15-744: Computer Networking

L-4 TCP

TCP Basics

- TCP reliability
- Congestion control basics
- TCP congestion control
- Assigned reading
  - [JK88] Congestion Avoidance and Control
  - [CJ89] Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks
  - [FF96] Simulation-based Comparisons of Tahoe, Reno, and SACK TCP
  - [FHPW00] Equation-Based Congestion Control for Unicast Applications

Key Things You Should Know Already

- Port numbers
- TCP/UDP checksum
- Sliding window flow control
  - Sequence numbers
- TCP connection setup

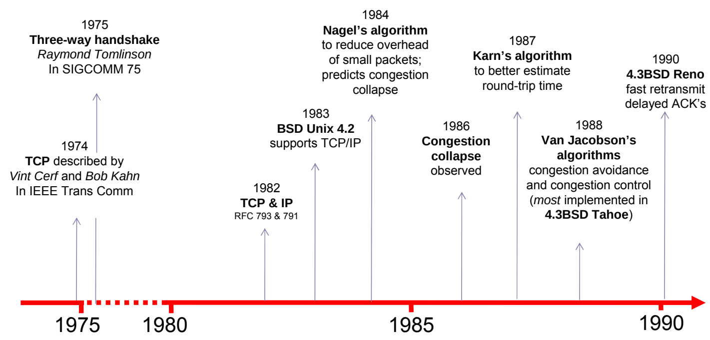
Overview

- TCP reliability: timer-driven
- TCP reliability: data-driven
- Congestion sources and collapse
- Congestion control basics
- TCP congestion control
- TCP modeling

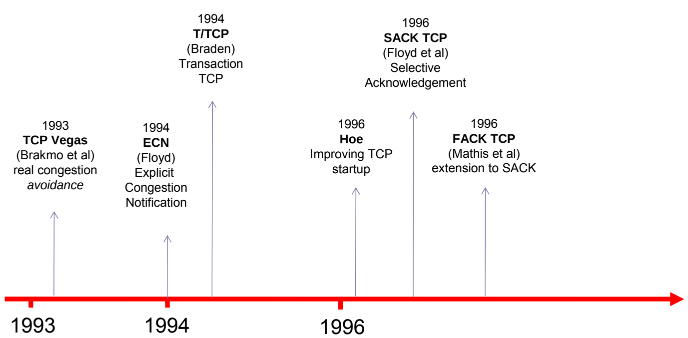
Introduction to TCP

- Communication abstraction:
  - Reliable
  - Ordered
  - Point-to-point
  - Byte-stream
  - Full duplex
  - Flow and congestion controlled
- Protocol implemented entirely at the ends
  - Fate sharing
- Sliding window with cumulative acks
  - Ack field contains last in-order packet received
  - Duplicate acks sent when out-of-order packet received

Evolution of TCP



TCP Through the 1990s



What's Different From Link Layers?

- Logical link vs. physical link
  - Must establish connection
- Variable RTT
  - May vary within a connection
- Reordering
  - How long can packets live → max segment lifetime
- Can't expect endpoints to exactly match link
  - Buffer space availability
- Transmission rate
  - Don't directly know transmission rate

Timeout-based Recovery

- Wait at least one RTT before retransmitting
- Importance of accurate RTT estimators:
  - Low RTT → unneeded retransmissions
  - High RTT → poor throughput
- RTT estimator must adapt to change in RTT
  - But not too fast, or too slow!
- Spurious timeouts
  - "Conservation of packets" principle – more than a window worth of packets in flight

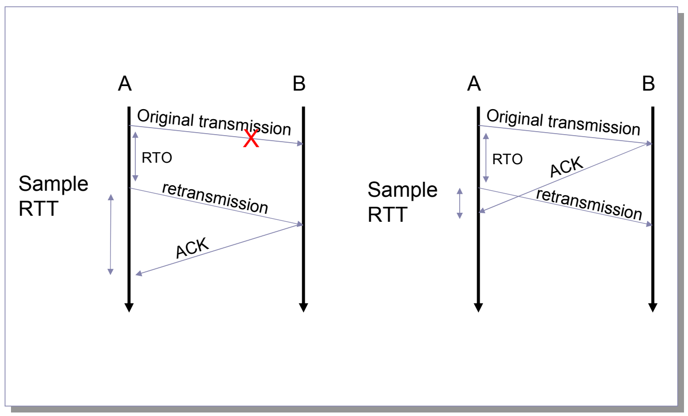
Initial Round-trip Estimator

- Round trip times exponentially averaged:
  - New RTT = α (old RTT) + (1 - α) (new sample)
  - Recommended value for α: 0.8 - 0.9
    - 0.875 for most TCP's
- Retransmit timer set to β RTT, where β = 2
  - Every time timer expires, RTO exponentially backed-off
  - Like Ethernet
- Not good at preventing spurious timeouts

Jacobson's Retransmission Timeout

- Key observation:
  - At high loads round trip variance is high
- Solution:
  - Base RTO on RTT and standard deviation or RRTT
  - rttvar = χ * dev + (1- χ)rttvar
    - dev = linear deviation
    - Inappropriately named – actually smoothed linear deviation

Retransmission Ambiguity



Karn's RTT Estimator

- Accounts for retransmission ambiguity
- If a segment has been retransmitted:
  - Don't count RTT sample on ACKs for this segment
  - Keep backed off time-out for next packet
  - Reuse RTT estimate only after one successful transmission

Timestamp Extension

- Used to improve timeout mechanism by more accurate measurement of RTT
- When sending a packet, insert current timestamp into option
  - 4 bytes for seconds, 4 bytes for microseconds
- Receiver echoes timestamp in ACK
  - Actually will echo whatever is in timestamp
- Removes retransmission ambiguity
  - Can get RTT sample on any packet

Timer Granularity

- Many TCP implementations set RTO in multiples of 200,500,1000ms
- Why?
  - Avoid spurious timeouts – RTTs can vary quickly due to cross traffic
  - Make timers interrupts efficient

Delayed ACKS

- Problem:
  - In request/response programs, you send separate ACK and Data packets for each transaction
- Solution:
  - Don't ACK data immediately
  - Wait 200ms (must be less than 500ms – why?)
  - Must ACK every other packet
  - Must not delay duplicate ACKs

Overview

- TCP reliability: timer-driven
- TCP reliability: data-driven
- Congestion sources and collapse
- Congestion control basics
- TCP congestion control
- TCP modeling

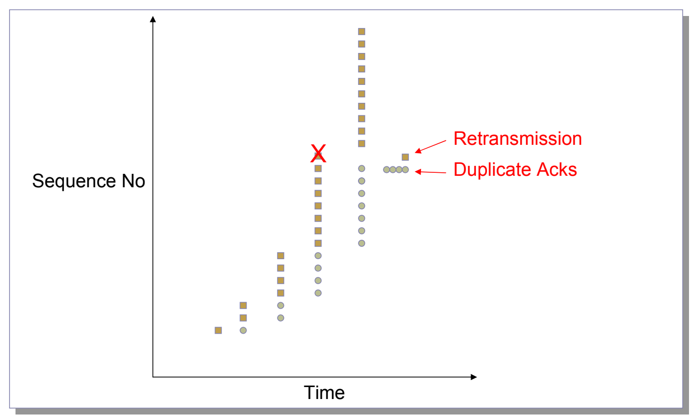
TCP Flavors

- Tahoe, Reno, Vegas → differ in data-driven reliability
- TCP Tahoe (distributed with 4.3BSD Unix)
  - Original implementation of Van Jacobson's mechanisms (VJ paper)
  - Includes:
    - Slow start
    - Congestion avoidance
    - Fast retransmit

