

Lecture 6:

CC and

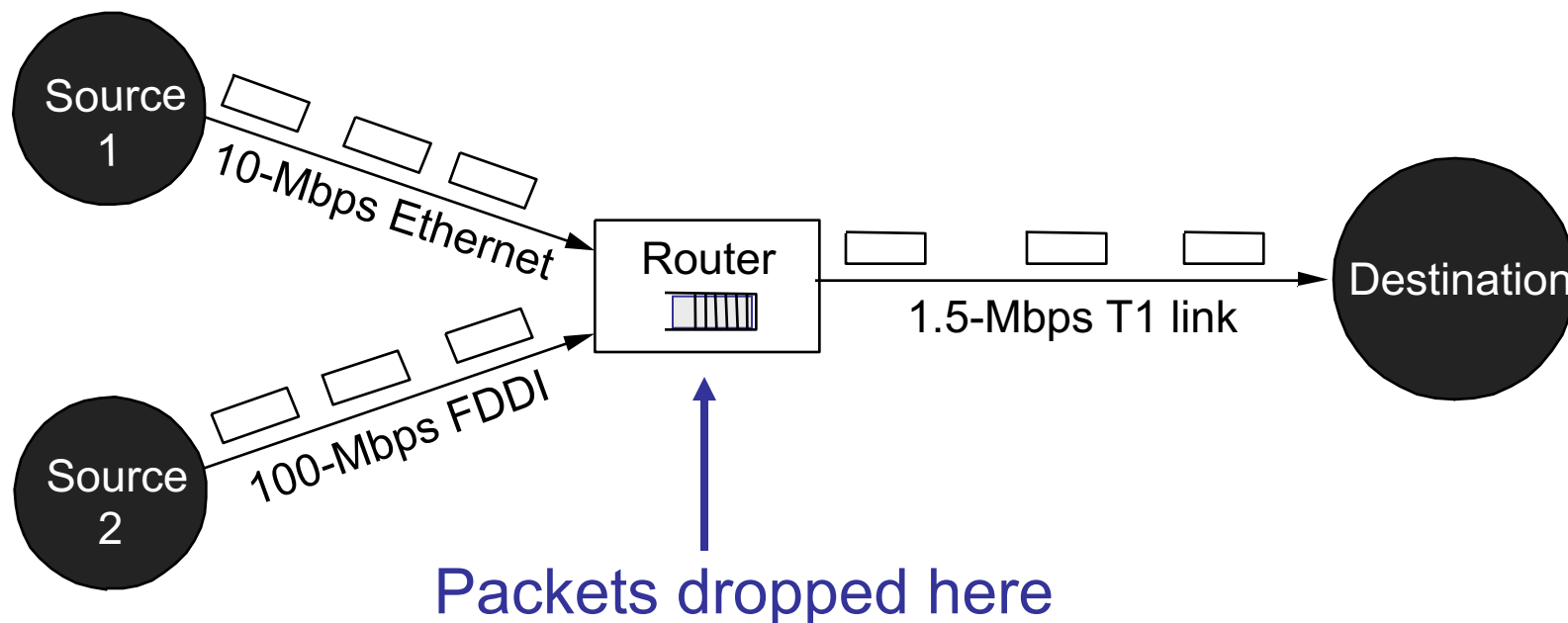
Transport Layer Protocols

CSE 123: Computer Networks

Aaron Schulman

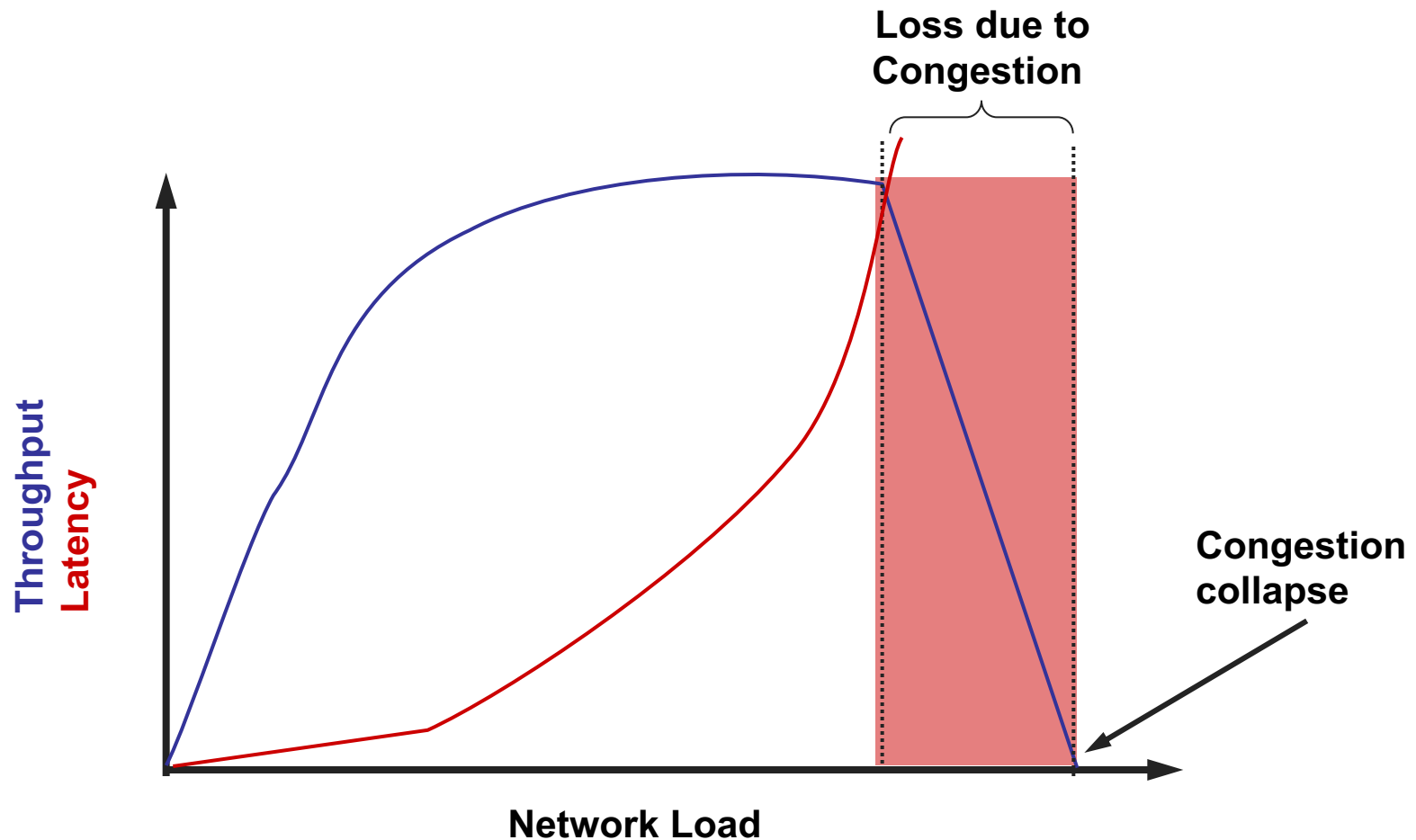


Congestion occurs at routers



- Buffer intended to absorb bursts when input rate $>$ output
- But if sending rate is persistently $>$ drain rate, queue builds
- Dropped packets represent wasted work; goodput $<$ throughput

Drop-Tail Queuing Problem...





Congestion Collapse

- Rough definition: “When an increase in network load produces a decrease in useful work”
- Why does it happen?
 - ◆ Sender sends faster than **bottleneck link** speed
 - ◆ Packets queue up until they are dropped
 - ◆ In response to packets being dropped, sender retransmits
 - ◆ All hosts repeat in steady state...



Mitigation Options

- Increase network resources
 - ◆ More buffers for queuing (this lead to “Bufferbloat”)
 - ◆ Increase link speed
 - ◆ Pros/Cons of these approaches?
- **Reduce network load**
 - ◆ Hosts send data more slowly
 - ◆ How much more slowly?
 - ◆ When to slow down?

Designing a Control Scheme



How?

- Open loop
 - ◆ Explicitly reserve bandwidth in the network in advance
- *Closed loop*
 - ◆ Respond to feedback and adjust bandwidth allocation

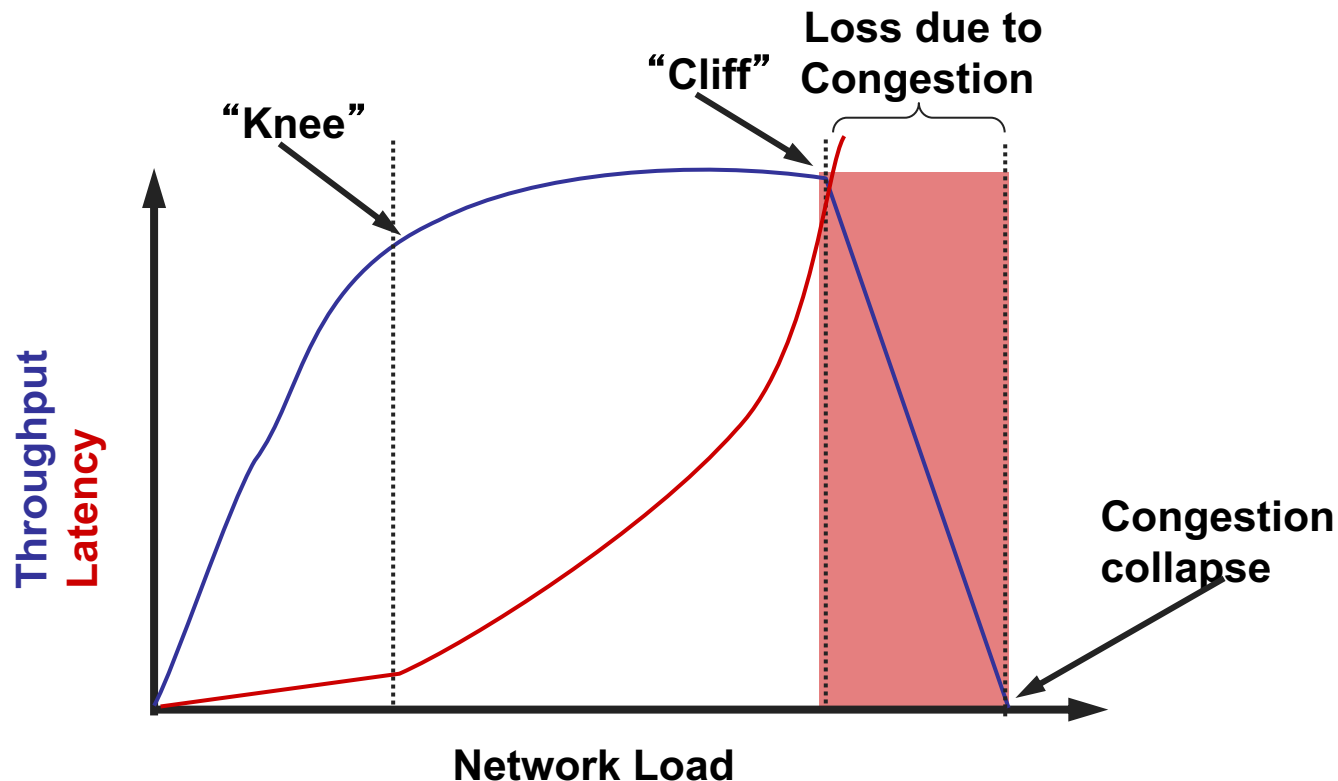
Where?

