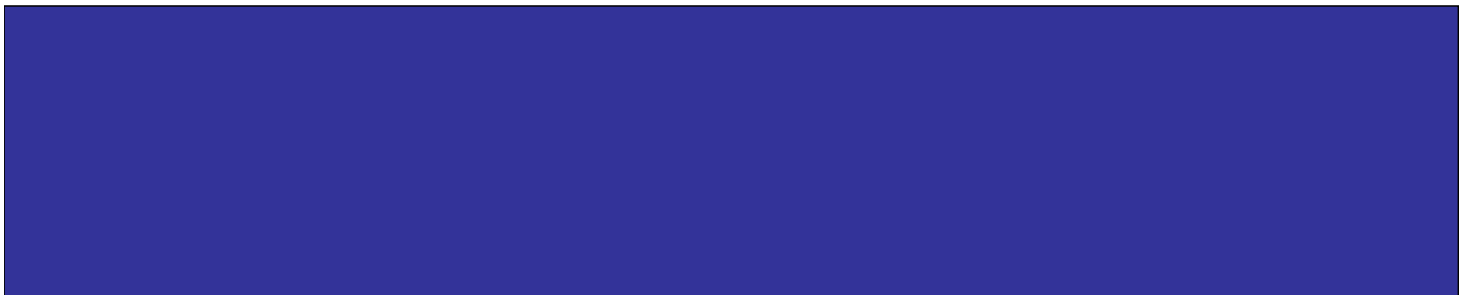


Transport Layer Protocols

EE4204: Computer Networks

Mehul Motani

motani@nus.edu.sg



Outline

- Principles underlying transport-layer services
 - (De)multiplexing
 - Detecting corruption
 - Reliable delivery
 - Flow control & Congestion control
- Transport-layer protocols in the Internet
 - User Datagram Protocol (UDP)
 - Transmission Control Protocol (TCP)

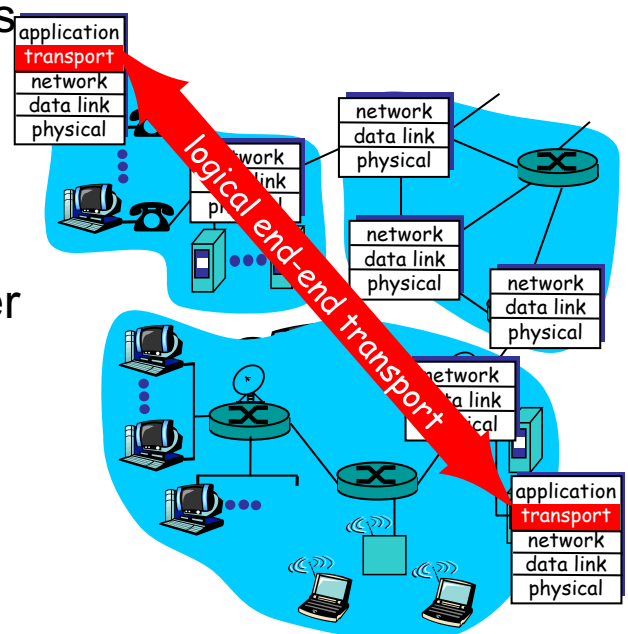
Note: Some slides & graphics adapted from Kurose & Ross, Computer Networking

Role of Transport Layer

- Application layer
 - Communication for specific applications
 - E.g., HyperText Transfer Protocol (HTTP), File Transfer Protocol (FTP), Network News Transfer Protocol (NNTP)
- Transport layer
 - Communication between processes (e.g., socket)
 - Relies on network layer and serves the application layer
 - E.g., TCP and UDP
- Network layer
 - Logical communication between nodes
 - Hides details of the link technology
 - E.g., IP

Transport Protocols

- Provide *logical communication* between application processes running on different hosts
- Run on end hosts
 - Sender: breaks application messages into **segments**, and passes to network layer
 - Receiver: reassembles segments into messages, passes to application layer
- Multiple transport protocol available to applications
 - Internet: TCP and UDP

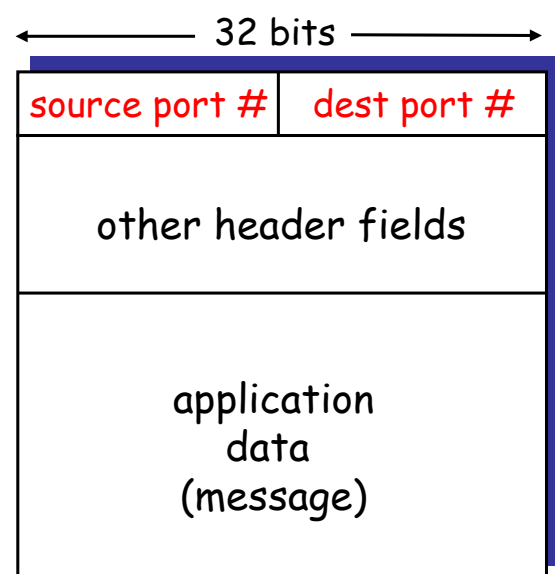


Internet Transport Protocols

- Datagram messaging service (UDP)
 - No-frills extension of “best-effort” IP
- Reliable, in-order delivery (TCP)
 - Connection set-up
 - Discarding of corrupted packets
 - Retransmission of lost packets
 - Flow control
 - Congestion control
- Other services not available
 - Delay guarantees
 - Bandwidth guarantees

Multiplexing and Demultiplexing

- Host receives IP datagrams
 - Each datagram has source and destination IP address,
 - Each datagram carries one transport-layer segment
 - Each segment has source and destination port number
- Host uses IP addresses and port numbers to direct the segment to appropriate socket



TCP/UDP segment format

Unreliable Message Delivery Service

- Lightweight communication between processes
 - Avoid overhead and delays of ordered, reliable delivery
 - Send messages to and receive them from a socket
- User Datagram Protocol (UDP)
 - IP plus port numbers to support (de)multiplexing
 - Optional error checking on the packet contents

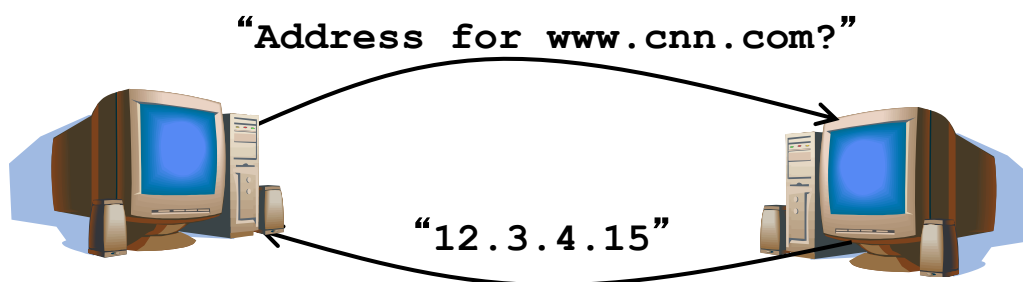
SRC port	DST port
checksum	length
DATA	

Why Would Anyone Use UDP?

