

End-to-end transport protocols, TCP, congestion control, and DNS

George Porter
CSE 124
January 27, 2015

Announcements

- Homework 1 extended to Thursday (1/29)
- For Thursday (1/29):
 - Eric Brewer, "Lessons from Giant-Scale Internet Services", IEEE Internet Computing. Vol. 5, No. 4. pp. 46-55. July/August 2001
 - Link on Syllabus
- Write up (to **ted.ucsd.edu**)
 - 1 paragraph: Summary of the paper
 - 1 paragraph: main contributions
 - 1 paragraph: Lessons you learn for building large-scale systems

Review from last time

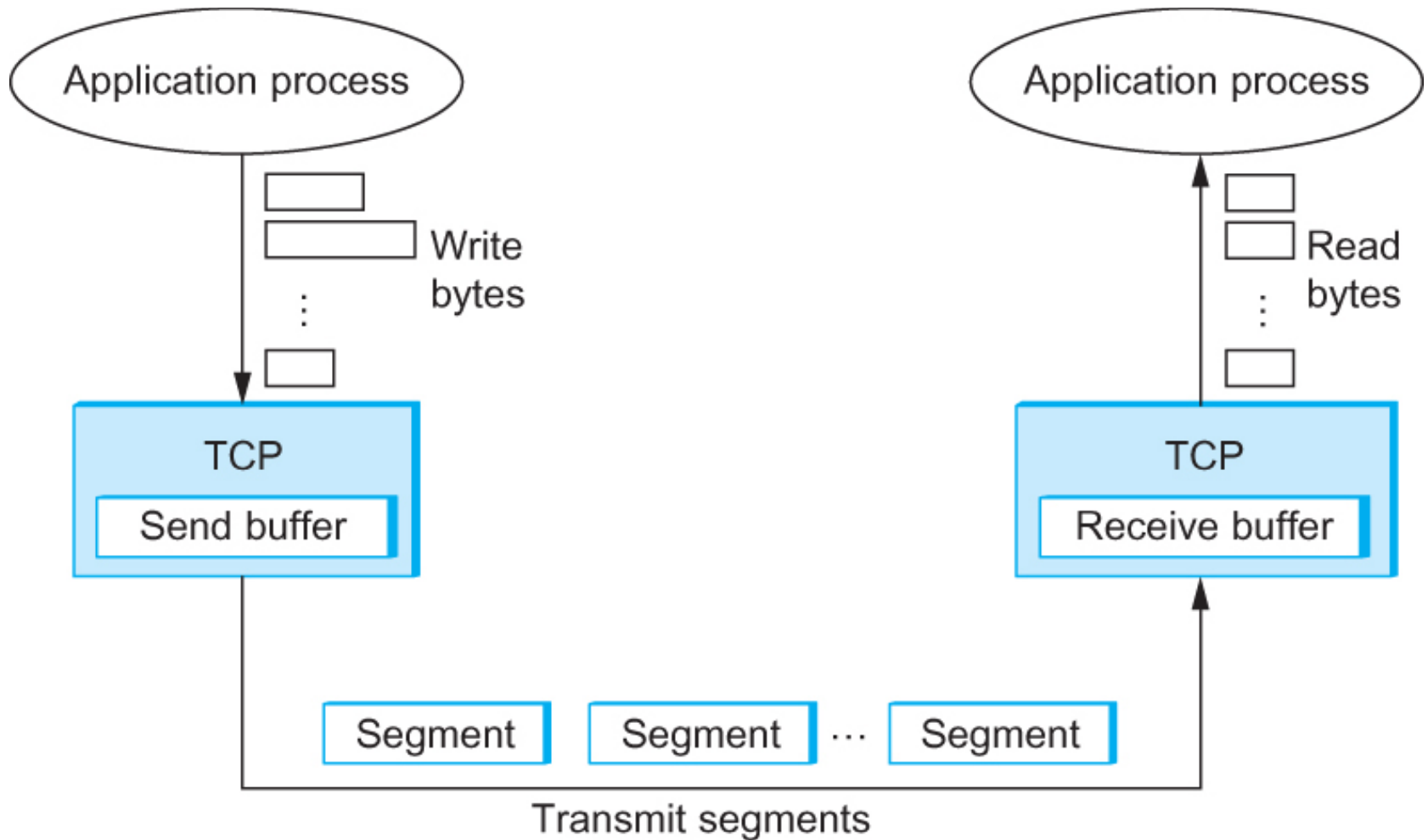
Host-to-Host vs. Process-to-Process Communication

- Ethernet
 - Link-layer
 - Host-to-host, on same local network
 - Unreliable, packet-oriented
- Internet Protocol/IP
 - Network-layer
 - Host-to-host, on remote networks
 - Unreliable, packet-oriented
- Transmission Control Protocol (TCP)
 - Transport layer
 - Process-to-process
 - Reliable, in-order byte stream

TCP

- Transmission Control Protocol (TCP)
 - Reliable in-order delivery of byte stream
 - Full duplex (endpoints simultaneously send/receive)
e.g., single socket for web browser talking to web server
 - Flow-control
To ensure that sender does not overrun *receiver*
Fast server talking to slow client
 - Congestion control
Keep the sender from overrunning the *network*
Many simultaneous connections across routers (cross traffic)

