# 04832250 – Computer Networks (Honor Track)

## A Data Communication and Device Networking Perspective

### Module 5: End-to-End Transport

**Prof. Chenren Xu（许辰人）**

Center for Energy-efficient Computing and Applications

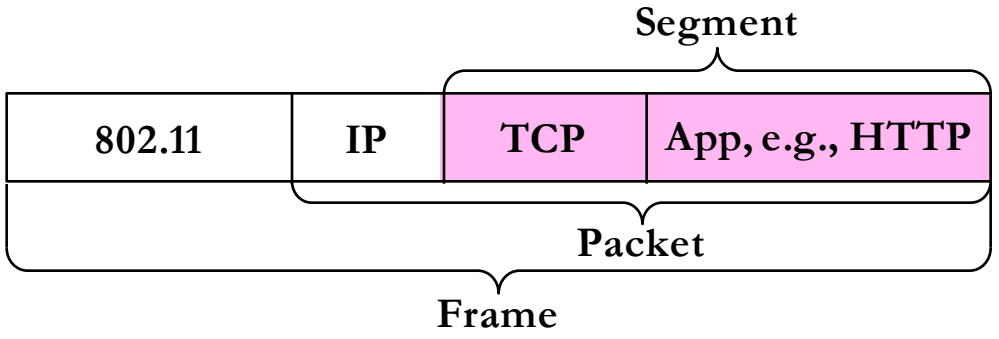
Computer Science, Peking University

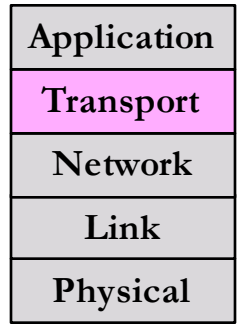
chenren@pku.edu.cn

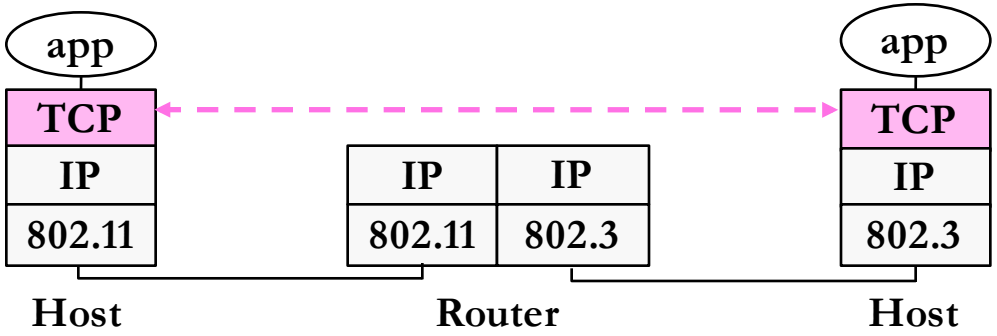
http://soar.pku.edu.cn/

# Where we are in the Course

- Starting the Transport Layer!
  - Builds on the network layer to deliver data across networks for applications with the desired reliability or quality
- Recall
  - Transport layer provides end-to-end connectivity across the network
  - Segments carry application data across the network
  - Segments are carried within packets within frames



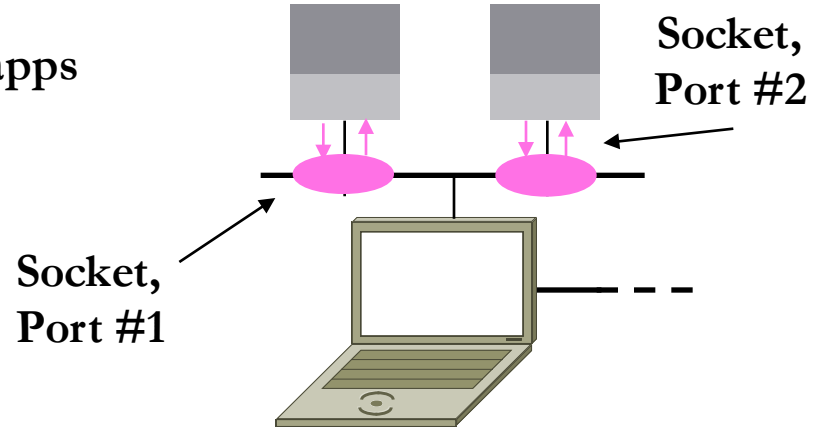

| TCP (Streams) | UDP (Datagrams) |
|---|---|
| Connections | Datagrams |
| Bytes are delivered reliably, and in order | Messages may be lost, reordered, duplicated |
| Arbitrary length content | Limited message size |
| Flow control matches sender to receiver | Can send regardless of receiver state |
| Congestion control matches sender to network | Can send regardless of network state |

# Topics

- Service Models
  - Socket API and ports
  - Datagrams, Streams
- User Datagram Protocol (UDP)
- Transmission Control Protocol (TCP)
  - Connections
  - Sliding Window
  - Flow control
  - Retransmission timers
  - Congestion control

# Socket API

- Simple abstraction to use the network
  - The "network" API (really Transport service) used to write all Internet apps
  - Part of all major OSes and languages; originally Berkeley (Unix) ~1983
- Supports both Internet transport services (Streams and Datagrams)
- Sockets let apps attach to the local network at different ports
- Same API used for Streams and Datagrams




| | Primitive | Meaning |
|---|---|---|
| | SOCKET | Create a new communication endpoint |
| | BIND | Associate a local address (port) with a socket |
| Only needed for Streams | LISTEN | Announce willingness to accept connections |
| Only needed for Streams | ACCEPT | Passively establish an incoming connection |
| Only needed for Streams | CONNECT | Actively attempt to establish a connection |
| To/From forms for Datagrams | SEND (TO) | Send some data over the socket |
| To/From forms for Datagrams | RECEIVE (FROM) | Receive some data over the socket |
| To/From forms for Datagrams | CLOSE | Release the socket |

# Ports

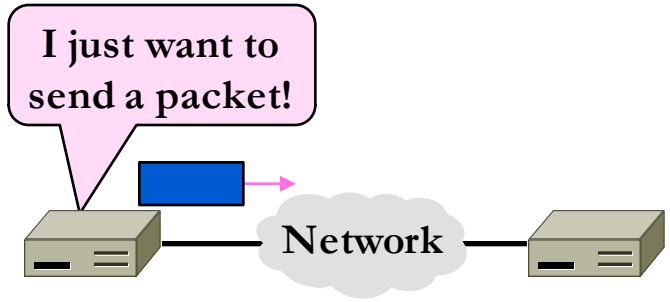
- Application process is identified by the tuple IP address, protocol, and port
  - Ports are 16-bit integers representing local "mailboxes" that a process leases
- Servers often bind to "well-known ports"
  - <1024, require administrative privileges
- Clients often assigned "ephemeral" ports
  - Chosen by OS, used temporarily

- Some Well-Known Ports

| Port | Protocol | Use |
|---|---|---|
| 20, 21 | FTP | File transfer |
| 22 | SSH | Remote login, replacement for Telnet |
| 25 | SMTP | Email |
| 80 | HTTP | World Wide Web |
| 110 | POP-3 | Remote email access |
| 143 | IMAP | Remote email access |
| 443 | HTTPS | Secure Web (HTTP over SSL/TLS) |
| 543 | RTSP | Media player control |
| 631 | IPP | Printer sharing |

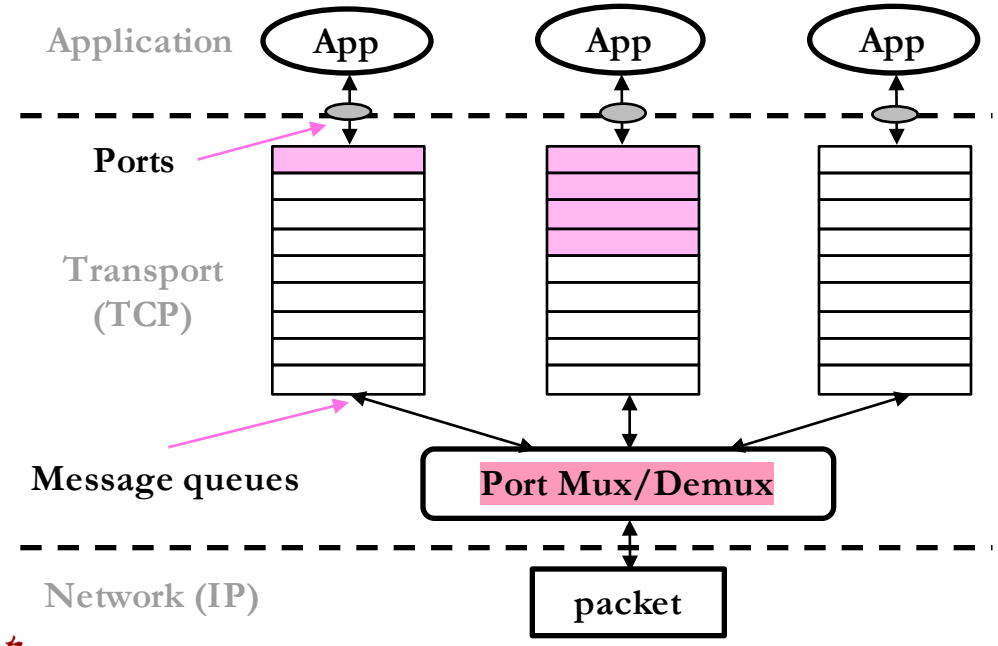
# Topics

- Service Models
  - Socket API and ports
  - Datagrams, Streams
- User Datagram Protocol (UDP)
- Transmission Control Protocol (TCP)
  - Connections
  - Sliding Window
  - Flow control
  - Retransmission timers
  - Congestion control

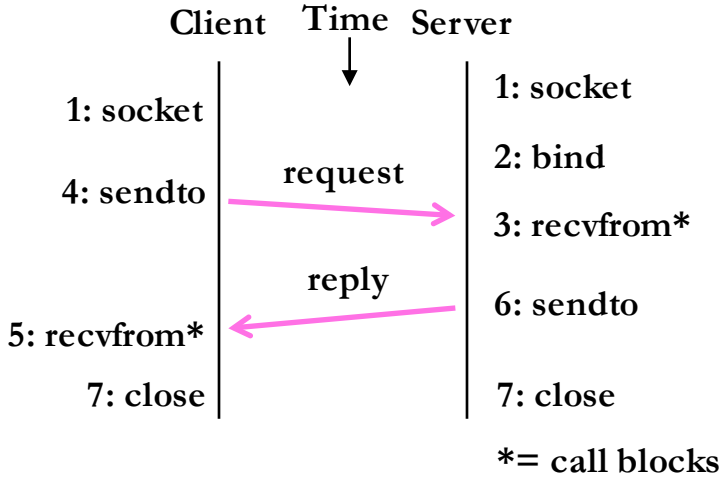


# User Datagram Protocol (UDP)

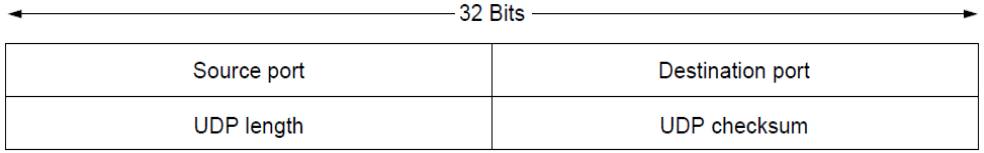
- A shim layer on packets
- Used by apps that don't want reliability or bytestreams
  - Voice-over-IP (unreliable)
  - DNS, RPC (message-oriented)
  - DHCP (bootstrapping)
- UDP Buffering




- Datagram Sockets

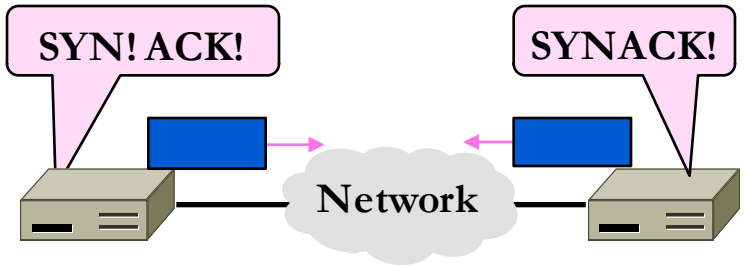



- UDP Header
  - Uses ports to identify sending and receiving application processes
  - Datagram length up to 64K
  - Checksum (16 bits) for reliability



# Topics

- Service Models
  - Socket API and ports
  - Datagrams, Streams
- User Datagram Protocol (UDP)
- Transmission Control Protocol (TCP)
  - Connections (establish and release)
  - Sliding Window
  - Flow control
  - Retransmission timers
  - Congestion control

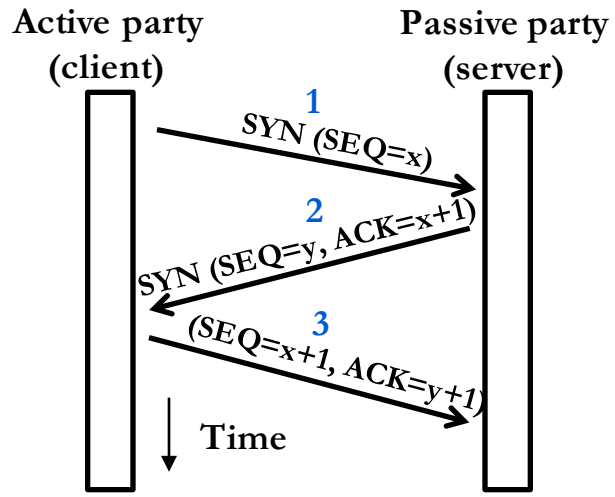


# Connection Establishment

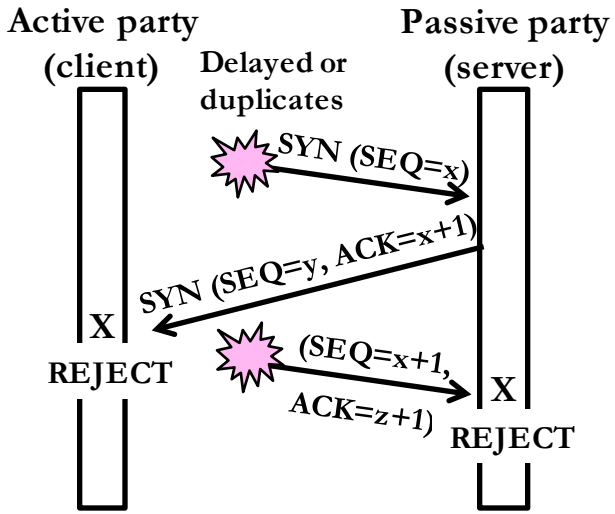
- How to set up connections
  - We'll see how TCP does it
- Both sender and receiver must be ready before we start the transfer of data
  - Need to agree on a set of parameters
  - e.g., the Maximum Segment Size (MSS)
- This is signaling
  - It sets up state at the endpoints
  - Like "dialing" for a telephone call

# Three-Way Handshake

- Used in TCP
- Opens connection for data in both directions
- Each side probes the other with a fresh Initial Sequence Number
  - Sends on a SYNchronize segment
  - Echo on an ACKnowledge segment
  - SYNs are retransmitted if lost
- Sequence and ack numbers carried on further (data) segments



