COMP3331: Networks and Applications

Haeohreum Kim (z5480978)

I. Introduction to the internet

A. What is the internet?

A ground up explanation.

At the *edge* of the internet, there are **billions** of connected devices. These devices are clients and servers, which have very specific meanings.

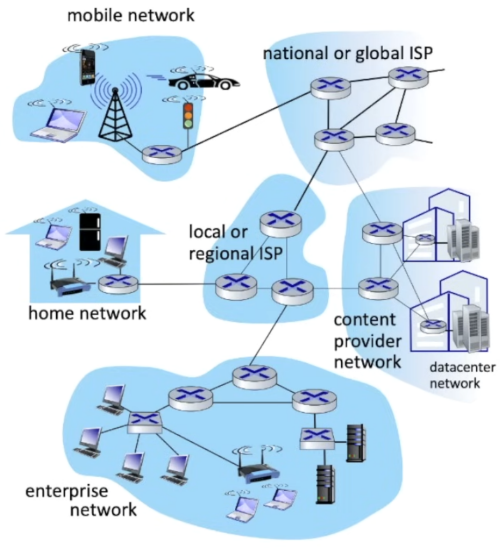
- Computers
- Refigerators
- Gaming consoles
- Security cameras
- and so much more...

Going deeper, we find *packet switches* - these forward packets (chunks of data) across the network. There are two main types: **routers** and **switches**.

Between these packet switches, we have communication links - fiber, copper, radio and satellite. The transmission rate (the speed at which packets are transferred) between these links is what we call *bandwidth*.

And finally, we have this abstract idea of *networks*, which are managed by an organisation and are a collection of devices, routers and links.

Protocols are everywhere - and they control the receiving and sending of data across the network defined above. There must be standards that all these devices, routers and links agree on - being the Request for Comments (RFC).



A 'service' view. The internet, at the end of the day, is the infrastructure that provides services to applications. Furthermore, they provide a programming interface (API) to distributed applications:

- "Hooks" allow sending/receiving apps to connect to, using the Internet transport service
- Providing service options, analogous to postal services

B. What is an internet protocol?

Between humans, we have protocols. Languages themselves are specific protocols with rules, grammar, words and more so that we understand eachother. Internet protocols are the same!

For devices to be able to communicate to eachother, they need to have some agreed protocol of communication.

C. The network edge

The *network edge* refers to end points, like devices, modems and the devices that connect them. This includes *access networks*.

Access networks and physical media are generally the first point of contact for our devices to reach the 'Internet'. These can be residential (like our routers), institutional (like at university) or mobile (like 5G).

D. Wired access

Cable access networks. A cable access network connects multiple devices through a cable headend. Different households are split using *frequency division multiplexing (FDM)*, where different frequencies are used for different purposes.

Cable access networks are generally asymmetric - that is, the downstream (or download speed) is faster than the upstream (or upload speed). This is because most people are larger consumers than producers of data.

Digital Subscriber Line (DSL). uses existing telephone lines to a central office. DSL is also asymmetric. Data over the DSL line goes to the Internet, whereas voice goes to the telephone net.

The home network is generally powered by one of the above two - which is then connected to a router which distributes connections either wired (through Ethernet) or wireless.

E. Wireless access networks

Wireless Local Area Networks (WLANs). generally have a smaller proximity (10-100m), with either 11, 54 or 450 Mbps transmission rates.

Wide-area cellular access networks. provided by mobile, cellular network operators (10's km). 10's Mbps (but with 5G has increased this to gigabyte speeds).

F. Packets and hosts

A host has some data it wants to send. The host breaks up the data into chunks called *packets*, of length `L` bits. The packet will have some extra data called a *packet header*.

Overhead.
We call `overhead` the percentage of data in the packet that is not payload. That is if 40 bits of a 1000 bit packet were headers, then we have a 4% overhead.

The host transmits this packet through some medium at a transmission rate `R`.

Of course, the time taken is then $\frac{L}{R}$.

G. Physical media

Some important terminology and concepts

- Bit: A partition of data that propagates between transmitter/receiver pairs
- Physical link: what lies between transmistter and receiver
- Guided media: signals propagate in solid media: copper, fiber, coax
- Unguided media: signals propagate freely: e.g radio
- Twisted pair (TP): two insulated copper pairs in a wire - 100 Mbps to 10Gbps speed
- Coaxial cable: two concentric copper conductors, bidirectional, 100's Mbps per channel
- Fiber optic cable: glass fibers carrying light pulses, with each pulse being a bit. High speed and low error.
- Wireless radio: signal carried in various bands - and is 'broadcast'; any receiver can technically receive it. There are challenges with propagating waves.

H. Network core

I. Packet switching

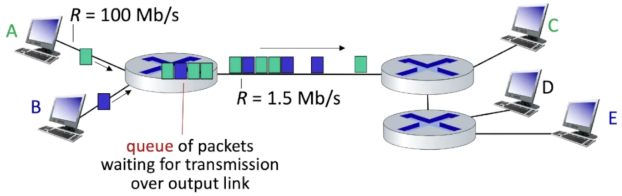
Hosts break application-layer messages into packets. These packets are then forwarded from one router to the next along a path until it reaches the destination.

The network core has two main functions:

1) Forwarding: forwarding is a local action - it moves arriving packets from a router's input link to the appropiate output link.
2) Routing: routing is a global action - which determines the source-destination paths taken by packets, which requires *routing algorithms*.

So routing requires knowledge beyond the local network - it needs to find a path; forwarding is done without the knowledge beyond the device the packet is currently at. Intermediate routers along a route are called **hop routers**. Packets **store and forward** packets by waiting for their entire transmission, and then forwarding them onto the next destination.

When multiple devices try to transmit packets through an access network at a faster rate then it's transmission rate, then **queuing** will occur. A problem occurs when packets are queued and router's memory fill up. In this case, packets are dropped, causing the problem of *packet loss*.



J. Circuit switching

Resources are reserved through a 'call' - a path is determined and then that path is reserved. This creates a direct link from source-destination; there's no loss, and no delay. However, since resources are entirely reserved for some source-destination, resources are largely wasted.

1) **Frequency Division Multiplexing (FDM)***:* Optical, electromagnetic frequencies are divided into narrow frequency bands. Each call is allocated it's own band, and can transmit at the max rate of that band.

2) **Time Division Multiplexing (TDM)***:* Time is divided into slots - each call is allocated a periodic slot, and can transmit at maximum rate (wide band) for the designated time slot.

K. Packet switching v.s circuit switching

Consider a router that routes N users. They all use 100 Mb/s when active, but are only active 10% of the time. The

router has a 1 Gbps link to the internet. With 100 Mb/s, we need 10 users to fill the entire link, which means we need to find the probability that more than 10 users are on the network at the same time. This is given by:

$$P(\text{Network overloaded}) = 1 - \sum_{i=1}^{10} \binom{N}{i} \left(\frac{1}{10}\right)^i \left(\frac{9}{10}\right)^{N-i}$$

The probability ends up being 0.0004. This means it's extremely unlikely, and was a major argument for the usage of packet switching.

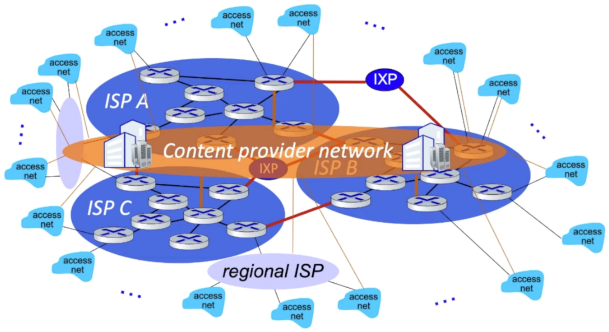
So is packet switching a slam dunk win?

