What is the Internet

- At the edge/outermost areas of the internet we have the devices we use to connect to the internet examples include computers and phones of course, but also any smart devices and datacenters
  - ↳ these are called hosts/end systems/end devices
- In between the end devices are the packet switches These forward and transfer chunks of data (also called packets) between end devices
  - ↳ 2 types: routers and switches
  - ↳ use various communication links to actually transfer data (e.g. ethernet cable, radio transmitters, satellites, etc)
- collections of devices, packet switches, comm links managed by a single entity form networks
  - ↳ Examples: In a home, all devices connected to the internet can form a network. In a company/organisation the 'work devices' can be linked together to form a network
- The internet is really these networks connected to each other, i.e. internet is a network of networks
- Protocols determine how information/data is transferred between networks/devices
  - ↳ IETF: Internet Engineering Task Force define the standards and protocols

Alternatively can view the Internet as a means/infrastructure/method for transferring information from point A in a network to a diff point in a possibly diff network

Protocols define and determine the format and order of messages between network devices. They also define the actions to be performed based on the messages received

The Network Edge

End-system devices = hosts

↳ 2 types: clients and servers

clients ← recieve some kind of service

servers ← provide service

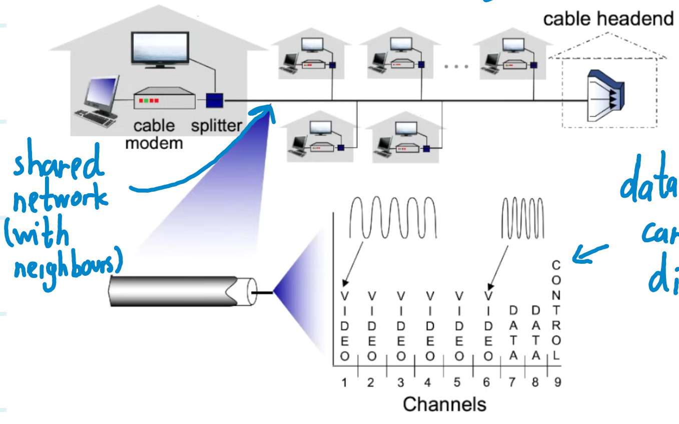
Access networks are the networks that connect end system devices to the Internet

↳ 3 types: residential, institutional, mobile

mobile ← wiFi, 4G/5G, etc

Cable-access network

lots of homes (physically) connect to single cable headend



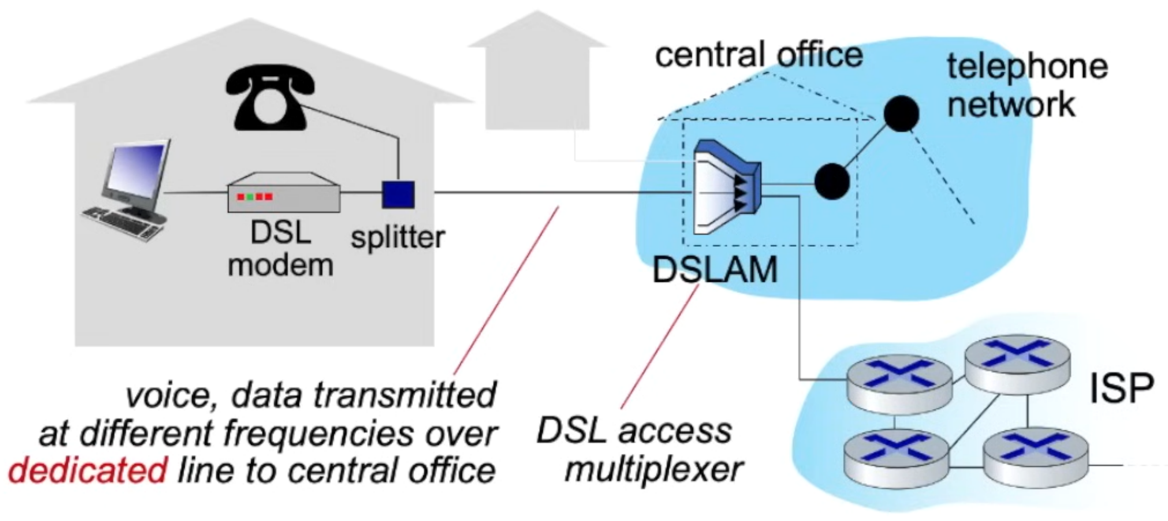
shared network (with neighbours)

data in cable is carried over diff frequencies

frequency division multiplexing (FDM): different channels transmitted in different frequency bands

