CO212 - Networks and Communications

14th January 2020 Week 1, Lecture 1

Evolution of the Internet

Literally only writing this part so I have something for the first lecture.

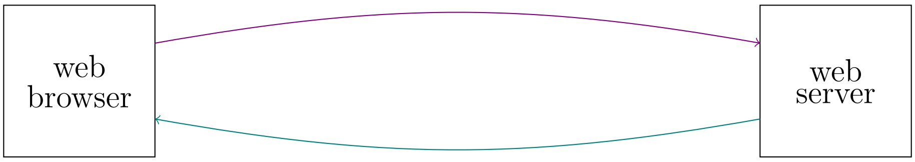
- (1969 - October) first message sent on ARPANET; ”login”, crashed after ”l” and ”o” were sent
- (1971) universities in West and East coast of USA connected
- (1980) London connected

14th January 2020 Week 2, Lecture 1

World Wide Web (WWW)

This is an example of an **application** on the internet, based on HTTP (HyperText Transfer Protocol). A web browser (the client) sends a request to the web server over a pipe, which can be any form of connection between the two devices (can also be the same device), which in turn sends back a response.

```
GET /pixel.gif HTTP/1.1
Host:  www.example.com
...
```



```
HTTP/1.1 200 OK
...
[binary data]
...
```

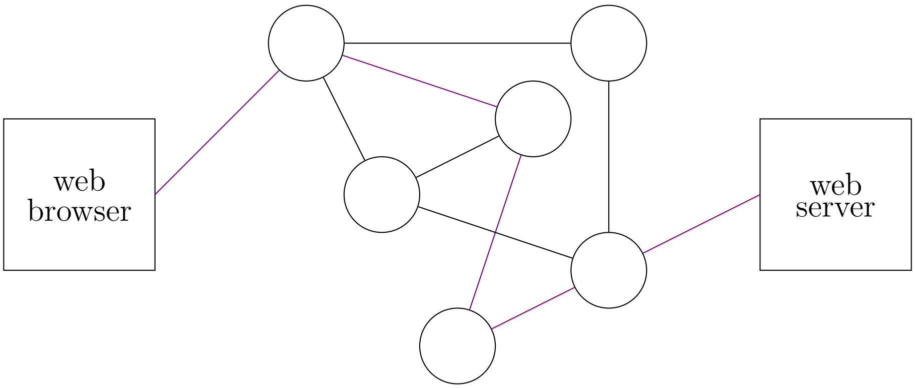
Layers

- **application layer**

  Any software written for the internet is on the application layer.
- **transport layer**

  In the **transport layer**, packets leave your (client) machine to the server, and the server sends back packets to your client. This layer divides a (big) message into smaller chunks, and sends them to the other side (re-ordered) to be presented to the recipient.
- **network layer**

  The **route / path** (sequences of switches a packet goes through) each packet takes can be different from the others, and is often the most optimal route available. This is done on the **network layer**, which routers are a part of.



- **data link layer**

  Our devices are linked to the network on the **data link layer**, via network interface controllers (NICs). Examples of this include Ethernet, fiber optic network cards, as well as wireless devices such as WiFi access points, and USB dongles for 4G. A communication link is any connection between packet switches and / or end systems.

- **physical layer**

  Finally, on the **physical layer**, there are various forms of communication media, including fiber-optic cables, twisted-pair copper wire, coaxial cables, and wireless local-area links (802.11, Bluetooth, etc).

16th January 2020 Week 2, Lecture 2

Circuit Switching

Old phones used circuit switching, which creates a connection between the two points, which is used for the entire communication. This isn't used for the internet as the failure of one node in the circuit would lead the the entire communication dropping - whereas a different route would be calculated in packet switching.

Compared to packet switching, it has an expensive setup phase, but will need very little processing once the connection is established. However, it is inefficient for sharing resources - if a node is used as part of a circuit, it cannot be used by another connection for a different circuit. The resources are blocked once a connection is established (hence it is an inefficient way to use the network). On the other hand, packet switching has no setup cost, but has a processing cost, as well as space overhead, for every packet. It has a processing cost for forwarding the packets, as well as space overhead as there must be redundant data for each packet, such that it is self contained. It is specifically designed to share links, hence it allows for a better utilisation of network resources.

Protocols

A protocol is a set of rules (an agreement between communicating parties on how communication is to proceed), run by end systems as well as packet switches. It must be unambiguous, complete (includes actions and / or responses for all possible situations), and also define all necessary message formats. The phases are as follows;

- **handshake** establishes identities and / or context
- **conversation** free-form exchange
- **closing** terminating the conversation

The internet protocol stack is based on the 5 layers briefly covered in the previous lecture. Some examples of design issues that can be encountered are as follows;

- **addressing** how to denote the intended recipient
- **error control** how to detect (and possibly fix) transmission errors, e.g. checksums
- **flow control** ensure data travels through communication media without issues
- **multiplexing / demultiplexing** conversion of data into binary, and parallel communications
- **routing** which route is chosen

Most network layers are either connection-oriented, where a connection is first established, data is exchanged, and the connection is finally released, or connectionless, where data is marked with its destination.

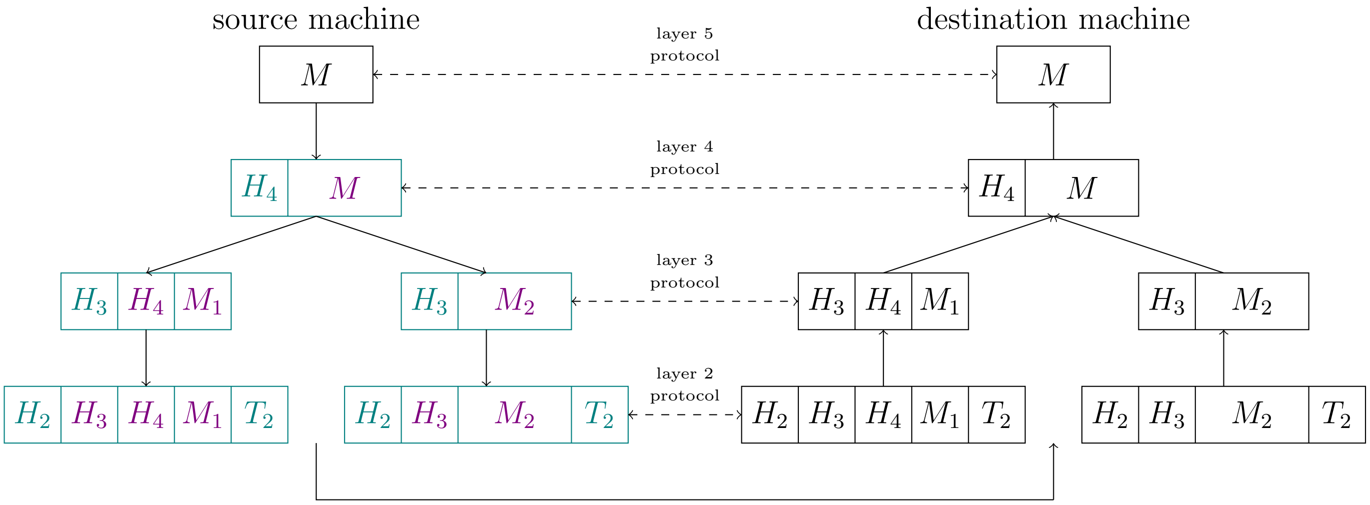
The TCP/IP (4 layer) stack consists of application, transport, internet, and network access (which combines data link and physical). On the other hand, the OSI (7 layer) model consists of the application layer, presentation, session, transport, network, data link, and physical.