- Packet-switching is great for bursty data; so data that is sent at an irregular interval
- Conge stion is certainly possible
- Can we provide a consistent stream from source-destination like circuit switching using packet-switching?

Having direct end-to-end paths between every single access network is not feasible. Rather, we create smaller networks. On the internet, hosts connect to Internet Service Providers (ISPs). ISPs connect mutliple access networks into a shared network.

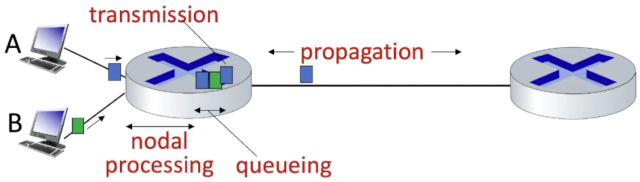
Of course, there are multiple ISPs. ISPs peer through eachother at points called *Internet Exchange Points* (IXP). Different companies that provide different ISPs to different access networks must still be able to communicate across eachother, so they create agreements to have peering locations.

Content provider netowrks, like Google and Facebook, are private networks that connect it's data centres to the Internet, bypassing ISPs. They do this with *Content Delivery Neteworks* (CDN), which are edge servers for customers, but also have agreements with ISPs to connect to IXPs.



L. Network performance

M. The four sources of packet delay



Packet delay is defined by:

$$d_{nodal} = d_{proc} + d_{queue} + d_{trans} + d_{prop}$$

- d_{proc}: processing delay - forwarding, error checking, ¡ microsecs
- d_{queue}: time waiting at output link for transmission
- d_{trans}: defined by L/R for L amount of bits at rate R.
- d_{prop}: length of the physical link, divided by the propagation speed.

Consider the following formula for *traffic intensity*:

$$t = \frac{L \cdot a}{R}$$

where L is the length of the packet, a is the average arrival rate of packets, and R is the bandwidth. We see as 1 is the average queue time, and < 0 as smaller queueing delays. Where $aL > R$, queueing occurs.

N. Tracking real delays in transmission (traceroute)

Using the tool `traceroute` allows us find the route of a packet, as well as the delay of a packet along it's route for each hop-router.

O. Throughput

Definition. Throughput is the rate at which bits are being sent from sender to receiver. It is given as a *rate* (bits/sec) at which bits are being sent from sender to receiver:

- instantaneous: rate at given point in time
- average: rate over longer period of time

$$\frac{\text{number of bits sent}}{\text{total time taken}}$$

You must consider the bottle necks within the network - and find where the transmission rate is minimised. Let's consider a situation where some packets are being sent through two links, with rates R_S and R_C. The throughput will be given by $\min(R_S, R_C)$.

What if we have 10 devices going through their own respective links R_{S_i}, into one shared link to the destination R_d? Then our throughput is $\min(R_{S_i}, R_d/10)$.

II. Application layer

The application layer is how users and software access the network.

Before we step into the application layer, what is a layer? Each layer *implements a service*. Of course, these layers interact with eachother to provide the whole experience when we use the internet.

But why do we layer?

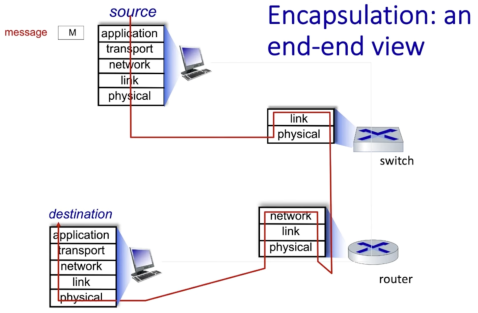
- Explicit structure allows identification and relationship of system's pieces
- Modularisation eases maintenance, and the updating of the system

Remember *shotgun surgery* from COMP2511. That's what we want to avoid! **What layers does the internet have?**

- Application: supports network applications
- Transport: Process-process data transfer
- Network: routing of datagrams from source to destination
- Link: data transfer between neighboring network elements
- Physical: physically transfers the bits between the links

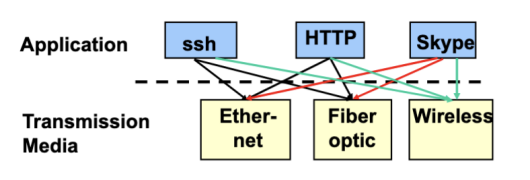
Definition. The application layer exchanges messages to implement some application service using services of the transport layer

As data exchanges from application layer to application layer, it of course first goes down the layers of one source. As the data passes down, headers are added at each layer (besides the physical); we call this *encapsulation*.



The importance of encapsulation.

Consider the alternative to layering. If there were no layering principle in networking, every single application would have to be reimplemented for every network technology.



Layering implements ways to pass any application through the same layer, by the usage of *headers*. This is the cost of layers.

Hosts, routers and switches; who implements what?.

Across the internet, there are three main participants: **hosts, routers and switches**. Who must implement what layers?

- Hosts
  - Hosts have applications. Application layer.
  - Applications must send segmented data. Transport layer.
  - Transport layer must send packets. Network layer.
  - Network layer must send bits. Passes to data link.
  - Data link passes to physical.

  So hosts must implement all layers.
- Routers and switches
  - Bits arrive on wire to routers. Physical layer.
  - Routers have local delivery (forwarding).
  - Routers participate in routing. Network necessary.
  - Routers don't need to segment. No transport.
  - Routers don't have applications. No application.

A. *Client-server paradigm*

Server:

- Always-on host
- Permanent IP address
- often in data centers, for scaling

Client:

- contact, communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do *not* communicate directly with eachother

Examples of client-server protocols: HTTP, IMAP, FTP

In a client-server architecture, processes on each of the client and server's machines are communicating with eachother. That is the client process initiates contact, and the server process responds.

Sockets.

- processes send and receive messages to and from it's socket
- Sockets are like doors; sending sockets send processes out the door, and receiving through it.
- There must be a sending and receiving layer.

Addressing processes.

- to receive messages, processes must have identifiers
- identifiers must have an **IP address** and **port number**
- but, some port numbers are for specific services.

Required transport services.

- Data integrity; how much data reliability do we need?
- Timing; how fast does the data transfer have to be?
- Throughput; how much minimum bandwidth do we need?
- Security; does our data have to be secure?

Transport protocol services.

- TCP: reliable data, connection oriented, flow and congestion controlled
- UDP: unreliable data, a lot less controlled than TCP. These services can however be built ontop of UDP.

B. Peer-peer architecture

- No always-on server
- arbitrary end systems directly communicate
- peers request service from other other peers, provide service in return to other peers
- peers are intermittently connected, and have dynamic IP addresses

Example: P2P file sharing (torrents!)

C. Web and HTTP

A little bit about the web, and URLs.

The `World Wide Web` is a *distributed database* of pages linked through Hypertext Transport Protocol (HTTP). Webpages, are defined by Uniform Resource Locators (URL).

`protocol://host[:port]/dir/res`

- protocol: http, fttp, https, etc.
- host: DNS name, IP address
- port: defaulst to protocol's stand port (e.g `http: 80`)
- dir: hiearchical directory, reflecting file system
- res: resource in directoy `dir` client wishes to access

HTTP: hypertext transfer protocol.

- Web's application layer protocol
- Applies the client-server paradigm
  - client: browser that requests, receives then displays Web objects
  - server: web server sends (using HTTP protocol) objects
- HTTP uses the TCP protcool
  - client initiates TCP connection to server, using port 80
  - server accepts TCP connection
  - HTTP messages are then transacted
  - TCP connection is closed
  - HTTP is stateless - server maintains no information about past requests

There are three flavours of HTTP connections

1) Non-persistent: : TCP connection opened, at most one object is sent, and then TCP connection is closed.

