eecs489 COMPUTER NETWORKS

Lecture 29:
TCP Connection Establishment

Internet Protocol Stack

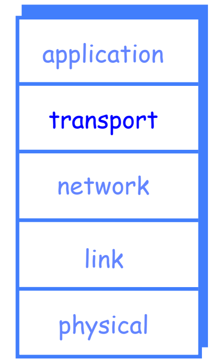
application: supporting network applications
- HTTP, SMTP, FTP, etc.

transport: endhost-endhost data transfer
- TCP, UDP

network: routing of datagrams from source to destination
- IP, routing protocols

link: data transfer between neighboring network elements
- Ethernet, WiFi

physical: bits "on the wire"

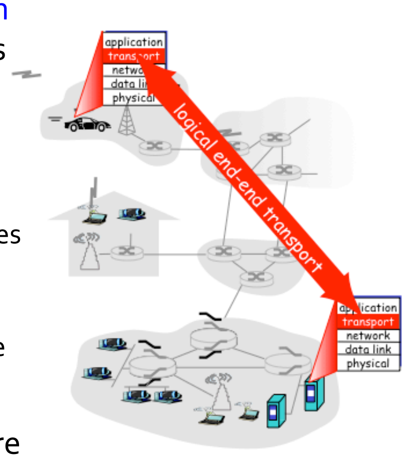


Transport Protocols

Provide logical communication between application processes running on different hosts

Run on end hosts
- sender: breaks each application message into segments, and passes them onto the network layer
- receiver: reassembles segments into messages, passes them to the application layer

Multiple transport protocols are available to applications

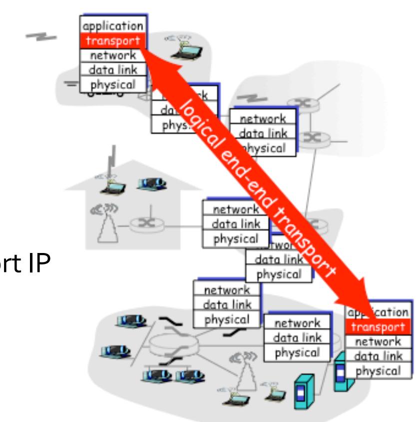


Internet Transport Protocols

Reliable, in-order delivery (TCP)
- connection setup
- flow control
- congestion control

Unreliable, unordered delivery (UDP)
- no-frills extension of best-effort IP

Services not available
- delay guarantees
- bandwidth guarantees



Why Would Anyone Use UDP?

Lightweight communication between processes:

- faster than TCP, no connection establishment/tear-down stages (1 vs. 2.5 rtts)
  - simply send messages to and receive them from a socket
- no connection state at server and client
  - avoid overhead and delays of ordered, reliable delivery
  - no allocation of buffers, parameters, sequence #s, etc.
  - making it easier to handle many active clients at once

Why Would Anyone Use UDP?

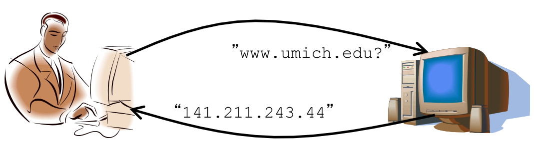
Lightweight communication between processes:

- small segment header
  - UDP header is only eight-bytes long
- fine control over what data is sent and when
  - as soon as an application process writes into the socket
  - UDP packages the data and sends the packet
- no congestion control
  - UDP can blast away as fast as the network can handle
- broadcast & multicast can only use UDP (Why?)

Popular Applications That Use UDP

Simple query protocols like DNS

- overhead of connection establishment is overkill
- easier to have the application retransmit if necessary



Multimedia streaming

- loss tolerant, rate sensitive
  - by the time a lost packet is retransmitted, it's too late
  - congestion control introduces too much jitter
  - e.g., calls, video streaming, gaming

Reliable UDP: add reliability at application layer

- application-specific error recovery!

