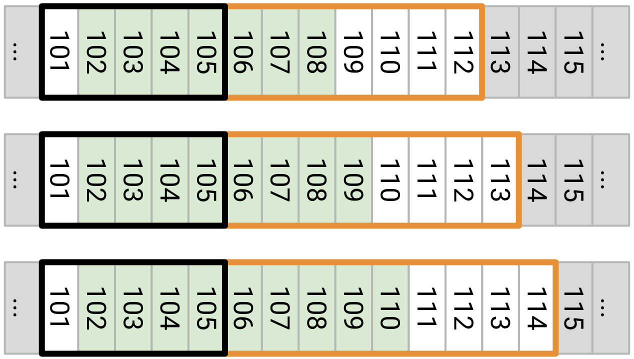


Lecture 14

Congestion Control, Part 2

CS 168, Spring 2025 @ UC Berkeley

Slides credit: Sylvia Ratnasamy, Rob Shakir, Peyrin Kao

Fast Recovery

Lecture 15, CS 168, Spring 2025

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.

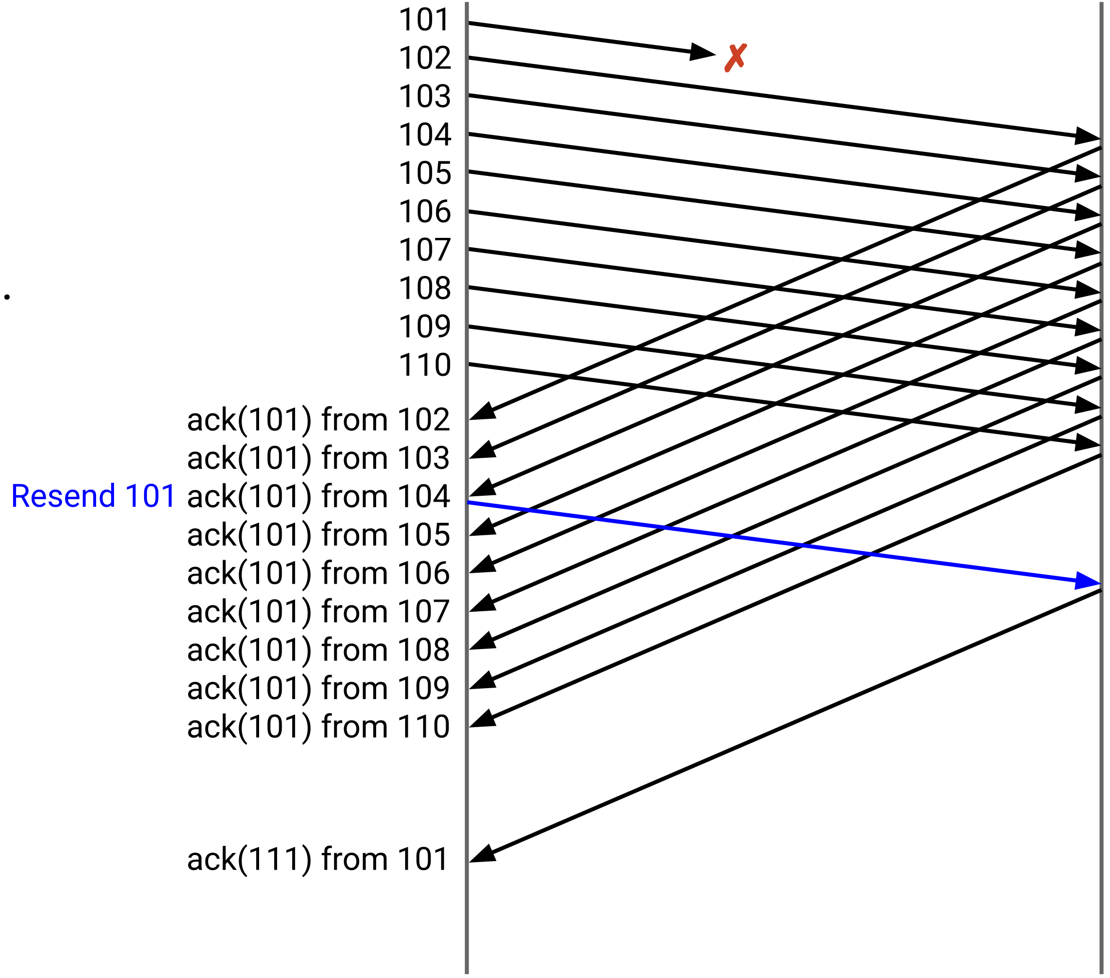
The first one (101) is dropped.

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?



Fast Recovery: Windows Without Fast Recovery

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

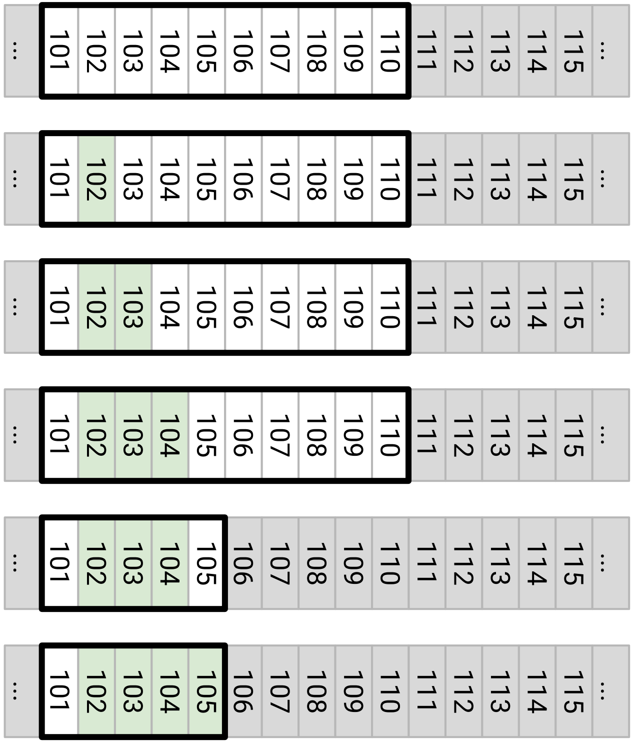
After ack(101) from 102: Can't send anything new.

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.

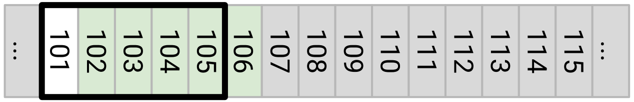
After ack(101) from 105: Can't send anything new.



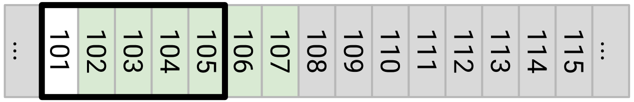
Note: We're marking acked packets in green for convenience, but the sender cannot deduce from duplicate acks that these exact packets were received.

Fast Recovery: Windows Without Fast Recovery

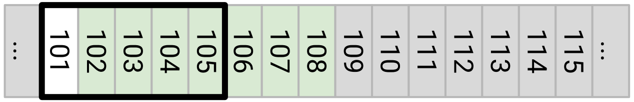
After ack(101) from 106: Can't send anything new.



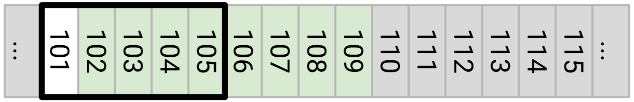
After ack(101) from 107: Can't send anything new.



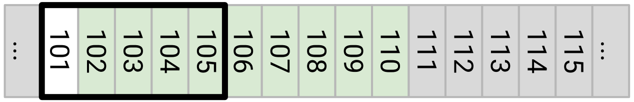
After ack(101) from 108: Can't send anything new.



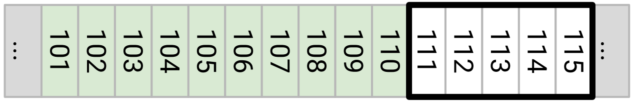
After ack(101) from 109: Can't send anything new.



