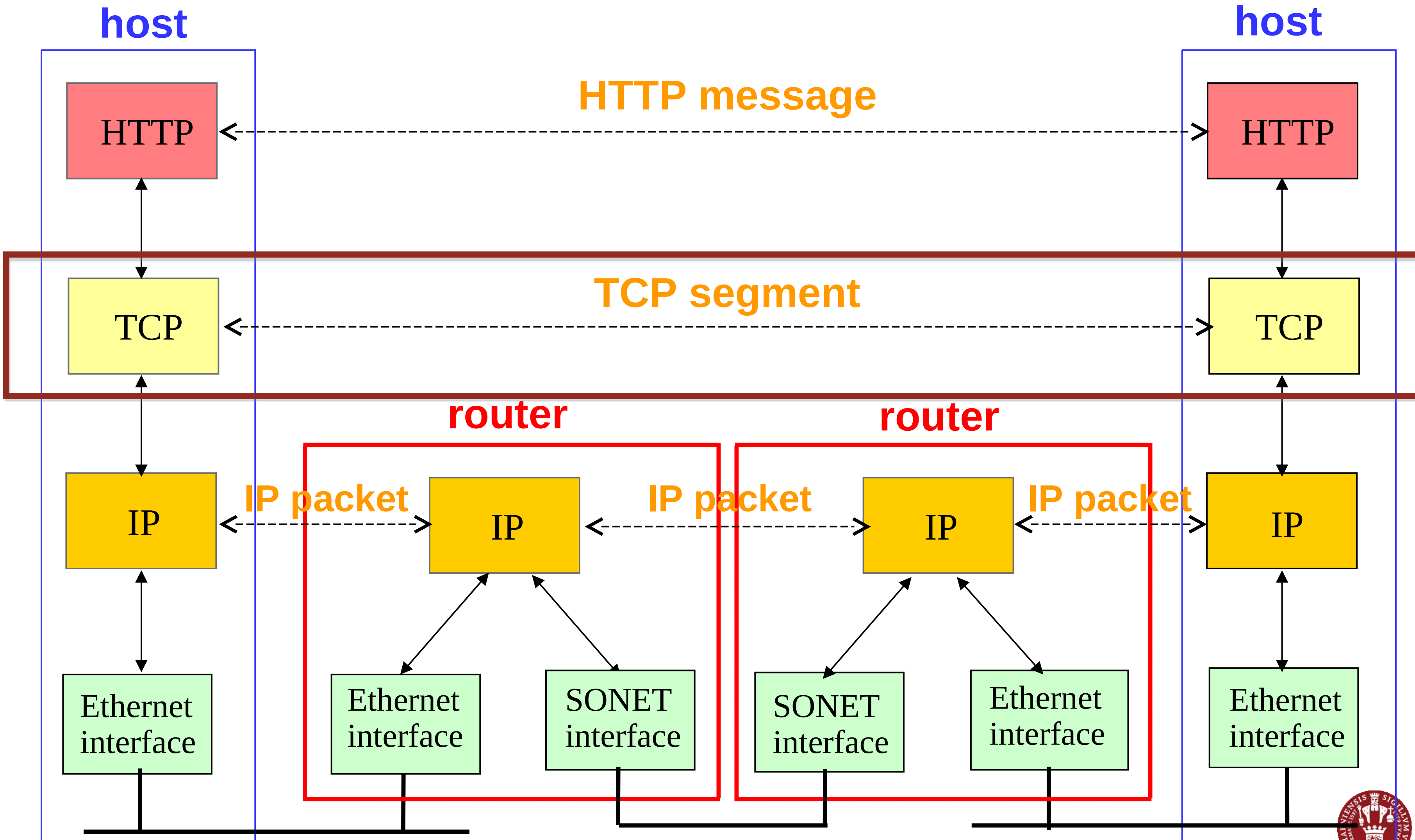
Transport Layer: UDP + Reliable Data Transfer + TCP

David Marchant

Based on slides compiled by Marcos Vaz Salles, with adaptions by Vivek Shah and Michael Kirkedal Thomsen

Internet Layering Model



Source: Freedman

Transport Layer

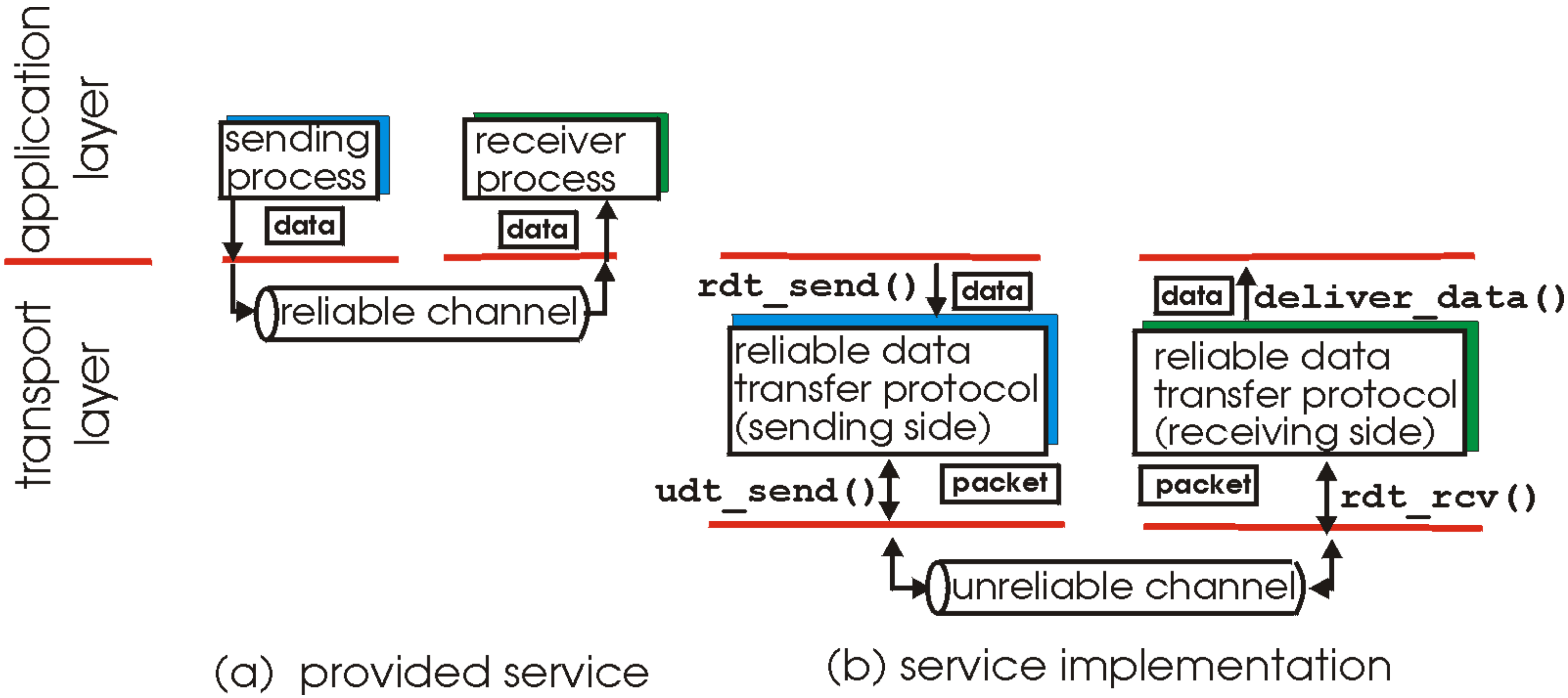
- Logical Communication between processes
  - Sender divides messages into segments.
  - Receiver re-assembles messages into segments.

- Principles underlying transport-layer services
  - (De)multiplexing
  - Detecting corruption
  - Optional: Reliable delivery, Flow control, Congestion control

- Transport-layer protocols in the Internet
  - User Datagram Protocol (UDP)
    - Simple (unreliable) message delivery
  - Transmission Control Protocol (TCP)
    - Reliable bidirectional stream of bytes

Source: Freedman

Reliable Data Transfer



Source: Kurose & Ross

- What can go wrong on the unreliable channel?
- How can you deal with it?
  - Suppose you want to transfer TCP segments, reliably and in order! ☺

Challenges of Reliable Data Transfer

- Over a perfectly reliable channel: Done
- Over a channel with bit errors
  - Receiver detects errors and requests re-transmission
- Over a lossy channel with bit errors
  - Some data missing, others corrupted
  - Receiver cannot easily detect loss
- Over a channel that may reorder packets
  - Receiver cannot easily distinguish loss vs. out-of-order

Source: Freedman

An Analogy

- Alice and Bob are talking
  - What if Alice couldn’t understand Bob?
  - Alice asks Bob to repeat what he said

- What if Bob hasn’t heard Alice for a while?
  - Is Alice just being quiet? Has she lost reception?
  - How long should Bob just keep on talking?
  - Maybe Alice should periodically say “uh huh”
  - … or Bob should ask “Can you hear me now?”

Source: Freedman

Take Aways from the Example

- Acknowledgments from receiver
  - Positive: “okay” or “uh huh” or “ACK”
  - Negative: “please repeat that” or “NACK”
- Retransmission by the sender
  - After *not* receiving an “ACK”
  - After receiving a “NACK”
- Timeout by the sender (“stop and wait”)
  - Don’t wait forever without some acknowledgment

Source: Freedman

TCP Support for Reliable Delivery

- **Detect bit errors:** checksum
  - Used to detect corrupted data at the receiver
  - …leading the receiver to drop the packet
- **Detect missing data:** sequence number
  - Used to detect a gap in the stream of bytes
  - … and for putting the data back in order
- **Recover from lost data:** retransmission
  - Sender re-transmits lost or corrupted data
  - Two main ways to detect lost packets

