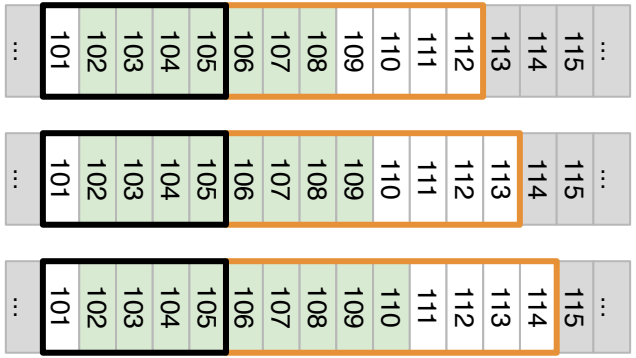


Lecture 13 (Transport 4)

Congestion Control, Part 2

CS 168, Fall 2026 @ UC Berkeley

Slides credit: Sylvia Ratnasamy, Rob Shakir, Peyrin Kao, Ion Stoica

Recall: Congestion Control Problem Statement

Congestion control: Ensuring senders send at an optimal rate. That is, a rate that:

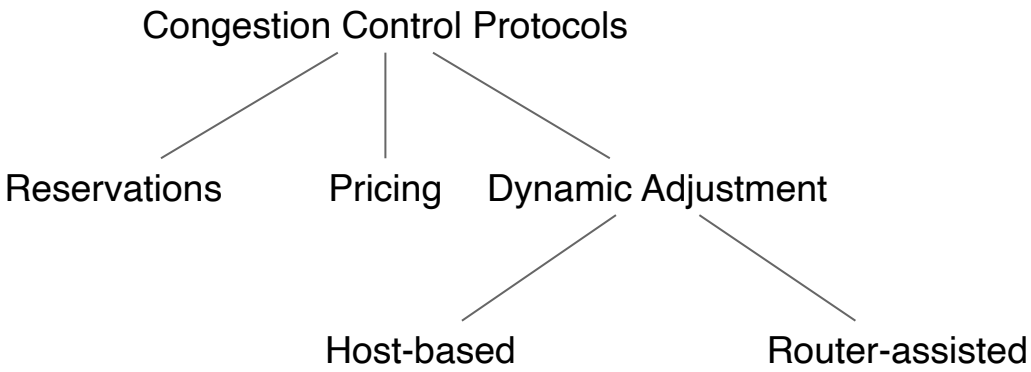
- Does not overwhelm the network
- Yet uses the network efficiently
- And uses the network fairly

Challenging problem since a host's optimal sending rate depends on several factors:

- E.g., Destination, path to the destination, other connections sharing links, etc.

No end host has a global view of the network

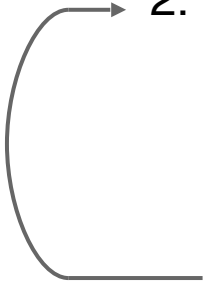
Recall: Possible Approaches



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.

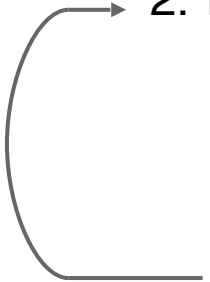


Components of a solution:

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

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?
   - If no, increase *R*.
   - If yes, reduce *R*.
   - Repeat.



Components of a solution:

- **Discovering an initial rate: Slow start (i.e., double CWND every RTT)**
- Detecting congestion.
- Reacting to congestion (or lack thereof) by increasing/decreasing rate.

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.

Components of a solution:

- Discovering an initial rate:
- Detecting congestion:
- **Reacting to congestion: AIMD (best to avoid congestion, only fair approach)**

Implementing Congestion Control with Event-driven Updates

Lecture 15, CS 168, Spring 2026

Congestion Control Implementation

- **Event-driven updates**
- Fast Recovery
- State Machine and Variants

TCP Throughput Model

Congestion Control Issues

Router-Assisted Congestion Control

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.

Event-Driven Updates

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

Sender adjusts CWND each time something interesting happens in TCP.

Three interesting events that sender reacts to:

- We get an **ack for new data**.
  - This means there was no congestion.
- We get 3 duplicate acks and declare a packet lost.
  - Probably an isolated loss, since we're still getting subsequent acks.
- The timer expires and we declare a packet lost.
  - We probably lost several packets – didn't get any duplicate acks. Bad news!

How to react to these events:

- 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.**

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

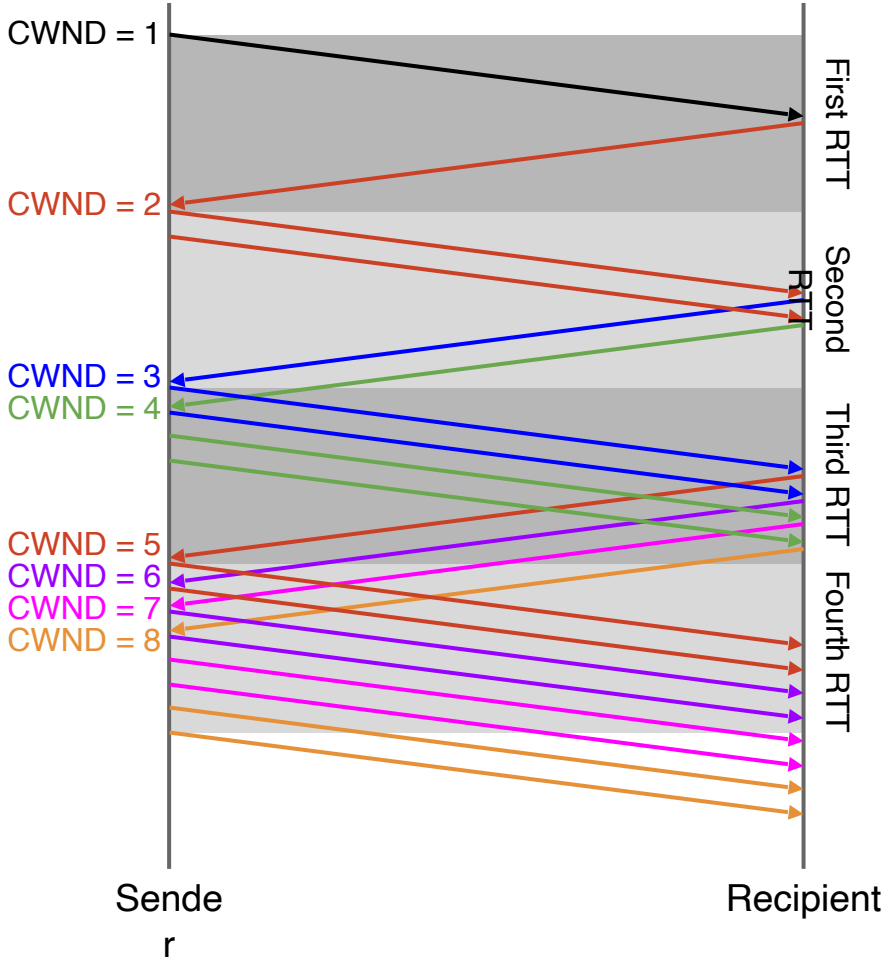
Conceptual slow start: Double CWND after each RTT.

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 AIMD Adjustments: Additive Increase

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

Event-driven AIMD increase: Increase CWND by 1/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:

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

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

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 ≈ 4

- In bytes:
  $CWND \leftarrow CWND + \frac{MSS}{CWND} \times MSS$

(1/*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.

Event-Driven AIMD Adjustments: Multiplicative Decrease

So far, we've looked at increasing the CWND.

- But if we keep increase, we will eventually exceed the rate and detect a loss.
- How do we decrease upon detecting a loss?

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* ← *CWND* / 2

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 *CWND* ← 1 packet, and re-enter slow start mode.

Event-Driven Slow Start: SSTHRESH

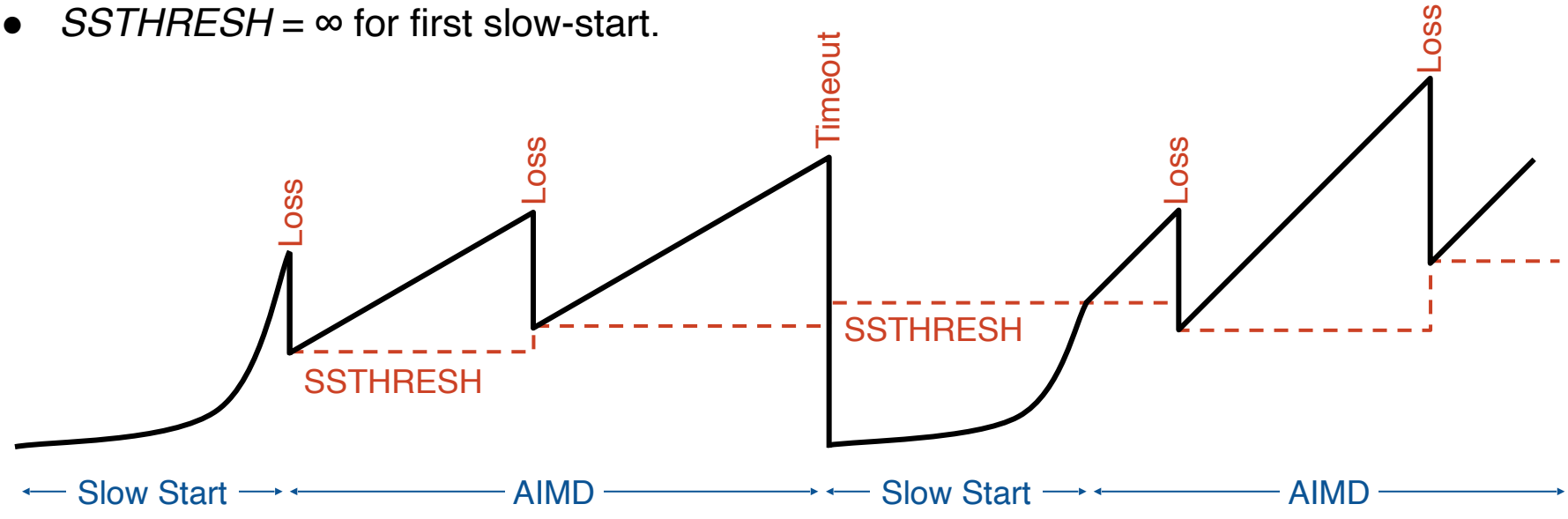
Denotes the boundary between slow start and congestion avoidance.

