

Lecture 13 (Transport 3)

Congestion Control, Part 1

CS 168, Spring 2025 @ UC Berkeley

Slides credit: Sylvia Ratnasamy, Rob Shakir, Peyrin Kao

Congestion Control: Why do we need it?

Lecture 13, CS 168, Spring 2025

Congestion Control Principles

- **Why do we need it?**
- Why is it hard?
- Design Goals

Dynamic Adjustment

- Algorithm Sketch
- Reacting to Congestion (AIMD)

Recall: Congestion at Routers

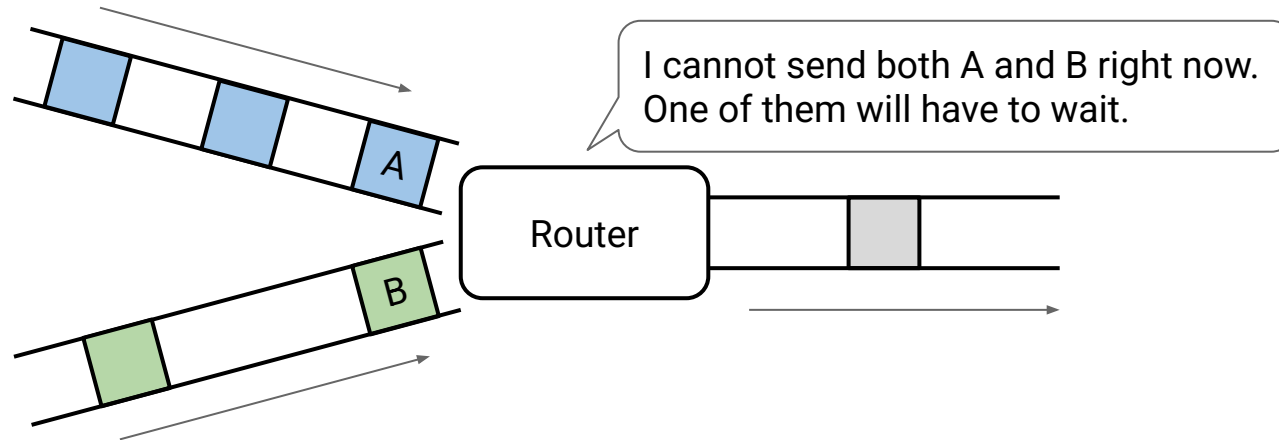
If two packets arrive at the same time, the router will send one, and buffer the other.

The router maintains a *queue* of packets that still need to be sent.

- If the queue is full, and a packet arrives, the packet gets dropped.

If too many packets arrive close in time:

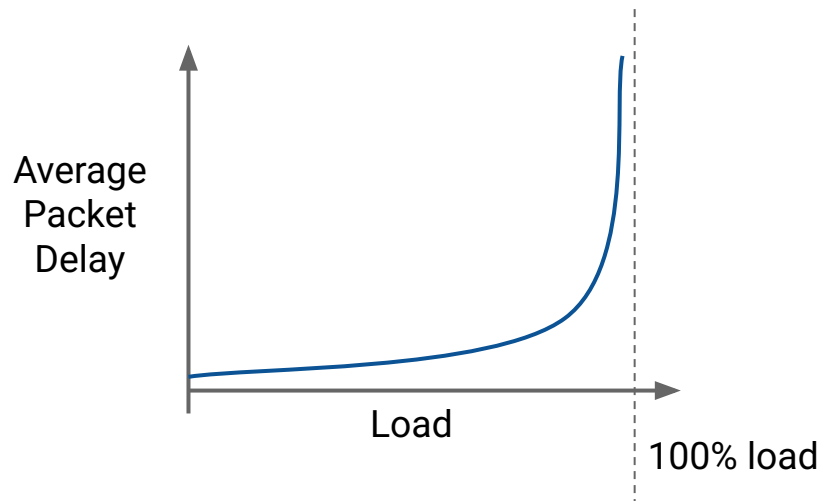
- The router cannot keep up, and gets congested.
- Packets can be delayed (stuck waiting in queue) or dropped (queue is full).



Congestion Delays Packets

For a typical queuing system with bursty arrivals:

- Higher load = more packets sent = higher packet delay.
- Delay gets really bad, even before we reach 100% load.
- Trade-offs: Need to balance high link utilization and low delay.



Brief History of Congestion Control

In the 1980s, TCP did not implement congestion control.

- Sending rate was only limited by flow control (recipient buffer capacity).
- If packets are dropped, resend them over and over, at the same fast rate.

This led to a *congestion collapse*.

- 1986: Capacity of a major link dropped to 0.1% of its original capacity.
- Why? Network was flooded with thousands of copies of every packet.

In October of '86, the Internet had the first of what became a series of 'congestion collapses.' During this period, the data throughput from LBL to UC Berkeley (sites separated by 400 yards and two IMP hops) dropped from 32 Kbps to 40 bps. We were fascinated by this sudden factor-of-thousand drop in bandwidth and embarked on an investigation of why things had gotten so bad. In particular, we wondered if the 4.3BSD (Berkeley Unix) TCP was mis-behaving or if it could be tuned to work better under abysmal network conditions. The answer to both of these questions was "yes".

– Karels (UCB) and Jacobson (LBL)

Brief History of Congestion Control

How was congestion collapse solved in the 1980s?

- Michael Karels and Van Jacobson were working on BSD (an early OS).
- They changed a few lines of TCP code to fix the problem.
 - Recall: TCP is implemented in everybody's OS.
 - The fix worked, and everybody quickly adopted it.

The history of congestion control had a big impact on its design.

- Invented as an ad-hoc patch to an existing system. (*Not in original TCP design.*)
- Implemented in the OS. (*Not in applications, routers, etc.*)

Fast-forward to today:

- Extensive research and improvements since the original patch.
- The core ideas in the original patch still persist.
- No major congestion collapses ever since.

Congestion Control: Why is it hard?

Lecture 13, CS 168, Spring 2025

Congestion Control Principles

- Why do we need it?
- **Why is it hard?**
- Design Goals

Dynamic Adjustment

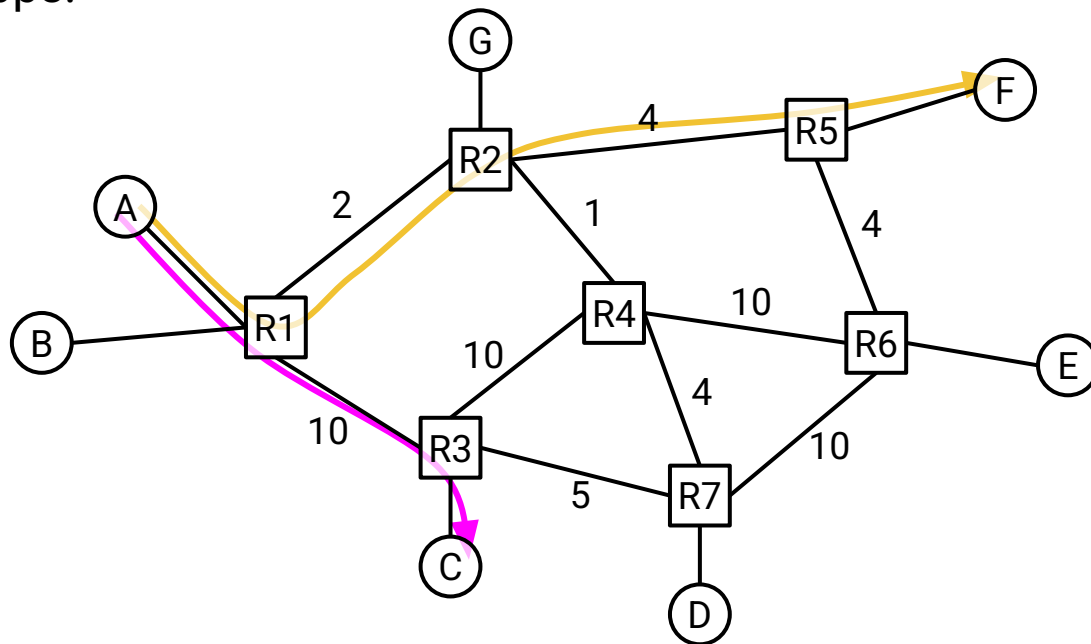
- Algorithm Sketch
- Reacting to Congestion (AIMD)

Congestion Control Challenges: Depends on Destination

At what rate should A send traffic?

It depends on the destination.

- $A \rightarrow C$ can send at 10 Gbps.
- $A \rightarrow F$ can only send at 2 Gbps.



Numbers are bandwidths, in Gbps.

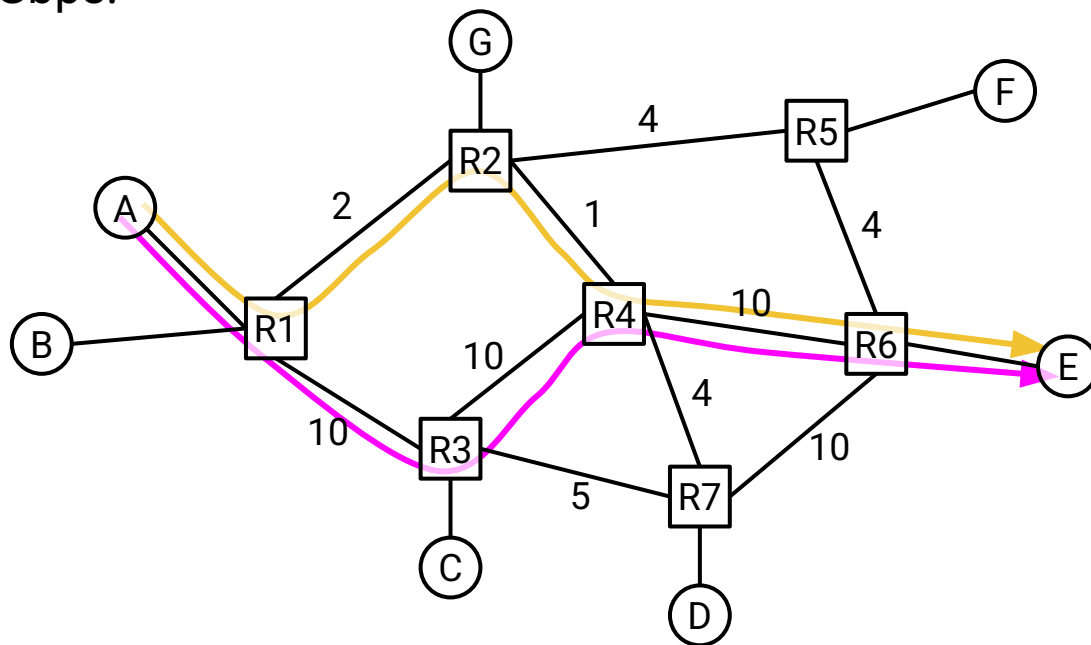
Ignore bandwidth of host-to-router links.