Source: Freedman

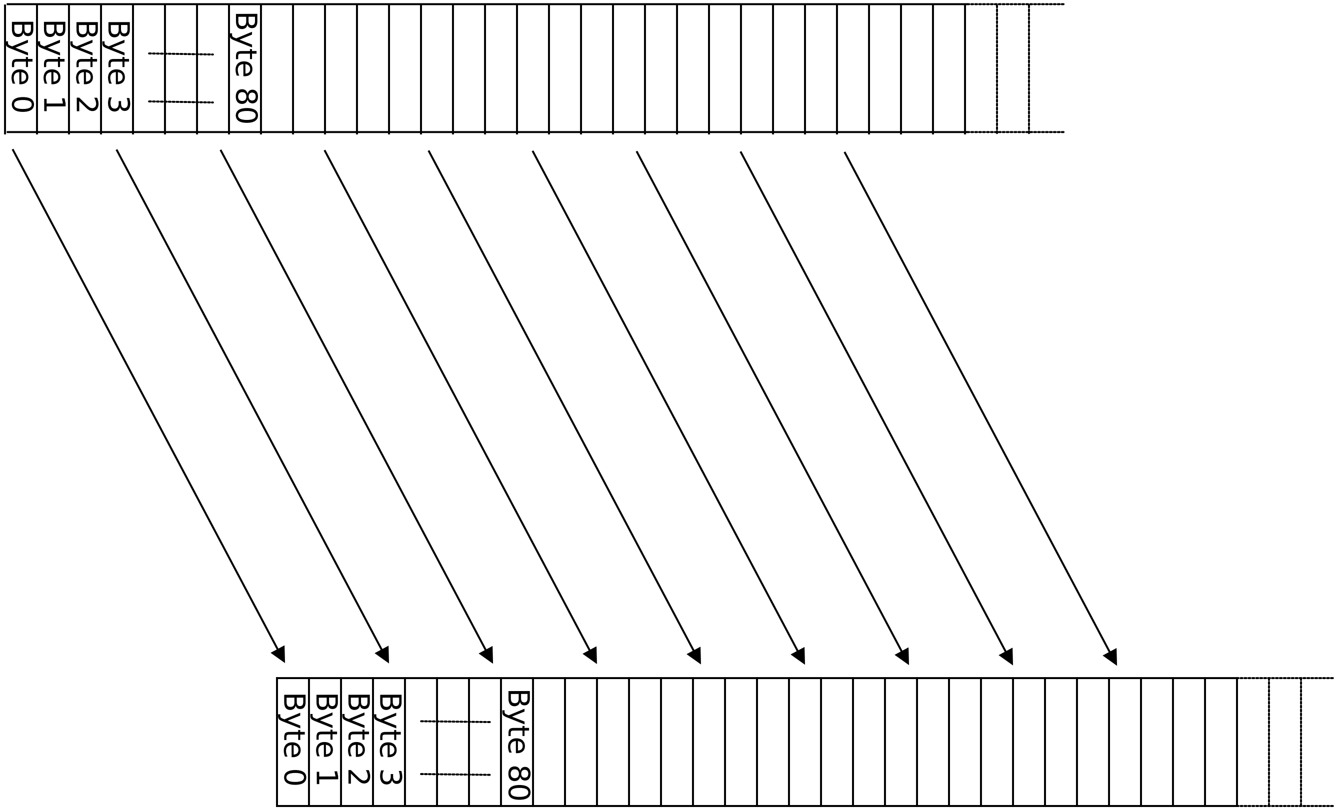
Transmission Control Protocol (TCP)

- **Stream-of-bytes service:** Send/recv streams, not msgs
- **Reliable, in-order delivery**
  - Checksums to detect corrupted data
  - Sequence numbers to detect losses and reorder data
  - Acknowledgments & retransmissions for reliable delivery
- **Connection oriented:** Explicit set-up and tear-down
- **Flow control:** Prevent overload of receiver’s buffer
- **Congestion control:** Adapt for greater good

Source: Freedman

TCP "Stream of Bytes" Service

Host A

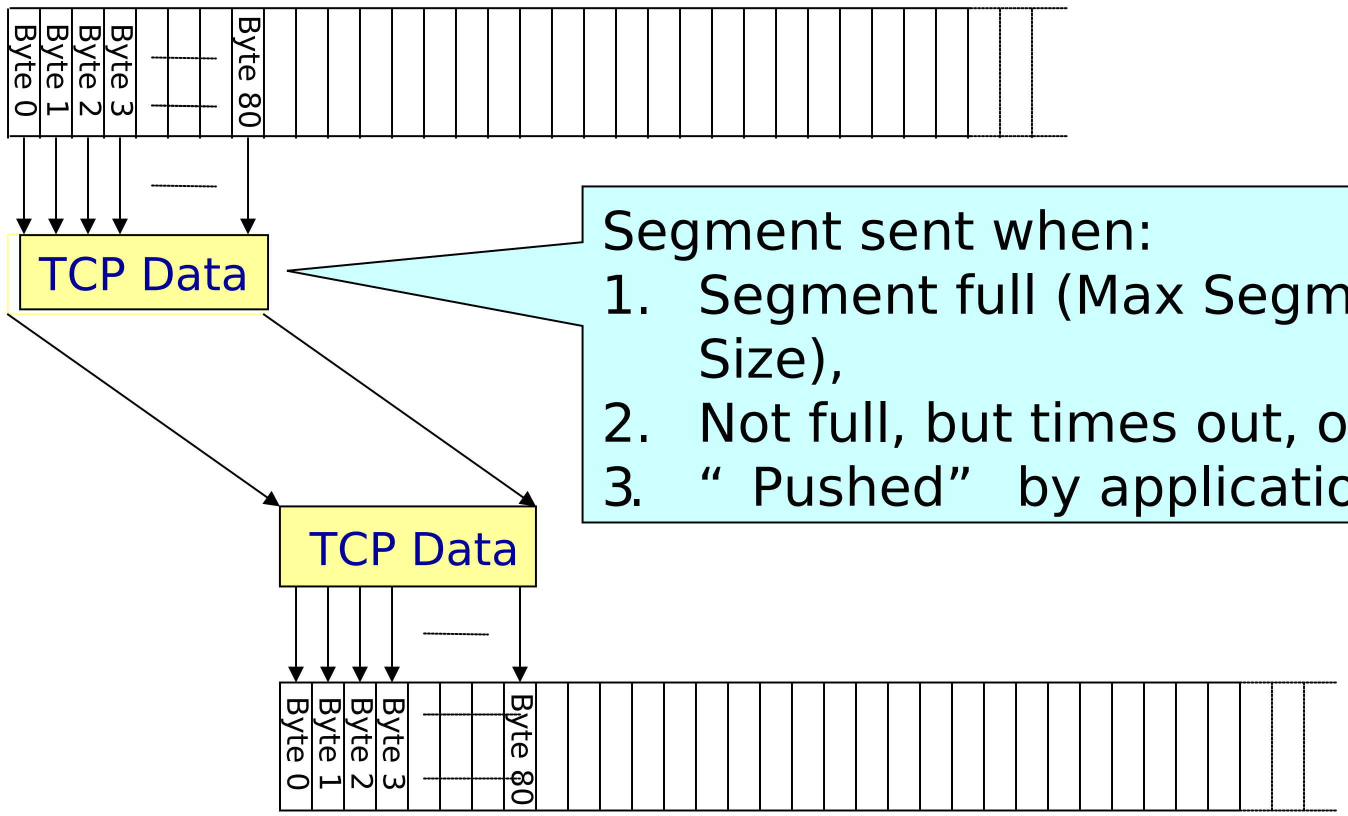


Host B

Source: Freedman

… Emulated Using TCP "Segments"

Host A



Host B

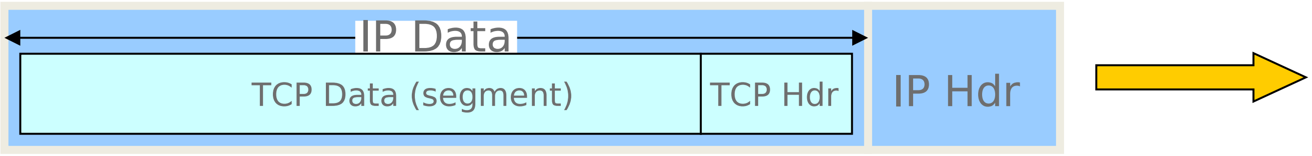
Segment sent when:

1. Segment full (Max Segment Size),
2. Not full, but times out, or
3. " Pushed" by application

Source: Freedman

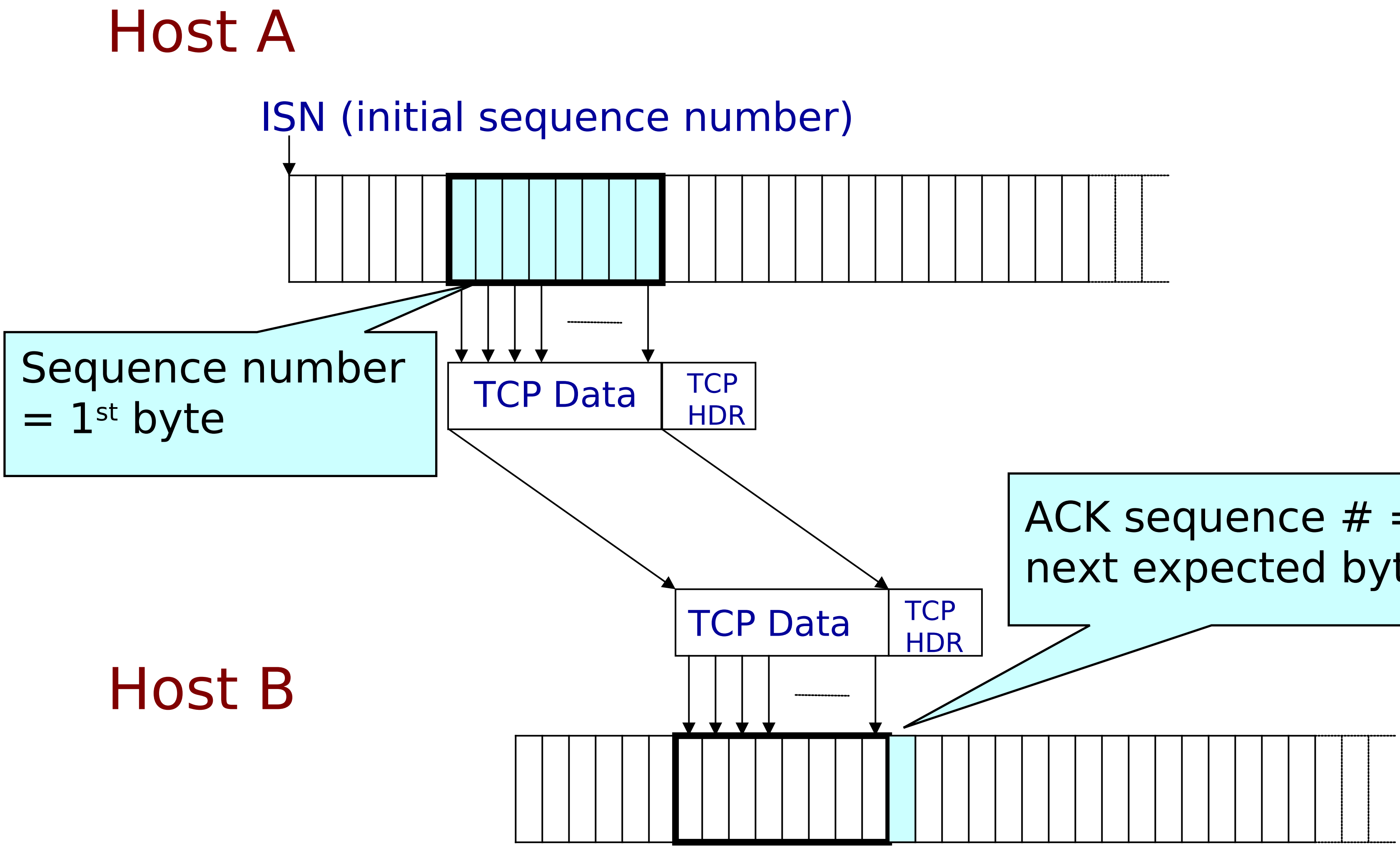
TCP Segment

- IP packet
  - No bigger than Maximum Transmission Unit (MTU)
  - E.g., up to 1500 bytes on an Ethernet link
- TCP packet
  - IP packet with a TCP header and data inside
  - TCP header is typically 20 bytes long
- TCP segment
  - No more than Maximum Segment Size (MSS) bytes
  - E.g., up to 1460 consecutive bytes from the stream



