P561: Network Systems
Week 5: Transport #1

Tom Anderson
Ratul Mahajan

TA: Colin Dixon

Administrivia

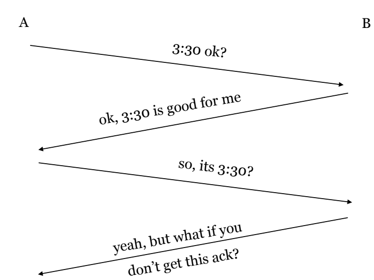
Homework #2
- Due next week (week 6), start of class
- Catalyst turnin

Fishnet Assignment #3
- Due week 7, start of class

Homework #1: General's Paradox

Can we use messages and retries to synchronize two machines so they are guaranteed to do some operation at the same time?
- No. Why?

General's Paradox Illustrated



Consensus revisited

If distributed consensus is impossible, what then?

1. TCP: can agree that destination received data
2. Distributed transactions (2 phase commit)
   - Can agree to eventually do some operation
3. Paxos: non-blocking transactions
   - Always safe, progress if no failures

Transport Challenge

IP: routers can be arbitrarily bad
- packets can be lost, reordered, duplicated, have limited size & can be fragmented

TCP: applications need something better
- reliable delivery, in order delivery, no duplicates, arbitrarily long streams of data, match sender/receiver speed, process-to-process

Reliable Transmission

How do we send packets reliably?

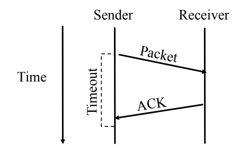
Two mechanisms

- Acknowledgements
- Timeouts

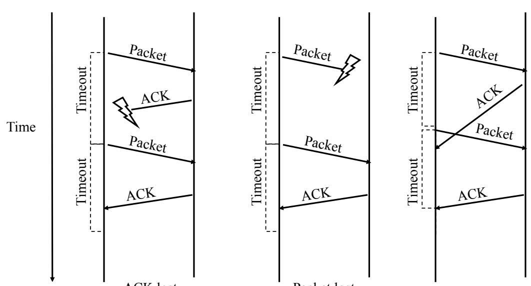
Simplest reliable protocol: Stop and Wait

Stop and Wait

- Send a packet, wait until ack arrives
  - retransmit if no ack within timeout
- Receiver acks each packet as it arrives



Recovering from error



ACK lost | Packet lost | Early timeout

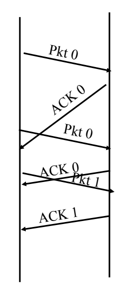
How can we recognize resends?

Use unique ID for each pkt

- for both packets and acks

How many bits for the ID?

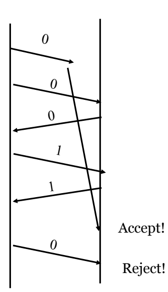
- For stop and wait, a single bit!
- assuming in-order delivery…



What if packets can be delayed?

Solutions?

- Never reuse an ID?
- Change IP layer to eliminate packet reordering?
- Prevent very late delivery?
  - IP routers keep hop count per pkt, discard if exceeded
  - ID's not reused within delay bound
- TCP won't work without some bound on how late packets can arrive!



What happens on reboot?

How do we distinguish packets sent before and after reboot?

- Can't remember last sequence # used unless written to stable storage (disk or NVRAM)

Solutions?

- Restart sequence # at 0?
- Assume/force boot to take max packet delay?
- Include epoch number in packet (stored on disk)?
- Ask other side what the last sequence # was?
- TCP sidesteps this problem with random initial seq # (in each direction)

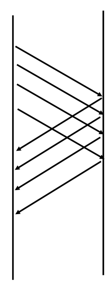
How do we keep the pipe full?

Unless the bandwidth*delay product is small, stop and wait can't fill pipe

Solution: Send multiple packets without waiting for first to be acked

Reliable, unordered delivery:
- Send new packet after each ack
- Sender keeps list of unack'ed packets; resends after timeout
- Receiver same as stop&wait

How easy is it to write apps that handle out of order delivery?
- How easy is it to test those apps?



