


Computer Networks

CMSC 417 : Spring 2024

COMPUTER SCIENCE
UNIVERSITY OF MARYLAND

Topic: TCP Flow-control (cont..) and Congestion-control (Textbook chapter 5 & 6)

Nirupam Roy

Tu-Th 2:00-3:15pm
CSI 2117

March 26th, 2024

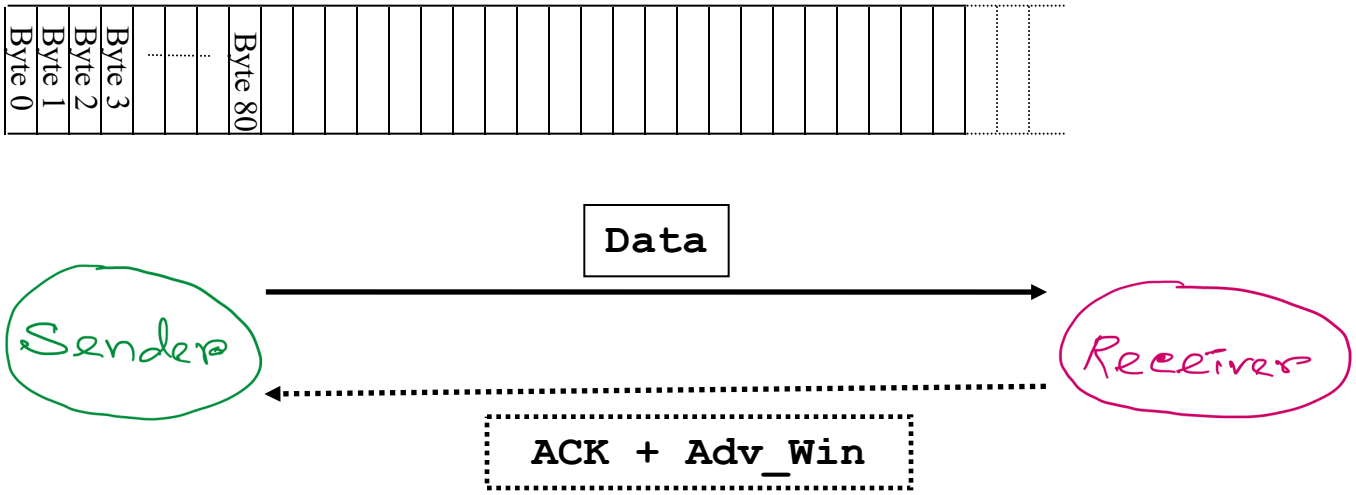



A few points of discussions on TCP flow control

- Silly window syndrome
- Nagle's algorithm (self-clocking)
- Karn-Partridge algorithm

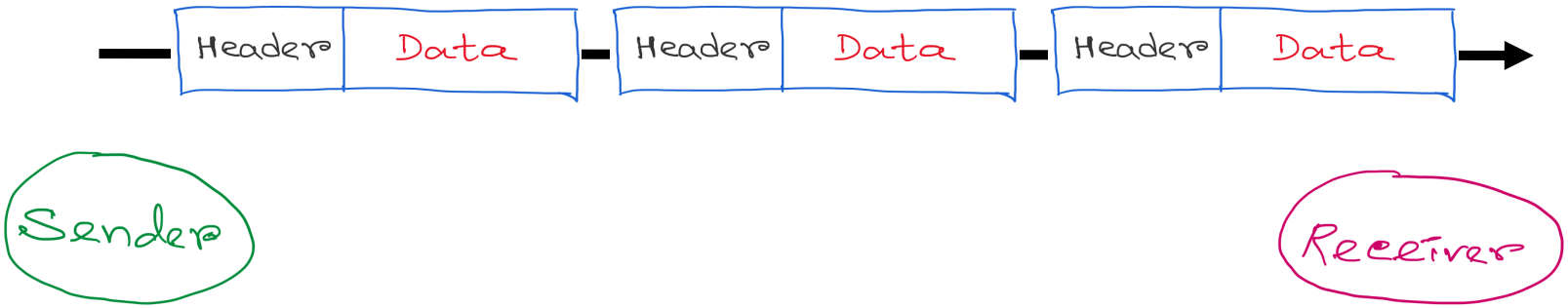
- Silly-window syndrome

Host A



Host B

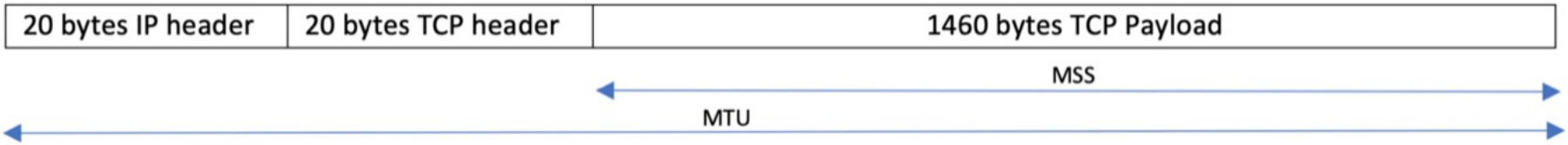
- Silly window syndrome



What happens when the receiving application program consumes data slowly?

Leads to smaller advertised_window

Info: Maximum Segment Size (MSS)

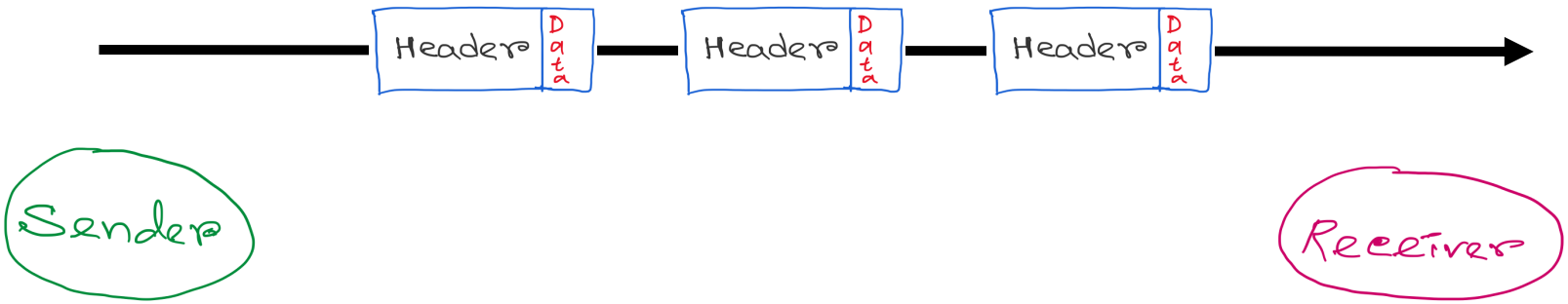


What if…

(1) Adv_window > MSS

(2) Adv_window < MSS

- Silly window syndrome



Discussion: What can be the possible solutions?

Solution 1: Receiver side approach

(1) Delayed ACK (a receiver-side approach)

Solution 1: Receiver side approach

(1) Delayed ACK (a receiver-side approach)

- How much delay is good enough?

Solution 1: Receiver side approach

(1) Delayed ACK (a receiver-side approach)

- How much delay is good enough?

[RFC 1122] A TCP SHOULD implement a delayed ACK, but an ACK should not be excessively delayed; in particular, the delay MUST be less than 0.5 seconds, and in a stream of full-sized segments there SHOULD be an ACK for at least every second segment.

Typically, OSs are more conservative. Use 200ms of delay.

Solution 2: Sender side approach

(2) Nagle's algorithm (a sender-side approach)

```
if there is new data to send then
    if the window size ≥ MSS and available data is ≥ MSS then
        send complete MSS segment now
    else
        if there is unconfirmed data still in the pipe then
            enqueue data in the buffer until an acknowledge is received
        else
            send data immediately
        end if
    end if
end if
```

Solution 2: Sender side approach

Condition 0: Assumes there are some data to send. Otherwise, the sender will have to wait anyway.

```
if there is new data to send then
    if the window size ≥ MSS and available data is ≥ MSS then
        send complete MSS segment now
    else
        if there is unconfirmed data still in the pipe then
            enqueue data in the buffer until an acknowledge is received
        else
            send data immediately
        end if
    end if
end if
```

Solution 2: Sender side approach

Condition 1: The system is capable of sending a full MSS.

Send immediately.

```
if there is new data to send then
    if the window size ≥ MSS and available data is ≥ MSS then
        send complete MSS segment now
    else
        if there is unconfirmed data still in the pipe then
            enqueue data in the buffer until an acknowledge is received
        else
            send data immediately
        end if
    end if
end if
```

Solution 2: Sender side approach

```
if there is new data to send then
    if the window size ≥ MSS and available data is ≥ MSS then
        send complete MSS segment now
    else
        if there is unconfirmed data still in the pipe then
            enqueue data in the buffer until an acknowledge is received
        else
            send data immediately
        end if
    end if
end if
```

Condition 2: Not enough data OR window size for a full MSS. But some packets are in-flight (yet to receive ACK for).

Buffer the data and wait.

Solution 2: Sender side approach

```
if there is new data to send then
    if the window size ≥ MSS and available data is ≥ MSS then
        send complete MSS segment now
    else
        if there is unconfirmed data still in the pipe then
            enqueue data in the buffer until an acknowledge is received
        else
            send data immediately
        end if
    end if
end if
```

Condition 3: No packet is in-flight (i.e., all ACKs are received).

Send immediately, no matter how small the packet is.

Solution 2: Sender side approach

(2) Nagle's algorithm

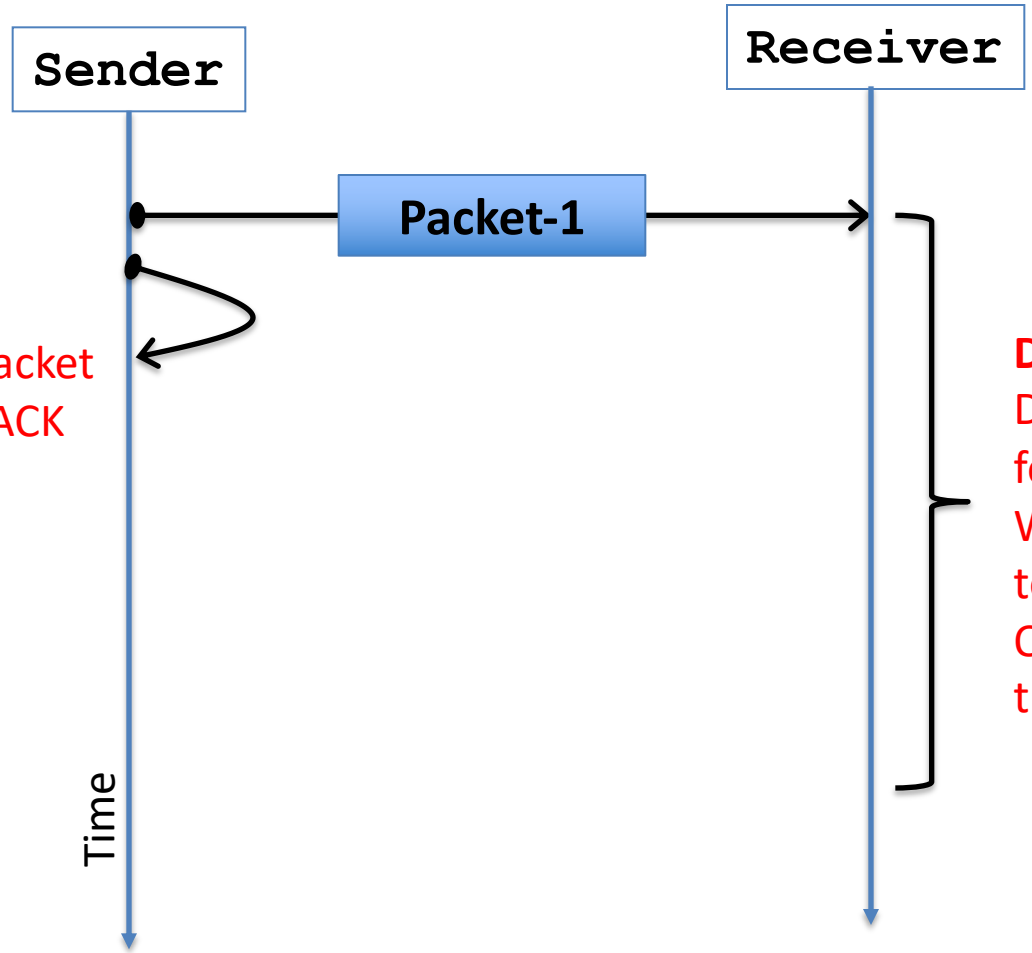
```
if there is new data to send then
    if the window size ≥ MSS and available data is ≥ MSS then
        send complete MSS segment now
    else
        if there is unconfirmed data still in the pipe then
            enqueue data in the buffer until an acknowledge is received
        else
            send data immediately
        end if
    end if
end if
```