- Finer control over what data is sent and when
 - As soon as an application process writes into the socket
 - ... UDP will package the data and send the packet
- No delay for connection establishment
 - UDP just blasts away without any formal preliminaries
 - ... which avoids introducing any unnecessary delays
- No connection state
 - No allocation of buffers, parameters, sequence #s, etc.
 - ... making it easier to handle many active clients at once
- Small packet header overhead
 - UDP header is only eight-bytes long

Popular Applications That Use UDP

- Multimedia streaming
 - Retransmitting lost/corrupted packets is not worthwhile
 - By the time the packet is retransmitted, it's too late
 - E.g., telephone calls, video conferencing, gaming
- Simple query protocols like Domain Name System
 - Overhead of connection establishment is overkill
 - Easier to have application retransmit if needed



Transmission Control Protocol (TCP)

- Connection oriented
 - Explicit set-up and tear-down of TCP session
- Stream-of-bytes service
 - Sends and receives a stream of bytes, not messages
- Reliable, in-order delivery
 - Checksums to detect corrupted data
 - Acknowledgments & retransmissions for reliable delivery
 - Sequence numbers to detect losses and reorder data
- Flow control
 - Prevent overflow of the receiver's buffer space
- Congestion control
 - Adapt to network congestion for the greater good

An Analogy: Talking on a Cell Phone

- Alice and Bob on their cell phones
 - Both Alice and Bob are talking
- What if Alice couldn't understand Bob?
 - Bob asks Alice to repeat what she said
- What if Bob hasn't heard Alice for a while?
 - Is Alice just being quiet?
 - Or, have Bob and Alice lost reception?
 - How long should Bob just keep on talking?
 - Maybe Alice should periodically say "uh huh"
 - ... or Bob should ask "Can you hear me now?" 😊

Some Reflections on the Example

- Acknowledgments from receiver
 - Positive: "okay" or "ACK"
 - Negative: "please repeat that" or "NACK"
- Timeout by the sender ("stop and wait")
 - Don't wait indefinitely without receiving some response
 - ... whether a positive or a negative acknowledgment
- Retransmission by the sender
 - After receiving a "NACK" from the receiver
 - After receiving no feedback from the receiver

Challenges of Reliable Data Transfer

- Over a perfectly reliable channel
 - All of the data arrives in order, just as it was sent
 - Simple: sender sends data, and receiver receives data
- Over a channel with bit errors
 - All of the data arrives in order, but some bits corrupted
 - Receiver detects errors and says “please repeat that”
 - Sender retransmits the data that were corrupted
- Over a lossy channel with bit errors
 - Some data are missing, and some bits are corrupted
 - Receiver detects errors but cannot always detect loss
 - Sender must wait for acknowledgment (“ACK” or “OK”)
 - ... and retransmit data after some time if no ACK arrives

TCP Support for Reliable Delivery

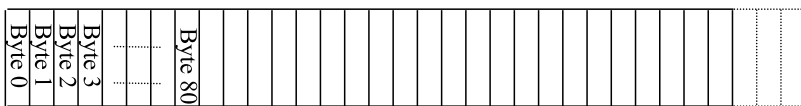
- Checksum
 - Used to detect corrupted data at the receiver
 - ...leading the receiver to drop the packet
- Sequence numbers
 - Used to detect missing data
 - ... and for putting the data back in order
- Retransmission
 - Sender retransmits lost or corrupted data
 - Timeout based on estimates of round-trip time
 - Fast retransmit algorithm for rapid retransmission

TCP Segments

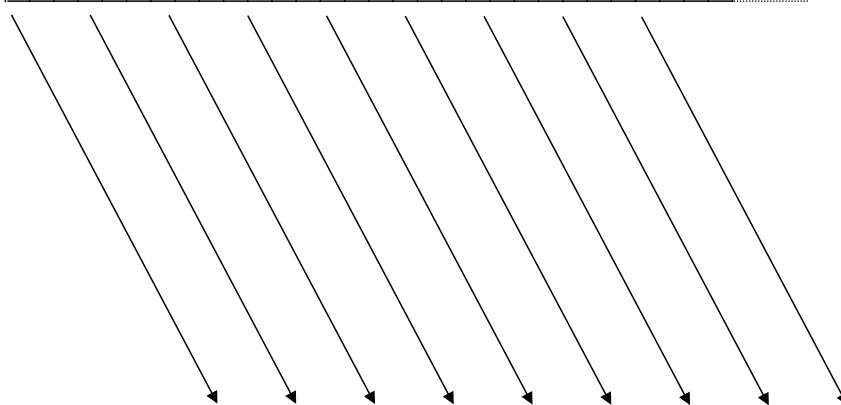


TCP “Stream of Bytes” Service

Host A

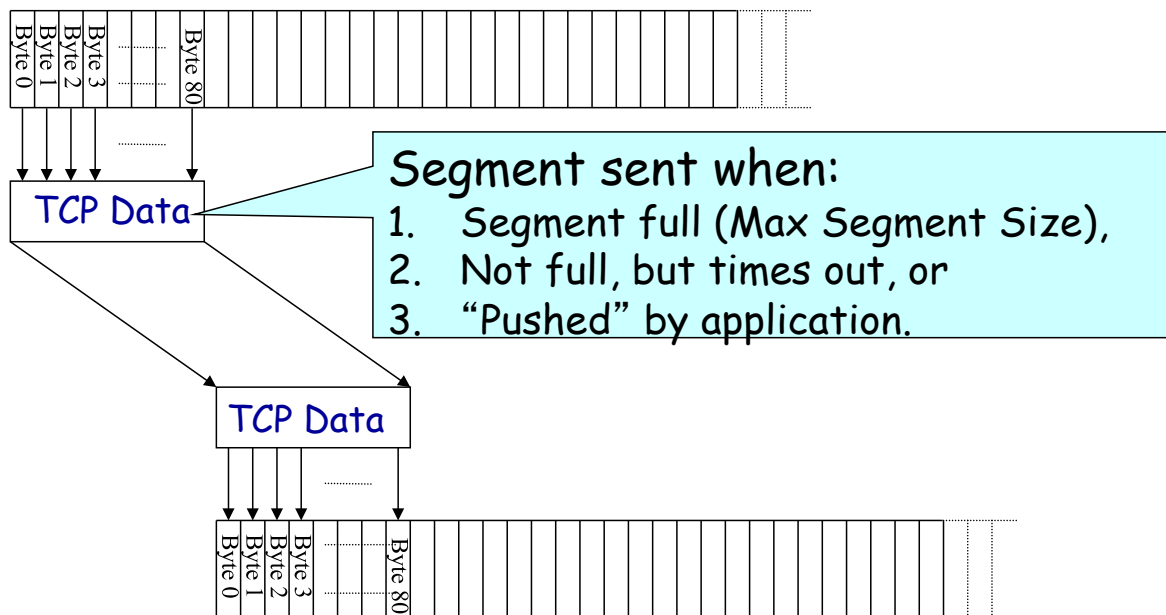


Host B



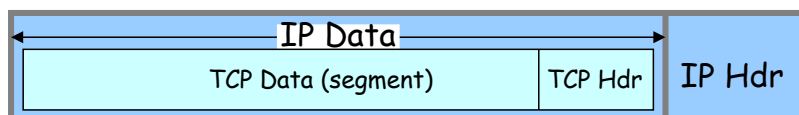
...Emulated Using TCP “Segments”

