Congestion Control

Recap: Flow Control vs. Congestion Control

- Flow control
  - Prevents *one sender* from overrunning the capacity of a *slow receiver*
- Congestion control
  - Prevents a *set of senders* from overloading the *network*!

Learning Objectives

- **Congestion control basics**
- Queuing disciplines
- TCP Congestion Control

Congestion Control Basics

- Problem
  - Demand for network resources can grow beyond the resources available
  - Want to avoid network congestion while providing “fair” amount to each user
- Example scenarios
  - Bandwidth between Chicago and San Francisco
  - Bandwidth in a network link
  - Buffer in a router/switch

Congestion Collapse

- Definition
  - Sending more traffic actually causes the goodput to drop — often dramatically
  - i.e., network is busy but little useful work is getting done
- Many possible causes
  - Spurious retransmissions of packets still in flight
    - Classical congestion collapse
    - Solution: better retransmission timer and congestion control
  - Undelivered packets
    - Packets consume resources and are dropped elsewhere in network due to buffer overflow
    - Solution: congestion control for *all* traffic

Dealing with Congestion

- Range of solutions
  - Congestion control
    - Cure congestion when it happens
  - Congestion avoidance
    - Predict when congestion might occur and avoid causing it
  - Resource allocation
    - Prevent congestion from occurring
- Model of network
  - Packet-switched network (or internet)
  - Connectionless flows (logical channels/connections) between hosts

04/16 Lecture

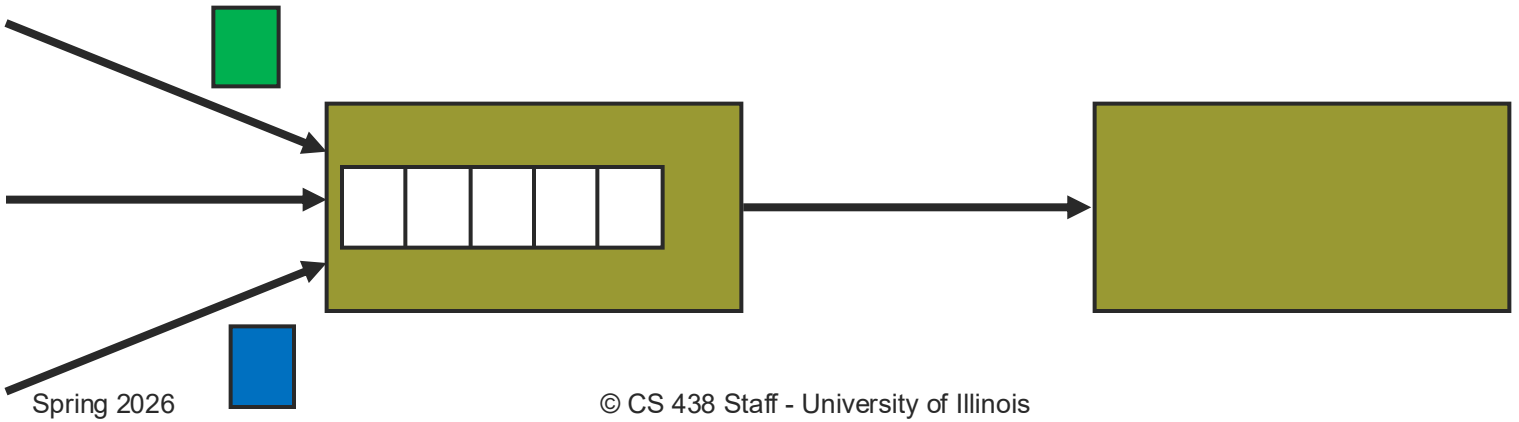
- Announcements:
  - Minor edits to HW4 handout
  - Check Campuswire and redownload
- Big picture:
  - TCP implements a specific type of congestion control
    - end-to-end, feedback-based, window-based
  - But congestion control is broader and more fundamental
    - related to resource allocation and queuing theory
  - Relevance: it remains an active research topic

Congestion Control

- Goal
  - Effective and fair allocation of resources among a collection of competing users
  - When to say no and to whom
- Resources
  - Bandwidth
  - Buffers
- Problem
  - Contention at routers causes packet delays & loss

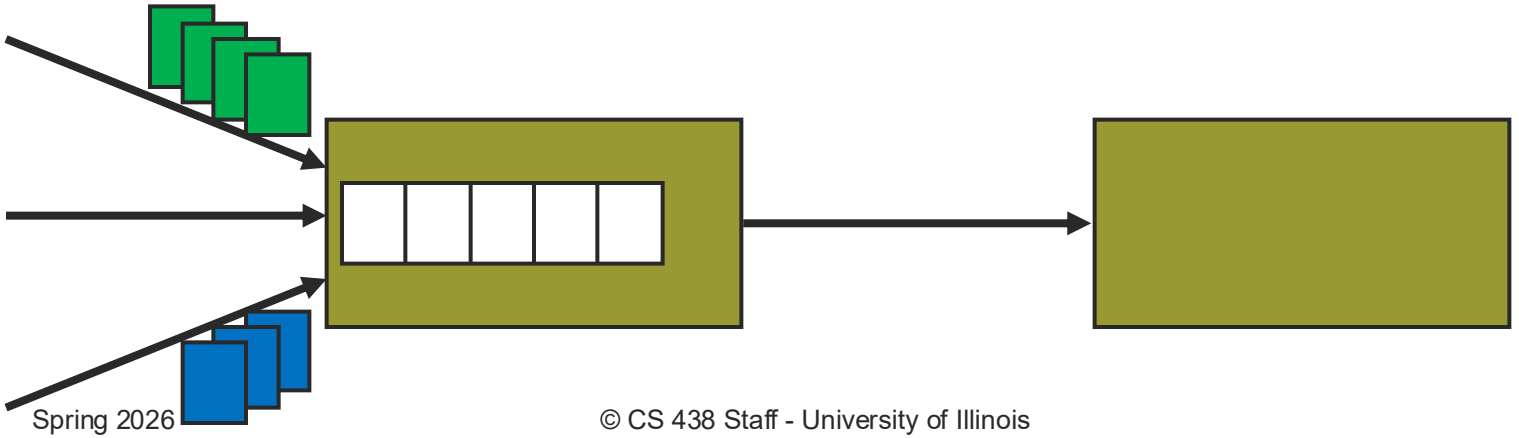
Congestion is Natural

- Because network traffic is bursty!
- If two packets arrive at the same time
  - Router can only transmit one and buffer the other



Congestion is Natural

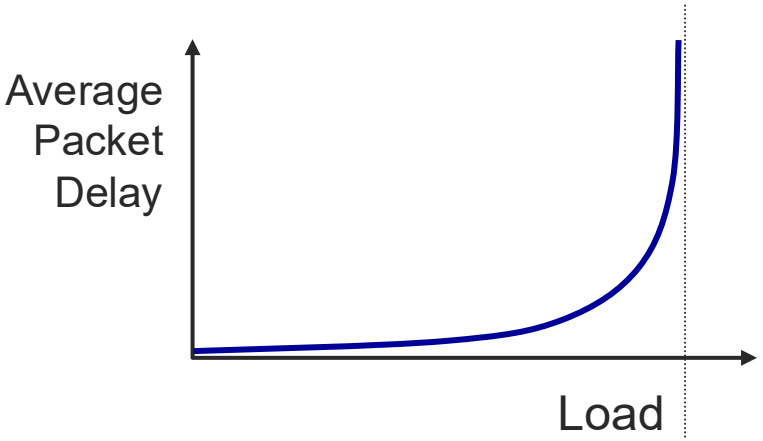
- Because network traffic is bursty!
- If two packets arrive at the same time
  - Router can only transmit one and buffer the other
- If many packets arrive in a short period of time
  - Router cannot keep up with the arriving traffic
  - Causes delays, and the buffer may eventually overflow



Throughput and Delay

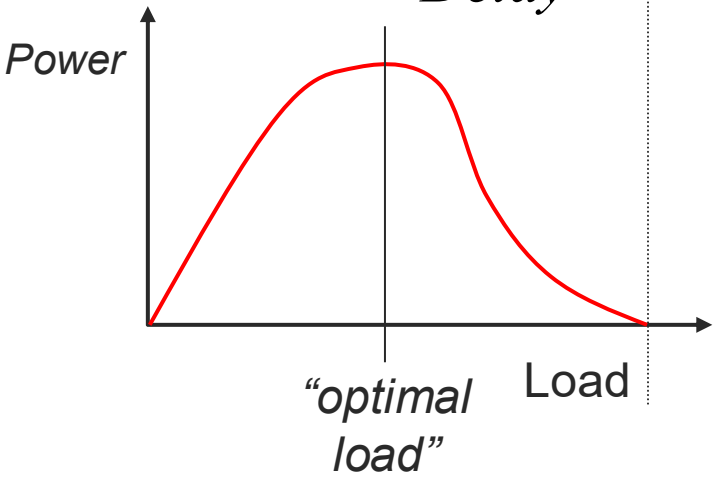
Ideal: low delay & high throughput
Reality: must balance the two

Typical behavior of queueing systems with bursty arrivals:



Maximizing "power" is an example

$$Power = \frac{Load}{Delay}$$

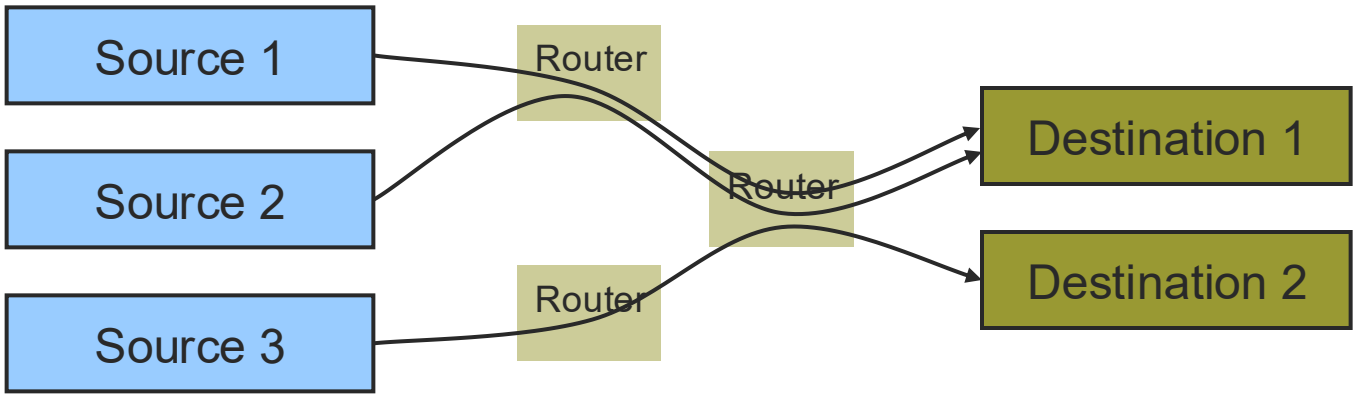


Basic Design Choices

- Prevention or Remedy?
  - Pre-allocate resources to avoid congestion
  - Send data and control congestion if and when it occurs
- Possible implementation points
  - Hosts at the edge of the network
    - Transport protocol (e.g., TCP)
  - Routers inside the network
    - Queueing disciplines
- Underlying service model
  - Best effort vs. quality of service (QoS)

Flows

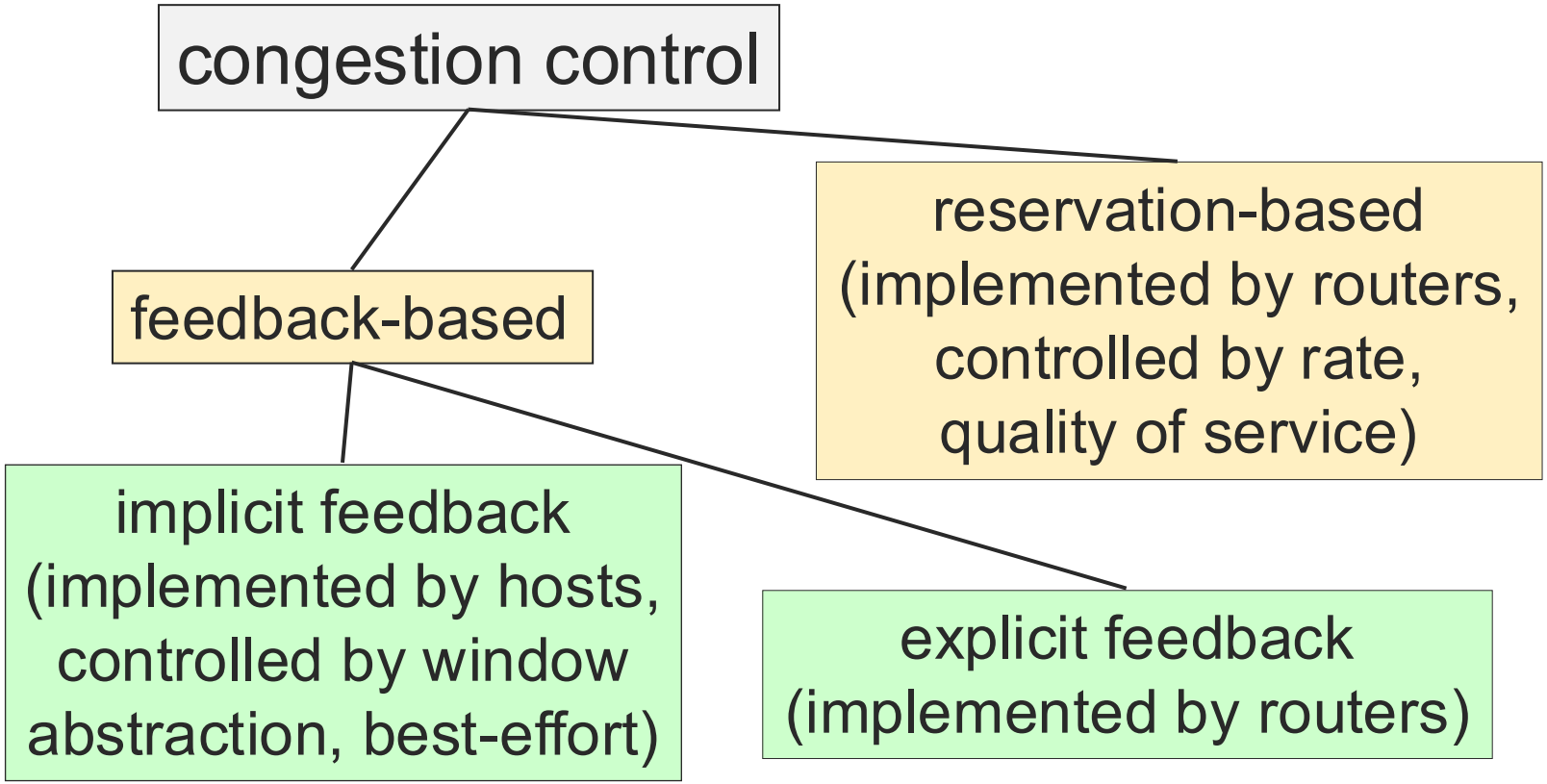
- Flow abstraction: Sequence of packets sent between a source/destination pair
- Visible to the routers
  - Per-flow state *can* be maintained at routers



Congestion Control

- Router role
  - Controls forwarding and dropping policies
  - Can send feedback to source
- Host role
  - Monitors network conditions
  - Adjusts accordingly
- Routing vs. congestion
  - Effective adaptive routing schemes can sometimes help congestion (not always)

Congestion Control Taxonomy



Router-Centric vs. Host-Centric

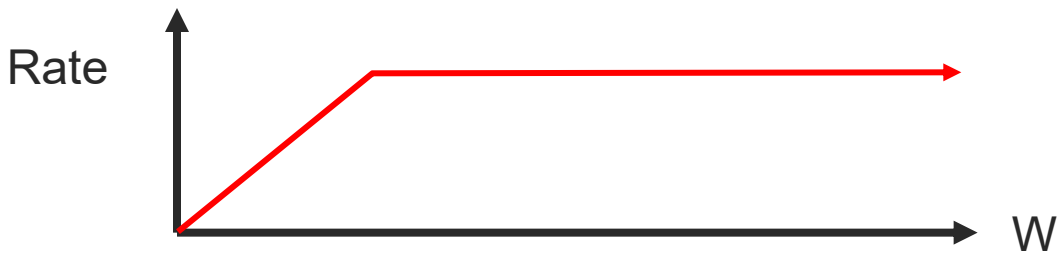
- Router-centric
  - Each router takes responsibility for deciding
    - When packets are forwarded
    - Which packets are dropped
    - Informing hosts of sending limitations
- Host-centric
  - Hosts observe network conditions and adjust their behavior accordingly

Reservation-Based vs. Feedback-Based

- Reservation-based
  - End host asks network for capacity at flow establishment time
  - Routers along flow's route allocate appropriate resources
  - If resources are not available, flow is rejected
  - Implies the use of router-centric mechanisms
- Feedback-based
  - End host begins sending without asking for capacity
  - End host adjusts sending rate according to feedback
    - Explicit vs. implicit feedback mechanisms
  - May use router-centric (explicit) or host-centric (implicit) mechanisms

