional communication between sender and receiver
- `Socket(String remoteHost, int port)`
  - Constructor, starts a connection with remote host at port
- `InputStream getInputStream()`
  - Allows data from the other party to be received
- `OutputStream getOutputStream()`
  - Allows data to be sent to the other party

DATE SERVER EXAMPLE

CLIENT CODE

```
20  public class DateClient {
21
22      public static final int LISTENING_PORT = 32007;
23
24      public static void main(String[] args) {
25          String hostName; // Name of the server computer to connect to.
26          Socket connection; // A socket for communicating with server.
27          BufferedReader incoming; // For reading data from the connection.
28          /* Get computer name from command line. */
29          if (args.length > 0)
30              hostName = args[0];
31          else {
32              hostName = "localhost";
33          }
34          /* Make the connection, then read and display a line of text. */
35          try {
36              connection = new Socket(hostName, LISTENING_PORT);
37              incoming = new BufferedReader(new InputStreamReader(connection.getInputStream()));
38              String lineFromServer = incoming.readLine();
39              if (lineFromServer == null) { // Null indicates end-of-stream
40                  throw new IOException("No data was sent!");
41              }
42              System.out.println(lineFromServer);
43              incoming.close();
44          } catch (Exception e) {
45              System.out.println("Error:  " + e);
46          }
47      }
48  }
```

Opens a connection to host computer with port 32007

Reads line from server

Prints received line

DATE SERVER EXAMPLE

SERVER CODE

```
17  public class DateServer {
18
19      public static final int LISTENING_PORT = 32007;
20
21      public static void main(String[] args) {
22          ServerSocket listener;  // Listens for incoming connections.
23          Socket connection;      // For communication with the connecting program.
24          // Accept and process connections forever, or until some error occurs.
25          try {
26              listener = new ServerSocket(LISTENING_PORT);
27              System.out.println("Listening on port " + LISTENING_PORT);
28              while (true) {
29                      // Accept next connection request and handle it.
30                  connection = listener.accept();
31                  sendDate(connection);
32              }
33          }
34          catch (Exception e) {
35              System.out.println("Sorry, the server has shutdown.");
36              System.out.println("Error:  " + e);
37              return;
38          }
39      }
```

Listens to connection requests at port 32007

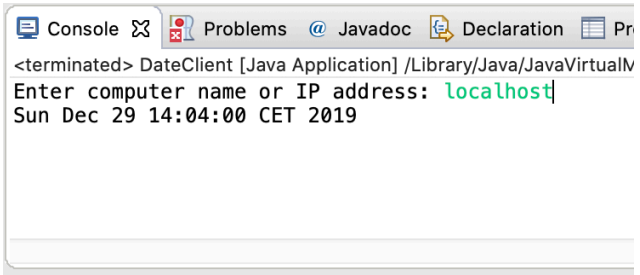
Accepts connections and sends date (indefinitely)

DATE SERVER EXAMPLE

SEND THE DATE

```
44    /**
45     * The parameter, client, is a socket that is already connected to another
46     * program.  Get an output stream for the connection, send the current time,
47     * and close the connection.
48     */
49    private static void sendDate(Socket client) {
50        try {
51            System.out.println("Connection from " +
52                    client.getInetAddress().toString() );
53            Date now = new Date();  // The current date and time.
54            PrintWriter outgoing;   // Stream for sending data.
55            outgoing = new PrintWriter( client.getOutputStream() );
56            outgoing.println( now.toString() );
57            outgoing.flush();  // Make sure the data is actually sent!
58            client.close();
59        }
60        catch (Exception e){
61            System.out.println("Error: " + e);
62        }
63    } // end sendDate()
```

Executing client



MORE EXAMPLES

- Until now in our examples a client queries a server
  - Webcat and DateClient/DateServer
- Alternative: `CLChatClient` and `CLChatServer` from Eck
  - Client and server are symmetric after connection is established
  - Server and client do handshake and send and receive messages
  - I modified the code to stress the symmetry by defining a `CLChatHandler` with behaviours shared by the Client and the Server

CLCHATAPP

Server

- Listens to a client connection request on port 1728 (default)
- After connection is established creates a `CLChatHandler` and performs handshake (call to `doHandshake`)
- In a loop, receives and sends messages with the `CLChatHandler`

CLCHATSERVER

```
70          /*
71           * Wait for a connection request. When it arrives, close down the listener.
72           */
73          try {
74              listener = new ServerSocket(port);
75              System.out.println("Listening on port " + listener.getLocalPort());
76              connection = listener.accept();
77              listener.close();
78          } catch (IOException e) {
79              System.out.println("An error occurred while opening connection.");
80              System.out.println(e.toString());
81          }
82          /*
83           * Create a handler to do the handshake and exchange messages.
84           */
85          try {
86              CLChatHandler handler = new CLChatHandler(connection);
87              handler.doHandshake();
88              System.out.println("Connected.  Waiting for the first message.");
89              while (true) {
90                  handler.receiveMessage();
91                  handler.sendMessage();
92              }
93          } catch (IOException e) {
94              System.out.println("Sorry, an error has occurred.  Connection lost.");
95              System.out.println("Error:  " + e);
96          }
97      }
```