Sliding Window: Reliable, ordered delivery

Two constraints:
- Receiver can't deliver packet to application until all prior packets have arrived
- Sender must prevent buffer overflow at receiver

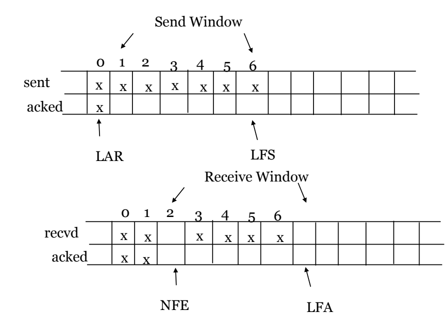
Solution: sliding window
- circular buffer at sender and receiver
  - packets in transit <= buffer size
  - advance when sender and receiver agree packets at beginning have been received
- How big should the window be?
  - bandwidth * round trip delay

Sender/Receiver State

sender
- packets sent and acked (LAR = last ack recvd)
- packets sent but not yet acked
- packets not yet sent (LFS = last frame sent)

receiver
- packets received and acked (NFE = next frame expected)
- packets received out of order
- packets not yet received (LFA = last frame ok)

Sliding Window



What if we lose a packet?

Go back N (original TCP)
- receiver acks "got up through k" ("cumulative ack")
- ok for receiver to buffer out of order packets
- on timeout, sender restarts from k+1

Selective retransmission (RFC 2018)
- receiver sends ack for each pkt in window
- on timeout, resend only missing packet

Can we shortcut timeout?

If packets usually arrive in order, out of order delivery is (probably) a packet loss
- Negative ack
  - receiver requests missing packet
- Fast retransmit (TCP)
  - receiver acks with NFE-1 (or selective ack)
  - if sender gets acks that don't advance NFE, resends missing packet

Sender Algorithm

Send full window, set timeout
On receiving an ack:
  if it increases LAR (last ack received)
    send next packet(s)
      -- no more than window size outstanding at once
  else (already received this ack)
    if receive multiple acks for LAR, next packet may have been lost; retransmit LAR + 1 (and more if selective ack)
On timeout:
  resend LAR + 1 (first packet not yet acked)

Receiver Algorithm

On packet arrival:
  if packet is the NFE (next frame expected)
    send ack
    increase NFE
    hand any packet(s) below NFE to application
  else if < NFE (packet already seen and acked)
    send ack and discard // Q: why is ack needed?
  else (packet is > NFE, arrived out of order)
    buffer and send ack for NFE – 1
      -- signal sender that NFE might have been lost
      -- and with selective ack: which packets correctly arrived

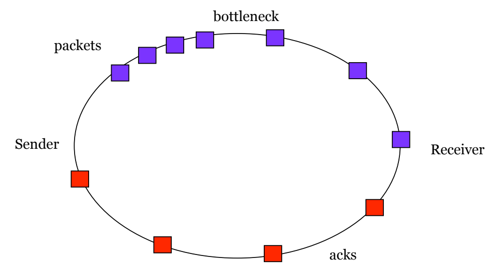
What if link is very lossy?

Wireless packet loss rates can be 10-30%
- end to end retransmission will still work
- will be inefficient, especially with go back N

Solution: hop by hop retransmission
- performance optimization, not for correctness

End to end principle
- ok to do optimizations at lower layer
- still need end to end retransmission; why?

Avoiding burstiness: ack pacing



Window size = round trip delay * bit rate

How many sequence #'s?

Window size + 1?
- Suppose window size = 3
- Sequence space: 0 1 2 3 0 1 2 3
- send 0 1 2, all arrive
  - if acks are lost, resend 0 1 2
  - if acks arrive, send new 3 0 1

Window <= (max seq # + 1) / 2

How do we determine timeouts?

If timeout too small, useless retransmits
- can lead to congestion collapse (and did in 86)
- as load increases, longer delays, more timeouts, more retransmissions, more load, longer delays, more timeouts …
- Dynamic instability!

If timeout too big, inefficient
- wait too long to send missing packet

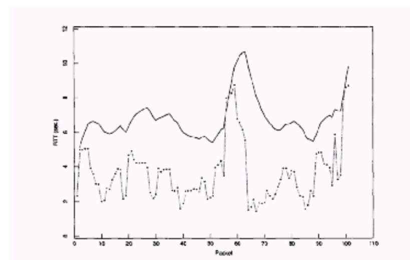
Timeout should be based on actual round trip time (RTT)
- varies with destination subnet, routing changes, congestion, …

Estimating RTTs

Idea: Adapt based on recent past measurements

- For each packet, note time sent and time ack received
- Compute RTT samples and average recent samples for timeout
- EstimatedRTT = α x EstimatedRTT + (1 - α) x SampleRTT
- This is an exponentially-weighted moving average (low pass filter) that smoothes the samples. Typically, α = 0.8 to 0.9.
- Set timeout to small multiple (2) of the estimate

Estimated Retransmit Timer

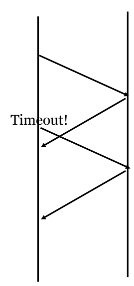


Retransmission ambiguity

How do we distinguish first ack from retransmitted ack?