Source: Freedman

TCP Acknowledgements

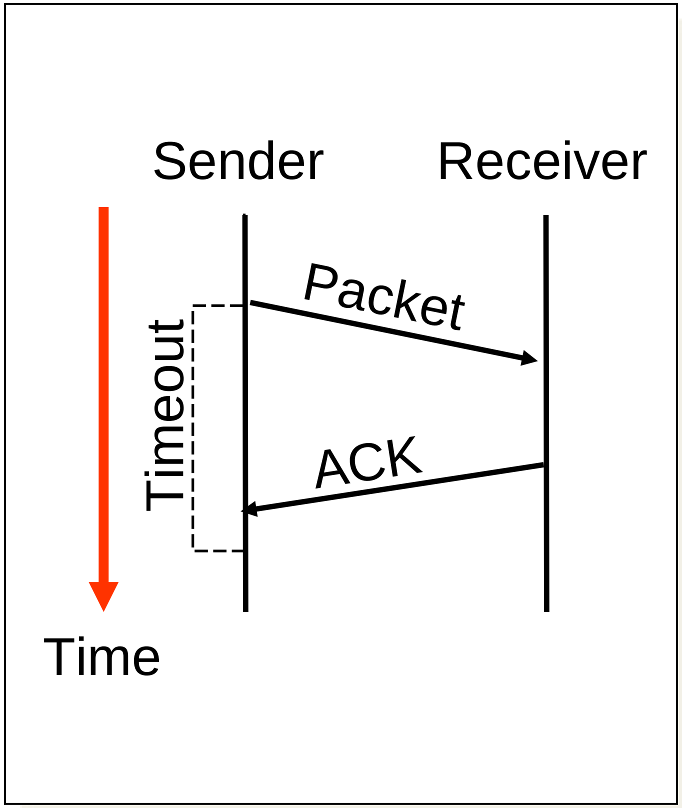


Source: Freedman

Automatic Repeat Request (ARQ)

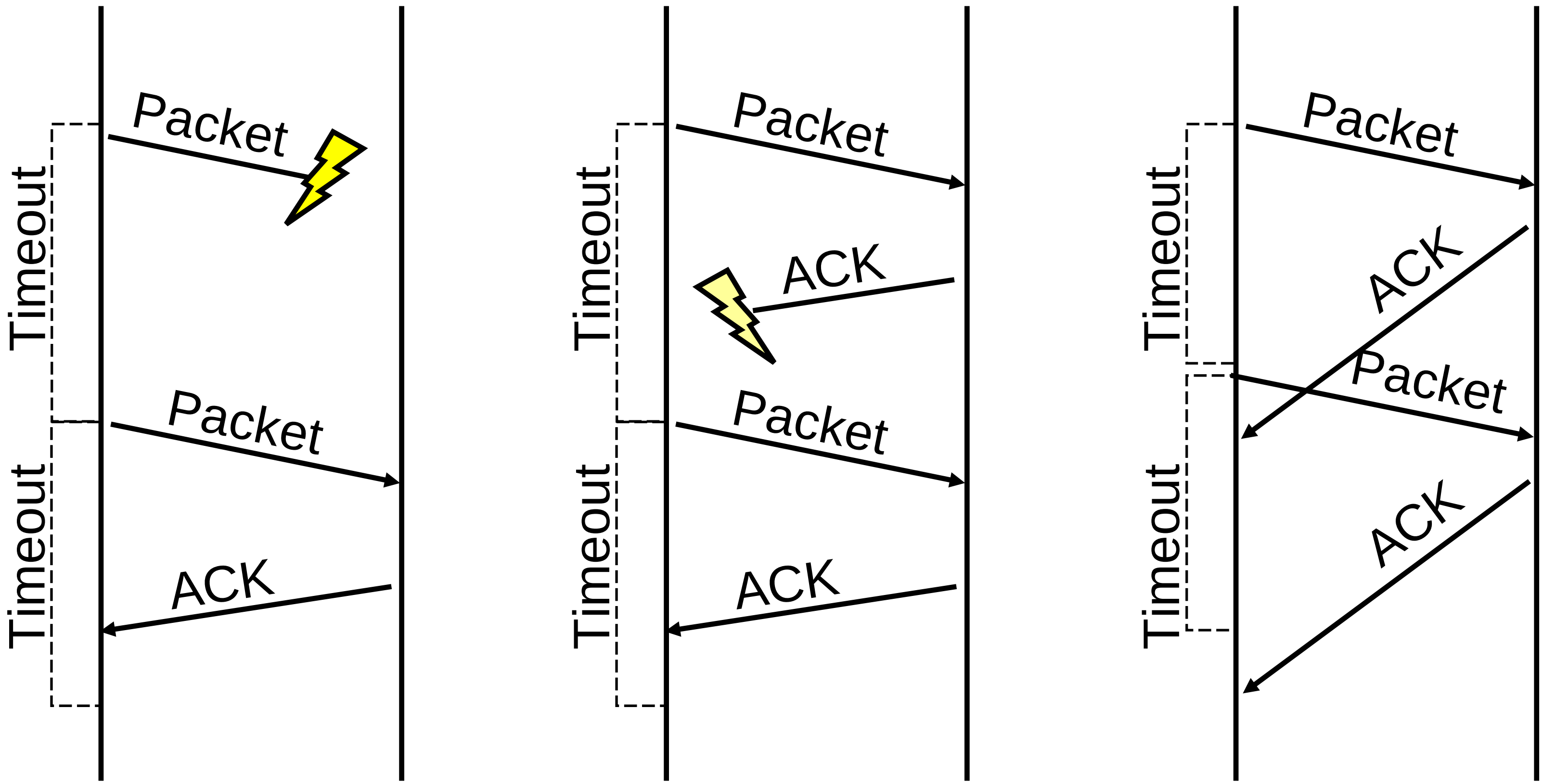
- Receiver sends ACK when it receives packet
- Sender waits for ACK.
- If ACK not received within some timeout period, resend packet

- "stop and wait"
  - One packet at a time…



Source: Freedman (partial)

Reasons for Retransmission



Packet lost

ACK lost
DUPLICATE
PACKET

Early timeout
DUPLICATE
PACKETS

Source: Freedman

How Long Should Sender Wait?

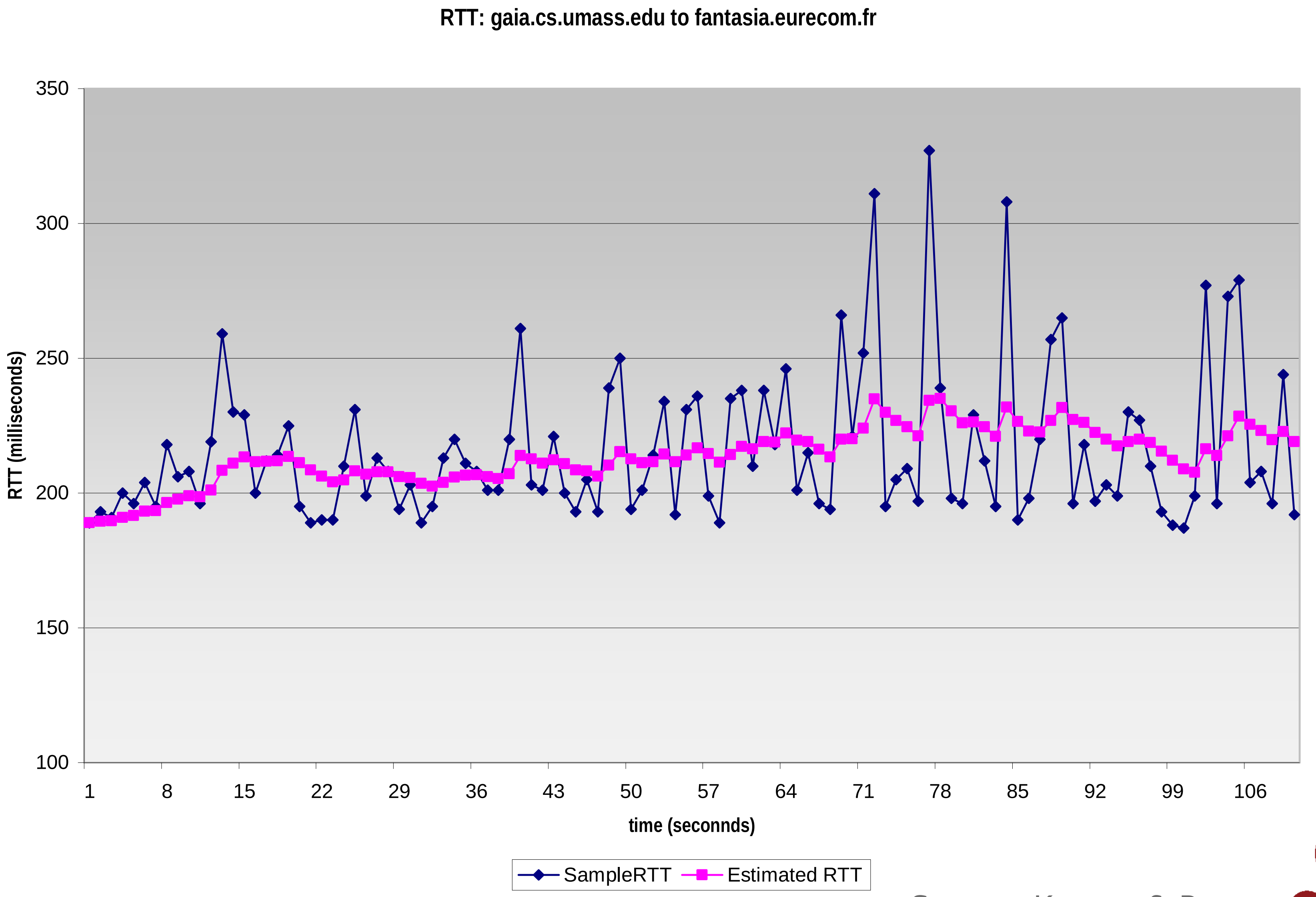
- Too short? Wasted re-transmissions
- Too long? Excessive delays when packet lost

- TCP sets timeout as function of Round Trip Time
  - ACK should arrive after RTT + fudge factor for queuing

- How does sender know RTT?
  - Can estimate RTT by watching the ACKs
  - Smooth estimate: Exponentially-weighted moving avg (EWMA)
    - **`EstimatedRTT = (1–a) * EstimatedRTT +  a * SampleRTT`**
  - Typical value: `a = 0.125`

Source: Freedman (partial) + Kurose & Ross (partial)

Example RTT Estimation



Source: Kurose & Ross

TCP Round Trip Time and Timeout

Setting the timeout

- **`EstimatedRTT`** plus "safety margin"
  - large variation in **`EstimatedRTT`** → larger safety margin
- first estimate of how much SampleRTT deviates from EstimatedRTT:

DevRTT = (1-β)*DevRTT + β*|SampleRTT-EstimatedRTT|

(typically, β = 0.25)

- Then set timeout interval:

TimeoutInterval = EstimatedRTT + 4*DevRTT

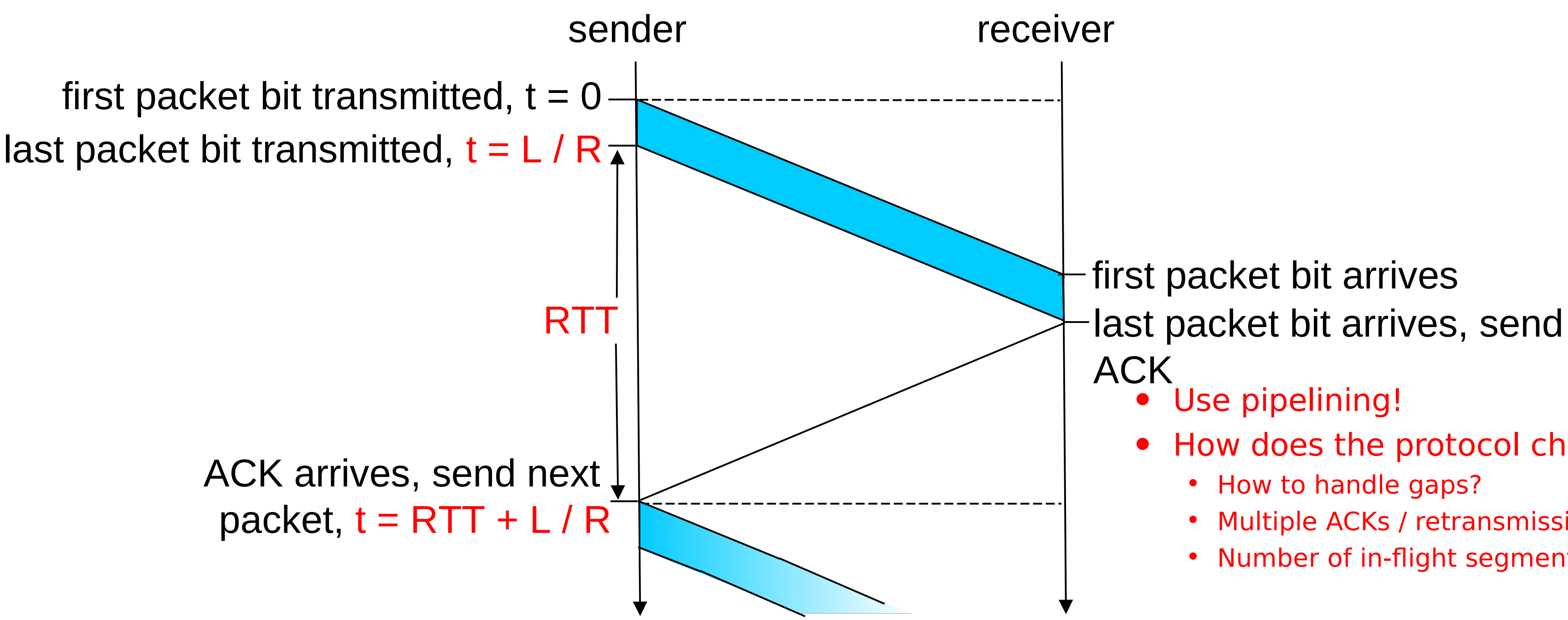
Source: Kurose & Ross

A Flaw in This Approach

- ACK acknowledges receipt of data, not transmission
- Consider a retransmission of a lost packet
  - If assume ACK with 1st transmission, SampleRTT too large
- Consider a duplicate packet
  - If assume ACK with 2nd transmission, SampleRTT too small
- Simple solution in the Karn/Partridge algorithm
  - Only collect samples for segments sent one single time
  - On retransmission, new_timeout = 2 * timeout

Source: Freedman (partial)

Well tuned timeouts help, but…

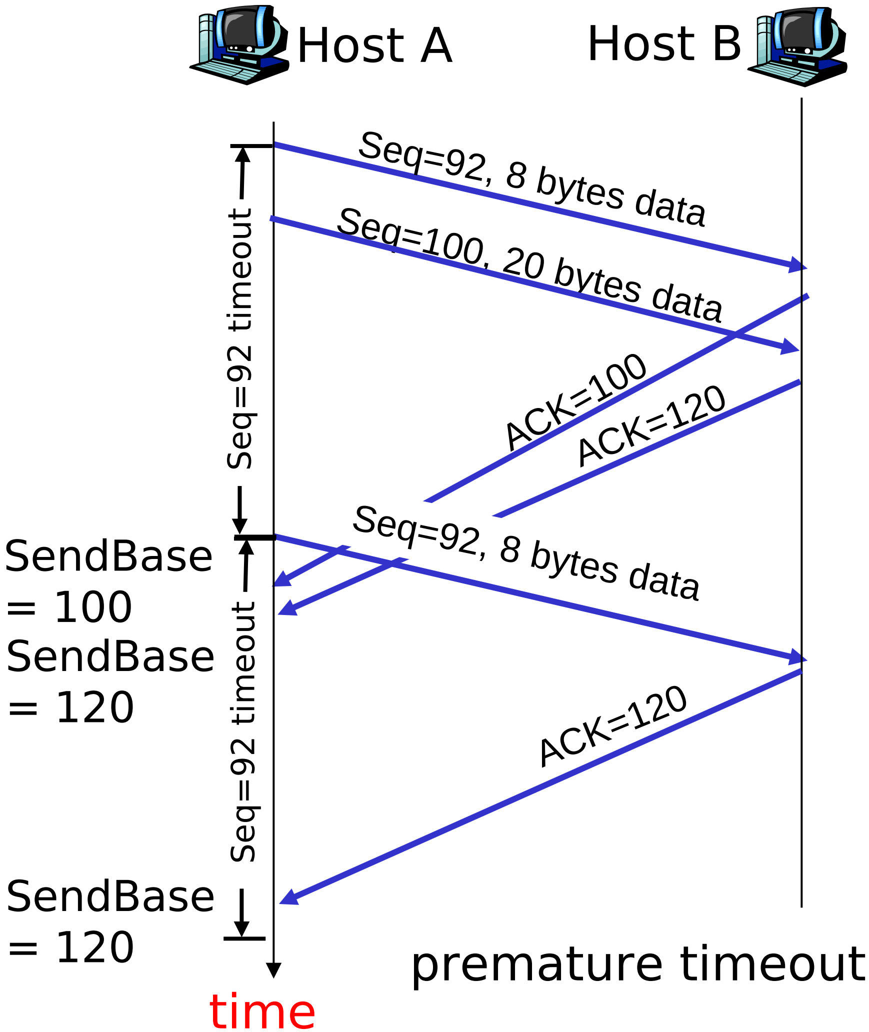


- Use pipelining!
- How does the protocol change?
  - How to handle gaps?
  - Multiple ACKs / retransmissions?
  - Number of in-flight segments?

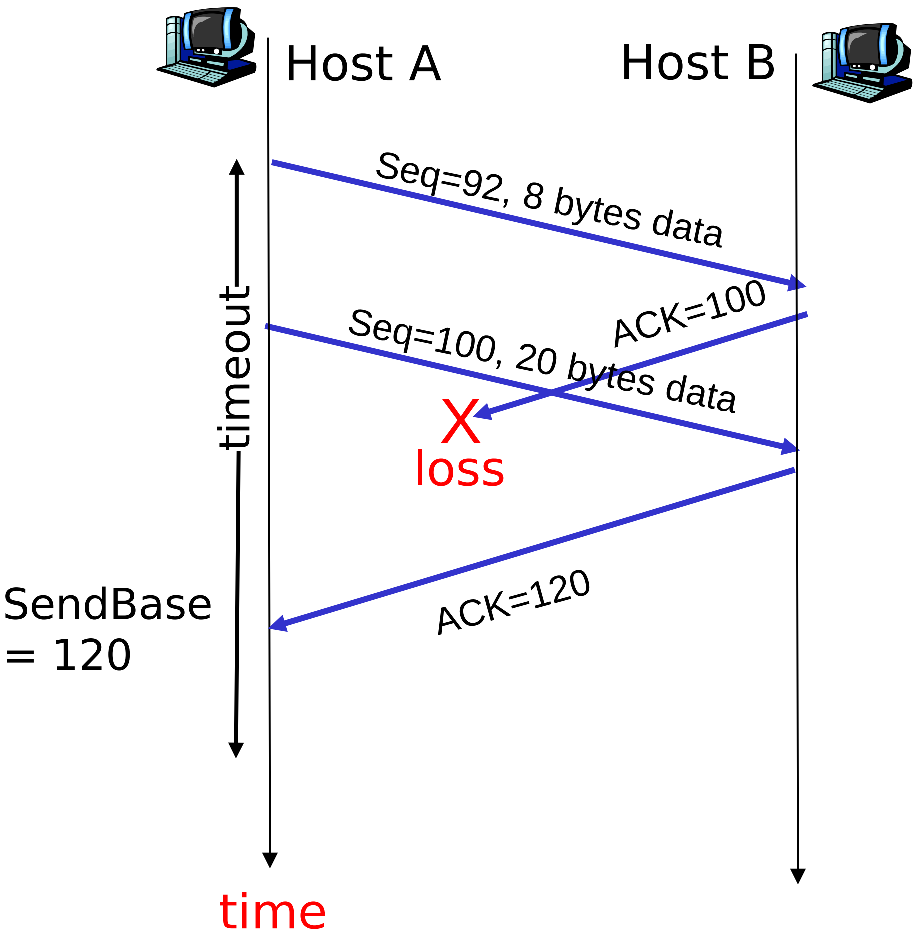
$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

Source: Kurose & Ross (partial)

TCP Retransmission and Cumulative ACK



premature timeout



Cumulative ACK scenario

Source: Kurose & Ross (partial)

TCP Fast Retrasmit

- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

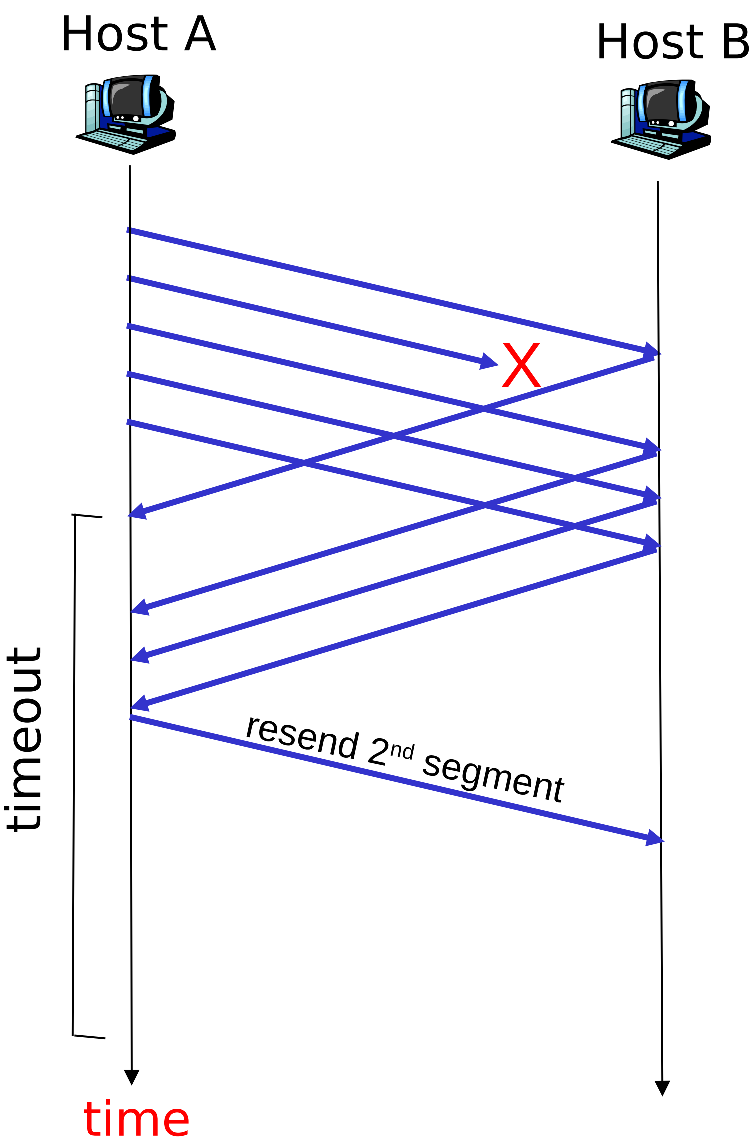
- if sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - fast retransmit: resend segment before timer expires