Because some applications cannot afford such a delay for each write it does to a TCP connection, the socket interface allows the application to turn off Nagel's algorithm by setting the TCP_NODELAY socket option.
TCP_QUICKACK option disables the Delayed Ack protocol.

What if sender and receiver simultaneously applies the Nagle's algo. and delayed ack (TCP's default setting) ?

[Ref] https://betterprogramming.pub/redis-internals-client-connects-to-redis-by-tcp-faf6ad1a7df5

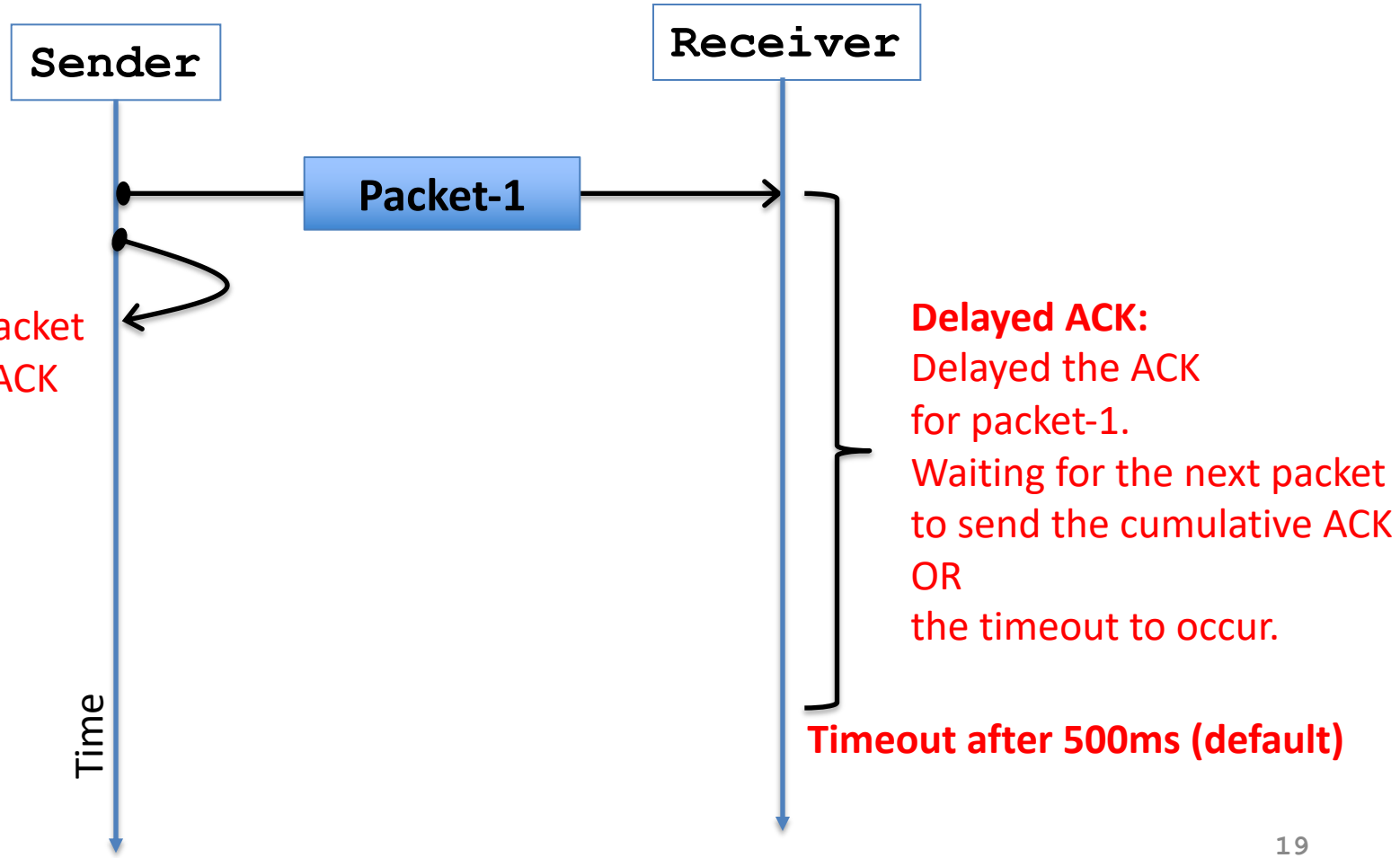
What if sender and receiver simultaneously applies the Nagle's algo. and delayed ack (TCP's default setting) ?



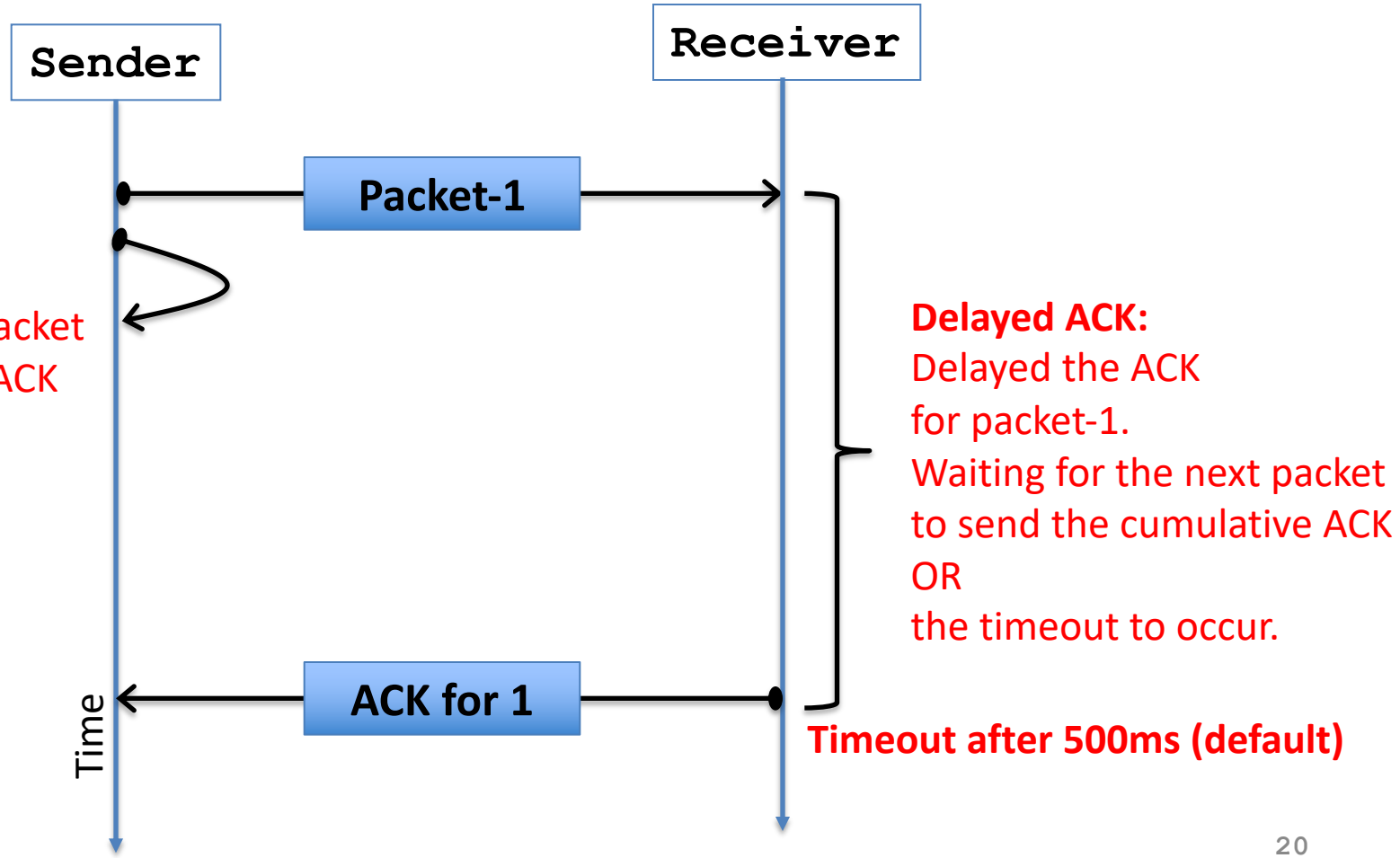
Nagle's algo:
Buffered next packet waiting for the ACK for packet-1.

Delayed ACK:
Delayed the ACK for packet-1.
Waiting for the next packet to send the cumulative ACK
OR
the timeout to occur.

