Lecture 8

Congestion Control

Start Network Layer

TCP Summary (so far)

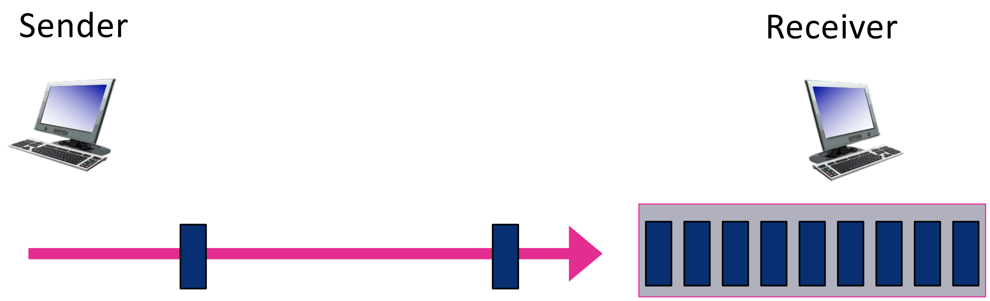
- 3-Way Handshake – Connection Oriented
- Protocol builds reliability on an unreliable network
- Mix of Go-Back-N and Selective Repeat
  - Cumulative acks
  - Timeouts
  - Store un-acknowledged packets at sender for selective retransmission
- Uses “bytes” instead of segments for the sequence numbers
- Both sender and receiver keep windows
  - What is “N”?

TCP Flow Control

- **Flow Control** key idea: sender should not transmit faster than the receiver can process!

Recall: Flow Control

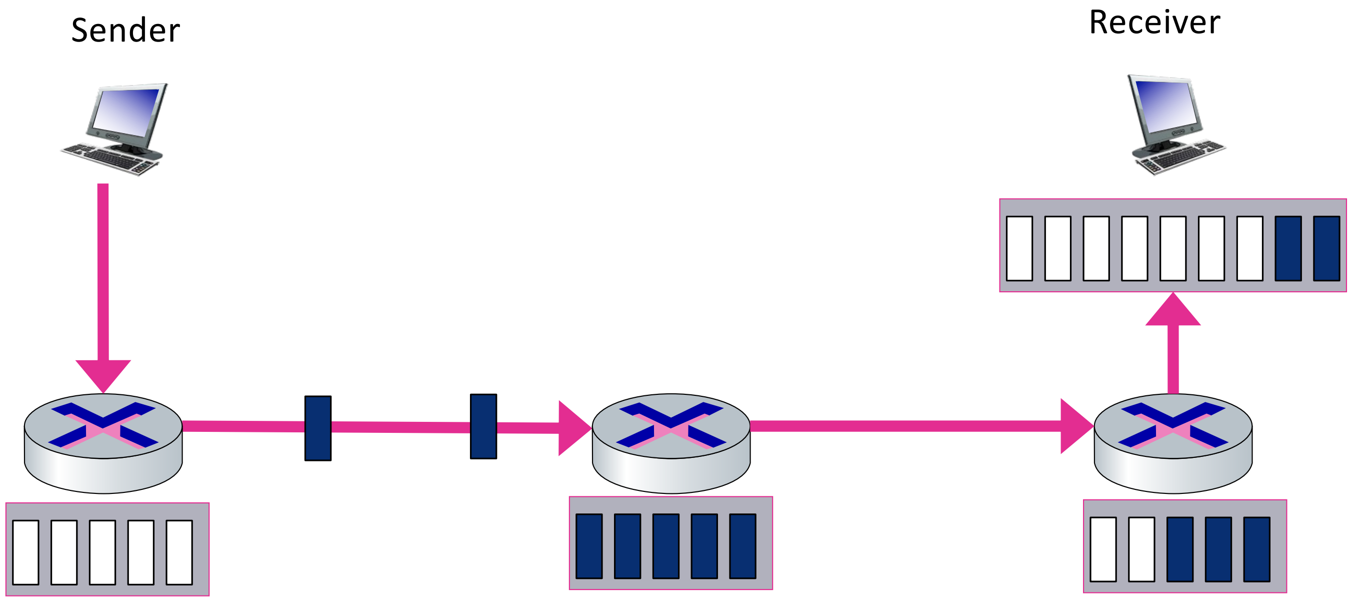
Problem:
- New data arriving at a receiver with a full buffer will be dropped
- To provide reliable transfer, it will need to be retransmitted…
- Sender needs to do a lot of extra work, and we waste bandwidth transmitting the same data multiple times



Solution:
- Flow control!
- Receiver tells sender how much new data it can accept, and sender agrees not to send more than that
- Results in “speed matching” of sending and processing/delivering data

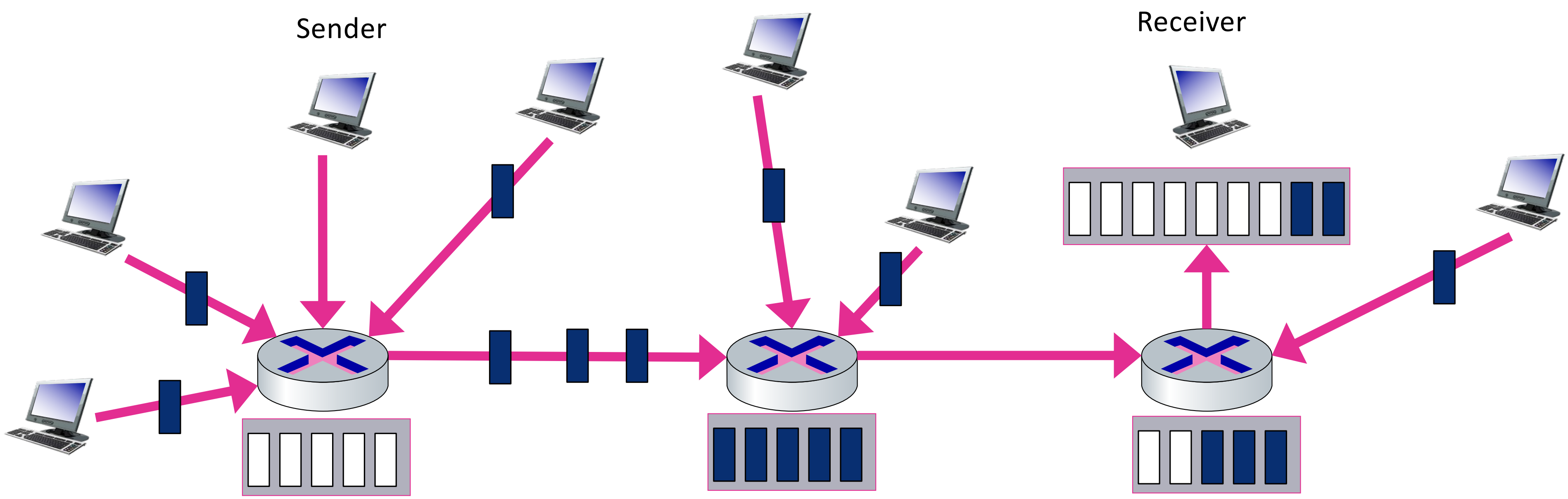
Congestion Control

- But, buffers at the receiver aren’t the only ones we need to worry about…



Congestion Control

- …and many senders may “compete” for the same network resources



Congestion Control vs Flow Control

- **Both** try to **limit sending rate** to avoid overwhelming finite resources

- **Flow control** aims to avoid overwhelming the **receiver**
- **Congestion control** aims to avoid overwhelming the **network**

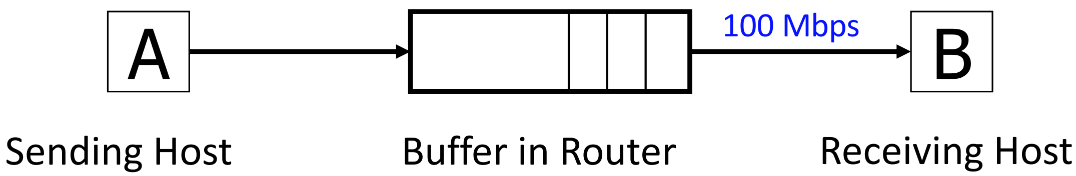
- Flow control benefits an individual source->destination flow
- Congestion control is (mainly) for the general benefit of the network

Congestion Control

- **Goal**: allow senders to transmit as fast as possible **without** overloading the network

Congestion Control Challenges

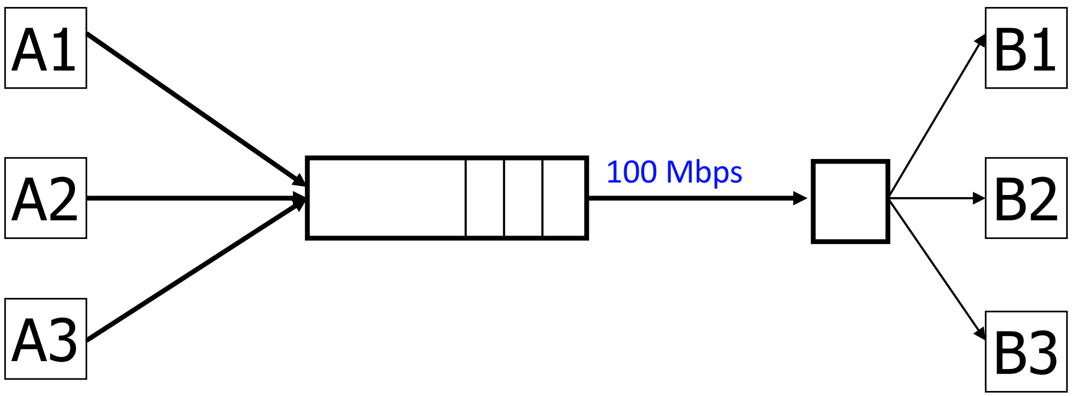
Consider an extremely simple scenario:



- Can we experience congestion in this case?
  - Yes, if A sends at a rate > 100 Mbps
- How can we avoid congestion in this case?
  - A needs to send at < 100 Mbps…but how does it know what the available transmission rate is??

Congestion Control Challenges

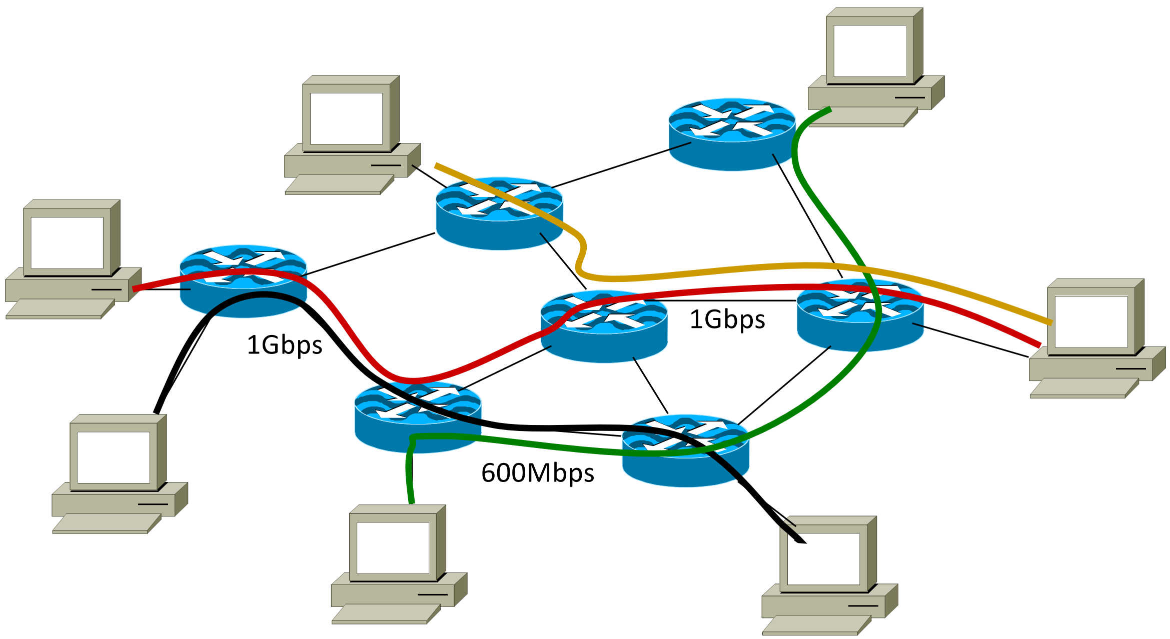
Consider a slightly more complex (but still simple) scenario:



- How can we avoid congestion?
  - A1, A2, A3 need to **collectively** send at a rate of at most 100 Mbps
  - But none of them knows that the bottleneck link is 100 Mbps, how many other senders there are, or how fast the others want to send…

Congestion Control Challenges

And reality looks more like this:



Congestion control is a resource allocation problem involving many flows, many links, and complicated global dynamics

What can we do?

