Chapter 1: Introduction

Internet Overview
- hosts = end systems (run network apps) - all 5 layers of protocol stack implemented at end host
- communication links (fiber, copper, radio)
- Routers & switches (packet-switching) (NOT application or transport layer - **IS** network layer)

Network structure
- network edge: hosts (clients/servers)
- access networks: wire/less communication links - bottleneck of internet speed
- network core: interconnected routers - routing and forwarding packets to destination
- connect hosts to network core via access networks

Transferring Data
- packet switching: hosts break app layer messages into packets, forward packets from router to router
- packets of length L bits, transmission rate (link bandwidth) R: **trans delay = L bits / R bits/sec**

Store and Forward: entire pckt must arrive at router before it can be transmitted on next link
- **end-end delay = 2L/R (assume 0 prop delay)**

Circuit Switching (Domain Multiplexing = DM)
- dedicate resources, no sharing, guaranteed perf, circuit idle if not used (no sharing), needs to be reserved in advance
- **Frequency DM**: frequency band divided into sub-bands;user can use the allocated sub-band only
- **Time DM**: divide time into time slots, user can use whole frequency band but only at allocated time slots
- # users: total bandwidth/bandwidth per user

Packet Switching: (better for bursty traffic)
Given N users, probability that x users are active is:

$$P(N,x)=\binom{N}{x}p^{x}(1-p)^{N-x}$$

- Probability that there are ≥ Y users: $\sum_{x=Y}^{N} P(N,x)$
- can use store and forward

pckt Delay
- d_nodal = d_proc + d_queue + d_trans + d_prop
- **d_proc:** nodal processing (at router) - check for bit errors, determine output link
- **d_queue:** queueing delay - time waiting at output link for transmission - depends on congestion level
- **d_trans:** transmission delay - L (pckt length)/ R (bandwidth)
- **d_prop:** propagation delay - d (length of physical link) / s (propagation speed - 2x10^8 m/sec)
- Average queueing delay: (N-1)L/2R

traceroute URL: provides delay measurement from source to router along end-end internet path towards destination

Packet loss
- buffer before link has finite capacity
- when packet arrives at full queue, it is dropped

Throughput: rate at which bits are transferred
- **bottleneck link:** link on end-end path that constrains end-end throughput
- time gap between rcv last bit of 1st pckt vs last bit of 2nd pckt = L (bytes)/min{bandwidth_s, bandwidth_c}
- time to get all = pckt1 delay + #pckts*bottleneck

Internet Protocol Stack (bottom to top)
- physical, link, network, transport, application
- may duplicate lower functionality, more overhead, useful as it achieves goal of divide and conquer

Chapter 2: Application Layer

- network apps run on end hosts

Application Architectures
- Client-Server: (server is always-on host, perm IP address) (client can turn off, comm with server, dynamic IP address, client don't intercommunicate)
- Peer-to-Peer: (no always on server, end systems directly communicate, peers request/provide service to each other, self-scalable (new peers = more service & demands), dynamic IP, peer can join/leave any time

Sockets - Client/Server communication
- process (program on host) send/get messages from socket-can comm. with many processes @ other hosts

application	data loss	throughput	time sensitive
file transfer	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	no loss	elastic	no
real-time audio/video	loss-tolerant	audio: 5kbps-1Mbps video:10kbps-5Mbps	yes, 100's msec
stored audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	few kbps up	yes, 100's msec
text messaging	no loss	elastic	yes and no

application	application layer protocol	underlying transport protocol
e-mail	SMTP [RFC 2821]	TCP
remote terminal access	Telnet [RFC 854]	TCP
Web	HTTP [RFC 2616]	TCP
file transfer	FTP [RFC 959]	TCP
streaming multimedia	HTTP (e.g., YouTube), RTP [RFC 1889]	TCP or UDP
Internet telephony	SIP, RTP, proprietary (e.g., Skype)	TCP or UDP

HTTP(Hypertext transfer protocol)(PORT 80(80))
- web pages consist of objects - base HTML file which includes referenced objects - addressable by URL
- **client** (browser) requests/displays web objects
- **server** (web) sends objects in response
- HTTP is stateless-no info about past requests
- PULL Model - client pulls info available to a server

