Introduction Made by Ivan Koh and Li Mingyang

What is the Internet

• a computer network that interconnects billions of computing devices throughout the world.

• All devices hooked up to the Internet are called **hosts/end systems** -> further split hosts into **clients** and **servers**, run network applications. eg browsers <-> web servers

• End systems are connected together by a network of **communication links** and **packet switches**, (**Routers** are packet switches used in the network core. **Link-layer switches** are used in access networks.) access Internet through **Internet Service Providers (ISPs)** & run protocols.

Network Edge: The **access network:** network that physically connects an end system to the first router on a path from that end system to any distant end system. eg residential, mobile, institutional. Hosts connect via WLAN or physical cable.

Network Core: a mesh of interconnected routers. Data transmitted through network via circuit and packet switching.

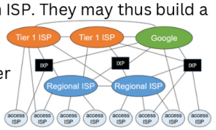
Circuit Switching: end-end resources alloc to & reserved for "call" btw src, dest. Call setup required, guaranteed constant rate performance. Circuit segment idle->no sharing. Commonly used in traditnal telephone networks. Divide link bandwidth into "pieces" (freq, time div).

Packet Switching: End systems break down long messages into smaller chunks known as **packets** (length L bits) and transmitted onto link (**transmission rate R,** aka **link capacity or link bandwidth**). packet transmission delay=L bits/R bits/s

- Store-and-Forward: Most common. Router must receive the entire packet from an inbound link before they can transmit the first bit onto an outbound link. e2e delay: 2*L/R
- **Routing & Addressing**: Routers determine src-dest route taken by packets via routing algorithms. (addressing: each packet needs to carry source and destination information)

Packet is generally better than Circuit: no setup/teardown req, resources are shared on demand vs reserved, best effort vs guaranteed service. Internet **uses packet switching.** -> excessive congestion possible.

Connecting the Internet: Hosts are connected to the Internet via an access ISP which can provide wired/wireless connectivity. The access ISPs themselves must further be interconnected, resulting in a network of networks, with a hierarchy of ISPs: Regional ISP: ISP that the access ISPs in the region connect to, act as middlemen. Tier-1 ISP: ISP that regional ISPs connect to. They are the highest level of ISP. Internet Exchange Point (IXP): When ISPs of the same level peer with each other (peering link), so they can skip the upstream ISP. They may thus build a IXP where multiple ISPs can peer together.



Content-Provider Networks: A company's own network to bring services content closer to users, eg Google.

Who Runs Internet? Network Information Centre (NIC): IP address and Internet naming. **The Internet Society (ISOC)**: Internet related standards, education and policy. **Internet Architecture Board (IAB)**: Issue and update technical standards regarding Internet protocols. **Internet Engineering Task Force (IETF)**: Protocol engineering, development and standardization arm of the IAB. Standards r published as RFCs (request for comments)

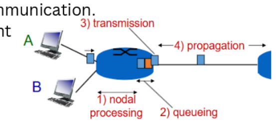
Delay, Loss & Throughput in Networks: Packet Loss: router queue has finite capacity. Packet arriving at full q = dropped. May be retransmitted by previous node, source host, or not at all.

E-2-E Packet Delay = all 4 delays added together. use *traceroute*

DELAYS

Nodal Processing Delay: check for bit-level errors and determine where to direct the packet. < microseconds. **Queuing Delay**: Time spent waiting to be transmitted onto the link. Depends on the number of earlier-arriving packets being queued or transmitted. (microsec - millisec). **Transmission Delay**: L: packet length (bits), R: link bandwidth (bits/s). Delay=L/R. (microsec - millisec) (pour into pipe) **Propagation Delay**: d: length of physical link, s: propagation speed in medium (~2*10^8 m/sec), delay=d/s (usually millisec) (travel across pipe). Use traceroute -> show E2E delay **Throughput:** how many bits transmitted per unit time, measured for e2e communication.