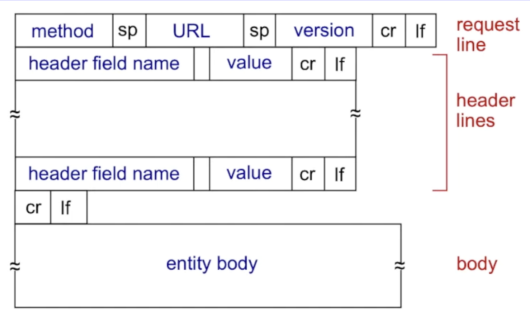
2) Persistent: : TCP connection opened, multiple object can be sent over single TCP connection, and then TCP connection is closed.

3) Persistent with pipelining: : TCP connection opened, multiple objects are requested concurrently over a single TCP connection, and then TCP connection is closed.

Round trip time (RTT): time for a small packet to travel from the client and back

By leaving the connection open, persistent connections can save RTTs for future messages, as a new 'acknowledgement' request is not required for the TCP connection - cutting the RTTs required by half.

Anatomy of a HTTP request message.



Do note that:

- The body is `text`, so each character uses a `byte`.
- That means, `256` uses 3 bytes.

The four main types of HTTP requests:

- GET: for sending data to server
- POST: for user input
- PUT: to upload new files to the server
- HEAD: request headers that would be returned if a GET request was sent instead

D. Cookies

Cookies. Web sites and client browsers use cookies to maintain some state between transactions. There are four components to use cookies

- cookie header line of HTTP response message (sent to client)
- cookie header line in next HTTP request message (sent to server from client)
- cookie file kept on user's host, managed by user's browser (state)
- back-end database at website

User states are of course extremely useful, and can be used for:

- authorisation
- shoping carts
- recommendations
- user session state

E. HTTP Performance Improvements

Web caches/Proxy Servers.

We'd like to limit repeat requests where possible, and to avoid going all the way to the host server.

- User configures browser to point to a local web cache.
- Browser sends all HTTP requests to the cache
  - if the objects in the cache, cache returns object to client
  - else, cache requests object to the origin server, and then forwards it along to client.

Essentially, proxy servers are a middle man between the client and server, that hopes to return cached information.

The origin server would tell the cache how long to cache the data for. The cache serves both the server and the client role.

Conditional GET.

Don't send object if cache already has an up-to-date cached version. This will entirely mitigate transmission delay, as no request is sent at all!

- The client sends a date it would like to update after `if-modified-since`.
- If the cache already has a copy that is recent enough, it well send a 304 Not Modified response. No request to the server will be sent
- If a update is required, a normal request will be sent, and a 200 OK will be sent back pending a successful request.

ETags and Conditional GET.

`ETags`, are essentially identifiers for a web object. Conditional GETs can also utilise ETags to check whether an object of the ETag x, exists, and if it does not exist, to fetch a new copy from the server.

HTTP/1.1 and HTTP/2.

`HTTP/1.1` introduced multiple, pipelined GETs over a single TCP connection, which allowed for concurrent HTTP requests.

- The server responds to requests *in-order*
- Since it is first come, first serve, smaller objects that could be processed might wait behind large objects (`head-of-line (HOL)` blocking)
- Loss recovery (due to `Go-Back-N`) stalls object transmission.

`HTTP/2` aims to decrease delay in multi-object HTTP requests. There is increased flexibity at `server` in sending objects to clients

- methods, status codes, etc. are unchanged
- transmission order of requested objects can be based on client-specific priority
- push unrequested objects to clients
- divide objects into frames, schedule frames to mitigate HOL blocking - so that large objects doesn't necessarily block small objects in queue.

F. E-mail

Three major components for email

- User agents (us)
- Mail servers (which send the emails around)
- Simple mail transfer protocol (the protocol used to send emails)

User agents, mail servers and SMTP protocol.

User agents

- Compose, edit and read mail messages
- Outgoing and incoming messages which are stored on the mail servers

Mail servers

- Mailbox - which contains incoming messages for user
- Message queue - which contains outgoing messages from a user

SMTP protocol

- Client: sending mail server
- Server: receiving mail server

Thus, for e-mails, there is server-to-server requests which are then relayed to clients.

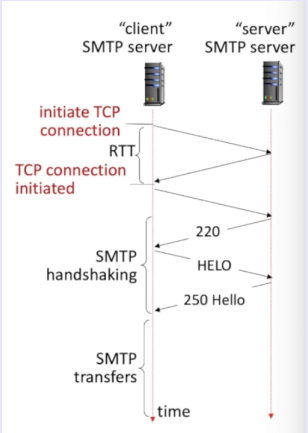
Alice sends an email to Bob.

1) Alice uses her user agent (for example, outlook) to compose an email message to `bob@someschool.edu`.
2) Alice's user agent sends the message to her mail server using SMTP or HTTP, and the

message is placed in the message queue.
   - Whether the user agent uses SMTP or HTTP depends on the user agent
3) Alice's mail server opens a TCP connection with Bob's mail server
4) The mail server then sends Alice's message over the TCP connection
5) Bob's mail server places the message in Bob's inbox, which he can then read.

SMTP RFC (5321).

- uses TCP to reliably transfeer email messages from client to server
- **uses port 25**
- Of course, there is the usual SYN, SYN-ACK then ACK to establish the TCP connection.
- Three phrases of transfer for SMTP
  - An SMTP handshake (which establishes and SMTP connection)
  - SMTP transfer of messages
  - SMTP closure
- Uses persistent connections



Mail messsage format.

Header

- To:
- From:
- Subject:

A blank line

Body: the "message", 7-bit ASCII

But how are the emails actually accessed from a user agent? There is of course a protocol for this too!

Internet Mail Access Protocol (IMAP).

IMAP defines how users interact with mail servers to access email messages. Commands are sent using a TCP connection using ASCII commands, such as

- `LOGIN username password` to authenticate a user.
- `SELECT INBOX` to open the inbox.
- `FETCH 1 BODY[]` to retrieve message 1.
- `SEARCH SUBJECT "Meeting"` to search emails with "Meeting" in the Subject/
- `LOGOUT` to close the session.

G. Domain Name System (DNS)

DNS and it's function.

Hosts and routers will have atleast two identifiers:

1) A human readable name (such as google.com.au)
2) IP address (32 bit) - used to address datagrams (packets)

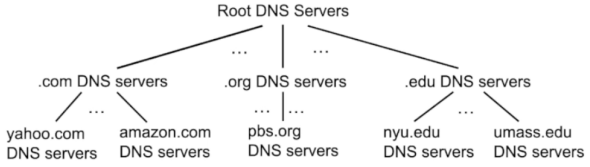
But how do we translate between addresses and names? We use DNS!

DNS is a distributed, hiearchical database implemented in hierachy of many *name servers*. It's also an application-layer protocol, implemented in the network edge. It also has the following functions

- host aliasing - aliased names can also be translated.
- mail server aliasing - IP address of a mail server, associated with a domain
- load distribution - many IP addresses may be allocated to one name, which the DNS can distribute

So why distributed and not centralised?

- A central DNS server would be a single point of failure could bring down a network
- Intense traffic volume
- Distance from the server
- Maintenance
- Lack of scaleability



If a client wants an address at `www.amazon.com`:

1) client requires root server to find .com DNS server
2) queries .com server to find amazon.com server
3) queries amazon.com to get IP address

Root DNS servers.

Official, contact-of-last-resort by name servers that can not resolve names. They forward clients to proper name servers.

There are 13 logical root name servers world

wide, but each server is replicated many times in each region.

Layers of the DNS tree.