(0) **Give up** – let everyone send as fast as they want

- Many packet drops, delays, retransmissions, potential for *congestive collapse*

What can we do?

(0) **Give up** – let everyone send as fast as they want

(1) **Require reservations (virtual circuits)**

- Pre-arrange bandwidth allocations
- Requires negotiation before sending packets
- Low utilization

What can we do?

(0) **Give up** – let everyone send as fast as they want

(1) **Require reservations (virtual circuits)**

(2) **Charge more money**

- Don’t drop packets for the high-bidders
- Requires payment model

What can we do?

(0) **Give up** – let everyone send as fast as they want

(1) **Require reservations (virtual circuits)**

(2) **Charge more money**

(3) **Routers talk with end hosts**

- Layering issues
- More overhead

What can we do?

(0) **Give up** – let everyone send as fast as they want

(1) **Require reservations (virtual circuits)**

(2) **Charge more money**

(3) **Routers talk with end hosts**

(4) **Dynamically adjust sending rates**

- Hosts infer level of congestion; adjust rates
- Simple to implement but suboptimal, messy dynamics

What can we do?

(0) **Give up** – let everyone send as fast as they want

(1) **Require reservations (virtual circuits)**

(2) **Charge more money**

(3) **Routers talk with end hosts**

(4) **Dynamically adjust sending rates**

- Hosts infer level of congestion; adjust rates
- Simple to implement but suboptimal, messy dynamics
- But, provides a very general solution: doesn’t assume business model, traffic characteristics, application requirements; maintains layered model
- But, requires good citizenship!

What can we do?

(0) **Give up** – let everyone send as fast as they war

(1) **Require reservations (virtual circuits)**

(2) **Charge more money**

(3) **Routers talk with end hosts**

(4) **Dynamically adjust sending rates**

- Hosts infer level of congestion; adjust rates
- Simple to implement but suboptimal, messy dynamic
- But, provides a very general solution: doesn't assume characteristics, application requirements
- But, requires good citizenship!

Aside: congestion arises from **competition for limited resources**

An alternative is **overprovisioning** such that congestion becomes extremely unlikely.

What is the drawback/tradeoff?

TCP Congestion Control: Dynamic Adaptation

- **Two key questions:**
  - How do senders **infer** that there is congestion?
  - How do senders **adjust** their sending rates in response?

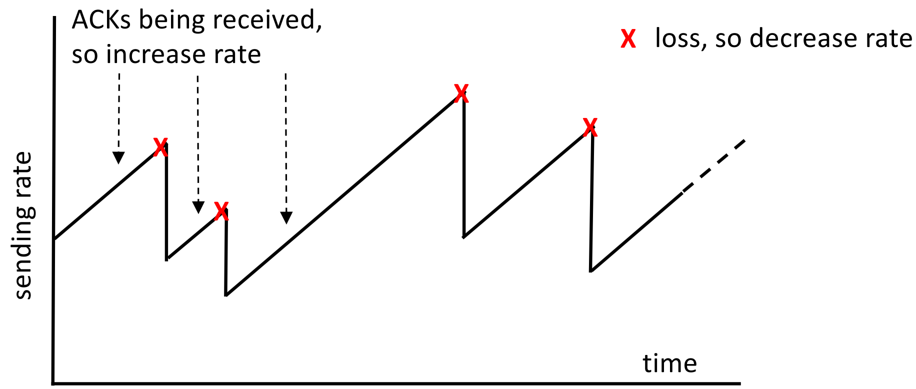
- **High-level answers:**
  - Assume **loss** implies congestion
  - Maintain a **window** that shrinks whenever congestion is detected (and increases when no congestion is detected)

Congestion Window: Controlling the Send Rate

- **Congestion Window**: CWND
  - Bytes that can be sent without overflowing routers
  - *Computed by sender using congestion control algorithm*
- (Recall) **Flow control window**: RWND
  - Bytes that can be sent without overflowing receiver
  - Determined by the receiver and reported to the sender
- **Sender-side window** = min {CWND, RWND}
  - Assume for this lecture that RWND >> CWND
  - Recall from flow control discussion: sending rate ~ Window/RTT

Adjusting the Window

- Basic idea:
  - **Probing for available bandwidth**
  - ACK: segment received → network is not congested → increase sending rate (increase window size)
  - Lost segment: (timeout or 3 duplicate ACKs) → network is congested → decrease sending rate (decrease window size)



TCP's "sawtooth" behavior

Adjusting the Window

- Note: Not all losses are the same
- **Duplicate ACKs**: isolated loss
  - Still getting ACKs
- **Timeout**: much more serious
  - Not enough duplicate acks
  - Must have suffered several losses
- Will adjust rate differently for each case

Adjusting the Window

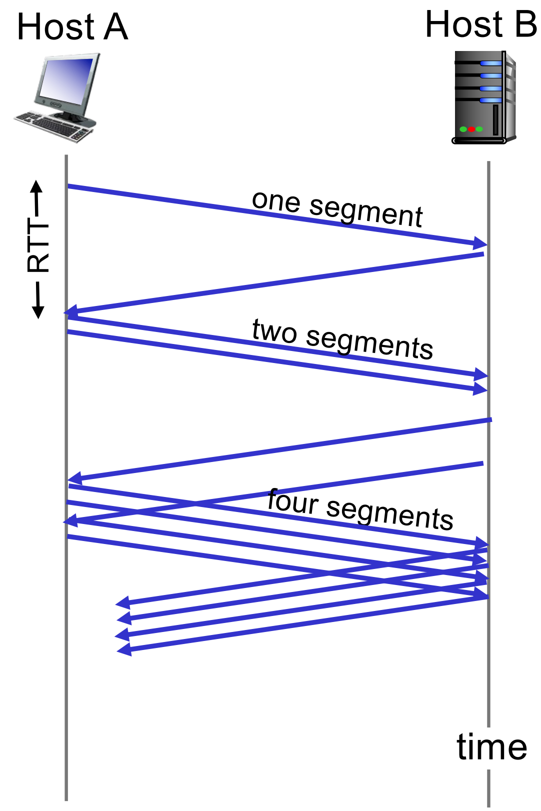
- High level approach:
  - TCP **probes** for the available transmission rate by increasing its window
    - Eventually, we'll set it too high, and congestion/loss will occur
  - Find out **approximately** the available transmission rate as fast as possible (discovery / *slow start* phase)
  - Then, once we hit loss, cut `cwnd` and probe more carefully (*congestion avoidance* phase)

Bandwidth discovery: “Slow Start” Phase

- **Goal: estimate available transmission rate**
  - Start slow (for safety)
  - Ramp up quickly (for efficiency)
- **Consider**
  - RTT = 100ms, MSS=1000bytes
  - Window size to “fill” 1Mbps of Transmission Rate = 12.5 segments
    - (10^6 bits/sec) / (8000 bits/packet) * (0.1sec) = 12.5 segments
  - Window size to “fill” 1Gbps = 12,500 segments
  - Either is possible!

Slow Start

- **Start with a small congestion window**
  - Initially, **CWND = 1 MSS**
  - So, initial sending rate is MSS/RTT
- **But ramp up fast (exponentially)**
  - **Double** the CWND for each RTT with no loss
  - **How? Increase by 1 MSS for each ACK**

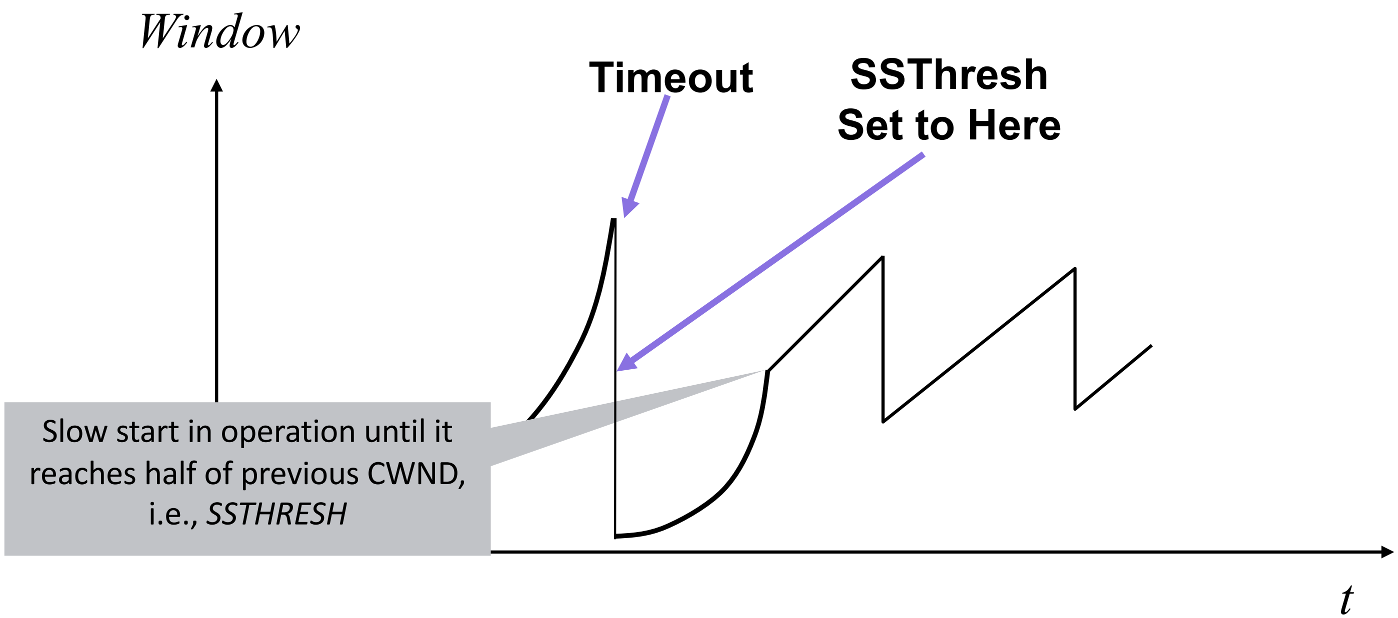


Slow Start

- **When should we stop exponential increase of the window size?**
  - When we think our estimate of the available bandwidth is *close* to reality
- Mechanism: **Slow start threshold** (ssthresh)
  - When loss is detected, set ssthresh to half the size of the current congestion window, then restart growth
- **Slow start ends** when cwnd >= ssthresh

- On timeout,
  - Set ssthresh = cwnd / 2
  - Set cwnd = 1 MSS
  - Restart slow start (exponential growth)

Example



Slow-start restart: Go back to CWND = 1 MSS, but take advantage of knowing the previous value of CWND

Bandwidth Adaptation: Congestion Avoidance

- When slow start ends, move to **Congestion Avoidance** phase
- Stop rapid growth and focus on maintenance
- Now, want to **track variations in this available bandwidth, oscillating around its current value**
  - Repeated probing (rate increase) and backoff (decrease)
- TCP uses: "**Additive Increase Multiplicative Decrease**" (AIMD)