Congestion Control Challenges: Depends on Path

At what rate should $A \rightarrow E$ send traffic?

It depends on the path through the network.

- $A \rightarrow E$ via R3 can send at 10 Gbps.
- $A \rightarrow E$ via R2 can send at 1 Gbps.

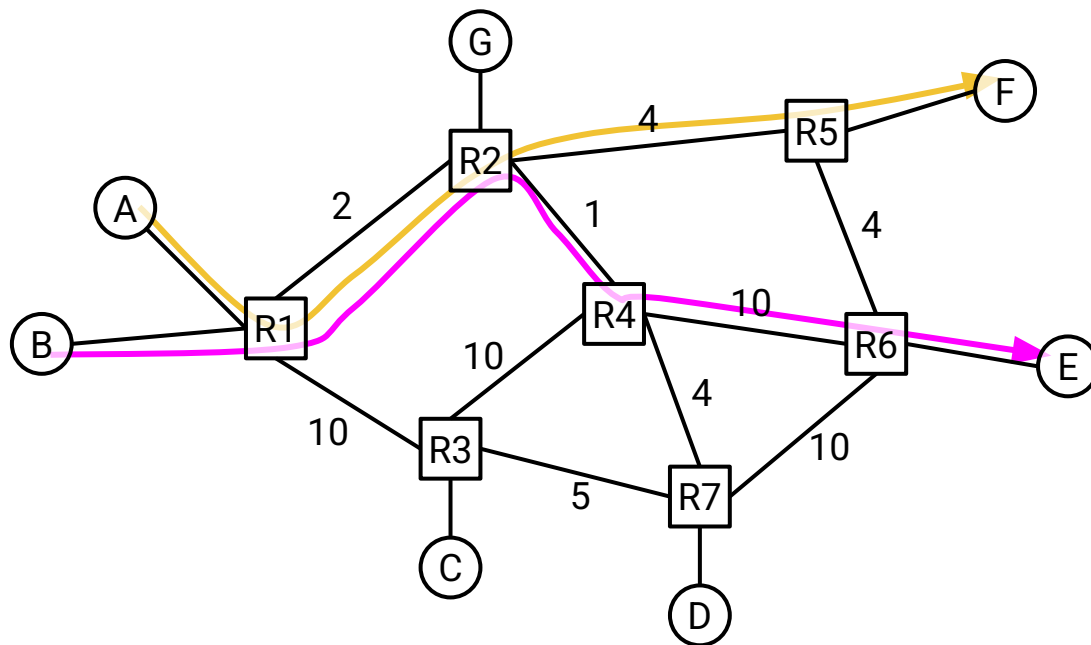


Congestion Control Challenges: Depends on Competing Flows

At what rate should $A \rightarrow F$ send traffic?

It depends on others using the network.

- $A \rightarrow F$ and $B \rightarrow E$ share a link.
- $B \rightarrow E$ can send at 1 Gbps.
- $A \rightarrow F$ can send at 1 Gbps.

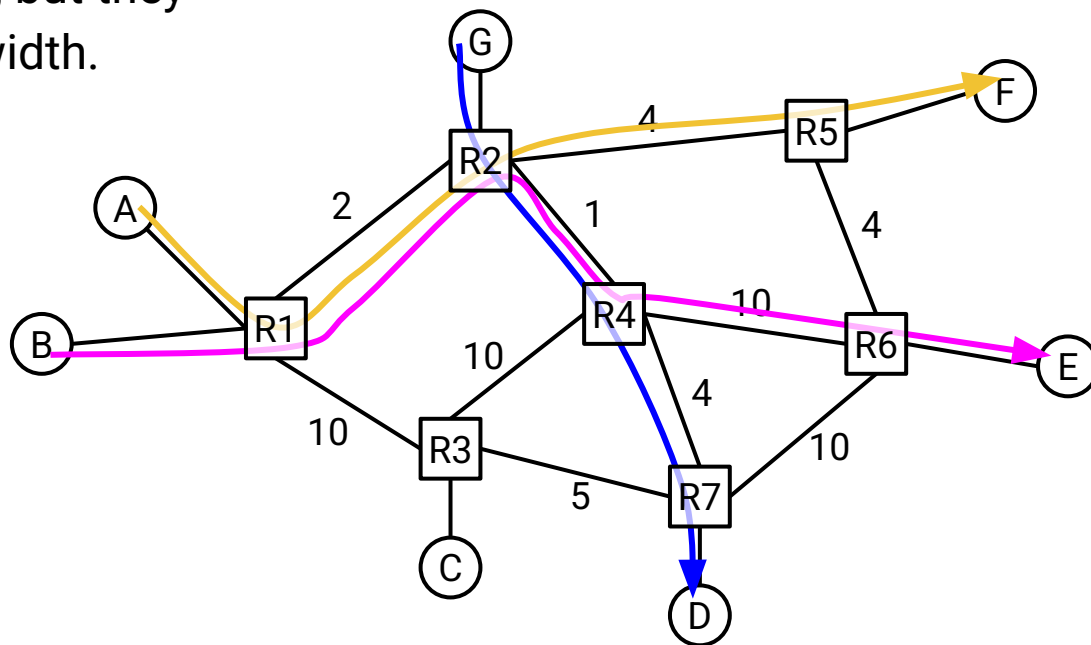


Congestion Control Challenges: Depends on Indirect Competition

What if $G \rightarrow D$ starts sending traffic?

- Maybe $B \rightarrow E$ should slow to 0.5 Gbps.
- Then $A \rightarrow F$ could increase to 1.5 Gbps.

$G \rightarrow D$ and $A \rightarrow F$ share no links, but they still affected each other's bandwidth.



Congestion Control is a Global Problem

The sender's optimal rate depends on:

- The destination.
- The path to the destination.
- Other connections sharing links along that path.
- Connections sharing links with those other connections.
- And so on...

Congestion control is a global problem.

- Connections in the network are highly interdependent.
- Solving it optimally requires a global view of the network.

Congestion Control as Resource Allocation Problem

Fundamentally, congestion control is a *resource allocation problem*.

- Bandwidth is a limited resource.
- Each connection demands a certain amount of bandwidth.
- We decide how much to allocate to each connection.

Resource allocation is a classic problem (e.g. CPU scheduling, memory allocation).

But, congestion control has additional challenges.

- A change in one connection can have a global impact.
- Network topology is constantly changing.
- Connections are constantly starting and ending.
- Everybody must solve it locally. No global view of network.

Design Goals

Lecture 13, CS 168, Spring 2025

Congestion Control Principles

- Why do we need it?
- Why is it hard?
- **Design Goals**

Dynamic Adjustment

- Algorithm Sketch
- Reacting to Congestion (AIMD)

Goals for a good congestion control algorithm:

1. Avoid congestion: Minimize packet delay and loss.
2. Efficiency: Maximize link utilization.
3. Fairness: Connections should share the available bandwidth.
4. Practical to implement: Scalable, decentralized, adaptive, etc.

Our ultimate goal is a good tradeoff between these metrics.

- Could avoid congestion (1) by sending really slowly.
- But then we aren't using all the bandwidth on the link (2).

Reservations:

- Connections reserve bandwidth at the start, and release bandwidth at the end.
- Requires connections to know bandwidth ahead of time.
- Other problems: Hard to implement, inefficient bandwidth usage, etc.

Pricing/priorities:

- Analogy: Express toll lanes on the highway. Price changes depending on traffic.
- Prioritize traffic from users who pay more.
- Requires a payment model.

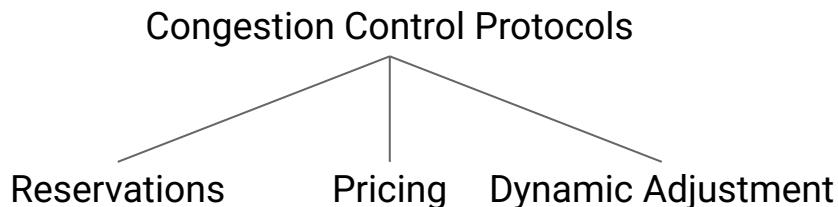
Dynamic adjustment:

- Learn the current congestion level, and adjust your rate accordingly.
- Requires good citizenship (no cheating).

3 possible approaches: reservations, pricing, and dynamic adjustment.

All the algorithms we'll see are based on dynamic adjustment.

- If we rebuilt the Internet from scratch, another approach might be feasible.
- But congestion control was invented as a patch on the existing Internet.
- Dynamic adjustment is the best fit for the existing Internet:
 - Doesn't require knowing bandwidth ahead of time.
 - Doesn't require a payment model.



