Transport
Part II

Lecture 6, Computer Networks (198:552)

Fall 2019

RUTGERS
UNIVERSITY | NEW BRUNSWICK

Two Main Transport Layers

- User Datagram Protocol (UDP)
  - Abstraction of independent messages between endpoints
  - Just provides demultiplexing and error detection
  - Header fields: port numbers, checksum, and length
  - Low overhead, good for query/response and multimedia

- Transmission Control Protocol (TCP)
  - Provides support for a stream of bytes abstraction

Transmission Control Protocol (TCP)

- Multiplexing/demultiplexing
  - Determine which conversation a given packet belongs to
  - All transports need to do this

- Reliability and flow control
  - Ensure that data sent is delivered to the receiver application
  - Ensure that receiver buffer doesn’t overflow

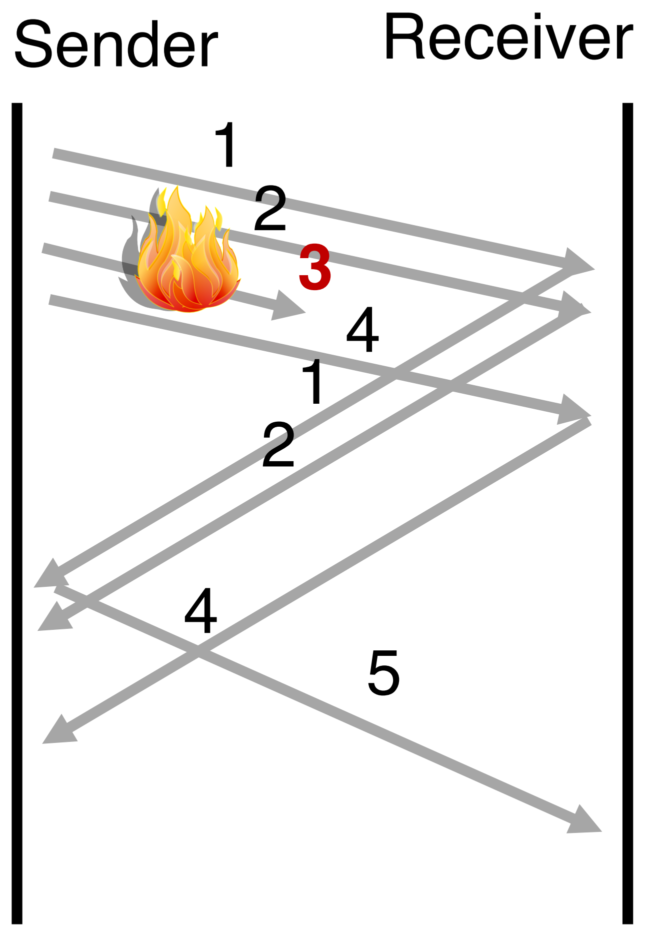
- Ordered delivery
  - Ensure bits pushed by sender arrive at receiver app in order
  - Q: why would packets ever be received out of order?

- Congestion control
  - Ensure that data sent doesn’t overwhelm network resources
  - Q: which network resource?

Ordered Delivery

Reordering packets at the receiver side

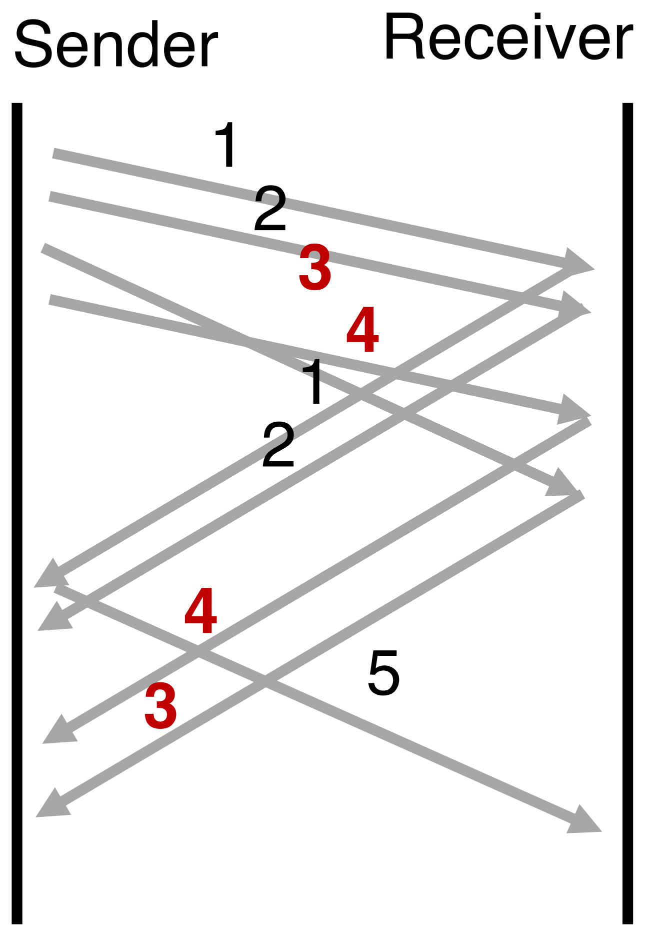
- Let's suppose receiver gets packets 1, 2, and 4, but not 3 (dropped)
- Suppose you're trying to download a Word document containing a report
- What would happen if transport at the receiver directly presents packets 1, 2, and 4 to the Word application?