- Network-based
 - ◆ Network implements and enforces bandwidth allocation
- *Host-based*
 - ◆ Hosts are responsible for controlling their sending rate



Proactive vs. Reactive

- Congestion avoidance: try to stay to the left of the knee
- Congestion control: try to stay to the left of the cliff





Challenges to Address

- How to detect congestion?
- How to limit sending data rate?
- How fast to send?



Detecting Congestion

- Explicit congestion signaling
 - ◆ Source Quench: ICMP message from router to sender
 - ◆ DECEBit / Explicit Congestion Notification (ECN):
 - » Router *marks* packet based on queue occupancy (i.e. indication that packet encountered congestion along the way)
 - » Receiver tells sender if queues are getting too full
- *Implicit congestion signaling*
 - ◆ Packet loss (remember router drop policy?)
 - » Assume congestion is primary source of packet loss
 - » Lost packets indicate congestion
 - ◆ Packet delay
 - » Round-trip time increases as packets queue
 - » Packet inter-arrival time is a function of bottleneck link



Throttling Options

- Sender Window-based (TCP)
 - ◆ Constrain number of outstanding packets allowed in network
 - ◆ Increase window to send faster; decrease to send slower
 - ◆ Pro: Cheap to implement (already have windowing), good failure properties
 - ◆ Con: Creates traffic **bursts** (requires bigger buffers)
- Rate-based (many streaming media protocols)
 - ◆ Two parameters (period, packets)
 - ◆ Allow sending of x packets in period y
 - ◆ Pro: smooth traffic
 - ◆ Con: fine-grained per-connection timers, what if receiver fails?

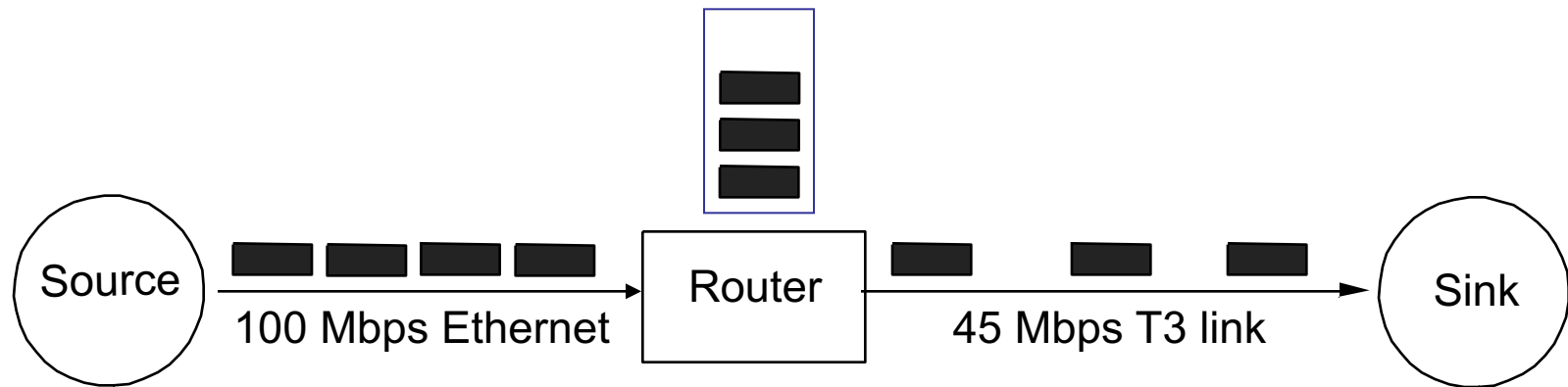


Choosing a Send Rate

- Ideally: Keep equilibrium at “knee” of power curve
 - ◆ Find “knee” somehow
 - ◆ Keep number of packets “in flight” the same
 - ◆ Don’t send a new packet into the network until you know one has left (I.e. by receiving an ACK)
 - ◆ What if you guess wrong, or if bandwidth availability changes?
- Compromise: adaptive approximation
 - ◆ If congestion signaled, reduce sending rate by x
 - ◆ If data delivered successfully, increase sending rate by y
 - ◆ How to relate x and y ? Most choices don’t converge...



TCP's Probing Approach



- Each source independently probes the network to determine how much bandwidth is available
 - ◆ Changes over time, since everyone does this
- Assume that packet loss implies congestion
 - ◆ Since errors are rare; also, requires no support from routers



Basic TCP Algorithm

- Window-based congestion control
 - ◆ Unified congestion control and flow control mechanism
 - ◆ *rwin*: advertised flow control window from receiver
 - ◆ *cwnd*: congestion control window
 - » Estimate of how much outstanding data network can deliver in a round-trip time
 - ◆ Sender can only send $\text{MIN}(rwin, cwnd)$ at any time
- Idea: decrease *cwnd* when congestion is encountered; increase *cwnd* otherwise
 - ◆ Question: how much to adjust?



Congestion Avoidance

- Goal: Adapt to changes in available bandwidth
- Additive Increase, Multiplicative Decrease (AIMD)
 - ◆ Increase sending rate by a constant (e.g. TCP MSS)
 - ◆ Decrease sending rate by a linear factor (e.g. divide by 2)
- Rough intuition for why this works
 - ◆ Let L_i be queue length at time i
 - ◆ In steady state: $L_i = N$, where N is a constant
 - ◆ During congestion, $L_i = N + yL_{i-1}$, where $y > 0$
 - ◆ Consequence: queue size increases multiplicatively
 - » Must reduce sending rate multiplicatively as well



TCP Bandwidth Probing

- TCP uses AIMD to adjust congestion window
 - ◆ Converges to fair share of bottleneck link
 - ◆ Increases modestly in good times
 - ◆ Cuts drastically in bad times
- But what rate should a TCP flow use initially?
 - ◆ Need some initial congestion window
 - ◆ We'd like to TCP to work on all manner of links
 - ◆ Need to span 6+ orders of magnitude, e.g., 10 K to 10 Gbps.
 - ◆ Starting too fast is catastrophic!

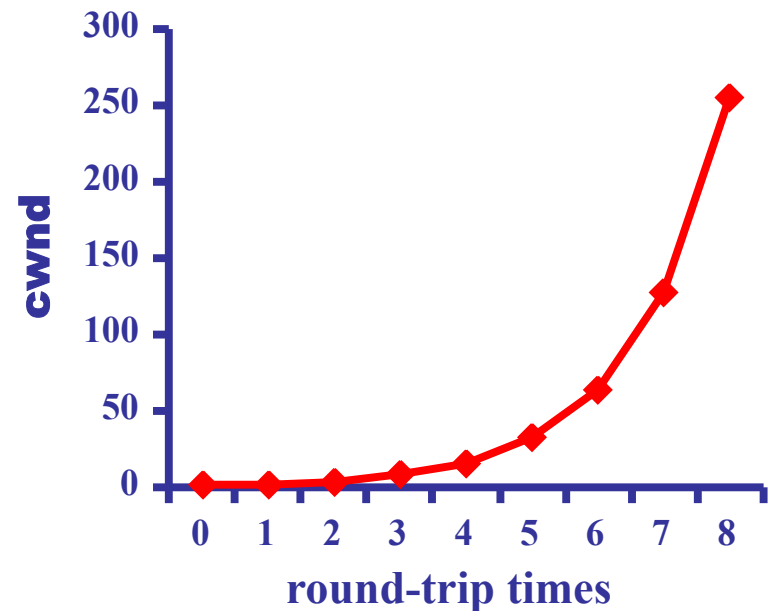
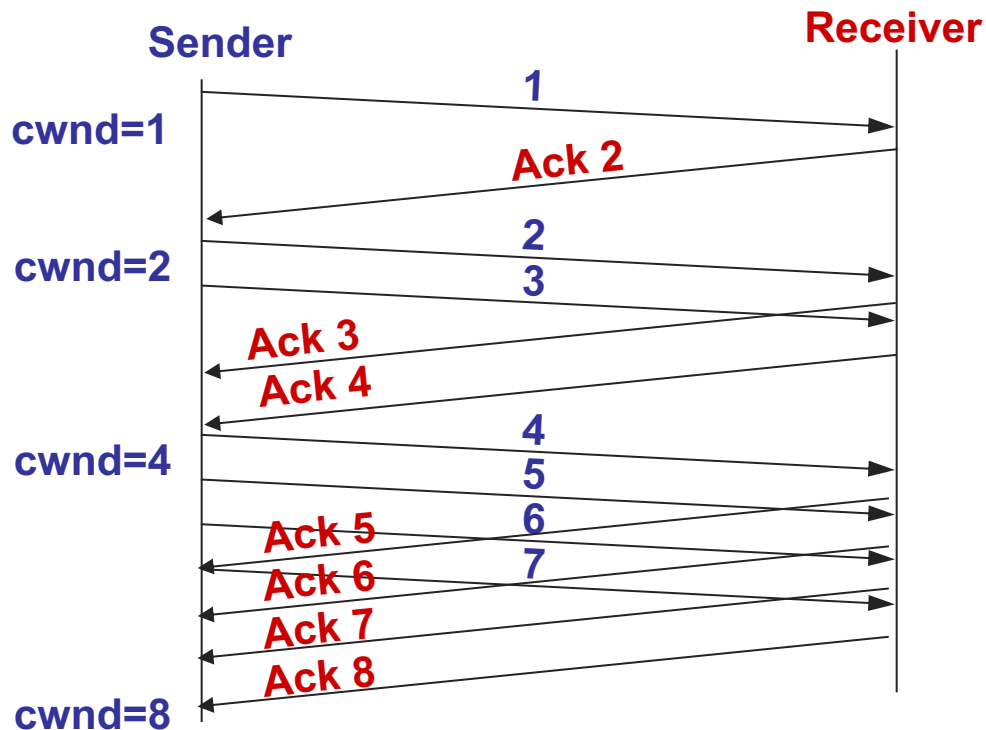
Slow Start