Why do we need it?

- Ramping up from a CWND of 1 after a timeout is too slow.
- We can remember a “safe-ish” rate and increase the window exponentially until that point.
- Heuristic: SSTHRESH = CWND (at the time of loss)/2.

SSTHRESH helps us remember a likely safe rate.

- SSTHRESH = CWND/2 upon loss. Leads to a safe rate since:
  - During slow start since we would only double if the previous rate was safe.
  - During congestion avoidance, CWND/2 is safe as long > 2
- If *CWND* exceeds *SSTHRESH*, switch to additive increases.
- *SSTHRESH* = ∞ 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.

Fast Recovery

Lecture 15, CS 168, Spring 2026

Congestion Control Implementation

- **Fast Recovery**
- State Machine and Variants

TCP Throughput Model

Congestion Control Issues

Router-Assisted Congestion Control

Fast Recovery

Problem: Additive increases are too slow to recover from isolated loss.

This last feature is an optimization to improve performance.

- Bit of a hack, but it's effective.

Fast Recovery: Setting the Stage

Consider sending 10 packets.

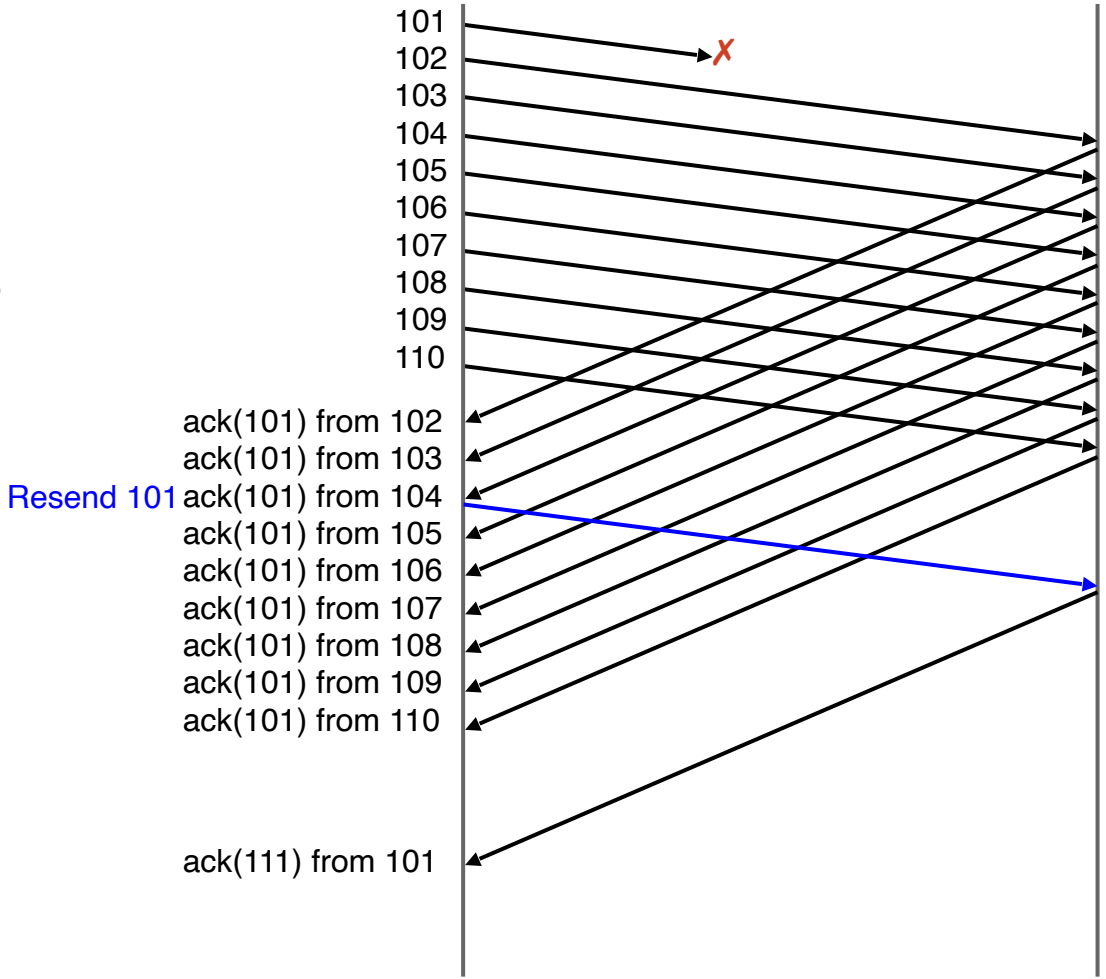
The first one (101) is dropped.

Acks for 102–110 all say ack(101) because packet 101 is still missing.

After the third duplicate ack(101), we resend 101.

Eventually, ack for the re-sent 101 says ack(111) because packets 102–110 were all received earlier.

What does *CWND* look like during this process?

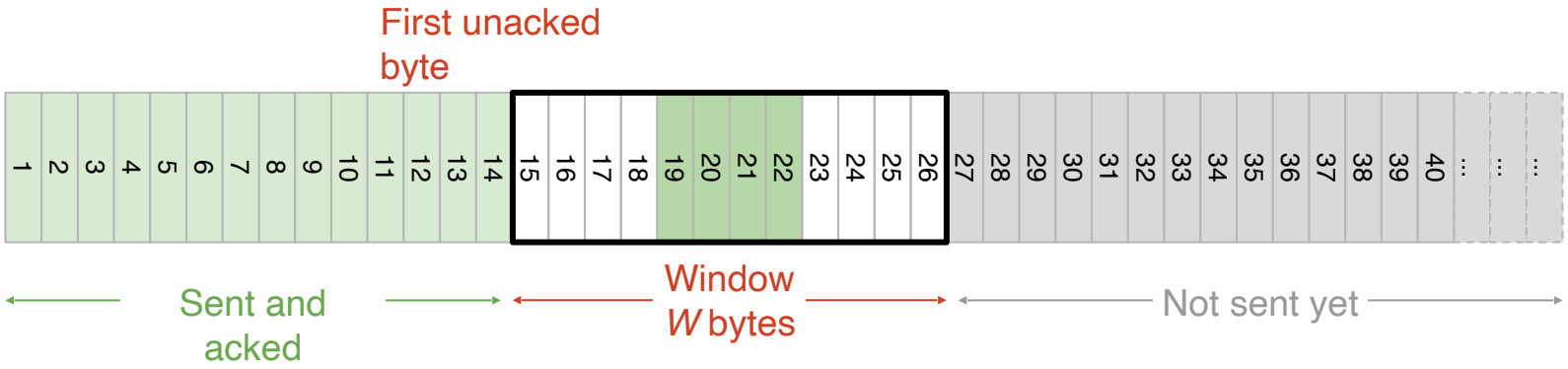


The window is a range of *W* contiguous bytes, starting at the first unacked byte.

- Only these *W* bytes are allowed to be in flight.

The window slides right if and only if its *leftmost* bytes are acked.

- Acking non-leftmost bytes in the window (e.g. 19–22) does not slide the window.
- The window is determined by the first unacked byte (e.g. still 15).



Fast Recovery: Windows Without Fast Recovery

After sending 10 packets: *CWND* = 10. 101–110 can be in flight.

After ack(101) from 102: Can't send anything new.

After ack(101) from 103: Can't send anything new.

After ack(101) from 104: Can't send anything new.

3 duplicate acks detected. *CWND* = 5.

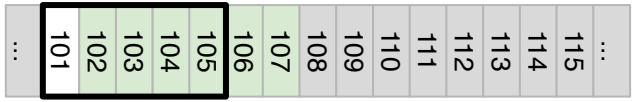
After ack(101) from 105: Can't send anything new.

Note: We're marking acked packets in green for convenience, but the sender cannot deduce from duplicate acks that these exact packets were received.

Fast Recovery: Windows Without Fast Recovery

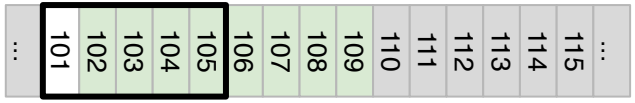
After ack(101) from 106: Can't send anything new.

After ack(101) from 107: Can't send anything new.

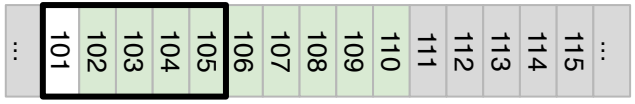


After ack(101) from 108: Can't send anything new.