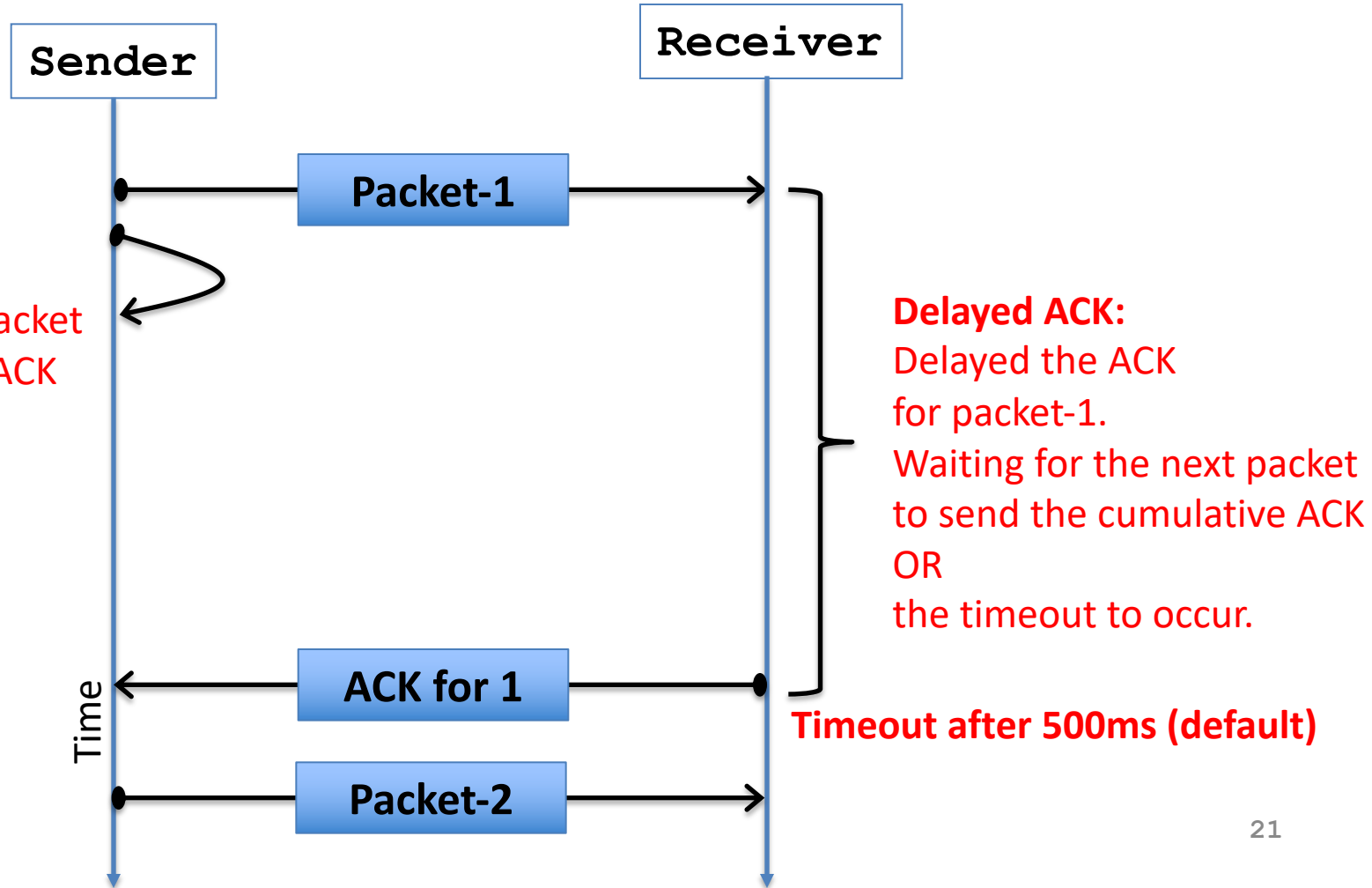
What if sender and receiver simultaneously applies the Nagle's algo. and delayed ack (TCP's default setting) ?



What if sender and receiver simultaneously applies the Nagle's algo. and delayed ack (TCP's default setting) ?



What if sender and receiver simultaneously applies the Nagle's algo. and delayed ack (TCP's default setting) ?

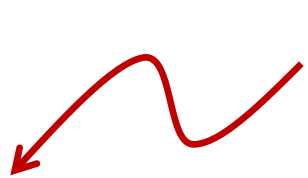


Timeout value calculation and its limitations

Calculating TCP timeout value: The original idea

$$\text{EstimatedRTT} = \alpha \times \text{EstimatedRTT} + (1 - \alpha) \times \text{SampleRTT}$$

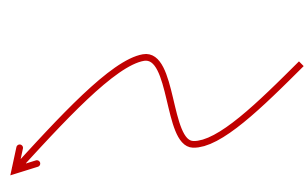
Calculating TCP timeout value: The original idea



Smoothing factor (recommended between 0.8 -0.9)

$$\text{EstimatedRTT} = \alpha \times \text{EstimatedRTT} + (1 - \alpha) \times \text{SampleRTT}$$

Calculating TCP timeout value: The original idea

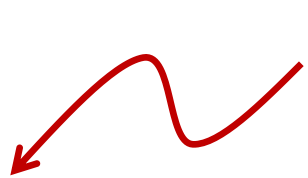


Smoothing factor (recommended between 0.8 -0.9)

$$\text{EstimatedRTT} = \alpha \times \text{EstimatedRTT} + (1 - \alpha) \times \text{SampleRTT}$$

$$\text{TimeOut} = 2 \times \text{EstimatedRTT}$$

Calculating TCP timeout value: The original idea



Smoothing factor (recommended between 0.8 -0.9)

$$\text{EstimatedRTT} = \alpha \times \text{EstimatedRTT} + (1 - \alpha) \times \text{SampleRTT}$$

$$\text{TimeOut} = 2 \times \text{EstimatedRTT}$$

RTT estimation can be wrong

- RTT estimation can be wrong

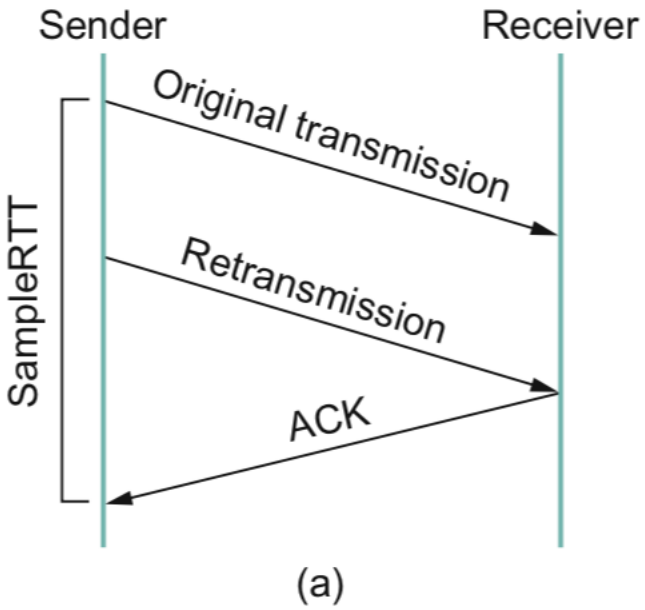


FIGURE 5.10 Associating the ACK with (a) original transmission versus

- RTT estimation can be wrong

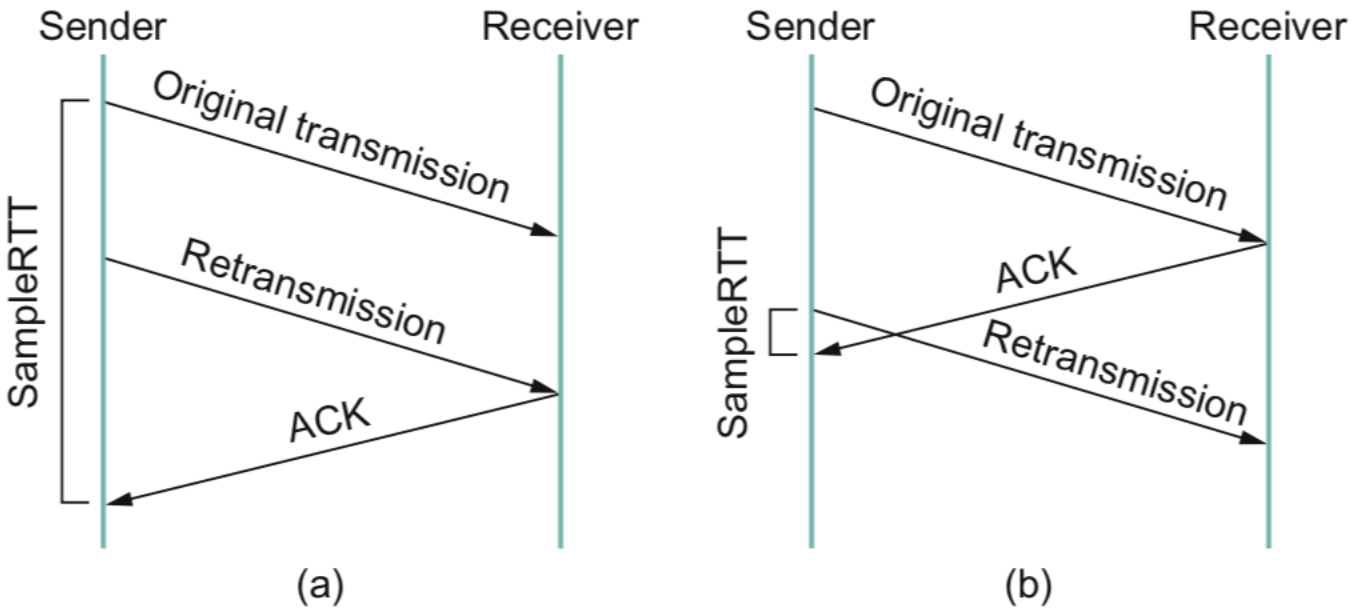


FIGURE 5.10 Associating the ACK with (a) original transmission versus (b) retransmission.

- Solution: Karn/Partridge algorithm

1) Stop taking RTT samples when TCP retransmits

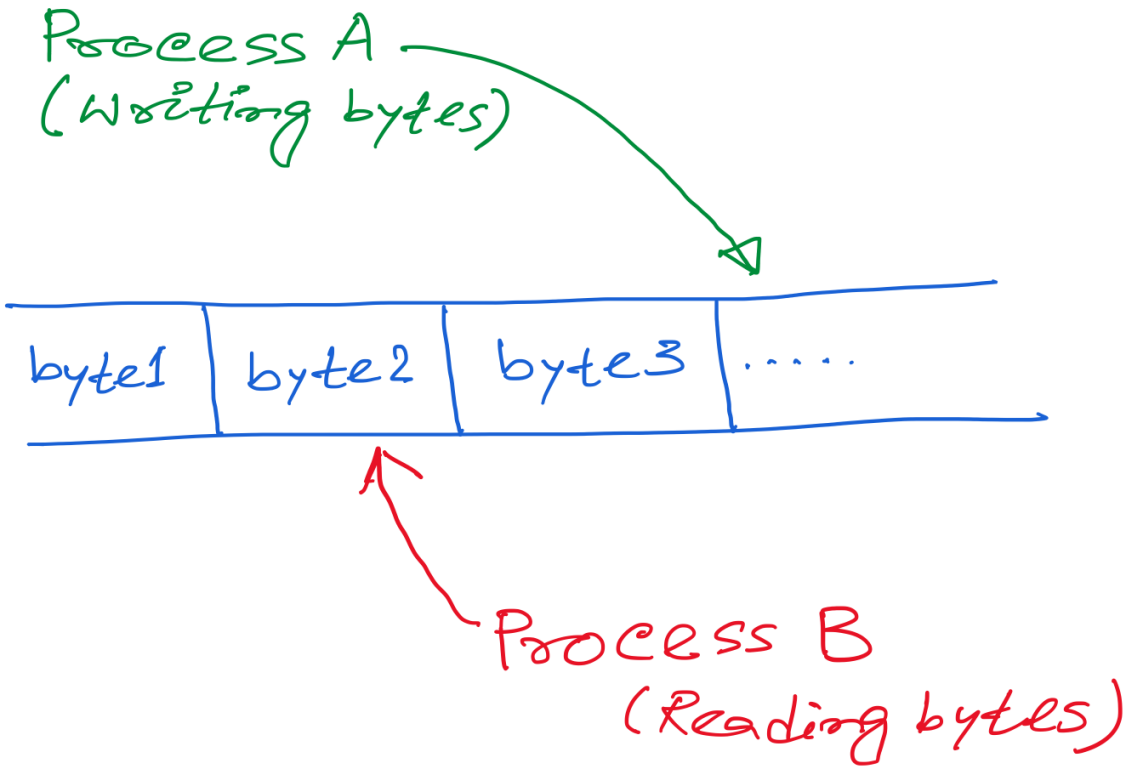
2) Start again after a regular transmission/reception

Can rules #1 & #2 create issues?