- Goal: quickly find the equilibrium sending rate
- Quickly increase sending rate until congestion detected
 - ◆ Remember last rate that worked and don't overshoot it
- TCP Reno Algorithm:
 - ◆ On new connection, or after timeout, set $cwnd=1$ MSS
 - ◆ For each segment acknowledged, increment $cwnd$ by 1 MSS
 - ◆ If timeout then divide $cwnd$ by 2, and set $ssthresh = cwnd$
 - ◆ If $cwnd \geq ssthresh$ then exit slow start
- Why called slow? Its exponential after all...



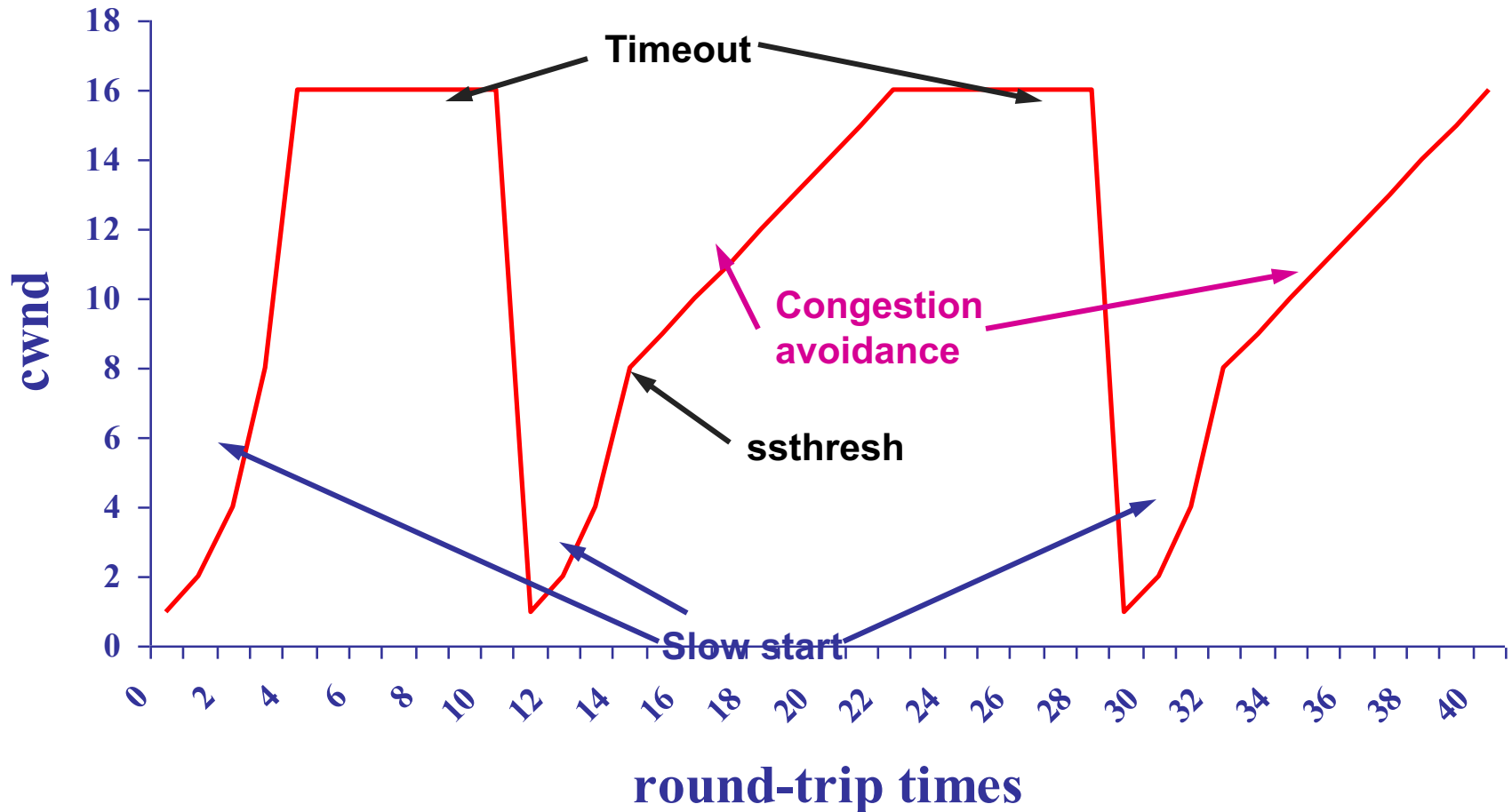
Slow Start Example





Basic Mechanisms

Slow Start + Congestion Avoidance



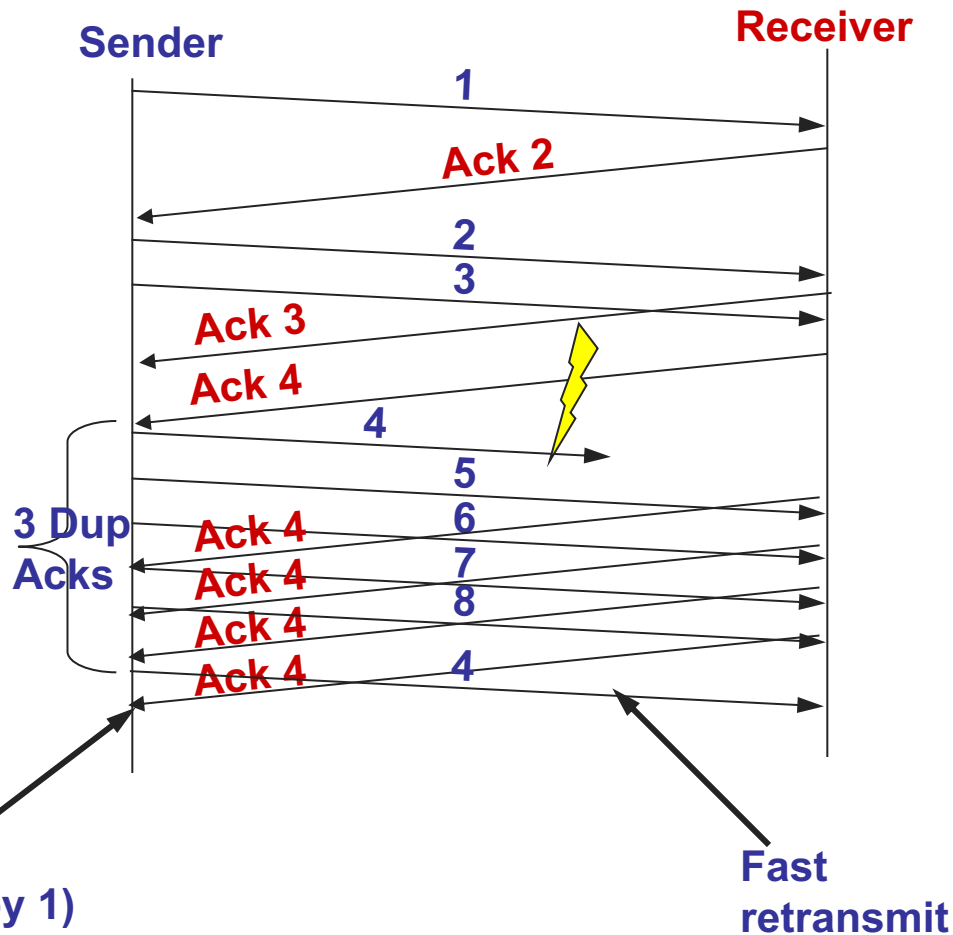
Fast Retransmit & Recovery



- Fast retransmit (we already covered this)
 - ◆ Timeouts are slow (default often 200 ms or 1 second)
 - ◆ When packet is lost, receiver still ACKs last in-order packet
 - ◆ Use 3 duplicate ACKs to indicate a loss; detect losses quickly
- Fast recovery
 - ◆ Goal: avoid stalling after loss
 - ◆ If there are still ACKs coming in, then no need for slow start
 - ◆ If a packet has made it through -> we can send another one
 - ◆ Divide *cwnd* by 2 after fast retransmit
 - ◆ Increment *cwnd* by 1 MSS for each additional duplicate ACK

