Telecoms - Transport Layer

=> Logical communication between app processes.

=> Run in end systems

* Send side: breaks message into segments, passed to network layer
* Rcv side: reassembles segments into message, passes to app layer

**Multiplexing/Demultiplexing**

Multiplexing @ sender

=> Handles data from multiple sockets

=> Adds transport header (used for demultiplexing)

Demultiplexing @ recv

=> Uses header to deliver rcvd segs into correct sockets

=> Host recv’s datagram:

* Contains header
  + TCP socket identified by src adr, dst adr, src port, dst port (4 tuple)
  + UDP socket identified by dst adr, dst port3
* Each segment has its own dst, src port to direct segment to correct socket to support multiple sockets
* Webservers different socket for each client

UDP

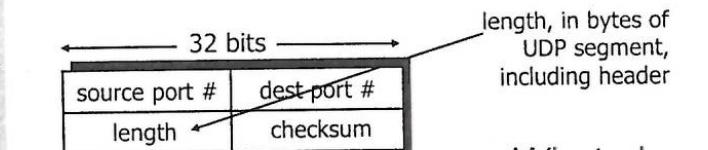
=> Barebones: simple transport protocol, connectionless, small header

=> Best efforts: segs may be lost, delivered out of order, no congestion control

=> Connectionless: no handshaking between sndr/recv, segs handled independently

=> Used: for streaming multimedia (loss tolerant/rate sensitive), DNS

=> Header:



**RDT (Reliable data transfer)**

=> We want to give the illusion of a reliable channel

=> Characteristics of the realistic unreliable channel will determine complexity of protocol

=> Separate FSMs for sndr/recv, although both sides may send or recieve

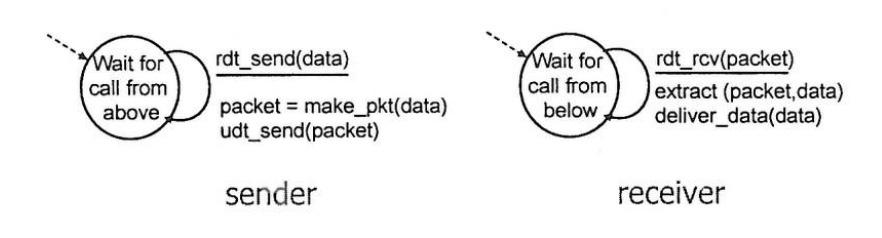
RDT 1.0 (Reliable channel)

=> No bit errors / no packet loss

=> Circle = current state | Event

Dashed arrow = starting state | ----------

Arrow = next state due to event | Actions



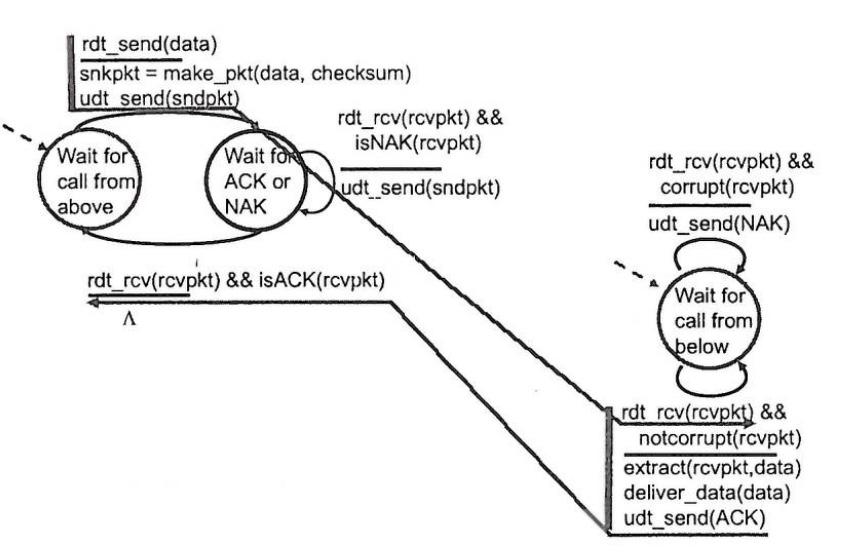
RDT 2.0 (Channel has bit errors)

=> Includes error detection & feedback

=> Use checksum to detect errors

=> ACKs/NAKs used for error recover

=> Cannot deal with corrupted ACK/NAK



RDT 2.1 (Handles corrupted ACKs/NAKs)

Sender:

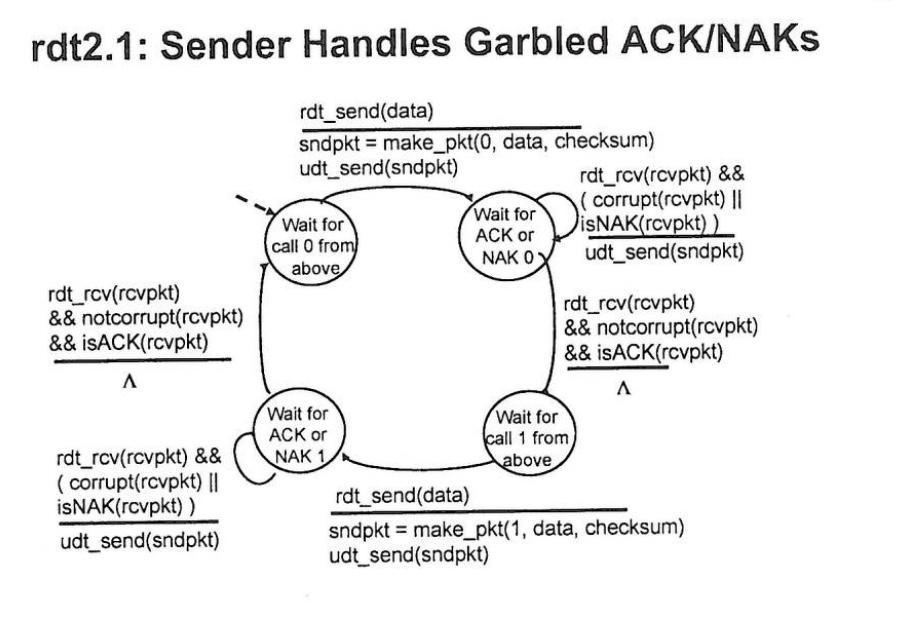
=> Seq number added to pkt (1/0)

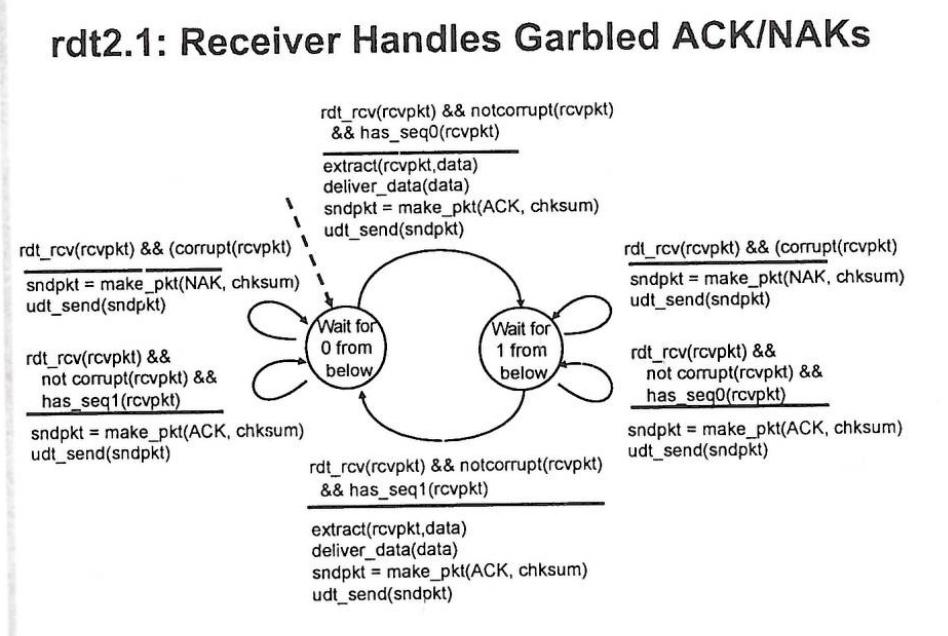
=> Checks if ACK/NAK corrupted

Reciever:

=> Checks if pkt is duplicate

=> Cannot tell if ACKs/NAKs received by sender



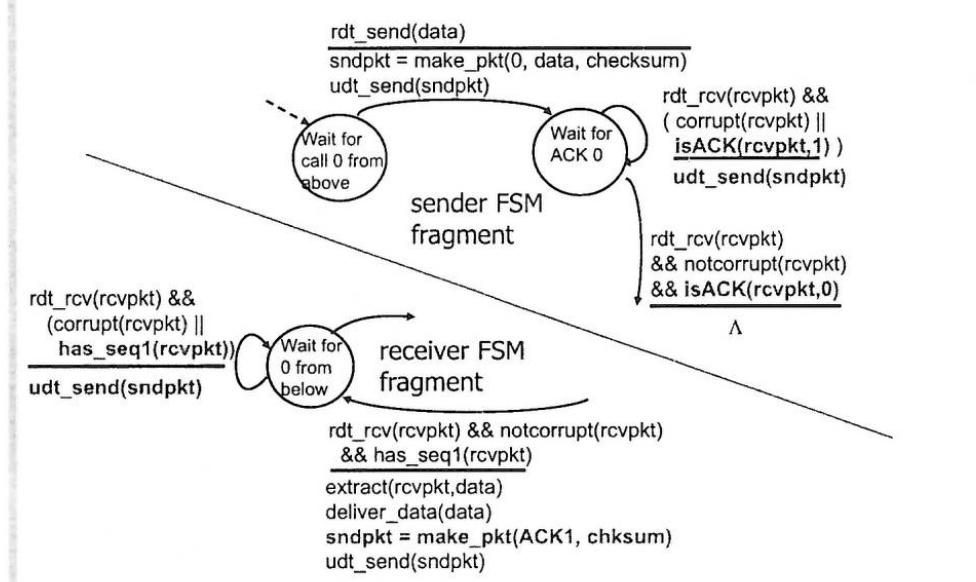


RDT 2.2 (NAK free)