- First send to first ack?
  - What if ack dropped?
- Last send to last ack?
  - What if last ack dropped?

Might never be able to fix too short a timeout!



Retransmission ambiguity: Solutions?

TCP: Karn-Partridge

- ignore RTT estimates for retransmitted pkts
- double timeout on every retransmission

Add sequence #'s to retransmissions (retry #1, retry #2, ...)

Modern TCP (RFC 1323): Add timestamp into packet header; ack returns timestamp

Jacobson/Karels Algorithm

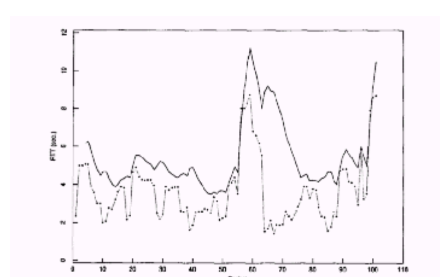
Problem:

- Variance in RTTs gets large as network gets loaded
- Average RTT isn't a good predictor when we need it most

Solution: Track variance too.

- Difference = SampleRTT – EstimatedRTT
- EstimatedRTT = EstimatedRTT + (δ x Difference)
- Deviation = Deviation + δ(|Difference|- Deviation)
- Timeout = μ x EstimatedRTT + ϕ x Deviation
- In practice, δ = 1/8, μ = 1 and ϕ = 4

Estimate with Mean + Variance



Transport: Practice

Protocols

- IP -- Internet protocol
- UDP -- user datagram protocol
- TCP -- transmission control protocol
- RPC -- remote procedure call
- HTTP -- hypertext transfer protocol
- And a bunch more...

How do we connect processes?

IP provides host to host packet delivery

- header has source, destination IP address

For applications to communicate, need to demux packets sent to host to target app

- Web browser (HTTP), Email servers (SMTP), hostname translation (DNS), RealAudio player (RTSP), etc.
- Process id is OS-specific and transient

Ports

Port is a mailbox that processes "rent"

- Uniquely identify communication endpoint as (IP address, protocol, port)

How do we pick port #'s?

- Client needs to know port # to send server a request
- Servers bind to "well-known" port numbers
  - Ex: HTTP 80, SMTP 25, DNS 53, ...
  - Ports below 1024 reserved for "well-known" services
- Clients use OS-assigned temporary (ephemeral) ports
  - Above 1024, recycled by OS when client finished

Sockets

OS abstraction representing communication endpoint

- Layer on top of TCP, UDP, local pipes

server (passive open)

- bind -- socket to specific local port
- listen -- wait for client to connect

client (active open)

- connect -- to specific remote port

User Datagram Protocol (UDP)

Provides application – application delivery

Header has source & dest port #'s

- IP header provides source, dest IP addresses

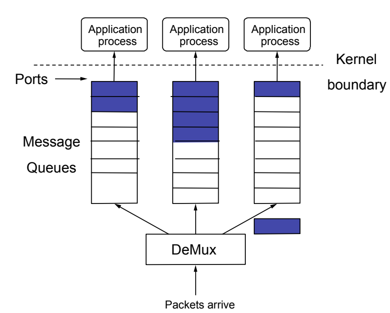
Deliver to destination port on dest machine

Reply returns to source port on source machine

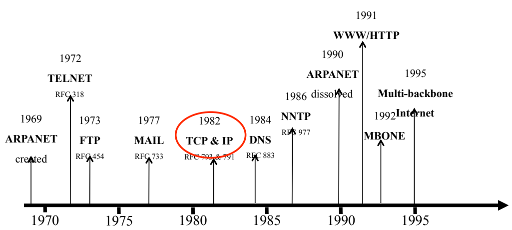
No retransmissions, no sequence #s

=> stateless

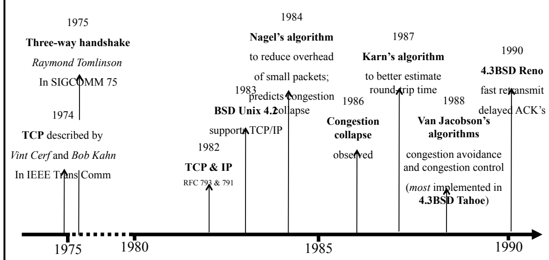
UDP Delivery



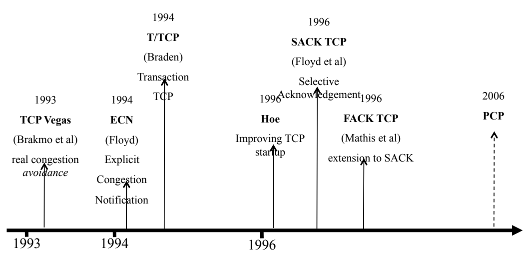
A brief Internet history...