TCP: Transmission Control Protocol

Connection oriented

- explicit set-up and tear-down of TCP session

Stream-of-bytes service

- sends and receives a stream of bytes, not discrete messages

Flow control

- prevent overflow of the receiver's buffer space

Reliable delivery

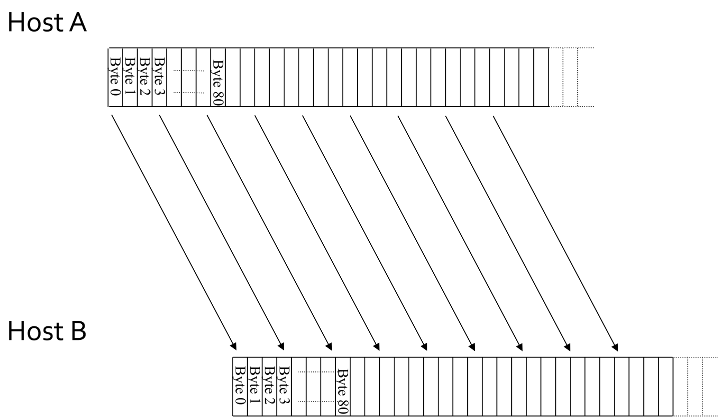
- retransmission of lost packets

Congestion control

- adapt to network congestion for the greater good

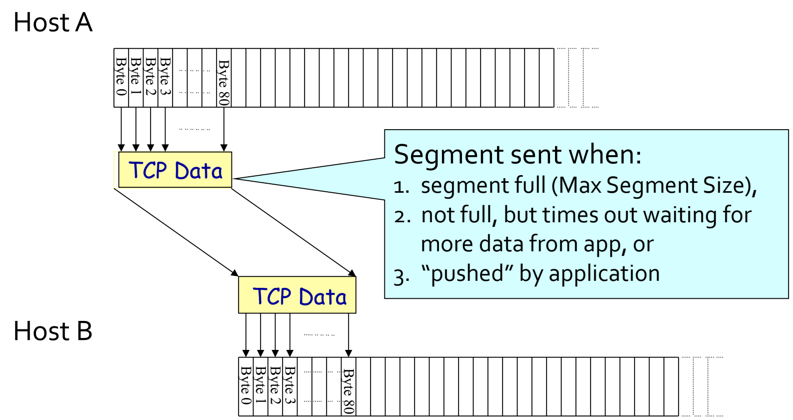
Does not provide timing guarantee nor minimum bandwidth guarantee

TCP "Stream of Bytes" Service



[Rexford]

Emulated Using TCP "Segments"



[Rexford]

TCP: Transmission Control Protocol

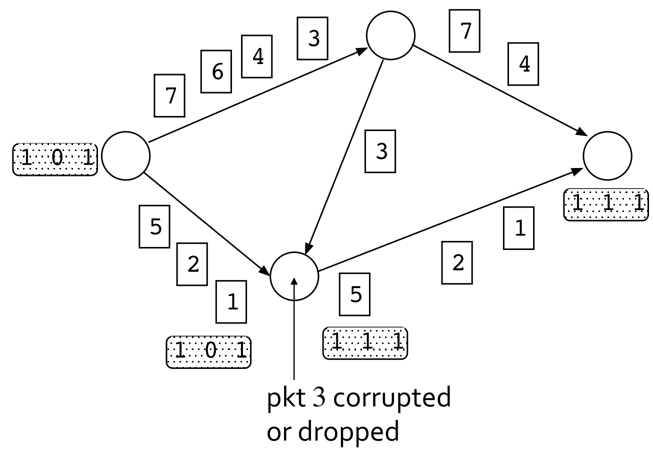
Provides reliability on datagram network
What does reliable delivery entail?

-
-
-
-

Link layer already provides reliable delivery
Why do we need provide it again at the transport layer?