application
presentation
session
transport
network
data link
physical

application
transport
network / internet
data link
physical

application
transport
internet
network access

A **service** is a set of primitives that a layer provides to the layer above it, whereas a **protocol** is a set of rules that prescribe the layout and meaning of packets. In a protocol stack, layer k puts its entire packet as data into a layer $k-1$ packet, the latter may add a header and / or a trailer. This may have to be split across several lower level packets (**fragmentation**). An example of protocol layering is as follows;



Note that the connection between the two machines can have multiple nodes in between, that can read up to a different physical layer. For example, if there was a link-layer switch after the source machine, it can read up to the link layer (layer 2), remove and add headers / trailers, and then send it on to the next device. The next device may be a router for example, which can read up to the network layer (layer 3), and do the same.

The data from the layer above is known as the SDU (service data unit), and the SDU combined with a header, added by the current layer, is known as the PDU (protocol data unit).

16th January 2020 Week 2, Lecture 3

Protocol Layers

The types of protocols on each layer are as follows;

- **application layer**

  Protocols on the application layer defines functionality and message formats. Some examples are as follows;

  - **traditional** name services (DNS), sending email (SMTP), file transfer (FTP), web (HTTP(S))
  - **modern**

    includes middleware protocols to support distributed systems with special protocols to handle replication, fault tolerance, caching, etc.
  - **high-level** special application-level protocols for e-commerce, banking etc.
  - **peer-to-peer** BitTorrent, old-Skype (awful API)

- **transport layer**

  These protocols generally offer connection-oriented (TCP) as well as connectionless (UDP) services, which have varying degrees of reliability. These often provide a network interface to application via sockets. It's also important to note there is a difference between reliability and security; the former guarantees that the data is sent, whereas the latter ensures that it is encrypted in some form. This layer also provides flow control; mechanisms to ensure fast senders don't overwhelm slow receivers.

- **network layer**

  In this layer, the protocols generally describe how routing is performed, such as determining which computers / routers are in the network, the best route between two points, how to handle faults (such as a device going down), as well as handling congestion (when a router is overloaded and packets are dropped).

- **data link layer**

  In this layer, we need to detect bit transmission errors. This can be done by adding redundancy bits in frames to detect errors - for example adding a parity bit to every 7 bits, where a 1 indicates an odd number of 1s, and a 0 indicates an even number of 1s, or adding a checksum (cyclic redundancy check) which should match the bits before it.

  It also specifies how many computers can share a common channel, with the medium access control sub-layer (MAC). A well known protocol is the Ethernet protocol.

- **physical layer**

  This describes the transmission of raw bits, in terms of the physical mechanical and electrical issues. For example, when two computers are connected with a wire, -3V may indicate to a binary 1, and +4V may correspond to a binary 0. It may also specify the number of times the voltage can be changed per second.

  For example if the voltage can be changed 20,000 times per second, it indicates that the maximum transfer rate is 20,000 bits per second, which is 20 Kbps (20 kilobits per second).

Units

- 1 Byte = 8 bits note that a byte has an uppercase B, whereas a bit has lowercase b
- 1000 Bytes = 1 KB (Kilobyte)
- 1024 Bytes = 1 KiB (Kibibyte)

Typically we use powers of 10 for networks, but as long as we are consistent in what we use it is fine.

Quantifying Data Transfer

- **bandwidth**

  the amount of information that **can** get into the connection in a time unit

- **throughput**

  the amount of information that **actually** get into the connection in a time unit - at steady-state, we assume zero accumulation of traffic therefore the input and output throughputs are equal

- **latency**

  the time it takes for one bit to go through the connection

Note that for the following formula, we use these values;

- t_0 the time the first packet leaves the source
- t_1 the time the first packet reaches the destination
- t_2 the time all the packets reach the destination
- L the size of the packet in bits
- latency (propagation delay) $d = t_1 - t_0$ (generally $\frac{\text{distance}}{\text{wave propagation speed}}$)

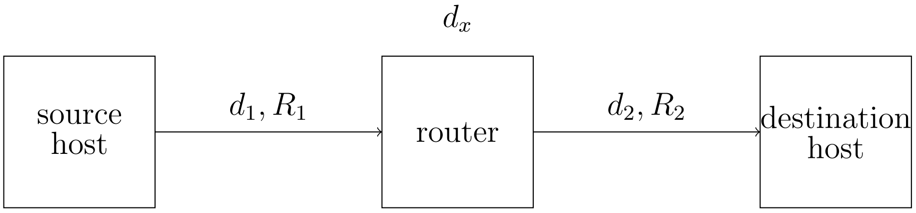
  note that this is half the RTT (round-trip time)

- throughput (link bandwidth) $R = \frac{L}{t_2 - t_1}$ (generally $\frac{\text{transferred bits}}{\text{duration}}$)
- packetization (transmission delay / store-and-forward delay) $\frac{L}{R}$

  time it takes for the entire packet to be received after the first bit is received

- transfer time (propagation delay + transmission delay) $\Delta = d + \frac{L}{R}$

However, it's important to note that the connections are (almost) never direct, and therefore will have additional delays at each router. Note that our bandwidth is also bottlenecked by the lowest bandwidth.



$$
\begin{aligned}
d_{\text{end-end}} &= \begin{cases} d_1 + \frac{L}{R_1} + d_x + d_2 & R_1 < R_2 \\ d_1 + d_x + \frac{L}{R_2} + d_2 & R_1 \geq R_2 \end{cases} \\
&= d_1 + d_x + d_2 + \frac{L}{\min(R_1, R_2)}
\end{aligned}
$$