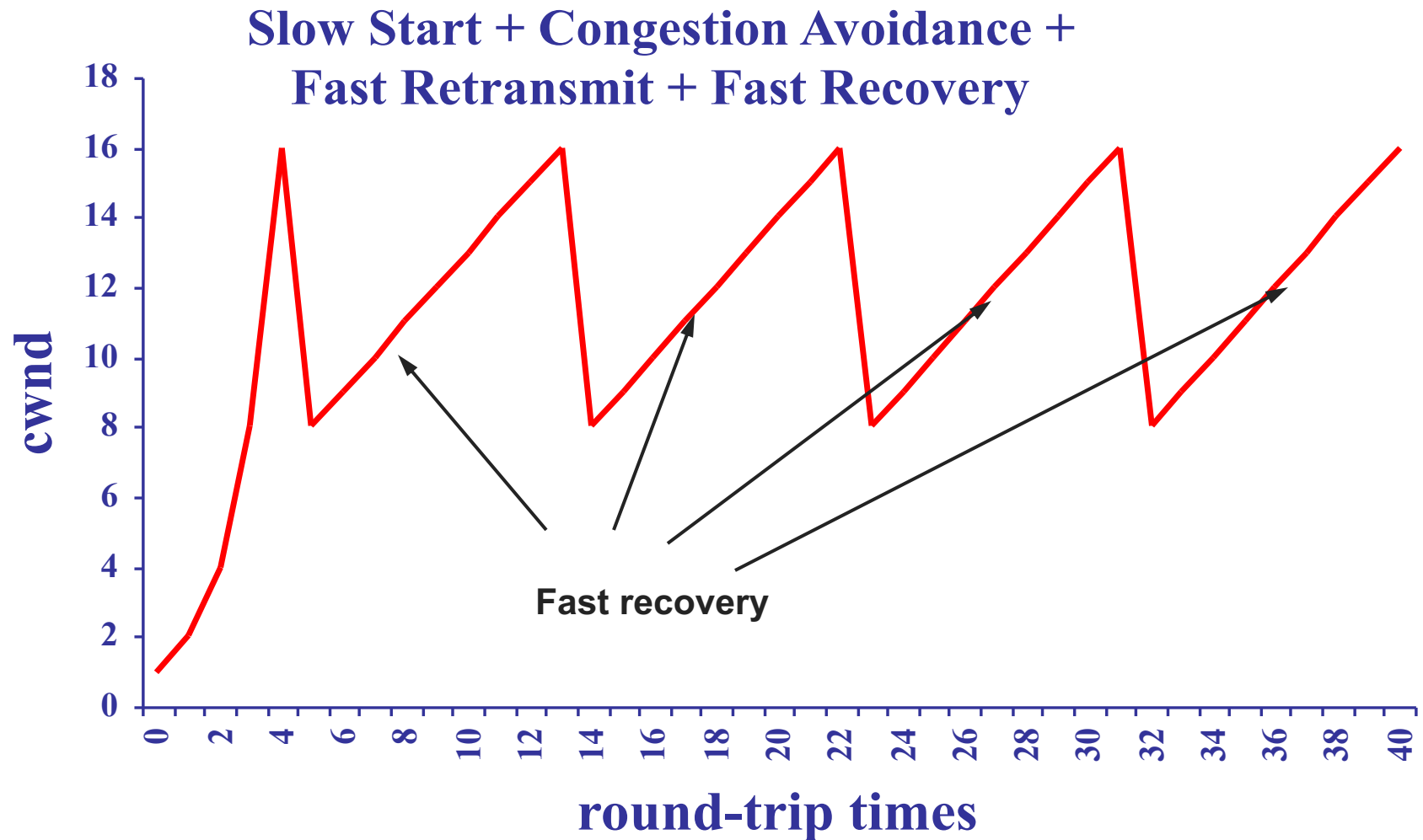


Fast Retransmit Example





More Sophistication





Short Connections

- Short connection: only contains a few pkts
- How do short connections and Slow-Start interact?
 - ◆ What happens when a packet is lost during Slow-Start?
 - ◆ What happens when the SYN is dropped?
- Bottom line: Which packet gets dropped matters a lot
 - ◆ SYN
 - ◆ Slow-Start
 - ◆ Congestion avoidance
- Do you think most flows are short or long?
- Do you think most traffic is in short flows or long flows?

Open Issues



- TCP is designed around the premise of cooperation
 - ◆ What happens to TCP if it competes with a UDP flow?
 - ◆ What if we divide *cwnd* by 3 instead of 2 after a loss?
- There are a bunch of magic numbers
 - ◆ Decrease by 2x, increase by $1/cwnd$, 3 duplicate acks, initial timeout = 3 seconds, etc.
- But overall it works really well!
 - ◆ Still being constantly tweaked...

TCP CC Summary



- TCP Probes the network for bandwidth, assuming that loss signals congestion
- The congestion window is managed with an additive increase/multiplicative decrease policy
 - ◆ It took fast retransmit and fast recovery to get there
 - ◆ Fast recovery keeps pipe “full” while recovering from a loss
- Slow start is used to avoid lengthy initial delays
 - ◆ Ramp up to near target rate, then switch to AIMD

Overview of all Transport Layer Tasks



- Define and provide specific delivery semantics
 - ◆ To which end point (which application/process)?
 - ◆ When?
 - ◆ How?
 - ◆ If?
- Multiplexing different processes between the same hosts
- ... and Reliability (if the application needs it)

What we will talk about next



- Process naming/demultiplexing within a host
- Overview of two popular transport layer protocols
 - ◆ User Datagram Protocol (UDP)
 - » Unordered and unreliable delivery
 - ◆ Transport Control Protocol (TCP)
 - » Three-way handshake
 - » Share state about flow control

Naming Processes/Services



- Process here is an abstract term for your Web browser (HTTP), Email servers (SMTP), hostname translation (DNS)
- How do we identify for remote communication?
 - ◆ Process id or memory address are OS-specific and transient
- So TCP and UDP use **ports**
 - ◆ 16-bit integers representing mailboxes that processes “rent”
 - ◆ Identify process uniquely as (IP address, protocol, port)



Picking Port Numbers

- We still have the problem of allocating port numbers
 - ◆ What port should a Web server use on host *X*?
 - ◆ To what port should you send to contact that Web server?
- Servers typically bind to **well-known** port numbers
 - ◆ e.g., HTTP 80, SMTP 25, DNS 53, ... look in /etc/services
 - ◆ Ports below 1024 traditionally reserved for well-known services
- Clients use OS-assigned temporary (**ephemeral**) ports
 - ◆ Above 1024, recycled by OS when client finished

Establishing well-known ports



Network Working Group
Request for Comments #322
NIC #9609

V. Cerf
J. Postel
UCLA-NMC
26 March 72

Well Known Socket Numbers

The Network Measurement Group would like to establish a network standard socket number for a Process Discard service (not all HOSTs need cooperate.) To do this, it is necessary to know which sockets at each HOST have already been allocated to standard network functions.

At all HOSTs which permit login for services, socket 1 is the Network Logger on which ICP may be performed.

We would like to catalog other sockets which are supposed to be well-known, so we would appreciate having a note or phone call from each HOST Technical Liaison describing the function and socket numbers of network service programs at each HOST. The catalog will be published and we would recommend that it be maintained at NIC.

Dr. Vint Cerf or Jon Postel
3804 Boelter Hall
UCLA Computer Science Department
Los Angeles, California 90024