-
-

Need for E2E Reliability



E2E Reliability

Lack of reliable delivery due to:
- re-routed packets
- bit error
- dropped/lost packets (due to congestion)
- system reboots

What are some of the tools available to us to achieve reliability at the transport layer, given unreliable network layer?
-
-
-
-

Sequence Number

What are the uses of sequence number in providing reliability?
-
-
-

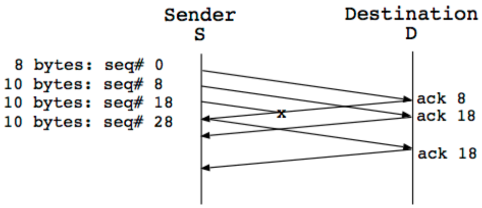
TCP's Cumulative ACK

ACKs the last byte received in-order

Tells sender the next-expected seq#

If bytes 0 to n have been received, ACK says $n+1$

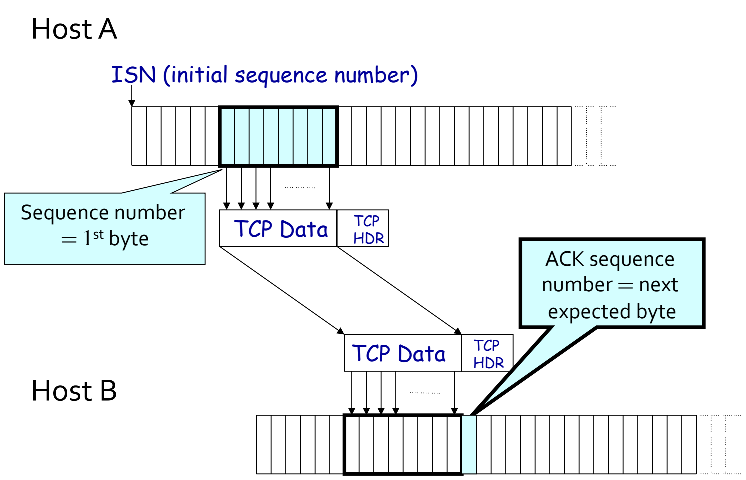
subsequent out-of-order packets generate the same cumulative ACK:



Advantage: lost ACK can be "covered" by later ACKs

Disadvantage: size of gap between two packets not known to sender

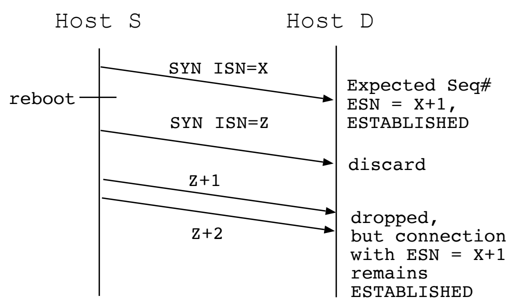
TCP Cumulative ACK



Connection Establishment

TCP SYNchronization packet to establish a connection carries the Initial Sequence Number (ISN)

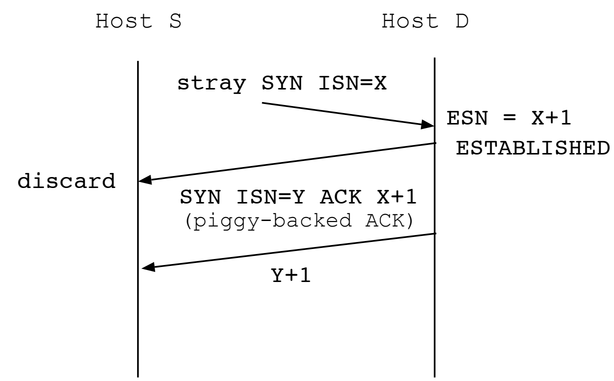
First try:



Lesson: connection request must be ACKed

Connection Establishment

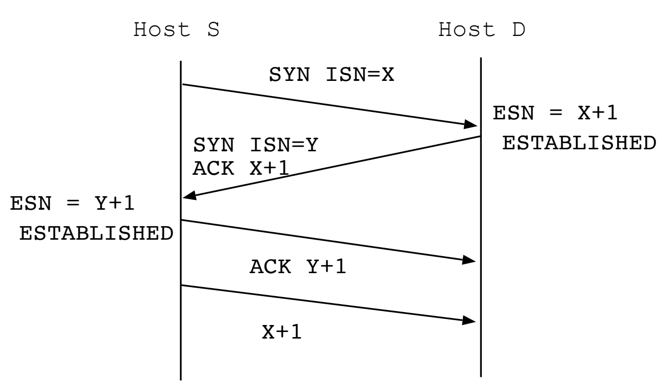
Second try:



Lesson: connection ACK must be ACKed or rejected

TCP Connection Establishment

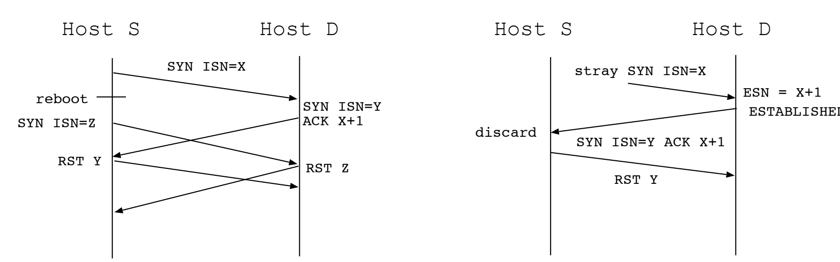
Three-way handshake:



SYN uses a seq#

Three-Way Handshake

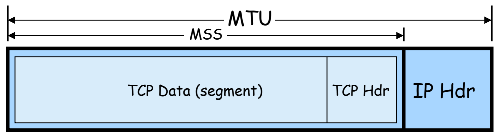
How three-way handshake solves the original problems:



What if the SYN packet is lost?

- since there's no good way to gauge RTT
- some TCP sender times out after 3-6 seconds

TCP Segment



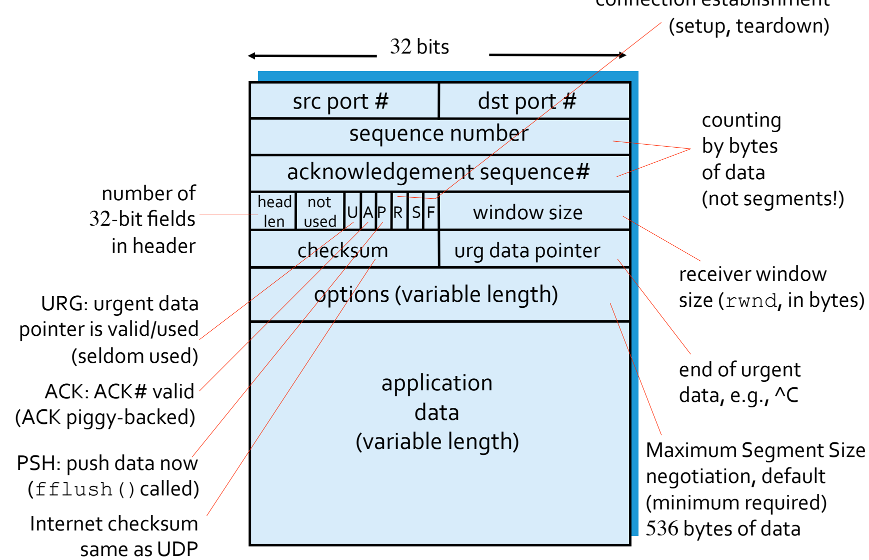
IP packet
- no bigger than Maximum Transmission Unit (MTU)
- e.g., up to 1500 bytes on an Ethernet

