TCP: Flow control and Congestion control

TCP

- TCP provides the following abstraction
  - In-order delivery
  - Reliability

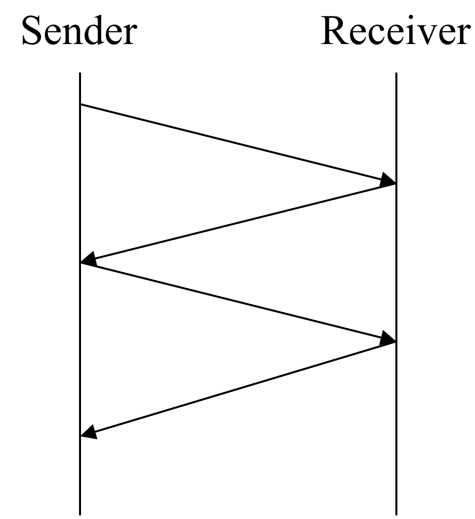
- Both of these are solved using a simple component: Sequence number and acknowledgements

Two common ways for reliability in networking

- Automatic Repeat Request (ARQ)
  – Usually used in all layers.

- Forward Error Correction (FEC)
  – Usually used in lower layers

Simple ARQ solution: Stop and Wait



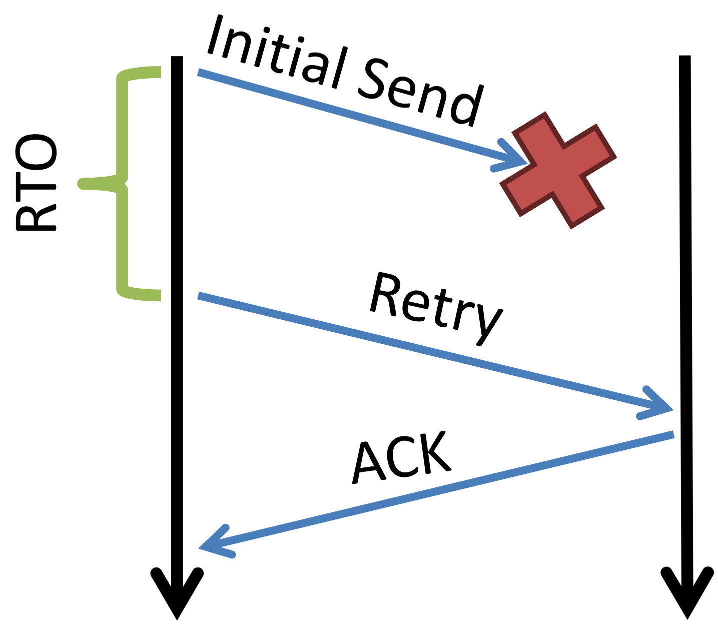
- Sender doesn’t send next packet until she’s sure receiver has last packet
- What happens if ack is not received?
- **Retransmission after a timeout**

Timeouts

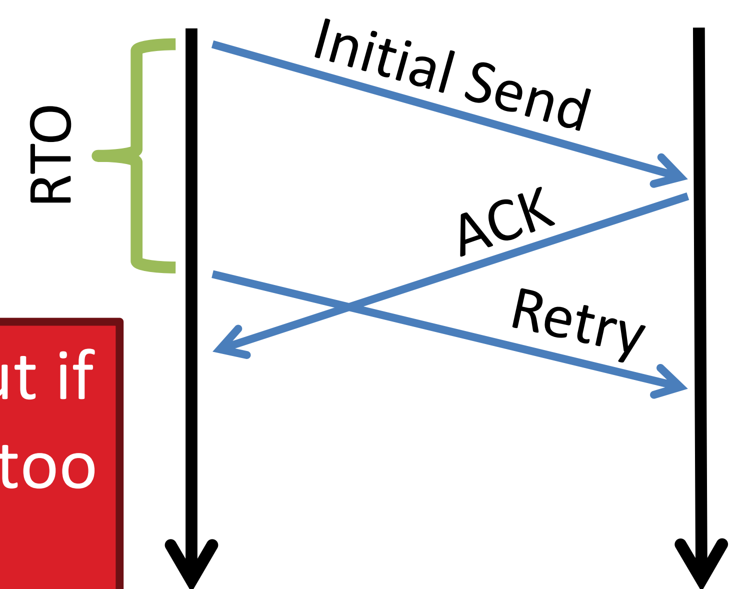
- Lost segments detected by sender
  - Use timeout to detect missing ACKs
  - What should the timeout depend on?