=> Same functionality as 2.1

=> Instead of NAK, receiver sends ACK for last OK pkt received

Bhy => Must include seq numbers of pkts being ACKd

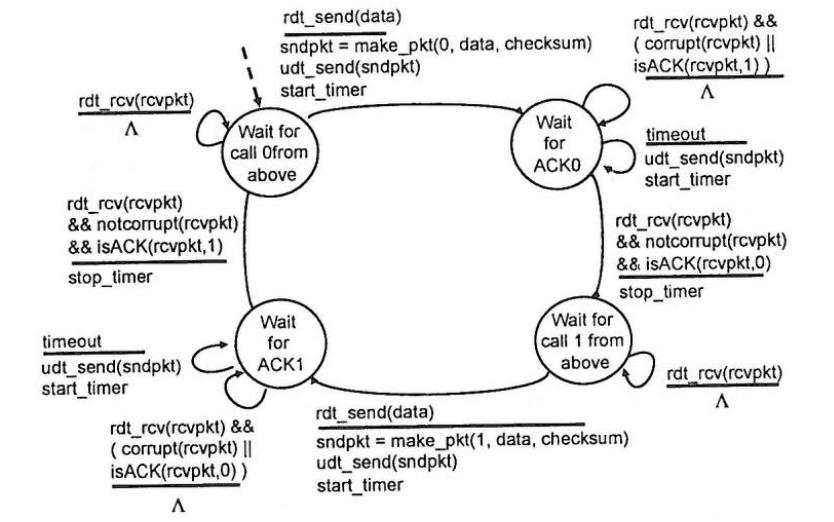
=> Duplicate ACKs at sender results in same action as NAK (retransmit current pkt)

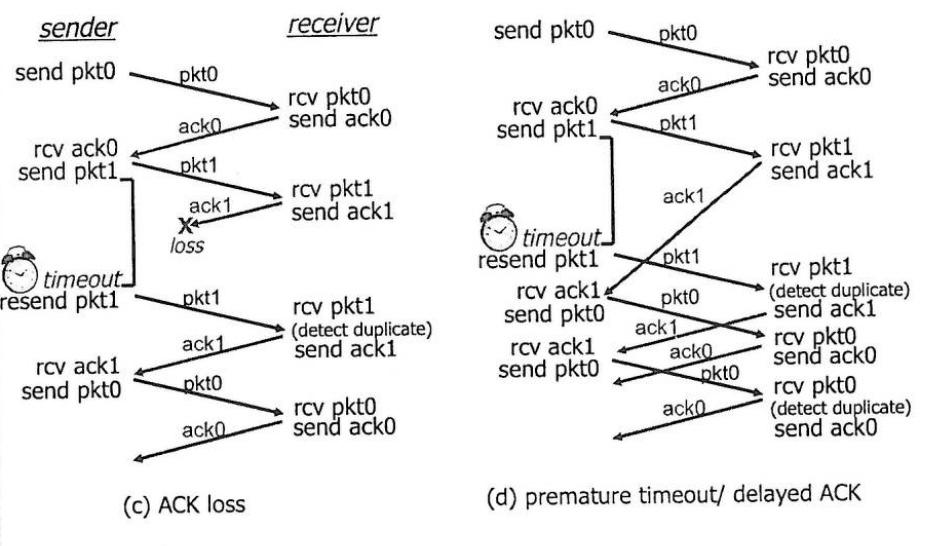
RDT 3.0 (Channel with errors and losses)

=> Channel can lose packets (data + ACKs)

=> Approach:

* Sender waits a set time (countdown timer required)
* Retransmits if no ACK recv in time
* If pkt/ACK just delayed (not lost):
  + Retransmission will be duplicate, but seq numbers handle this
  + Recv must specify seq number of the pkt being ACKd





**Pipelined protocols**

=> Allows for multiple “in-flight” pkts, so the RTT won’t affect the throughput as much,

As we don’t have to wait on ACKs (increased bandwidth utilisation)

=> Range of seq numbers must be increased rather than 0/1

=> Allows for buffering at sender/receiver

Go-back-N

=> Sender can have N unACKd pkts in pipeline

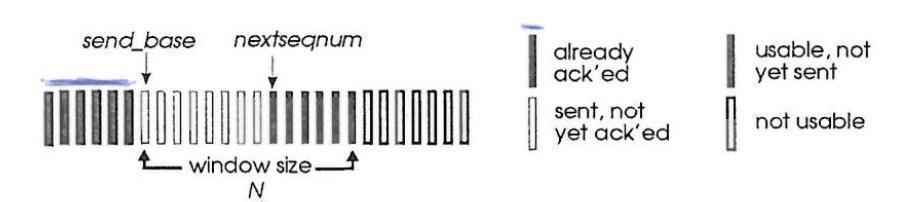
=> Recv can send cumulative ACKs (only sending the ACK for last pkt recv)

=> Sender has timer for oldest unACKd pkt

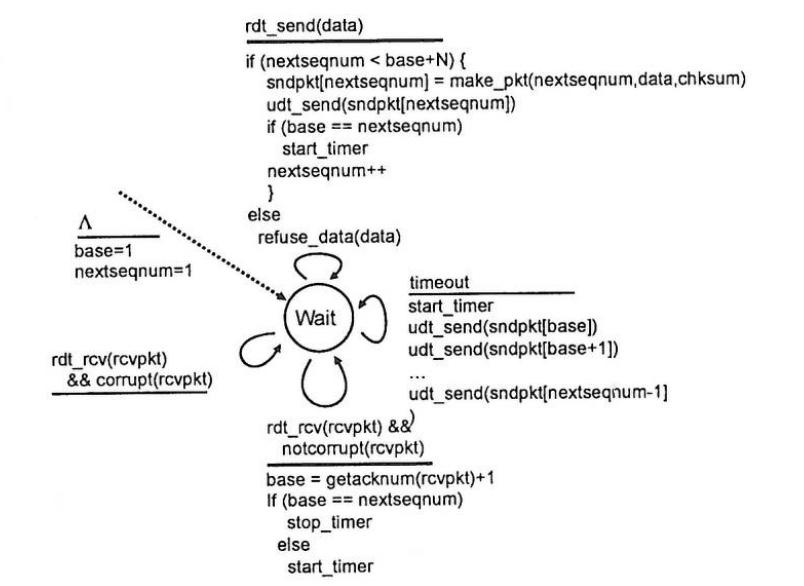
=> When timer expires, all unACKd pkts are retransmitted (retransmit current + all higher seq number pkts in window)

=> Seq number in pkt header

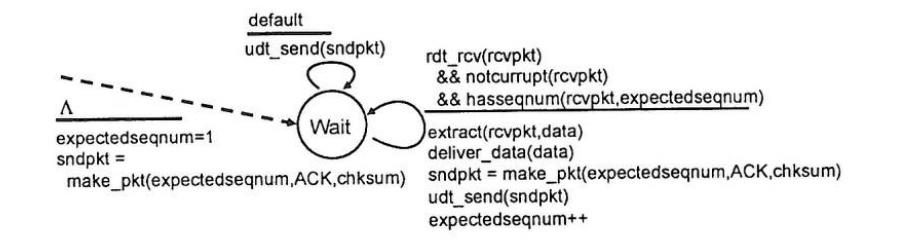
=> Out of order pkts discarded, reACK packet with highest seq number