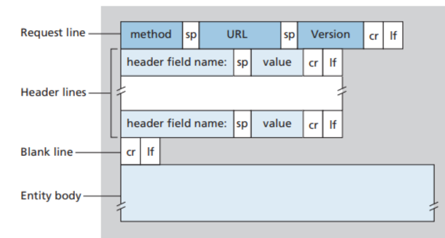
Non-persistent HTTP
- ≤ 1 object sent using TCP, then conn closed
- downloading multiple objects == multiple connections
- Steps: 1 RTT set up TCP conn, 1 RTT to get base HTML, for each referenced object, repeat prev steps
- 2RTT + file transmission time for every object

Parallel Non-persistent HTTP
- 2RTT for TCP setup + getting base html
- 2RTT * (#objects/number parallel conns)
- consumes much more server resources

Persistent HTTP
- multiple objects sent over single TCP connection
- server leaves connection open after sending response
- 2RTT for TCP setup + getting base html
- 1 RTT for each referenced object

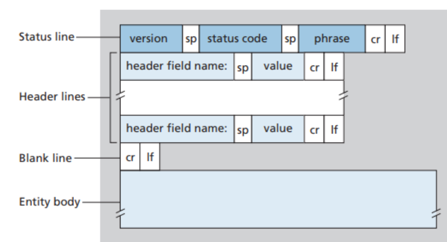
Pipelined Persistent HTTP (Parallel Persistent)
- 2 RTT for TCP setup + getting base html
- 1 RTT overall for every referenced object

HTTP Request Message Format



HTTP/1.0: Get (request object), POST (upload input to server), HEAD (Get but leave out object - testing)
HTTP/1.1 (Pers): 1.0 + PUT (upload file in entity body to path in URL), DELETE (delete file from URL)

HTTP Response Format (body can be empty)

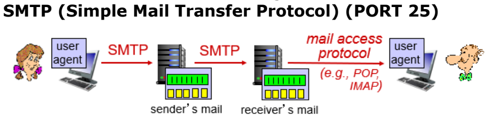


Status Codes: 200 OK, 301 Moved Permanently (new object location specified), 400 Bad Request, 404 Not Found, 505 HTTP version Not Supported
telnet URL PORT: cmd line interface to communicate with server (telnet URL 80)
HTTP/2.0(2): multiplexing multiple streams (pipelining), header compression, server push (server sends info to client before request)

HTTP Features (State & Web Cache)
- Cookies**:** client sends http request, server creates cookie (ID) for users and stores in backend database, server sends http response with **set-cookie: X**, client stores cookie value in cookie file, next time client sends http request, adds **cookie: X** to request, server uses this cookie to identify client, used when you login to a site multiple times
- Proxy Server: satisfy client request without going to origin server - browser sends HTTP request to proxy, if object in cache, returns object, else request object from origin server and then return
- **Conditional Get:** (if page is updated) - don't send object if cache has up-to-date cached version - proxy specifies **If-modified-since: <date>** in HTTP request to origin server, if not modified since the date, sends no data back, otherwise it sends the data back

Electronic Mail
- user agent(UA): mail client: outlook, iphone mail app
- mail server (MS): mailbox contains incoming messages for user, message queue of outgoing (to be sent) mail messages
- Steps: UA for **Y** to make email to **X**, send msg to **Y**'s MS (placed in msg queue), **Y**'s MS opens connection to **X**'s MS, **Y**'s MS sends msg over TCP, **X**'s MS places msg in mailbox, **X** uses UA to read msg

SMTP (Simple Mail Transfer Protocol) (PORT 25)



- client = sending MS, server = receiving mail server - messages sent using port 25
- header/body of msg must be in 7-bit ASCII
- direct transfer - sending to receiving server
- three phases: handshaking, transfer, closure
- uses persistent connections
- all 5 layers of protocol stack are @SMTP server
- PUSH model - sending mail server pushes the data onto the receiving mail server