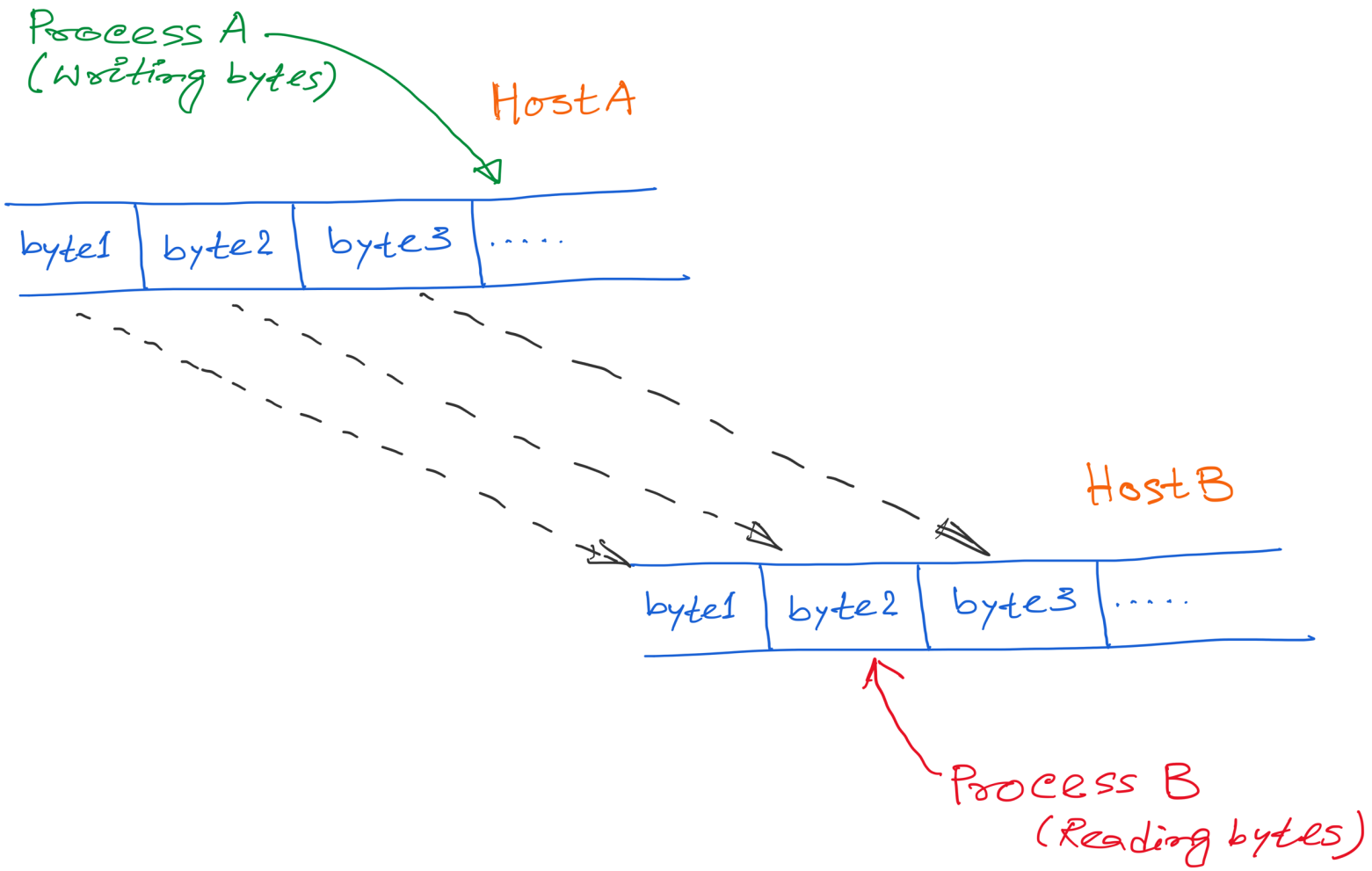
3) Each time TCP retransmits, set next timeout to be twice the last timeout value

Summary of TCP learned so far

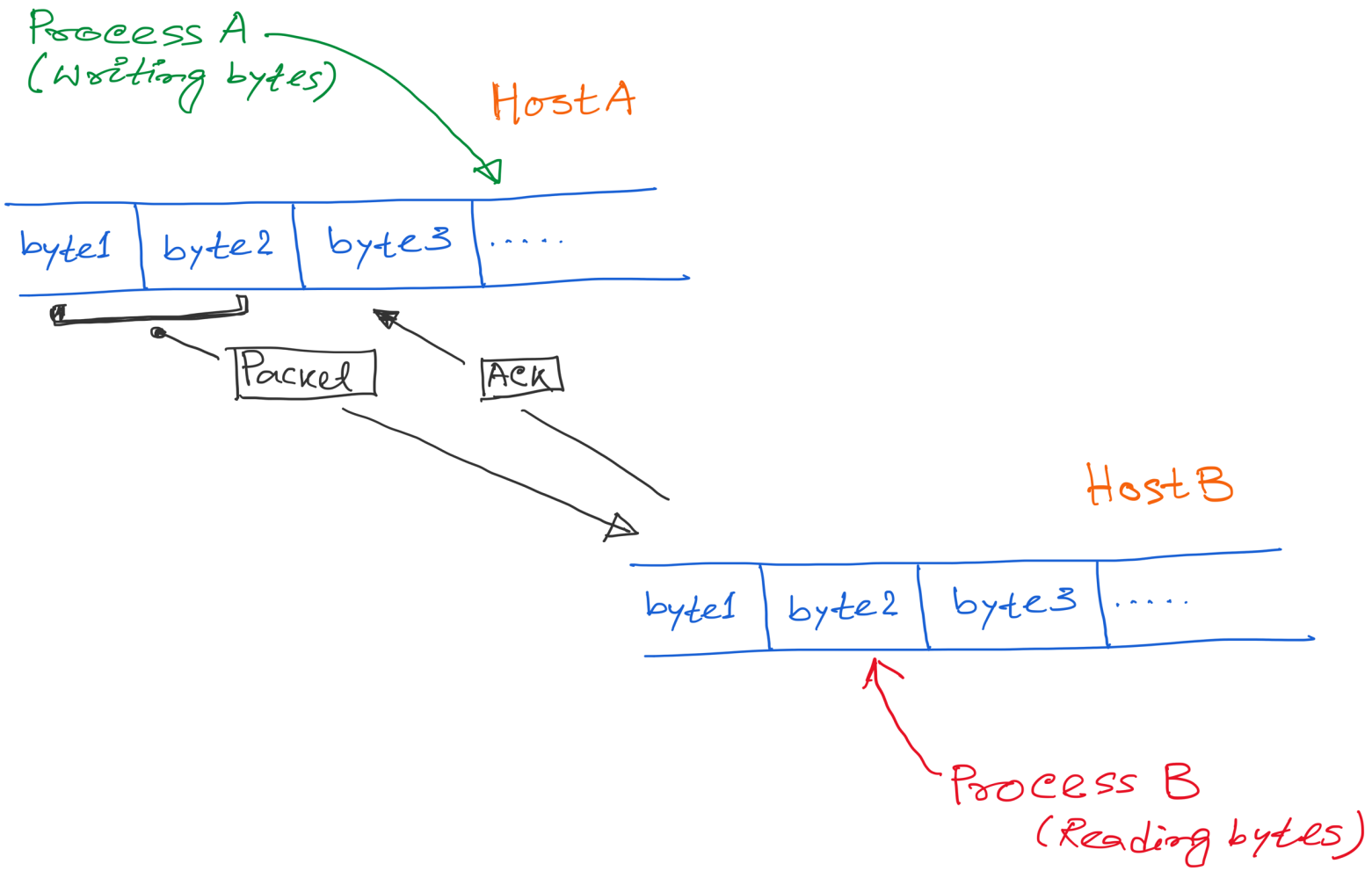
- TCP: Abstraction of Reliable Bytes Stream



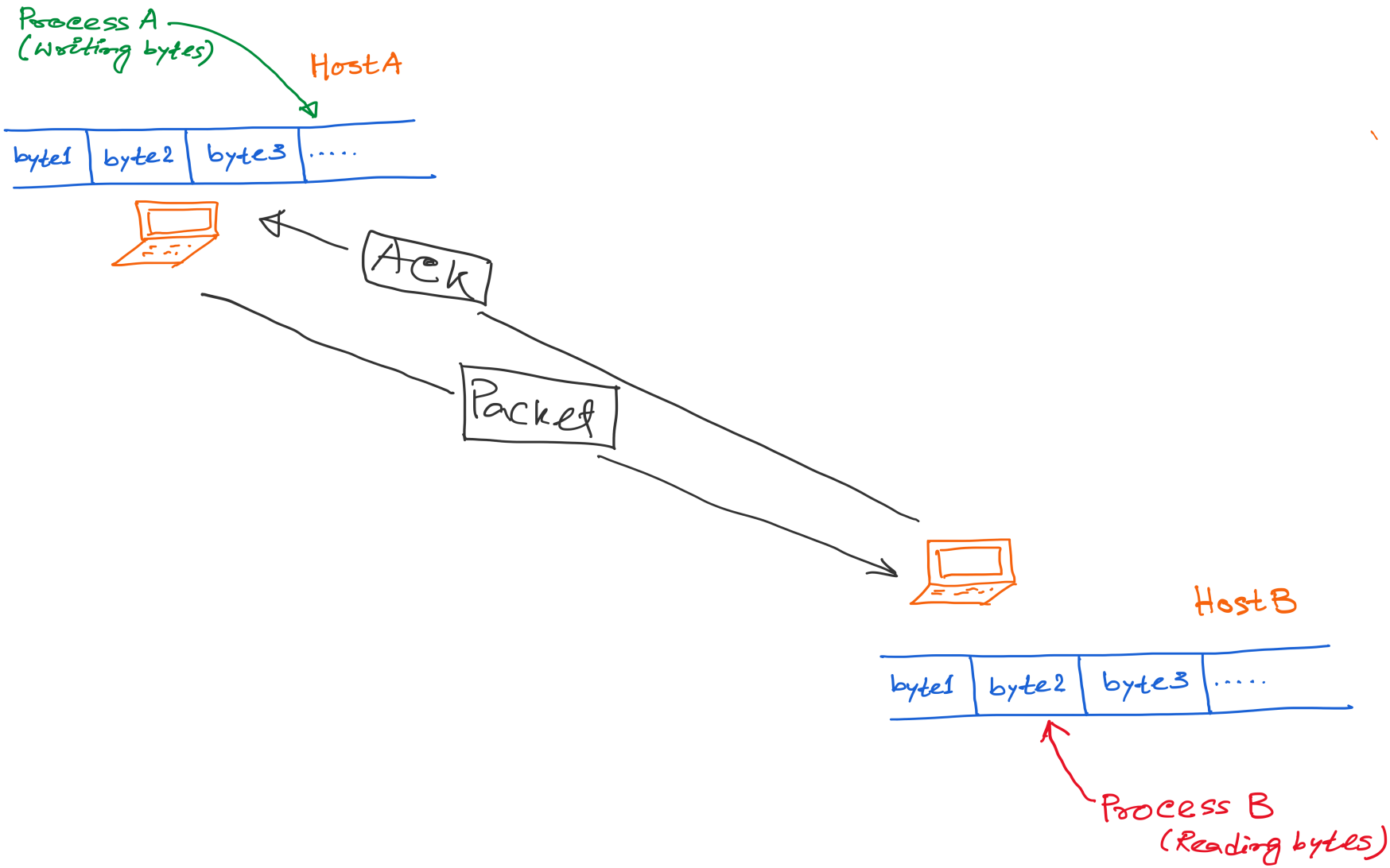
- TCP: Abstraction of Reliable Bytes Stream



