

# Computer Networking

Lent Term

M/W/F 11:00-12:00

LT1 in Gates Building

Slide Set 3 (Topic 5)

Andrew W. Moore

Andrew.Moore@cl.cam.ac.uk

2018-2019

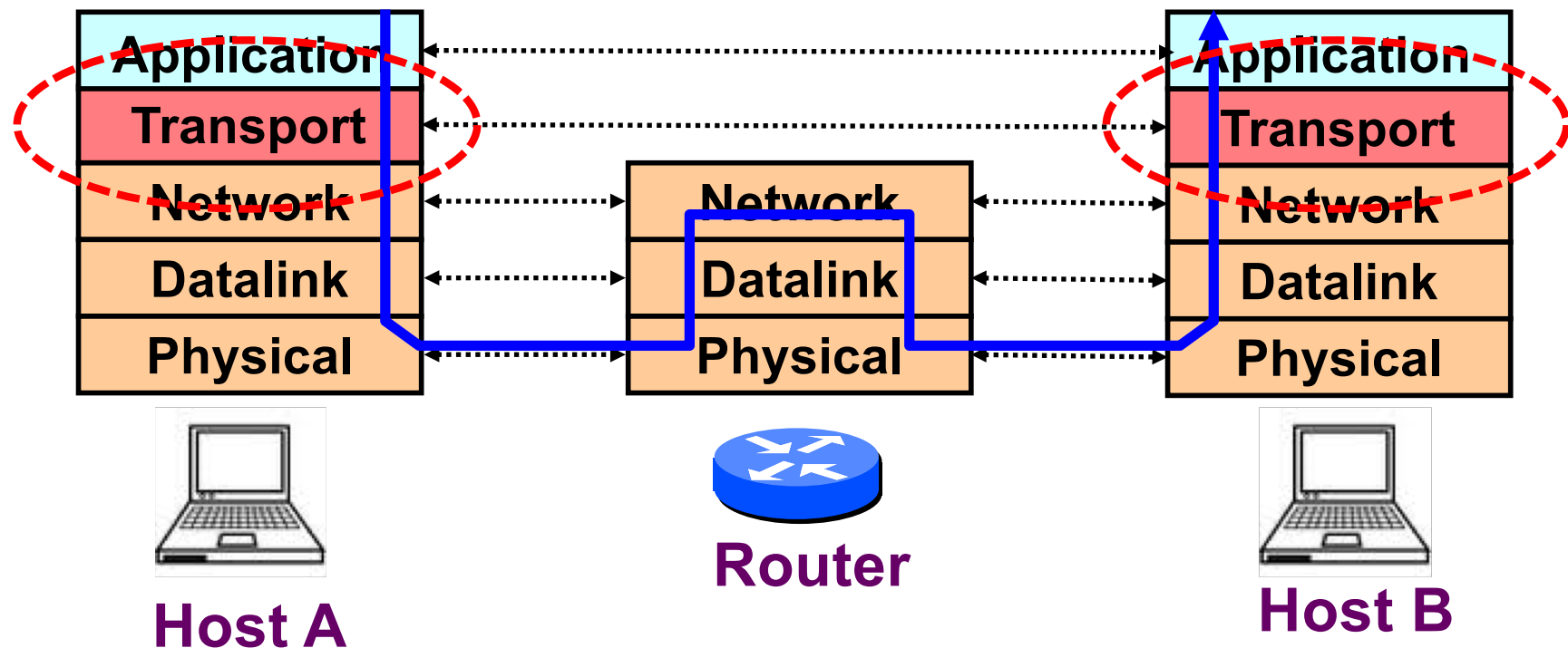
# Topic 5 – Transport

## Our goals:

- understand principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
  - beyond TCP
- learn about transport layer protocols in the Internet:
  - UDP: connectionless transport
  - TCP: connection-oriented transport
  - TCP congestion control
  - TCP flow control

# Transport Layer

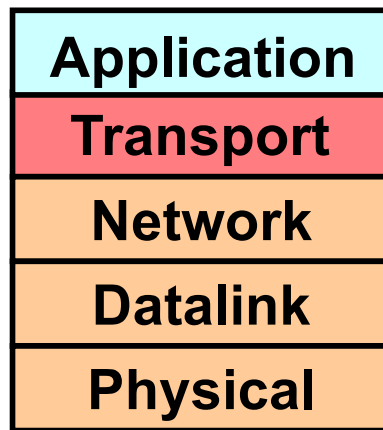
- Commonly a layer **at end-hosts**, between the application and network layer



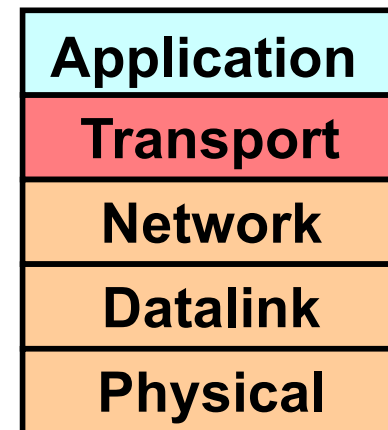
# Why a transport layer?

- IP packets are addressed to a host but end-to-end communication is between application processes at hosts
  - Need a way to decide which packets go to which applications (*more multiplexing*)

# Why a transport layer?

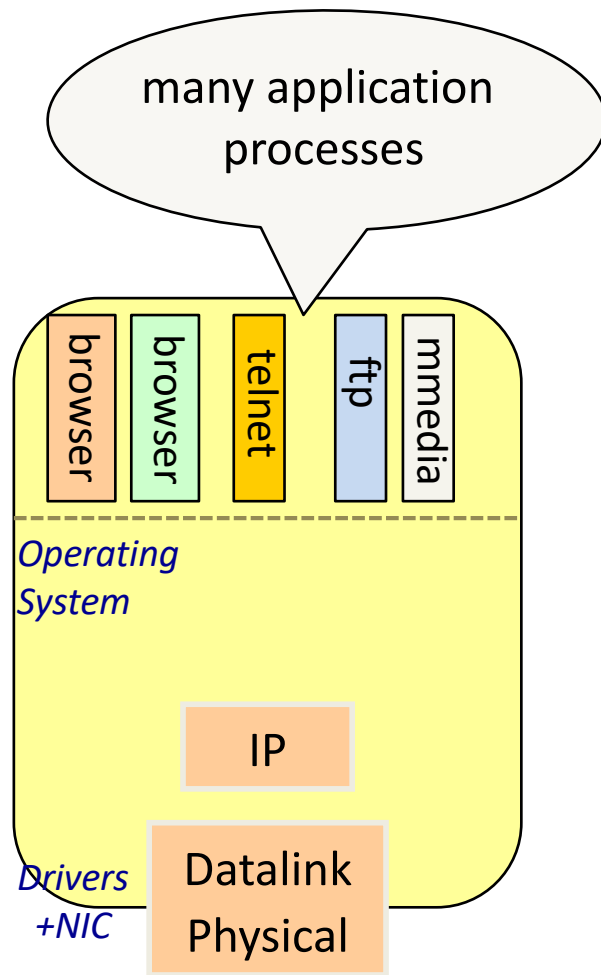


**Host A**

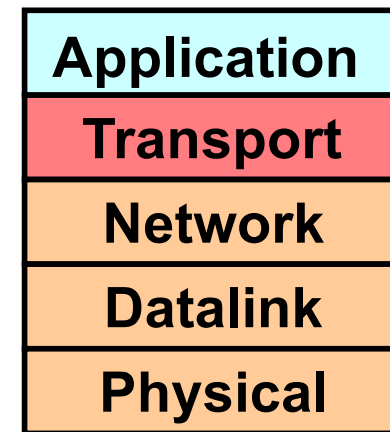


**Host B**

# Why a transport layer?

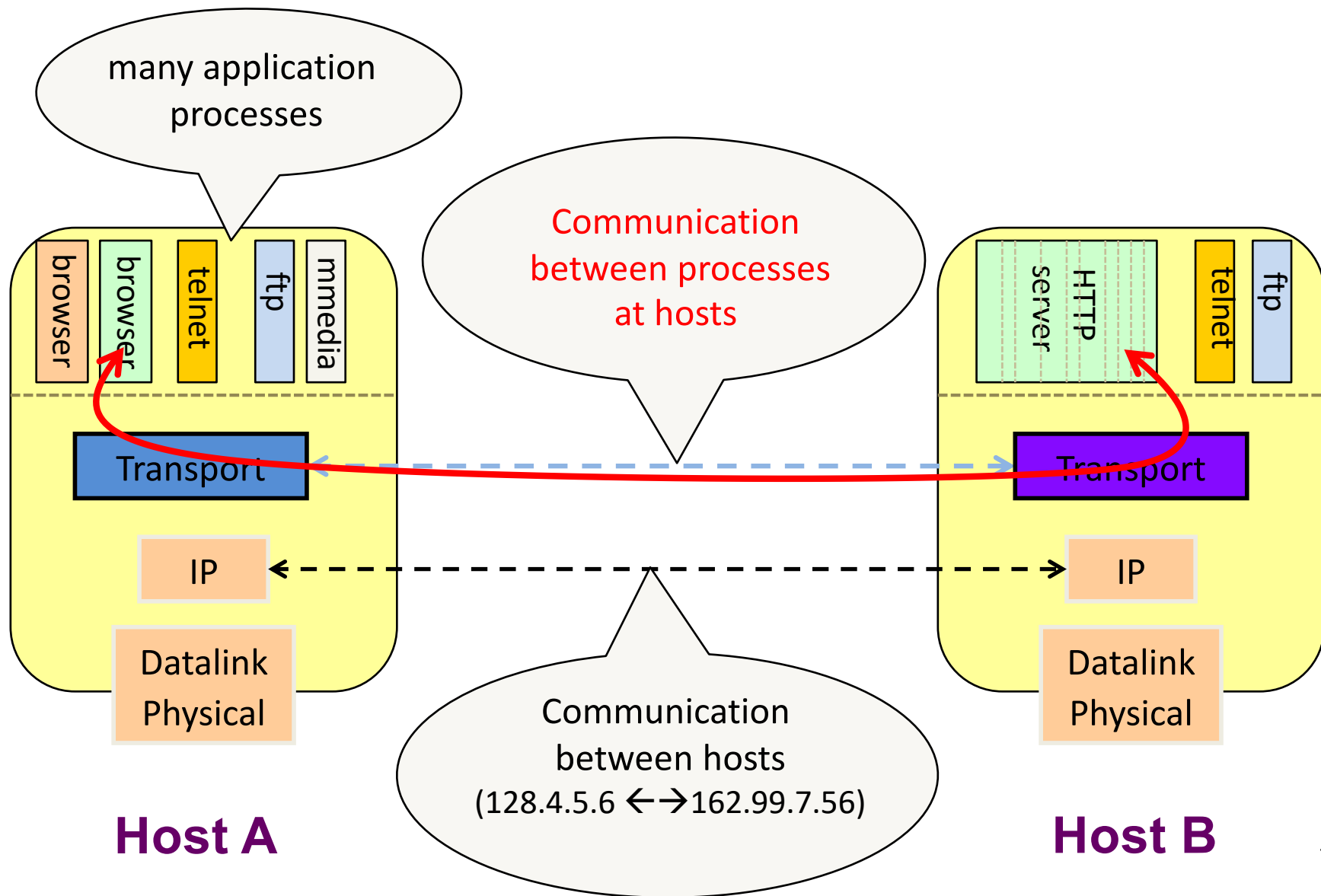


**Host A**



**Host B**

# Why a transport layer?



# Why a transport layer?

- IP packets are addressed to a host but end-to-end communication is between application processes at hosts
  - Need a way to decide which packets go to which applications (mux/demux)
- IP provides a weak service model (*best-effort*)
  - Packets can be corrupted, delayed, dropped, reordered, duplicated
  - No guidance on how much traffic to send and when
  - Dealing with this is tedious for application developers



# Role of the Transport Layer

- Communication between application processes
  - Multiplexing between application processes
  - Implemented using *ports*

# Role of the Transport Layer

- Communication between application processes
- Provide common end-to-end services for app layer [optional]
  - Reliable, in-order data delivery
  - Paced data delivery: flow and congestion-control
    - too fast may overwhelm the network
    - too slow is not efficient

# Role of the Transport Layer

- Communication between processes
- Provide common end-to-end services for app layer [optional]
- TCP and UDP are the common transport protocols
  - also SCTP, MTCP, SST, RDP, DCCP, ...

# Role of the Transport Layer

- Communication between processes
- Provide common end-to-end services for app layer [optional]
- TCP and UDP are the common transport protocols
- **UDP is a minimalist, no-frills transport protocol**
  - only provides mux/demux capabilities

# Role of the Transport Layer

- Communication between processes
- Provide common end-to-end services for app layer [optional]
- TCP and UDP are the common transport protocols
- UDP is a minimalist, no-frills transport protocol
- TCP is the *totus porcus* protocol
  - offers apps a reliable, in-order, byte-stream abstraction
  - with congestion control
  - but **no** performance (delay, bandwidth, ...) guarantees

# Role of the Transport Layer

- Communication between processes
  - mux/demux from and to application processes
  - implemented using ports

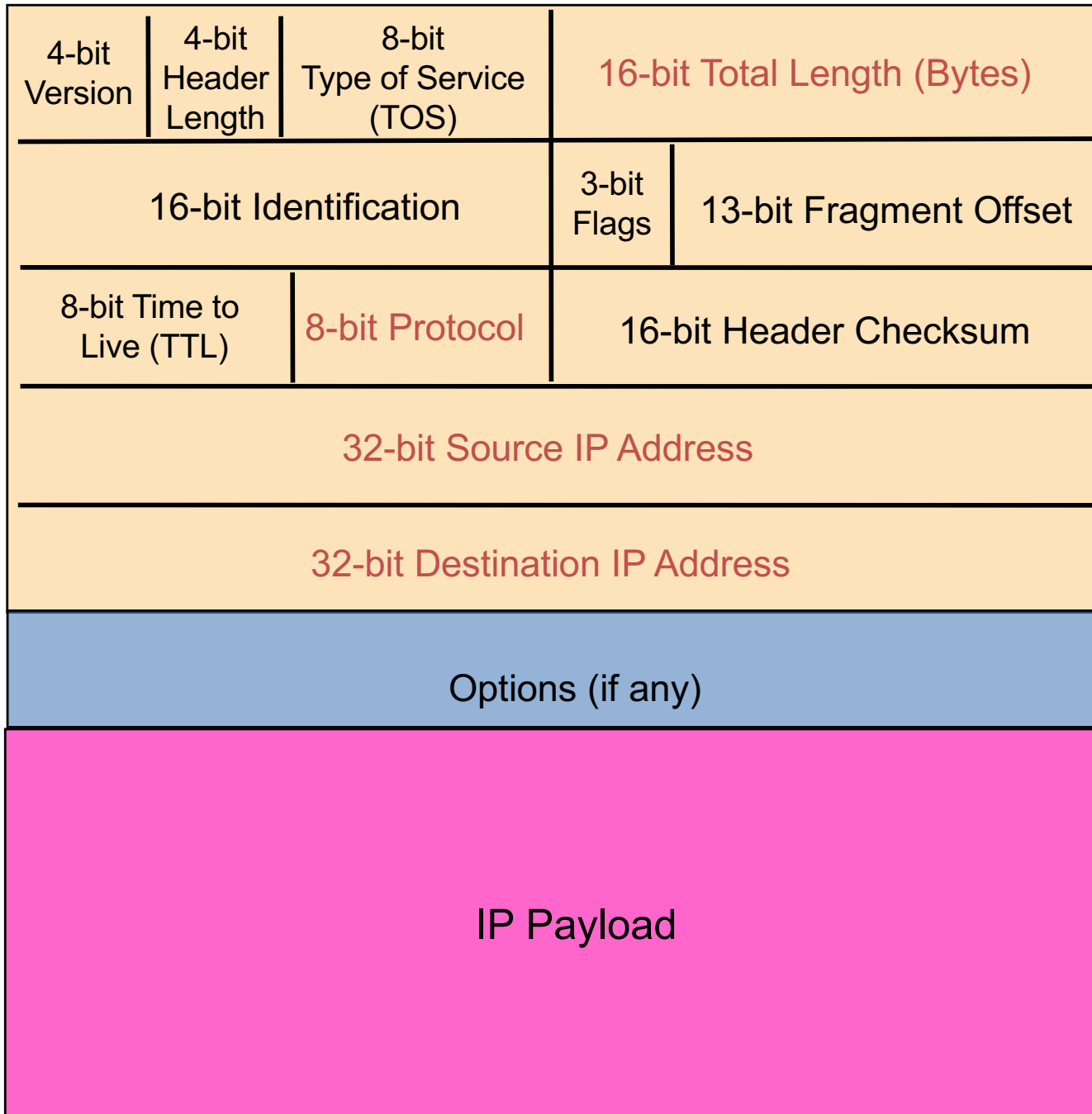
# Context: Applications and Sockets

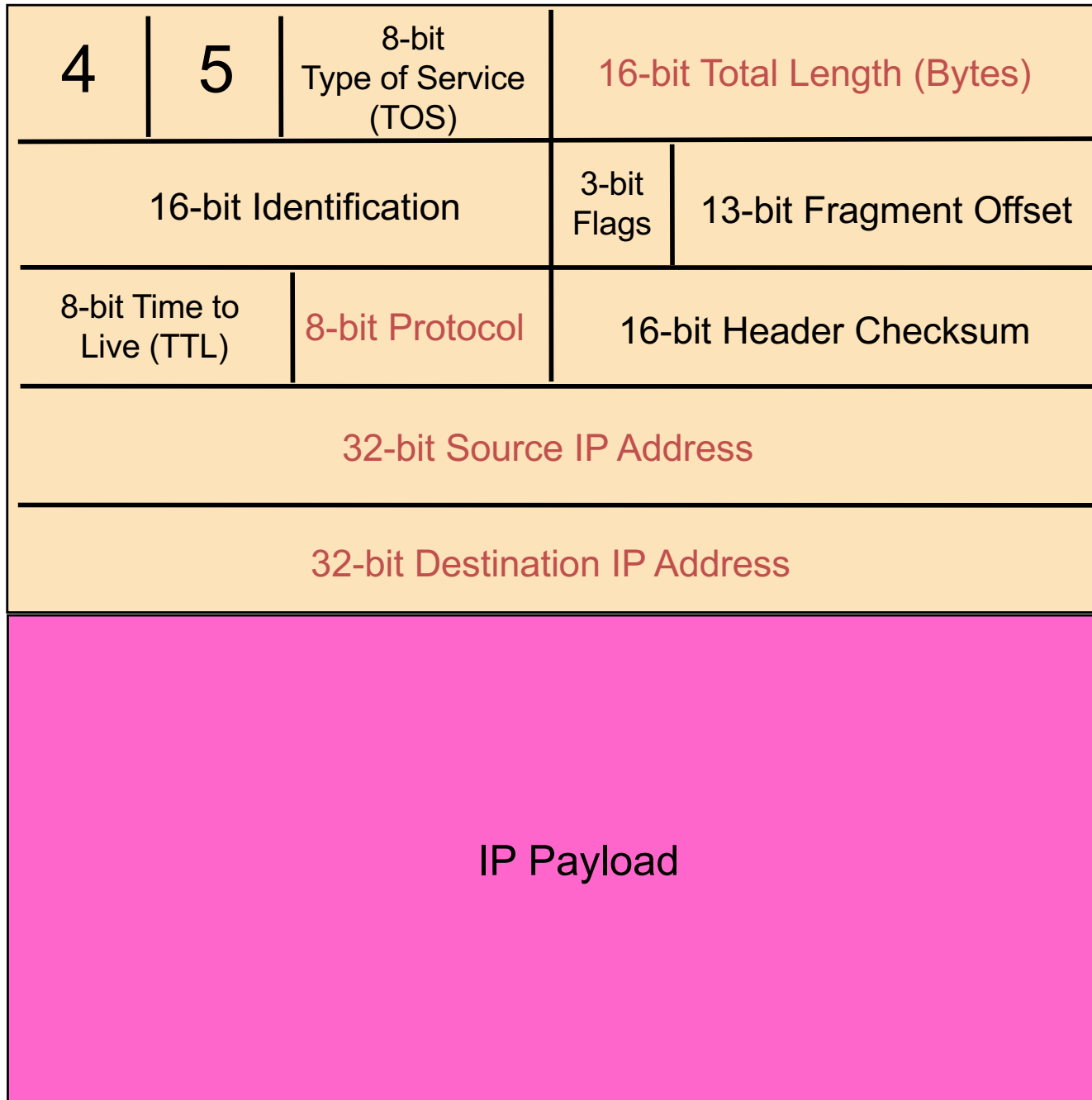
- Socket: software abstraction by which an application process exchanges network messages with the (transport layer in the) operating system
  - `socketID = socket(..., socket.TYPE)`
  - `socketID.sendto(message, ...)`
  - `socketID.recvfrom(...)`
- Two important types of sockets
  - UDP socket: TYPE is `SOCK_DGRAM`
  - TCP socket: TYPE is `SOCK_STREAM`

