Lec 15 - Congestion Control

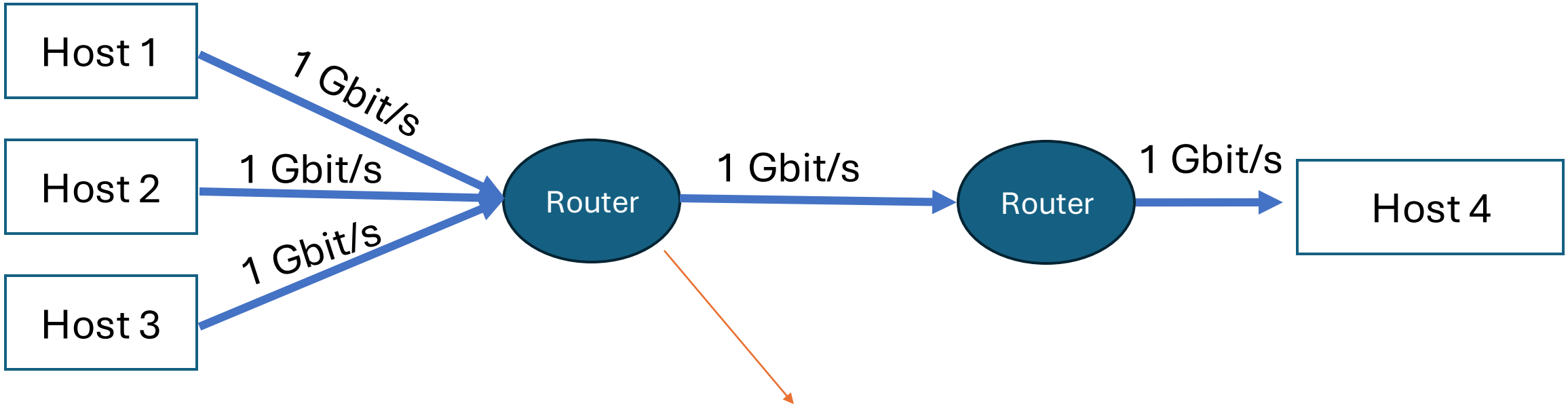
Lecturer: Venkat Arun

Chapter 6.3 in the book

Why Congestion Control?

- **Problem:** Network links do not have infinite bandwidth. Sometimes, applications want to send data faster than the network can handle.
  - How do senders know to slow down?
  - How much should they slow down?
  - If multiple hosts are sharing the same bandwidth, the *total* rate at which they can send is limited. In this case, who sends how much?

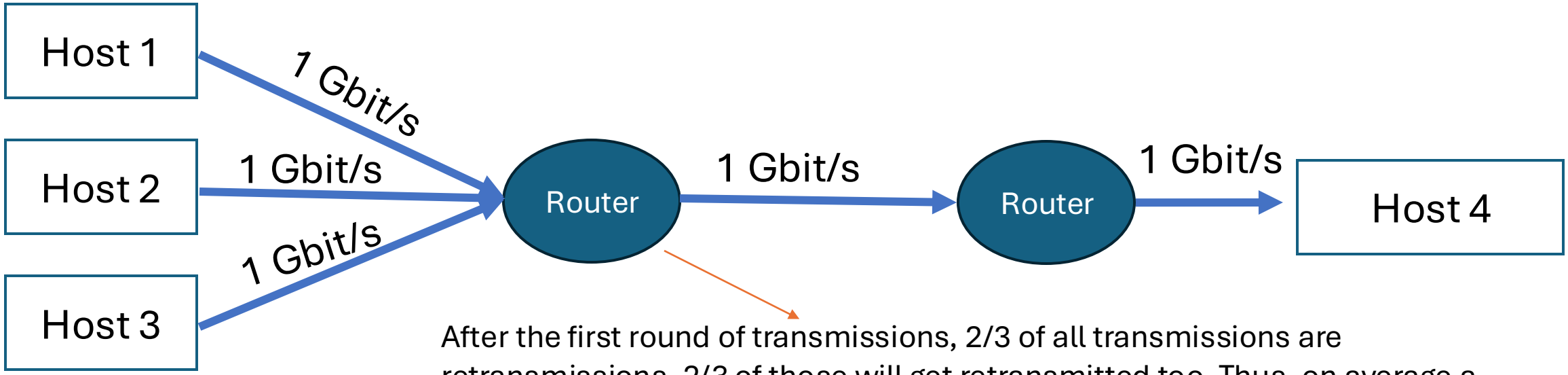
What happens if everyone sends at the maximum rate that they can?