Mail Access Protocol (POP3 IMAP HTTP use TCP)
- SMTP only used for delivery, can't be used for mail retrieval - end clients can't always be online to get emails, but SMTP acts like a client
- POP (post office protocol), IMAP (Internet Mail Access Prot, not used for email exchange btwn servers), HTTP
- POP3: authorization phase (user logs in), transaction phase (client) (**retr** msg, **dele** msg), update phase (server) (deletes marked msgs)

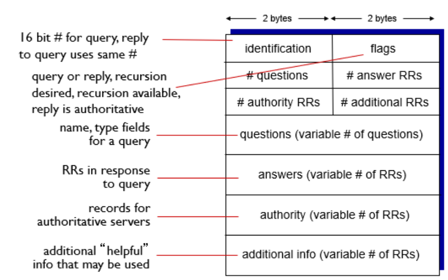
DNS (Domain Name System)
- hostname to IP address - used when user srchs URL
- distributed database - hierarchy of name srvrs
- app layer protocol - hosts, name servers comm to resolve names to IP addresses
- used in email as well to resolve (@gmail ip)
- needs to be distributed == scales better

DNS Design
- hierarchical names - ucla.edu not ucla_edu (flat name) - form the namespace hierarchy
- Name servers (resolve names) organized into hierarchy - each name server handles small portion of namespace hierarchy
- some queries can be iterative, others recursive in sequence to translate hostname
- **Root Name Servers** (".") - contacts TLD server if name mapping unknown
- **TLD (Top Level Domain)**- .com, .edu, etc
- **Authoritative Servers** - organizations own DNS server, providing correct name to IP map
- **Local Name Server** - not in hierarchy - each ISP, company, uni has one - when host makes DNS query, sent to local DNS server (which can query hierarchy or return cache answer)
- **DNS Iterative Query**: query local server, then root, TLD, etc til we get to authoritative server
- **DNS Recursive Query**: query local, which queries root, then root itself queries TLD, and other name servers till it gets the mapping -large overhead - but local only queries once
- **DNS Cache**: once any name server learns mapping, caches mapping (entries timeout after time-to-live expires), TLD servers typically cached in local name servers (root not visited)

DNS Records
- Format: (name, value, type, ttl)
- Type A (name = hostname, value = IP addr)
- Type NS (name = domain, value = hostname of authoritative name server for domain)
- Type CNAME (name = alias name for real name, value = real (canonical) name)
- Type MX (value = name of mail server associated with name)

nslookup -type _ URL (hostname to IP/DNS record)
dig URL - to get query time of website

DNS Protocol Messages



- DNS: fixed length (bytes) for each field
- HTTP/SMTP: ASCII, flexible length fields
- DNS Protocol vs Records: protocol messages sent between host and local DNS resolver, records stored at DNS resolvers (entry in DB)

Client Server VS P2P
- How much time to distribute file (size F) from one server to N Peers:
- u_s = server upload capacity, d_min = min peer download rate, u_i = peer i upload capacity, S = Σ # peers that own the file
- Client/Server: must upload one file copy per client, each client must download file copy:
T ≥ max{N*F/u_s, F/d_min}
- P2P: must upload at least 1 copy, each client must download file copy, aggregate must download N*F bits
T ≥ max{F/u_s, F/d_min, N*F/(u_s + S*u_i)}
- As N grows, so does S, so P2P scales

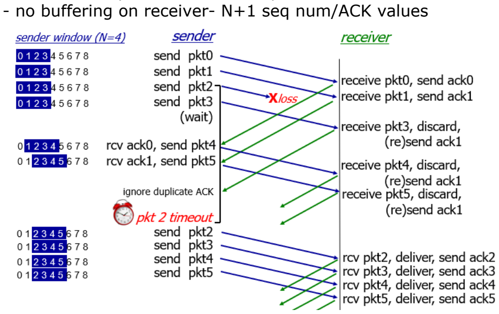
Streaming Multimedia (DASH) (HTTP GET)
- Dynamic Adaptive Streaming over HTTP
- server divides video file into multiple chunks, encoded at different rates, manifest file provides URL for different chunks - intelligence at client side
- client measures server-client bandwidth, using manifest, requests one chunk at a time, choosing maximum coding rate given current bandwidth - can choose different rate over time, where to request from