TCP: This is your life...



TCP: After 1990

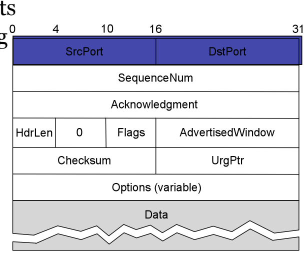


Transmission Control Protocol (TCP)

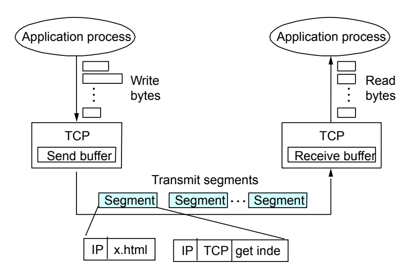
- Reliable bi-directional byte stream
  - No message boundaries
  - Ports as application endpoints
- Sliding window, go back N/SACK, RTT est, ...
  - Highly tuned congestion control algorithm
- Flow control
  - prevent sender from overrunning receiver buffers
- Connection setup
  - negotiate buffer sizes and initial seq #s
  - Needs to work between all types of computers (supercomputer -> 8086)

TCP Packet Header

- Source, destination ports
- Sequence # (bytes being sent)
- Ack # (next byte expected)
- Receive window size
- Checksum
- Flags: SYN, FIN, RST



TCP Delivery



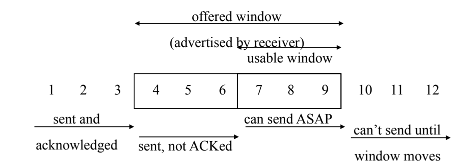
TCP Sliding Window

Per-byte, not per-packet (why?)
- send packet says "here are bytes j-k"
- ack says "received up to byte k"

Send buffer >= send window
- can buffer writes in kernel before sending
- writer blocks if try to write past send buffer

Receive buffer >= receive window
- buffer acked data in kernel, wait for reads
- reader blocks if try to read past acked data

Visualizing the window



Left side of window advances when data is acknowledged.

Right side controlled by size of window advertisement

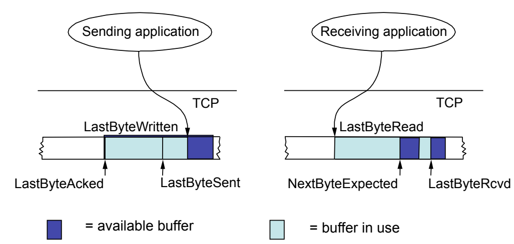
Flow Control

What if sender process is faster than receiver process?
- Data builds up in receive window
- if data is acked, sender will send more!
- If data is not acked, sender will retransmit!

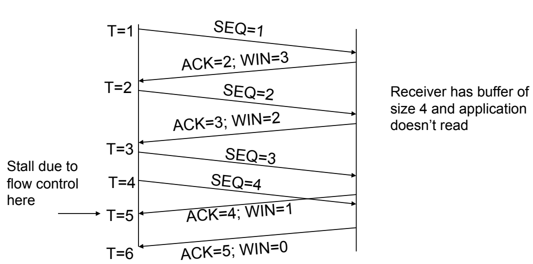
Sender must transmit data no faster than it can be consumed by the receiver
- Receiver might be a slow machine
- App might consume data slowly

Sender sliding window <= free receiver buffer
- Advertised window = # of free bytes; if zero, stop

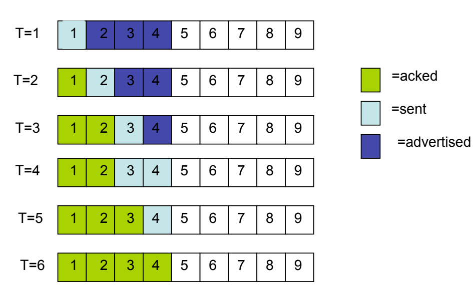
Sender and Receiver Buffering



Example – Exchange of Packets



Example – Buffer at Sender



How does sender know when to resume sending?

- If receive window = 0, sender stops
  - no data => no acks => no window updates
- Sender periodically pings receiver with one byte packet
  - receiver acks with current window size
- Why not have receiver ping sender?

Should sender be greedy (I)?

- Should sender transmit as soon as any space opens in receive window?
  - Silly window syndrome
    - receive window opens a few bytes
    - sender transmits little packet
    - receive window closes
- Solution (Clark, 1982): sender doesn't resume sending until window is half open

Should sender be greedy (II)?