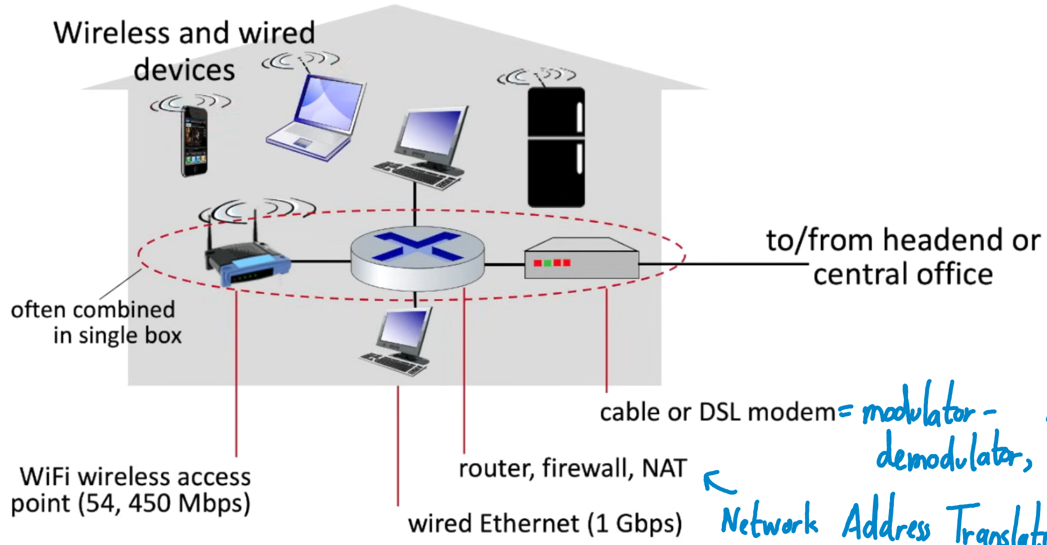
asymmetric carry more data away than to homes (i.e downstream transmission >> upstream)

Digital Subscriber Line (DSL)


central office
telephone network
DSL modem
splitter
DSLAM
ISP
voice, data transmitted at different frequencies over dedicated line to central office
DSL access multiplexer

- Uses existing telephone lines for data transmission
- NOT shared with neighbours
- transmission rates depend on distance to central office

Home Networks


Wireless and wired devices
to/from headend or central office
often combined in single box
cable or DSL modem
router, firewall, NAT
WiFi wireless access point (54, 450 Mbps)
wired Ethernet (1 Gbps)

= modulator - demodulator, ← converts digital signal to analogue (radio) signal

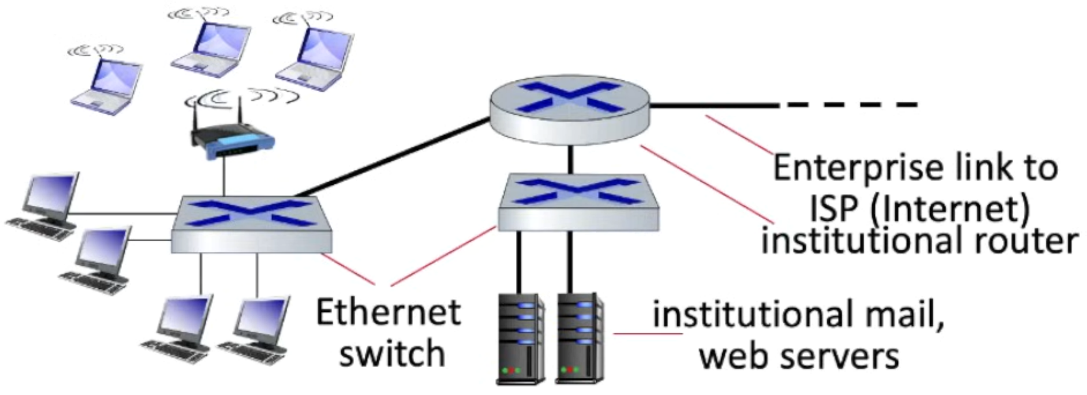
← Network Address Translation
↳ lets multiple (private) devices access internet through (single) public IP address

Wireless Access Networks

protocol defined by IEEE instead of IETF

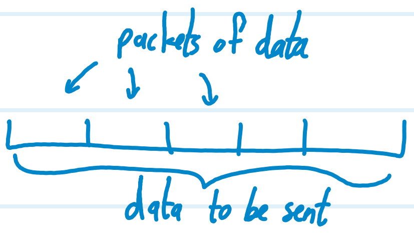
- 2 types: Wireless Local area networks (WLAN e.g. WiFi) and Wide-area cellular access networks (e.g. 5G)
- in both types devices send and recieve data from base station or access point

Enterprise Networks



- used by companies, universities, institutions, etc.
  - ↳ kind of like a very large home network
- data center networks are also an enterprise network (look v different from above)
- connect 100s – 1000s of servers to each other and to internet

Sending data



- hosts send data in chunks called "packets"
- there is usually a capacity limit on how much data can be sent at once (this includes some packet header info determined by protocol)
- there is also transmission rate R determined by the access network. R = num bits sent/sec

Links: physical media

We want to understand (at least a little bit) of how data is literally sent over wires, radio waves, etc.