Dynamic Adjustment: Algorithm Sketch

Lecture 13, CS 168, Spring 2025

Congestion Control Principles

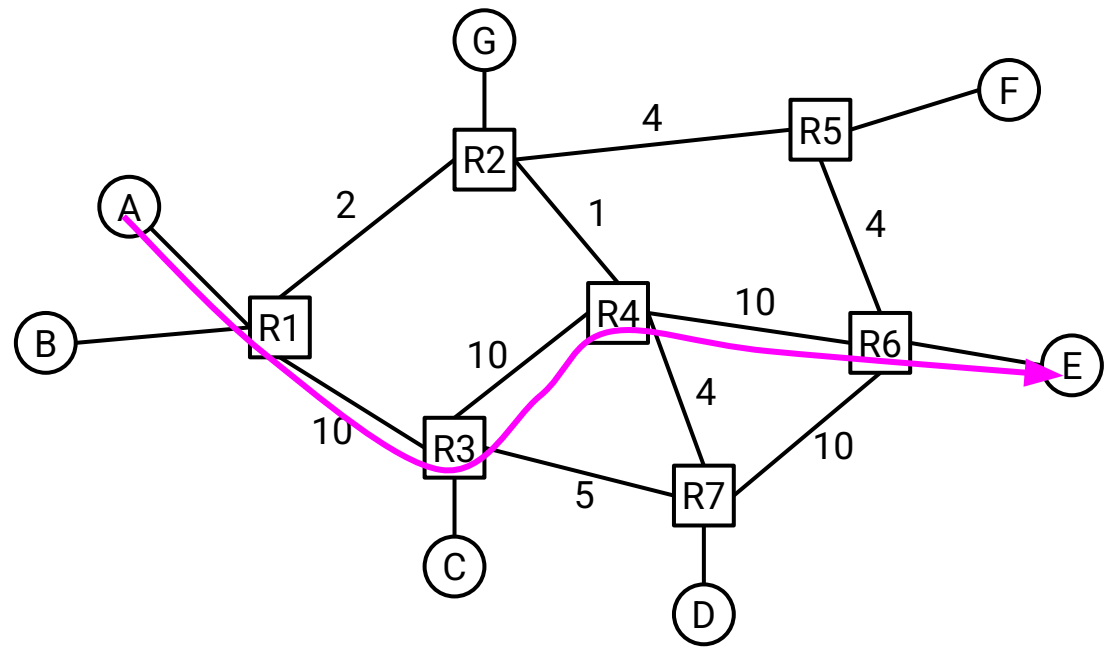
- Why do we need it?
- Why is it hard?
- Design Goals

Dynamic Adjustment

- **Algorithm Sketch**
- Reacting to Congestion (AIMD)

Dynamic Adjustment: Discovery

Discovery: Host A discovers it can send data at 10 Gbps.

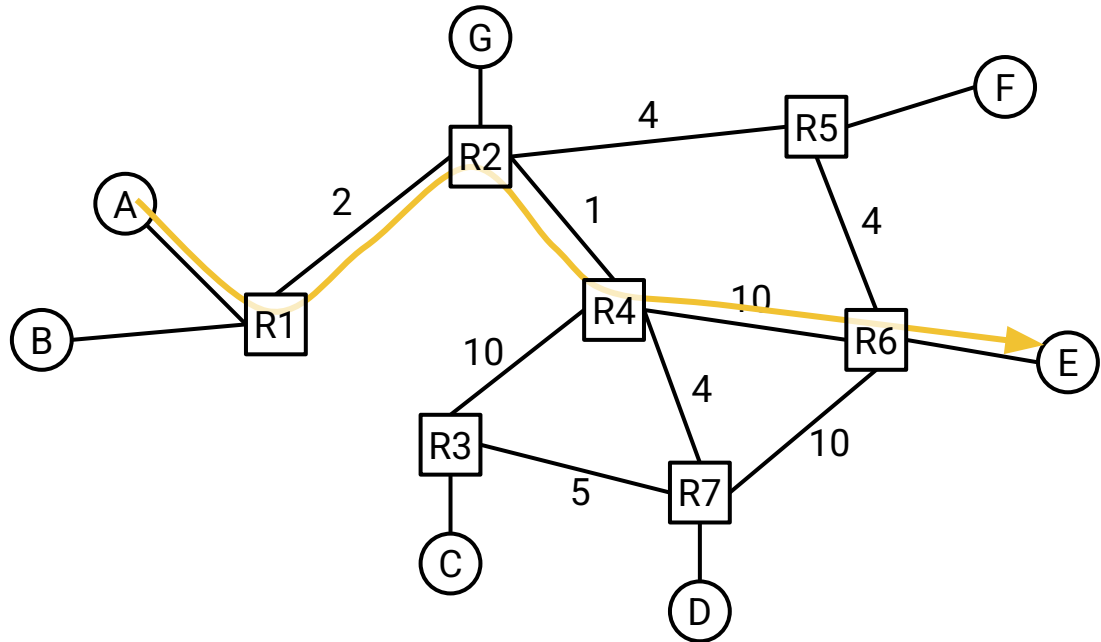


Dynamic Adjustment: Detection and Adjustment

Later, the $R1 \rightarrow R3$ link goes down. A's traffic is rerouted.

Detection: A notices that 10 Gbps is congesting the network.

Adjustment: A decides to slow down to 1 Gbps.

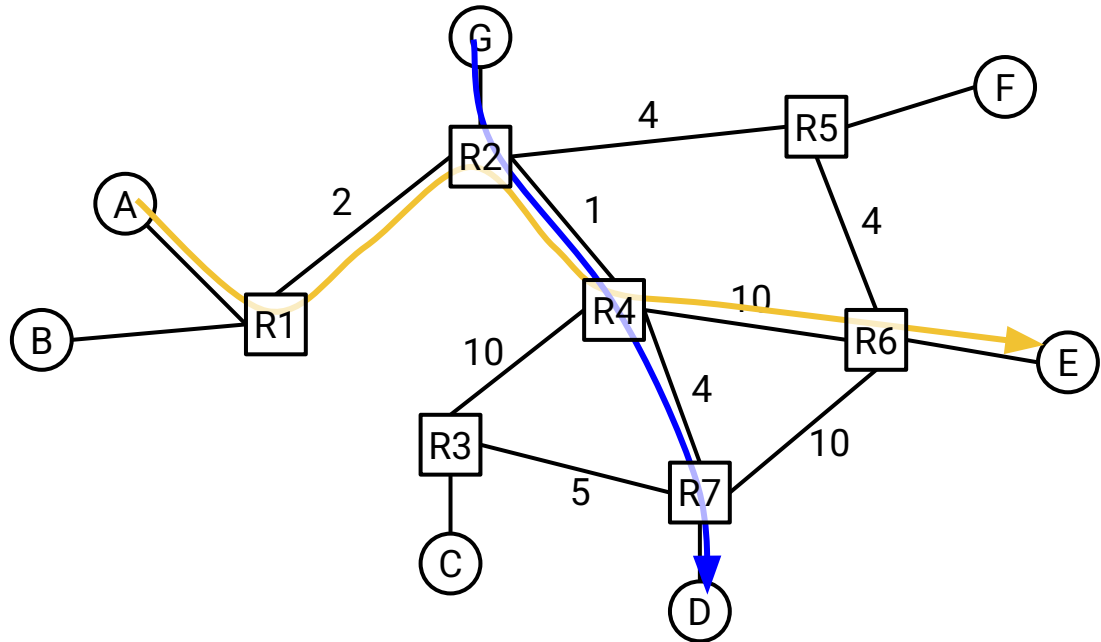


Dynamic Adjustment: Detection and Adjustment

Later, $G \rightarrow D$ starts sending traffic.

Detection: A notices that 1 Gbps is congesting the network.

Adjustment: A decides to slow down to 0.5 Gbps.



Types of Dynamic Adjustment Algorithms

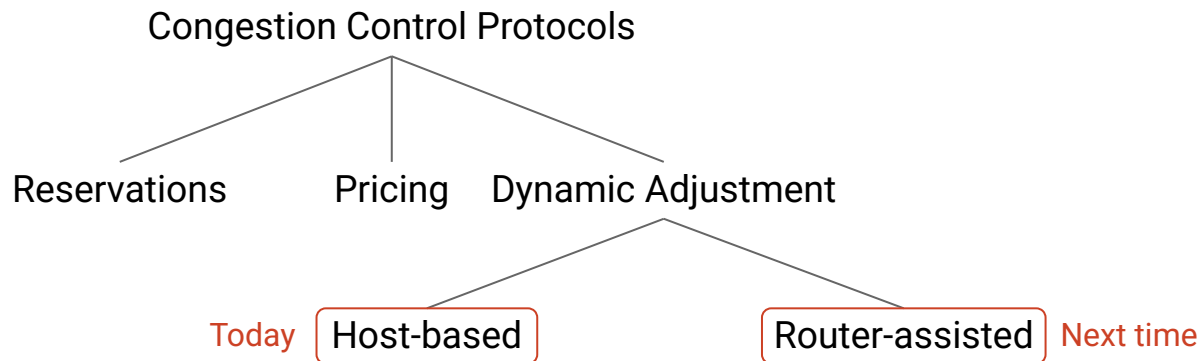
Solutions can be classified into 2 types, depending on where they're implemented:

Host-based congestion control: (*TCP uses this one*)

- Implemented at the end hosts. No special support from routers.
- Hosts adjust rate based on *implicit* feedback from routers.

Router-assisted congestion control:

- Routers signal congestion back to end hosts.
- Hosts pick rate based on *explicit* feedback from routers.



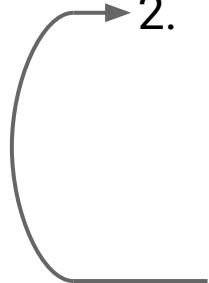
Host-Based Dynamic Adjustment: Algorithm Sketch

Every host *independently* runs the same algorithm:

1. Pick an initial rate R .

2. Try sending at rate R for some period of time.

- Did I experience congestion in this time period?
- If no, increase R .
- If yes, reduce R .
- Repeat.

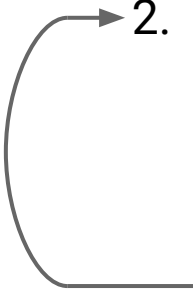


Host-Based Dynamic Adjustment: Algorithm Sketch

Every host *independently* runs the same algorithm:

1. Pick an initial rate R . (*How do we pick the initial rate?*)

→ 2. Try sending at rate R for some period of time.