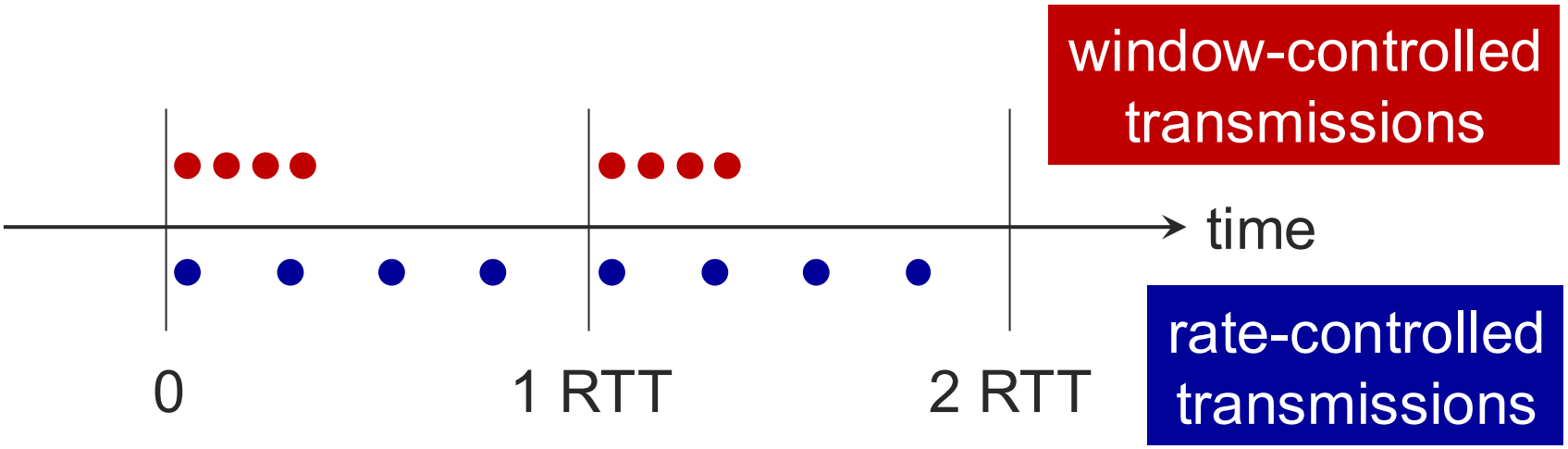
Window-based vs. Rate-based

- Given an RTT and window size W, long term throughput rate is
  - Rate = min(link speed, W/RTT)
- Since rate can be controlled by the window size, is there really any difference between controlling the window size and controlling the rate?



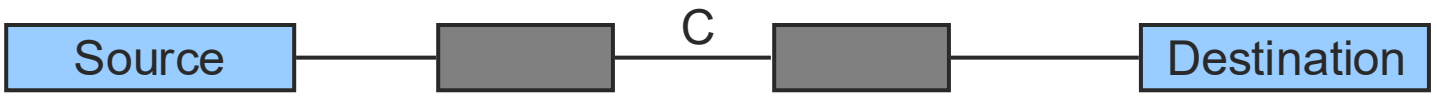
Window-based ≠ Rate-based

- RTT is unknown
- Different transmission patterns: “self-clocking” vs. “evenly spread out”
- Window directly corresponds to buffer space
- Rate is more suitable for multimedia applications and reservation-based control (e.g., to provide different QoS)



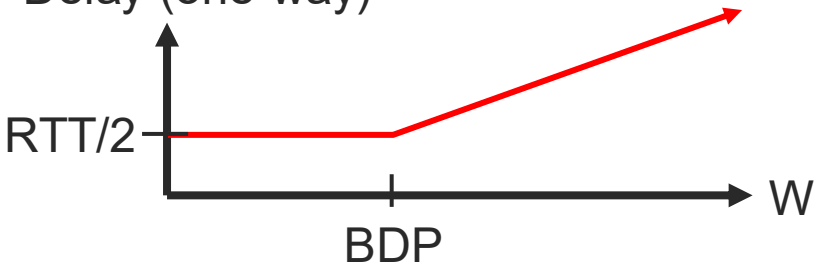
Window Size and BDP

For non-random network with bottleneck capacity C:

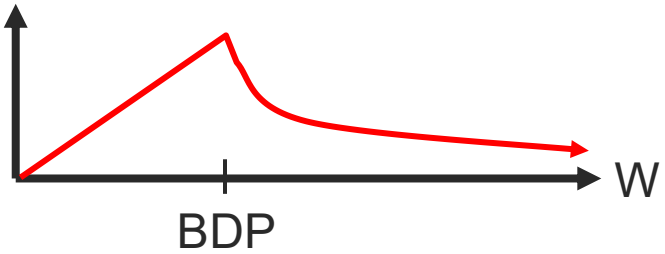


Rate = Throughput

Delay (one-way)



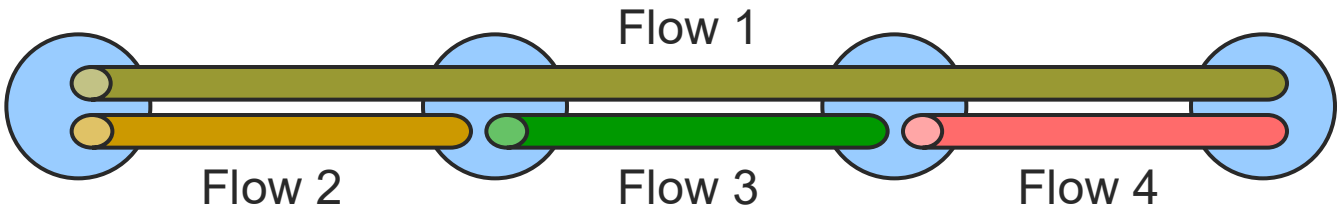
Power = throughput/delay



Fairness

- Goals
  - Allocate resources “fairly”
  - Isolate ill-behaved users
  - Still achieve statistical multiplexing
    - One flow can fill entire pipe if no contenders
    - Work conserving → scheduler never idles link if it has a packet
- At what granularity?
  - Flows, connections, domains?

What's Fair?



Globally Fair?
F1 = C/4, F2 = F3 = F4 = 3C/4

or

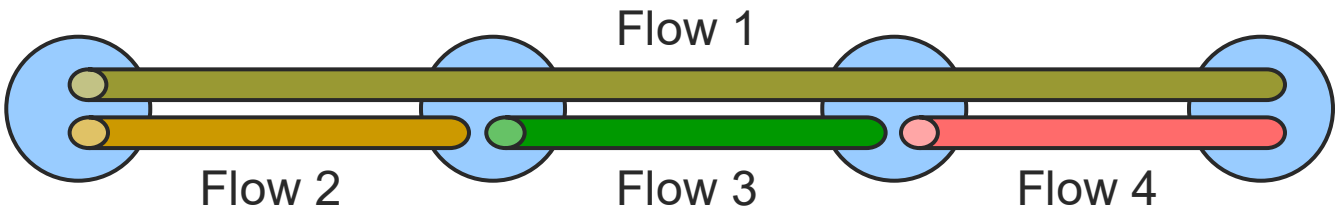
Locally Fair?
F1 = F2 = F3 = F4 = C/2

Max-Min Fairness

Max-min fairness:

1. The allocation (x_1, ..., x_n) is feasible
2. No x_i can be increased without strictly decreasing some $x_j \leq x_i$

Question: What's the max-min fair allocation?



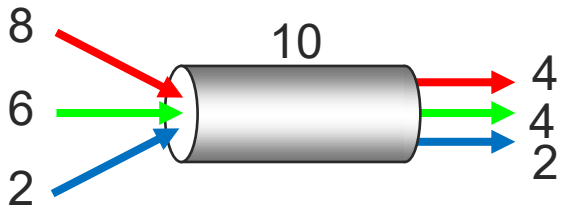
Consider demands:

1) No user receives more than requested bandwidth
2) No other allocation with 1) has higher min bandwidth
3) 2) remains true recursively on removing minimal user

$x_i = MIN(x_{fair}, r_i)$

Max-Min Fairness: Example

- Capacity(C) = 10
  - 3 flows with demands: r1 = 8, r2 = 6, r3 = 2
- C/3 = 3.33 →
  - Can service all of r3
  - Remove r3 from the accounting: C = C – r3 = 8; N = 2
- C/2 = 4 →
  - Can't service all of r1 or r2
  - So hold them to the remaining fair share: f = 4



Learning Objectives

- Congestion control basics
- **Queuing disciplines**
- TCP Congestion Control

Queueing Disciplines

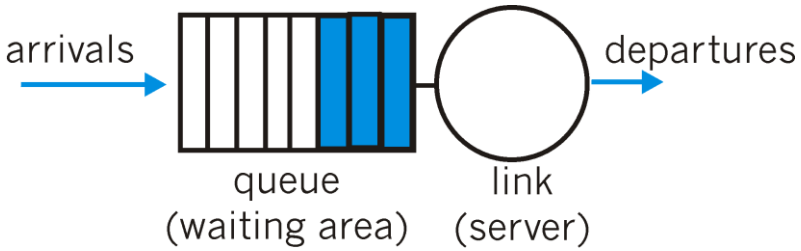
- Goal
  - Decide how packets are buffered while waiting to be transmitted
  - Provide protection from ill-behaved flows
  - Each router MUST implement some queuing discipline regardless of what the resource allocation mechanism is
- Impact
  - Directly impacts buffer space usage
  - Indirectly impacts congestion control

Queueing Disciplines

- Allocate bandwidth
  - Which packets get transmitted
- Allocate buffer space
  - Which packets get discarded
- Affect packet latency
  - When packets get transmitted