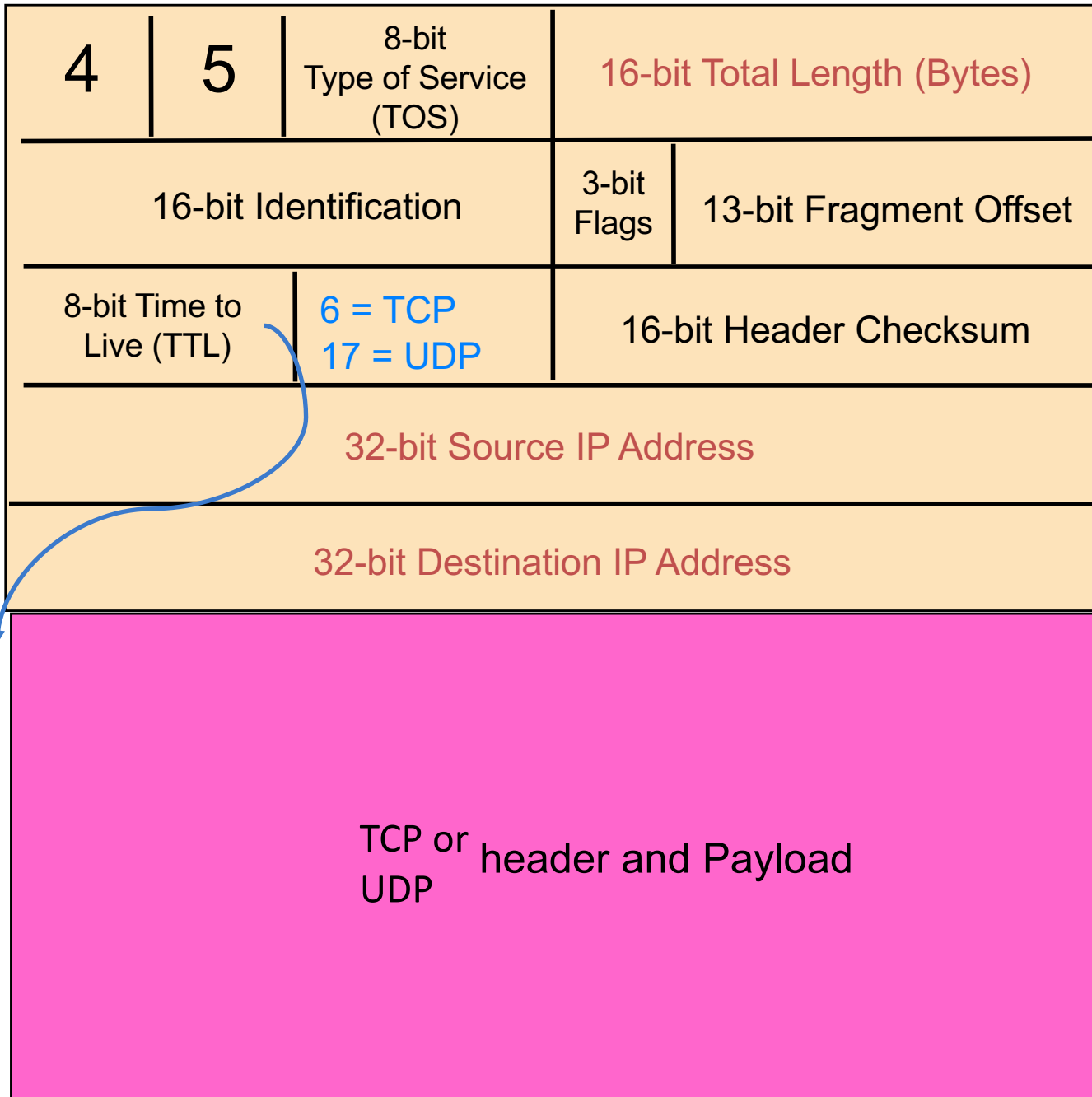
# Ports

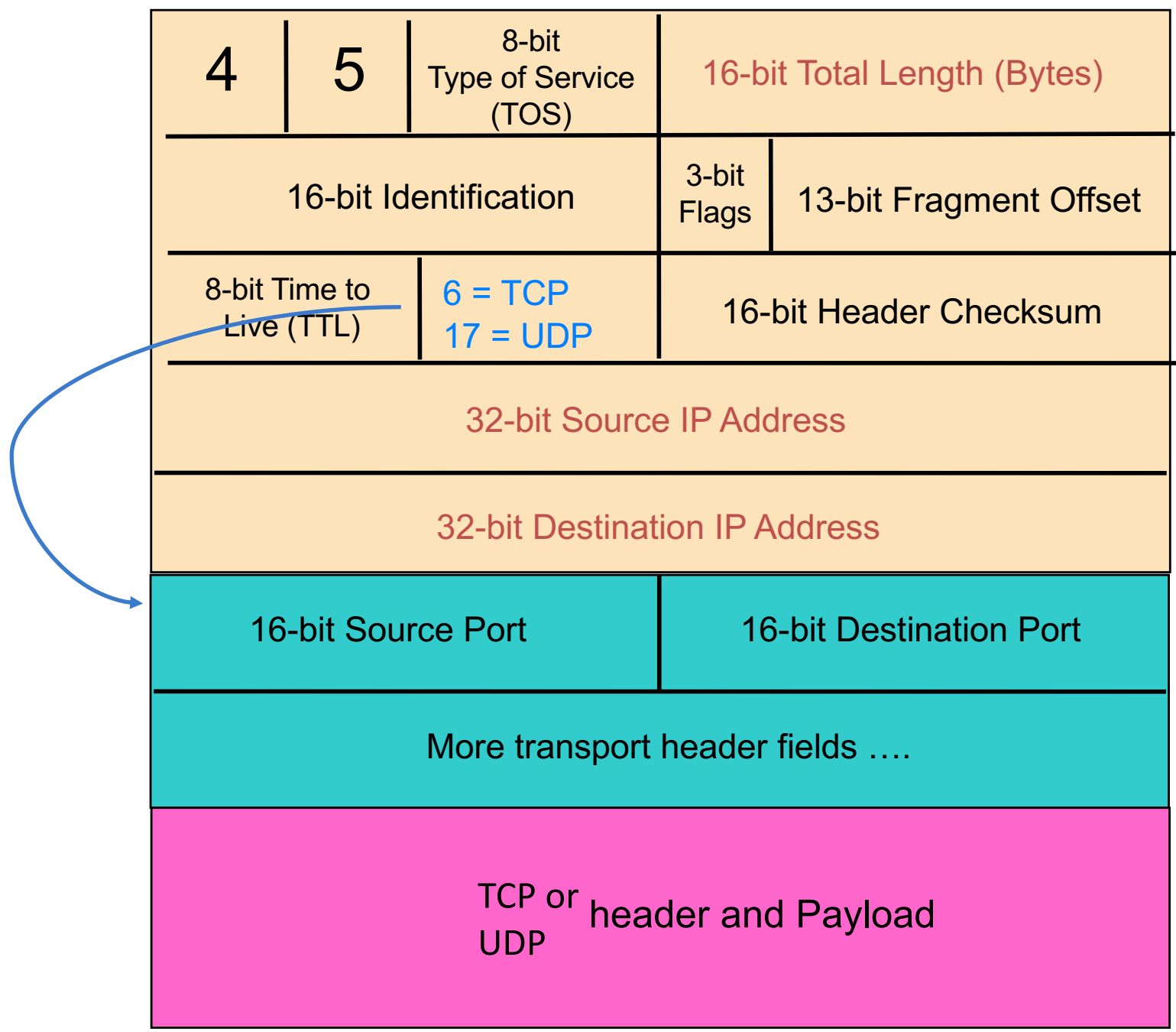
- Problem: deciding which app (socket) gets which packets
- Solution: *port* as a transport layer identifier
  - 16 bit identifier
    - OS stores mapping between sockets and *ports*
    - a packet carries a source and destination port number in its transport layer header
- For UDP ports (SOCK\_DGRAM)
  - OS stores (local port, local IP address)  $\leftrightarrow$  socket
- For TCP ports (SOCK\_STREAM)
  - OS stores (local port, local IP, remote port, remote IP)  $\leftrightarrow$  socket











# Recap: Multiplexing and Demultiplexing

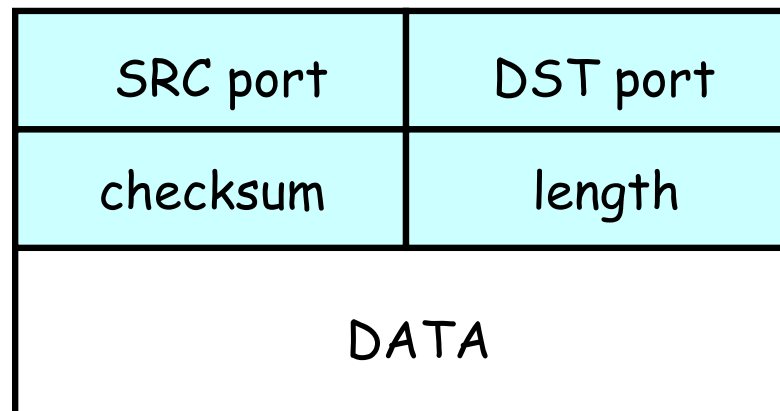
- Host receives IP packets
  - Each IP header has source and destination **IP address**
  - Each Transport Layer header has source and destination **port** number
- Host uses IP addresses and port numbers to direct the message to appropriate **socket**

# More on Ports

- Separate 16-bit port address space for UDP and TCP
- “Well known” ports (0-1023): everyone agrees which services run on these ports
  - e.g., ssh:22, http:80
  - helps client know server’s port
- Ephemeral ports (most 1024-65535): dynamically selected: as the source port for a client process

# UDP: User Datagram Protocol

- Lightweight communication between processes
  - Avoid overhead and delays of ordered, reliable delivery
- UDP described in RFC 768 – (1980!)
  - Destination IP address and port to support demultiplexing
  - Optional error checking on the packet contents
    - (checksum field of 0 means “don’t verify checksum”)
    - ((this idea of optional checksum is removed in IPv6))



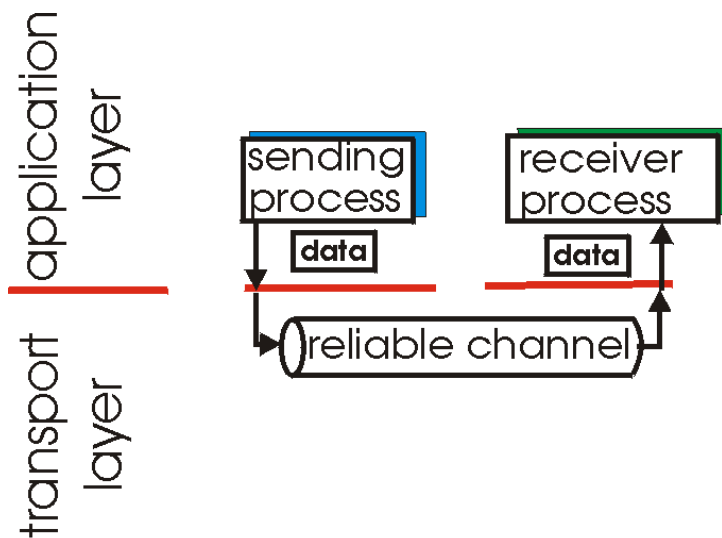
# Why a transport layer?

- IP packets are addressed to a host but end-to-end communication is between application processes at hosts
  - Need a way to decide which packets go to which applications (mux/demux)
- IP provides a weak service model (*best-effort*)
  - Packets can be corrupted, delayed, dropped, reordered, duplicated



# Principles of Reliable data transfer

- important in app., transport, link layers
- top-10 list of important networking topics!



(a) provided service

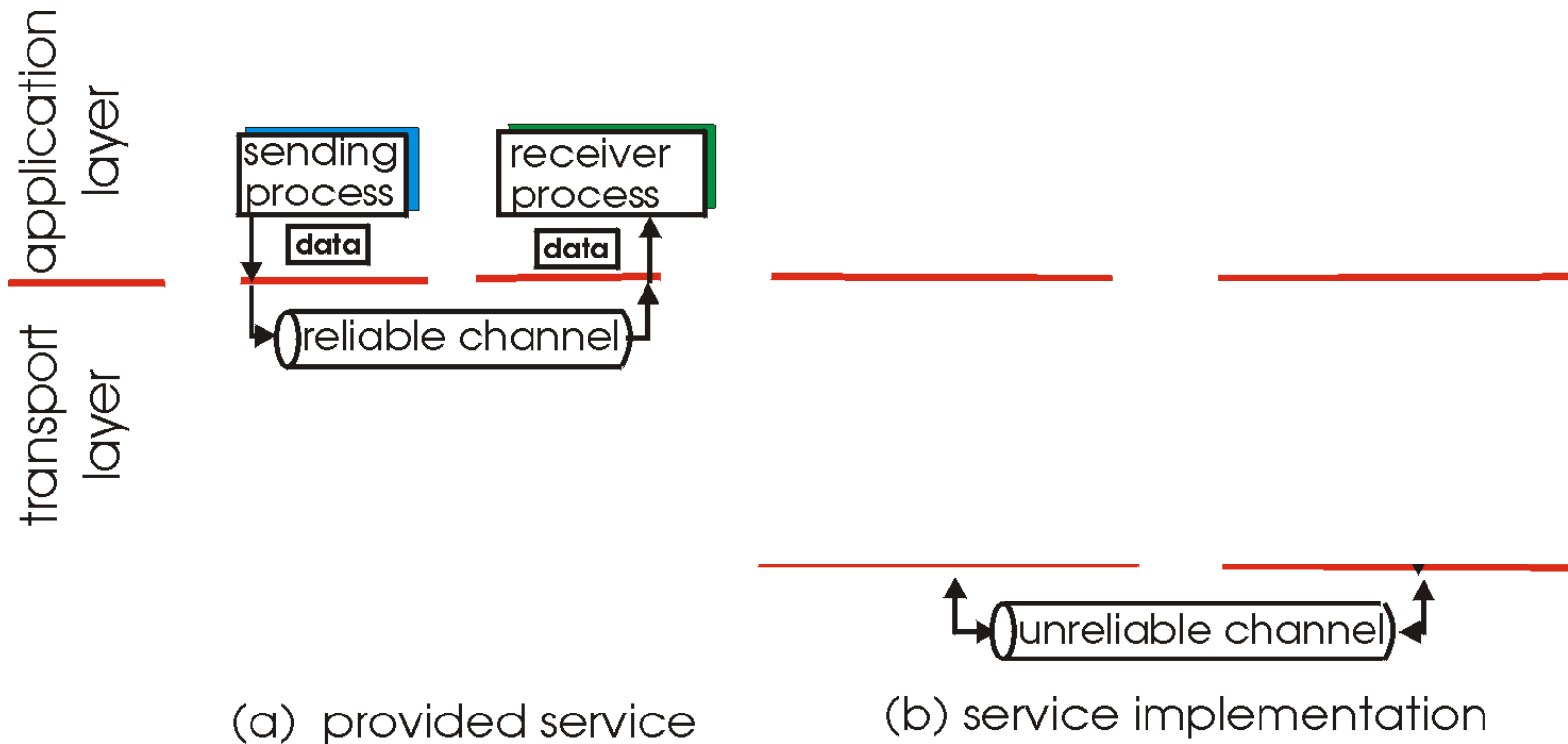
- In a perfect world, reliable transport is easy

But the Internet default is *best-effort*

- All the bad things best-effort can do
  - a packet is corrupted (bit errors)
  - a packet is lost
  - a packet is delayed (*why?*)
  - packets are reordered (*why?*)
  - a packet is duplicated (*why?*)

# Principles of Reliable data transfer

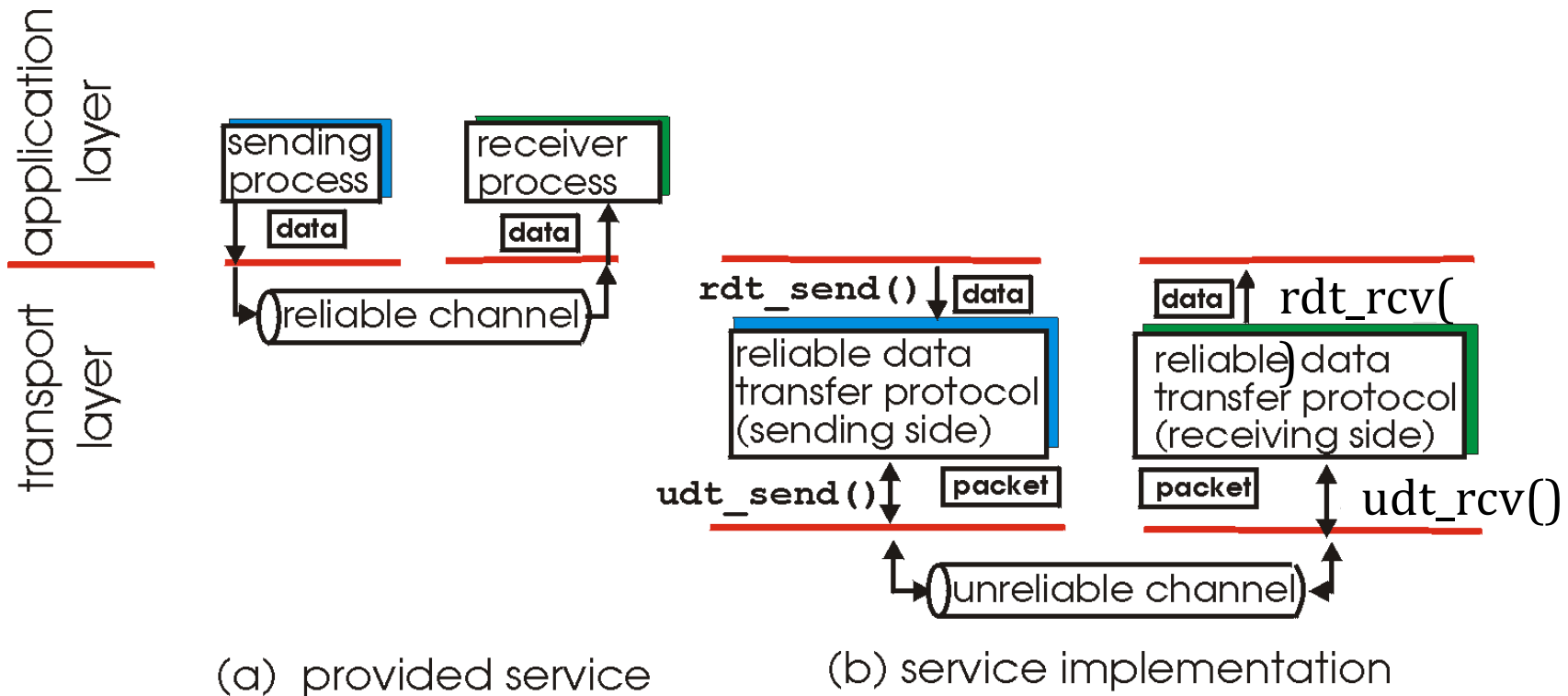
- important in app., transport, link layers
- top-10 list of important networking topics!



- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

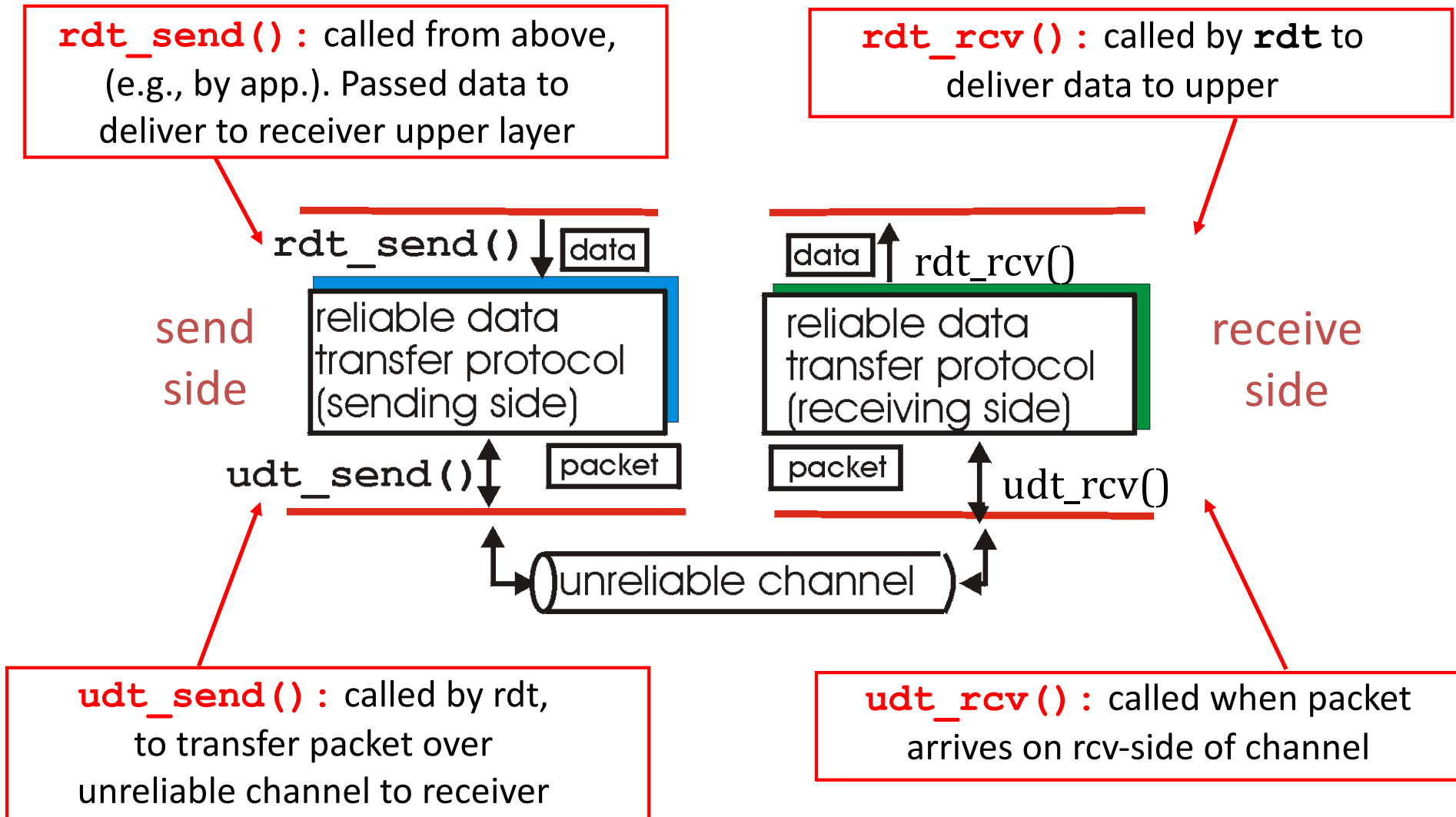
# Principles of Reliable data transfer

- important in app., transport, link layers
- top-10 list of important networking topics!



- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

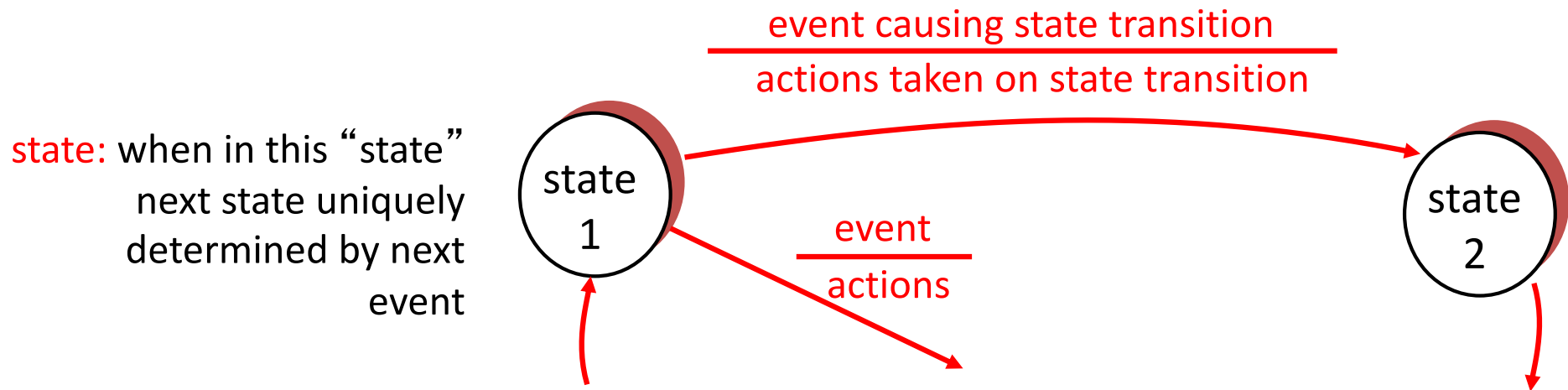
# Reliable data transfer: getting started



# Reliable data transfer: getting started

## We' ll:

- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver



# KR state machines – a note.

## Beware

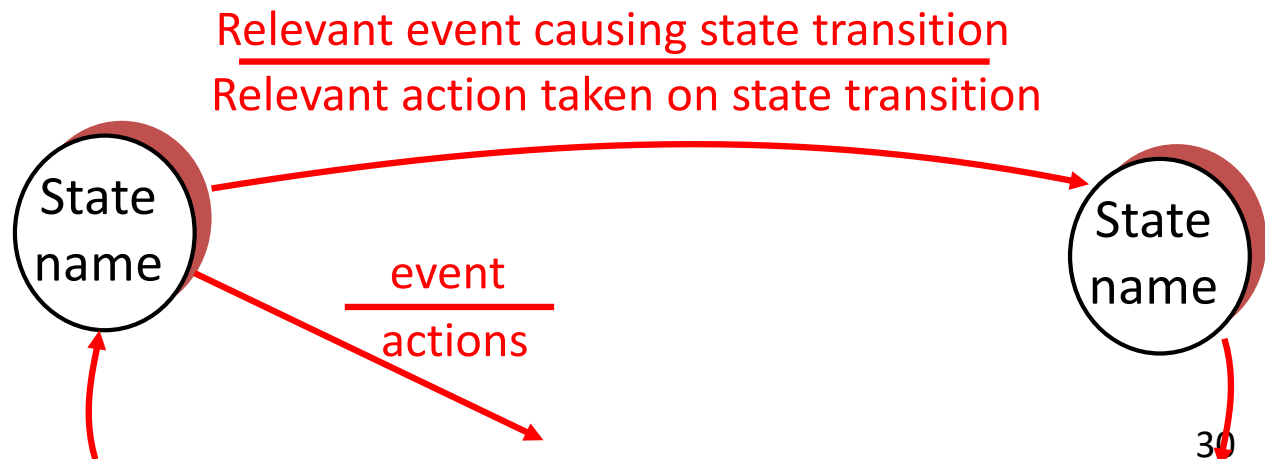
Kurose and Ross has a confusing/confused attitude to state-machines.