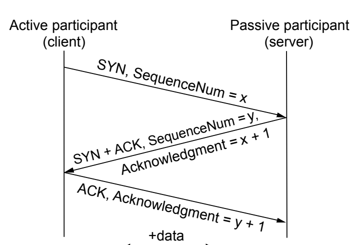
- App writes a few bytes; send a packet?
  - Don't want to send a packet for every keystroke
  - If buffered writes >= max segment size
  - if app says "push" (ex: telnet, on carriage return)
  - after timeout (ex: 0.5 sec)
- Nagle's algorithm
  - Never send two partial segments; wait for first to be acked, before sending next
  - Self-adaptive: can send lots of tinygrams if network is being responsive
- But (!) poor interaction with delayed acks (later)

TCP Connection Management

- Setup
  - assymetric 3-way handshake
- Transfer
  - sliding window; data and acks in both directions
- Teardown
  - symmetric 2-way handshake
- Client-server model
  - initiator (client) contacts server
  - listener (server) responds, provides service

Three-Way Handshake

- Opens both directions for transfer



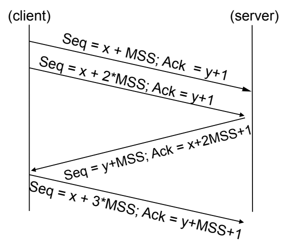
Do we need 3-way handshake?

- Allows both sides to
  - allocate state for buffer size, state variables, …
  - calculate estimated RTT, estimated MTU, etc.
- Helps prevent
  - Duplicates across incarnations
  - Intentional hijacking
    - random nonces => weak form of authentication
- Short-circuit?
  - Persistent connections in HTTP (keep connection open)
  - Transactional TCP (save seq #, reuse on reopen)
  - But congestion control effects dominate

TCP Transfer

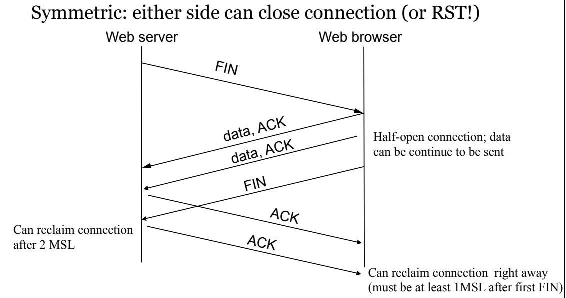
Connection is bi-directional

- acks can carry response data

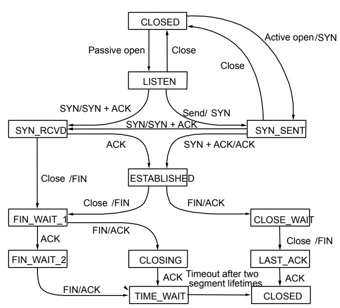


TCP Connection Teardown

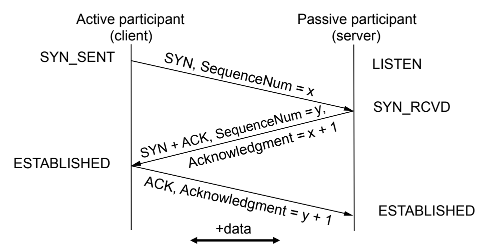
Symmetric: either side can close connection (or RST!)



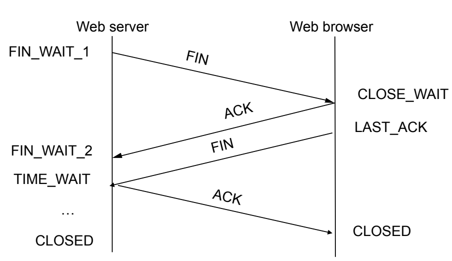
TCP State Transitions



TCP Connection Setup, with States



TCP Connection Teardown



The TIME_WAIT State

We wait 2MSL (two times the maximum segment lifetime of 60 seconds) before completing the close

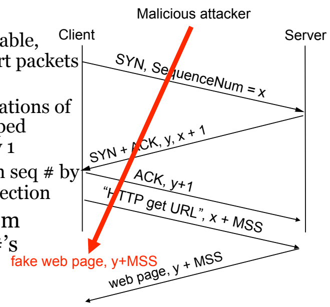
Why?