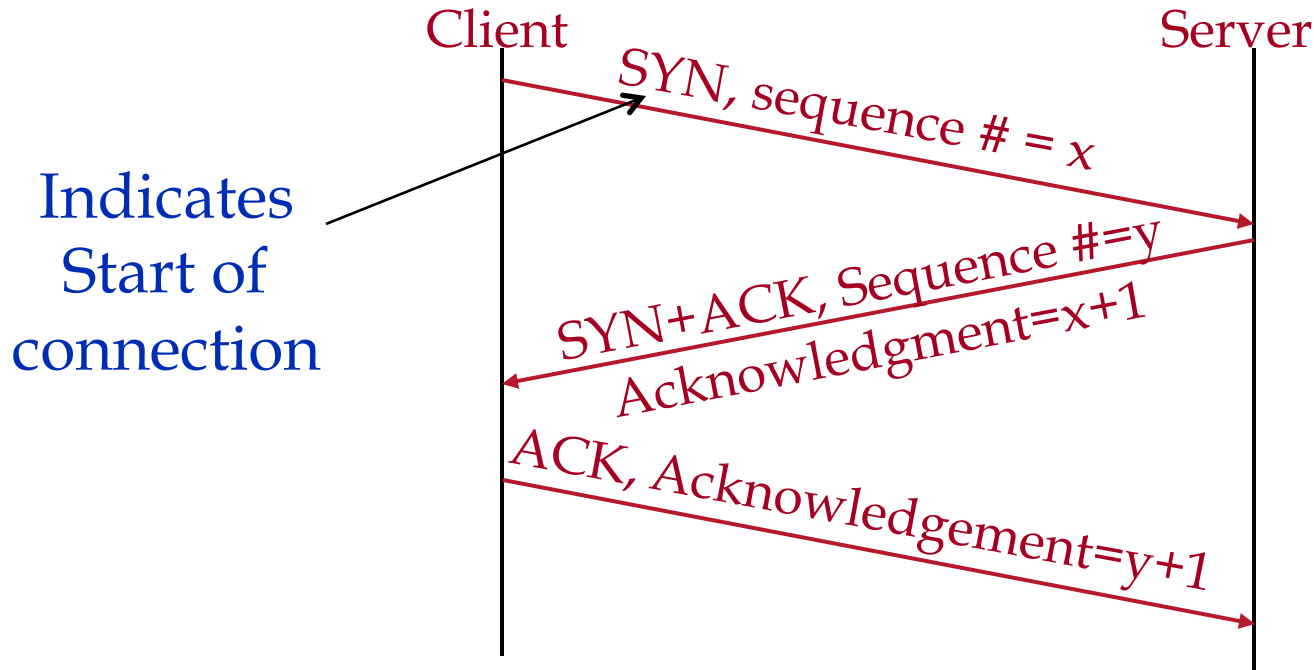
TCP Segment



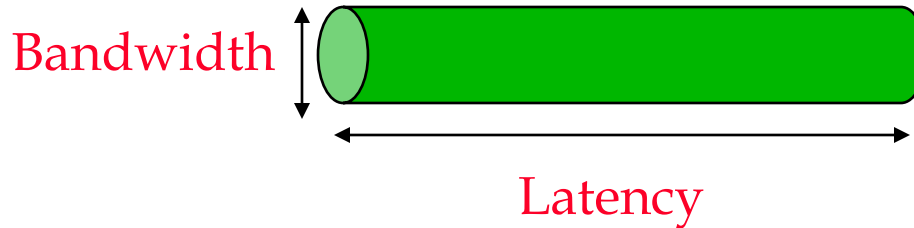
How TCP manages a byte stream.

TCP Connection Establishment

- Exchange necessary information to begin communication
- Three-way handshake
 - E.g., server listening on socket



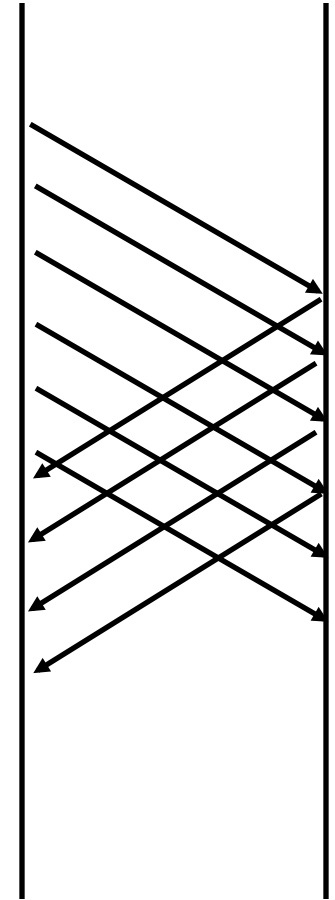
Keeping the Pipe Full



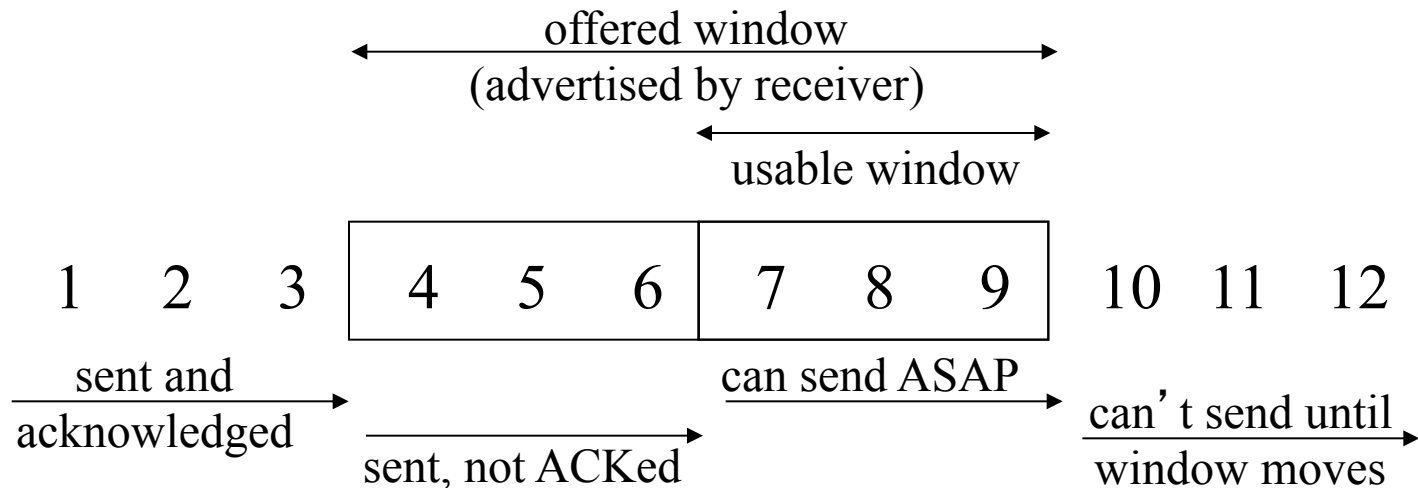
- Bandwidth-Delay product measures network capacity
- How much data can you put into the network before the first byte reaches receiver
- Stop and Wait: 1 data packet per RTT
 - Ex. 1.5-Mbps link with 45-ms RTT
 - Stop-and-wait: 182 Kbps
- Ideally, send enough packets to fill the pipe before requiring first ACK

How Do We Keep the Pipe Full?

- Send multiple packets without waiting for first to be ACK'd
- Reliable, unordered delivery:
 - Send new packet after each ACK
 - Sender keeps list of unac'd packets; resends after timeout
- Ideally, first ACK arrives immediately after pipe is filled
 - Opens up another “slot”
- Example 10 Mbit link, 100 ms RTT:
 - Send 125 KB to keep pipe full



