Computer Networks

2018A7PS0193P

CHAPTER 1

Networks

1. Introduction

DEFINITION 1.1. A **network** is a shared infrastructure that allows users to communicate with each other.

The basic building blocks of a network are **nodes** and **links**. Nodes may be hosts or forwarding nodes. The end hosts communicate with one another through the **core network**, consisting of forwarding nodes. These nodes are connected via links called **network edges**.

End hosts are physically connected to the core network via the **access network**. This consists of many parts, such as the ethernet switch, router, etc.

2. Communication Models

Networks could have multiple communication models:

- **Client-Server model**: The client host requests, and receives the service from an always-on server. This server is also an end host, but has some special privileges.
- **Peer-peer model**: Here, there is minimal or no use of dedicates servers, as in BitTorrent. The clients directly communicate with one another.

3. Core Network Models

3.1. Circuit Switching. How is our core network made? One way to do this is via **circuit switching**. There are end-to-end resources reserved for a "call", like on a telephone network. There is no sharing of resources. Call setup needs to be done as a preparatory step. Circuit switching generally can be implemented by two different methods:

- **FDM**, which stands for Frequency Domain Multiplexing. The total frequency bandwidth is divided among the users, allowing them to send data simultaneously.
- **TDM**, which stands for Time Domain Multiplexing. Here the time is divided among the users (perhaps in a round robin fashion). As such, one user gets access to the entire bandwidth of the circuit, but only for a period in time.

3.2. Packet Switching. Another way is through **packet switching**, where data is sent over the net in discrete "chunks", called packets. This is how it is done on the Internet. The host takes the application message, and breaks into packets of length L bits. It then transmits packets into the access networks at transmission rate R, also called the bandwidth. Of course, this means that each packet faces a transmission delay of L/R.

Packet switching uses **store and forward**. The packets are stored at intermediate nodes before sending to the next node. The intermediate node checks for errors in the packets before transmitting, assuring the integrity of the packets.

Unlike circuit switching, packet switching does not provide guarantees of the bandwidth remaining constant. This is because it does not have any reserved end-to-end resources for a "call".

When using packet switching, there may be four sources of packet delay:

- **Nodal processing** : The node performs error checking and checks the header for the destination of the packet.
- **Queueing** : When the arrival rate of the packets is faster than the sending rate, the node will keep the packets in a buffer queue. As such, there is a delay when the packet wait in the node queue.
- **Propagation** : This is the delay from propagation of the packets from a node to the next node, i.e., the outgoing delay. This depends on the medium of the wire, and is given by d/s, where d is the length of the connection and s is the speed.
- **Transmission** : This is the delay from transmission of packets into the link. If the transmission rate is R, the delay for one packet of length L will be L/R.

4. Performance of a Network

The performance of a network can be measured by the following parameters:

- Delay
- Packet loss : This is the number of packets lost when transmitting. Some applications, like streaming, might not care too much about this.
- Throughput : This is the amount of bits transferred in unit time. This is important in some applications, such as for file transfer.

CHAPTER 2

The Internet

1. Introduction

The Internet is, in fact, a network of networks. The networks must be able to communicate despite using different applications running on different devices - i.e. it is heterogeneous.

As such, the Internet is full of different access ISP networks. How do end hosts on different access ISPs communicate with one another? Of course, if we directly connect them all, it would not be scalable as it would need $O(N^2)$ connections. We also cannot use a single global hub, since it would be difficult to find a single place to put it and connect the entire world.

Since a single global ISP cannot scale to connect the entire world, we use multiple global ISPs. These must be interconnected themselves. One way to do this is using **peering links**, which directly link two global ISPs. Another is to use **Internet Exchange Points**, called IXPs, to which multiple global ISPs can connect.

The Internet uses this system in a tiered manner - end hosts might connect to a regional ISP, which may then connect to a higher level country ISP, and so on.

Some corporations, like Google, have their own Content Distribution Networks (CDNs), and have their own network to bring services and content closer to users.

2. Layered Network Model

The Internet is based on a Layered Network Model known as **OSI**. Any device under OSI can have the following layers:

(1) Physical

(2) Data Link

(3) Network

(4) Transport

(5) Session

(6) Presentation

(7) Application

Each of these layers depend on the one below (lower number) and export their services to the ones above (higher number).

The end hosts implement all 7 layers of the model. Those which implement the first 3 layers are called **routers** or Layer 3 devices. Routers are used to connect two different networks. Those which implement the first 2 layers are called **switches** or Layer 2 devices. They connect devices within a network, i.e., in Local Area Networks.

The Internet stack does not actually use all 7 layers - in fact it uses only 5. It removes the Presentation layer, which allows applications to interpret the meaning of data. It also removes the Session layer, which is used for synchronization, check pointing and recovery of data exchange. These functions are generally performed by the Application layer. This Internet stack is known as **TCP/IP model**.

In the context of the Internet, these layers perform the following functions:

(1) **Physical:** This layer delivers bits between the two endpoints of a link, e.g. copper, fiber, wireless, etc.

(2) **Data Link:** This layer delivers packets between two hosts in a local area network. These are bridges and switches.

(3) **Network:** This layer connects multiple networks, e.g. routers. This uses the Internet Protocol (IP).

(4) **Transport:** This layer does process-process data transfer. It may use a multitude of protocols, including TCP, UDP, etc.

(5) **Application:** This layer supports network applications. It may use FTP, SMTP, HTTP, etc.

3. IP Hourglass Architecture

One way to imagine the Internet architecture is as a hourglass. The IP interconnects multiple existing networks, and hides the underlying technology from applications. This provides minimal functionality (has a "narrow waist"). The trade-off of this approach is that there are no assumptions being made, and as such no guarantee that something works.

CHAPTER 3

The Application Layer

1. Network Applications

A **Network application** is a program that runs on different end systems and communicates over a network. These are run only on the end hosts - core network devices do not run user application code.

The application architecture can run different application architectures:

- **Client-Server**: The server is an "always on" host, which has a permanent IP address. To be able to scale, there are generally large data centers acting as a virtual server. The clients communicate with the server. Unlike the server, they may be intermittently connected, and may have a changing dynamic IP address. These clients never directly communicate with one another.
- **Peer to peer** : Here, there is not always on server. Instead the end hosts directly communicate with one another. The peers are connected and may change IP addresses dynamically.
- **Hybrid of client server and peer to peer** : This is the case in Instant Messaging and Skype. In instant messaging, the chatting between two users is P2P, but to get the IP addresses of a user's friends, a central server is needed.

Processes communicate within the same host using interprocess communication, but they must communicate with different hosts by exchanging messages.

Definition 1.1. A **socket** is the interface between the application layer and the transport layer within the host.

A process (a network application) send and receives messages using it's socket. This is a software entity, not a physical one.