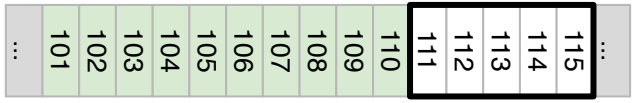
After ack(101) from 109: Can't send anything new.



After ack(101) from 110: Can't send anything new.



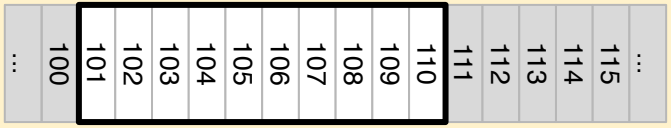
After ack(111) from resent 101:
Window slides to first unacked packet (111).
Sender is now able to send 111–115.



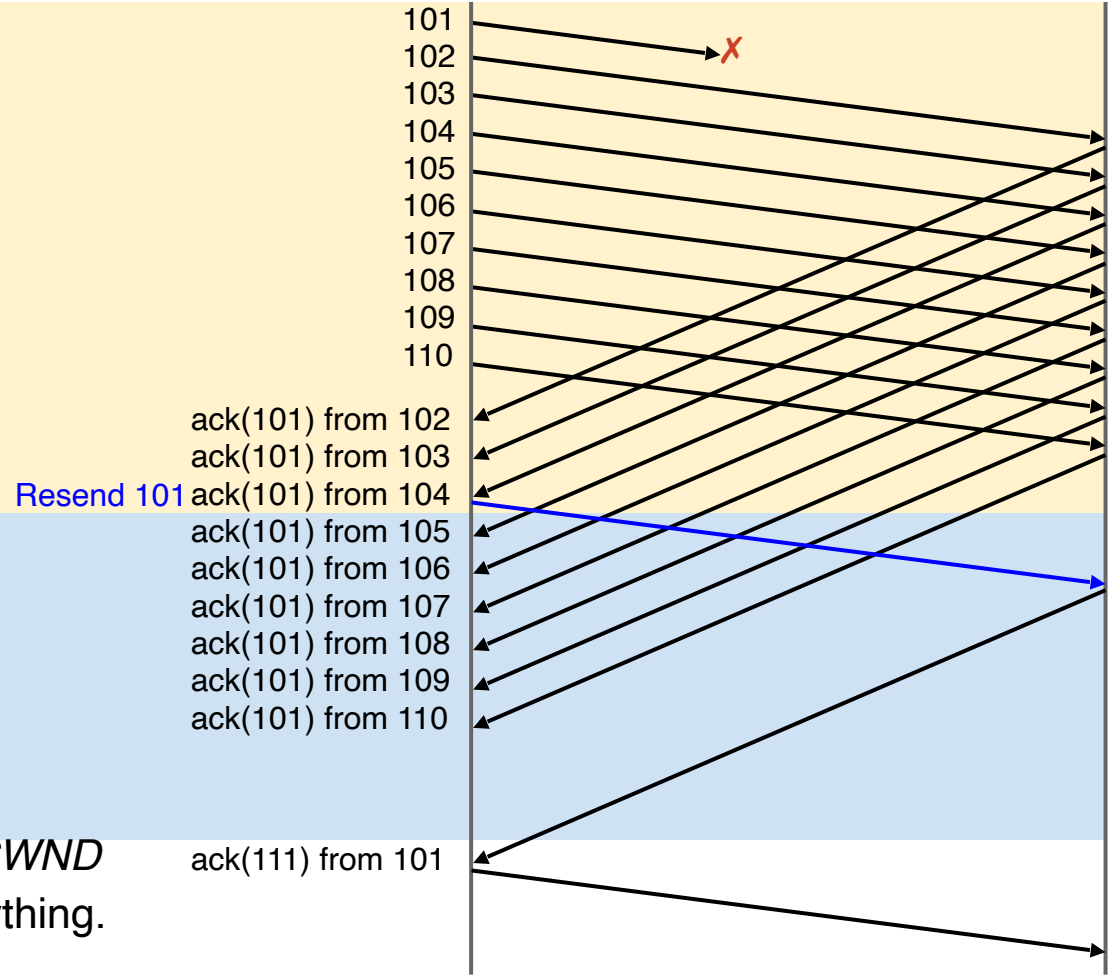
Problem: After we decreased *CWND*, we had to wait a long time before sending again.

Fast Recovery: Windows Without Fast Recovery

CWND = 10. 101–110 can be in flight.



CWND = 5. 101–105 can be in flight.



Problem: In the entire blue area, after *CWND* decreased, the sender isn't sending anything.

Fast Recovery: Windows Without Fast Recovery

CWND = 5. 101–105 can be in flight.

ack(101) from 102
ack(101) from 103
Resend 101 ack(101) from 104
ack(101) from 105
ack(101) from 106
ack(101) from 107
ack(101) from 108
ack(101) from 109
ack(101) from 110

ack(111) from 101

CWND = 5. 111–115 can be in flight.

111
112
113
114
115

Secondary problem: Sender is forced to send a large burst at once, and so has to wait a full RTT for ack(112) to arrive before sending 116+.

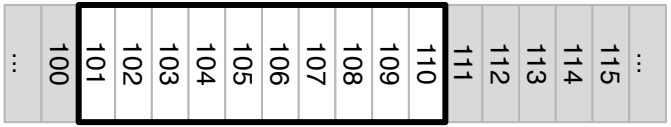
The problem, restated again:

- After *CWND* decreased, the window was too small for us to send packets after 110.
- The sender must wait (sending nothing) until the re-sent 101 is acked.

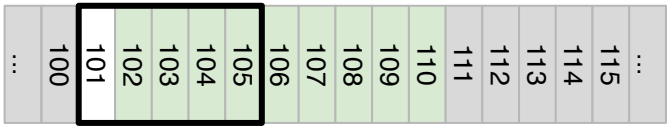
The secondary problem, restated again:

- When 101 is acked, the window jumps forward, and 111–115 are all sent at once.
- The sender must wait again (idling) until ack(111) before sending more.

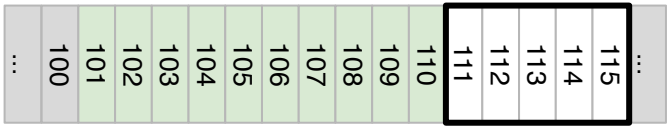
CWND = 10. 101–110 can be in flight.



CWND = 5. 101–105 can be in flight.



CWND = 5. 111–115 can be in flight.



The sender can deduce from the duplicate acks that fewer packets are in flight:

- Initially: 10 packets in flight.
- After ack(101) from 102: 9 packets in flight.
- After ack(101) from 103: 8 packets in flight.
- After ack(101) from 104: 7 packets in flight.
- After ack(101) from 105: 6 packets in flight.
- After ack(101) from 106: 5 packets in flight.
- After ack(101) from 107: 4 packets in flight.
- After ack(101) from 108: 3 packets in flight.
- After ack(101) from 109: 2 packets in flight.
- After ack(101) from 110: 1 packet in flight. (*The packet still in flight is the re-sent 101.*)

The sender can deduce from the duplicate acks that fewer packets are in flight.

- Even though there are eventually <5 packets left in flight, the sliding window is stopping us from sending more.

Key idea:

- Grant the sender temporary "credit" for each duplicate ack.
- When a duplicate ack arrives, we know that one fewer packet is in flight.
- *Artificially extend* the window to let the sender send one more packet.

Reminder: We had seen this in action before!

1 2 3 4 5 6 7 8 9 10 11 12 13 14

duplicate acks = 6

- (for packet 4) ACK 3 → Send 9
- (for packet 6) ACK 3 → Send 10
- (for packet 7) ACK 3 → Send 11
  - ACK 3 duplicated 3 times. **Resend 3.**
- (for packet 8) ACK 3 → Send 12
- (for packet 9) ACK 3 → Send 13
- (for packet 10) ACK 3 → Send 14
  - ACK 3 duplicated 6 times.
    Resend 3? Resend 4? Resend 5?

Assumptions:

- $k = 3$
  (*lost after 3 later acks*)
- $W = 6$
  (*6 packets in flight*)
- red = in flight
- green = acked
- Packets 3 and 5 are lost (but sender doesn't know).
- 1 and 2 already acked.

Fast Recovery: Windows with Fast Recovery

After sending 10 packets: *CWND* = 10.
101–110 can be in flight.

After ack(101) from 102: Can't send anything new.

After ack(101) from 103: Can't send anything new.

After ack(101) from 104: Can't send anything new.

3 duplicate acks detected. *CWND* = 5.
Extend window to account for the 3 acks: *CWND* = 5 + 3 = 8.

After ack(101) from 105: Extend window to *CWND* = 9.
Can't send anything new.

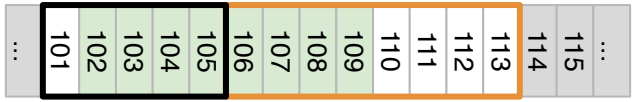
Fast Recovery: Windows with Fast Recovery

After ack(101) from 106: Extend window to $C = 10$.
Can't send anything new.

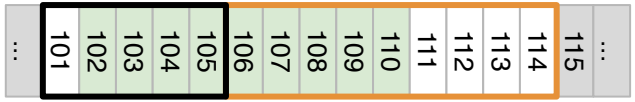
After ack(101) from 107: Extend window to $C = 11$.
We can send 111!

After ack(101) from 108: Extend window to $C = 12$.
We can send 112!

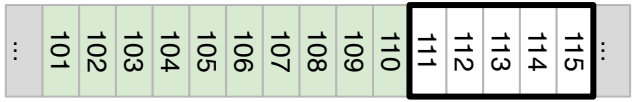
After ack(101) from 109: Extend window to $C = 13$.
We can send 113!