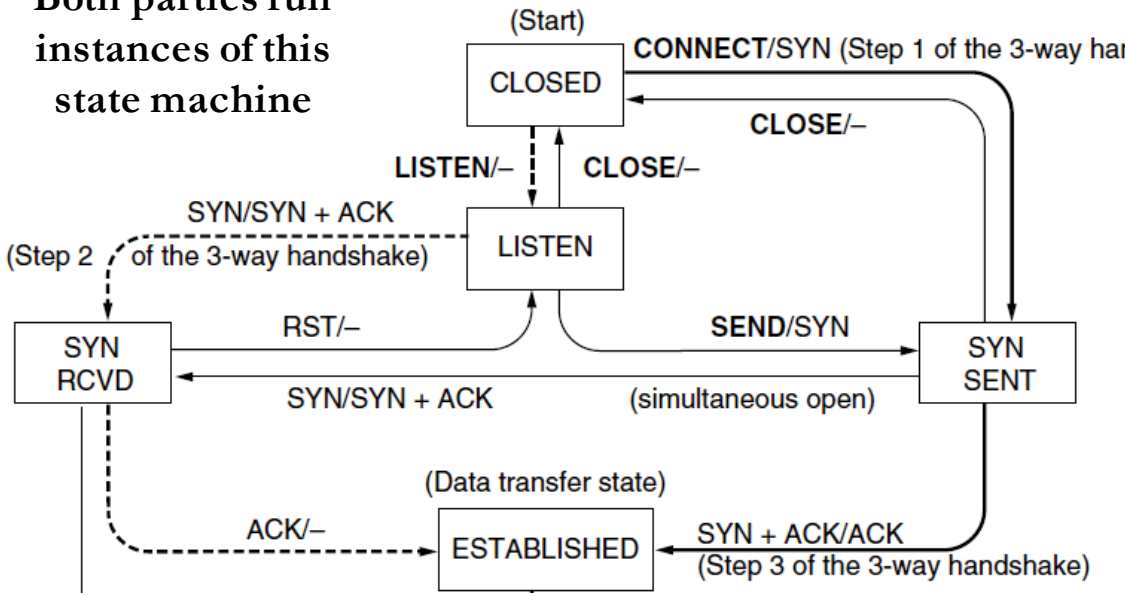

- Suppose delayed, duplicate copies of the SYN and ACK arrive at the server!
  - Connection will be cleanly rejected on both sides ☺
  - DoS attack …

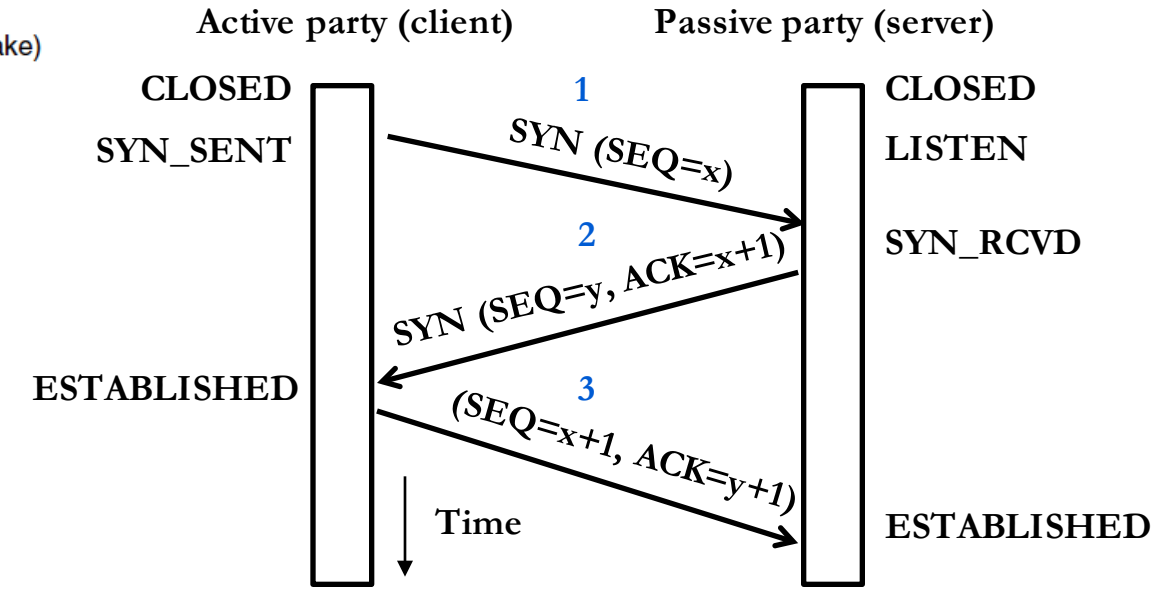


# TCP Connection State Machine (Connection Establishment)

- Captures the states (rectangles) and transitions (arrows)
  - A/B means event A (active or passive) triggers the transition, with action B

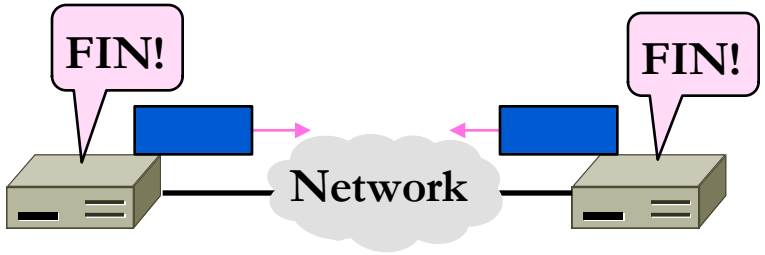
Both parties run instances of this state machine



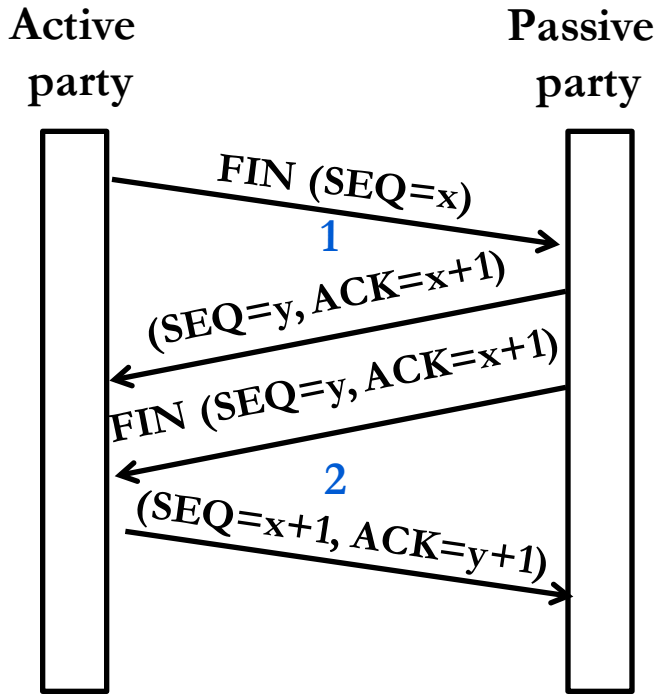




- Finite state machines are a useful tool to specify and check the handling of all cases that may occur
- TCP allows for simultaneous open
  - i.e., both sides open at once instead of the client-server pattern

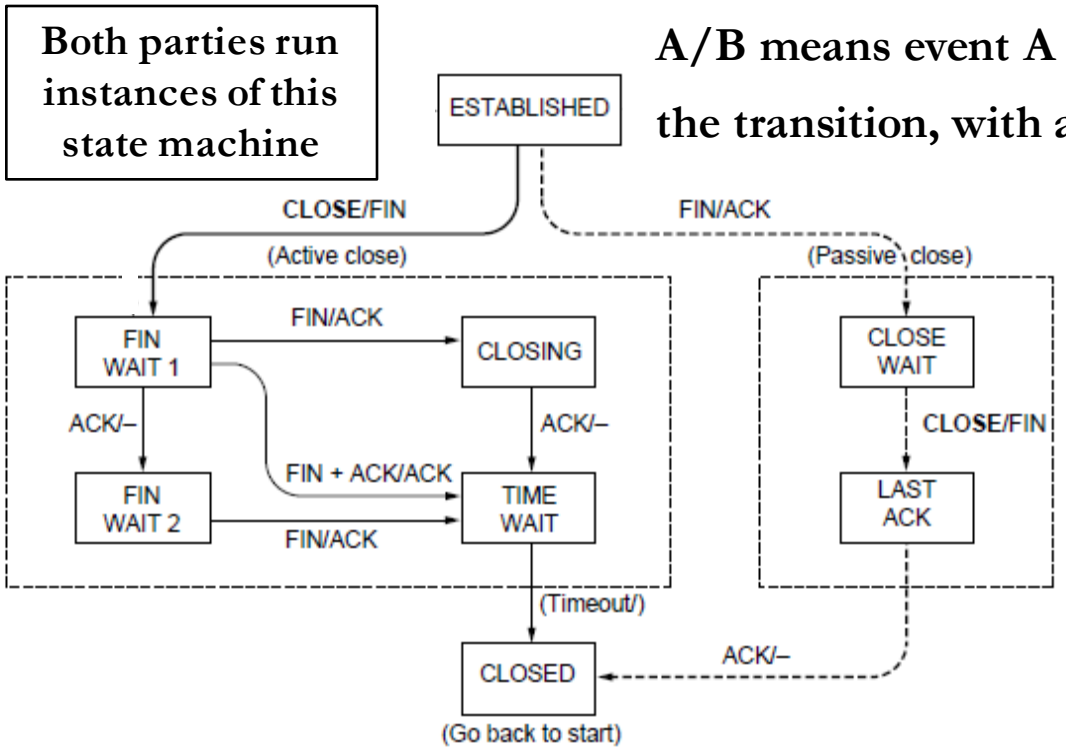
# Connection Release

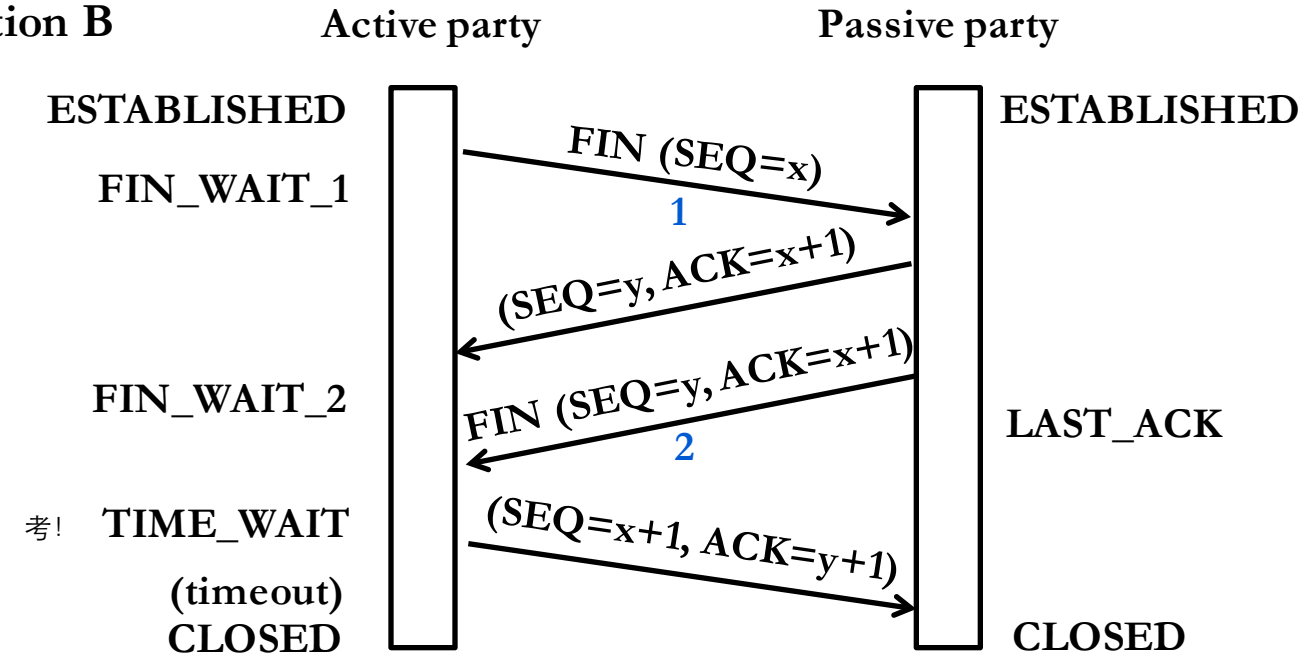



- How to release connections
  - We'll see how TCP does it
- Orderly release by both parties when done
  - Delivers all pending data and "hangs up"
  - Cleans up state in sender and receiver
- Key problem is to provide reliability while releasing
  - TCP uses a "symmetric" close in which both sides shutdown independently
- TCP Connection Release
  - Two steps:
    - Active sends FIN(x), ACKs
    - Passive sends FIN(y), ACKs
    - FINs are retransmitted if lost
  - Each FIN/ACK closes one direction of data transfer



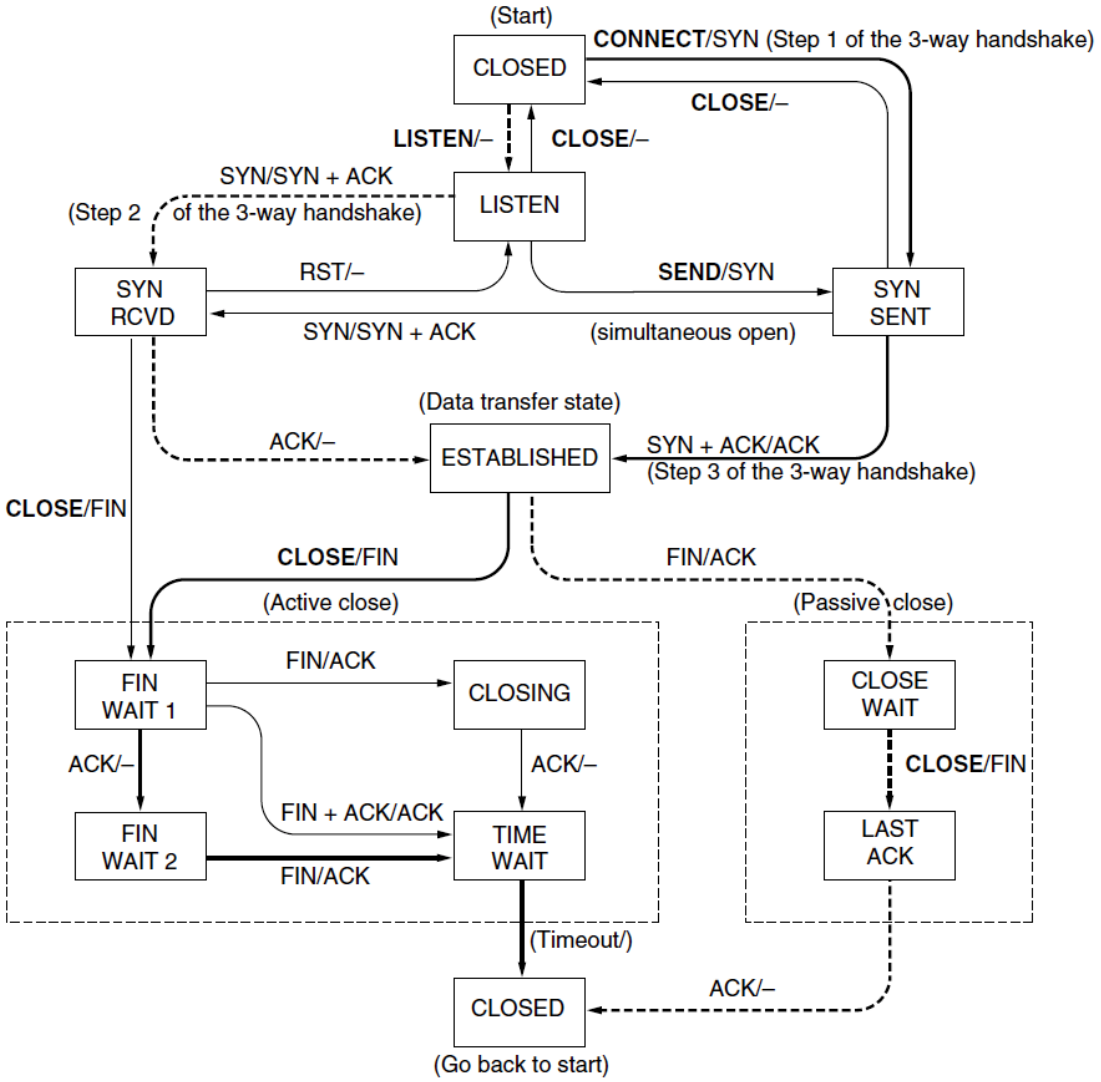
# TCP Connection State Machine (Connection Release)