Host A



Host B

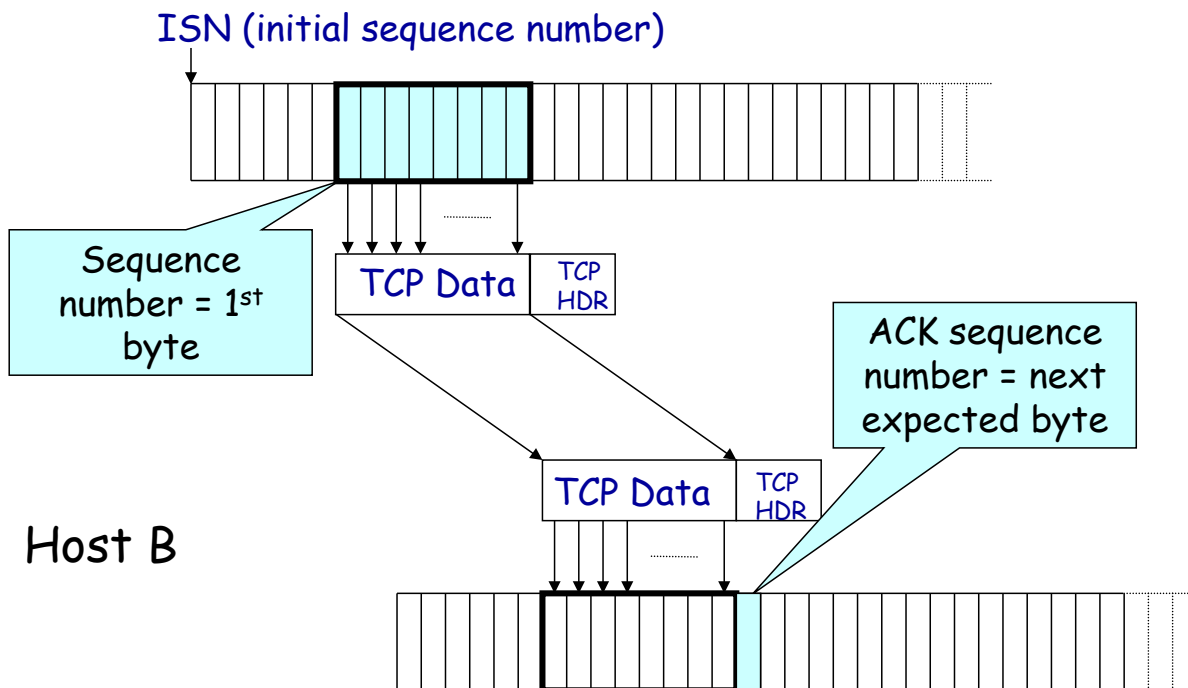
TCP Segment



- 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

Sequence Numbers

Host A



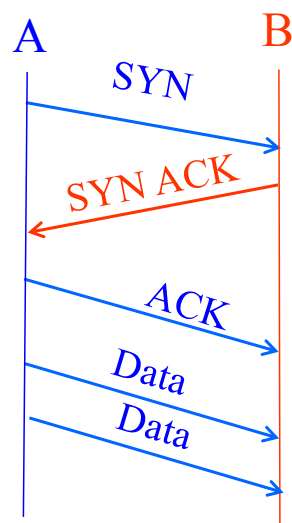
Host B

Initial Sequence Number (ISN)

- Sequence number for the very first byte
 - E.g., Why not a de facto ISN of 0?
- Practical issue
 - IP addresses and port #s uniquely identify a connection
 - Eventually, though, these port #s do get used again
 - ... and there is a chance an old packet is still in flight
 - ... and might be associated with the new connection
- So, TCP requires changing the ISN over time
 - Set from a 32-bit clock that ticks every 4 microseconds
 - ... which only wraps around once every 4.55 hours!
- But, this means the hosts need to exchange ISNs

TCP Three-Way Handshake

Establishing a TCP Connection



Each host tells its ISN to the other host.

- Three-way handshake to establish connection
 - Host A sends a **SYN** (open) to the host B
 - Host B returns a SYN acknowledgment (**SYN ACK**)
 - Host A sends an **ACK** to acknowledge the SYN ACK

TCP Header

Flags: SYN
FIN
RST
PSH
URG
ACK

Source port		Destination port	
Sequence number			
Acknowledgment			
HdrLen	0	Flags	Advertised window
Checksum		Urgent pointer	
Options (variable)			
Data			

Step 1: A's Initial SYN Packet

Flags: **SYN**
FIN
RST
PSH
URG
ACK

A' s port		B' s port	
A' s Initial Sequence Number			
Acknowledgment			
20	0	Flags	Advertised window
Checksum			Urgent pointer
Options (variable)			

A tells B it wants to open a connection...

Step 2: B's SYN-ACK Packet

Flags: **SYN**
FIN
RST
PSH
URG
ACK

B' s port			A' s port		
B' s Initial Sequence Number					
A' s ISN plus 1					
20	0	Flags	Advertised window		
Checksum			Urgent pointer		
Options (variable)					

B tells A it accepts, and is ready to hear the next byte...
... upon receiving this packet, A can start sending data

Step 3: A's ACK of the SYN-ACK

Flags: **SYN**
FIN
RST
PSH
URG
ACK

A's port		B's port	
Sequence number			
B's ISN plus 1			
20	0	Flags	Advertised window
Checksum			Urgent pointer
Options (variable)			

A tells B it wants is okay to start sending

... upon receiving this packet, B can start sending data

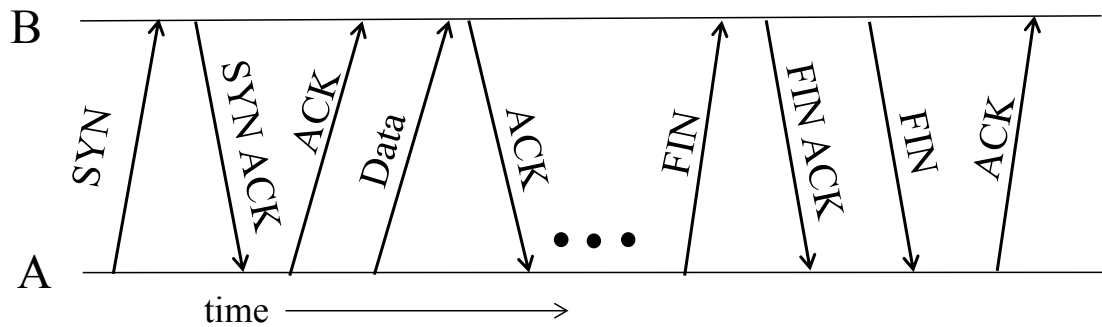
What if the SYN Packet Gets Lost?

- Suppose the SYN packet gets lost
 - Packet is lost inside the network, or
 - Server rejects the packet (e.g., listen queue is full)
- Eventually, no SYN-ACK arrives
 - Sender sets a timer and wait for the SYN-ACK
 - ... and retransmits the SYN if needed
- How should the TCP sender set the timer?
 - Sender has no idea how far away the receiver is
 - Hard to guess a reasonable length of time to wait
 - Some TCPs use a default of 3 or 6 seconds

SYN Loss and Web Downloads

- User clicks on a hypertext link
 - Browser creates a socket and does a “connect”
 - The “connect” triggers the OS to transmit a SYN
- If the SYN is lost...
 - The 3-6 seconds of delay may be very long
 - The user may get impatient
 - ... and click the hyperlink again, or click “reload”
- User triggers an “abort” of the “connect”
 - Browser creates a new socket and does a “connect”
 - Essentially, forces a faster send of a new SYN packet!
 - Sometimes very effective, and the page comes fast

Tearing Down the Connection



➤ Closing the connection

- Finish (FIN) to close and receive remaining bytes
- And other host sends a FIN ACK to acknowledge
- Reset (RST) to close and not receive remaining bytes

TCP Retransmissions

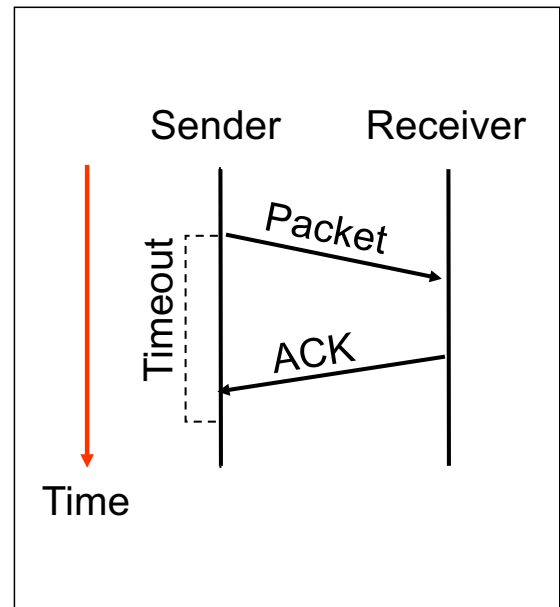
Automatic Repeat reQuest (ARQ)