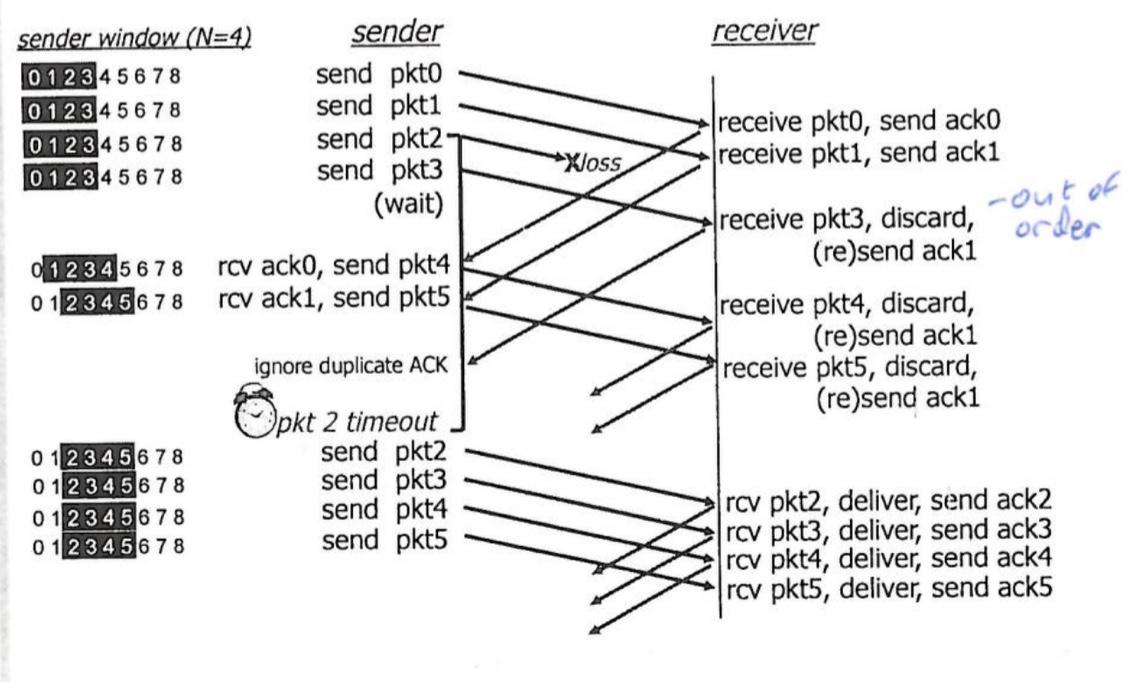


Sender FSM



Recv FSM





Selective Repeat

=> Recv ACKs all individual OK packets

=> Packets are buffered to achieve in-order delivery for the upper layer

=> Sender only retransmits pkts for which ACK not recv’d

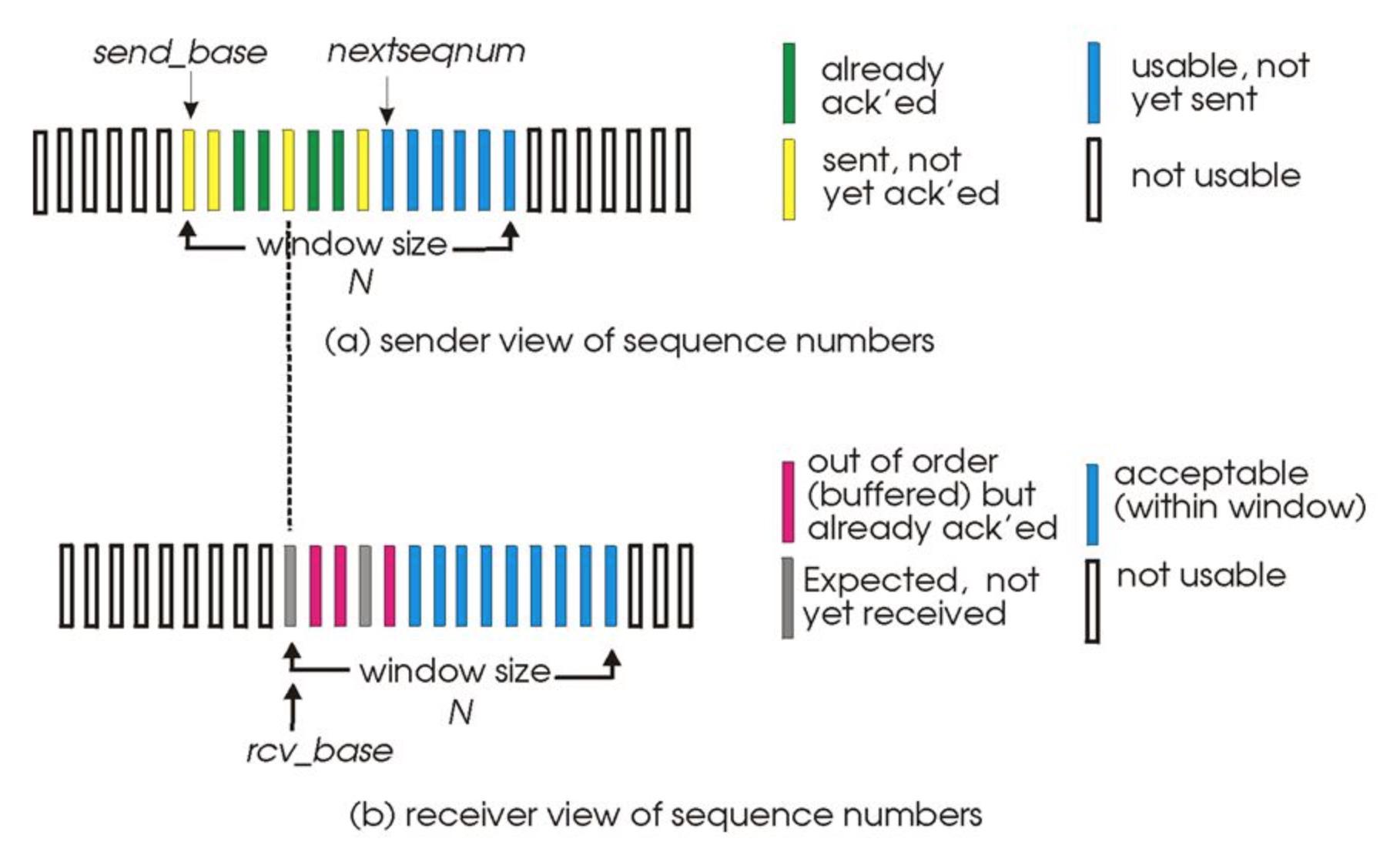
=> Need a timer for each packet to do this

=> Sender:

* If data sent from above ->
  + if next available seq# in window, send pkt
* Timeout(seq# n) ->
  + resend pkt n, restart timer
* Recv ack(seq# n) where seq# n is in window ->
  + mark pkt n as recv’d
  + if seq# n is the lowest seq# in window, advance window by 1

=> Recv:

* Pkt seq# n in recv window ->
  + Send ack(n)
  + If pkt out of order then buffer
  + If pkt in order, deliver (and deliver buffered out of order pkts)
    - Advance window to next not yet recv’d pkt
* Pkt seq# before recv window ->
  + ack(n) previously ack’d pkt
* Otherwise -> Ignore



**TCP**

=> Point to point (1 sender, 1 recv)

=> Reliable, in order byte stream (no message boundaries)

=> Pipelined (TCP congestion control/flow control set window size)

=> Full duplex data

* Bi directional data flow in same connection
* MSS (max seg size) e.g. 1500 bytes for ethernet
* MTU (max transmission unit) - 40 bytes (TCP/IP header)

=> Connection oriented

* Handshaking (exchange of control msgs)
* Initialising sender/receiver state before data exchange

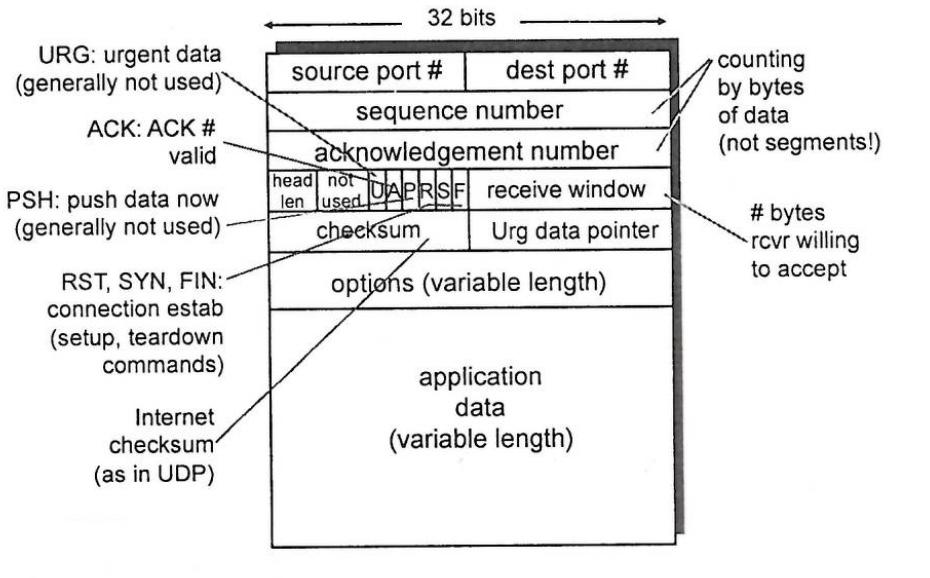
=> Flow controlled

* Sender will not overwhelm receiver

TCP Header

=> Seq# = number of first byte in segment

=> ACK# = seq# of next byte expected from other side



TCP Round Trip Time

=> Timeout value

* Should be longer than RTT, but RTT varies
* Too short -> unnecessary retransmissions
* Too long -> slow reaction to segment loss

=> Estimating RTT

* Use sampleRTT (the time it takes from seg transmission to ACK receipt)
* SampleRTT varies due to load and congestion

=> Use EstimatedRTT

* (1 - ⍺) \* EstimatedRTT + ⍺ \* SampleRTT
* ⍺ usually 0.125 (between 0-1)
  + If too small -> sudden change in network not reflected fast enough
  + If too large -> transient fluctuations affect EstimatedRTT makes it unstable
  + Both lead to under/over EstimatedRTT
* Exponential weighted moving average

=> DevRTT

* TimeoutInterval = EstimatedRTT + 4\*DevRTT (safety margin)
* DevRTT = (1-β \* DevRTT + β\*abs(SampleRTT - EstimatedRTT)
* Β usually 0.25

=> TimeoutInterval starts at 1, doubled when timeout occurs

TCP reliable data transfer

=> Pipelined segments

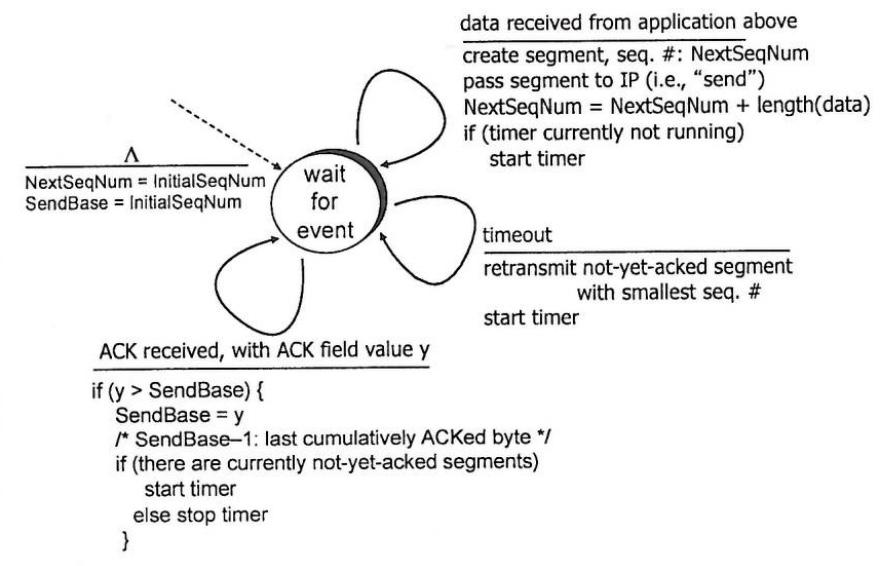
=> Cumulative ACKs

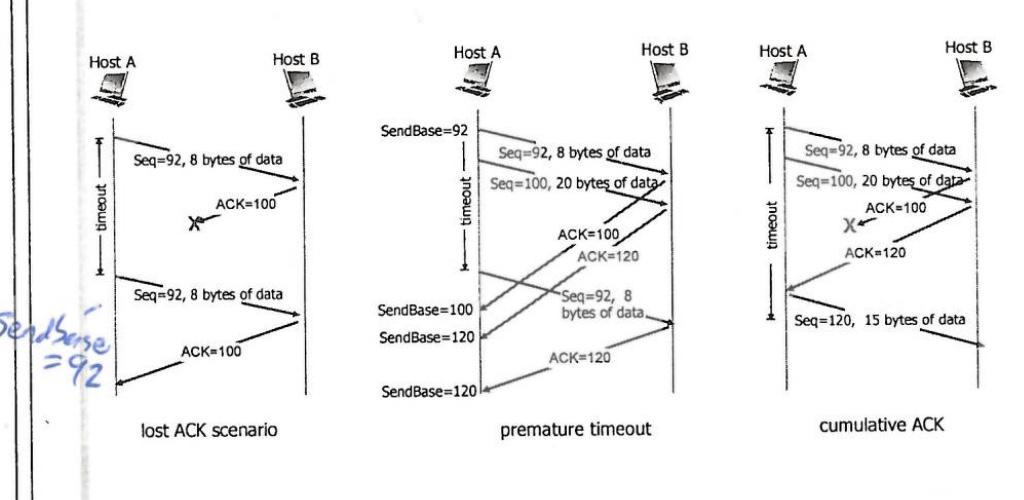
=> Single retransmission timer

=> TCP Sender

* Data recv’d from app above, create seg with seq# of 1st data byte in segment
* Start timer if not already running (timer for oldest unACKd pkt (expires after TimeoutInterval))
* If timeout occurs, retransmit seg that caused timeout + restart timer
* ACK recv’d
  + If ACK acknowledges previously unACKd segments, updated what is known to be ACKd and start timer if there are still unACKd segments