I've attempted to normalise the representation.

UPSHOT: these slides have differing information to the KR book (from which the RDT example is taken.)

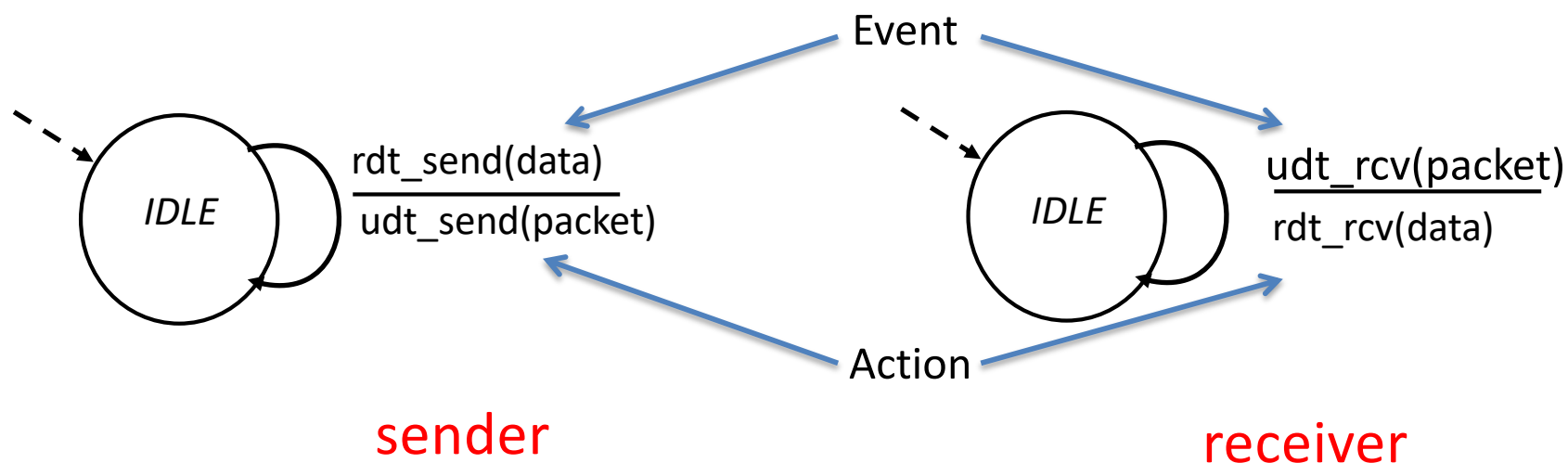
in KR “actions taken” appear wide-ranging, my interpretation is more specific/relevant.

**state:** when in this “state”  
next state uniquely  
determined by next  
event



# Rdt1.0: reliable transfer over a reliable channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver read data from underlying channel

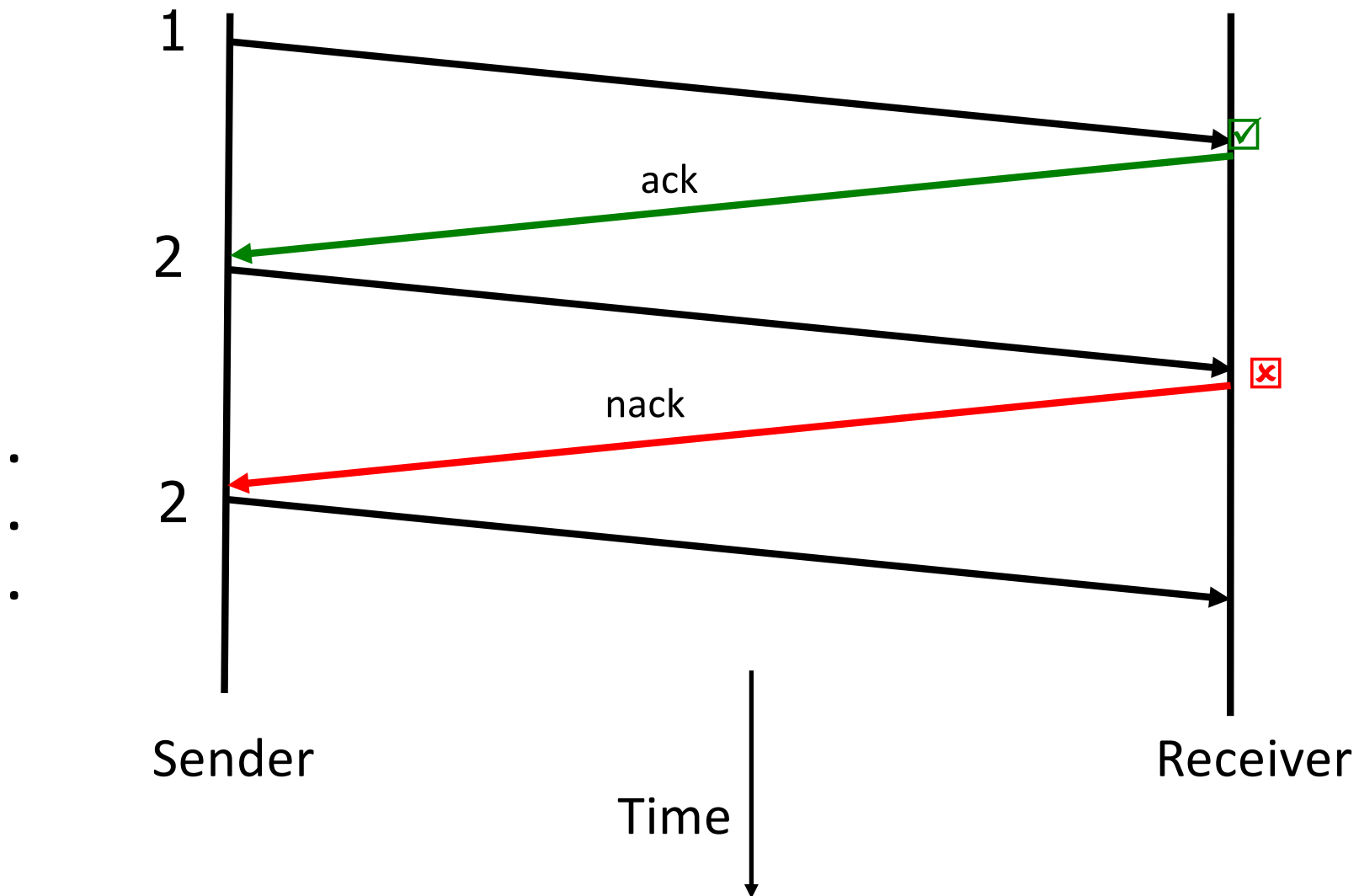


# Rdt2.0: channel with bit errors

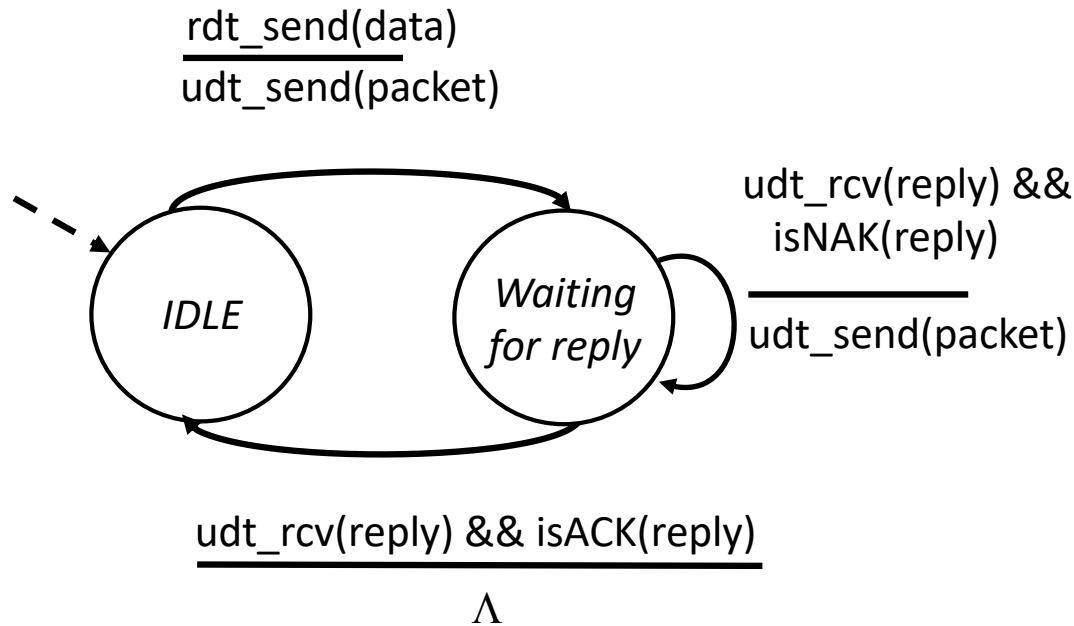
- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:
  - *acknowledgements (ACKs)*: receiver explicitly tells sender that packet received is OK
  - *negative acknowledgements (NAKs)*: receiver explicitly tells sender that packet had errors
    - sender retransmits packet on receipt of NAK
- new mechanisms in **rdt2.0** (beyond **rdt1.0**):
  - error detection
  - receiver feedback: control msgs (ACK,NAK) receiver->sender



# Dealing with Packet Corruption



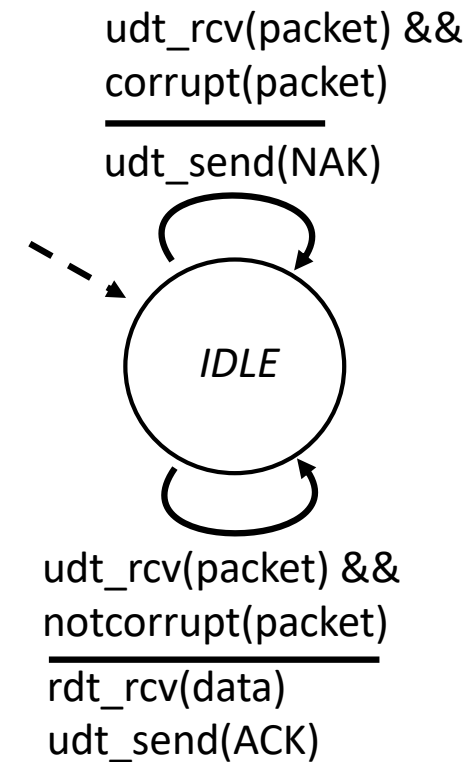
# rdt2.0: FSM specification



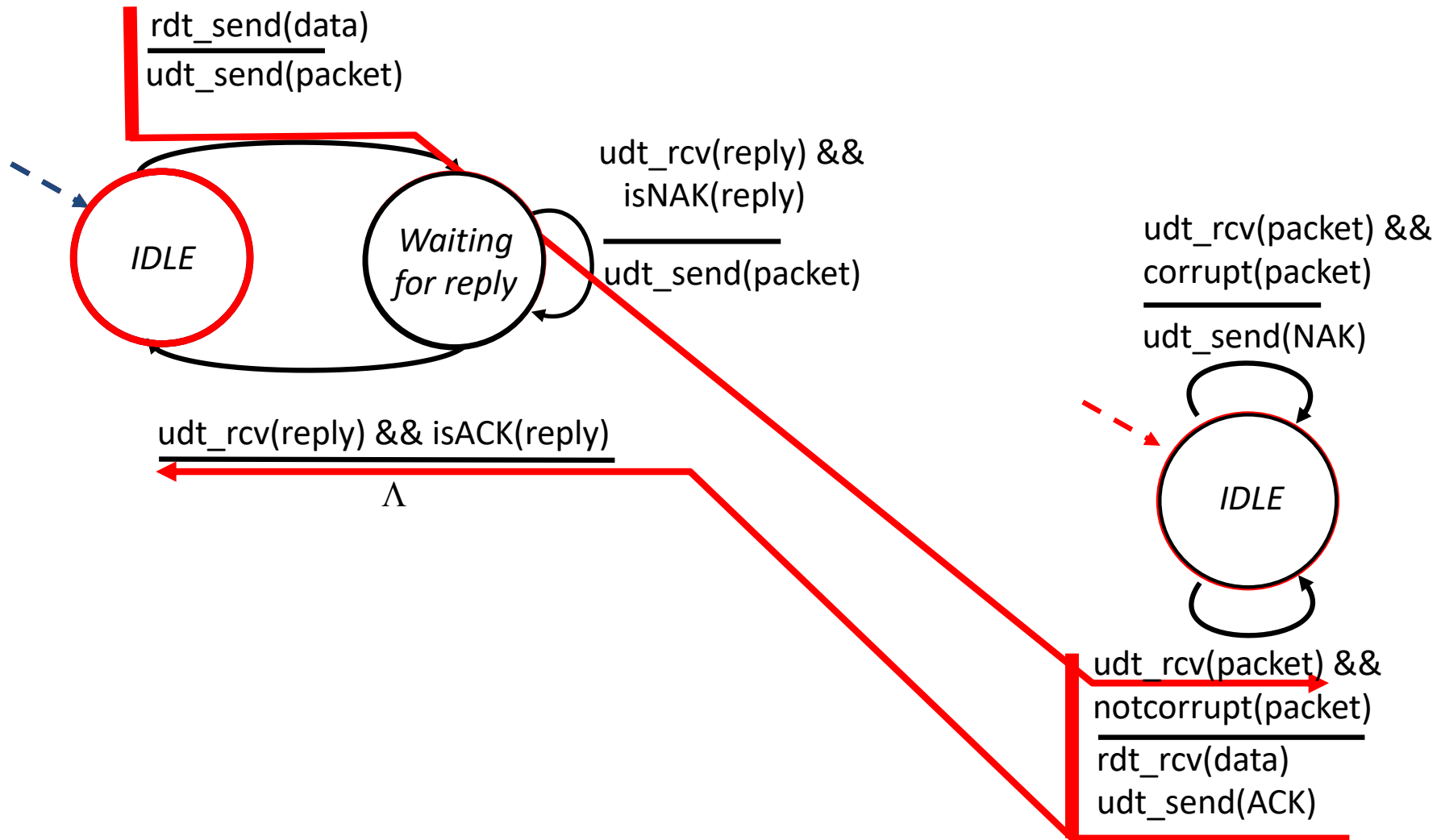
sender

**Note:** the sender holds a copy of the packet being sent until the delivery is acknowledged.

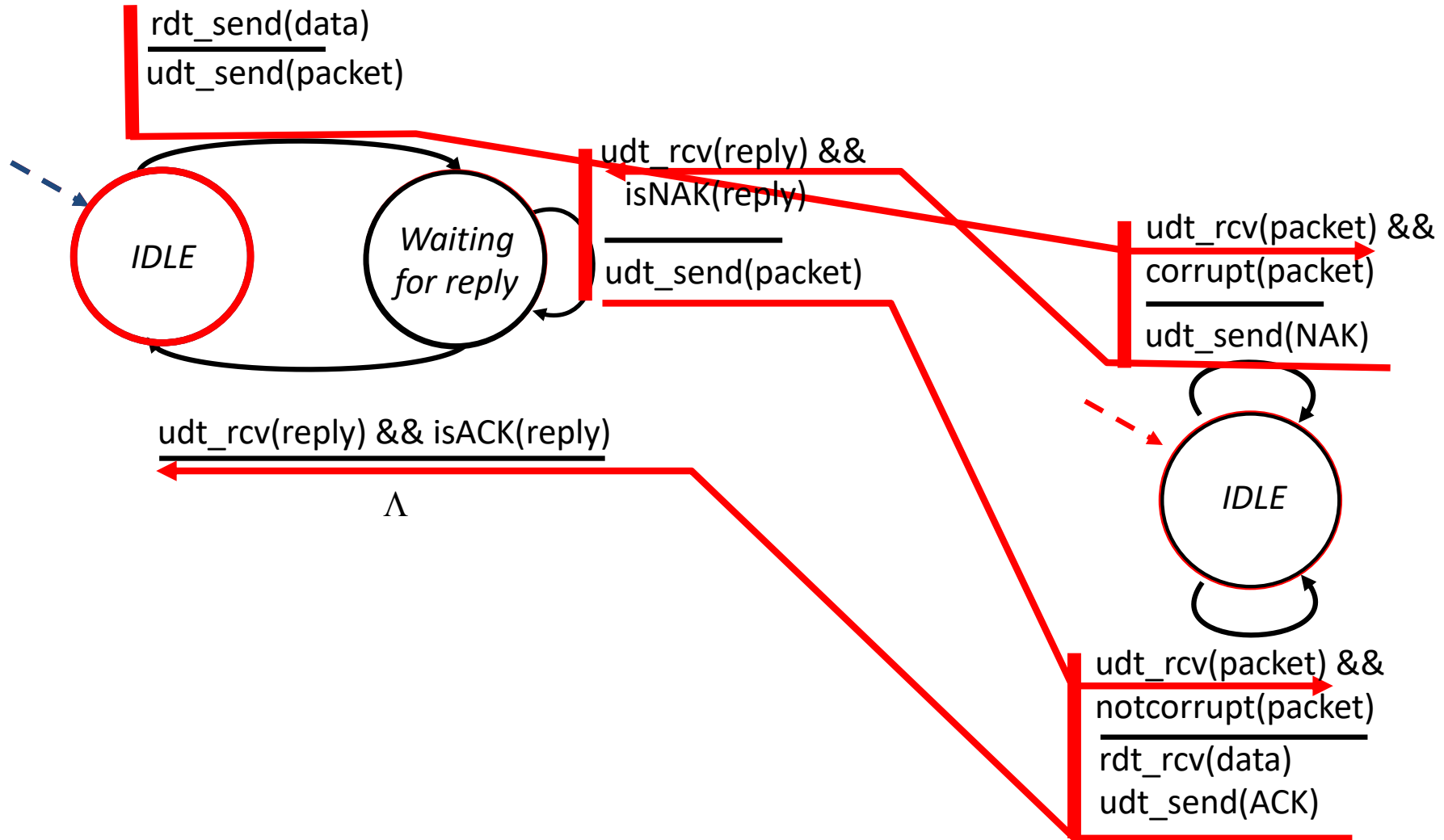
receiver



# rdt2.0: operation with no errors



# rdt2.0: error scenario



# rdt2.0 has a fatal flaw!

## What happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!
- can't just retransmit: possible duplicate

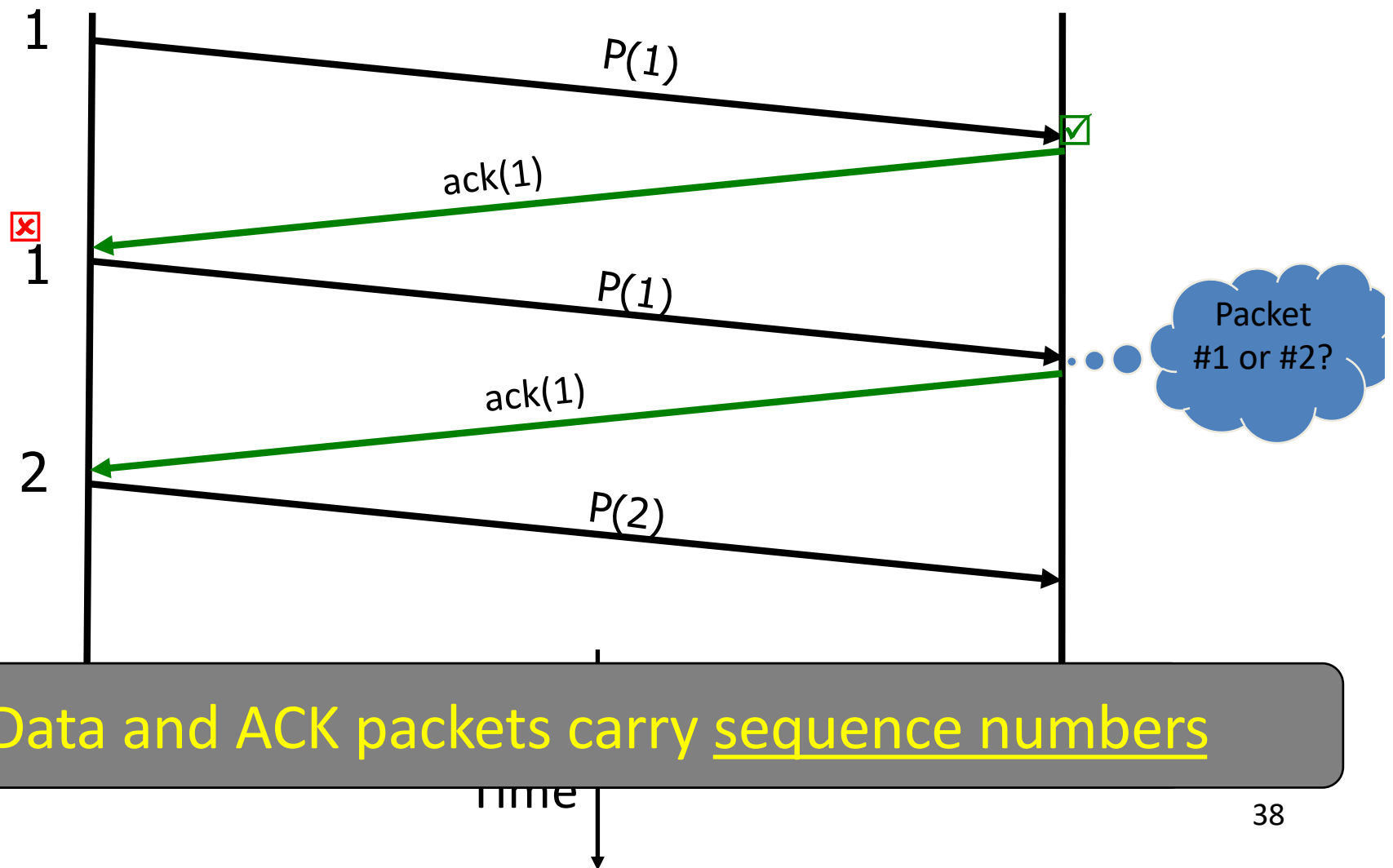
## Handling duplicates:

- sender retransmits current packet if ACK/NAK garbled
- sender adds *sequence number* to each packet
- receiver discards (doesn't deliver) duplicate packet

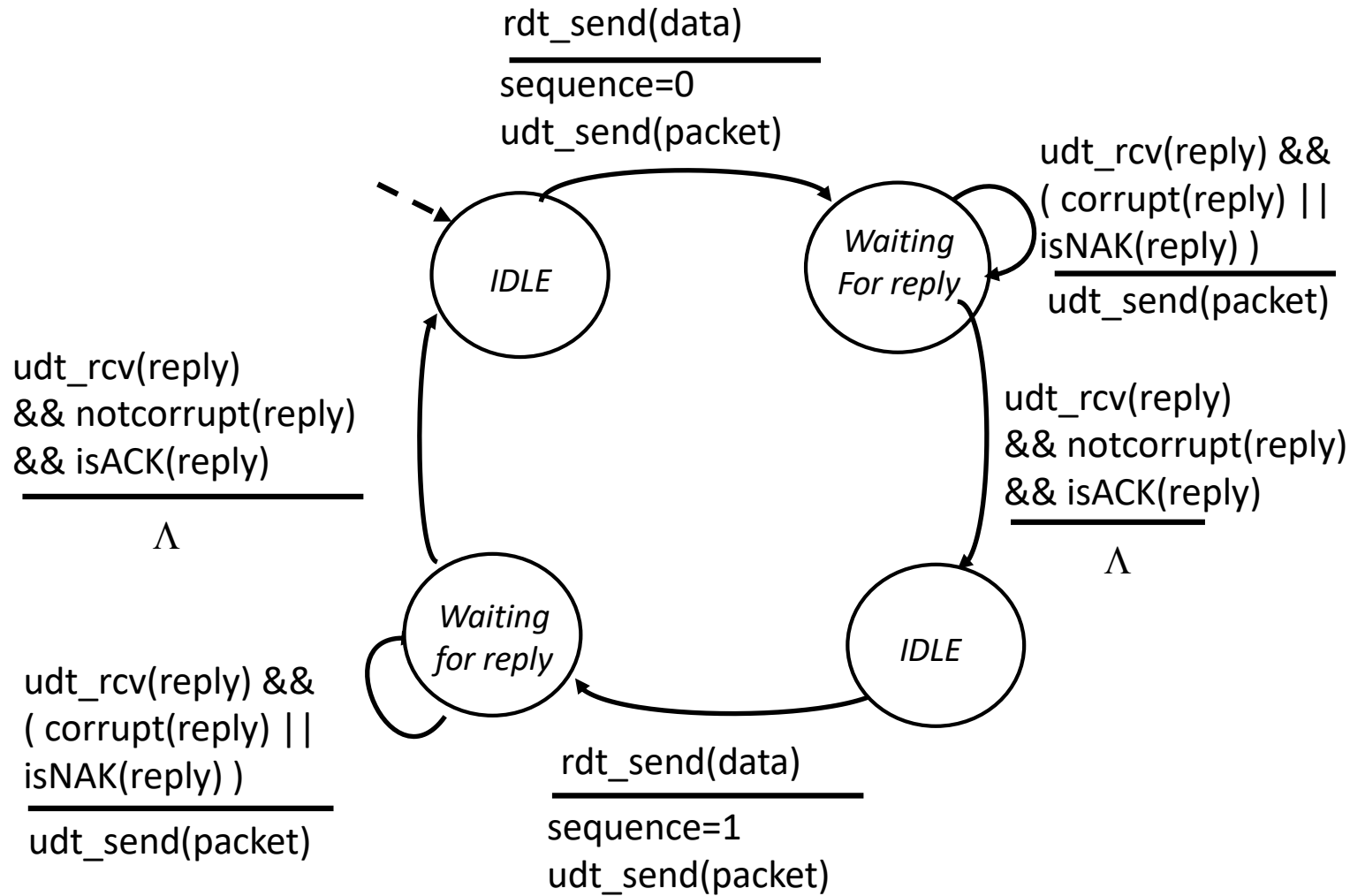
### stop and wait

Sender sends one packet, then waits for receiver response

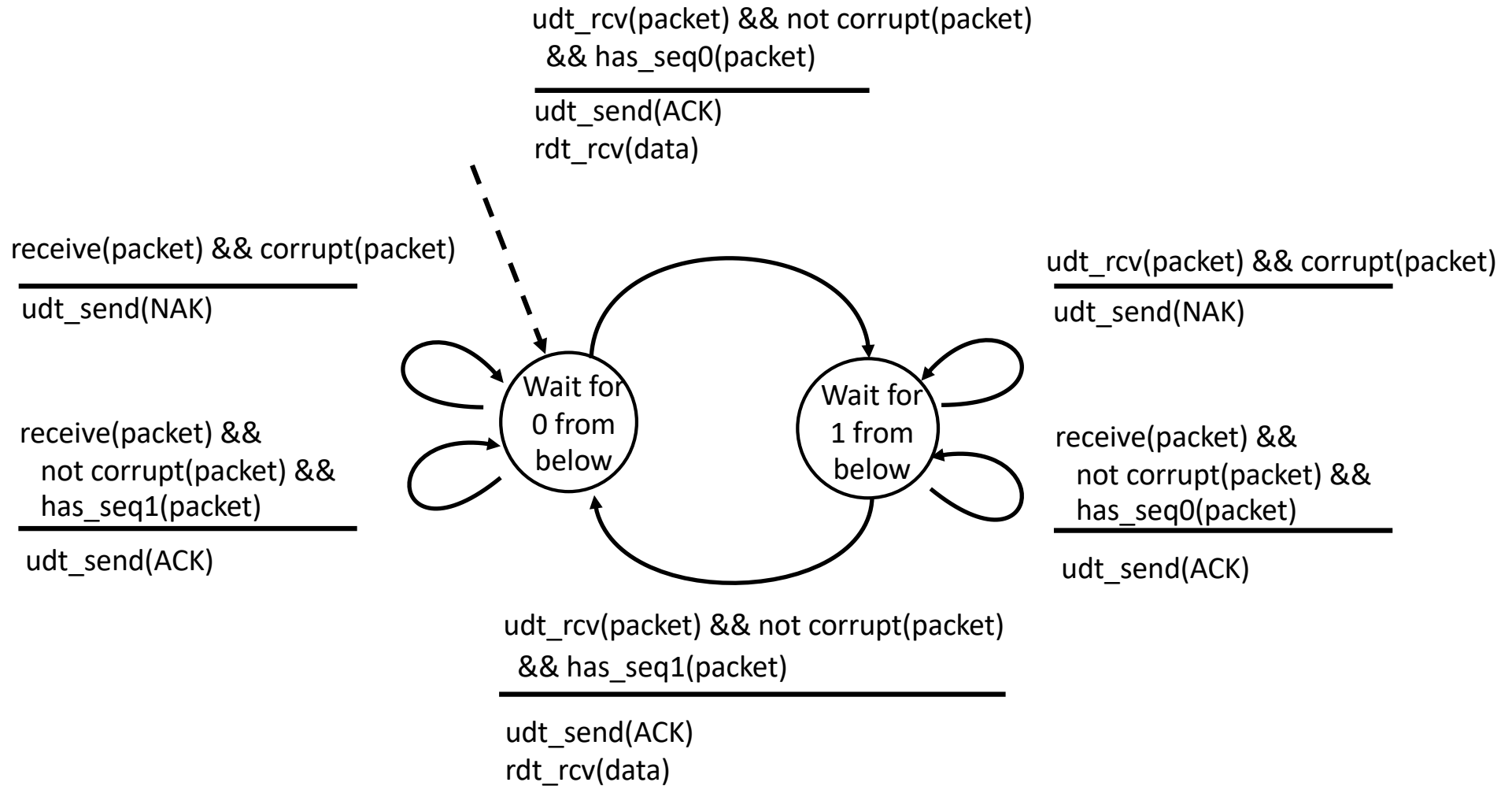
# Dealing with Packet Corruption



# rdt2.1: sender, handles garbled ACK/NAKs



# rdt2.1: receiver, handles garbled ACK/NAKs





# rdt2.1: discussion

## Sender:

- seq # added to pkt
- two seq. #'s (0,1) will suffice. Why?
- must check if received ACK/NAK corrupted
- twice as many states
  - state must “remember” whether “current” pkt has a 0 or 1 sequence number

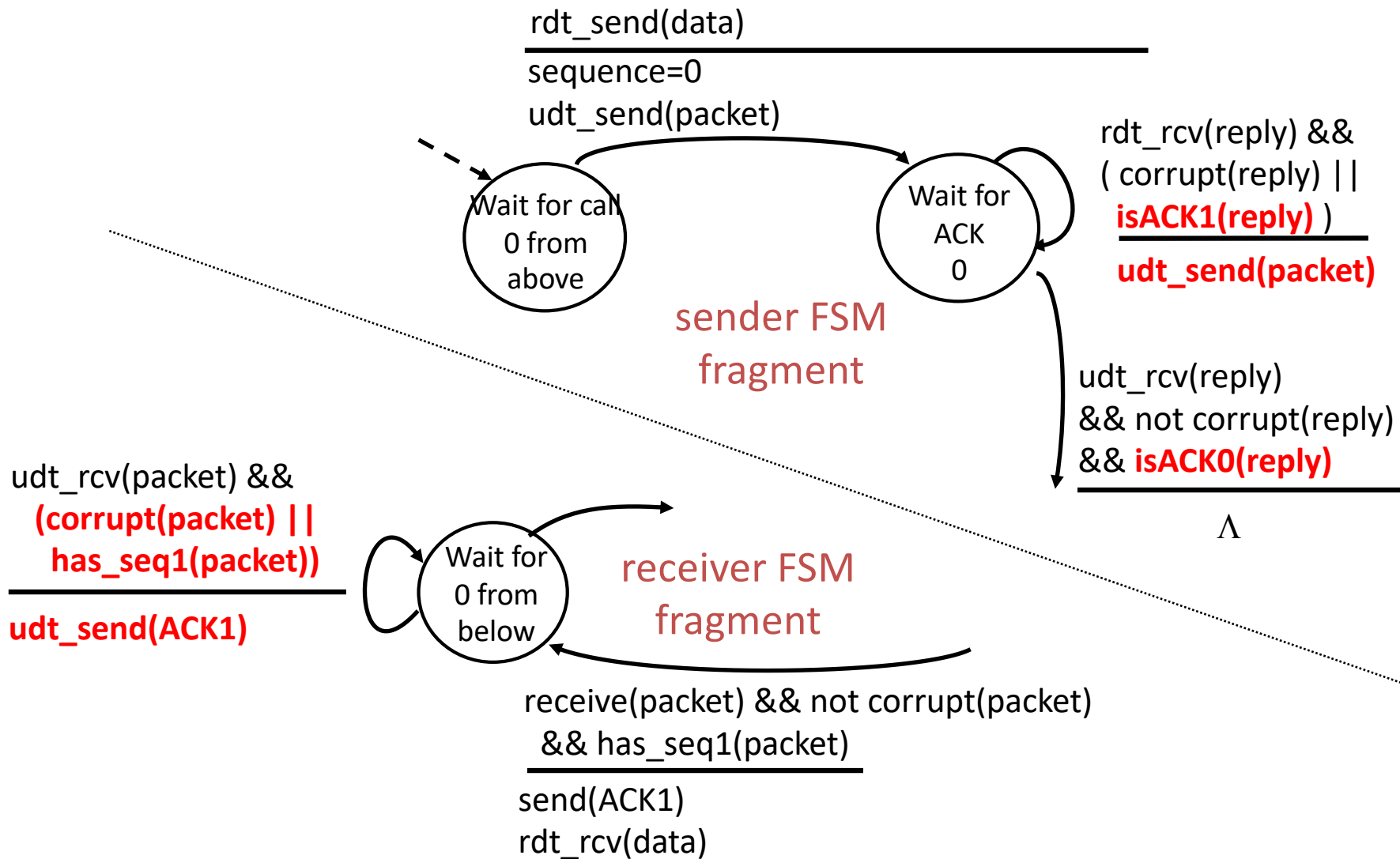
## Receiver:

- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
- note: receiver can *not* know if its last ACK/NAK received OK at sender

## rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must *explicitly* include seq # of pkt being ACKed
- duplicate ACK at sender results in same action as NAK:  
*retransmit current pkt*

# rdt2.2: sender, receiver fragments



# rdt3.0: channels with errors *and* loss

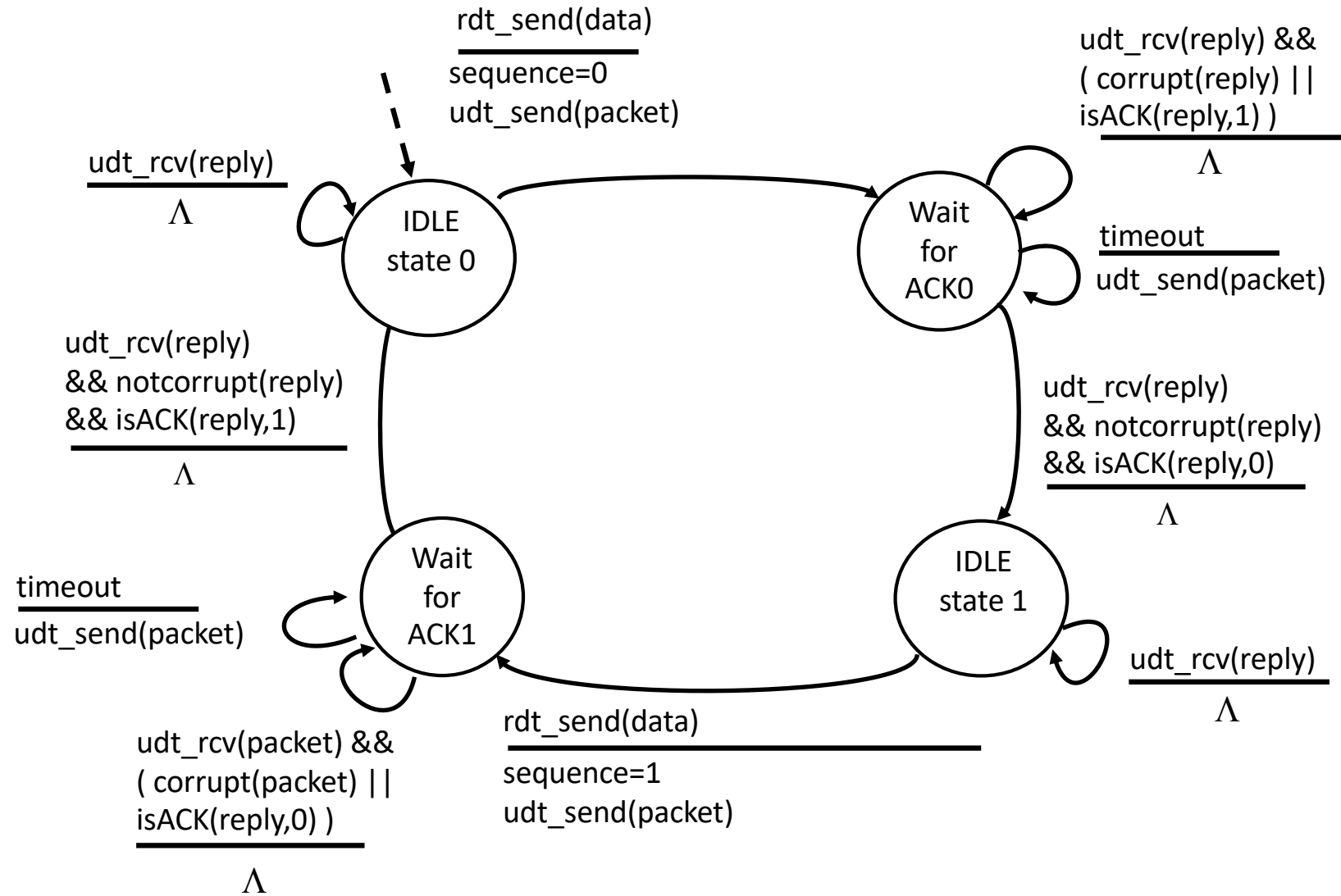
New assumption: underlying channel can also lose packets (data or ACKs)

- checksum, seq. #, ACKs, retransmissions will be of help, but not enough

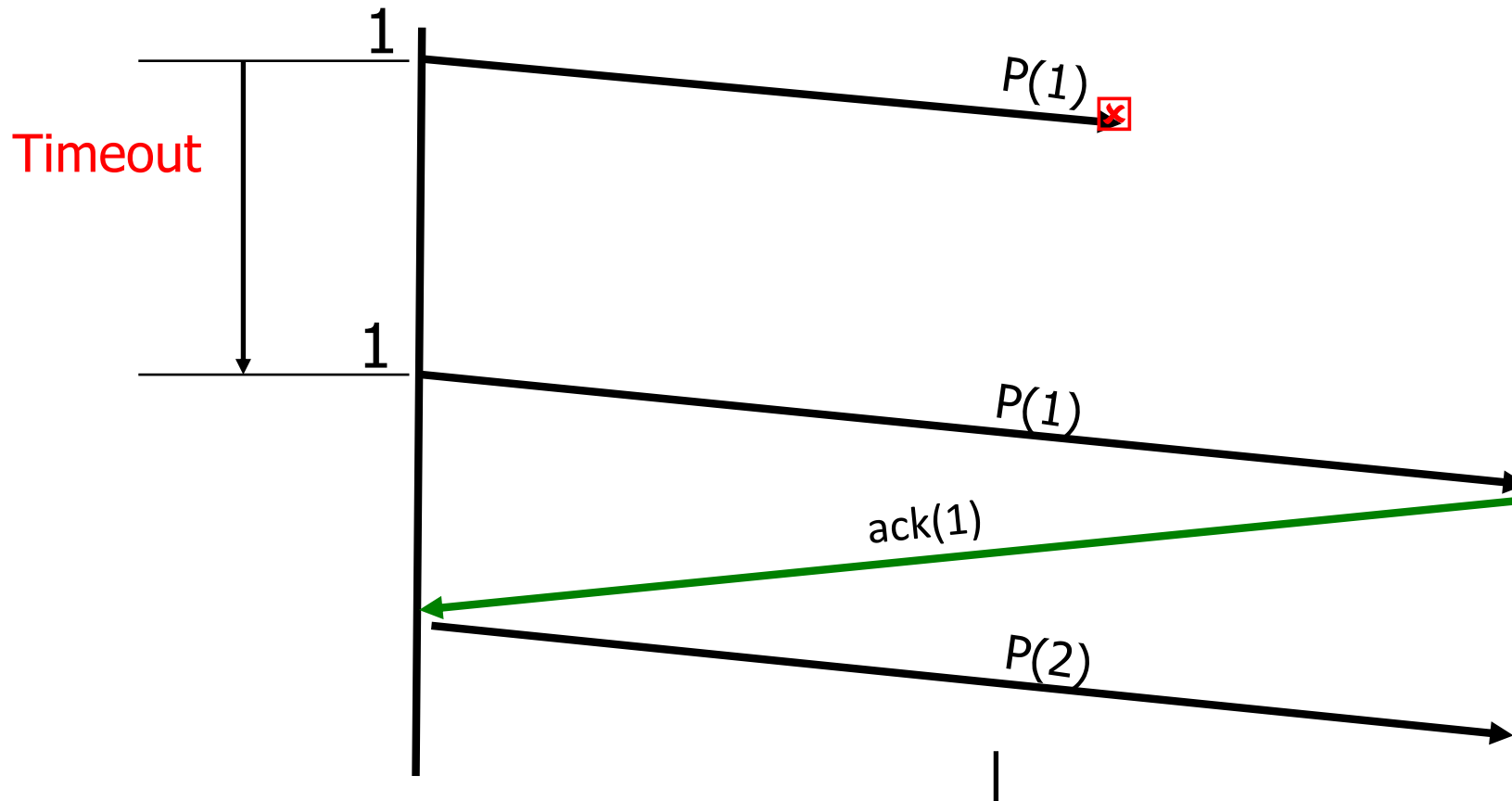
Approach: sender waits “reasonable” amount of time for ACK

- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but use of seq. #'s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer

# rdt3.0 sender



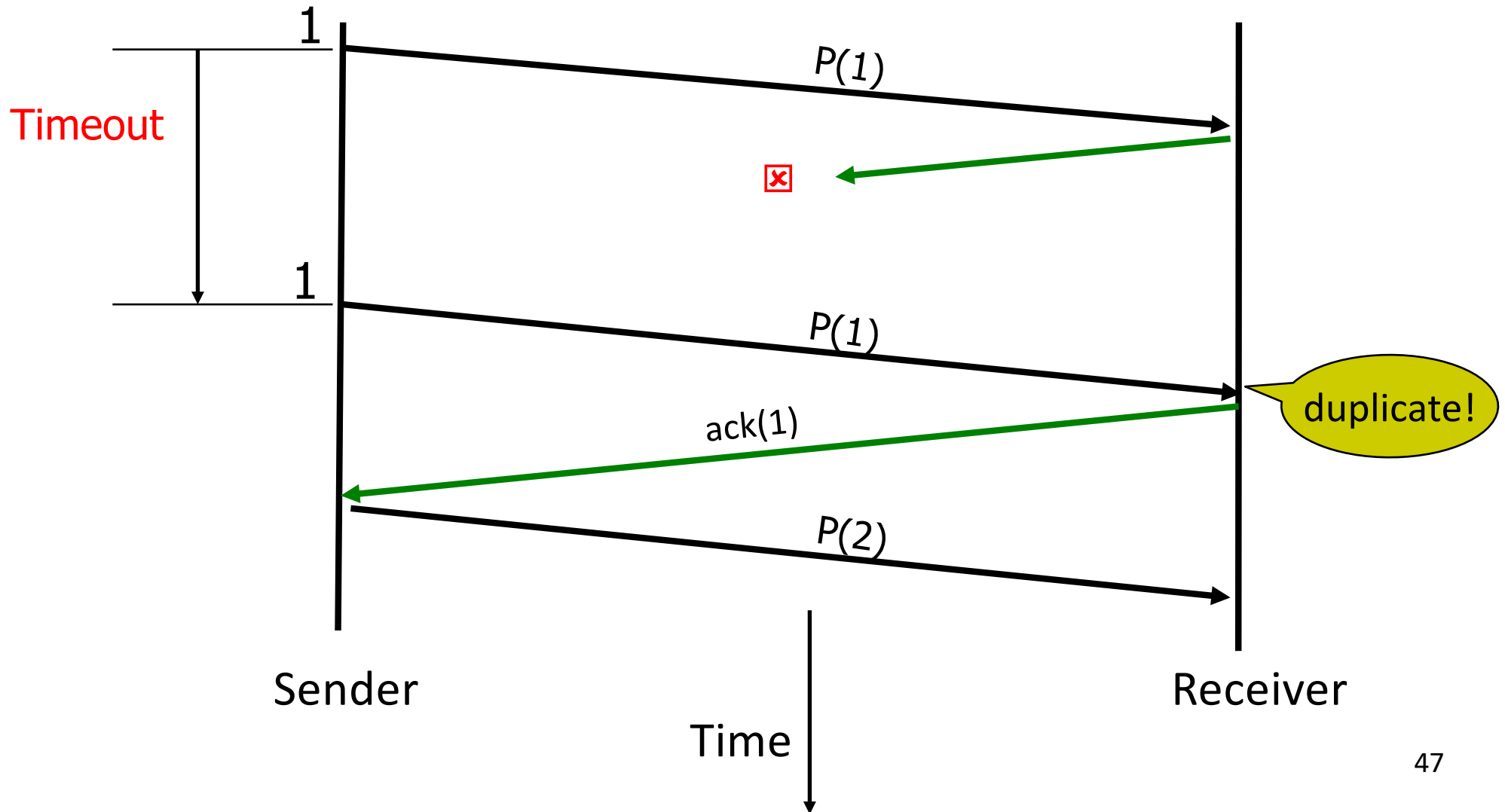
# Dealing with Packet Loss



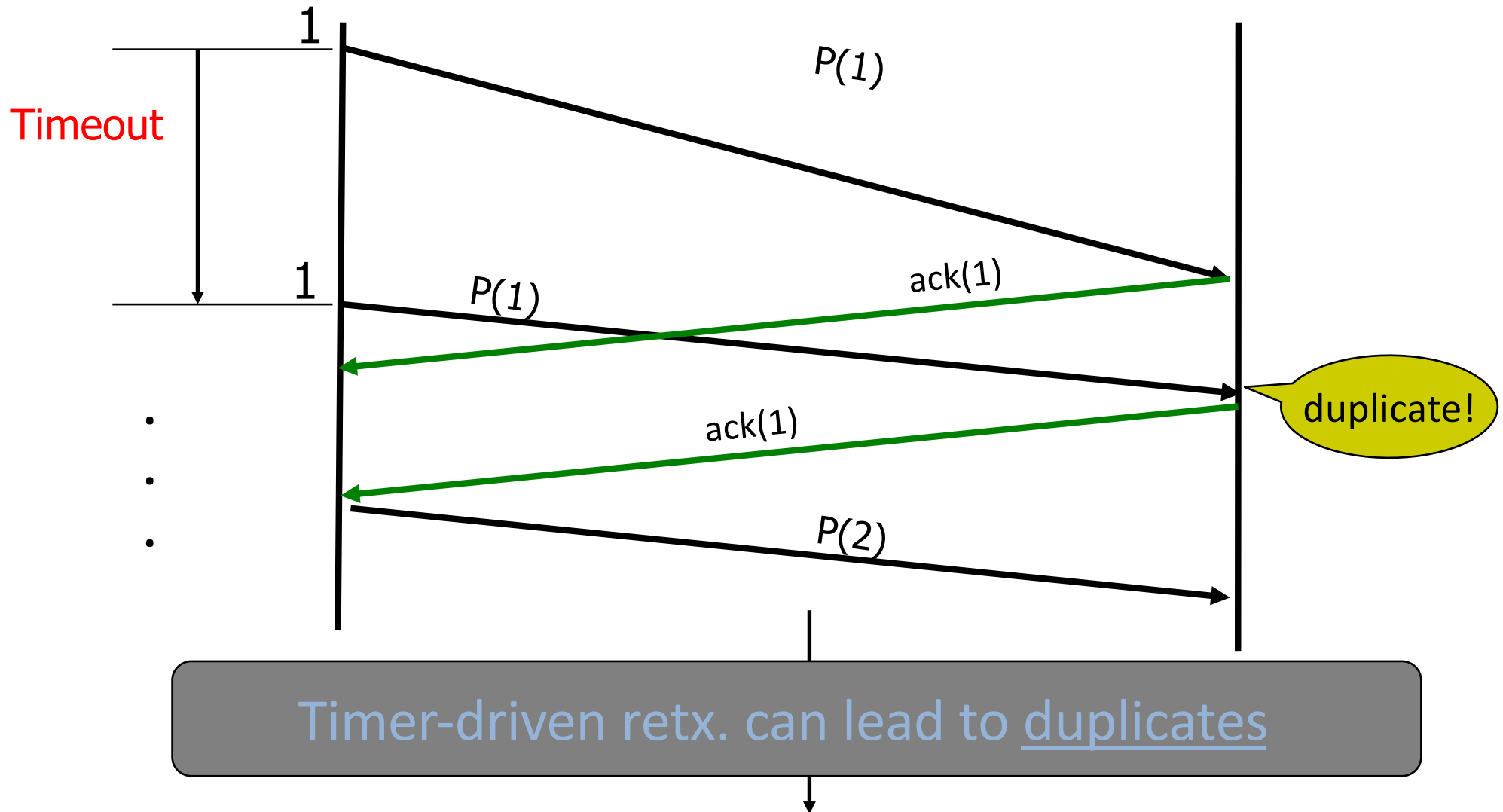
Timer-driven loss detection

Set timer when packet is sent; retransmit on timeout

# Dealing with Packet Loss



# Dealing with Packet Loss





# Performance of rdt3.0

- rdt3.0 works, but performance stinks
- ex: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

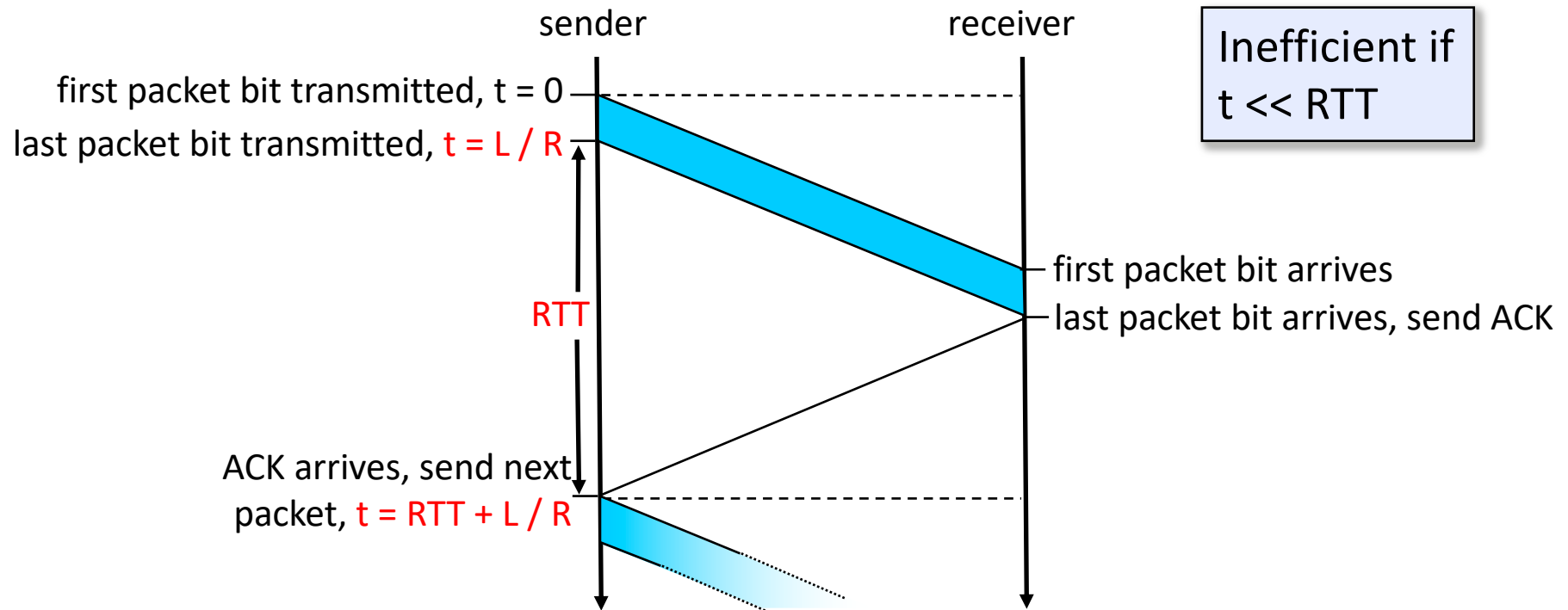
$$d_{trans} = \frac{L}{R} = \frac{8000\text{bits}}{10^9\text{bps}} = 8\text{microseconds}$$

m  $U_{\text{sender}}$ : **utilization** – fraction of time sender busy sending

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

- m 1KB pkt every 30 msec -> 33kB/sec throughput over 1 Gbps link
- m network protocol limits use of physical resources!

# rdt3.0: stop-and-wait operation

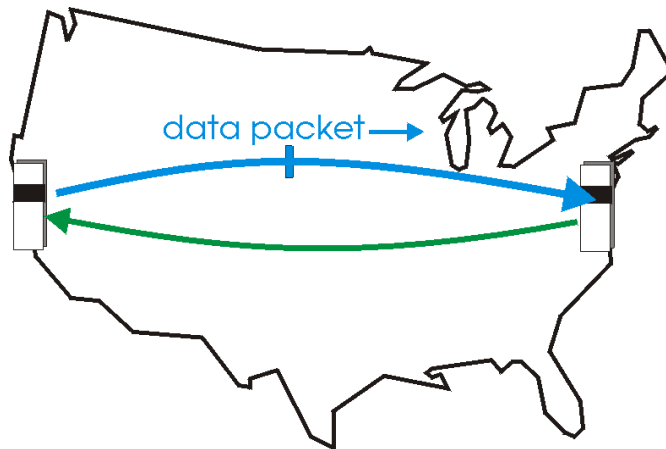


$$U_{\text{sender}} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

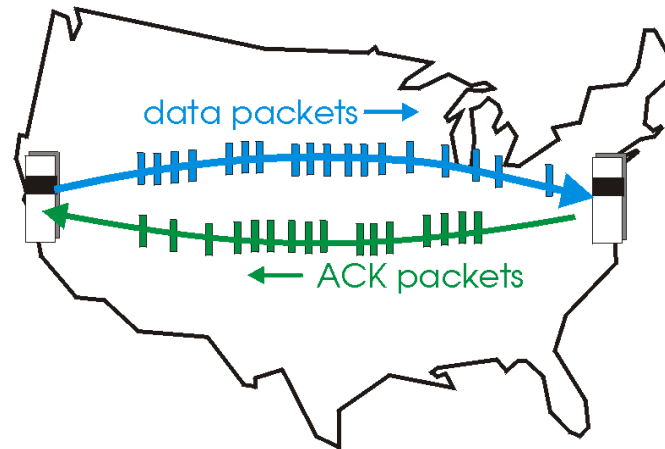
# Pipelined (Packet-Window) protocols

**Pipelining:** sender allows multiple, “in-flight”, yet-to-be-acknowledged pkts

- range of sequence numbers must be increased
- buffering at sender and/or receiver



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



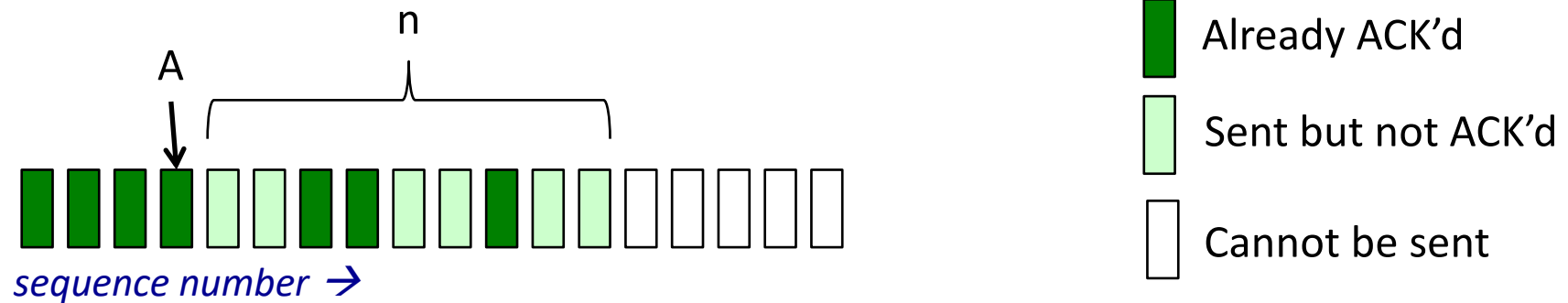
(b) a pipelined protocol in operation

# A Sliding Packet Window

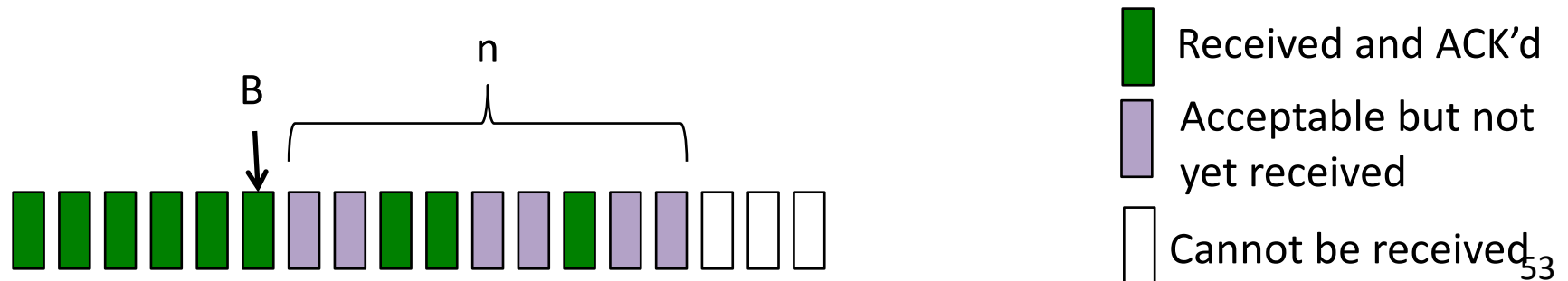
- **window** = set of adjacent sequence numbers
  - The size of the set is the **window size**; assume window size is  $n$
- General idea: send up to  $n$  packets at a time
  - Sender can send packets in its window
  - Receiver can accept packets in its window
  - Window of acceptable packets “slides” on successful reception/acknowledgement

# A Sliding Packet Window

- Let A be the **last ack'd packet of sender without gap**;  
then window of sender =  $\{A+1, A+2, \dots, A+n\}$



- Let B be the **last received packet without gap** by receiver,  
then window of receiver =  $\{B+1, \dots, B+n\}$

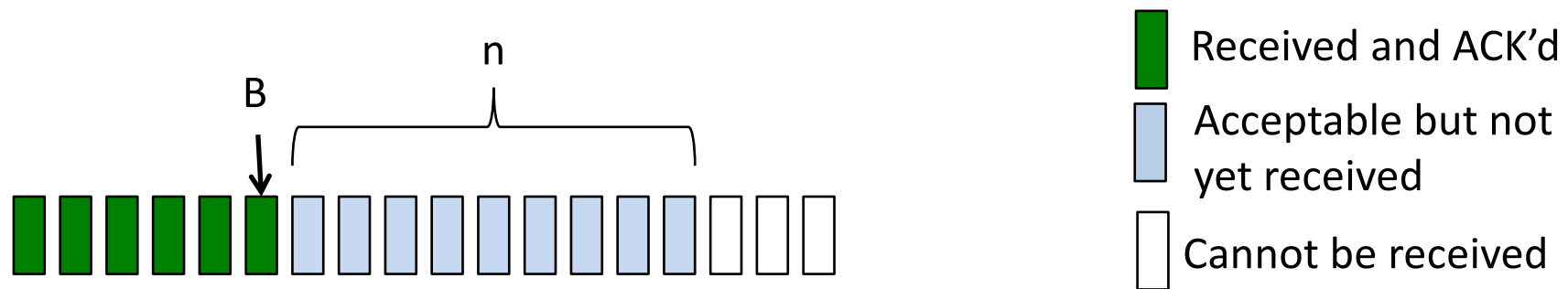


# Acknowledgements w/ Sliding Window

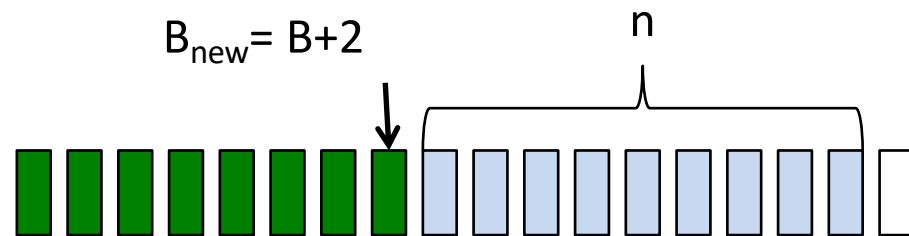
- Two common options
  - cumulative ACKs: ACK carries next in-order sequence number that the receiver expects

# Cumulative Acknowledgements (1)

- At receiver



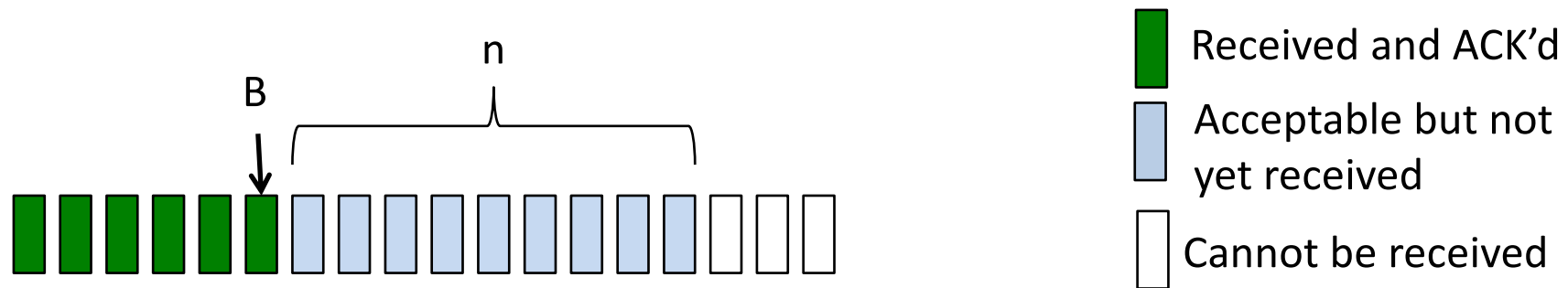
- After receiving  $B+1, B+2$



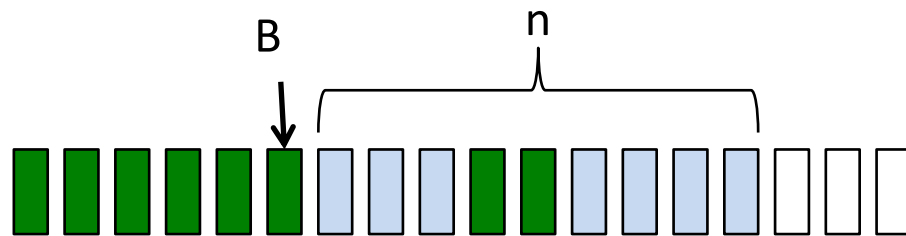
- Receiver sends  $\text{ACK}(B_{\text{new}}+1)$

# Cumulative Acknowledgements (2)

- At receiver



- After receiving B+4, B+5



**How do we recover?**

- Receiver sends **ACK(B+1)**

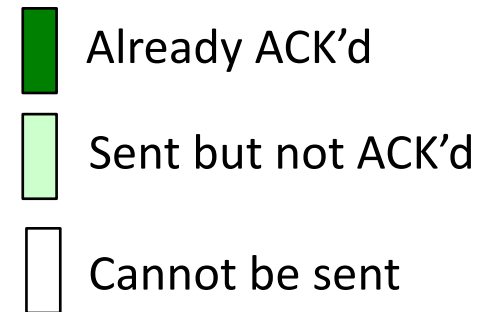
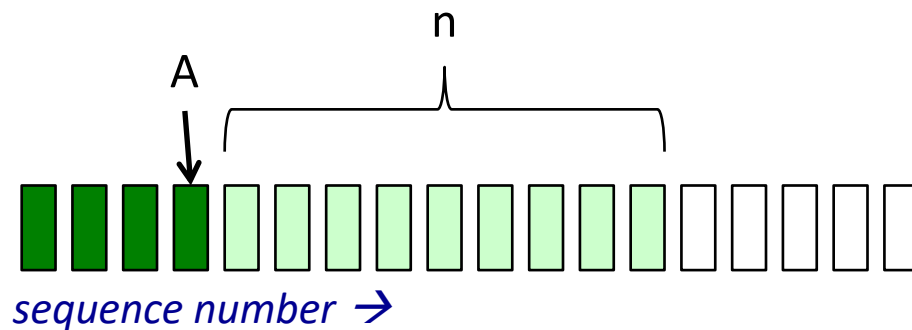


# Go-Back-N (GBN)

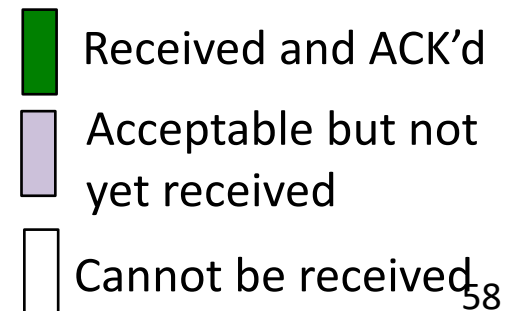
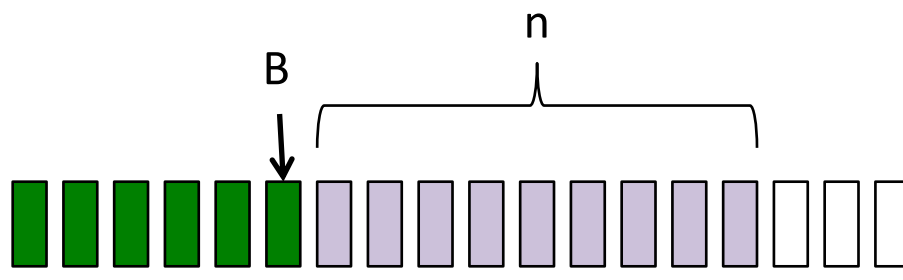
- Sender transmits up to  $n$  unacknowledged packets
- Receiver only accepts packets in order
  - discards out-of-order packets (i.e., packets other than  $B+1$ )
- Receiver uses **cumulative acknowledgements**
  - i.e., sequence# in ACK = next expected in-order sequence#
- Sender sets timer for 1<sup>st</sup> outstanding ack ( $A+1$ )
- If timeout, retransmit  $A+1, \dots, A+n$