The router delay, d_x has two components; the processing delay d_{proc} which is the processing time (checking for bit errors and determining the output link), as well as d_q, which is the queueing delay - the time waiting at the output link for transmission, which depends on how congested the router is. We can quantify the **traffic intensity** as follows;

$$
\begin{aligned}
R &= \text{link bandwidth} \\
L &= \text{packet length (bits)} \\
a &= \text{average packet arrival rate} \\
\frac{La}{R} &= \text{traffic intensity}
\end{aligned}
$$

$\frac{La}{R} \approx 0$ small average queueing delay

$\frac{La}{R} \to 1$ large average queueing delay

$\frac{La}{R} > 1$ more working arriving than can be serviced, infinite delay

21st January 2020 Week 3, Lecture 1

Clients and Servers

We can distinguish between the two roles in a pair of communicating processes, in a connection-oriented model;

- **client** initiates communication
- **server** waits to be contacted

On the other hand, some applications have processes that act as both the client and the server, which is referred to P2P (or peer-to-peer) architecture.

End System Applications

Internet applications are processes on the end systems. They must have a way of addressing each other, either via the internet (such as chat servers), or they can directly communicate (such as FTP) - but this depends on the protocol in use. Note that a single end system (or host) can run multiple programs, which run multiple processes, all of which connect through a network API provided by the operating system. Each process is addressed within its host by a **port number**. When an application wants to communicate with another application, the OS opens a socket, which allows data to be transferred between the two machines.

client application

1. create a socket C by connecting to server application (connecting to host H on port P)
2. read and write data to socket C
3. disconnect and destroy C

server application (on host H)

1. create a socket S by accepting connection on part P (port is often called a server socket)
2. read and write data to socket S
3. disconnect and destroy S

The server application can open more sockets to server multiple clients at a time. A DDoS (distributed denial of service) attack works by opening many sockets, preventing the server from serving legitimate requests.

The World Wide Web

Invented in 1989 (formally defined in 1991) by Sir Tim Berners-Lee. Based on the idea of hypertext and hyperlinks (based on a proposal by William Tunnicliffe in late 1960s). Commonly used terminology is as follows;

- **document** a webpage is called a document
- **object** any called within a document (images, stylesheets, etc.)
- **Uniform Resource Locator (URL)** specifies the address of an object
- **browser** also called user agent - client used to access documents
- **web server** application that makes documents and objects available through HTTP

Protocol

In general, the request and reply would include the following;

request

- protocol version
- URL specification
- connection attributes
- content / feature negotiation

reply

- protocol version
- reply status / value
- connection attributes
- object attributes
- content specification
- content

A protocol should always include a version number, as it allows the protocol design to change. HTTP/2 will replace HTTP/1.x in the next few years (hopefully), as the former is able to fully multiplex a connection since all content is binary, and can use a single TCP connection for parallelism.

The URL contains the host name, which determines where the requests goes, by mapping to a network address. The request consists of a request line, such as `GET /path/to/index.html HTTP/1.1`, zero or more header lines, an empty line, followed by the object body (which can be empty). The request line contains a method, such as;

• `GET`	retrieve the object identified by the URL
• `POST`	allows for submission of data to the server
• `HEAD`	similar to `GET` but only receives headers
• `PUT`	requests the enclosed object to be stored under the given URL
• `DELETE`	deletes the given object
• `OPTIONS`	requests the available communication options for the given object

The status code in a server response is generally as follows;

• `1xx`	informational
• `2xx`	successful
• `3xx`	redirection (e.g. object has temporarily or permanently moved)
• `4xx`	client error (e.g. malformed request, unauthorised, object not found, method not allowed)
• `5xx`	server error (e.g. internal server error, service overloaded)

23rd January 2020 Week 3, Lecture 2

How HTTP uses TCP

HTTP uses TCP as it is essentially a file transfer protocol, which needs to be connection-oriented. The first version of HTTP uses a TCP connection for each object, which was an inefficient use of both the network and the operating system. HTTP/1.1 introduced persistent connection, which allows for an existing connection to be used to issue multiple requests (either sending a request, waiting for a response, sending the next request and so on, or through pipelining) - which will eventually close with a timeout. Pipelining allows the client to send all its requests without waiting for a response, and the server delivers them in response.

Web Caching

A proxy is a server which acts as an intermediate between the client and the destination server. This can be used for caching by storing a copy of the content, which reduces load on the origin server, and also allows for lower latency. Data isn't cached for an extended period of time as it can lead to stale data (where old content is served to a user, even after the content is changed on the origin server). Proxies can also protect the clients by providing anonymity, as well blocking malicious content through the use of a single firewall on the proxy. However, this also acts as a single point of failure - if the proxy fails then the clients may not be able to connect.

A `HEAD` request could be used to see if an object has been updated, which is less expensive than retrieving the entire object with a `GET` request. The request can also include `Cache-Control: no-cache`, which indicates it does not want cached objects, thus requiring proxies to go to the origin server, or `Cache-Control: max-age=20`, which only gets a cached object if the cache is less than 20 seconds old.

On the other hand, the reply can also include `Cache-Control: no-cache`, which informs the proxy not to cache the object, or `Cache-Control: maxage=100; must-revalidate`, which specifies to the proxy that it must revalidate the object after 100 seconds.

Sessions

Note that HTTP is a **stateless** protocol, which means that responses have no memory of past requests. However, HTTP allows higher-level applications to maintain **stateful** sessions, via the use of cookies. The `Set-Cookie` header is sent from a server, informing the client to store the cookie as a session identifier for that site. On the other hand, the client sends a `Cookie` header, which tells the server which session the request belongs to. This can be useful for storing identifying a user on a page (allowing for personalised pages). However, this can be also be used to track users for profiling and targeted advertising - leading to privacy issues.

Dynamic Web Pages

Servers often generate pages on-the-fly, instead of only serving statically stored pages. CGI (common gateway interface) allows you to identify a program and its parameters in a URL, which then starts a process to execute the program and return any results as a regular web page. On the other hand, servelets in Java maintain state (whereas CGI is stateless), and the webserver contains an instance of the JVM.

Another approach is for the web page to incorporate interpretable code, which is executed when the page is being processed on the client-side (via JavaScript). It's important to distinguish this from server-side processing in something such as PHP, which the user should not see.

IP and Hosts

Each end system is identified and addressed by its IP address, which is 32 bits in IPv4, or 128 bits in IPv6. This is easy to process by a computer, as it can easily work in powers of two, but not practical for people to use.