- guided media: signals go through some physical media (copper wires, fiber optics, etc).
- unguided media: signals propogate freely (e.g. radio waves)

Types of cables:

high transmission rate but prone to noise

- Twisted pair (TP): pair of copper wires twisted around one another; originally used for telephone signals - now used by ethernet
- Coaxial cable: used to carry cable access (lower transmission rate)
- Fiber optics: transmit using light pulses (high transmission rate and immune to EM noise but more expensive)

Wireless radio

- data transmitted on specific frequencies
- broadcast freely so anyone can listen/tune in
- hard to transmit data over! (susceptible to noise, require proximity, etc)

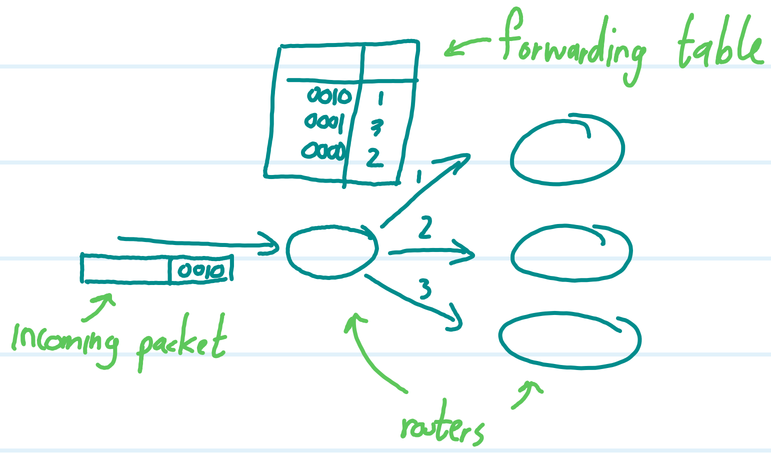
Network core

- network core is the set/network of routers and corresponding communication links

Packet switching is how data is transferred. The data is split into chunks called packets and these packets are sent from source to destination along some path of routers and communication links in network core

A router has 2 main functions: forwarding/switching and routing

↳ Forwarding : packet arrives which has destination address
all routers have a forwarding table; look up address in this table to see how to forward



↳ Routing is used to determine the forwarding tables. A routing algorithm is what determines the exact path a packet will take. This then defines the table

Store-and-forward refers to a specific method of data transmission between 'adjacent' routers. Recall we can only send L bits of data at a time. In the store-and-forward case, router B (the reciever) will wait until the entire packet has been sent over before forwarding it along (it stores the bits of the packet till completion and only then forwards it)

Queveing occurs when data arrives at a router faster than it can transmit it. If queve (i.e. memory) is full then incoming packets are lost (dropped). This is bad (

Circuit switching

An alternative to packet switching. Used to be used by telephone lines

In this case resources (e.g. memory, comm links) are reserved for a "call" between source and destination So there are no packet losses or delays (except propogation delays, i.e. literally the time it takes for a signal to get from point A to point B). However because resources allocated in advance they may be idle or unused when no data being transferred

each call given a frequency band

Circuit switching is carried out using FDM (frequency division multiplexing) or TDM (time division multiplexing)