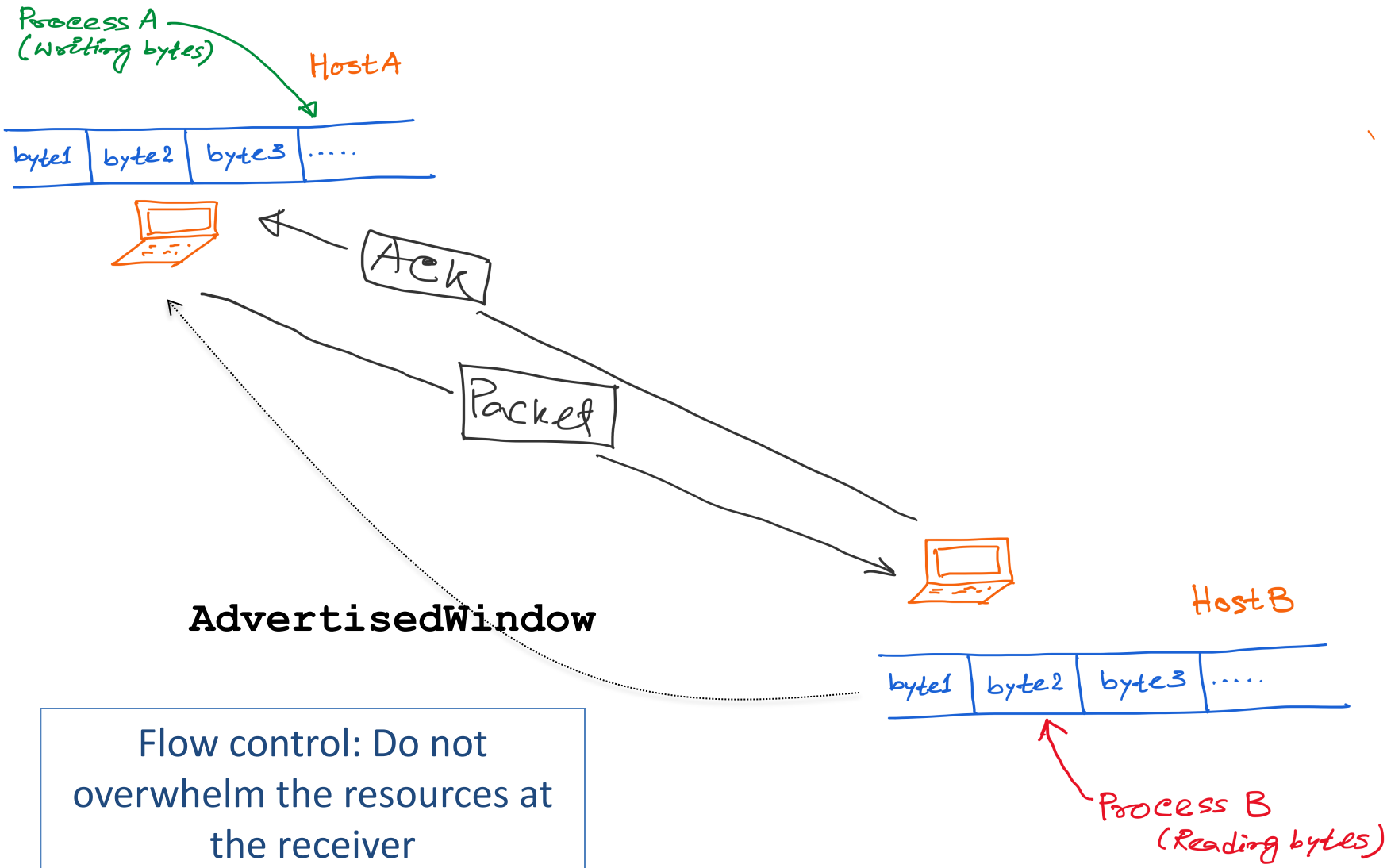
- TCP: Abstraction of Reliable Bytes Stream