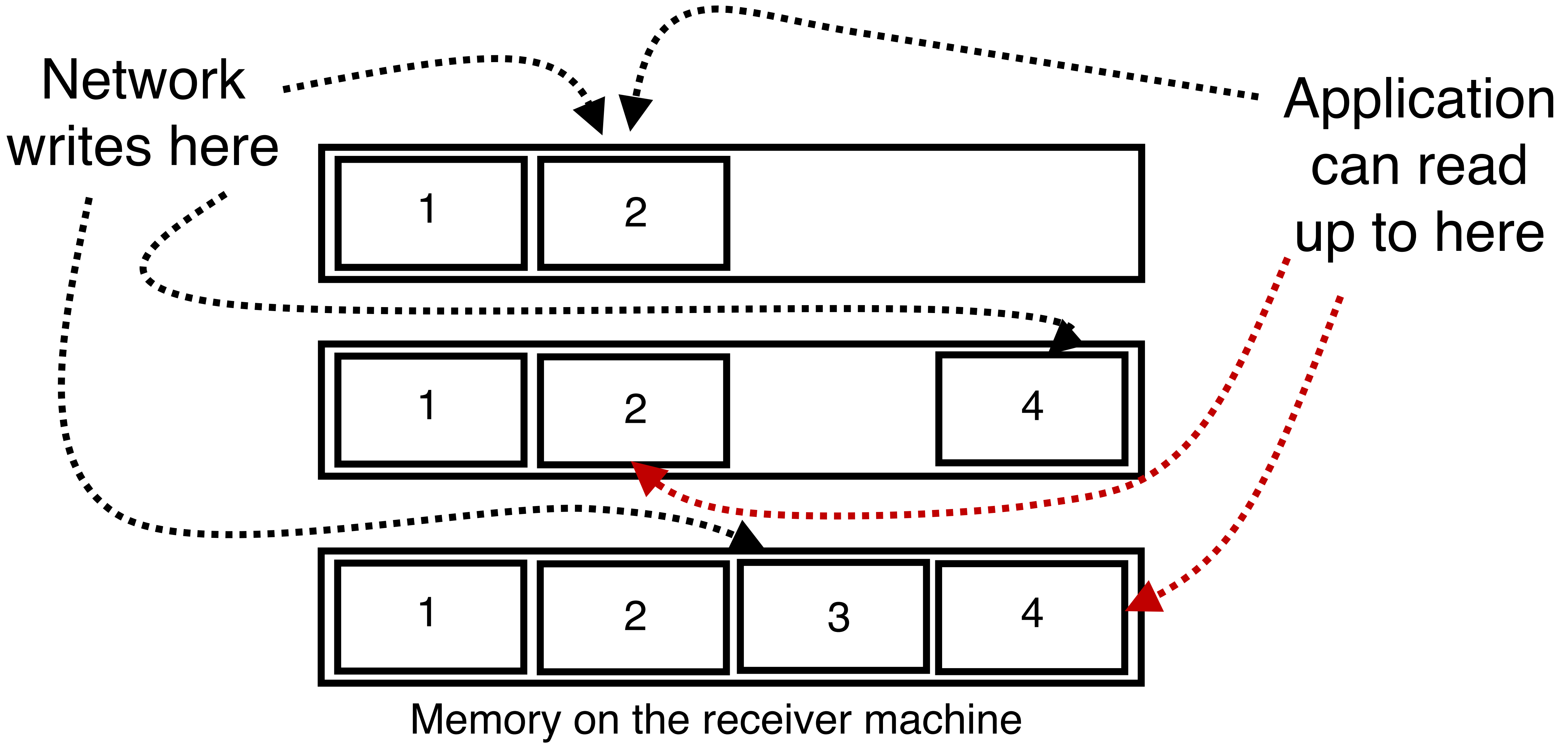
Reordering at the receiver side

- Reordering can also happen due to packets taking different paths through a network

- Receiver needs a general strategy to ensure that data is presented to the application in the same order of sender side bytes pushed



Buffering at the receiver side


Network writes here
Application can read up to here
1
2
1
2
4
1
2
3
4
Memory on the receiver machine

Buffering at the receiver side

- The TCP receiver uses a memory buffer to hold packets until they can be read by the application in order
- This process is known as TCP reassembly

Implications of ordered delivery

- Packets cannot be delivered to the application if there is an in-order packet missing from the receiver's buffer
  - The receiver can only buffer so much out-of-order data
  - Subsequent out-of-order packets dropped

- It doesn't matter that the packets successfully arrive at the receiver NIC from the sender over the network

- TCP application throughput will suffer if there is too much packet "reordering" in the network

Implications of ordered delivery

- A TCP sender can only send as much as the free receiver buffer space available before packets are dropped at the receiver
  - This number is called the receiver window size
  - TCP is said to implement flow control