If everyone is sending at 1 Gbit/s, this router will be forced to drop 1/3 of the packets

Is it so bad though? Why can't we just retransmit?

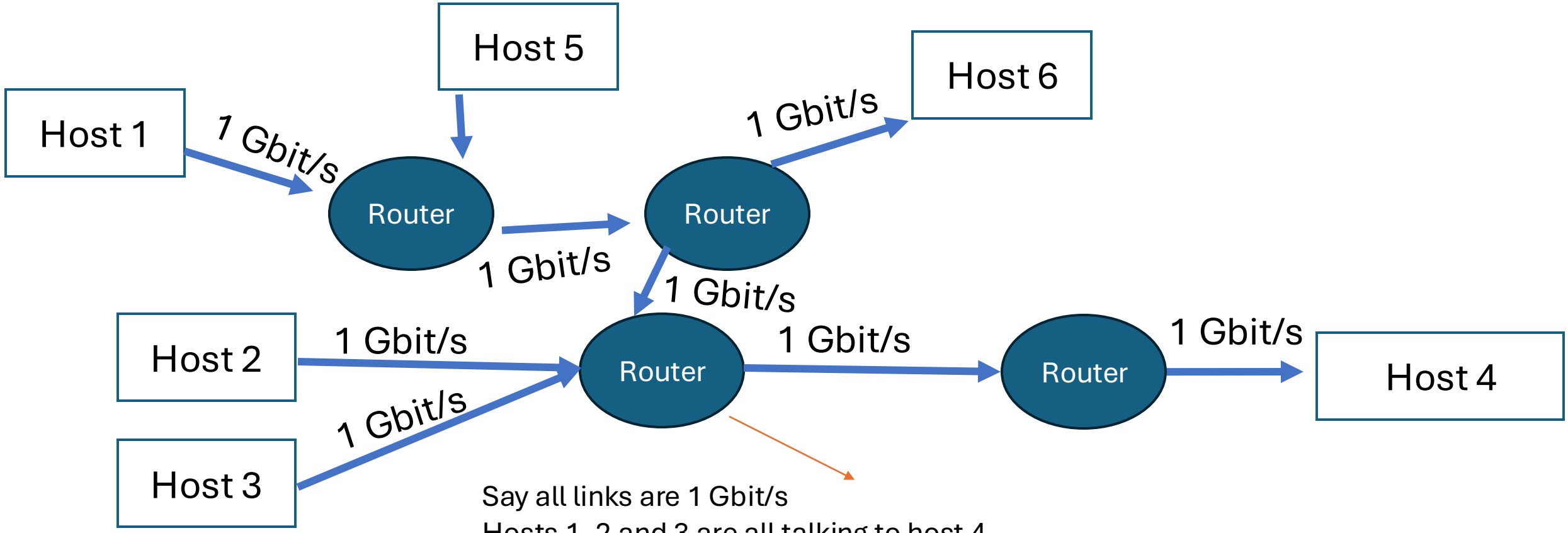
Problem with bursting: congestion collapse



After the first round of transmissions, 2/3 of all transmissions are retransmissions. 2/3 of those will get retransmitted too. Thus, on average a packet will have to be transmitted 3 times before it goes through successfully.