(213) 825-4864 Vint

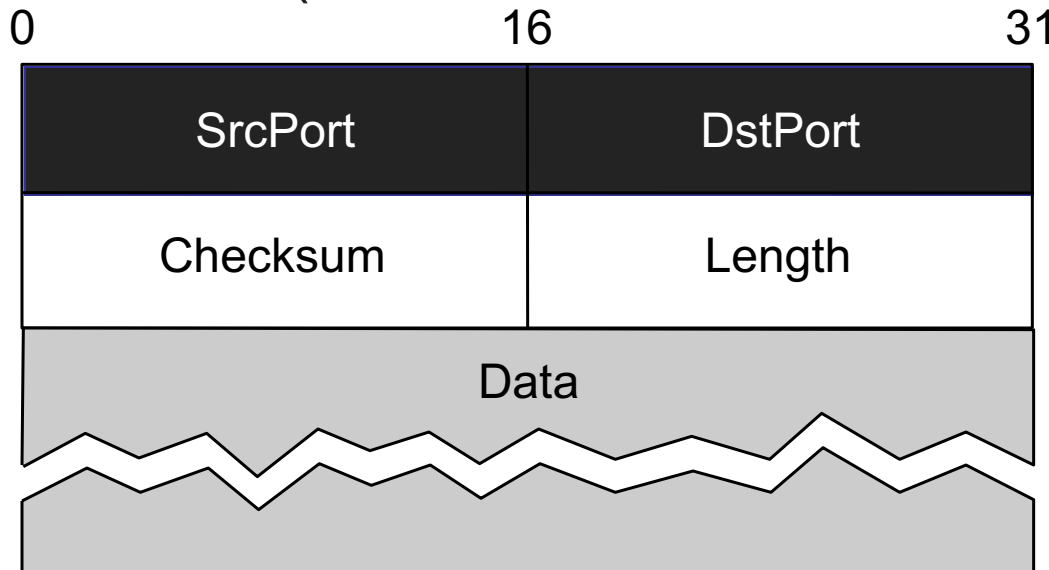
(213) 825-4733 Jon

(213) 825-2368 Secretary

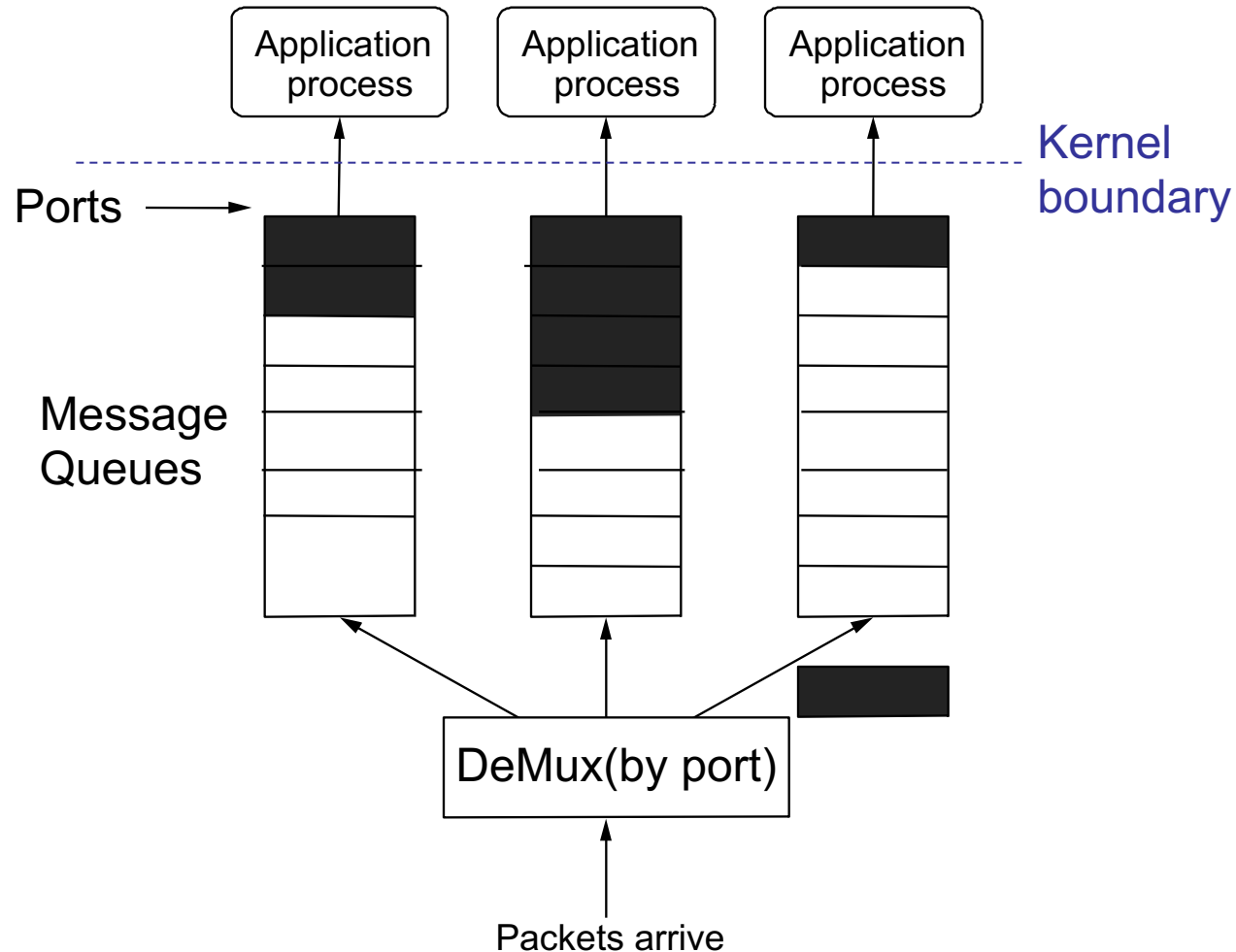
User Datagram Protocol (UDP)



- Provides *unreliable message delivery* between processes
 - ◆ Source port filled in by OS as message is sent (using ephemeral ports)
 - ◆ Destination port identifies UDP delivery queue at endpoint
- Connectionless (no state about who talks to whom)



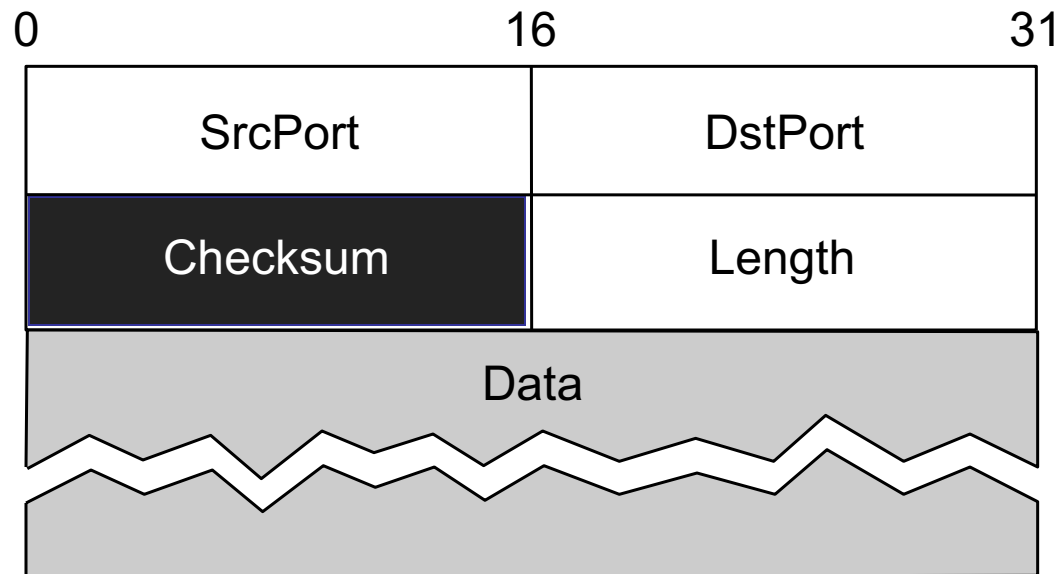
UDP Delivery





UDP Checksum

- UDP includes optional protection against errors
 - ◆ Checksum intended as an end-to-end check on delivery
 - ◆ So it covers data, UDP header, and **IP pseudoheader (history)**





Applications for UDP

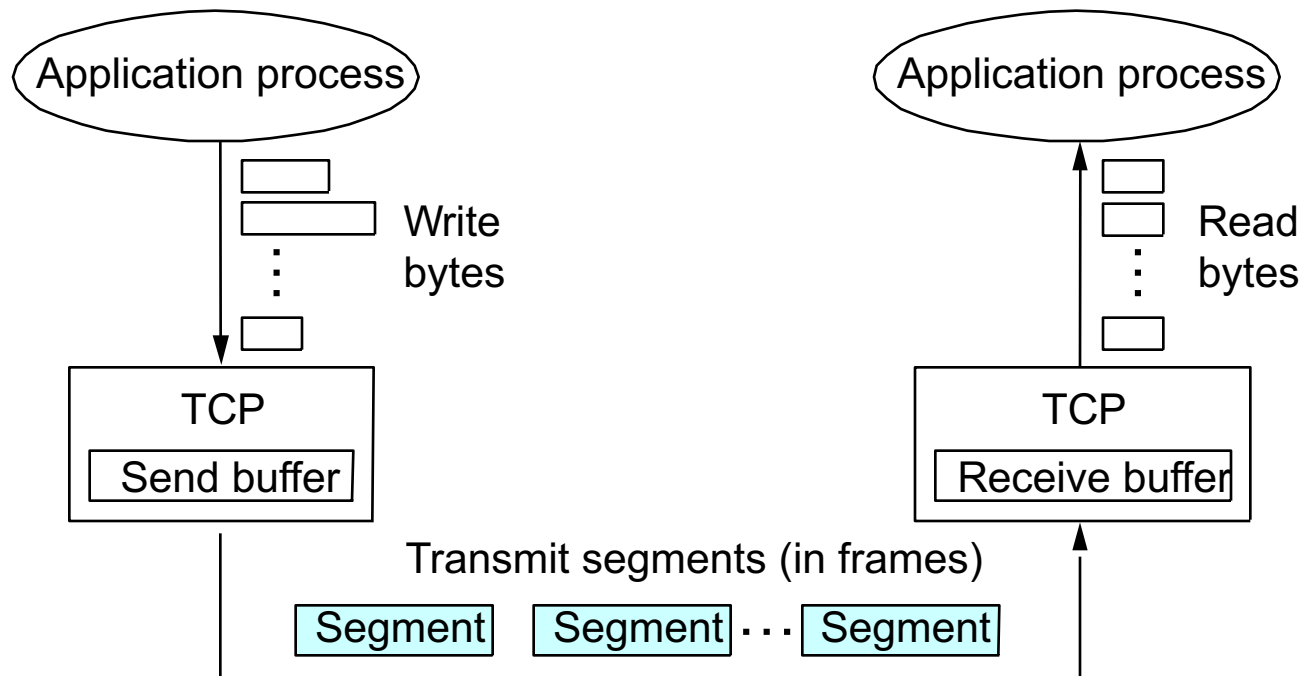
- Streaming media (e.g., live video)
- DNS (Domain Name Service)
- NTP (Network Time Protocol) (synchronizing clocks)
- FPS multi-player video games (e.g., Call of Duty)
- Why might UDP be appropriate for these?

