Network systems

Notes on the lectures

Week 1

Introduction

A network system is a group of related parts that work together. An example of this is the Internet, consisting of millions of connected computing devices (hosts), connected via communication links, like fiber, copper, radio and satellite, and packet switches forwarding data. The hosts run applications for end-users, like web servers, e-mail, instant messaging and audio and video streaming services. A network should be scalable, manageable and it should fit the needs of the application, like reliability and speed.

A naive approach for a network would be to create a link between every host on the network. This, obviously, does not scale, as you'll need a lot of links. Another approach is a shared link. This scales quite well, but the shared link becomes a bottleneck, making the network slow.

Usually, a switched network is used. A switched network consists of hosts and switches, connected to each other. The switches forward data between hosts and switches. Switched networks can be interconnected, like the network of the UT is connected with the network of the University of Delft via the Internet.

Internetworking brings a lot of challenges. Every device in the network needs to be addressable, and preferably, routable: every host should be able to reach every other host. Furthermore, different approaches for distributing data exist. While unicast is a method of getting a packet from host A to host B, techniques like broadcast and multicast enable packets to reach many hosts (subscribers) efficiently.

But even on a lower level challenges appear. If multiple hosts from one network communicate with multiple hosts on another network, they all have logical flows of data that need to go through one shared link. As described before, this is a bottleneck that should be tackled.

To solve these problems, many solutions exist, like circuit switching (used in voice channels like telephony) and packet switching (used in most data networks).

A network consists of multiple layers, one layer depending on the features that the next provides. In the slides, we assume the application, transport, network, link and physical layers.

Application	Network applications, like FTP, SMTP, HTTP.
Transport	Process-to-process data transfer, like TCP, UDP.
Network	Routing of packets from source to destination, like IP.
Link	Data transfer between **neighboring** network elements, like Ethernet, 802.11.
Physical	Physical aspects of the network, like a piece of copper or "the air".

The different layers in a network.

Except for the physical layer, most layers encapsulate the data of the layer above. For example, the transport layer might encapsulate a HTTP GET-request in one or more TCP packets, and the network layer encapsulates those in possibly even more IP packets. The layers only need to disassemble the information they care about. A switch only cares about information in the link layer, so it will not parse information about the HTTP request. Only at the destination (with a few exceptions, of course) will the packet be fully disassembled, in this example at the web server parsing the GET-request.

Performance and applications

The performance of a network can roughly be described in two ways: a quick response, or a quick download. A quick response means that there is a low round-trip-time, while a quick download means there is a high bandwidth.

The one-way-delay describes how long a packet takes to travel through the network in one way, say, form host A to host B. The round-trip-time describes how long a packet takes to travel through the network and back. As the paths through the network don't have to be symmetric, the RTT is not always twice the one-way-delay. If messages in a network are short, the RTT determines how quickly a system can respond.

Source	Origin	Calculation
d_{trans}: transmission delay	The technical limit of the link layer, for example the speed of the switch.	$d_{trans} = \frac{L}{R}$ L: packet length (bits) R: link bandwidth (bits / s)
d_{prop}: propagation delay	The physical limit of the link layer, for example the time it takes for electricity to flow through a copper wire.	$d_{\text{prop}} = \frac{d}{s}$ d: length of the physical link (m) s: propagation speed in medium (m/s, usually ~2×10^8 m/s)
d_{proc}: processing delay	Determining where a packet should go. Typically less than a ms.	N/A
d_{queue}: queuing delay	The time a packet waits at the output link for transmission, depends on router congestion level.	N/A

Different sources of packet delay, their origin and how to calculate them.

Bandwidth indicates how many bits per second can be sent though a single link or network path, and it roughly determines how quickly a large file can be downloaded. The bandwidth-delay product indicates how many bits "fit" in a link. It's used for performance and protocol design. For example, a telephone modem link can only contain 56 bits, while a transatlantic fiber link can contain 10^9 bits. That means that you'll have sent 1/5th (125 MB) of a CD before the other host even starts receiving it. The bandwidth-delay product is literally calculated by multiplying the bandwidth and the delay (most of the times the RTT) of a link.

Sometimes, not only the delay is important, but also the variation in delay (jitter). If an application that streams audio or video experiences too much jitter, interruptions in the streams will occur.

There are a lot of network applications, and they can use different architectures. In a client-server model, there is one always-on server, with a permanent address, with which clients connect and communicate. The clients do not communicate with each other. An example is the world wide web, or e-mail.

Peer-to-peer applications do not depend on a centralized server. The peers communicate with each other. These networks are difficult to manage, as peers come and go. An example is BitTorrent.

Some applications use a hybrid of client-server and P2P. Skype, for example, uses a central server for accounting, but a peer-to-peer connection for calling and chatting. This makes the application more scalable, as the server does not need to forward every active call.

Applications on a network follow one or more protocols. Web browsers use the HTTP protocol, e-mail clients use SMTP. Ideally, they all comply to the protocol (which is standardized and published) and thus they're interchangeable. This means that sending mail using Thunderbird to someone using Outlook should not be a problem.