- Did I experience congestion in this time period? (*How do we detect congestion?*)
 - If no, increase R .
 - If yes, reduce R . (*How much do we increase/decrease by?*)
 - Repeat.
- 

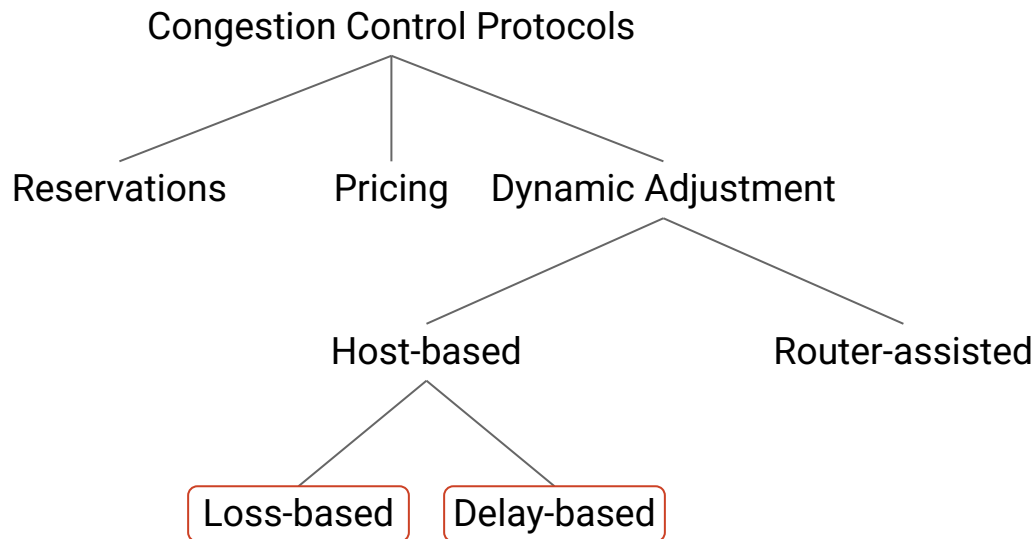
Components of a solution:

- Discovering an initial rate.
- Detecting congestion.
- Reacting to congestion (or lack thereof) by increasing/decreasing rate.

Detecting Congestion

There are 2 ways for the host to detect congestion.

Remember: The host can't see the whole network.



Loss-based detection: If we lose a packet, declare congestion.

- This is what TCP uses.

Benefits:

- Unambiguous signal: Every packet is either lost (resent), or not lost.
- TCP already detects loss for reliability.

Cons:

- Not all loss is caused by congestion. (e.g. checksum error).
- A delayed packet could be mistakenly declared lost.
- When we detect loss, the network is already congested (queues are full).

Delay-based detection: If packets are delayed, declare congestion.

- Challenge: Packet delay length varies with queue size and other traffic.
- Long considered tricky to get right.
- Google's BBR protocol (2016) is challenging this assumption.

Discovering an Initial Rate

Goal for discovering initial rate: Estimate available bandwidth.

Start slow for safety.

- Starting too fast would cause congestion.

Ramp up quickly for efficiency.

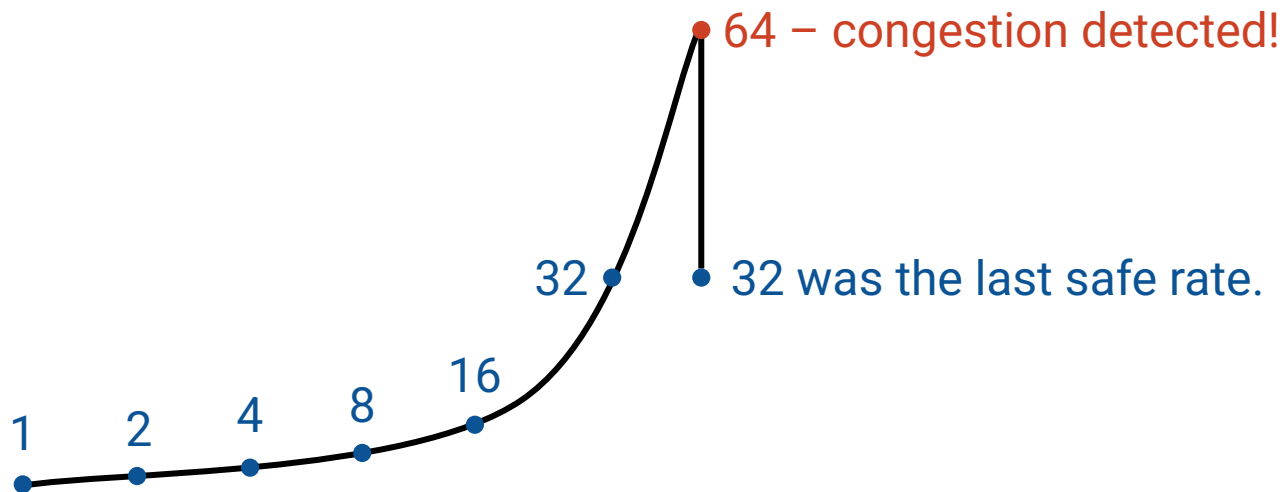
- Example of inefficiency: Suppose we added 0.5 Mbps every 0.1 seconds.
- If 1 Mbps available, we'll reach the target in 0.2 seconds.
- If 1 Gbps available, we'll reach the target in 200 seconds.
 - Lots of time wasted sending at suboptimal rates.

What kind of mathematical function starts slow but ramps up quickly?

Discovering an Initial Rate: Slow Start

Slow start is an algorithm for discovering the initial rate:

- Start with a small rate (could be much less than actual bandwidth).
- Increase exponentially (e.g. double rate each time) until we detect loss.
- A safe rate is the most recent rate before detecting loss.
 - $0.5 \times$ rate when loss occurred.



Reacting to Congestion

Lecture 13, CS 168, Spring 2025

Congestion Control Principles

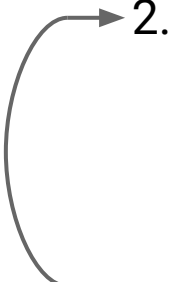
- Why do we need it?
- Why is it hard?
- Design Goals

Dynamic Adjustment

- Algorithm Sketch
- **Reacting to Congestion (AIMD)**

Host-Based Dynamic Adjustment: Algorithm Sketch

Every host *independently* runs the same algorithm:

1. Pick an initial rate R . (*Slow start.*)
 2. Try sending at rate R for some period of time.
 - Did I experience congestion in this time period? (*Detect packet loss.*)
 - If no, increase R .
 - If yes, reduce R . (*How much do we increase/decrease by?*)
 - Repeat.
- 

Rate adjustment: How much do we increase/decrease by?

- Critical part of congestion control design.
- Determines how quickly a host adapts to changes in available bandwidth.
- Determines how effectively bandwidth is consumed.
- Determines how bandwidth is shared (fairness).

Goals for rate adjustment:

- Efficiency: High utilization of link bandwidth.
- Fairness: Each flow gets an equal share of bandwidth.

At a high level, we can either adjust quickly or slowly.

- Fast: **Multiplicative** changes.
 - Increase by doubling: $R \rightarrow 2R$
 - Decrease by halving: $R \rightarrow R/2$
- Slow: **Additive** changes.
 - Increase by adding: $R \rightarrow R + 1$
 - Decrease by subtracting: $R \rightarrow R - 1$

We can combine these rates in four ways:

- AIAD: Additive Increase, Additive Decrease.
- AIMD: Additive Increase, Multiplicative Decrease.
- MIAD: Multiplicative Increase, Additive Decrease.
- MIMD: Multiplicative Increase, Multiplicative Decrease.

Why AIMD? Intuition

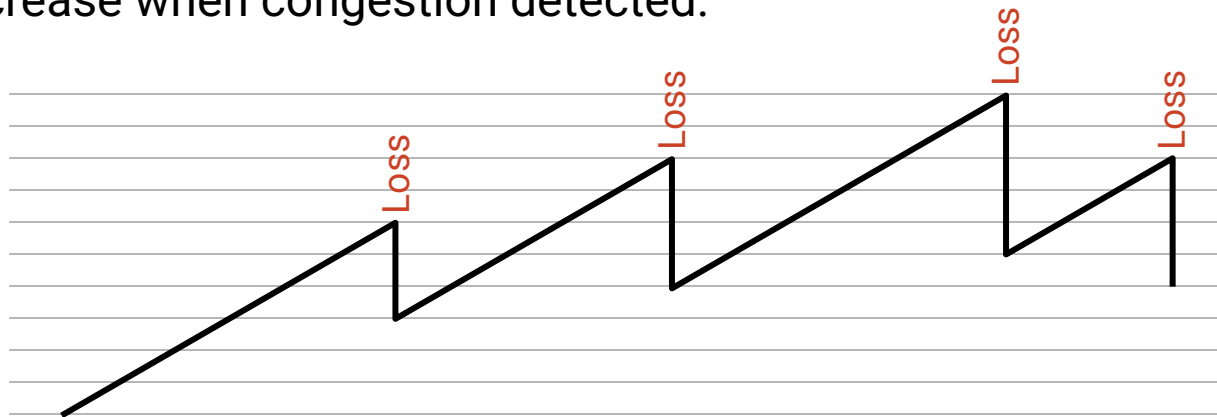
Non-obvious fact: AIMD (Additive Increase, Multiplicative Decrease) is the best option.

Intuition: Sending too much is worse than sending too little.

- Sending too much: Congestion, packets dropped and retransmitted.
- Sending too little: Somewhat lower throughput.

General approach:

- Gentle increase when uncongested. (exploration)
- Rapid decrease when congestion detected.



Non-obvious fact: Of the four options, AIMD is the only one that ensures fairness.

To see why, let's build a simple model:

- Two flows using a single link of fixed capacity C .
- Both flows are *independently* running *identical* dynamic adjustment algorithms.
- Flows are sending at rates X and Y .
 - If $X + Y > C$, the network is congested.
 - If $X + Y < C$, the network is underloaded.
- Goals:
 - Full link utilization: $X + Y = C$.
 - Fair sharing: $X = Y$.

Rate Adjustment Graph

Assume $C = 1$.