- TIME_WAIT State
  - We wait a long time (two times the maximum segment lifetime of 60 seconds) after sending all segments and before completing the close, but why?
    - ACK might have been lost, in which case FIN will be resent for an orderly close
    - Could otherwise interfere with a subsequent connection

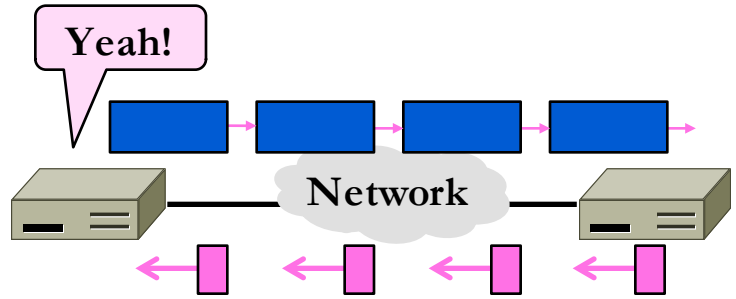
# TCP Connection State Machine Complete




| State | Description |
|---|---|
| CLOSED | No connection is active or pending |
| LISTEN | The server is waiting for an incoming call |
| SYN RCVD | A connection request has arrived; wait for ACK |
| SYN SENT | The application has started to open a connection |
| ESTABLISHED | The normal data transfer state |
| FIN WAIT 1 | The application has said it is finished |
| FIN WAIT 2 | The other side has agreed to release |
| TIME WAIT | Wait for all packets to die off |
| CLOSING | Both sides have tried to close simultaneously |
| CLOSE WAIT | The other side has initiated a release |
| LAST ACK | Wait for all packets to die off |

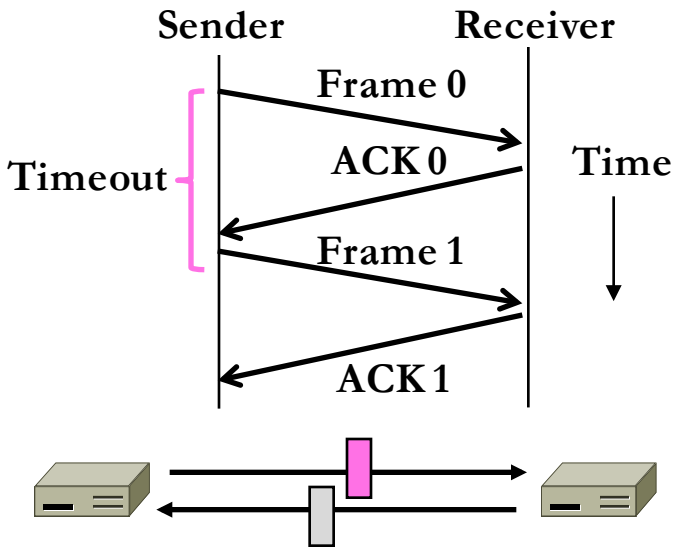
# Topics

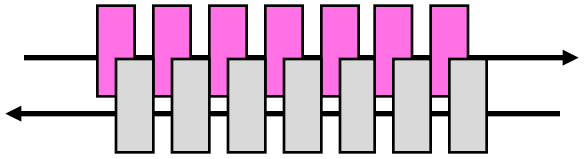
- Service Models
  - Socket API and ports
  - Datagrams, Streams
- User Datagram Protocol (UDP)
- Transmission Control Protocol (TCP)
  - Connections (establish and release)
  - Sliding Window
  - Flow control
  - Retransmission timers
  - Congestion control



# Sliding Window

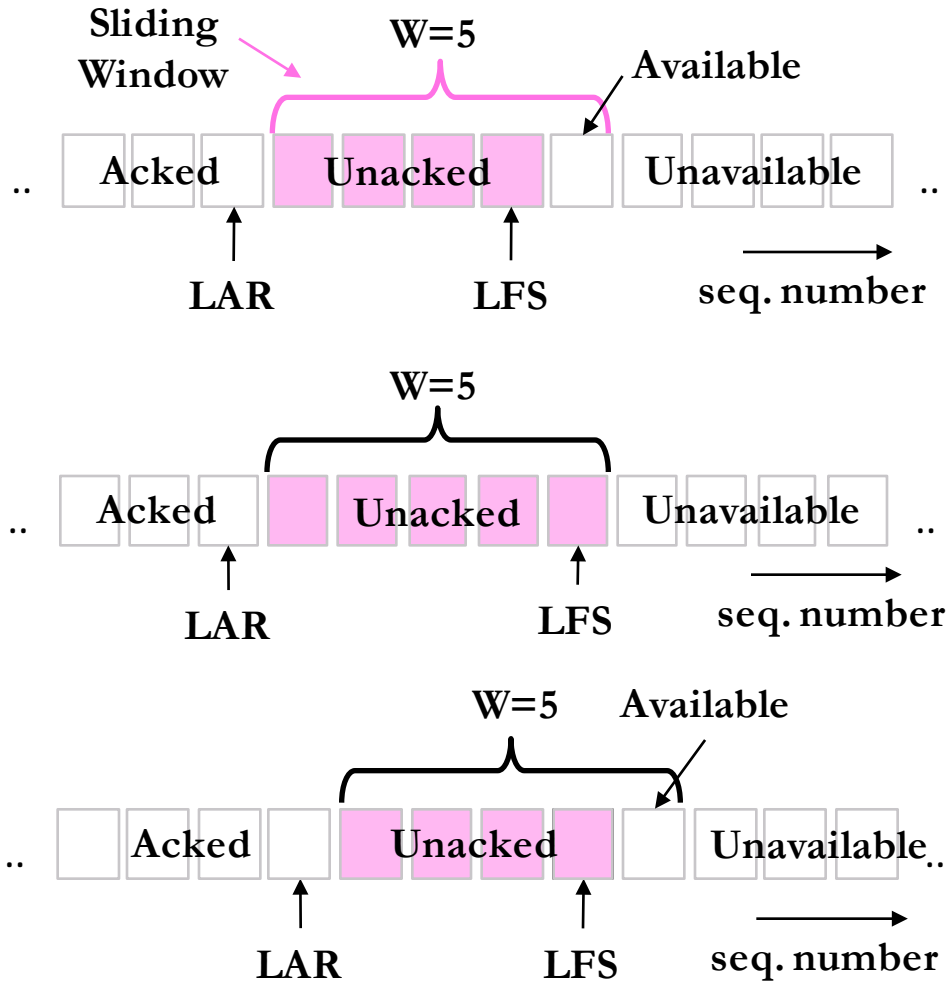
- Principles of the Algorithm
  - Pipelining and reliability
  - Building on Stop-and-Wait – ARQ with one message at a time
- Limitations of Stop-and-Wait
  - It allows only a single message to be outstanding from the sender:
    - Fine for LAN (only one frame fit)
    - Not efficient for network paths with BD >> 1 packet
  - Example: R = 1 Mbps, D = 50 ms, RTT = 2D = 100 ms
    - Assume pkt is 1250 Byte = 10 Kb, 10 Kb / 100 ms = 100 Kbps = 0.1 Mbps = only 10% channel utilization
    - What if R = 10 Mbps?
- Generalization of Stop-and-Wait
  - Allows W packets to be outstanding – can send W packets per RTT (=2D)
    - Need W = 2BD to fill network path



# Sliding Window – Sender

- Sender buffers up to W segments until they are acknowledged
  - LFS = last frame sent, LAR = last ack received
  - Sends while LFS – LAR ≤ W
- Transport accepts another segment of data from the Application ...
  - Transport sends it (as LFS – LAR = 5)
- Next higher ACK arrives from peer…
  - Window advances, buffer is freed
  - LFS – LAR → 4 (can send one more)



# Sliding Window Protocol Optimizations – Receiver Coordination

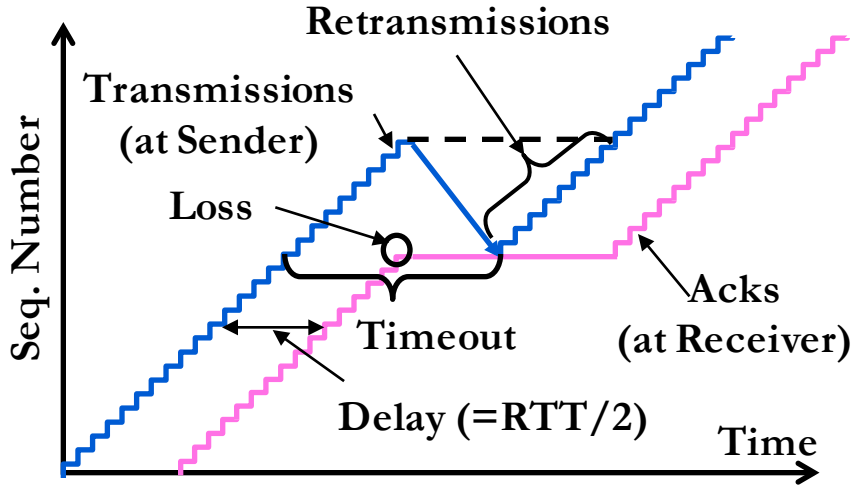
- Go-Back-N
  - Receiver keeps only a single packet buffer for the next segment
    - State variable, LAS (LAST ACK SENT)
  - On receive:
    - If seq. number is LAS+1, accept and pass it to app, update LAS, send ACK
    - Otherwise discard (as out of order) and resend an ACK of LAS
  - Retransmission: sender uses a single timer to detect losses
    - On timeout or receiving an duplicate ACK of LAR, resends buffered packets starting at LAR+1

- Selective Repeat
  - Receiver passes data to app in order, and buffers out-of-order segments to reduce retransmissions
  - TCP uses a selective repeat design
  - Buffers W segments, keeps state variable
  - On receive:
    - Buffer segments [LAS+1, LAS+W]
    - Pass up to app in-order segments from LAS+1, and update LAS
    - Send ACK for LAS regardless
  - Retransmission: sender uses a timer per unacked segment to detect losses
    - On timeout for segment, resend it
    - Hope to resend fewer segments

http://www.ccs-labs.org/teaching/rn/animations/gbn_sr/

# Sequence Numbers

- Need more than 0/1 for Stop-and-Wait …
  - But how many?
- For Selective Repeat, need W numbers for packets, plus W for acks of earlier packets
  - 2W seq. numbers
  - Fewer for Go-Back-N (W+1)
- Typically implement seq. number with an N-bit counter that wraps around at $2^N - 1$
  - E.g., N = 8: …, 253, 254, 255, 0, 1, 2, 3, …
  - TCP uses 32-bit




Go-Back-N scenario

How about selective repeat?

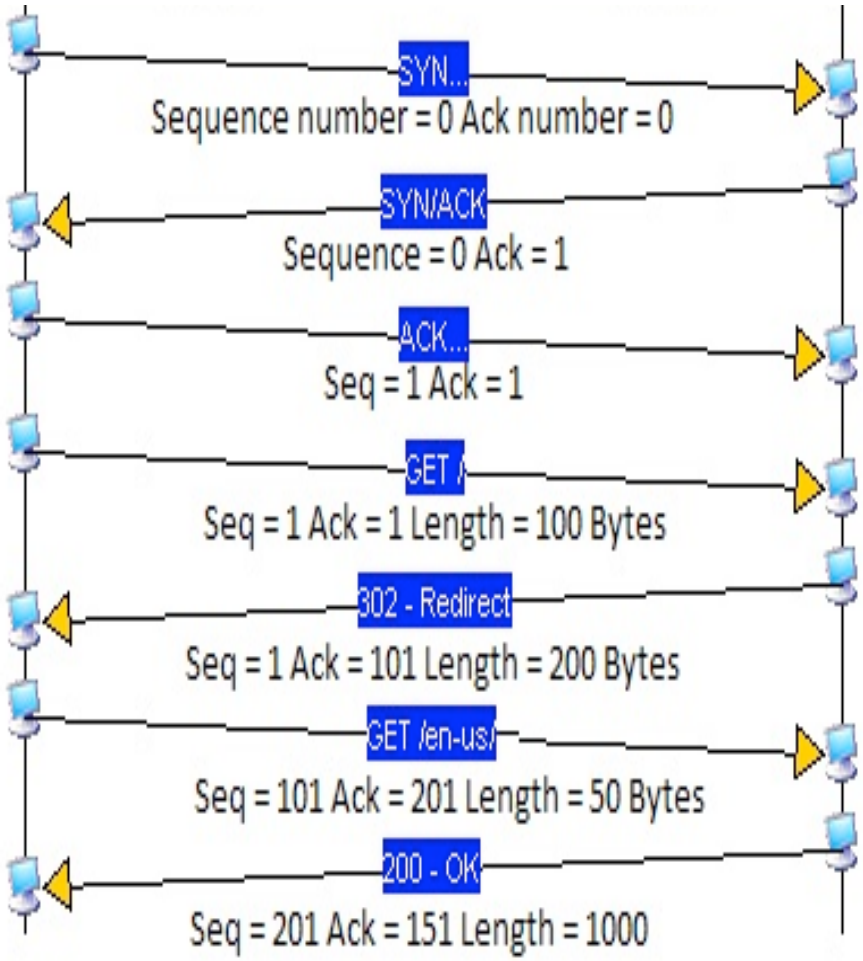
# A Simple TCP Example

Pkt 1: A sends a SYN to B with seq# 0 and ack# 0

Pkt 3: A adds a 1 to B's ISN and sends an ack to B to acknowledge its ISN

Pkt 4: A sends a 100 byte long GET request to B

Pkt 6: A sends another request of 50 bytes. Its seq# starts at 101. The next expected seq# should be 151. It acknowledges the 200 bytes sent by B by sending an ack# of 201



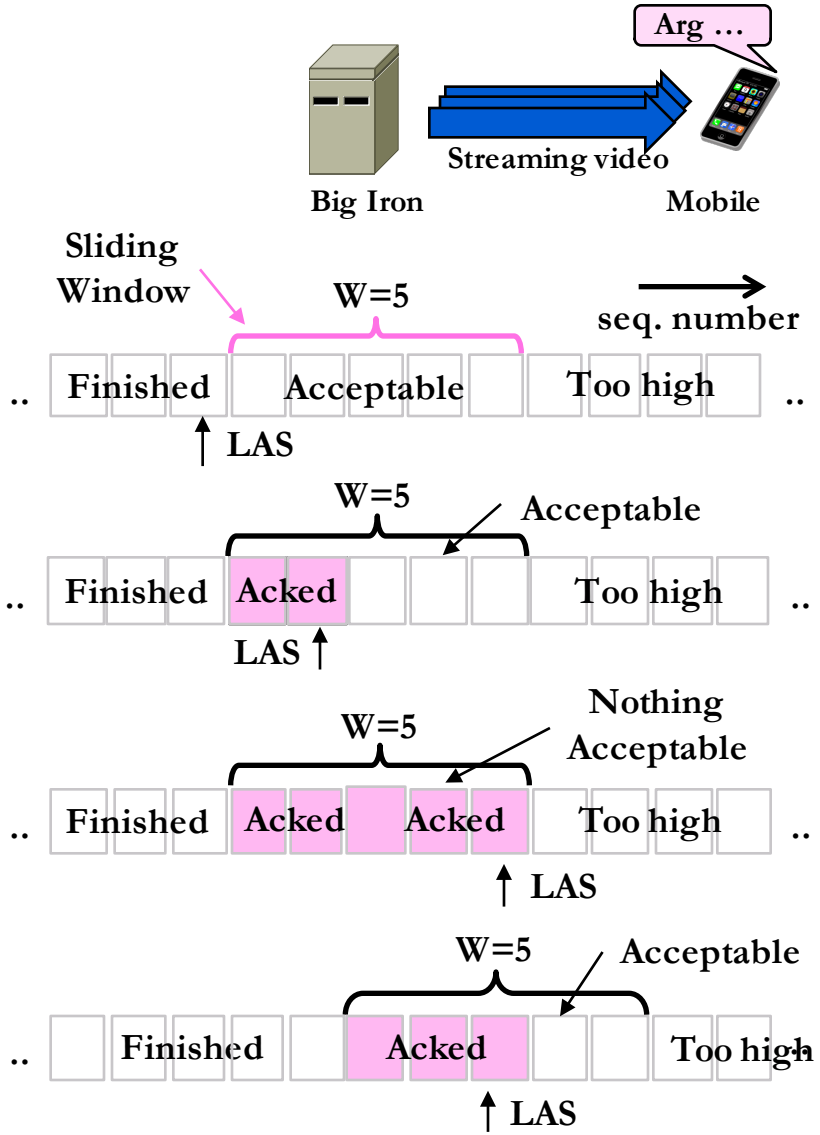

Pkt 2: B starts its seq# of 0 and acknowledges A's seq # by adding a 1

Pkt 5: B responds to the request from A. Since B has not sent data yet, its seq # is still 1. It sends the packet with ack = 101 to acknowledge receipt of the 100 bytes from A

Pkt 7: B responds to the request with a 1000 byte packet. The starting seq is 201, so the next expected seq# is 1201. It receives 50 bytes from A, so it acks with 151.