To receive messages, each process must have some identifier. The IP address is not enough since it will only uniquely identify the host, but not the individual processes running on the host. So, we also use the port number to identify a process. For instance, an HTTP server would use port number 80, and a mail server would use port number 25.

2. Application Transport Services

What transport services does an application need? This changes from application to application. Here are some examples:

- **Data Loss** : Depending on the application, it might be necessary that data transfer is 100% reliable. For instance, in the case of file transfer, there must be no data loss, but some loss can be tolerated in the case of streaming.
- **Bandwidth** : Applications may also have bandwidth requirements. Streaming needs some minimum amount of bandwidth to work, but elastic applications like email and file transfer will make do with whatever bandwidth is available.
- **Timing** : Another parameter to consider is timing. Some applications, like games, require low delay, but for others this is unnecessary.

3. HTTP

The application layer for web pages is HTTP. A web page consists of objects, like HTML files, JPEG images, etc. Each of these objects is addressable by a URL. HTTP applications use TCP as it's transfer protocol.

(1) Client initiates the TCP connection on port 80. The time taken to establish this is called the **Round Trip Time** or RTT.

(2) Client sends a request to the server

(3) Server receives message through it's socket and sends the response.

(4) Server attempts to close the TCP connection. The client may refuse to do so if the client has not received the message.

(5) The client receives the message and closes the connection.

It is important to remember that the time taken to send a file would be $2RTT$+File Transmission Time - one RTT to establish connection, one RTT to send the first message and the time taken to send the file. Let us say the received file is an HTML file, which as 10 images embedded. Then, the client will request these images as well, by reopening the TCP connection and sending requests.

If at most one object is sent over a TCP connection, it is called **Non-persistent HTTP**. This was the case in HTTP version 1.0. In HTTP version 1.1, **Persistent HTTP** was introduced, which allows multiple objects to be sent over a single TCP connection between client and server. This could be done with or without a pipeline. If there is no pipeline, the client requests for each object, waits for the response, requests the next, and so on. To speed

this up, we can use a pipeline and send requests for multiple objects one after another, while waiting for the response. The responses are always received in the same order in which they were sent.

3.1. HTTP Requests. A HTTP request message consists of:

- A request line. It contains the method (POST,GET,HEAD), the URL being requested, and the HTTP version being used.
- Header lines. Each header line has the header field name and a value. An example of a header line can be `User-Agent: Firefox/3.6.10` or `Keep-Alive:115`.
- The body, which contains all the data of an entity

Each line is delimited by a carriage return character \r and a line feed character `\n`.

The request methods could have the following meanings:

- `GET` : This method is used to retrieve data from the server.
- `POST` : This is used to submit an entity to the server.
- `HEAD` : This asks the server for a response identical to a GET request, but without the response body.
- `PUT` : This is present in HTTP version 1.1. It uploads a file in entity body to the path specified in the URL field.
- `DELETE` : This is present in HTTP version 1.1. It deletes the file specified in the URL field.

3.2. HTTP Responses. A HTTP response message consists of:

- A status line, which consists of the protocol, a status code and a status phrase. An example is `HTTP/1.1 200 OK`.
- Header lines, which are in the same format as in the request message. It could contain the time, the OS, etc.
- The body, which contains the requested data.

The HTTP response status codes are:

- `200 OK` : Request succeeded, requested object later in this message.
- `301 Moved Permanently` : Requested object moved, new location specified in the Location header line.

- `400 Bad Request` : Request message not understood by the server.
- `404 Not Found` : Requested document not found on this server.
- `505 HTTP Version not supported` : The HTTP version used in the request is not supported by the server.

3.3. States and HTTP. HTTP is a stateless protocol - it does not save anything from it's previous actions. To save the state in HTTP, we use **cookies**. When a client sends a http request to the server, it will create an ID for the user and create an entry in the backend database. Now, when the server sends it's response, it will include a header line called `set-cookie`. From then onwards, the usual HTTP request message will send the cookie ID through a header line called `cookie`.

Hence, using cookies, we can save state with HTTP. This can be used to save user information and track them on other sites.

3.4. Proxy Servers. Proxy servers are also called web caches. Clients send their requests to a proxy server, which would forward it to the destination. The incoming responses are also forwarded to their respective clients. The proxy server can cache information, and hence answer requests itself to save time. This is especially useful in institutions when the content that users access has a large overlap.

One issue with this is that the cached data may turn stale. This problem is solved by the means of **Conditional GETs**. When a proxy server receives a response from the origin server, it will cache it before forwarding it to the end host. This response will have a header line called `Last-modified`, reflecting the time when that web page was last modified. Now the next time an end host requests that page, the proxy server will forward that GET request but with a header line `If-modified-since`, along with the date we got from the `Last-modified` header line. This effectively asks the server the question - has it been modified in the time since I last requested it? If it has, then the server removes the normal response, but otherwise it will get a `304 Not Modified` response. This response will not have any data, saving on bandwidth and time. This way, the proxy server is able to maintain the freshness of the cached data.

3.5. HTTP/2 Protocol.

Definition 3.1. A stream is a bidirectional sequence of text format frames sent over the HTTP/2 protocol exchanged between the server and client.

HTTP/1 was capable of transmitting only one stream at a time. This made receiving large amount of media content inefficient and time consuming. HTTP/2 allows transmission

of parallel multiplexed requests and responses. A binary framing layer is created, which allows the client and sever to disintegrate the HTTP payload into small independent and manageable interleaved sequence of frames. This information is then reassembled at the other end.

HTTP/2 also allows the server to send additional cacheable information to the client that isn't requested but is anticipated to be needed in future requests. This mechanism saves a RTT and reduces network latency. This is called **Server PUSH**.

4. Domain Name System

When a client requests the URL, it must obtain the IP address of the destination host to send the request to. The **Domain Name System** maps the name people use to locate a website to the IP address that a computer uses to locate a website.

The DNS is formed by a distributed hierarchical database. The 13 **root DNS** servers contain reference to the **top level domain** servers, like the .com servers, the .org servers, etc. These then contain references to the **authoritative servers**, which are maintained by organizations with publicly accessible hosts to map the hostnames to IP addresses. Examples of this would be servers for yahoo.com or google.com.

One issue we would face is that since we would need to traverse the hierarchy recursively to query a host, it can take a long time to process a DNS query. In practice, instead of traversing the tree on a direct path, the local DNS server would first get information about the TLD DNS server from the root server, and then contact the TLD server directly, instead of letting the root DNS server do this. This is done recursively, contacting lower and lower levels in the hierarchy directly from the local DNS server and getting the IP of the next level.

DNS responses are generally cached to improve the delay performance and to reduce the number of DNS messages.