Flow control headers

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

                         TCP Header Format

       Note that one tick mark represents one bit position.
```

Implications of buffering at receiver side

- Flow control has implications for TCP data delivery, even when packets arrive in order
- Typically TCP packets arrive in a burst
    - Why?
    - The receiver application won’t read this data in one shot
- Q: What’s the size of the maximum burst?

Sizing the receiver window

- Bandwidth-delay product: enough data to “fill the pipe”

- Implications to achieve high throughput:
  - More memory required at high bandwidth
  - More memory required at high RTT
  - Consider 100 Gbit/s connection at 100 ms RTT

- With this window size, can you guarantee that you will never “block” the connection due to a filled-up receiver buffer?
  - You can’t! If app never reads from receiver buffer, it will fill up and not allow any more data to come in.

Implications of ordered delivery

- What if packets travel from sender to receiver over multiple paths?

- Imagine a situation where one path is much faster than another

- First (faster) path sends packets: 1, 3, 5, …
- Second (slower) path sends packets: 2, 4, 6, …

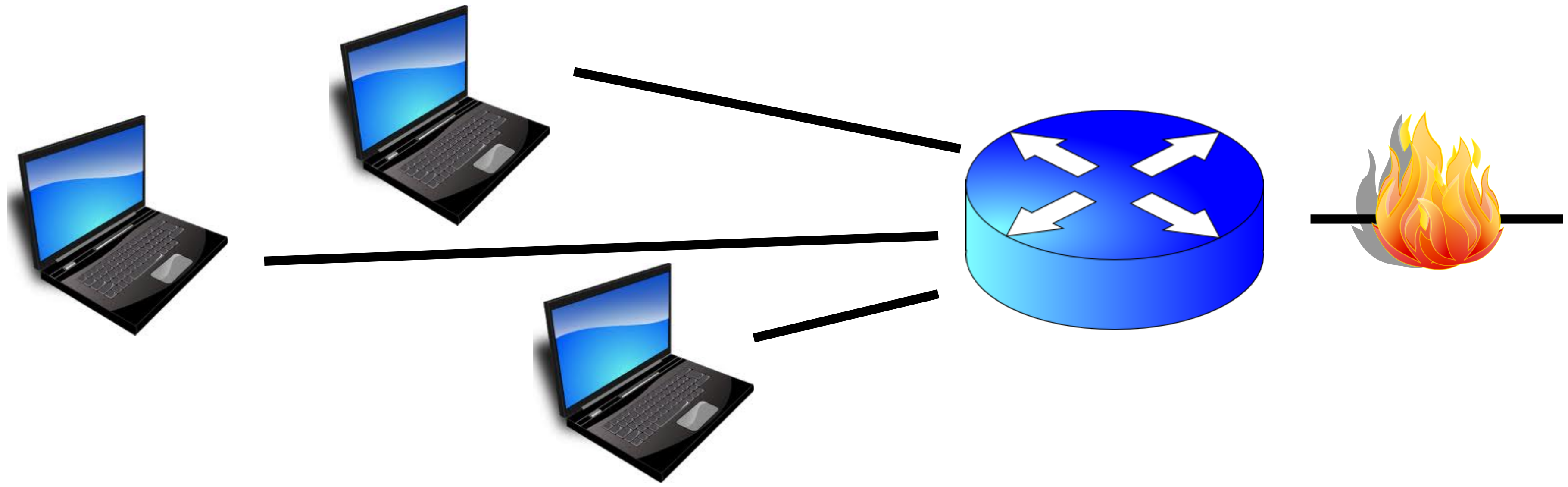
- Reassembly will require dropping the connection’s throughput to match the slower one

Implications of ordered delivery

- Reordering and reassembly are bad for application throughput
- Most network-level load balancing mechanisms avoid per-packet multi-path forwarding
  - Balance load at per-flow or per-flowlet (burst) granularity
- Multi-path TCP variants exist, but all need to solve the path scheduling problem:
  - Schedule outgoing packet transmissions and adjust windows
  - … so that the packets arrive at the receiver (roughly) in order

Congestion control

How should multiple endpoints share net?



- It is difficult to know where the bottleneck link is
- It is difficult to know how many other endpoints are using that link
- Endpoints may join and leave at any time
- Network paths may change over time, leading to different bottleneck links (with different link rates) over time