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

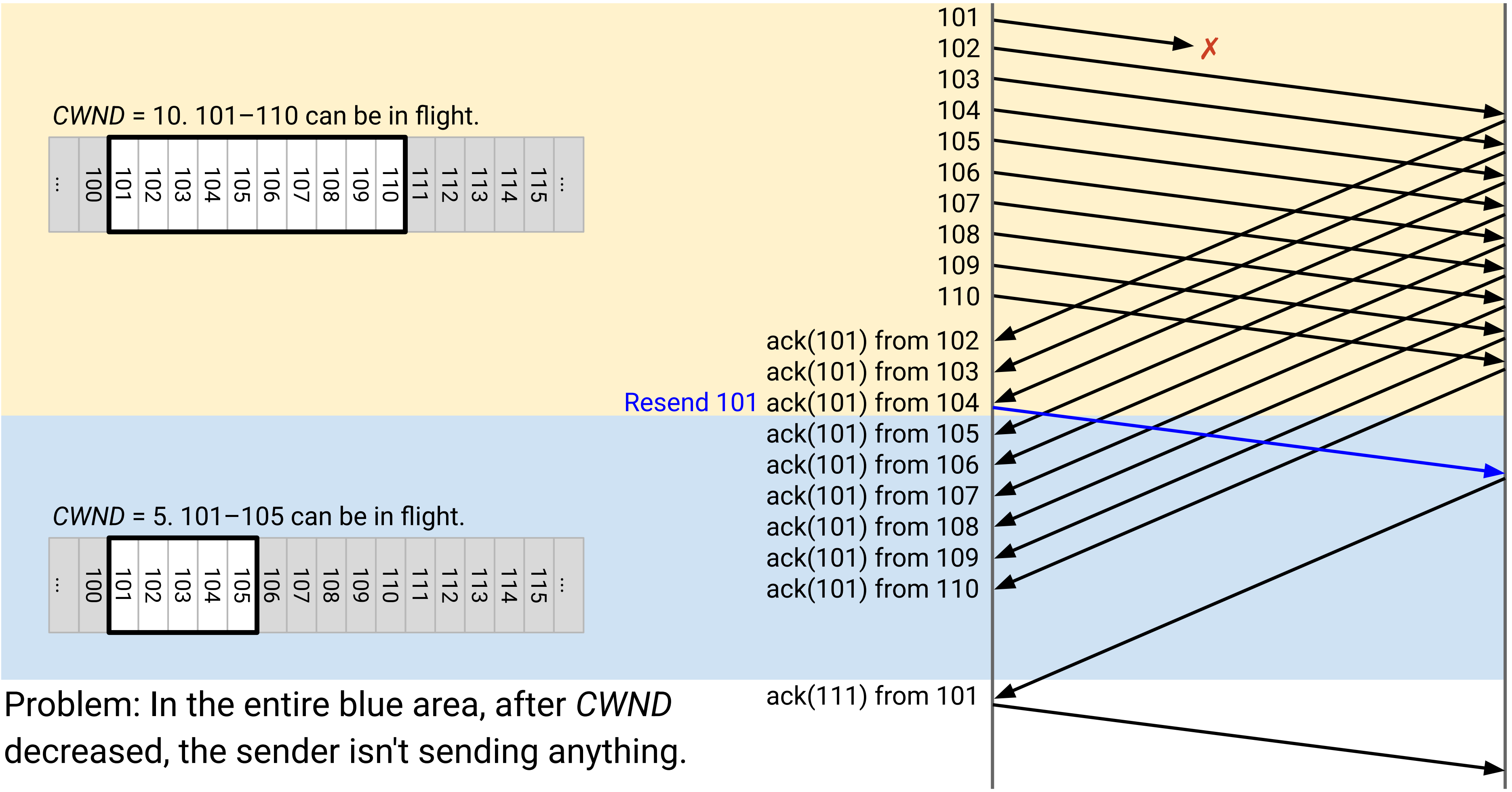


After ack(111) from resent 101:
Window slides to first unacked packet (111).
Sender is now able to send 111–115.



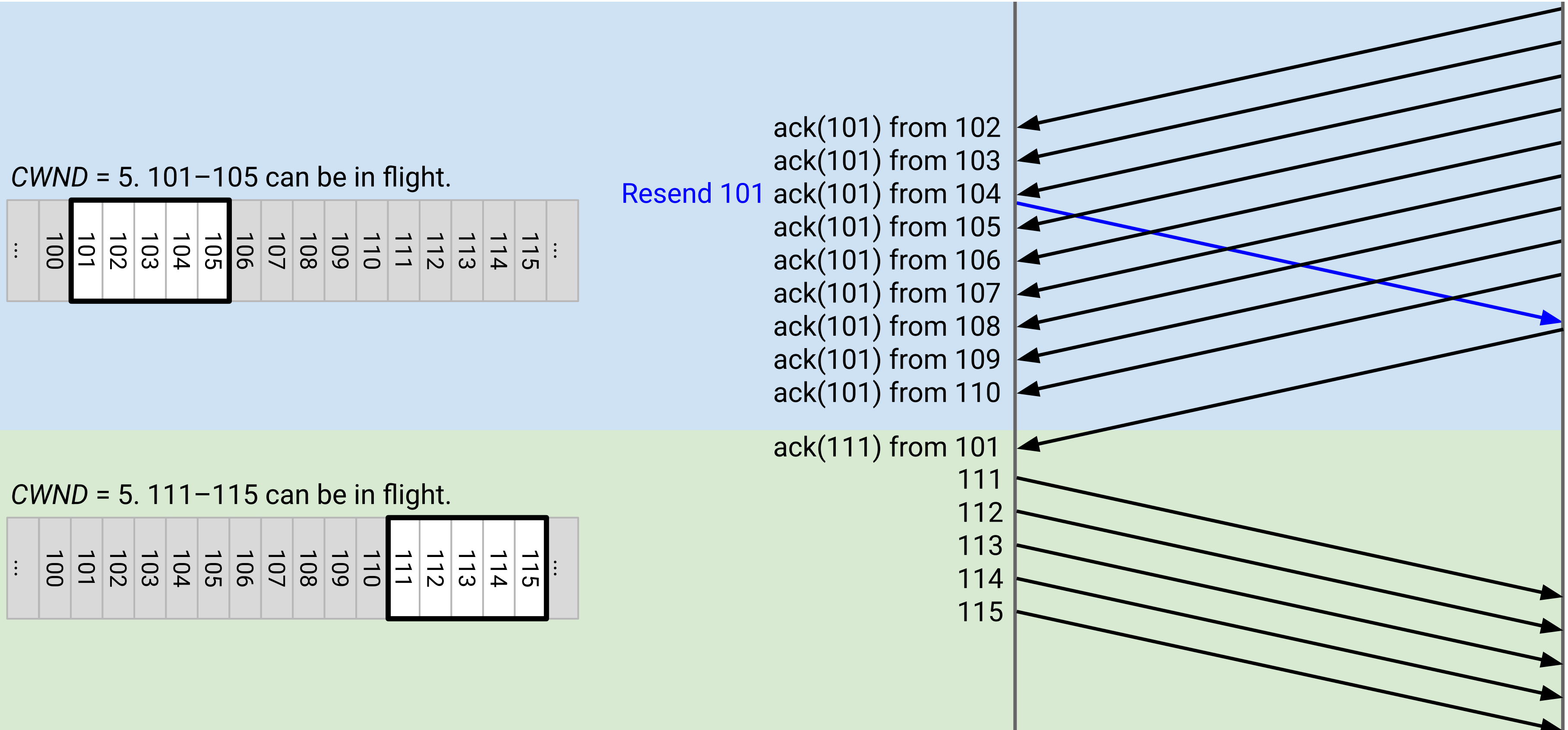
Problem: After we decreased *CWND*, we had to wait a long time before sending again.