Source: Kurose & Ross

TCP Fast Retransmit

- Resending a segment after triple duplicate ACK

- Triple duplicate ACK works as a logical NACK



Source: Kurose & Ross (partial)

Effectiveness of Fast Retransmit

- When does Fast Retransmit work best?
  - Long transfers: High likelihood of many pkts in flight
  - Large window: High likelihood of many packets in flight
  - Low loss burstiness: Higher likelihood that later pkts arrive

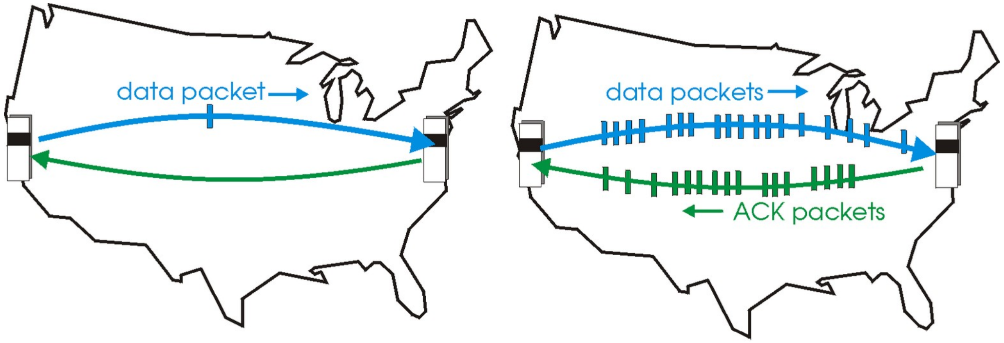
- Implications for Web traffic
  - Most Web objects are short (e.g., 10 packets)
  - So, often aren't many packets in flight
  - … making fast retransmit less likely to "kick in"
  - … another reason for persistent connections!

Source: Freedman

Increasing TCP throughput

- **Problem:** Stop-and-wait + timeouts are inefficient
  - Only one TCP segment " in flight" at time ✔
- **Solution:** Send multiple packets at once

- **Problem:** How many w/o overwhelming receiver?
- **Solution:** Determine " window size"



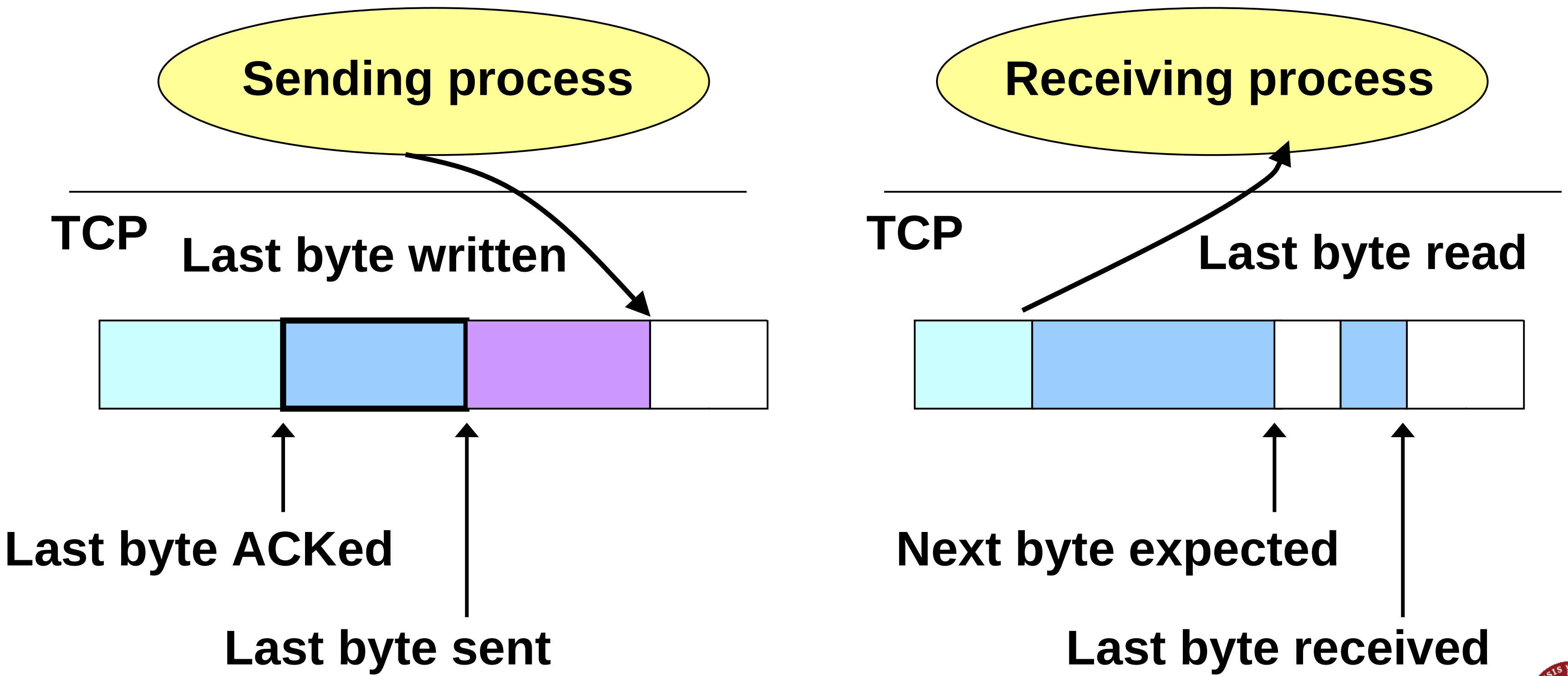
(a) a stop-and-wait protocol in operation (b) a pipelined protocol in operation

Image Source: Kurose & Ross

Source: Freedman (partial)

Flow Control: Sliding Window

- Allow a larger amount of data "in flight"
  - Sender can get ahead of receiver, though not *too far*

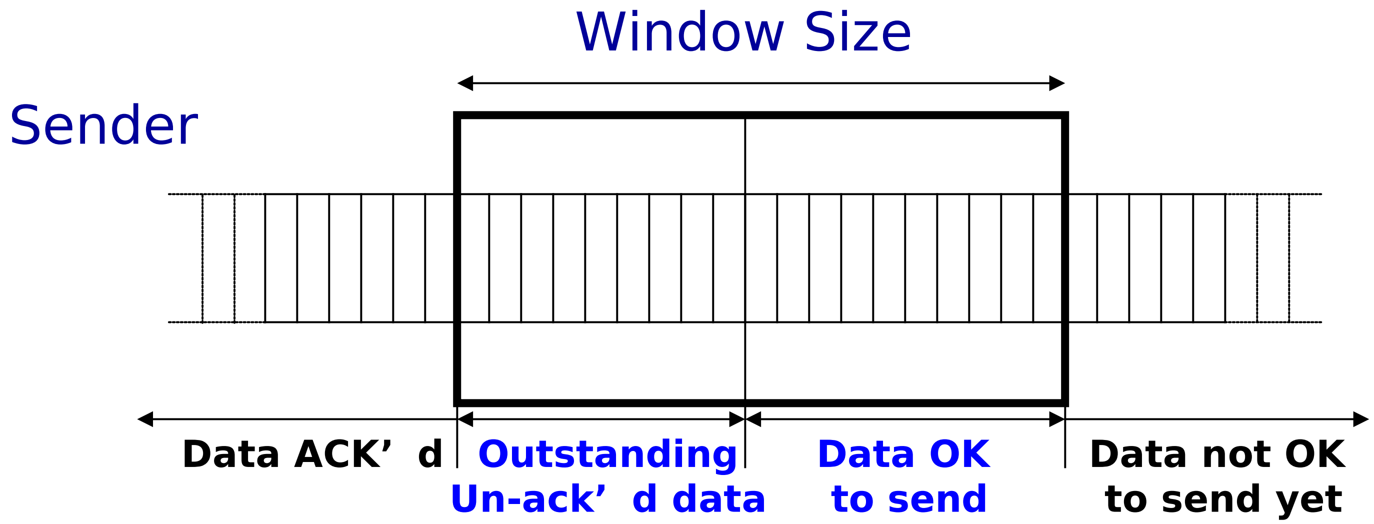


Source: Freedman (partial)

Flow Control: Receiver Buffering

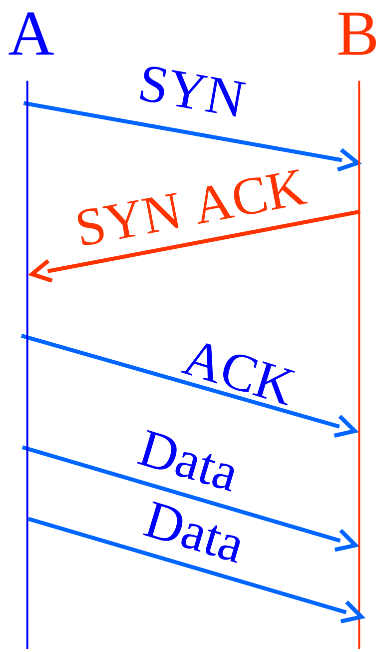
- Window size
  - Amount that can be sent w/o ACK, because receiver can buffer
- Receiver advertises window to sender
  - Tells amount of free space left (in **bytes**)

    `RcvWindow = RcvBuffer-[LastByteRcvd – LastByteRead]`
  - Sender agrees not to exceed this amount



Source: Freedman (partial)

Establishing a TCP Connection



Each host tells its ISN to other host

- Three-way handshake to establish connection
  - Host A sends a **SYN**chronize (open) to the host B
  - Host B returns a SYN ACKnowledgment (**SYN ACK**)
  - Host A sends an **ACK** to acknowledge the SYN ACK

Source: Freedman

Step 1: A’s Initial SYN Packet

Flags: SYN
FIN
RST
PSH
URG
ACK

<table>
<tr><td colspan="3">A’s port</td><td>B’s port</td></tr>
<tr><td colspan="4">A’s Initial Sequence Number</td></tr>
<tr><td colspan="4">Acknowledgment</td></tr>
<tr><td>20</td><td>0</td><td>Flags</td><td>Advertised window</td></tr>
<tr><td colspan="3">Checksum</td><td>Urgent pointer</td></tr>
<tr><td colspan="4">Options (variable)</td></tr>
</table>

A tells B it wants to open a connection…

Source: Freedman

Step 2: B’s SYN-ACK Packet

Flags: SYN
FIN
RST
PSH
URG
ACK

B’s port			A’s port
B’s Initial Sequence Number			
A’s ISN plus 1			
20	0	Flags	Advertised window
Checksum			Urgent pointer
Options (variable)			