ACK might have been lost and so FIN will be resent
Could interfere with a subsequent connection

TCP Handshake in an Uncooperative Internet

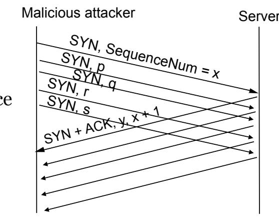
TCP Hijacking
- if seq # is predictable, attacker can insert packets into TCP stream
- many implementations of TCP simply bumped previous seq # by 1
- attacker can learn seq # by setting up a connection

Solution: use random initial sequence #'s
- weak form of authentication



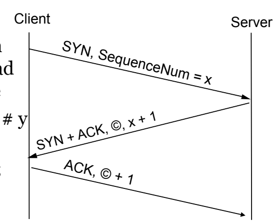
TCP Handshake in an Uncooperative Internet

TCP SYN flood
- server maintains state for every open connection
- if attacker spoofs source addresses, can cause server to open lots of connections
- eventually, server runs out of memory



TCP SYN cookies

Solution: SYN cookies
- Server keeps no state in response to SYN; instead makes client store state
- Server picks return seq # y = © that encrypts x
- Gets © +1 from sender; unpacks to yield x



Can data arrive before ACK?

How can TCP choose segment size?

Pick LAN MTU as segment size?
- LAN MTU can be larger than WAN MTU
- E.g., Gigabit Ethernet jumbo frames

Pick smallest MTU across all networks in Internet?
- Most traffic is local!
  - Local file server, web proxy, DNS cache, ...
- Increases packet processing overhead

Discover MTU to each destination? (IP DF bit)

Guess?

Layering Revisited

IP layer "transparent" packet delivery
- Implementation decisions affect higher layers (and vice versa)
  - Fragmentation => reassembly overhead
    - path MTU discovery
  - Packet loss => congestion or lossy link?
    - link layer retransmission
  - Reordering => packet loss or multipath?
    - router hardware tries to keep packets in order
  - FIFO vs. active queue management

IP Packet Header Limitations

Fixed size fields in IPv4 packet header
- source/destination address (32 bits)
  - limits to ~ 4B unique public addresses; about 600M allocated
  - NATs map multiple hosts to single public address
- IP ID field (16 bits)
  - limits to 65K fragmented packets at once => 100MB in flight?
  - in practice, fewer than 1% of all packets fragment
- Type of service (8 bits)
  - unused until recently; used to express priorities
- TTL (8 bits)
  - limits max Internet path length to 255; typical max is 30
- Length (16 bits)
  - Much larger than most link layer MTU's

TCP Packet Header Limitations

Fixed size fields in TCP packet header

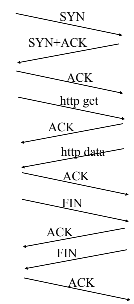
- seq #/ack # -- 32 bits (can't wrap within MSL)
  - T1 ~ 6.4 hours; OC-192 ~ 3.5 seconds
- source/destination port # -- 16 bits
  - limits # of connections between two machines (NATs)
  - ok to give each machine multiple IP addresses
- header length
  - limits # of options
- receive window size -- 16 bits (64KB)
  - rate = window size / delay
  - Ex: 100ms delay => rate ~ 5Mb/sec
  - RFC 1323: receive window scaling
  - Defaults still a performance problem

HTTP on TCP

How do we reduce the # of messages?

Delayed ack: wait for 200ms for reply or another pkt arrival

TCP RST from web server

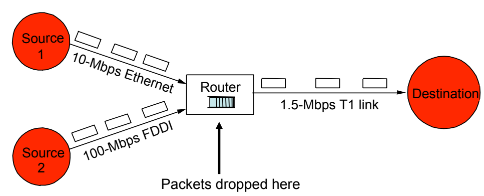


Bandwidth Allocation

How do we efficiently share network resources among billions of hosts?

- Congestion control
  - Sending too fast causes packet loss inside network -> retransmissions -> more load -> more packet losses -> ...
  - Don't send faster than network can accept
- Fairness
  - How do we allocate bandwidth among different users?
  - Each user should (?) get fair share of bandwidth

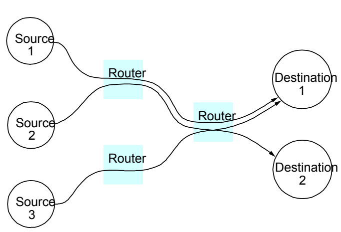
Congestion



Buffer absorbs bursts when input rate > output
If sending rate is persistently > drain rate, queue builds
Dropped packets represent wasted work

Chapter 6, Figure 1

Fairness



Each flow from a source to a destination should (?) get an equal share of the bottleneck link … depends on paths and other traffic

Chapter 6, Figure 2

The Problem

Original TCP sent full window of data

When links become loaded, queues fill up, and this can lead to:

- *Congestion collapse: w*hen round-trip time exceeds retransmit interval -- every packet is retransmitted many times
- Synchronized behavior: network oscillates between loaded and unloaded

TCP Congestion Control

Goal: efficiently and fairly allocate network bandwidth

- Robust RTT estimation
- Additive increase/multiplicative decrease
  - oscillate around bottleneck capacity
- Slow start
  - quickly identify bottleneck capacity
- Fast retransmit
- Fast recovery

Tracking the Bottleneck Bandwidth

Sending rate = window size/RTT

Multiplicative decrease

- Timeout => dropped packet => cut window size in half
  - and therefore cut sending rate in half

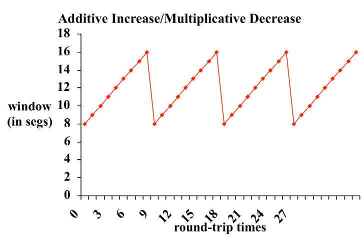
Additive increase

- Ack arrives => no drop => increase window size by one packet/window
  - and therefore increase sending rate a little

TCP "Sawtooth"

Oscillates around bottleneck bandwidth

- adjusts to changes in competing traffic



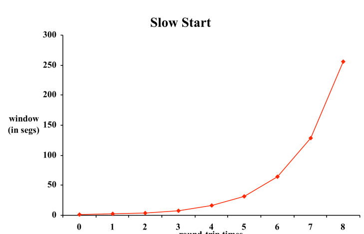
Slow start

How do we find bottleneck bandwidth?

- Start by sending a single packet
  - start slow to avoid overwhelming network
- Multiplicative increase until get packet loss
  - quickly find bottleneck
- Remember previous max window size
  - shift into linear increase/multiplicative decrease when get close to previous max ~ bottleneck rate
  - called "congestion avoidance"

Slow Start

Quickly find the bottleneck bandwidth



TCP Mechanics Illustrated

Source

Router

Dest

100 Mbps

0.9 ms latency

10 Mbps

0 latency

78

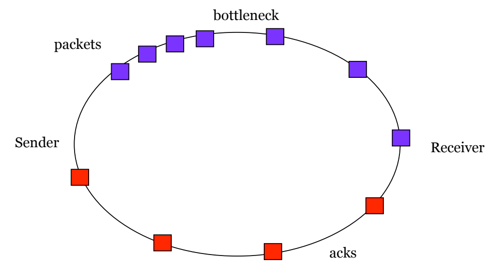
Slow Start Problems

Bursty traffic source
- will fill up router queues, causing losses for other flows
- solution: ack pacing

Slow start usually overshoots bottleneck
- will lose many packets in window
- solution: remember previous threshold

Short flows
- Can spend entire time in slow start!
- solution: persistent connections?

Avoiding burstiness: ack pacing

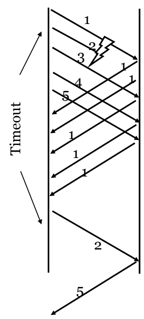


Window size = round trip delay * bit rate

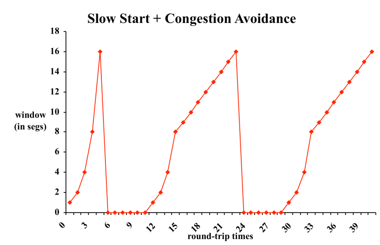
Ack Pacing After Timeout

Packet loss causes timeout, disrupts ack pacing
- slow start/additive increase are *designed* to cause packet loss

After loss, use slow start to regain ack pacing
- switch to linear increase at last successful rate
- “congestion avoidance”



Putting It All Together



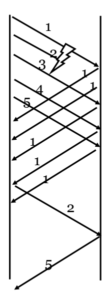
Timeouts dominate performance!

Fast Retransmit

Can we detect packet loss without a timeout?
- Receiver will reply to each packet with an ack for last byte received in order

Duplicate acks imply either
- packet reordering (route change)
- packet loss

TCP Tahoe
- resend if sender gets three duplicate acks, without waiting for timeout



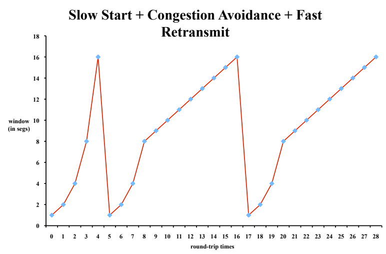
Fast Retransmit Caveats

Assumes in order packet delivery
- Recent proposal: measure rate of out of order delivery; dynamically adjust number of dup acks needed for retransmit