➤ Automatic Repeat Request

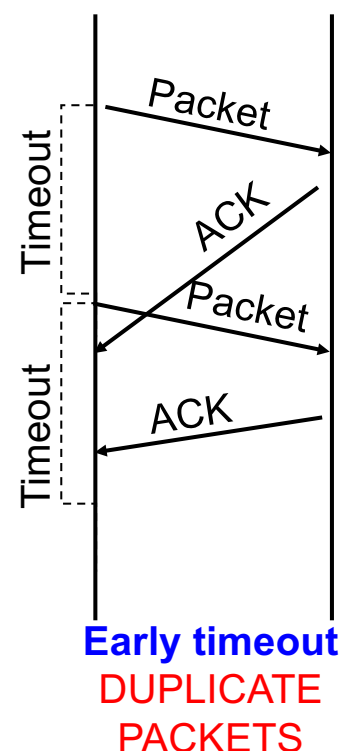
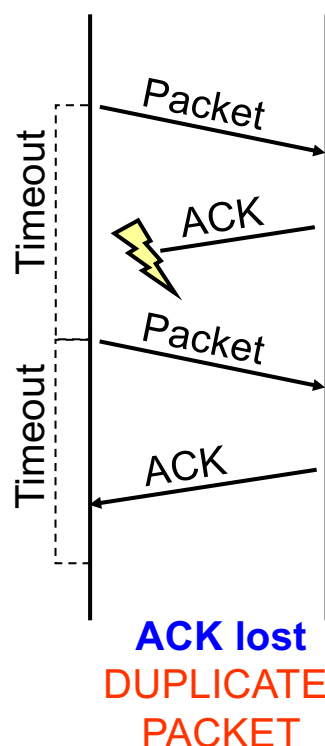
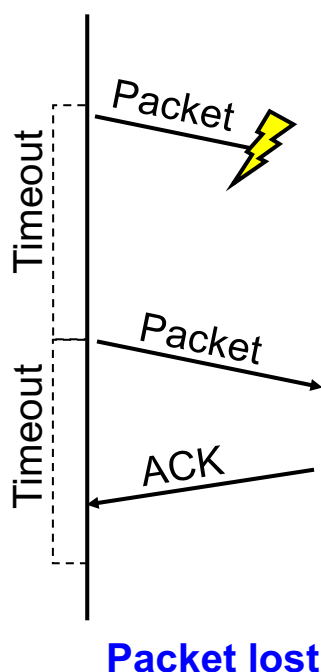
- Receiver sends acknowledgment (ACK) when it receives packet
- Sender waits for ACK and timeouts if it does not arrive within some time period

➤ Simplest ARQ protocol

- Stop and wait
- Send a packet, stop and wait until ACK arrives



Reasons for Retransmission

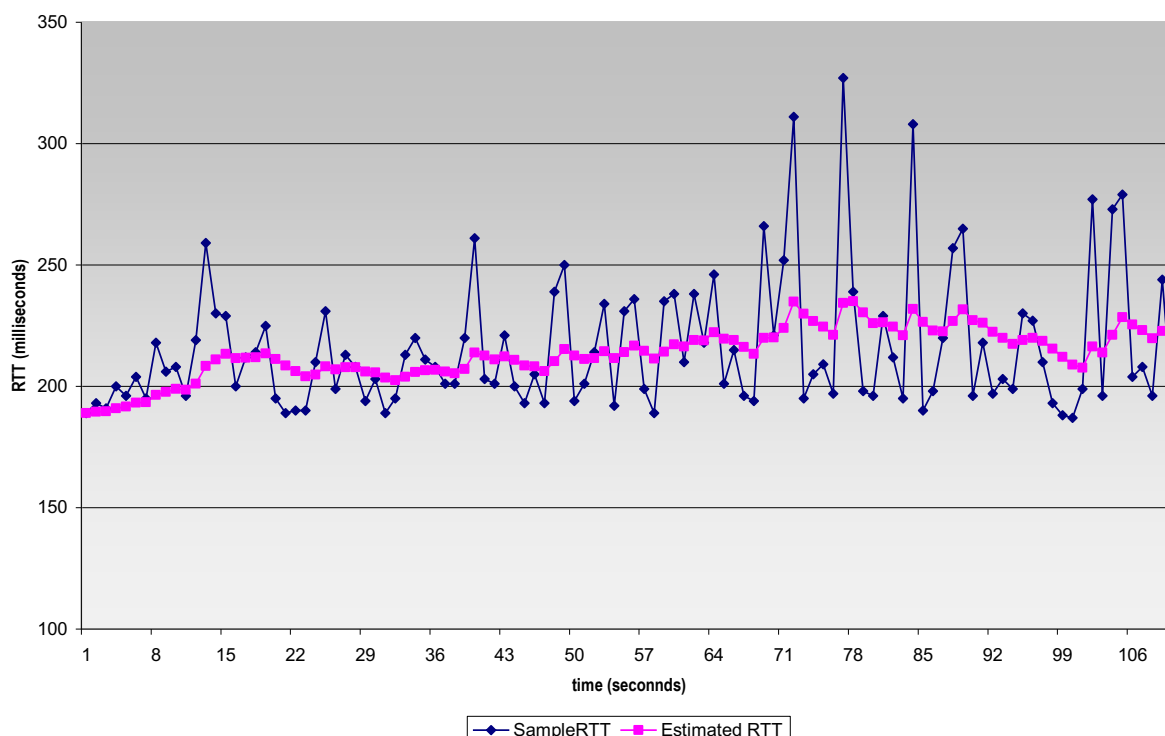


How Long Should Sender Wait?

- Sender sets a timeout to wait for an ACK
 - Too short: wasted retransmissions
 - Too long: excessive delays when packet lost
- TCP sets timeout as a function of the RTT
 - Expect ACK to arrive after an RTT
 - ... plus a fudge factor to account for queuing
- But, how does the sender know the RTT?
 - Can estimate the RTT by watching the ACKs
 - Smooth estimate: keep a running average of the RTT
 - $\text{EstimatedRTT} = a * \text{EstimatedRTT} + (1 - a) * \text{SampleRTT}$
 - Compute timeout: $\text{TimeOut} = 2 * \text{EstimatedRTT}$

Example RTT Estimation

RTT: gaia.cs.umass.edu to fantasia.eurecom.fr



A Flaw in This Approach

- An ACK doesn't really acknowledge a transmission
 - Rather, it acknowledges receipt of the data
- Consider a retransmission of a lost packet
 - If you assume the ACK goes with the 1st transmission
 - ... the SampleRTT comes out way too large
- Consider a duplicate packet
 - If you assume the ACK goes with the 2nd transmission
 - ... the Sample RTT comes out way too small
- Simple solution in the Karn/Partridge algorithm
 - Only collect samples for segments sent one single time

Yet Another Limitation...

- Doesn't consider variance in the RTT
 - If variance is small, the EstimatedRTT is pretty accurate
 - ... but, if variance is large, the estimate isn't all that good
- Better to directly consider the variance
 - Consider difference: $\text{SampleRTT} - \text{EstimatedRTT}$
 - Boost the estimate based on the difference
- Jacobson/Karels algorithm
 - See Section 5.2 of the Peterson/Davie book for details