The approach that the Internet takes is to use a distributed algorithm to converge to an efficient and fair outcome.

The approach that the Internet takes is to use a distributed algorithm to converge to an efficient and fair outcome.

No one can centrally view or control all the endpoints and bottlenecks in the Internet.

Every endpoint must try to reach a globally good outcome by itself: i.e., in a distributed fashion.

This also puts a lot of trust in endpoints.

The approach that the Internet takes is to use a distributed algorithm to converge to an efficient and fair outcome.

If there is spare capacity in the bottleneck link, the endpoints should use it.

The approach that the Internet takes is to use a distributed algorithm to converge to an efficient and fair outcome.

If there are N endpoints sharing a bottleneck link, they should be able to get equitable shares of the link's capacity.

The approach that the Internet takes is to use a distributed algorithm to converge to an efficient and fair outcome.

So, how to achieve this?

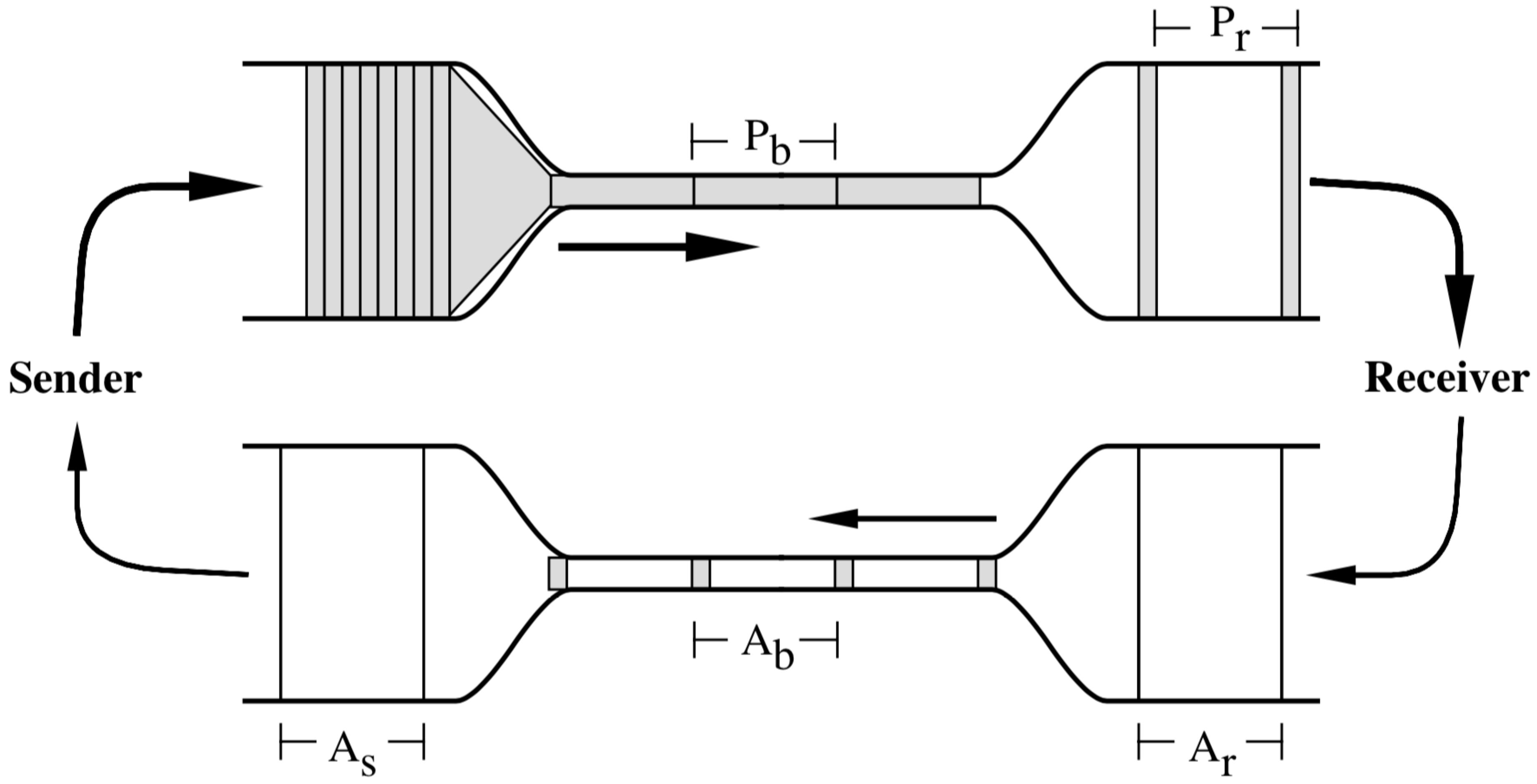
Feedback from network offers clues…