Visualizing the Window

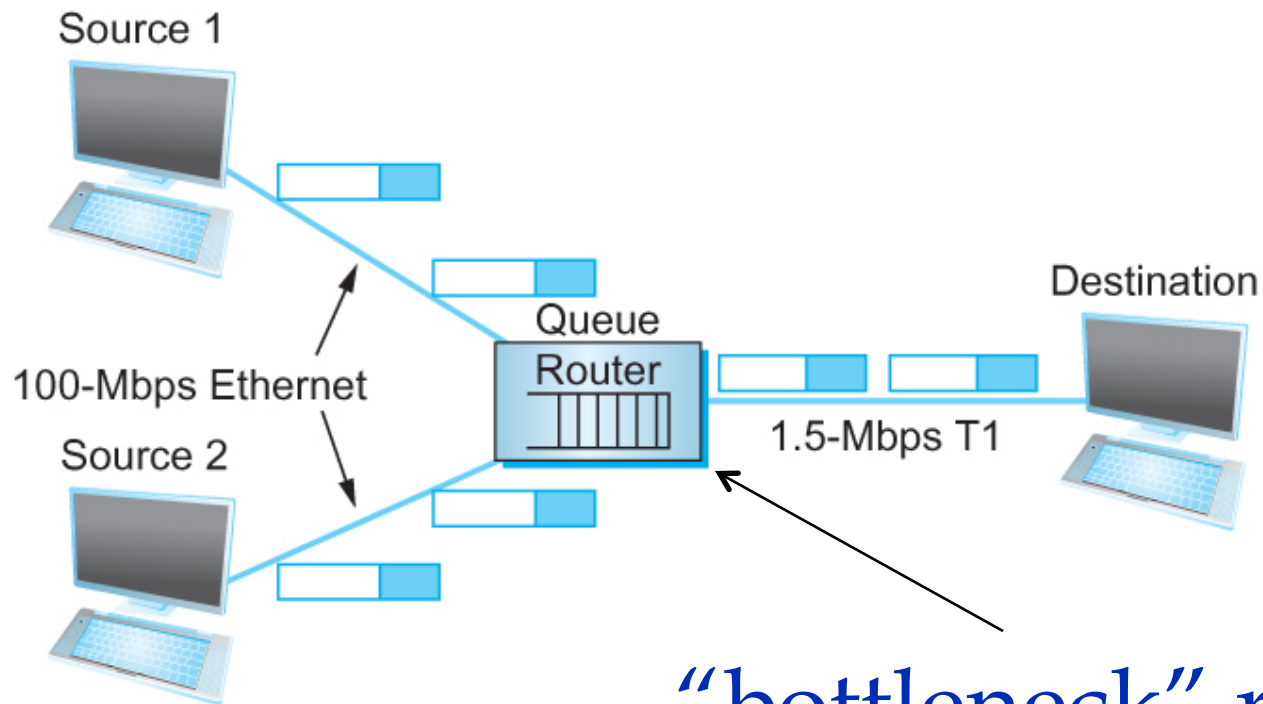


Left side of window advances when data is acknowledged.
Right side controlled by size of window advertisement.

TCP Congestion Control

Congestion control

- Key idea is to prevent different senders from overloading the network
 - Want to ensure fairness across users



“bottleneck” router

Summary of TCP Congestion Control

- Adjust transmission rate to match network bandwidth
 - Additive increase/multiplicative decrease
 - Slow start
 - Fast retransmit
 - Fast recovery

Jacobson Solution

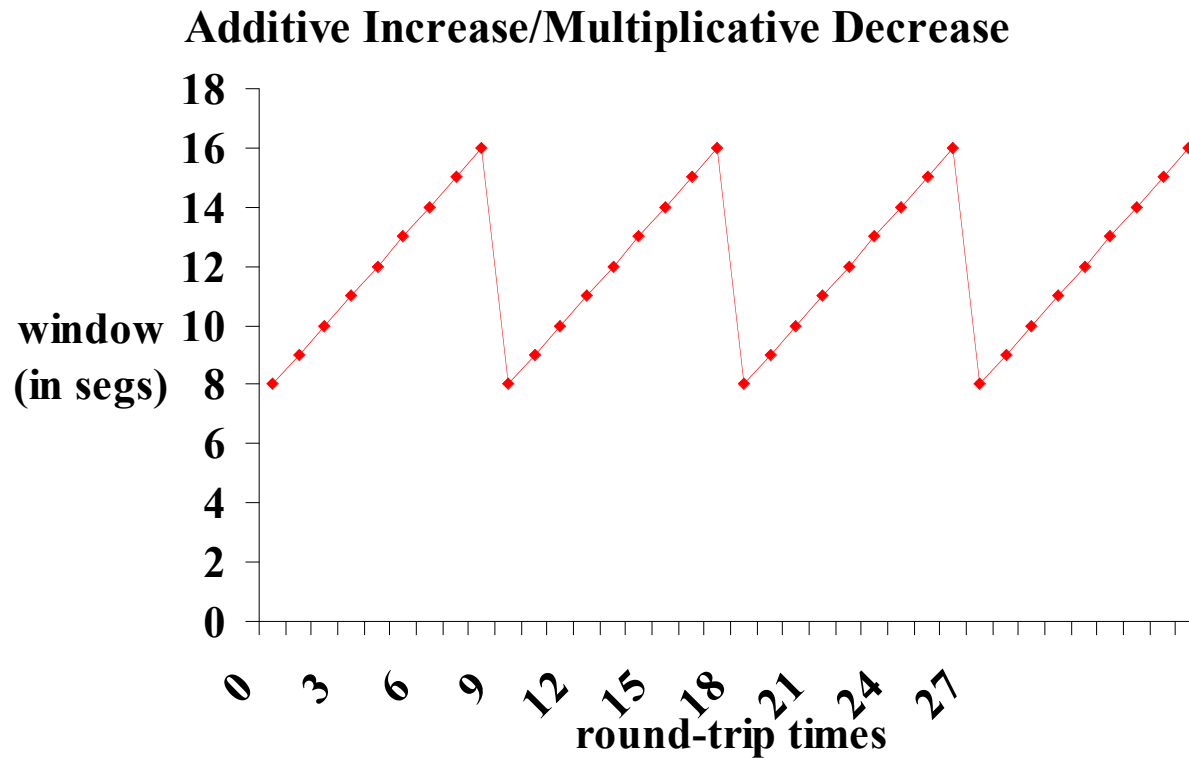
- Transport protocols should obey *conservation of packets*
 - Use ACKs to clock injection of new packets
- Modify retransmission timer to adapt to variations in delay
- Infer network bandwidth from packet loss
 - Drops → congestion → reduce rate
 - No drops → no congestion → increase rate
- Limit send rate based on minimum of *congestion window* and advertised window
 - Thus, interaction between flow control and congestion control

Tracking the Bottleneck Bandwidth

- Throughput = window size/RTT
- Multiplicative Decrease
 - Timeout → dropped packet → cut window size in half
- Additive Increase
 - ACK arrives → no drop → increase window size by one packet/window
- AIMD fundamental for network stability

TCP “Sawtooth”

- Oscillates around bottleneck bandwidth
 - Adjusts to changes in competing traffic