- TCP: Abstraction of Reliable Bytes Stream

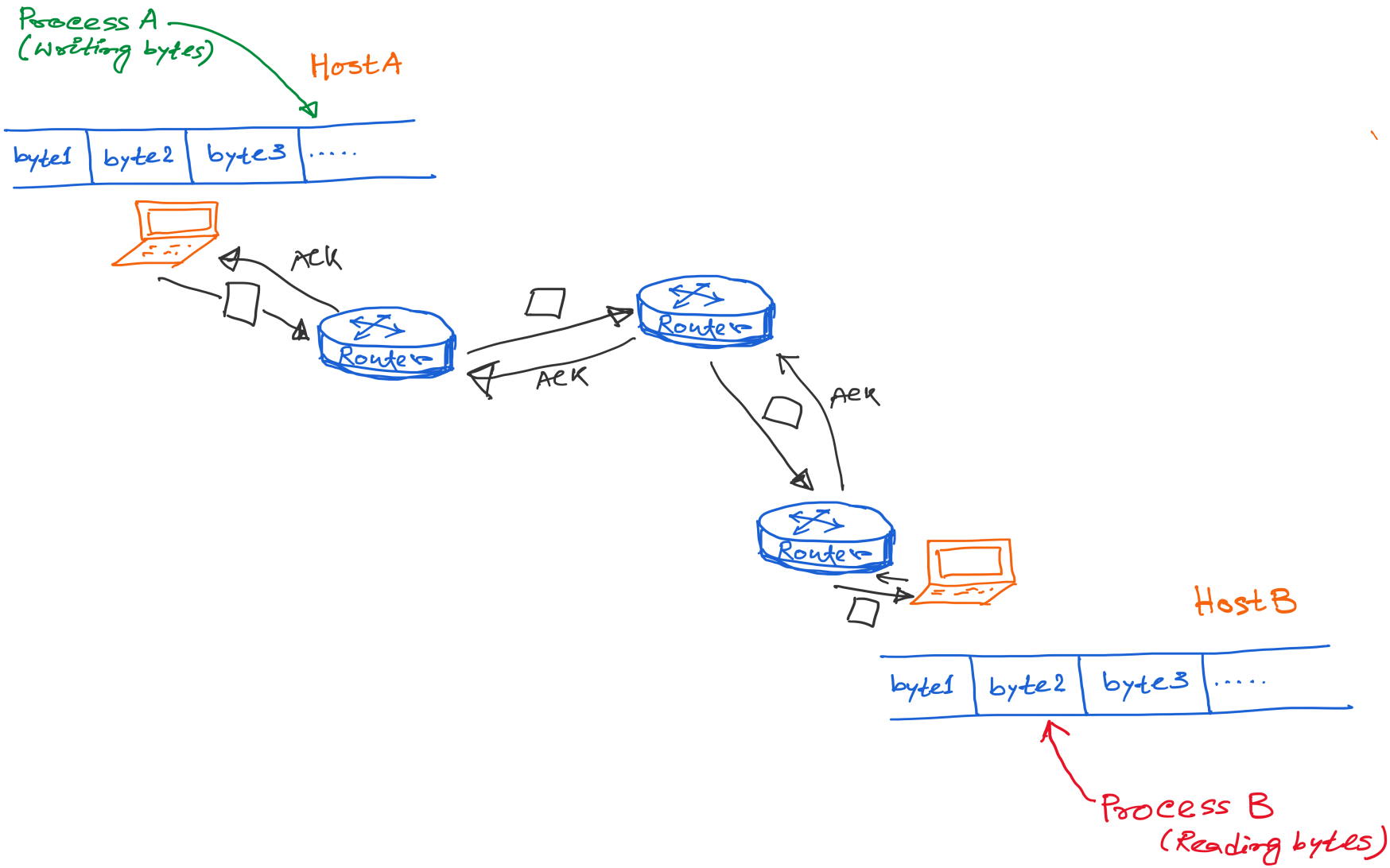


- TCP: Abstraction of Reliable Bytes Stream

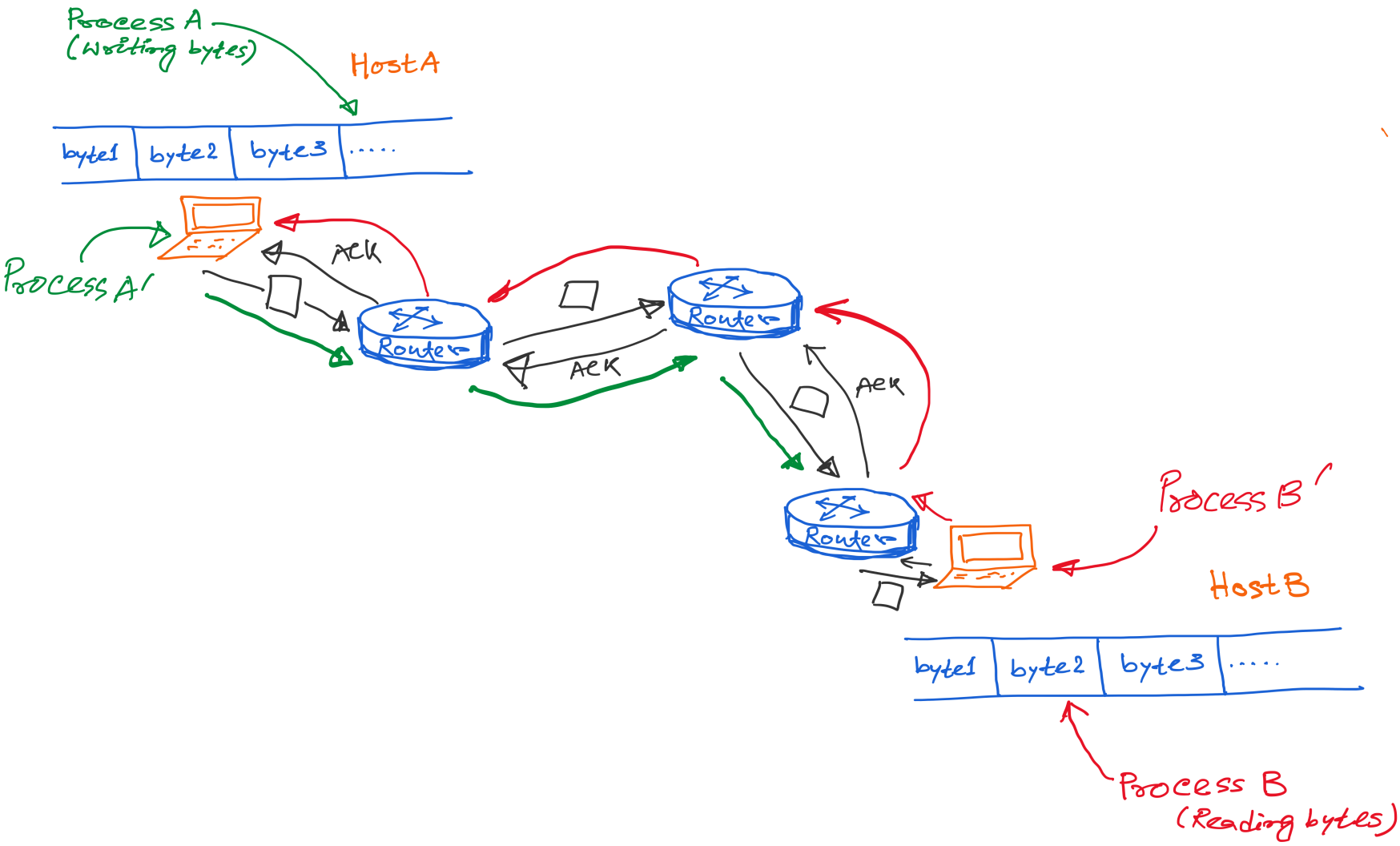


Flow control: Do not overwhelm the resources at the receiver

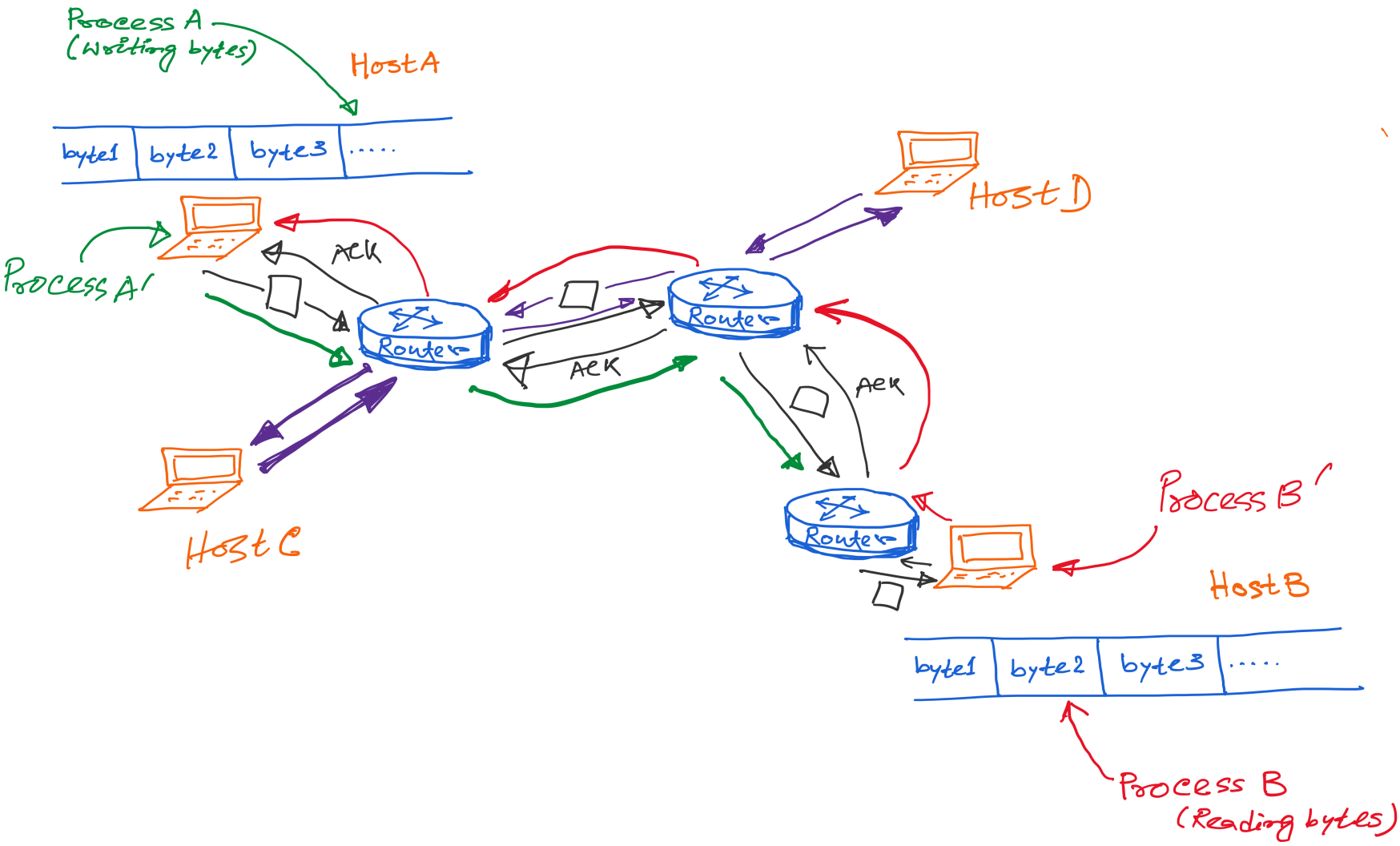
- TCP: Abstraction of Reliable Bytes Stream



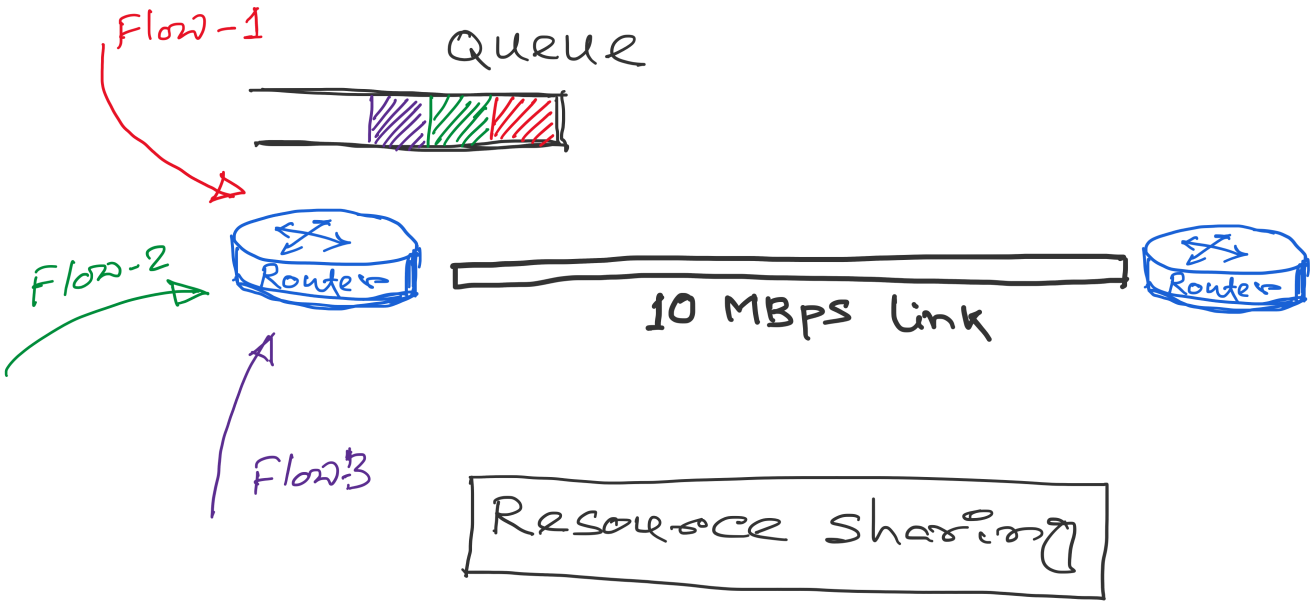
- TCP: Abstraction of Reliable Bytes Stream



- TCP: Abstraction of Reliable Bytes Stream



- TCP: Abstraction of Reliable Bytes Stream



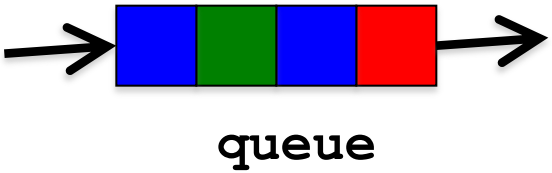
Flows contend for two resources:

1) Link bandwidth
2) Queue space

Congestion

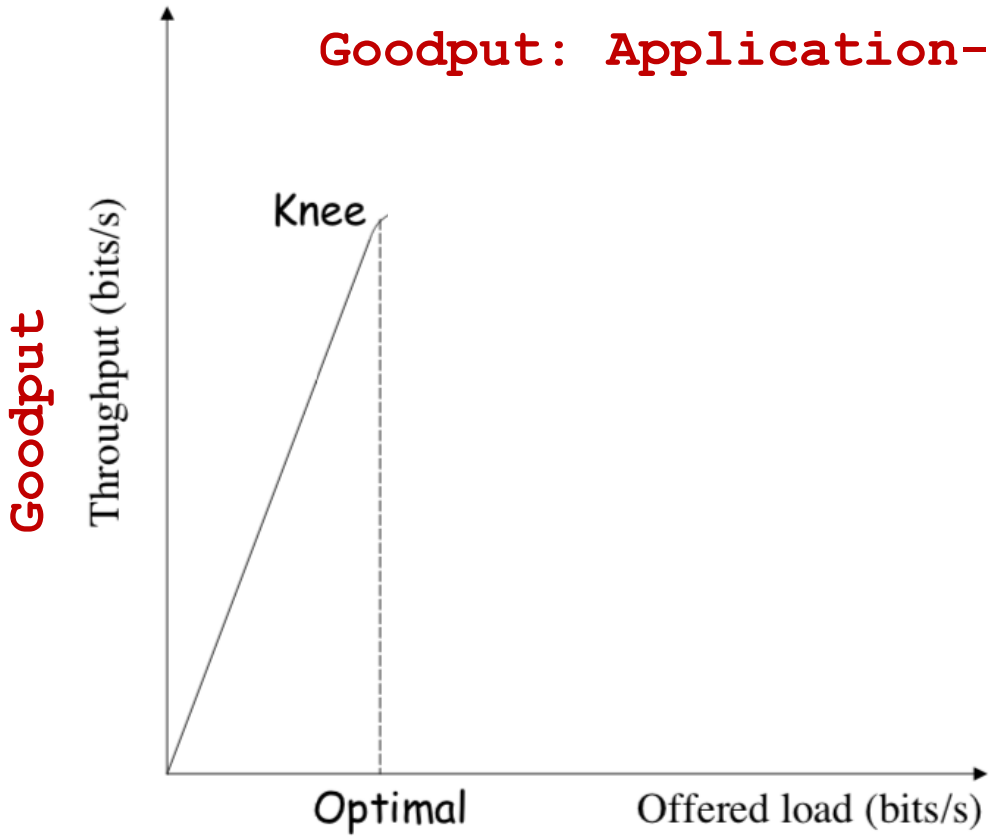
A network is considered congested if contention for resources leads toward queue build-ups and packet drops

- Best-effort network does not "block" calls
  - So, they can easily become overloaded
  - Congestion == "Load higher than capacity"
- Examples of congestion
  - Link layer: Ethernet frame collisions
  - Network layer: full IP packet buffers
- Excess packets are simply dropped
  - And the sender can simply retransmit