What is the problem though? Who is it harming really? Everyone is still getting a net of 1/3 Gbit/s as long as we never incorrectly retransmit packets

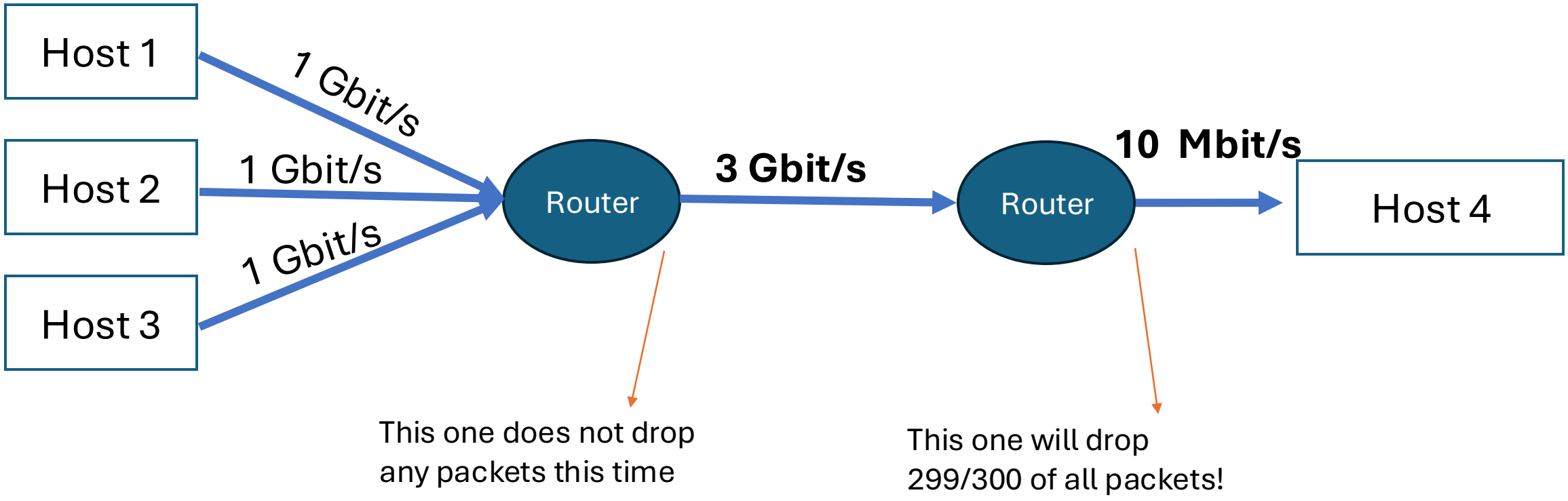
Problem with bursting: congestion collapse



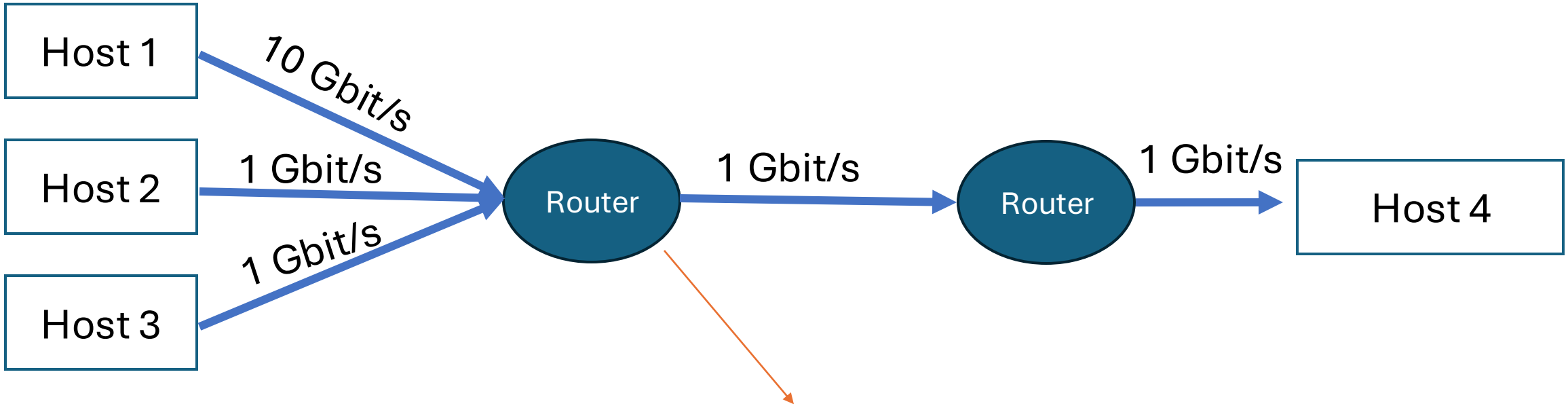
Say all links are 1 Gbit/s
Hosts 1, 2 and 3 are all talking to host 4
Host 5 is talking to host 6