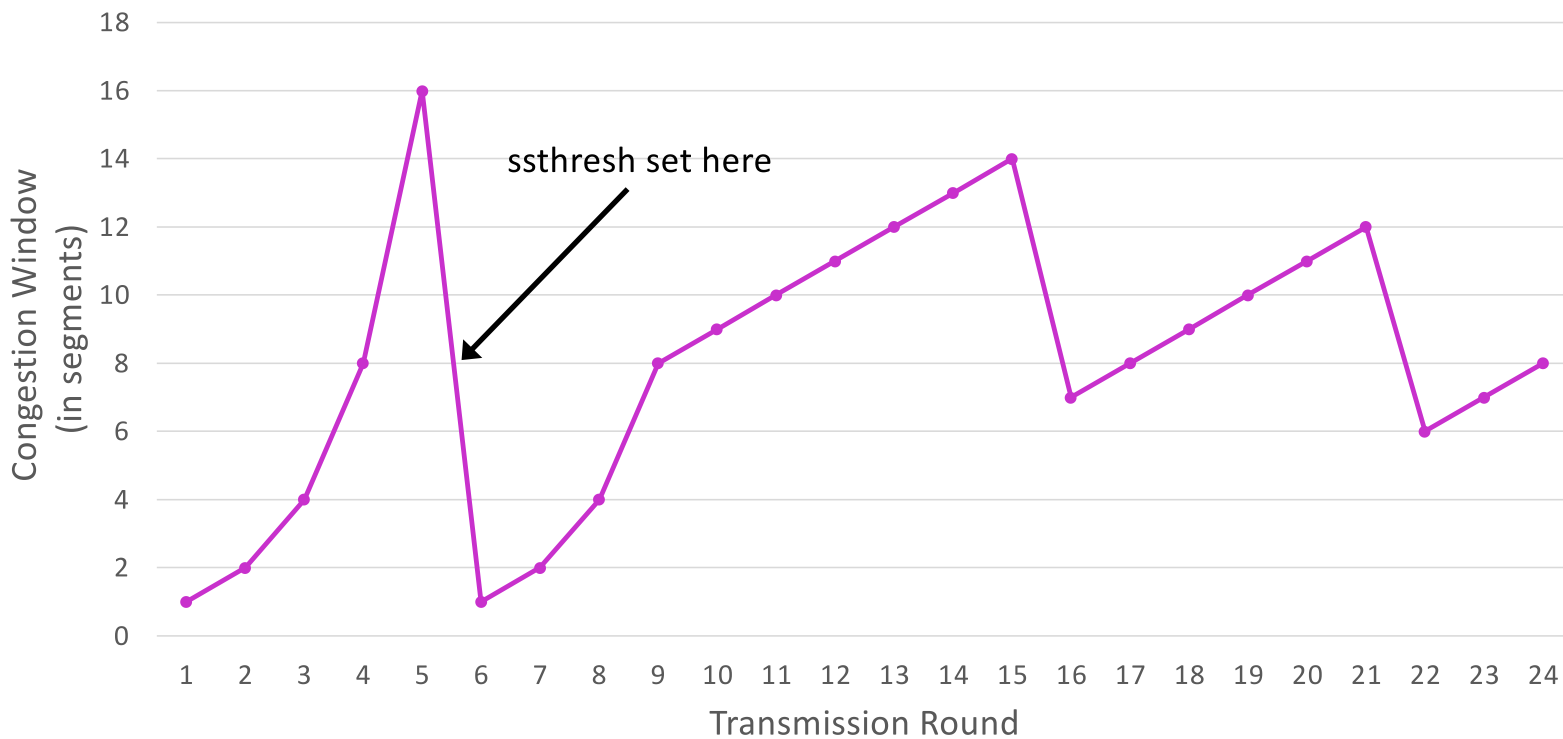
Congestion Avoidance

e.g. let MSS = 1460, CWND = 10 segments = 14600 bytes

MSS * MSS/CWND
= 1460 * (1460/14600) = 1460 * 1/10 = 146

- **Additive increase**
  - Increase CWND by 1 MSS per RTT
    - For each ACK, CWND += MSS * MSS/CWND
- **Multiplicative decrease**
  - On triple duplicate ACK, cut CWND in half (and cut ssthresh in half) and move to fast recovery
    - ssthresh = CWND/2
    - CWND = CWND/2 + 3 MSS
    - Transition to Fast Recovery state
  - On timeout, cut ssthresh in half and restart slow start
    - ssthresh = CWND/2
    - CWND = 1 MSS
    - Initiate Slow Start

Example (with numbers)



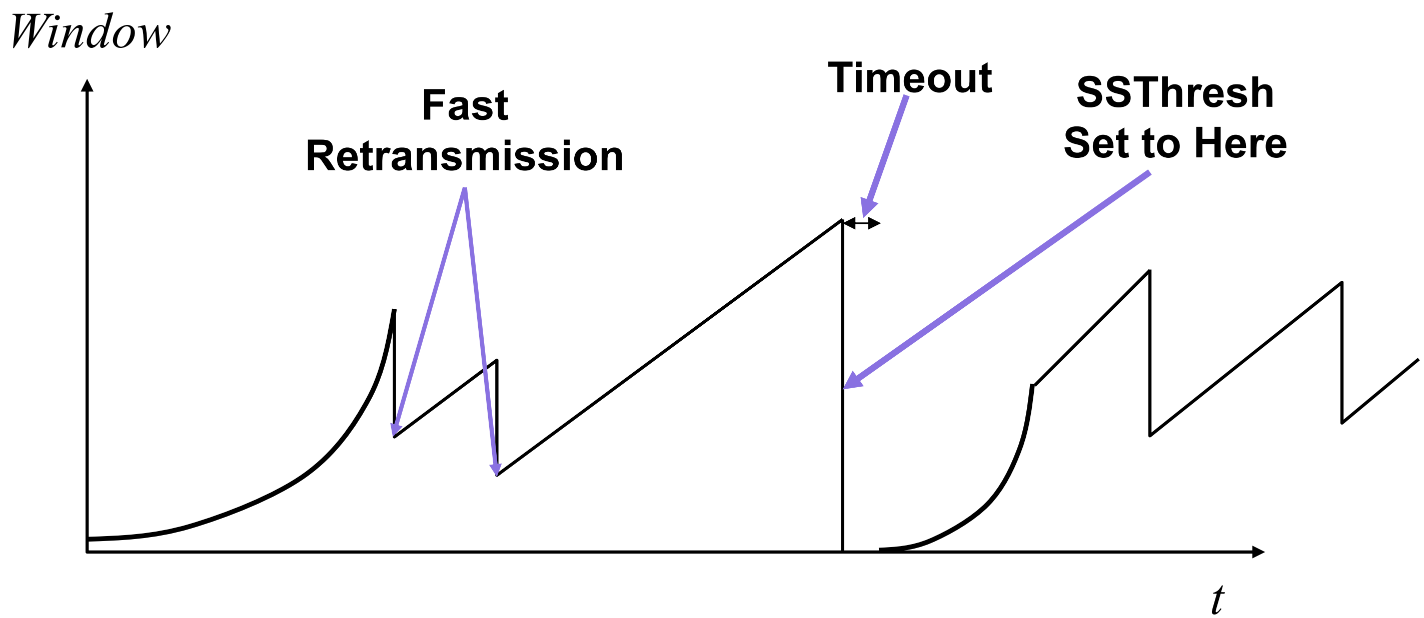
Adjusting the Window: Overview

- Increasing cwnd
  - Slow start: increase **exponentially** (double each RTT)
    - Beginning of connection or after timeout
    - Start from very small initial window and ramp up fast
  - Congestion Avoidance: increase **linearly** (1 MSS per RTT)
    - Normal case: already have a "pretty good" view of available bandwidth, cautiously probing for more
- Reducing cwnd
  - Timeout: cut cwnd to **1 MSS**
  - 3 duplicate ACKs: cut cwnd in **half** (introduced in TCP Reno)

Additive Increase

Multiplicative Decrease

Example



TCP Fast Recovery (Completing TCP Reno Specification)

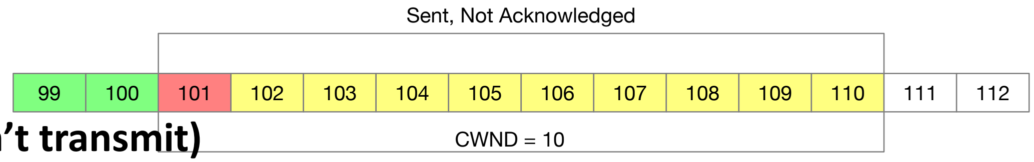
- Idea: grant temporary “credit” for duplicate ACKs to make recovery from isolated loss faster
- Why?

Why Fast Recovery?: Example

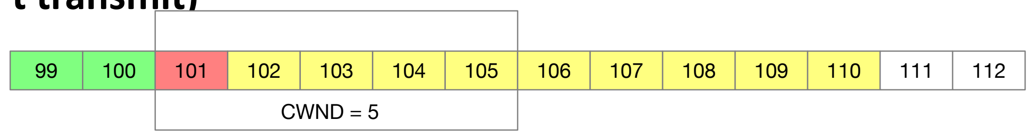
- **Consider a TCP connection with:**
  - CWND=10 packets
  - Last ACK was for packet # 101
    - i.e., receiver expecting next packet to have seq. no. 101
- **10 packets [101, 102, 103,…, 110] are in flight**
  - Packet 101 is dropped

Why Fast Recovery?: Example

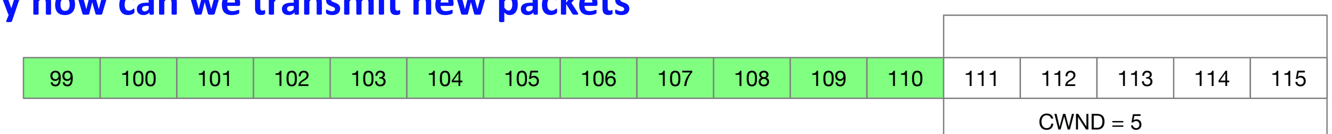
- Packets 101-110 in flight, Packet 101 is LOST



- ACK 101 (due to 102) cwnd=10 dupACK#1 (can't transmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (can't transmit)
- ACK 101 (due to 104) cwnd=10 dupACK#3 (can't transmit)
- RETRANSMIT 101 ssthresh=5 cwnd= 5



- ACK 101 (due to 105) cwnd=5 (can't transmit)
- ACK 101 (due to 106) cwnd=5 (can't transmit)
- ACK 101 (due to 107) cwnd=5 (can't transmit)
- ACK 101 (due to 108) cwnd=5 (can't transmit)
- ACK 101 (due to 109) cwnd=5 (can't transmit)
- ACK 101 (due to 110) cwnd= 5 (can't transmit)
- ACK 111 (due to 101) 🡸 only now can we transmit new packets



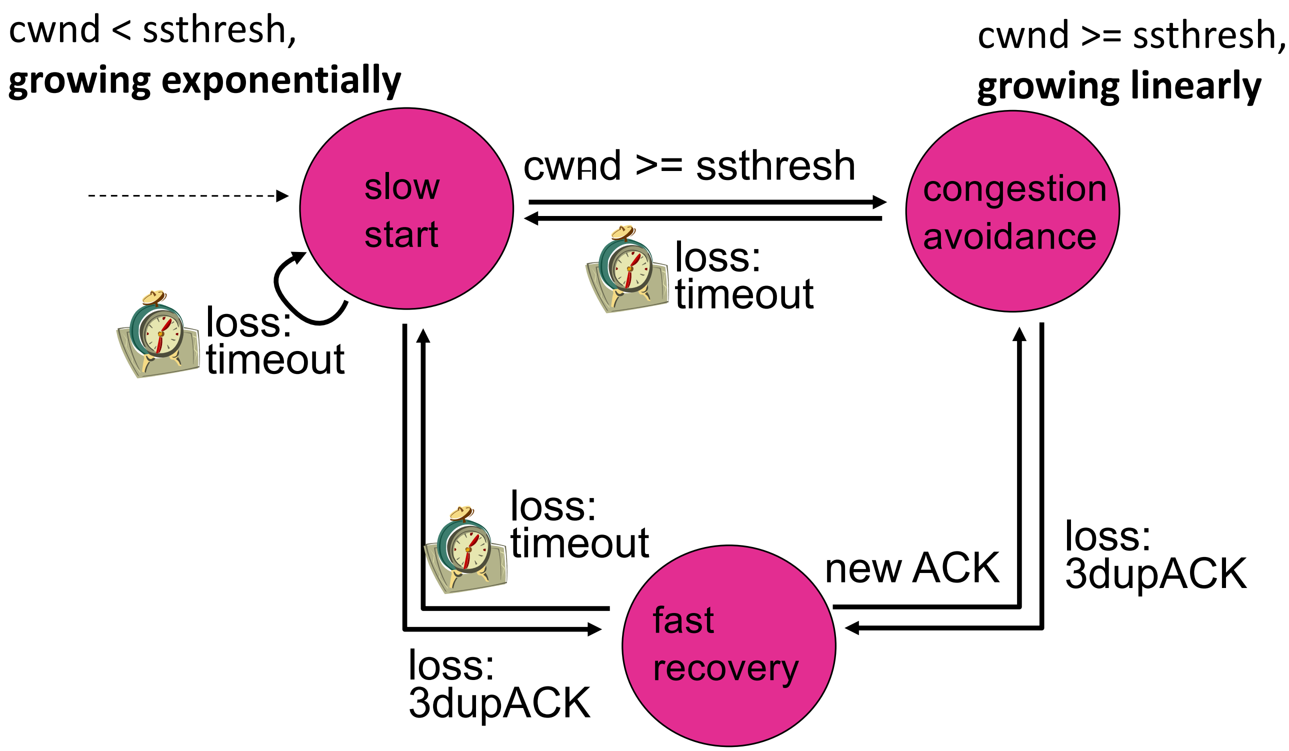
Fast Recovery (Completing TCP Reno Specification)