Multimedia and P2P applications

Audio signals are sampled at a constant rate. Telephone conversations are sampled at 8000 samples per second, while higher quality CD music is sampled at 44100 samples per second. Each sample is quantized, or rounded. If the sample is represented by 8 bits, $2^8 = 256$ possible quantized values are possible. Often, data compression techniques like MP3 are applied, which don't store frequencies humans can't hear.

Video signals are a sequence of images, displayed at a constant rate, like 24 or 60 frames per second. Every frame is a digital image, which is an array of pixels. Every pixel is represented by a few bits.

If every frame were to be stored as a bitmap image, without any compression, video files would be huge. However, as a lot of frames only differ slightly compared to the one before them, this redundancy is used to decrease the file size. Furthermore, the redundancy within a frame is often used to compress the file. The first is called temporal, while the latter is spatial.

A very basic approach of spatial coding is not to send repeating pixels of the same color, but to send that the following n pixels are color c, vastly decreasing the amount of data. Temporal encoding is done by sending the differences between frame i and frame $i + 1$, rather than sending frame i as a whole.

Video can be encoded using a constant bit rate (CBR) or variable bit rate (VBR). With constant bit rate, the amount of data per second is always the same. With variable bit rate, the amount of data per second depends on the amount of spatial and/or temporal coding. For example, a still image of a sky will have very little spatial data, while a video of a stroboscope will have a lot of spatial data.

In networking, three application types exist for multimedia. Streaming stored audio or video means that the media already exists, and is requested from the server whenever the uses wishes to view it. Examples include YouTube and Netflix.

Streaming live audio or video means that the content does not exist yet, but that the server sends the content as it is created. A subset of this type of application is conversational media, like Skype, where conversations need to be transmitted. Conversational media has less tolerance for delays.

As delaying may occur at any point in the network from the server up to the client, simply playing data as it arrives at the client will not work. The client buffers a small bit of media before it starts to play it. If there is a delay in the streaming, the client can "empty out" its buffer and continue playing, without the user ever noticing that a delay occurred. The longer the client waits until it starts playing the media, the less likely the video will ever freeze or stutter. However, this also means that the user has to wait longer before the video starts.

Conversational media has tough requirements. A small delay of over 400 ms is already considered very bad by end users. A lot of factors impact the quality of the "call", at both the application and the network layer. The application needs to process incoming and outgoing packets fast enough, while the network should have low delays and little jitter.

A popular, open protocol for conversational media is SIP (Session Initiation Protocol), which is used in a lot of business telephony setups. It indicates where a user is, if they are available and what type of media (encoding techniques) they support. Furthermore, it helps set up and manage a session, enabling features like call forwarding.

Peer-to-peer applications

As explained before, peer-to-peer applications are an alternative to the client-server model, requiring no centralized infrastructure. A peer-to-peer network is distributed and self-organizing. This makes it very scalable. Examples include file distribution (BitTorrent) and VoIP (Skype).

In a client-server model, distributing the same file to a lot of clients is a tedious and slow process. Given a file with size F, and an upload speed of u_s of the server, transmitting the file to one client takes F/U_s. Sending the file to N clients then takes $N\ F/U_s$. Furthermore, every client needs to download the file. Given a download rate of d_{min}, this takes F/d_{min}. In conclusion, sending a file with size F to N clients takes

$$D_{cs} \geq max\left\{\frac{NF}{u_s}, \frac{F}{d_{min}}\right\} \quad (1)$$

In the P2P model, the server only needs to upload at least one copy, which, as we know, takes F/u_s. The client then downloads the copy, which takes F/d_{min}. After this, every client can upload the file at their upload speed u_i. The maximum upload rate of the system then becomes $\sum u_i$. So, the time it takes to distribute a file with size F to N clients then takes

$$D_{P2P} \geq max\left\{\frac{F}{u_s}, \frac{F}{d_{min}}, \frac{NF}{u_s + \sum_{i=1}^{N} u_i}\right\} \quad (2)$$

In (1), the first argument for *max* increases linearly in N. In (2), the time does not increase linearly in N, as this is corrected by the sum in the last argument of *max*. This proves that P2P is way quicker than the client-server model for an increasing amount of clients. ∎

BitTorrent is a very popular P2P file distribution technology. In BitTorrent, the files are split up into chunks of equal size. The peers sharing the torrent then send and receive those chunks. A tracker keeps track of the peers participating in the torrent exchange and helps the peers find each other. Such a collection of peers is called a swarm. A swarm is very dynamic, as people may join and leave it whenever they want.

However, a tracking server is not always necessary. Often times a Distributed Hash Table (DHT) is used. A DHT is a distributed P2P key-value database. Any peer can query the DHT for a certain key, and the DHT responds with the value that matches that key. A peer can also insert (key, value) pairs.

Which peer stores what key is determined using a hash function. The key gets converted into an integer, and the peer with the ID (for example, the hash of their IP address) closest to that key stores that value.

In a circular DHT, every peer is only aware of its immediate successor and predecessor. To query for a certain key, the peer requests the value from the peer “above” or “below” it, depending on the requested key and the ID of the requesting peer.