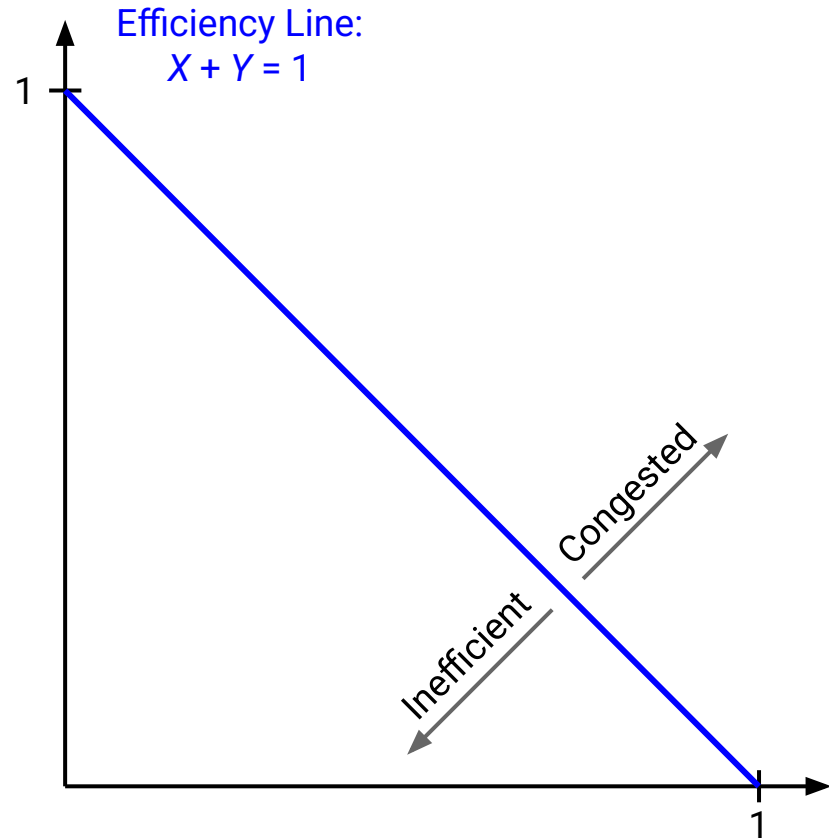
Efficiency line: $X + Y = 1$

Congestion: $X + Y > 1$

Everything above the efficiency line.

Unused capacity: $X + Y < 1$

Everything below the efficiency line.

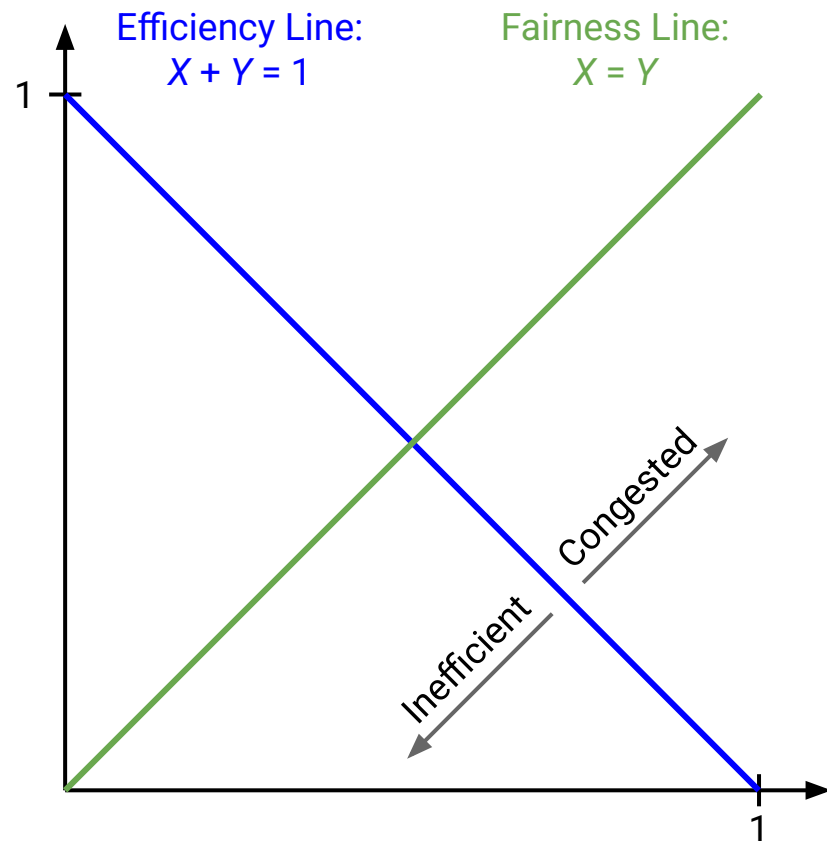


Rate Adjustment Graph

Fairness line: $X = Y$

The lines intersect at $(0.5, 0.5)$.

This allocation is both fair and efficient.



Rate Adjustment Graph

$(0.2, 0.5)$ is inefficient (below blue line).

Only 0.7 bandwidth used.

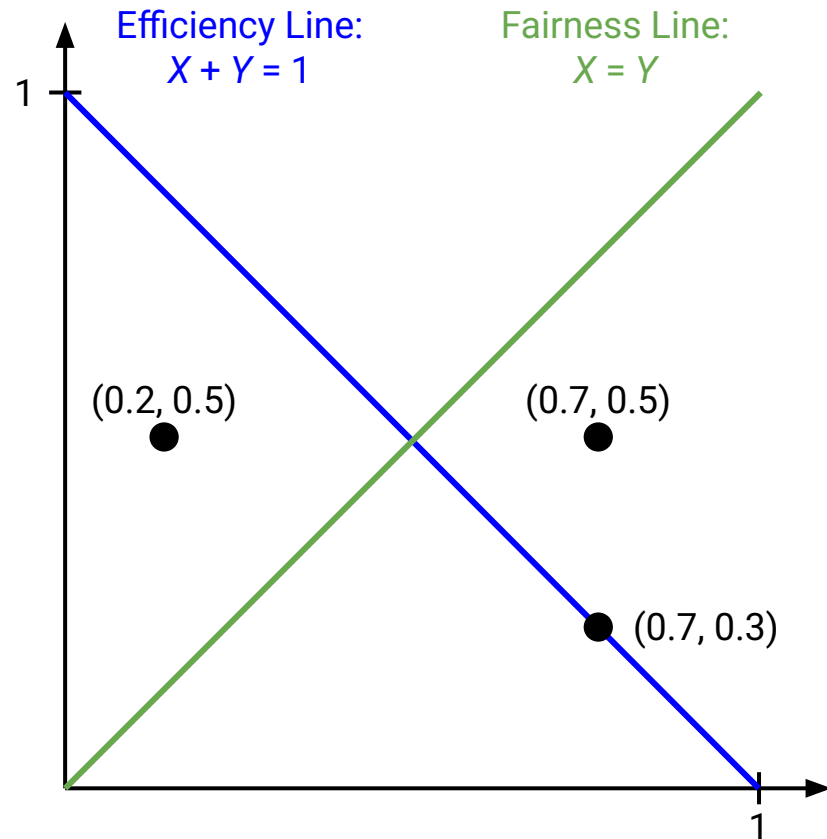
$(0.7, 0.5)$ is inefficient (above blue line).

1.2 bandwidth used (capacity is $C = 1$).

$(0.7, 0.3)$ is efficient, but unfair.

On blue line, but not on green line.

1 bandwidth used, but, $0.7 \neq 0.3$.



Additive Adjustments on Graph

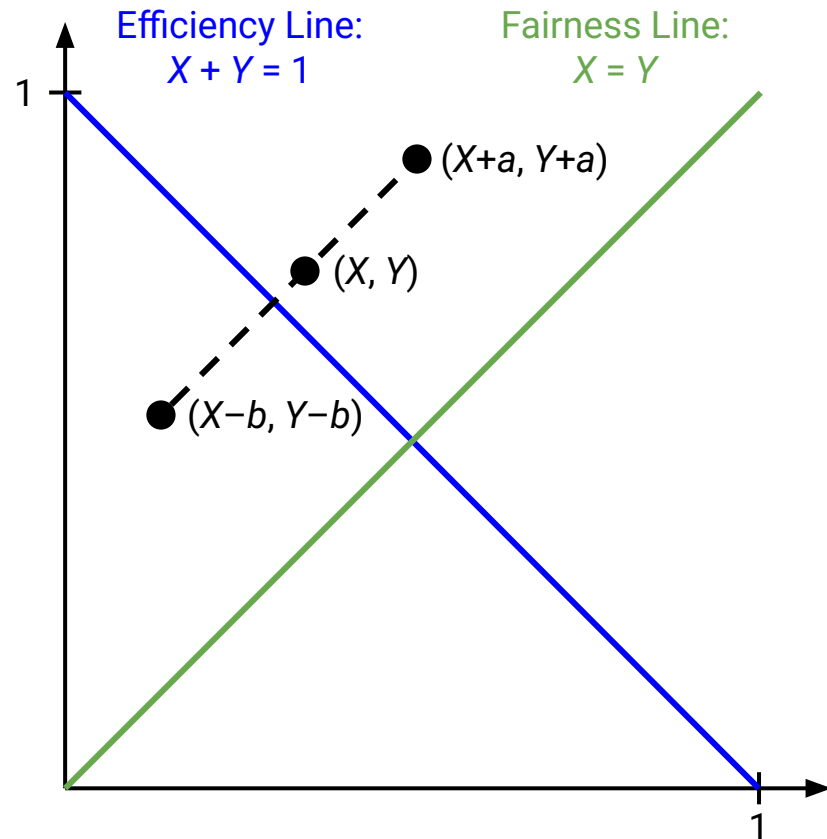
Recall: Both flows *independently* running *identical* algorithms.

Suppose current allocation is (X, Y) .

If both increase additively: $(X + a, Y + a)$.

If both decrease additively: $(X - b, Y - b)$.

Notice: Allocations always stay on a line with slope = 1.



Multiplicative Adjustments on Graph

Recall: Both flows *independently* running *identical* algorithms.

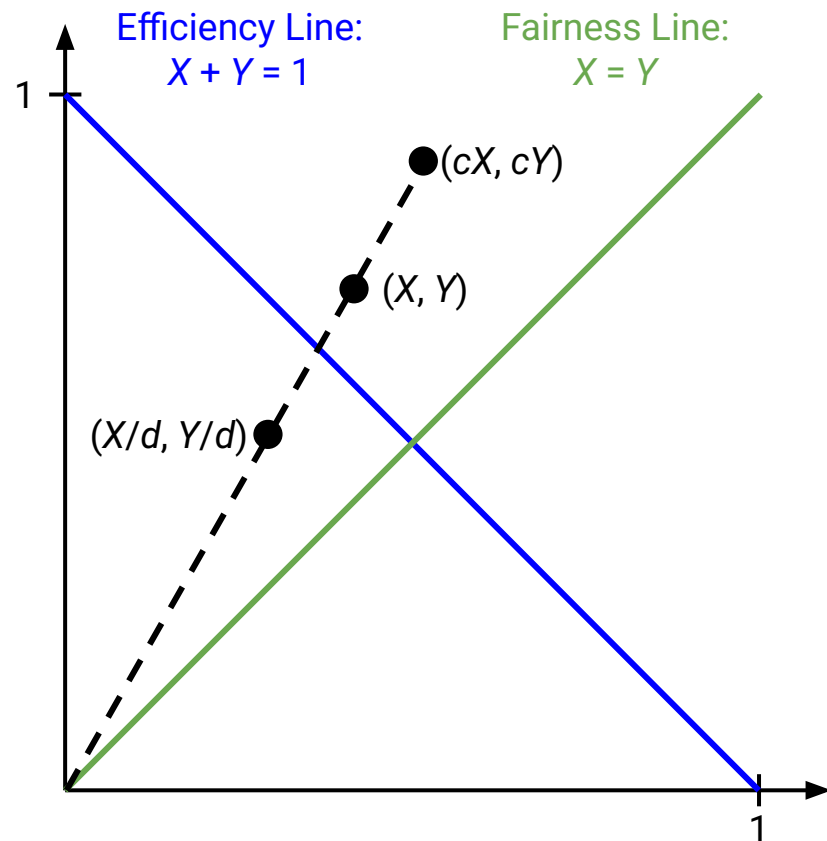
Suppose current allocation is (X, Y) .