DNS provides the following services:

- **Host name to IP address mapping** : Websites are generally identified by their hostname. DNS provides a mapping of hostnames to IP addresses, which is what the end host needs to connect to the server.
- **Host aliasing** : Sometimes, hosts have complicated names, and hence can have one or more alias names. The non-alias hostname is called the canonical hostname. DNS can also be used to find the canonical hostname for a supplied alias.

- **Mail server aliasing** : It is highly desirable that an email address is easy to remember. The hostname of the Gmail mail server may be very complicated. DNS can be invoked by a mail application to obtain the canonical hostname for the given email address alias.
- **Load distribution** : DNS is used to perform load distribution in the case of replicated servers, where one canonical hostname is associated with multiple IP address. The DNS server rotates these IP addresses upon every query to ensure load management.

The DNS servers store the necessary information in the form of resource records(RR). They are in the format `(name, value, type, ttl)`. Here `ttl` stands for "time to live", which tells the DNS resolver how long to cache a query before requesting a new one. The values of the other fields depends on the type of the resource record. There are four types of resource records, depending on the `type`:

- `type=A` : `name` refers to the host name, and `value` refers to the IP address.
- `type=NS` : `name` is the domain name, and `value` is the host name of authoritative name server for this domain.
- `type=CNAME` : `name` is the alias name for some canonical name, `value` is the canonical name.
- `type=MX` : `value` is the name of the mail server associated with the `name`.

To insert a record into DNS, a newly created domain name should be first registered at a registrar.

5. File Transfer Protocol

This protocol is used to transfer file from client to client. It uses the TCP protocol, since it must be reliable and error free.

In a typical session, FTP will use two connections - a control connection and a data connections. The control connection, which is generally established on port 21, is the primary connection and is used to send commands back and forth between the client and the server. The data connection, established on port 20, is used solely to transfer the requested data. It stays open until the transfer is complete. Since FTP uses two parallel TCP connections to transfer a file, it is said to send it's control information **out of band**. This is in contrast to HTTP, which works **in band**.

Unlike HTTP, the FTP server must maintain the state of the user. In particular, the server must associate the control connection with a specific user account and keep track of the user's

current directory as the user wanders about the remote directory tree. This unfortunately constrains the total number of sessions that FTP can can maintain simultaneously.

Some FTP commands are:

- `USER username`
- `PASS password`
- `LIST` returns list of file in current directory
- `RETR filename` retrieves file
- `STOR filename` stores file onto remote host

These commands can give return codes and phrases as in HTTP:

- `331 Username OK, password required`
- `125 data connection already open; transfer starting`
- `425 can't open data connection`
- `452 Error writing to file`

6. eMail

e-Mail consists of three major components

- User agents, e.g. Outlook, mutt
- Mail servers, Contains incoming messages for user
- Simple Mail Transfer Protocol (STMP)

A user creates a mail and sends it to their mail server using the user agent. The sender's mail server will forward this mail to the recipient's mail server over a TCP connection on port 25. The recipient's user agent will then try to access this mail from the server. The protocol used in this case is called **Mail Access Protocol**, like IMAP or POP3. Everywhere else in this process, we use SMTP. We cannot use SMTP for the recipient to access the mail, since SMTP is a push-based protocol, not a pull based protocol.

POP3 is the Post Office Protocol. It allows the user to download and keep the mails. The user can create folders and move the messages into them locally. It is stateless across sessions.

A more feature-rich protocol is IMAP or the Internet Mail Access Protocol. It allows the user to create remote folders and maintains user state information across sessions, It also

permits a user agent to obtain components of messages, which is ideal for low bandwidth connections.

The user agent could be web-based, like in Hotmail or Gmail. Now, the transfer of the message from the sender to the mail server happens over HTTP. The access of mails by the receiver is also done over HTTP.

7. Peer To Peer Architecture

As mentioned before, there is no always-on server in peer-to-peer architectures. The end hosts directly communicate, and the peers are intermittently connected.

Let us say we want to send N file copies of size F. In a client-server system, ignoring the delays, the time taken to send to a server would be NF/u_s, where u_s is the upload rate. Each client must download a file copy, and it the slowest download speed is d_{min}, then the slowest download time would be F/d_{min}. So, the minimum time to distribute to the N clients would be:

$$D_{c-s} \geq \max(NF/u_s, F/d_{min})$$

In the case of P2P system, at least one copy must be uploaded, taking time F/u_s. Each client must download a file copy, with the slowest download time once again being F/d_{min}. Unlike in the client-server scheme, as the file download gets completed on peers, the number of possible "providers" increases. As this happens the maximum upload rate increases, up till $u_s + \sum u_i$. So, the time needed is:

$$D_{P2P} \geq \max(F/u_s, F/d_{min}, NF/(u_s + \sum u_i))$$

In BitTorrent, a peer-to-peer application, the file is divided into chunks, typically 256 KB in size. There are trackers that track peers participating in a torrent. A new peer joins a torrent and registers with the tracker to get a list of peers, and connects to some subset of them. While torrenting, a group of peers exchange chunks of a particular file. At any given time, each peer will have a subset of the chunks, and asks its neighbours for a list of which chunks they have. The peer will then take a call on which chunks it should request from the neighbour, and to which of its neighbours it should send requested chunks. Ideally, the peer would request and send the most scarce chunks.

One P2P protocol is **Napster**. Here, we have a central database where information is available, and the peers contact it for the necessary information. However, if the database is down then the peer-to-peer application cannot run.

Another P2P protocol is **Gnutella**. Unlike Napster, it is a completely decentralized protocol. On startup, an end host finds at least one other node to connect to. It then queries this

node for more nodes to connect to, until it reaches some quota. Due to the completely decentralized network, this protocol is not scalable - searching takes time exponential in the number of nodes, and often end hosts are only connected intermittently.

Yet another P2P protocol is **Kazaa**, which is the basis for Skype. Kazaa is also a decentralized system. The users are divided into two groups - supernodes and ordinary nodes. Supernodes are powerful computers that act like traffic hubs, processing data requests from ordinary nodes.

8. Distributed Databases

Distributed databases are a common application of the P2P framework. In a distributed database, each peer holds a small subset of the total (key,value) pairs. Any peer can query the distributed database with a particular key. The distributed DB locates the peers that have the corresponding (key,value) pairs and return it to the querying peer. Any peer can insert a new (key,value) pair into the database.

A non-scalable way of doing this is as a distributed hash table. The key,value pairs are randomly scattered across all the peers. They maintain a list of the IP addresses of all the peers, and send their queries to all these peers. Those who contain the required (key,value) pairs respond with the matching pairs. This is, as mentioned before, not scalable since all peers must be queried.