Retransmission Time Outs (RTO)

- Problem: time-out could be too short



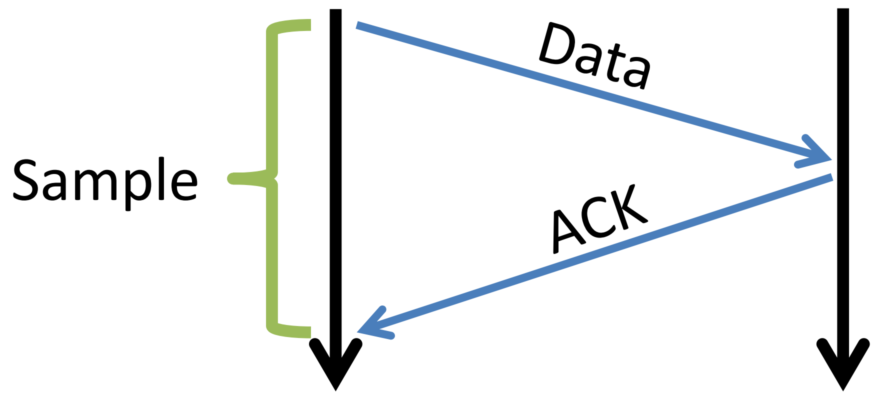
What about if timeout is too long?



TCP’s timeout

- One idea
  - RTO = 2 * RTT’
  - RTT’ is the estimated RTT
  - TCP is conservative
- RFC gives the exact RTO measure
  - https://tools.ietf.org/html/rfc6298
- How should we estimate RTT?

Round Trip Time Estimation



- RTT = (1-α) (RTT) + α(new_sample)
  - Recommended α: 0.125

Back to stop-and-wait.

- Stop and wait is inefficient.
- Why?

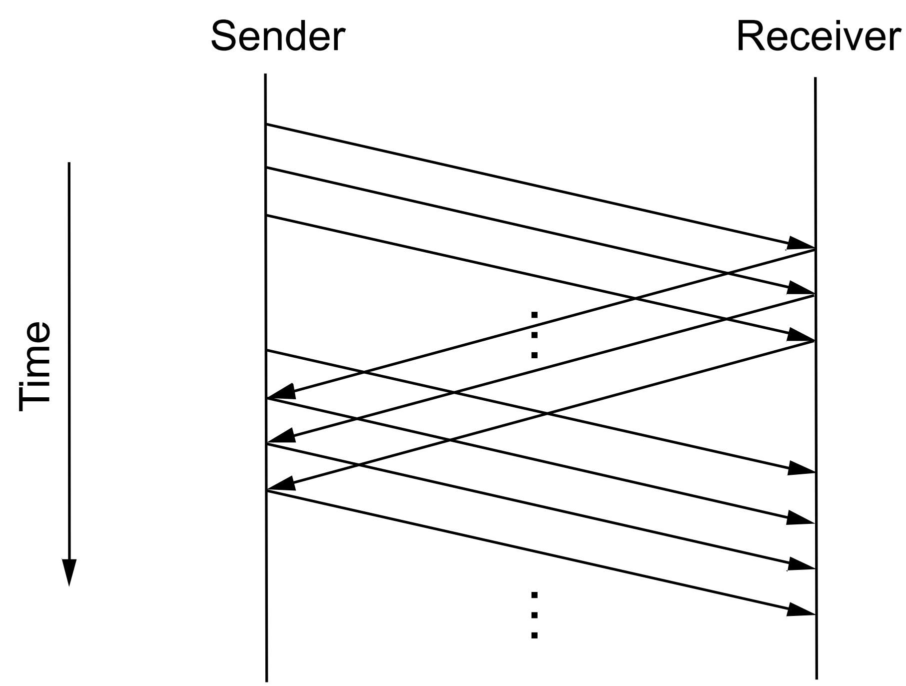
Performance problem with stop and wait

- Problem: “keeping the pipe full”
  - If the bandwidth-delay product [BDP] is much larger than a packet size, the sender will be unable to keep the link busy
  - **Bandwidth-delay product** estimates the amount of data that can be pushed on the pipe

Bandwidth delay product

- Determine the Bandwidth-Delay Product (BDP)
  - Bandwidth Delay Product = Bandwidth * (Round Trip Time/2)
  - BDP = BW * (RTT/2)
  - e.g. 10Gbps*70ms =700,000,000bits = 87,500,000Bytes
- What does a large/small BDP mean?

Alternate approach: Sliding window



Sounds familiar?