# Problem of Sliding Window

- Sliding window uses pipelining to keep the network busy
  - What if the receiver is overloaded?
- Sliding window – Receiver
  - Consider receiver with W buffers
    - LAS = last ack sent, app pulls in-order data from buffer with recv() call
  - Suppose the next two segments arrive but app does not call recv()
    - LAS rises, but we can't slide window!
  - If further segments arrive (even in order) we can fill the buffer
    - Must drop segments until app recvs!
  - App recv() takes two segments
    - Window slides

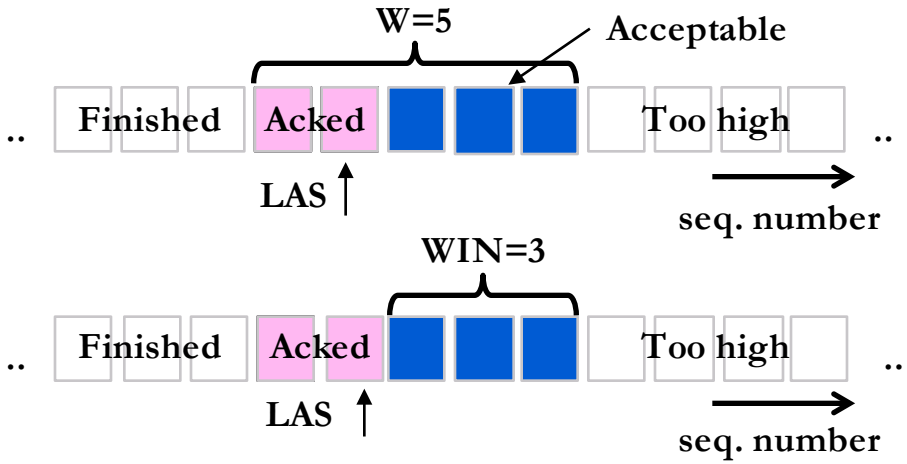


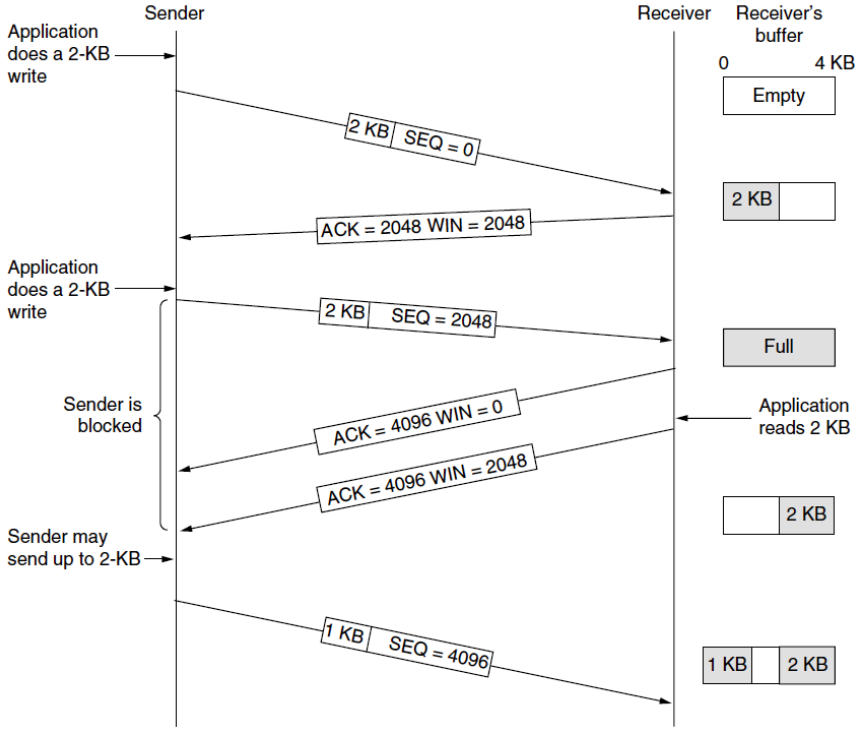
# Topics

- Service Models
  - Socket API and ports
  - Datagrams, Streams
- User Datagram Protocol (UDP)
- Transmission Control Protocol (TCP)
  - Connections (establish and release)
  - Sliding Window
  - Flow control
  - Retransmission timers
  - Congestion control

# Flow Control

- Solution: Adding flow control to the sliding window algorithm
  - To slow the over-enthusiastic sender
- Avoid loss at receiver by telling sender the available buffer space
  - WIN = #Acceptable, not W (from LAS)
- Sender uses the lower of the sliding window and flow control window (WIN) as the effective window size
- TCP-style example
  - SEQ/ACK sliding window
  - Flow control with WIN
  - SEQ + length < ACK + WIN
  - 4 KB buffer at receiver
  - Circular buffer of bytes



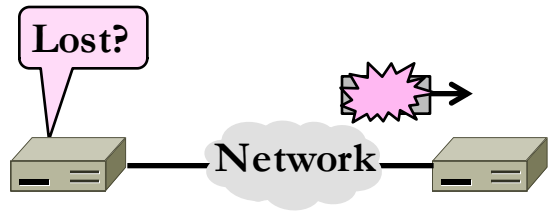



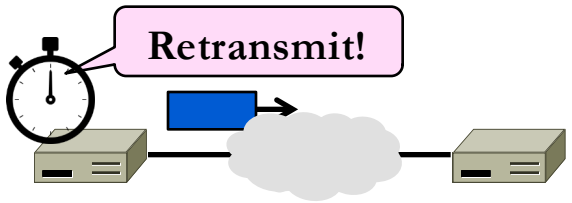
# Topics

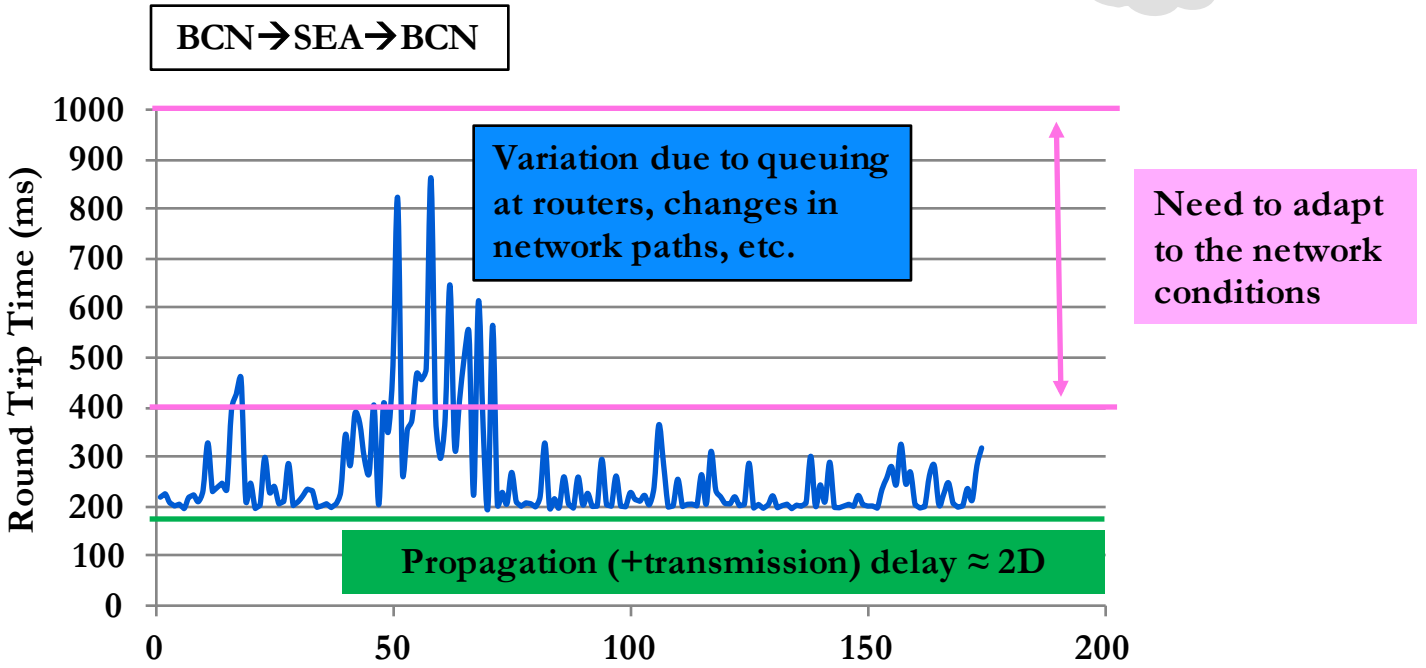
- Service Models
  - Socket API and ports
  - Datagrams, Streams
- User Datagram Protocol (UDP)
- Transmission Control Protocol (TCP)
  - Connections (establish and release)
  - Sliding Window
  - Flow control
  - Retransmission timers
  - Congestion control

# Retransmission Timeouts

- How to set the timeout for sending a retransmission
  - Adapting to the network path
- Retransmissions
  - With sliding window, the strategy for detecting loss is the timeout
    - Set timer when a segment is sent
    - Cancel timer when ack is received
    - If timer fires, retransmit data as lost
- Timeout Problem
  - Timeout should be "just right"
    - Too long wastes network capacity
    - Too short leads to spurious resends
  - Easy to set on a LAN (Link)
    - Short, fixed, predictable RTT
  - Hard on the Internet (Transport)
    - Wide range, variable RTT

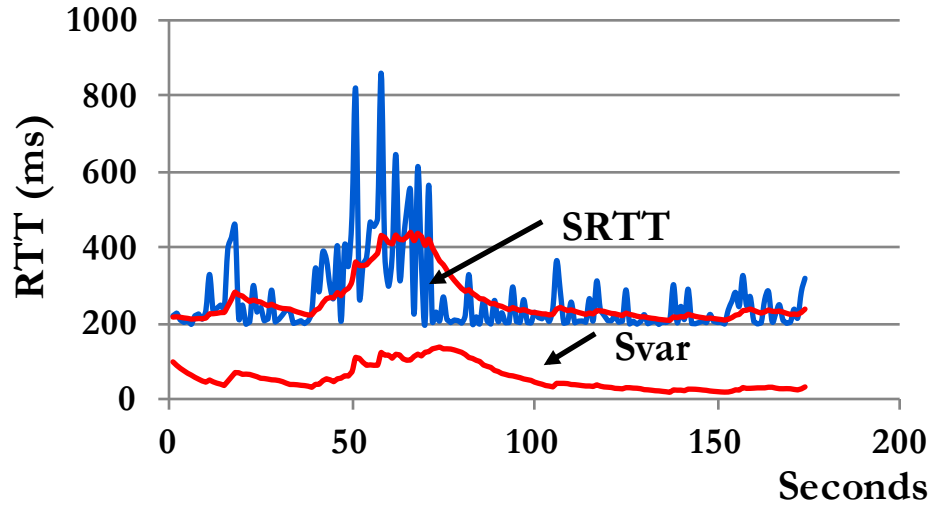


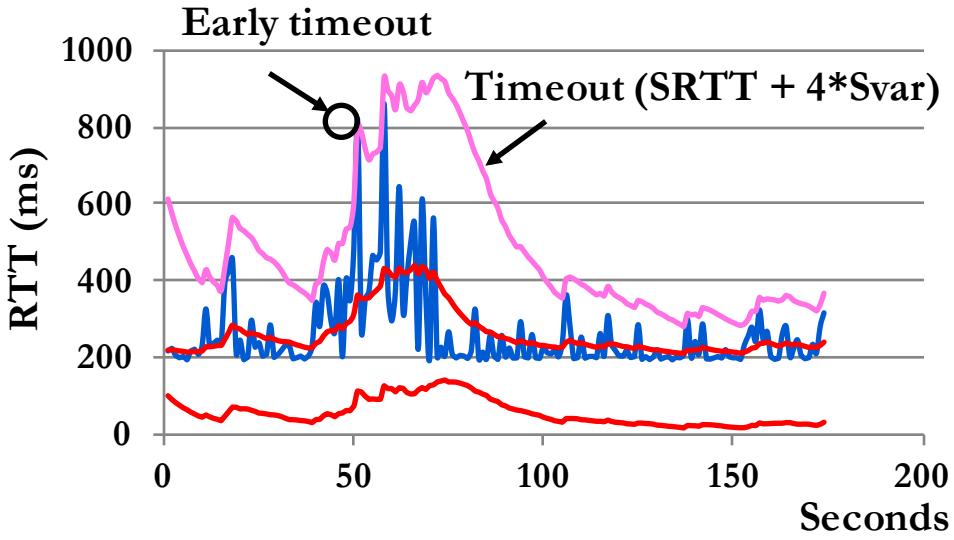



# Adaptive Timeout

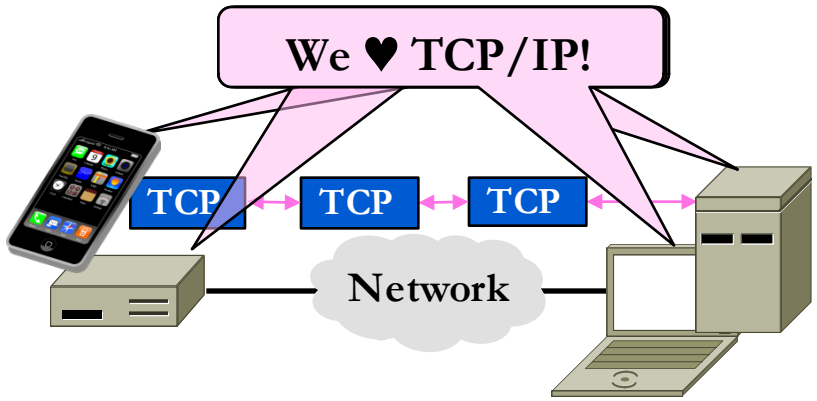
- Keep smoothed estimates of the mean SRTT and variance Svar of RTT
  - Update estimates with a moving average
    - $SRTT_{N+1} = 0.9*SRTT_N + 0.1*RTT_{N+1}$
    - $Svar_{N+1} = 0.9*Svar_N + 0.1*|RTT_{N+1} - SRTT_{N+1}|$
- Set timeout to a multiple of estimates
  - To estimate the upper RTT in practice
  - TCP $Timeout_N = SRTT_N + 4*Svar_N$
- Simple to compute, does a good job of tracking actual RTT
  - Little "headroom" to lower
  - Yet very few early timeouts
- Important for good performance and robustness