In this case, half the packets between the communication between Host 5 and 6 could have happened at 2/3 Gbit/s if host 1 had slowed down to its actual bandwidth. But now, it can only happen at ½ Gbit/s

Problem with bursting: almost all packets may be dropped



Problem with bursting: it may be unfair

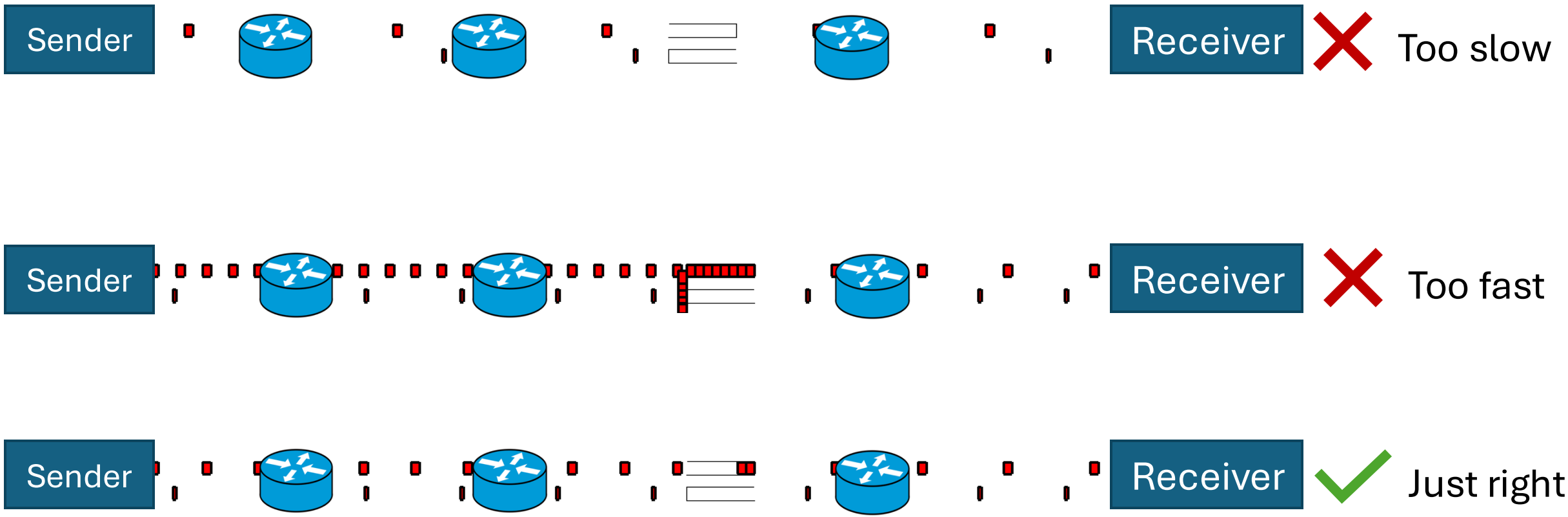


If host 1's incoming link is 10Gbit/s, it will likely get much more than 1/3 of the bandwidth.
Plus, even if all the incoming bandwidths are the same, which packets get dropped is determined in a messy way