Instead we may implement this with the **Circular DHT** protocol. A hash function assigns each node and key an m bit identifier using a base hash function such as SHA-1. The node's ID could be a hash of the port and the IP address, while the key's ID can be a hash of the original key. Since there is a finite number of hashes, these ID values will lie on an imaginary circle, ranging values from 0 to 2^{m-1}. We now assign (key,value) pairs to the peer that has the next closest node ID to the pair's key ID.

One protocol that uses this concept is the **Chord protocol**. Every node keeps track of 2 pointers - the predecessor, a pointer to the previous node on the ID circle, and the successor, a pointer to the next node on the ID circle. When doing a lookup operation, a node will check whether the key's ID is between the ID of the node and it's successor. In this case, it will know that the key is present in the successor. If not, the query is forwarded to the successor, and the same operation is done recursively. The number of messages is $O(n)$, where n is the number of nodes. So, this is not scalable.

A scalable way to do this is that each node n contains a routing table with up to m entries (m is the number of bits), known as a **finger table**. The i^{th} entry in the table at node n contains the first node s that succeeds n by at least 2^{i-1}, i.e. $s = succ(n + 2^{i-1})$. s is called

the i^{th} **finger** of node n. With this information, we can jump to the furthest successor that precedes the key ID.

In a P2P system, peers can connect and disconnect without warning. Thus, when designing the DHT, we have to be able to maintain the correct successor pointers. To achieve this, each node maintains a successor list of its r nearest successors on the ring. If node n notices that it's successor has failed, then it will replace it with the next entry on the list. This checking needs to be done often based on the frequency of nodes leaving and joining.

9. Socket Programming

A socket is created whenever we desire communication between two applications running on different machines. In UNIX, this is merely another file descriptor. The socket is created with the `socket(domain, type, protocol)` system call. The domain contains the address family, the type specifies the semantics of communication, e.g. SOCK_STREAM for stream sockets, SOCK_DGRAM for datagram sockets, etc. The protocol specifies whether it is UDP (IPPROTO_UDP) or TCP(IPPROTO_TCP).

To establish a connection, the client does the following:

(1) Create a socket

(2) Connect the socket to the address of the server

(3) Send/Receive data

(4) Close the socket

To establish a connection, the server does the following:

(1) Create a socket

(2) Bind the socket to the port number known to all clients.

(3) Listen for the connection request

(4) Accept the connection request

(5) Send/receive data

The socket address is stored in the following struct

```
struct sockaddrs {
  unsigned short sa_family; // address family, AF_xxx or PF_xxx
  char sa_data[14]; // 14 bytes protocol address
};
```

AF stands for address family and PF stands for protocol family. For IPv4 internet protocols, we use AF_INET.

There is also the following construct, that contains information about the address family, port number, internet address and the size of the struct sockaddr.

```
struct sockaddr_in{
  short int sin_family; // Address family
  unsigned short int sin_port // Port number
  struct in_addr sin_addr; // Internet address
  unsigned char sin_zero[8]; // same size as struct sockaddr
}
```

Some systems are little endian, while others could be big endian. To communicate between these two systems, a standard has been defined for the data representation in the network called **Network Byte Order**, which is big endian. The system calls that help us convert a short/long from host byte order to NBO and vice versa are:

- `htons()` : Host to Network short
- `htonl()` : Host to Network long
- `ntohs()` : Network to Host short
- `ntohl()` : Network to Host long

CHAPTER 4

Transport Layer

1. Introduction

The transport layer is responsible for logical communication between the hosts, unlike the network layer which is responsible for logical communication between processes. Since the transport layer works on top of the network layer, it is constrained by the underlying network layer protocol. However, in some cases it can offer services even when the network layer doesn't offer it.

The transport layer provides two protocols - UDP, an unreliable connectionless service, and TCP, a reliable, connection oriented service. The transport layer works with transport layer packets called **segments**.

2. Multiplexing and Other Functions

The transport layer multiplexes messages at sending time by handling data from multiple sockets and adding a transport header for each message. It is also responsible for demultiplexing the received messages by using the header information to deliver the received messages to the correct sockets. These are the only two services UDP provides, but TCP provides some additional services. As mentioned before, it also proved reliable data transfer. Besides this, it provides what is called congestion control. TCP congestion control prevents any one TCP connection from swamping links and routers between hosts with excess traffic, and tries to give every connection an equal share of traffic.

In **connectionless multiplexing**, as seen in UDP, the sockets are identified by a port number. So to perform multiplexing, the segments must contain information of a source port number and a destination port number.This information is passed from the sender's transport layer to it's network layer, which encloses it within a **datagram** (network layer packet). This datagram will now also have information about the sender's IP address and the destination IP address. Finally, when this message reaches the destination, that host checks the destination port number and redirects the message to that port. The source port number seems unnecessary, but acts as a "return address" for the application to use to communicate.

As such, a UDP segment has the following content:

- 16 bit Source port number
- 16 bit Destination port number
- 16 bit length
- 16 bit checksum, which is used to detect errors. This is calculated by treating the segment contents as a sequence of 16-bit integers, and summing them all. The one's complement of the sum is put in this field.
- The rest is the application data

TCP uses **connection oriented multiplexing**. Unlike a UDP socket, a TCP socket is identified by a 4-tuple of (source IP address, source port number, destination IP address, destination port number). Thus, when a TCP segment arrives from the network to a host, the host uses all four values to demultiplex the segment to the appropriate socket. In contrast with UDP, two arriving TCP segments with different source IP addresses or source port numbers will be directed to two different sockets.

Knowing that UDP is unreliable, one might wonder why we use UDP at all. Some reasons are as follows:

- No connection establishment, which would add delay
- It is simple, since there is no connection state to maintain at the sender and the receiver end
- It has a small header size
- It has no congestion control. While this is undesirable in some cases, it means that UDP can run as fast as desired.

3. Designing a Reliable Data Transfer

Let us assume that the underlying channel is perfectly reliable. Then, let us design a protocol called `rdt1.0`. We diagrammatically show this as two Finite State Machines - one for the sender, and another for the receiver.

As we can see in Fig 1, sender simply sends data into the underlying channel, and the receiver simply reads from the underlying channel.