Link capacity (bandwidth): (R) meant for a specific link in the chain. 10^3 = K, 6 = M, 9 = G, 12 = T, 15 = P -3 = milli, -6 = micro, -9 = n, -12 = p, -15 = f



Protocol Layers & Service Models: Internet supports many network apps (games, email etc) that exchange msgs, comm among peers according to **protocols**.

Protocol: define the format and order of messages exchanged and the actions taken after messages are sent or received. Logically organized into "layers". Each layer provide service, simple interfaces between layers, hide details from each other. Explicit structure -> identification, r/s of complex system's pieces. Modularization eases maintenance, updating of system.

Stack: Application: apps treats Internet as black box. Info packet here = message. eg HTTP, FTP. **Transport**: Transports application-layer messages between application endpoints. transport-layer packet=segment. eg TCP, UDP. **Network**: Moves network-layer packets (datagrams) from one host to another. eg. IP protocol. **Link**: Moves packet btw neighboring network elements. eg WiFi. **Physical**: Moves bits from one node to another.

Application Layer

Architectures: **Client-server:** Server waits for incoming requests and provides required services to client. Client initiates contact with server and requests for service. Client usually implemented in browser. **P2P:** No dedicated server, uses direct communication between pairs of intermittently connected hosts called peers. Highly scalable, but difficult to manage. **Hybrid:** eg instant msging-> chatting is P2P, presence detection is centralized (user registers IP w central server when it comes online, contacts server to find IP address of buddies.

Transport services: **Reliable Data Transfer**: Some apps req. data to be sent 100% correctly to the receiver (file xfer), while others are loss-tolerant apps (Spotify). **Throughput**: Some apps need some minimum throughput (Youtube), while other apps (file xfer) can use whatever is available. **Timing**: Real-time applications (gaming) require low delays to be effective. **Security**: encryption, data integrity, authentication

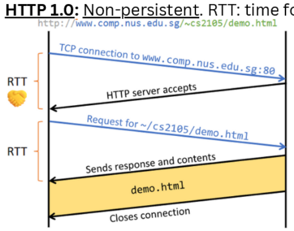
Application-Layer Protocols define: **Types of messages exchanged**: e.g request, response messages. **Message syntax:** eg. what fields in msgs, how fields r delineated. **Message semantics:** what field info means. **Rules** for when and how to send and respond to messages. There are **open protocols** defined in RFCs (Request for Comments) eg HTTP that allow interoperability, while there are some **proprietary protocols** eg Skype.

The Web & HTTP Web Page: HTML file & several referenced objects. **Objects**: A file, eg HTML file, JPEG, Java applet, etc. that is addressable by a **Uniform Resource Locator (URL)**. **Hostname**: eg. http://www.comp.nus.edu.sg **Path name**: eg. /cs2105/img/doge.jpg

HyperText Transfer Protocol (HTTP): the Web's app layer protocol, uses client/server model. Client is browser that req, recv, displays Web obj. Server sends obj in response to req.

http 1.0: RFC 1945, http 1.1: RFC 2616. Uses TCP. Stateless.

HTTP 1.0: Non-persistent. RTT: time for packet to travel from client to server and go back.

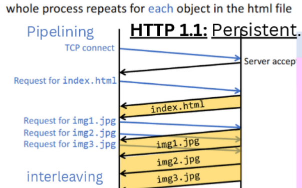


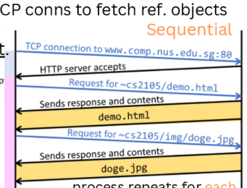
Non-persistent HTTP Response time: 1 RTT to establish TCP connection, 1 RTT for HTTP req & 1st few B of response to return file transmission time

1.0 issues: req 2 RTT/obj, OS overhead for each TCP conn, browsers often open parallel TCP conns to fetch ref. objects

whole process repeats for each object in the html file

HTTP 1.1: Persistent.