Sliding window problems

- Can overwhelm the receiver
- Can cause congestion
- Solution: flow control

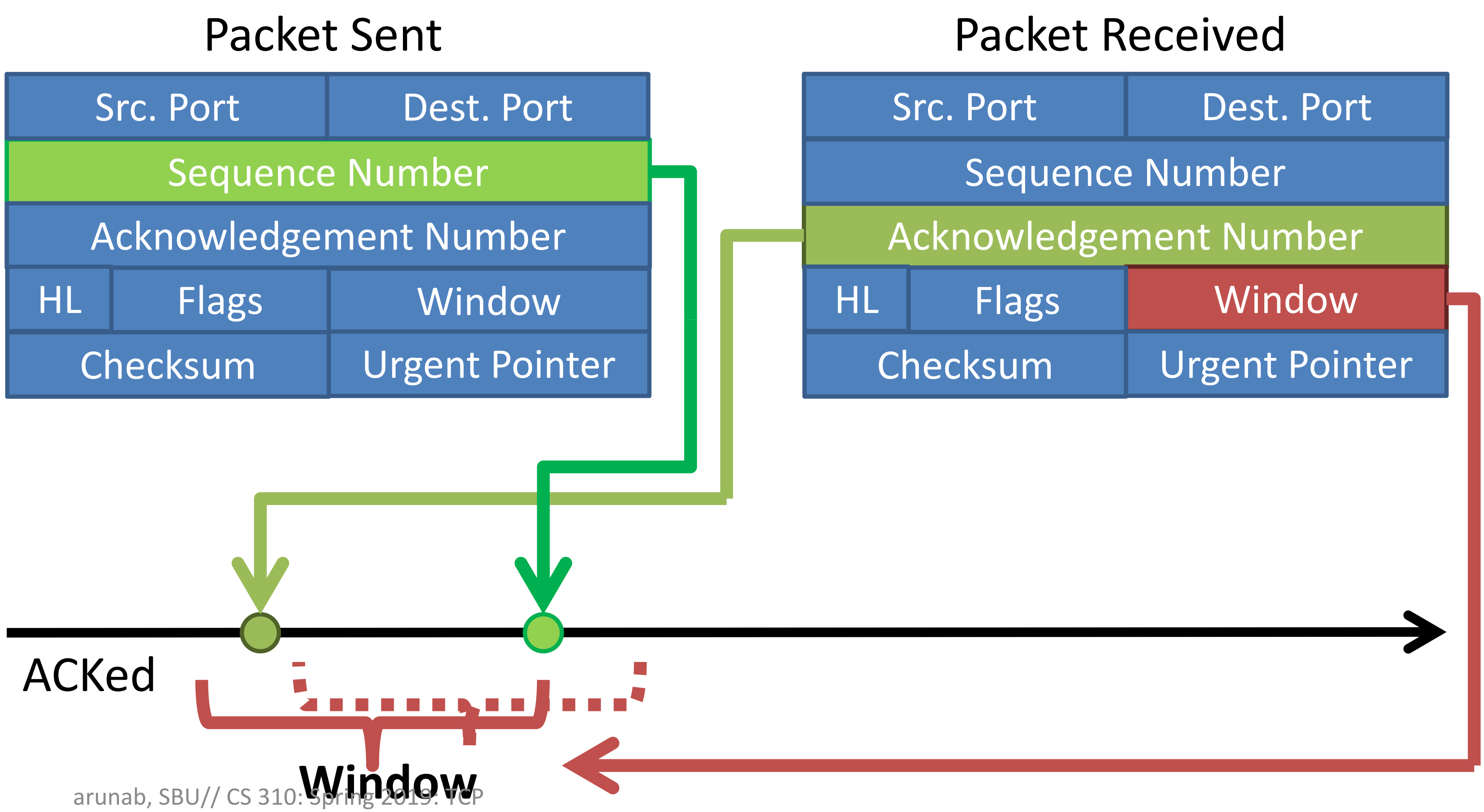
Sliding window

- TCP sends a window size of packets instead of stop-and-wait
- For window size *n*, sender may transmit *n* bytes without receiving an ACK
  - After each ACK, the window slides forward

Flow Control

- Problem: how many packets should a sender transmit?
  - Too many packets may overwhelm the receiver
- Solution
  - Receiver tells the sender how big their buffer is
  - This is called the advertised window