What if our underlying channel has some bit errors? Then the packet may have some flipped bits! For this we design our second protocol, `rdt2.0`. For this, we can use the checksum to detect the bit errors. If there were no bit errors, the receiver will return an **acknowledgement** (ACK), explicitly telling the sender that the packet is OK. If not, the

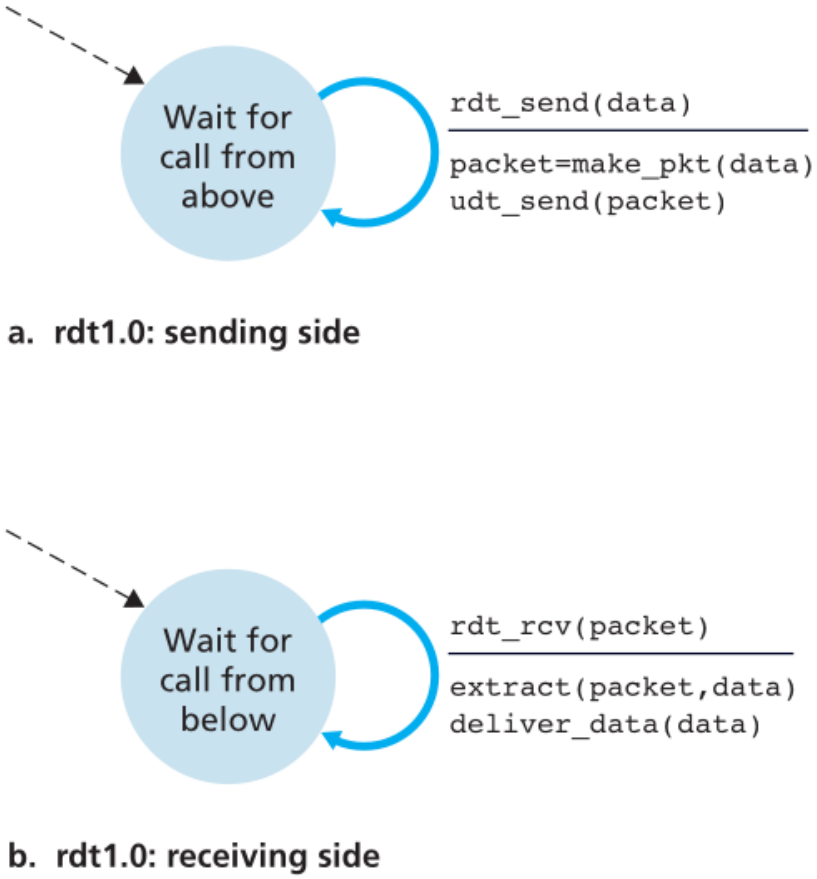

FIGURE 1. FSM for rdt1.0

receiver will return a **negative acknowledgement** (NAK), saying that the packet had errors. In this case, the sender will retransmit packets. The FSM for the same is in Fig 2.

However, this has a fatal flaw. What if ACK/NAK is corrupted? One way to handle this is to be safe, and send the packet again if the ACK/NAK is corrupted. But this creates a new problem of handling duplicate packets, where the receiver is unsure if this is new information of old data. To fix this, the sender adds a sequence number to each packet. The receiver checks this to decide which packet to discard. For instance, say A sends a packet with sequence number 0 to B. B sends back an ACK, but it gets corrupted on the way back. Then, A will send the packet again with the same sequence number. However, B will know this is a duplicate, since it has the same sequence number as the last packet. In this mechanism (called a stop and wait mechanism), we need only 1 bit for the sequence numbers.

This creates a new protocol - `rdt2.1`. The FSMs are as in Fig 3 and 4.

FIGURE 2. FSM for rdt2.0

We can remove the need for NAKs by sending ACK for the last correctly received packet instead. If a sender receives two ACKs for the same packet, it knows that the receiver did not correctly receive the packet following the packet that is being ACKed twice. This new protocol is `rdt2.2`.

Let us now assume that the underlying channel could also lose packets entirely. `rdt2.2` gives us the base to start from, but we need to handle the lost packets. We put the burden of detecting and recovering from lost packets on the sender. Suppose that the sender transmits a data packet and either that packet or the receiver's ACK of that packet gets lost. If the sender is willing to wait long enough so that it is certain that a packet has been lost, it can retransmit the data packet. But how long is long enough? We could choose the worst case, but that is difficult to approximate and may be too slow. So we just choose some time value that packet loss is very likely, but not guaranteed to have happened.

Even in this case, we could have duplicate data packets, since the ACK could just have been received very late due to some delays, even if the packet was not lost. Luckily, rdt2.2 can