If both increase multiplicatively: (cX, cY) .

If both decrease multiplicatively: $(X/d, Y/d)$.

Notice: Allocations always stay on a line with slope $= Y/X$, going through the origin.

$$\frac{\text{Rise}}{\text{Run}} = \frac{cY - Y}{cX - X} = \frac{Y(c - 1)}{X(c - 1)} = \frac{Y}{X}$$



AIAD (Additive Increase, Additive Decrease) Dynamics

Increase: +1

Decrease: -2

$C = 5$

Start:	$X = 1$	$Y = 3$	sum = 4, no congestion	$Y - X = 2$
--------	---------	---------	------------------------	-------------

First iteration:	$X = 2$	$Y = 4$	sum = 6, congested	$Y - X = 2$
------------------	---------	---------	--------------------	-------------

Second iteration:	$X = 0$	$Y = 2$	sum = 2, no congestion	$Y - X = 2$
-------------------	---------	---------	------------------------	-------------

Third iteration:	$X = 1$	$Y = 3$	back to where we started!	$Y - X = 2$
------------------	---------	---------	---------------------------	-------------

Notice: The *difference* in allocations stays the same!

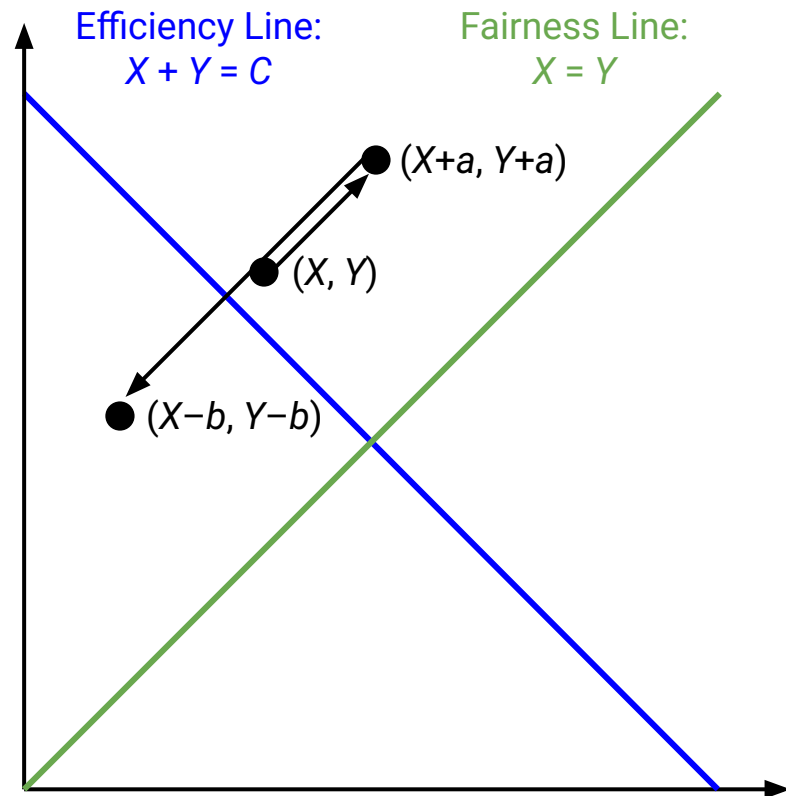
AIAD maintains the same unfairness across iterations.

AIAD (Additive Increase, Additive Decrease) Adjustments on Graph

Increase: $+a$ Decrease: $-b$

We keep moving along the line with slope 1.

AIAD does not converge to fairness.



MIMD (Multiplicative Increase, Multiplicative Decrease) Dynamics

Increase: $\times 2$

Decrease: $\div 4$

$C = 5$

Start:	$X = 0.5$	$Y = 1$	sum = 1.5, no congestion	$Y/X = 2$
--------	-----------	---------	--------------------------	-----------

First iteration:	$X = 1$	$Y = 2$	sum = 3, no congestion	$Y/X = 2$
------------------	---------	---------	------------------------	-----------

Second iteration:	$X = 2$	$Y = 4$	sum = 6, congestion	$Y/X = 2$
-------------------	---------	---------	---------------------	-----------

Third iteration:	$X = 0.5$	$Y = 1$	back to where we started!	$Y/X = 2$
------------------	-----------	---------	---------------------------	-----------

Notice: The *ratio* of allocations stays the same!

MIMD maintains the same unfairness across iterations.

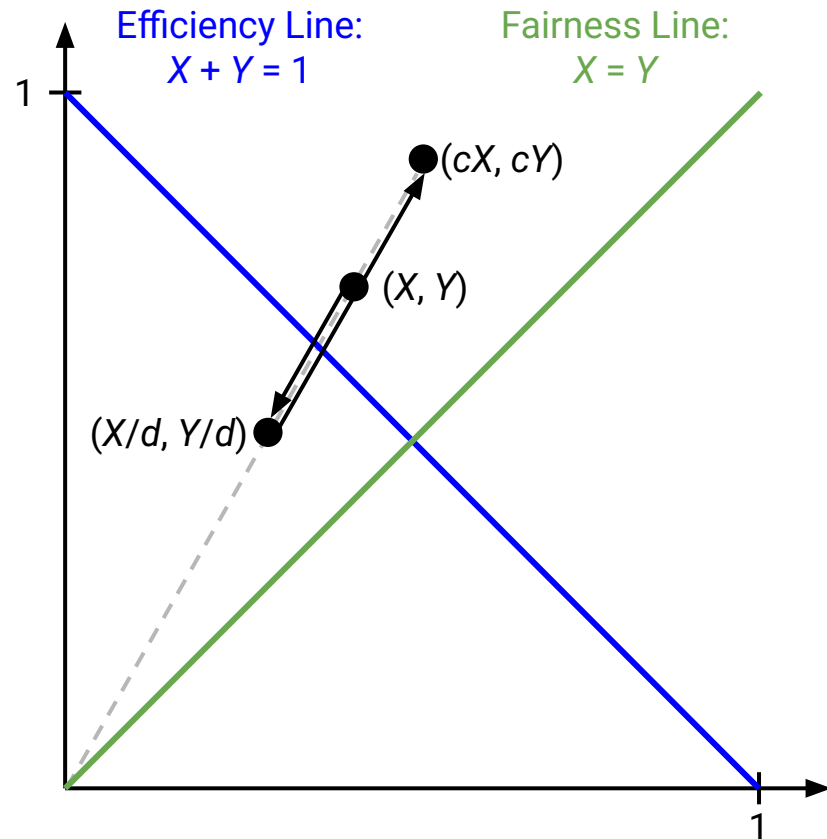
MIMD (Multiplicative Increase, Multiplicative Decrease) Adjustments on Graph

Increase: $\times c$

Decrease: $\div d$

We keep moving along the line connected to the origin.

MIMD does not converge to fairness.



MIAD (Multiplicative Increase, Additive Decrease) Dynamics

Increase: $\times 2$

Decrease: -1

$C = 5$

Start:	$X = 1$	$Y = 3$	sum = 4, no congestion	$Y - X = 2$
--------	---------	---------	------------------------	-------------

First iteration:	$X = 2$	$Y = 6$	sum = 8, congestion	$Y - X = 4$
------------------	---------	---------	---------------------	-------------

Second iteration:	$X = 1$	$Y = 5$	sum = 6, congestion	$Y - X = 4$
-------------------	---------	---------	---------------------	-------------

Third iteration:	$X = 0$	$Y = 4$	sum = 4, no congestion	$Y - X = 4$
------------------	---------	---------	------------------------	-------------

Fourth iteration:	$X = 0$	$Y = 8$	X stuck at 0 forever!	$Y - X = 8$
-------------------	---------	---------	-------------------------	-------------

The gap in allocation doubles when we increase. before: $Y - X$. after: $2Y - 2X = 2(Y - X)$

The gap in allocation stays the same when we decrease. (Same as AIAD.)

MIAD is *maximally unfair*. The gap increases until one person has all the bandwidth!

AIMD (Additive Increase, Multiplicative Decrease) Dynamics

Increase: +1	Decrease: ÷2	C = 5		
Start:	X = 1	Y = 2	sum = 3, ok	$Y - X = 1$
First iteration:	X = 2	Y = 3	sum = 5, ok	$Y - X = 1$
Second iteration:	X = 3	Y = 4	sum = 7, congestion	$Y - X = 1$
Third iteration:	X = 1.5	Y = 2	sum = 3.5, ok	$Y - X = 0.5$
Fourth iteration:	X = 2.5	Y = 3	sum = 5.5, congestion	$Y - X = 0.5$
Fifth iteration:	X = 1.25	Y = 1.5	sum = 2.75, ok	$Y - X = 0.25$
Sixth iteration:	X = 2.25	Y = 2.5	sum = 4.75, ok	$Y - X = 0.25$
Seventh iteration:	X = 3.25	Y = 3.5	sum = 6.75, congestion	$Y - X = 0.25$
Eighth iteration:	X = 1.625	Y = 1.75	sum = 3.375, ok	$Y - X = 0.125$
Ninth iteration:	X = 2.625	Y = 2.75	approaching 2.5 each!	$Y - X = 0.125$

AIMD (Additive Increase, Multiplicative Decrease) Dynamics

The gap in allocation stays the same when we increase.

The gap in allocation halves when we decrease.