Fast Recovery: Windows Without Fast Recovery



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

Fast Recovery: Windows Without Fast Recovery



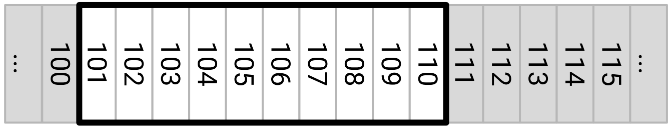
Secondary problem: Sender has to wait a full RTT for ack(112) to arrive before sending 116+.

The Fast Recovery Problem

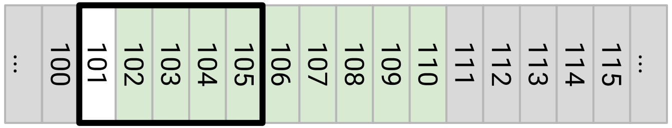
The problem, restated again:

- 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.

CWND = 10. 101–110 can be in flight.



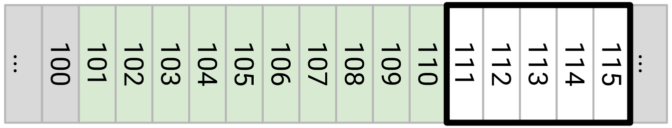
CWND = 5. 101–105 can be in flight.



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 = 5. 111–115 can be in flight.



The Fast Recovery Problem

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 Fast Recovery Problem

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

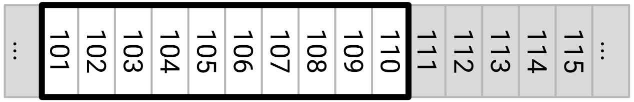
- Even though there are eventually <5 packets left in flight, the sliding window is stopping us from sending more.

Key idea:

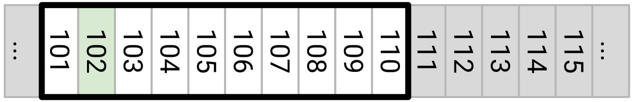
- Grant the sender temporary "credit" for each duplicate ack.
- When a duplicate ack arrives, we know that one fewer packet is in flight.
- *Artificially extend* the window to let the sender send one more packet.

Fast Recovery: Windows with Fast Recovery

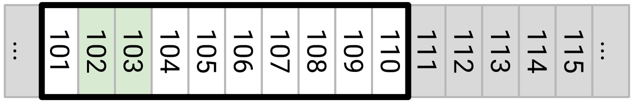
After sending 10 packets: *CWND* = 10.
101–110 can be in flight.



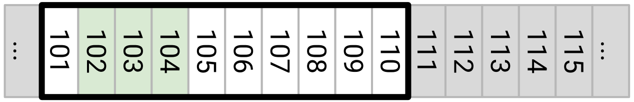
After ack(101) from 102: Can't send anything new.



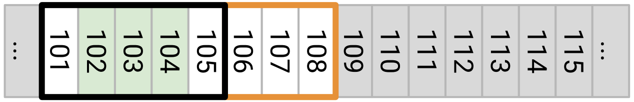
After ack(101) from 103: Can't send anything new.



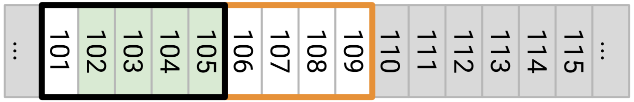
After ack(101) from 104: Can't send anything new.



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

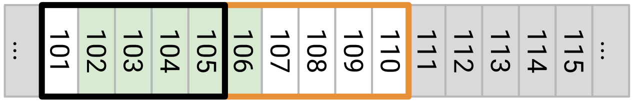


After ack(101) from 105: Extend window to *CWND* = 9.
Can't send anything new.

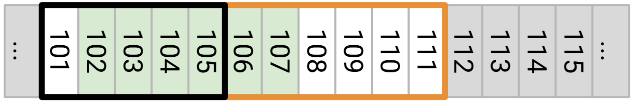


Fast Recovery: Windows with Fast Recovery

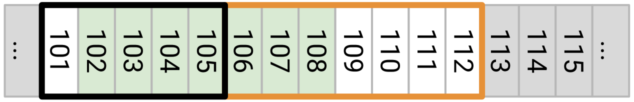
After ack(101) from 106: Extend window to C = 10.
Can't send anything new.



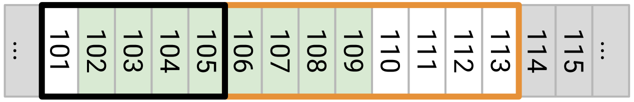
After ack(101) from 107: Extend window to C = 11.
We can send 111!



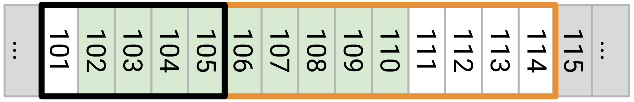
After ack(101) from 108: Extend window to C = 12.
We can send 112!



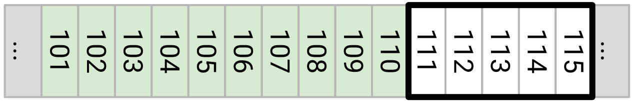
After ack(101) from 109: Extend window to C = 13.
We can send 113!



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.



Fast Recovery: Windows with Fast Recovery

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

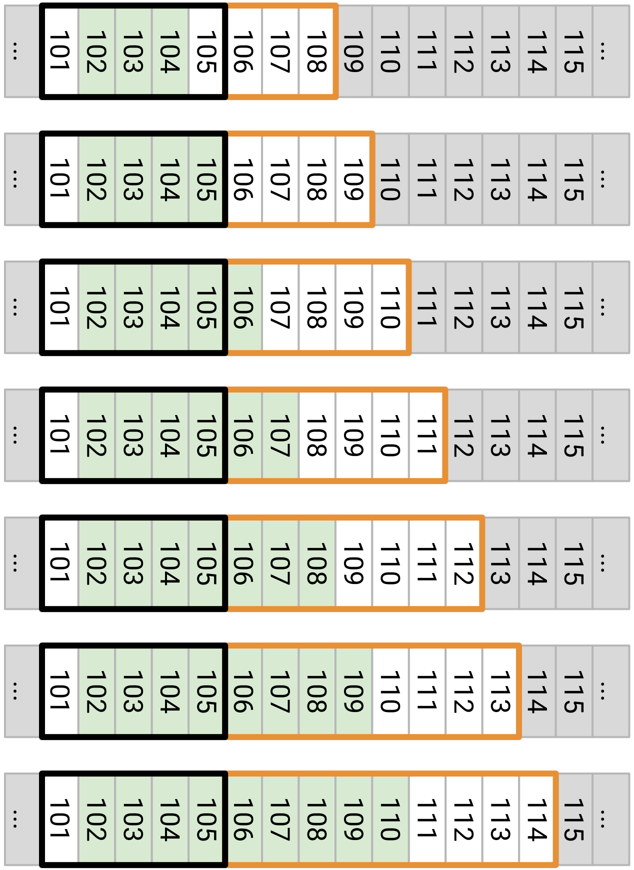
- The sender only sees duplicate acks.