Content Distribution Networks (CDN)
- store/send multiple copies of videos at diff sites

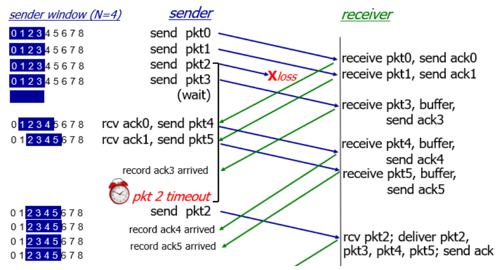
-- enter deep: push CDN servers deep into many access networks - close to users
-- bring home: smaller number of larger clusters near but not within access networks
- Steps: get url of video from webpage, query DNS server for IP, get URL for CDN, query DNS server for URL for CDN, get IP address for CDN, request video from CDN using DASH

Chapter 3: Transport Layer

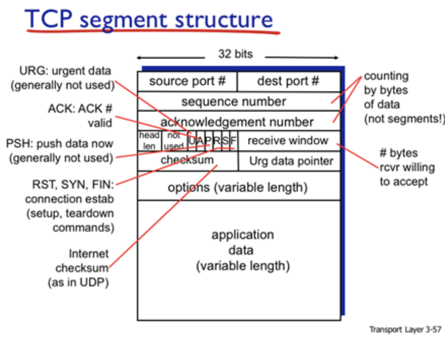
- Transport layer = provide logical comm between app processes on different hosts
Multiplexing: combining data from multiple sockets on sender side to single datastream, adding transport header for demultiplexing
Demultiplexing: use header info to deliver received segments to correct socket
- Steps: host gets IP datagrams, each datagram has source/dest IP addr/port number
- **Connectionless:** when host gets UDP segment, checks dest port # only for demux (checks dest IP @ network layer), directs UDP segment to socket with that port - datagrams with same dest port # but diff source IP/port directed to the same socket at dest
- **Connection-oriented demux**: TCP socket ID by 4 tuple (source/dest IP/port), demux uses all 4 values to direct segment, check dest/source port used for demux, check source/dest IP @ network layer
- server host can support multiple simultaneous TCP sockets (specified by 4 tuple), web servers have different sockets for each connecting client (non persistent HTTP has different socket for each request)
UDP: User Datagram Protocol
- best effort service - segments may be lost, delivered out of order, no handshaking between sender/receiver, each segment handled independently of each other
- need reliable transfer over UDP? add reliability at app layer, app-specific recovery
- has small header size, no handshaking = less delay, simple (no connection state), no congestion control - send segments fast
- Segment header: source/dest port (32 bits), length of UDP segment including header (16 bits), checksum (16 bits) - treat segment contents+header as 16 bit integers, do one's complement addition of segment contents = **checksum** - receiver computes checksum and compares values between header - diff = error (when adding, carryout from MSB added to result, then invert all bits to get checksum)
Reliable Data Transfer (RDT)
- RDT 1.0: underlying channel is reliable - no bit errors/lost packets - just send data normally
Stop and Wait: one pckt a time, 1 sender/receiver, infinite packets, senders starts one pckt trans at a time, waits for ACK
- **RDT 2.0:** channel with bit errors (flip bits) - How to detect/recover from errors?
- Detect: checksum, Recover: receiver sends ACK that pckt received, receiver sends NAK that pckt had errors
- retransmit after NAK
- **RDT 2.1:** ACK or NAK can be corrupted, sender doesn't know what happened
- sender retransmits current pckt if ACK/NAK corrupted, adds sequence number to each pckt, receiver discards duplicate packets - only needs 2 bits for seq # bc stop and wait
- **RDT 2.2:** same as 2.1, but using ACK only, receiver sends ACK for last packet received OK, including sequence number of pckt, duplicate ACK at sender results in same action as NAK, retransmit current pckt
- **RDT 3.0**: channel can also lose packets
- sender waits "reasonable" time for ACK, then retransmits if no ACK received in time based on timer
- if corrupted ACK received, no action taken
- RDT 3.0 performance is very poor - sender utilization = (L/R)/(RTT + L/R)
Pipelined Protocols
- sender allows multiple unACKed packets sent
Go Back N: up to N unacked packets in pipeline, receiver only sends cumulative ACK - no ACK if there's a gap in seq number, timer for oldest unACKed pckt, when timer expires, all unACKed packets retransmitted
- no buffering on receiver- N+1 seq num/ACK values