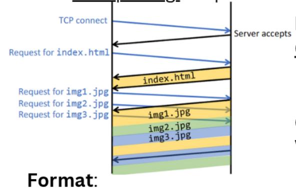
process repeats for each obj in html file

Persistent fix: server leaves conn open aft sending response, subsequent HTTP msgs btw same client/server sent over same TCP connectn. Client sends req as soon as it encounters a ref. obj (*only for pipelining*). As little as 1 RTT for all ref. obj.

Transport Protocols: **Transmission Control Protocol (TCP): connection oriented** (handshake done before message flow), **reliable** transport between sending and receiving process **flow control**: sender won't overwhelm receiver. **congestion control**: throttle sender when network is overloaded does not provide: timing, minimum throughput guarantee, security.

User Datagram Protocol (UDP): unreliable data transfer between sending and receiving process. does not provide: green stuff above. UDP forms DatagramPackets.

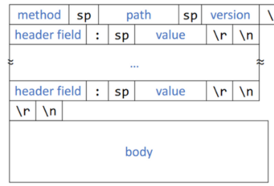
HTTP 2: Multiplexing. Resp. comes back in any order, even partially.



HTTP Request Example:

GET /index.html HTTP/1.1\r\n
Host: www.poopy.org\r\n
User-Agent: Mozilla/5.0\r\n
Connection: close\r\n
\r\n -> (marks end of header)

Format:



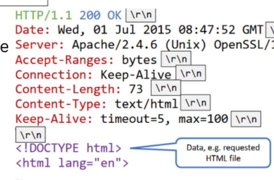
Headers: Host: used by Web proxy caches

Connection: value = close if its final req

Methods: GET: gets obj
POST: post form data
HEAD: get header w/o body
1.1 adds PUT, DELETE

HTTP Response:

status line: protocol + response code.



Status Codes:

200 OK (req successful, req obj follows), 301 Moved Permanently (new location to follow), 304 Not Modified (since specified date/time), 403 Forbidden (server declines to show pg), 404 Not Found, 500 Internet Server Error (unspecified)

Cookies: help to carry state. (stateless -> server maintains no info abt past client req). Steps: 1. Server responds w/ Set-Cookie header field, creates entry in backend database. 2. Cookie stored on user's end, sent via Cookie header field in future messages. 3. Server checks cookie for each req against backend database in future requests.

Conditional GET (Caching): server only sends obj if obj modified after date specified in If-modified-since: <date> header line. Returns 200 OK + obj if modified, else 304 Not Modified + no obj.

Domain Name System (DNS): translates btw host name, IP address. Client must carry out DNS query to determine IP address corresponding to the server name before connecting

DNS Resource Record (RR): map of <name, value, type, TTL>.

Type = A (address): Name = hostname, Value = IP address.

Type = NS (name server): name = domain, value = hostname of authoritative NS for domain. Type = CNAME (canonical name): name = alias for real name, value = real name. Type = MX (mail exchange): name = email address domain, value = mail server name.

NOTE: A hostname may be mapped to multiple IP addresses

DNS Hierarchy: DNS stores RR in distributed databases implemented in hierarchy of many name servers.

Classes: **Root servers**: Provides IP addr of TLD servers. **Top-level domain (TLD) servers:** for each TLD (.com, .org, etc) & all country TLDs (.sg etc), there is a TLD server providing the IP addr for authoritative DNS servers. **Authoritative servers**: Org's own DNS server(s) that provide authoritative hostname to IP mappings for org's named hosts (eg Web, mail). **Local DNS server:** outside of hierarchy. Each ISP has 1 local DNS svr (aka **default name server**). When host makes DNS query, it is sent to local DNS server. If ans in cache, return else it acts as proxy, fwds query into hierarchy.

DNS Query: uses UDP/53.

Iterative: local DNS server goes back n forth w each server in hierarchy
Recursive: local DNS -> root DNS -> TLD -> auth.