Doesn’t work with small windows (e.g. modems)
- what if window size <= 3

Doesn’t work if many packets are lost
- example: at peak of slow start, might lose many packets

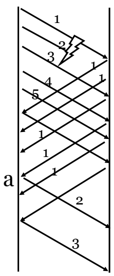
Fast Retransmit



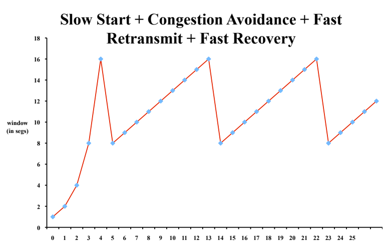
Regaining ack pacing limits performance

Fast Recovery

- Use duplicate acks to maintain ack pacing
  - duplicate ack => packet left network
  - after loss, send packet after every other acknowledgement
- Doesn't work if lose many packets in a row
  - fall back on timeout and slow start to reestablish ack pacing



Fast Recovery



Delayed ACKS

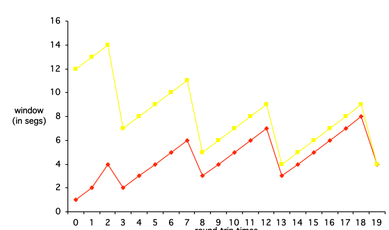
- Problem:
  - In request/response programs, server will send separate ACK and response packets
    - computing the response can take time
- TCP solution:
  - Don't ACK data immediately
  - Wait 200ms (must be less than 500ms)
  - Must ACK every other packet
  - Must not delay duplicate ACKs

Delayed Acks

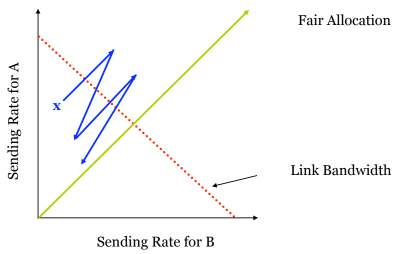
- Recall that acks are delayed by 200ms to wait for application to provide data
- But (!) TCP congestion control triggered by acks
  - if receive half as many acks => window grows half as fast
- Slow start with window = 1
  - ack will be delayed, even though sender is waiting for ack to expand window

What if two TCPs share link?

- Reach equilibrium independent of initial bw
  - assuming equal RTTs, "fair" drops at the router

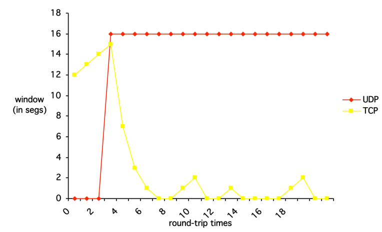


Equilibrium Proof



What if TCP and UDP share link?

Independent of initial rates, UDP will get priority! TCP will take what's left.



What if two different TCP implementations share link?

If cut back more slowly after drops => will grab bigger share

If add more quickly after acks => will grab bigger share

Incentive to cause congestion collapse!

- Many TCP "accelerators"
- Easy to improve perf at expense of network

One solution: enforce good behavior at router

What if TCP connection is short?

Slow start dominates performance

- What if network is unloaded?
- Burstiness causes extra drops

Packet losses unreliable indicator

- can lose connection setup packet
- can get drop when connection near done
- signal unrelated to sending rate

In limit, have to signal every connection

- 50% loss rate as increase # of connections

Example: 10KB document
10Mb/s Ethernet,70ms RTT, 536 MSS

Ethernet ~ 10 Mb/s

64KB window, 70ms RTT ~ 7.5 Mb/s

can only use 10KB window ~ 1.2 Mb/s

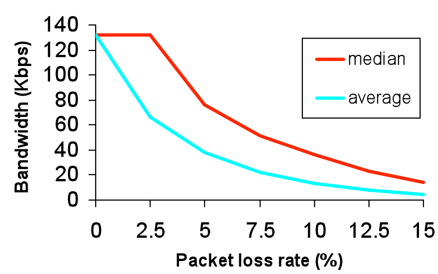
5% drop rate ~ 275 Kb/s (steady state)

model timeouts ~ 228 Kb/s

slow start, no losses ~ 140 Kb/s

slow start, with 5% drop ~ 75 Kb/s

Short flow bandwidth



Flow length=10Kbytes, RTT=70ms

TCP over Wireless

What's the problem?

How might we fix it?

97

TCP over 10Gbps Pipes

What's the problem?

How might we fix it?

98

TCP and ISP router buffers

What's the problem?

How might we fix it?

99

TCP and Real-time Flows

What's the problem?

How might we fix it?

100