- **Root name servers**: has information about TLD DNS servers.
- **Top-Level Domain (TLD) servers**: responsible for .com, .org, .cn, etc.
- **Authoritative DNS servers**: organisation's own DNS servers, providing authoritative hostname to IP mappings for organisations named hosts
- **Local DNS servers**: When a host makes a DNS query, it is sent to it's local DNS query, local DNS server returns reply answering:
  - recent name-to-address translation pairs (which might be out of date)
  - forwarding request to the DNS hiearchy above

There are two types of querying:

1) Iterative, where the local DNS server sequentially requests down the DNS hiearchy, until it finds the address
2) Recursive, where the requests are placed on the sequential servers (for example, root goes to TLD directly, rather than going back to local).

Caching DNS information.
Once any name server learns mappings, it caches the mapping and immediately returns the cached mapping in response to a query. Caching of course improves response time (as it does not have to go down the hiearchy). Cache entries do time out.

DNS records. DNS servers store records in the form `resource records`

`(name, value, type, ttl)`

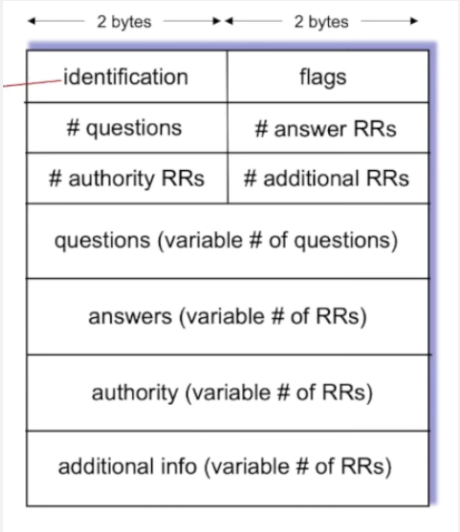
Depending on the type, these keys hold different values. `ttl` refers to Time to Live, which holds how long the record should be held in cache before being discarded.
Common types

- A: `name` is hostname, `value` is IP address
- NS: `name` is domain (e.g foo.com), `value` is hostname of authoritative name server
- CNAME: `name` is an alias, `value` is canonical name
- MX: value is the name of the mailserver associated with name

Getting your info into the DNS. example: new startup 'Network Utopia'

- register name `networkutopia.com` at DNS registrat
  - provide names, IP addresses of authoritative name server
  - registra inserts NS, A RRs into .com TLD server
- create authoritative server locally

DNS protocol messages. DNS query and reply messages both have the same format.



- identification header is the same for reply and query, to identify which reply belongs to which query
- flag contains which type of query/reply it is
- number of questions and answers

DNS Security.
DDoS attacks

- bombard root servers with traffic
  - traffic filtering
- bombard TLD servers

Re-direct attacks

- man-in-middle: intercept DNS queries
- DNS poisoning: send bogus replies to DNS server, which caches
  - preventable by only caching from authoritative name servers.

Exploit DNS for DDoS

- send queries with spoofed source address: target IP
- requires amplification

H. Peer-to-peer applications

Peer-to-peer architecture.

- no always-on server
- arbitrary end systems directly communicate

- peers request service from other peers, provide service in return to other peers
- peers are intermittently connected and change IP addreses

Why peer-to-peer?.

How much time to distribute file (size F) from one server to N peers? where μ_s is the bandwidth of the server, and d_i is the individual download rates of the clients, we have the throughput to be

$$\text{download time} \geq \max\left\{\frac{NF}{\mu_s}, \frac{F}{d_{min}}\right\}$$

Note that this grows linearly with N. In the peer-to-peer model, each client must download a file copy $\frac{F}{d_{min}}$, but as more clients begin to share bandwidth, we have that

$$\text{download time} \geq \max\left\{\frac{F}{\mu_s}, \frac{F}{d_{min}}, \frac{NF}{\mu_s + \sum_{u_i}}\right\}$$

where u_i is the upload speed of the clients. So more clients, lower download time. $\frac{NF}{\mu_s}$ is the initial time for all the clients to download the file.

BitTorrent.

- file divided into 256Kb chunks
- peers in torrent send/receive file chunks

A *torrent* is a group of peers exchanging chunks of a file. A *tracker* tracks peers participating in a torrent. So it's not a pure peer-to-peer architecture, as there is an always-on server.

If a new user wants to join a torrent, then the user contacts the tracker. The tracker gives the list of peers in the torrent. They initially have no chunks, but as they accumulate them, they also begin to upload them.

BitTorrent: requesting and sending file chunks.

Consider two participants, *Alice* and *Bob*.

- Requesting chunks
  - at any given time, different peers have different subsets of file chunks.
  - periodically, Alice asks each peer fora list of chunks they have.
  - Alice requests missing chunks, *rarerst first*, to ensure these chunks have copies.
- Sending chunks: `tit for tat`
  - Alice sends chunks to those four peers currently *sending* her chunks at **highest rate**.
    - other peers are `choked` by Alice (Alice does not send them chunks)
    - re-evaluate top 4 every 10 seconds
  - Every 30 seconds, Alice randomly chooses another peer, and starts sending chunks
    - `optimistically unchoke` this peer
    - new chosen peer may join top 4

I. Video streaming and content distribution networks (CDNs)

Videos.

- video: sequence of images displayed at a constant rate
- digital image: an array of examples
- coding: use redundancy within and between images to decrease # bits used to encode image

Two main encodings

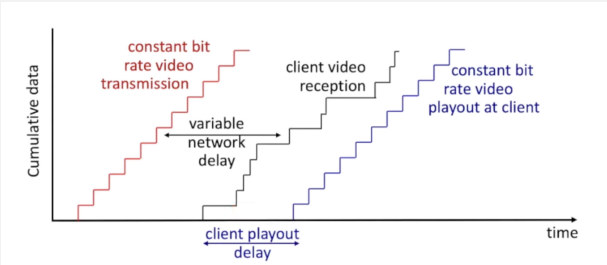
- CBR (constant bit rate): video encoding rate fixed
- VBR (variable bit rate): video encoding rate changes as amount of spatial, temporal coding changes.

But the amount of available bandwith will vary over time, and we'll have to adapt to that. We also have to deal with packet loss - delay due to congestion will delay (causing buffering).

Challenges on the client side.

- continuous playout constraint: during client video playout, playout timing must match original timing.
  - but network delays are variable (jitter), so we'll need a **buffer** to match continuous playout constraints

So how is the continuous playout constraint solved?



Dynamic, Adaptive Streaming over HTTP (DASH).

Server:

- divides video file into multiple chunks
- each chunk encoded at different rates
- `manifest file`: provides URLs for different chunks

Client:

- periodically estimate server-to-client bandwidth
- consults `manifest file`:

  - chooses maximum coding rate sustainable at current bandwidth
  - can choose different coding rates for different points in time

But how do we stream all of this content to hundreds of thousands of simultaneous viewers? One option would be a *single, large "mega-server"*. Of course, this suffers from congestion, single point of failure, as well as distance issues to some clients.

Content distribution networks (CDN). Content Distribution Networks (CDNs), store/serve multiple copies of videos at multiple geographicaly distributed sides. There are two types of CDNs

1) enter deep: push CDN servers deep into many access networks (at the network edge)
2) bring home: smaller number of CDN servers near access nets (deeper into the network)

So how is a video streamed using CDNs?.
Let's say `Bob` wishes to stream a video on Netflix

1) `Bob` selects the video, which in turn creates a DNS request to find the CDN server.
2) Then, we request the CDN server the DNS query returned for a `manifest` file.
   - This begins the process of video streaming - we now have the URLs for the chunks of video.
3) If at any point, the CDN server becomes congested - it may redirect `Bob` to a new CDN server.

III. Transport Layer

The transport layer is all about how we segment and prepare data for delivery.

- Providing logical communication between application processes running on different hosts
- Transport protocols actions in end systems (on the network edge)
- Two transport protocols available to internet apps; TCP and UDP

The distinction between the *network* layer and *transport* layer - is that the network layer provides logical communication between *hosts* - whereas the transport layer provides logical communication between *processes*.