DNS Caching: Once NS learns mapping, it caches it & return it as non-authoritative ans next time its queried. Cached entries may be out-of-date & expire after some time (**Time-to-live**). If host changes IP addr, may not propagate globally until all caches expire. Update/notify mechanisms: RFC 2136

Socket Programming

Sockets: abstraction interface a process uses to send msgs into, & recv msgs from network. (a set of API calls). Generally consists of IP address (globally unique addr, identifies host. 32 or 128-bit.), port no. (locally unique name, identifies process. 16-bit no, 1-1023 reserved. IANA assigns port no.) Port number is from 0 to 65535. Port number is passed from application layer to transport layer.

Processes: top-lvl execution container, independent memory space. Apps run in hosts as processes. Host can run several processes. In same host, 2 proc comm with IPC (in OS). In diff host, 2 processes communicate by exchanging messages (according to protocols like HTTP). Processes/applications treat the internet as a black box, sending and receiving messages through sockets.

Threads: run in a process, shared same memory.

Types: datagram socket (UDP): only 1 socket needed. App creates each packet, attaches dest IP addr + port no while OS attaches src IP + port. Rcvr IDs sender by extracting src IP + port from rcvd pkt.

Steps:

(server) Create socket sock with socket.socket(socket.AF_INET, socket.**SOCK_DGRAM**). Bind to local port with sock.bind(address). Read data, address frm client with sock.recvfrom(4096). Send data to client with sock.sendto(data, addr).

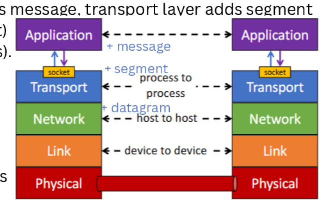
(client) Create socket **(no bind as the OS binds)**. Send data to server with sock.sendto(text.encode(), address). Read packet from server and remember to decode(). Close socket with sock.close().

Stream socket (TCP): 1 proc establish connection to another process While connection in place, data flows between process in continuous streams.

TCP server creates welcome/listening socket. When contacted by client, forks new socket for server process to communicate with that client. (**NOTE:** each client gets a unique communication socket on the server) Client creates socket to establish connection with server. Every connection has its own socket instance (welcome socket is not unique to any client).

Steps: (server) Creates welcome socket to listen with *socket(AF_INET, SOCK_STREAM)*. Bind to local port. Listen with *sock.listen(1)* (how many to put on hold). while True Accept incoming client. *conn, addr = sock.accept(). text = [] in=conn.makefile('r')*. Read with *while(data:=in.readline())!='\n':* *text.append(data)*. Write text to socket with *conn.sendall(''.join(text).encode())* and close socket with *conn.shutdown(soc ket.SHUT_RDWR)* then *conn.close()*. (shutdown = EOF so recv() returns empty byte). (client) create socket & connect with *client_sock.connect((hostname, port))*. Get input with *while text:=input():* . Write text to socket with *client_socket.send(text. encode())* & client_socket.*send(b'\r\n')*. Flush socket with *sendall(b'\r\n')*. Wait for server to shutdown with *while data:=client_skt.recv(10)*. when while loop exits, means empty byte received = server closed. Lastly, close the socket with client_socket.close().

Layering: application layer adds message. transport layer adds segment (contains src & destination port) network layer adds datagram (contains source & destination IP address).



Transport Layer

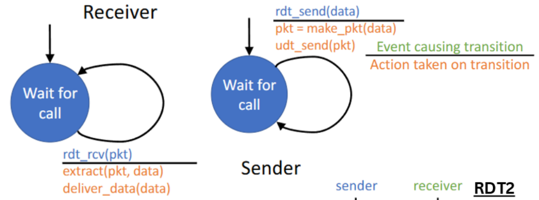
(lives on end hosts, proc-2-proc comm)

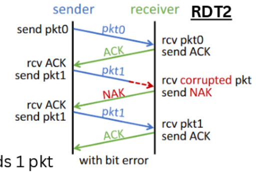
Transport layer Services:

protocols = TCP, UDP which run in hosts. Senders break app msgs into segments, pass to network layer. Rcvr rejoins segments into msg, passes to app layer. Packet switches (routers) in btw check destination IP address to decide routing.