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

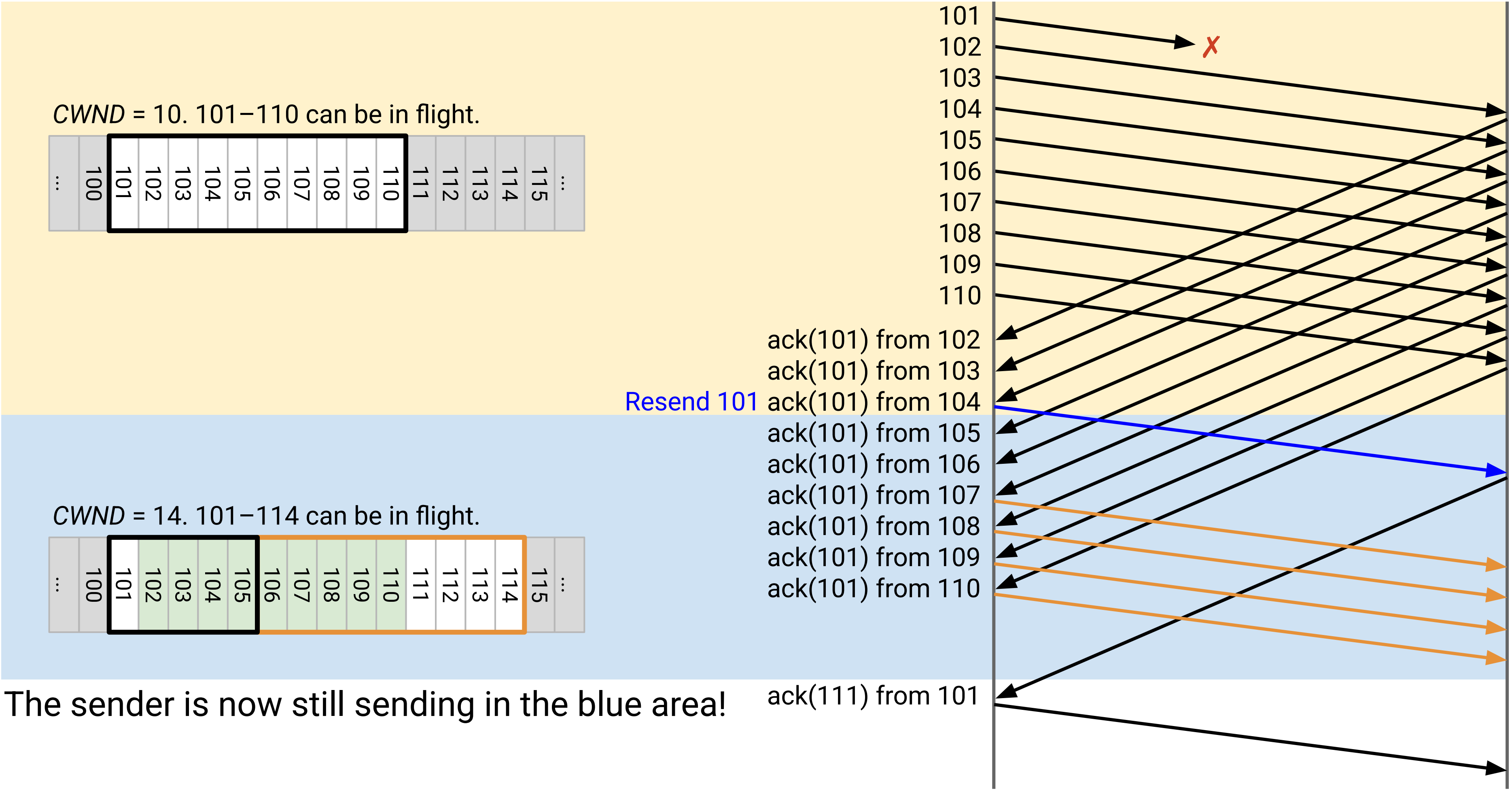
- The sender can count duplicate acks.

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*).

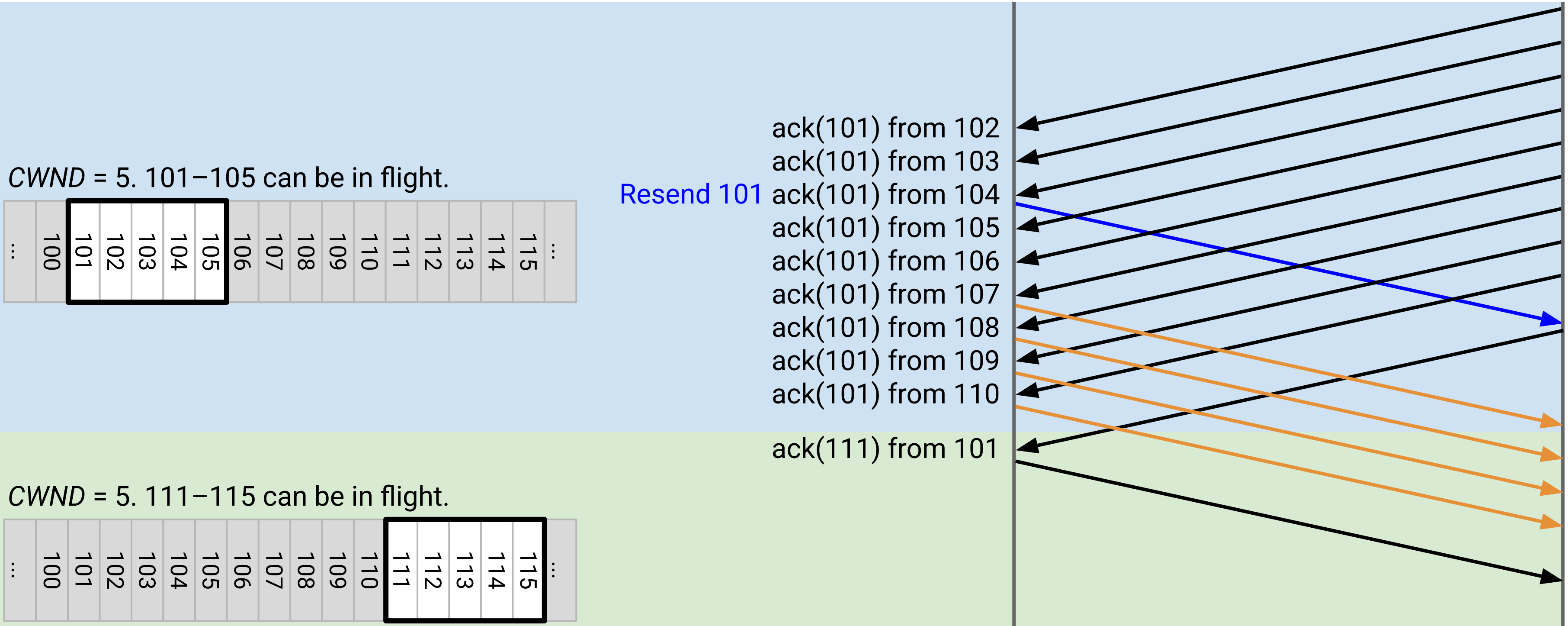


Fast Recovery: Windows Without Fast Recovery



The sender is now still sending in the blue area!

Fast Recovery: Windows Without Fast Recovery



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:
  - $SSTHRESH \leftarrow CWND/2$
  - $CWND = CWND/2 + 3$ *(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 2025

Putting It All Together: TCP with 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.

Putting It All Together: TCP with Congestion Control

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

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

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

Putting It All Together: TCP with Congestion Control

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

When we receive a duplicate ack:

- Increment duplicate ack count.

If *dupACKcount* = 3: Do fast retransmit.

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

If *dupACKcount* > 3: Do fast recovery.