- Signals
  - Packets being dropped (ex, RTO fires)
  - Packets being delayed
  - Rate of incoming ACKs

$\Big\}$ "Implicit" feedback signals (more on explicit signals later)

- Knobs
  - What can you change to "probe" the sending rate?
  - Suppose receiver buffer is unbounded:
  - Let's call the amount of in-flight data per RTT the congestion window
  - Increase congestion window: e.g., by x or by a factor of x
  - Decrease congestion window: e.g., by x or by a factor of x

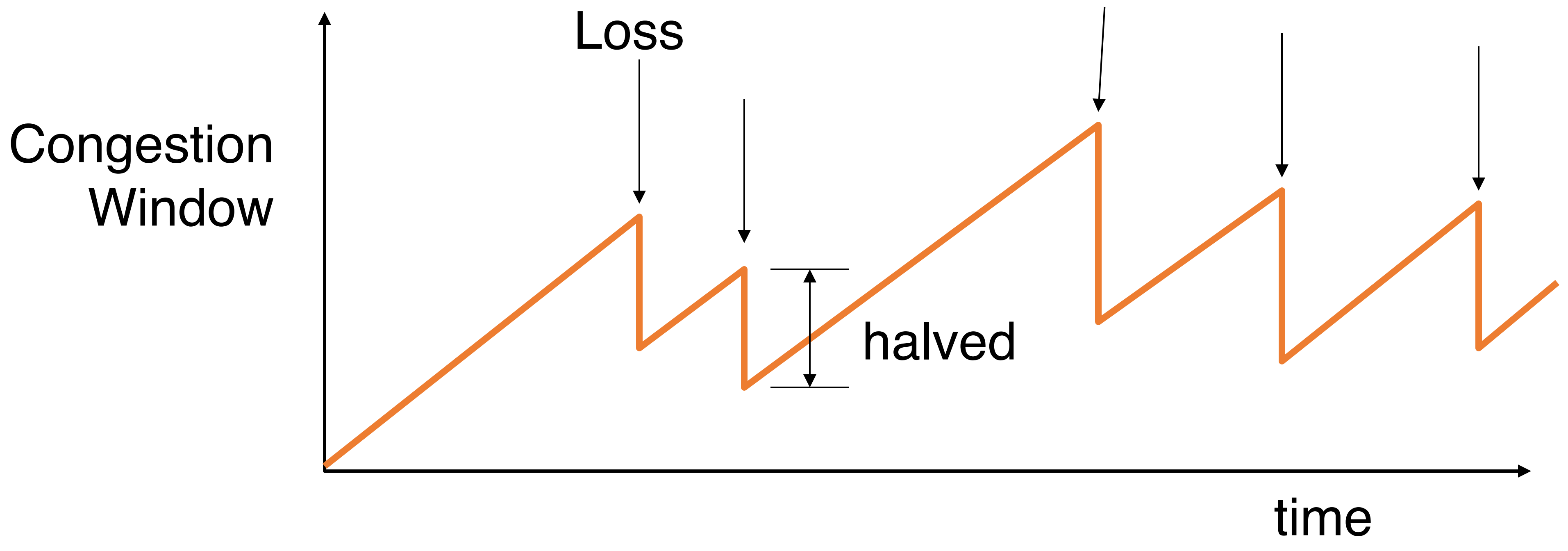
Steady state: ACK clocking



Time for an activity

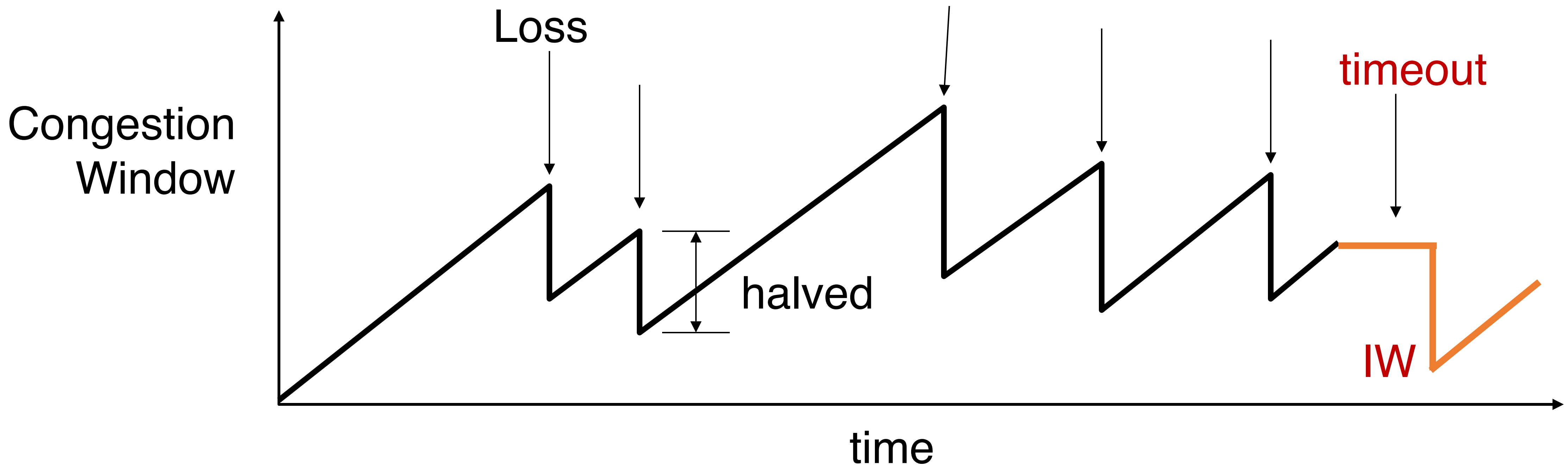
How to get to steady state?

- Slow start
- Congestion avoidance: Additive increase, multiplicative decrease (AIMD)



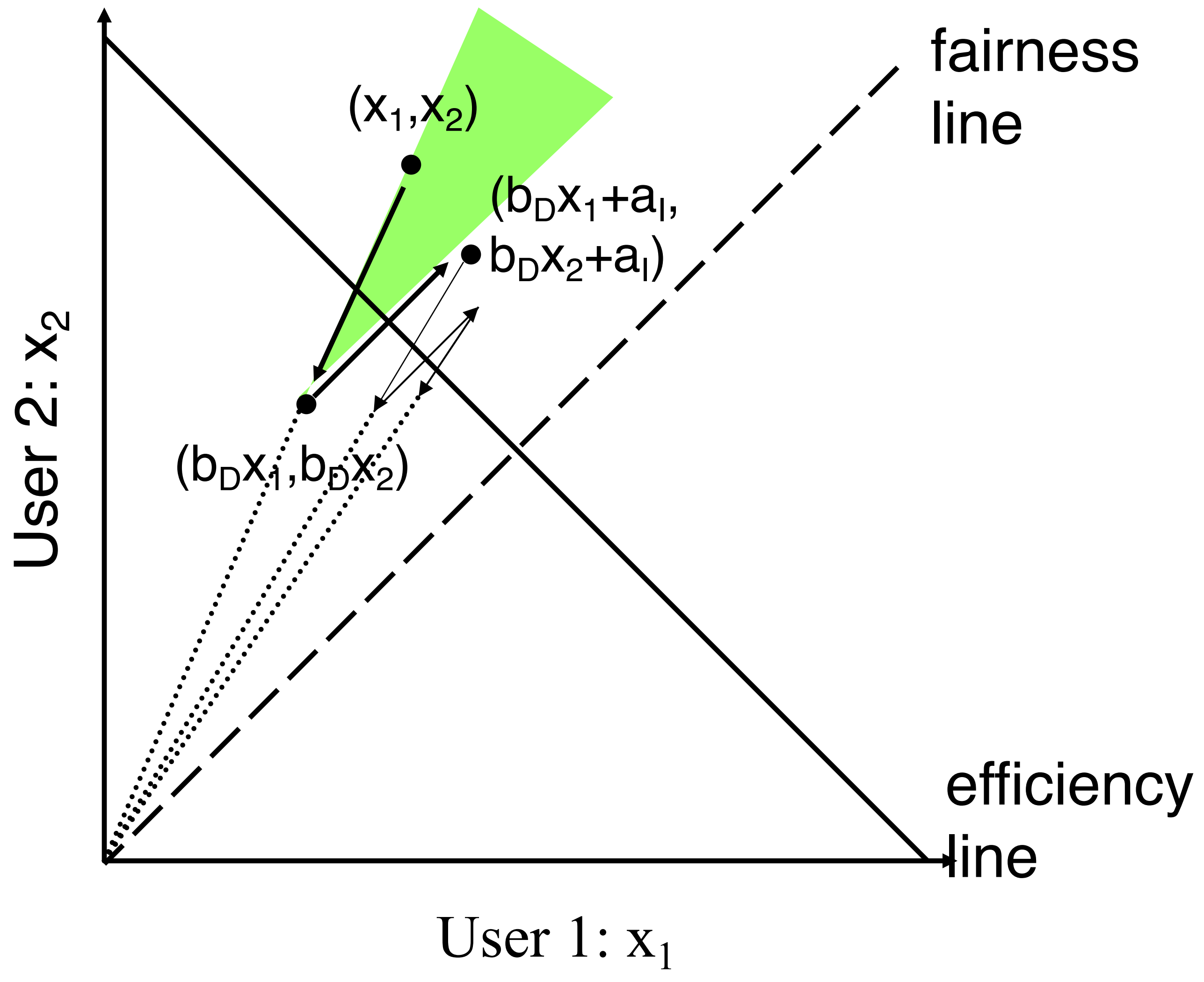
How to get to steady state?

- Upon a timeout, drop the window to a small fixed value (IW)
- Upon idling, drop the window to a small fixed value (RW)



Why AIMD?

- Converges to fairness
- Converges to efficiency
- Increments to rate smaller as fairness increases



TCP’s steady state is not static