TCP Sliding Window

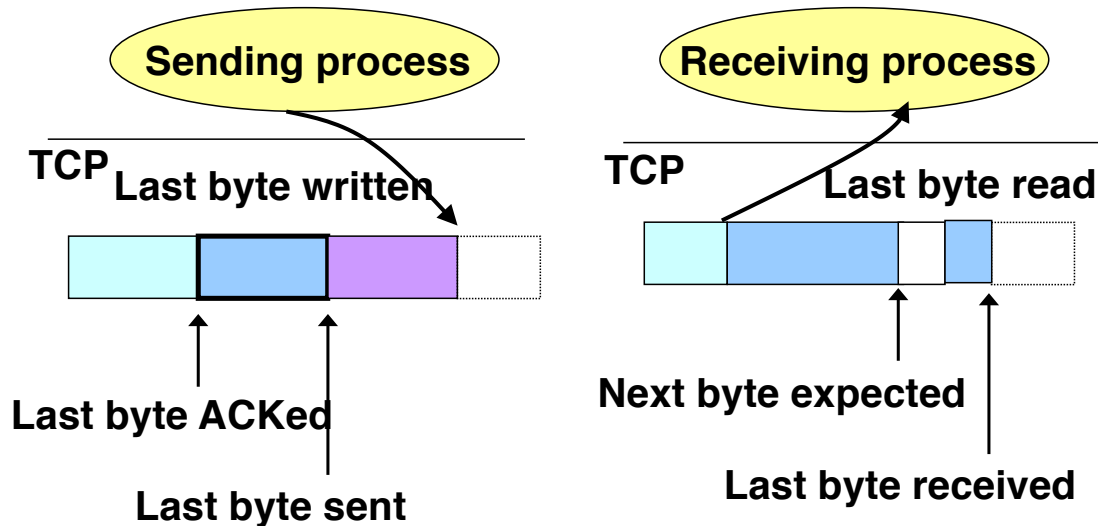


Motivation for Sliding Window

- Stop-and-wait is inefficient
 - Only one TCP segment is “in flight” at a time
 - Especially bad when delay-bandwidth product is high
- Numerical example
 - 1.5 Mbps link with a 45 msec round-trip time (RTT)
 - Delay-bandwidth product is 67.5 Kbits (or 8 KBytes)
 - But, sender can send at most one packet per RTT
 - Assuming a segment size of 1 KB (8 Kbits)
 - ... leads to 8 Kbits/segment / 45 msec/segment → 182 Kbps
 - That's just one-eighth of the 1.5 Mbps link capacity

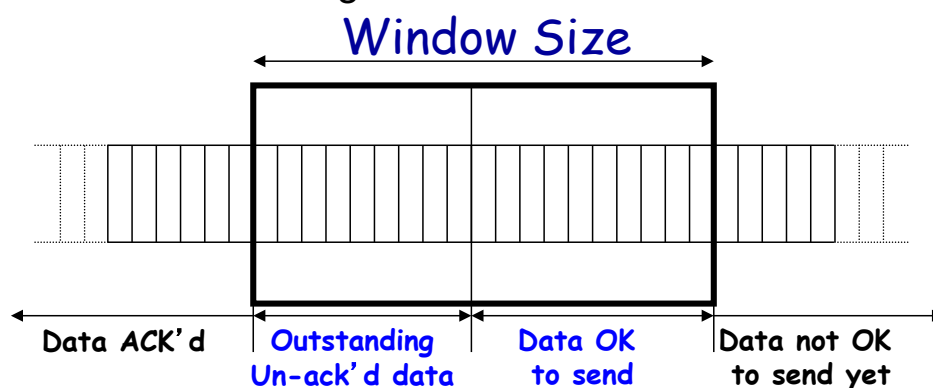
Sliding Window

- Allow a larger amount of data “in flight”
 - Allow sender to get ahead of the receiver
 - ... though not *too far* ahead

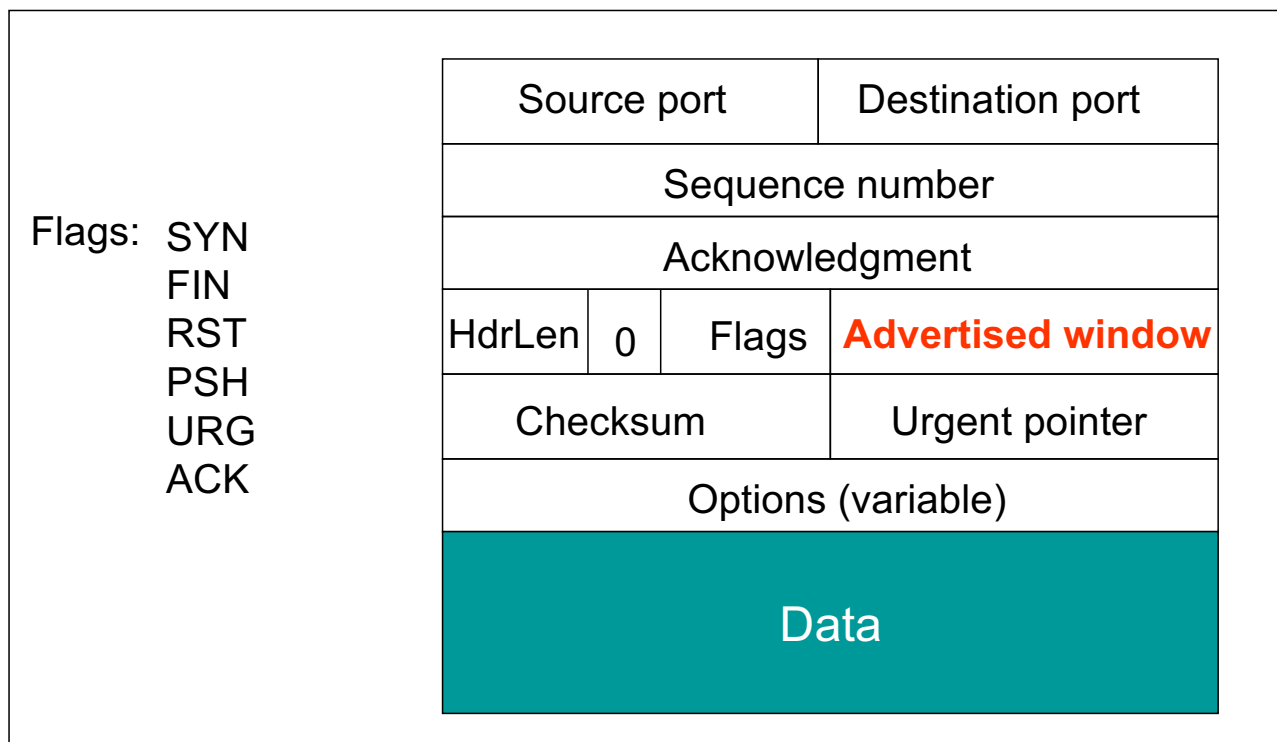


Receiver Buffering

- Window size
 - Amount that can be sent without acknowledgment
 - Receiver needs to be able to store this amount of data
- Receiver advertises the window to the sender
 - Tells the receiver the amount of free space left
 - ... and the sender agrees not to exceed this amount



TCP Header for Receiver Buffering

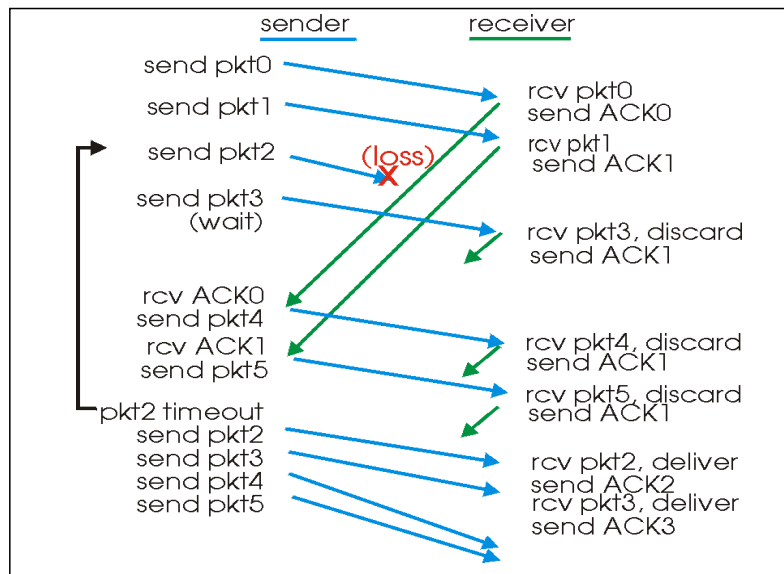


Fast Retransmission

Timeout is Inefficient

➤ Timeout-based retransmission

- Sender transmits a packet and waits until timer expires
- ... and then retransmits from the lost packet onward



Fast Retransmission

➤ Better solution possible under sliding window

- Although packet n might have been lost
- ... packets $n+1$, $n+2$, and so on might get through

➤ Idea: have the receiver send ACK packets

- ACK says that receiver is still awaiting n^{th} packet
 - And *repeated* ACKs suggest later packets have arrived
- Sender can view the “duplicate ACKs” as an early hint
 - ... that the n^{th} packet must have been lost
 - ... and perform the retransmission early

➤ Fast retransmission

- Sender retransmits data after the triple duplicate ACK

Effectiveness of Fast Retransmit

- When does Fast Retransmit work best?
 - Long data transfers
 - High likelihood of many packets in flight
 - High window size
 - High likelihood of many packets in flight
 - Low burstiness in packet losses
 - Higher likelihood that later packets arrive successfully
- Implications for Web traffic
 - Most Web transfers are short (e.g., 10 packets)
 - Short HTML files or small images
 - So, often there aren't many packets in flight
 - ... making fast retransmit less likely to "kick in"
 - Forcing users to like "reload" more often...