After ack(101) from 110: Extend window to $C = 14$.
We can send 114!



After ack(111) from resent 101: Back to normal. $C = 5$.
We can send 115.



Remember: The sender does *not* know exactly which packets were acked.

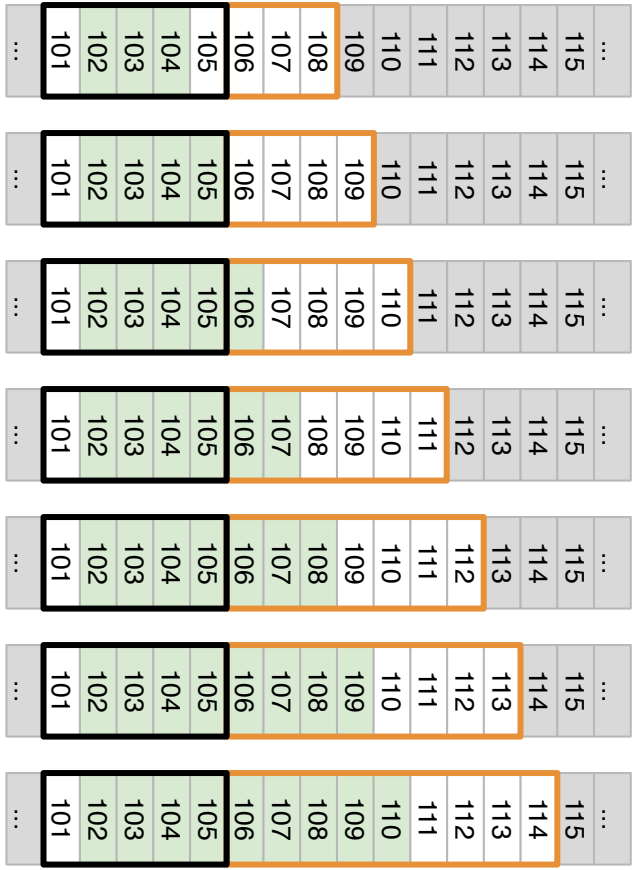
- The sender only sees duplicate acks.

The sender *does* know how many packets were acked.

- The sender can count duplicate acks.

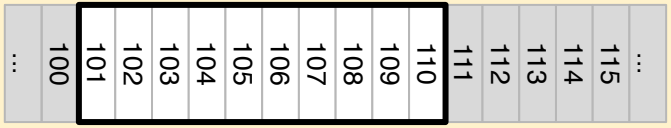
The artificially extended window ensures that the total number of packets in flight is still 5 (the new *CWND*).

- Size of extended window
  – number of duplicate acks
  = 5 (the new *CWND*).

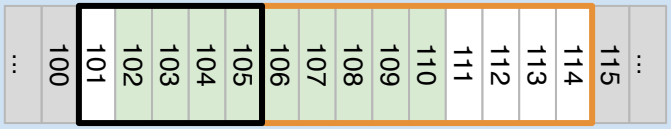


Fast Recovery: Windows Without Fast Recovery

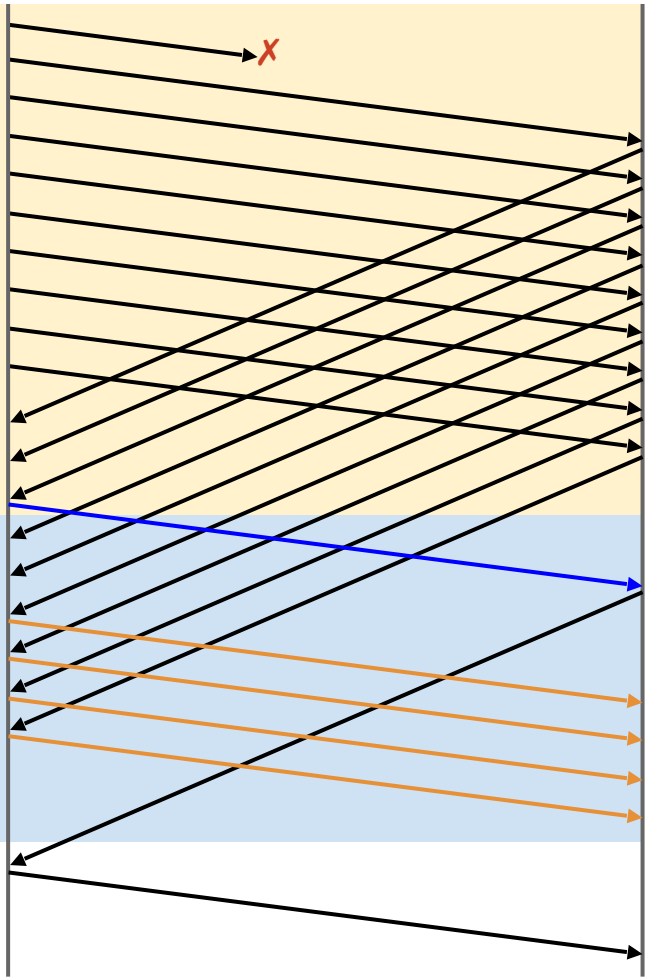
CWND = 10. 101–110 can be in flight.



CWND = 14. 101–114 can be in flight.



The sender is now still sending in the blue area!



Fast Recovery: Windows Without Fast Recovery

CWND = 5. 101–105 can be in flight.

Resend 101

ack(101) from 102
ack(101) from 103
ack(101) from 104
ack(101) from 105
ack(101) from 106
ack(101) from 107
ack(101) from 108
ack(101) from 109
ack(101) from 110

ack(111) from 101

CWND = 5. 111–115 can be in flight.

Because we sent 111–114 during the blue area, when the new ack(111) arrives, we only have to send 115.
Also, ack(112) will arrive sooner so we can keep sending.

Fast Recovery: Implementation

Conceptually: When a duplicate ack arrives, *artificially extend* the window to let the sender send one more packet.

Implementation:

- When we receive 3 duplicate acks:
  - *SSTHRESH* ← *CWND*/2
  - *CWND* = *CWND*/2 + 3 (*artificially extend for the 3 duplicate acks*)
- While in fast recovery mode, when we receive a duplicate ack:
  - *CWND* = *CWND* + 1 (*artificially extend for each duplicate ack*)
- Exit fast recovery when we receive a new, non-duplicate ack:
  - *CWND* = *SSTHRESH* (*back to 0.5 × rate when the loss happened*)

State Machine and Variants

Lecture 15, CS 168, Spring 2026

Congestion Control Implementation

- Fast Recovery
- **State Machine and Variants**

TCP Throughput Model

Congestion Control Issues

Router-Assisted Congestion Control

The sender maintains 5 values:

- *dupACKcount* for detecting loss. Initialized to 0.
- *Timer* for detecting loss.
- *RWND* for flow control.
- *CWND* for congestion control. Initialized to 1 packet.
- *SSTHRESH* to remember latest safe rate. Initialized to ∞.

The recipient maintains a buffer of out-of-order packets.

The sender responds to 3 events:

- Ack for new data (not previously acked).
- Duplicate ack.
- Timeout.

The recipient responds to receiving a packet: Reply with an ack and a *RWND* value.

Events at sender: **new ack**, duplicate ack, timeout.

When we receive an ack for new data (not previously acked):

- If in slow-start mode:
  - *CWND* ← *CWND* + 1 packet (*so CWND doubles per RTT*)
- If in fast recovery mode:
  - *CWND* ← *SSTHRESH* (*so we leave fast recovery*)
- If in congestion avoidance mode:
  - *CWND* ← *CWND* + 1/*CWND* (*so CWND increases by 1 per RTT*)
- Reset timer.
- Reset duplicate ack count.
- If window allows, send new data.

Events at sender: new ack, **duplicate ack**, timeout.

When we receive a duplicate ack:

- Increment duplicate ack count.

If *dupACKcount* = 3: Do fast retransmit.

- *SSTHRESH* ← *CWND* / 2
- *CWND* ← (*CWND* / 2) + 3 (*the +3 is for fast recovery*)
- Resend leftmost packet in window.

If *dupACKcount* > 3: Do fast recovery.