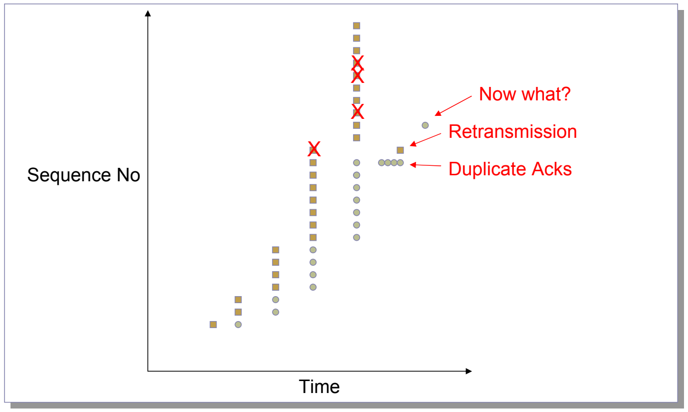
Fast Retransmit

- What are duplicate acks (dupacks)?
  - Repeated acks for the same sequence
- When can duplicate acks occur?
  - Loss
  - Packet re-ordering
  - Window update – advertisement of new flow control window
- Assume re-ordering is infrequent and not of large magnitude
  - Use receipt of 3 or more duplicate acks as indication of loss
  - Don't wait for timeout to retransmit packet

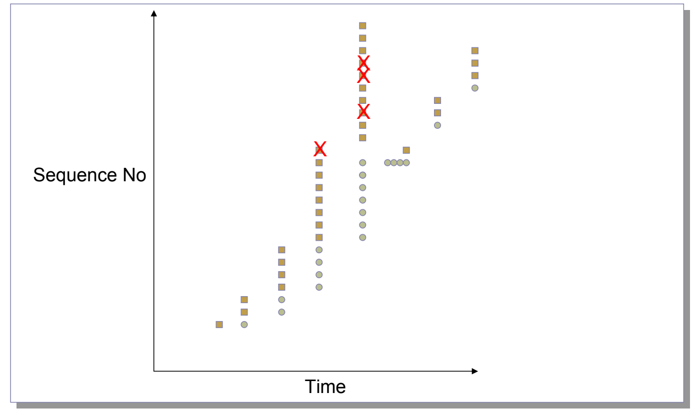
Fast Retransmit



Multiple Losses



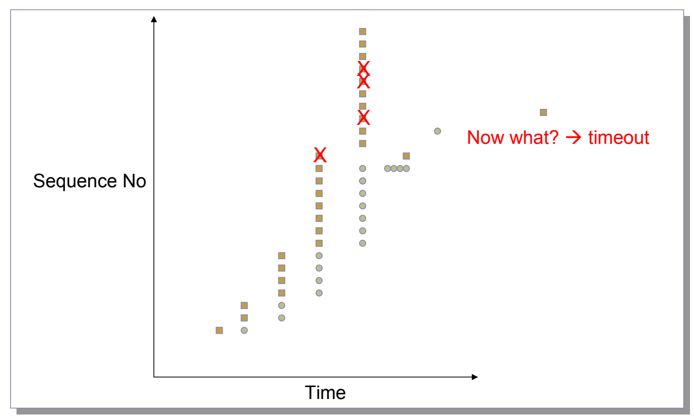
Tahoe



TCP Reno (1990)

- All mechanisms in Tahoe
- Addition of fast-recovery
  - Opening up congestion window after fast retransmit
- Delayed acks
- Header prediction
  - Implementation designed to improve performance
  - Has common case code inlined
- With multiple losses, Reno typically timeouts because it does not receive enough duplicate acknowledgements

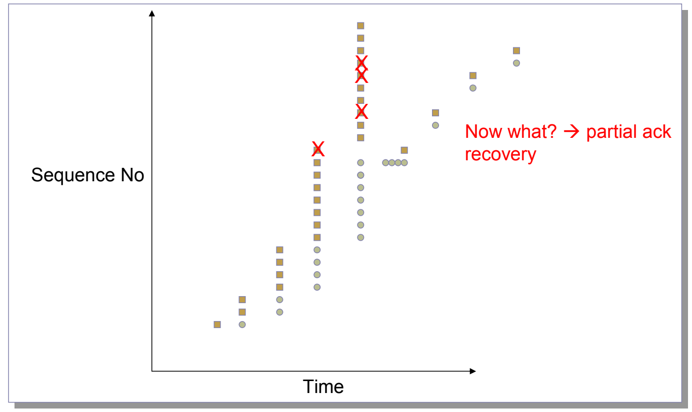
Reno



NewReno

- The ack that arrives after retransmission (partial ack) should indicate that a second loss occurred
- When does NewReno timeout?
  - When there are fewer than three dupacks for first loss
  - When partial ack is lost
- How fast does it recover losses?
  - One per RTT

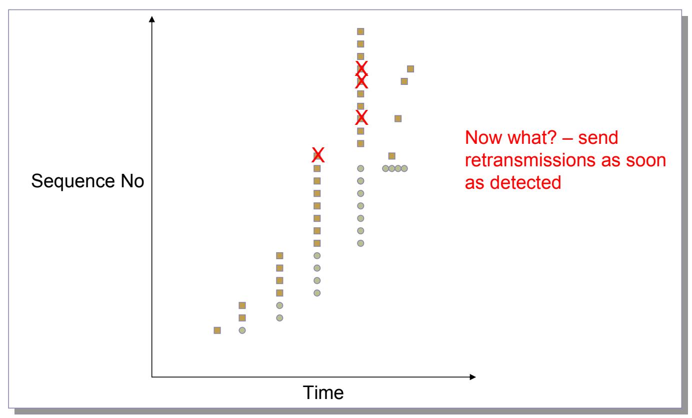
NewReno



SACK

- Basic problem is that cumulative acks provide little information
  - Ack for just the packet received
    - What if acks are lost? → carry cumulative also
    - Not used
  - Bitmask of packets received
    - Selective acknowledgement (SACK)
- How to deal with reordering

SACK



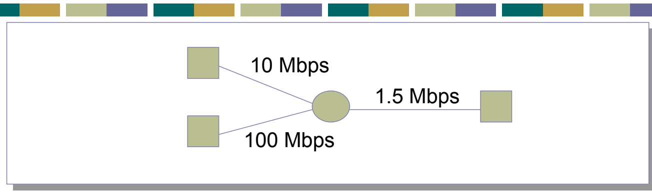
Performance Issues

- Timeout >> fast rexmit
  - Need 3 dupacks/sacks
  - Not great for small transfers
    - Don't have 3 packets outstanding
  - What are real loss patterns like?