Lecture Strategy Today

- **Goal:** Everyone should send at their "fair" rate and the net should never exceed the bandwidth of any link on the network (too much)
- In this lecture, we will study a narrow version of congestion control. While narrow, it serves as a crucial building block for almost all forms of congestion control
- We make the following assumptions:
  - Congestion control (CC) is performed entirely at the senders by TCP. Thus, it is fully decentralized
  - There is only a single congested link along the path of any TCP flow
  - Our notion of "fairness" is that all TCP flows sharing a congested link should get the same bandwidth
  - All routers are fitted with a first-in-first-out (FIFO) queue at the all *outgoing* ports. Packets are dropped when they can no longer fit into this queue.

The problem: finding the right sending rate



Let’s understand the congestion window

Step 1: Revisit the receive window from the sender side

Algorithm implemented at the sender side

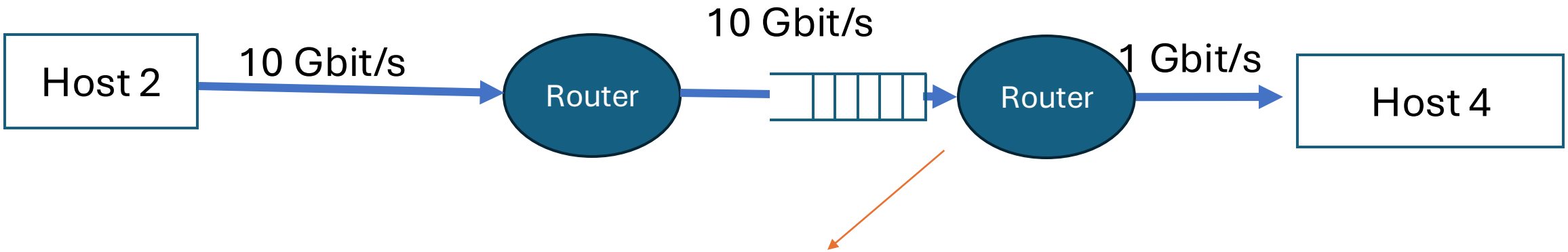
```
# Global variable
inflight = 0
# Receive window, a global variable whose value is set by the receiver in code that is not shown here
rwnd = 10000000
def on_ack(bytes_acked: int, bytes_lost: int)
    # bytes_lost is the loss detected using dupACKs
    inflight = inflight - bytes_acked - bytes_lost
def on_timeout():
    # We assume all bytes are lost
    inflight = 0
def on_send(bytes_sent: int)
    inflight += bytes_sent

# Whenever inflight < rwnd, send packets until inflight = rcv_window.
Which bytes should we put into the packet? If there are bytes we believe are lost, retransmit those first. Else send new bytes
```

Step 2: invent a new variable called congestion window

- Let us call the congestion window as "cwnd". Instead of using just the receive window in the previous algorithm, use min(cwnd, rwnd)
- The sender plays with cwnd to adjust its sending rate. The smaller cwnd is, the slower it will send.
- Why use cwnd and not just limit packets per second?
  - Modern algorithms do this too, but cwnd is easy to reason about and still used almost universally

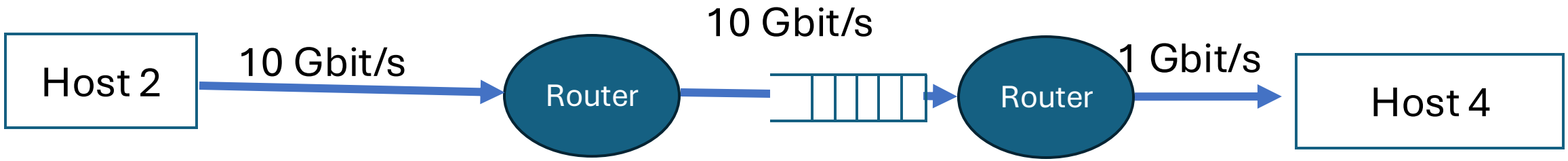
Step 3: understand what cwnd does



A queue will build up here if ingress rate is greater than egress rate

Q: How large will the queue be?