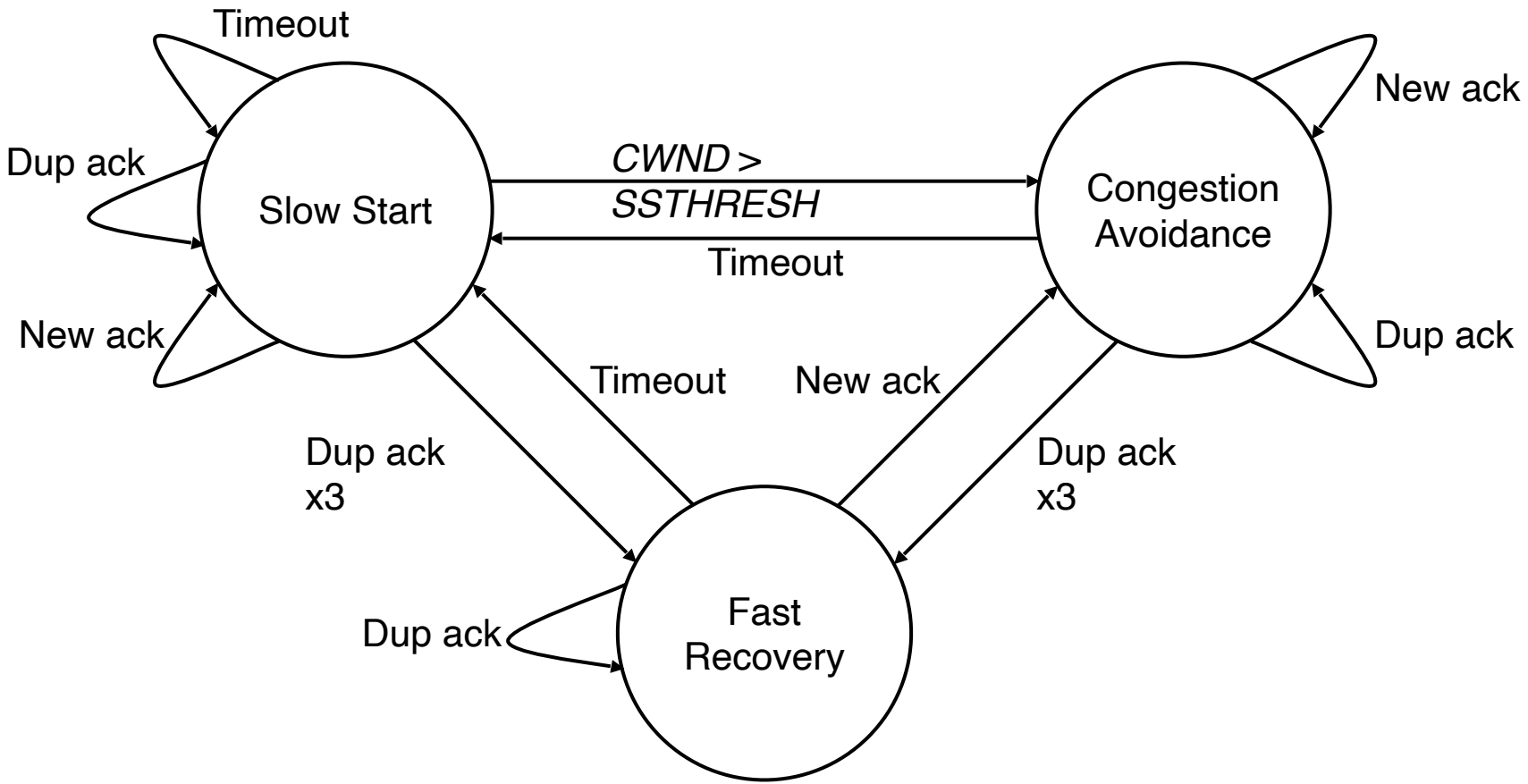
- *CWND* ← *CWND* + 1

Events at sender: new ack, duplicate ack, **timeout.**

When the timer expires:

- *SSTHRESH* ← *CWND* / 2
- *CWND* ← 1 packet
- Switch to slow-start mode.
- Resend leftmost packet in window.

TCP Congestion Control State Machine



TCP Congestion Control Variants

TCP Tahoe:

- *CWND* = 1 after triple duplicate acks.

TCP Reno:

- *CWND* = 1 on timeout.
- *CWND* = *CWND* / 2 after triple duplicate acks.

TCP New Reno:

- TCP Reno + improved fast recovery.

Unless otherwise specified, we're using this one.

TCP-SACK:

- Adds selective acknowledgements.
- Acks describe byte ranges received.

How can all these variants co-exist? Don't we need a single, uniform standard?

- Congestion control is implemented at the sender.
- The sender can run whatever code they want.

What happens if I use Reno, you use Tahoe, and we try to communicate?

- This should work fine. The variants change the rate we send data, but the packet format is the same.

What happens if I use Tahoe, and you use SACK?

- Problem: You're expecting selective acks, and I'm only providing cumulative acks.
- Solution: SACK supported is negotiated during 3-way handshake via TCP options

TCP Throughput Model

Lecture 15, CS 168, Spring 2026

Congestion Control Implementation

- Fast Recovery
- State Machine and Variants

TCP Throughput Model

Congestion Control Issues

Router-Assisted Congestion Control

Our Goal: The TCP Throughput Equation

Given a path through the network, what TCP throughput can we expect?

- The congestion control algorithm told us *how* to adjust rates.
- But it didn't tell us *what* the actual rates are.

We'll derive a simple model that expressed TCP throughput in terms of path properties:

- RTT.
- *p*, the packet loss rate.

TCP Throughput Equation: Simplifying Assumptions

Simplifying assumptions:

- There is a single TCP connection.
- Ignore the slow-start phase.

We'll assume a single, non-changing path through the network:

- RTT is some fixed number.
- Bottleneck bandwidth is some fixed number.
- Therefore: RTT × bandwidth = W_{max} (a constant value).
  - When the window size exceeds W_{max}, we lose exactly one packet.
  - Loss detected by duplicate acks. No timeouts.

From our assumptions: We lose a packet when *CWND* reaches W_{max}.

Window size changing over time:

- After detecting loss: $(0.5 \times W_{max})$
- One RTT later: $(0.5 \times W_{max}) + 1$
- Two RTTs later: $(0.5 \times W_{max}) + 2$

...

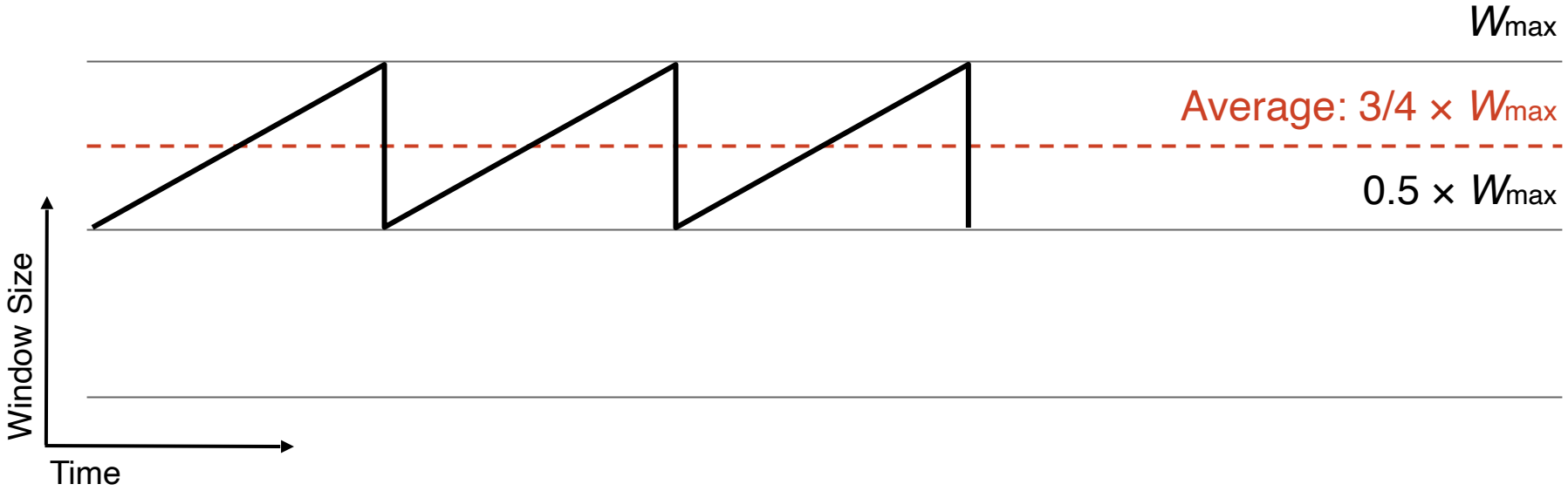
- $(0.5 \times W_{max})$ RTTs later: W_{max}
- After detecting loss: $(0.5 \times W_{max})$
- One RTT later: $(0.5 \times W_{max}) + 1$
- Two RTTs later: $(0.5 \times W_{max}) + 2$

...

Window size changing over time:

- Increase linearly from (0.5 × W_{max}) to (W_{max}).
- Drop back down to (0.5 × W_{max}).
- Repeat.

Average window size is 3/4 × W_{max}.



Unit conversion:

- Average window size is 3/4 × W_{max}.
- This is measured in packets.
- Each packet is *MSS* bytes.
- Average window size, in bytes, is 3/4 × W_{max} × *MSS*.