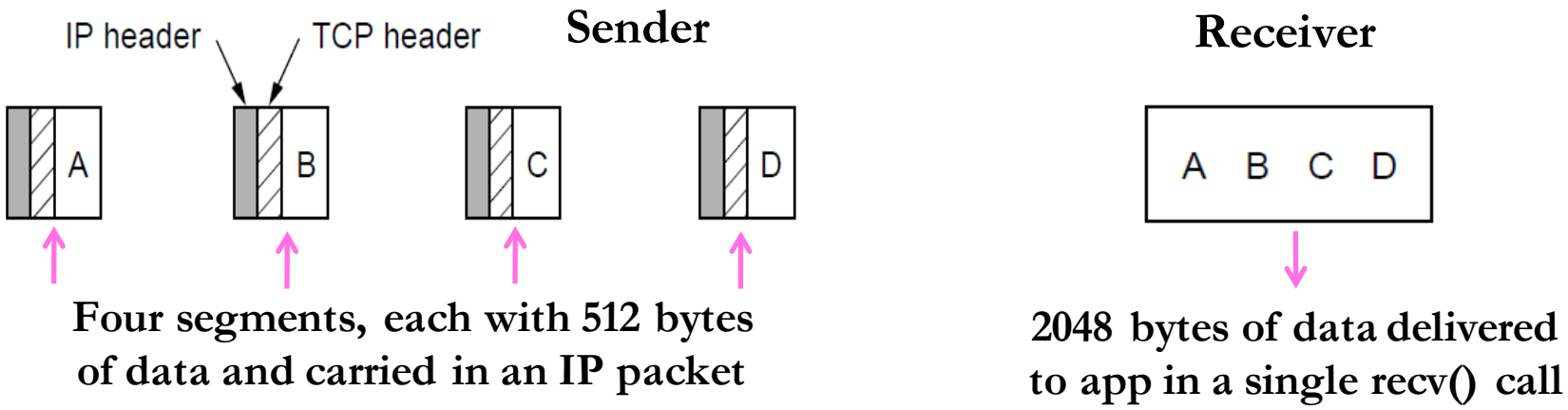
# Transmission Control Protocol

- How TCP works!
  - The transport protocol used for most content on the Internet
- TCP Features
  - A reliable bytestream service
  - Based on connections
  - Sliding window for reliability
    - With adaptive timeout
  - Flow control for slow receivers
  - Congestion control to allocate network bandwidth

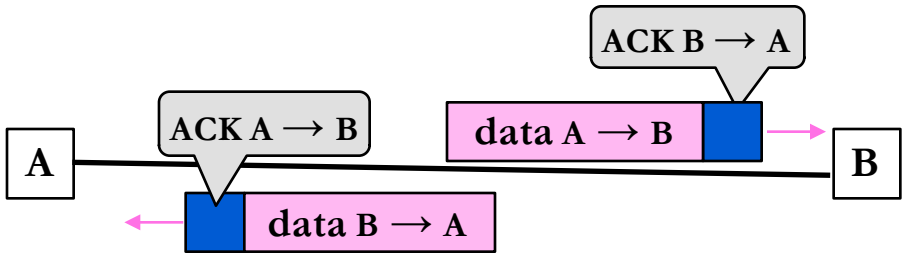


# Reliable Bytestream

- Message boundaries not preserved from send() to recv()
  - But reliable and ordered (receive bytes in same order as sent)

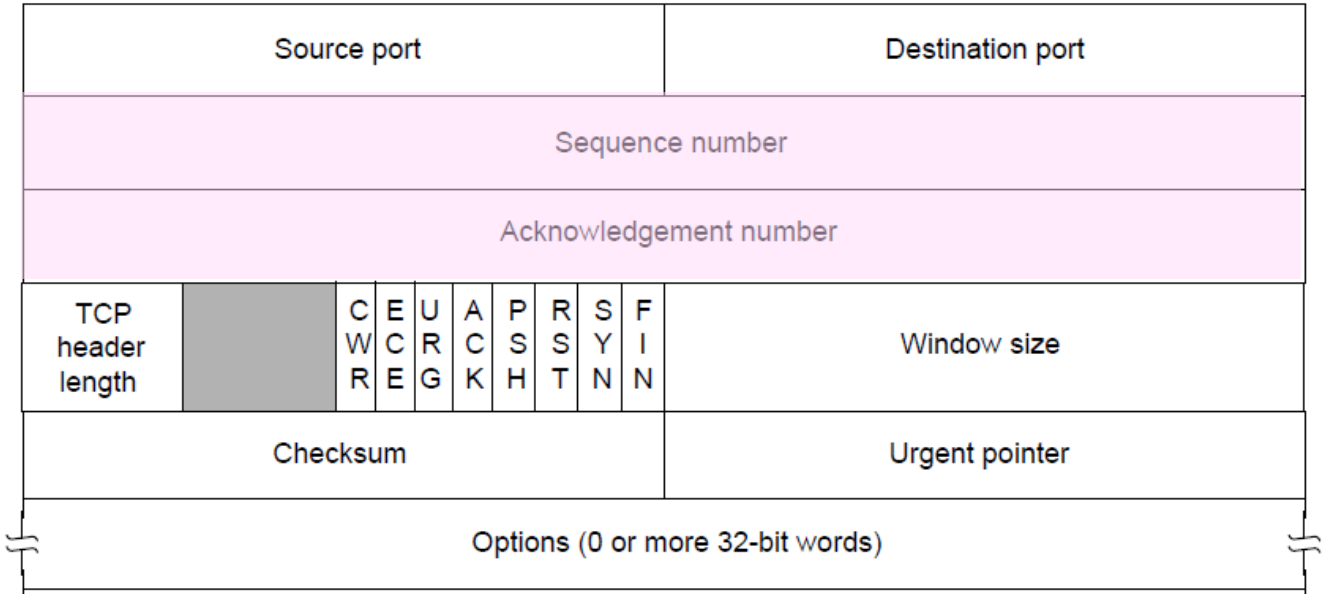



- Bidirectional data transfer
  - Control information (e.g., ACK) piggybacks on data segments in reverse direction



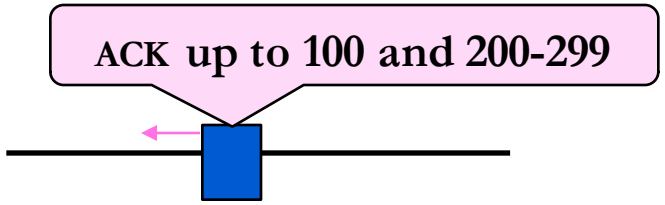
# TCP Header

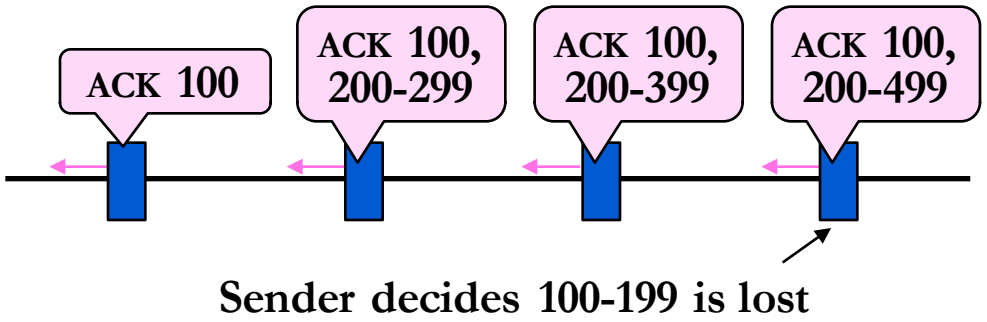
- Ports identify apps (socket API)
  - 16-bit identifiers
- SEQ/ACK used for sliding window
  - Selective Repeat, with byte positions
- SYN/FIN/RST flags for connections
  - Flag indicates segment is a SYN etc.
- Window size for flow control
  - Relative to ACK, and in bytes



# TCP Sliding Window

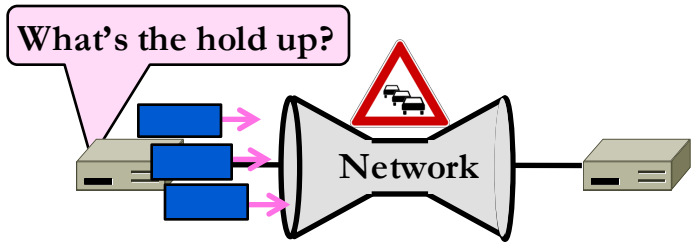
- **Receiver**
  - Cumulative ACK tells next expected byte sequence number ("LAS+1")
  - Optionally, selective ACKs (SACK) give hints for receiver buffer state
    - List up to 3 ranges of received bytes
- **Sender**
  - Uses an adaptive retransmission timeout to resend data from LAS+1
  - Uses heuristics to infer loss quickly and resend to avoid timeouts
    - "Three duplicate ACKs" treated as loss



# Topics

- Service Models
  - Socket API and ports
  - Datagrams, Streams
- User Datagram Protocol (UDP)
- Transmission Control Protocol (TCP)
  - Connections (establish and release)
  - Sliding Window
  - Flow control
  - Retransmission timers
  - Congestion control

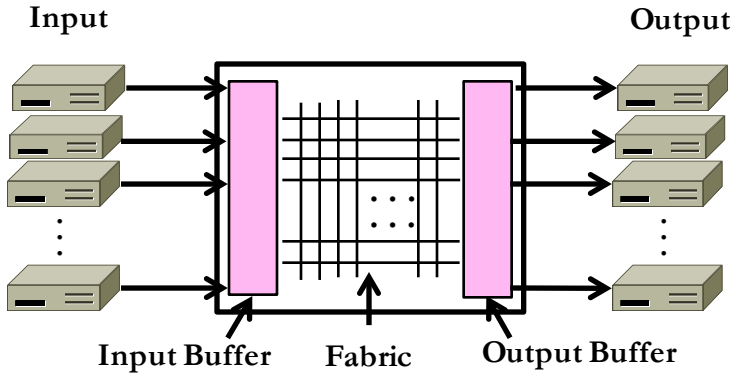


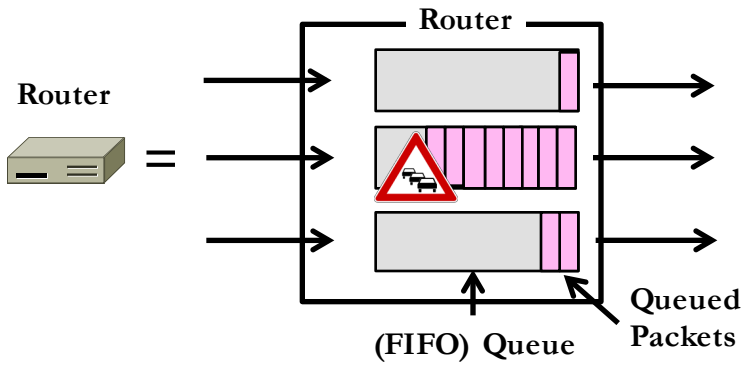
# Congestion Overview

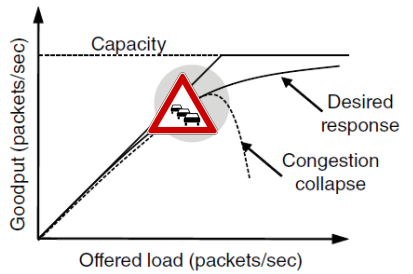
- More fun in the Transport Layer!
  - The mystery of congestion control
  - Depends on the Network layer too
- Understanding congestion, a "traffic jam" in the network
  - Later we will learn how to control it
- Topics
  - Nature of congestion
  - Fairness of Bandwidth Allocation
  - AIMD Control Law
  - TCP Congestion Control history
  - ACK Clocking
  - TCP Slow-start
  - TCP Fast Retransmit/Recovery
  - Congestion Avoidance (ECN)

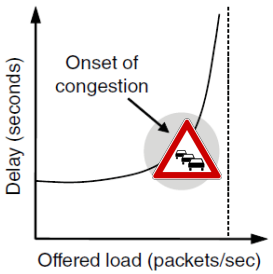
# Nature of Congestion

- Routers/switches have internal buffering for contention
- Simplified view of per port output queues
  - Typically FIFO (First In First Out), discard when full
- Queues help by absorbing bursts when input > output rate
- But if input > output rate persistently, queue will overflow
  - http://www.ccs-labs.org/teaching/rn/animations/queue/
- Congestion is a function of the traffic patterns – can occur even if every link have the same capacity
- Effects of Congestion
  - What happens to performance as we increase the load?
  - As offered load rises, congestion occurs as queues begin to fill:
    - Delay and loss rise sharply with more load
    - Throughput falls below load (due to loss)
    - Goodput may fall below throughput (due to spurious retransmissions)




Want to operate network just before the onset of congestion

# Bandwidth Allocation

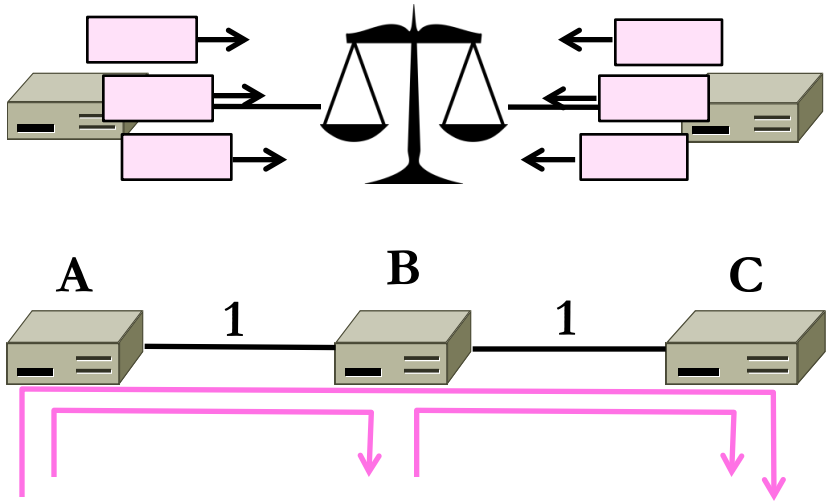
- Important task for network is to allocate its capacity to senders
  - Good allocation is efficient and fair
- Efficient means most capacity is used but there is no congestion
- Fair means every sender gets a reasonable share the network

- Key observation:
  - In an effective solution, Transport and Network layers must work together
- Network layer witnesses congestion
  - Only it can provide direct feedback
- Transport layer causes congestion
  - Only it can reduce offered load

- Why is it hard? (Just split equally!)
  - Number of senders and their offered load is constantly changing
  - Senders may lack capacity in different parts of the network
  - Network is distributed; no single party has an overall picture of its state
- Solution context:
  - Senders adapt concurrently based on their own view of the network
  - Design this adaption so the network usage as a whole is efficient and fair
  - Adaption is continuous since offered loads continue to change over time

# Fairness of Bandwidth Allocation

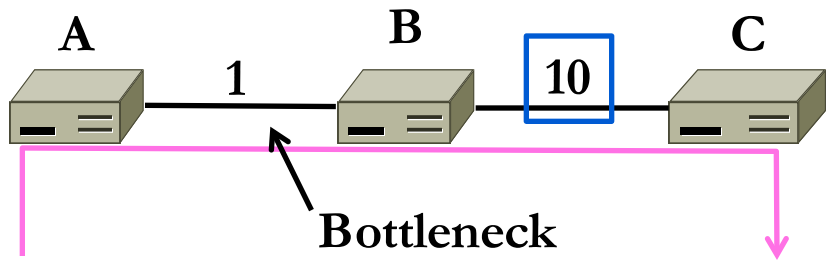
- What's a "fair" bandwidth allocation?
- Recall
  - We want a good bandwidth allocation to be fair and efficient
    - Now we learn what fair means
  - Caveat: in practice, efficiency is more important than fairness
- Efficiency vs. Fairness
  - Cannot always have both!
    - Example network with traffic A → B, B → C and A → C
    - How much traffic can we carry?
- The Slippery Notion of Fairness
  - Why is "equal per flow" fair anyway?
    - A → C uses more network resources (two links) than A → B or B → C
    - Host A sends two flows, B sends one
  - Not productive to seek exact fairness
    - More important to avoid starvation; "Equal per flow" is good enough