TCP packet
- IP packet with a TCP header and data inside
- TCP header is typically 20 bytes long

TCP segment
- no more than Maximum Segment Size (MSS) bytes
- e.g., up to 1460 consecutive bytes from the stream
- (note: PA3's MSS includes IP header)

[Rexford]

TCP Header

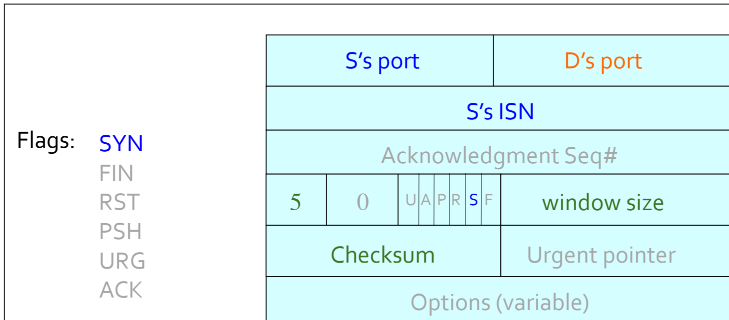


TCP Header Fields

Sequence number:
- sequence numbers count bytes sent
- seq# of a packet is the seq# of the first byte it carries

Acknowledgement sequence number:
allows for piggy-backed ACK
- TCP traffic is often bidirectional
  - both data and ACKs travel in both directions
- ACK packets have high overhead
  - 40 bytes for the TCP/IP headers, carrying no data
- piggy-backing allows a host D to send its ACK to host S along with its data for host S
- delayed ACK: TCP allows the receiver to delay sending of ACKs to increase chances of piggy-backing and to reduce number of ACKs (since they're cumulative)

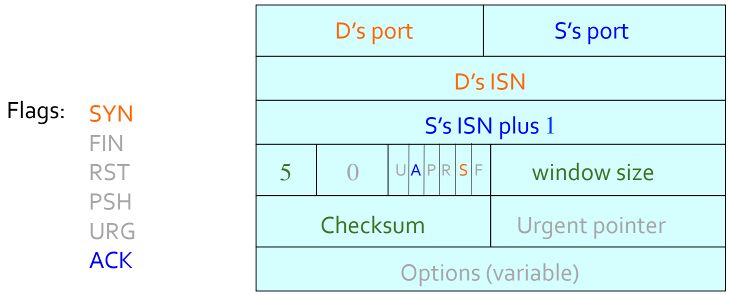
Step 1: S's Initial SYN Packet



Host S tells Host D it wants to open a connection…

[Rexford]

Step 2: D's SYNACK Packet

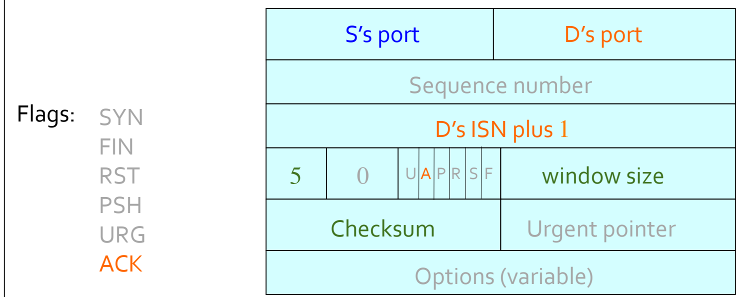


Host D tells Host S it accepts, and is ready to hear the next byte…

Upon receiving Host D's SYNACK packet, Host S can start sending data

[Rexford]

Step 3: S's ACK of the SYNACK



Host S tells Host D it is okay to start sending

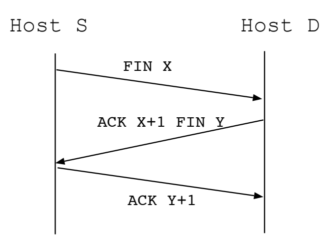
Upon receiving Host S's packet, Host D can start sending data

[Rexford]

Connection Tear-down

When to release a connection?
How do you know the other side is done sending and all sent packets have arrived?