Network layer: host-2-host, "best-effort", unreliable. IP datagram has source & destination IP address & receiving host identified by destination IP address. 1 datagram has 1 segment, which has source, destination port no.

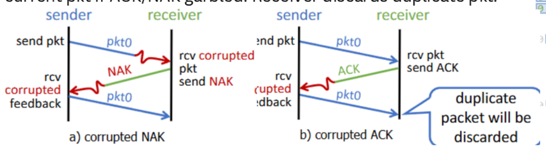
Reliable Delivery Transfer (RDT): mitigate pkt corruptn, pkt loss, pkt reordering, pkt delivery after arbitrarily long delay.
We must guarantee pkt delivery, correctness, same order.
Model: sender app to trpt lyr: rdt_send(), trpt to ntwk lyr: udt_send(), ntwk to rcvr trpt: rdt_recv(), trpt to app: deliver_data(). **RDT 1.0:** Assume underlying channel is reliable.



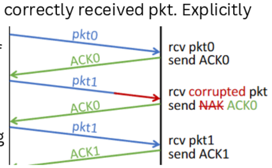


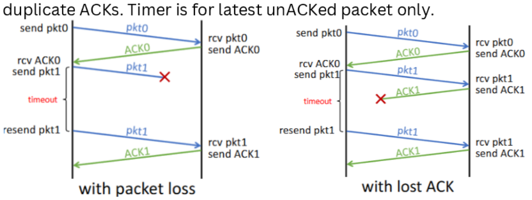
RDT 2.0: Assume channel may flip bits in pkts. Mitigate: use checksum to detect bit errors, and use ACKs for OK, NAKs for error declaration. On receiving NAK, sender retransmits pkt.
Stop-and-wait protocol: Sender sends 1 pkt at a time and waits for receiver response.

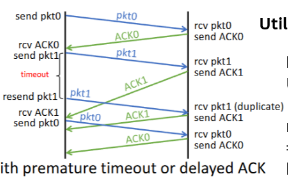
2.0 Flaw: if ACK/NAK corrupted, can't recover. If sender resends pkt when receiving garbage, receiver doesn't know if it's a duplicate.
RDT 2.1: Sender adds seq. no to each pkt (alts btw 0 & 1), retransmits current pkt if ACK/NAK garbled. Receiver discards duplicate pkt.



RDT 2.2: Replace NAK w ACK of last correctly received pkt. Explicitly include seq. no. of pkt being ack-ed. Sender resend current packet if dupe ACK received.
RDT 3.0: Assume packets can be lost/corrupted/delayed.
Sndr handles packet loss by waiting "reasonable" amt of time for ACK & resend if no ACK received till timeout. Sender does not respond to duplicate ACKs. Timer is for latest unACKed packet only.





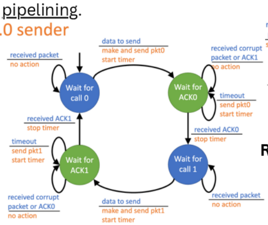


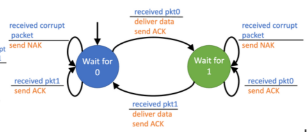
Utilization: $\frac{\text{Time spent sending}}{\text{Total time}}$

Performance so far: Utilization rate very low RTT = last bit transmitted to next bit transmitted, d_trans = 1st bit transmitted to last bit transmitted.

Util sender = very bad for stop-and-wait protocol = dtrans / (dtrans+RTT). throughput = L / (dtrans+RTT) = bad. Fix: Use pipelining.