B tells A it accepts, and is ready to hear the next byte…

… upon receiving this packet, A can start sending data

Source: Freedman

Step 3: A's ACK of the SYN-ACK

Flags:
- SYN
- FIN
- RST
- PSH
- URG
- ACK

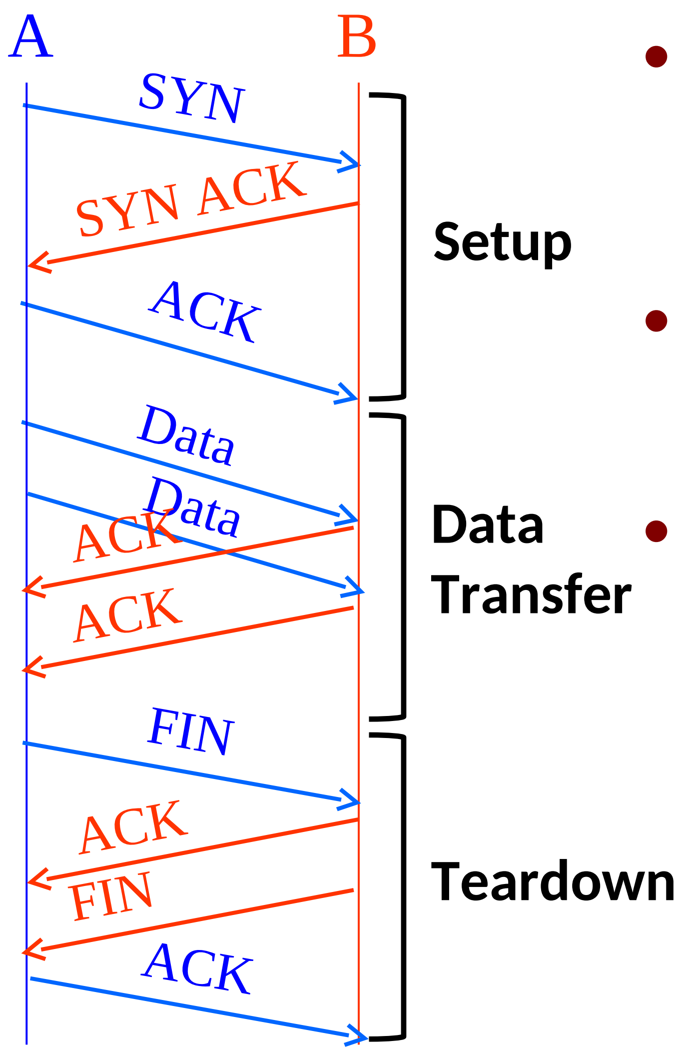
A's port		B's port	
Sequence number			
B's ISN plus 1			
20	0	Flags	Advertised window
Checksum		Urgent pointer	
Options (variable)			

A tells B it is okay to start sending...

... upon receiving this packet, B can start sending data

Source: Freedman

Tearing Down the Connection



- Closing a connection
  - Process done writing: invokes close()
  - Once TCP sends all outstanding byte, TCP sends a FINish message
- Receiving a FINish
  - Process reading data from socket
  - Eventually, read attempt returns EOF
- Tear-down is two-way
  - FIN to close, but receive remaining
  - Other host ACKs the FIN
  - Reset (RST) to close and not receive remaining: error condition

Source: Freedman

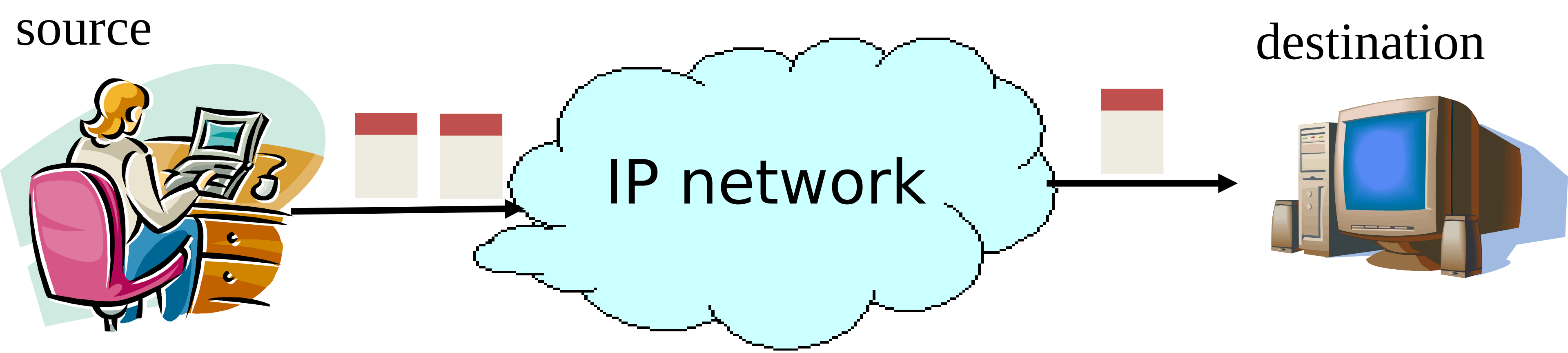
Congestion Control

- Congestion in IP networks
  - Unavoidable due to best-effort service model
  - IP philosophy: decentralized control at end hosts
- Congestion control by the TCP senders
  - Infers congestion is occurring (e.g., from packet losses)
  - Slows down to alleviate congestion, for the greater good
- TCP congestion-control algorithm
  - Additive-increase, multiplicative-decrease
  - Slow start and slow-start restart

Source: Freedman

IP Best-Effort Design Philosophy

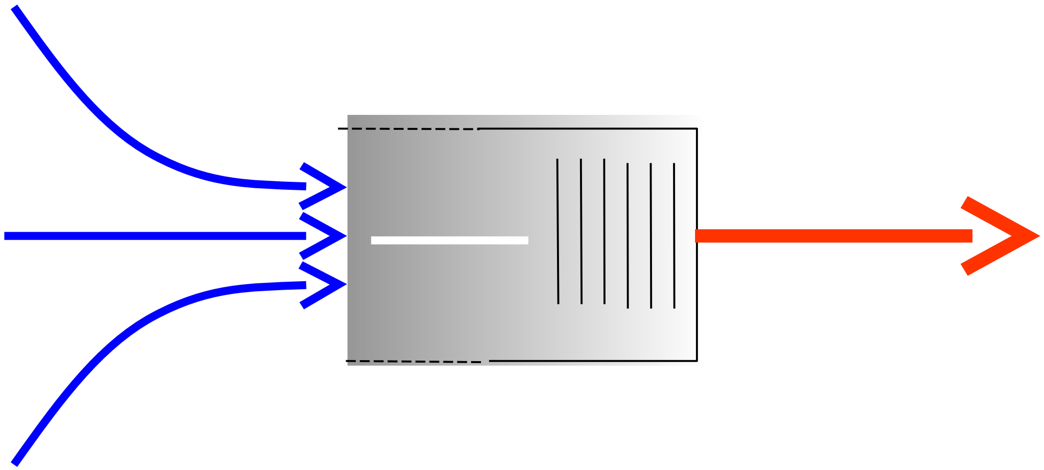
- Best-effort delivery
  - Let everybody send
  - Network tries to deliver what it can
  - … and just drop the rest



Source: Freedman

Congestion is Unavoidable

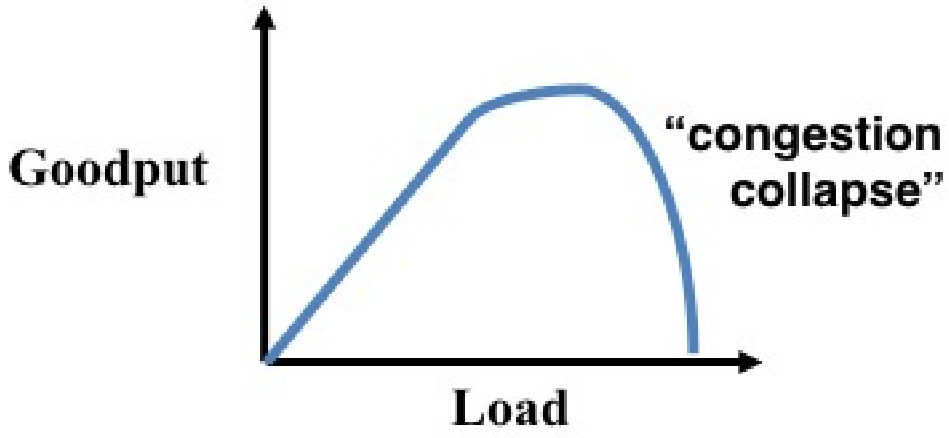
- Two packets arrive at same time
  - Router can only transmit one: must buffer or drop other
- If many packets arrive in short period of time
  - Router cannot keep up with the arriving traffic
  - Buffer may eventually overflow



Source: Freedman

The Problem of Congestion

- What is congestion?
  - Load is higher than capacity
- What do IP routers do?
  - Drop the excess packets
- Why is this bad?
  - Wasted bandwidth for re-transmissions



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

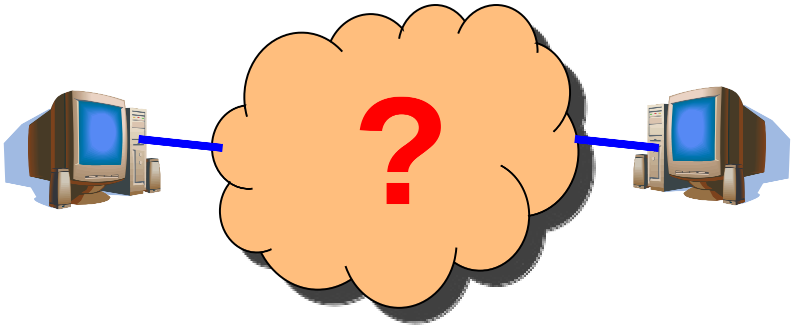
Source: Freedman

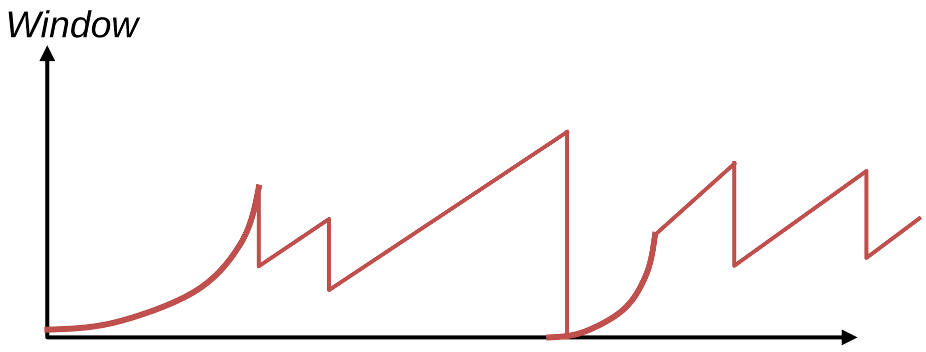
Ways to Deal With Congestion

- Ignore the problem
  - Many dropped (and retransmitted) packets
  - Can cause congestion collapse
- Reservations, like in circuit switching
  - Pre-arrange bandwidth allocations
  - Requires negotiation before sending packets
- Pricing
  - Don’t drop packets for the high-bidders
  - Requires a payment model, and low-bidders still dropped
- Dynamic adjustment (TCP)
  - Every sender infers the level of congestion
  - Each adapts its sending rate “ for the greater good”

Source: Freedman

Inferring From Implicit Feedback





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

- TCP keeps congestion window, as in the graph
- Can you explain behavior? Why are there increases and drops?
- Why is there a “sawtooth”?