Overview

- TCP reliability: timer-driven
- TCP reliability: data-driven
- Congestion sources and collapse
- Congestion control basics
- TCP congestion control
- TCP modeling

Congestion

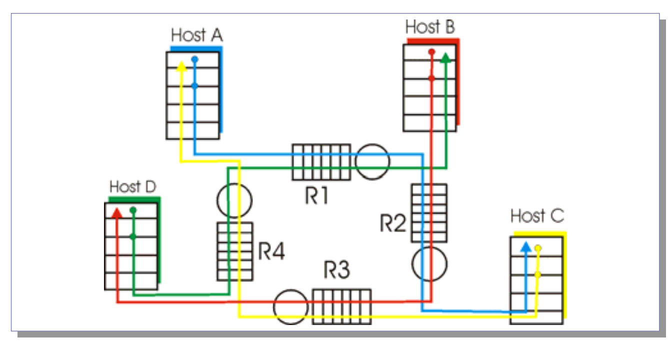


- Different sources compete for resources inside network
- Why is it a problem?
  - Sources are unaware of current state of resource
  - Sources are unaware of each other
  - In many situations will result in < 1.5 Mbps of throughput (congestion collapse)

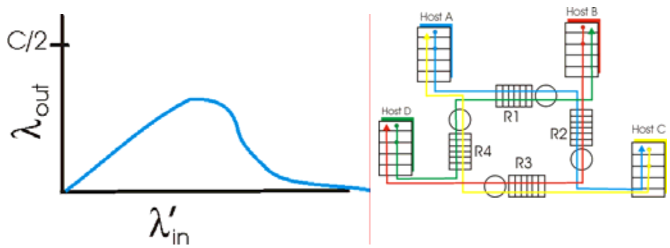
Causes & Costs of Congestion

- Four senders – multihop paths
- Timeout/retransmit

Q: What happens as rate increases?



Causes & Costs of Congestion



- When packet dropped, any "upstream transmission capacity used for that packet was wasted!

Congestion Collapse

- Definition: *Increase in network load results in decrease of useful work done*
- Many possible causes
  - Spurious retransmissions of packets still in flight
    - Classical congestion collapse
    - How can this happen with packet conservation
    - Solution: better timers and TCP congestion control
  - Undelivered packets
    - Packets consume resources and are dropped elsewhere in network
    - Solution: congestion control for ALL traffic

Other Congestion Collapse Causes

- Fragments
  - Mismatch of transmission and retransmission units
  - Solutions
    - Make network drop all fragments of a packet (early packet discard in ATM)
    - Do path MTU discovery
- Control traffic
  - Large percentage of traffic is for control
    - Headers, routing messages, DNS, etc.
- Stale or unwanted packets
  - Packets that are delayed on long queues
  - "Push" data that is never used

Where to Prevent Collapse?

- Can end hosts prevent problem?
  - Yes, but must trust end hosts to do right thing
  - E.g., sending host must adjust amount of data it puts in the network based on detected congestion
- Can routers prevent collapse?
  - No, not all forms of collapse
  - Doesn't mean they can't help
  - Sending accurate congestion signals
  - Isolating well-behaved from ill-behaved sources

Congestion Control and Avoidance

- A mechanism which:
  - Uses network resources efficiently
  - Preserves fair network resource allocation
  - Prevents or avoids collapse
- Congestion collapse is not just a theory
  - Has been frequently observed in many networks

Overview

- TCP reliability: timer-driven
- TCP reliability: data-driven
- Congestion sources and collapse
- Congestion control basics
- TCP congestion control
- TCP modeling

Objectives

- Simple router behavior
- Distributedness
- Efficiency: $X_{knee} = \Sigma x_i(t)$
- Fairness: $(\Sigma x_i)^2/n(\Sigma x_i^2)$
- Power: $(\text{throughput}^{\alpha}/\text{delay})$
- Convergence: control system must be stable

Basic Control Model

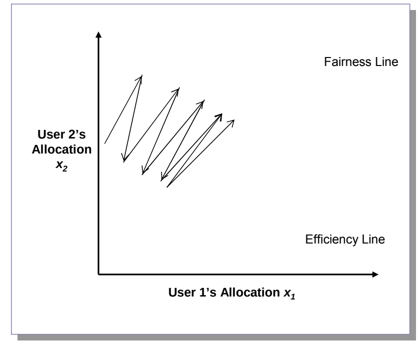
- Let's assume window-based control
- Reduce window when congestion is perceived
  - How is congestion signaled?
    - Either mark or drop packets
  - When is a router congested?
    - Drop tail queues – when queue is full
    - Average queue length – at some threshold
- Increase window otherwise
  - Probe for available bandwidth – how?

Linear Control

- Many different possibilities for reaction to congestion and probing
  - Examine simple linear controls
  - Window(t + 1) = a + b Window(t)
  - Different a_i/b_i for increase and a_d/b_d for decrease
- Supports various reaction to signals
  - Increase/decrease additively
  - Increased/decrease multiplicatively
  - Which of the four combinations is optimal?

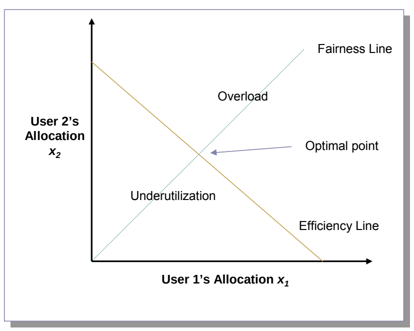
Phase plots

- Simple way to visualize behavior of competing connections over time



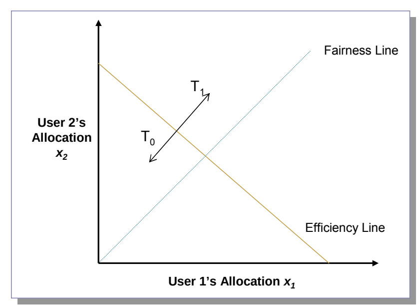
Phase plots

- What are desirable properties?
- What if flows are not equal?



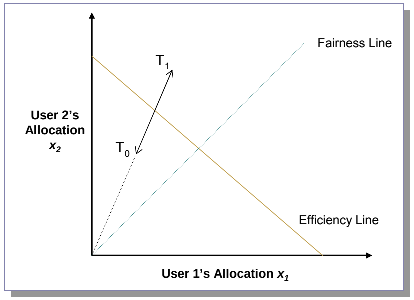
Additive Increase/Decrease

- Both X_1 and X_2 increase/decrease by the same amount over time
  - Additive increase improves fairness and additive decrease reduces fairness

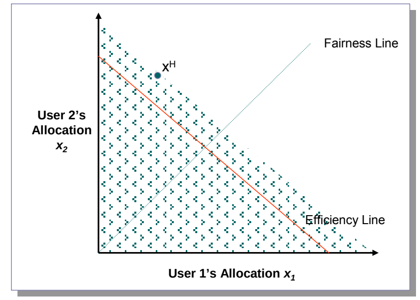


Multiplicative Increase/Decrease

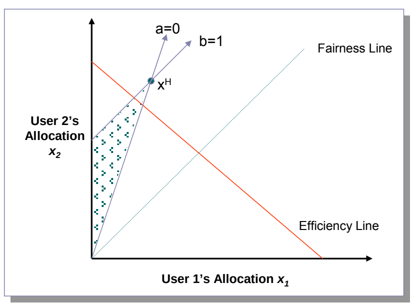
- Both X_1 and X_2 increase by the same factor over time
  - Extension from origin – constant fairness