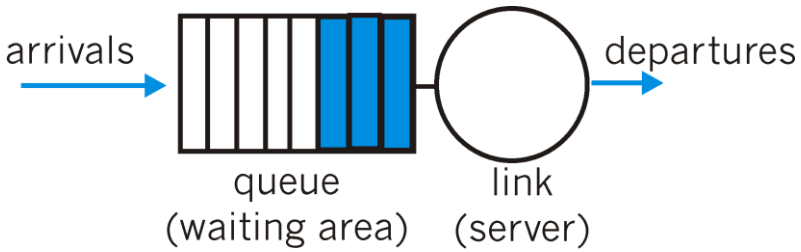
Scheduling Policies

- FIFO (First In First Out) a.k.a. FCFS (First Come First Serve)
  - Service
    - In order of arrival to the queue
  - Drop policy
    - Tail drop: Packets that arrive to a full buffer are dropped
    - Generally: drop policy determines which packet to discard (new arrival or something already queued)



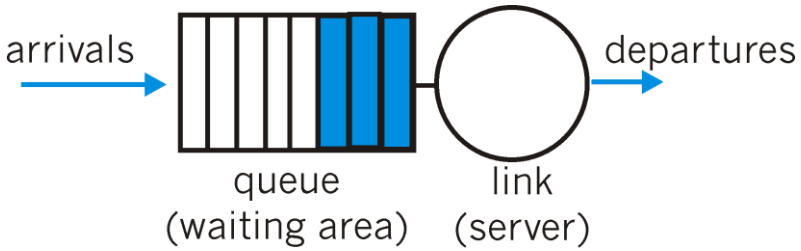
Scheduling Policies

- FIFO (First In First Out)
  - Problem 1: send more packets, get more service
    - Selfish senders trying to grab as much as they can
    - Malicious senders trying to deny service to others
  - Problem 2: not all packets should be equal



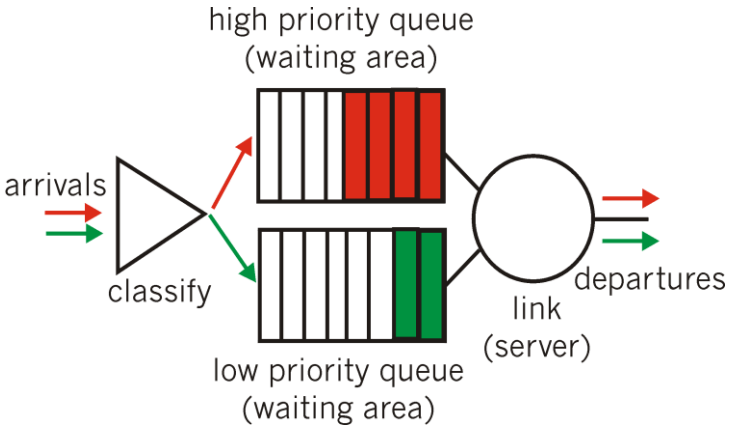
Scheduling Policies

- FIFO with tail drop
  - Simplest, most widely used in Internet routers
  - Treats all packets the same no matter of their sources
  - Fairness for latency
  - Bandwidth not considered (not good for congestion)
  - Congestion control left to the sources



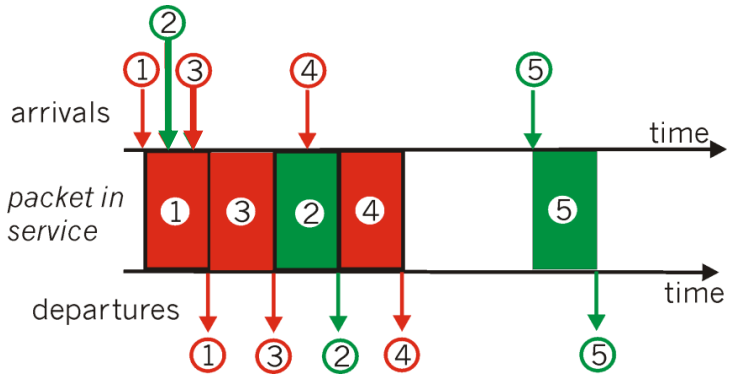
Scheduling Policies

- Priority Queuing
  - Classes have different priorities
    - May depend on explicit marking or other header info
      - e.g., IP source or destination, TCP port numbers, etc.
  - Service
    - Transmit packet from highest priority class with a non-empty queue



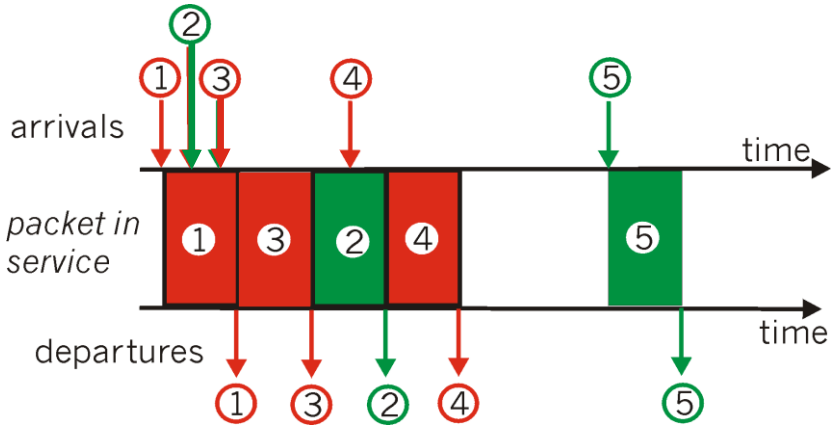
Scheduling Policies

- Priority Queuing
  - Isolation for the high-priority traffic
    - Almost like it has a dedicated link
  - Except for a small transmission delay when
    - High-priority packet arrives during transmission of low-priority
    - Router completes sending the low-priority traffic first
  - Preemptive variant postpones the low-priority processing



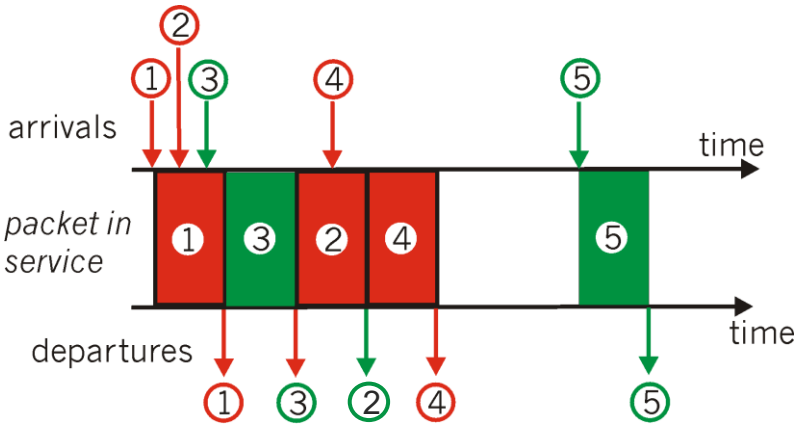
Scheduling Policies

- Limitation of priority queuing?
  - May starve lower priority flows
- How can we address this?



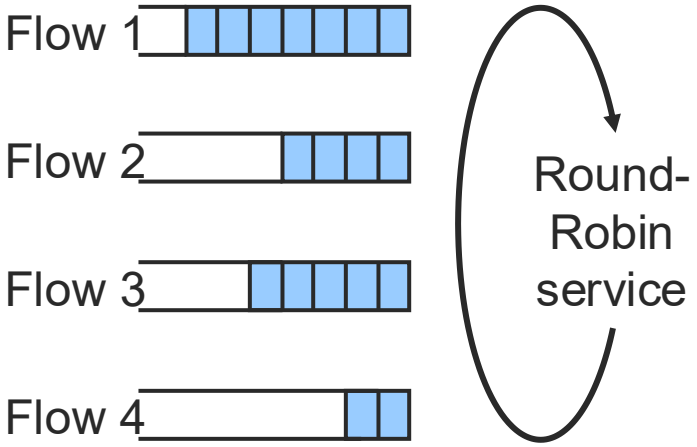
Scheduling Policies

- Round Robin
  - Each flow gets its own queue
  - Circulate through queues, process one packet (if queue non-empty), then move to next queue



Scheduling Policies

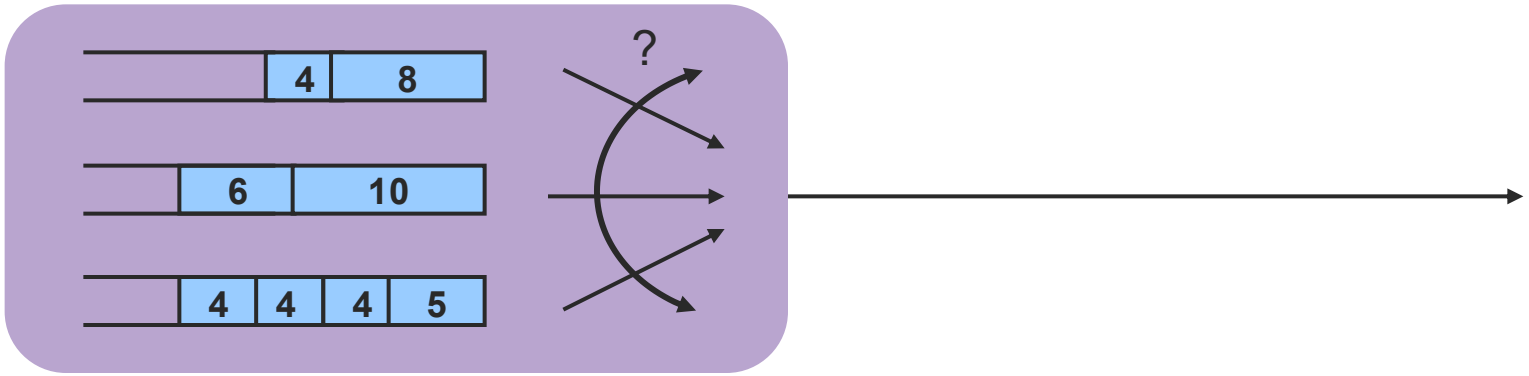
- Fair Queueing (FQ)
  - Explicitly segregates traffic based on flows
  - Ensures no flow captures more than its share of the capacity
  - Work conserving
    - Never leave the link idle if there is a packet to send
    - Unused bandwidth will be evenly divided between other sources
  - Fairness for bandwidth
  - Delay not considered