Host names are used to create human-readable "aliases" for IP addresses. Originally, before 1983, all mappings were in the `hosts` file, since there weren't many different hosts. However, as more hosts became present, DNS (Domain Name System) was developed, which provides a distributed lookup facility.

There are 13 root DNS servers, which know where the top-level servers are located. The top-level domain servers are each associated with a top-level domain. However, knowing where to connect to requires a large amount of communication between servers. This can therefore be a bottleneck for applications, and also be a critical point of failure. To circumvent this, we can also cache DNS lookups.

This improves performance as we do not have to do as much communication, and it also reduces the overall load on the DNS infrastructure.

However, this can be an issue if enough DNS servers advertise an incorrect lookup, causing subsequent requests to point to an incorrect IP (DNS cache poisoning).

DNS query types are as follows;

- `A` maps a host name to its address, **name** is a host name, and **value** is its IP address
- `NS`

  a query for a name server, **name** is a domain name, and **value** is the authoritative name server for that domain
- `CNAME`

  a query for a canonical name, **name** is a host name alias, **value** is the primary host name
- `MX`

  a query for the mail exchange server **name** is a host or domain name, and **value** is the name of the mail server handling incoming mail

The DNS protocol is connectionless, and runs on UDP port 53. UDP (best effort) is used since it only involves two network packets (request and response), setting up and closing a TCP (reliable) connection every time would be wasteful. If it fails, it can just try again.

Round Robin DNS is a load balancing technique, as it responds to DNS requests with a list of IP addresses instead of a single IP address. The IP at the top of list is returned a set number of times before it is moved to the bottom of the list (the IP that was previously second in the list is now at the top). If load balancing is used, the TTL should be small, as we want this to constantly change depending on server load.

23rd January 2020 Week 3, Lecture 3

Content Distribution Networks (CDNs)

We have the following options when we want to provide a streaming service from millions of videos to many simultaneous users;

1. single, large "mega-server"

   - single point of failure, and point of network congestion
   - long path to distant clients (slow)
   - multiple copies of video sent over outgoing link

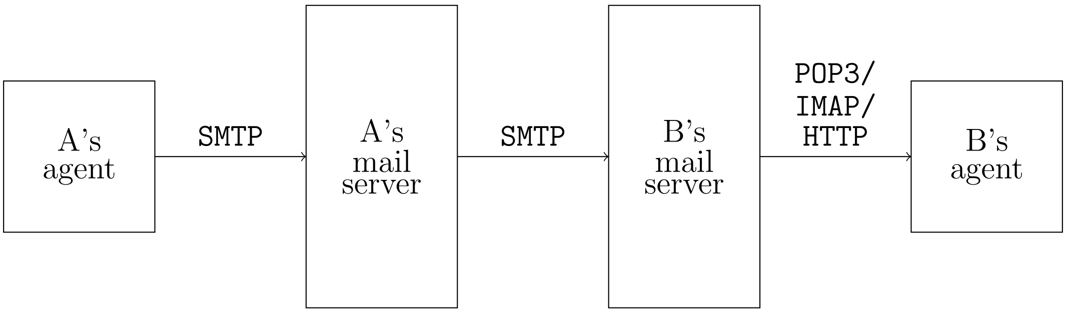
   This solution does not scale to a large amount of users.
2. store multiple copies of videos at multiple geographically distributed sites

   - **enter deep** push CDN servers into many access networks (close to users)
     used by Akamai (216000+ servers, 120+ countries, 1500+ networks)
   - **bring home** smaller number of large clusters in PoPs near (not within) access networks
     used by Limelight (80+ PoPs - Points of Presence)

A CDN DNS can select a good CDN node by picking the CDN node closest to the client, or a CDN node with the shortest delay to the client. However, it's important to note that the CDN doesn't know the IP address of the client, only the address of the local DNS which may not be fully accurate. An alternative is to let the client decide by giving a list of CDN servers. The client can then select the "best" based on the lowest RTT.

Electronic Mail

While email was able to achieve asynchronous communication, one-to-many communication, as well as multimedia content it had a number of limitations. Privacy and security was an issue, as there was initially no authentication - hence messages could be modified or forged, and messages could be read by others. Additionally, it was unreliable as there were no delivery guarantees, and had no reliable acknowledgement system.



The user agent is the client the user uses to read, compose, reply, send and forward messages. The mail server can do the following;

- accept messages for remote delivery (delivers message to remote destination server with transport protocol)
- accept messages for local delivery (saves messages in local persistent mailbox)
- allows user agents to access local mailboxes (to allow for retrieval and / or deletion of messages)

Simple Mail Transfer Protocol

It's important to note that the 'S' in SMTP does not stand for secure (no authentication). This is a connection-oriented protocol (TCP), on port 25. As it is a very simple protocol, it can be left unsecured - this is often targeted by spammers and phishers who use unsecured mail servers to send mail without using their own resources.

The general format of using SMTP is headers, followed by an empty line, and then content. A single dot is used to end the email, and `QUIT` is used to exit. SMTP is completely oblivious to the contents of a message, other than the **received** header, which is added by each receiving SMTP server. This can be used to trace the origins of an email.

The initial message format had many limitations - it only supported 7-bit text content, and was essentially only usable for the English language. The MIME (multipurpose internet mail extensions) format defined useful extensions on top of SMTP, which includes the following types;

• `text/plain`	normal ASCII message
• `text/html`	HTML-formatted message
• `image/jpeg`	contains only an image file
• `multipart/mixed`	consists of multiple parts

Post Office Protocol (POP3) and IMAP

The user's mailbox is often stored on a different machine than the user agent, but we need remote access to incoming (and outgoing) messages. However, POP3 is insecure (at least on port 110), due to the transmission of credentials in plain text. It also implicitly assumes the retrieved mail is deleted at the server, which isn't useful if people wanted to access mail from different clients.

The internet message access protocol (IMAP) solves this, by storing messages on a server, and requiring the client to be online to read mail.

Dark Web

TOR (The Onion Router) hides the user behind a series of machines (proxies), and also allows for access to `.onion` sites (as well as regular ones). This can be used for privacy purposes, as well as for circumventing censorship. The exit nodes of TOR can be owned by law enforcement agencies, allowing for a user to be compromised if they were to subpoena all intermediate proxies.

Not that this shouldn't be confused with the **deep web**, which is typically just not indexed on the surface web.

27th January 2020 Week 4, Lecture 1

Transport Layer

The transport layer provides both reliable connection-oriented services (TCP - transmission control protocol), as well as unreliable connection-less services (UDP - user datagram protocol). This provides for logical communication between application processes, and only runs on end hosts (not routers / switches). TCP data are called segments, whereas UDP data are called datagrams.