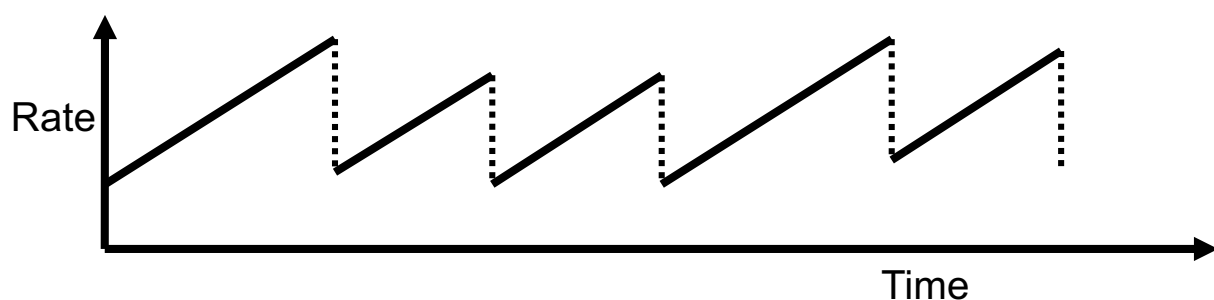
TCP Congestion Control

TCP Congestion Control

- Flow control addresses congestion at receiver, not in middle of network (for example, at intermediate routers)
- Suppose you initially send up to the size of flow control window (FW)
 - Intermediate routers may not be able to handle so much traffic
 - Congestion overflows router buffers causing lost packets causing retransmissions causing more congestion ... → congestion collapse
- Idea behind TCP Congestion Control:
 - Send only enough packets into the network that the network has the capacity to handle without loss
- Define a congestion window (CW) that can be used to respond to network congestion
 - Distinct from flow control window FW
 - Actual window size $W = \min(CW, FW)$
 - # of data bytes that can be on the link
 - Send no more data than the bottleneck can handle

TCP Congestion Control

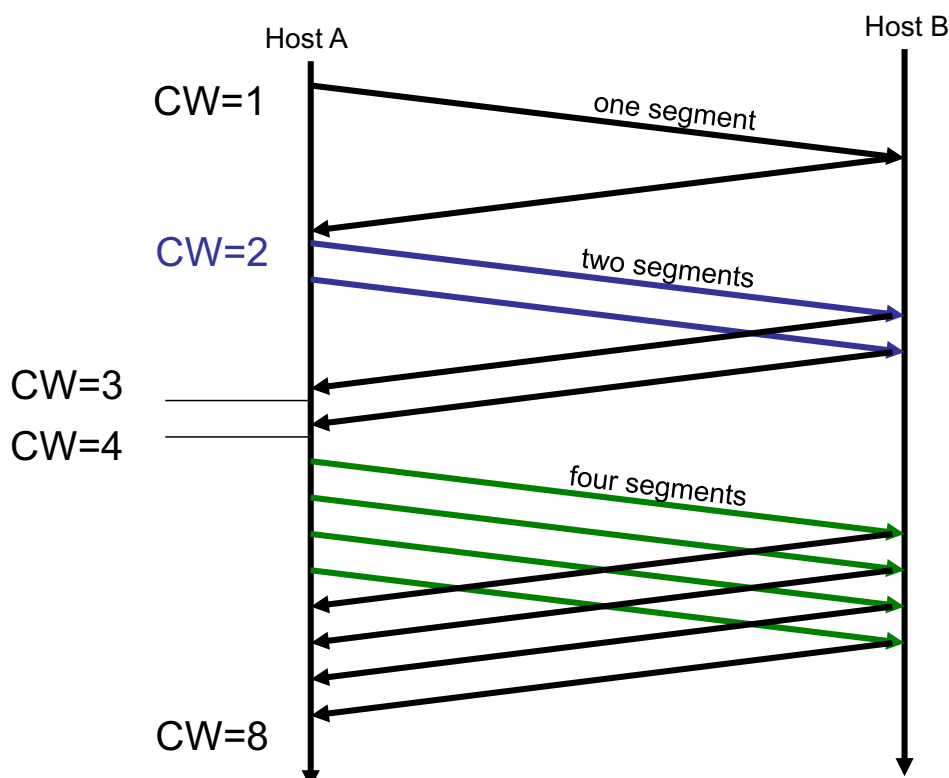
- How does sender set CW?
 - Adaptively probe the network with data segments
 - Keep expanding CW until a segment is lost, then contract CW.
 - Continue with expand/contract cycle throughout connection – “sawtooth” behavior
- Additive Increase / Multiplicative Decrease (AIMD)
 - Increment CW by one packet per RTT (*linear increase*)
 - Divide CW by two whenever a timeout occurs (*multiplicative decrease*)



TCP Slow Start

- The rate at which new packets should be injected into network is the rate at which ACKs are returned by other end
 - Use ACK's to pace transmission of packets: "self-clocking"
 - Start by setting $CW = 1$ segment (in bytes)
 - Initial segment size set by receiver
 - For each ACK that returns, increment CW by one.
 - Send 1 packet. When ACK returns, increment CW , $CW=2$
 - Send 2 packets. When 1st ACK returns, increment CW to $CW=3$, when 2nd ACK returns, increment CW to $CW=4$
 - Can send 4 packets. After 4 ACKs return, CW will be up to 8
 - Exponential increase – not "slow", quickly reach window size that the network can accommodate

TCP Slow Start

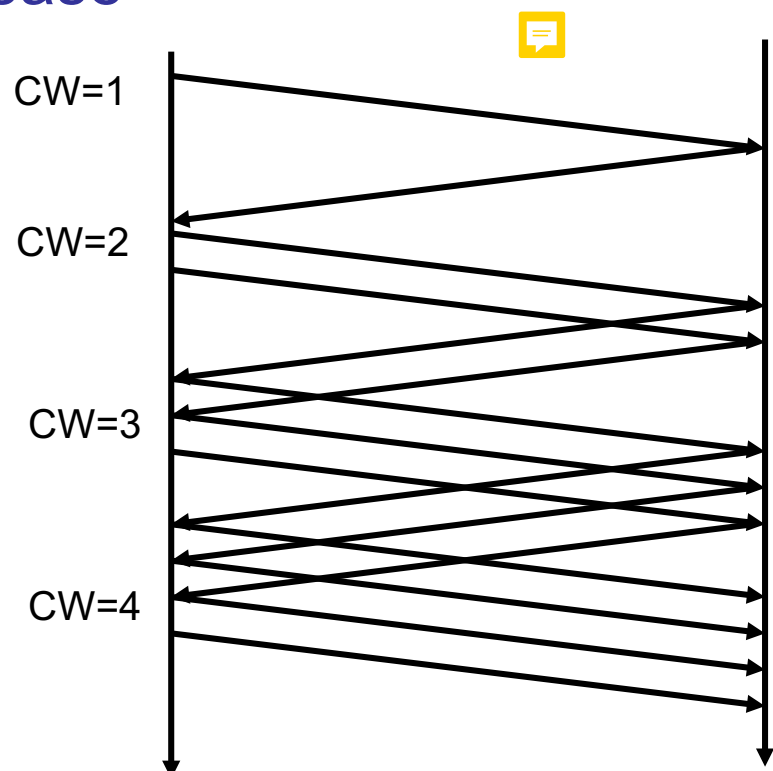


TCP Additive Increase/ Multiplicative Decrease

- How does a sender detect that CW is too large?
 - It starts to see timeouts, which are interpreted as packet loss due to congestion
- After a timeout:
 - TCP drastically resets $CW=1$ and slow starts again
 - TCP remembers that congestion occurred near CW by storing $CW/2$ in $ssthresh = CW/2$
 - $ssthresh$ = Slow Start Threshold
 - But TCP exponentially increases only to $ssthresh$, halfway to old congestion mark
 - After $CW > ssthresh$, *additively increase* CW
 - Rationale: Be cautious about sending new data packets once you get near old mark that caused timeouts/congestion