How large will the queue be? – intuitive way



- We will look at it in two ways: intuition and algebra. Intuition first
- There are cwnd unacknowledged packets (by definition of cwnd)
- Assume the queue length is stable (it may not be), then packets are coming into the queue at the same rate as packets going out.
  This is C = 1 Gbit/s
- Let the RTT (round trip time) when the queue is empty be R_m
- C* R_m Packets are in the "pipe", i.e. the non queue part of the network.
  C* R_m is the "Bandwidth Delay Product (BDP)"
- The rest **(cwnd – C * R_m)** are in the queue

How large will the queue be? – the algebra way

Claim 1: The sender will send at a rate cwnd / RTT, where RTT is the Round-Trip Time

Claim 2: The sender will maintain q = max(cwnd – BDP, 0) bytes in the bottleneck buffer, assuming the bottleneck buffer is larger than q

The above two statements are the main thing to know about congestion window

Here, RTT = R_m + time spent in queue. Recall R_m is the RTT when the queue is empty

Claim 1

Claim 1: The sender will send at a rate cwnd / RTT, where RTT is the Round-Trip Time assuming packets do not get lost

At the beginning, suppose inflight begins at 0. The sender can send cwnd bytes before it receives any ACKs. The first ACK arrives an RTT later.

In steady state, inflight=cwnd. What is the rate then? Consider an arbitrary packet P. Consider the period between when P was sent and ACKed. It is 1 RTT long. If we find the number of packets that were sent in that period, X, we are done.

In that period, all packets that were in flight when P was sent would have been ACKed (and therefore as many new packets would have been sent). But we know that cwnd packets were in flight and therefore sent. Thus the rate is cwnd / RTT

Claim 2:

Claim 2: The sender will maintain $q = \max(cwnd - BDP, 0)$ bytes in the bottleneck buffer, assuming the bottleneck buffer is larger than q

Note 1: the RTT is not constant. Let R_m be the RTT when the queue is empty and let C be the link rate. Then RTT, $R = R_m + q / C$ because it takes 1/C seconds to dequeue every byte (bytes are dequeued in units of packets, but we'll ignore that detail)

Note 2: In steady state when inflight = 0, sending rate <= C. Why? Because sending rate equals ACK rate and ACK rate <= C

Case 1 (sending rate = C): This means

$cwnd / R = cwnd / (R_m + q) = C$
Solving this equation, we get, $cwnd = C R_m + q$. Or, $w = cwnd - C R_m = cwnd - BDP$

Claim 2:

Case 2 (sending rate < C): In this case, q = 0 since if q > 0, the link would have been fully utilized.

Thus, cwnd / R_m < C, which implies cwnd < C R_m. Thus, case 1 applies only when cwnd >= C R_m

This means q = max(cwnd – BDP, 0) and we have proved the result.

Another way to think about this is that packets are either “flying” or waiting in queue. Every packet “flies” for R_m seconds. When the link is fully utilized, C R_m can fly. The rest must wait in queue

Note: we assumed in this proof that q exists and is constant. We did not prove this

Takeaways

When using congestion window, it is easy to reason about queue lengths. This is why we use them.

Goals:

- If cwnd > BDP, q > 0 and the link is fully utilized.
- If q = cwnd – BDP < buffer size, packets do not get lost.