Flow Control: Sender Side

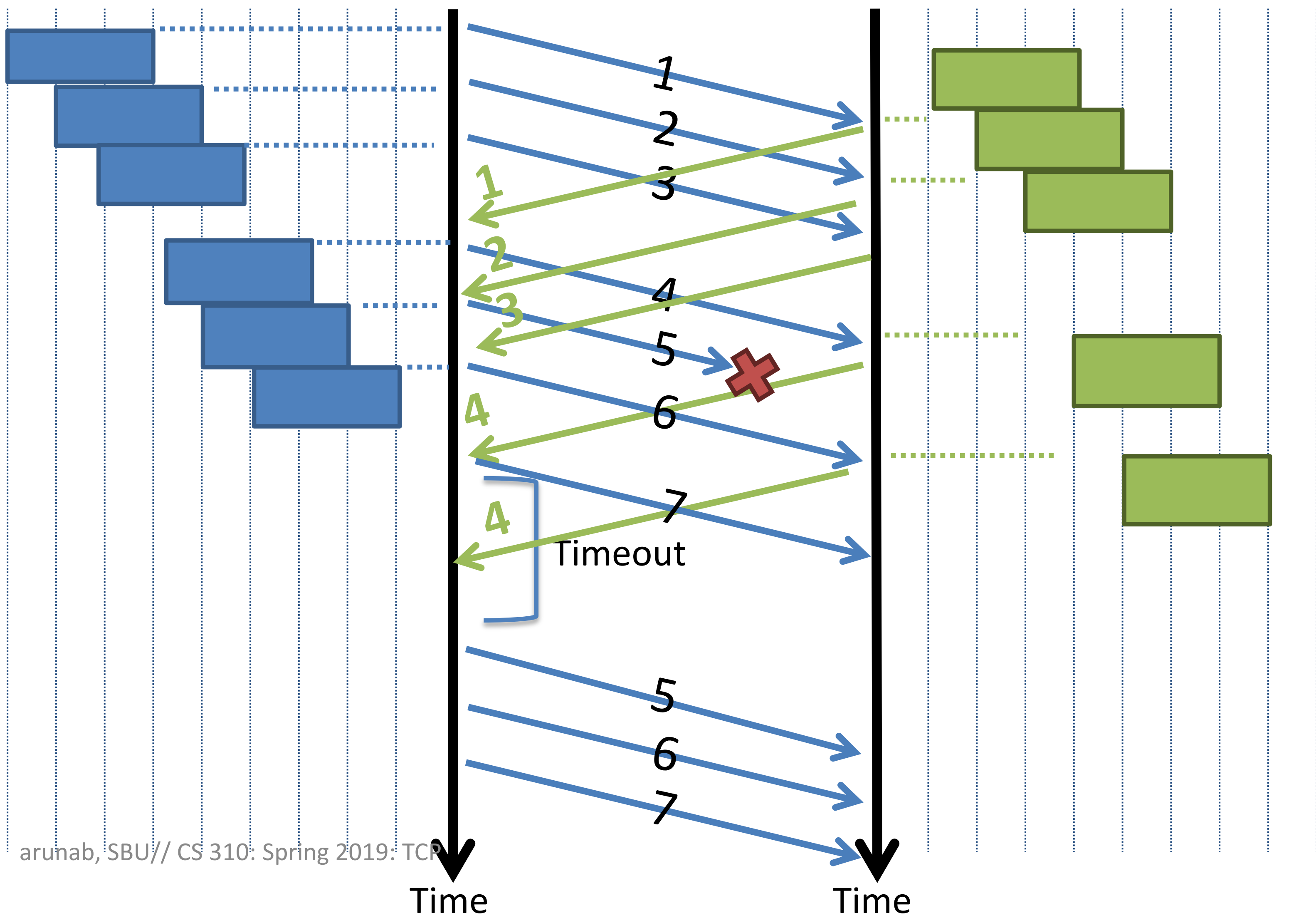


What Should the Receiver ACK?