Computing throughput:

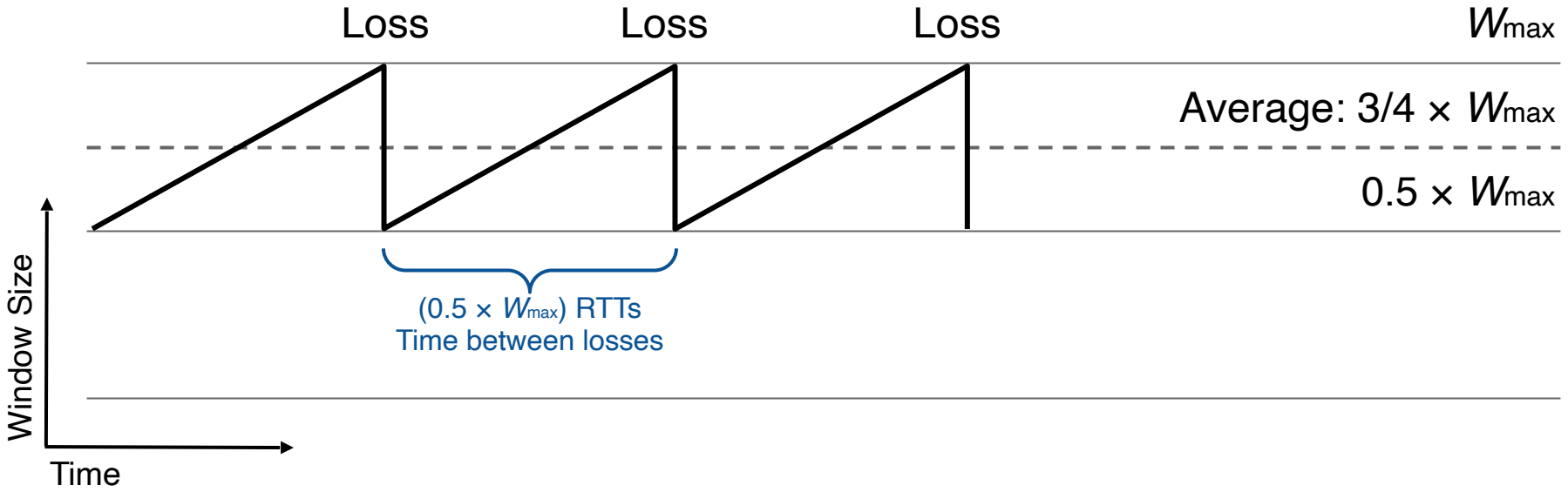
- Window size tells us how much data we can send per RTT.
- To compute rate, divide window size (data) by RTT (time):

$$\text{Throughput} = \frac{3}{4} W_{max} \times \frac{MSS}{RTT}$$

Next step: Express W_{max} in terms of p, the loss rate.

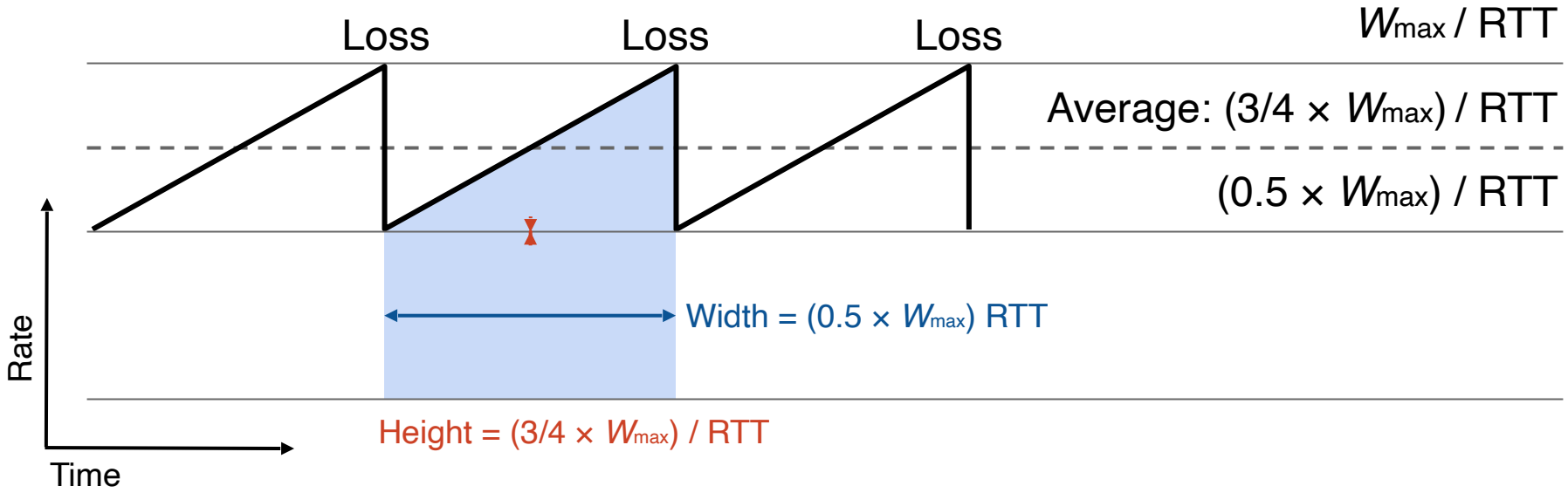
Next step: Express W_{max} in terms of p, the loss rate.

- We know one packet is lost every ($0.5 \times W_{max}$) RTTs.
- So, we just need to figure out how many packets are sent in ($0.5 \times W_{max}$) RTTs.
- If that number is x, $p = 1/x$



How many packets are sent in $(0.5 \times W_{max})$ RTTs?

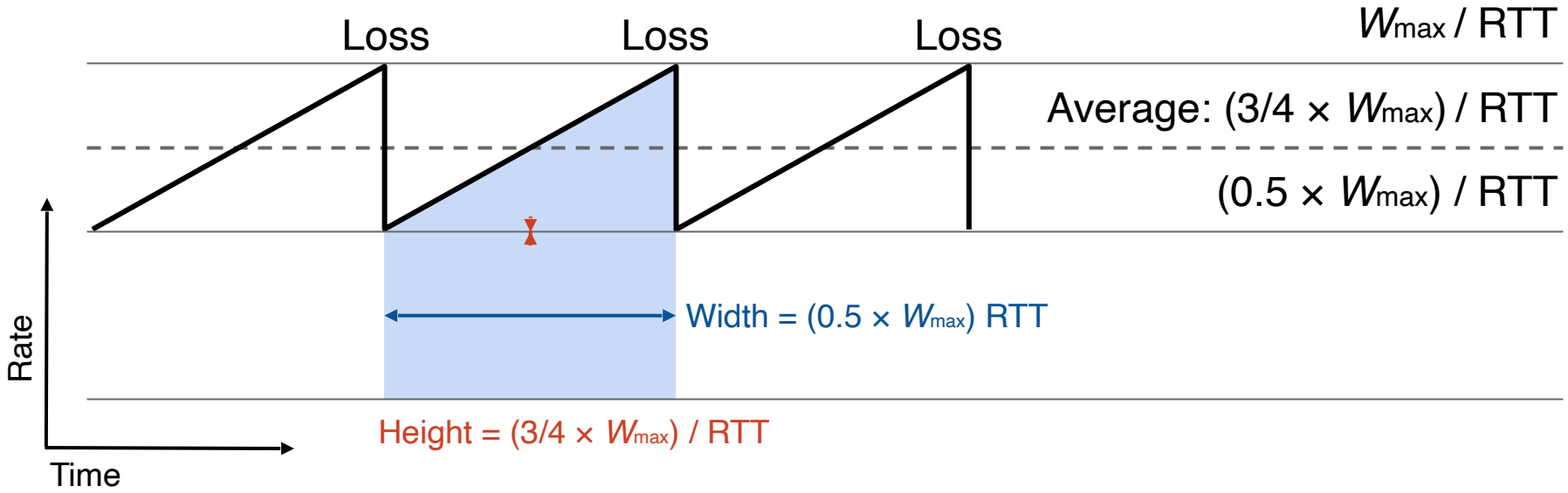
- Average window size is $3/4 \times W_{max}$.
- Answer: $(3/4 \times W_{max}) \times (0.5 \times W_{max}) = 3/8 \times W_{max}^2$ packets.



Next step: Express W_{max} in terms of p, the loss rate.

- We now know: $3/8 \times W_{max}^2$ packets are sent between losses.
- One packet lost out of that many packets.

$$\text{Loss rate} = p = \frac{8}{3W_{max}^2}$$



Loss rate = $p = \frac{8}{3W_{max}^2}$

Do some algebra to isolate W_{max} in terms of p:

$$p = \frac{8}{3W_{\text{max}}^2}$$

$$3W_{\text{max}}^2 p = 8$$

$$W_{\text{max}}^2 = \frac{8}{3p}$$

$$W_{\text{max}} = \frac{2\sqrt{2}}{\sqrt{3p}}$$

$$W_{\max} = \frac{2\sqrt{2}}{\sqrt{3p}}$$

Do some more algebra to plug this into our original throughput equation:

$$\begin{aligned}
\text{throughput} &= \frac{3}{4} W_{\max} \times \frac{\text{MSS}}{\text{RTT}} \\
&= \frac{3}{4} \left(\frac{2\sqrt{2}}{\sqrt{3p}} \right) \times \frac{\text{MSS}}{\text{RTT}} \\
&= \sqrt{\frac{3}{2}} \times \frac{\text{MSS}}{\text{RTT}\sqrt{p}}
\end{aligned}$$

The TCP Throughput Equation

$$\text{Throughput} = \sqrt{\frac{3}{2}} \times \frac{\text{MSS}}{\text{RTT}\sqrt{p}}$$

This tells us that:

- Throughput is inversely proportional to RTT.
  - Shorter RTT = higher throughput.
- Throughput is inversely proportional to square root of loss rate.
  - Lower loss rate = higher throughput.

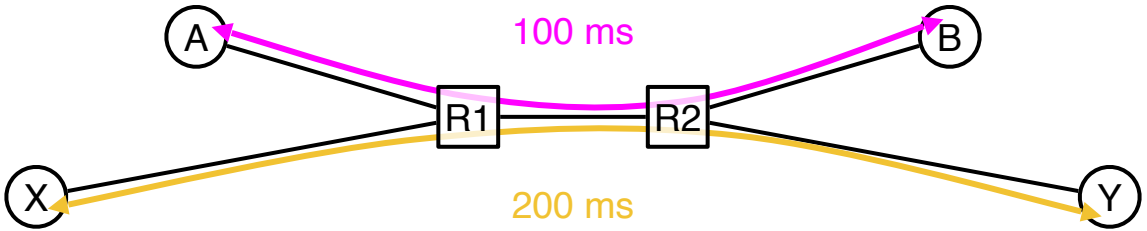
Consequences of Equation (1/2): Flows with Different RTTs

$$\text{Throughput} = \sqrt{\frac{3}{2}} \times \frac{\text{MSS}}{\text{RTT}\sqrt{p}}$$

TCP is inherently unfair when flows have different RTTs.