Sender:





TCP ACK generation (at receiver)

Event -> Action

=> arrival of in-order segs, with expected seq#, all data up to expected seq# ACKd ->

Send delayed ACK 500ms for next seg, if no next seg send ACK

=> arrival of in-order segs, with expected seq#, one other seg has ACK pending ->

Immediately send cumulative ACK, ACKing both in-order segs

=> arrival of out-of-order seg, with higher than expected seq#, gap detected ->

Immediately send duplicate ACK, with seq# of next expected byte

=> arrival of segment that fills a gap (partially or fully) ->

Immediately send ACK, provided seg starts at lower end of gap

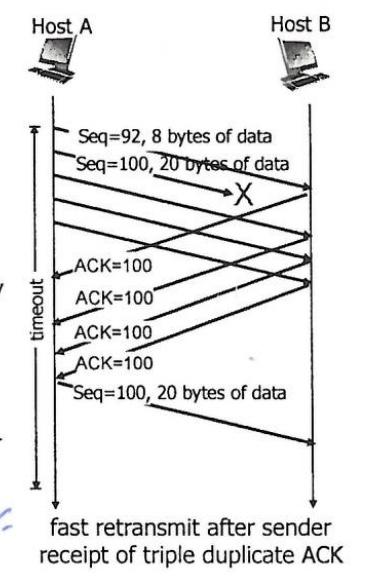
TCP Fast Retransmit

=> Long timeout period

=> Lost segs detected by duplicate ACKs

=> If 3 duplicate ACKs, resend seg with smallest seq#

=> Likely that unACKd seg was lost, so don't wait for timeout



TCP Flow Control

=> Receiver controls sender, so sender won't overflow recv buffer by transmitting too much too fast, prevents buffer from overflowing

=> Recv advertises free buffer space

=> By using rwnd (recv window) value included in header of recv-to-sender segs

=> RcvBuffer size set via socket options, auto-adjusted by OS

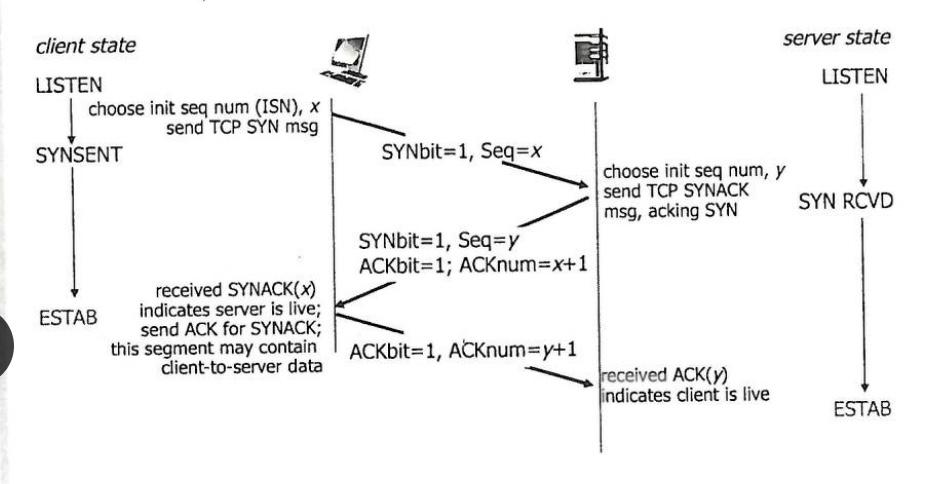
Rwnd = RcvBuf - [ lastByteRcvd - lastByteRead ]

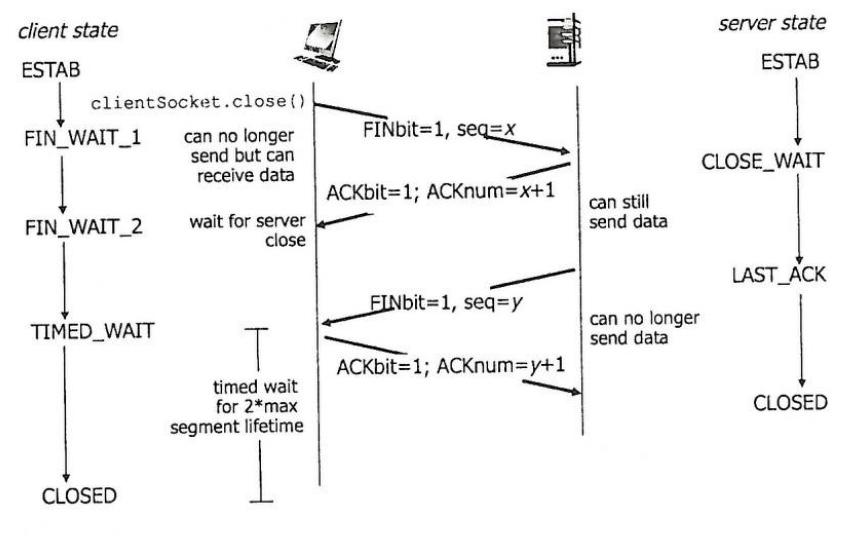
TCP Connections

=> Send/recv handshake, establishes connection

=> Agree on connection params

=> 3 way handshake

Establishing Connection

Closing Connection 

SYN flood could (DOS)

* Large number of TCP SYN segs sent to server without completing step 3 of handshake
* Use SYN cookies to ensure legitimate user
* Server creates initial sequence number (ISN) from hash of Src adr/port, Dst adr/port, timestamp
* Server sends SYNACK
* Legit user returns and ACK, using cookie info (ISN+1) in ACK

**Congestion Control**

=> Too many sources sending too much data too fast for network to handle

=> Causes lost packets(buffer overflow), long delays(queuing in buffers)

=> 2 Approaches

* End-to-end(TCP)
  + No explicit feedback
  + Congestion inferred from end-system observed loss&delay
* Network Assisted
  + Routers provide feedback to end systems
  + Single bit indicating congestion (ATM)

ATM ABR

ABR available bit rate

=> Senders path underloaded -> sender uses more bandwidth

=>Senders path Congested -> sender throttled to minimum guaranteed rate

=>RM Cells (resource management)

* Sent by sender, between data cells
* Bits in rm cells set by switches (network assisted) -> NI/CI bits -> mild congestion/congestion
* Returned to sender by receiver

AIMD (additive increase, multiplicative decrease)

=> Sender increases transmission rate (window size)

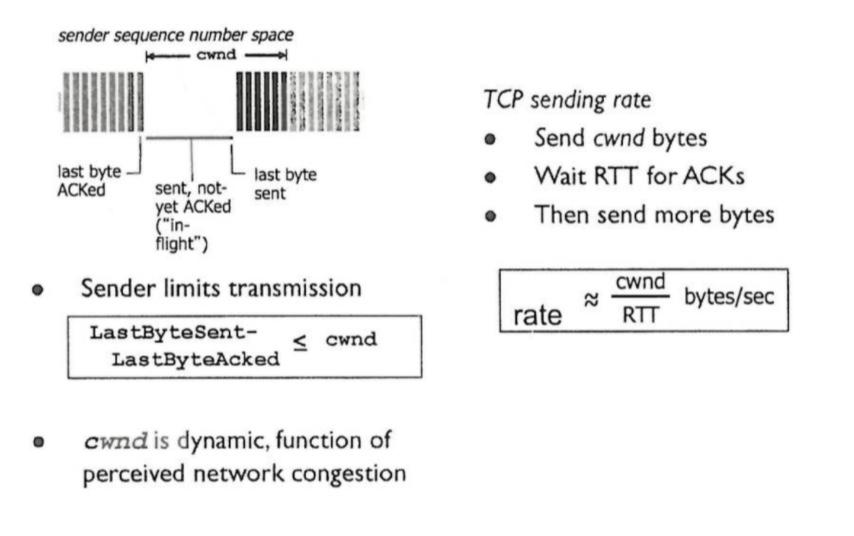
=> Probing for usable bandwidth until loss occurs

=> Increase congestion window (cwnd) by 1 mss every rtt until loss detected

=> Cut cwnd in half after loss

=> Saw tooth behaviour

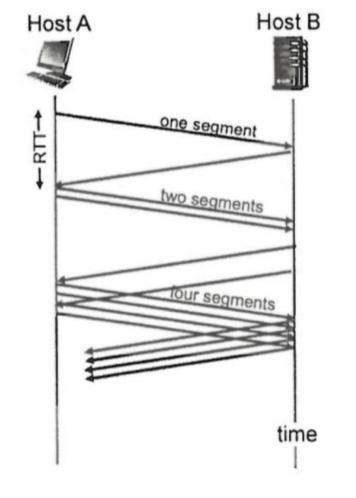
TCP Congestion Control



Slow Start

=> Rate increases exponentially until 1st loss occurs (starts at cwnd=1mss)

=> Double cwnd every RTT (incrementing cwnd for every ACK received)