Source: Freedman (partial)

TCP Congestion Window

- Each TCP sender maintains a congestion window
  - Max number of bytes to have in transit (not yet ACK' d)
- Adapting the congestion window
  - Decrease upon losing a packet: backing off
  - Increase upon success: optimistically exploring
  - Always struggling to find right transfer rate
- Tradeoff
  - Pro: avoids needing explicit network feedback
  - Con: continually under- and over-shoots " right" rate

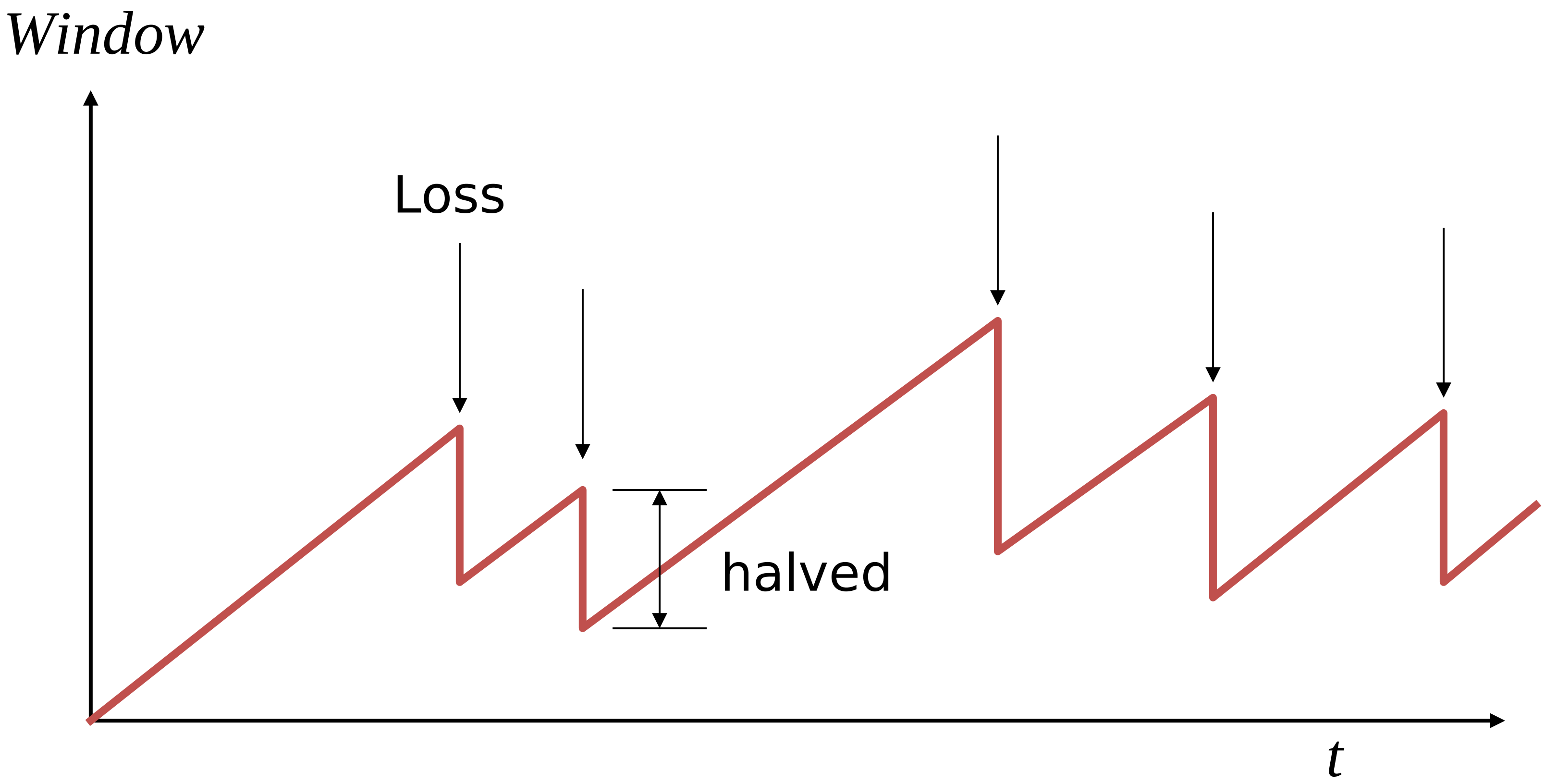
Source: Freedman

Additive Increase, Multiplicative Decrease (AIMD)

- How much to adapt?
  - Additive increase: On success of last window of data, increase window by 1 Max Segment Size (MSS)
  - Multiplicative decrease: On loss of packet, divide congestion window in half
- Much quicker to slow than speed up!
  - Over-sized windows (causing loss) are much worse than under-sized windows (causing lower throughput)
  - AIMD: A necessary condition for stability of TCP

Source: Freedman (partial)

Leads to TCP "Sawtooth"



Source: Freedman

Receiver Window vs. Congestion Window

- Flow control
  - Keep a *fast sender* from overwhelming *a slow receiver*
- Congestion control
  - Keep a *set of senders* from overloading the *network*

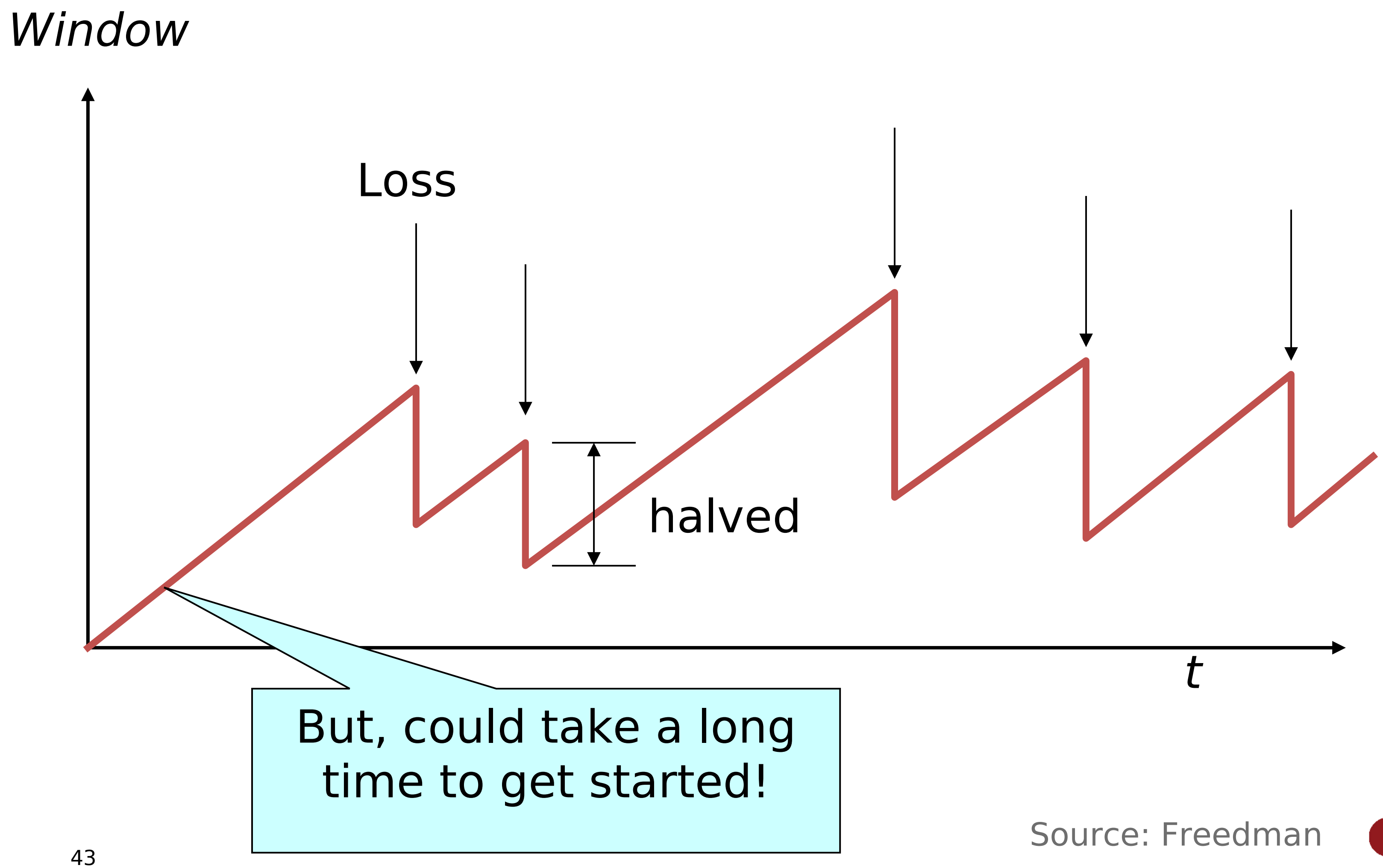
- Different concepts, but similar mechanisms
  - TCP flow control: receiver window
  - TCP congestion control: congestion window
  - Sender TCP window =

    min { congestion window, receiver window }

Source: Freedman

How Should a New Flow Start?

Start slow (a small CWND) to avoid overloading network



Source: Freedman

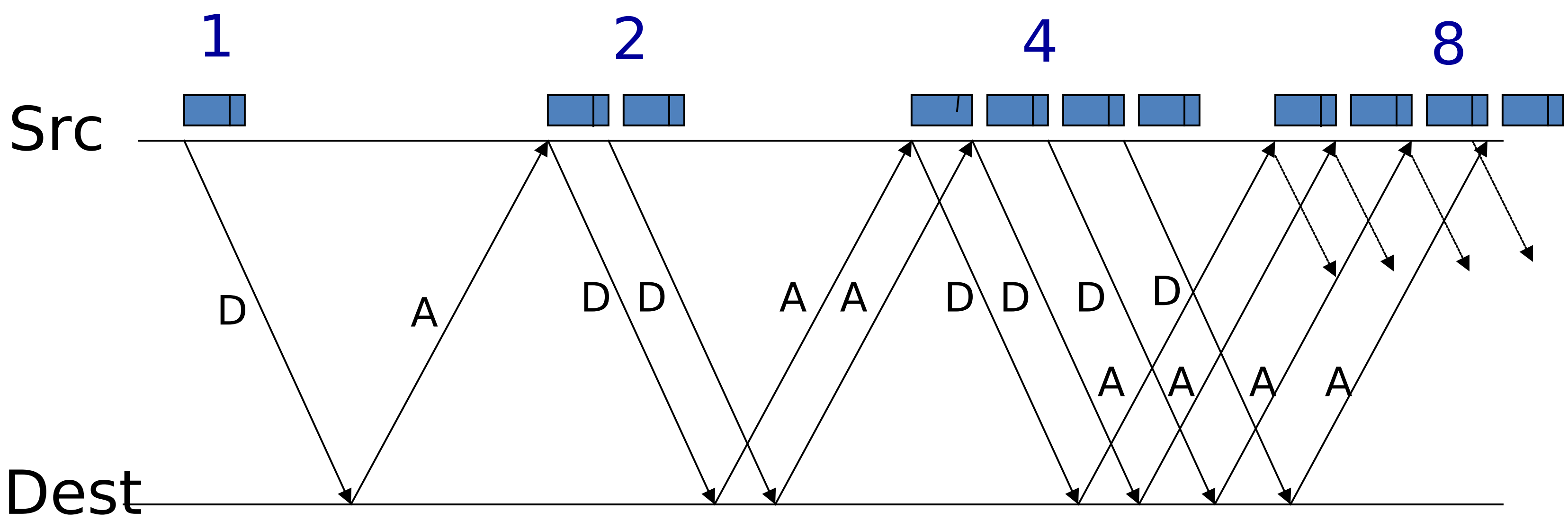
“Slow Start” Phase

- Start with a small congestion window
  - Initially, CWND is 1 MSS
  - So, initial sending rate is MSS / RTT
- Could be pretty wasteful
  - Might be much less than actual bandwidth
  - Linear increase takes a long time to accelerate
- Slow-start phase (really “ fast start”)
  - Sender starts at a slow rate (hence the name)
  - … but increases rate exponentially until the first loss