CLCHATAPP

Client

- Starts connection on port 1728
- After connection is started, creates a `CLChatHandler` and performs handshake (call to `doHandshake`)
- In a loop, sends and receives messages with the `CLChatHandler`

CLChatHandler

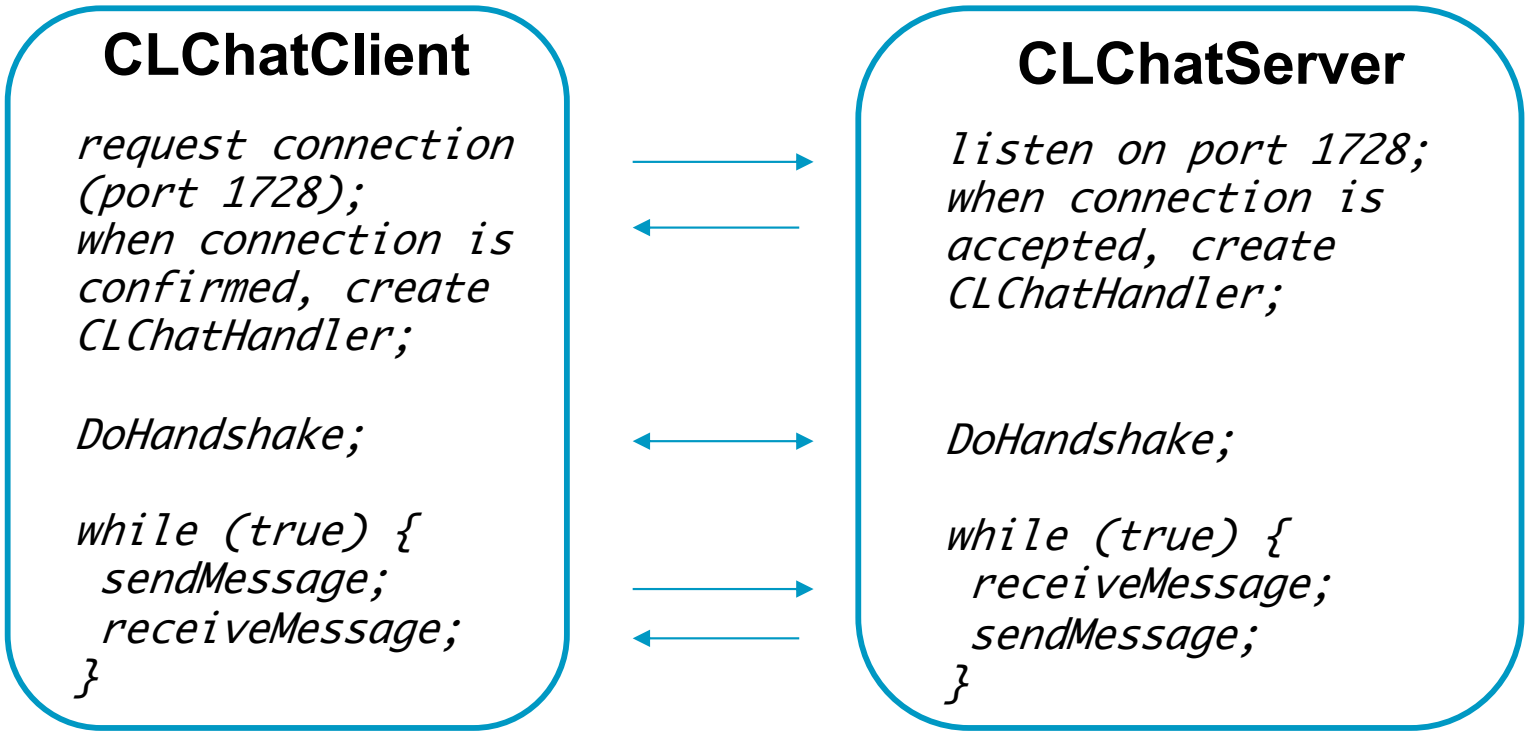
- Methods `doHandshake`, `sendMessage` and `receiveMessage`