Simplest possible congestion control algorithm (CCA)

Run the following every RTT

If a packet got lost in the last RTT

    cwnd -= 1

Else

    cwnd += 1

Problem: Packet will get lost every other RTT

Fix: Additive Increase, Multiplicative Decrease (AIMD)

Run the following every RTT

If a packet got lost in the last RTT

    cwnd -= 1

Else

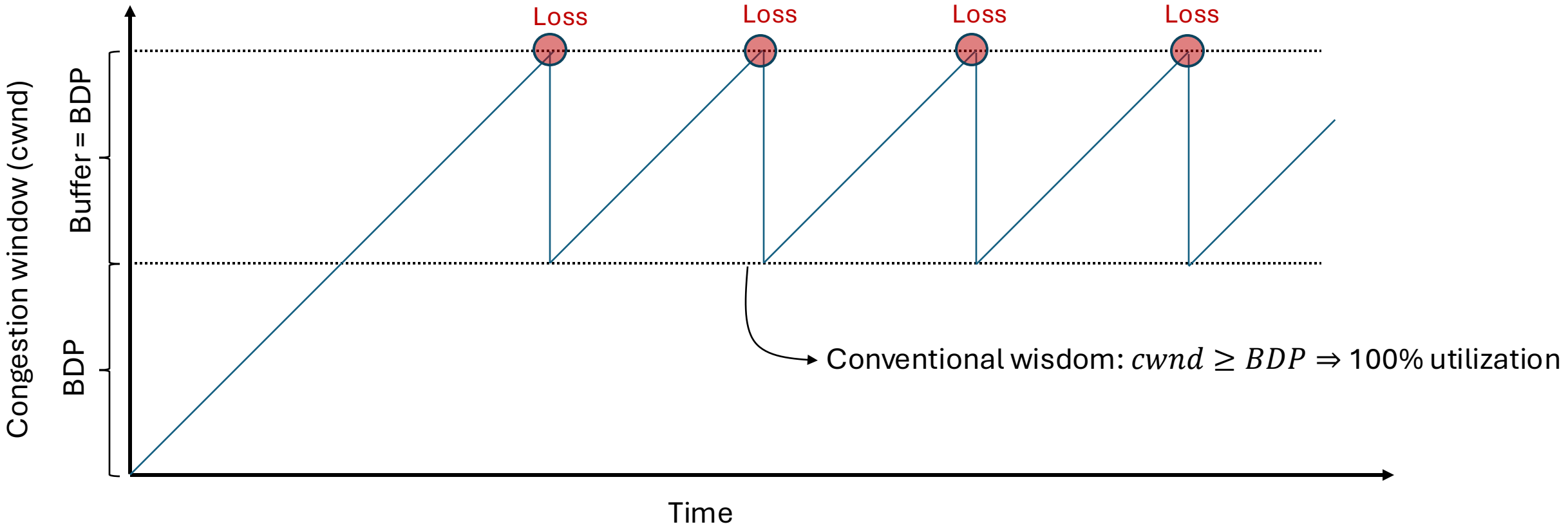
    cwnd /= 2

Why it is good:

- Because we decreased cwnd a lot, it takes time before the next loss. In particular, it takes cwnd / 2 RTTs before the next loss
- Is the utilization high? Recall, we want cwnd > C R_m.
  - Thus, if loss happens when cwnd > 2 C R_m, then utilization will be high.
  - Loss happens when q >= buffer. That is, loss_cwnd > buffer + C R_m
  - Thus, we want buffer to be large enough that loss_cwnd > 2 C R_m. This means, buffer + C R_m > 2 C R_m
  - As long as buffer > C R_m we are fine.

AIMD

Conventional wisdom: buffer≥BDP ensures 100% utilization



AIMD Takeaways

Takeaway 1: As long as buffer size > BDP, we get 100% utilization.

Takeaway 2: As a bonus, AIMD is fair when multiple people are sharing the bottleneck. Intuitively, all flows increase at the same rate, but flows with the larger cwnd decreases more.

Suppose two flows have cwnd X and Y. We assume that either both flows lose a packet or neither flow loses a packet. Let us consider how X – Y evolves in the two cases:

- Both flows increase additively: X – Y becomes (X + 1) – (Y + 1) = X – Y
- Both flows decrease multiplicatively: X – Y becomes (X / 2) – (Y / 2) = (X – Y) / 2, which decreases the gap

AIMD is the canonical CCA and a version of it is called TCP Reno and TCP NewReno. It is rarely used today, but its descendants are

There are a bazillion ways to do congestion control, each with its own tradeoffs