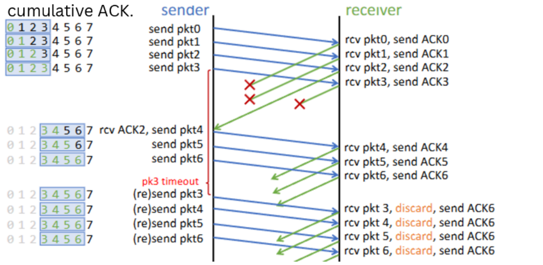
.0 sender

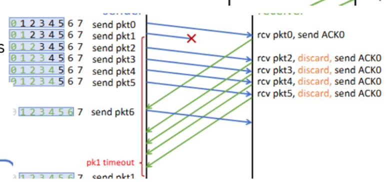




Recv: Send ACK of the latest correct packet **Sender:** Wait for time out unless recv the correct ACK

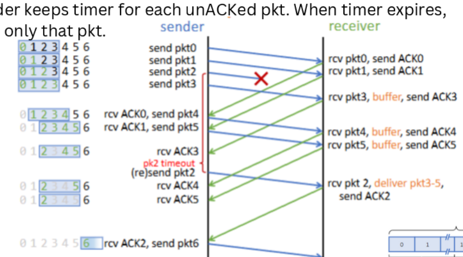
Pipelining: sender allows multiple "in-flight", yet-to-be-ack-ed pkts. Hence seq. no. range needs to increase, and must buffer at sndr &/or rcvr. (assumptns same as rdt 3.0). Util_sender = (no. of transmitted packet * dtrans) / (dtrans + RTT). With pipelining for n packets, throughput=L/(dtrans*n + RTT/2). (NOTE: RTT/2 is for the last packet).
Forms:
Go-Back-N (cumulative ACK): "ACK *n*" = all pkts <= *n* received. Sliding window that slides fwd when ACK recvd for leftmost pkt in window. Req. k bits in pkt header for 2^k seq no. Sndr can hv <= N unACKed pkt in pipeline, & keeps timer for oldest unACKed pkt. Only 1 timer kept. On timeout, retransmit ALL pkt in window. Recvr only accepts ACK pkts in order, else discards. ACKS last in-order seq. no -> cumulative ACK.

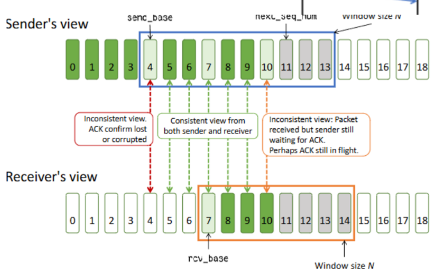


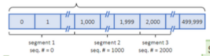


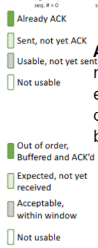
GBN only the sender needs a sliding window, receiver don't need.
For SR, both sender and receiver needs window

Selective Repeat: Key Features: receiver individually ACKs all correctly received pkts, buffers out-of-order pkts for eventual in-order delivery to upper layer. Sender keeps timer for each unACKed pkt. When timer expires, retransmit only that pkt.







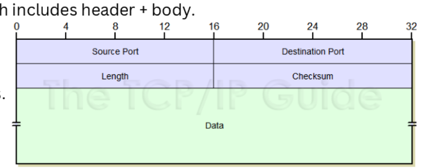


Sender	Receiver
data from above:	pkt n in [rcvbase, rcvbase+N-1]
- if next available seq # in window, send pkt	- send ACK(n)
timeout(n):	- out-of-order: buffer
- resend pkt n, restart timer	- in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt
ACK(n) in [sendbase, sendbase+N]:	pkt n in [rcvbase-N, rcvbase-1]
- mark pkt n as received	- ACK(n)
- if n is smallest unACKed pkt, advance window base to next unACKed seq. #	otherwise:
	- ignore

	Go-Back-N	Selective Repeat
#unACK packets	N packets in pipeline	N packets in pipeline
ACK style	cumulative	selective
out-of-order	discarded	buffered
timer	oldest unACK	each unACK
retransmit	all unACK	one unACK