- **Idea: Grant the sender temporary “credit” for each dupACK so as to keep packets in flight**
- **If dupACKcount == 3**
  - ssthresh = CWND/2
  - CWND = ssthresh + 3 MSS
- While in fast recovery
  - CWND += 1 MSS for each additional dupACK
- Exit fast recovery after receiving new ACK
  - set CWND = ssthresh

Fast Recovery: Example

- Packets 101-110 in flight, Packet 101 is LOST
- ACK 101 (due to 102) cwnd=10 dupACK#1 (can't transmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (can't transmit)
- ACK 101 (due to 104) cwnd=10 dupACK#3 (can't transmit)
- RETRANSMIT 101 ssthresh=5 cwnd= 8 (5+3)
- ACK 101 (due to 105) cwnd=9 (can't transmit)
- ACK 101 (due to 106) cwnd=10 (can't transmit)
- ACK 101 (due to 107) cwnd=11 (transmit 111)
- ACK 101 (due to 108) cwnd=12 (transmit 112)
- ACK 101 (due to 109) cwnd=13 (transmit 113)
- ACK 101 (due to 110) cwnd= 14 (transmit 114)
- ACK 111 (due to 101) cwnd = 5 (transmit 115) ← exiting fast recovery (deflate window)
- Packets 111-114 already in flight
- ACK 112 (due to 111) cwnd= 5 + 1/5 <- back in congestion avoidance

TCP Congestion Control: Overview



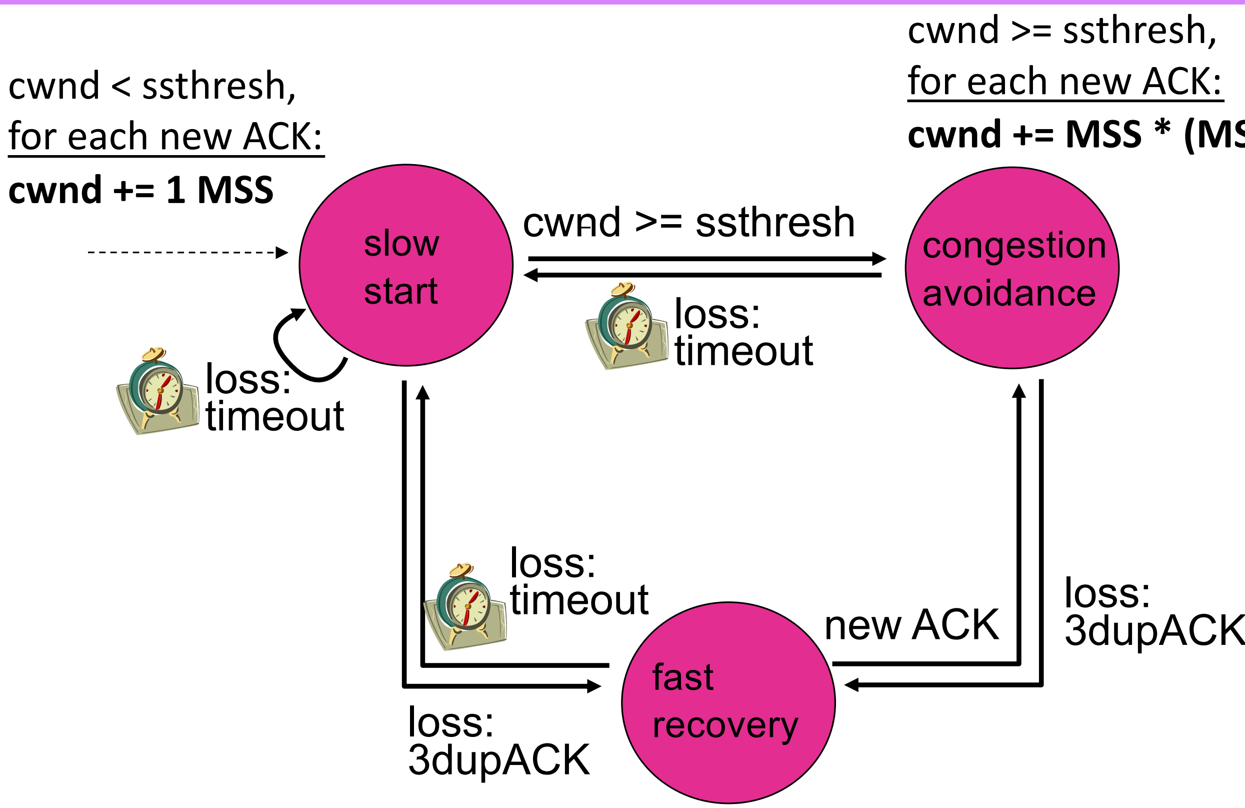
Timeout -> serious loss: go back to **slow start** to re-do bandwidth estimation

- ssthresh = cwnd/2
- cwnd = 1 MSS

3dupACK -> isolated loss, try to fine-tune estimate

- ssthresh = cwnd/2
- cwnd = cwnd/2 + 3 MSS

TCP Congestion Control: Overview



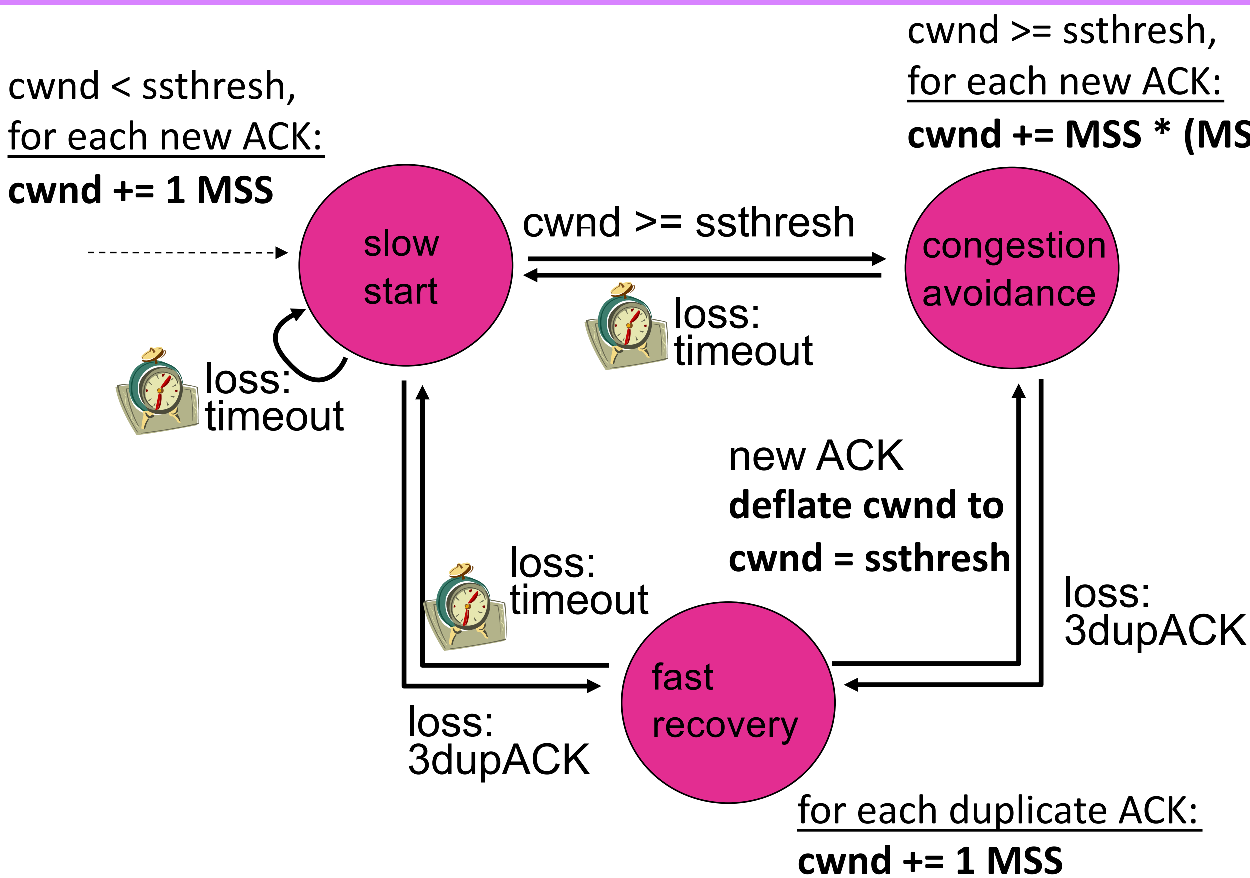
Timeout -> serious loss: go back to **slow start** to re-do bandwidth estimation

- ssthresh = cwnd/2
- cwnd = 1 MSS

3dupACK -> isolated loss, try to fine-tune estimate

- ssthresh = cwnd/2
- cwnd = cwnd/2 + 3 MSS

TCP Congestion Control: Overview



Timeout -> serious loss: go back to **slow start** to re-do bandwidth estimation

- ssthresh = cwnd/2
- cwnd = 1 MSS

3dupACK -> isolated loss, try to fine-tune estimate

- ssthresh = cwnd/2
- cwnd = cwnd/2 + 3 MSS

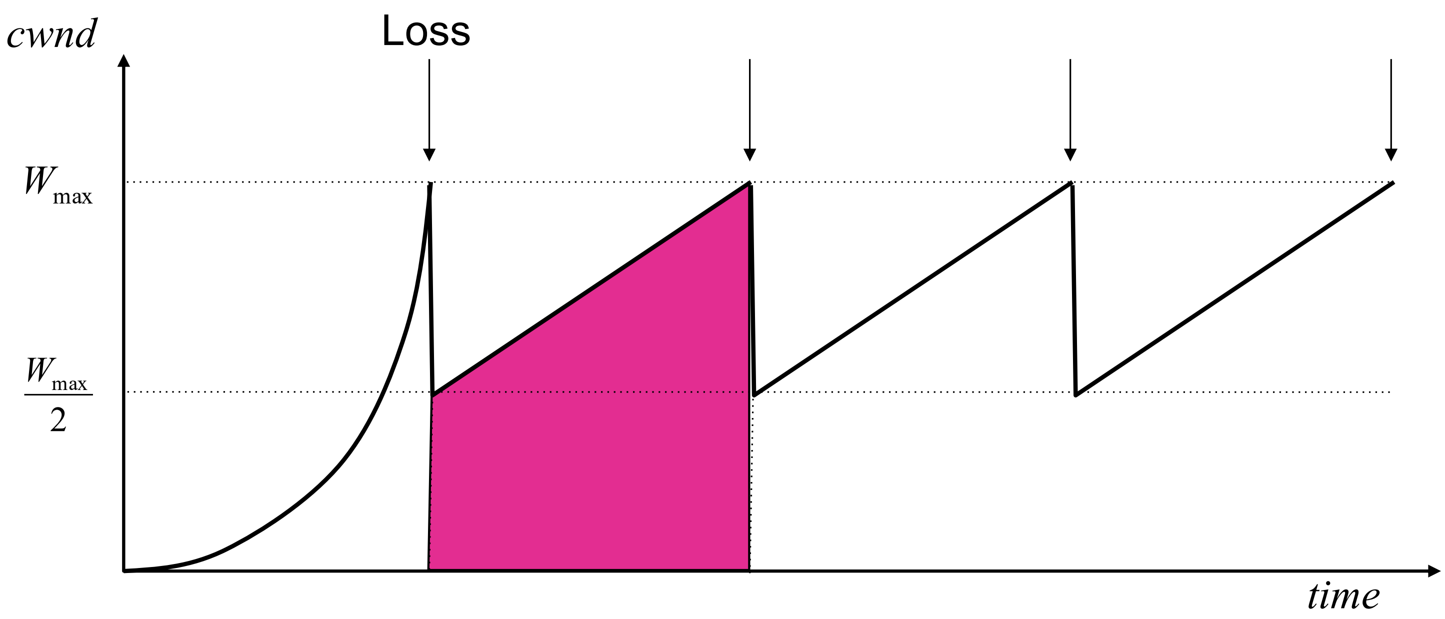
TCP Throughput: A Simple Model

- If our congestion window at some point in time is *W* bytes, then we are allowed to **send *W* bytes per RTT**
- So, max throughput at that time is: $\frac{W}{RTT}$