TCP Additive Increase

- If entire window's worth (CW) of packets in a RTT is ACKed w/o error, then increment CW by one
- In practice, TCP adds α/CW to CW as each ACK returns, rather than waiting for a full CW of ACKs to return



Rationale behind AIMD

- Why not just slow start exclusively (exponential increase) after timeout, instead of additive increase?
 - Be more cautious about adding new packets once you're near old congestion point.
- Each time a timeout occurs, divide CW by half and store in ssthresh: multiplicative decrease
 - Minimum ssthresh and minimum CW is one
- Why not additive decrease instead of multiplicative decrease after congestion?
 - Consequences of having a too-large congestion window are worse than having a too-small CW
 - Additive decrease can keep CW too large for too long compared to multiplicative decrease

More on TCP AIMD

- What happens if the amount of unacknowledged data is greater than CW?
 - Can't send new data
 - Retransmit unacknowledged data
 - Wait for ACKs for unacknowledged data to increase CW above size of unacknowledged data, then can send new data
- After a timeout, TCP slows down in two ways:
 - Congestion window collapses, restricting new data
 - RTO backs off exponentially, slowing down retransmission of old unacknowledged data

TCP Saw Tooth Behavior (TCP Reno)

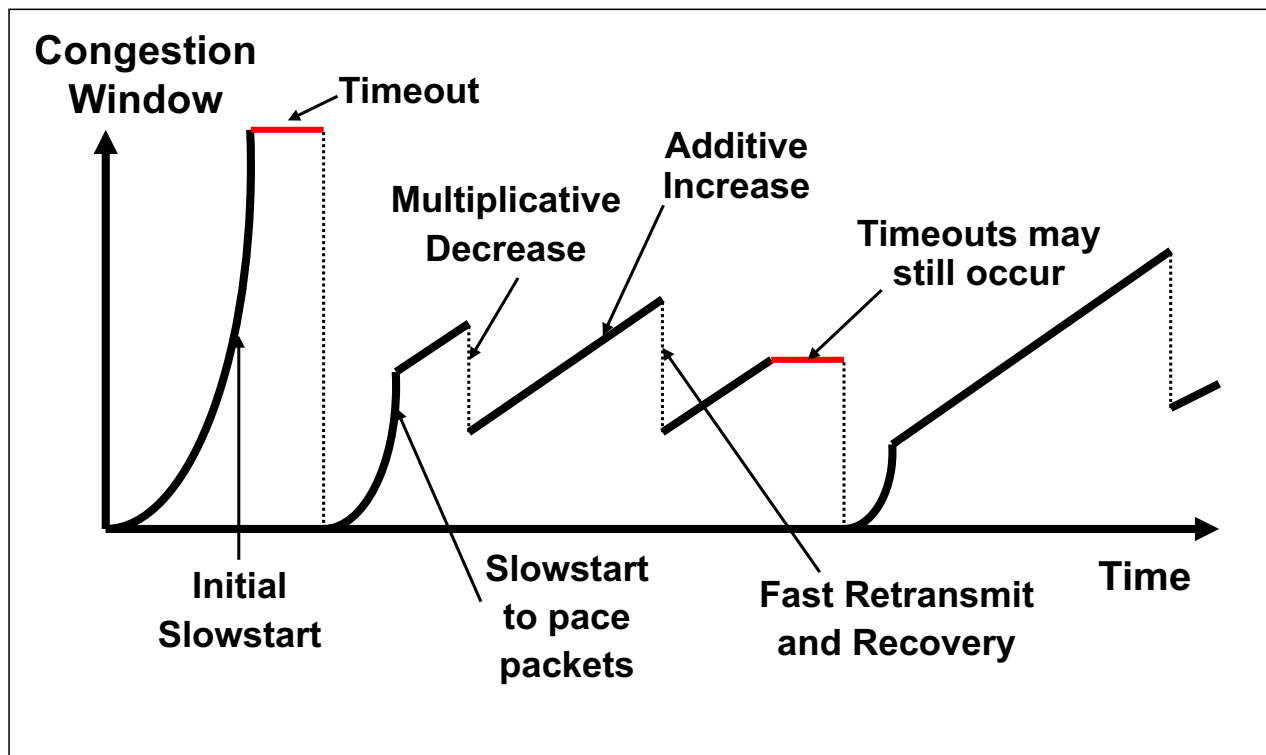
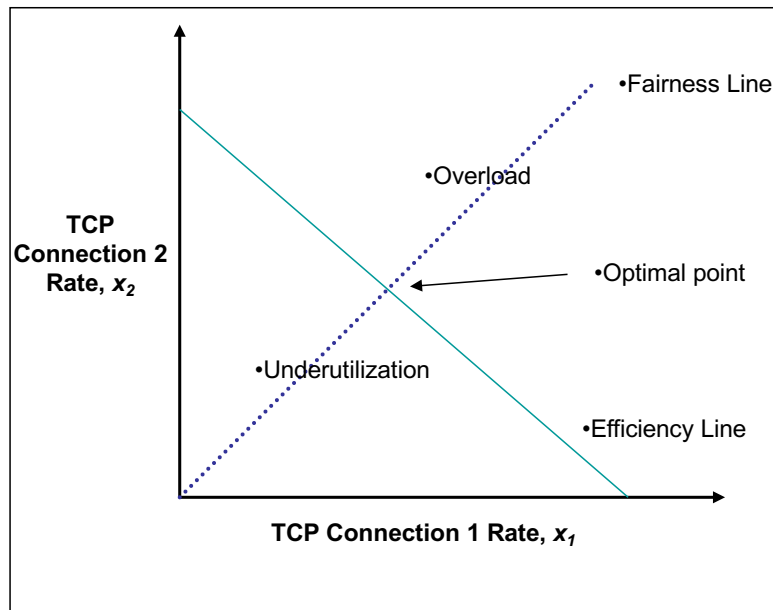


Diagram courtesy of Srinu Seshan

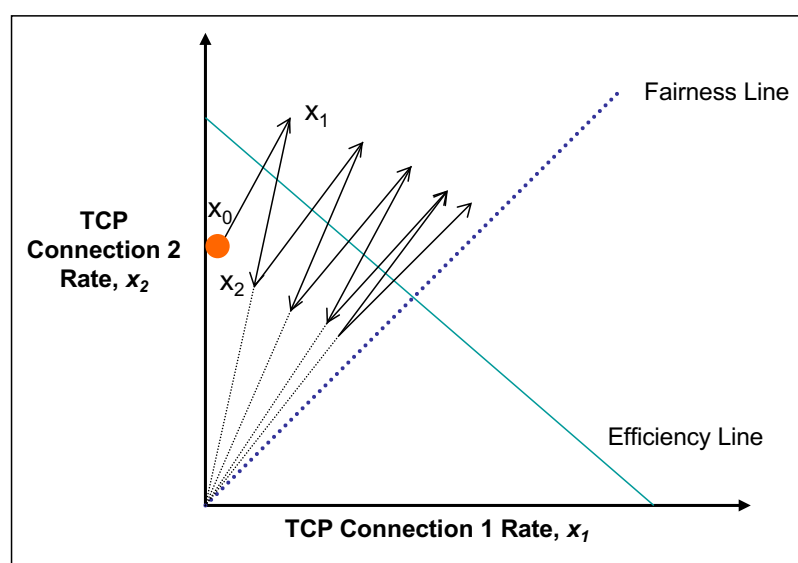
Variants of TCP

- TCP Reno & TCP Tahoe
 - Both handle Timeouts in the same way
 - TCP Tahoe handles Triple duplicate acks in the same way a Timeout
 - TCP Reno handles Triple duplicate acks differently (by cutting CW by half)

Visualizing behavior of competing TCP connections over time




Visualizing TCP's AIMD



Q: Is Additive Increase / Multiplicative Decrease fair?