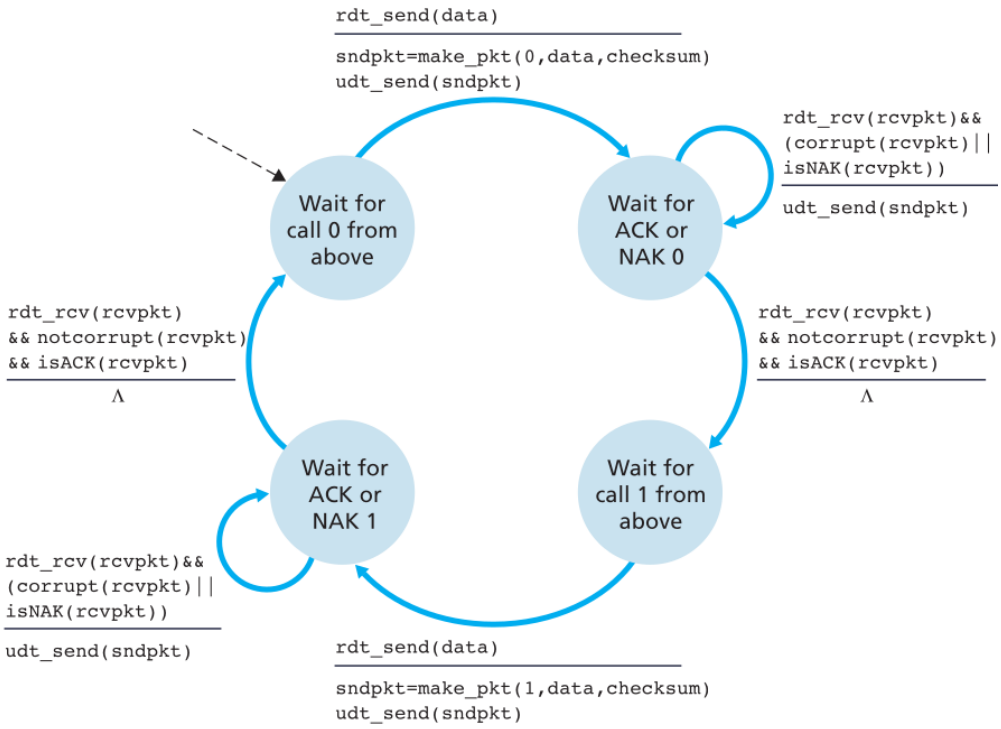

FIGURE 3. rdt2.1 Sender

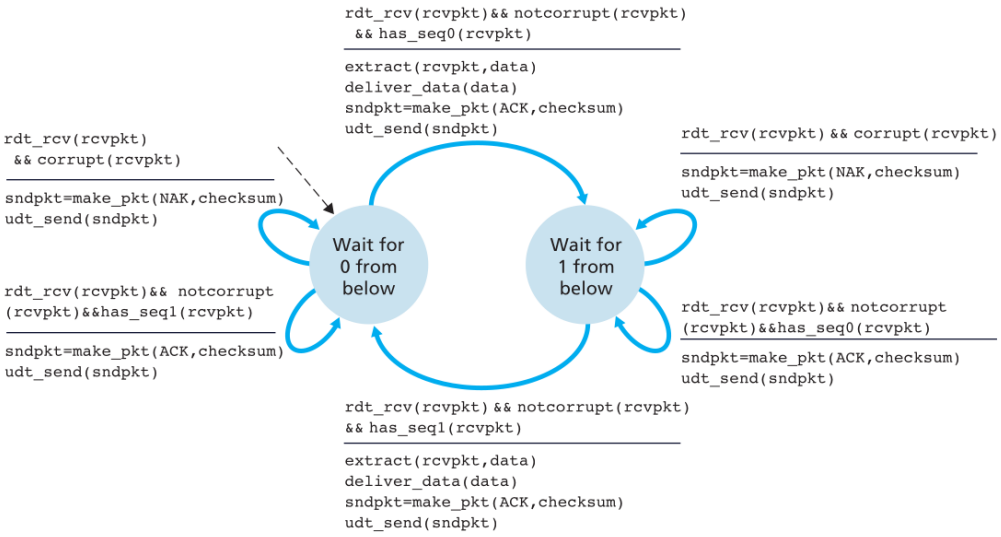

FIGURE 4. rdt2.1 Receiver

already handle this case using sequence numbers. Thus, we now have `rdt3.0`. Since the sequence numbers are either 0 or 1, it's also called a alternating bit protocol. In Fig 5 we can see the FSM for the sender in `rdt3.0`.

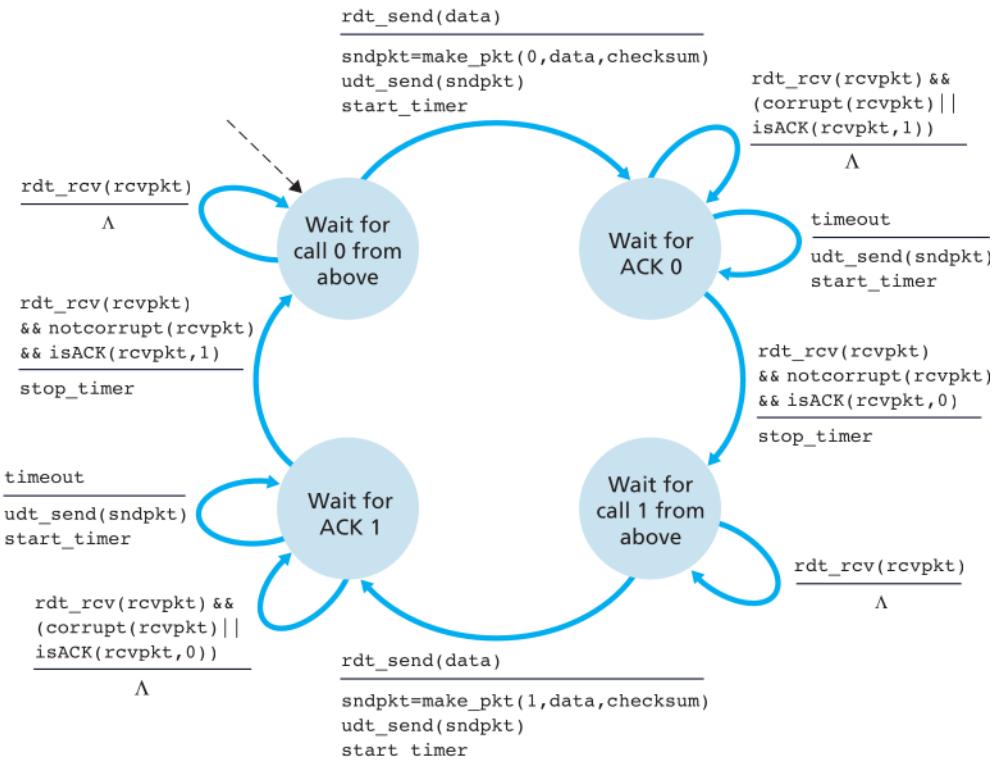

FIGURE 5. rdt3.0 sender

`rdt3.0` is good, but it has it's own issue. Let us define **utilization** of the sender as the fraction of time the sender is actually busy sending bits into the channel. If we are sending L bits per packet, and R is the rate of transmission, the sender will spend $\frac{L}{R}$ time sending bits. However, the time taken to receive the acknowledgements for the packet will be RTT. So,

$$U = \frac{L/R}{RTT + L/R}$$

This U, if calculated, is generally found to be dismal in the case of `rdt3.0`. Instead, we use a pipelined protocol.

REMARK. As an aside, we can also define the utilization as the fraction of the bandwidth being used. So if we take T time to send L bits, the amount of bandwidth being used is L/T. This is also called the effective bandwidth or the throughput. As such, we can define the utilization as:

$$U = \frac{Throughput}{R}$$

where R is the total bandwidth of the connection.

In a pipelined protocol, the sender may send multiple packets despite not having received the ACK. If one sends n packets at once, we can increase the utilization by a factor of n. However, this comes with multiple consequences:

- The range of sequence numbers must be increased.
- The sender and receiver sides of the protocols need to buffer more than one packet.

We can have two pipeline protocols - Go-Back-N (GBN) or Selective Repeat (SR).

In a GBN protocol, the sender is allowed to send multiple packets without waiting for an ACK, but it is constrained to some maximum allowable number N. Let us define `base` as the sequence number of the oldest unacknowledged packet and `nextseqsum` to be the smallest unused sequence number. Then, we have four intervals:

- `[0,base-1]` : corresponds to the packets that have already been transmitted and acknowledged.
- `[base,nextseqsum-1]` corresponds to packets that have been sent but not acknowledged.
- `[nextseqnum,base+N-1]` can be used for packets that can be sent immediately
- `base+N` and greater cannot be used until an unacknowledged packet currently in the pipeline has been acknowledged

So, the window of permissible numbers transmitted but not yet acknowledged packets can be viewed as a sliding window of size N.

In the GBN protocol, an ACK for a packet with sequence number n will be taken as cumulative acknowledgement, indicating that all packets with a sequence number up to and including n have been correctly received at the receiver. Just like in `rdt3.0`, a timer is used to recover from lost data or acknowledgement protocols. If a timeout occurs, the sender resends all packets that have been previously sent but not acknowledged.

In GBN, the receiver also discards out of order packets. This means that the receiver need not buffer out of order packets. The receiver hence maintains the sequence number of the next in order packet, called `expectedseqnum`. If an out of order packet is received, it is discarded. This is equivalent to saying that GBN has a receiver's window of size 1.