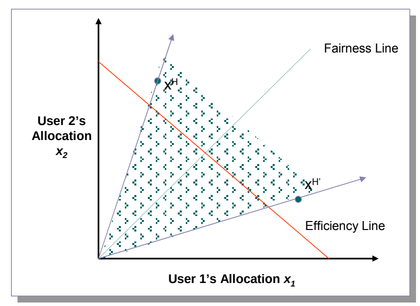
Convergence to Efficiency



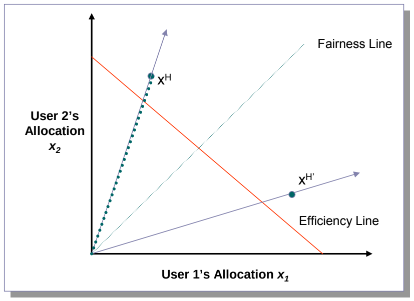
Distributed Convergence to Efficiency



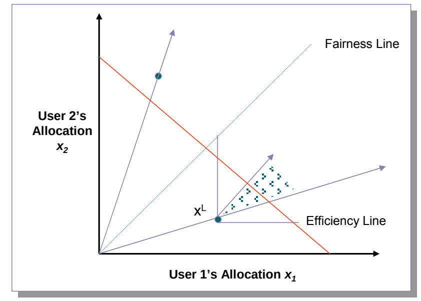
Convergence to Fairness



Convergence to Efficiency & Fairness



Increase

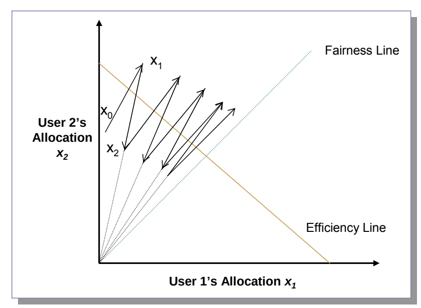


Constraints

- Distributed efficiency
  - I.e., Σ Window(t+1) > Σ Window(t) during increase
    - $a_i > 0$ & $b_i \geq 1$
    - Similarly, $a_d < 0$ & $b_d \leq 1$
- Must never decrease fairness
  - a & b's must be ≥ 0
  - $a_i/b_i > 0$ and $a_d/b_d \geq 0$
- Full constraints
  - $a_d = 0$, $0 \leq b_d < 1$, $a_i > 0$ and $b_i \geq 1$

What is the Right Choice?

- Constraints limit us to AIMD
  - Can have multiplicative term in increase (MAIMD)
  - AIMD moves towards optimal point



Overview

- TCP reliability: timer-driven
- TCP reliability: data-driven
- Congestion sources and collapse
- Congestion control basics
- TCP congestion control
- TCP modeling

TCP Congestion Control

- Motivated by ARPANET congestion collapse
- Underlying design principle: packet conservation
  - At equilibrium, inject packet into network only when one is removed
  - Basis for stability of physical systems
- Why was this not working?
  - Connection doesn't reach equilibrium
  - Spurious retransmissions
  - Resource limitations prevent equilibrium

TCP Congestion Control - Solutions

- Reaching equilibrium
  - Slow start
- Eliminates spurious retransmissions
  - Accurate RTO estimation
  - Fast retransmit
- Adapting to resource availability
  - Congestion avoidance

TCP Congestion Control

- Changes to TCP motivated by ARPANET congestion collapse
- Basic principles
  - AIMD
  - Packet conservation
  - Reaching steady state quickly
  - ACK clocking

AIMD

- Distributed, fair and efficient
- Packet loss is seen as sign of congestion and results in a multiplicative rate decrease
  - Factor of 2
- TCP periodically probes for available bandwidth by increasing its rate

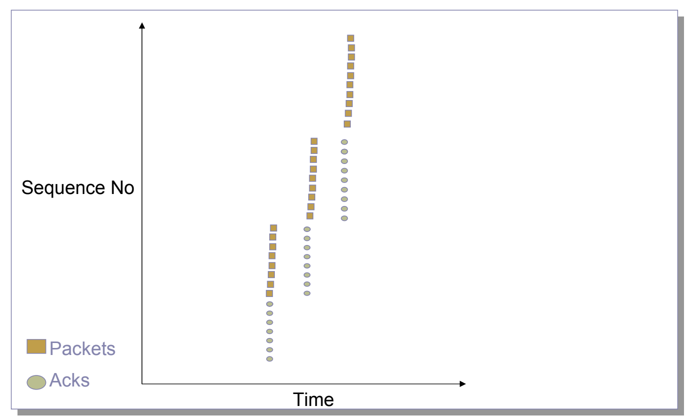
Implementation Issue

- Operating system timers are very coarse – how to pace packets out smoothly?
- Implemented using a congestion window that limits how much data can be in the network.
  - TCP also keeps track of how much data is in transit
- Data can only be sent when the amount of outstanding data is less than the congestion window.
  - The amount of outstanding data is increased on a "send" and decreased on "ack"
  - (last sent – last acked) < congestion window
- Window limited by both congestion and buffering

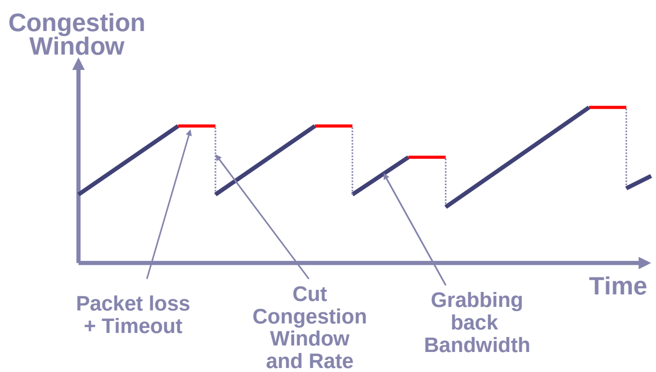
Congestion Avoidance

- If loss occurs when cwnd = W
  - Network can handle 0.5W ~ W segments
  - Set cwnd to 0.5W (multiplicative decrease)
- Upon receiving ACK
  - Increase cwnd by (1 packet)/cwnd
    - What is 1 packet? → 1 MSS worth of bytes
    - After cwnd packets have passed by → approximately increase of 1 MSS
- Implements AIMD

Congestion Avoidance Sequence Plot



Congestion Avoidance Behavior

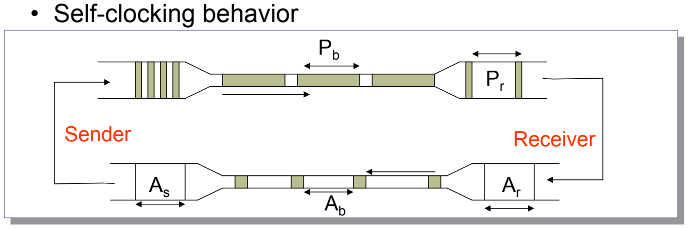


Packet Conservation

- At equilibrium, inject packet into network only when one is removed
  - Sliding window and not rate controlled
  - But still need to avoid sending burst of packets → would overflow links
    - Need to carefully pace out packets
    - Helps provide stability
- Need to eliminate spurious retransmissions
  - Accurate RTO estimation
  - Better loss recovery techniques (e.g. fast retransmit)