- $CWND \leftarrow CWND + 1$

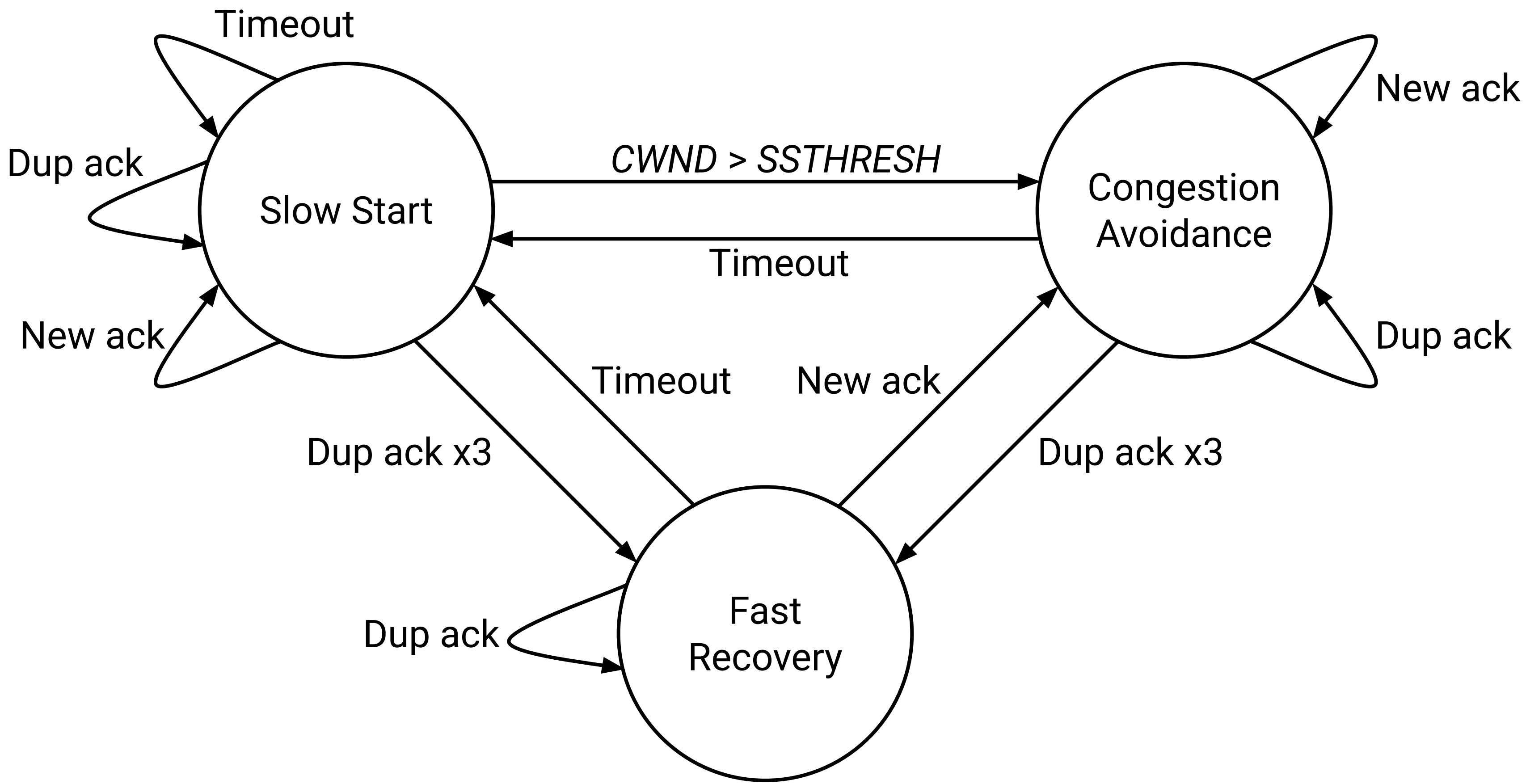
Putting It All Together: TCP with Congestion Control

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

When the timer expires:

- $SSTHRESH \leftarrow CWND$ / 2
- $CWND \leftarrow$ 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.

Interoperability Between TCP Congestion Control Variants

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.

TCP Throughput Model

Lecture 15, CS 168, Spring 2025

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.

TCP Throughput in Terms of Window Size

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: $(0.5 \times W_{max}) + 2$

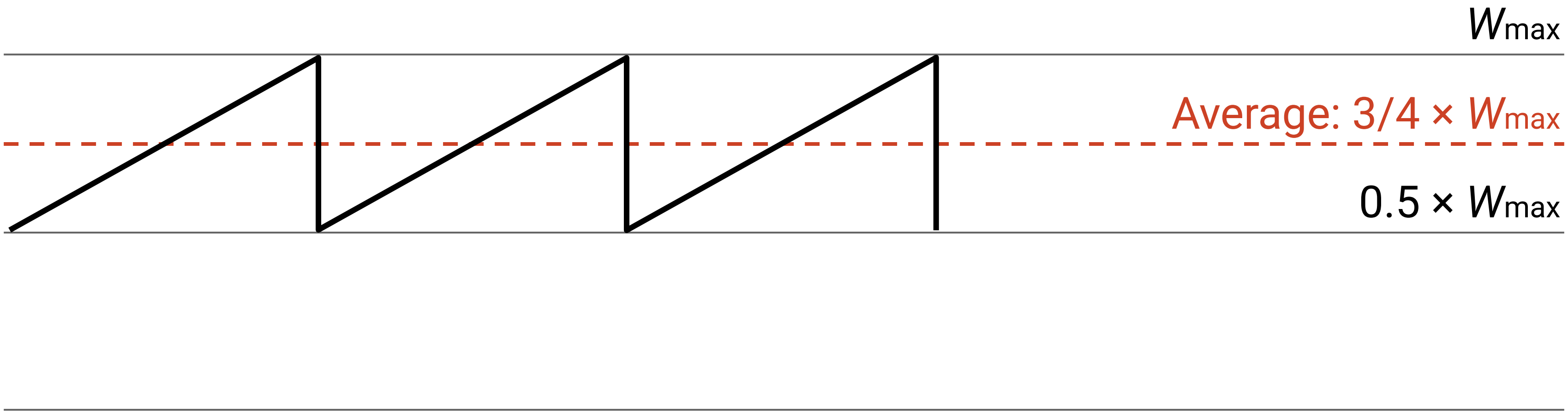
  ...

TCP Throughput in Terms of Window Size

Window size changing over time:

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

Average window size is $3/4 \times W_{max}$.



TCP Throughput in Terms of Window Size

Unit conversion:

- Average window size is $3/4 \times W_{max}$.
- This is measured in packets (since we were adding 1 packet per iteration).
- Each packet is *MSS* bytes.
- Average window size, in bytes, is $3/4 \times W_{max} \times MSS$.

Computing throughput:

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

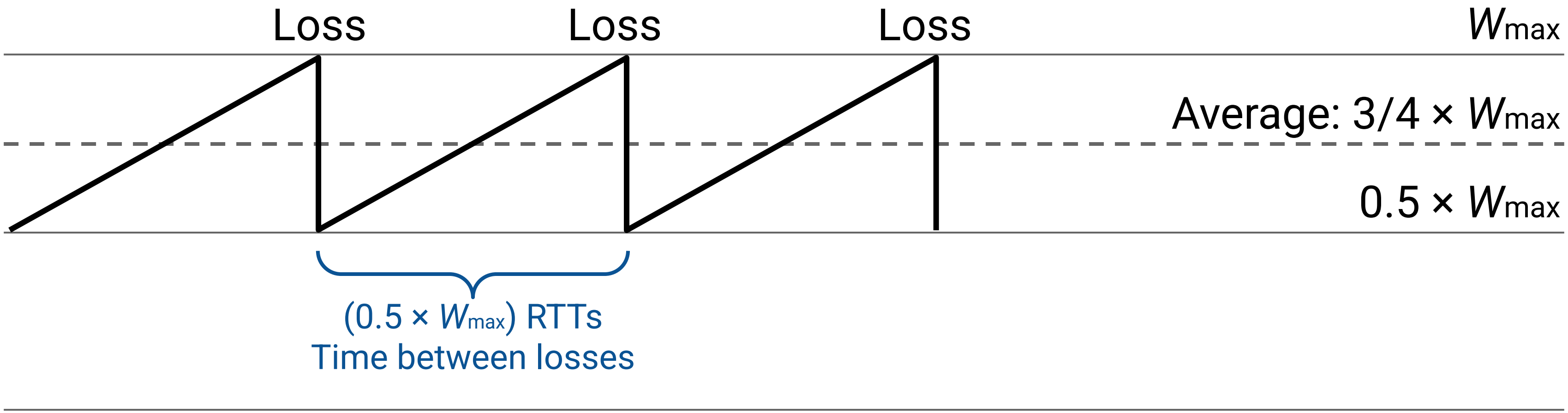
$$\text{Throughput} = \frac{3}{4} W_{max} \times \frac{MSS}{RTT}$$

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

Relating Window and 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.
  - This is how long it takes to climb from ($0.5 \times W_{max}$) back up to (W_{max}).
  - Each RTT adds 1, and we have to climb ($0.5 \times W_{max}$) in total.
- So, we just need to figure out how many packets are sent in ($0.5 \times W_{max}$) RTTs.