- Before: $Y - X$
- After: $(0.5 Y - 0.5 X) = 0.5(Y - X)$

The gap approaches 0, so AIMD approaches fairness!

$$Y - X = 1$$

$$Y - X = 1$$

$$Y - X = 1$$

$$Y - X = 0.5$$

$$Y - X = 0.5$$

$$Y - X = 0.25$$

$$Y - X = 0.25$$

$$Y - X = 0.25$$

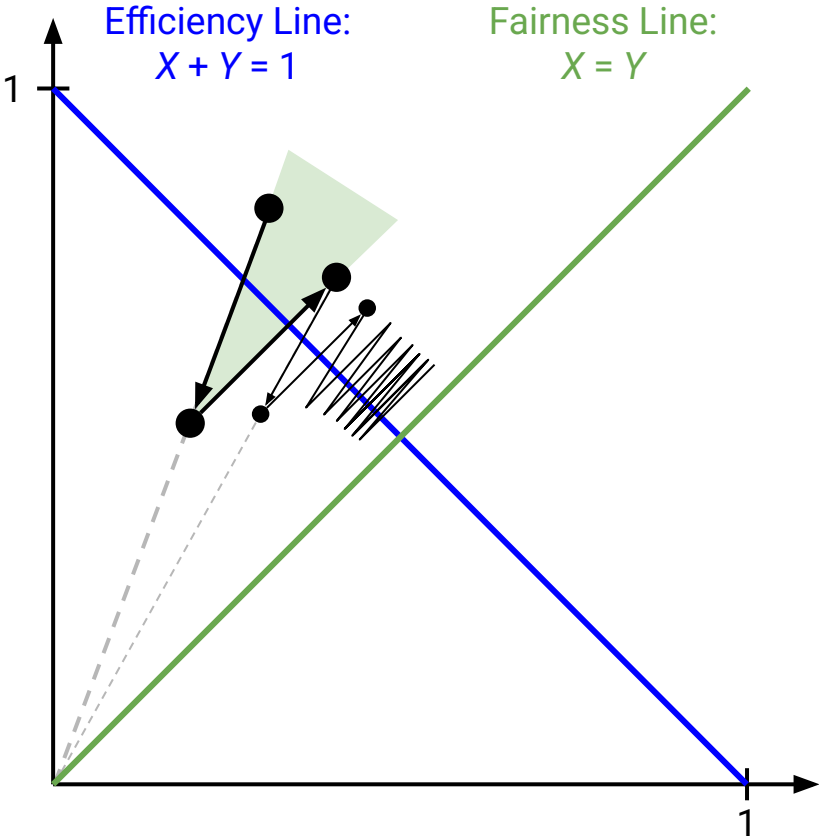
$$Y - X = 0.125$$

$$Y - X = 0.125$$

AIMD (Additive Increase, Multiplicative Decrease) Adjustments on Graph

Increase: $+a$ Decrease: $\div d$

AIMD converges to fairness!



Why AIMD? Intuition

Non-obvious fact: Of the four options, AIMD is the only one that ensures fairness.

Hopefully more obvious now!



AIMD embodies gentle increase, rapid decrease.

Out of the four options:

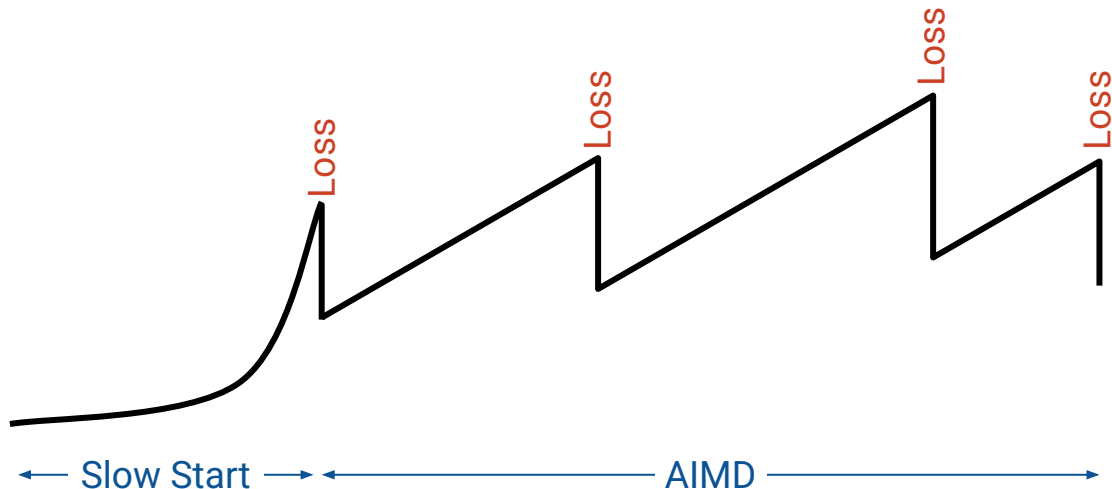
- AIAD and MIMD retain unfairness.
- MIAD approaches maximum unfairness (one person with all the bandwidth).
- AIMD approaches fairness across iterations.

Host-Based Dynamic Adjustment: Algorithm Sketch

Every host *independently* runs the same algorithm:

1. Pick an initial rate R . (*Slow start.*)
2. Try sending at rate R for some period of time.
 - Did I experience congestion in this time period? (*Detect packet loss.*)
 - If no, increase R . (*Additive increase.*)
 - If yes, reduce R . (*Multiplicative decrease.*)
 - Repeat.

This algorithm creates a sawtooth-shaped graph as the rate is adjusted.



Implementing Congestion Control with Event-Driven Updates

Lecture 13, CS 168, Spring 2025

Congestion Control Principles

- Why do we need it?
- Why is it hard?
- Design Goals

Dynamic Adjustment

- Algorithm Sketch
- **Reacting to Congestion (AIMD)**

Review: Setting Window Size

The sender maintains a window of W packets in flight.

Pick window size W to balance three goals:

1. Take advantage of network capacity ("fill the pipe").
 - Not actually used (impractical to calculate, and never the minimum).
2. Don't overload the recipient.
 - Receiver window (RWND): Recipient reports how much space it has left.
3. Don't overload links.
 - Congestion window (CWND): Sender runs a congestion control algorithm.

W is the minimum of RWND and CWND.

- For today, we'll assume $RWND > CWND$, so window is determined by CWND.
- Often true in practice too.

Review: Measuring Packets vs. Bytes

Real TCP measures data in bytes.

- Thus, real TCP maintains CWND in bytes.

In this lecture, we'll measure CWND in packets for convenience.

- To convert to bytes, recall: 1 packet = *MSS* bytes.
- *MSS* = Maximum Segment Size.

Congestion control is used to set the window size (CWND).

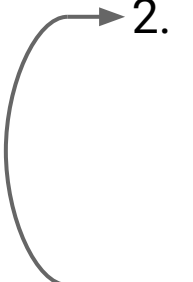
We talked about congestion control as adjusting the rate.

Possibly confusing fact: Window size and rate are directly correlated.

- Recall: $\text{Rate} = \text{CWND} / \text{RTT}$.
- Key thing to remember: Larger window = higher rate.

Host-Based Dynamic Adjustment: Algorithm Sketch

Every host *independently* runs the same algorithm:

1. Pick an initial rate R . (*Slow start.*)
 2. Try sending at rate R for some period of time. ???
 - Did I experience congestion in this time period? (*Detect packet loss.*)
 - If no, increase R . (*Additive increase.*)
 - If yes, reduce R . (*Multiplicative decrease.*)
 - Repeat.
- 

How long is "some period of time"?

- Ideally, each iteration lasts for one RTT.
- Problem: RTT is hard to measure, and changes dynamically.

Instead of updating after "some period of time," CWND updates are **event-driven**.

Sender adjusts CWND each time something interesting happens in TCP.

- We get an ack for new data.
 - This means there was no congestion.
 - Increase CWND (using slow-start, or additive increase in AIMD).
- We get 3 duplicate acks and declare a packet lost.
 - Probably an isolated loss, since we're still getting subsequent acks.
 - Decrease CWND (multiplicative decrease in AIMD).
- The timer expires and we declare a packet lost.
 - We probably lost several packets – didn't get any duplicate acks. Bad news!
 - Abandon current rate and start over from slow-start.

Event-Driven Slow Start

Conceptual slow start: Double CWND after each RTT.

Event-driven slow start: Increase CWND by 1 after each ack.

Intuition:

t = 0 RTTs:		CWND = 1	Send 1 packet
t = 1 RTT:	get 1 ack	CWND = 2	Send 2 packets
t = 2 RTTs:	get 2 acks	CWND = 4	Send 4 packets
t = 3 RTTs:	get 4 acks	CWND = 8	Send 8 packets
t = 4 RTTs:	get 8 acks	CWND = 16	Send 16 packets

Event-Driven Slow Start

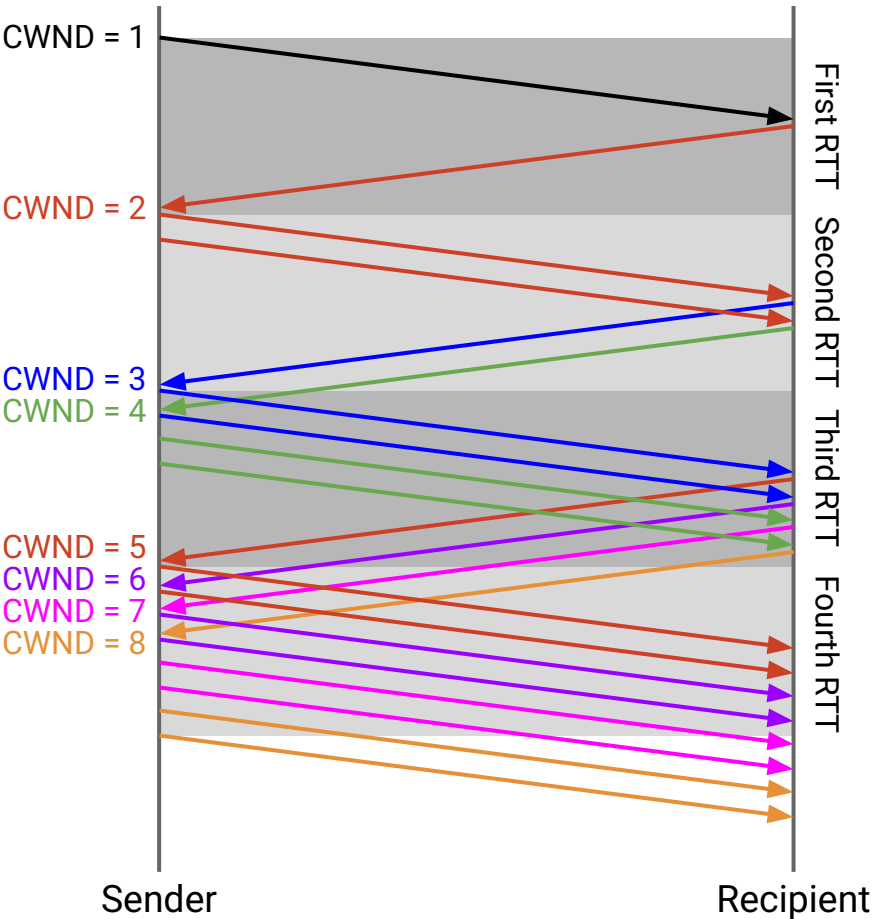
Conceptual slow start: Double CWND after each RTT.