TCP Packet Pacing

- Congestion window helps to “pace” the transmission of data packets
- In steady state, a packet is sent when an ack is received
  - Data transmission remains smooth, once it is smooth
  - Self-clocking behavior

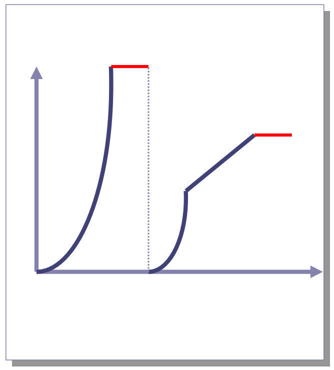


Reaching Steady State

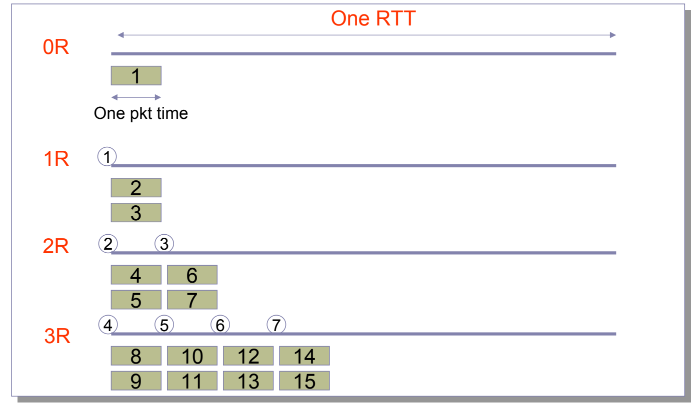
- Doing AIMD is fine in steady state but slow…
- How does TCP know what is a good initial rate to start with?
  - Should work both for a CDPD (10s of Kbps or less) and for supercomputer links (10 Gbps and growing)
- Quick initial phase to help get up to speed (slow start)

Slow Start Packet Pacing

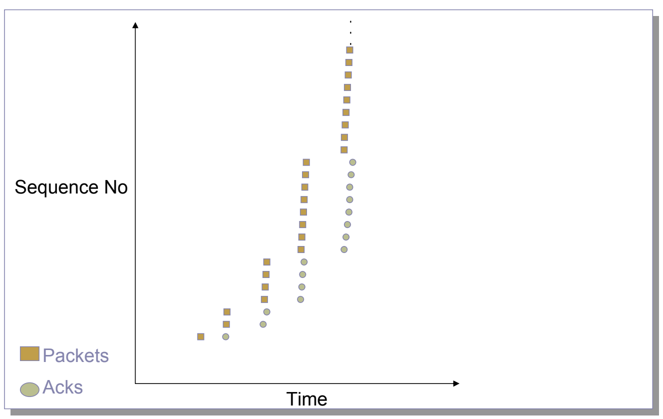
- How do we get this clocking behavior to start?
  - Initialize cwnd = 1
  - Upon receipt of every ack, cwnd = cwnd + 1
- Implications
  - Window actually increases to W in RTT * $\log_2(W)$
  - Can overshoot window and cause packet loss



Slow Start Example



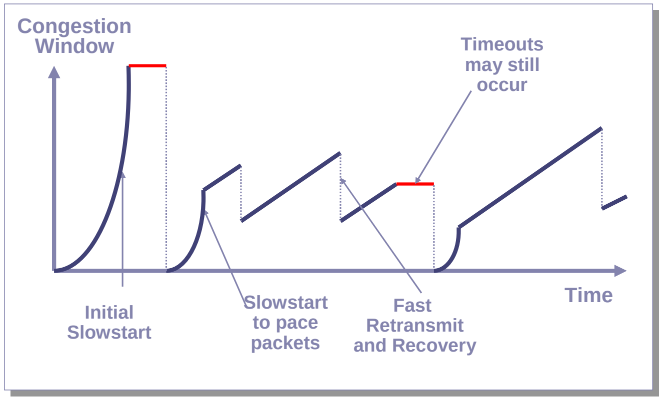
Slow Start Sequence Plot



Return to Slow Start

- If packet is lost we lose our self clocking as well
  - Need to implement slow-start and congestion avoidance together
- When timeout occurs set ssthresh to 0.5w
  - If cwnd < ssthresh, use slow start
  - Else use congestion avoidance

TCP Saw Tooth Behavior



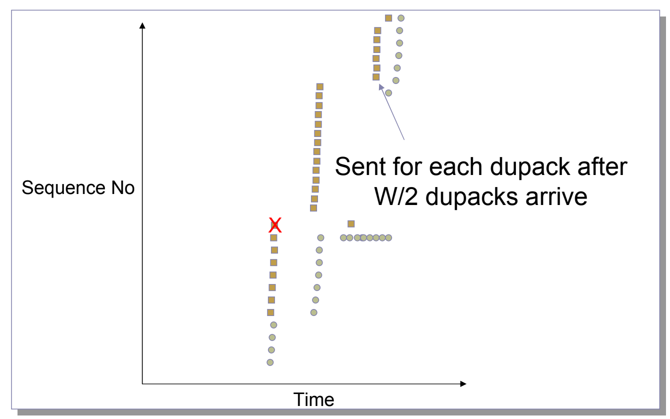
How to Change Window

- When a loss occurs have W packets outstanding
- New cwnd = 0.5 * cwnd
  - How to get to new state?

Fast Recovery

- Each duplicate ack notifies sender that single packet has cleared network
- When < cwnd packets are outstanding
  - Allow new packets out with each new duplicate acknowledgement
- Behavior
  - Sender is idle for some time – waiting for ½ cwnd worth of dupacks
  - Transmits at original rate after wait
    - Ack clocking rate is same as before loss

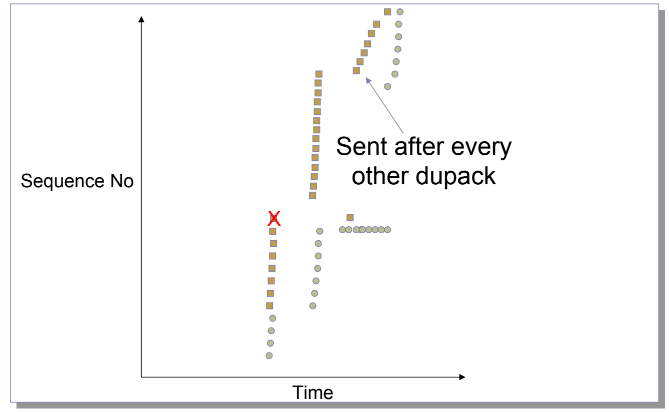
Fast Recovery



NewReno Changes

- Send a new packet out for each pair of dupacks
  - Adapt more gradually to new window
- Will not halve congestion window again until recovery is completed
  - Identifies congestion events vs. congestion signals
- Initial estimation for ssthresh

Rate Halving Recovery



Delayed Ack Impact

- TCP congestion control triggered by acks
  - If receive half as many acks → window grows half as fast
- Slow start with window = 1
  - Will trigger delayed ack timer
  - First exchange will take at least 200ms
  - Start with > 1 initial window
    - Bug in BSD, now a “feature”/standard