Source: Freedman

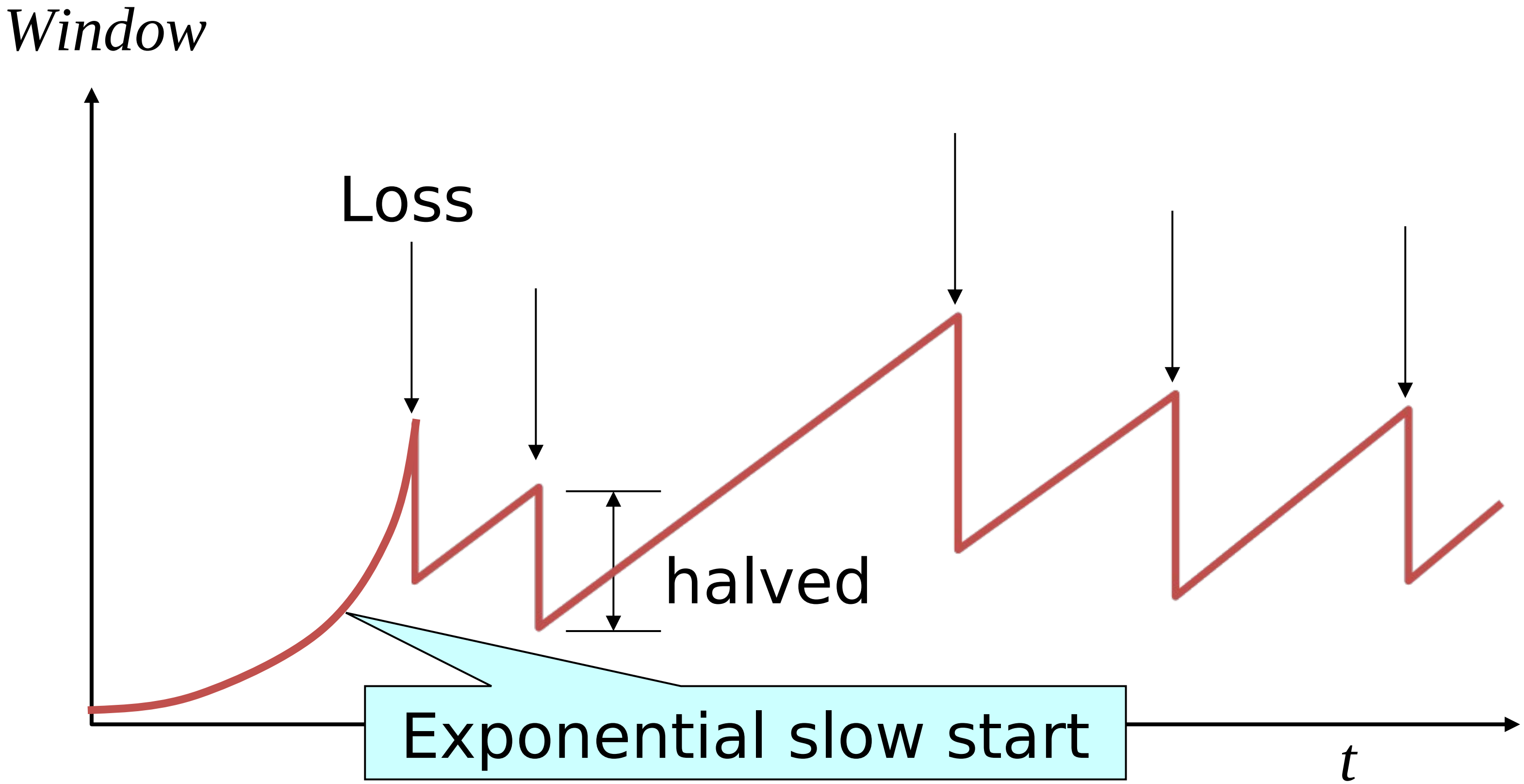
Slow Start in Action

Double CWND per round-trip time



Source: Freedman

Slow Start and the TCP Sawtooth



- So-called because TCP originally had no congestion control
  - Source would start by sending an entire receiver window
  - Led to congestion collapse!

Source: Freedman

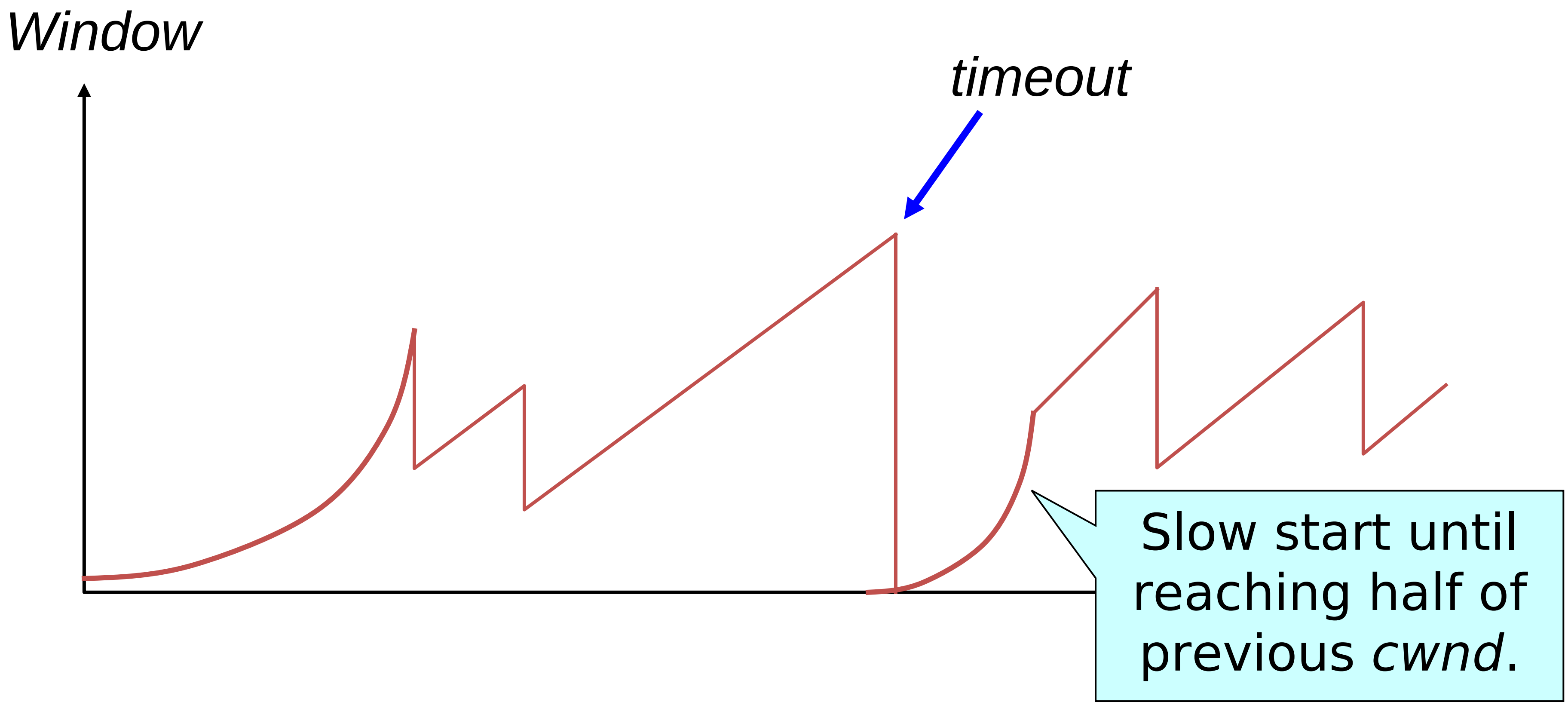
Two Kinds of Loss in TCP

- Timeout
  - Packet n is lost and detected via a timeout
    - When? n is last packet in window, or all packets in flight lost
  - After timeout, blasting entire CWND would cause another burst
  - Better to start over with a low CWND

- Triple duplicate ACK
  - Packet n is lost, but packets n+1, n+2, etc. arrive
    - How detected? Multiple ACKs that receiver waiting for n
    - When? Later packets after n received
  - After triple duplicate ACK, sender quickly resends packet n
  - Do a multiplicative decrease and keep

Source: Freedman

Repeating Slow Start After Timeout



Slow-start restart: Go back to CWND of 1, but take advantage of knowing the previous value of CWND.

Source: Freedman

Repeating Slow Start After Idle Period

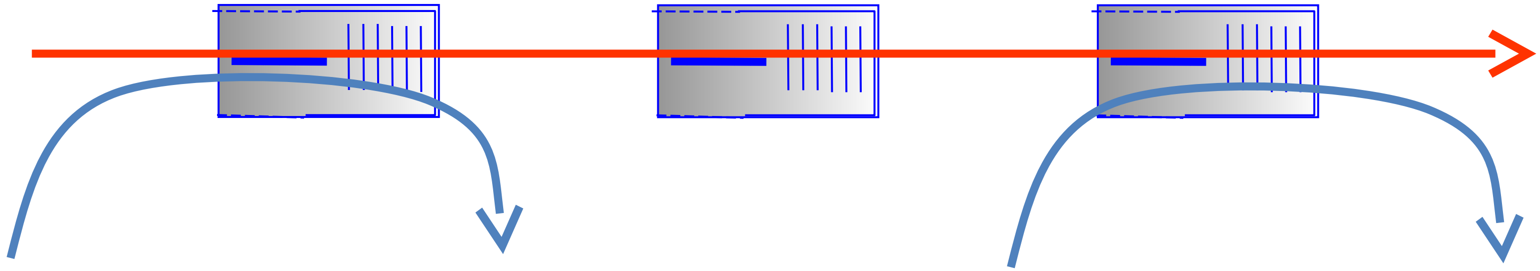
- Suppose a TCP connection goes idle for a while
- Eventually, the network conditions change
  - Maybe many more flows are traversing the link
- Dangerous to start transmitting at the old rate
  - Previously-idle TCP sender might blast network
  - … causing excessive congestion and packet loss
- So, some TCP implementations repeat slow start
  - Slow-start restart after an idle period

Source: Freedman

TCP Achieves Some Notion of Fairness

- Effective utilization is not only goal
  - We also want to be *fair* to various flows
  - … but what does *that* mean?

- Simple definition: equal shares of the bandwidth
  - N flows that each get 1/N of the bandwidth?
  - But, what if flows traverse different paths?
  - Result: bandwidth shared in proportion to RTT



Source: Freedman

What About Cheating?

- Some folks are more fair than others
  - Running multiple TCP connections in parallel (BitTorrent)
  - Modifying the TCP implementation in the OS
    - Some cloud services start TCP at > 1 MSS
  - Use the User Datagram Protocol
- What is the impact
  - Good guys slow down to make room for you
  - You get an unfair share of the bandwidth
- Possible solutions?
  - Routers detect cheating and drop excess packets?
  - Per user/customer fairness?
  - Peer pressure?

Source: Freedman

Summary

- UDP
  - basic multiplexing, checksums
- TCP & reliable transfer
  - Segments, sequence numbers, automatic repeat requests
  - Timeout estimation
  - Pipelining, cumulative ACK, fast retransmit
  - Flow control: receiver window
  - Congestion control: congestion window, AIMD, slow start, slow start restart