each call given a periodic interval of time

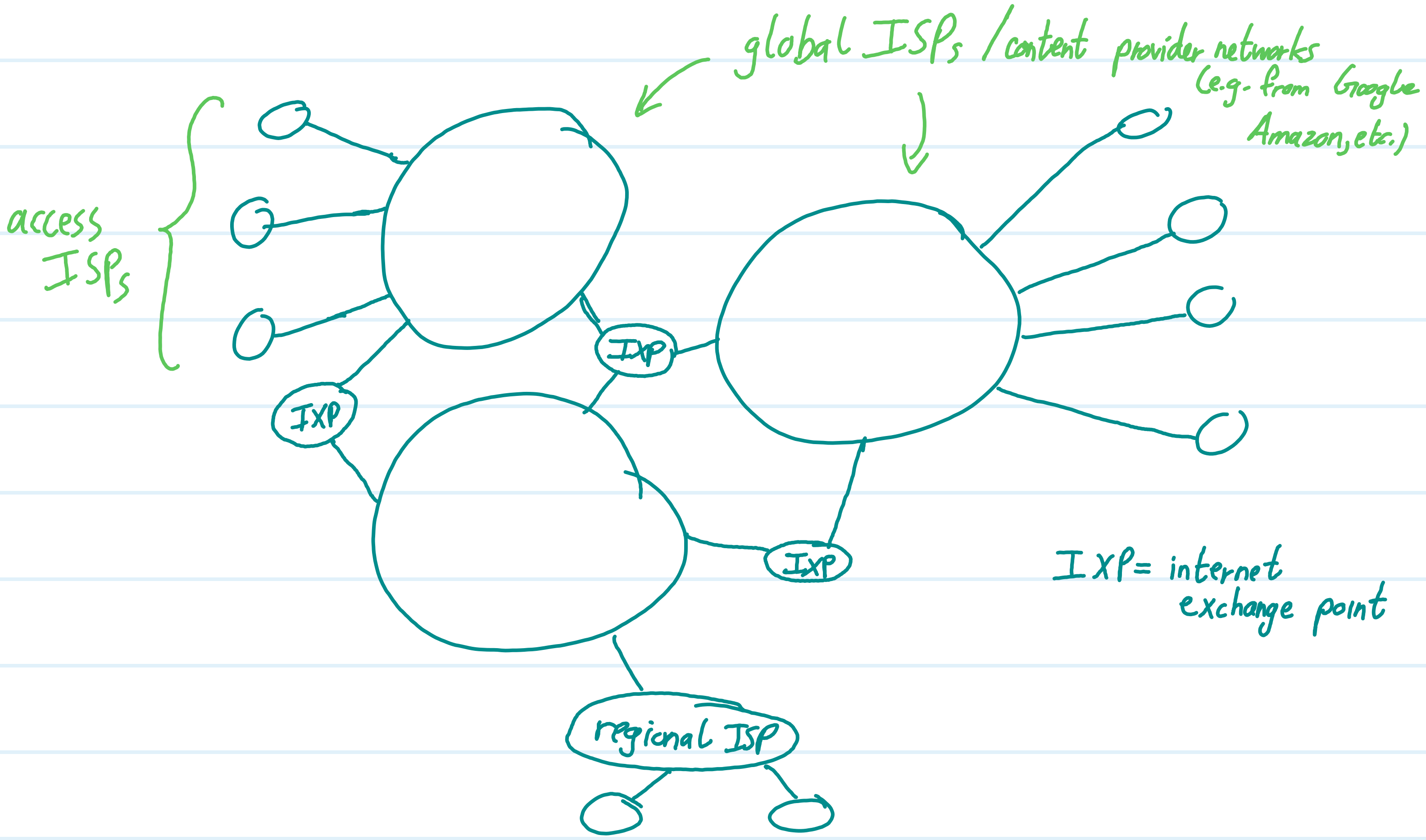
Packet switching vs Circuit switching

Packet switching's biggest advantage is it allows more users on the same connection because (in most cases) not all of them will need to send/recieve data at the same time. Although packet delays and losses are possible, these can be mitigated against with clever protocols. Finally there are some techinques to make packet switching more "circuit-like".

Internet Structure

internet service provider

Hosts connect to internet via access ISPs. There are global / large ISPs that access ISPs connect to.


global ISPs / content provider networks (e.g. from Google Amazon, etc.)
access ISPs
IXP
IXP
IXP
regional ISP
IXP = internet exchange point

Network are peers if they are directly interconnected

So at "center" of internet we have a (relatively) small number of well-connected large networks. Includes so-called "tier-1" ISPs (the big global guys) and content provider networks. Then we have regional networks connected to these and finally the access ISPs at the edge

Network performance

Packet delay:

$$\underbrace{d_{nodal}}_{\text{total delay on a node / router}} = \underbrace{d_{proc}}_{\text{processing incoming packet } (\sim \text{microseconds})} + \overset{\text{time spent in queue}}{d_{queue}} + \underbrace{d_{trans}}_{\text{time taken by transmission from previous router / device} \; \to \; L/R} + \overset{\text{time taken by data to travel from curr router to next one (literally the speed of the electric signals / radio waves)}}{d_{prop}}$$

Diff between d_{trans} and d_{prop}. d_{trans} comes from the fact that the packet has L bits but data can only be carried over at a rate of R bits/sec (if we tried to send too many things at once they might start interfering w/ one another). d_{prop} comes from the fact that it literally takes time for data/signals to get from point A to point B (the length of delay will depend on comm link, distance between routers, etc.).

Queueing delay

a = num packets arriving on avg / s

L = length of packet (in bits)

R = link bandwidth (num bits transmitted per sec)

$$\text{traffic intensity} = \frac{La}{R} = \frac{\text{"incoming"}}{\text{"outgoing"}}$$

ideally want this (much) less than 1

traceroute is a program that lets us see the delays that occur as data moves from source to destination

throughput = rate at which bits are sent from sender to reciever

↳ could be instantaneous or average.