Each flow is guaranteed ¼ of capacity

Fair Queueing with Variable Packet Length

- How should we implement FQ if packets are not all the same length?
  - Bit-by-bit round-robin
    - Not feasible to implement, must use packet scheduling
    - Solution: approximate



Fair Queueing with Variable Packet Length

- Idea
  - Let S_i = amount of service flow i has received so far
  - Always serve a flow with minimum value of S_i
    - Can also use minimum (S_i + next packet length)
  - Upon serving a packet of length P from flow i, update:
    - $S_i = S_i + P$

Fair Queueing with Variable Packet Length

- Problem
  - A flow resumes sending packets after being quiet for a long time
- Effect
  - Such a flow could be considered to have "saved up credit"
  - Can lock out all other flows until credits are level again
- Solution
  - Enforce "use it or lose it policy"
    - Compute S_{min} = min(S_i such that queue i is not empty)
    - If queue j is empty, set $S_j = S_{min}$

Fair Queueing with Variable Packet Length

- Problem
  - A flow resumes sending packets after being quite for a long time
- Effect
  - Such a flow could be considered to have “saved up credit”
  - Can lock out all other flows until credits are level again
- Solution
  - Enforce “use it or lose it policy”
    - Compute S_{min} = min(S_i such that queue i is not empty)
    - If queue j is empty, set $S_j = S_{min}$

Extension: Weighted Fair Queueing

- Extend fair queueing
  - Notion of importance for each flow
- Suppose flow i has weight w_i
  - Example: w_i could be the fraction of total service that flow i is targeted for
- How to update the FQ implementation?
  - Currently, $S_i = S_i + P$
- Need only change basic update to
  - $S_i = S_i + P/w_i$

Fair Queuing Tradeoffs

- FQ can control congestion by isolating flows
  - Non-adaptive flows can still be a problem – why?
- Implementation challenges in routers?
  - Complex state: must keep queue per flow
    - Hard in routers with many flows (e.g., backbone routers)
    - Flow aggregation is one possibility (e.g., can do fairness per domain instead)
  - Complex computation
    - Classification into flows may be hard
    - Keep queues sorted by the amount of service received
    - Changes whenever the flow count changes

Learning Objectives

- Congestion control basics
- Queuing disciplines
- **TCP Congestion Control**

Host Solutions

- Host has very little information
  - Assumes best-effort network
  - Acts independently of other hosts
- Host actions
  - Reduce transmission rate below congestion threshold
  - Continuously monitor network for signs of congestion

Detecting Congestion

- How can a TCP sender determine that the network is congested?
- Network could tell it
  - ICMP Source Quench: during times of overload the signal itself could be dropped (and add to congestion)!
  - ECN: Explicit Congestion Notification
- Packet delays go up (knee of load-delay curve)
  - Tricky: noisy signal (delay often varies considerably)
- Packet loss
  - Fail-safe signal that TCP already has to detect
  - Complication: non-congestive loss

TCP Congestion Control

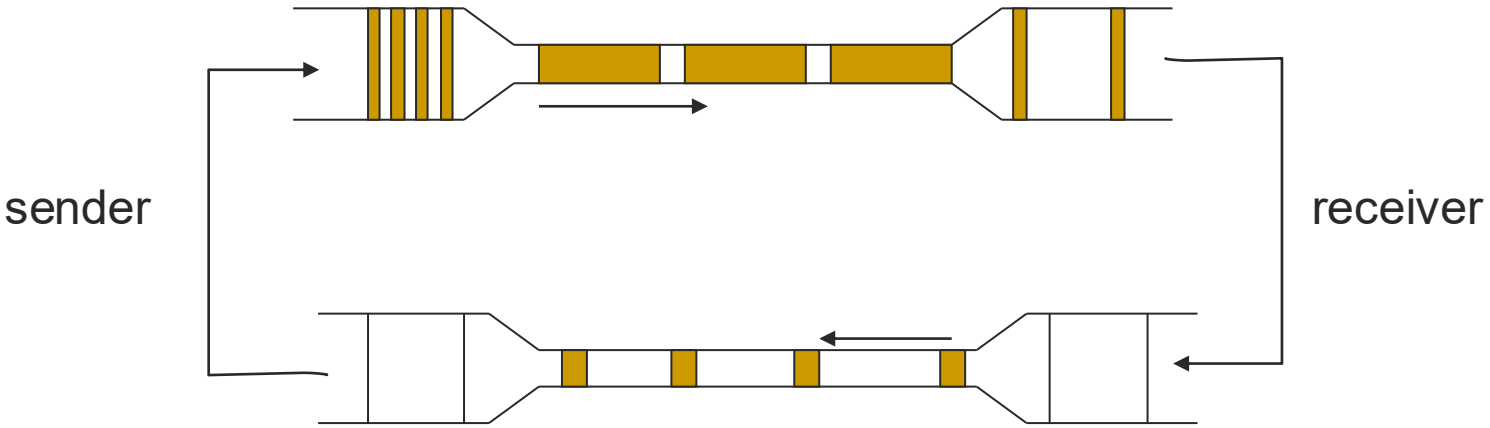
- Idea
  - Assumes best-effort network
    - FIFO or FQ
  - Each source determines network capacity for itself
  - Implicit feedback
  - ACKs pace transmission (self-clocking)
- Challenge
  - Determining initial available capacity
  - Adjusting to changes in capacity in a timely manner

TCP Congestion Control

- Basic idea
  - Add notion of congestion window
  - Effective window is smaller of
    - Advertised window (flow control)
    - Congestion window (congestion control)
  - Changes in congestion window size
    - Slow increases to absorb new bandwidth
    - Quick decreases to eliminate congestion

TCP Congestion Control

- Specific strategy
  - Self-clocking
    - Send data only when outstanding data ACK'd
    - Equivalent to send window limitation mentioned



TCP Congestion Control

- Additive Increase, Multiplicative Decrease (AIMD)
  - Growth
    - Linearly increase the congestion window
    - Add one maximum segment size (MSS) per congestion window of data ACK'd
  - Decrease
    - Cut window in half when congestion occurs

AIMD Rationale

- Objective
  - Adjust to changes in available capacity
- Basic idea
  - Consequences of over-sized window much worse than having an under-sized window
    - Over-sized window: packets dropped and retransmitted
    - Under-sized window: somewhat lower throughput

AIMD Rationale

- Tools
  - React to observance of congestion
  - Probe channel to detect more resources
- Observation
  - On notice of congestion
    - Decreasing too slowly will not be reactive enough
  - On probe of network
    - Increasing too quickly will overshoot limits

AIMD Implementation

- New TCP state variable
  - **`CongestionWindow`**
    - Similar to **`AdvertisedWindow`** for flow control
  - Limits how much data source can have in transit
    - **`MaxWin = MIN(CongestionWindow, AdvertisedWindow)`**
    - **`EffWin = MaxWin - (LastByteSent - LastByteAcked)`**
    - TCP can send no faster than the slowest component, network or destination