Transmission Control Protocol (TCP)



- Reliable bi-directional **bytestream** between processes
 - ◆ Uses a sliding window protocol!
- Connection-oriented
 - ◆ Conversation between two endpoints with a beginning and end
- Flow control (last lecture) [*receiver window*]
 - ◆ Prevents sender from over-running receiver buffers
 - ◆ Tells sender how many bytes are left in receiver window.
- Congestion control (later in term) [*sender window*]
 - ◆ Prevents sender from over-running network capacity

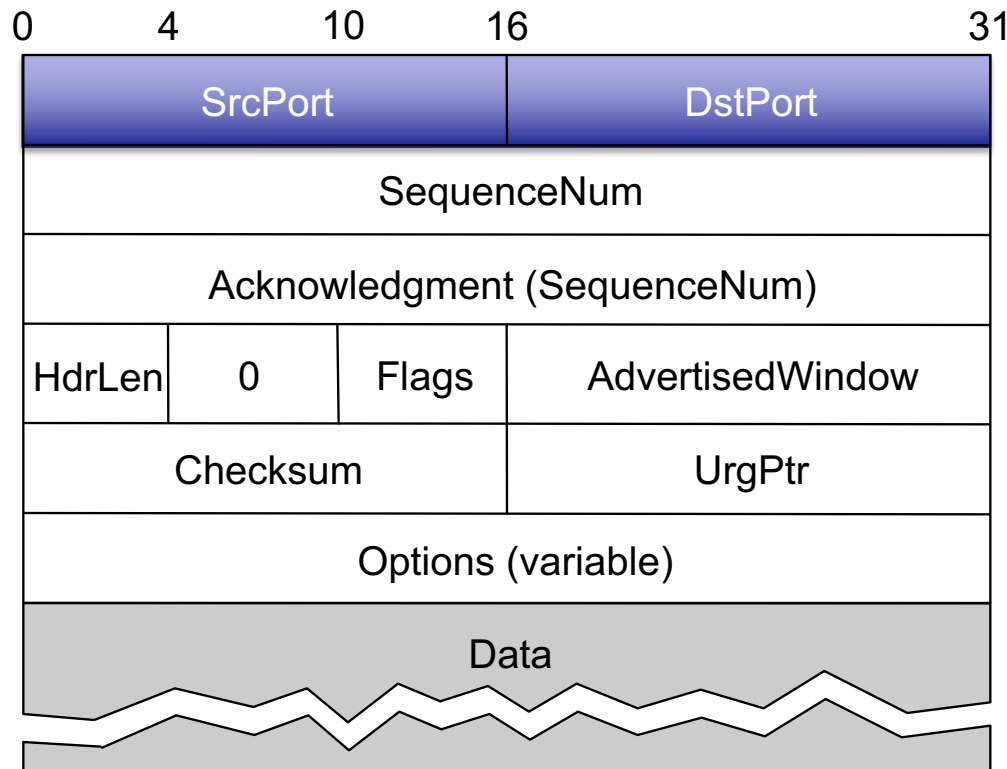
TCP Delivery





TCP Header Format

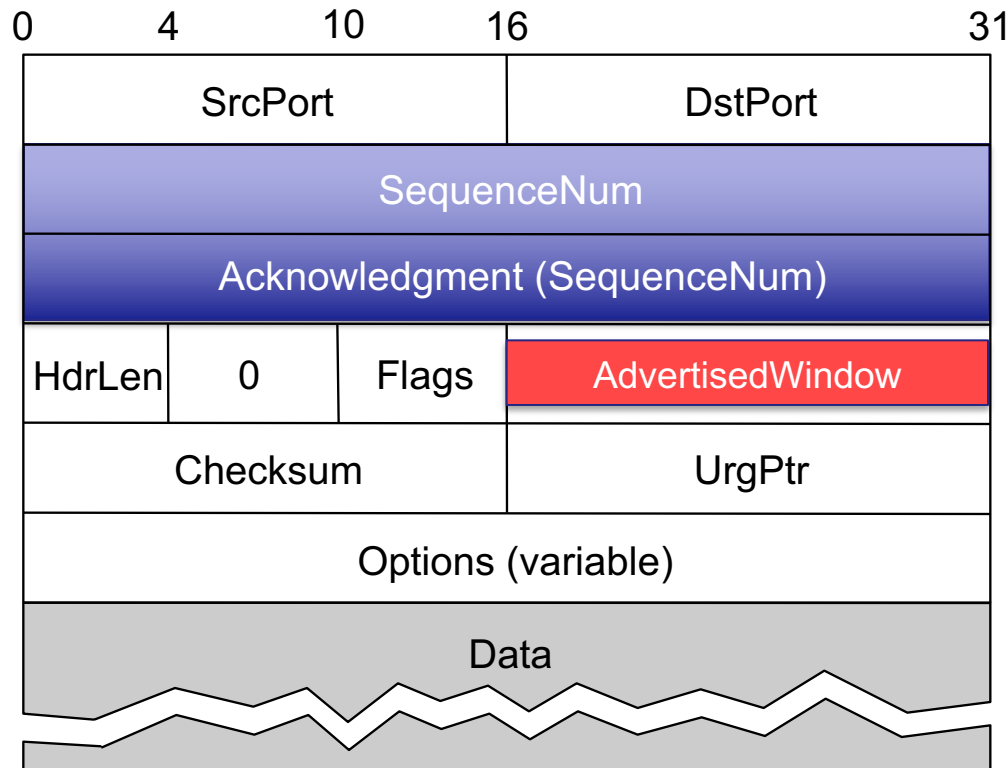
- Ports plus IP addresses identify a connection (4-tuple)





TCP Header Format

- Sequence, Ack numbers used for the sliding window
 - ◆ How big a window? Flow control/congestion control determine

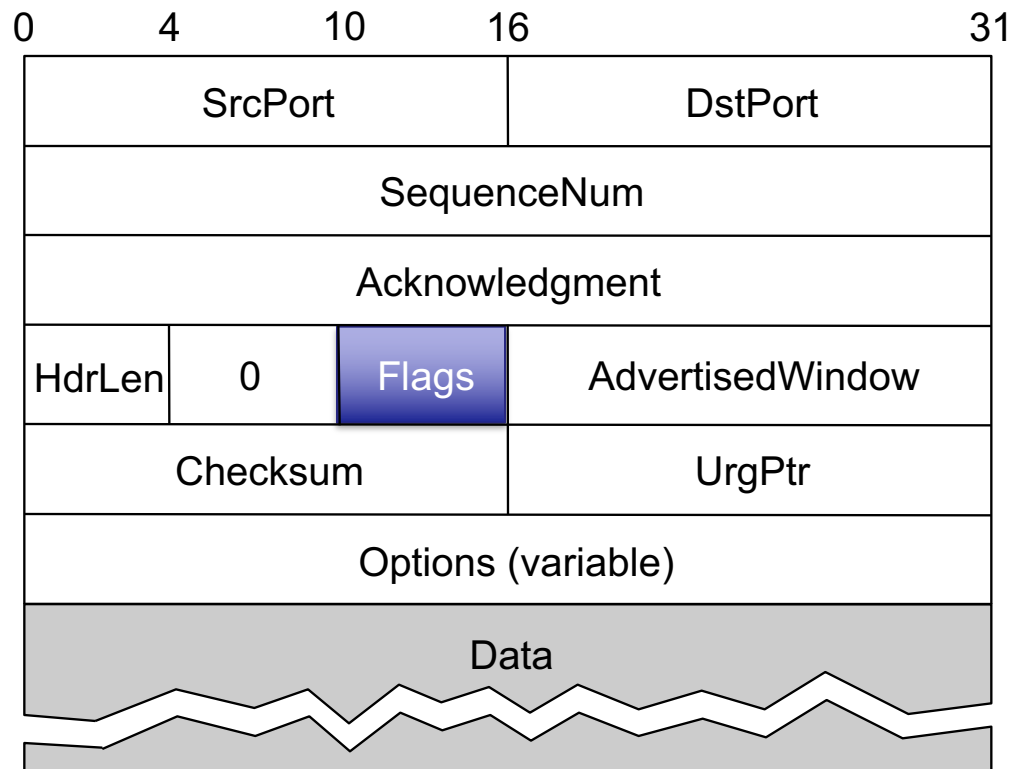


Flow control –
Receiver tells the
transmitter how big its
remaining window is



TCP Header Format

- Flags may be ACK, SYN, FIN, URG, PSH, RST



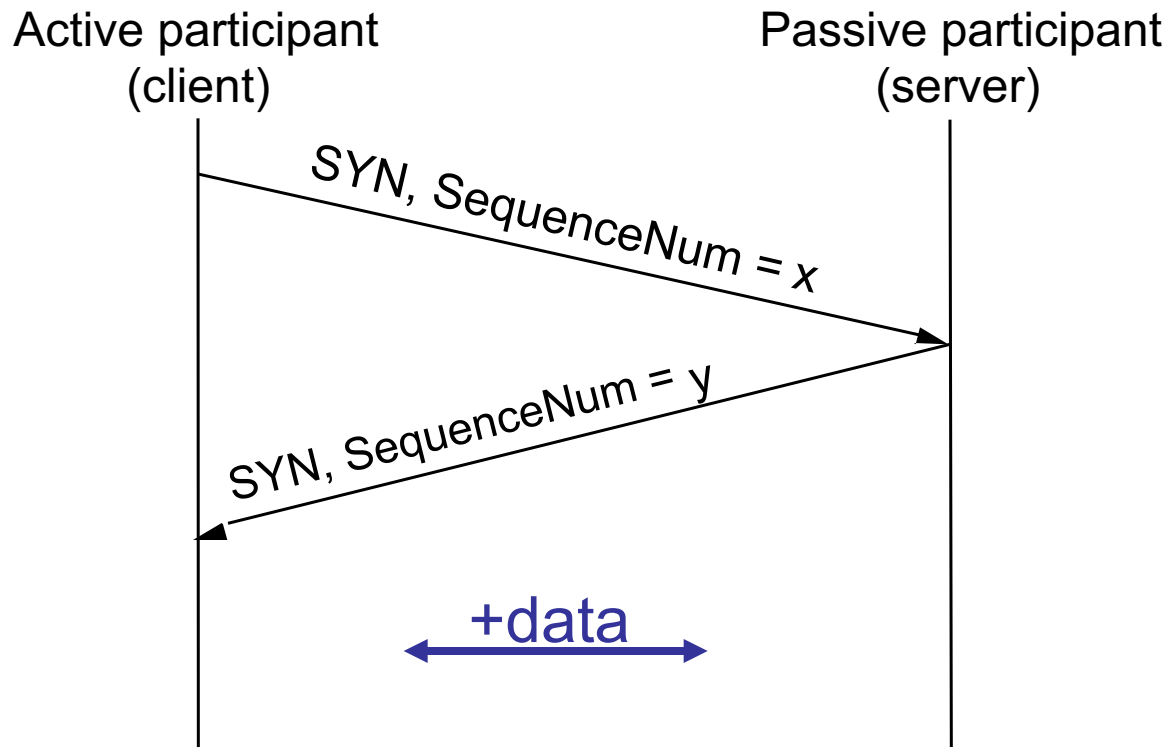
Connection Establishment



- Both sender and receiver must be ready before we start to transfer the data
 - ◆ Sender and receiver need to agree on a set of parameters
 - ◆ Most important: sequence number space in each direction
 - ◆ Lots of other parameters: e.g., the Maximum Segment Size
- Handshake protocols: setup state between two oblivious endpoints
 - ◆ Need to deal with **delayed** and **reordered** packets in the handshake itself!