=> Loss indicated by timeout, cwnd set to 1 mss

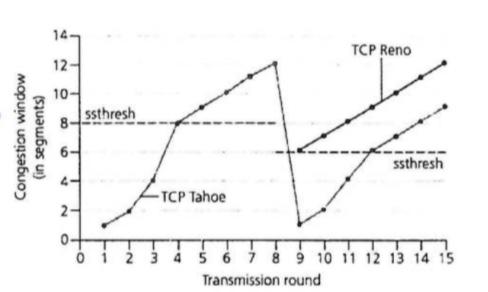
=> Window grows exponentially to ssthresh, then grows linearly

TCP Reno

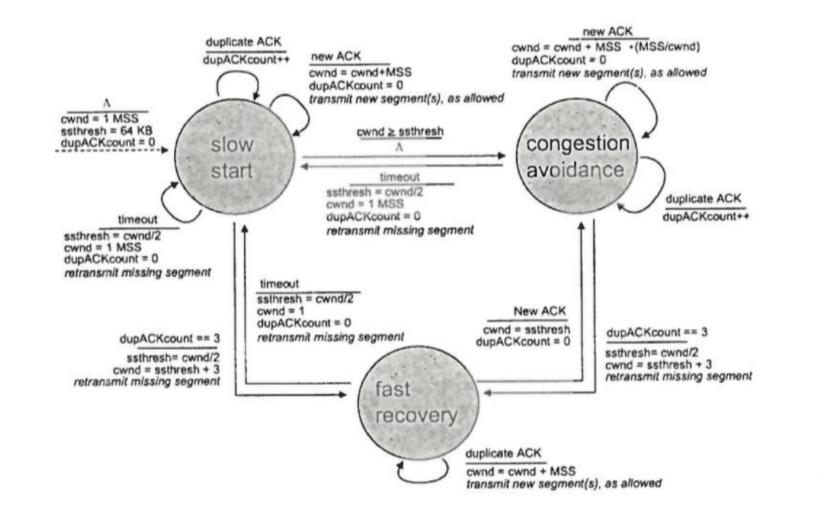
* 3 duplicate ACKs indicates loss
* Cwnd cut in half, then grows linearly

TCP Tahoe

* Always sets cwnd to 1
* Timeout or 3 duplicate ACKs



=> Exponential switches to linear when cwnd set to ½ of its value before timeout

=> Variable ssthresh

Telecoms - Application Layer

**Client-Server Architecture**

=> Server

* Always on host
* Permanent IP adr

=> Client

* Communicates with server
* Intermittent connection
* May have dynamic IP adr
* Do not communicate directly with each other

**P2P peer to peer architecture**

=> No always on server

=> End systems directly communicate

=> Peers request service from other peers, provide service in return to other peers

=> Self scalability -> new peers bring new service capacity, as well as demands

=> Peers are intermittently connected, change IP adrs, complex management

**Processes**

=> Client process -> inits communication

=> Server process -> waits to be contacted

=> Communication by exchanging messages

=> p2p architecture applications contain both client&server processes

**Application’s Transport Services Needs**

=> Data loss

* File transfer / transactions 100% reliable data transfer
* Audio / video tolerates loss

=> Timing -> telephony / games are time sensitive

=> Throughput -> multimedia needs minimum amount of throughput

=> Security -> authentication / encryption

**WWW**

=> URL (uniform resource locator) allows access to each object on webpage

[hostname]/[pathname]

**HTTP**

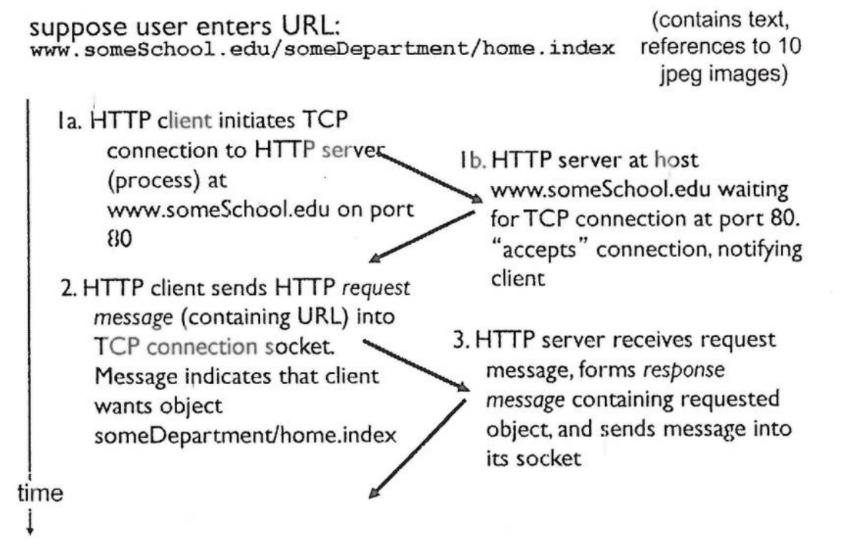
=> Uses TCP

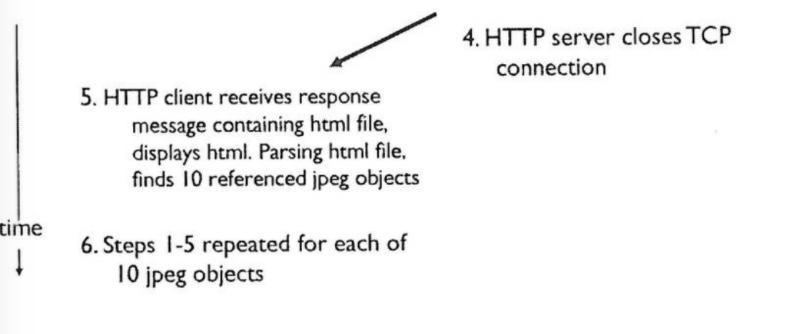
* Client inits TCP connection (creates socket) to server @ port 80
* Server accepts connection
* HTTP msgs exchanged
* Connection closed

=> HTTP is stateless as TCP retains information.

=> Non-persistent HTTP

- 1 object sent over TCP connection, then connection closed





* 1 RTT to init TCP connection
* 1 RTT for HTTP request + first few bytes of HTTP response to return
* File transmission time
* Response time = 2\*RTT + file transmission time
* Overhead for each TCP connection
* Browsers can open parallel TCP connections

=> Persistent HTTP (default with pipelining)

- Multiple objects send over single TCP connection

- Server leaves connection open after sending response

- Subsequent client/server messages sent over open connection

**HTTP Request Message**

=> Either request or response

=> PUT

* Uploads file to path in url field

=> DELETE

* Deletes file at path in url field

=> POST

* Input uploaded to server

=> GET

* Input is uploaded in URL field

**HTTP Response Codes**

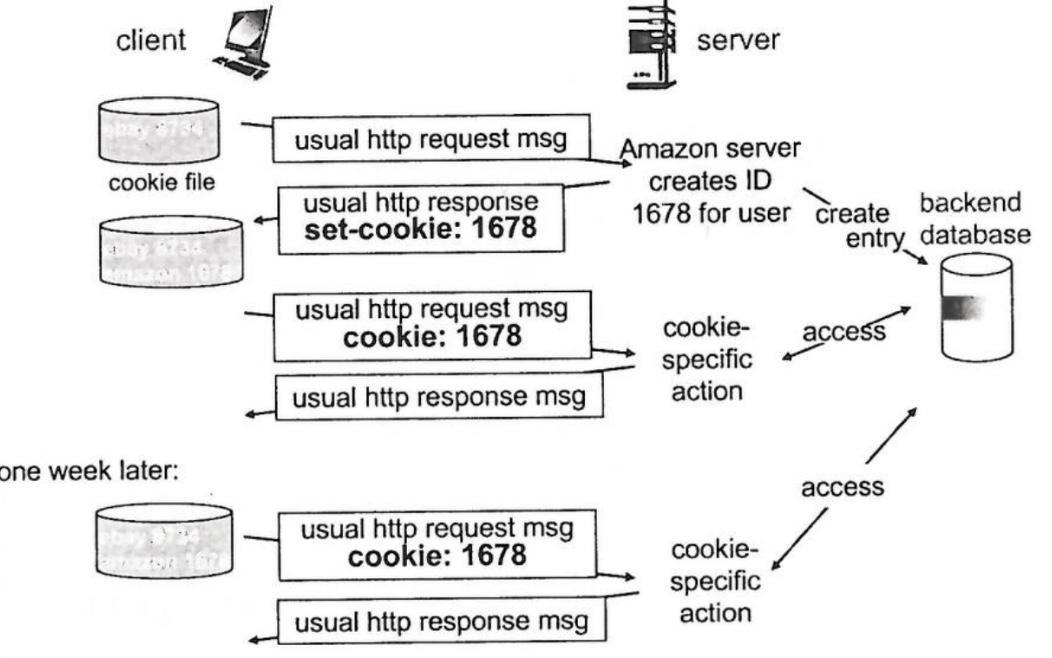
=> Occur on 1st line of response message

=> 200 OK, 301 Moved Permanently, 400 Bad Request, 404 Not Found, 505 Version Not Supported

**Cookies**

=> Used for authorization, shopping carts, user session state

=> As HTTP is stateless, cookies may be used to maintain state at sender/receiver