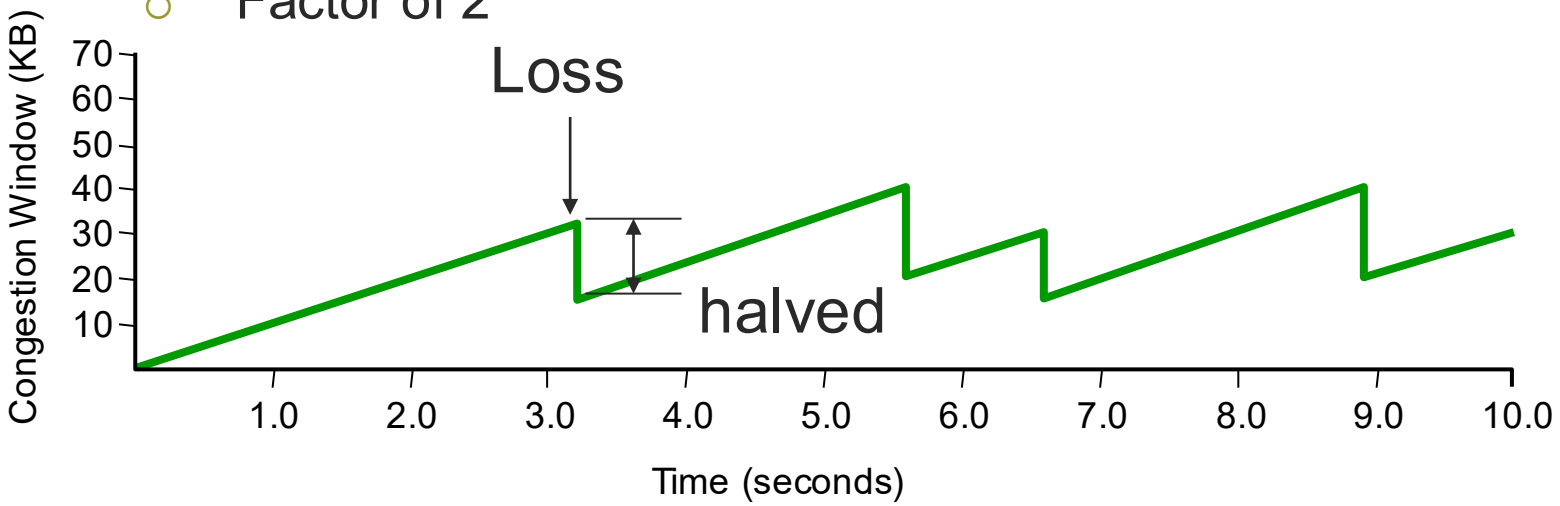
AIMD Implementation

- Increment **`CongestionWindow`** by one packet per RTT
  - In practice, increment a little per ACK
  - CWND += 1/CWND (if CWND is in packets)
  - **`CongestionWindow`** in bytes
    - **`Inc = MSS * MSS/CongestionWindow`**
    - **`CongestionWindow += Inc`**

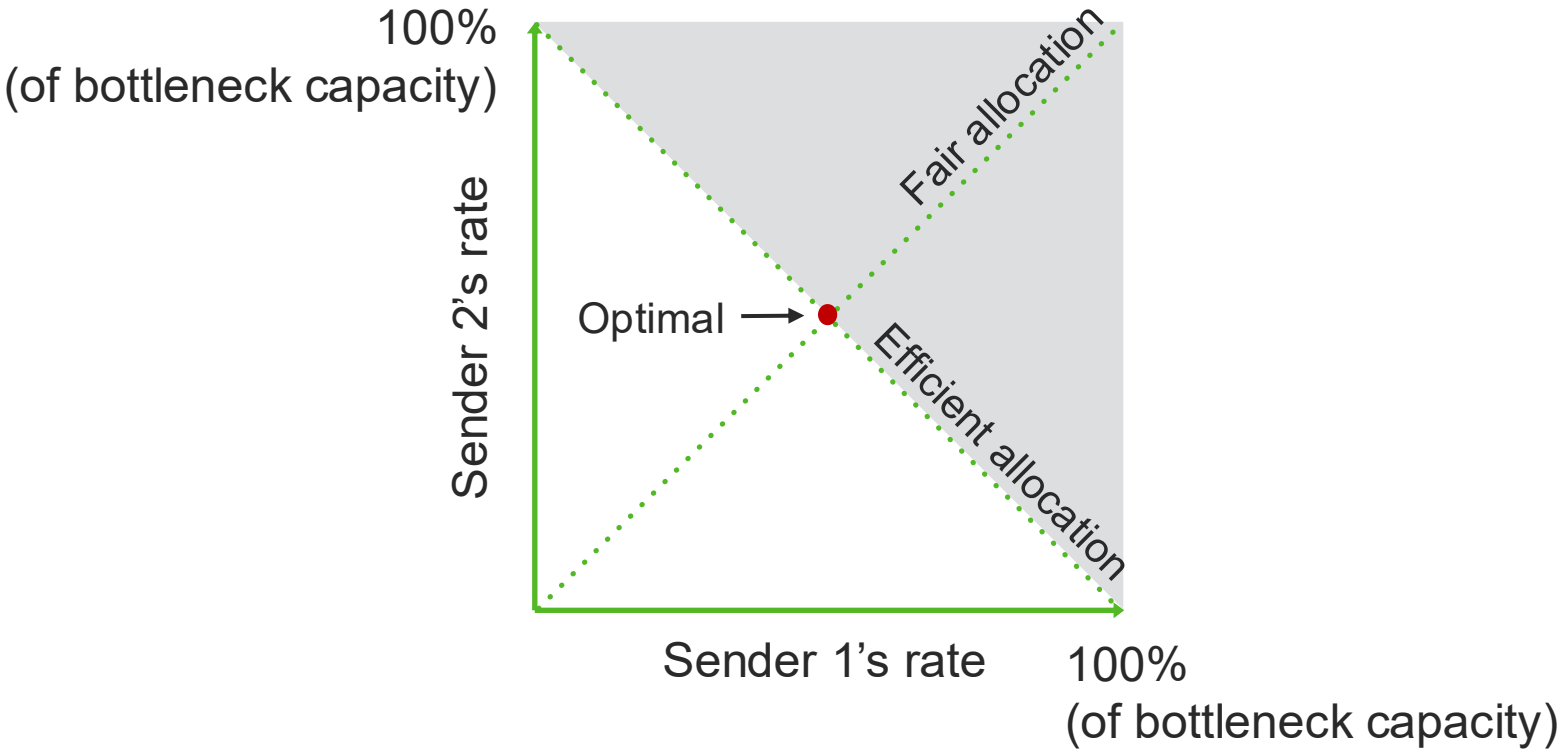
- Divide **`CongestionWindow`** by two whenever a timeout occurs
  - Timeout signals packet loss
  - Packet loss is rarely due to transmission error
  - Lost packet implies congestion

AIMD – Sawtooth Trace

- TCP periodically probes for available bandwidth by increasing its rate
- Packet loss is seen as sign of congestion and results in a multiplicative rate decrease
  - Factor of 2

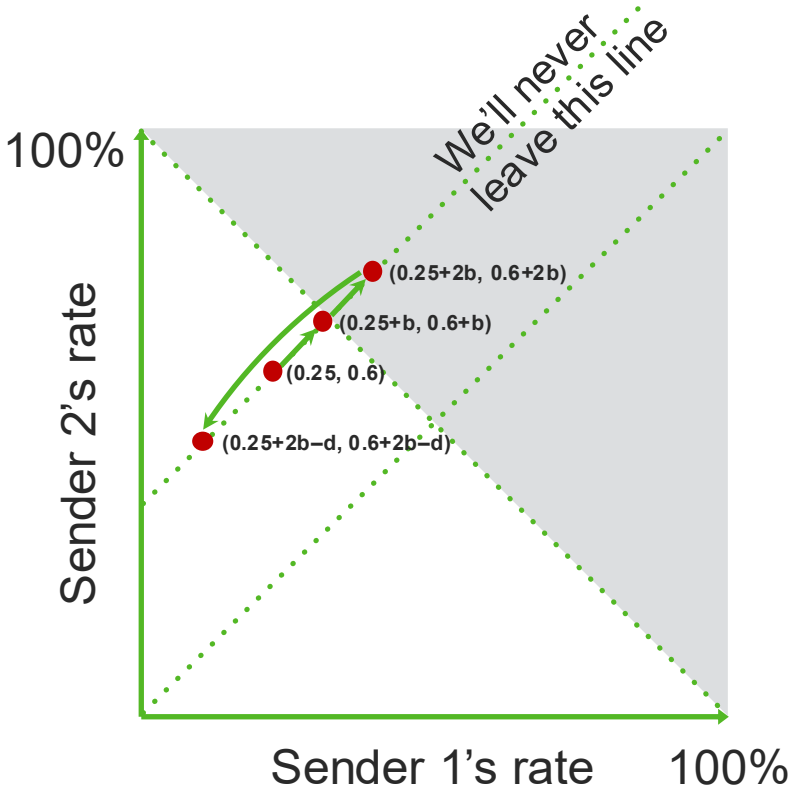


Why is AIMD fair?