1. ACK every packet
2. Use *cumulative ACK*, where an ACK for sequence n implies ACKS for all $k < n$

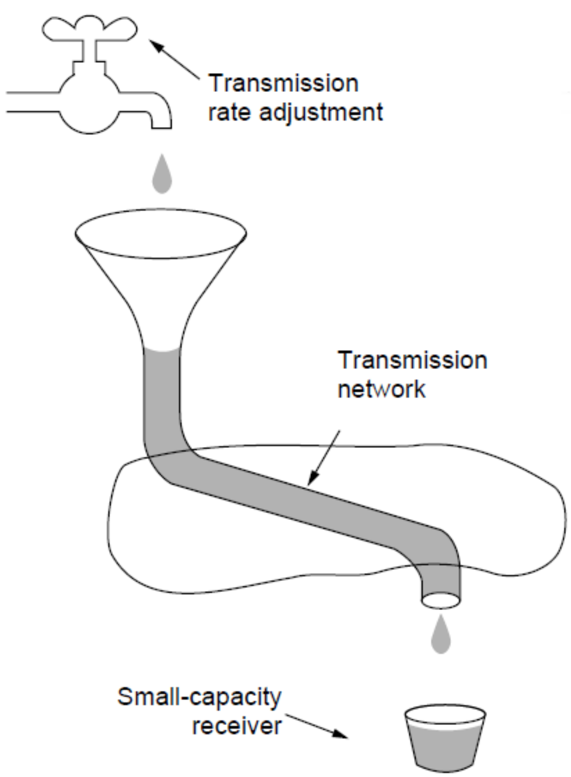
There are also other types of ack semantics.

Example: Sliding Window = 3

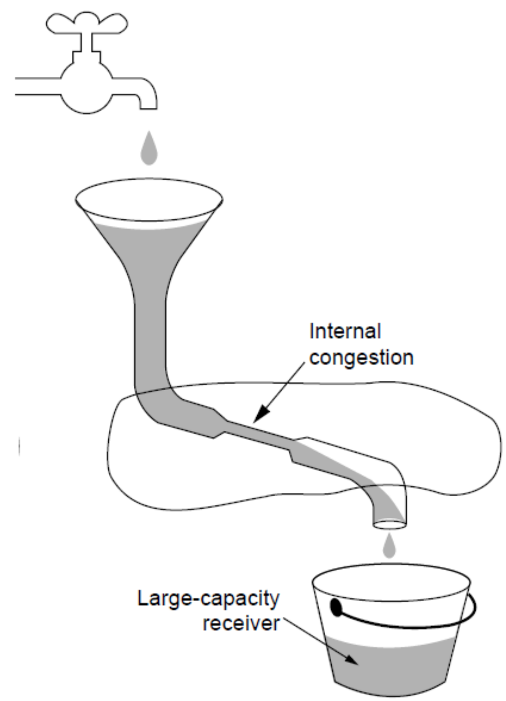


Congestion control

How is congestion related to sending rate?



A fast network feeding a low-capacity receiver → flow control is needed



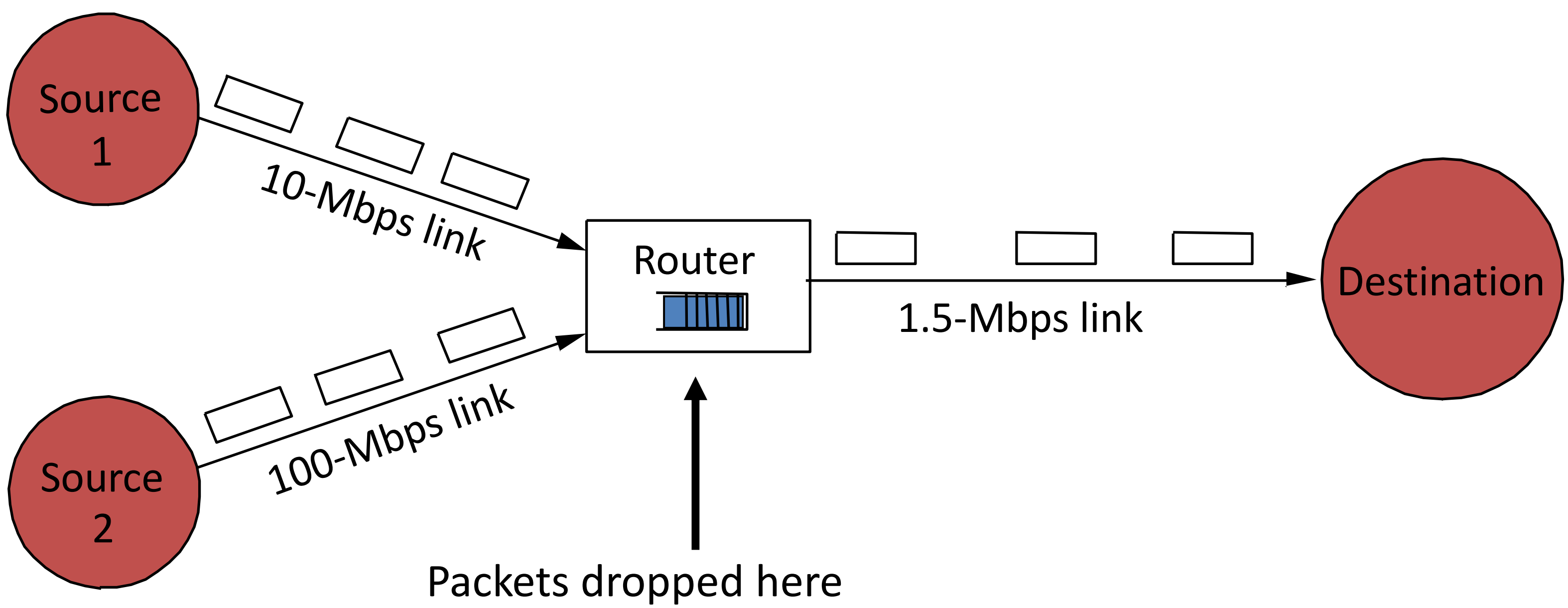
A slow network feeding a high-capacity receiver → congestion control is needed