GBN has it's downsides:

- If window size and bandwidth-delay product are both large, many packets can be in the pipeline.

- A single packet error can cause retransmission of many packets

Selective Repeat avoids unnecessary retransmissions by having the sender retransmit only the packets that it suspects were received in error at the receiver. This means that the receiver must individually acknowledge the correctly received packets. Like in GBN, a window of size N will limit the number of outstanding unacknowledged packets in the pipeline, but unlike GBN, the sender will have already received ACK for some of the packets in the window. Of course, the window will not move past any unacknowledged packets until they have been acknowledged.

The SR receiver will also ACK packets even if they are received out of order. Out of order packets are buffered until any missing packets are received, at which point they are delivered to the upper layer.

In SR protocols, the sender and receiver do not always have the same windows. This lack of synchronization can result in issues when we deal with windows of finite sizes. It is possible to show that the window size should ideally be less than or equal to half the size of the sequence number space.

4. TCP

4.1. Introduction. Just like the protocols we have designed before, TCP is a point to point protocol that allows bidirectional flow of data. It provides a reliable in-order byte stream - this means there are no message boundaries. It is a pipeline protocol, where the window size is set by the congestion and flow control in TCP. Unlike UDP, it is connection oriented.

4.2. The TCP Segment. The important fields in a TCP segment are:

- 16 bit Source port number
- 16 bit Destination Port number
- 32 bit sequence number, indicating the sequence number of the first byte.
- 32 bit acknowledgement number. If a 200 byte message is sent with sequence number 100, then the receiver will return a segment with an acknowledgement number of 300, indicating that sequence numbers up to 299 has been received, and we expect to receive sequence number 300 next.
- 4 bit header length. This indicates that the header length is not always fixed, since the options in the header can be of variable length
- Urgent data flag. It is generally unused.

- Valid acknowledgement bit, indicating whether or not the acknowledgement number is to be considered. If it is 0, then the acknowledgement bit can be a garbage number.
- PSH flag, telling to push the data now
- RST,SYN,FIN flags, used to setup and destroy connections.
- 16 bit receive window, which is the number of bytes that the receiver is willing to accept. This is used for flow control.
- 16 bit checksum, used to detect bit errors during transmission.

4.3. Handshaking. Before exchanging data, the sender and receiver must do a handshake, agreeing on the parameters of the connection. One way to do this is a 2 way handshake - the sender "asks" for a connection with the receiver, and the receiver responds with a "yes" or "no". If it is a "yes", then we can establish it. However, this is not actually enough. Since the underlying channel is not reliable, the reply (yes or no) could be dropped, and the receiver is unsure if the sender has received the reply. Hence, a third segment is sent, confirming the receipt of the "yes" or "no". This is called 3-way handshaking. Formally, the steps are as follows:

(1) The client chooses an initial sequence number x and sends a TCP SYN message. This message has it's SYN flag set to one. The segment has no application layer data - it is just used to establish a connection. After doing this, the client enters the SYNSENT state.

(2) On receiving the SYN message, the server enters the SYN RCVD state. The server chooses an initial sequence number y and sends a TCP SYNACK message. This has it's SYN and ACK flags set to 1. The acknowledge number will be equal to $x + 1$, indicating the server expects to receive a sequence number $x + 1$ next.

(3) The client on receiving SYNACK believes the server is live, and enters the ESTAB state. It now sends an ACK for the SYNACK, so the ACK bit is set to 1 and the acknowledgement number is $y + 1$. This segment may contain application data. If it does, this is called **piggybacking**.

(4) The server receives ACK and believes the client is live. The connection has been established, and the server also enters the ESTAB state.

4.4. Sending Data. When sending data from the client to the server, the client passes a stream of data through the socket, into the hands of the transport layer and the TCP protocol. TCP directs this data to the send buffer of the client, and when possible, will forward chunks of data from it to the network later. The maximum amount of data that can

be placed in a segment is called the **maximum segment size** or MSS. This is decided by considering the length of the largest link-layer packet that can be sent, called the **maximum transmission unit** or MTU. The MSS is a multiple of 8 bytes chosen such that the TCP segment and the TCP/IP header length will fit into a single link layer frame. This stream of data is received by the server in the receive buffer and duly processed. This process happens on both sides of the connection, since TCP is bidirectional.

To prevent the sender from overflowing the receiver's buffer, TCP provides **flow control**. The sender maintains a variable called a **receive window**, used to give the sender an idea of how much free buffer space is available to the receiver. Let `LastByteRead` be the number of last byte in the data stream read from the buffer by the application process in B, and `LastByteRvcd` be the number of the last byte in the data stream that has arrived from the network and has been placed in the receive buffer at B. Since TCP must not overflow the allocated buffer, we must have:

```
LastByteRvcd - LastByteRead <= RcvBuffer
```

The receive window denoted by rwnd is then the amount of spare room in the buffer, given by

```
rwnd = RcvBuffer - (LastByteRvcd - LastByteRead)
```

While the receiver B keeps track of `rwnd`, A keeps track of two variables - `LastByteSent` and `LastByteAcked`. `LastByteSent - LastByteAcked` is obviously the amount of unacknowledged data A has sent into the connection. A makes sure this amount is less than the value of `rwnd` to ensure that the receive buffer is not being overflowed.

If `rwnd` is ever 0, A will keep sending segments with one data byte to B to check whether space has freed up in the receive buffer.

4.5. Closing The Connection. When closing the connection, the following steps are taken:

(1) The client sends a closing request using `clientsocket.close()`. This segment will have a sequence number of x and have the FIN flag set. The client will no longer send data, but will be able to receive data. The client is said to be in the FIN_WAIT_1 state.

(2) The server responds with an ACK. The server enters the CLOSE_WAIT state, while the client (on reception) enters the FIN_WAIT_2 state. The client is now waiting for the server to close.

(3) The server will enter the LAST_ACK state, and sends it's own closing segment, where FIN bit is set to 1. The client will acknowledge this with an ACK, which will put the server in a CLOSED state.

(4) The client now enters a TIMED_WAIT state, where it will wait for some predetermined time (say 30 seconds) before considering the connection closed. This allows the client time to resend the final acknowledgement in case the ACK is lost.

From the above discussion, it is clear that unlike the connection establishment, connection closure is a 4-way handshake.

4.6. Timeouts. TCP, just like the `rdt` protocol, uses a timeout mechanism to recover from lost packets. How do we decide the length of the timeout interval? It must be larger than the RTT, but how do we estimate it?