Q: What if we had used MIMD? Or AIAD?

Congestion Avoidance

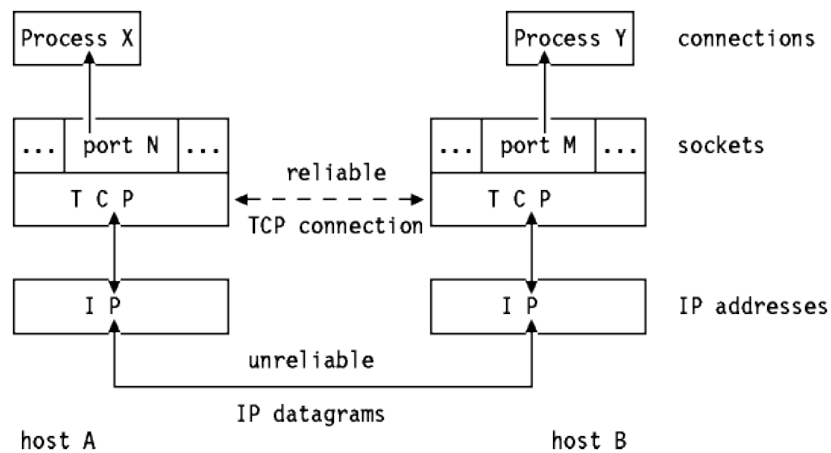
- Congestion control:
 - Cycle of actively probing, transmitting more than the network can handle, then backing off
- Congestion avoidance:
 - Back off before there are packet losses 
 - How can you tell that congestion is increasing?
 - Look at RTT – is it expanding?
 - Implicit: Random Early Dropping (RED) of packets by intermediate routers
 - Explicit: Intermediate routers indicate that there is congestion by setting a bit in the packet and receiver send that information back in the ACK (DECbit)

Summary of TCP

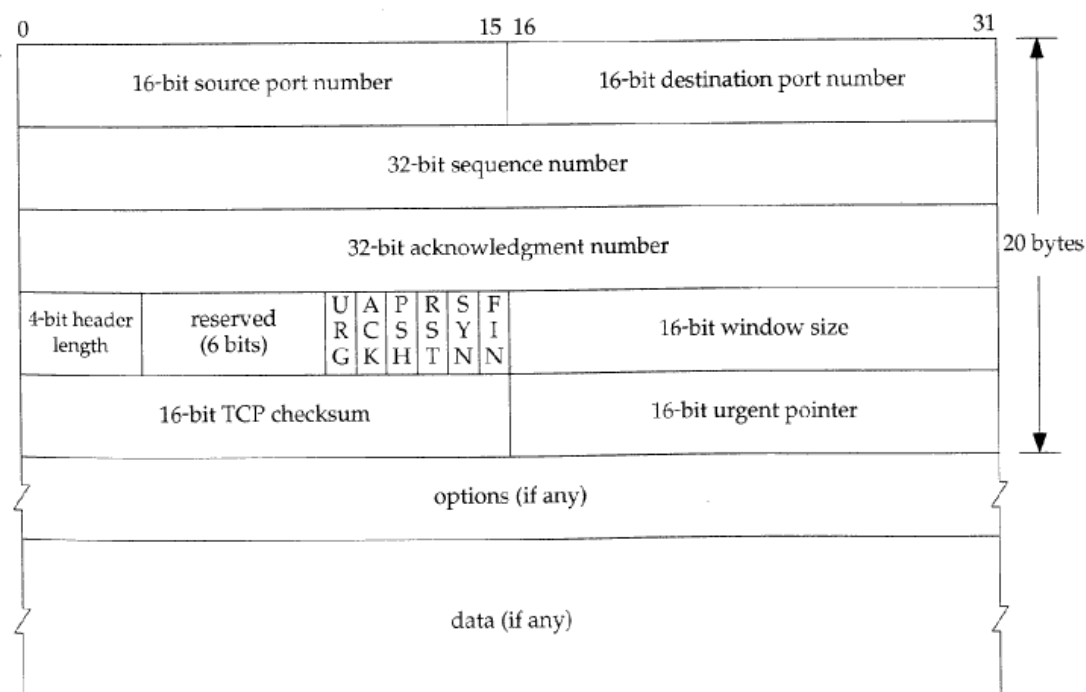
Transport Layer Services

- Connection-oriented communication
- Multiplexing
- Byte orientation
- In-order delivery
- Reliability: Error Detection & Re-transmissions
- Flow control
- Congestion control and congestion avoidance

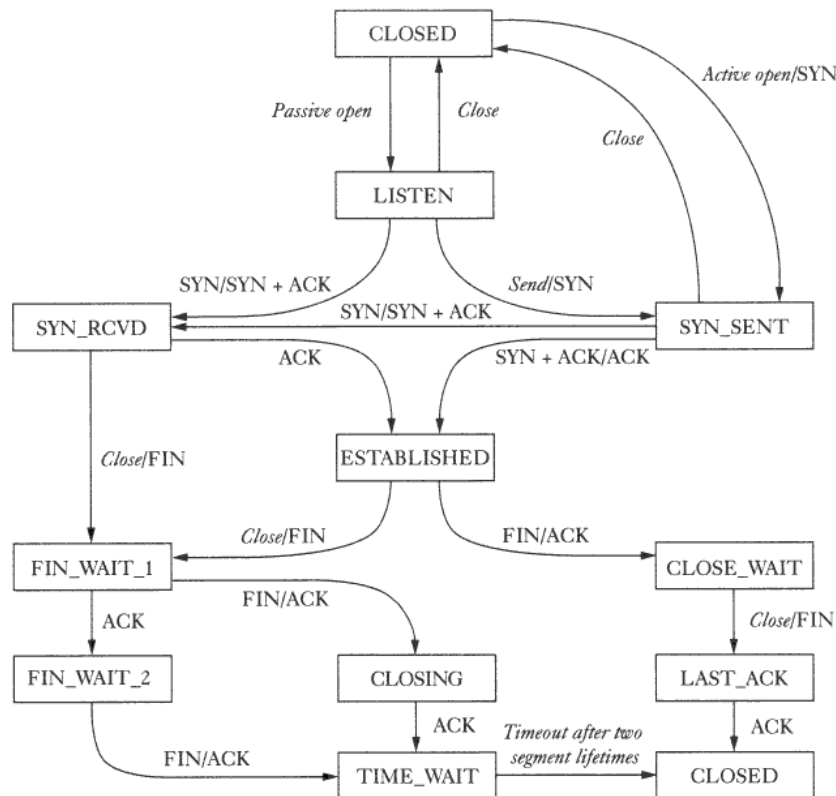
TCP Socket



TCP Header



TCP Finite State Machine Diagram



Conclusion

- Transport protocols
 - Multiplexing and demultiplexing
 - Sequence numbers
 - Window-based flow control
 - Timer-based retransmission
 - Checksum-based error detection
 - Congestion control
- TCP – provides end-to-end reliability
 - TCP also encompasses congestion control
 - AIMD: Additive Increase Multiplicative Decrease
- References for transport layer protocols
 - Chapter 6/7 of Peterson & Davie
 - Chapter 3 of Kurose & Ross