# Sliding Window with GBN

- Let A be the **last ack'd packet of sender without gap**;  
then window of sender =  $\{A+1, A+2, \dots, A+n\}$



- Let B be the **last received packet without gap** by receiver,  
then window of receiver =  $\{B+1, \dots, B+n\}$

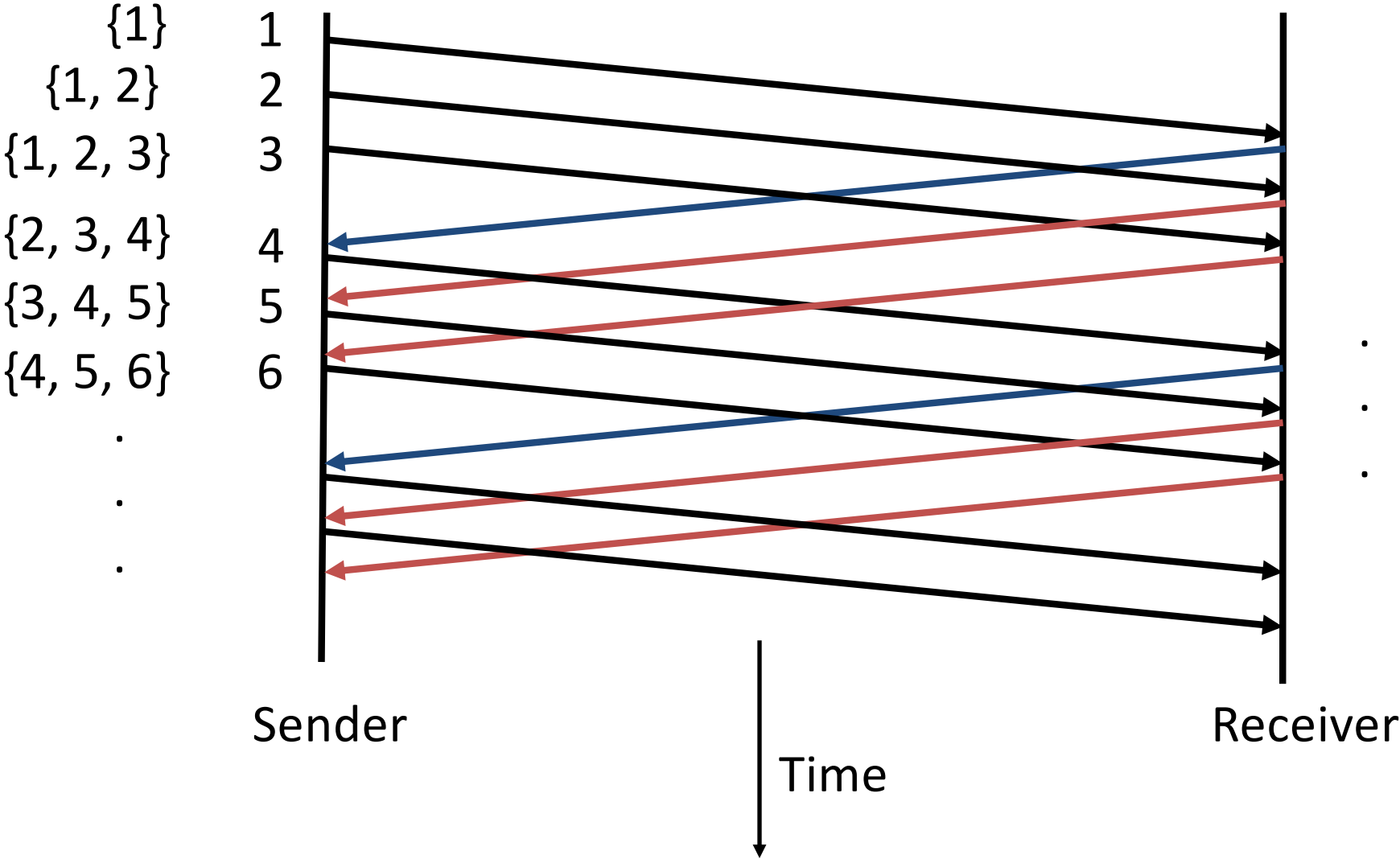


# GBN Example w/o Errors

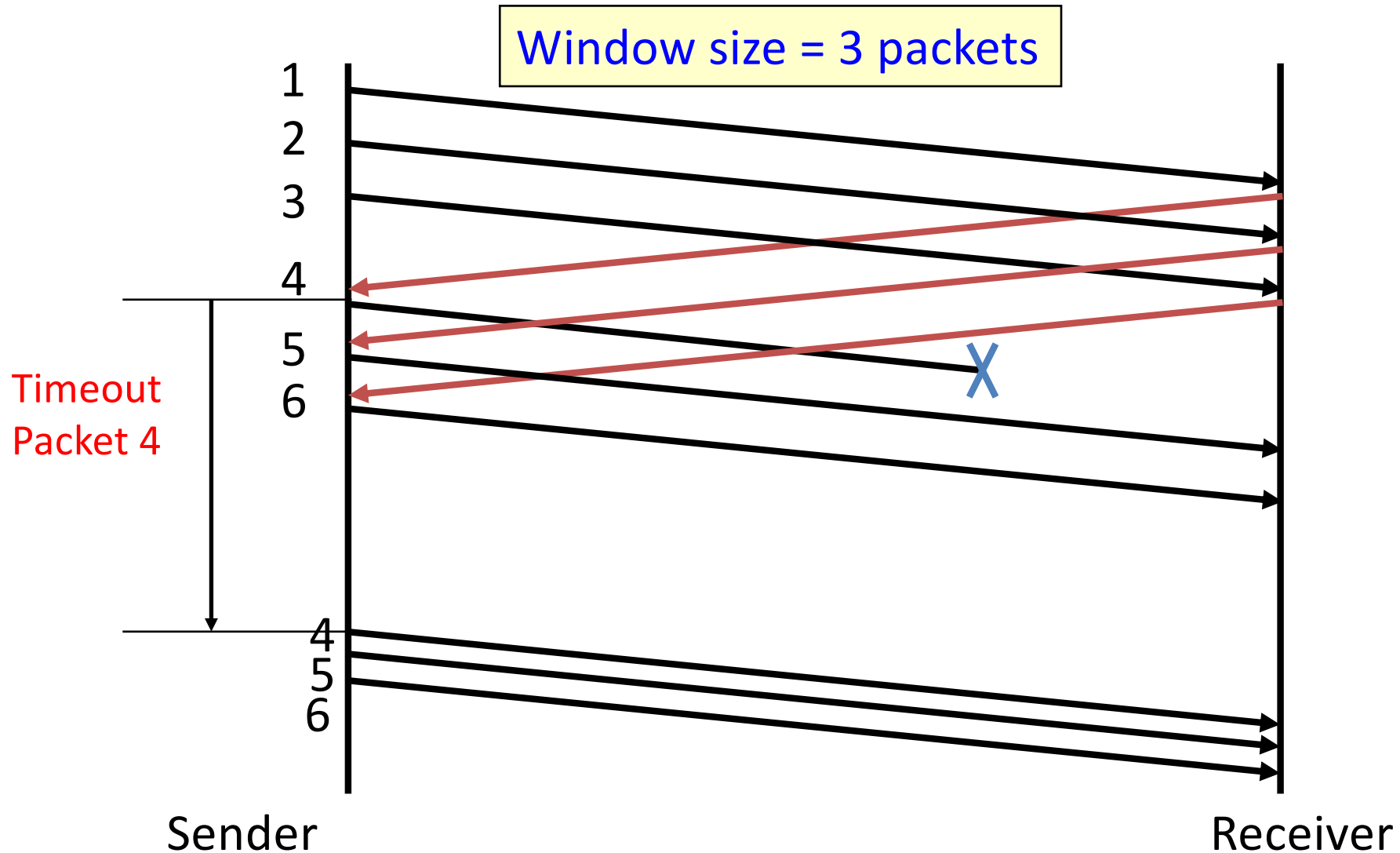
Sender Window

Window size = 3 packets

Receiver Window



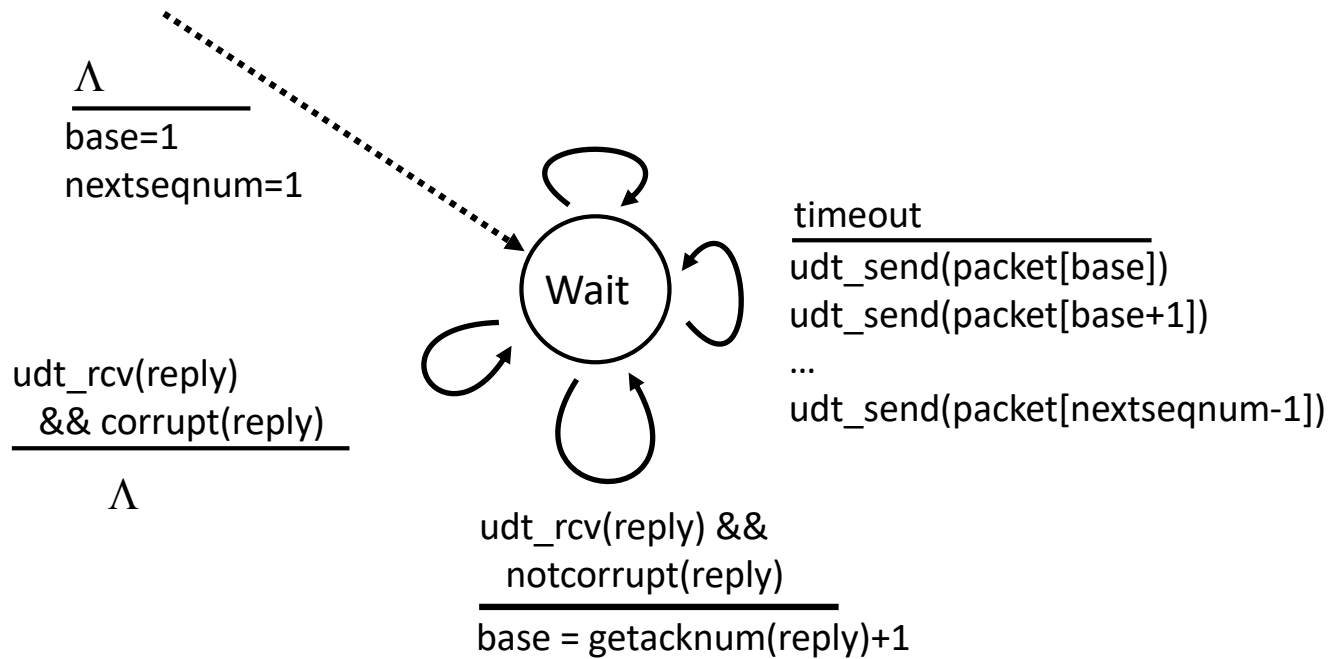
# GBN Example with Errors



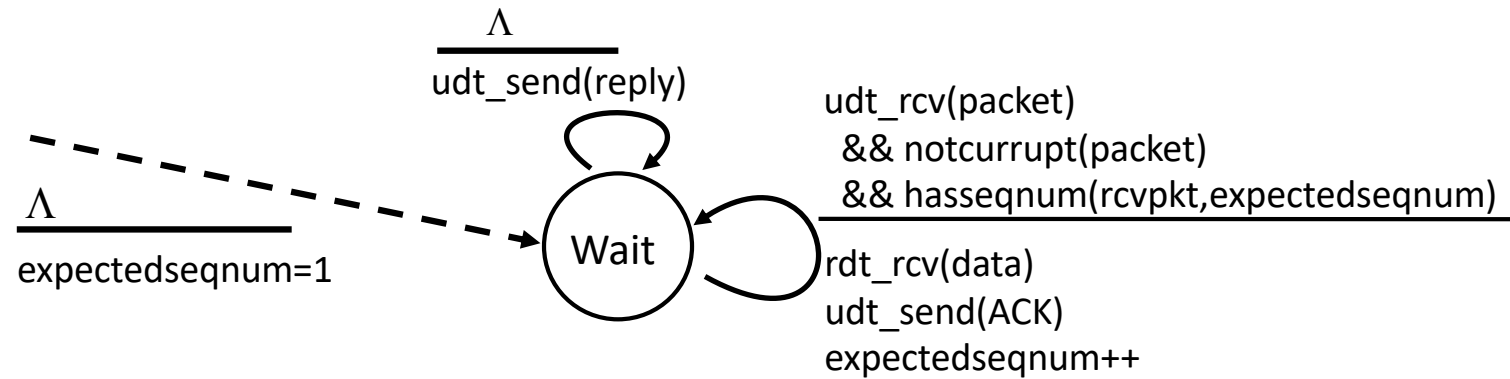
# GBN: sender extended FSM

```

rdt_send(data)
if (nextseqnum < base+N) {
  udt_send(packet[nextseqnum])
  nextseqnum++
}
else
  refuse_data(data) Block?
  
```



# GBN: receiver extended FSM



ACK-only: always send an ACK for correctly-received packet with the highest *in-order* seq #

- may generate duplicate ACKs
- need only remember **expectedseqnum**
- out-of-order packet:
  - discard (don't buffer) -> **no receiver buffering!**
  - Re-ACK packet with highest in-order seq #

# Acknowledgements w/ Sliding Window

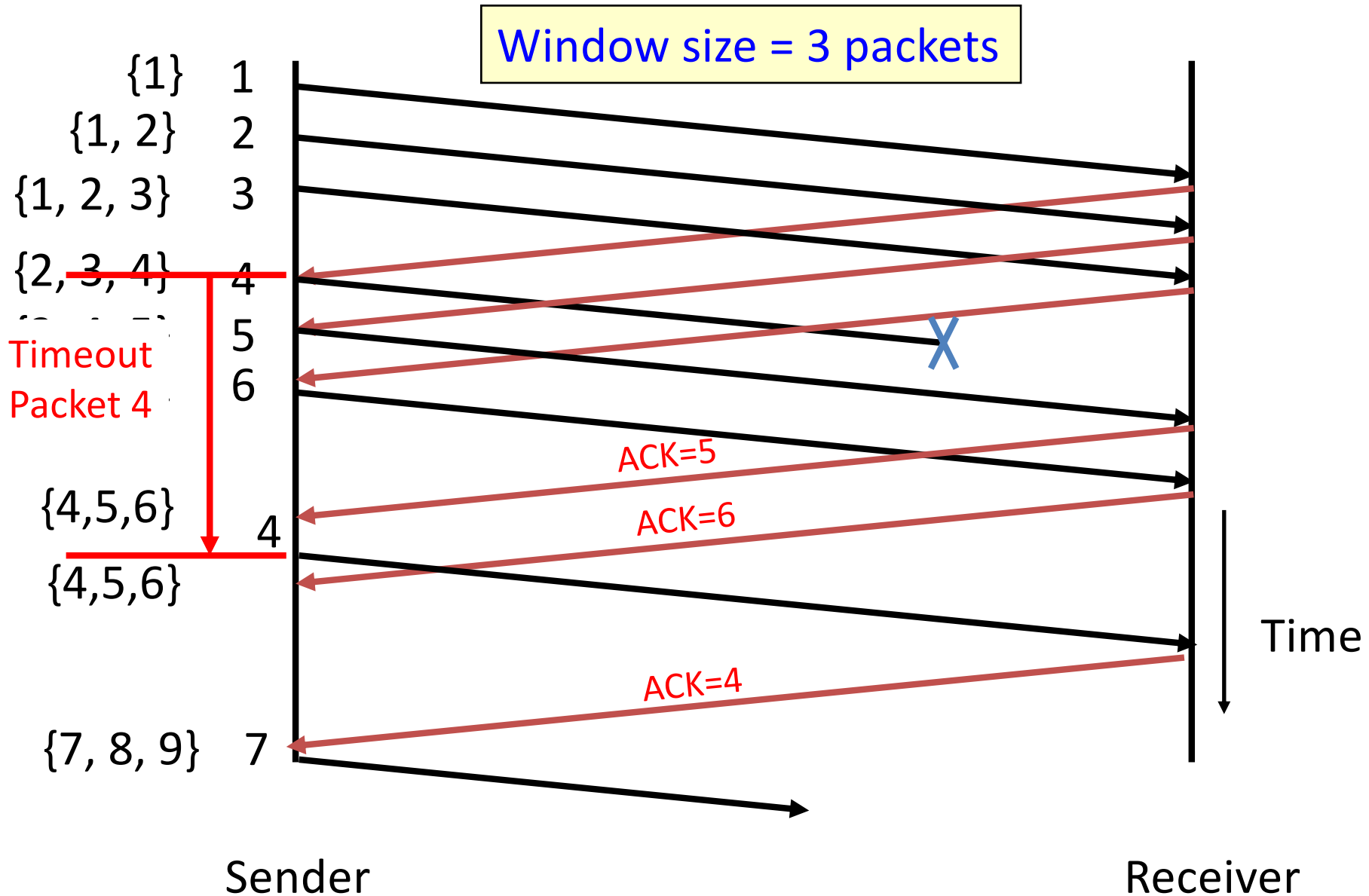
- Two common options
  - cumulative ACKs: ACK carries next in-order sequence number the receiver expects
  - selective ACKs: ACK individually acknowledges correctly received packets
- Selective ACKs offer more precise information but require more complicated book-keeping
- Many variants that differ in implementation details

# Selective Repeat (SR)

- Sender: transmit up to  $n$  unacknowledged packets
- Assume packet  $k$  is lost,  $k+1$  is not
- Receiver: indicates packet  $k+1$  correctly received
- Sender: retransmit only packet  $k$  on timeout
- Efficient in retransmissions but complex book-keeping
  - need a timer per packet



# SR Example with Errors



# Observations

- With sliding windows, it is possible to fully utilize a link, provided the window size ( $n$ ) is large enough. Throughput is  $\sim (n/RTT)$ 
  - Stop & Wait is like  $n = 1$ .
- Sender has to buffer all unacknowledged packets, because they may require retransmission
- Receiver may be able to accept out-of-order packets, but only up to its buffer limits
- Implementation complexity depends on protocol details (GBN vs. SR)

# Recap: components of a solution

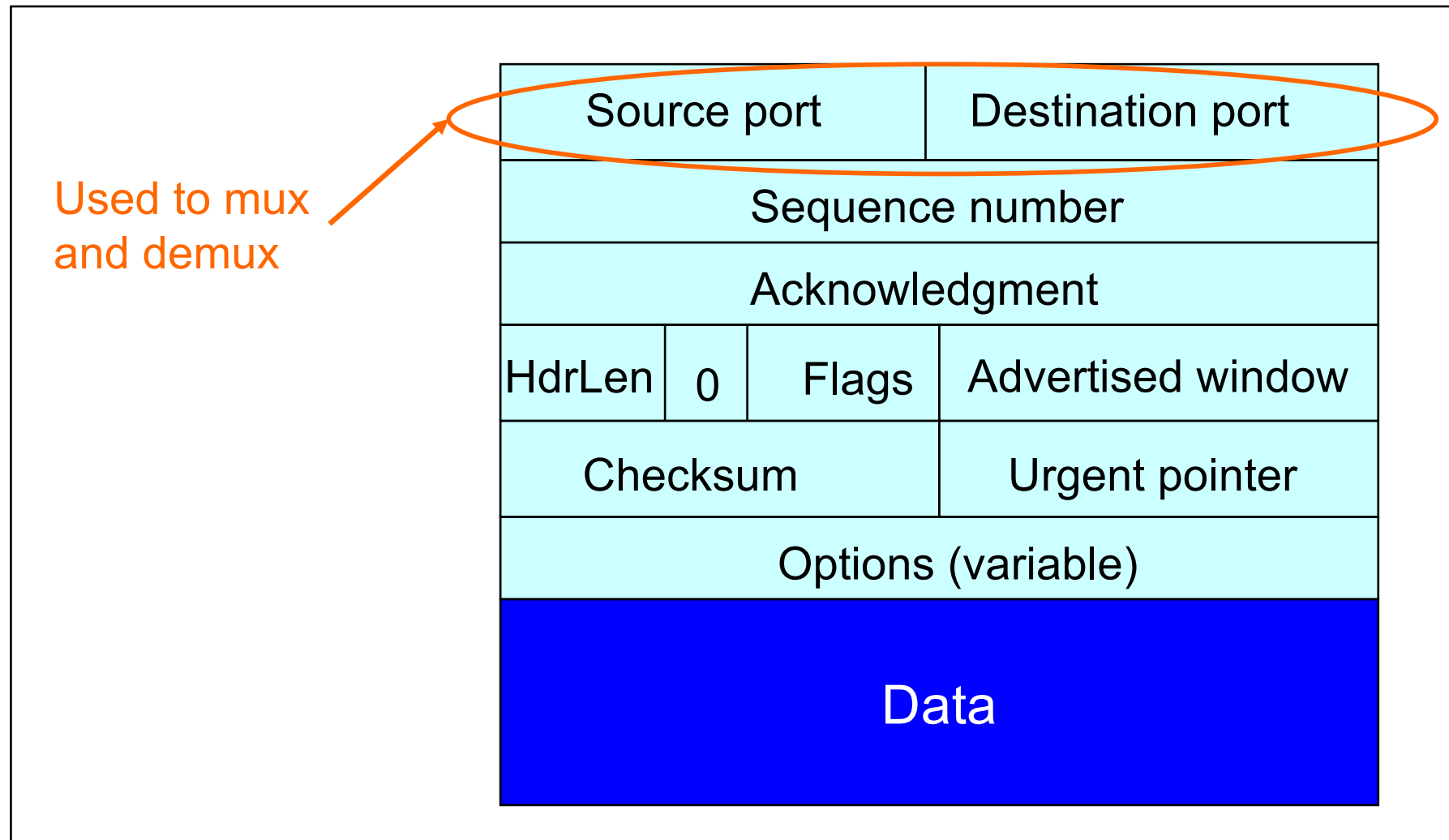
- Checksums (for error detection)
- Timers (for loss detection)
- Acknowledgments
  - cumulative
  - selective
- Sequence numbers (duplicates, windows)
- Sliding Windows (for efficiency)
  
- Reliability protocols use the above to decide when and what to retransmit or acknowledge

# What does TCP do?

Most of our previous tricks + a few differences

- Sequence numbers are byte offsets
- Sender and receiver maintain a sliding window
- Receiver sends cumulative acknowledgements (like GBN)
- Sender maintains a single retx. timer
- Receivers do not drop out-of-sequence packets (like SR)
- Introduces **fast retransmit** : optimization that uses duplicate ACKs to trigger early retx
- Introduces timeout estimation algorithms