- Shorter RTT = higher throughput.
- The flow with shorter RTT gets higher throughput.
- Nothing we can really do about it. Treat it as a feature of TCP.

From the equation: A–B gets twice as much bandwidth as X–Y.



$$\text{Throughput} = \sqrt{\frac{3}{2}} \times \frac{\text{MSS}}{\text{RTT}\sqrt{p}}$$

Look up RFC 5348 for spec.

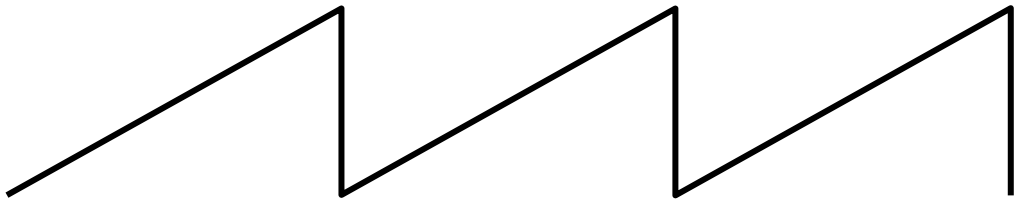
TCP throughput is choppy: Rate oscillates between *W*/2 and *W*.

Some applications (e.g. video streaming) would prefer sending at a steady rate.

Solution: Equation-based congestion control.

- Abandon TCP's adjustment rules, and just follow the equation.
- Measure RTT, measure *p*, and compute the rate accordingly.

Using the equation ensures fairness: We don't use more bandwidth than TCP would.



Congestion Control Issues

Lecture 15, CS 168, Spring 2026

Congestion Control Implementation

- Fast Recovery
- State Machine and Variants

TCP Throughput Model

Congestion Control Issues

Router-Assisted Congestion Control

We'll look at 5 issues (and potential solutions):

1. Confusing corruption and congestion.
2. Short connections complete before discovering available capacity.
3. Router queues get filled up, causing high delays.
4. Cheating.
5. Congestion control and reliability are intertwined.

TCP detects congestion by checking for loss.

- Loss could also occur due to corruption.
- TCP will confuse corruption with congestion.

From the equation: Higher loss = lower throughput.

- Still true, even if the losses aren't due to congestion!
- Equation could be used to analyze how TCP would perform on a lossy link (high corruption rate).

Most real-life TCP connections are really short.

- 50% of connections send fewer than 1.5 KB.
- 80% of connections send fewer than 100 KB.
- Very few packets (maybe only one) are sent.

Many connections stay in slow-start the whole time.

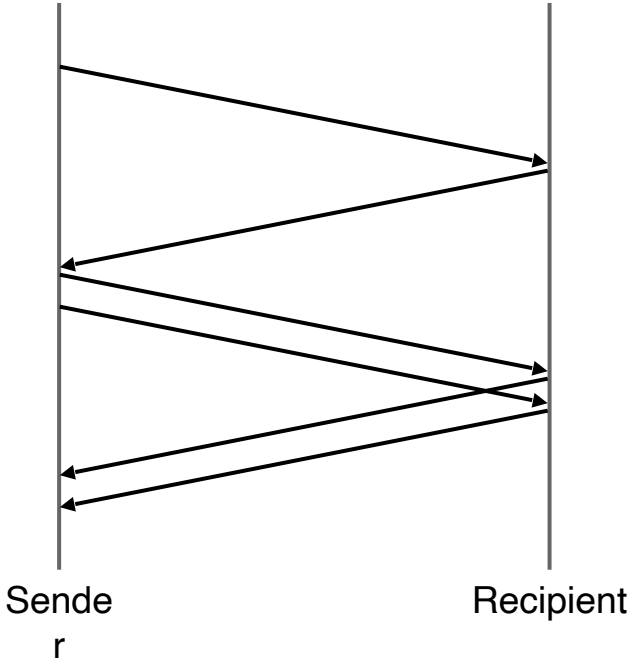
- Short connections likely to suffer from high latency.

Not enough packets to trigger duplicate acks.

- Isolated loss may lead to timeouts.
- Timeouts can severely increase latency.

Partial fix: Start with a higher initial *CWND*.

Look up RFC 5348 for experiments on this.



Recall slow start: Start with *CWND* = 1, double every RTT.

It took ≈2 RTTs to send 3 packets.

TCP deliberately overshoots capacity until packets get dropped.

- Recall: Loss occurs when the queue is full.
- By then, the queue is already full, and packets are delayed.

Result: Delays are large for everybody.

- Someone is transferring a 10 GB file. Queues are filled with their packets.
- You want to download a 100 byte file. You're stuck waiting in the queue.

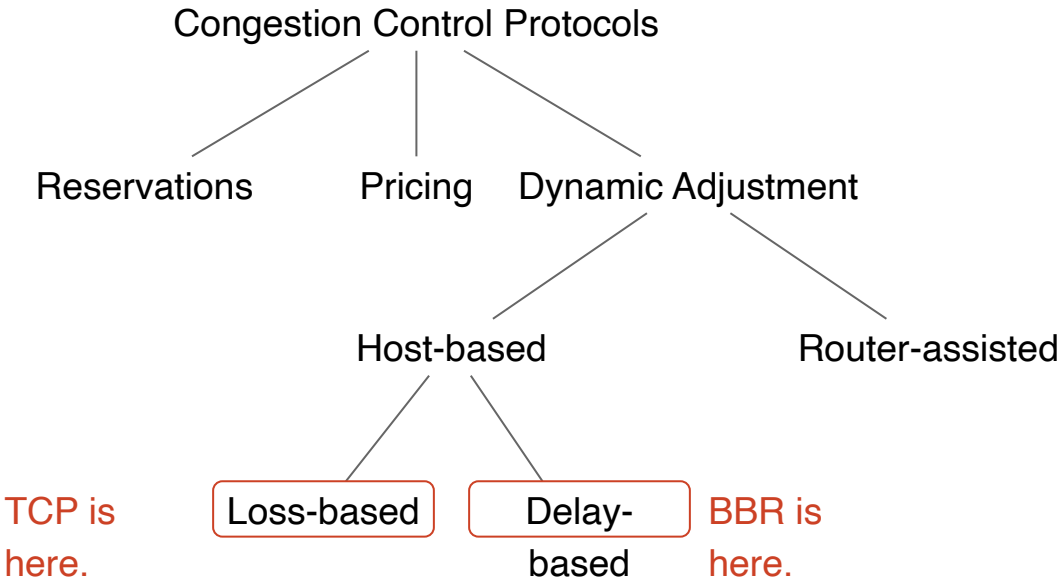
The problem is even worse if routers keep really long queues.

- **Bufferbloat**: Routers have excessive memory, and maintain long queues.
- When loss occurs, everybody is already waiting in a really long queue.

Possible solution: Google's BBR algorithm (2016).

Link to BBR paper.

- Detects congestion using *delay* instead of *loss*.
- Sender learns its minimum RTT.
- Sender slows down if it observes RTTs exceeding the minimum.



Nobody is enforcing that users follow the TCP congestion control algorithm.

Cheating strategy: Change the algorithm.

- Increase the window faster (+2, instead of +1).
- Start with a large initial *CWND*.

Cheating strategy: Open lots of connections.

- TCP shares bandwidth between connections.
- If Alice opens 10 connections, and Bob opens 1, then Alice gets 10x more bandwidth.

Applying graphical model to a cheating host:

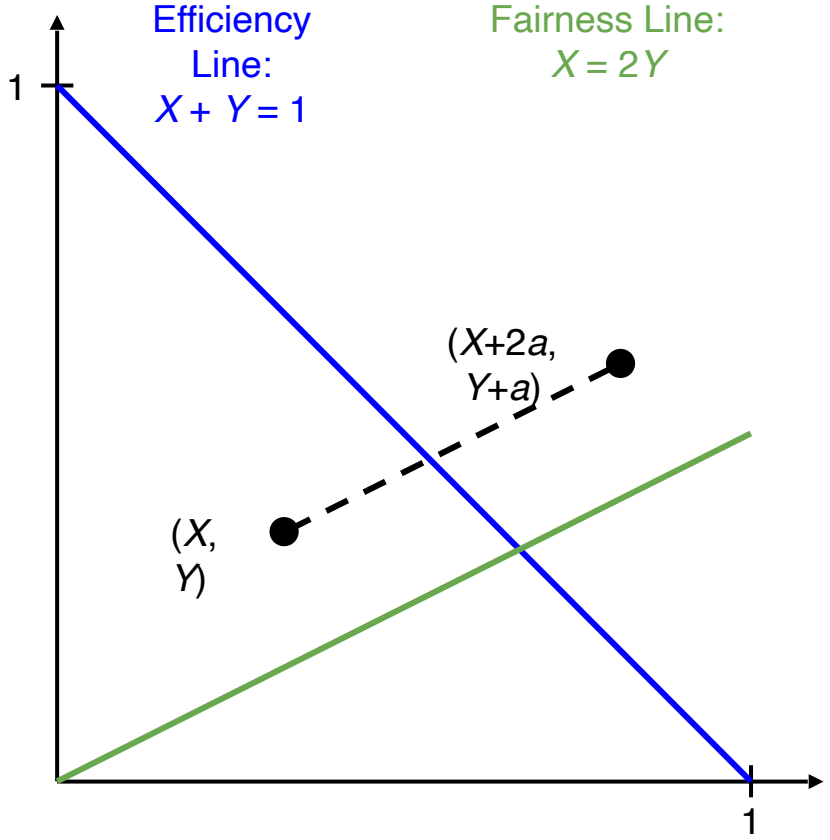
- X increases by 2 per RTT.
- Y increases by 1 per RTT.