We also assume that the underlying layer is working, with every host having a unique IP address, and that IP is a "best-effort" delivery service. This means that it has no guarantees on the integrity of data (or packet) transmission, nor does it guarantee the order of delivery of packets or segments.

Data Encapsulation

For each layer, we have the following terminology - we can't refer to everything as packets;

- application layer data
- transport layer TCP segments or UDP datagrams

  (TCP only) segmentation can be done when the segments are too large - if the UDP datagrams are too large, it cannot be sent
- network / internet layer IP datagrams (or packets)

  similarly, fragmentation can be done when the packets are too large
- data link layer frames
- physical layer bits

Ports

Note that it's possible for a client (one IP) to communicate with the same host (one IP) via multiple applications, such as HTTP and SMTP. Each application on ta host is identified with a unique port number - they can be thought of as cross-platform process identifiers. A socket consists of two pairs of `ip_address` + `port_number` + `TCP/UDP`; such as

`146.179.40.24:80 TCP` ⇔ `192.168.1.1:7155 TCP`

The first 1024 ports (0 - 1023) are reserved, and cannot be used to form a connection as a client (unless it is done as a superuser).

Transmission Control Protocol

TCP is the Internet's primary transport protocol. It is a connection-oriented service, and endpoints initially perform a handshake to establish a connection. This is a full-duplex service, thus both endpoints can send a receive at the same time. Some definitions for TCP are as follows;

- **TCP segment** "envelope" for TCP data

- **maximum segment size (MSS)**

  maximum amount of application data transmitted in a single segment (does not include headers) - typically related to MTU to avoid network-level fragmentation

- **maximum transmission unit (MTU)** largest link-layer frame available to the sender host

  path MTU discovery (PMTUD) is used to determine the largest link-layer frame that can be sent on all links from the sender to the receiver

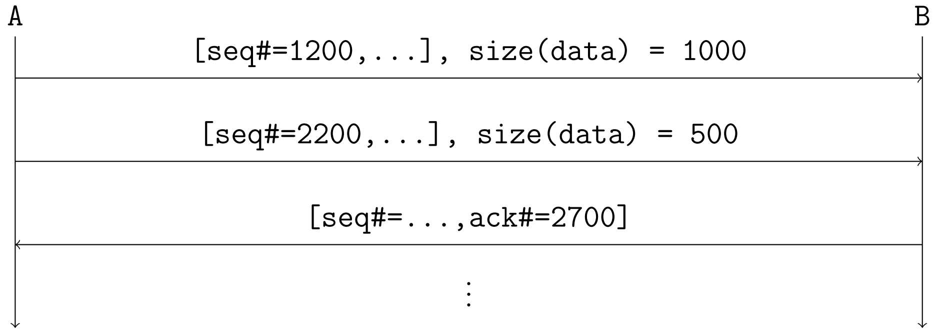
The TCP header consists of the following fields;

- source and destination ports 16-bit each (identifies applications)
- sequence number 32-bit (used to implement reliable data transfer)

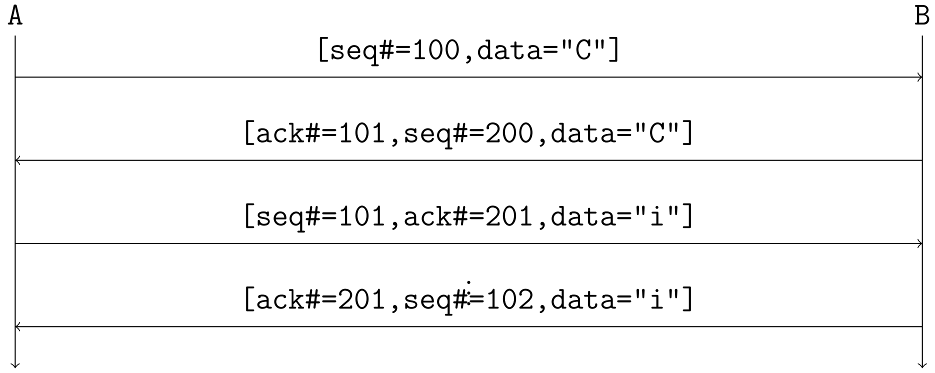
  These numbers are not associated with the segments, but instead are associated with the bytes in the data stream. It indicates the sequence number (the place) of the first byte carried by the TCP segment. When the connection is initialised, a random ISN (initial sequence number) is decided upon to avoid accidentally receiving leftover segments.

- acknowledgement number 32-bit (used to implement reliable data transfer)

  Represents the first number not yet seen by the receiver (one higher than the sequence number of the last bit received). These can be cumulative, and typically TCP implementations acknowledge every other packet;


A
B
[seq#=1200,...], size(data) = 1000
[seq#=2200,...], size(data) = 500
[seq#=...,ack#=2700]
⋮

  Note that because a TCP connection consists of a full-duplex link, there are two streams, hence two different sequence numbers (see the example below, of an ”echo” application). Acknowledgements are ”piggybacked” on data segments.


A
B
[seq#=100,data="C"]
[ack#=101,seq#=200,data="C"]
[seq#=101,ack#=201,data="i"]
[ack#=201,seq#=102,data="i"]

- receive window 16-bit (size of window on receiver end)
- header length / offset 4-bit (size of TCP header in 32-bit words)
- optional fields variable length (may be used to negotiate protocol parameters)
- `URG` flag 1-bit (informs receiver some data is marked as urgent)

- ACK flag 1-bit (value contained in the acknowledge number is a valid acknowledgement)

  See **acknowledgement number** above, and the **three-way handshake** section.
- PSH flag 1-bit (solicit receiver to pass data to application immediately)
- RST flag 1-bit (used during connection setup and shutdown)
- SYN flag 1-bit (used during connection setup and shutdown)

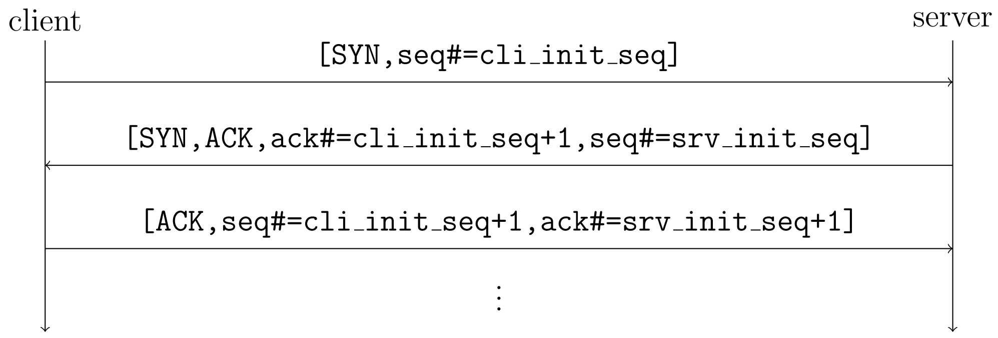
  See the **three-way handshake** section below.
- FIN flag 1-bit (used during connection shutdown)