**Web Caches (Proxy server)**

=> Want to satisfy request without involving origin server

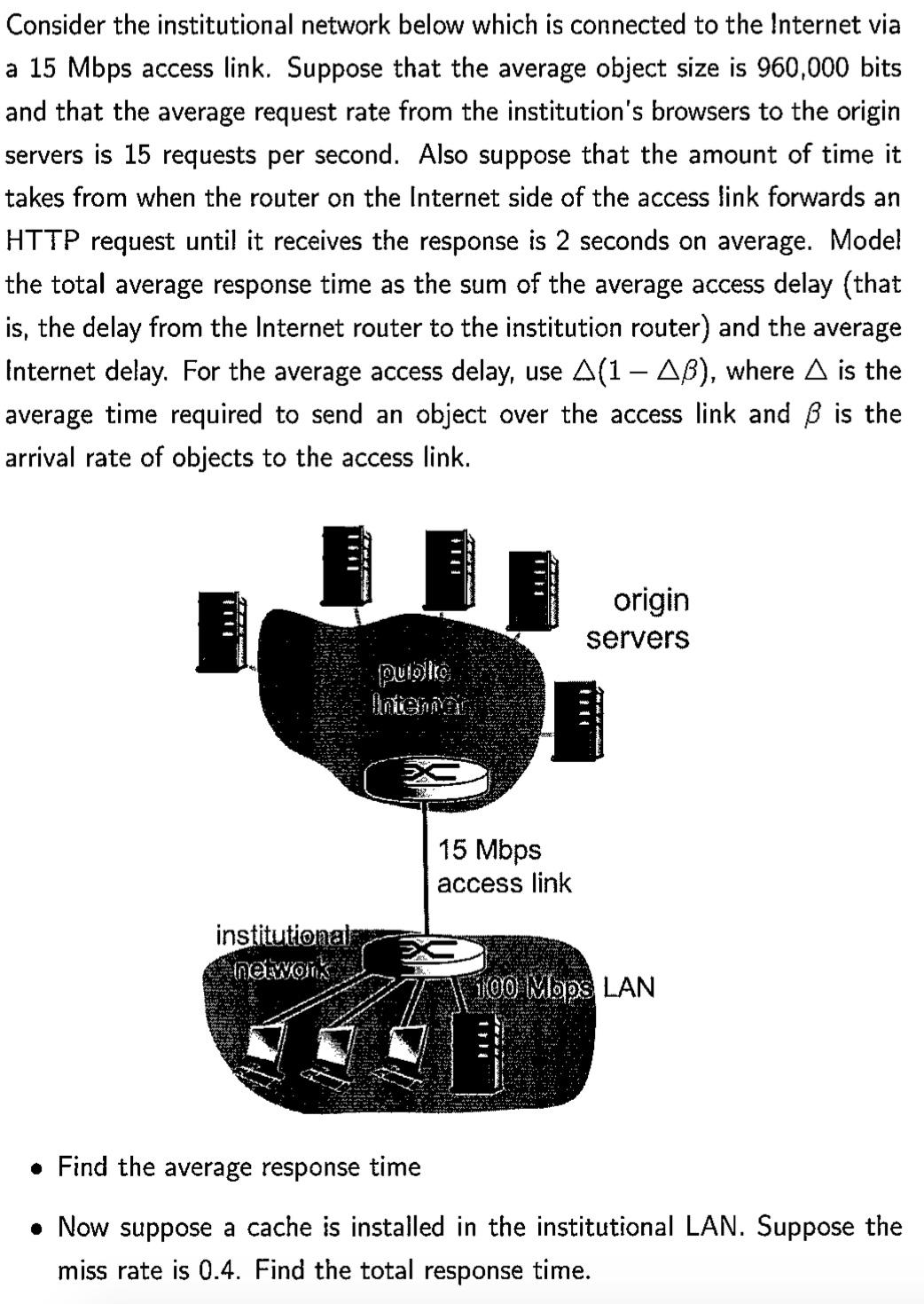
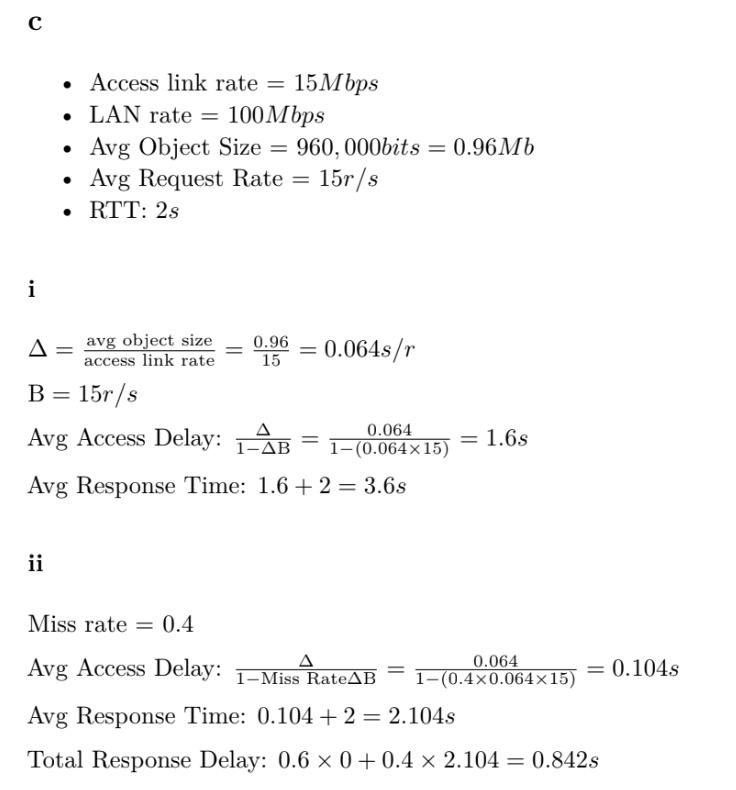
=> Browser sends HTTP requests to proxy

=> If object in cache -> proxy returns object

=> Otherwise -> proxy requests object from origin server, returns obj to client

=> Cache acts as client&server

=> Caches reduce traffic on ISP, reduces response time for client request



=> Conditional GET used (if-modified-since:<date> in HTTP request), if cache has up-to-date version, send it with “304 Not Modified” response

=> If the cache’s version is out of date, send response from origin server

**Email**

=> User Agent

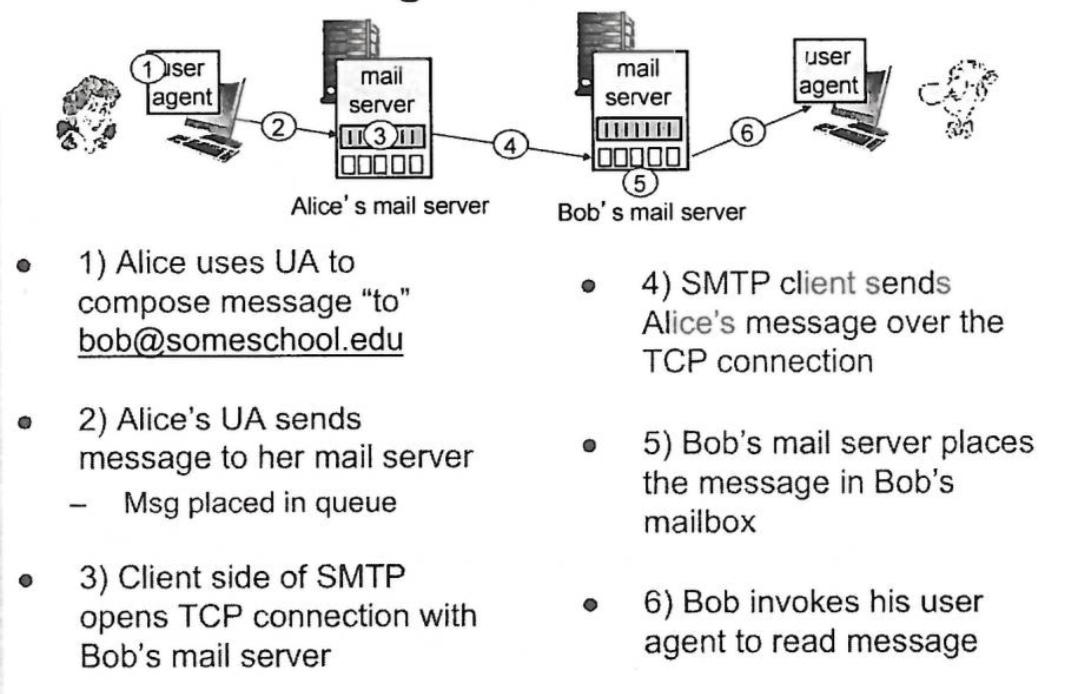
* Compose, edit, read messages

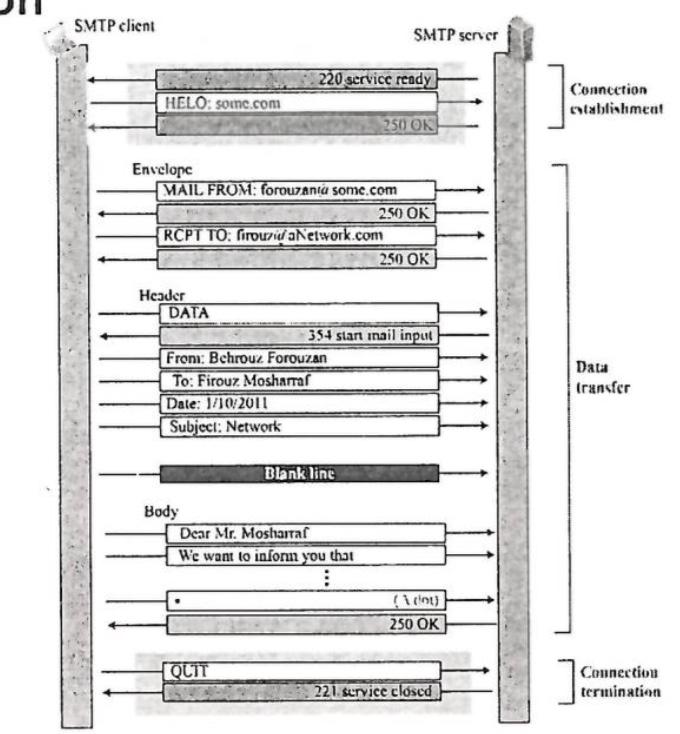
=> Mail Server

* Contains incoming messages, queue of outgoing messages
* SMTP protocol used between mail servers

=> SMTP

* Uses TCP, port 25
* Command/response interaction
* Uses persistent connections
* All messages (header&body) to be sent in 7 bit ASCII
* Both HTTP and STMP have ASCII command/response interaction, status codes





**Multipurpose Internet Mail Extension (MIME)**

=> Used to convert non ASCII data to ASCII to be sent via SMTP

=> Different MIME types for different data types (e.g. text, audio, image)