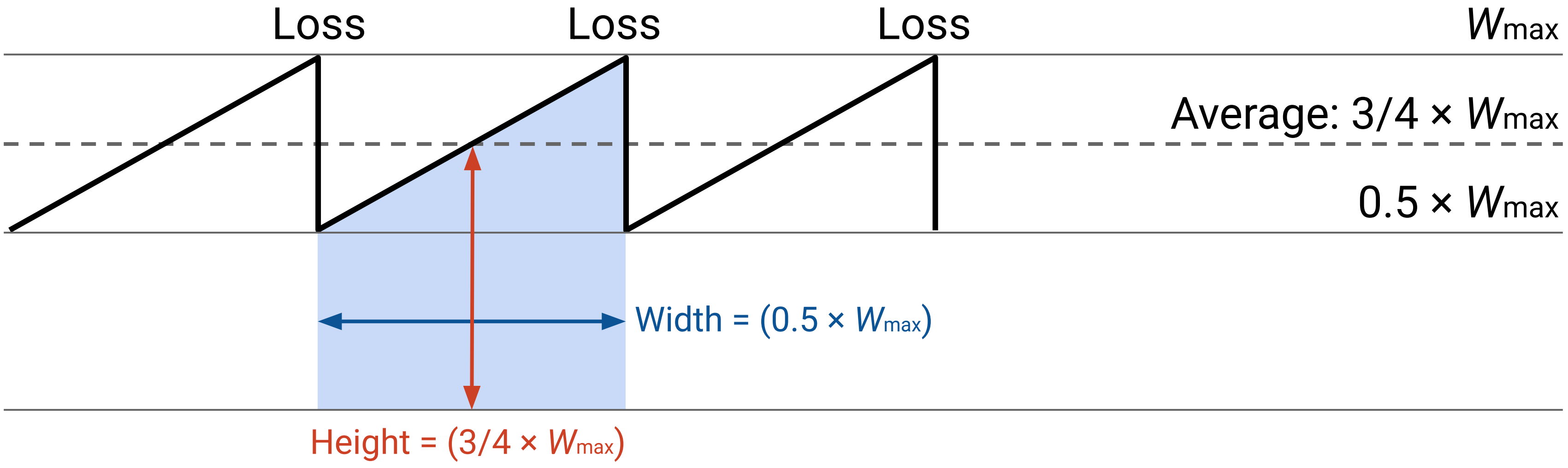


Relating Window and Loss Rate

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

- Average window size is $3/4 \times W_{max}$. That's how many packets we send per RTT.
- Answer: $(3/4 \times W_{max}) \times (0.5 \times W_{max}) = 3/8 \times W_{max}^2$ packets.

Can also be computed as the area of the shape (rate × time), or area under the curve (integral of rate).

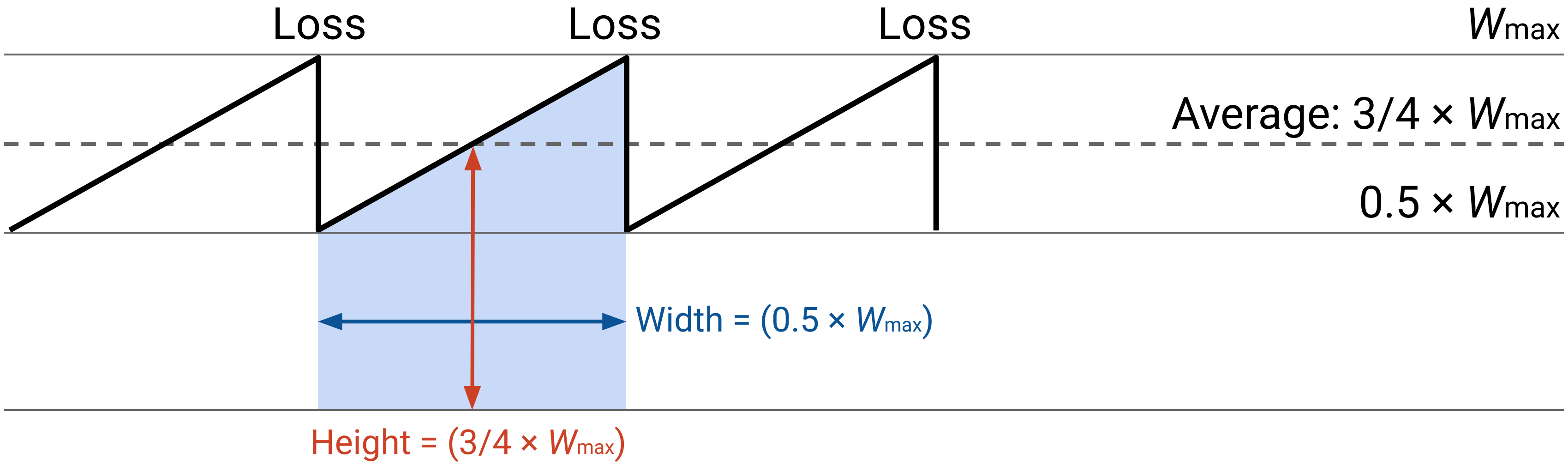


Relating Window and Loss Rate

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.

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



Relating Window and Loss Rate

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

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

$$
\begin{aligned}
p &= \frac{8}{3W_{\max}^2} \\
3W_{\max}^2 p &= 8 \\
W_{\max}^2 &= \frac{8}{3p} \\
W_{\max} &= \frac{2\sqrt{2}}{\sqrt{3p}}
\end{aligned}
$$

Relating Window and Loss Rate

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

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

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.

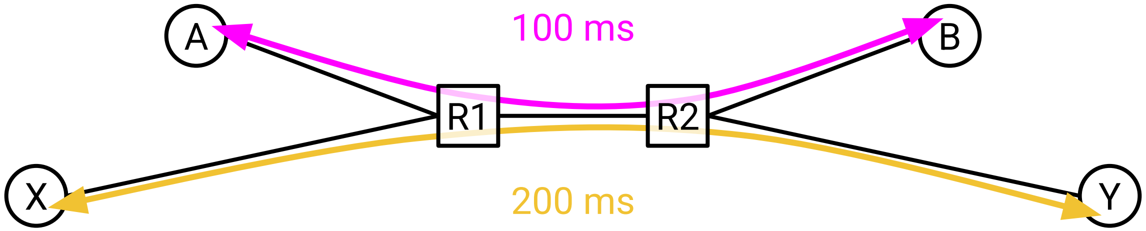
Consequences of Equation (1/2): Flows with Different RTTs

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.



Consequences of Equation (2/2): Rate-Based Congestion Control

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

Look up RFC 5348 for spec.

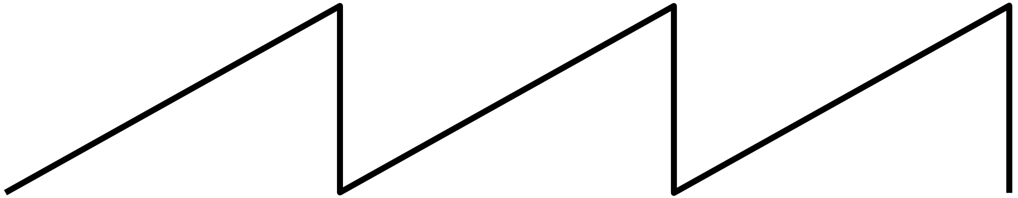
TCP throughput is choppy: Rate oscillates between $W/2$ and W.

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 2025

Congestion Control Issues

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.

Issues (1/5): Confusing Corruption and Congestion

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).