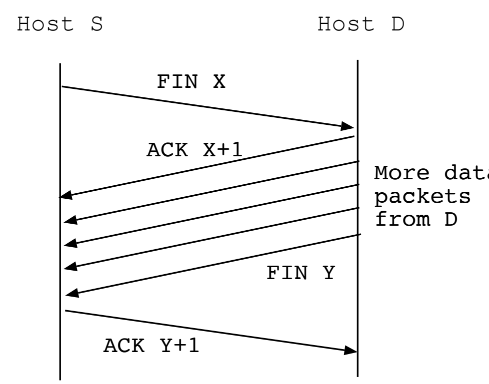
Use 3-way handshake to tear-down connection:



FIN also uses a seq#

Connection Tear-down

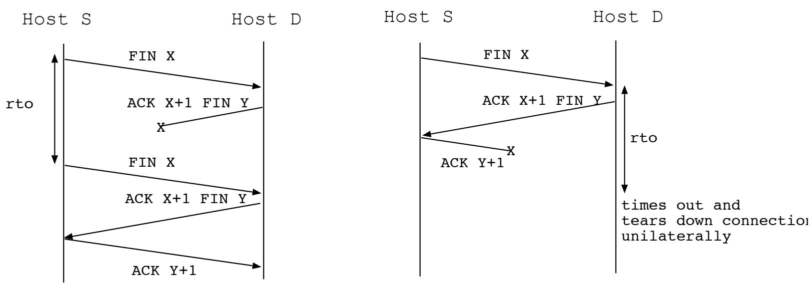
If the other side still has data to send:



Why not delay `ACK X+1` until `FIN Y`?

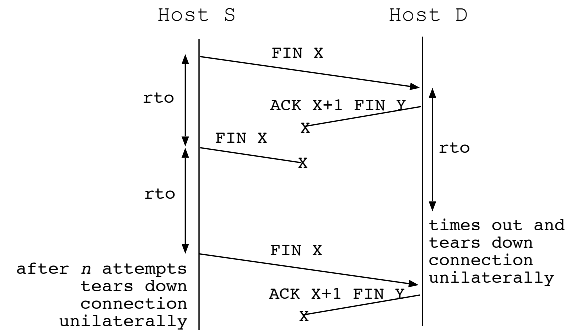
Connection Tear-down

Still depends on timeout for correctness:



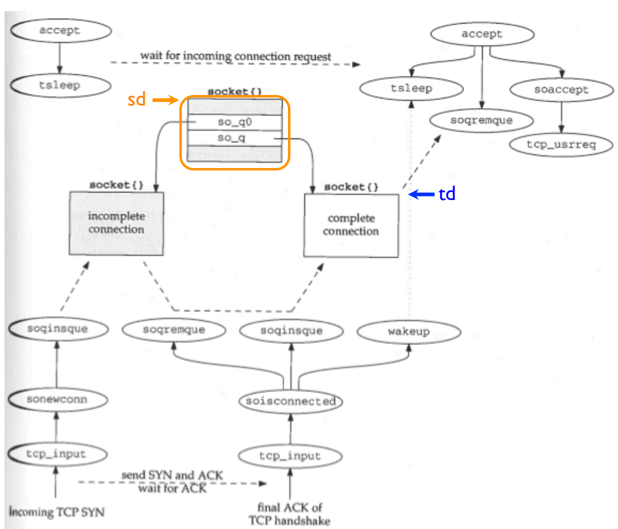
Connection Tear-down

Still depends on timeout for correctness:



TCP connection tear-down depends on timers for correctness, but uses 3-way handshake for performance improvement