Overview

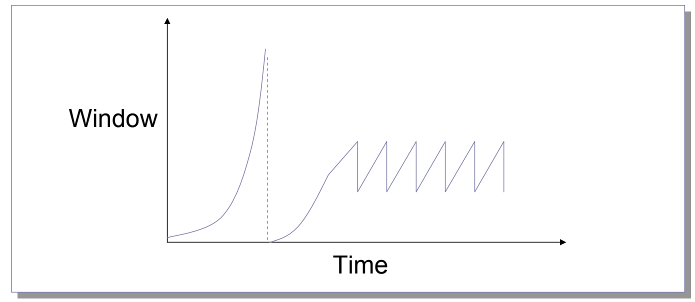
- TCP reliability: timer-driven
- TCP reliability: data-driven
- Congestion sources and collapse
- Congestion control basics
- TCP congestion control
- TCP modeling

TCP Modeling

- Given the congestion behavior of TCP can we predict what type of performance we should get?
- What are the important factors
  - Loss rate
    - Affects how often window is reduced
  - RTT
    - Affects increase rate and relates BW to window
  - RTO
    - Affects performance during loss recovery
  - MSS
    - Affects increase rate

Overall TCP Behavior

- Let's concentrate on steady state behavior with no timeouts and perfect loss recovery



Simple TCP Model

- Some additional assumptions
  - Fixed RTT
  - No delayed ACKs
- In steady state, TCP losses packet each time window reaches W packets
  - Window drops to W/2 packets
  - Each RTT window increases by 1 packet→W/2 * RTT before next loss
  - BW = MSS * avg window/RTT = MSS * (W + W/2)/(2 * RTT) = .75 * MSS * W / RTT

Simple Loss Model

- What was the loss rate?
  - Packets transferred = (.75 W/RTT) * (W/2 * RTT) = $3W^2/8$
  - 1 packet lost → loss rate = p = $8/3W^2$
  - W = sqrt(8 / (3 * loss rate))
- BW = .75 * MSS * W / RTT
  - BW = MSS / (RTT * sqrt (2/3p))

TCP Friendliness

- What does it mean to be TCP friendly?
  - TCP is not going away
  - Any new congestion control must compete with TCP flows
    - Should not clobber TCP flows and grab bulk of link
    - Should also be able to hold its own, i.e. grab its fair share, or it will never become popular
- How is this quantified/shown?
  - Has evolved into evaluating loss/throughput behavior
  - If it shows 1/sqrt(p) behavior it is ok
  - But is this really true?

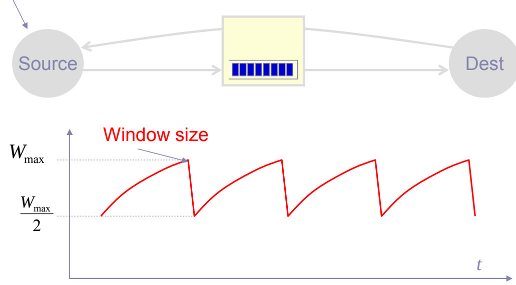
TCP Performance

- Can TCP saturate a link?
- Congestion control
  - Increase utilization until… link becomes congested
  - React by decreasing window by 50%
  - Window is proportional to rate * RTT
- Doesn't this mean that the network oscillates between 50 and 100% utilization?
  - Average utilization = 75%??
  - No…this is *not* right!

TCP Congestion Control

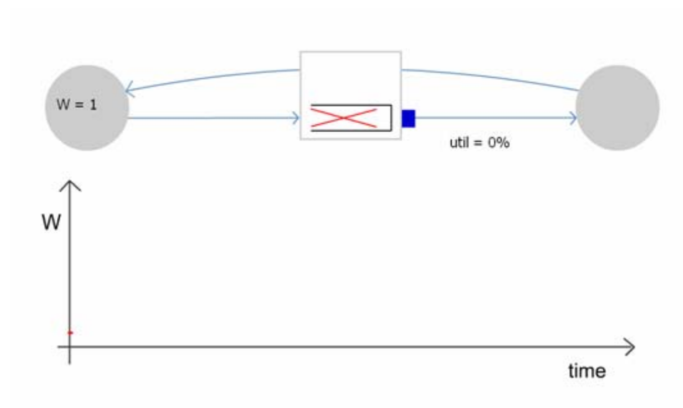
Rule for adjusting W
- If an ACK is received: W ← W+1/W
- If a packet is lost: W ← W/2

Only W packets may be outstanding

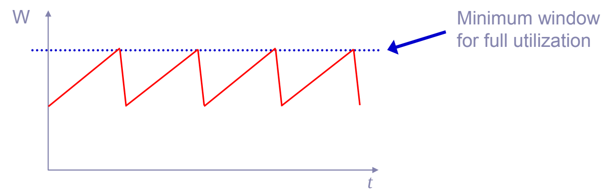


Single TCP Flow

Router without buffers



Summary Unbuffered Link



- The router can't fully utilize the link
  - If the window is too small, link is not full
  - If the link is full, next window increase causes drop
  - With no buffer it still achieves 75% utilization

TCP Performance

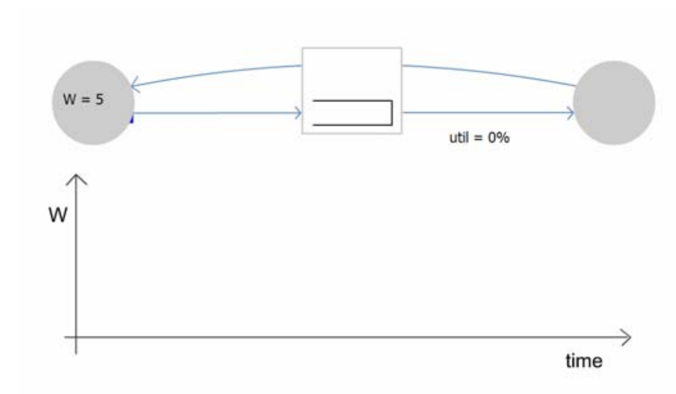
- In the real world, router queues play important role
  - Window is proportional to rate * RTT
    - But, RTT changes as well the window
  - Window to fill links = propagation RTT * bottleneck bandwidth
    - If window is larger, packets sit in queue on bottleneck link

TCP Performance

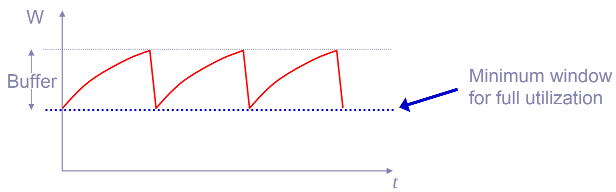
- If we have a large router queue → can get 100% utilization
  - But, router queues can cause large delays
- How big does the queue need to be?
  - Windows vary from W → W/2
    - Must make sure that link is always full
    - W/2 > RTT * BW
    - W = RTT * BW + Qsize
    - Therefore, Qsize > RTT * BW
  - Ensures 100% utilization
  - Delay?
    - Varies between RTT and 2 * RTT

Single TCP Flow

Router with large enough buffers for full link utilization



Summary Buffered Link



- With sufficient buffering we achieve full link utilization
  - The window is always above the critical threshold
  - Buffer absorbs changes in window size
    - Buffer Size = Height of TCP Sawtooth
    - Minimum buffer size needed is 2T*C
  - This is the origin of the rule-of-thumb

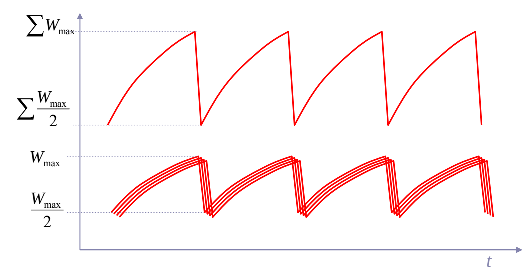
Example

- 10Gb/s linecard
  - Requires 300Mbytes of buffering.
  - Read and write 40 byte packet every 32ns.
- Memory technologies
  - DRAM: require 4 devices, but too slow.
  - SRAM: require 80 devices, 1kW, $2000.
- Problem gets harder at 40Gb/s
  - Hence RLDRAM, FCRAM, etc.