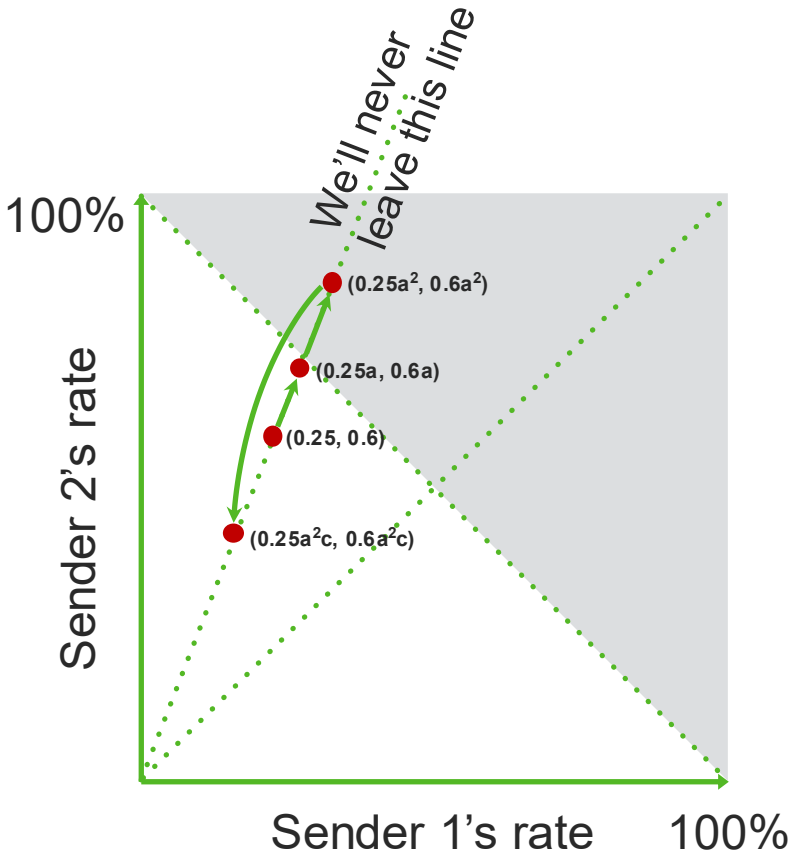
Two sender state diagram illustration adapted from Brighten Godfrey's slides

Additive Increase, Additive Decrease (AIAD)



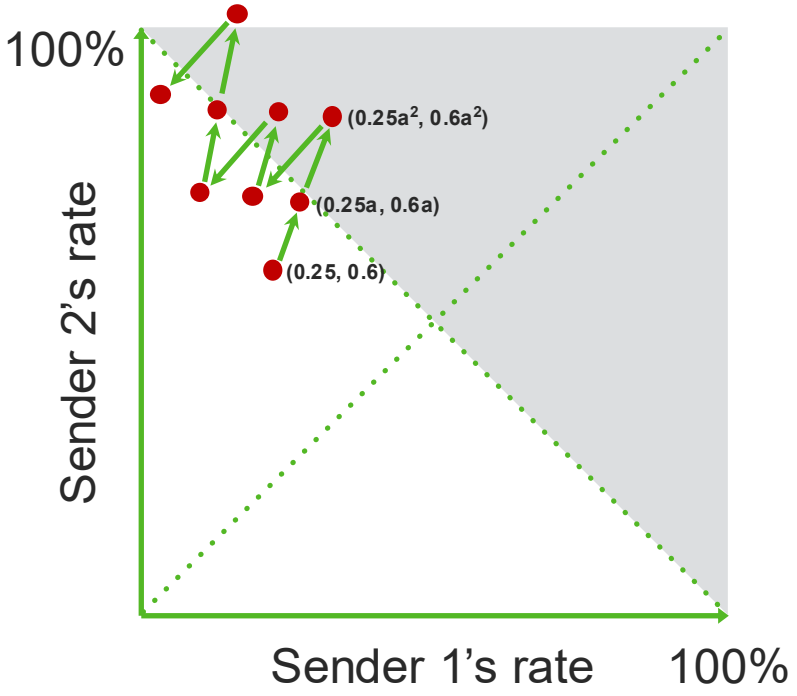
Does not converge to fair point

Multiplicative Increase, Multiplicative Decrease (MIMD)



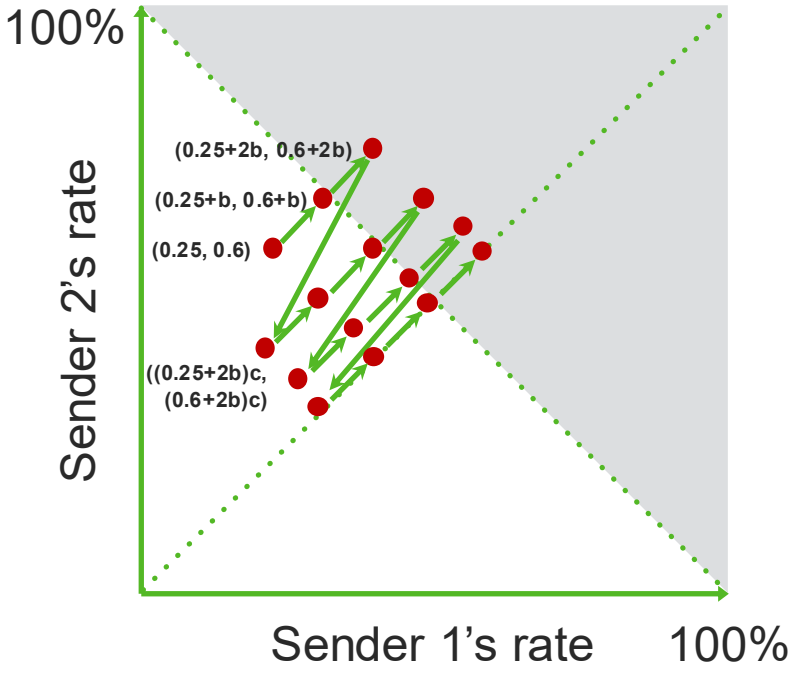
Does not converge to fair point

Multiplicative Increase, Additive Decrease (MIAD)



Converges to one flow dominating!

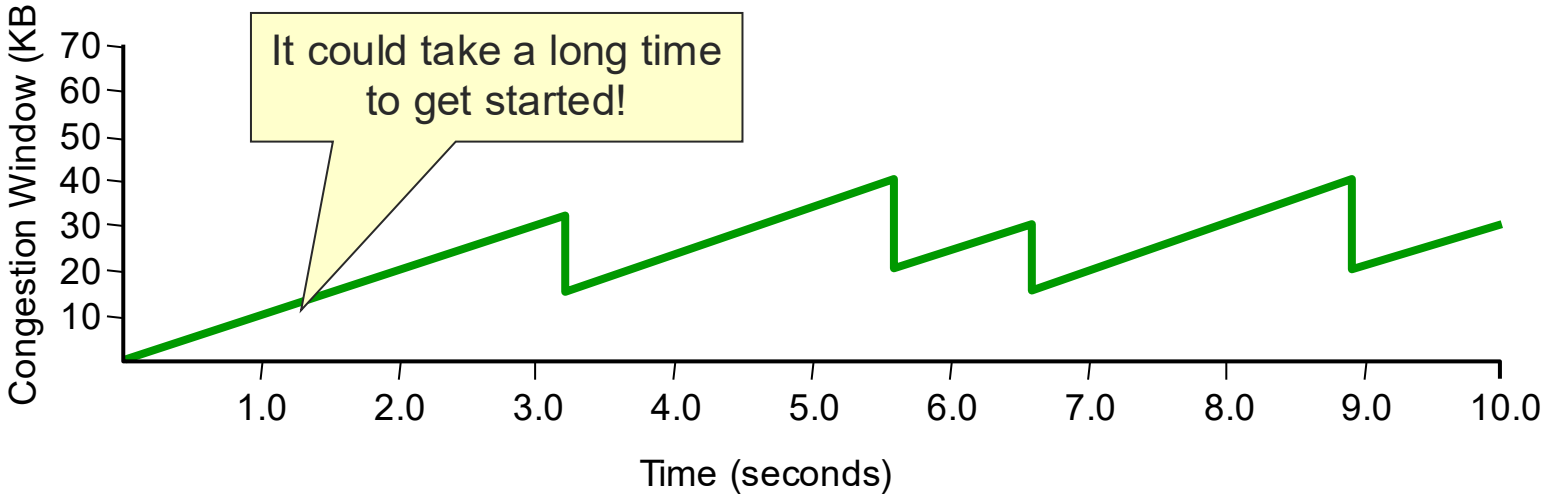
Additive Increase, Multiplicative Decrease (AIMD)



Fair and efficient equilibrium!

TCP Start Up Behavior

- How should TCP start sending data?
  - AIMD is good for channels operating at capacity
  - AIMD can take a long time to ramp up to full capacity from scratch



TCP Start Up Behavior

- How should TCP start sending data?
  - AIMD is good for channels operating at capacity
  - AIMD can take a long time to ramp up to full capacity from scratch
  - *TCP Slow Start*
    - Begins with a small window size (start slowly)
      - Slow compared with sending a whole window immediately in original TCP
    - Increases the window exponentially (fast!)