Event-driven slow start: Increase CWND by 1 after each ack.

Intuition:

- Each time we get an ack, we can send one more packet (sliding window).
- But, we can also increase CWND by 1 and send another packet.
- In total: Each time we get an ack, we can send out 2 packets.
- Example: In one RTT, if we get 8 acks, we can send out 16 packets.

Event-Driven Slow Start



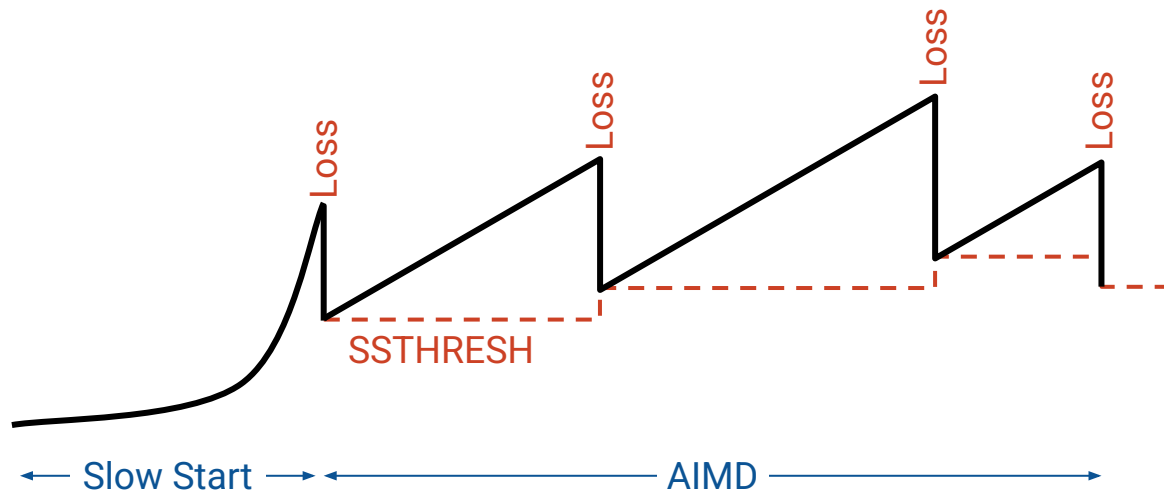
Event-Driven Slow Start: SSTHRESH

Increase CWND by 1 after each ack.

Eventually, we encounter a packet loss.

- Set $\text{CWND} = \text{CWND}/2$.
- Set $\text{SSTHRESH} = \text{CWND}/2$.

SSTHRESH helps us remember the last safe rate.



Event-Driven AIMD Adjustments: Additive Increase

Conceptual AIMD increase: Add 1 to CWND after each RTT.

Event-driven AIMD increase: Increase CWND by $1/\text{CWND}$ after each ack.

Intuition:

- Increase CWND by a fraction of a packet on each ack.
- After one RTT's worth of packets, CWND will increase by 1.

Implementation:

- $$\text{CWND} \leftarrow \text{CWND} + \frac{1}{\text{CWND}}$$

Note: Technically, increase on each ack is different, but approximation is close enough.

- In bytes:

$$\text{CWND} \leftarrow \text{CWND} + \frac{\text{MSS}}{\text{CWND}} \times \text{MSS}$$

($1/\text{CWND}$ in packets) fraction rewritten in bytes by multiplying top and bottom by packet size: (MSS/CWND in bytes)

Total increase after each RTT should be MSS bytes, not 1 packet.

First update: $3.00 + (1/3.00) = 3.33$

Second update: $3.33 + (1/3.33) = 3.63$

Third update: $3.63 + (1/3.63) = 3.93 \approx 4$

Event-Driven AIMD Adjustments: Additive Increase

Event-driven AIMD increase: Increase CWND by a fraction after each ack.

After one RTT (aka one window) of packets, CWND increases by 1.

	Measured in packets:	Measured in bytes: (<i>MSS</i> = 50 bytes)
Old CWND:	3 packets	150 bytes (3 packets)
Fraction of increase per ack:	$1/CWND = 1/3$	$MSS/CWND = 50/150$
Total increase after 1 RTT:	+1 packet	<i>MSS</i> = +50 bytes
Increase after each ack:	+1 packet $\times 1/3 = +1/3$	+50 bytes $\times 50/150 = +50/3$
Increase after 3 acks:	$3 + (1/3) + (1/3) + (1/3)$ = 4 packets	$150 + (50/3) + (50/3) + (50/3)$ = 200 bytes

Event-Driven AIMD Adjustments: Multiplicative Decrease

Conceptual AIMD decrease: Divide CWND in half when loss detected.

Event-driven AIMD decrease: Divide CWND in half when 3 *duplicate acks* detected.

- Set $CWND \leftarrow CWND / 2$
- Set $SSTHRESH \leftarrow CWND / 2$. (*Remember the last safe rate.*)

CWND should never decrease below 1 packet (MSS bytes).

- If dividing by 2 causes $CWND$ to fall below 1, it's okay to round up to 1 packet.

Event-Driven Adjustments: Timeout

When a packet is lost from timeout, abandon current rate and restart from slow-start.

- Rationale: We didn't get any duplicate acks before timing out, so we probably lost multiple packets.
- The current *CWND* might be way off. Safest to rediscover a good rate from scratch.
- This design decision errs on the side of caution.

Implementation: When timeout occurs:

- Set $SSTHRESH \leftarrow CWND/2$. (*Remember the last safe rate.*)
- Set $CWND \leftarrow 1$ packet, and re-enter slow start mode.

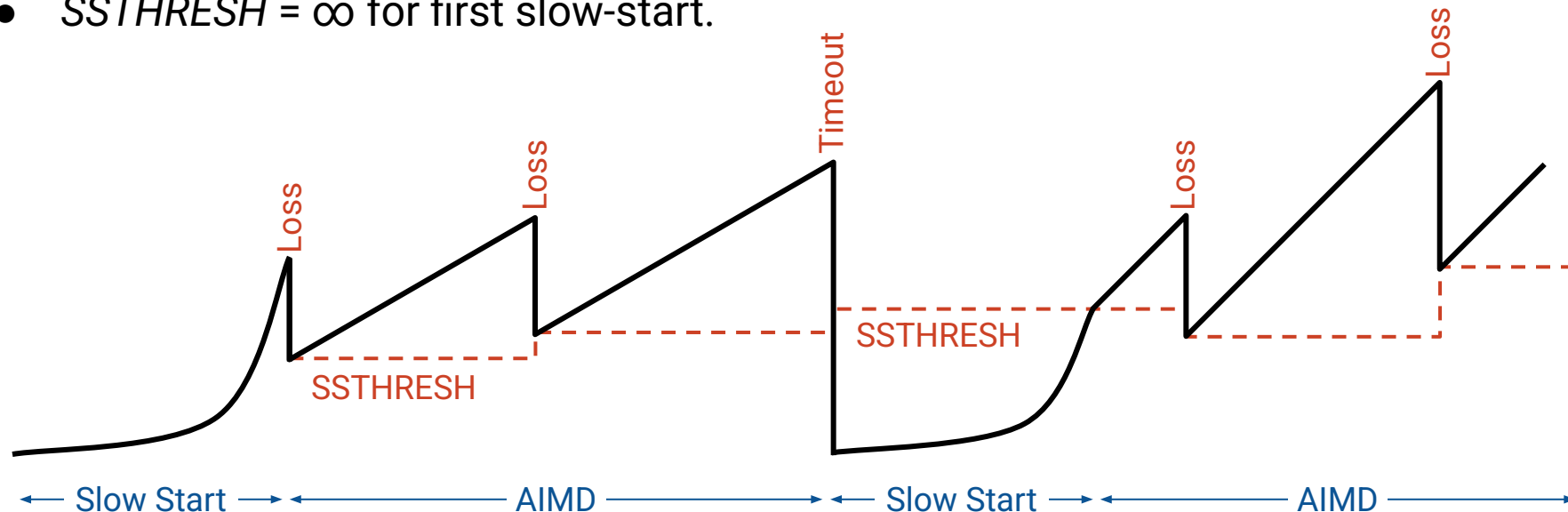
Event-Driven Updates: SSTHRESH

SSTHRESH helps us remember the last safe rate.

- When slow-start detects loss: Update $SSTHRESH \leftarrow CWND/2$.
- When a packet times out in AIMD: Update $SSTHRESH \leftarrow CWND/2$.

During slow start: If $CWND$ exceeds $SSTHRESH$, switch to additive increases.

- $SSTHRESH = \infty$ for first slow-start.



Summary: TCP Congestion Control

Detecting congestion:

- Loss-based. (Delay-based is the alternative.)

Discovering an initial rate:

- Slow-start. Double rate until loss or safe rate (*SSTHRESH*) exceeded.
- Event-driven: Add 1 to *CWND* on each ack.

Adapting rate to congestion:

- AIMD (additive increase, multiplicative decrease) approaches fairness.
- Event-driven:
 - Add $1/CWND$ on each ack. After one RTT, *CWND* will have increased by 1.
 - Divide *CWND* in half when there are 3 duplicate acks.
- When a packet is lost on timeout, restart from slow-start.