Socket Connection Queues

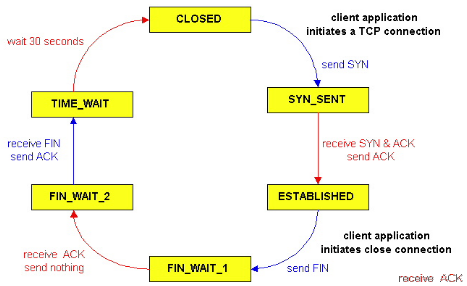


Stevens TCP/IP Illustrated v. 2 pp. 441, 461

TCP Connection Establishment Demo

```
% ifconfig -a
% sudo tcpdump -i en0 -S host web.eecs.umich.edu
```

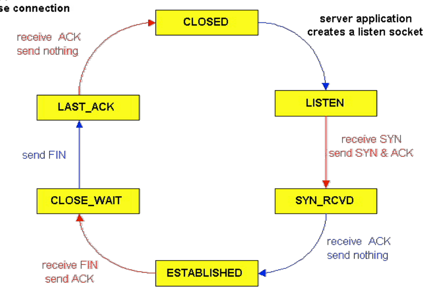
TCP Connection Management FSM

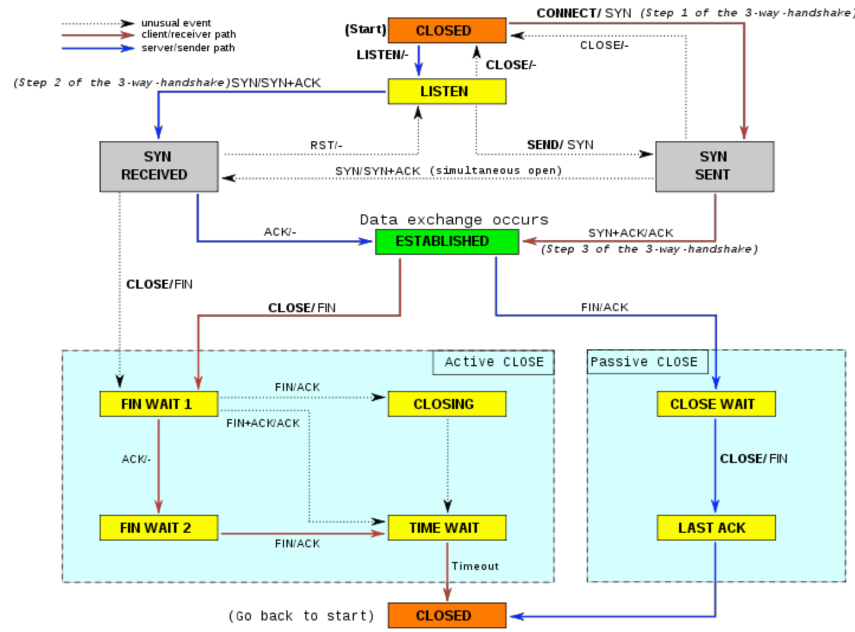


TCP client lifecycle

Put together . . .

TCP server lifecycle





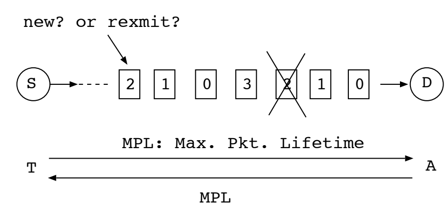
Finite Sequence Number Space

Issues arising from having finite sequence number space:

1. choice of sequence space size
2. sequence number wrap around
3. initial sequence number (ISN) choice

Sequence Number Space Size

If we had only 2 bits to keep track of sequence numbers:



Let:
A: time taken by receiver to ACK packet
T: time sender continues retransmitting if an ACK is not received