  A client sends a TCP segment with FIN set, and the server responds with ACK, and then the server sends a FIN TCP segment, client then responds with ACK. Closing a connection is simpler than initialising one.
- checksum 16-bit (used to detect transmission errors)

Three-Way Handshake

The three steps are as follows;

1. **client** sends a TCP segment with SYN set, and an initial sequence number
2. **server** responds with another TCP segment with SYN set, as well as ACK, the first unseen client sequence number, and the ISN for the server
3. **client** reserved with a TCP segment with ACK set, the first unseen server sequence number, and the client's new sequence number


client
server
[SYN,seq#=cli_init_seq]
[SYN,ACK,ack#=cli_init_seq+1,seq#=srv_init_seq]
[ACK,seq#=cli_init_seq+1,ack#=srv_init_seq+1]
⋮

User Datagram Protocol

UDP only provides the two most basic functions of a transport protocol, which are application identification (multiplexing and demultiplexing), and an integrity check via a CRC-type checksum. There is no flow control, no error control, nor any retransmissions. The datagrams cannot be larger than 65K, there is no segmentation, and the router will just drop it. The 65K consists of 20B IP header, 8B UDP header, 65507B of data, coming to a total of 65535 bytes. In practice, only 500 to 1000 bytes are used (the smaller the datagram, the more likely it will arrive intact).

Instead of just using IP, it adds port numbers on top of it, allowing us to differentiate between applications. This is a connection-less protocol, therefore there is no need to connect (just send the data) but each datagram packet must carry the full address and port of the recipient. The UDP header consists of the following fields;

- source and destination ports 16-bit each
- length of data 16-bit
- checksum 16-bit

Any application where we care more about speed, we can use UDP, since it is generally faster (due to the lack of connection establishment), and also has a smaller packet overhead.

Berkeley Socket Interface

The **Berkeley** socket interface is as follows;

- SOCKET — create a new communication endpoint
- BIND — attach a local address to a socket

  the client and server each bind a transport-level address and a name to the locally created socket
- LISTEN — announce willingness to accept N connections

  server starts listening on this socket, thus telling the kernel it will wait for connections from clients
- ACCEPT — block until a remote client wishes to establish a connection

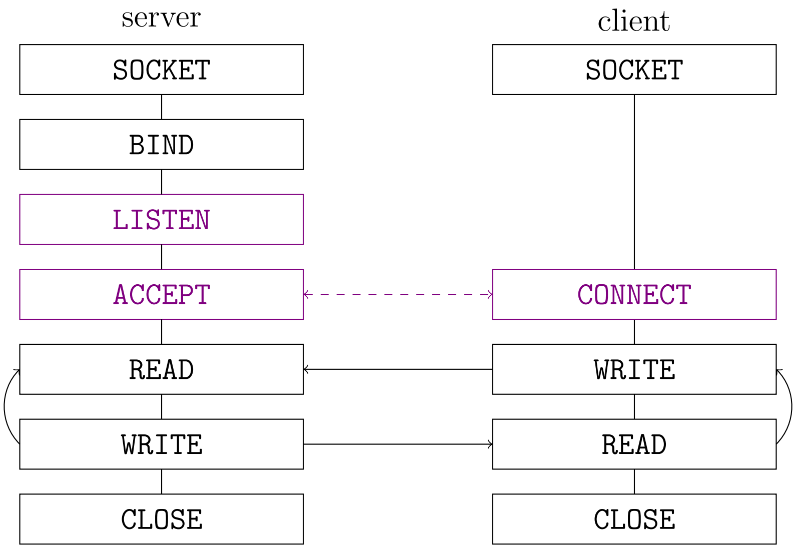
  this blocks the current thread, thus it is a synchronous operation

  from here the server can accept or select connections from clients
- CONNECT — attempt to establish a connection

  a client connects to the socket, it needs to provide the full transport-level address to locate the socket
- SEND — send data over a connection
- RECEIVE — receive data over a connection

  now the client and server communicate through these operations on their respective sockets
- CLOSE — release the connection

  end the communication, must be closed otherwise connections may be continued, and may run out of ports



Note that the part in violet would not be present for UDP.

30th January 2020 — Week 4, Lecture 2

TCP vs UDP

The scenario is as follows; movie player app is being built that allows (paying) users to stream public domain movies. Since the users are paying, we should be using TCP, as it guarantees quality of service, whereas UDP does not. The difference in performance (TCP being slower than UDP) can be circumvented by pre-buffering (sending some data in advance for the client to store locally). This however doesn't work for live events - we can either switch to UDP, or introduce a delay between the feed and the live event.

QUIC

QUIC (Quick UDP Internet Connections) is a new layer 4 protocol, implemented by a Google engineer in 2012. Originally, it was designed for general purpose, but now it is being used for HTTP (layer 5). This is still a draft, but is actively being used by some web servers.

Finite-State Machines

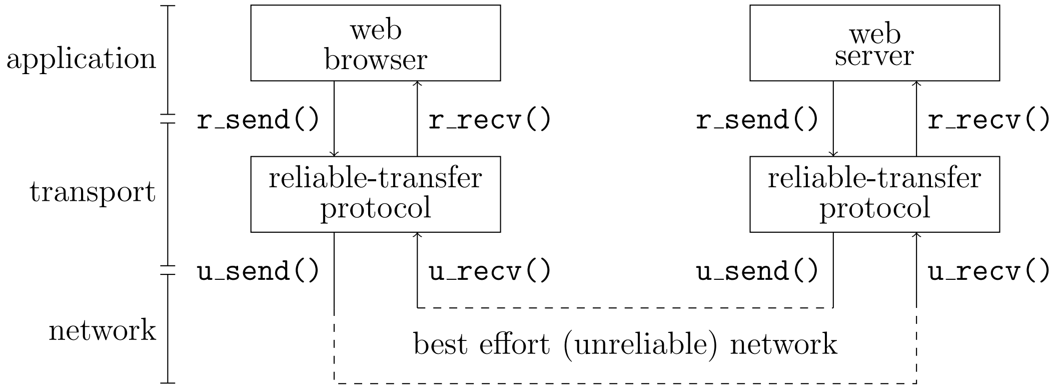
A finite-state machine (FSM) is a mathematical abstraction, where states are represented as nodes, and transitions are directed edges between states, labelled with events. They are also known as finite-state automaton (FSA), deterministic finite-state automaton (DFA), and non-deterministic finite-state automaton (NFA) - multiple edges with the same label from one node. This can be used to specify protocols, where the states represent the state of a protocol, and transitions are characterised by an event / action label (event typically consists of an input message or timeout, whereas an action typically consists of an output message).

$$\frac{\text{input / timeout (event)}}{\text{output (action)}}$$

Examples of this are in the slides, for the TCP state machines for both the client and server.