  - If we care about fairness:
    - Give equal bandwidth to each flow
    - A → B: ½ unit, B → C: ½, and A → C: ½
    - Total traffic carried is 1 ½ units
  - If we care about efficiency:
    - Maximize total traffic in network
    - A → B: 1 unit, B → C: 1, and A → C: 1
    - Total traffic carried is 2 units

# Generalizing "Equal per Flow"

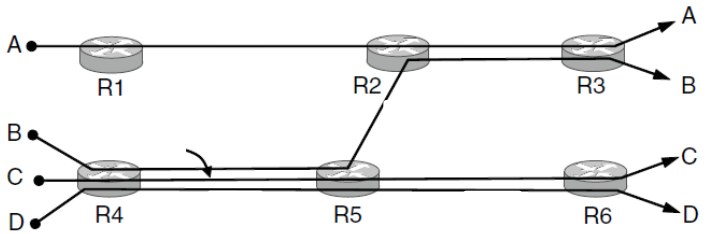
- Bottleneck for a flow of traffic is the link that limits its bandwidth
  - Where congestion occurs for the flow
  - For A → C, link A – B is the bottleneck
- Flows may have different bottlenecks
  - For A → C, link A–B is the bottleneck
  - For B → C, link B–C is the bottleneck
  - Can no longer divide links equally …



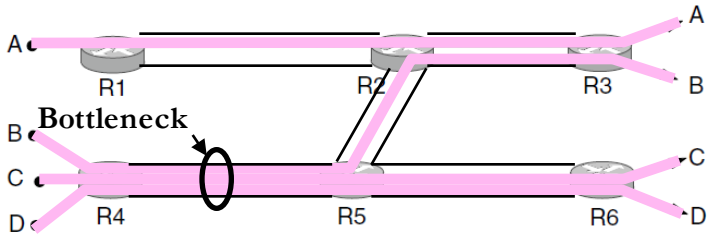
# Max-Min Fairness

- Intuitively, flows bottlenecked on a link get an equal share of that link
- Max-min fair allocation is one that:
  - Increasing the rate of one flow will decrease the rate of a smaller flow
  - This "maximizes the minimum" flow
- To find it given a network, imagine "pouring water into the network"
  1. Start with all flows at rate 0
  2. Increase the flows until there is a new bottleneck in the network
  3. Hold fixed the rate of the flows that are bottlenecked
  4. Go to step 2 for any remaining flows

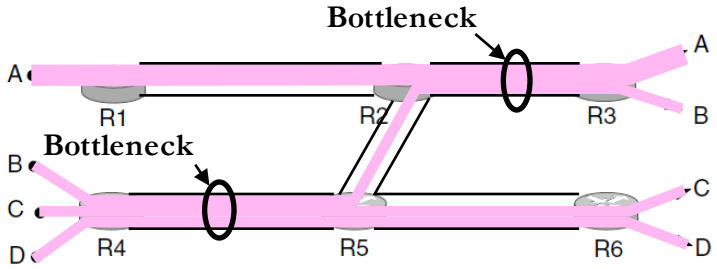
- Example: network with 4 flows, links equal bandwidth



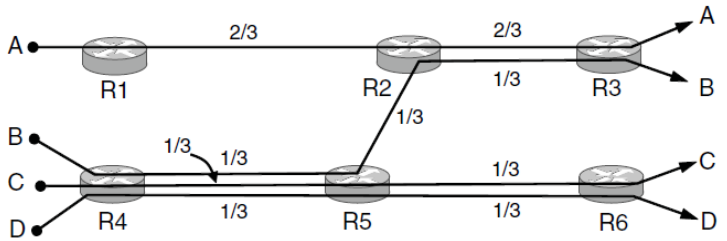

When rate=1/3, flows B, C, and D bottleneck R4 – R5. Fix B, C, and D, continue to increase A




When rate=2/3, flow A bottlenecks R2 – R3. Done.

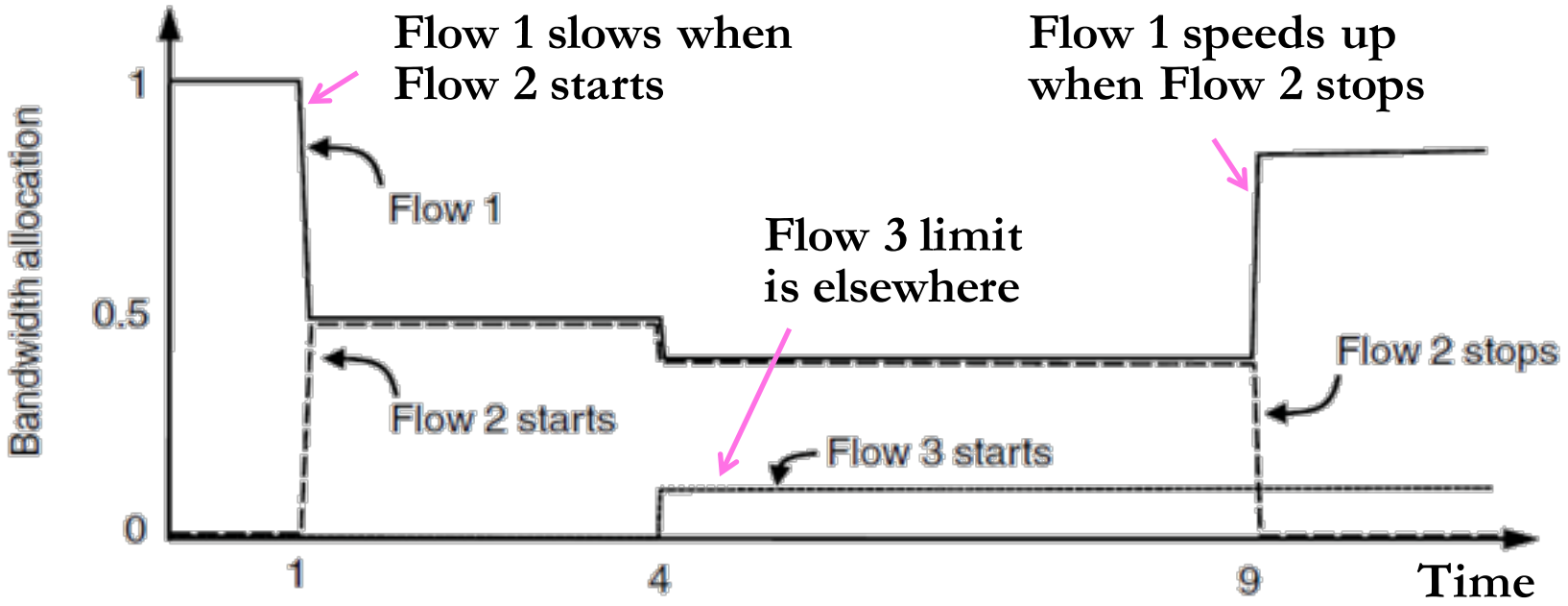



End with A=2/3, B, C, D=1/3, and R2 – R3, R4 – R5 full. Other links have extra capacity that can't be used



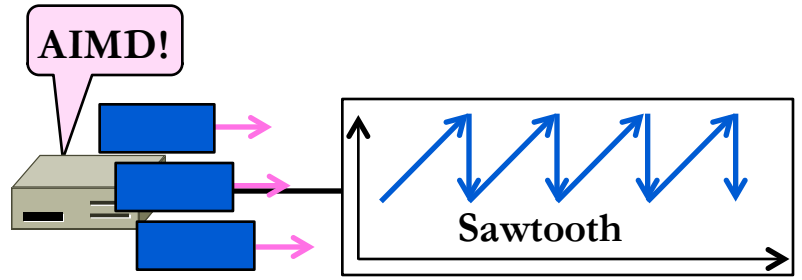
# Adapting over Time

- Allocation changes as flows start and stop



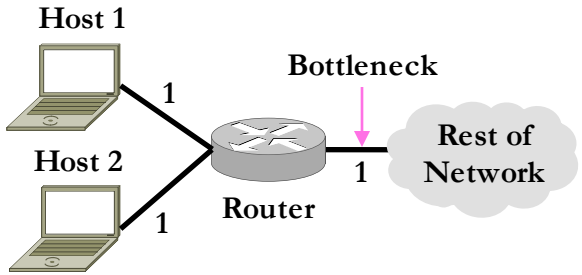
# Bandwidth Allocation Models

- Want to allocate capacity to senders, but how?
- Open loop versus closed loop
  - Open: reserve bandwidth before use
  - Closed: use feedback to adjust rates
- Host versus Network support
  - Who sets/enforces allocations?
- Window versus Rate based
  - How is allocation expressed?
- We'll look at closed-loop, host-driven, and window-based approach which TCP adopts
- Network layer returns feedback on current allocation to senders
  - At least tells if there is congestion
- Transport layer adjusts sender's behavior via window in response
  - How senders adapt is a control law
    - Example: Additive Increase Multiplicative Decrease



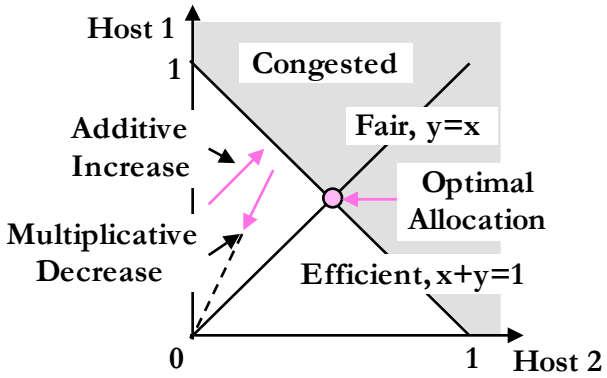
# Additive Increase Multiplicative Decrease (AIMD)

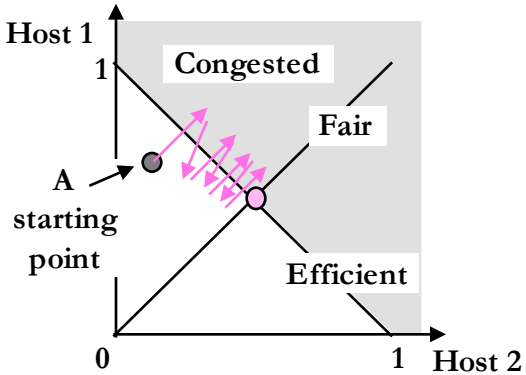
- AIMD is a control law hosts can use to reach a good allocation
  - Hosts additively increase rate while network is not congested
  - Hosts multiplicatively decrease rate when congestion occurs
  - Used by TCP
- Let's explore the AIMD game …
  - Hosts 1 and 2 share a bottleneck
    - But do not talk to each other directly
  - Router provides binary feedback
    - Tells hosts if network is congested




Each point is a possible allocation

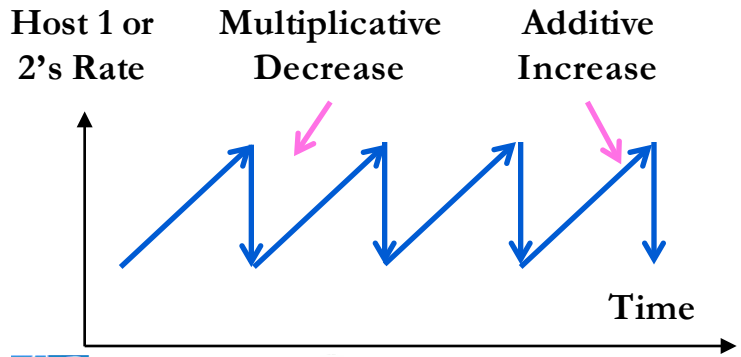
AI and MD move the allocation




Always converge to good allocation!

- Properties
  - Produces a "sawtooth" pattern over time for rate of each host
  - Converges to an allocation that is efficient and fair when hosts run it
    - Holds for more general topologies
  - Other increase/decrease control laws do not! (Try MIAD, MIMD, AIAD)
  - Requires only binary feedback from the network



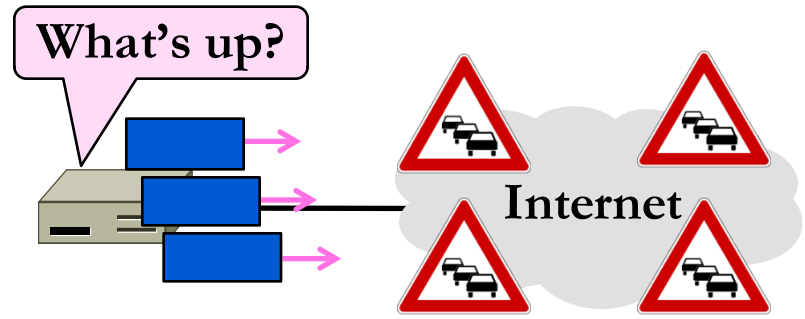
# Feedback Signals

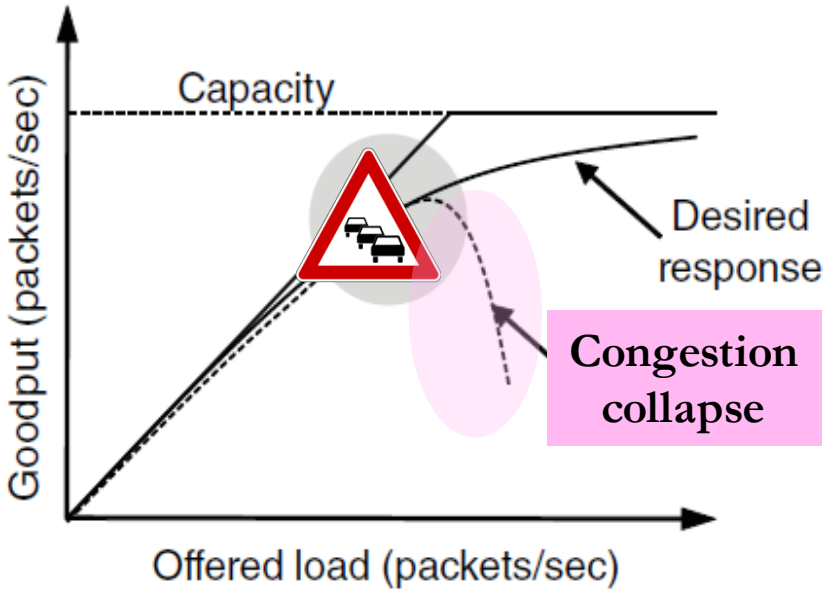
- Several possible signals, with different pros/cons
  - We'll look at classic TCP that uses packet loss as a signal

| Signal | Example Protocol | Pros / Cons |
|---|---|---|
| Packet loss | TCP NewReno<br>Cubic TCP (Linux) | Hard to get wrong<br>Hear about congestion late |
| Packet delay | Compound TCP (Windows) | Hear about congestion early<br>Need to infer congestion |
| Router indication | TCPs with Explicit Congestion Notification | Hear about congestion early<br>Require router support |

# History of TCP Congestion Control

- The story of TCP congestion control
  - Collapse, control, and diversification
- Congestion Collapse in the 1980s
  - Early TCP used a fixed size sliding window (e.g., 8 packets)
    - Initially fine for reliability
  - But something strange happened as the ARPANET grew
    - Links stayed busy but transfer rates fell by orders of magnitude!
  - Queues became full, retransmissions clogged the network, and goodput fell