Bandwidth Allocation

- How fast should the Web server send packets?
  - Two bottlenecks: Receiver-centric and Network-centric
- Receiver centric: Flow control

- Network centric
  - Congestion control: sending too fast will cause packets to be lost in the network
  - Fairness: different users should get their fair share of the bandwidth
  - Often treated together

Why does this congestion collapse occur?



- 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

Congestion and delay

- Packet delay = queuing delay at the buffer + transmission delay (usually very small)

- If buffer size is high
  - Queuing delay is high
- If buffer sizes are small
  - Higher packet loss

- Wireless networks are different!!

Challenges in bandwidth allocation for congestion control and fairness

- Who tells the end host
  - The available capacity or the operating point?
  - The presence of a bottleneck link?

- What happens if new flows join the network and flows leave (not necessarily from the same host)

- How to ensure fairness of flows that only have part overlapping paths?

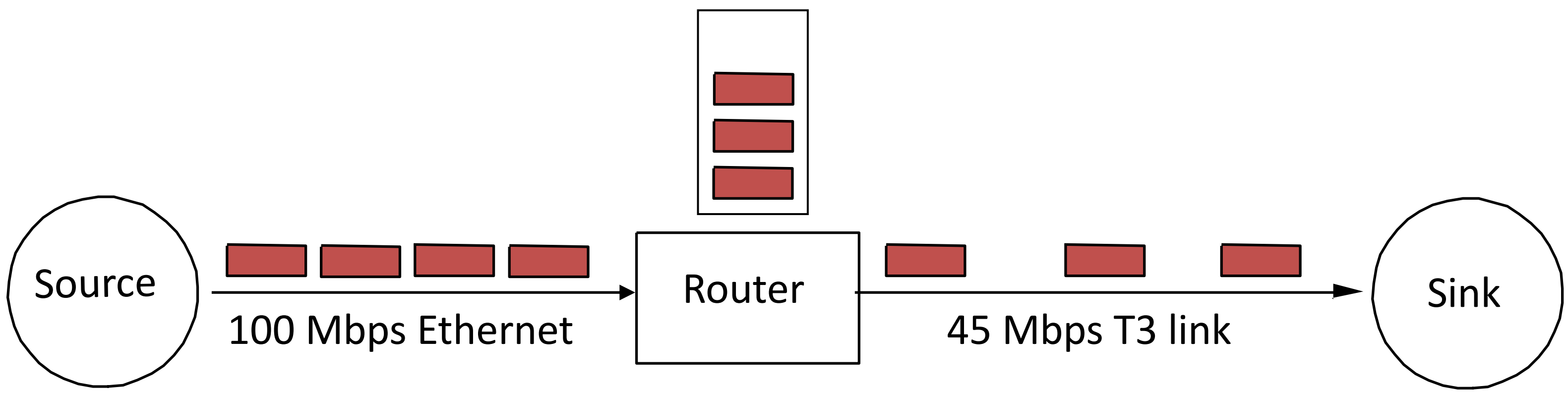
Possible approaches

- Do nothing, send packets indiscriminately
  - Many packets will drop, totally unpredictable performance
  - May lead to congestion collapse
- Reservations
  - Pre-arrange bandwidth allocations for flows
  - Requires negotiation before sending packets
  - Must be supported by the network
- Dynamic adjustment
  - Use probes to estimate level of congestion
  - Speed up when congestion is low
  - Slow down when congestion increases

TCP’s End-to-End argument

- End host “learns” the rate at which it can pump data into the network using probes
- End host “learns” the existence of a congestion
- End host automatically adapts sending rate according to congestion

First probe the network, but start slow



- Each source independently probes the network by sending packets at a slow rate
- The rate is then adapted.