Often times, shortcuts are implemented, like in a binary search. Here, the peer keeps track of shortcuts to peers further ahead or behind them, to ask for values of keys that are “far away”. This decreases the amount of necessary messages from *O(N)* to *O(log N)*.

To handle peer churn (coming and going of peers), each peer periodically pings its two successors. If the immediate successor leaves, the next successor is chosen as the new immediate successor, maintaining the integrity of the circular DHT.

Week 2

Error detection & correction, Shannon’s information theory

In the physical layer, bit errors may occur, which causes individual bits to be received incorrectly. The cause may be any kind of physical noise. Remedies include detecting the bit error, and correcting the bit error.

Error detecting codes are based on the principle of sending extra bits, calculated based on the data. The receiver does the same calculations, and if they come out the same, the data is intact. The general principle is to map the set of 2^k possible k-bit messages to 2^n code words, of n bits each. All other $2^n - 2^k$ patterns of bits are invalid. The message can then be discarded, or corrected based on the code word that is least different from the received code word.

A very naive approach is to send every bit twice. This is very inefficient, as it doubles the frame size, but also not very robust, as it’s possible that both copies are damaged, preventing the error from being detected.

Another technique is parity. One bit is added to the frame, such that the total number of 1s is even, or odd, depending on the implementation. It’s very efficient, and can detect any single-bit error. However, double-bit errors are not detected.

A more advanced technique is a parity matrix. Here, the data is laid in a matrix, and a parity check is done for every row and column in the matrix. The overhead is $2n + 1$ bits for n^2 bits of data. Parity matrices significantly reduce possible flaws in error detection, but they don’t eliminate them.

With a checksum, the data is considered a sequence of integers. Those integers can be added together, making the sum the checksum. It is quite efficient, but a double-bit error can still fool it. This technique is used in IP and TCP headers.

Cyclic redundancy check (CRC) is a very efficient but complicated technique. CRC does some math magic and creates a checksum, which can be verified. The hardware implementation for CRC is quite simple.

Error correction is possible for a number of the mentioned approaches. When using a repetition code, like repeating every bit three times, the error can be corrected *and* corrected, using a majority decision. For a parity matrix, a single bit error can be corrected using matrix magic.

Much better error-correcting codes have been designed than the ones described above. They are capable of correcting more bit errors, and/or they have less overhead. They're often quite complicated and use a lot of mathematical magic. They require quite an amount of computation, and sometimes a larger block of data, making them not always the best choice.

Measuring information

The amount of information in a message depends on how much you already knew about the message. If you knew the entire message in advance, there is no information in it. If you knew it was going to be an *n*-digit binary message, there were *n* bits of information. If you knew it was 90% sure to be A, but it could be B, then the amount of information is $-0.9 \cdot \log_2(0.9) - 0.1 \cdot \log_2(0.1) \approx 0.47$ bits.

These numbers are calculated using the formula for entropy. If a message is going to be M_1, M_2, ..., or M_n, with probabilities p_1 ... p_n then the amount of information in the message can be can be calculated as follows.

$$H = \sum_{i=1}^{n} p_i \log_2\left(\frac{1}{p_i}\right)$$

This is called the amount of information, or *entropy* of the message. It also means that you can encode the message using on average *H* bits per message.

Every noisy channel, like a physical medium, has a certain channel capacity *C*. If a source generates information at a rate less than *C*, then there exists a coding technique, such that the output of the source may be transmitted over the channel with an arbitrarily low probability of symbol error.

Given a probability *p* of a bit being received correctly, and a channel rate of *R* bits per second, the channel capacity can be calculated as follows.

$$C = R \cdot \left[1 - p \log_2\left(\frac{1}{p}\right) - (1-p) \log_2\left(\frac{1}{1-p}\right)\right]$$

Given *k*-bit messages encoded with *n* bits on a channel, information theory tells us that the error rate can be held arbitrarily low by ensuring $\frac{k}{n} \cdot R < C$.

Reliable data transfer

On the link layer, packets may be lost for multiple reasons. The packet may be dropped, because of bit errors, the queue might overflow, there might be routing errors or there might be connectivity errors, like in mobile connections. The solution is to retransmit lost packets.

One approach is to let the receiver ask for a retransmission of a packet whenever that's needed. However, if a packet is lost, the receiver does not know about it and thus will never ask for a retransmission. If a received packet is damaged, the receiver will not know where the error is, and might not know the source of the packet. In general, such packets are simply dropped. This makes it clear that the sender needs to take the initiative for retransmissions, not the receiver.

A simple solution is to retransmit packets after a timeout. The sender sends one packet at a time, and waits for an acknowledgement. If that does not arrive, the sender retransmits the packet. This technique can fail when acknowledgement packets get lost, as the sender will send the same packet twice, possibly without the receiver knowing it, resulting in data corruption.

A solution to that problem is to add sequence numbers. The acknowledgement packets then also describe what packet exactly they're acknowledging. In a link with a large bandwidth-delay product, a larger sequence number space is needed, as the sender will stop sending data when all possible sequence numbers have been used.