A. Transport layer multiplexing and demultiplexing

Multiplexing and demultiplexing.
Multiplexing and demultiplexing is how the transport layer keeps track of which segments of data belong to which application (sockets), by using port numbers contained in the header.

These headers are added (and interpreted) at the transport layer, and contain the `source` and `destination` ports.

How demultiplexing works.

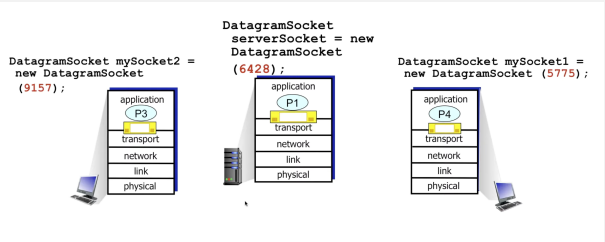
1) the host receives IP datagrams
2) each datagram has source IP address, destination IP address
3) each datagram carries one transport layer segment
4) each segment has source, destination port number
5) the host uses IP addresses and port numbers to direct segment into socket

UDP Connectionless Demultiplexing.
When creating a UDP socket, you have to specify a host-local port number. When creating a datagram to send into a UDP socket, you must specify:

- destination IP address
- destination port number

When the receiving host receives UDP segment, it uses the destination port number in the segment into the respective socket.

UDP datagrams with the same destination port number but with different source IP addresses and port numbers will be directed to the same socket.



Both `mySocket1` and `mySocket2`'s datagrams will be sent to the same socket, on port 6428 on `serverSocket`.

TCP Connection-oriented demultiplexing.
TCP sockets use 4-tuples:

- source IP address
- source port number
- destination IP address
- destination port number

Since TCP sockets have a distinct connection between sender and receiver, it requires to know all four of the above values to send it to the respective sockets.
Therefore, if many processes use the same port; under a TCP connection, we can ensure that packets

can be sent to specific sockets held by processes on the server.

The welcoming socket.
In TCP connections, a `handshake` is required to establish the connection. This handshake is done with a `welcoming socket` at the server, which is seperate to the actual socket that the client will connect to.

It's main purpose is to listen for incoming connections.

B. *Connectionless transport: UDP*

UDP: User Datagram Protocol.
- connectionless; no handshakes with sender receiver
- UDP segments are handle independently
- no connection establishment - saves an RTT
- simple, smaller header size
- no congestion control (so no intrinsic buffering of requests)

Thus, UDP is used for data that needs to be transferred quickly, where packet loss isn't critical; like media streaming.

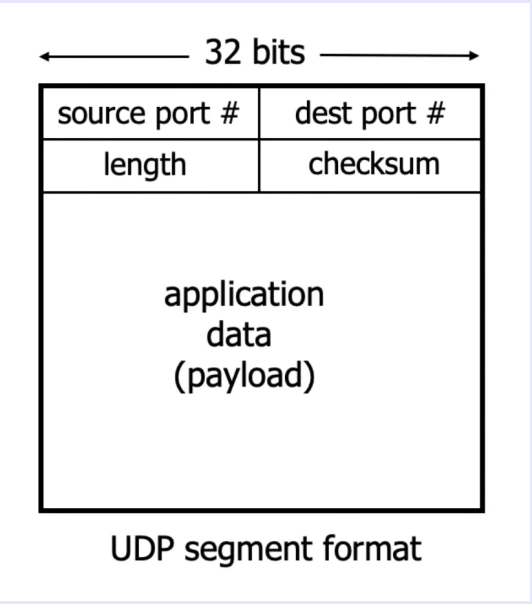
On the sender side

- A sender passes an appliaction-layer message
- UDP segment header field values are identified
- UDP segment is created, and sent to IP

On the receiver side

- receive segment from IP
- checks UDP checksum header value
- extracts application layer message
- demultiplex message up to application socket

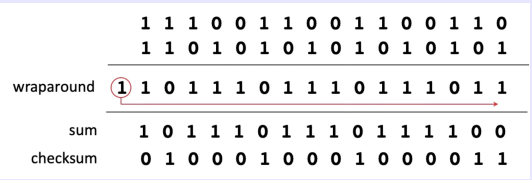
UDP segment.



UDP segment format

- source port number and destination port number are straight forward
- the length of the UDP segment, including the header can be variable
- checksum (which is an error checking header)

Checksum.
UDP checksum is aimed at detecting errors (or flipped bits) in a transmitted segment. Checksum works by sending numbers, and the sum of these numbers. If the sum of these numbers don't equal to the transmitted sum, then there has been an error while transmitting.

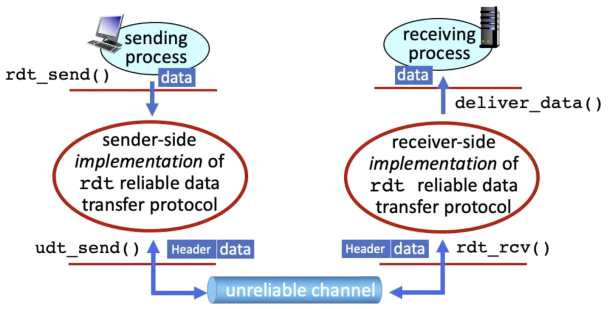


- Begin by doing a normal bit addition - if there is a wrap around, add it to the back once more
- The check sum is then the complement of this final sum

Do note that checksum isn't a perfect error check - you can convince yourself of this by considering flipping the first two bits of the two numbers in the example. The checksum would still be the same!

C. *Principles of Reliable Data Transfer*

How do we reliably send packets from a sender to receiver - ensuring they are sent properly? This mechanism is of course implemented in the transport layer, over an *unreliable channel*.



So how do we develop a **Reliable Data Transfer** (`rdt`) protocol?

- consider only unidirectional data transfer
  - but control info will flow on both directions
- use finite state machines (FSM) to specify sender, receiver

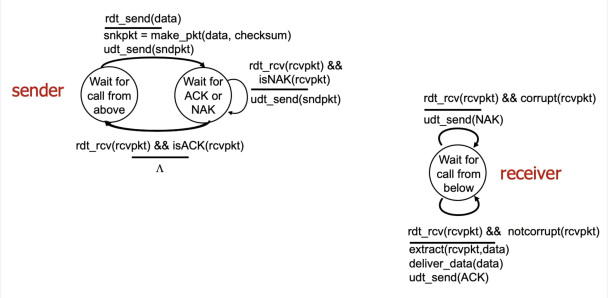
Let us try to make finite state machines for differing levels of contingencies.

rdt1.0: reliable transfer over a reliable channel.
Since the underlying channel is reliable, the sender and receiver don't have to do much besides send and receive the data. They can be sure that the packets are sent and received.

rdt2.0: channel with bit errors.
Now, the underlying channel is unreliable and may flip bits in packets. We can use checksum to detect bit errors. How do we recover from errors?

- acknowledgements (ACKs): receiver explicitly tells sender that packet received is OK
- negative acknowledgements (NAKs): receiver explicitly tells sender that packet had errors
- sender retransmits the packet on a receipt of a NAK

Now the sender sends one packet, then waits for the receivers response.

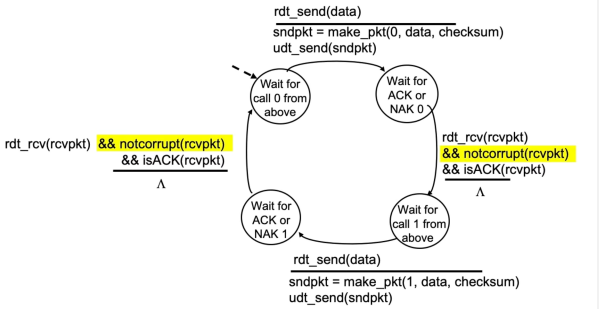


There should be one obvious question that arises.