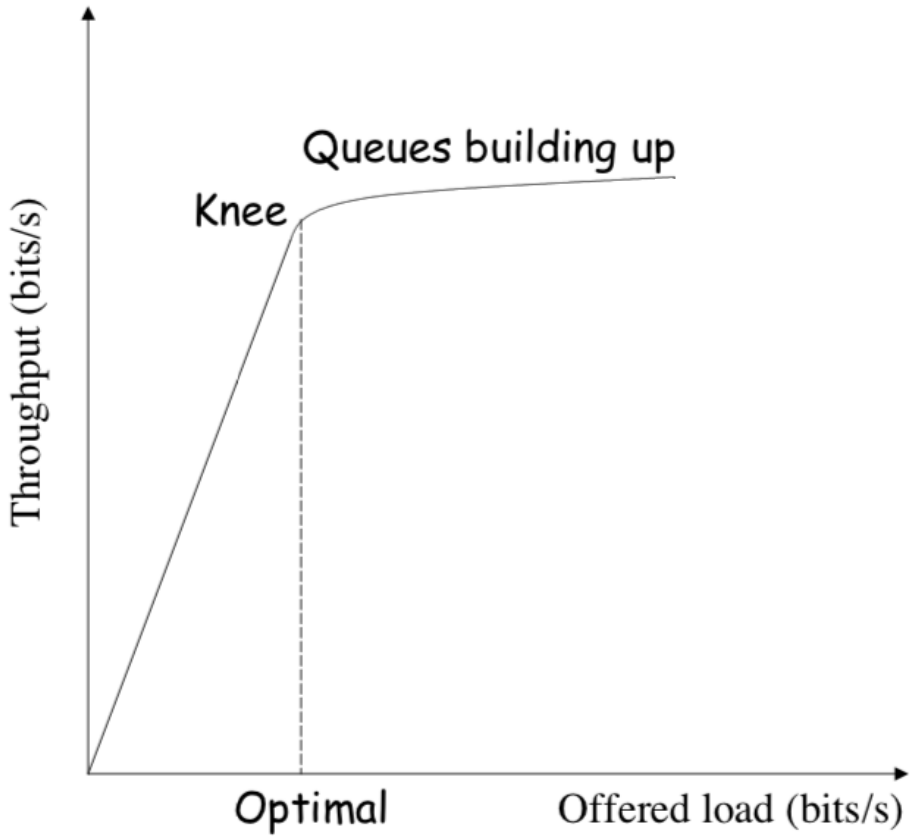


Throughput vs Load

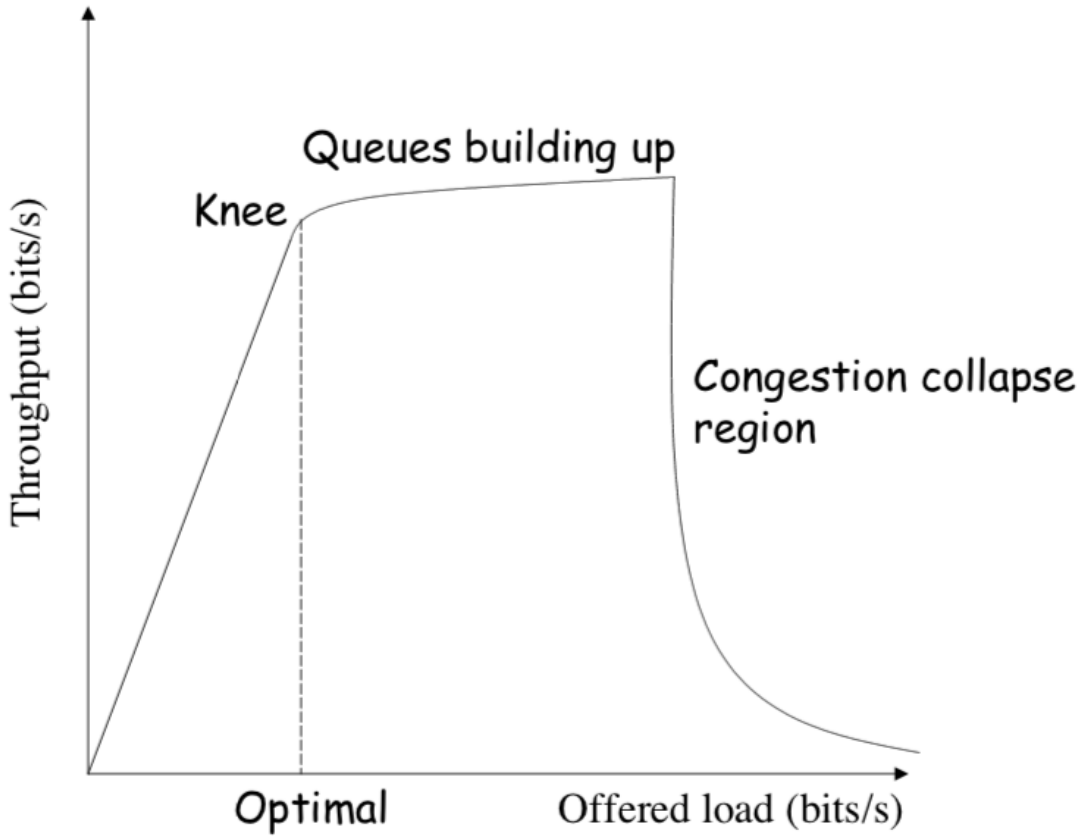
Goodput: Application-level throughput.



Throughput vs Load

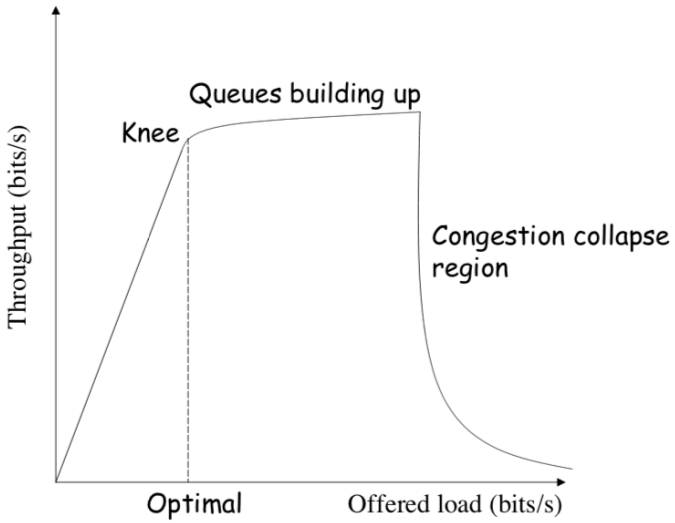


Congestion Collapse



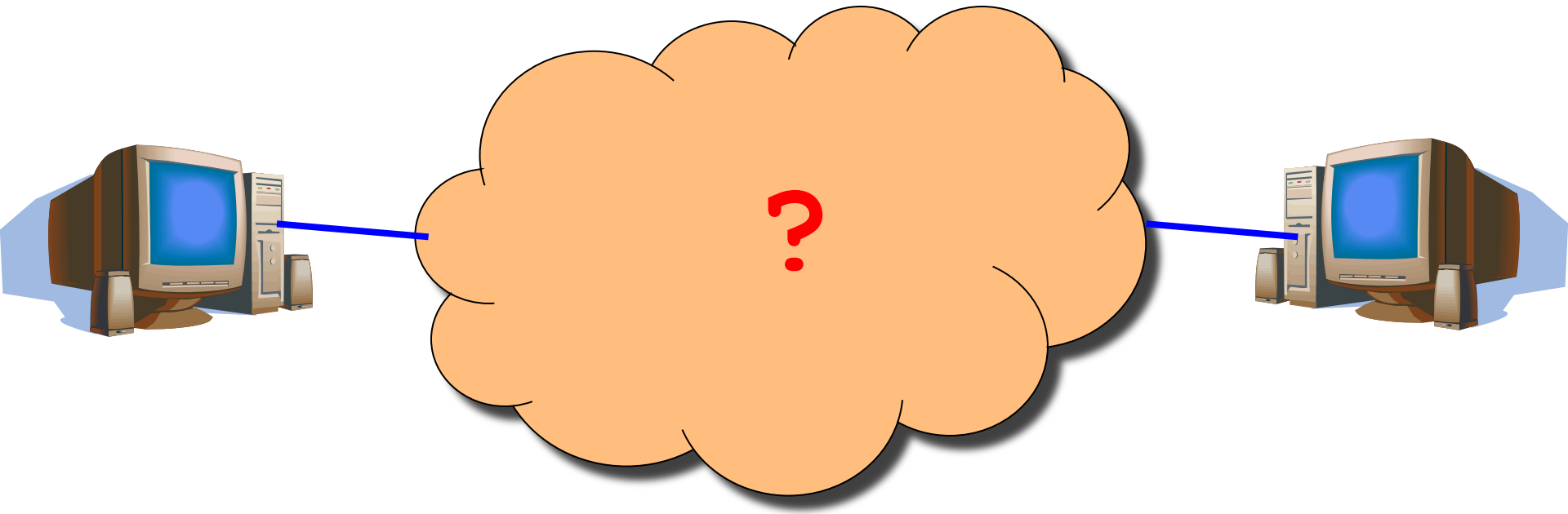
Congestion Collapse is real !!

- Easily leads to *congestion collapse*
  - Senders retransmit the lost packets
  - Leading to even *greater* load
  - … and even *more* packet loss



Increase in load that results in a *decrease* in useful work done.

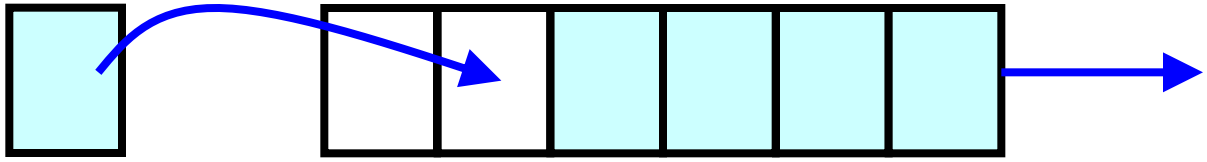
Detect and Respond to Congestion



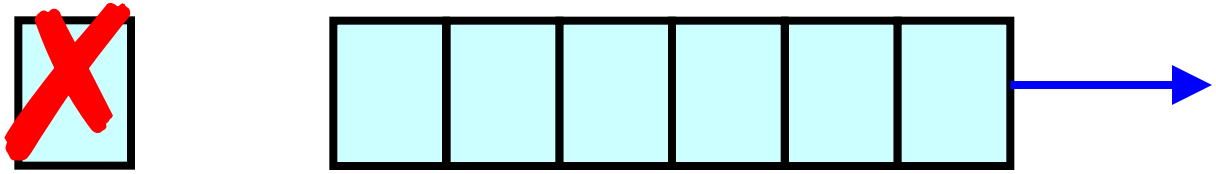
- What does the end host see?
- What can the end host change?

Congestion in a Drop-Tail FIFO Queue

- Access to the bandwidth: first-in first-out queue
  - Packets transmitted in the order they arrive

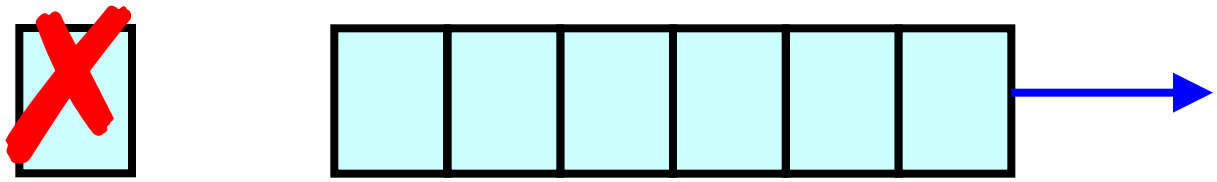


- Access to the buffer space: drop-tail queuing
  - If the queue is full, drop the incoming packet



How it Looks to the End Host

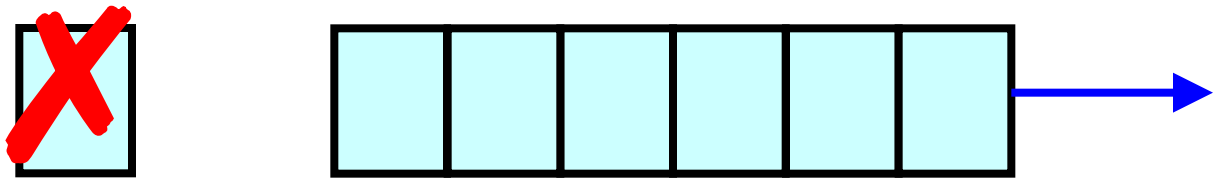
- Delay: Packet experiences high delay
- Loss: Packet gets dropped along path



How it Looks to the End Host

- Delay: Packet experiences high delay
- Loss: Packet gets dropped along path

- How does TCP sender learn this?