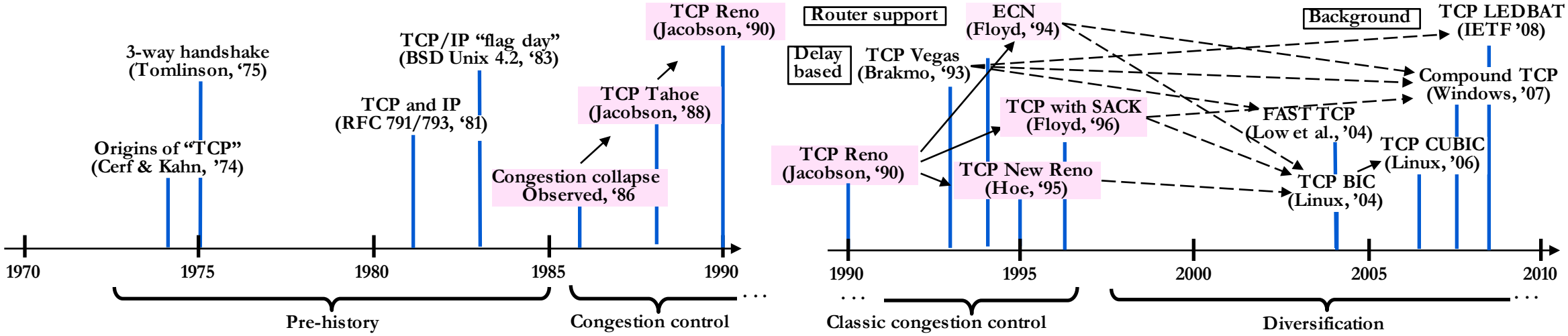
# Van Jacobson (1950—)

- Widely credited with saving the Internet from congestion collapse in the late 80s
  - Introduced congestion control principles
  - Practical solutions (TCP Tahoe/Reno)
  - V. Jacobson and M. J. Karels, Congestion Avoidance and Control, ACM SIGCOMM 1988.
- Much other pioneering work:
  - Tools like traceroute, tcpdump, pathchar
  - IP header compression, multicast tools

# TCP Tahoe/Reno

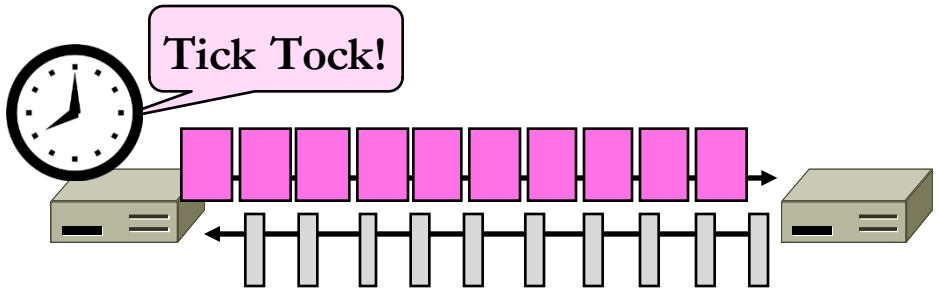
- Avoid congestion collapse without changing routers (or even receivers)
- Idea is to fix timeouts and introduce a congestion window (cwnd) over the sliding window to limit queues/loss
- TCP Tahoe/Reno implements AIMD by adapting cwnd using packet loss as the network feedback signal

- TCP behaviors we will study:
  - ACK clocking
  - Adaptive timeout (mean and variance)
  - Slow-start
  - Fast Retransmission
  - Fast Recovery
- Together, they implement AIMD

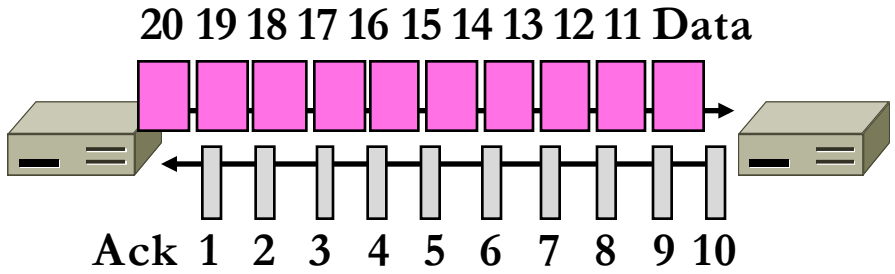


# TCP ACK Clocking

- The self-clocking behavior of sliding windows, and how it is used by TCP
  - The "ACK clock"

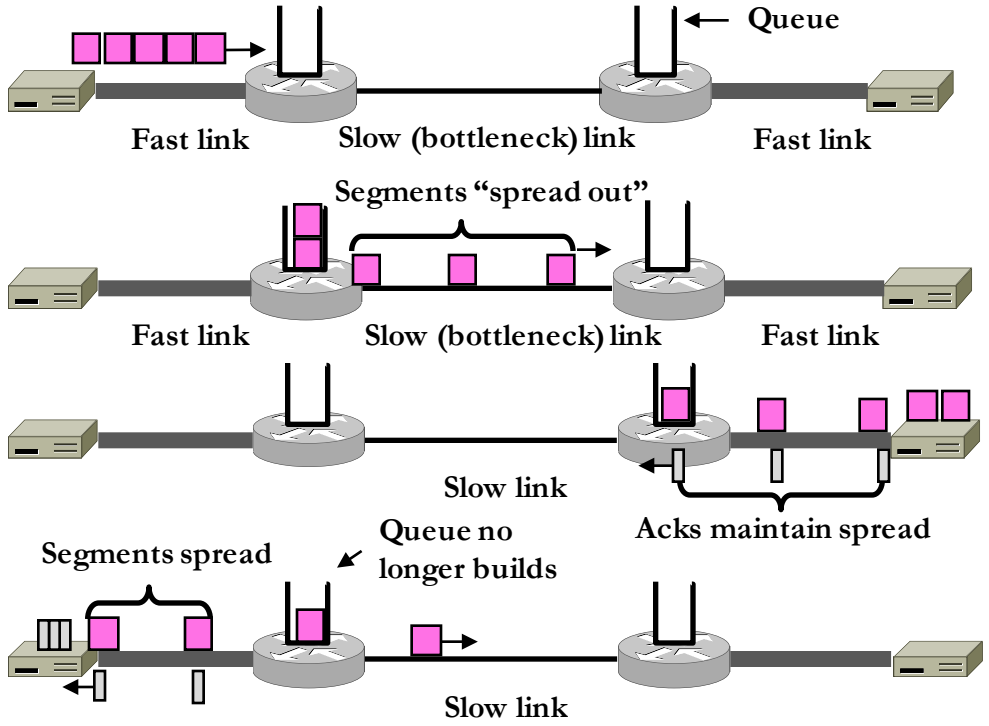



- Sliding Window ACK Clock
  - Each in-order ACK advances the sliding window and lets a new segment enter the network
    - ACKs "clock" data segments



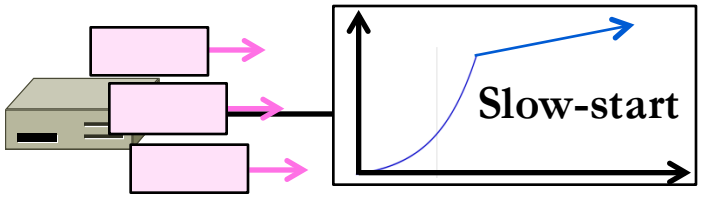
# Benefit of ACK Clocking

- Consider what happens when sender injects a burst of segments into the network
- Segments are buffered and spread out on slow link
- ACKs maintain the spread back to the original sender
- Sender clocks new segments with the spread
  - Now sending at the bottleneck link without queuing!
- The network has smoothed out the burst of data segments
  - ACK clock transfers this smooth timing back to the sender
  - Subsequent data segments are not sent in bursts so do not queue up in the network
- TCP Uses ACK Clocking – Sliding window controls how many segments are inside the network
  - Called the congestion window, or cwnd; Rate is roughly cwnd/RTT



# TCP Slow Start

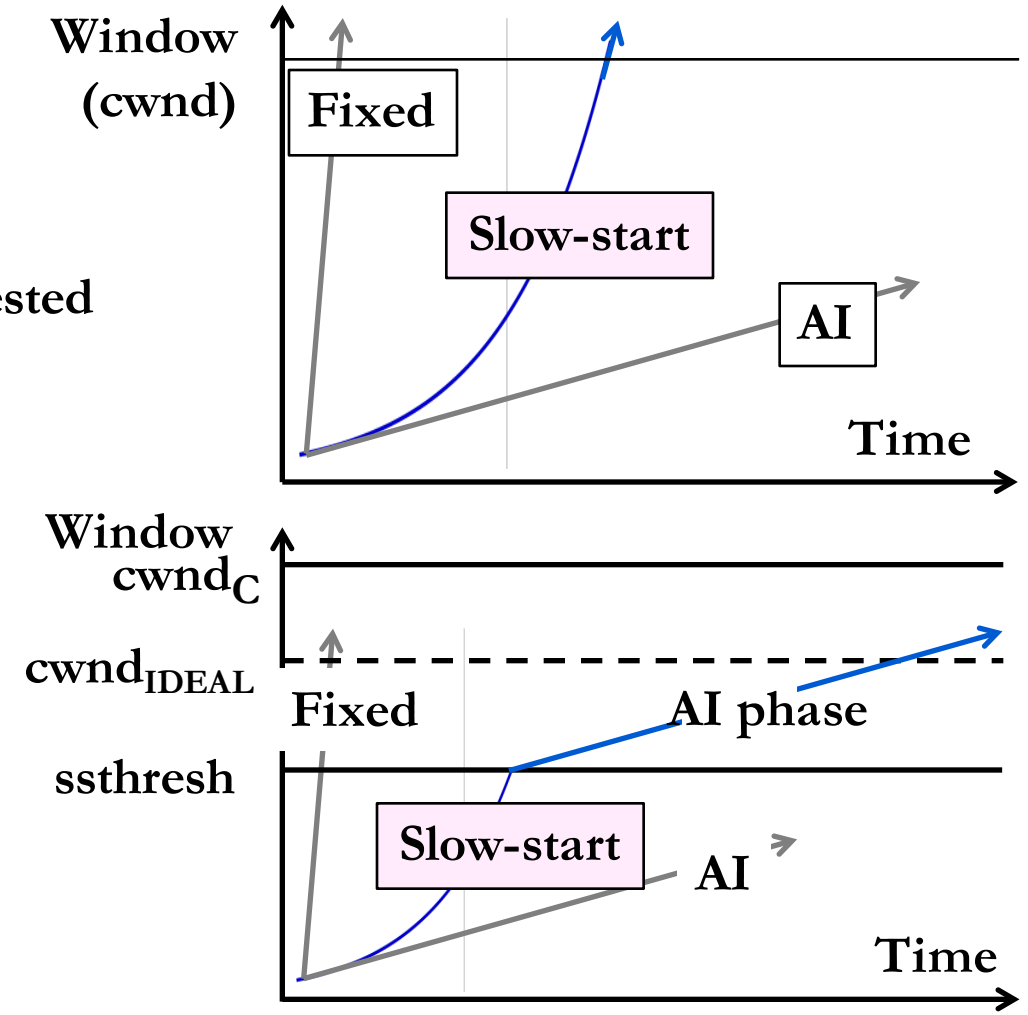
- How TCP implements AIMD, part 1
  - "Slow start" is a component of the AI portion of AIMD



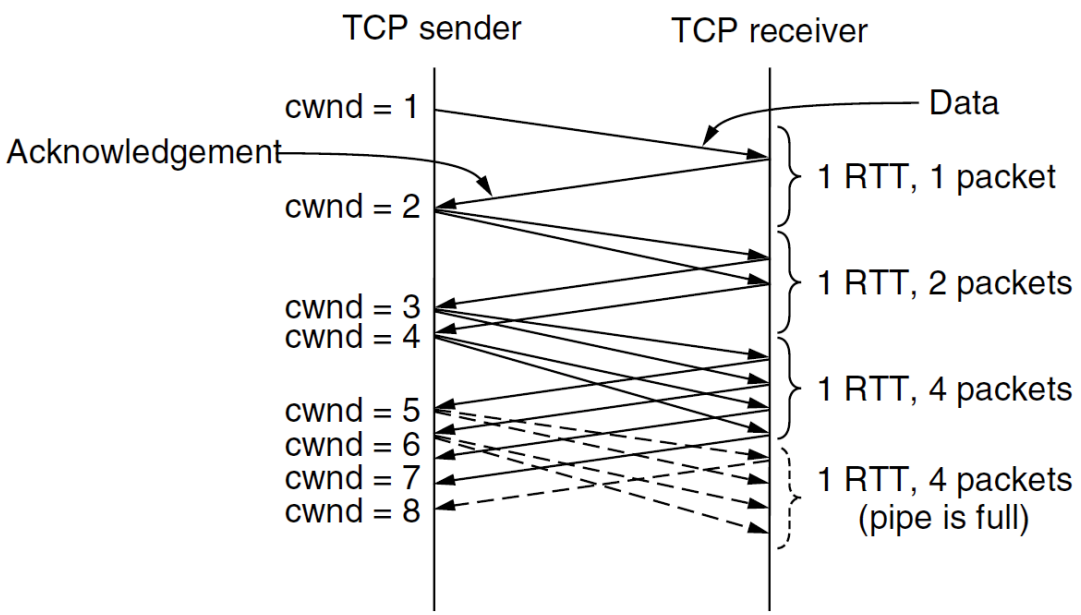

- Recall
  - We want TCP to follow an AIMD control law for a good allocation
  - Sender uses a congestion window or cwnd to set its rate (≈cwnd/RTT)
  - Sender uses packet loss as the network congestion signal
  - Need TCP to work across a very large range of rates and RTTs
- TCP Startup Problem
  - We want to quickly near the right rate, $cwnd_{IDEAL}$, but it varies greatly
    - Fixed sliding window doesn't adapt and is rough on the network (loss!)
    - AI with small bursts adapts cwnd gently to the network, but might take a long time to become efficient

# Slow-Start Solution

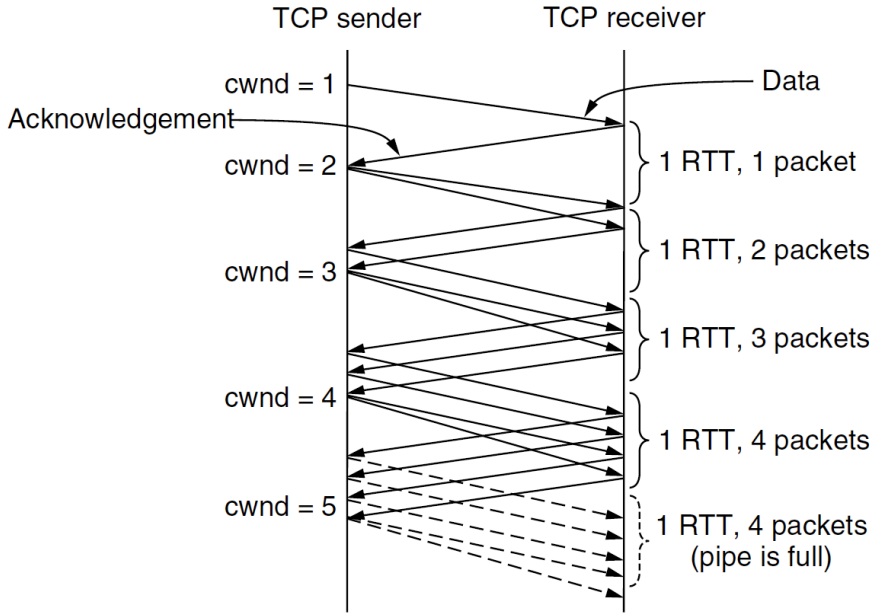
- Start by doubling cwnd every RTT
  - Exponential growth (1, 2, 4, 8, 16, …)
  - Start slow, quickly reach large values
- Eventually packet loss will occur when the network is congested
  - Loss timeout tells us cwnd is too large
  - Next time, switch to AI beforehand
  - Slowly adapt cwnd near right value
- In terms of cwnd:
  - Expect loss for $cwnd_C \approx 2BD + queue$
  - Use ssthresh $= cwnd_C/2$ to switch to AI
- Combined behavior, after first time
  - Most time spend near right value



# Comparison of Slow Start and Additive Increase




Increment cwnd by 1 packet for each ACK



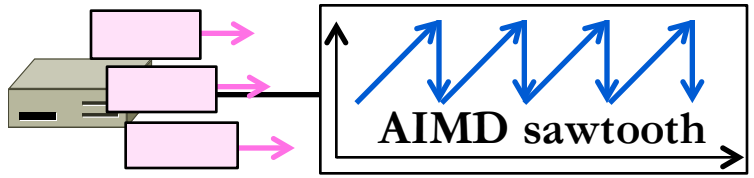

Increment cwnd by 1 packet every cwnd ACKs (or 1 RTT)