What happens if the acknowledgements are corrupted or dropped?

rdt2.1: handling corrupted ACKS and NAKS. Here's how rdt2.0 can be extended to deal with corrupted acknowledgements

- sender retransmits current packet if ACK/NAK is corrupted
- sender adds sequence number to each packet (so same seq. number for same packet)
- receiver discards duplicate packet



rdt2.2: NAK-free protocol.
Same functionality as rdt2.1, but implemented using ACKs only.

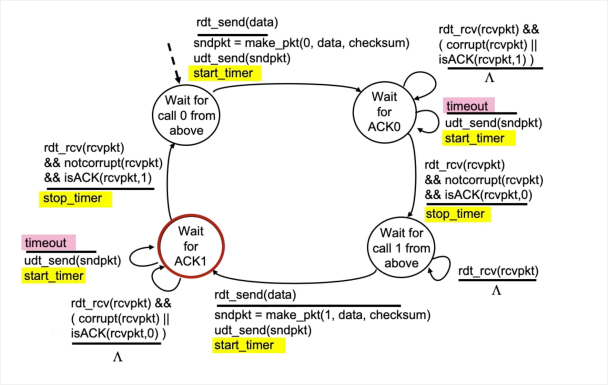
- Instead of a NAK, receive sends an ACK of the last packet # received.
- Therefore, for the sender, a duplicate ACK indicates a loss at # + 1.
- TCP uses this NAK free approach.

rdt3.0: channels with errors and loss.
In reality, channels can also lose packets.

- Checksum, sequence #s, and etc can be useful, but they won't solve this problem.
- What do humans do? If someone randomly stopped talking, we would ask them to acknowledge what we said.
- So the **motivation** is to wait a 'reasonable' amount of time for an ACK to come back, and if it doesn't, to take appropiate action.

Approach:

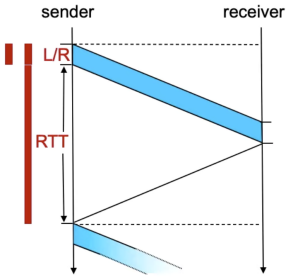
1) sender waits a reasonable time for an ACK
2) retransmit if no ACK is received in this time
3) if packet (or ACK) delayed, but not lost:
   a) retransmission will be a duplicate, but the seq #'s can deal with this
   b) receivers must specify sequence # of packet being ACKed
4) use a countdown timer to interrupt after a reasonable amount of time



Performance of `rdt3.0`:

- U_{sender}: utilisation - fraction of time sender busy sending
- 1 Gbps link, 15 ms prop. delay, 8000 bit packet
- $D_{trans} = L/R = 8000/10^9 = 8$ microseconds

$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$



That's very low utilisation. To increase the utilisation of the link, we can allow multiple packets to be sent concurrently while waiting for their acknowledgements.

D. Pipelining for reliable data transfer protocols

Go-Back-N.

Sending side

- Has a window of N transmitted packets, that are unacknowledged
- Has a k-bit seq # in the packet header
- *cumulative ACK:* when an ACK of seq # n is received, **all** of the packets until n will be acknowledged.
- There is a timer for the oldest in-flight packet.
  - if the receiver receives out of order, the packet is discarded and acknowledgment of the most recent in order packet is sent
  - if packet k out of n packets is timed out, then packets $k \rightarrow n$ must be resent (this is the cumulative idea).

Receiving side

- ACK-only: always sends ACK for correctly-received packet so far, with *highest* in-order seq #.
  - may generate duplicate ACKs
  - need only remember `rcv_base`, or next in-order seq #.
- on receipt of out-of-order packet:
  - discard or put in buffer
  - send ACK for highest seq # received so far

Convincing myself of Go-Back-N.

Consider sending four packets `0, 1, 2, 3` to a server. The server acknowledges all of these packets, but the acknowledgements for 0, 1 and 2 are lost. The acknowledgement for 3 arrives - and in Go-Back-N, we say **all of these packets are sent**.

We can do this as we know the server would *not* send an acknowledgement for 3, if it had not already received 0, 1 and 2.

Selective repeat.

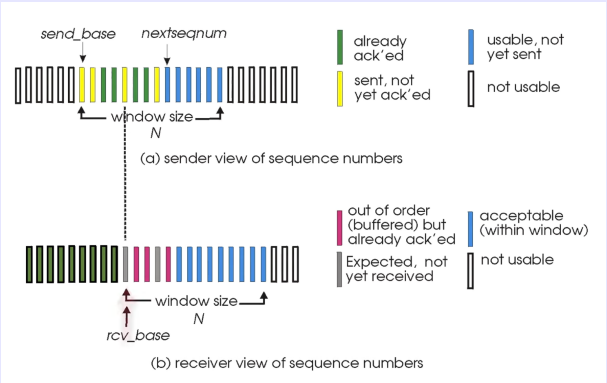
Receiving side

- receiver *individually* acknowledges all correctly received packets
- so packets must be buffered as needed, to guarantee in order delivery

Sending side

- sender times-out/retransmits individually for unacknowledged packets
- so the sender must time every unacknowledged packet, as there is no cumulative idea like in `Go-Back-N`
- The window is only advanced when the last packet to be received is acknowledged.

Sliding window defined by the smallest acknowledged k-th packet, up till the $k + n + 1$-th packet, which defines the maximum amount of packets allowed to be sent at one time.



Sequence numbers for Go-Back-N and Selective Repeat.

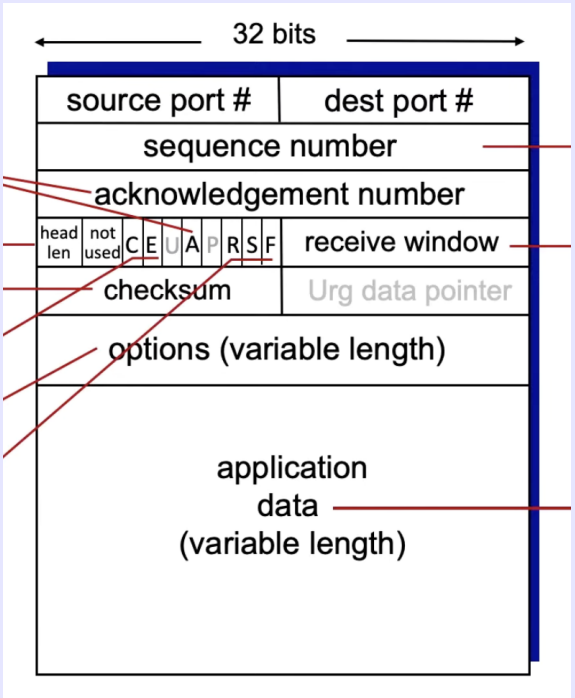
For window size W and and number of possible sequence number $|S|$

- For `Go-Back-N`, $W < |S|$.
  - For `[0, 1, 2, 3]`, if all ACKs are lost, then `0` will be sent as a resubmission, but the receiver believes it is a new packet.
- For `Selective Repeat`, $W \leq \frac{|S|}{2}$.
  - If W packets are lost, with $|S| \leq 2|W|$, retransmissions are interpreted as new data.

E. TCP Reliability

TCP Overview.