- Example: Window of 10 segments, each segment is 1500bytes, RTT is 20ms
  - Throughput = (10 * 1500 * 8) / .02 = 6 Mbps
  - If we want 10 Mbps throughput:
    - 10Mbps = (X * 1500 *8) / .02
    - (10*10^6 bits/sec * 0.02s) / (1500 bytes * 8 bits) = 16.667 segments ~= 17 segments

TCP Throughput: A Simple Model

- Ignore Slow Start, assume we are always in Congestion Avoidance

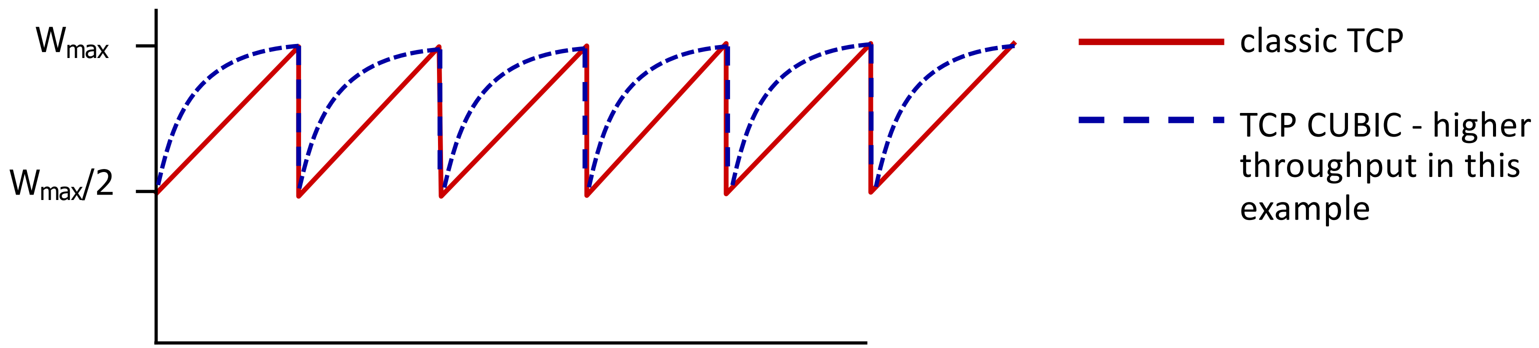


Window **increases linearly** from $\frac{W_{max}}{2}$ to W_{max} until loss occurs and process repeats

Average throughput:
$$\frac{0.75\ W_{max}}{RTT}$$

TCP CUBIC

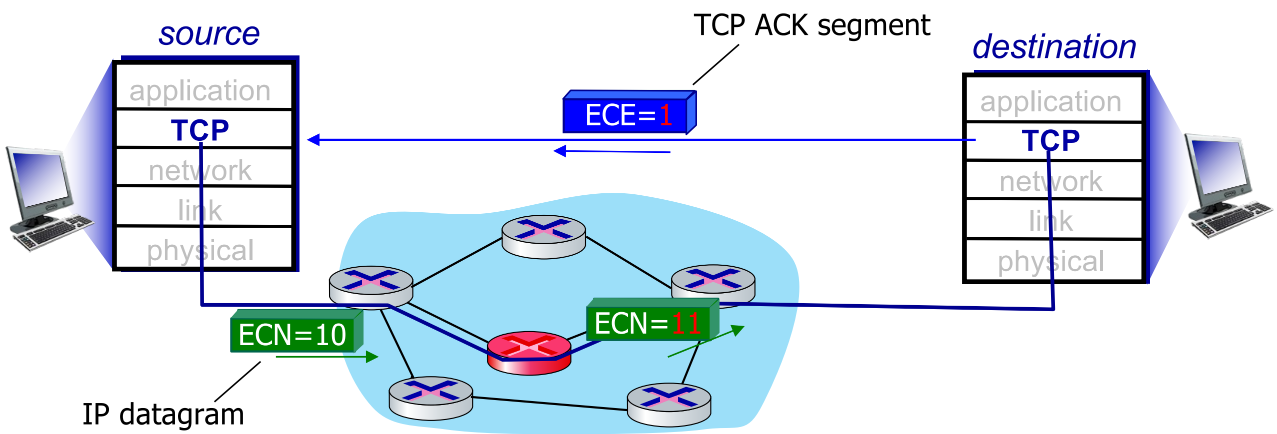
- Is there a better way than AIMD to "probe" for usable bandwidth?
- Insight/intuition:
  - W_{max}: sending rate at which congestion loss was detected
  - congestion state of bottleneck link probably (?) hasn't changed much
  - after cutting rate/window in half on loss, initially ramp to to W_{max} *faster*, but then approach W_{max} more *slowly*



Explicit congestion notification (ECN)

TCP deployments often implement *network-assisted* congestion control:

- two bits in IP header (ToS field) marked *by network router* to indicate congestion
  - *policy* to determine marking chosen by network operator
- congestion indication carried to destination
- destination sets ECE bit on ACK segment to notify sender of congestion
- involves both IP (IP header ECN bit marking) and TCP (TCP header C,E bit marking)

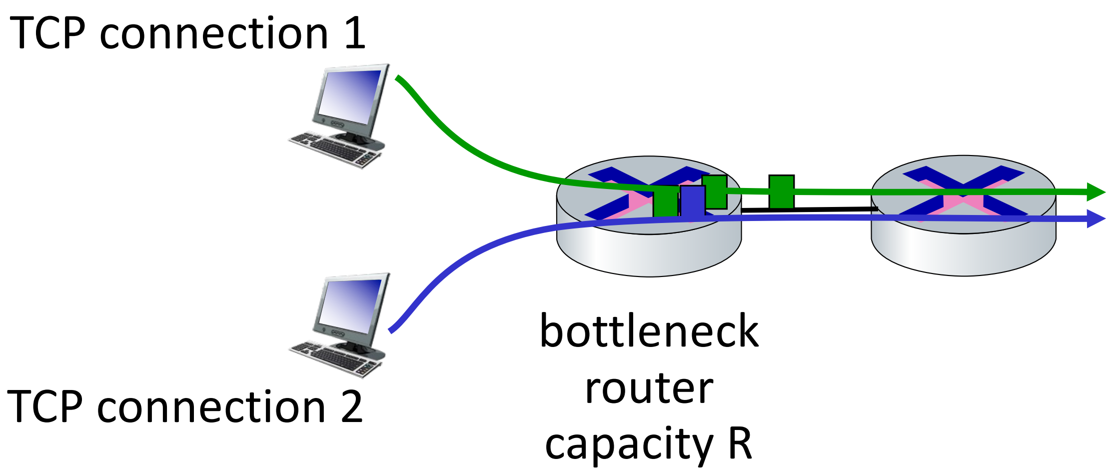


Fairness

- Mac OS Dictionary App
  - **Definition:** impartial and just treatment or behavior without favoritism or discrimination

TCP Fairness

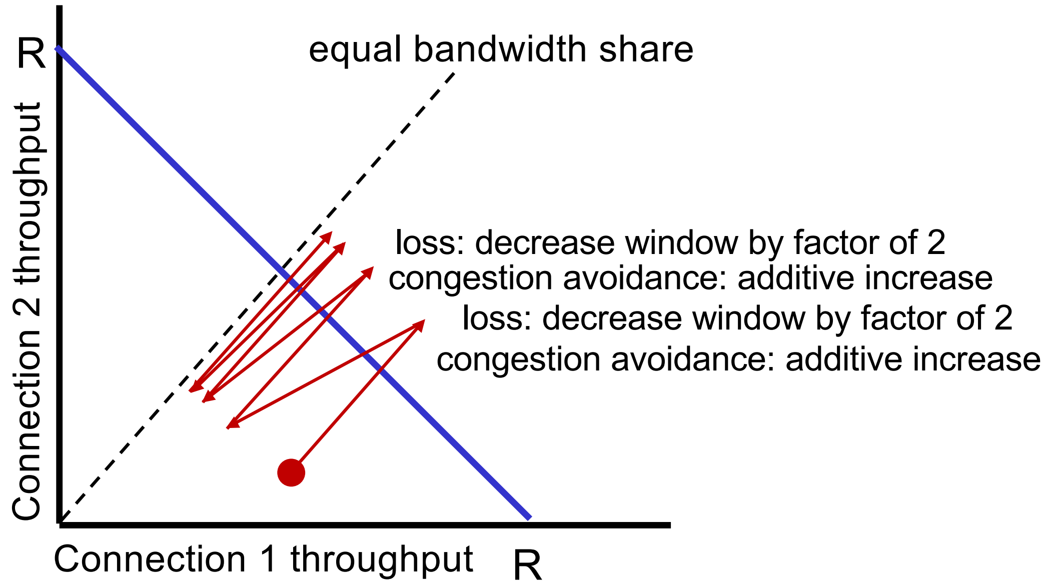
Fairness goal: if *K* TCP sessions share same bottleneck link of bandwidth *R*, each should have average rate of *R/K*



Is TCP Fair?

Example: two competing TCP sessions:

- additive increase gives slope of 1, as throughout increases
- multiplicative decrease decreases throughput proportionally



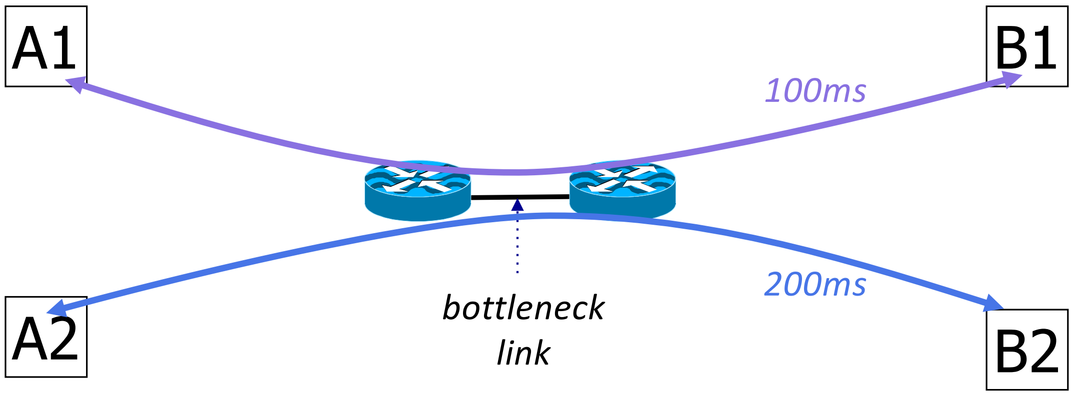
Is TCP fair?

A: Yes, under idealized assumptions:

- same RTT
- single connection per app
- only TCP traffic

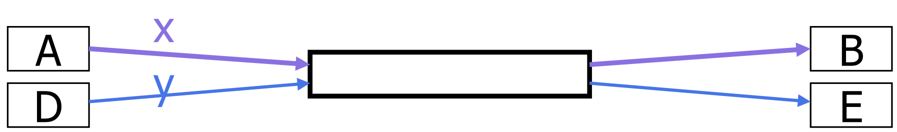
Unfairness and TCP

- TCP throughput depends on RTT
- Flows with shorter RTT can increase cwnd faster than flows with long RTT (feedback arrives faster)
- TCP unfair in the face of heterogeneous RTTs



Unfairness and TCP

- Applications (e.g., Web) can open many parallel TCP connections to transfer data
- Unfairness on an application basis
  - Using more TCP connections an application can obtain a higher aggregate throughput



- Assume
  - A starts 10 connections to B
  - D starts 1 connection to E
  - Each connection gets about the same throughput
- Then A gets 10 times more throughput than D

Unfairness and UDP

- UDP does not perform congestion control
- It’s possible for UDP traffic to increase congestion and crowd out TCP traffic