CLCHATCLIENT

```
88          /*
89           * Open a connection to the server.
90           */
91          try {
92              System.out.println("Connecting to " + computer + " on port " + port);
93              connection = new Socket(computer, port);
94          } catch (IOException e) {
95              System.out.println("An error occurred while opening connection.");
96              System.out.println(e.toString());
97          }
98          /*
99           * Create a handler to do the handshake and exchange messages.
100          */
101         try {
102             CLChatHandler handler = new CLChatHandler(connection);
103             handler.doHandshake();
104             System.out.println("Connected!!! Enter your first message.");
105
106             while (true) {
107                 handler.sendMessage();
108                 handler.receiveMessage();
109             }
110         } catch (Exception e) {
111             System.out.println("Sorry, an error has occurred.  Connection lost.");
112             System.out.println("Error:  " + e);
113         }
114     }
```

CLCHATAPP

END-TO-END COMMUNICATION



PROBLEM WITH CLCHATAPP

- Client and Server must wait for each other to send and receive messages!
- This is unrealistic since network communication is inherently asynchronous
  → no control of when messages are sent (and should be received)
- How to cope with asynchrony?
  - Multithreading: different threads for the network and user inputs!

BLOCKING AND I/O

- Program that reads data from input stream waits (blocks) until data are available or returns immediately if data were sent before
- Example: Date client

```
35      try {
36          connection = new Socket(hostName, LISTENING_PORT);
37          incoming = new BufferedReader(new InputStreamReader(connection.getInputStream()));
38          String lineFromServer = incoming.readLine();
39          if (lineFromServer == null) { // Null indicates end-of-stream
40              throw new IOException("No data was sent!");
41          }
```

- Programs may also block when performing output! Why?

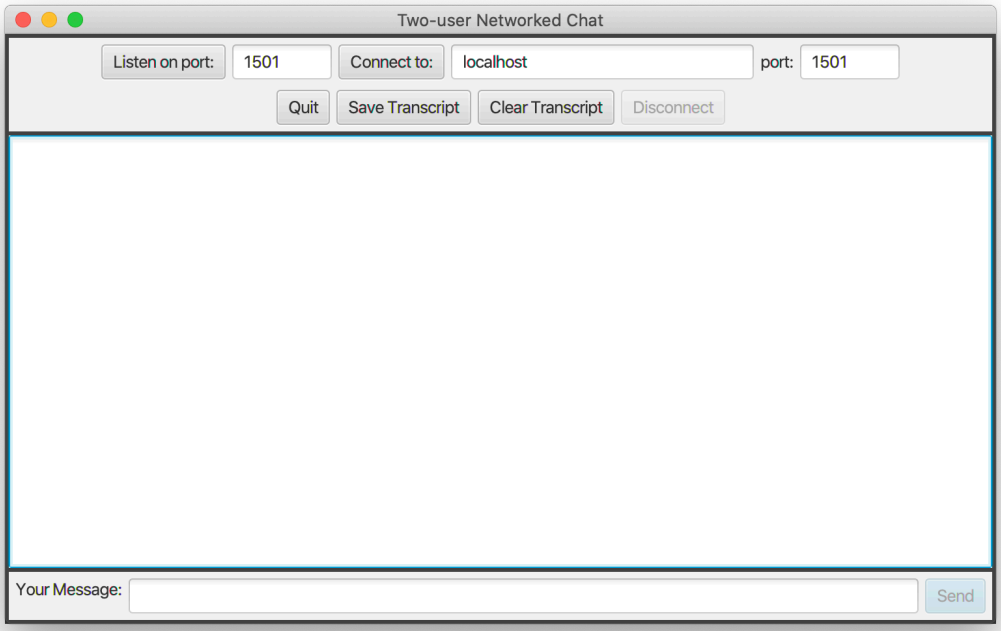
Attempts to send faster than data can be transmitted

BLOCKING AND MULTITHREADING

- If a program (thread) blocks it can only be unblocked when the event it is waiting for happens (e.g., data arrival)
- If the program needs to be reactive, for example, to the user interface, we need multiple concurrent threads so that some threads can be blocked while others keep on working
- Situation even worse with servers that must handle multiple clients!
- Good practice: after a connection is established, create a thread only to handle this connection

EXAMPLE: GUICHAT

CONCENTRATING ON THE NETWORKING CODE



- Works as client or server
- Start one program (‘server’), let it listen to a port and connect with another (‘client’)
- After connection is established, both `GUIChat` programs work in the same way