↳ determined by link with least bandwith. This link called bottleneck link

Layered architecture

Suppose you were trying to define a communication system over networks. How would we do it? More importantly how can we do it in a way that is flexible and expandable but also robust enough to handle all of the traffic of the internet? One way to accomplish this is to modularise, i.e. separate out the different components, so that we can separate the task and focus on the components individually and then make sure we can put things together nicely (this is of course a very common theme in computer science). This is why the internet has what is called a layered architecture.

The layers of the internet are:

- Application layer. This is the topmost layer consisting of the applications, programs and processes being run by the hosts themselves. This where messages that need to be sent over the internet typically originate from. Example. When you google "Tiny hamsters in speedos", the application is your web browser and the message will be your search query (potentially along with some metadata about your device and such.

official name for data in app layer

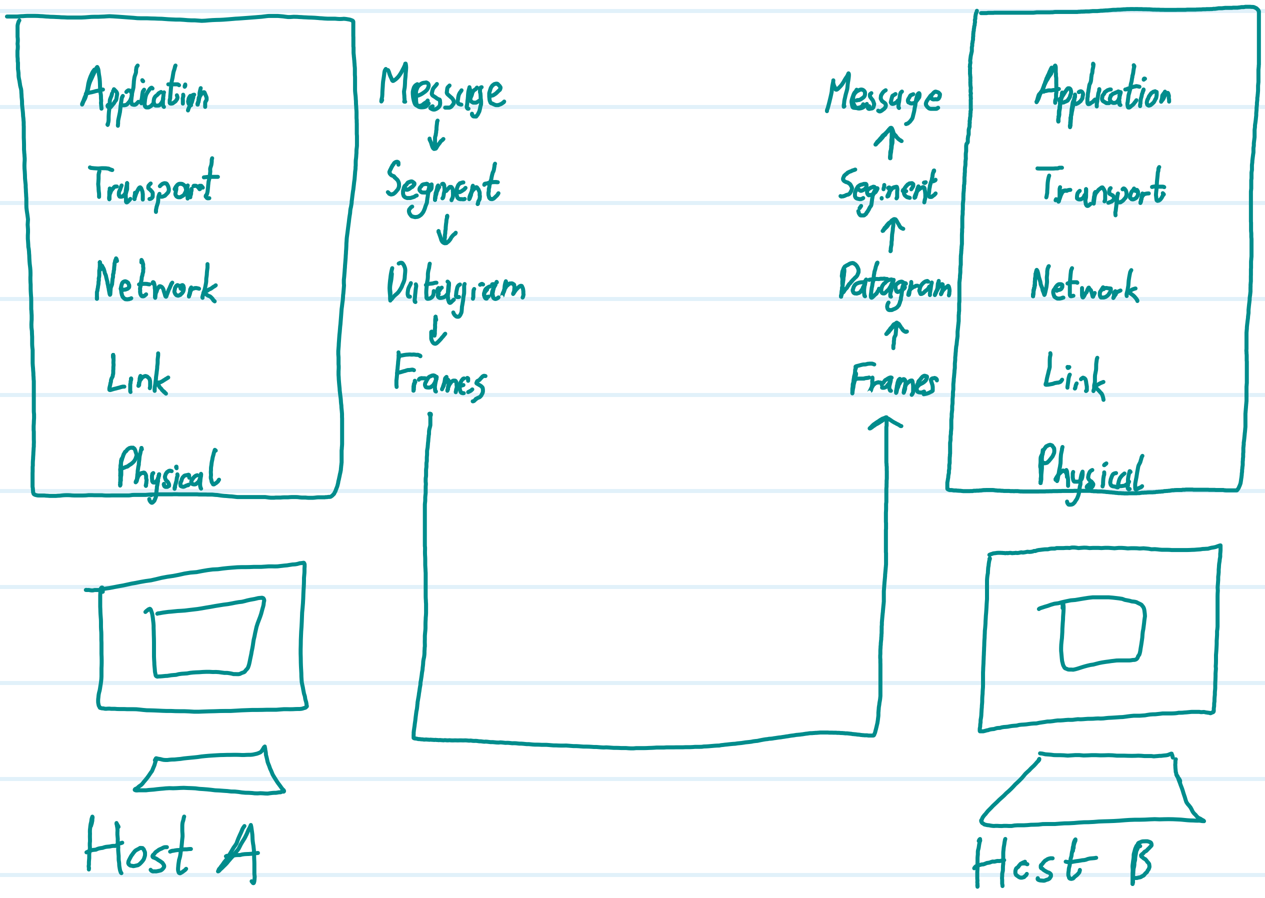
- The transport layer is analogous to the post office in real life. The transport layer is in charge of taking messages from app layer of host A to app layer of host B. The application layer says what needs to be sent and where it needs to go and the transport layer is what takes it there (or more precisely it manages and directs the lower layers to transport the data).