- No requirement for network applications to employ congestion control

Transport Layer Evolution

Bottleneck Bandwidth and Round-trip propagation time

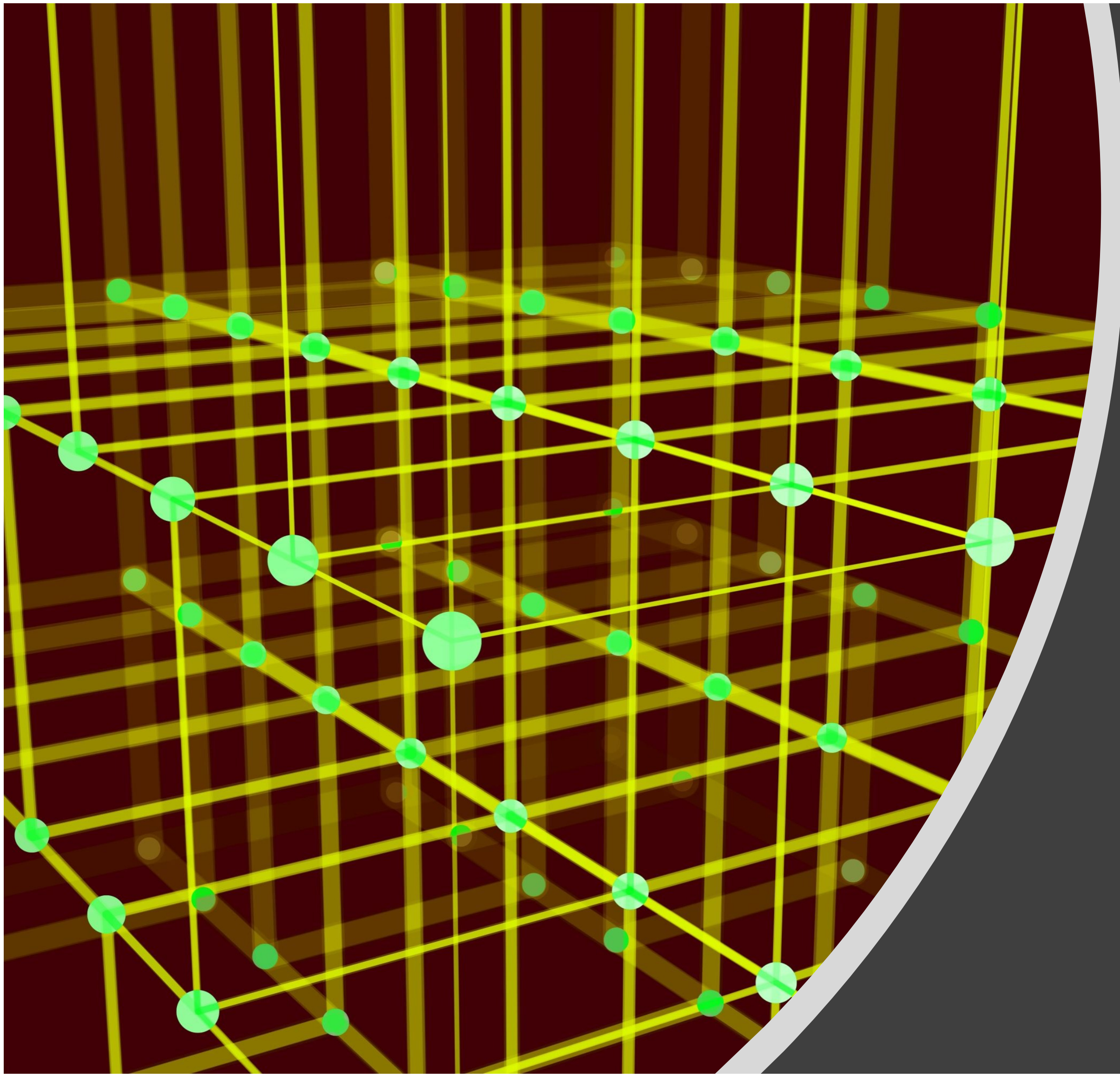
- TCP, UDP: principal transport protocols for 40 years
- *Many* different variants of TCP developed…TCP CUBIC now default on most OS (vs Reno), BBR used by Google
- and more for specific scenarios:

Scenario	Challenges
Long, fat pipes (large data transfers)	Many packets “in flight”; loss shuts down pipeline
Wireless networks	Loss due to noisy wireless links, mobility; TCP treat this as congestion loss
Long-delay links	Extremely long RTTs
Data center networks	Latency sensitive
Background traffic flows	Low priority, “background” TCP flows

- moving transport–layer functions to application layer, on top of UDP
  - HTTP/3: QUIC

Summary

- Congestion control goal: avoid overwhelming network resources
- Congestion control mechanism: senders infer congestion, voluntarily reduce sending rate (by shrinking window) when congestion is detected
- Classic TCP approach: infer congestion based on loss, adapt window via “additive increase, multiplicative decrease”
- Many TCP variants, transport layer is still evolving…



The Network Layer

The Internet Protocol

Plan for Today

- Network layer: overview
  - data plane
  - control plane

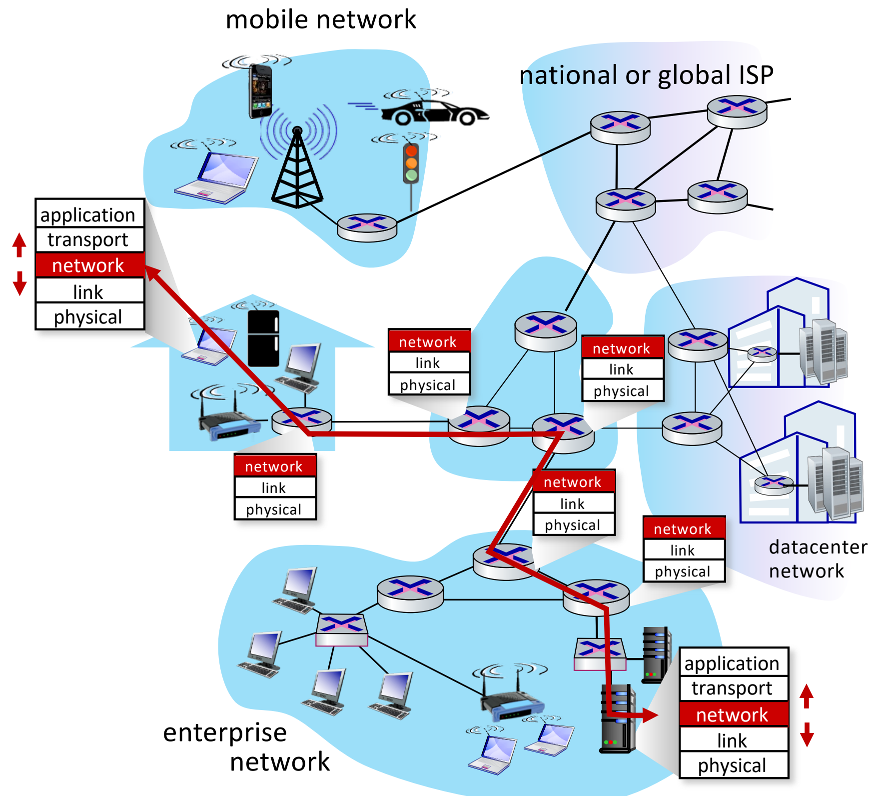
- Data plane forwarding: What's inside a router?
  - input ports, switching, output ports
  - buffer management, scheduling

Network Layer: Overview

Network-layer services and protocols

- **Network layer objective:**
  transport segment from sending to receiving host

- Network layer protocols in *every Internet device*: hosts, routers
  - Sending host: encapsulates segments into datagrams, passes to link layer
  - Receiving host: delivers segments to transport layer protocol
- **Routers**:
  - examine header fields in all IP datagrams passing through them
  - move datagrams from input ports to output ports to transfer datagrams along end-end path



Two key network-layer functions

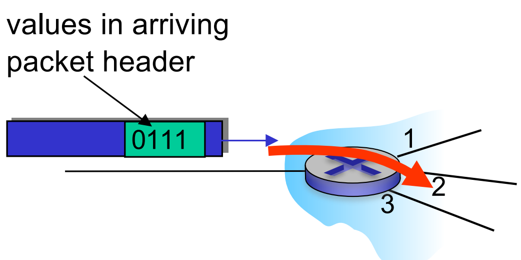
- *Forwarding:* move packets from a router's input link to appropriate router output link

- *Routing:* determine route taken by packets from source to destination
  - *routing algorithms*

Network layer: data plane, control plane

Data plane:

- *local*, per-router function
- determines how datagram arriving on router input port is **forwarded** to router output port

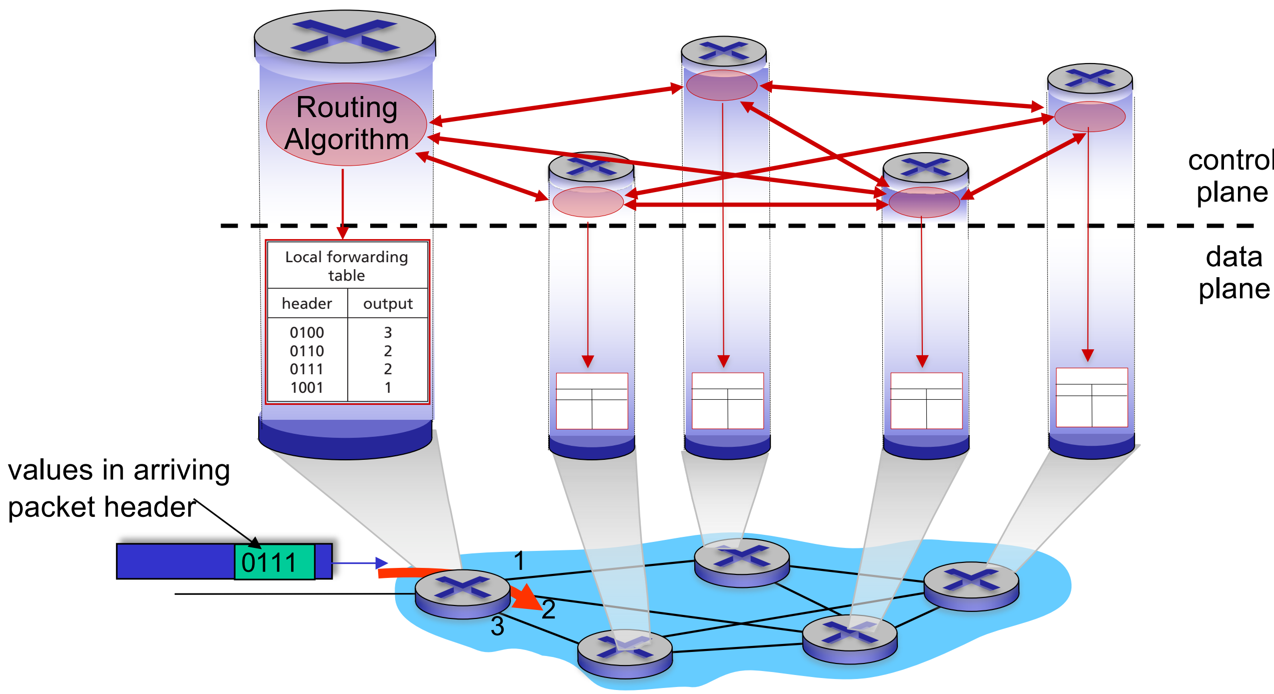


Control plane

- *network-wide* logic
- determines how datagram is **routed** among routers along end-end path from source host to destination host
- two control-plane approaches:
  - *traditional routing algorithms:* implemented in routers
  - *software-defined networking (SDN)*: implemented in (remote) servers