Issues (2/5): Short Connections

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.

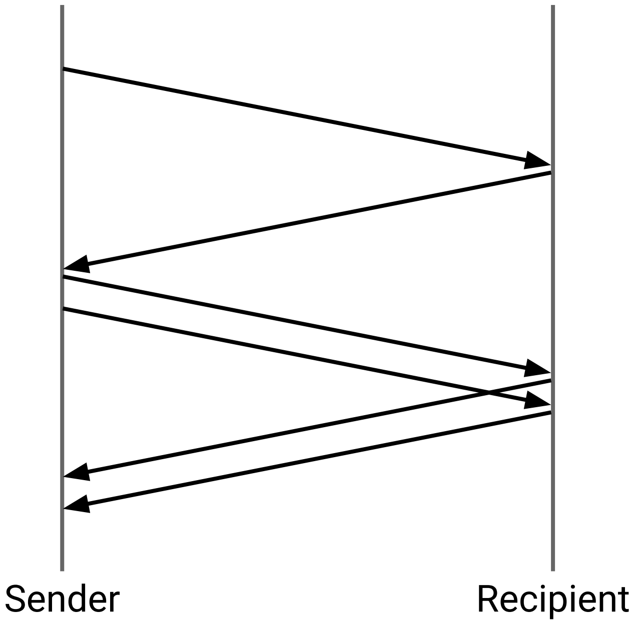
- Short connections likely to suffer from high latency.

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.

Issues (3/5): Router Queues Get Filled Up

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.

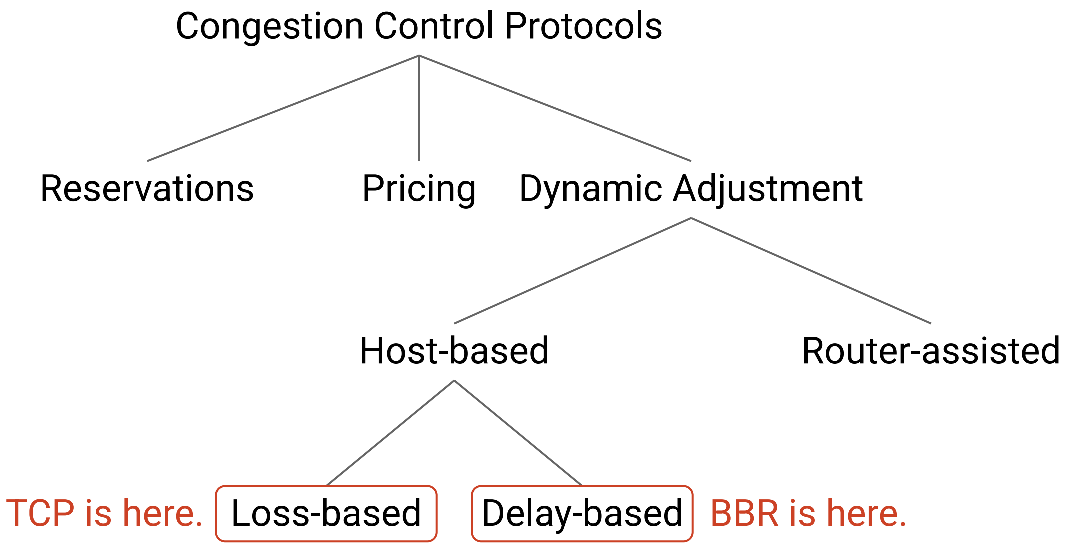
The problem is even worse if routers keep really long queues.

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

Issues (3/5): Router Queues Get Filled Up

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.



Issues (4/5): Cheating

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.

Issues (4/5): Cheating

Applying graphical model to a cheating host:

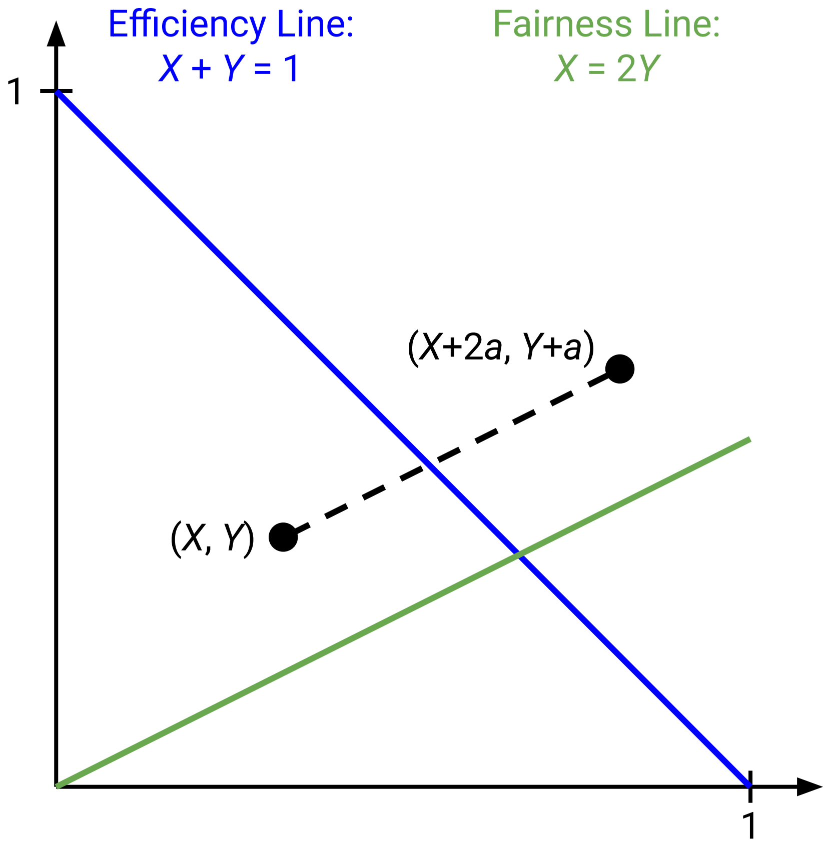
- X increases by 2 per RTT.
- Y increases by 1 per RTT.

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.

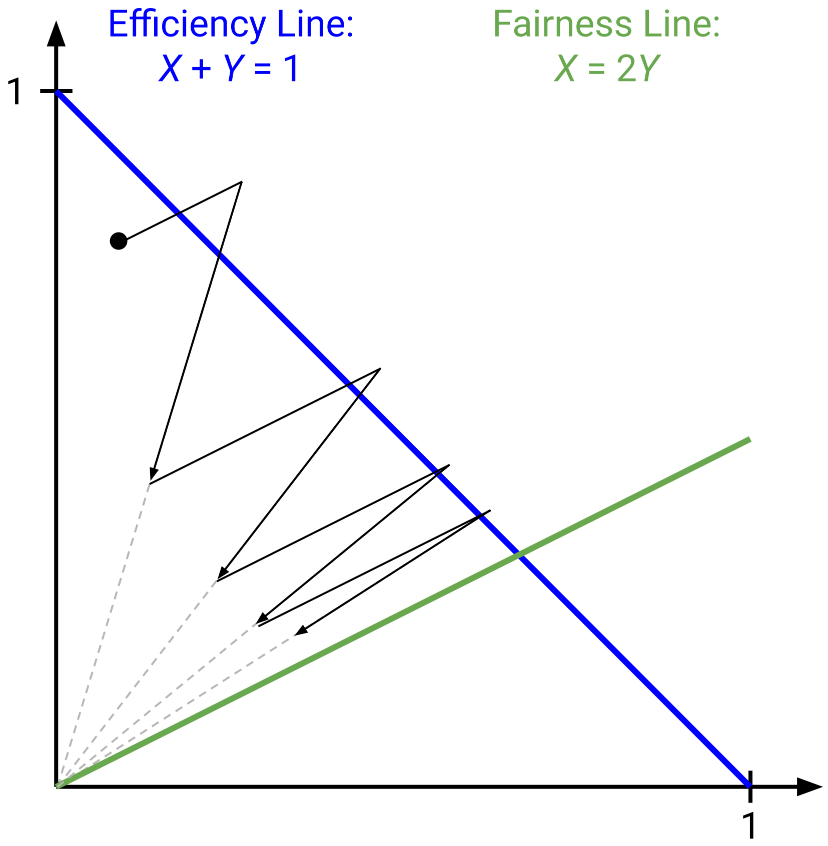
- Fairness/efficiency lines now meet at (2/3, 1/3).