- The network layer is in some sense where the internet starts*. This is when data finally passes into the router and enters the network of networks we call the Internet. This is where routing decisions for getting from source to destination are made. Although there are many protocols at the transport layer, there is only one at the network layer: the Internet Protocol (IP). This unified protocol is what binds the Internet together.

* This is of course not strictly true. All the layers together is what makes the Internet work. I just mean this is when we first enter the network

- The link layer handles moving data between adjacent nodes/routers in the network. While the network layer is in charge of moving data across the whole network, it does so by passing through a series of routers. Each of these steps corresponds to a 'call' to the link layer.

- The physical layer is the lowest layer we will consider. The physical layer is in charge of transporting the individual bits for the link layer. Notice that exact details of this will depend on the <u>physical</u> medium, e.g. the protocol is going to be different for delivering bits over WiFi vs over Ethernet for example. This is partly why it makes sense to separate the link layer and physical layer.



See above for schematic of flow of data As data moves down layers it is given a different name based on which layer it is in. This is not just a semantic difference. As data moves down the layers headers are added (and similarly when data gets to the other end these headers are unpacked), which contain information necessary for the layers below (for example the transport layer may add the address details for the destination, the network layer may add routing details, etc.). This process of adding headers is called encapsulation.

Security

The internet was initially built under the assumption that it would be a network of mutually trusting devices. This is why the 'vanilla' Internet actually has very few security checks in place. This means bad actors can attack in a variety of ways we need to be able to defend against them.

- Bad actors can send malware over the internet This is an umbrella term for small programs or pieces of code that spies on users, steals their information/data, etc.
- Denial-of-service attacks (DoS attacks) are attacks which prevent legitimate users from accesing or using a given service This can be done by sending malicious code to a vulnerable part of the system which causes it to stop or crash. Alternativly, attackers can flood the system by having many connections to the service which prevents other users from accessing the service. A slightly more sophisticated version of this are DDos (distributed Denial-of-service) attacks. This is where a single bad actor controls multiple devices/hosts and coordinates them to flood the victim's system. These are much harder to detect and defend against than DoS attacks.
- Remember that the data is more or less flowing freely through the network. A bad actor can (fairly easily) simply look at all the traffic passing through a router to learn potentially sensitive information. This is known as packet sniffing. Packet sniffing can be difficult to detect. The best way to combat it is to encrypt the data so even if someone sees the packet, they cannot learn anything from it.
- It is (very) easy to spoof your IP This means modifying the contents of a packet to change the source IP address. Because the Internet was built under the idea of mutual trust, the source address on a packet is just assumed to be the true address However this allows bad actors to masquerade as trustworthy users and inject harmful data/code into the system or even the whole network. We can try to fight this using endpoint authentication.

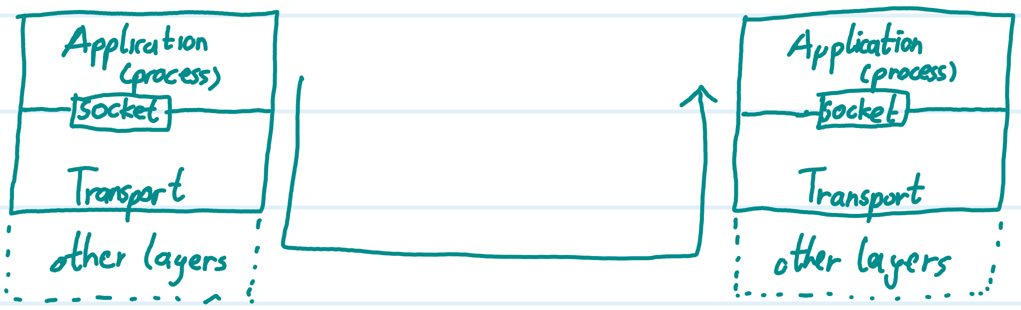
Application Layer

The benefit of the above architecture is that when building a network application, we can abstract away a lot of the underlying complexity. In effect all we need to worry about is what are the services provided by the transport layer and how do we communicate with it. The latter is done through something called the application layer interface or API (application programming interface).

The details of the above will depend on the 'style' of communication. These typically come in 2 flavours. client-server or peer-to-peer. In the client-server case, one of the hosts is a server while the other is a client A server is (typically) a host that is always on with a permanent IP and typically resides in a data center. Clients, on the other hand, may have dynamic IP addresses and only connect with the server intermittently. Moreover clients don't communicate with each other, only with the server. One of the best examples of a protocol that implements this client-server model is HTTP (hypertext transfer protocol).