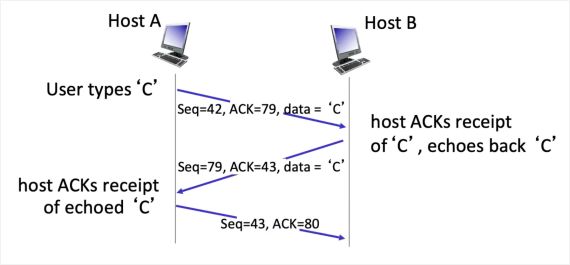
- point-to-point: one sender, one receiver
- reliable, in-order *byte stream*: no "message boundaries"
- full duplex data: bi-directional data flow
- cumulative ACKs: so uses Go-Back-N
- pipelining: tcp congestion and flow control set window size
- connection-oriented: so a handshake occurs to initialise sender-receiver state before data exchange
- flow controlled: sender will not overwhelm receiver



TCP Sequence Numbers.

A *byte stream* is simply a sequence of bytes - that is TCP sends *byte/stream-oriented* data that can be read by the receiver. UDP uses *message-oriented* which means when one message is sent, it will be sent as one packet.

- Sequence numbers: represents byte stream 'number' of first byte in segment's data.
- Acknowledgements: represents the seq # of the next byte expected from the other side
- What to do with out of order segments? TCP doesn't give a standard reasoning, so is up to the implementation



Note in the above statement, that the acknowledgement statement corresponds to the next byte that the *receiver* should **expect** from the sender.

TCP Maximum Segment Size.

- IP Packet (the packet actually sent across)
  - No bigger than Maximum Tramission Unit (MTU) of link layer
  - E.g up to 1500 bytes with Ethernet
- TCP packet
  - IP packet with a TCP header and data inside
  - TCP header $\geq$ 20 bytes long
- TCP **segment**
  - No more than Maximum Segment Size (`MSS`) bytes
  - E.g up to 1460 consecutive bytes from the stream
  - MSS = MTU - 20 (Min. IP Header) - 20 (min. TCP header)

TCP Acknowledgements, Sequence Numbers and Piggybacking.

- Sequence numbers in the TCP protocol begin with a random Initial Sequence Number (ISN).
  - This prevents malicious behaviour, as well as confusion for back-to-back connections.
- ACKs returns the **next sequence number** the receiver expects.
- Many TCP connections have duplex (double-sided) sending behaviour
  - To reduce delay, ACKs can be sent alongside payload
  - This combines two messages into one

TCP Go-Back-N + Selective Repeat.

TCP combines the ideas of Go-Back-N and Selective Repeat

- TCP will *typically* use cumulative acknowledgements like Go-Back-N
- TCP also uses a sliding window to control how much unACKed data can be "in flight"
- However, TCP also **buffers out-of-order segments**, using Selective Acknowledgements `SACK`.
- This means the sender can retrainsmit only the lost packet, while combining the ideas of cumulative ACK.

TCP round trip time, timeout.

How do we set the timeout value? Clearly these should depend on the round trip time, but if we do this too fast - we may do unnecessary transmissions, or too long, and react too slow.

$$RTT_{estimate} = (1-\alpha) \cdot RTT_{estimate} + \alpha \cdot RTT_{new}$$

Where $\alpha \in [0,1]$. A standard value for $\alpha = 0.125$. The equation above denotes an exponential weighted moving average, where new samples are given more weight over older ones.
Now to find the actual timeout interval, we use

$$\text{Timeout} = RTT_{estimate} + 4 \cdot \text{dev}(RTT_{estimate})$$

And to calculate the deviation

```
DevRTT = (1-β)*DevRTT + β*|SampleRTT-EstimatedRTT|
```

TCP Sender to Receiver Flow.

1) create segment with seq #
2) seq # is byte-strema number of first data byte in segment
3) start timer if not already running
   - think of timer as for older unACKed segment
   - expiration interval: `TimeOutInterval`

Receiver Scenarios

1) arrival of in-order segment with expected # arrives. all data up to expected seq # already acked.
   a) wait up to 500ms for another segment.
   b) if no next segment, send ACK.
   c) if another segment, then send a cumulative ACK.
2) arrival of in-order segment with expected seq #. one other segment has ACK pending
   a) immediately send single cumulative ACK, acking both in-order segments.
3) arrival of out-of-order segment, higher than expected seq #
   a) immediately send duplicate ACK
   b) ACK should contain # of next expected byte
4) arrival of segment that partially or completely fills gap
   a) immediately send ACK, provided that segment starts at lower end of gap.

TCP fast retransmit.

If multiple consecutive ACKs from the receiver come back with the same sequence number, the sender can know that there has been a missing segment.

1) If there is a receipt of three duplicate ACKs, this indicates 3 segments have been received after a missing segment
2) So retransmit the lost segment (as detailed by ACK #) immediately.

F. TCP Flow Control

What happens if network layer delivers data faster than application layer removes data from socket buffers? this is what flow control tries to achieve - as buffered data at the application layer isn't very helpful.

The basic idea.

- TCP receivers 'advertised' free buffer space in `rwnd` field in TCP header item sender limits amount of unAcked ('in-flight') data to received `rwnd`.
- guaratees receive buffer will not overflow.

G. TCP connection management

TCP Connection Management.

Before exchanging data, sender and receiver must 'handshake'

- agree to establish connection
- agree on connection parameters (e.g starting seq #'s, ...)

2-way handshakes (where the sender and receiver each send one establishing request) don't work, as this can create confusions with re-transmisted connection requests and packets.

Rather, TCP uses the 3-way handshake.

1) The client sends the server a `SYN` message, with `SYNbit=1` and `seq=x`
2) The server sends the client a `SYNACK` message, with `seq=y`, `ACKbit=1` and `ACKnum=x+1`
3) The client sends the server a `ACK` message, with `ACKbit=1` and `ACKnum=y+1`.

How to close a TCP connection?

1) Closer sends packet with `FIN` bit = 1

2) Closee responds with `ACK` bit = 1 and `FIN` bit = 1. These may be seperated into two different messages.
3) Closer responds with `ACK` bit = 1. Closee is now closed - closer remains open for any remaining packets from the closee.

Abrupt termination

1) Rather then a `FIN`, an abrupt termination is declared by the closer sending a `RST` packet.
2) The closee does not respond with an ACK - so it is not sent reliably.
3) Rather, if the closee sends another packet to A, another `RST` is sent.
4) Occurs due to **unexpected and invalid packets** as well as network or system loss.

TCP SYN Attacks and Cookies.

SYN Attacks

- An attacker creates a fake SYN packet, with the IP address of victim host
- Victim receives TCP connection, allocates buffers, creates variables, etc.
- This uses up resources on the victim's machine.
- The attacker never ACKs; yet even during this timeout, the victim has used resources.
- The attacker sends multiple of these ACKs, overwhelming the victim.

SYN Cookies

- On receipt of SYN, server does not creative a connection state.
- Creation of ISN, that is a hash of source & dest IP address, and port number of SYN packet.
  - Responds with SYN ACK containing ISN.
  - The server need not store this.
- If original SYN was genuine, then a connection state will be instantiated if the ACK returns ISN + 1
- Otherwise, a fake SYN, and no state is created.

H. Congestion control

Problem statement.

Too many sources sending too much data too fast for the **network** to handle.

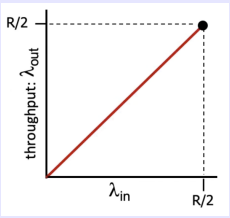
How congestion occurs.

Consider two hosts sending through a shared link of capacity R to a router R, which links to two seperate servers of shared link R. The two hosts send to seperate servers.

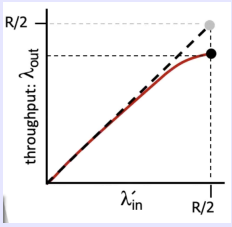
- λ_{in}: the arrival rate of data from a host
- λ_{out}: the throughput that the server receives

With buffers at R, and queuing delays as $\lambda_{in} \to \frac{R}{2}$, we may see retransmissions from the hosts to account for buffer overflows. Call this adjusted arrival rate λ'_{in}.