To choose the timeout interval, the sender first finds the RTT from the last segment sent, called the $SampleRTT$. TCP only does this for a single transmitted (but yet unacknowledged) segment at a time, and only those which have been transmitted only once. This value will obviously fluctuate with time, and hence TCP maintains an exponential weighted moving average called $EstimatedRTT$ calculated by the formula:

$$EstimatedRTT = (1 - \alpha)EstimatedRTT + \alpha SampleRTT$$

We also estimate the deviation of $SampleRTT$ from $EstimatedRTT$ in $DevRTT$, using the formula:

$$DevRTT = (1 - \beta)DevRTT + \beta|SampleRTT - EstimatedRTT|$$

Then, the timeout interval is generally chosen as:

$$TimeoutInterval = EstimatedRTT + 4DevRTT$$

In the case of a timeout, TCP retransmits the not-yet-acknowledged segment with the smallest sequence number, and doubles the next timeout interval. Whenever the timer is next started after ACK is received or data is received from the application layer, the timeout interval is restored back to the one derived from our formula.

4.7. Fast Retransmit. Let us see how the receiver generates ACKs depending on the situation:

- If an in-order segment arrives with the expected number, such that all data up to the expected sequence number has already been acknowledged, the receiver will delay the sending of the ACK. It does this to wait for the arrival of another in-order

segment, so that it can ACK that as well. If no more in-order segments arrive, it will send the ACK

- Assume the previous case, but one more in-order segment arrives. Then the receiver will immediately send a single cumulative ACK
- If an out-of-order segment arrives, that has a higher sequence number than expected, we have a "gap". The sequence numbers in the gap have not been received yet. In this case, we send a duplicate ACK, which has an acknowledgement number equal to the sequence number of the next expected byte. That sequence number would be the lower end of the gap.
- If a segment arrives that partially or completely fills the aforementioned gap, we immediately send ACK, as long as it starts at the lower end of the gap.

Since the sender will send a large number of segments at once, it will receive many duplicate ACKs. If the sender receives 3 or more duplicate ACKs, it takes this as indication that the segment following that has been lost. Then, it performs a **fast retransmit**, retransmitting the segment before the segment's timer expires. This way, we can quickly recover from lost segments.

4.8. Congestion Control. Congestion can be created when there are links of different bandwidths going in and out of a node. For instance, say a node is receiving two links of bandwidths 10 Mbps and 100 Mbps, but it has an outgoing link of bandwidth 1.5 Mbps. This results in queueing, and packets could even be lost when the queue exceeds capacity. As such congestion is the result of sending too much data too fast for the node to handle.

Let us consider a theoretical case. We have two senders and two receivers, using one router with infinite buffers. The output link capacity is R. Since the router has infinite buffers, there is no possibility of packet loss, since they can be queued indefinitely. The senders will also never need to retransmit due to timeout, since it would be queued in the router.

Obviously, the outgoing throughput will increase along with the incoming throughput, until the incoming throughput is $R/2$. After that, the extra packets will be queued, but the outgoing throughput will remain stagnant at $R/2$.

Now let us remove some degree of impossibility from our scenario - let our router have finite buffers. Since packets can be dropped, the sender needs to retransmit packets on timeout. This means that while at the application layer, the incoming throughput (λ_{in}) will be equal to the outgoing throughput (λ_{out}), the transport layer will face a higher incoming throughput (λ'_{in}). Hence, $\lambda'_{in} \geq \lambda_{in}$.

Due to the retransmission of the dropped packets, the outgoing throughput decreases:

$$\lambda_{out} + \text{Cost of retransmission} = \frac{R}{2}$$

The basic principle behind TCP congestion control is that sending rate is a function of perceived congestion. Congestion is perceived by measuring the amount of packet loss. This is quantified in the form a **congestion window** denoted by `cwnd`. The number of sequence numbers that have not been ACKed cannot exceed this congestion window, i.e.:

$$LastByteSent - LastByteAcked \leq \min\{cwnd, rwnd\}$$

Generally, `rwnd` is far larger than `cwnd`. We can make this assumption unless otherwise mentioned from here on. Because of this, the sender's send rate is roughly $cwnd/RTT$.

When a TCP connection begins, the value of `cwnd` is typically initialized to a small value of 1 MSS. This is known as **slow start**. Every time a transmitted segment is acknowledged, `cwnd` increases by 1 MSS. This in fact leads to an exponential growth rate - for instance, if `cwnd` is 2, then it will increase to 4 when receiving ACK for the 2 transmitted segments, and then double again, and so on.

When will our slow start end? If there is a loss event (indicated by timeout), the TCP sender will set the value to 1 again. It also sets the value of a second state variable, `ssthresh` or slow start threshold, to `cwnd/2`. Of course, it would not be wise to exceed this `ssthresh`, since we know congestion happened last time we did. So, if `cwnd` is equal to `ssthresh`, we leave the slow start state and enter **congestion avoidance**. If we ever experience fast retransmit, we will enter the **fast recovery state**.

When we start congestion avoidance, TCP is far more conservative. The TCP sender will increase the value by a single MSS every RTT. If we experience time out, then the TCP sender sets the value of `cwnd` to 1, `ssthresh` to `cwnd/2` and goes back to slow start.

Just like in slow start, if we experience fast retransmit, we begin fast recovery. However in this case, we only half the value of `cwnd` instead of setting it back to 1, and add 3 MSS to account for the triple duplicate ACKs. `ssthresh` is set to half the value of the old `cwnd`.

In fast recovery, the value of `cwnd` is increased by 1 MSS for every duplicate ACK received for the missing segment that caused TCP to enter the fast recovery state. Eventually, when an ACK arrives for that missing segment, TCP enters the congestion avoidance state after deflating `cwnd` (by how much?). If there is a timeout event, we go back to the slow start state by setting `cwnd` to 1 MSS and `ssthresh` to half the value of the old `cwnd`.

Fast recovery is not actually necessary according to the standard. **TCP Tahoe** has no fast recovery state, while **TCP Reno** has fast recovery.

4.9. AIMD and TCP. If you ignore the initial slow start period and assume that losses are indicated by triple duplicate ACKs rather than timeouts, TCP's congestion control consists of linear increase in `cwnd` and then a halving of `cwnd` on a triple duplicate ACK event. For this reason, we sometimes refer to TCP congestion control as **AIMD** (additive increase, multiplicative decrease).

AIMD congestion control is characterized by the "sawtooth" behaviour of the bandwidth graph. Given this sawtooth behaviour, what is the average throughput of a TCP connection over a long period of time?

Let us assume that the RTT and the window size when a loss event occurs (denoted by W) both remain approximately constant over time. Then, the TCP transmission rate will be in a range from $W/2RTT$ (when loss occurs) to W/RTT (right before loss occurs). In between these values, the transmission rate will increase linearly. Hence, the average throughput will be:

$$\frac{0.75 \cdot W}{RTT}$$