- As stated so far, TCP probes network capacity iteratively
- … Until it induces a loss
- … and then probes network capacity again

- It is important to have efficient mechanisms to detect and recover from packet loss

Loss detection & recovery in TCP

- Detecting loss before timeouts occur through fast retransmit
- Basic idea:
    - if the receiver did not receive a segment
    - but did receive a subsequent few segments (duplicate ACKs)
    - … the unreceived packet must have been dropped.

- Fast recovery: Don’t drop the window too much
    - If you’re receiving dup ACKs, packets are being delivered
    - Do congestion avoidance instead of slow start from IW

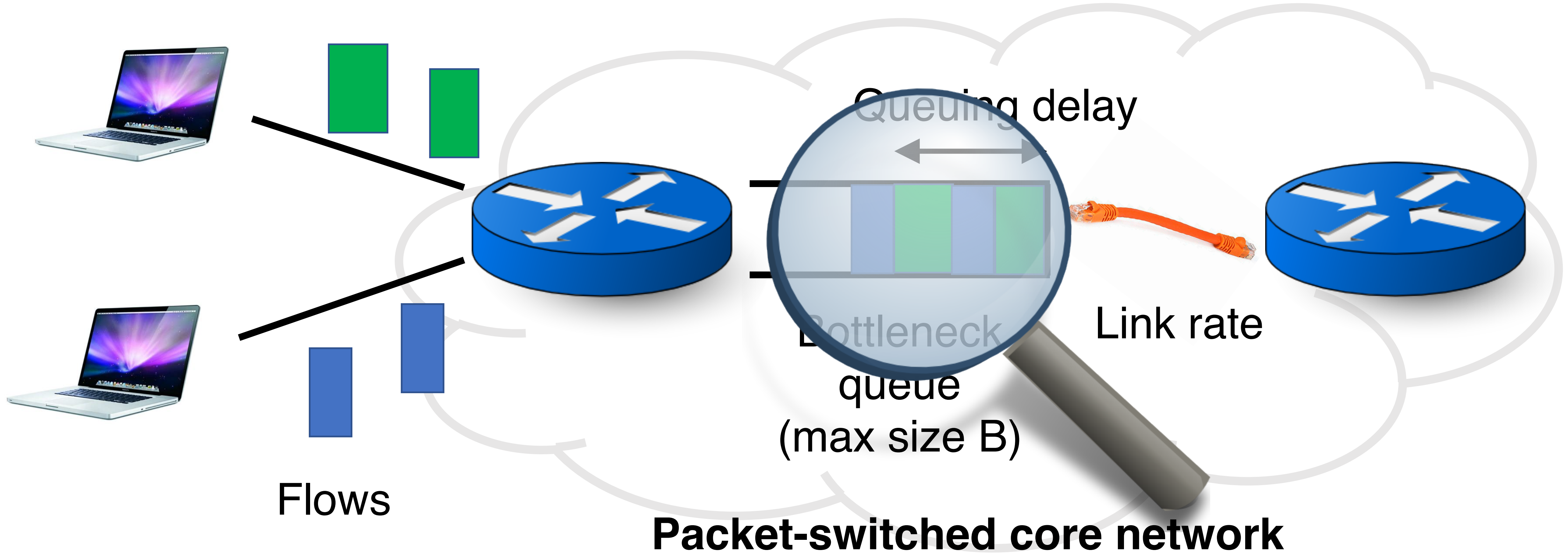
- Many more details in RFC 2581 and follow-on work

Packet Scheduling

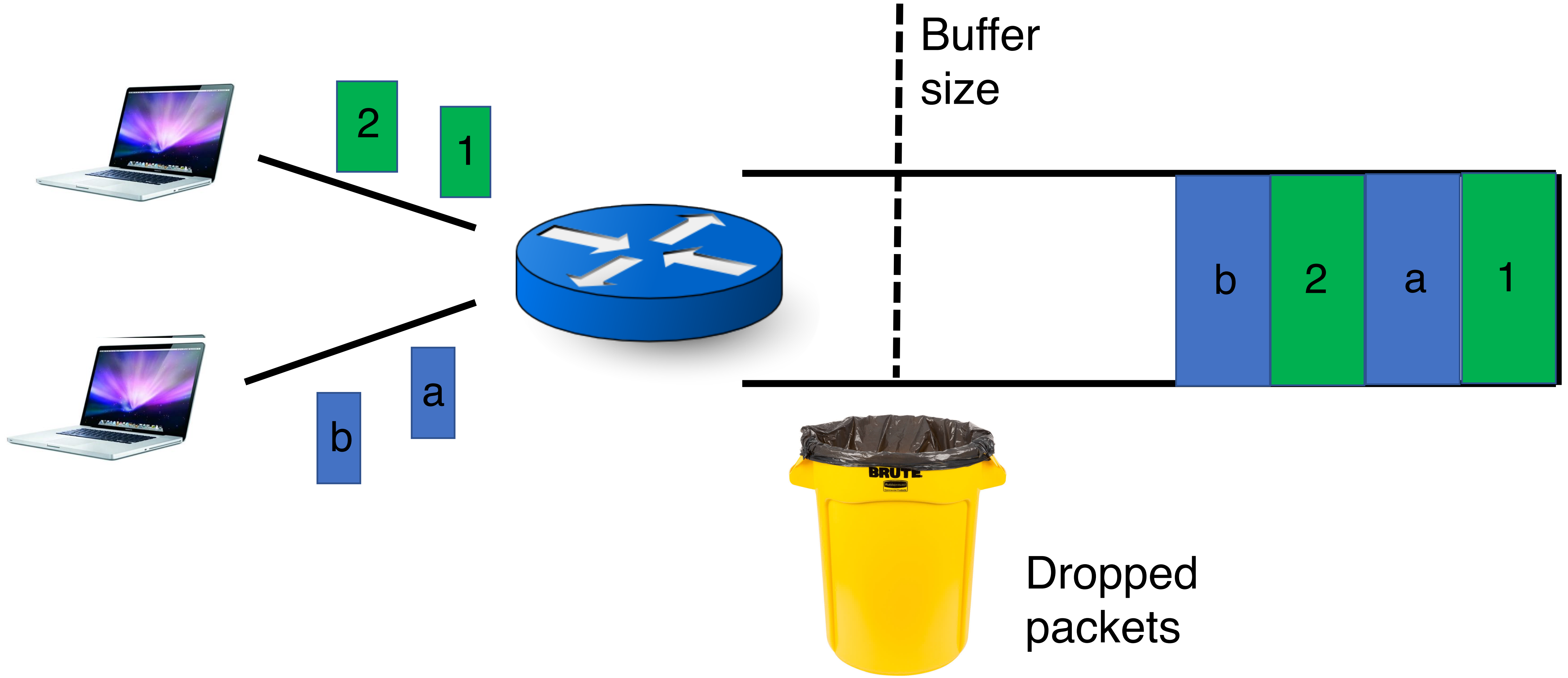
Are endpoint algorithms alone enough?

- What if an endpoint is malicious or buggy?
- Want the network core to do something more about resource allocation than best effort

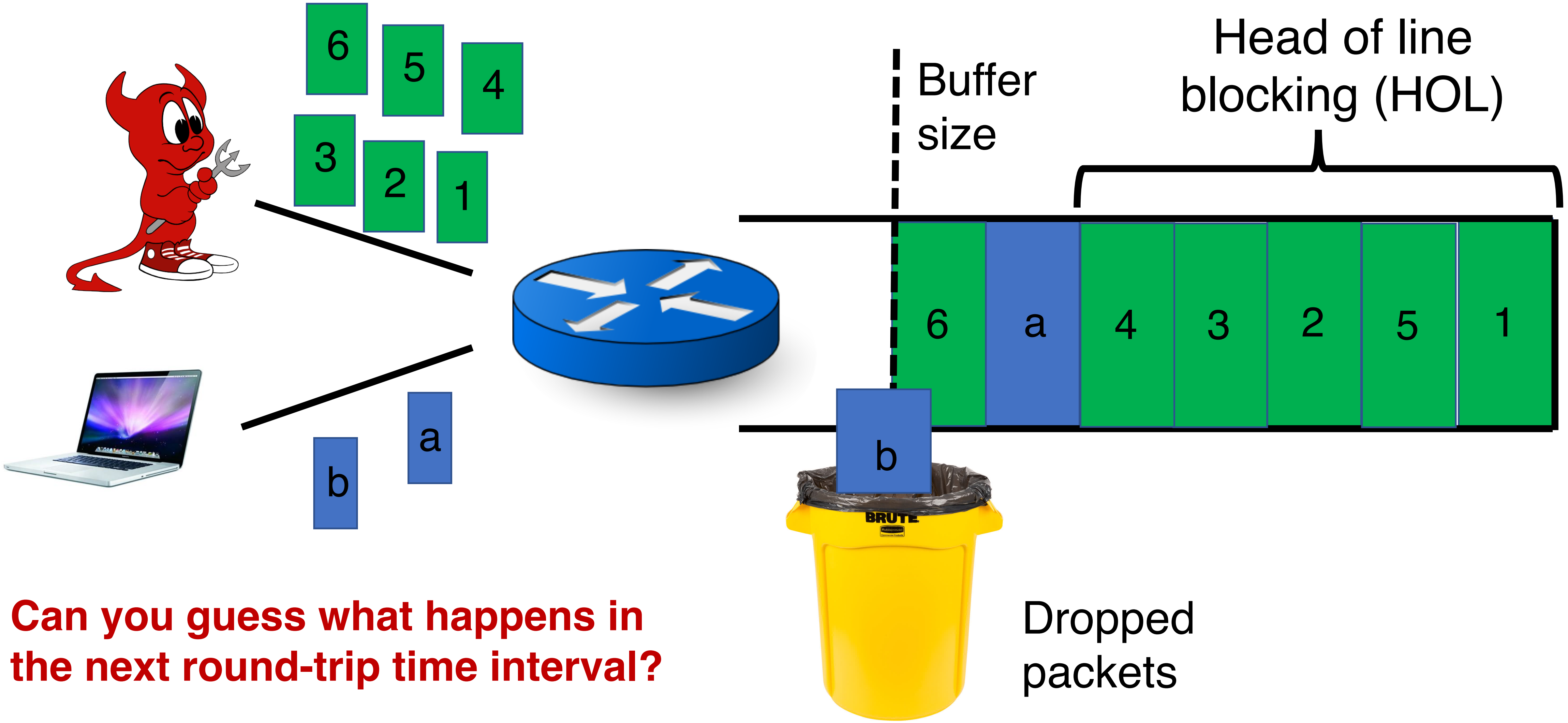
Network model



First-in first-out (FIFO) queue + tail-drop