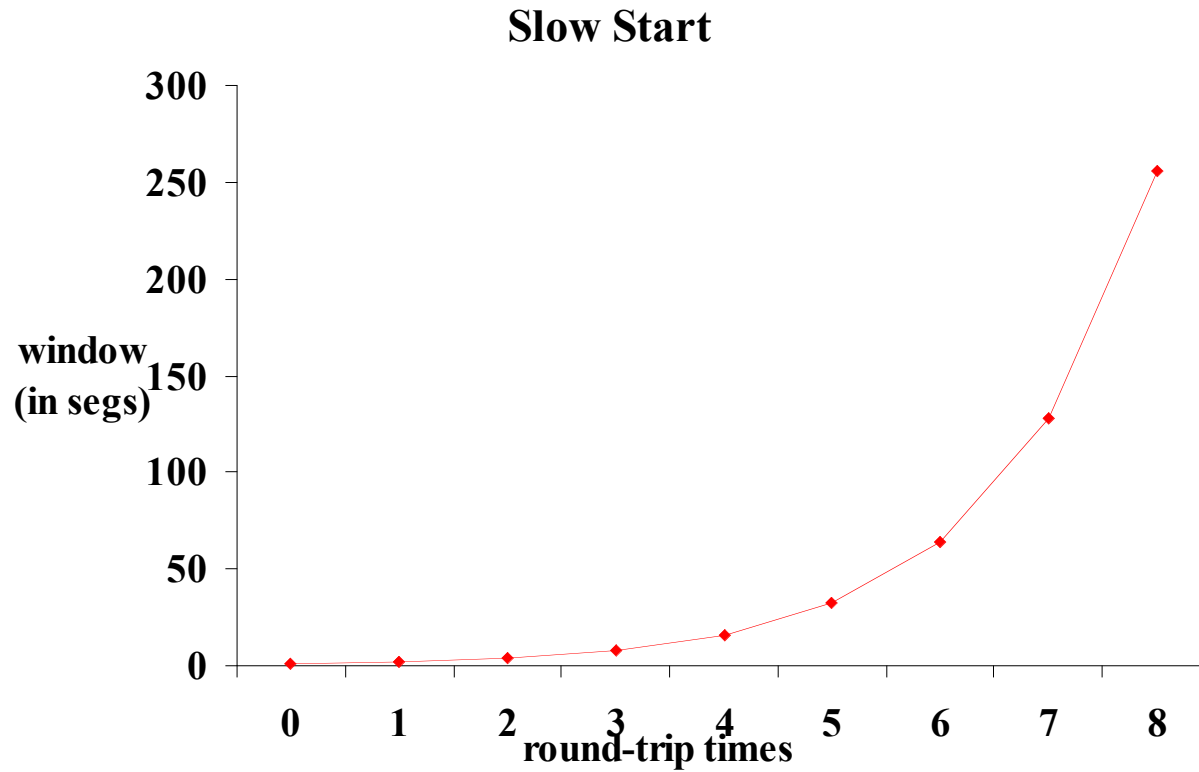


Slow Start

- How do we find bottleneck bandwidth?
- Cannot use ACKs to clock without reaching equilibrium
 - Start by sending a single packet
 - Start slow to avoid overwhelming network
 - Multiplicative increase until get packet loss
 - Quickly find bottleneck
 - Cut rate by half
 - Shift into linear increase/multiplicative decrease

Slow Start

- Quickly find the bottleneck bandwidth



Why is it called “Slow” start?

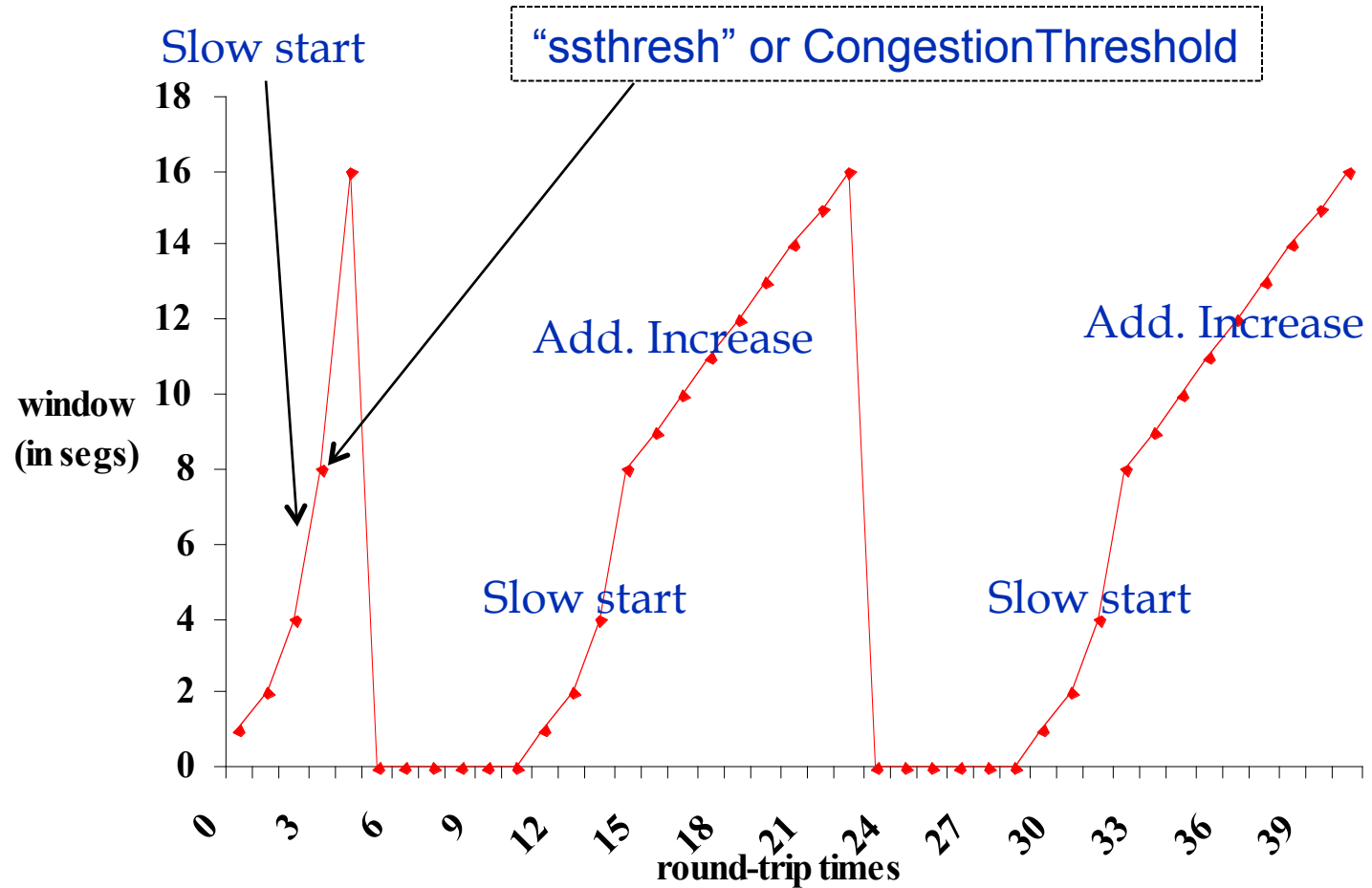
- What is the alternative?
 - Initially, sender sent entire window all at once
 - Many packet burst into network → drops
- Slow start is slower than that initial burst
- When does slow start occur?
 - When a new connection is opened
 - When a connection “times out” waiting for an ACK