**POP3 vs IMAP**

POP3

=> Download and delete mode, cannot re-read email if client is changed

=> Download and keep, copies of messages on different clients

=> Stateless across sessions

IMAP

=> Keeps all messages at server

=> Allows user to organise messages in folders

=> Keeps user state across sessions

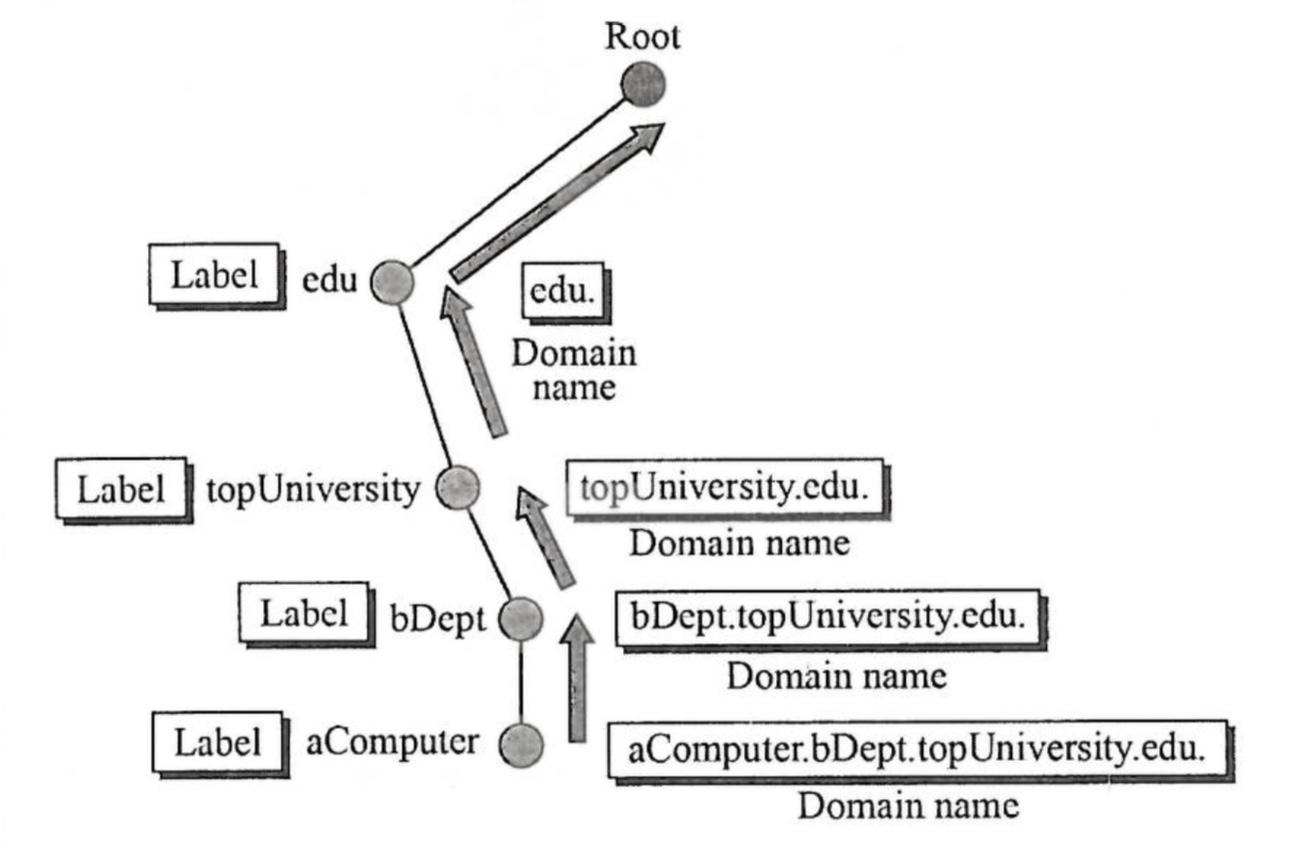
**Domain Name System (DNS)**

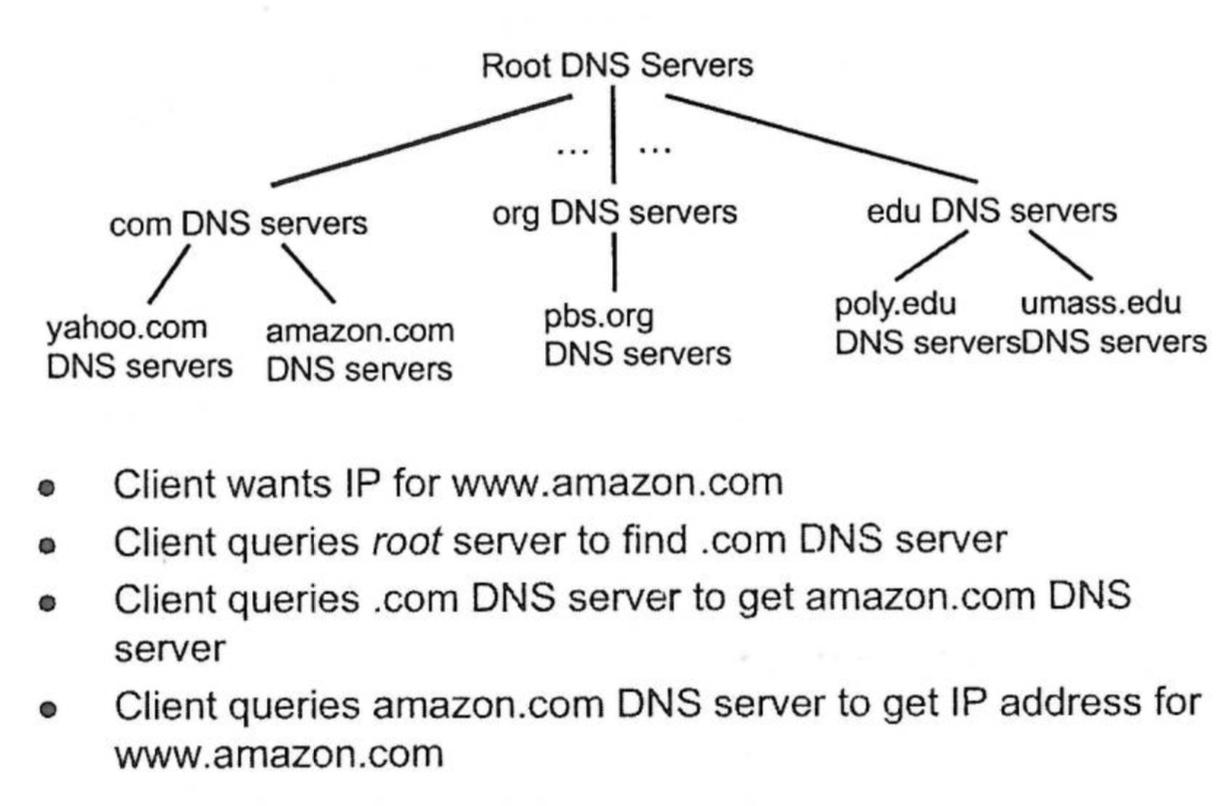
=> Distributed db implemented in hierarchy of many name servers

=> Hosts, name servers communicate to resolve names (adr/name translation)

=> Services:

* Hostname to IP adr translation
* Host aliasing
* Canonical, alias names
* Mail server aliasing
* Load distribution
* Replicated web servers
* Many IP adrs correspond to 1 name





=> Root servers contacted by local name server that cannot resolve name

=> Returns list of IP adrs for responsible TLD servers

=> Top level domain (TLD) servers, responsible for .com, .org etc

**Local DNS**

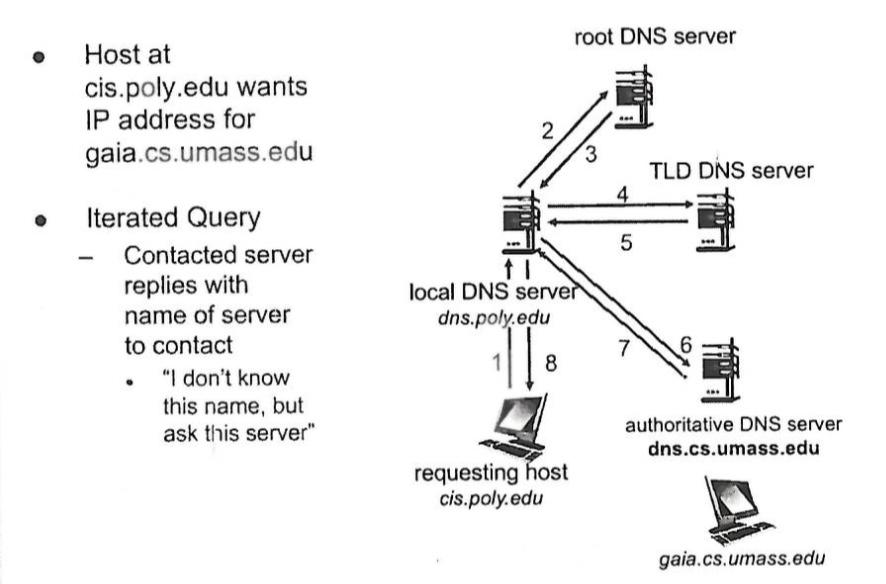
=> Does not strictly belong in the hierarchy

=> Each ISP has one

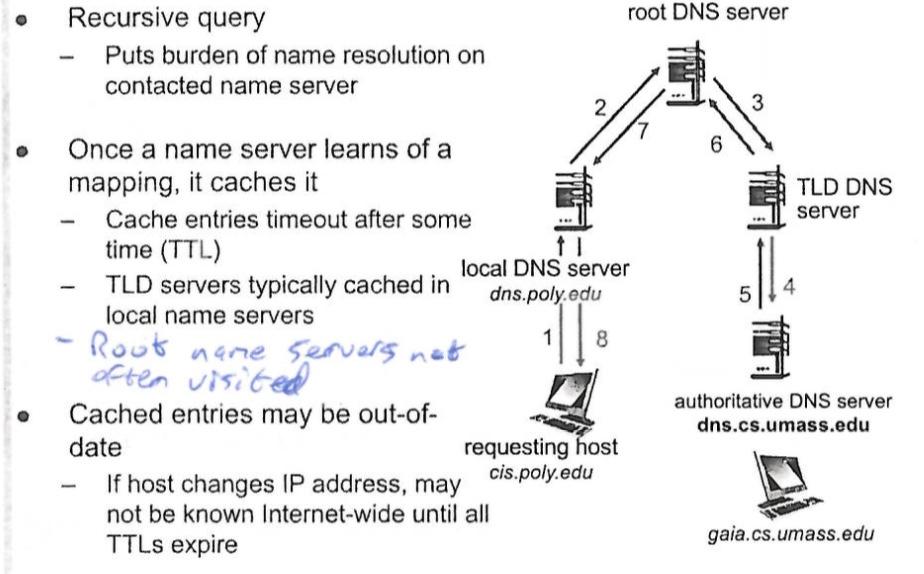
=> When host names DNS query, its sent to local DNS server