Reliable Data Transfer

TCP adds a reliable channel on top of an unreliable channel (IP) which is "best effort" delivery. The service provided by the transport layer sits on top of the service provided by the network layer.



Another issue is transmitting data through a noisy channel. There is a chance no packets will be lost, but a a bit may be modified during the transmission (flipped). The stages of dealing with this are as follows;

- **error detection** receiver must know when a received packet is corrupted

  Some strategies for detecting errors are as follows;

  - sending redundant information

    sending information twice, report an error if two different messages received - inefficient
  - error-detection codes

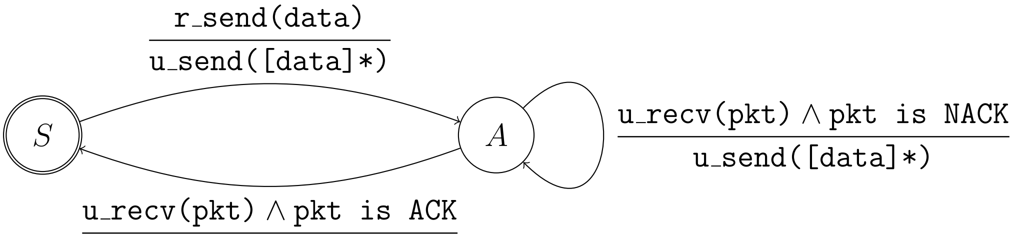
    parity bit, adding a single bit at the end that is the XOR of all other bits in the message, if the recomputed XOR (by receiver) is different, then an error occurred

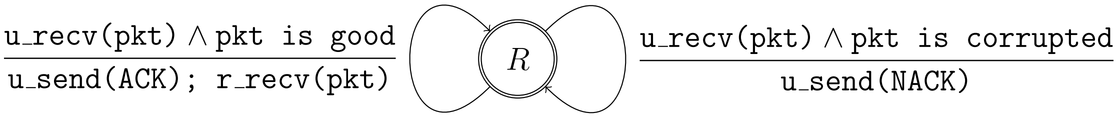
- **receiver feedback** receiver must be able to alert sender that a corrupted packet was received

  `ACK`s and `NACK`s are also protected with an error-detection code, with corrupted `ACK`s being treated as `NACK`s. This may possibly generate duplicate segments, however sequence numbers allow the receiver to ignore these data segments.

- **retransmission** the sender must retransmit corrupted packets

Note that in the diagram below, [data]* indicates a packet containing data and an error-detection code, and A represents the acknowledgement state.