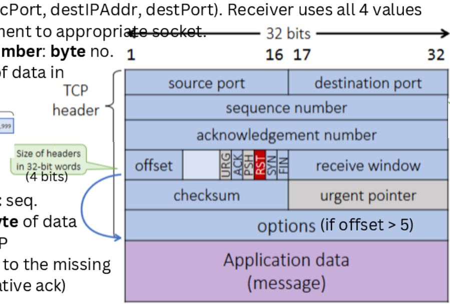
User Datagram Protocol: (UDP) (RFC 768)
Adds very little service on top of IP (connectionless mux/demux & checksum only)
. Transmission is unreliable (often used by Youtube etc since loss tolerant, rate sensitive). To achieve reliable transmission over UDP, **APPLICATION LAYER** implements error detection & recovery mechanisms.
Transport-layer muxing: data from multiple sources (sockets) to be sent using only 1 "transmission channel".
Why use UDP? -> No connection setup -> reduce delay. No connection state at sender/receiver -> need less resources. Small header size -> less overhead. No congestion control -> can blast as fast as desired. **UDP Mux & Demuxing: Mux:** get data from process, form segment & pass to network layer. **Demux**: examine destination port number of segment, pass segment to socket with that port number. IP datagrams (different sources) with the same destination port number will be directed to the same UDP socket at destination.
UDP Checksum: for error detection, not correction. may hv bit error so need this. Steps: Treat UDP segment as seq of 16 bit int. Do binary addition on every 16 bit int (checksum field intially 0). Carry (if any) from MSB added to result. Compute 1's complement to get UDP checksum. (1s complement = invert ur result). verifier will compute the checksum and check if result is all ones (0xFFFF).
UDP segment: 32 bits wide. line 1, bits 0-15: source port #, 16-31 destination port #. line 2 bit 0-15 data length, 16-31 checksum. rest = msg. Note: length includes header + body.

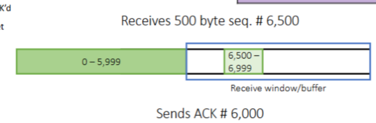
UDP length wo data is 8 bytes. TCP length wo data is 20 bytes. **MSS** doesn't include header. **MTU** does.



Transmission Control Protocol (TCP): RFC 793, 1122, 1323, 2018, 2581, ... -> conn-oriented (handshake before sending app data). Reliable, in-order byte stream (app passes data to TCP and TCP forms segment in view of **MSS** (max segment size, typically derived from link-layer's **MTU** - Maximum Transmission Unit). Flow ctrl, congestn ctrl prevents unnecessary losses. **Problem:** Communication channel is completely unreliable (pkt corrupt, loss, delayed, reordered). **Overview:** TCP socket can be IDed by 4-tuple (srcIPAddr, srcPort, destIPAddr, destPort). Receiver uses all 4 values to direct segment to appropriate socket.
Sequence Number: byte no. of first byte of data in segment.

ACK Number: seq. no. of next **byte** of data expected. TCP only ACKs up to the missing byte. (cumulative ack)





TCP Echo Server: TCP is full duplex, i.e bi-directional data flow. ACKs are "piggybacked" on data segment.
"Piggybacking": Data can be sent with ACK. ACK will be processed as long as ACK bit = 1
Out-of-order handling: TCP doesn't specify how to handle, but people usually just do buffering.

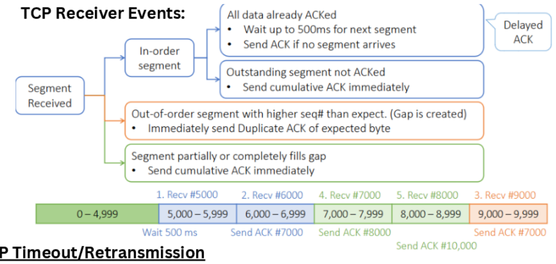
TCP Sender Events

```
loop(forever)
  switch(event)
    event: data received from application
        create TCP segment with nextSeqNum
        if (timer not currently running)
            start timer
        pass segment to IP
        nextSeqNum += length(data)

    event: timer timeout
      retransmit unacknowledged segment with smallest seq num
      start timer

    event: ACK received, with ACK num #y
      if (y > sendBase)
        sendBase = y
        if (there are still unacknowledged segments)
            start timer
```