* Has cache of recent name to adr translation pairs (may be out of date)
* Acts as proxy, forwards query into hierarchy

**DNS Name Resolution**



**DNS Name Resolution & Caching**

****

**DNS Records**

=> Distributed db storing resource records(RR)

=> A: IPv4, AAAA: IPv6 (128 bit)

* NS: name server
  + Name is domain, value is hostname
* CNAME: canonical name (alias)
  + Name is alias for real name, value is real name
* MX: mail exchange
  + Value is name of mail server

**DNS Inserting records**

=> Register name at DNS registrar

=> Registrar creates 2 RRs into .com TLD server

* example.com, NS, dns1.example.com
* Dns1.example.com, A, 212.212.212.1

=> Create

* Authoritative server type A record for example.com
* MX type record for example.com

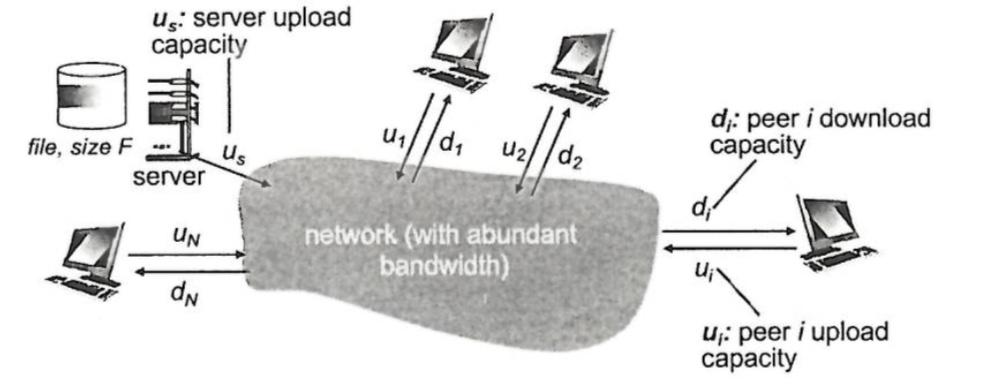
**DNS Attacks**

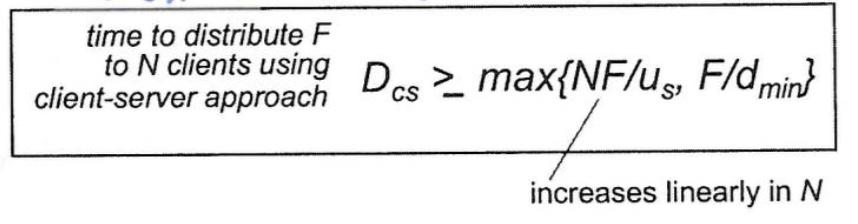
=> DDos -> bombard root servers with traffic, avoided as local DNS servers cache IPs, bypassing root

=> Redirect attacks (man in the middle - intercepting queries, DNS poisoning)

**P2P vs Client Server - File Distribution**

=> How long does it take to distribute file (size F) from one server to N peers?



Client-server: 

=> Time to upload N copies = N\*(F / us)

=> Time to download on a client = F/dmin

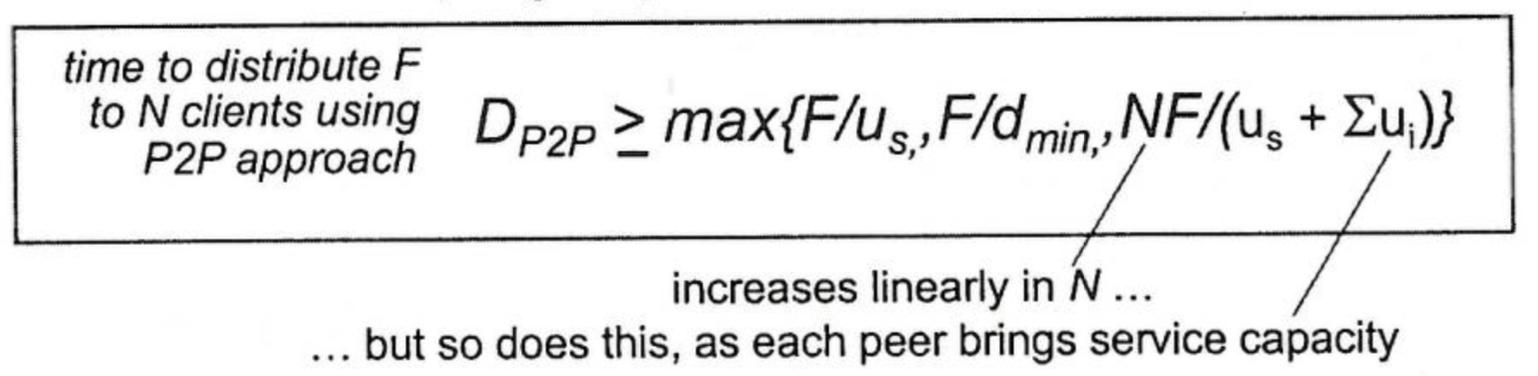
Where dmin  is the min client d/l rate

P2P

=> Server must upload at least one copy

=> Clients download file copy

=> Max upload rate = us + sum(ui)



**BitTorrent**

=> A torrent is a group of peers exchanging chunks of a file

=> File divided into 256kb chunks

=> Peers can send/recv chunks

=> Trackers track peers participating in torrent

=> Peer joins torrent

* No chunks yet, accumulates them over time from other peers
* Registers with tracker to get list of peers, connects to some peers (neighbours)
* Peers upload chunks to other peers
* Peers may change
* Once peer has entire file, may leave or remain in torrent

=> Peers will have different subsets of file chunks

=> Periodically, peers will ask each other for chunks that they have

=> The missing chunks are requested from other peers

=> Tit-for-tat, each peer has top 4 peers currently sending its chunks at highest rate

=> Top 4 re-evaluated every 10 secs

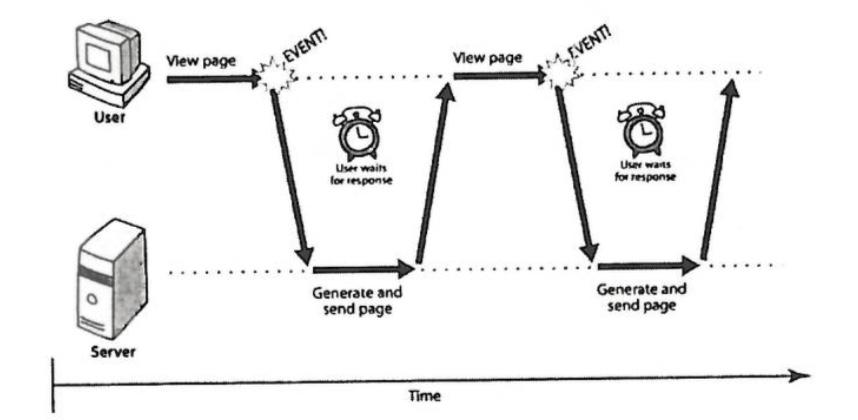
=> Other peers may be choked, so every 30 secs select a random peer & send chunks

**Synchronous vs Asynchronous Web Communication**

Synchronous (most common)

=> User must wait while new pages load

=> While request is being processed on the server, user cannot interact with the client



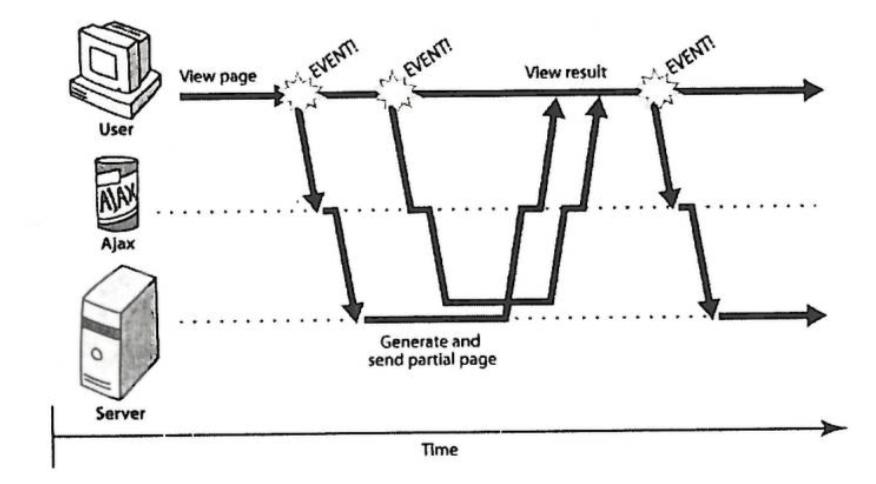
Asynchronous

=> Used in rich internet applications (RIAs)

=> Dynamic websites

=> Ajax (asynchronous XML and javascript)

* Download data from server in background
* Webpage doesn’t need to be refreshed
* User may interact with webpage while data loads



Typical Ajax Request

=> User clicks, invoking an event handler

=> Handler creates and XMLHttpRequest object

=> XHR object requests page from server

=> Server retrieves and sends data

=> XHR object fires an event when the object arrives (callback)

=> Callback event handler processes the data and displays it

=> Resources must be on the same domain (same origin policy)

**Websockets**

=> Two-way socket connection between web browser and server

=> Ajax has to poll server for data, HTTP connection overhead creates bottleneck

=> Websockets allow client side js to open and persist a connection to a server

=> Full duplex capability allows messages to be pushed to client from server

=> Without the overhead associated with traditional HTTP requests

=> Means that both parties can start sending data at any time

=> Websocket handshake establishes connection

=> Data is transferred as messages

=> Messages are framed, with 2 byte overhead per frame, reduces amount of non-payload data that is transferred, reductions in latency