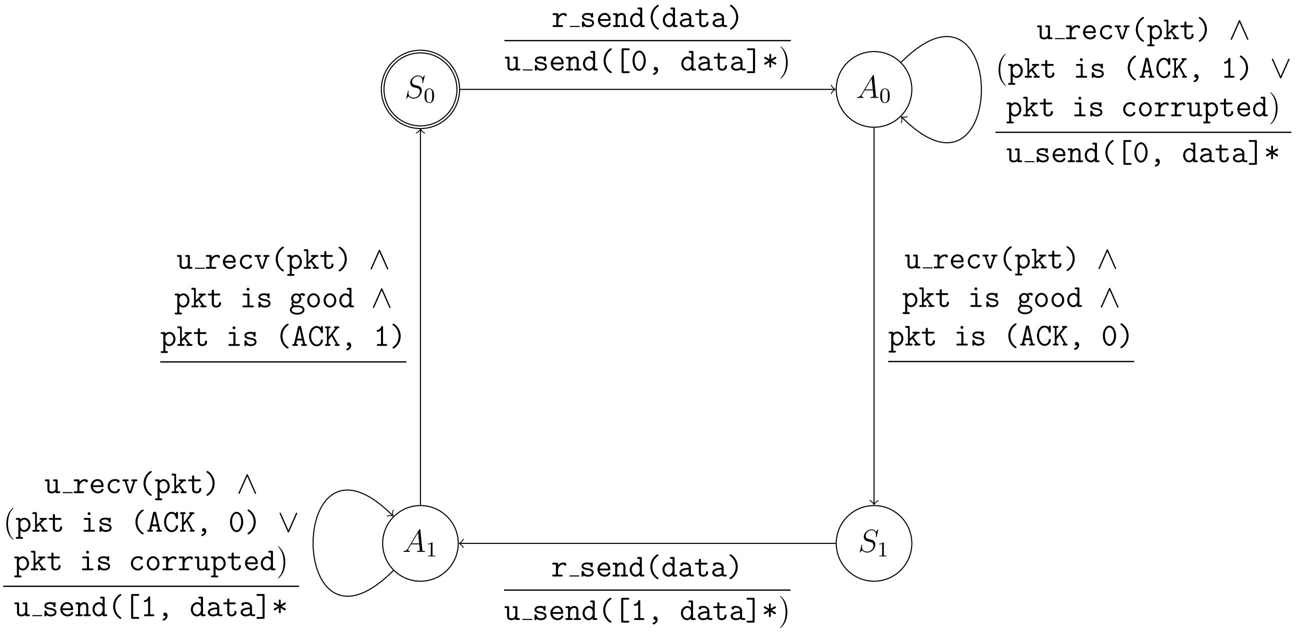


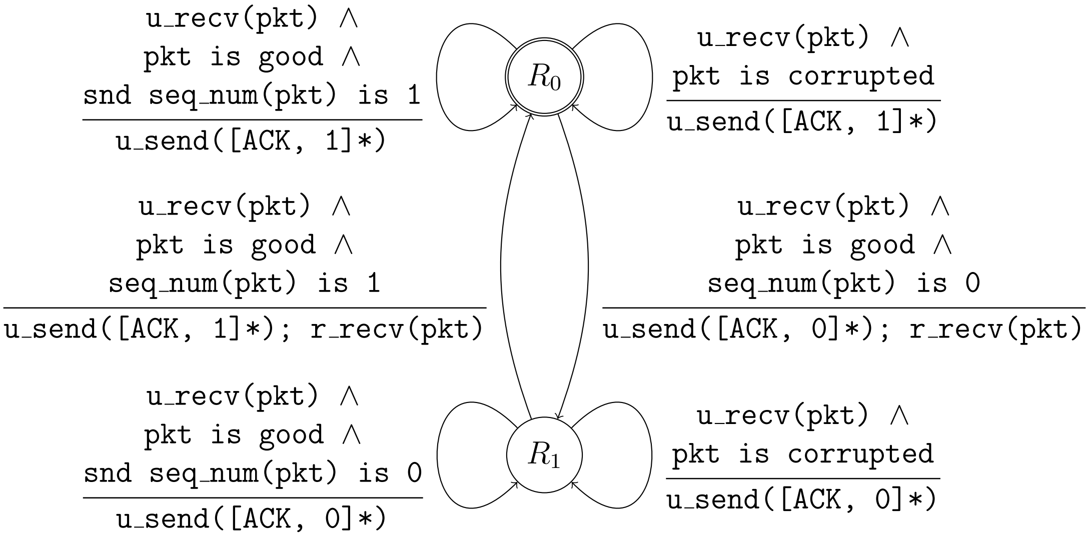


This is a synchronous (or stop-and-wait) for each segment, as the sender must receive a positive acknowledgement before it can take more data from the application layer. However, this can have issues when the ACK/NACK packets have issues. A way around this is for the sender to add a sequence number to each packet, thus allowing the receiver to determine whether a packet is a retransmission. If the sender does not hear an acknowledgement, due to some failure, it assumes a NACK, and retransmits. However, if the receiver did actually send an acknowledgement, it would ignore the packet, since it knows it is a retransmission. It then resends the ACK, and doesn't process the retransmitted.

Since this is a stop-and-wait protocol for each segment, only one bit is needed for the sequence number - as all we need to do is distinguish between the next segment and the retransmission of the current segment.

However, now that we have sequence numbers, we don't need both ACK and NACK, since we can convey the same semantics by sending an ACK for the last good packet it received.





Acknowledgement Generation

The following are cases of segment arrival, and how the protocol handles them;

- arrival of in-order segment with expected sequence number (all data up to expected sequence number already acknowledged)

  delayed ACK, wait up to 500ms for another in-order segment, if it doesn't arrive, send ACK for just this segment
- arrival of in-order segment with expected sequence number (one other in-order segment awaiting ACK, from above case)

  immediately send cumulative ACK for both segments
- arrival of out of order segment with higher-than-expected sequence number (gap detected)

  immediately send duplicate ACK - the sender will know to fill in the gap, as it would've already seen this acknowledgement
- arrival of segment that fills a gap in the received data

  immediately send ACK if segment starts at lower end of gap

Additionally, it's important to note that lost packets can easily be treated as corrupted packets. Since I don't really want to draw the FSM again, add a start_timer() after every u_send(...), and an additional transition loop to A_0 and A_1, which sends the same data, but the input is timeout.

There's also a small section after this on the alternating bit protocol, which isn't used in practice.

30th January 2020 Week 4, Lecture 3

Congestion Detection

If the queue of one or more routers between the sender and receiver overflow, we call it **congestion** - this has the visible effect of segments being droppped. The server can assume the network experiences congestion wheen it detects a segment loss; either on a timeout, hence no ACK, or multiple acknowledgements, which is equivalent to NACK.

TCP we've described so far is referred to as TCP Reno. Another popular implementation is TCP Vegas, which aims to detect congestion before losses occur. It does this by predircting imminent packet loss via observing the RTT - the longer it is, the greater the congestion in the routers. However, flows with small RTTs are advantaged, compared to the ones with large RTTs, as their window can grow faster. TCP CUBIC circumvents this bym making the window increase as a function of time rather than RTT.

Congestion Window and Congestion Control

The conguestion window, W, is maintained by the sender, and limits the amount of bytes that the sender pushes into the network before it blocks to wait for acknowleedgements.

$$\text{LastByteSent} - \text{LastByteAcked} \leq W = \min(\text{CongestionWindow}, \text{ReceiverWindow})$$

This gives a resulting maximum output of roughly

$$\lambda = \frac{W}{\text{RTT}}$$

The phases are as follows;

- **slow start**

  The inital value of W is the maximum segment size (MSS), which is quite low for modern (high-speed) networks. To result in a good throughput, TCP increases the sending rate exponentially for its first phase. It increases W by 1 MSS on every (positive) segment acknowledgement (doubles each time - thef first `ACK` increases it by 1 MSS, the second `ACK` will be after sending 2 segments, hence it increases by a further s2 MSS, and so on), until it reaches some slow strt threshold (ssthresh), or until congestion occurs. If W is greater than (ssthresh), then use congestion avoidance (if they are equal, either could be used).

- **congestion avoidance**

  TCP then increases W linearly in this phase.

  $$W = W + \underbrace{\text{MSS} \cdot \frac{\text{MSS}}{W}}_{\text{increment}}$$

  This happens until congestion is detected.

- **Additive-Increase / Multiplicative-Decrease (AIMD)**
  - at every good acknowledgement, increase W in accordance with congestion avoidance ($\frac{\text{MSS}^2}{W}$)
  - at every packet loss event, TCP halves the congestion window

    There are two possibilities for packet loss. TCP provides reliable ddata transfer by using a timer to detect lost segments. If there is a timeout, without an `ACK`, it means that there is likely a lost packet, hence a retranismission is required. This timeout interval, T, must be larger than the RTT to avoid unnecessary retransmissions, however it shouldn't be too far from RTT, as we want to detect and retransmit lost segments as quickly as posdible. TCP sets its timeouts as follows, where (1) is the estimated RTT, and (2) is the variablility estimate;

    $$T = \underbrace{\overline{\text{RTT}}}_{(1)} + 4 \cdot \underbrace{\overline{\text{DevRTT}}}_{(2)}$$

    The timeout is controlled by contimously estimating the current RTT.

    The agreement is that 3 duplicate `ACK`s (hence 4 identical `ACK`s overall), are interpreted as a `NACK` in **fast retransmit**. While both timeouts and `NACK`s signal a loss, they say different things about the status of the network, hence TCP should react differently. A timeout indicates congestion, whereas the duplicate `ACK`s suggests the network is able to still able to deliver segments along that path.

    In the formulae below, assume the current window size is $W = \overline{W}$;

- timeout

  In the event of a timeout, set $W = \text{MSS}$, and let