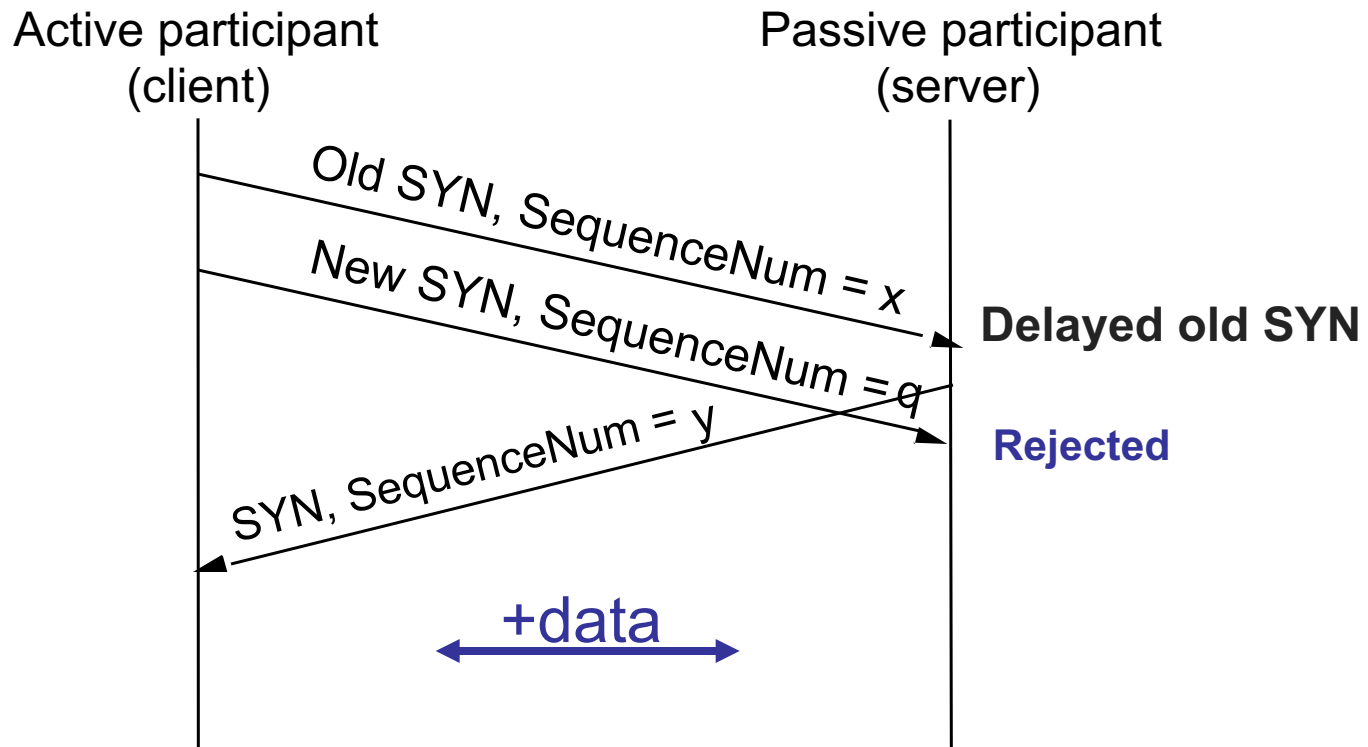
Two-way handshake?



What's wrong here?



Two-way handshake?

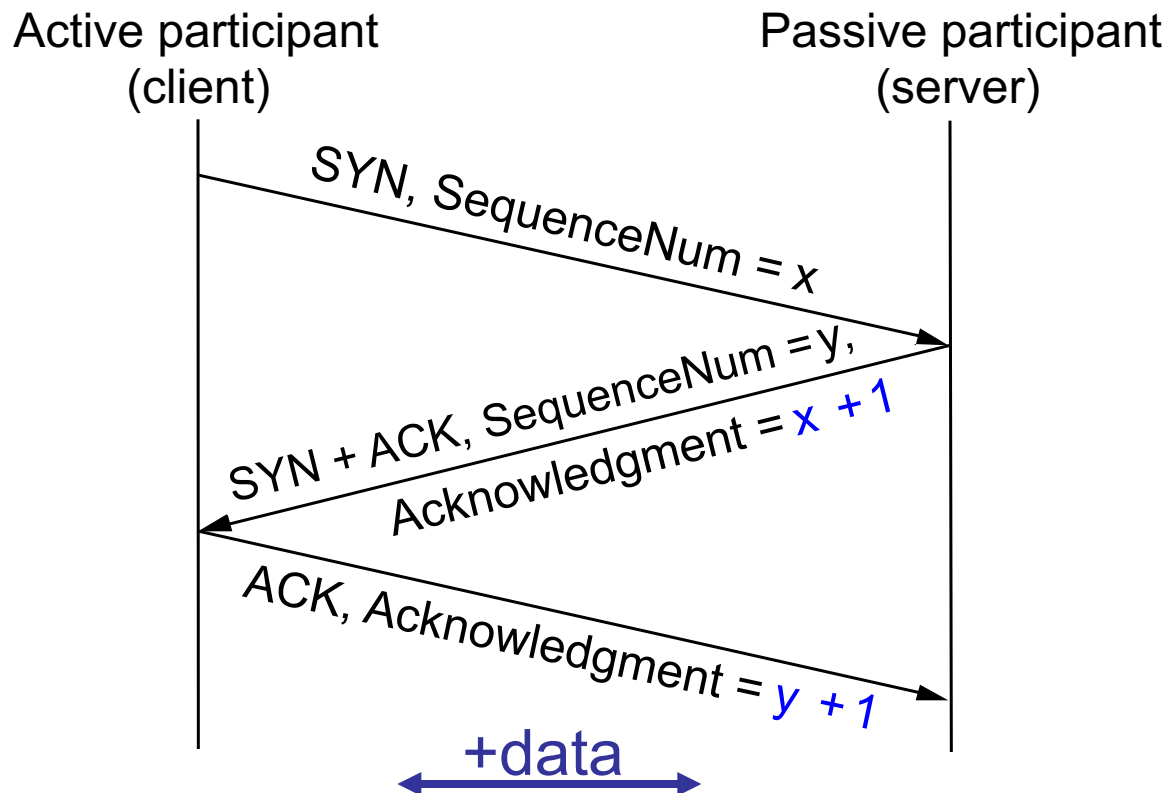


Client and server do not agree they are starting the same connection



Three-Way Handshake

- Opens both directions for transfer (both sides ack)