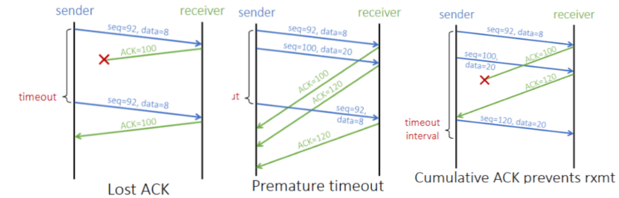
Sender only keeps one timer
Retransmit only oldest unACK segment
Cumulative ACK

Echo Server

TCP Receiver Events:



TCP Timeout/Retransmission

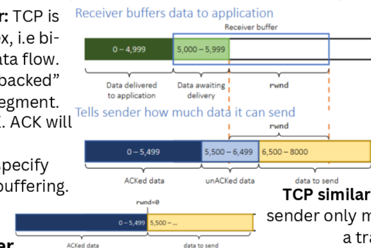


TCP Timeout Value: too short a value leads to premature timeout, unnecessary retransmission. too long leads to slow rxn to loss. Hence, timeout > estimated RTT. **Steps:** Take SampleRTT (RTT_s). Compute RTT_ε, **estimatedRTT** = $(1-\alpha)\cdot RTT_\varepsilon + \alpha\cdot RTT_s$ Typical value of α = 1/8
Deviation of RTT = $RTT_{dev} = (1-\beta)\cdot RTT_{dev} + \beta\cdot|RTT_s - RTT_\varepsilon|$
Typical value of β = 1/4. **Retransmission Time Out (RTO) Interval:** = $RTT_\varepsilon + 4\times RTT_{dev}$ (rttdev = "safety margin")
TCP Fast Retransmission (RFC 2001): timeout often relatively long. If 3 duplicate ACKs received, resend segment immediately.
TCP Connection Establishment: established using a 3-way handshake. Agree on connection & exchange parameters. **Half-open connections** lead to SYN flooding (DoS style attack by sending SYN), SYN/ACK flooding (DoS)
Steps:

- Client sends special TCP segmt w no app data, random initial seq no *client_isn* and SYN bit 1.
- Once server receives, it extracts TCP segmt from datagram, allocates TCP buffers & vars to conn, replies w conn-granted segmt (*SYNACK*), which has no app data, SYN bit 1, ACK no = *client_isn* + 1, ACK bit 1, random initial seq no *server_isn*.
- Once client recvd, client also allocates buffers, vars to conn. Third segmt can carry client-to-server data in payload. SYN bit 0, ACK no = *server_isn* + 1, ACK bit 1.

TCP Closing Connection:
1. Client sends special TCP (shutdown) segment to server with no app data, FIN bit 1, seq no. *u*.
Counts bytes, not seg #
2. Server sends back an ACK segment w ACK bit 1, ACK no *u* + 1, & client cannot send any info over but can receive info.
3. Server then sends its own shutdown segment, like step 1.
4. Client ACKs the shutdown segment like step 2.

TCP Flow Control (receiver too much data): Receiver buffers data to app, tells sender how much data it can send via "Receive Window". Sender will send a 0-data segment (zero-window probe). Receiver will send back updated receive window. If rcwd=0, probe again later else continue sending.



RST (reset) bit: set to 1 when error happens. (server send reset to client if it does not want to accept connection.**URG (urgent) bit**: when urgent data needs immediate processing. **Urgent pointer field** = byte where data ends. **PSH bit**: force TCP to send data to app layer immediately.
TCP similarities and differences with GBN: TCP sender only maintains the smallest seq number of a transmitted but but unacknowledged byte(send base) and the seq number of next byte to be sent(NextSeqNum) which looks like GBN. However, TCP will buffer correctly received out of order packets(not GBN).

Congestion control: for when network itself has too much traffic. receiver say its missing packets from sender, so sender reduce send rate.