# TCP Tahoe (Implementation)

- Initial slow-start (doubling) phase
  - Start with cwnd = 1 (or small value)
  - cwnd += 1 packet per ACK
- Later Additive Increase phase
  - cwnd += 1/cwnd packets per ACK
  - Roughly adds 1 packet per RTT
- Switching threshold (initially infinity)
  - Switch to AI when cwnd > ssthresh
  - Set ssthresh = cwnd/2 after loss
  - Begin with slow-start after timeout
- Timeout Misfortunes
  - Why do a slow-start after timeout?
    - Instead of MD cwnd (for AIMD)
  - Timeouts are sufficiently long that the ACK clock will have run down
    - Slow-start ramps up the ACK clock
  - We need to detect loss before a timeout to get to full AIMD
    - Done in TCP Reno

# TCP Fast Retransmit / Fast Recovery

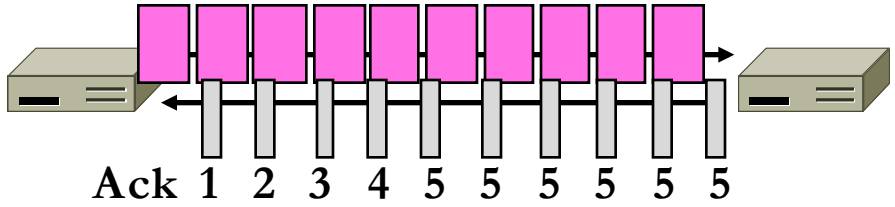
- How TCP implements AIMD, part 2
  - "Fast retransmit" and "fast recovery" are the MD portion of AIMD

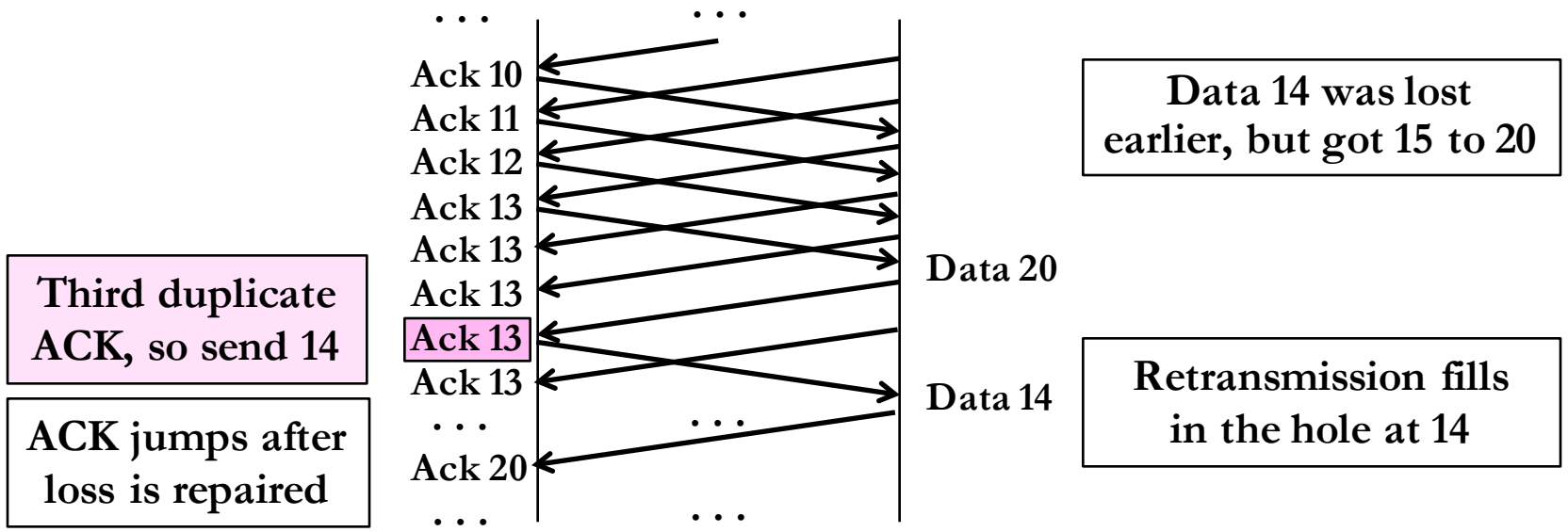



- Recall
  - We want TCP to follow an AIMD control law for a good allocation
  - Sender uses a congestion window or cwnd to set its rate (≈cwnd/RTT)
  - Sender uses slow-start to ramp up the ACK clock, followed by Additive Increase
  - But after a timeout, sender slow-starts again with cwnd=1 (as it no ACK clock)
- Inferring Loss from ACKs
  - TCP uses a cumulative ACK
    - Carries highest in-order seq. number
    - Normally a steady advance
  - Duplicate ACKs give us hints about what data hasn't arrived
    - Tell us some new data did arrive, but it was not next segment, thus the next segment may be lost

# Fast Retransmit

- Treat three duplicate ACKs as a loss
  - Retransmit next expected segment
  - Some repetition allows for reordering, but still detects loss quickly



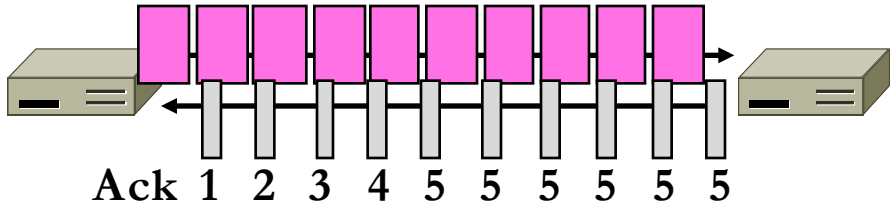

- It can repair single segment loss quickly, typically before a timeout
- However, we have quiet time at the sender/receiver while waiting for the ACK to jump
- And we still need to MD cwnd …

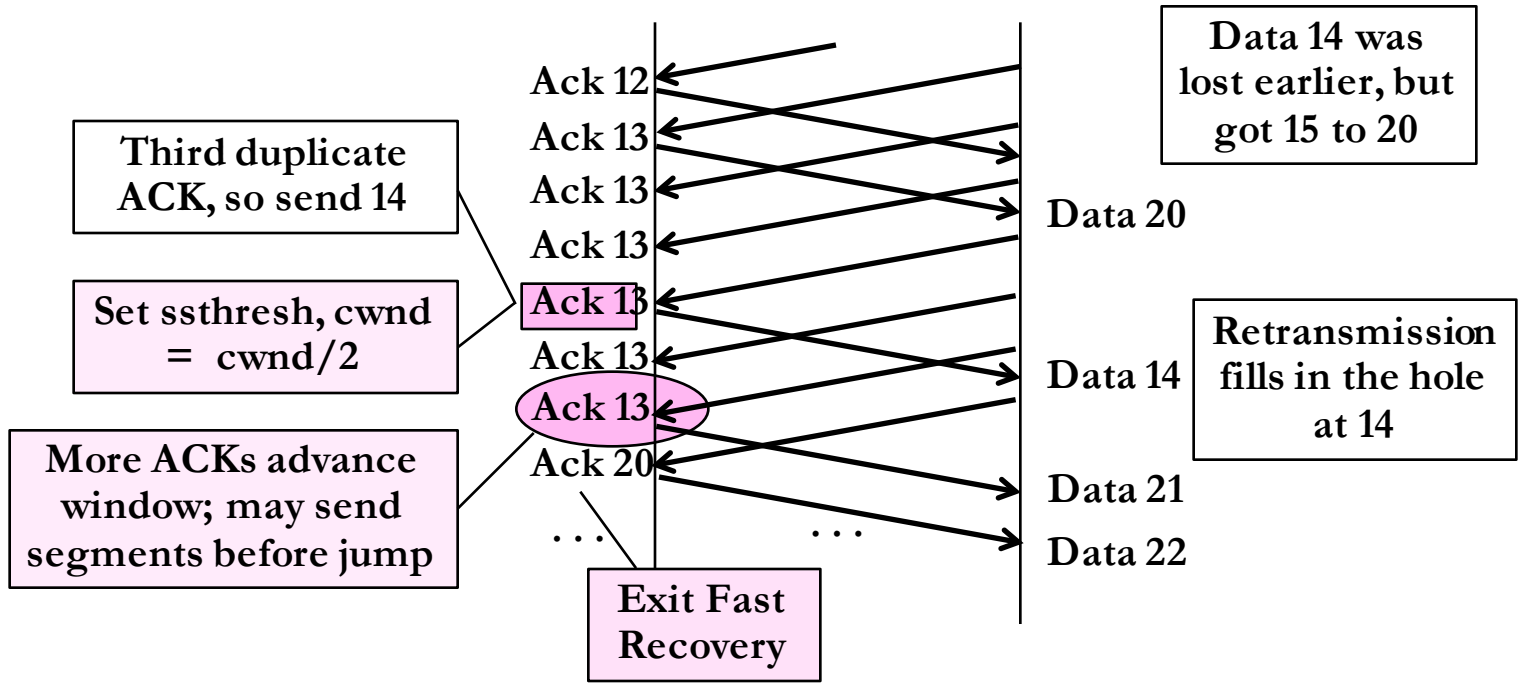
# Inferring Non-Loss from ACKs

- Duplicate ACKs also give us hints about what data has arrived
  - Each new duplicate ACK means that some new segment has arrived
  - It will be the segments after the loss
  - Thus advancing the sliding window will not increase the number of segments stored in the network

# Fast Recovery

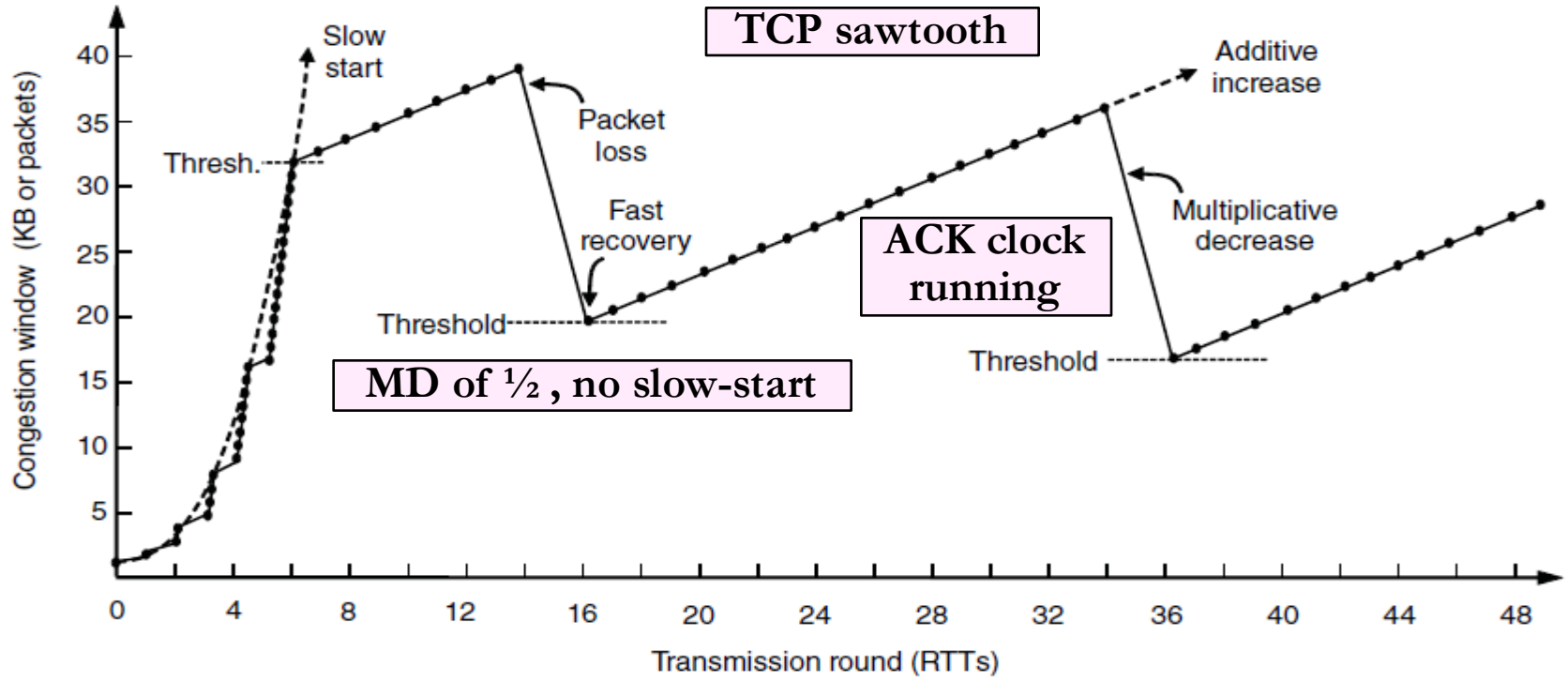
- First fast retransmit, and MD cwnd
- Then pretend further duplicate ACKs are the expected ACKs
  - Lets new segments be sent for ACKs
  - Reconcile views when the ACK jumps
- With fast retransmit, it repairs a single segment loss quickly and keeps the ACK clock running
- This allows us to realize AIMD
  - No timeouts or slow-start after loss, just continue with a smaller cwnd



# TCP Reno

- TCP Reno combines slow-start, fast retransmit and fast recovery
  - Multiplicative Decrease is ½



# TCP Reno, NewReno, and SACK

- Reno can repair one loss per RTT
  - Multiple losses cause a timeout
- NewReno further refines ACK heuristics
  - Repairs multiple losses without timeout
- Selective ACK is a better idea
  - Receiver sends ACK ranges so sender can retransmit without guesswork

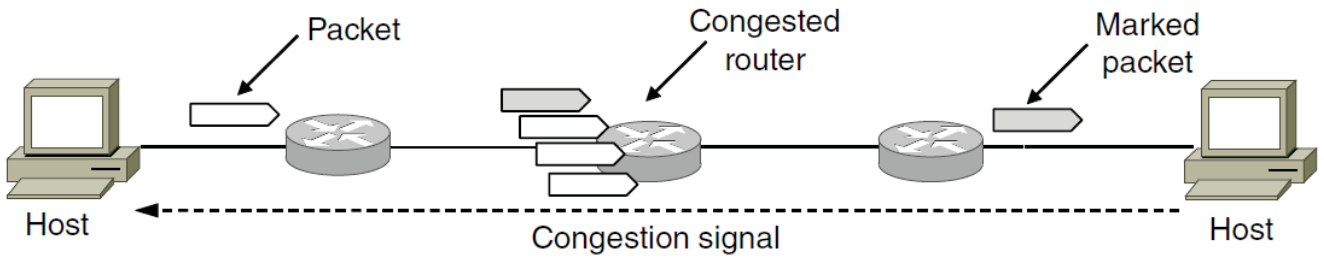
# Congestion Avoidance vs. Control

- Classic TCP drives the network into congestion and then recovers
  - Needs to see loss to slow down
- Would be better to use the network but avoid congestion altogether!
  - Reduces loss and delay
- But how can we do this? – Feedback Signals
  - Delay and router signals can let us avoid congestion

| Signal | Example Protocol | Pros / Cons |
|---|---|---|
| Packet loss | Classic TCP<br>Cubic TCP (Linux) | Hard to get wrong<br>Hear about congestion late |
| Packet delay | Compound TCP<br>(Windows) | Hear about congestion early<br>Need to infer congestion |
| Router indication | TCPs with Explicit Congestion Notification | Hear about congestion early<br>Require router support |

# Explicit Congestion Notification (ECN)

- Router detects the onset of congestion via its queue
  - When congested, it marks affected packets (IP header)
- Marked packets arrive at receiver; treated as loss
  - TCP receiver reliably informs TCP sender of the congestion




- Advantages:
  - Routers deliver clear signal to hosts
  - Congestion is detected early, no loss
  - No extra packets need to be sent
- Disadvantages:
  - Routers and hosts must be upgraded