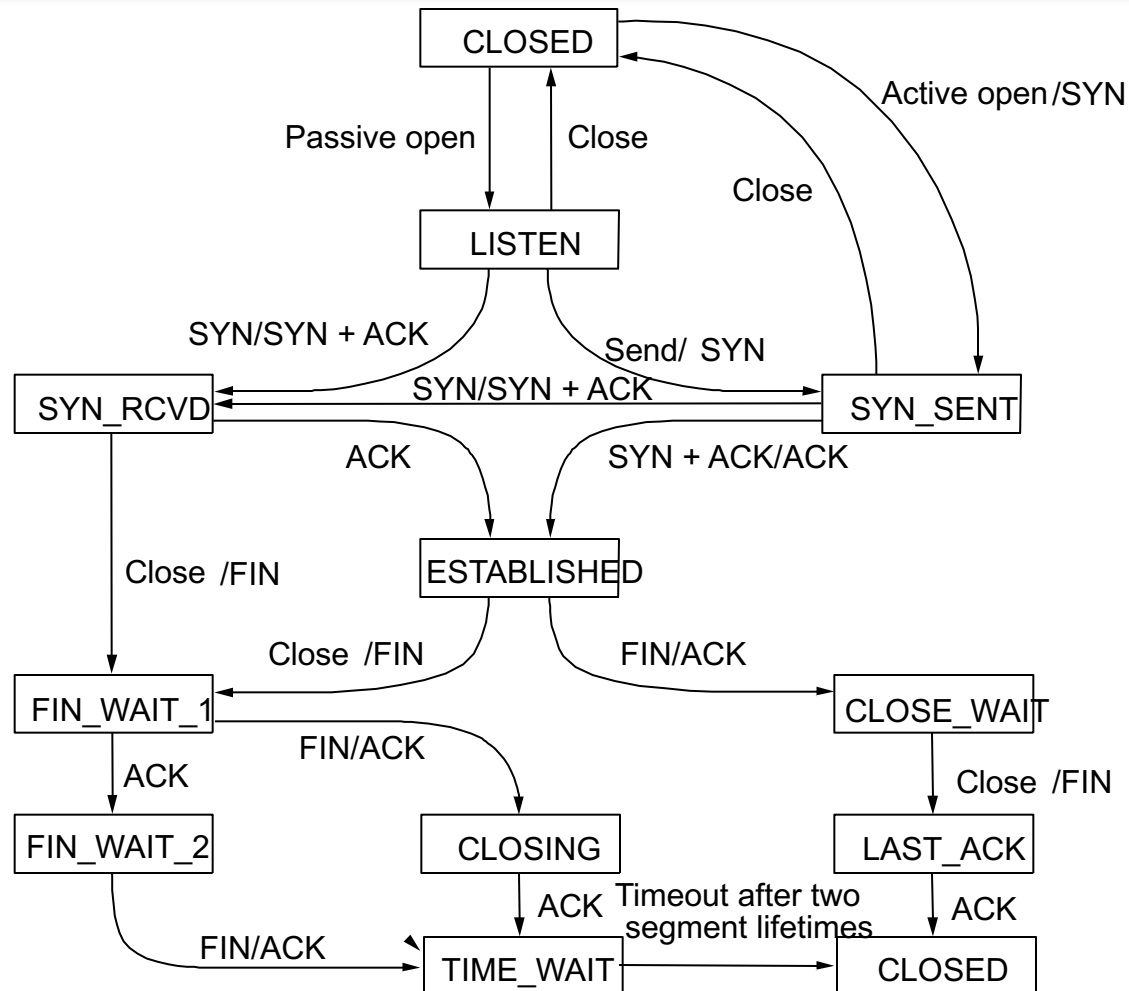




Some Comments

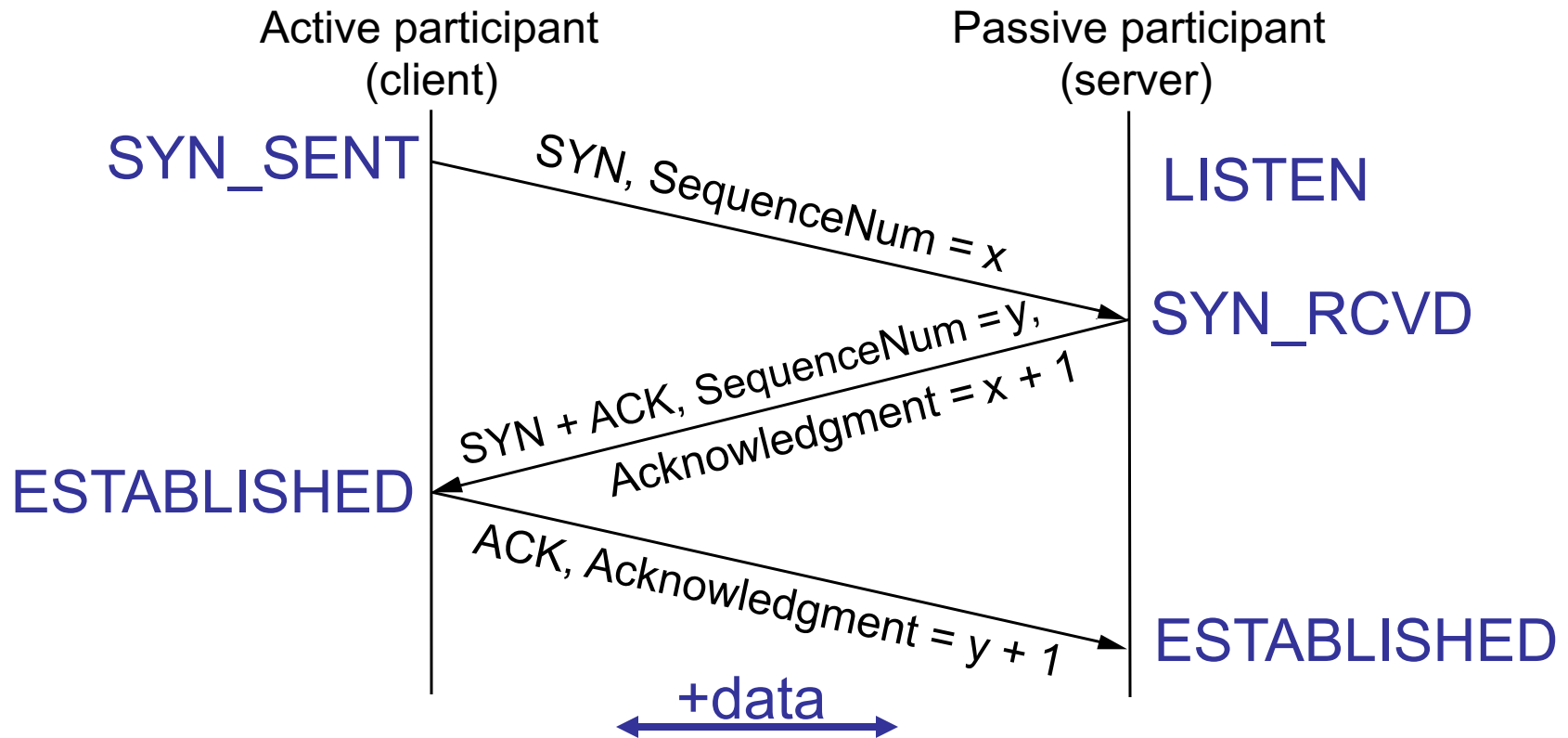
- We could abbreviate this setup, but it was chosen to be robust, especially against delayed duplicates
 - ◆ Three-way handshake from Tomlinson 1975
- Changing Initial sequence numbers (ISNs) minimizes the chance of hosts that crashes getting confused by a previous incarnation of a connection
 - ◆ It also has another benefit for security (see below)
- How to choose ISNs?
 - ◆ Maximize period between reuse
 - ◆ Minimize ability to guess (why?)

TCP State Transitions





Again, with States

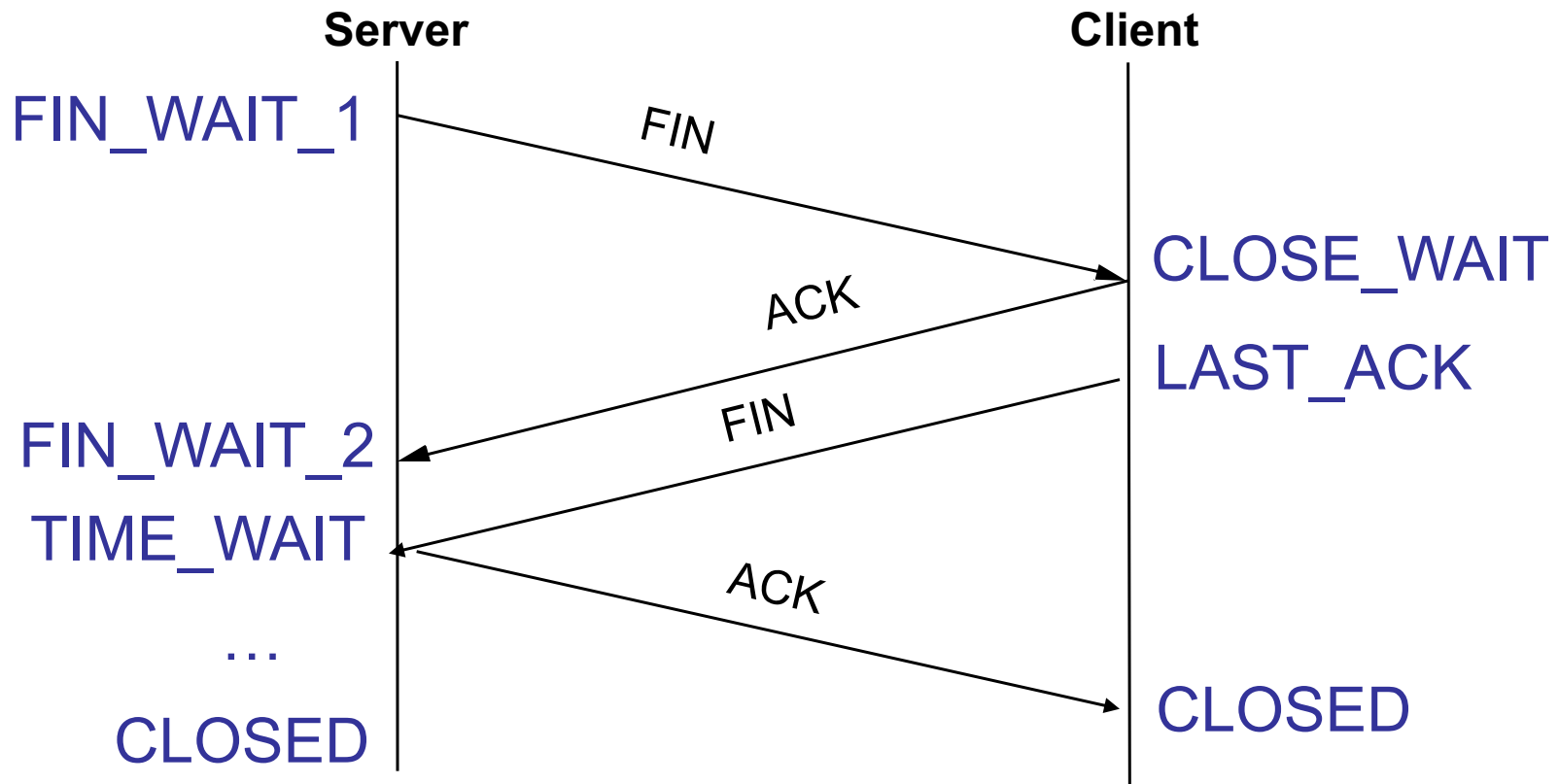




Connection Teardown

- Orderly release by sender and receiver when done
 - ◆ Delivers all pending data and “hangs up”
- Cleans up state in sender and receiver
- TCP provides a “symmetric” close
 - ◆ Both sides shutdown independently, why? You don’t know when the other side will be done!

TCP Connection Teardown





The TIME_WAIT State

- We wait $2 \times \text{MSL}$ (maximum segment lifetime) before completing the close
- Maximum segment lifetime is usually 60 seconds (longest period of time packet can be stuck in network)
 - ◆ Why?
- ACK might have been lost and so FIN will be resent
 - ◆ Could interfere with a subsequent connection
- Real life: Abortive close
 - ◆ Don't wait for $2 \times \text{MSL}$, simply send Reset packet (RST)
 - ◆ Why?



Lots of Icky Details

- Window probes
 - Silly Window Syndrome
 - Nagle's algorithm
 - PAWS
 - Etc...
-
- Steven's books "TCP/IP Illustrated (vol 1,2)" is a great source of information on this



For next time...

- (Back to the link layer)
Connecting links together!
- Read Ch in P&D 3-3.3