TCP Slow Start: Initialization of Congestion Window

- Congestion window should start small
  - Avoid congestion due to new connections
- Start at 1 MSS,
  - Initially, CWND is 1 MSS
  - Initial sending rate is MSS/RTT
- Reset to 1 MSS with each timeout

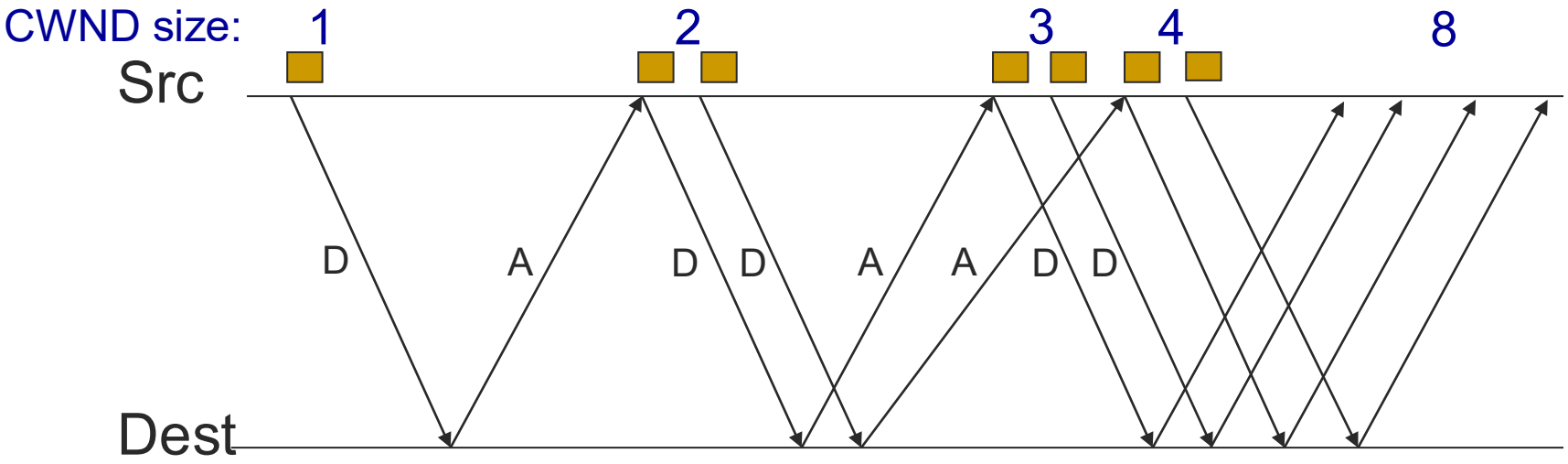
TCP Slow Start: Growth of Congestion Window

- Start slow but then grow fast
  - Sender starts at a slow rate
  - Increase the rate exponentially
  - Until the first loss event
- Linear growth could be pretty wasteful
  - Might be much less than the actual bandwidth
  - Linear increase takes a long time to accelerate

Slow Start: Summary

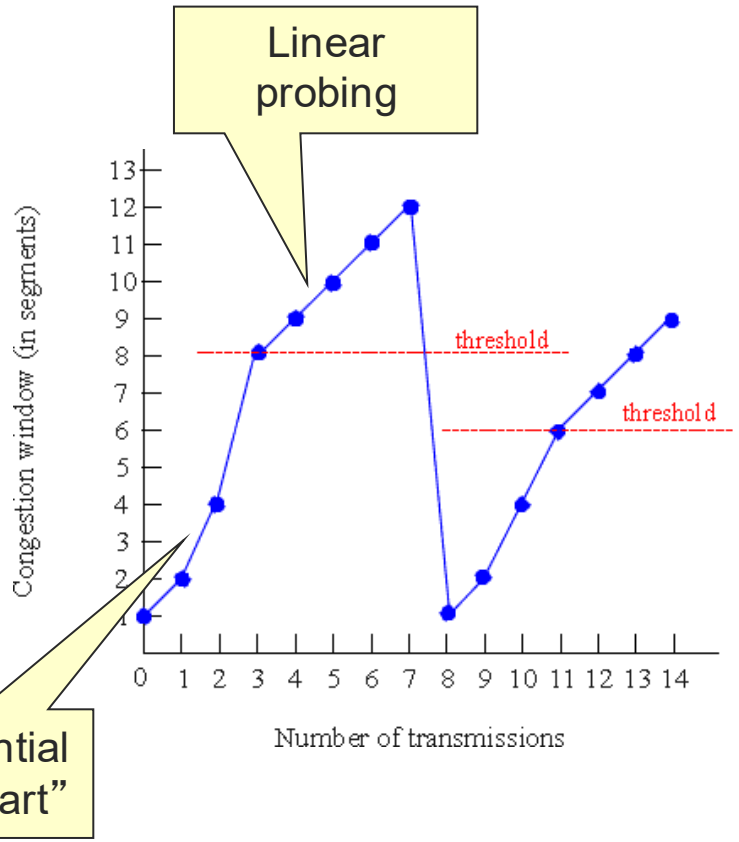
- Objective
  - Determine initial available capacity
- Implementation
  - Begin with **`CongestionWindow`** = 1 packet
  - Double **`CongestionWindow`** each RTT
    - Increment by 1 packet for each ACK
  - Continue increasing until timeout (loss)
    - Reset to 1 MSS and return to slow start

Slow Start Example



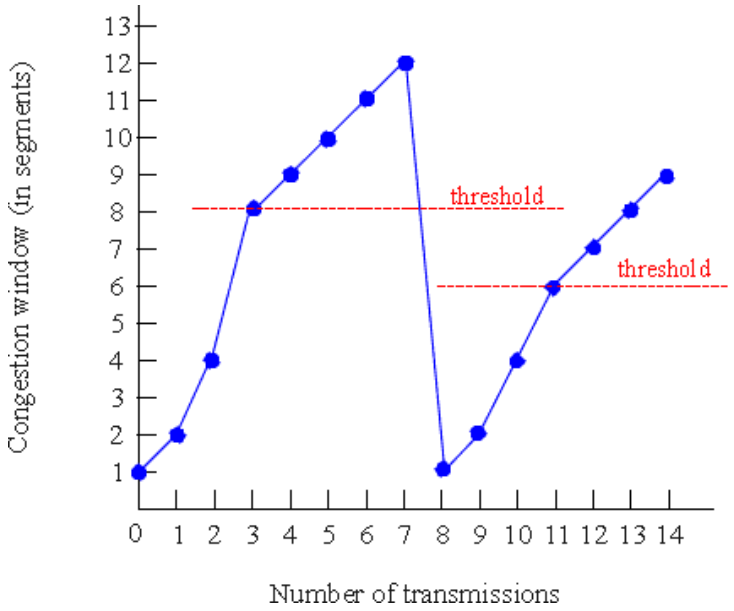
From Slow Start to Congestion Avoidance

- How long should the exponential increase from slow start continue?
  - Use **`CongestionThreshold`** as target window size
  - Estimates network capacity
  - When **`CongestionWindow`** reaches **`CongestionThreshold`** switch to additive increase
  - Set to 1/2 of current window on timeout
  - Initially set to a large value



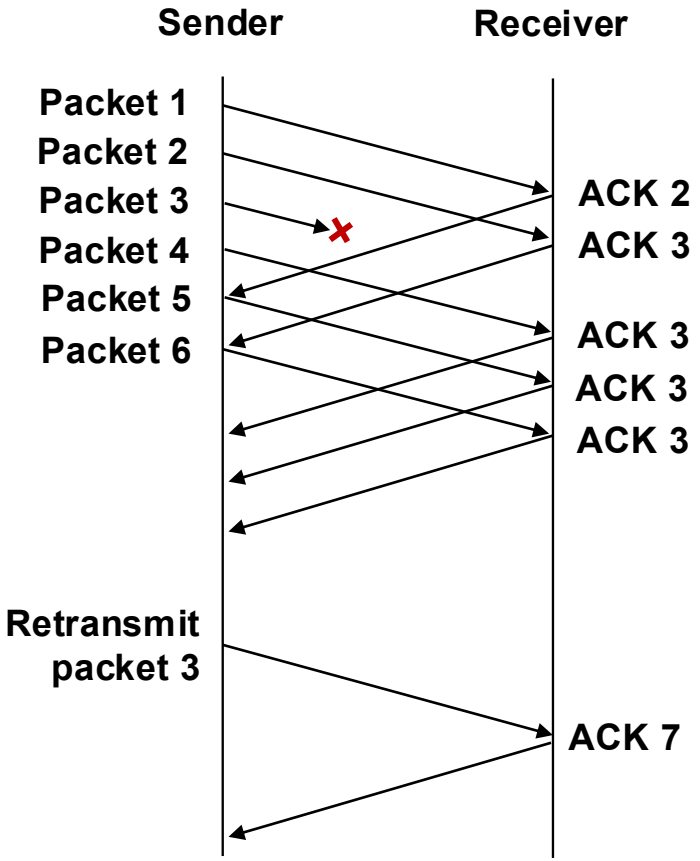
From Slow Start to Congestion Avoidance

- Example: initially,
  - **`CongestionThreshold = 8`**
  - **`CongestionWindow = 1`**
- Slow start until transmission 3
- Additive increase until transmission 7
- Timeout after transmission 7
  - **`CongestionWindow`** currently 12
  - Set **`Congestionthreshold = CongestionWindow/2`**
  - Set **`CongestionWindow = 1`**



Retransmission Revisited

- Problem
  - TCP timeouts lead to long idle periods
- Fast retransmit
  - Use duplicate ACKs to trigger retransmission
- When duplicate ACKs received
  - Resend lost segment immediately
  - Do not wait for timeout
  - In practice, retransmit on 3rd duplicate (triple dup ACK)



Fast Retransmit and Fast Recovery

- Fast recovery
  - When fast retransmission occurs, skip slow start
  - Congestion window becomes 1/2 previous
    - Restoring AIMD
  - Start additive increase immediately

TCP Congestion Window Dynamics Summary