How to ``learn” about bottlenecks?

- Implicit feedback: Losses = congestion

- Sending rate and losses are now intertwined
  - Increase the congestion window until there is a loss
  - Loss implies bottleneck, stop sending more data

- Reacting to losses also allows TCP to adjust as flows join or leave the network

AIMD: Additive Increase Multiplicative decrease

- Increase the sending rate slowly
- On loss, reduce the sending rate rapidly
- Sending rate == Window size.
  - Called the congestion window size of *cwnd*.

TCP congestion control idea

- Send congestion window size of packets (cwnd)

- Use AIMD to adapt cwnd
  - Increase cwnd slowly when bandwidth is available
  - Reduce cwnd rapidly when loss occurs.

- Use packet loss as a signal for congestion.

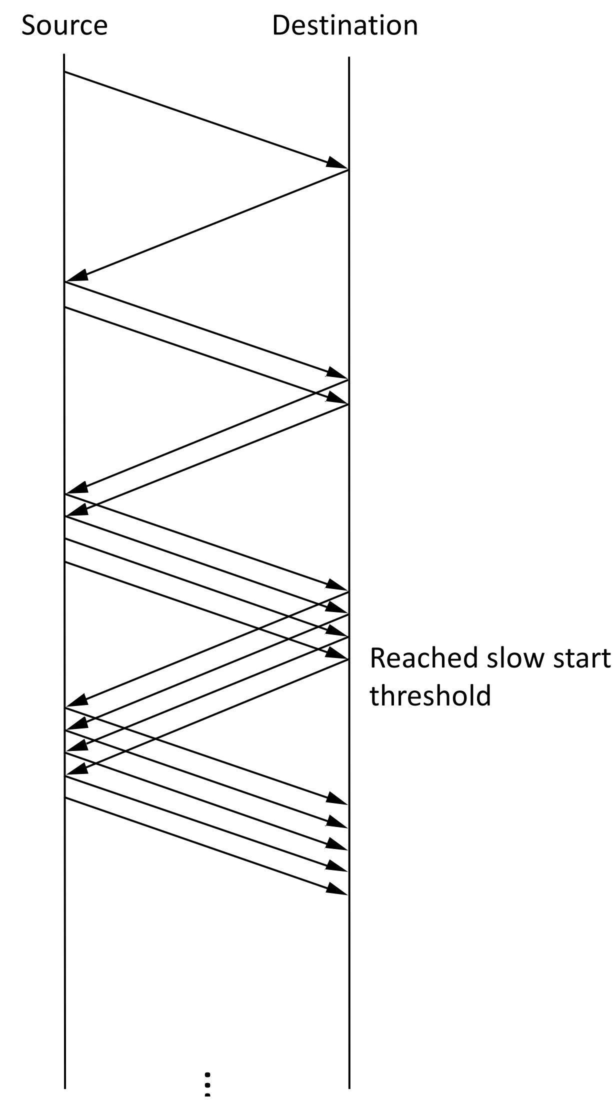
TCP congestion control algorithm:

- TCP start up problem
  - Starting very slowly can waste bandwidth
  - Starting too quickly can cause congestion

- Solution: Start quickly up to a point (slow start phase), and then go slow (congestion avoidance phase).

TCP "Slow Start"

- Until slow start threshold
  - Double cwnd every RTT
  - Cwnd *= 2 / RTT
- When the slow start threshold is reached, start additive increase
  - Cwnd +=1 / packet received



TCP congestion control algorithm: Tahoe(1 of 2)

- Slow start phase (actually not slow)
  - First send a small number of packets == initial congestion window size (icwnd)
  - For every RTT, double cwnd (i.e., for every ack send an additional packet)
  - (If cwnd >= slow start threshold *ssthresh*)
    - Go to congestion avoidance phase.
  - If there is a loss
    - cwnd = icwnd, ssthresh = cwnd/2.

TCP congestion control algorithm: Tahoe (2 of 2)

- Congestion avoidance phase
  - Send cwnd packets
  - For every RTT, increase cwnd by 1 (i.e., for every ack, increase cwnd by 1/cwnd)
  - When timeout occurs
    - cwnd = icwnd, ssthresh = cwnd/2

- This version is called TCP Tahoe.

But timeout is very expensive: Instead, use Fast Retransmit

- Fast Retransmit
  - Duplicate acks are a signal that a packet is lost
  - When triple duplicate acks arrive, assume that the packet is lost and retransmit the packet
  - How does this help?