# TCP Header

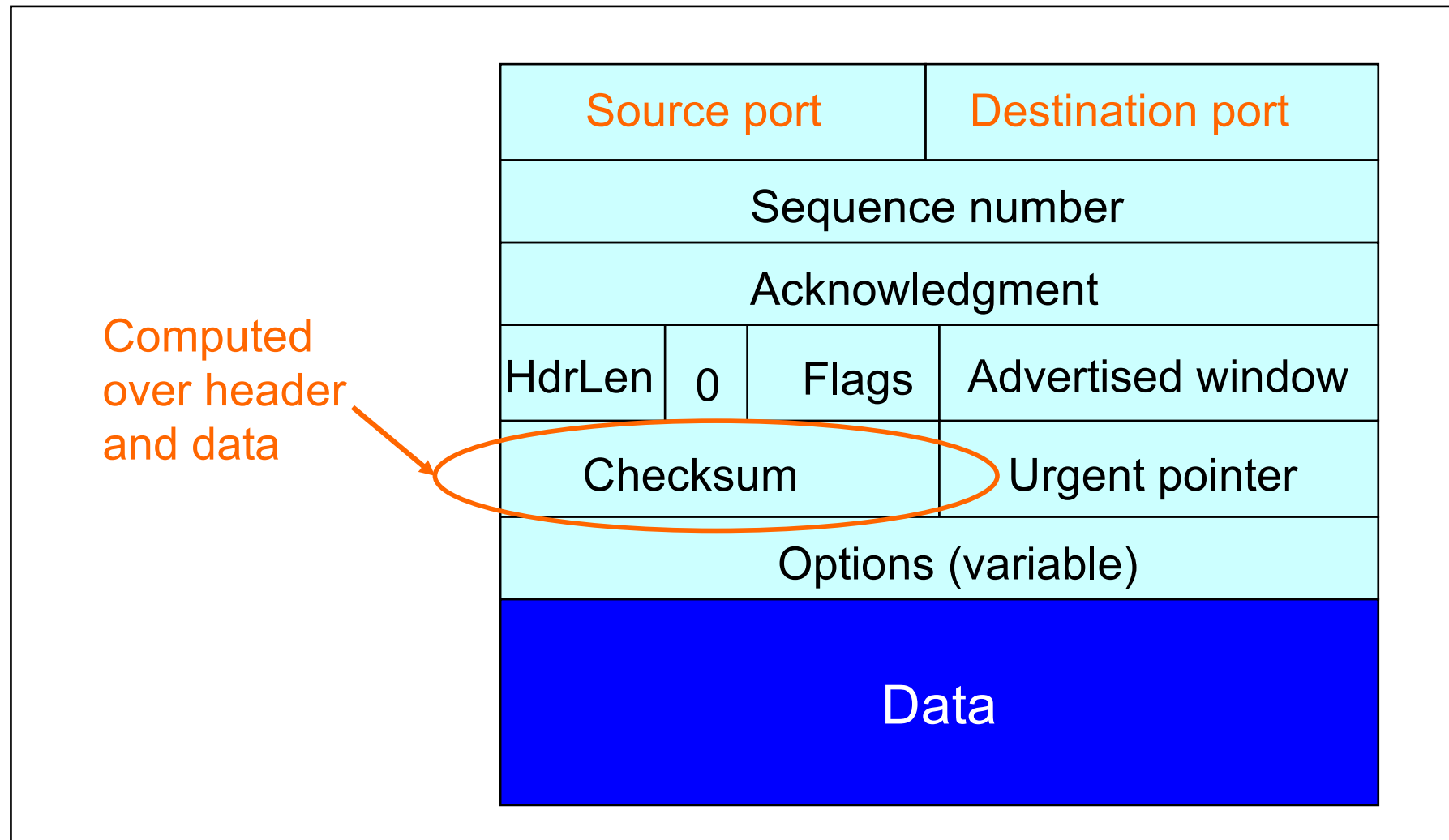


# What does TCP do?

Many of our previous ideas, but some key differences

- Checksum

# TCP Header



# What does TCP do?

Many of our previous ideas, but some key differences

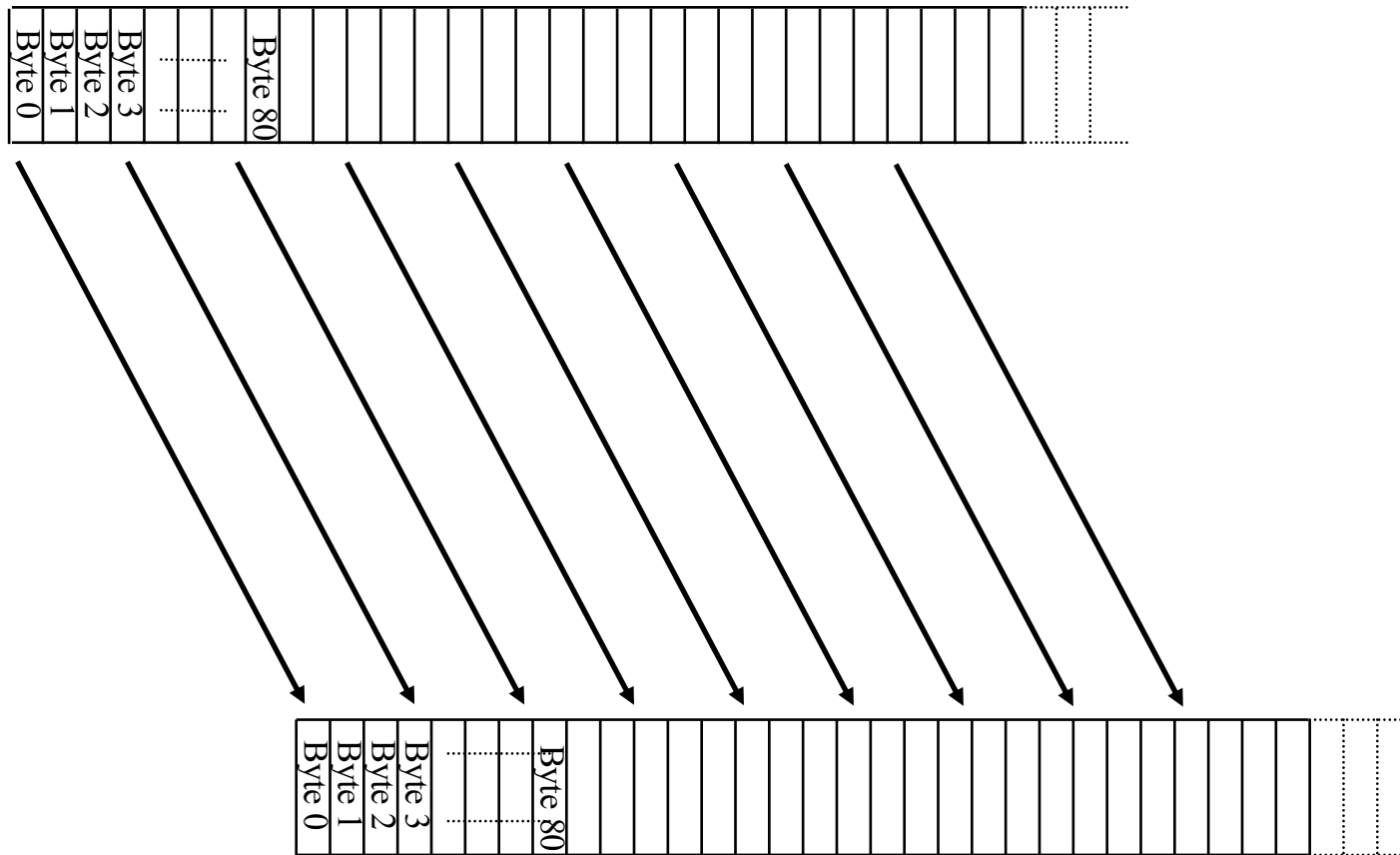
- Checksum
- **Sequence numbers are byte offsets**



# TCP: Segments and Sequence Numbers

# TCP “Stream of Bytes” Service...

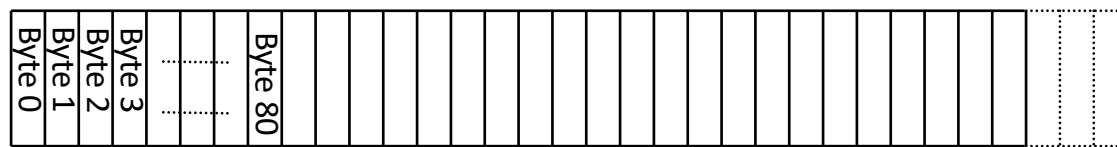
Application @ Host A



Application @ Host B

# ... Provided Using TCP “Segments”

Host A



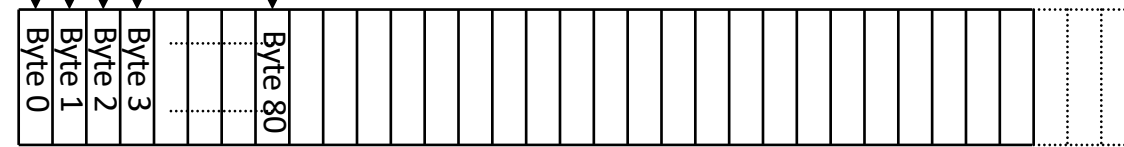
TCP Data

*Segment sent when:*

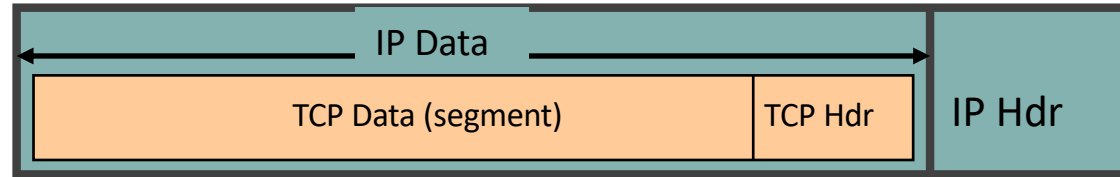
1. Segment full (Max Segment Size),
2. Not full, but times out

TCP Data

Host B

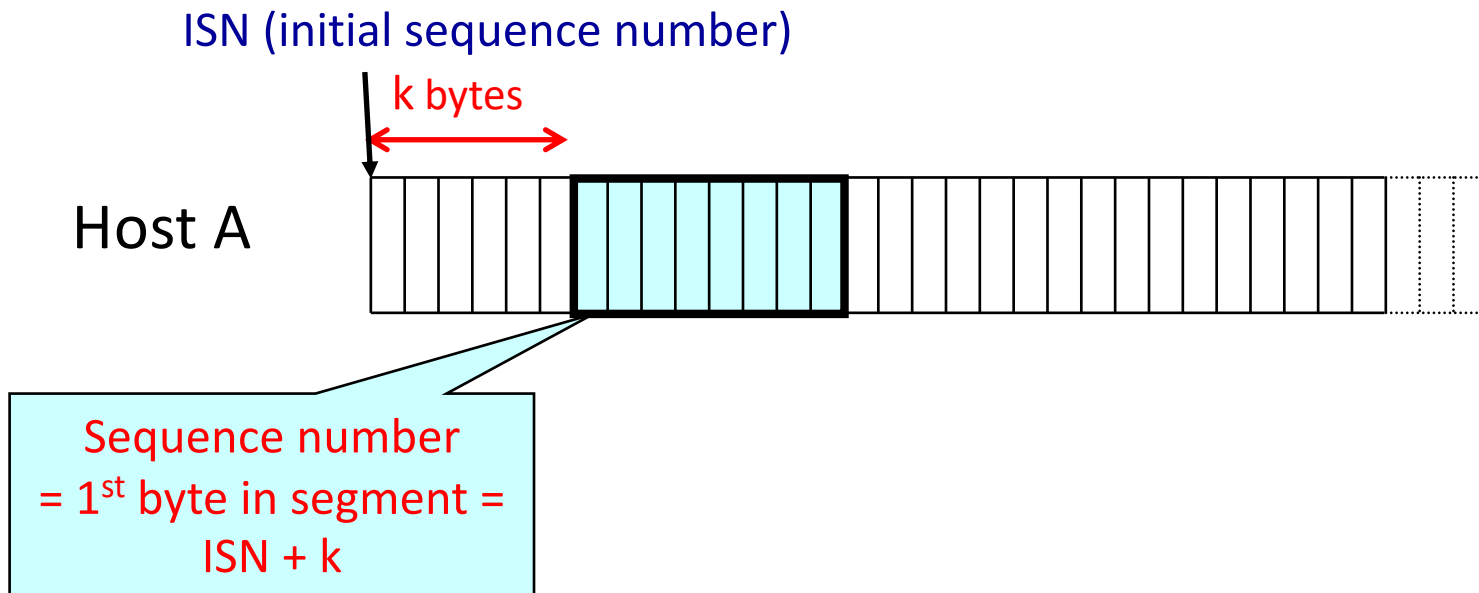


# TCP Segment

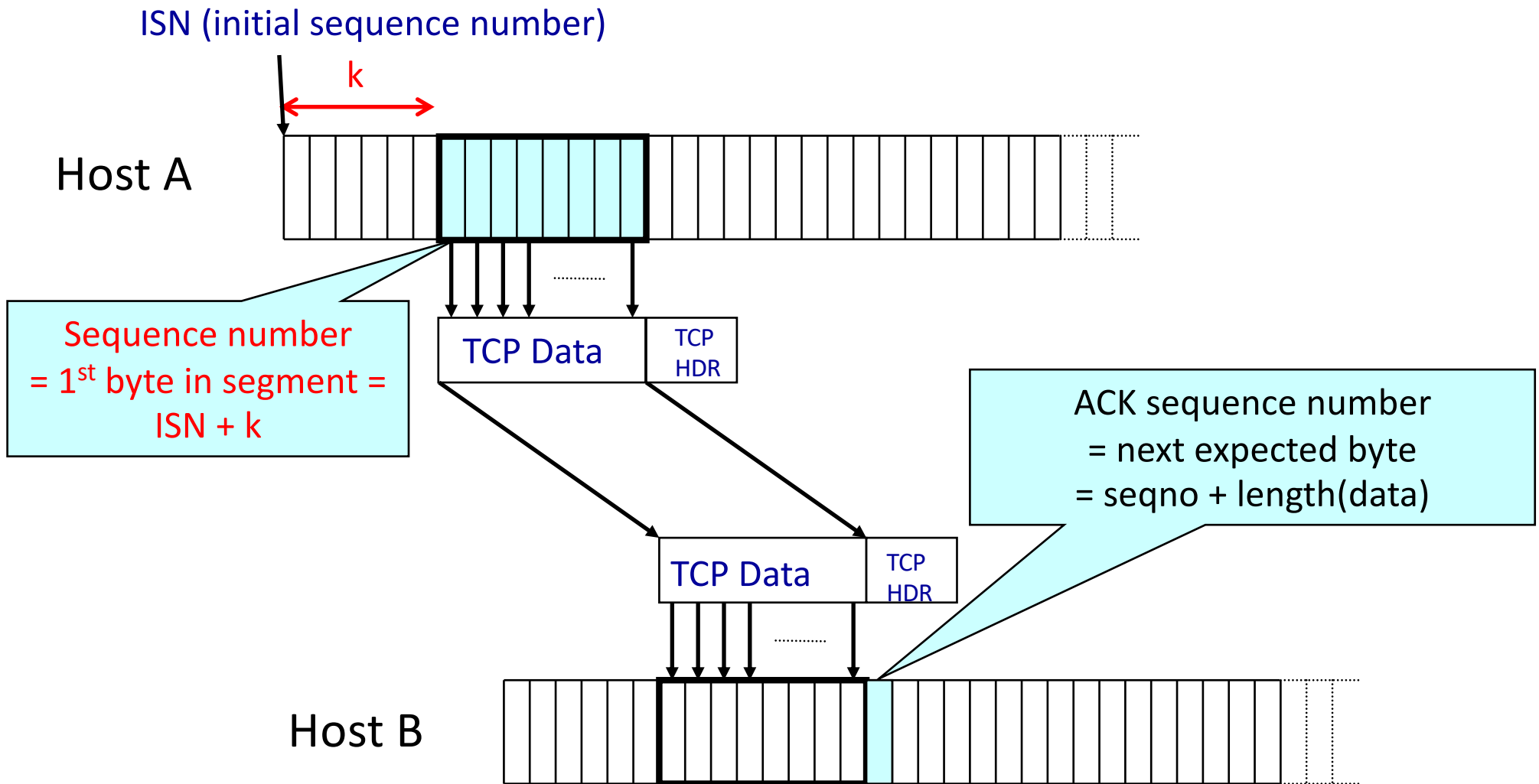


- IP packet
  - No bigger than Maximum Transmission Unit (**MTU**)
  - E.g., up to 1500 bytes with Ethernet
- TCP packet
  - IP packet with a TCP header and data inside
  - TCP header  $\geq 20$  bytes long
- **TCP segment**
  - No more than **Maximum Segment Size** (MSS) bytes
  - E.g., up to 1460 consecutive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

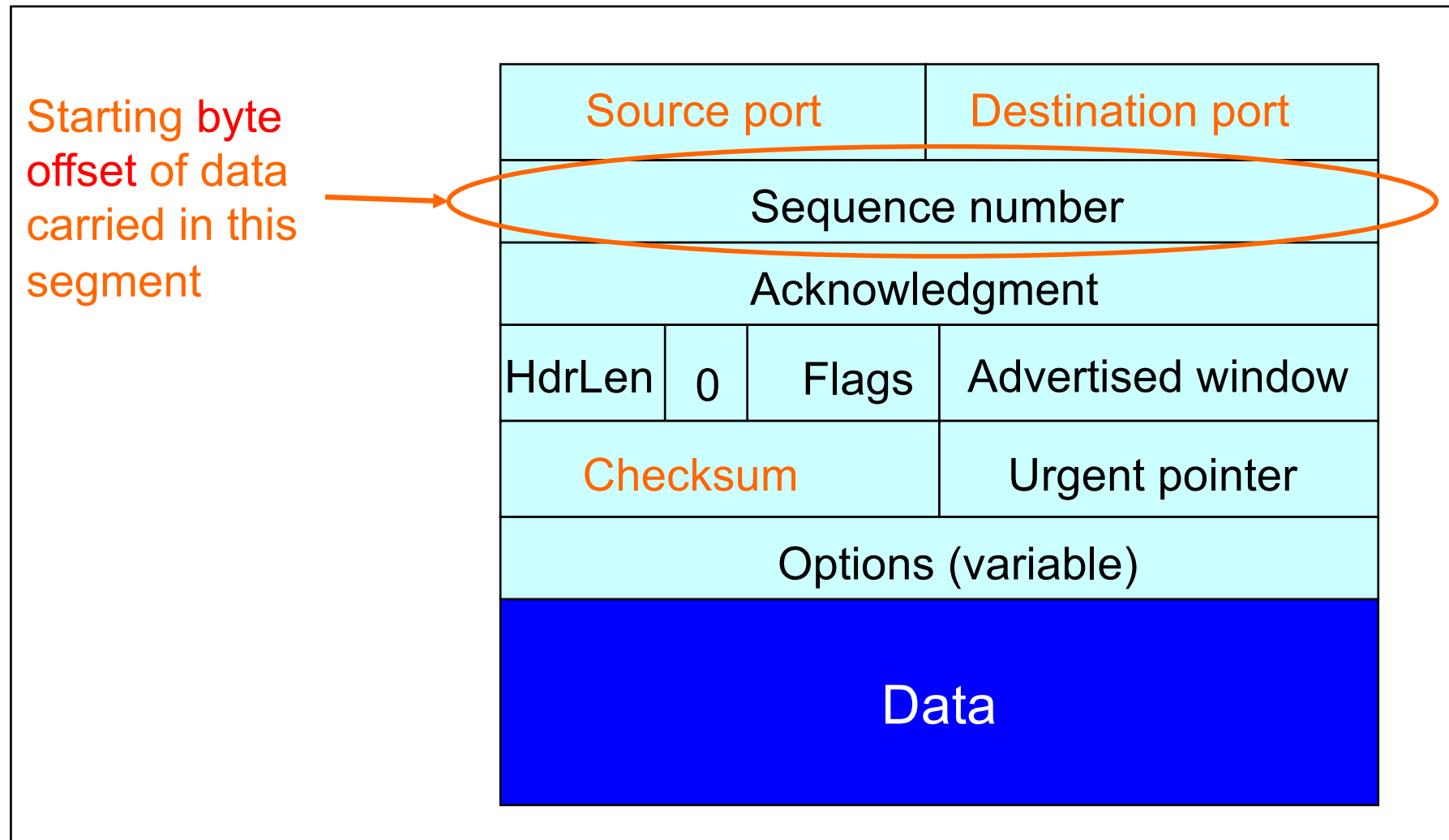
# Sequence Numbers



# Sequence Numbers



# TCP Header



- What does TCP do?



# What does TCP do?

Most of our previous tricks, but a few differences

- Checksum
- Sequence numbers are byte offsets
- Receiver sends cumulative acknowledgements (like GBN)

# ACKing and Sequence Numbers

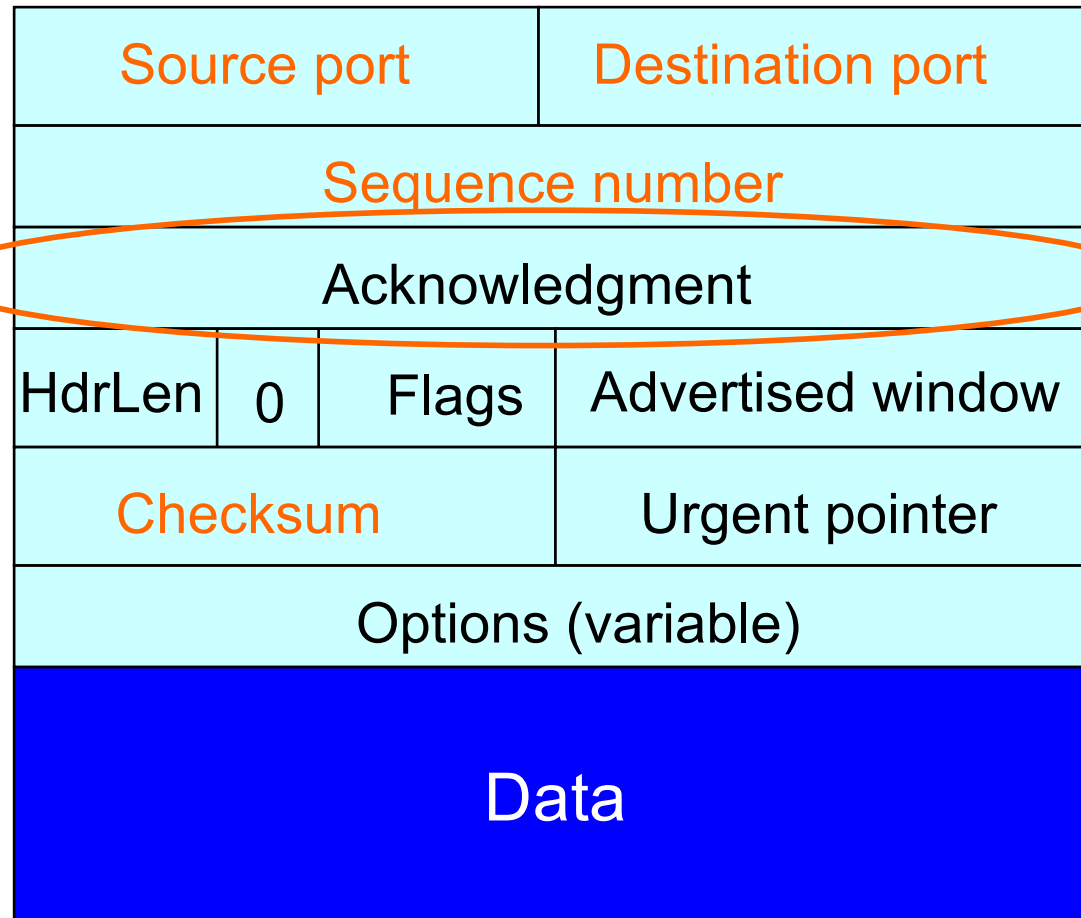
- Sender sends packet
  - Data starts with sequence number  $X$
  - Packet contains  $B$  bytes  $[X, X+1, X+2, \dots, X+B-1]$
- Upon receipt of packet, receiver sends an ACK
  - If all data prior to  $X$  already received:
    - ACK acknowledges  $X+B$  (because that is next expected byte)
  - If highest in-order byte received is  $Y$  s.t.  $(Y+1) < X$ 
    - ACK acknowledges  $Y+1$
    - Even if this has been ACKed before

# Normal Pattern

- Sender: seqno= $X$ , length= $B$
- Receiver: ACK= $X+B$
- Sender: seqno= $X+B$ , length= $B$
- Receiver: ACK= $X+2B$
- Sender: seqno= $X+2B$ , length= $B$
  
- Seqno of next packet is same as last ACK field

# TCP Header

Acknowledgment gives seqno just beyond highest seqno received **in order** (*“What Byte is Next”*)



# What does TCP do?

Most of our previous tricks, but a few differences

- Checksum
- Sequence numbers are byte offsets
- Receiver sends cumulative acknowledgements (like GBN)
- Receivers **can** buffer out-of-sequence packets (like SR)

# Loss with cumulative ACKs

- Sender sends packets with 100B and seqnos.:
  - 100, 200, 300, 400, 500, 600, 700, 800, 900, ...
- Assume the fifth packet (seqno 500) is lost, but no others
- Stream of ACKs will be:
  - 200, 300, 400, 500, 500, 500, 500, ...