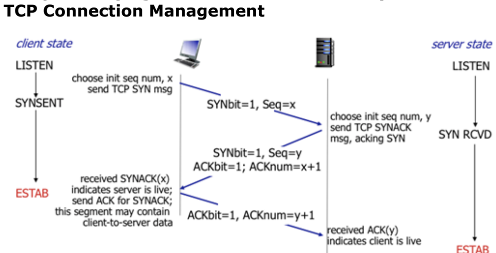
Selective Repeat: up to N unacked packets in pipeline, receiver sends individual ACK for each packet, timer for each unACKed packet, when timer expires, retransmit only that unACKed pckt, buffer N packets on receiver, 2N seq # (2N possible ACK values returning)

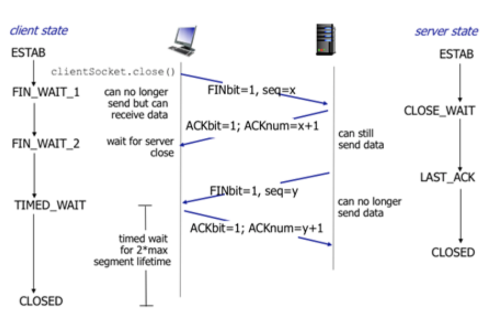


TCP
- one sender/receiver, reliable in order byte stream, pipelined, bidirectional data flow, connection oriented (ACK), sender will not overwhelm receiver



- header length (4 bits) = count in terms of 4 bytes - 60 bytes max, checksum computed using TCP header/payload + pseudo header (source/dest IP)
Setting timeout: Estimate RTT via Sample RTT (time from segment transmission til ACK)
- Karn's Alg - TCP ignores RTTs of re-sent segments - avoid ACK ambiguity (original or re-sent?)
EstimatedRTT = (1-a)*EstimatedRTT + a*SampleRTT
- exponential weighted moving average - a = 0.125 usually
DevRTT = (1-B)*DevRTT + B*abs(SampleRTT - EstimatedRTT) - large variation in EstRTT- larger safety margin, B = 0.25 usually
TimeoutInterval = EstimatedRTT + 4*DevRTT, four STD = avoids retransmission 99% of time
TCP sender events
- create segment with seq # - byte stream number of first data byte in segment, start timer if not already running (timer for oldest non acked segment)
TCP Flow Control
- receiver advertises free buffer space by including **rwnd** value in TCP header, RcvBuffer is total size of buffer on receiver side IN BYTES
- sender limits amount of in flight data to **rwnd** value - guarantees receiver buffer doesn't overflow
- to fully utilize network, **rwnd ≥ (delay*bandwidth)**
- delay = RTT, convert delay * bandwidth to BYTES
- seqnum ≥ (segment lifetime*bandwidth)
TCP Connection Management





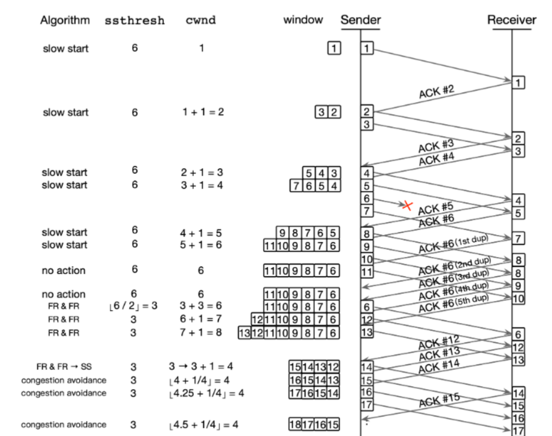
TCP Congestion Control
- too many sources sending too much data too fast for *network* (not receiver) to handle = lost packets, long delays, but we assume: all losses of TCP segments are due to internet congestion (generally true)
- idea: add congestion window - adjust window size for Selective Repeat to change TCP sending rate