Slow start to half the previous window, then go into additive increase mode

Slow Start Problems

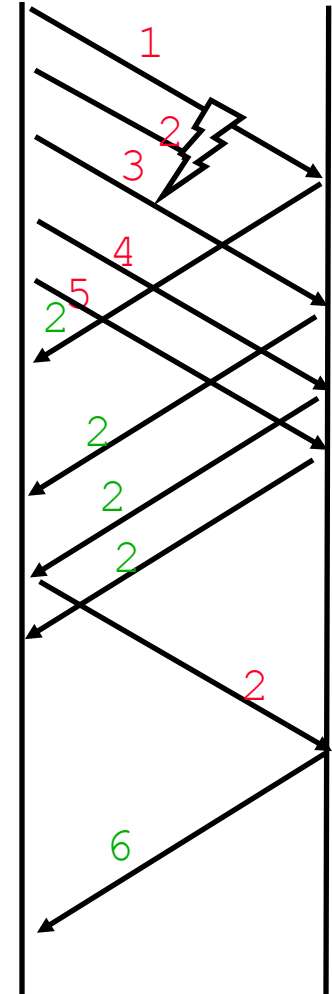
- Slow start usually overshoots bottleneck
 - Leading to many lost packets in window
 - Can lose up to half of window size
- Bursty traffic source
 - Will cause bursty losses for other flows
- Short flows
 - Can spend entire time in slow start
 - Especially for large bottleneck bandwidth
- Consider repeated connections to the same server
 - E.g., for web connections
 - Motivation for pipelined HTTP/1.1

Timeouts Dominate Performance



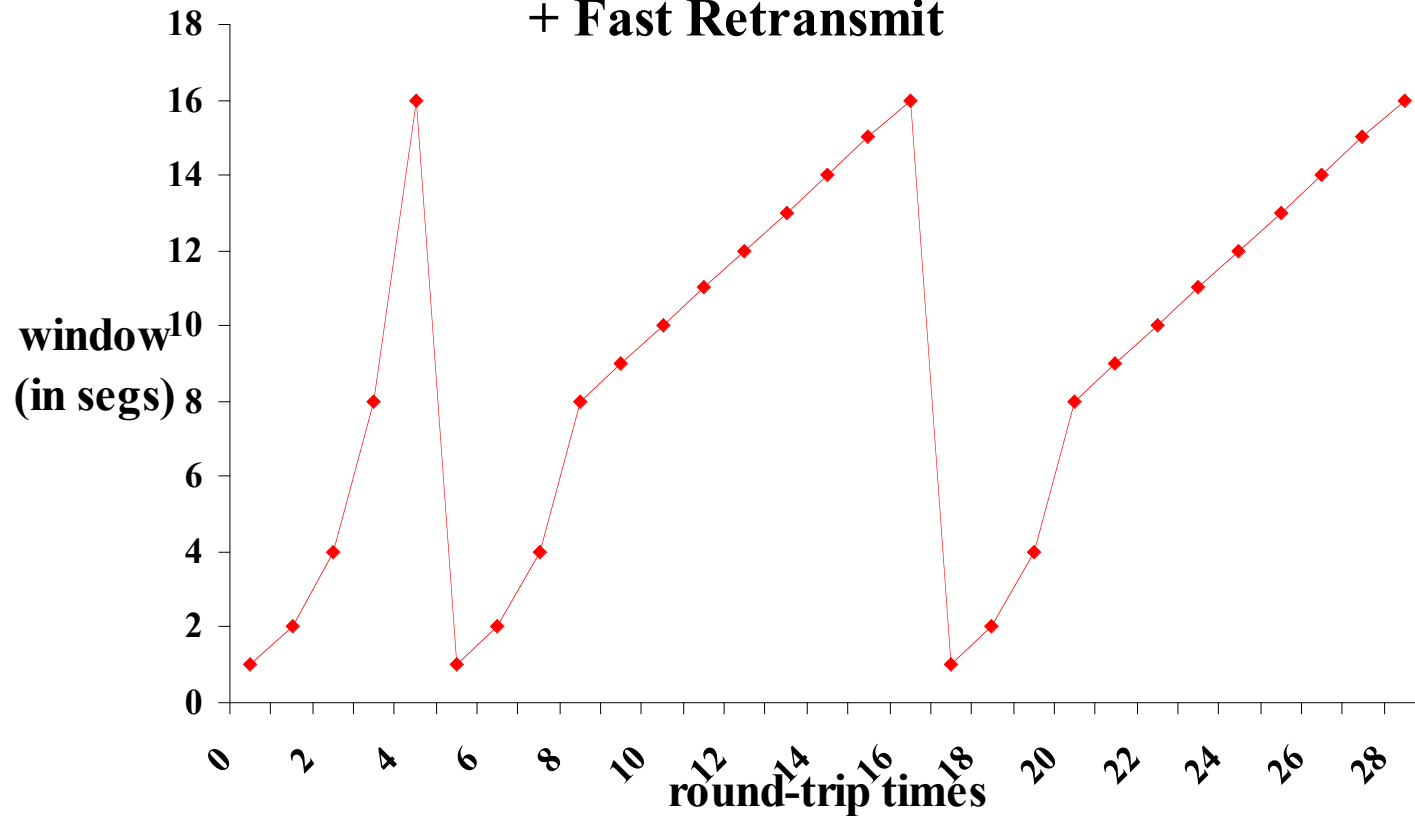
Fast Retransmit

- Can we detect packet loss without a timeout?
- Duplicate ACKs imply either
 - Packet reordering (route change)
 - Packet loss
- TCP Tahoe
 - Resend if see three dup ACKs
 - Eliminates timeout delay