Detecting Congestion

- Link layer
  - Carrier sense multiple access
  - Seeing your own frame collide with others
- Network layer
  - Observing end-to-end performance
  - Packet delay or loss over the path

How TCP detects packet losses

(1) Duplicate ACKs

- Packet# n is lost, but packets n+1, n+2, etc. are received and ACKs are sent.

(2) Timeout

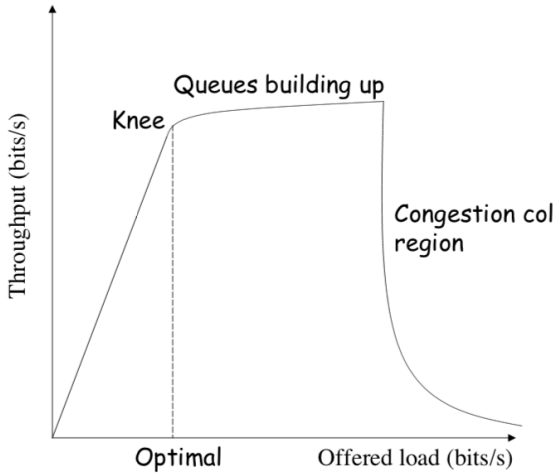
- Packet# n is lost and detected via a timeout

Responding to Congestion

- Upon detecting congestion
  - Decrease the sending rate

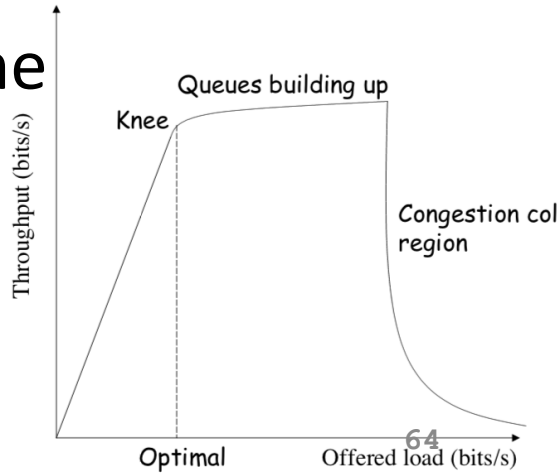
Responding to Congestion

- Upon detecting congestion
  - Decrease the sending rate
- But, what if conditions change?
  - If more bandwidth becomes available,
  - … unfortunate to keep sending at a low rate



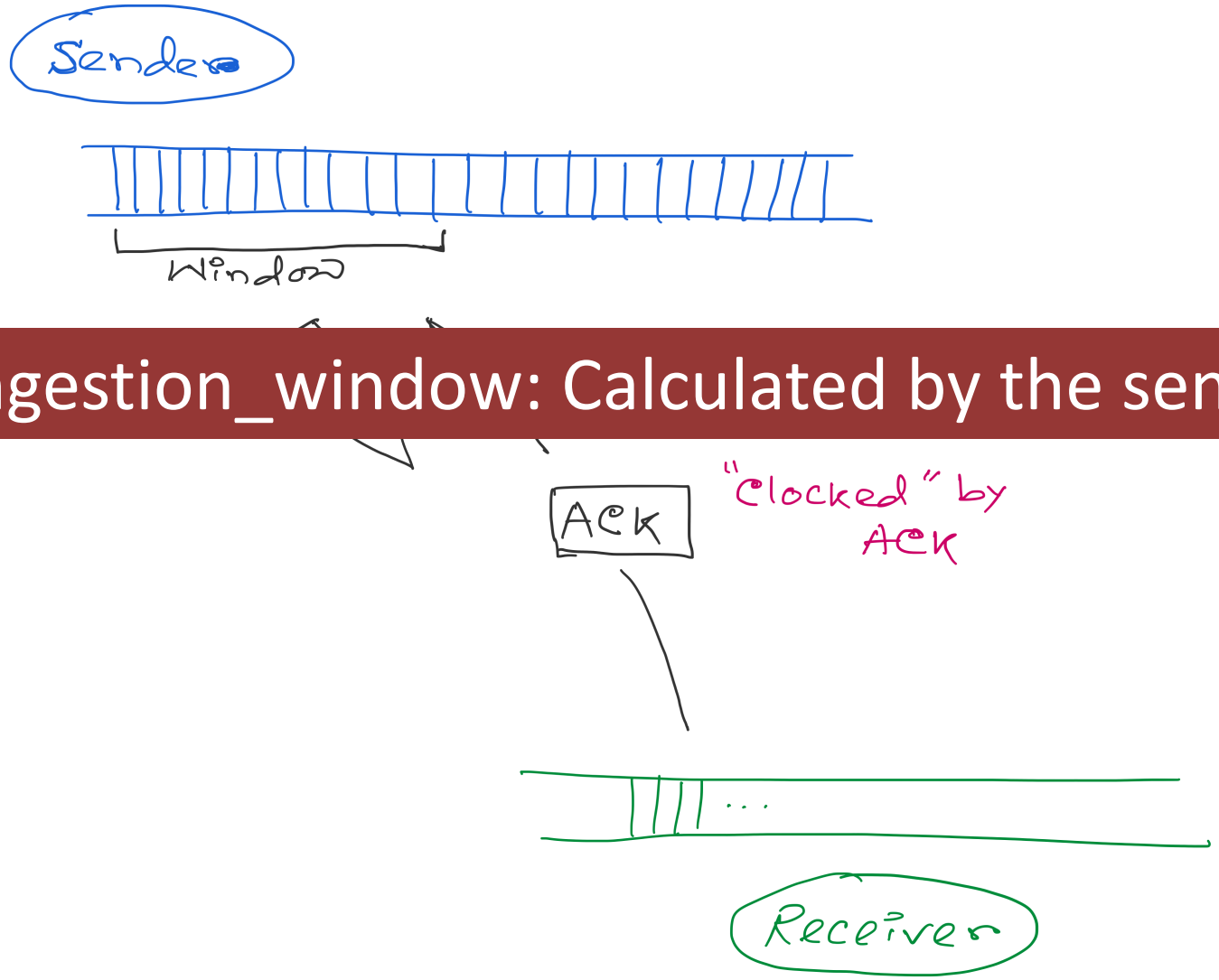
Responding to Congestion

- Upon detecting congestion
  - Decrease the sending rate
- But, what if conditions change?
  - If more bandwidth becomes available,
  - … unfortunate to keep sending at a low rate
- Upon *not* detecting congestion
  - Increase sending rate, a little at a time
  - See if packets get through

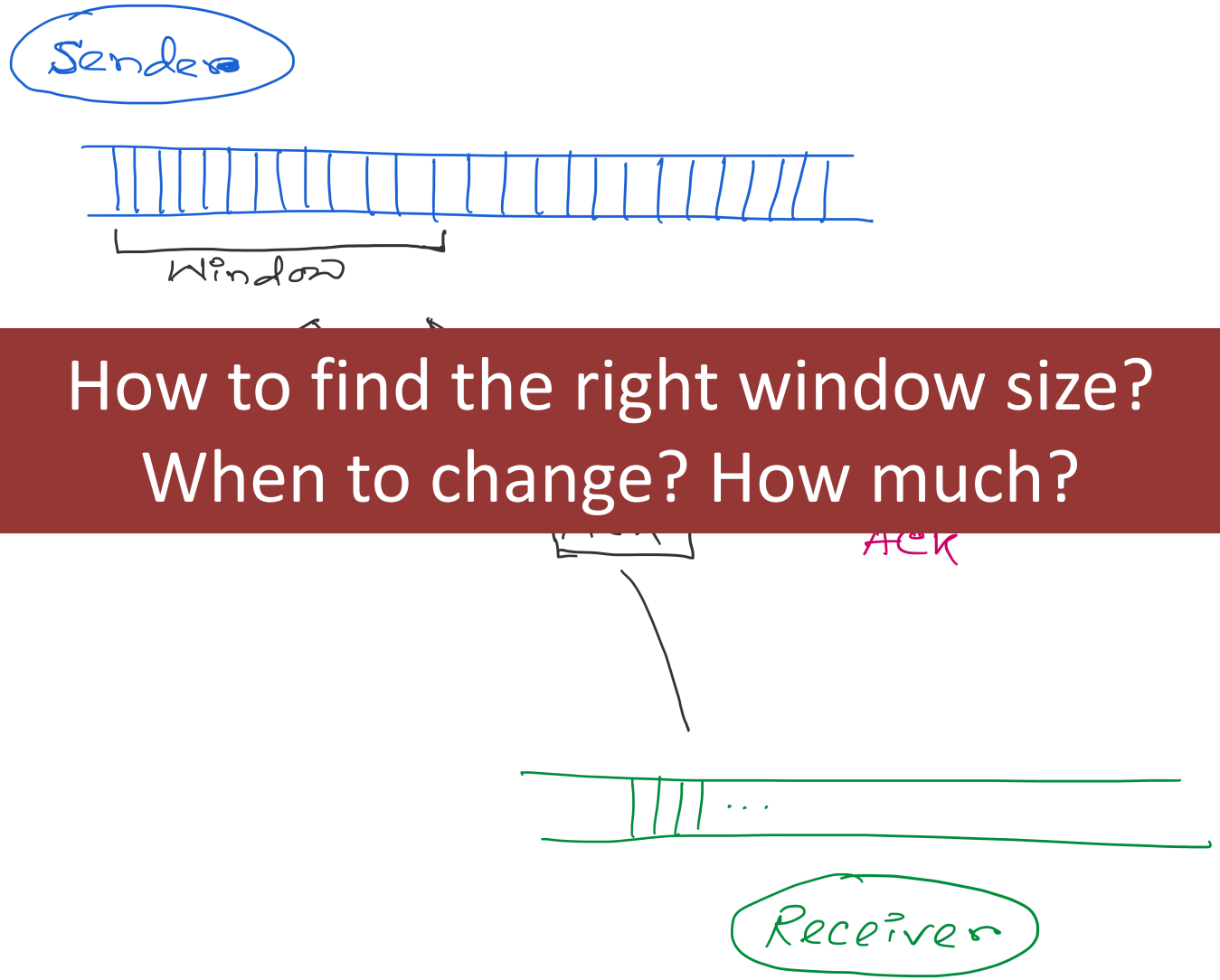


TCP Congestion Control

- ACK works as the indicator and the window controls the flow.



- ACK works as the indicator and the window controls the flow.



How to find the right window size?
When to change? How much?

TCP Congestion Control

- Additive increase, multiplicative decrease (AIMD)
  - On packet loss, divide congestion window in half
  - On success for last window, increase window linearly