CONNECTIONHANDLER CLASS

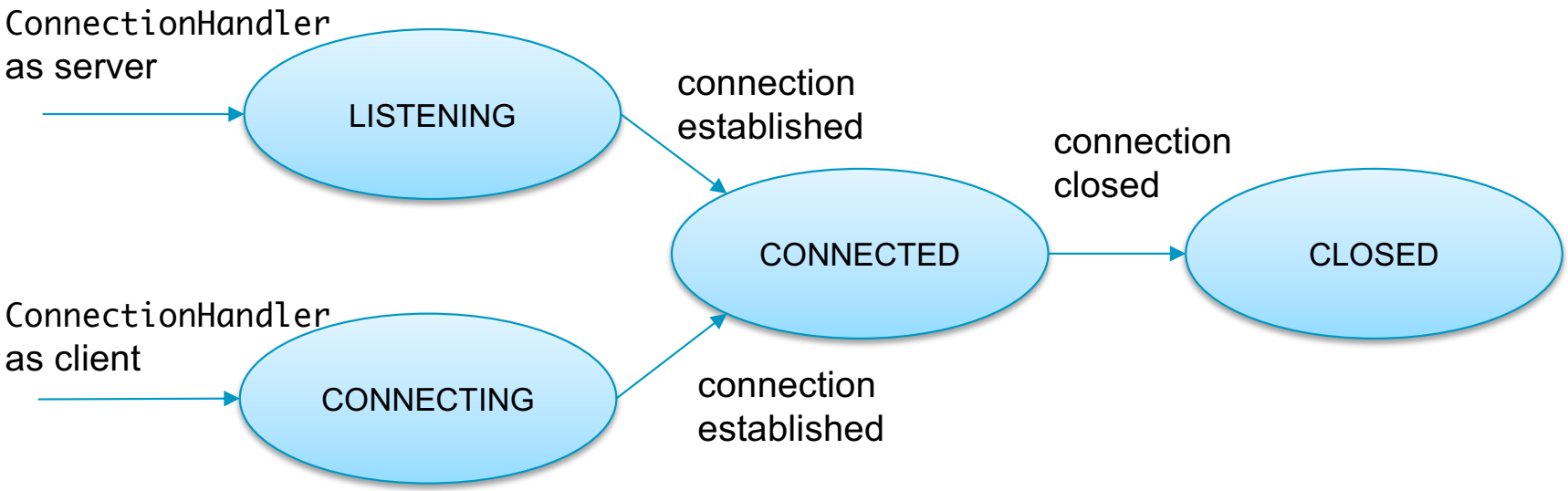
- A `ConnectionHandler` inner class extends `Thread` and has been defined to handle the chat protocol
- `ConnectionHandler` object is created and started when a `GUIChat` listens to a port (acts as server) or attempts to establish a connection (acts as client)

```
/**
 * Defines the thread that handles the connection.
 * for opening the connection and for receiving messa
 * several methods that are called by the main class,
 * executed in a different thread.  Note that by usir
 * connection, any blocking of the graphical user int
 * using a thread for reading messages sent from the
 * can be received and posted to the transcript asyn
 * time as the user is typing and sending messages.
 * that are made by this class are done using Platfor
 */
private class ConnectionHandler extends Thread {

    private volatile ConnectionState state;
    private String remoteHost;
    private int port;
    private ServerSocket listener;
    private Socket socket;
    private PrintWriter out;
    private BufferedReader in;

    /**
     * Listen for a connection on a specified port.
     * does not perform any network operations; it ju
     * instance variables and starts the thread.  Not
     * thread will only listen for one connection, ar
     * close its server socket.
     */
    ConnectionHandler(int port) {
        state = ConnectionState.LISTENING;
        this.port = port;
```

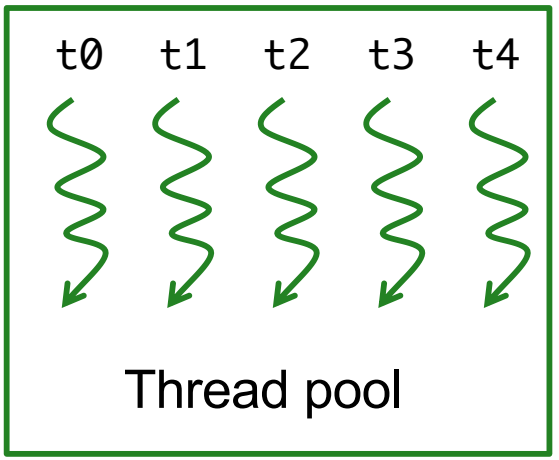
CONNECTION STATES




```
/**
 * Possible states of the thread that handles the network connection.
 */
private enum ConnectionState { LISTENING, CONNECTING, CONNECTED, CLOSED }
```

THREAD POOL

- Instead of creating threads on demand, we can create a pool of threads beforehand and allocate an available thread to a connection when necessary
- This can be used to avoid the thread creation overhead and limit the number of threads being created (avoid depleting resources)
- May be too sophisticated for this module...



See `DateServerWithThreadPool` example in Eck!

TAKE HOME MESSAGES

- Protocols define the rules for interoperability of system parts
- The client/server architecture is a quite popular for implementing distributed software systems
- Java offers support to build (client/server) applications on top of TCP
- For less than trivial applications, it is necessary to use multithreading, so that different connections can be supported while keeping the user interface responsive