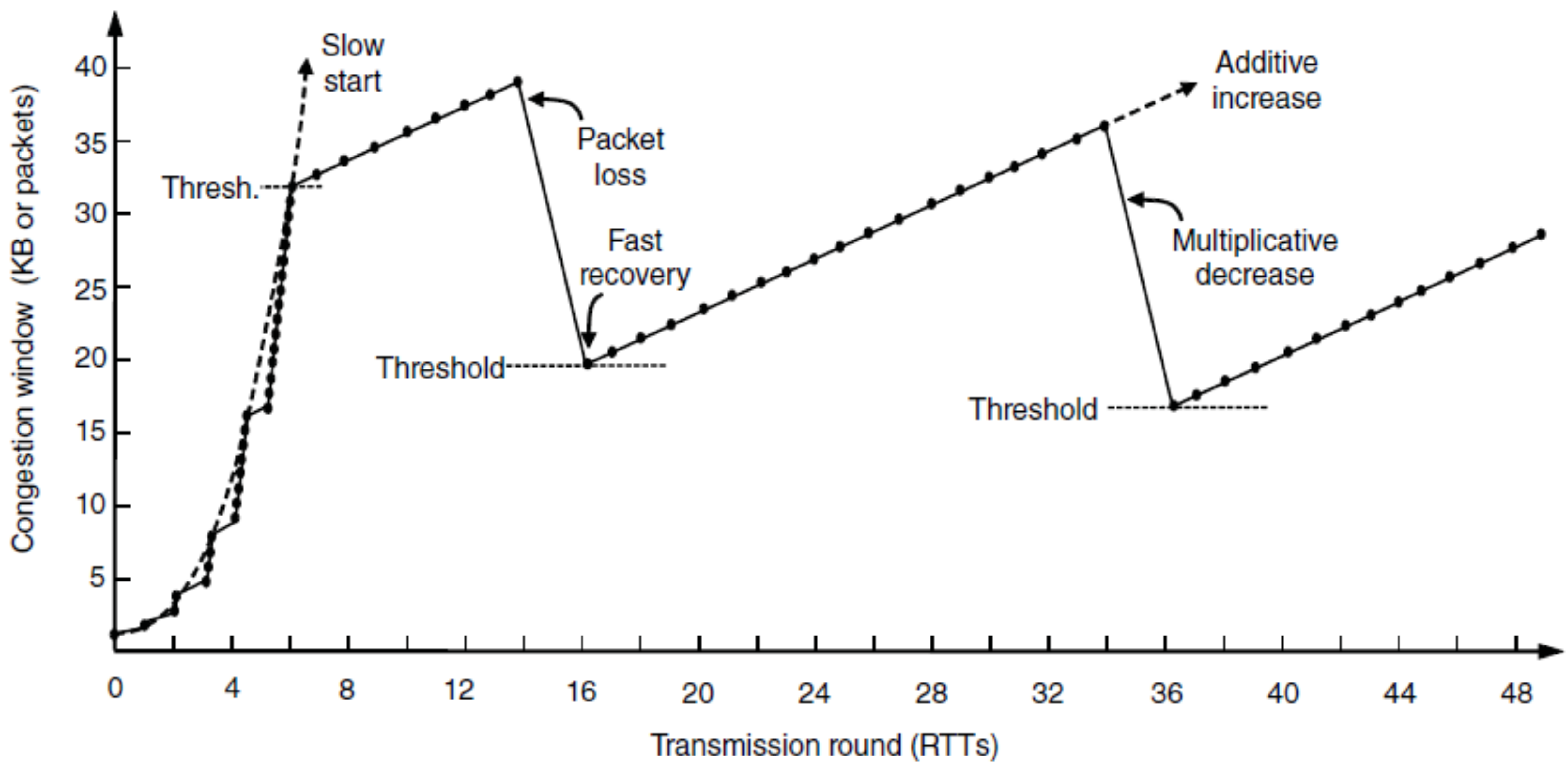
TCP with Fast Retransmit & Fast Recovery

- Congestion avoidance stage
  - When timeout occurs, same as before
  - On triple duplicate ack
    - Resend lost packet
    - ssthresh = cwnd/2, cwnd = cwnd/2

- Slow start
  - When timeout occurs, same as before
  - On triple duplicate ack
    - Resend lost packet
    - ssthresh = cwnd/2, cwnd = ssthresh (directly to congestion avoidance)

- This is TCP reno.

TCP + fast recovery + fast retransmit



Assume no timeout!