Peer-to-peer communication, meanwhile, is a model for communication between two arbitrary hosts. In particular there is no 'always on' server and peers may provide and recieve service from one another. Consider the example of file sharing where a peer (i.e. a host) may send or recieve files. Importantly peers need not have a permanent connection to the network and may not have a permanent IP address (this makes things more complicated).

Regardless of the communication model, at the end of the day we have messages that need to be sent between peers (be they clients or servers or peers). These messages originate from processes which are programs being run on the hosts The client process is the process that initiates communication and the server process is the process waiting to be contacted (the same terms are used for the P2P case). A socket serves as an interface between the application layer and the transport layer. Thus all communication will involve 2 sockets: one at the sending end and one at the recieving end.



Sockets require 2 pieces of information destination IP address and port number A host may be running multiple processes with different protocols which require and request different data. The port number helps distinguish these For example port number 80 is (typically) reserved for HTTP. Thus when data comes in to port 80, the server knows how to interpret the message and act on it. If we sent requests/messages based on a different protocol to port 80, this would lead to communication errors (something analogous to two people trying to talk to one another in different languages).

An application layer protocol (such as HTTP) defines the types and format of messages recieved and sent (e.g responses, requests, etc.), where the fields in a message are, what they mean and how to act/react based on response. Open protocols are publicly available for everyone to view/use. They are defined through RFCs (Requests for Comments). It's a bit of a circular definition since RFCs are outlines and reports that define industry standards (typically standards for the internet). The name is a bit of a holdover from early Internet days when these standards were much more open for discussion. The other type of protocols are proprietary protocols meaning they are owned by a company (think Skype, Zoom, etc). The use, definitions, etc. are not publicly available

Now let's look at services that the transport layer can provide us

- data integrity : we may or may not want 100% reliable data transfer Some applications (e.g. file transfers) need all data to be sent whereas other apps (like audio streaming) can tolerate some loss
- timing : online games, for example, need minimal delay to be effective. Email, on the other hand, can tolerate longer delays
- throughput : some applications (like video streaming) have minimum throughput requirements. Applications which don't have such requirements are called elastic
- security : some transport layer services may provide data security (e.g. through encryption)

There are 2 transport protocol services: TCP and UDP.

TCP (Transmission Control Protocol) provides:

- reliable transport from sender to reciever
- flow control so reciever is not overwhelmed

  congestion control, sender is 'calmed down' if network is very busy
- connection-oriented. link is set up between client and server before data sent

UDP provides none of the above. Neither provide security, timing or throughput guarantees. Understandably TCP is the more common protocol. However there are still good reasons to use UDP since the additional features can be built on top of it in the application layer.

HTTP and the Web

First let's quickly recall how webpages work. A webpage is really just a collection of objects stored on one or more server A webpage could be an HTML document, an image, Java applet, etc. Finally, we can access these objects by using their URL.

With this setup, let's look at how HTTP works. This protocol is based on the client-server model discussed above. The process typically looks like so:

HTTP (typically) uses TCP

1) client creates a TCP connection on a socket to a server (on port 80, as per convention)

2) The server accepts the TCP connection

3) Messages are sent back and forth as determined by the protocol

4) Connection terminated

One should keep in mind that HTTP is stateless, in other words, it does not keep track of past clients or past connections So for example if something fails we cannot roll back to a previous state. The upside of this is that it allows HTTP to be a fairly simple protocol

This is where cookies come in!

There are 2 types of HTTP connections. persistent and non-persistent In a non-persistent connection, at most 1 object can be sent over the TCP connection while for the persistent case multiple objects can be sent over the (same) TCP connection Persistent HTTP is HTTP v1.1 which is the most common form of HTTP in use right now.

Suppose a client requests an HTML page which references some images. In a non-persistent setting, the server closes the TCP connection after sending the HTML page so the client would need to (open a TCP connection and send an HTTP request) for each of the referenced images. For persistent HTTP, the server does not immediately close the TCP connection after replying to a request but leaves it open for a while. That way the client can use the same TCP connection to send requests.

HTTP messages

There are 2 types of HTTP messages: requests and responses. Request messages start with a request line that looks something like

command   version of HTTP

GET /index.html HTTP/1.1 \r\n

page being requested   new line character

This request line is followed by various header lines that contain other information such as the domain name of the host, client's web browser, etc. If a body is required, it comes after the header lines.

There are 4 main HTTP request messages. As we've seen the GET method is used to request objects from the server. The POST method is used when a user fills out a form (e.g. when entering a search query into a search bar). In this case the returning page depends on the users inputs (although a lot of websites don't use POST for searching and instead include the search terms in the URL e.g. Google). The PUT method is used to upload objects to servers and the HEAD method is similar to the GET method except it just returns the header of the response and not the object. This is useful for debugging purposes as well as doing things like seeing how big the requested object is without retrieving it.