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.



Issues (4/5): Cheating

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 2025

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.)

Enforcing Fairness

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?

Round-Robin (Equal Packet Size)

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 |

Round-Robin (Equal Packet Size)

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**.

Max-Min Fairness (Equal Packet Size)

Formal definition:

- Input: Total available bandwidth is C.
- Input: Each flow i has a bandwidth demand, r_i.
- Output: Find a fair allocation a_i to each flow i.

Max-min allocations are defined as:

- $a_i = \min(f, r_i)$, where f is the unique value such that $\sum a_i = C$.

Intuition:

- f is the fair share (same value for everybody) if you don't get full demand.
- The min term ensures that nobody gets more than they asked for.
- The sum ensures that all capacity is used.

Max-Min Fairness (Equal Packet Size)

Max-min allocations are defined as:

- $a_i = \min(f, r_i)$, where f is the unique value such that $\sum a_i = C$.

Applied to the previous example:

- $C = 10$
- $r_1 = 8$ $r_2 = 6$ $r_3 = 2$

Solution:

- The fair share is $f = 4$.
- $a_1 = \min(4, 8) = 4$
- $a_2 = \min(4, 6) = 4$
- $a_3 = \min(4, 2) = 2$

Round-Robin (Unequal Packet Size)

How do we deal with packets of different sizes?

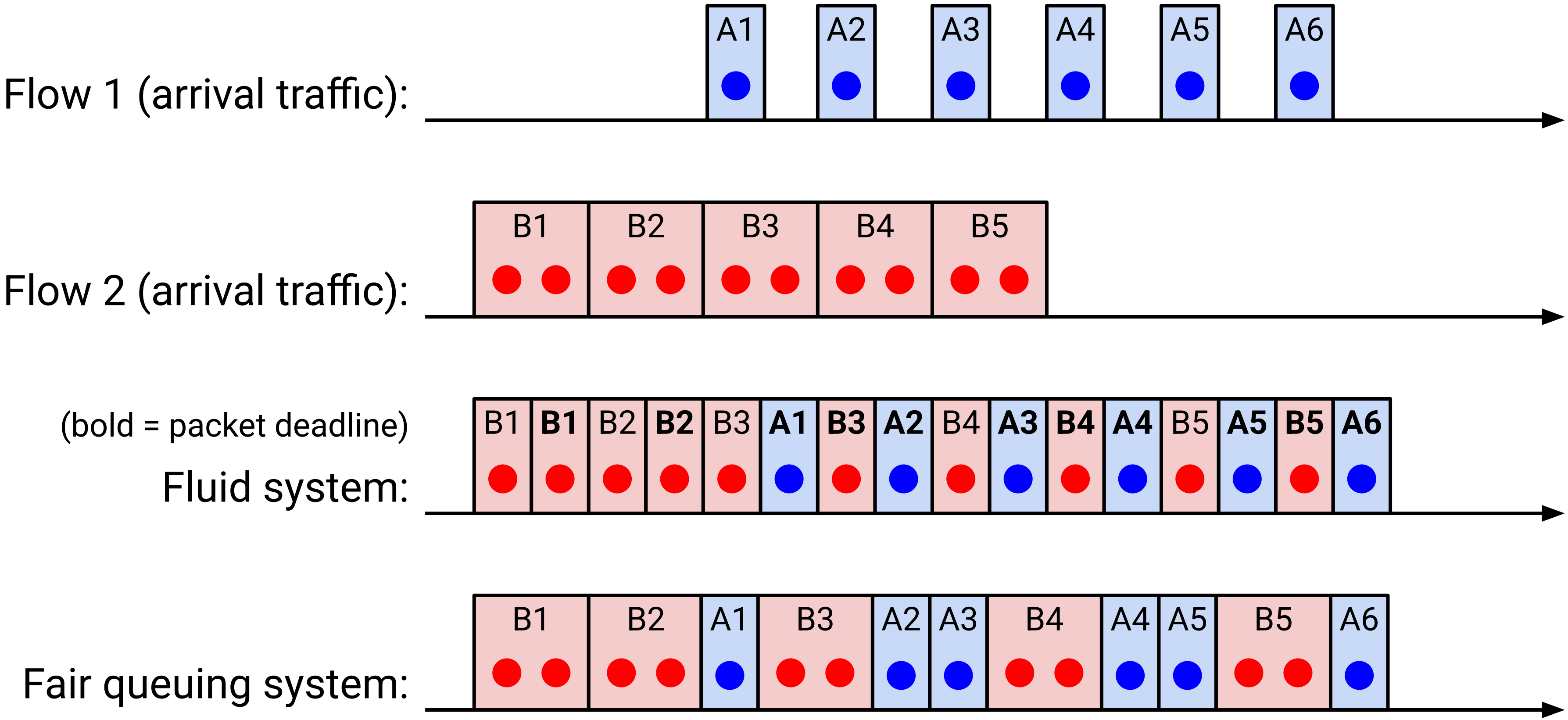
- Mental model: bit-by-bit round-robin ("fluid flow").
- We can't actually do this in practice!
- But we can approximate it. This is what **fair queuing** routers do.

Fair queuing:

- For each packet, compute the time when the last bit of the packet would have left the router, if flows were served bit-by-bit.
- This time is called the *deadline* for that packet.
- Then, serve packets in order of their deadlines.

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

Fair Queuing (Unequal Packet Size)



Fair Queuing in Practice

Perfect fair queuing is too complex to implement at high speeds.

But several approximations exist.

- Example: Deficit Round Robin (DRR)

Today:

- Routers typically implement approximate fair queuing (e.g. DRR).
- Routers only use a small number of queues.
  - This results in coarser-grained isolation.
  - Example: Separate queues per customer (not per flow).

Fair Queuing vs. FIFO Queues

Fair queuing pros:

- Isolation: Cheating flows don't benefit.
- Bandwidth share doesn't depend on RTT.
- Flows can pick any rate adjustment scheme they want.

Fair queuing cons:

- More complex than FIFO.
  - Separate queues per flow.
  - Additional bookkeeping per packet.
- Can only be approximated in practice.

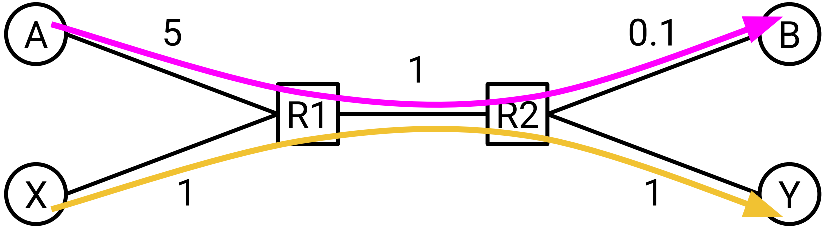
Fair Queuing Does Not Solve Congestion

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.



Fair Queuing: Philosophy of Fairness

Fair queuing gives us per-flow fairness. But is that really what we want?

- What if you have 8 flows, and I have 4?
  - Why do you get twice the bandwidth?
- What if your flow goes over 4 congested hops, and mine only goes over 1?
  - Shouldn't you be penalized for using more scarce bandwidth?
- What granularity should we enforce fairness at?
  - Per TCP connection?
  - Per source-destination pair?
  - Per source?

Fair queuing is a great way to ensure *isolation*.

- Ensures that no one person hogs all the bandwidth, even in the worst cases.

Router-Assisted Congestion Control

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.

Router-Assisted Congestion Detection

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.