Rule-of-thumb

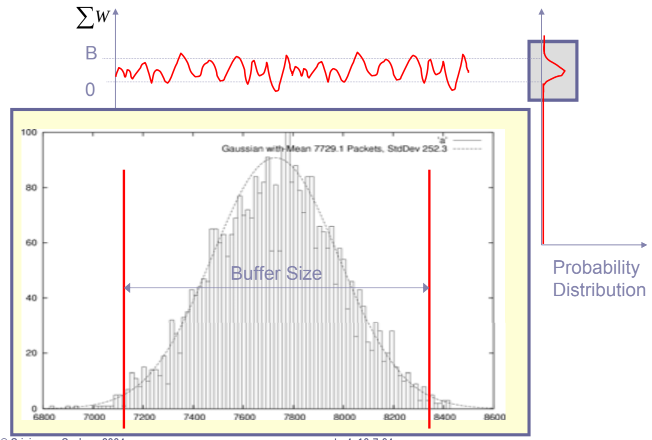
- Rule-of-thumb makes sense for one flow
- Typical backbone link has > 20,000 flows
- Does the rule-of-thumb still hold?

If flows are synchronized



- Aggregate window has same dynamics
- Therefore buffer occupancy has same dynamics
- Rule-of-thumb still holds.

If flows are not synchronized

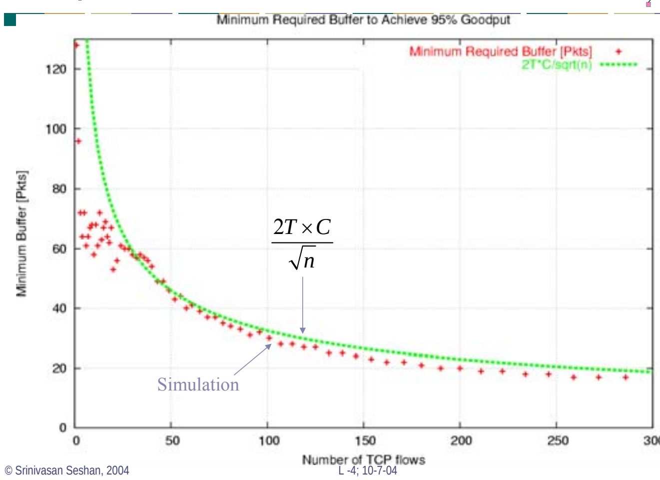


Central Limit Theorem

- CLT tells us that the more variables (Congestion Windows of Flows) we have, the narrower the Gaussian (Fluctuation of sum of windows)
  - Width of Gaussian decreases with $\frac{1}{\sqrt{n}}$
  - Buffer size should also decreases with $\frac{1}{\sqrt{n}}$

$$B \rightarrow \frac{B_{n=1}}{\sqrt{n}} = \frac{2T \times C}{\sqrt{n}}$$

Required buffer size



Important Lessons

- How does TCP implement AIMD?
  - Sliding window, slow start & ack clocking
  - How to maintain ack clocking during loss recovery → fast recovery
- Modern TCP loss recovery
  - Why are timeouts bad?
  - How to avoid them? → fast retransmit, SACK
- How does TCP fully utilize a link?
  - Role of router buffers

Next Lecture

- TCP Vegas/alternative congestion control schemes
- RED
- Fair queuing
- Core-stateless fair queuing/XCP
- Assigned reading
  - [BP95] TCP Vegas: End to End Congestion Avoidance on a Global Internet
  - [FJ93] Random Early Detection Gateways for Congestion Avoidance
  - [DKS90] Analysis and Simulation of a Fair Queueing Algorithm, Internetworking: Research and Experience
  - [SSZ98] Core-Stateless Fair Queueing: Achieving Approximately Fair Allocations in High Speed Networks
  - [KHR02] Congestion Control for High Bandwidth-Delay Product Networks

EXTRA SLIDES

The rest of the slides are FYI

Integrity & Demultiplexing

- Port numbers
  - Demultiplex from/to process
  - Servers wait on well known ports (/etc/services)
- Checksum
  - Is it sufficient to just checksum the packet contents?
  - No, need to ensure correct source/destination
    - Pseudoheader – portion of IP hdr that are critical
    - Checksum covers Pseudoheader, transport hdr, and packet body
- UDP provides just integrity and demux

TCP Header

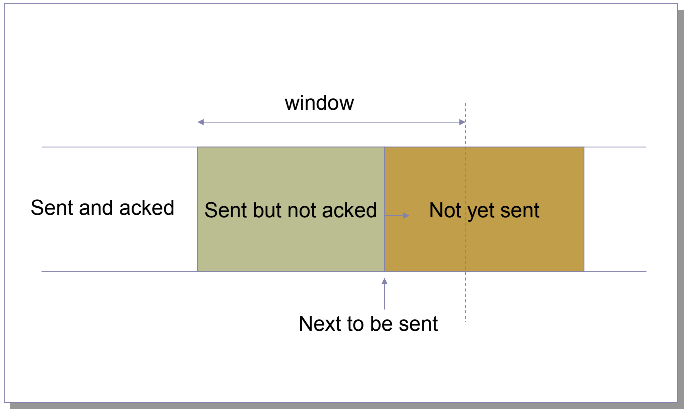
Flags: SYN
FIN
RESET
PUSH
URG
ACK

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

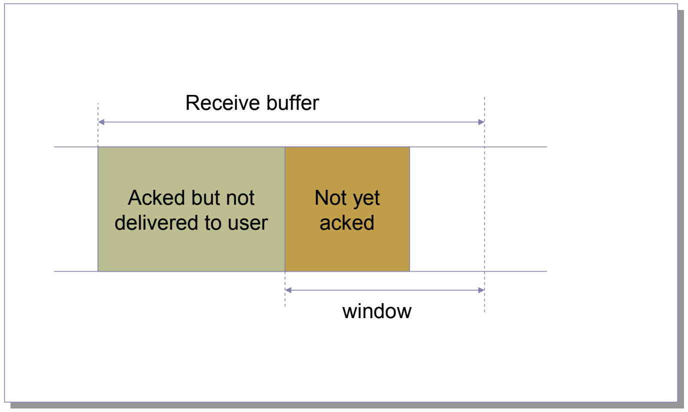
TCP Flow Control

- TCP is a sliding window protocol
  - For window size *n*, can send up to *n* bytes without receiving an acknowledgement
  - When the data is acknowledged then the window slides forward
- Each packet advertises a window size
  - Indicates number of bytes the receiver has space for
- Original TCP always sent entire window
  - Congestion control now limits this