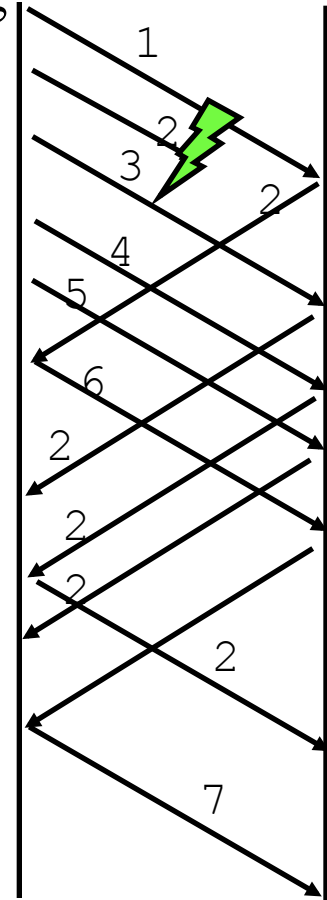
Fast Retransmit

**Slow Start + Congestion Avoidance
+ Fast Retransmit**



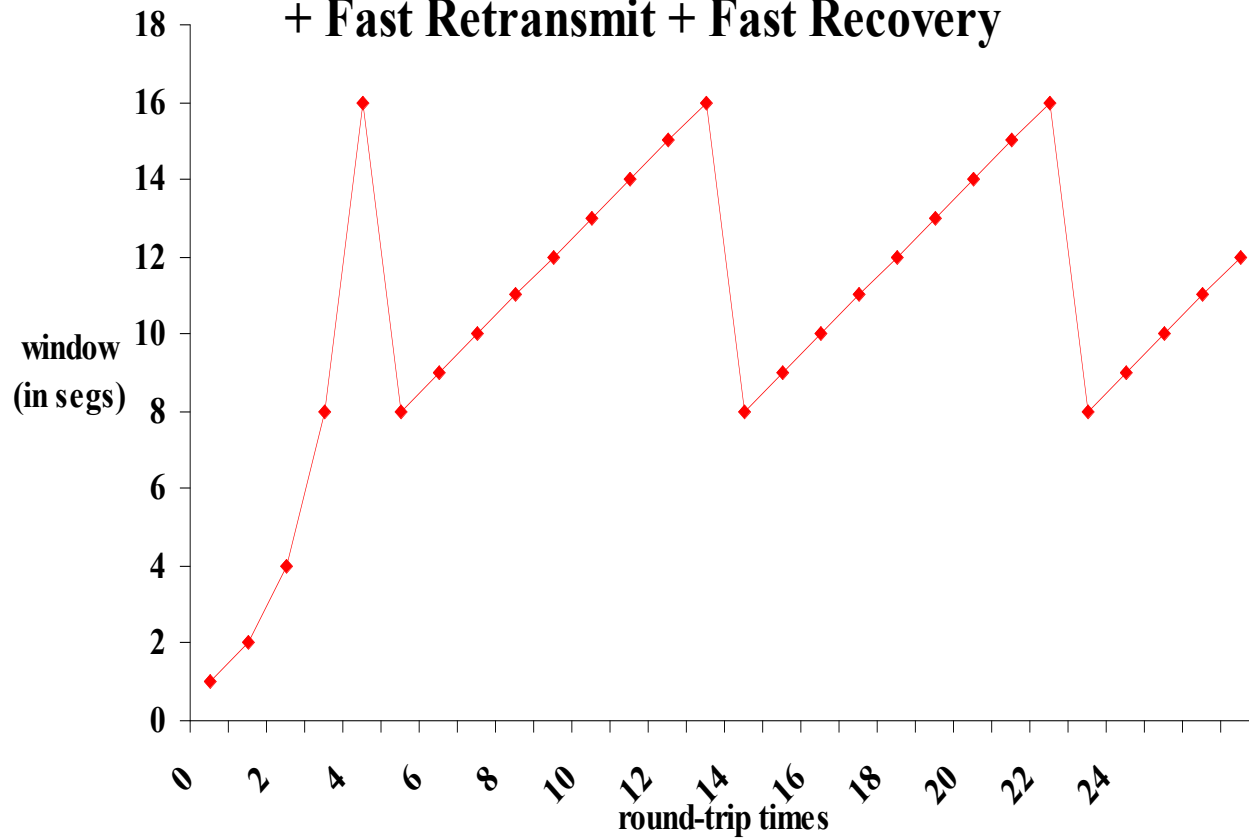
Fast Recovery

- Use duplicate ACKs to maintain ACK pacing
 - Dup ACK → packet left network
 - Every other ACK → send packet
- Fast recovery allows TCP to fall to half previous bottleneck bandwidth
 - Rather than all the way back to 1 packet/reinitiate slow start
 - **Slow start only at beginning/on timeout**