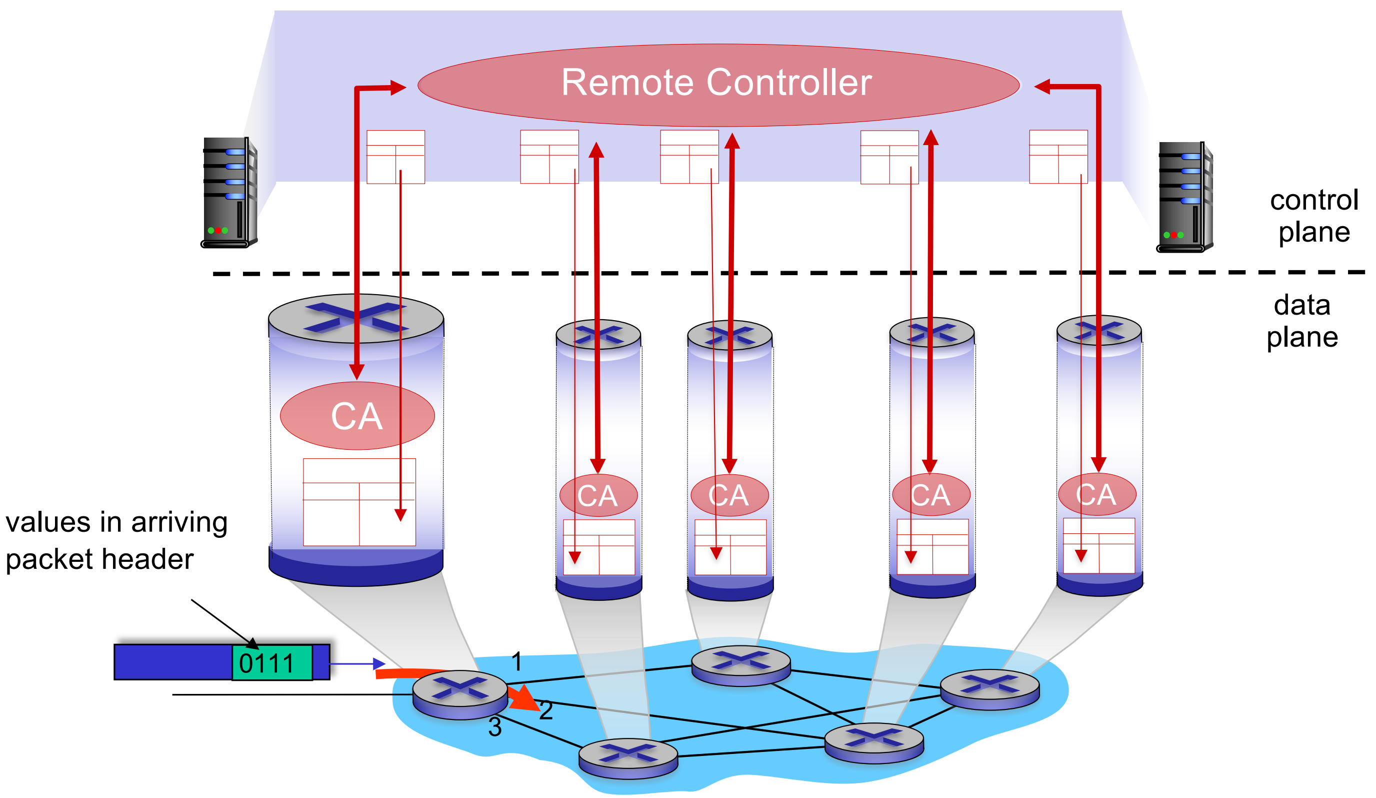
Per-router control plane

Individual routing algorithm components *in each and every router* interact in the control plane



Software-Defined Networking (SDN) control plane

Remote controller computes, installs forwarding tables in routers



Network-layer service model

- **Best-effort** host-to-host delivery

- **NO guarantees on:**
  - Reliable delivery
  - In-order delivery
  - Timely delivery
  - End-to-end bandwidth

- There have been some efforts to add quality-of-service guarantees (Intserv, Diffserv), but these approaches generally don't provide strict end-to-end guarantees (Diffserv) or suffer from scalability problems (Intserv) over the Internet
  - Can be useful in LANs / single-operator networks though

Reflections on best-effort service:

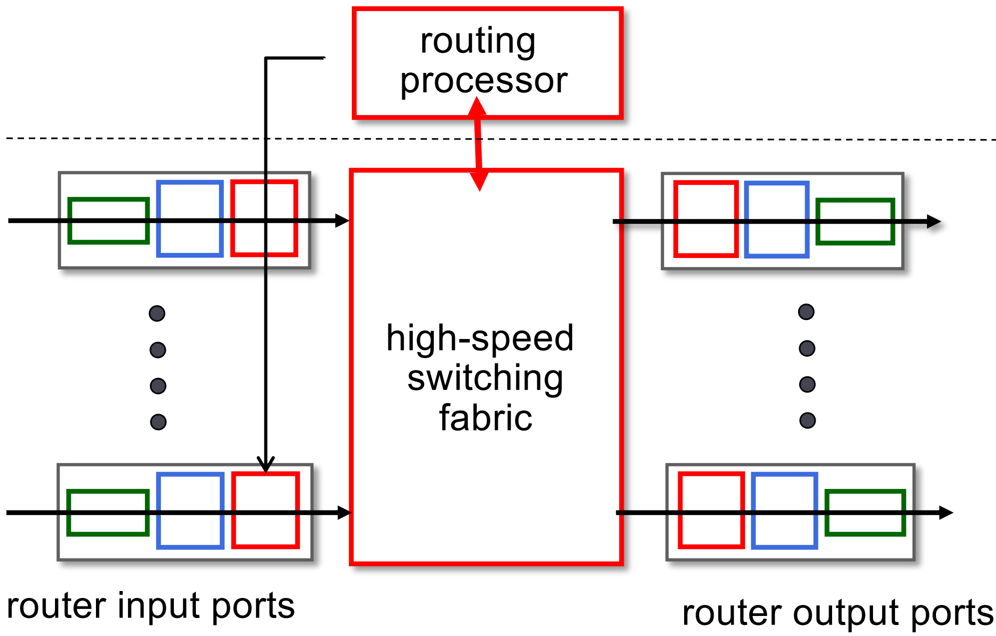
- simplicity of mechanism has allowed Internet to be widely deployed adopted
- sufficient provisioning of bandwidth allows performance of real-time applications (e.g., interactive voice, video) to be "good enough" for "most of the time"
- replicated, application-layer distributed services (datacenters, content distribution networks) connecting close to clients' networks, allow services to be provided from multiple locations
- congestion control of "elastic" services helps

It's hard to argue with success of best-effort service model

Routers and Forwarding

Router architecture overview

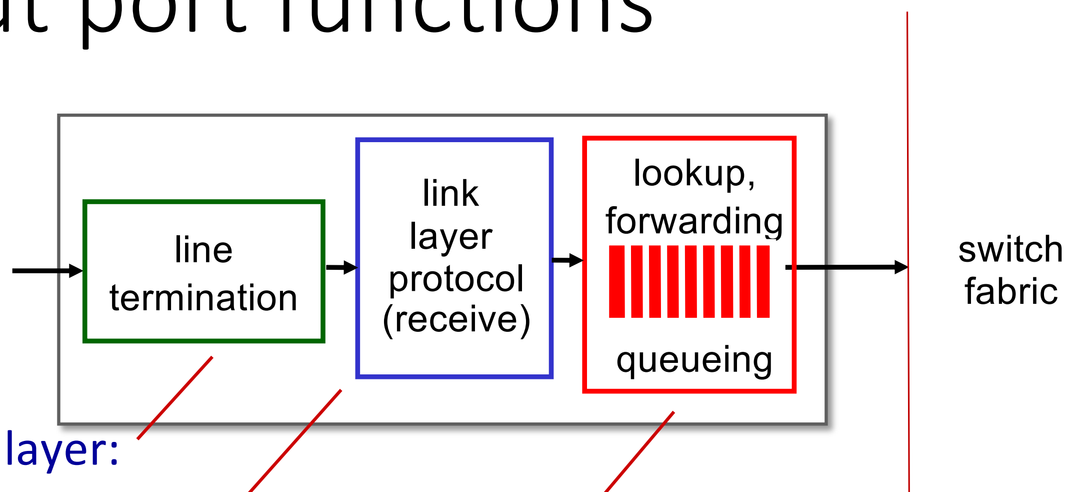
high-level view of generic router architecture:



routing, management control plane (software) operates in millisecond time frame

forwarding data plane (hardware) operates in nanosecond timeframe

Input port functions



physical layer:
bit-level reception

link layer:
e.g., Ethernet
(chapter 6)

network layer:

- **look up** output port using forwarding table
- goal: complete input port **processing** at 'line speed'
- **input port queuing**: if datagrams arrive faster than forwarding rate into switch fabric

Destination-based Forwarding

- The **forwarding table** is *indexed* using the destination address
- If the address is 32 bit long, there are 2^{32} different addresses. The table cannot contain 2^{32} (> 4 billion) entries!

Destination-based forwarding

forwarding table

Destination Address Range	Link Interface	
11001000 00010111 00010000 00000000 through 11001000 00010111 00010111 11111111	0	200.23.16.0 through 200.23.23.255
11001000 00010111 00011000 00000000 through 11001000 00010111 00011000 11111111	1	200.23.24.0 through 200.23.24.255
11001000 00010111 00011001 00000000 through 11001000 00010111 00011111 11111111	2	200.23.25.0 through 200.23.31.255
otherwise	3	

Longest prefix matching

longest prefix match

when looking for forwarding table entry for given destination address, use *longest* address prefix that matches destination address.

Destination Address Range	Link interface
11001000 00010111 00010*** ********	0
11001000 00010111 00011000 ********	1
11001000 00010111 00011*** ********	2
otherwise	3

examples:

11001000 00010111 00010110 10100001 which interface?

11001000 00010111 00011000 10101010 which interface?

Longest prefix matching

longest prefix match

when looking for forwarding table entry for given destination address, use *longest* address prefix that matches destination address.

Destination Address Range	Link interface
11001000 00010111 00010*** ********	0
11001000 00010111 00011000 ********	1
11001000 00010111 00011*** ********	2
otherwise	3

match!

examples:

11001000 00010111 00010110 10100001 which interface?

11001000 00010111 00011000 10101010 which interface?

Longest prefix matching

longest prefix match

when looking for forwarding table entry for given destination address, use *longest* address prefix that matches destination address.

Destination Address Range	Link interface
11001000 00010111 00010*** ********	0
11001000 00010111 00011000 ********	1
11001000 00010111 00011*** ********	2
otherwise	3

match!

examples:

11001000 00010111 00010110 10100001 which interface?

11001000 00010111 00011000 10101010 which interface?

Longest prefix matching

longest prefix match

when looking for forwarding table entry for given destination address, use *longest* address prefix that matches destination address.

Destination Address Range				Link interface
11001000	00010111	00010***	********	0
11001000	00010111	00011000	********	1
11001000	00010111	00011***	********	2
otherwise				3

match!

examples:

11001000	00010111	00010110	10100001	which interface?
11001000	00010111	00011000	10101010	which interface?

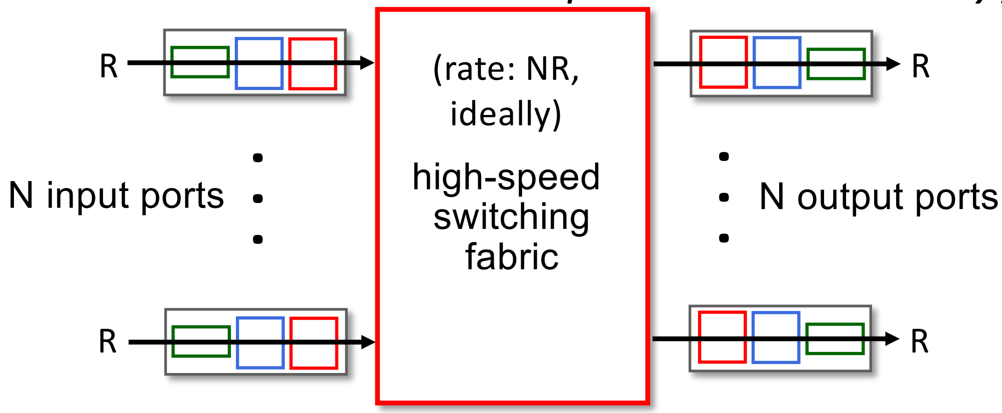
Forwarding

- **Fast lookups**: longest prefix matching is often performed using ternary content addressable memories (**TCAMs**)
  - *content addressable:* present IP address to TCAM: retrieve forwarding table entry in one clock cycle, regardless of table size
  - Cisco Catalyst: ~1M routing table entries in TCAM

- The forwarding **table can change at any time**
  - Packets belonging at the same flow might follow different paths to the destination

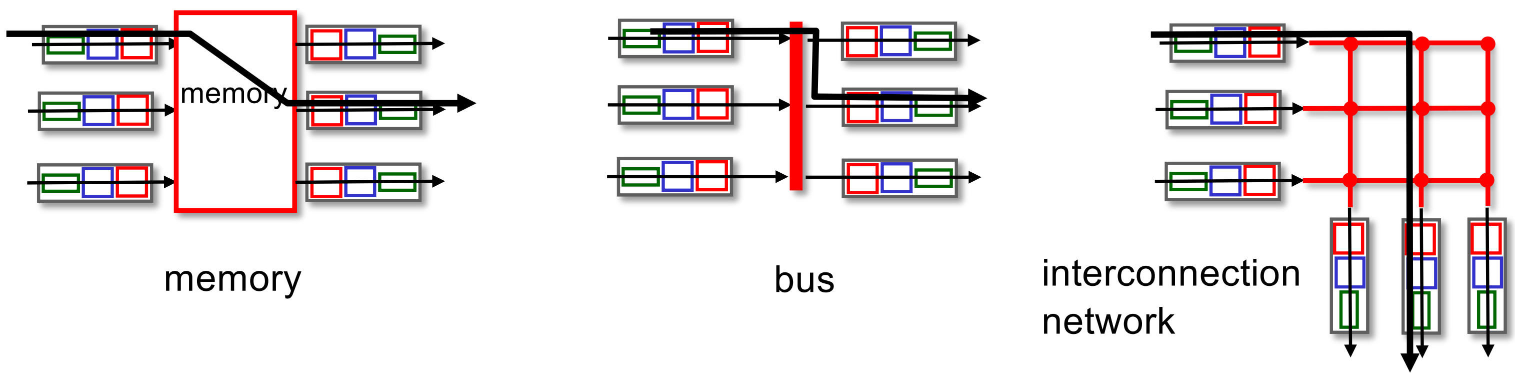
Switching fabrics

- transfer packet from input link to appropriate output link
- switching rate: rate at which packets can be transferred from inputs to outputs
  - often measured as multiple of input/output line rate (R)
  - N inputs: switching rate N times line rate desirable (i.e. 12-port 1 Gb switch should be able to send at 1 Gb on *all 12 ports simultaneously*)