Window Flow Control: Send Side



Window Flow Control: Receive Side

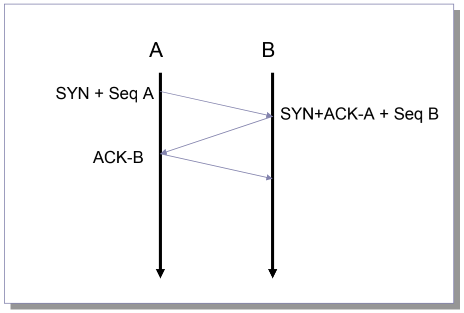


TCP Persist

- What happens if window is 0?
  - Receiver updates window when application reads data
  - What if this update is lost?
- TCP Persist state
  - Sender periodically sends 1 byte packets
  - Receiver responds with ACK even if it can't store the packet

Connection Establishment

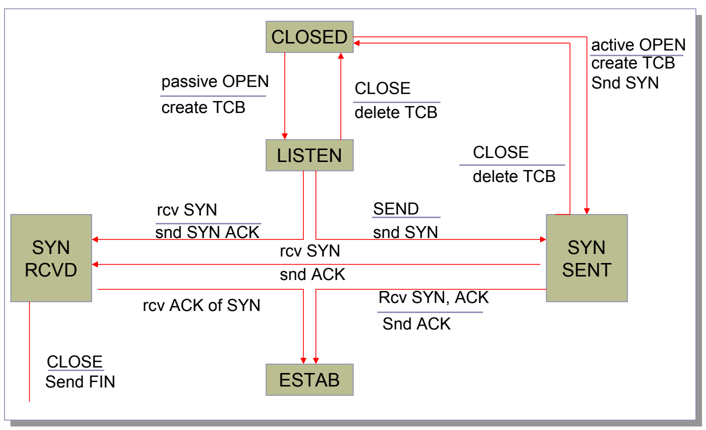
- A and B must agree on initial sequence number selection
  - Use 3-way handshake



Sequence Number Selection

- Why not simply chose 0?
- Must avoid overlap with earlier incarnation

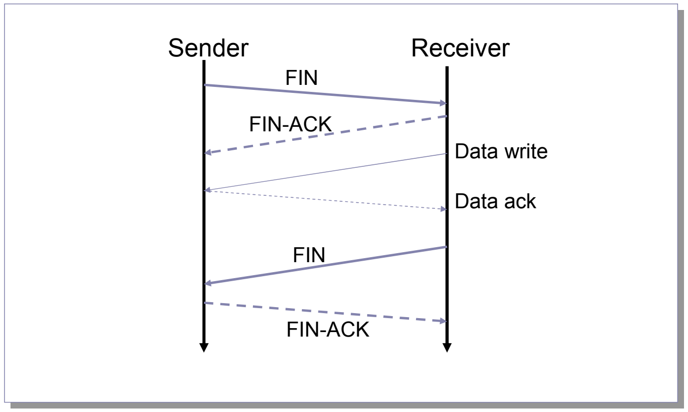
Connection Setup



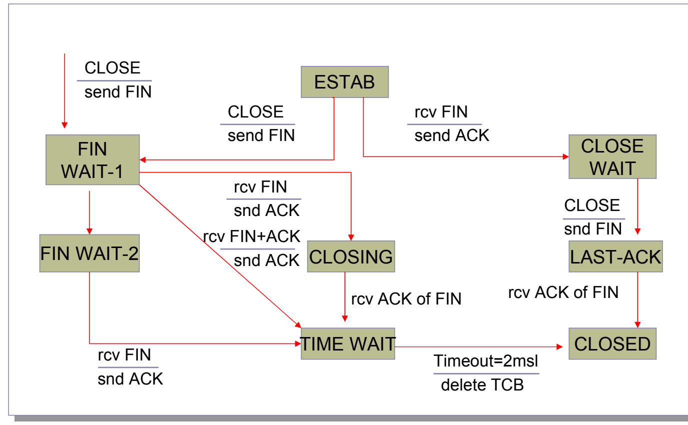
Connection Tear-down

- Normal termination
  - Allow unilateral close
- TCP must continue to receive data even after closing
- Cannot close connection immediately
  - What if a new connection restarts and uses same sequence number?

Tear-down Packet Exchange



Connection Tear-down



Detecting Half-open Connections

	TCP A		TCP B
1.	(CRASH)		(send 300, receive 100)
2.	CLOSED		ESTABLISHED
3.	SYN-SENT → <SEQ=400><CTL=SYN>	→	(??)
4.	(!!) ← <SEQ=300><ACK=100><CTL=ACK>	←	ESTABLISHED
5.	SYN-SENT → <SEQ=100><CTL=RST>	→	(Abort!!)
6.	SYN-SENT		CLOSED
7.	SYN-SENT → <SEQ=400><CTL=SYN>	→	

Observed TCP Problems

- Too many small packets
  - Silly window syndrome
  - Nagel's algorithm
- Initial sequence number selection
- Amount of state maintained

Silly Window Syndrome

- Problem: (Clark, 1982)
  - If receiver advertises small increases in the receive window then the sender may waste time sending lots of small packets
- Solution
  - Receiver must not advertise small window increases
  - Increase window by min(MSS,RecvBuffer/2)

Nagel's Algorithm

- Small packet problem:
  - Don't want to send a 41 byte packet for each keystroke
  - How long to wait for more data?
- Solution:
  - Allow only one outstanding small (not full sized) segment that has not yet been acknowledged

Why is Selecting ISN Important?

- Suppose machine X selects ISN based on predictable sequence
- Fred has .rhosts to allow login to X from Y
- Evil Ed attacks
  - Disables host Y – denial of service attack
  - Make a bunch of connections to host X
  - Determine ISN pattern a guess next ISN
  - Fake pkt1: [<src Y><dst X>, guessed ISN]
  - Fake pkt2: desired command

Time Wait Issues

- Web servers not clients close connection first
  - Established → Fin-Waits → Time-Wait → Closed
  - Why would this be a problem?
- Time-Wait state lasts for 2 * MSL
  - MSL is should be 120 seconds (is often 60s)
  - Servers often have order of magnitude more connections in Time-Wait

TCP Extensions

- Implemented using TCP options
  - Timestamp
  - Protection from sequence number wraparound
  - Large windows

Protection From Wraparound

- Wraparound time vs. Link speed
  - 1.5Mbps: 6.4 hours
  - 10Mbps: 57 minutes
  - 45Mbps: 13 minutes
  - 100Mbps: 6 minutes
  - 622Mbps: 55 seconds → < MSL!
  - 1.2Gbps: 28 seconds
- Use timestamp to distinguish sequence number wraparound

Large Windows

- Delay-bandwidth product for 100ms delay
  - 1.5Mbps: 18KB
  - 10Mbps: 122KB > max 16bit window
  - 45Mbps: 549KB
  - 100Mbps: 1.2MB
  - 622Mbps: 7.4MB
  - 1.2Gbps: 14.8MB
- Scaling factor on advertised window
  - Specifies how many bits window must be shifted to the left
  - Scaling factor exchanged during connection setup

Maximum Segment Size (MSS)

- Exchanged at connection setup
  - Typically pick MTU of local link
- What all does this effect?
  - Efficiency
  - Congestion control
  - Retransmission
- Path MTU discovery
  - Why should MTU match MSS?