The HTTP response messages have a fairly similar structure. The first line is the status line which looks like

HTTP version ↘   ↙ corresponding status message

HTTP/1.1 200 OK

↗ status code

The status line is followed by header lines which contain information such as the type of server as well as some metadata about the requested object such as its size, the type of data that it is and even when it was last modified Finally after this comes the requested object itself (if the client had used the HEAD command then this final part does not happen).

Cookies

Finally let's discuss how servers can actually remember past transactions and clients As we know this cannot be done through just HTML as it is stateless. Instead we use something called cookies. When a client first sends an HTTP request to a server, the server sends back an HTTP response as normal but also includes a cookie (which is basically just a unique ID number). The server will also store this cookie along with information about the HTTP transaction in its backend database. From now on whenever the client sends an HTTP request to the server, it will include the cookie in the header and server will be able to keep track of all the transactions and react accordingly.

Web Caches

Our goal is to satisfy client requests without involving the (origin) server in order to improve user-perceived performance while reducing the load on the server.

Suppose you are an institution housing a lot of devices that need to connect to the internet through your limited access link. You can set up a **Web cache server** to hold some of the requested objects (e.g the most recently or frequently requested objects). Users in the institutional network configure their browsers to point at the cache server. Then whenever a user makes an HTTP request to an $\overset{\text{origin server}}{\text{internet}}$ server, it goes to the cache server instead. If the cache server has the requested object, it returns it to the user without ever involving the origin server. If not, the cache will send a request to the origin server and forward the returned object to the user. The cache server will also hold on to this object so if any users make requests for it in the future, it can handle them directly. In fact this last part is partially determined by the origin server which can dictate, in its response, how long the returned object should be cached for (if at all)

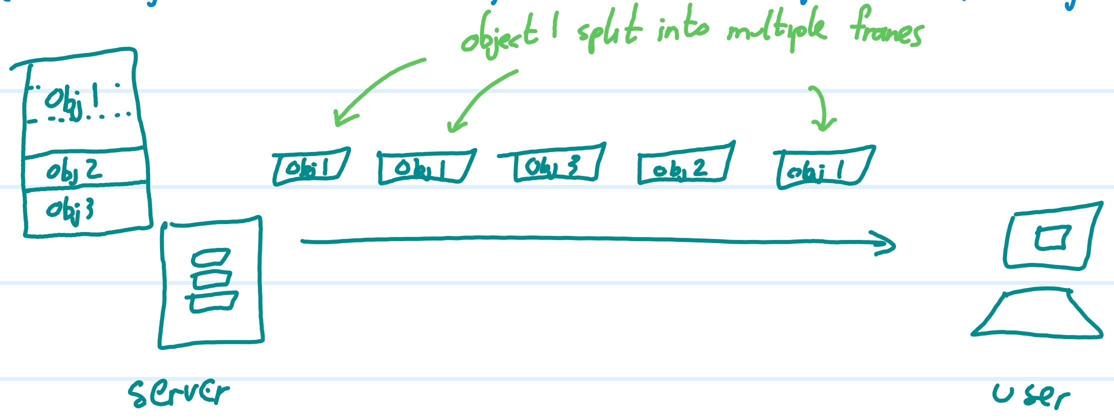
The benefits of a cache server are obvious. Since the cache server is on the same network as users, the latency (in response) is minimal. Furthermore since a large proportion of the requests are handled by the cache server, the load on the institutional access link (the link connecting the users to the wider internet) is reduced

There is a second way that users can use caching without needing a whole extra server, namely through the Conditional GET request The idea is that if a user already has the most up-to-date version of an object, then the user doesn't need to download the object from the server all over again. Thus what one can do is make a Conditional GET request(as opposed to a regular GET request) which has the additional line if-modified-since <date> If the object has been modified since the given date, then the interaction proceeds as normal (the server will (hopefully) reply with an HTTP 200 response and then the requested object) However if the object has not been modified, the server will reply with HTTP/1.0 304 Not Modified and no further data will need to be sent.

HTTP 2 and 3

Improvements have been made to HTTP to give servers flexibility and improve performance (especially from the user's perspective) in HTTP2 and then security and congestion control have been improved in HTTP3 Some of the changes made in HTTP2 are

- transmission order can be based on client-specified priorities
- servers can push objects to clients that have not been requested yet but may be requested in the future
- large files can be divided into frames This way if multiple objects are requested, the server can schedule the frames so as to minimise avg response time of objects. Otherwise small files may have to wait for an entire big file to be sent, before they can be sent.


object 1 split into multiple frames
obj 1
obj 2
obj 3
obj 1
obj 1
obj 3
obj 2
obj 1
server
user