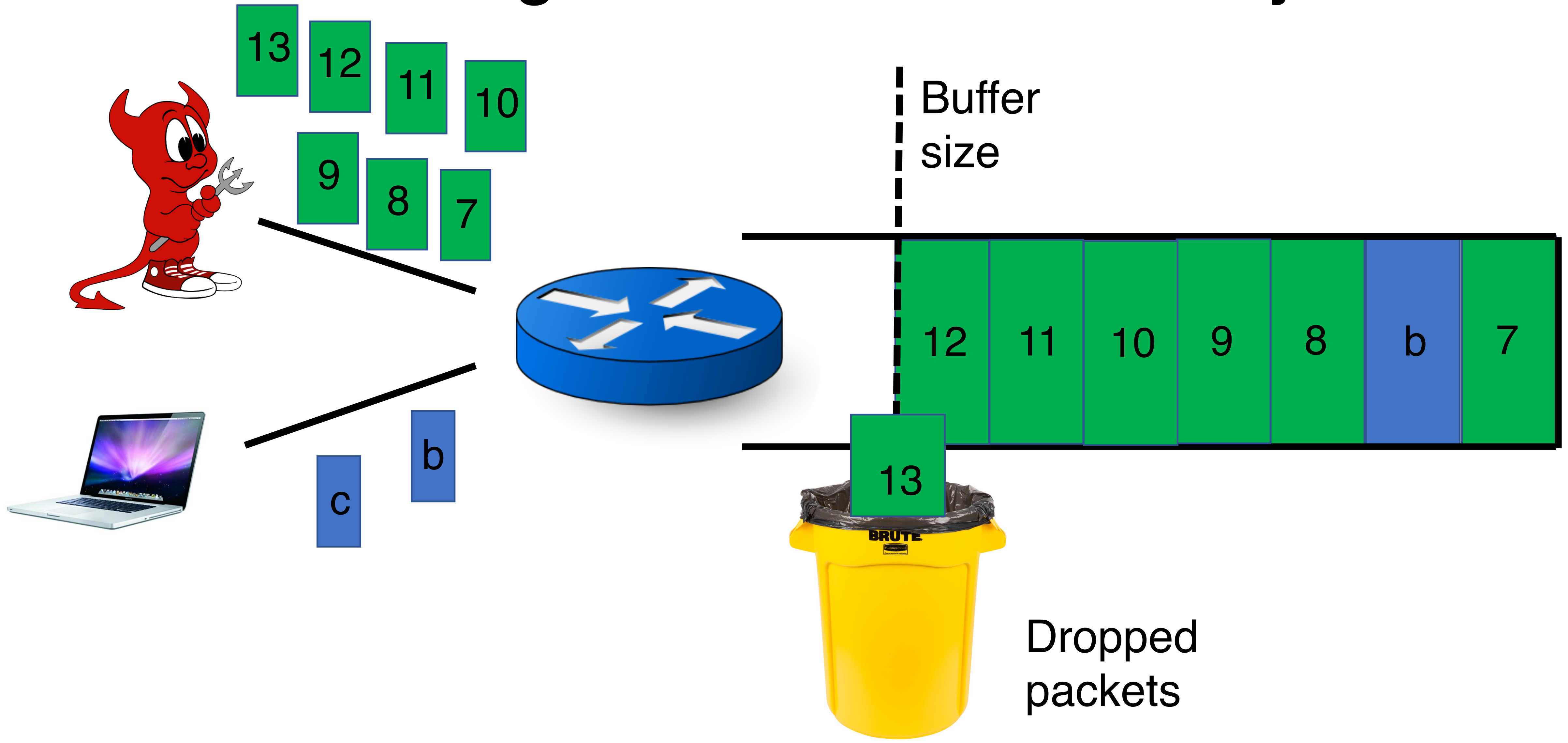


First-in first-out (FIFO) queue + tail-drop

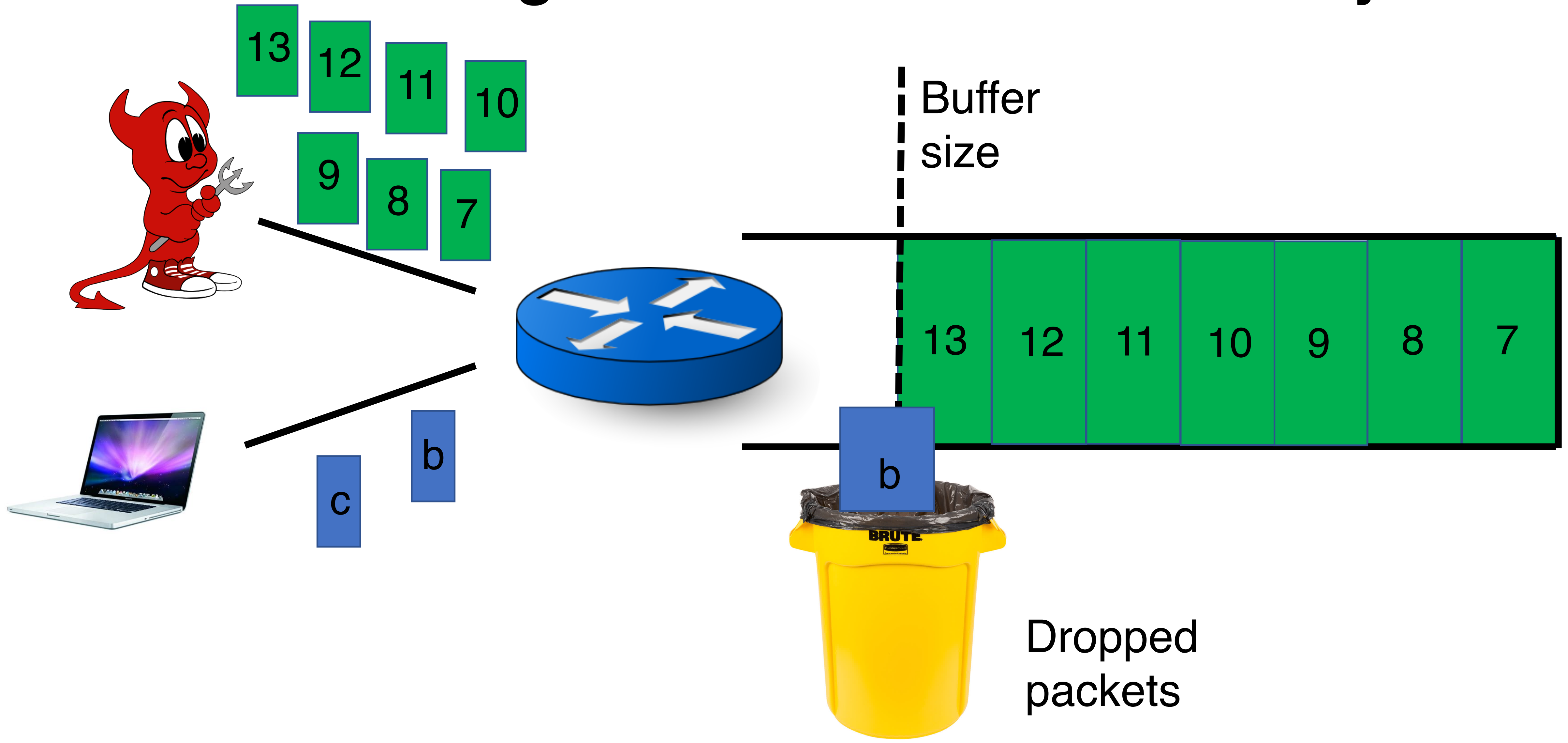


Can you guess what happens in the next round-trip time interval?

ACK-clocking makes it worse: lucky case



ACK-clocking makes it worse: unlucky case



Network monopolized by “bad” endpoints

- An ACK signals the source of a free router buffer slot
    - Further, ACK clocking means that the source transmits again

- Contending packet arrivals may not be random enough
    - Blue flow can’t capture buffer space for *a few* round-trips

- Sources which sent successfully earlier get to send again

- A FIFO tail-drop queue *incentivizes* sources to misbehave!

Packet scheduling on routers

- We will discuss packet scheduling algorithms implemented on routers in detail later in this course.
- Goal: Achieve a predetermined resource allocation regardless of endpoint behavior
- How to make such allocation “efficient”?
  - Implement on routers at high speeds and low cost
  - Achieve equitable sharing of network bandwidth & queues
  - Use available bandwidth effectively