Maximum Segment Lifetime (MSL): $2\text{MPL} + T + A$

Duplicated Sequence Number

Maximum Segment Lifetime (MSL): $2\text{MPL} + T + A$

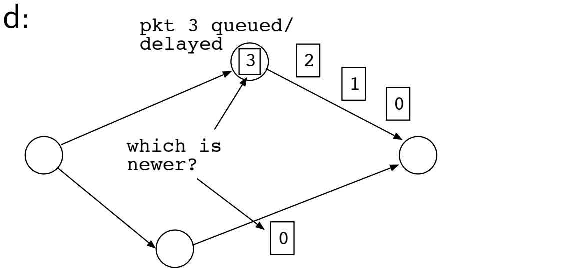
Want: no sequence number may be duplicated within an MSL

What would cause a sequence number to be duplicated within an MSL?

- sequence number wraps around (within a single connection)
- sequence number re-used (across connections)

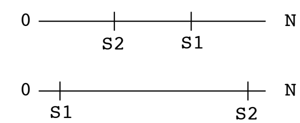
Sequence Number Wrap Around

Sequence space (N) is finite and sequence number can wrap around:



Assuming s_1 and s_2 are not more than $N/2$ apart, $s_1 > s_2$ if either:

1. $|s_1 - s_2| < N/2$ and $s_1 > s_2$, or
2. $|s_1 - s_2| > N/2$ and $s_1 < s_2$



Required Sequence Number Size

To prevent duplicate sequence number: for $s_1 > s_2$, s_1 and s_2 cannot be more than $N/2$ apart ($|s_1 - s_2| < N/2$) within an MSL

For TCP, $N = 2^{32}–1$, $N/2$ maximum separation requirement means that only $n = 31$ bits are usable

Let μ be the transmission bandwidth, worst case, assuming sender can "fill the pipe", want:
$\mu < (N/2)/\text{MSL}$ or, $2^n > \mu{*}\text{MSL}$

- example: SF-NY MPL is 25 msec.
  let $\text{MSL} = 2$ min, for $n = 31$ bits,
  μ must be < 17.8 MB/s (143 Mbps)

Required Sequence Number Size

But TCP transmission is constrained by the receiver's advertised window (`rwnd`), which is of 16-bit size

Only 64 KB can be outstanding at any one time, which takes less than ⅓ of a second to clear the network even at "slow" T1 speed (1.5 Mbps)

So we don't have to worry about sequence number re-use due to wrap around

Initial Sequence Number (ISN)

Sequence number for the very first byte

Why not always start with an ISN of 0?

IP addresses and port #s uniquely identify a connection, but port numbers get reused when:

-
-

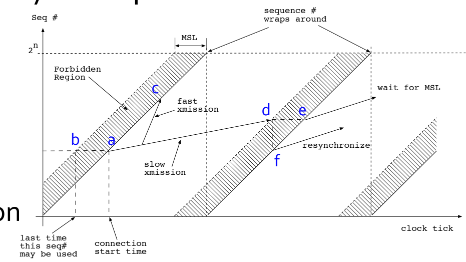
and connections get "reincarnated"

We want ISNs that will not let packets from an old connection that are still in flight to be mistaken for packets from the new connection

ISN from System Clock

Assume clock keeps ticking even when machine is down
Want: no seq. number may be duplicated within an MSL

Forbidden region: using sequence number currently in forbidden region could cause duplicate ISN chosen by "reincarnated" connection



What to do on hitting forbidden region (d)?

d) nothing: seqno duplicated by "reincarnated" connection
e) wait for MSL before resuming transmission
f) resynchronize sequence number
either (e) or (f), connection stalled

TCP's Handling of ISN

TCP requires changing the ISN over time

- set from a 32-bit clock that ticks every 4 μseconds
- which only wraps around once every 4.55 hours
  - unlikely for reincarnated connection to share seqno

Connection cannot be reused for MSL time

- on connection tear-down, wait for 2MSL (TIME-WAIT state, 30 secs)
  `bind: Address already in use`
- on reboot, do not create connection for MSL (2 minutes boot time, so not a problem)