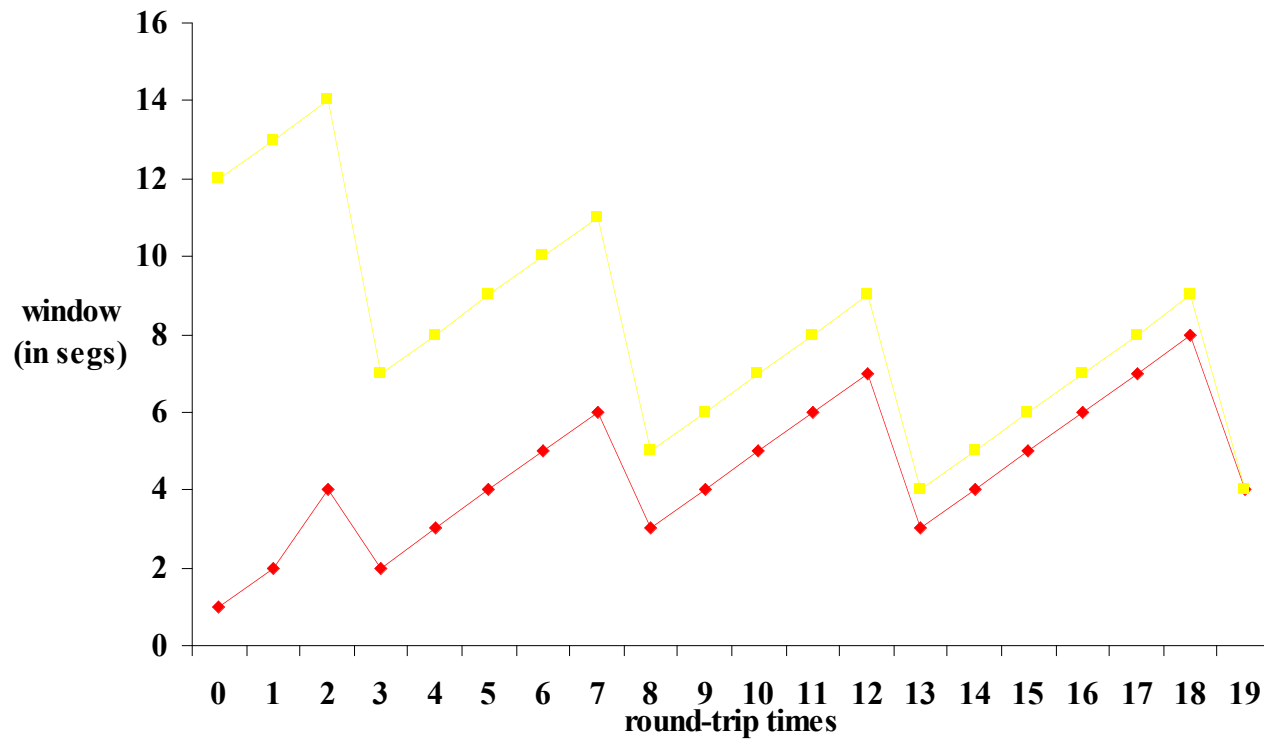
Fast Recovery

**Slow Start + Congestion Avoidance
+ Fast Retransmit + Fast Recovery**



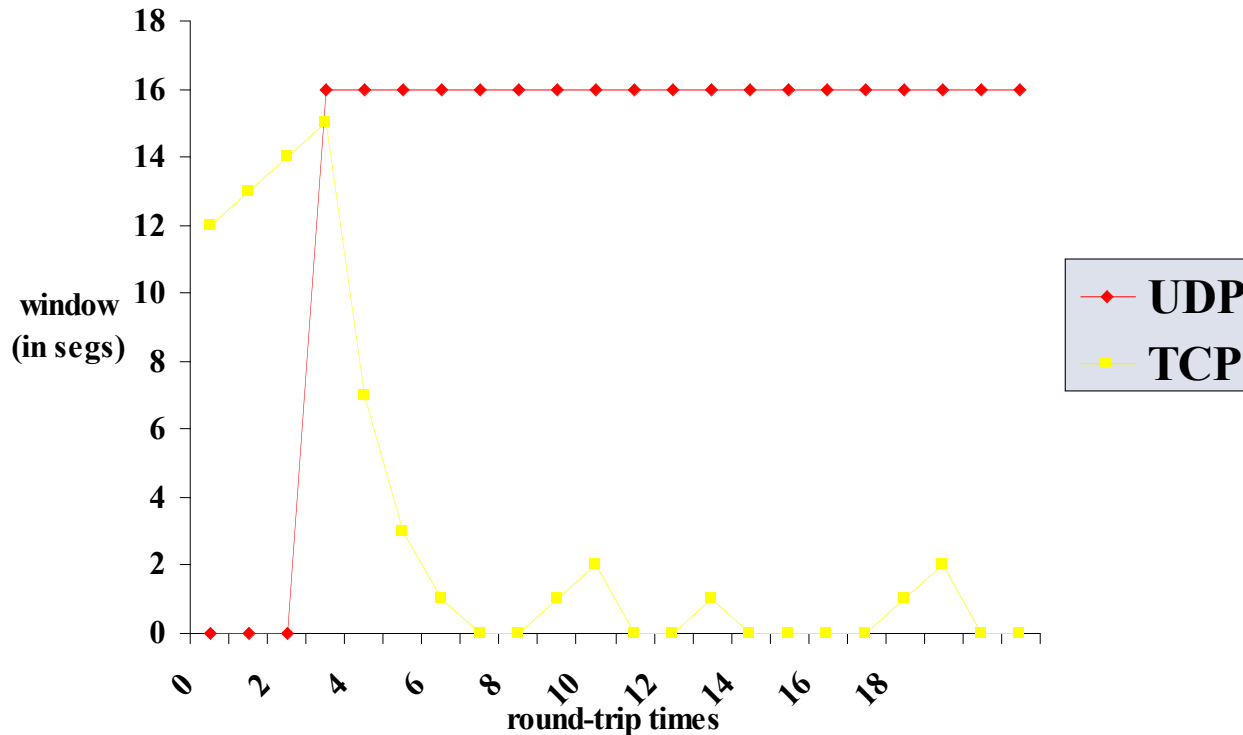
What if Two TCP Connections Share Link?

- Reach equilibrium independent of initial bandwidth (assuming equal RTTs)



What if TCP and UDP share link?

- Independent of initial rates, UDP will get priority! TCP will take what is left.



What if Two Different TCP Implementations Share Link?

- Problem: many different TCP implementations
- If cut back more slowly after drops → grab bigger share
- If add more quickly after ACKs → grab bigger share
- Incentive to cause congestion collapse
 - Many TCP “accelerators”
 - Easy to improve perf at expense of network
- Solutions?

What if Two Different TCP Implementations Share Link?

- Problem: many different TCP implementations
- If cut back more slowly after drops → grab bigger share
- If add more quickly after ACKs → grab bigger share
- Incentive to cause congestion collapse
 - Many TCP “accelerators”
 - Easy to improve perf at expense of network
- Q: Solutions?
 - Per-flow fair queuing at router

TCP Congestion Control Summary

- Slow Start
- Adaptive retransmission
 - Account for average *and* variance
- Fast retransmission
 - Triple duplicate ACKs
- Fast restart
 - Use ACKs in pipeline to avoid shrinking congestion window to one