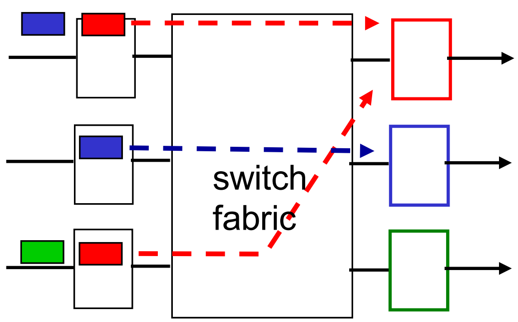
Switching fabrics

- transfer packet from input link to appropriate output link
- switching rate: rate at which packets can be transferred from inputs to outputs
  - often measured as multiple of input/output line rate
  - N inputs: switching rate N times line rate desirable
- three major types of switching fabrics:

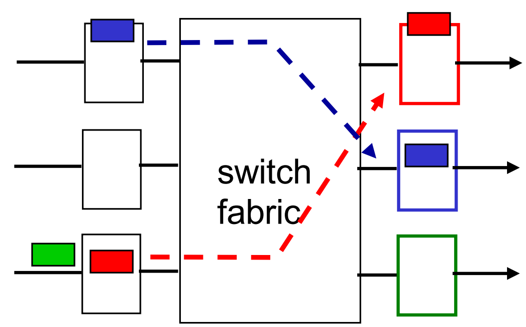


Input port queuing

- If switch fabric slower than input ports combined -> queueing may occur at input queues
  - queueing delay and loss can occur due to input buffer overflow!
- Head-of-the-Line (HOL) blocking: queued datagram at front of queue prevents others in queue from moving forward



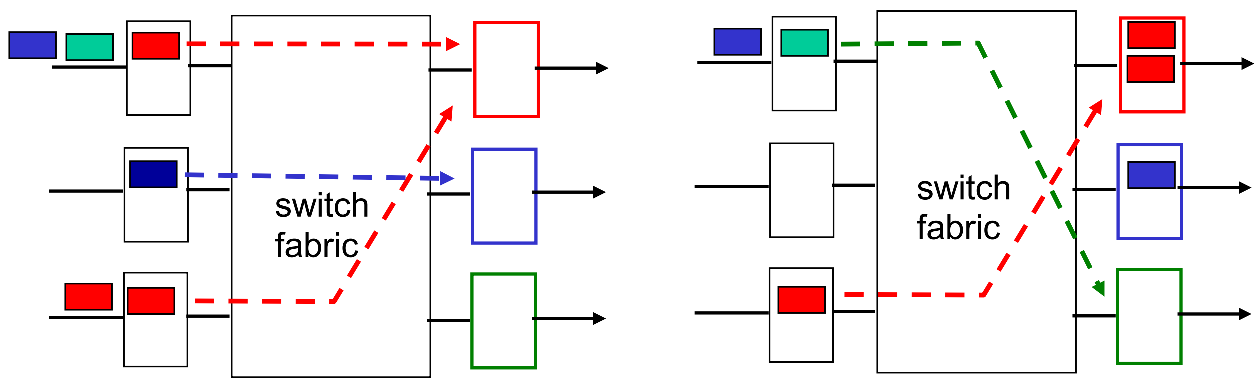
output port contention: only one red datagram can be transferred. lower red packet is *blocked*



one packet time later: green packet experiences HOL blocking

Output port queuing

Bigger problem than input port queuing (switch fabrics are typically fast enough today)

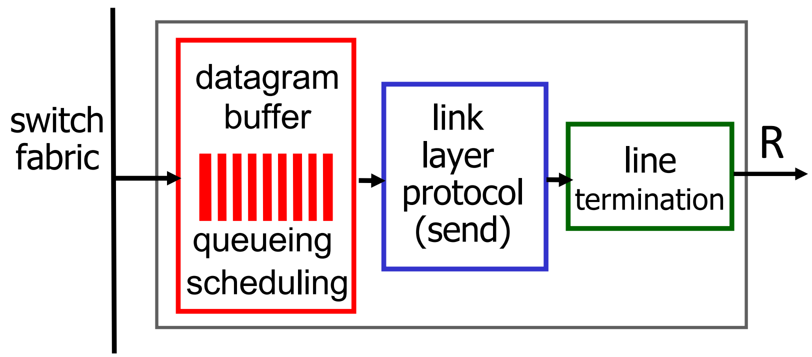


at *t,* packets move from input to output

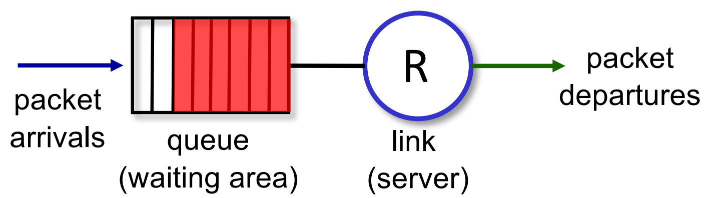
one packet time later

- buffering when arrival rate via switch exceeds output line speed
- *queueing (delay) and loss can occur due to output port buffer overflow!*

Managing queues



Abstraction: queue



- buffer management: **which packet to drop when buffer is full?**
  - tail drop: drop arriving packet
  - priority: drop/remove on priority basis

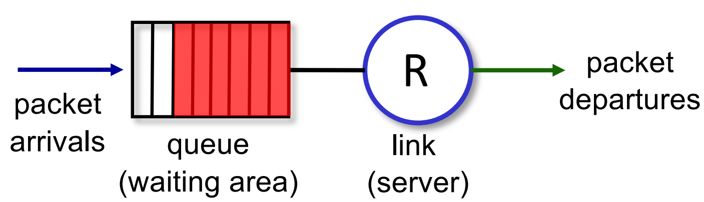
packet scheduling: **which packet to send next from the queue?**
- first come, first served
- priority
- round robin
- weighted fair queueing

Packet Scheduling: FIFO

FIFO (first-in-first-out): packets transmitted in order of arrival to output port

- "first-come-first-served"
- this is how we normally think of "waiting in line"

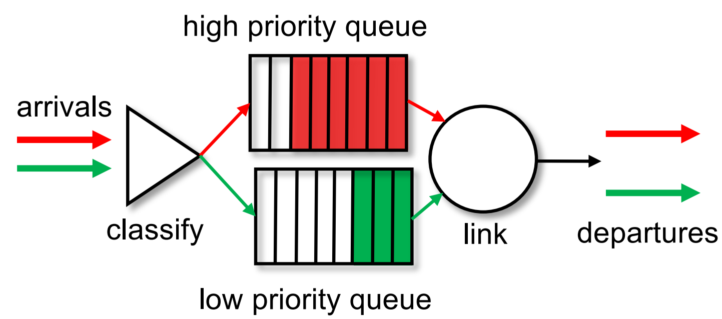
Abstraction: queue



Scheduling policies: priority

Priority scheduling:

- arriving traffic classified, queued by class
  - any header fields can be used for classification
- send packet from highest priority queue that has buffered packets
  - FIFO within priority class

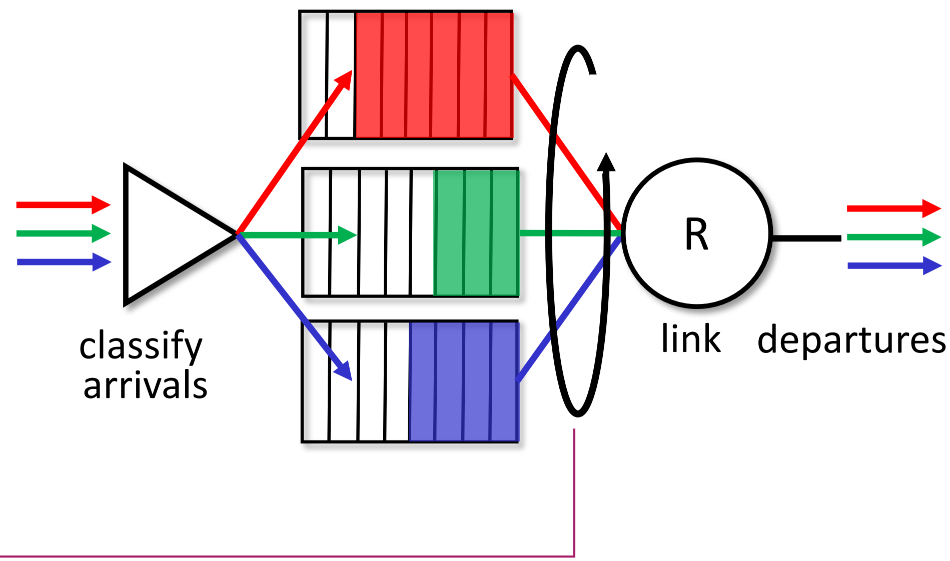


This is *basically* how DiffServ aims to provide quality-of-service guarantees for traffic that requires low latency…packets get tagged with high priority class that gets preference over other traffic

Scheduling policies: round robin

Round Robin (RR) scheduling:

- arriving traffic classified, queued by class
  - any header fields can be used for classification
- server cyclically, repeatedly scans class queues, sending one complete packet from each class (if available) in turn



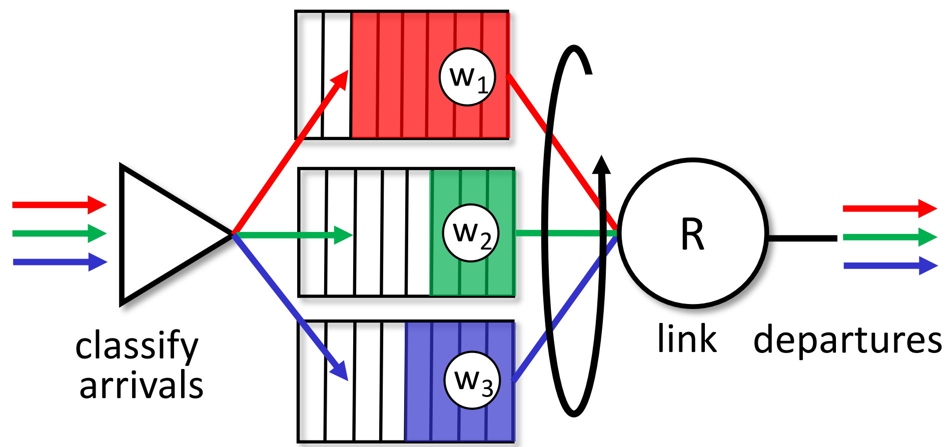
Scheduling policies: weighted fair queueing

Weighted Fair Queuing (WFQ):

- generalized Round Robin
- each class, i, has weight, w_i, and gets weighted amount of service in each cycle:

$$\frac{w_i}{\Sigma_j w_j}$$

- minimum bandwidth guarantee (per-traffic-class)



Scheduling policies: enforcing fairness

- Round-robin / weighted fair queuing **could** be used to enforce fair sharing of bandwidth (instead of relying on TCP congestion control for this)

- Consider 1 class per flow, all classes treated in round-robin manner
  - Good: we get fair bandwidth sharing, **even if hosts misbehave**
  - Bad: how much state does the router need to maintain???
    - Depends on how we define a flow. Definitely NOT possible for every TCP connection
    - Can be used for coarse-grained fairness based on source prefixes

Summary

- Network layer service: best-effort host-to-host transport
- 2 key functions:
  - *Forwarding:* move packets from a router's input link to appropriate router output link
  - *Routing:* determine route taken by packets from source to destination
- Routers forward packets based on destination IP address, by performing longest-prefix matching lookup in forwarding table
- Packet scheduling policies determine how routers manage queues