Slow Start: increase rate exponentially when cwnd < ssthresh - double cwnd every RTT - cwnd += 1 MSS for every ACK, if duplicate ACK, don't increase cwnd
Congestion Avoidance: increase cwnd by 1 MSS every RTT until loss detected - when cwnd > ssthresh, cwnd += (MSS/cwnd)*MSS upon every non dup ACK
Fast Retransmit/Recovery: 3 dup ACK = loss, set ssthresh = max(cwnd/2, 2MSS), cwnd = ssthresh + 3MSS, retransmit lost packet, increase cwnd by 1 MSS every duplicate ACK, upon new ACK, cwnd = ssthresh
- in fast retransmit, max # TCP seg inflight = ssthresh
Timeout: ssthresh = max(cwnd/2, 2), cwnd = 1, resend lost packet
- window of packets is always win = min(cwnd, rwnd)
- **FR** not activated but **timeout** is: cwnd very small, no 3rd dup ACK can be sent by the rcvr; 2. multiple consecutive losses seen in the same window; 3. TCP has segment losses at its early stage of running slow start; 4. Retrans-timeout is very small (data centers)

TCP Congestion Control

Algoritms	condition	Design	action
Slow Start	cwnd <= ssthresh;	cwnd doubles per RTT	cwnd+=1MSS per ACK
Congestion Avoidance	cwnd > ssthresh	cwnd++ per RTT (additive increase)	cwnd+=1/cwnd * MSS per ACK
fast retransmit	3 duplicate ACK	reduce the cwnd by half (multiticative decreasing)	ssthresh = max(cwnd/2,2) cwnd = ssthresh + 3 MSS; retx the lost packet
fast recovery	receiving a new ACK after fast retx	finish the 1/2 reduction of cwnd in fast retx/fast recovery	cwnd = ssthresh; tx if allowed by cwnd
	upon a dup ACK after fast retx before fast recovery	("transition phrase")	cwnd +=1MSS; Note: it is different from slow start.
RTO timeout	time out	Reset everything	ssthresh = max(cwnd/2,2) cwnd = 1; retx the lost packet

TCP Throughput
- Average = 0.75*WindowSize/RTT bytes/sec
- loss prob L: throughput = 1.22*MSS/(RTT*sqrt(L))



When ACK #15 is received, ssthresh = 3 and cwnd = 4

Sample Server:

```
int sockfd, new_fd; //listen on sock_fd, new connection on new_fd
struct sockaddr_in my_addr;  //my address
struct sockaddr_in their_addr; //connector addr
int sin_size;
if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1) {
            perror("socket");
            exit(1);
}
/* set the address info */
my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(MYPORT); //short, network byte order
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
/* INADDR_ANY allows clients to connect to any one of the host's
IP addresses. Optionally, use this line if you know the IP to use:
my_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
*/
memset(my_addr.sin_zero, '\0', sizeof(my_addr.sin_zero));
if (bind(sockfd,(struct sockaddr *)&my_addr, sizeof(my_addr))
== -1) {
            perror("bind");
            exit(1);
}
if (listen(sockfd, BACKLOG) == -1) {
            perror("listen");
            exit(1);
}
while (1) { /* main accept() loop */
  sin_size = sizeof(struct sockaddr_in);
  if ((new_fd = accept(sockfd, (struct sockaddr*)&their_addr,
&sin_size)) == -1) {
            perror("accept");
            continue;
  }
  char buffer[MESSAGE_SIZE];
  int nbytes;
  if ((nbytes = read(new_fd, buffer, MESSAGE_SIZE)) == -1) {
            perror("recv");
            continue;
  }
  if((nbytes = write(new_fd, buffer, nbytes)) == -1) {
    perror("send");
    continue;
  }
  close(new_fd);
```