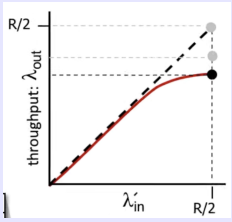
Imagine that somehow the sender knows how much space is available in the buffer, and thus no packets are dropped. In this case, $\lambda_{in} = \lambda_{out}$.



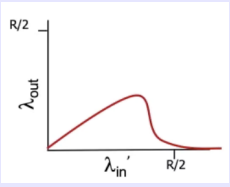
Now imagine the sender can perfectly know when a packet is dropped by the buffer. In this case, $\lambda'_{in} \geq \lambda_{out}$, as packets are dropped and do not reach the servers.



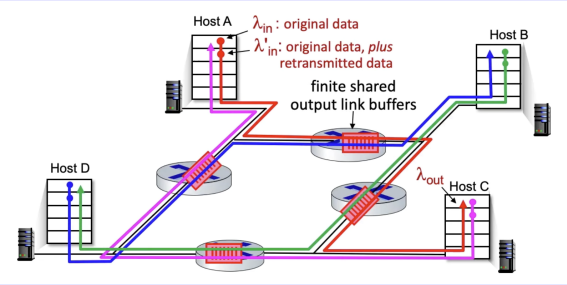
Now imagine the sender *does not know* whether a packet has been dropped or not, which require retransmissions. However, sender times can time out prematurely, sending two copies, both of which are delivered.



As we add more hops inbetween our hosts and servers, any upstream transmission capacity and buffering that was used for the packet gets wasted - and so congestion becomes worse and worse.



We call the above phenomena **congestion collapse**.



So how do we deal with this?

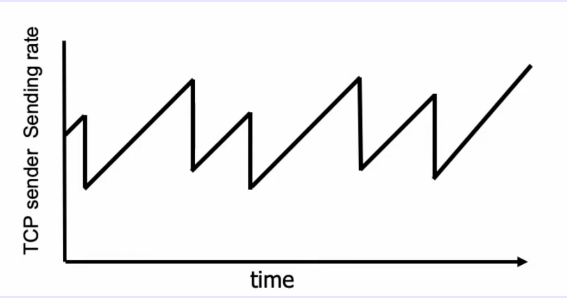
1) **End-end congestion control**: no explicit feedback from network, but infers congestion from loss and delay. This is the approach taken by original TCP.
2) **Network-assisted congestion control**: routers provide direct feedback to sending/receiving hosts with flows pasing through congested routers. may indicate congestion or explicitlys end rate.

I. TCP Congestion Control

Additive Increase Multiplicative Decrease (AIMD).

Approach: senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss events. Specifically:

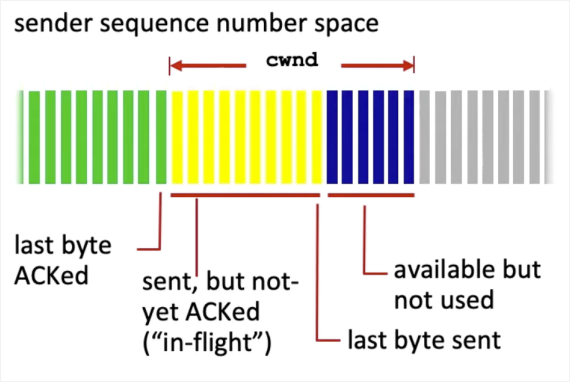
- **Additive increase**: increases rate by 1 maximum segment size every RTT until loss detected
- **Multiplicative decrease**: cut sending rate in half at each loss event



Multiplicative decreases are signalled by:

- Cut in half on loss detected by triple duplicate ACK
- Cut to 1 maximum segment size when loss detected by timeout

cwnd (congestion window) and TCP.



- Recall `cwnd` (congestion window), which is dynamically adjusted by receivers.
- The TCP sender limits tranmission such that:

$$\text{LastByteSent} - \text{LastByteAcked} \leq \texttt{cwnd}$$

- Then, TCP rate is given by

$$\text{TCP rate} = \frac{cwnd}{\text{RTT}} \text{ bytes/sec}$$

TCP slow start.

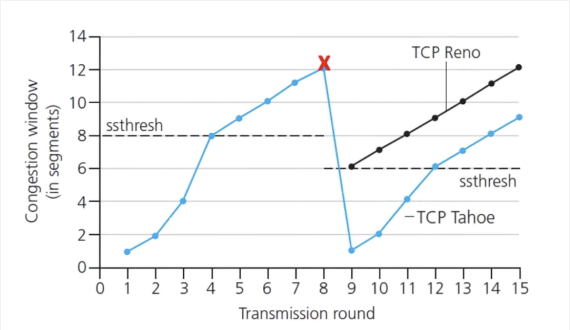
- When connection begins, increase rate exponentially until first loss event
  - initally `cwnd` = 1 MSS
  - double `cwnd` every RTT
  - done by incrementing `cwnd` for every ACK received
- so sending rate ramps up exponentially.

So how do we join these two ideas of AIMD and slow start together? More formally put:

When should we swap from exponential increasing sending rates to linear increasing?

Transitioning from slow start to AIMD.

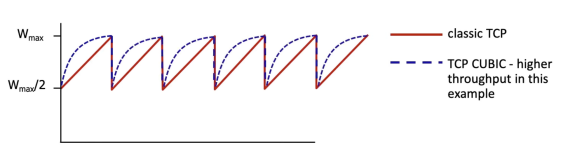
We switch from slow start exponential growth to AIMD when `cwnd` (or the amount of bytes that are able to be in flight) reaches half of it's value before the last timeout.



TCP CUBIC.

Is there a better way than AIMD to prove for usable bandwidth?

- W_{max}: sending rate at which congestion loss was detected
- congestion state of bottle neck link probably hasn't changed much
- after cutting rate/window in half on loss, initally ramp to W_{max} faster, but then begins to approach W_{max} slower.



How it actually works:

1) K: point in time when TCP window size will reach W_{max}
2) increase W as a function of the *cube* of the distance, between current time and K
   - that is, increase W a lot when it's further away
   - but as it reaches W_{max}, slow down
3) TCP CUBIC default in Linux, most popular TCP for popular web servers.

Delay-based TCP congestion control.

TCP increases the sending rate until packet loss ocurrs at some router's output; we refer to this as the **bottleneck link.**

Consider the link can only send a speci fic amount of bits per second. Clearly there is no point of continually sending more bits through this link if it cannot handle the traffic.

We can measure the round trip time. We can also consider the # of bytes sent in the last RTT interval. The minimum RTT value, is the *uncongested path*. Consider:

$$T_{measured} = \frac{\#\text{ of bytes in last RTT interval}}{RTT_{measured}}$$

The uncongested throughput with congestion window `cwnd` is then

$$T_{ideal} = \frac{cwnd}{RTT_{min}}$$

Thus

- If $T_{measured}$ close to T_{ideal}, increase `cwnd` linearly.
- If $T_{measured}$ far below T_{ideal}, decrease `cwnd` linearly.

Explicit congestion notification (ECN).

TCP deployments often implement *network-assisted* congestion control:

- two bits in IP header (TOS field) marked by network router to indicate congestion
- congestion indication carried to destination
- destionation sets ECE bit on ACK segment to notify sender of congestion
- involves both IP (IP header ECN bit marking) and TCP (TCP header C,E bit marking)

TCP fairness.

Fairness goal: if K TCP sessions share some bottleneck link of bandwidth R, each should ahve average rate of R/K.

Consider AIMD.

1) If connection 1 starts with more throughput then connection 2, both linearly increase.
2) However, connection 1 will reach a sending rate of R much faster - which then leads to a halving of the sending rate
3) While this happens, connection 2 begins to grow, until it also halves.
4) Therefore, the two connections converge to fairness