Domain Name System

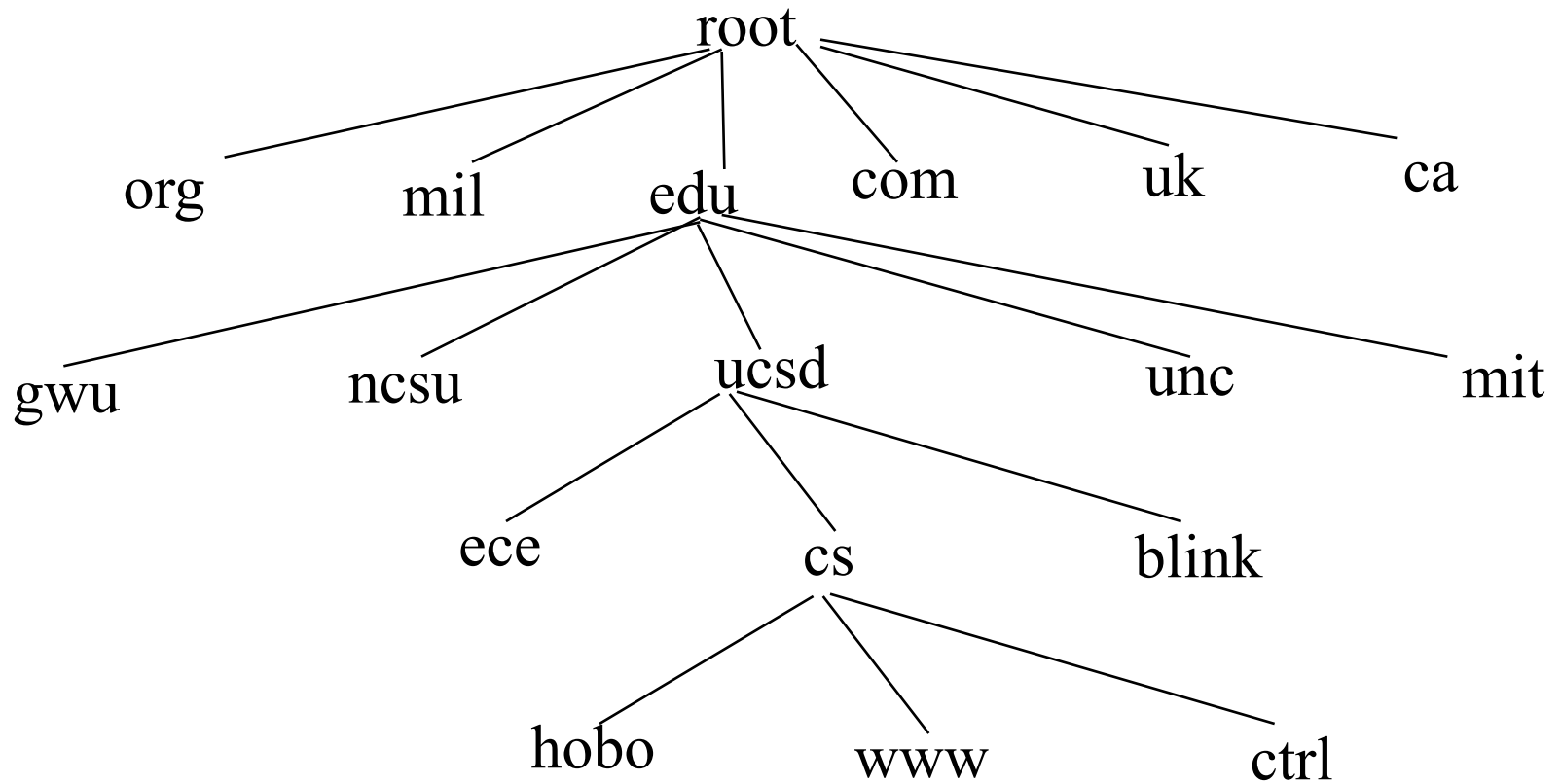
Motivation

- 1982: single hosts.txt file stored and distributed from a central site
- Contained all hostname to IP address mappings
- Centralized control did not fit with distributed management
- Number of hosts changed from number of timesharing systems to number of workstations
 - Organizations to users
 - Exponential resource usage for distributing the file

Domain Name System

- Hierarchical namespace with typed data
- Control delegated in hierarchical fashion
 - Convince node above you to delegate control
- Designed to be extensible w/support for new data types
- 1985: some hosts solely utilize DNS

Hierarchical Design



Domain Name System (DNS)

- Translate human understandable names to machine understandable names

- E.g., `www.cs.ucsd.edu` → `132.239.8.67`
- Hierarchical structure

Every DNS server knows where the “root” is

The root can tell you how to get to `.edu`

`.edu` server can tell you how to find `ucsd.edu`

`ucsd.edu` tells you about `cs.ucsd.edu`

`cs.ucsd.edu` translates `www.cs.ucsd.edu` → `132.239.8.67`

- Caching along the way to improve performance

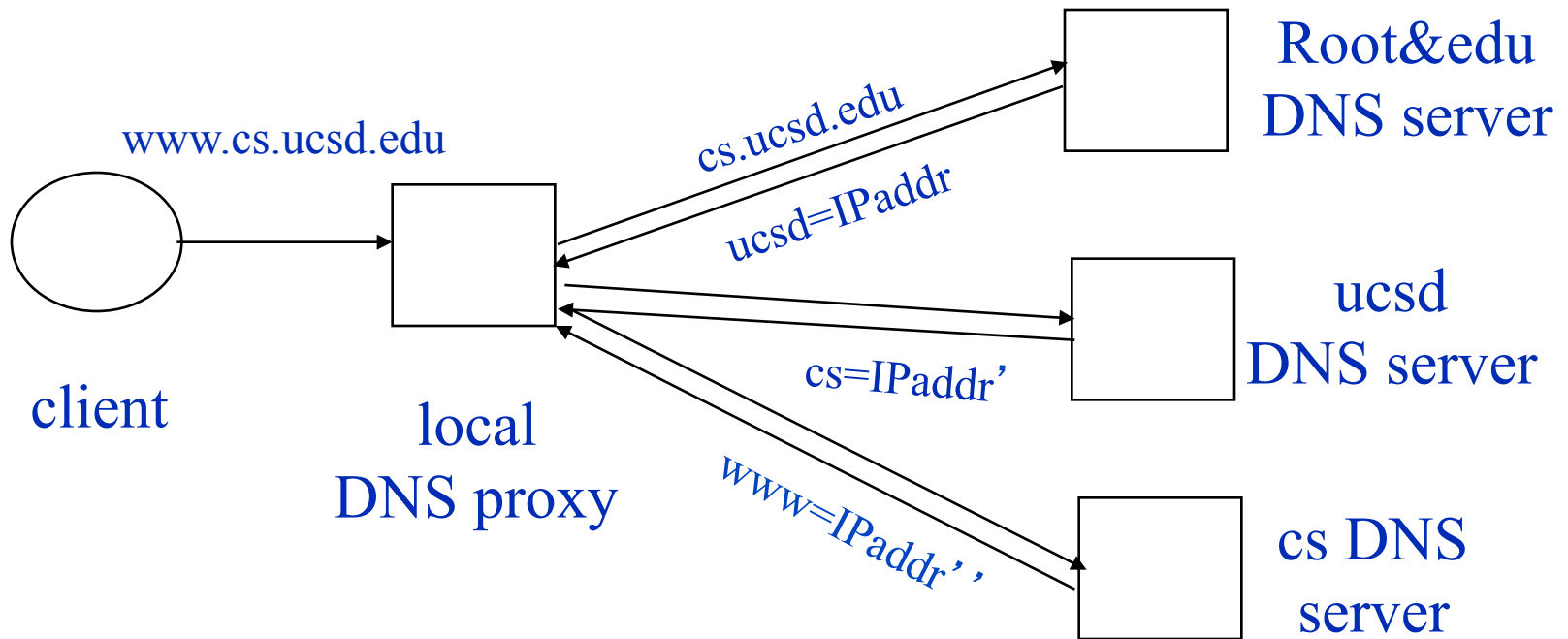
Query Processing

- Query local name server
 - Authoritative/cached answers
- Support both recursive and iterative queries
- If not cached locally, locate server lowest in the hierarchy with entry in local DB
 - In the worst case, contact root (.)
 - Cache locally with TTL

Zones and Caching

- Mechanisms for data distribution
- Zones
 - Provide local autonomy
 - Any contiguous set of nodes in the tree
 - Can be grown to arbitrary size
 - Each domain should provide redundant servers
- Caching
 - Time to live (TTL) associated with each name
 - low value => higher consistency
 - high value => better performance (less traffic)

DNS Lookup Example



1988 Status

- 20k hosts available through DNS (!)
- 30 top level domain names
- SRI managed all non-country top levels
- 7 Root servers
 - 1 query per second, driven by tuning of parameters
 - 50% of 1988 traffic could be eliminated with further tuning
- Query breakdown
 - All info (25-40%)
 - Hostname to address (30-40%)
 - Address to hostname (10-15%)
 - Mail MX record (<10%)

Performance

- Performance worse than designed for (distributed system)
- Clients see 500 ms to 5 second response time for root servers
- Delegated domain performance much worse
 - 3 to 10 seconds, with 30 to 60 seconds not unreasonable
- Negative caching
 - Initially 20-60% of requests were for bad data (old style mail addr)
 - Still 10-50% (typically 25%) for bad data
 - Programs produce steady stream of bad names
 - Negative caching initially an optional feature

Discussion

- Distributed debugging
- Write once, run anywhere? (debug everywhere)
- Mechanisms for pushing code updates
- Self-tuning system
 - Caching responses even if unreasonable (reverse data and TTL)
 - Developers do not want to tune system
 - Especially if they are getting reasonable performance
 - Globally vs. locally optimal
- Security?