Suppose current allocation is (X, Y).

If both increase additively: ($X + 2a$, $Y + a$).

Notice: Slope of this line is now 1/2, not 1.

- Fairness/efficiency lines now meet at (2/3, 1/3).

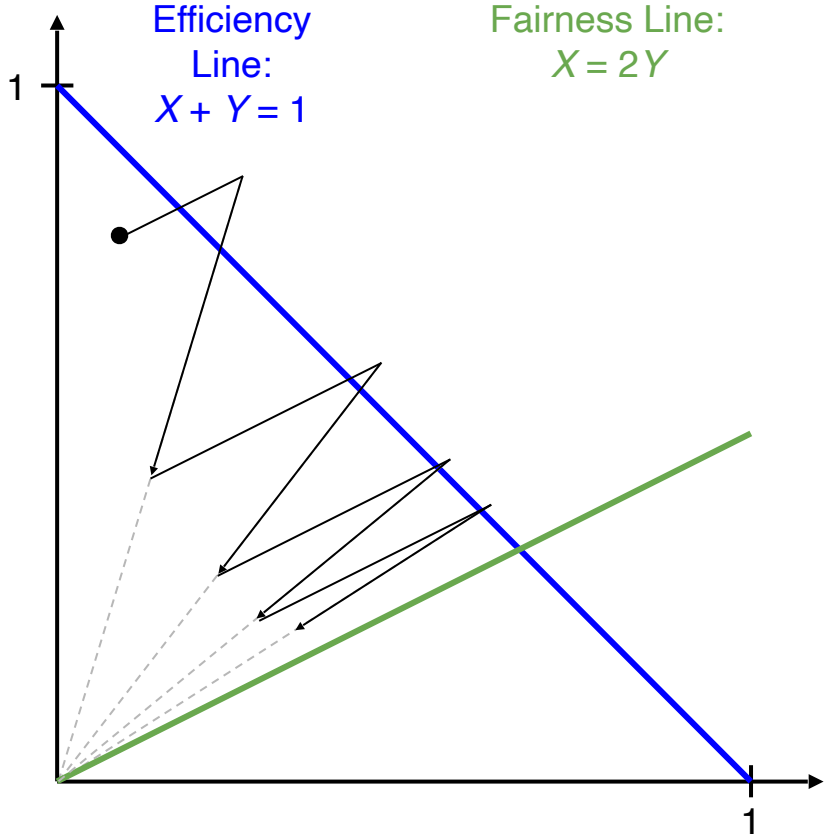


AIMD (Additive Increase, Multiplicative Decrease) Adjustments on Graph

Increase: +2 (X), +1(Y)

Decrease: ÷2

Because X is cheating, AIMD converges toward a "fairness" line where X has twice as much bandwidth as Y.



Why hasn't the Internet suffered another congestion collapse? Some theories:

- Cheaters might get an unfair share of bandwidth, but they still follow basic congestion control rules (e.g. slow down when loss occurs).
  - Contrast with the 1980s: Everybody sent at maximum rate, no adjusting.
- TCP is implemented in the operating system.
  - Most users probably aren't changing their OS code.

How much cheating occurs in practice?

- We don't really know. Measuring cheating is hard.

MOTHERBOARD
TECH BY VICE

Link

Google's Network Congestion Algorithm Isn't Fair, Researchers Say

Karl Bode *October 31, 2019*

Issues (5/5): Congestion Control and Reliability are Intertwined

Mechanisms for congestion control and reliability are tightly coupled.

- A design choice from the 1980s. (TCP was patched to stop congestion collapse.)
- Example: CWND is adjusted based on acks and timeouts.
- Example: We detect loss/congestion with duplicate acks, because TCP uses cumulative acks.

This complicates evolution.

- Example: If we wanted to switch from cumulative to selective acks, we'd have to change the congestion control algorithm too.
- This is a failure of modularity, not layering.

Sometimes we only want one, but not the other.

- Congestion control, no reliability: Video streaming.
- Reliability, no congestion control: Lightweight application (one packet per hour).

Router-Assisted Congestion Control

Lecture 15, CS 168, Spring 2026

Congestion Control Implementation

- Fast Recovery
- State Machine and Variants

TCP Throughput Model

Congestion Control Issues

Router-Assisted Congestion Control

Many of our issues could be fixed with some help from routers!

1. Confusing corruption and congestion.
2. Short connections complete before discovering available capacity.
3. Router queues get filled up, causing high delays.
4. Cheating.
5. Congestion control and reliability are intertwined.

Two ways routers can help:

- **Enforce fairness. (Helps with 4.)**
- Send information to hosts. (Helps with 1, 2, 3.)

How can routers ensure each flow gets its fair share?

- Consider a single router's actions.
- Router classifies incoming packets into flows (think: TCP connections).
- Each flow has its own separate FIFO queue in the router.
- Router picks a queue (i.e. flow) in a fair order, and transmits packet from the front of that queue.

What does fair mean exactly?

For now, assume all packets are the same size.

Suppose we have three connections:

- A has 8 packets to send: A1 A2 A3 A4 A5 A6 A7 A8
- B has 6 packets to send: B1 B2 B3 B4 B5 B6
- C has 2 packets to send: C1 C2

Suppose we only have capacity to send 10 packets.

Round-robin service: Take turns sending packets from each queue.

- Packets sent: A1 B1 C1 A2 B2 C2 A3 B3 A4 B4
  - We sent: 4x A packets, 4x B packets, and 2x C packets.
- Packets not sent: A5 A6 A7 A8 B5 B6

With round-robin service and capacity 10:

- A had 8 packets. We sent 4.
- B had 6 packets. We sent 4.
- C had 2 packets. We sent 2.

Property: If you don't get your full demand, nobody gets more than you.

- C got its full demand.
- A and B did not, but they each got an equal share of the remainder.
- This is called **max-min fairness**.

Max-Min Fairness (Equal Packet Size)

Instead of round-robin, we could mathematically solve max-min fairness.

Resource allocation problem:

- Capacity is 10.
- A wants 8. B wants 6. C wants 2.

Intuitive solution:

- If we split equally, everyone gets 10/3 = 3.33.
- But C only wants 2, so let's give C its full demand. **C = 2**. Now we have 8 left.
- If we split equally, A and B each get 8/2 = 4.
- We can't meet A or B's demands, so they each get a fair share of the remainder. **A = 4**, **B = 4**.

How do we deal with packets of different sizes?

- Mental model: bit-by-bit round-robin ("fluid flow").
- We can't actually do this in practice!
- But we can approximate it. This is what **fair queuing** routers do.

We won’t cover fair queueing in detail this class, but if you’re interested please refer to the paper below.

Analysis and Simulation of a Fair Queueing Algorithm	Link
Alan Demers, Srinivasan Keshav, Scott Shenker	*1990*

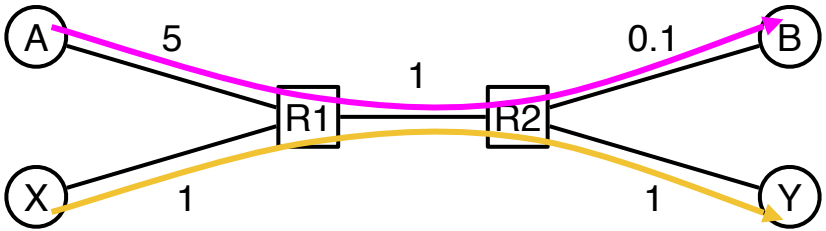
Note: Fair Queuing Does Not Solve Congestion

Fair queuing does not *eliminate* congestion.

- A → B wants to send at 5 Gbps. X → Y wants to send at 1 Gbps.
- R1 implements fair queuing, and gives 0.5 Gbps to each flow.
- If A → B runs at 0.5 Gbps, then R2 ends up dropping 0.4 Gbps (sending only 0.1).
- R1 fairly allocating didn't help. A needs to slow down.

Fair queuing can *manage* congestion.

- Resilient to cheating, RTT variations, etc.
- Congestion and packet drops still occur.
- We still want end hosts to discover/adapt to their fair share.



Two ways routers can help:

- Enforce fairness. (Helps with 4.)
  - Fair queuing (and approximations like DRR).
- **Send information to hosts. (Helps with 1, 2, 3.)**

Router-Assisted Rate Adaptation

Why not just let routers tell the end hosts what rate they should use?

Possible design:

- Packets carry an extra "rate" field in the header.
- Routers insert a flow's fair share in the header.
- End hosts set rate according to the header value.

Now, the sender doesn't need to dynamically adjust to find a good rate.

Explicit Congestion Notification (ECN) bit: Single bit in the IP packet header.

- Congested routers can set this bit.
  - When recipient gets a packet with ECN on, the ack also has ECN on.
- Many options for *when* routers set the bit.
  - Trade-offs between high link utilization and packet delay.
- Sender could treat an ECN bit as a packet drop and adjust accordingly.
- Pros:
  - Doesn't confuse corruption and congestion.
  - Allows routers to warn about congestion earlier (e.g. before queue is full). Reduces delays.
  - Lightweight to implement.

Used in some, but not all routers.

- Most useful in local networks (e.g. datacenters) where all routers use the bit.