# What does TCP do?

Most of our previous tricks, but a few differences

- Checksum
- Sequence numbers are byte offsets
- Receiver sends cumulative acknowledgements (like GBN)
- Receivers may not drop out-of-sequence packets (like SR)
- Introduces **fast retransmit**: optimization that uses duplicate ACKs to trigger early retransmission

# Loss with cumulative ACKs

- “Duplicate ACKs” are a sign of an isolated loss
  - The lack of ACK progress means 500 hasn’t been delivered
  - Stream of ACKs means some packets are being delivered
- Therefore, could trigger resend upon receiving k duplicate ACKs
  - TCP uses  $k=3$
- But response to loss is trickier....



# Loss with cumulative ACKs

- Two choices:
  - Send missing packet and increase  $W$  by the number of dup ACKs
  - Send missing packet, and wait for ACK to increase  $W$
- Which should TCP do?

# What does TCP do?

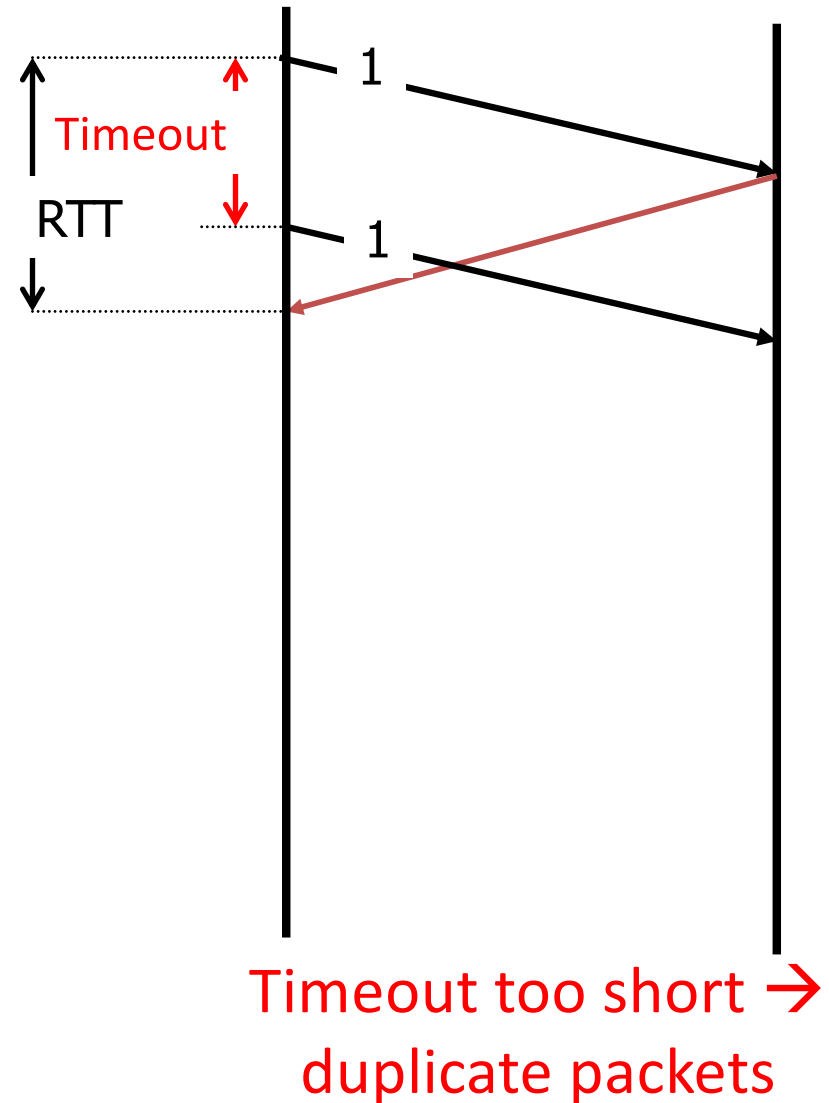
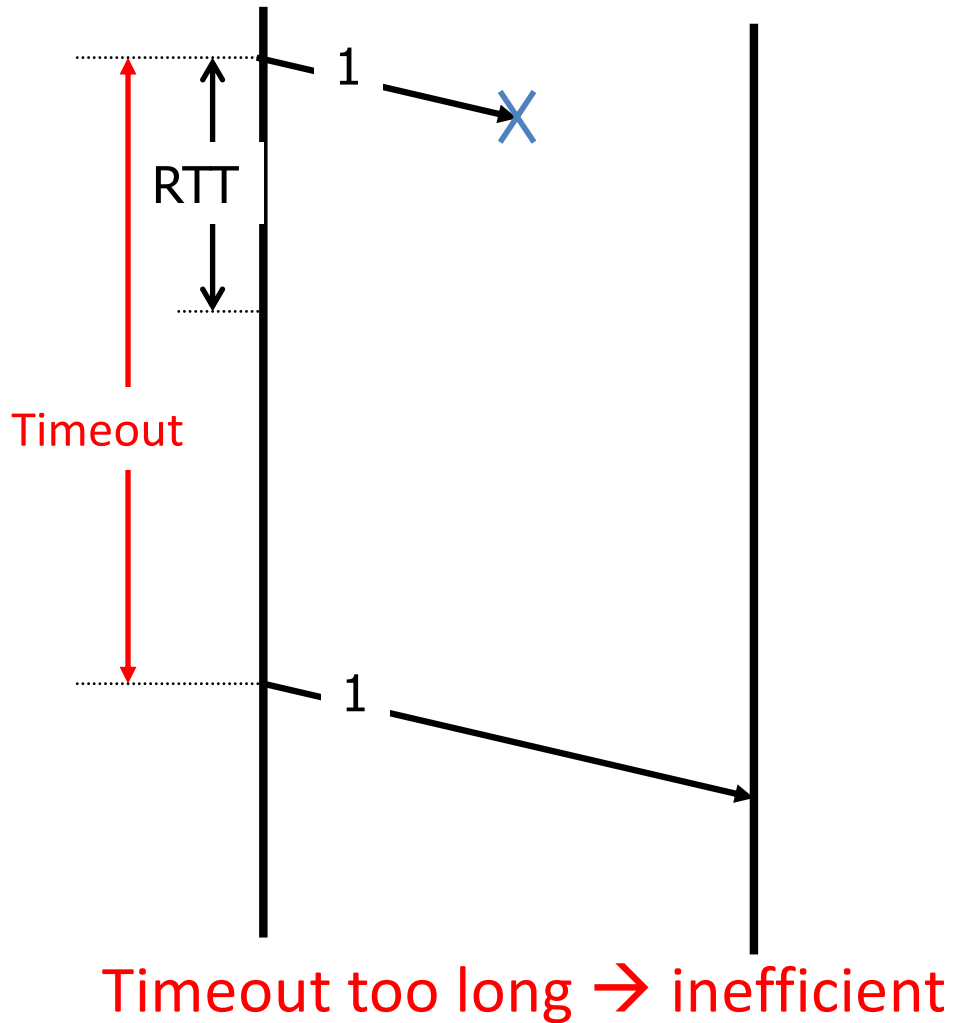
Most of our previous tricks, but a few differences

- Checksum
- Sequence numbers are byte offsets
- Receiver sends cumulative acknowledgements (like GBN)
- Receivers do not drop out-of-sequence packets (like SR)
- Introduces fast retransmit: optimization that uses duplicate ACKs to trigger early retransmission
- Sender maintains a single retransmission timer (like GBN) and retransmits on timeout

# Retransmission Timeout

- If the sender hasn't received an ACK by timeout, retransmit the first packet in the window
- How do we pick a timeout value?

# Timing Illustration



# Retransmission Timeout

- If haven't received ack by timeout, retransmit the first packet in the window
- How to set timeout?
  - Too long: connection has low throughput
  - Too short: retransmit packet that was just delayed
- Solution: make timeout proportional to RTT
- But how do we measure RTT?

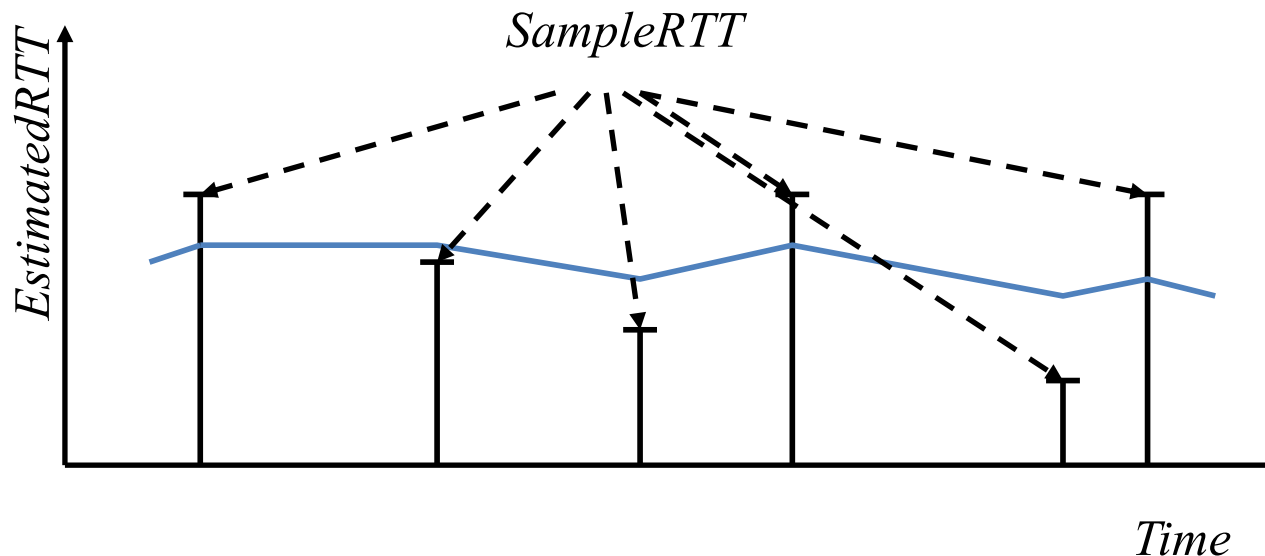
# RTT Estimation

- Use exponential averaging of RTT samples

$$\text{SampleRTT} = \text{AckRcvdTime} - \text{SendPacketTime}$$

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

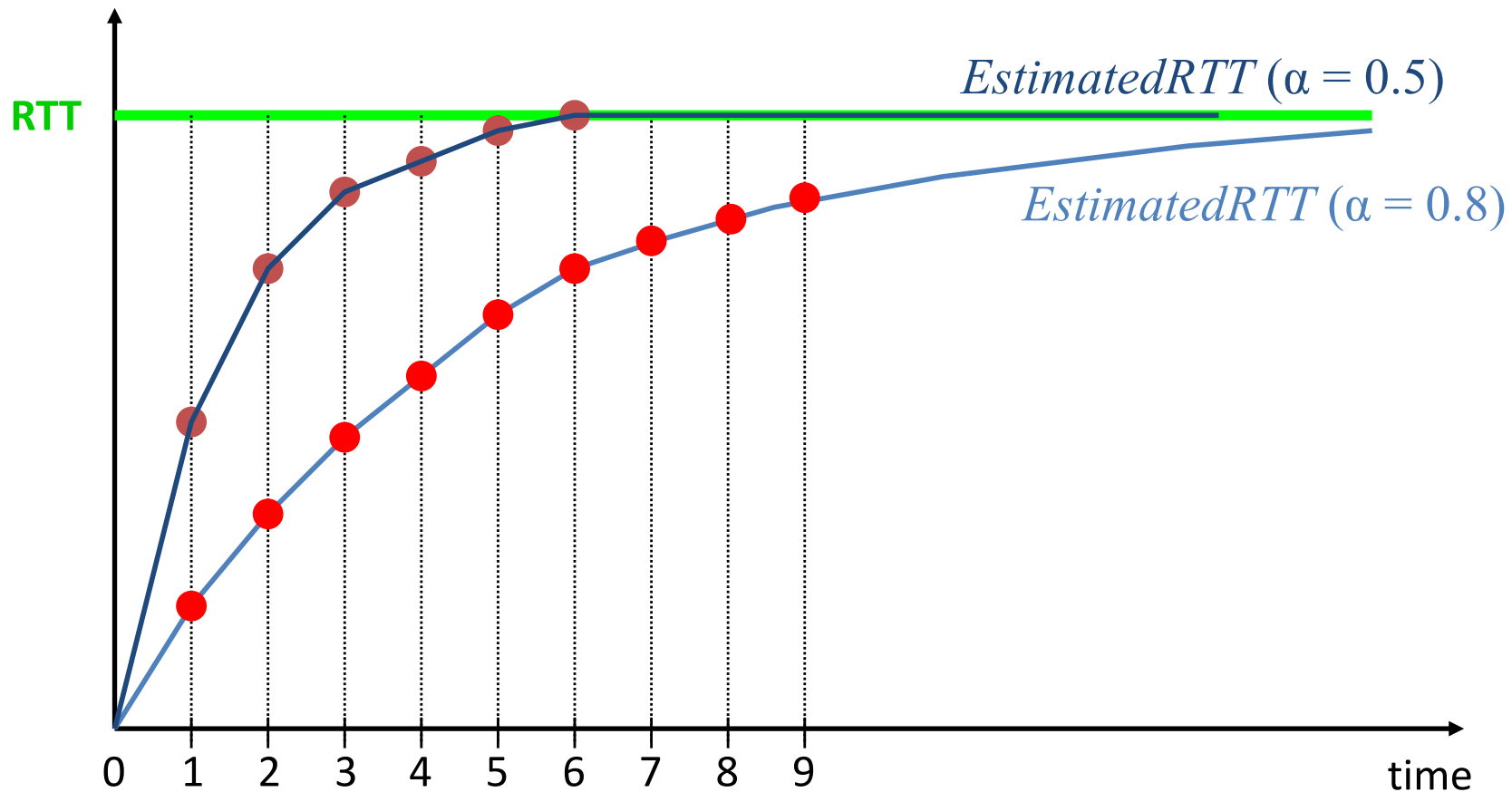
$$0 < \alpha \leq 1$$



# Exponential Averaging Example

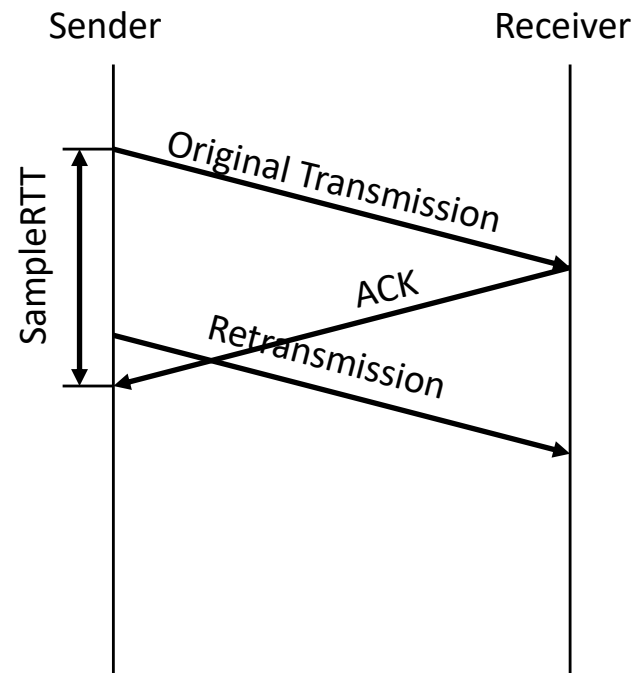
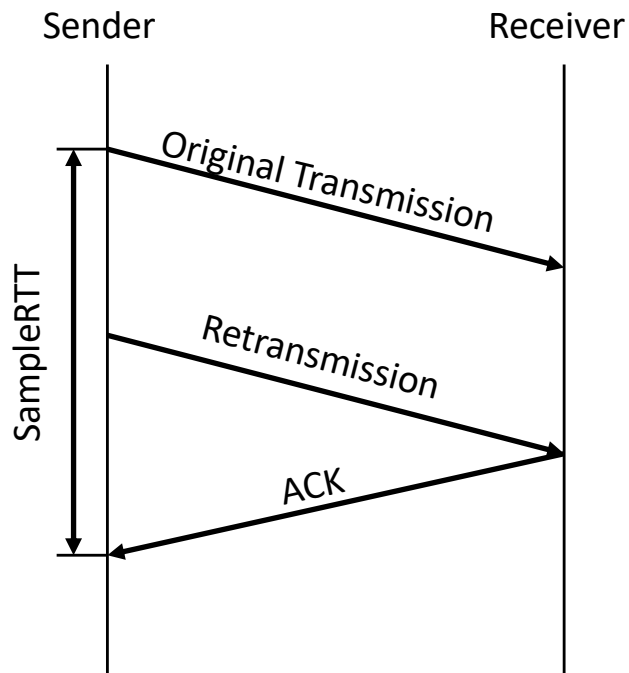
$$\text{EstimatedRTT} = \alpha * \text{EstimatedRTT} + (1 - \alpha) * \text{SampleRTT}$$

Assume RTT is constant  $\rightarrow$   $\text{SampleRTT} = \text{RTT}$



# Problem: Ambiguous Measurements

- How do we differentiate between the real ACK, and ACK of the retransmitted packet?



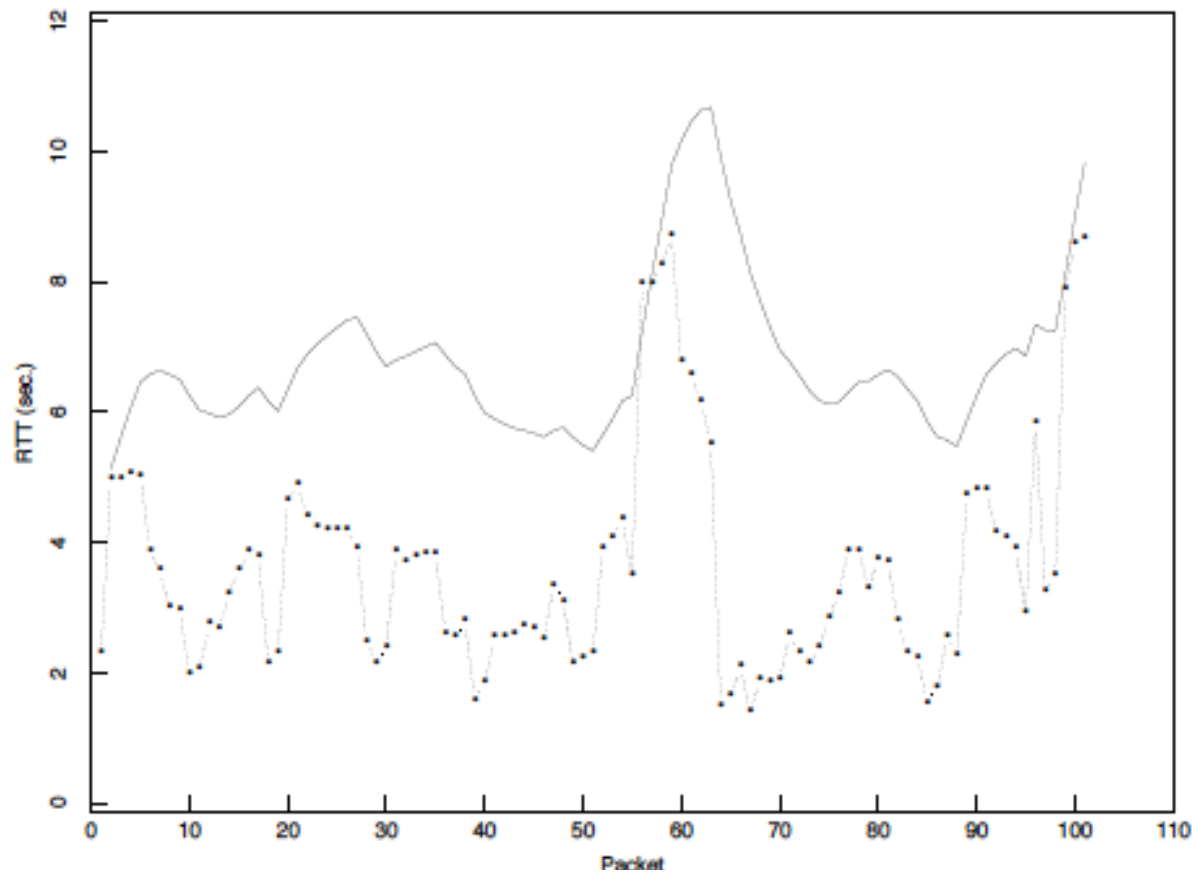


# Karn/Partridge Algorithm

- Measure *SampleRTT* only for original transmissions
  - Once a segment has been retransmitted, do not use it for any further measurements
- Computes EstimatedRTT using  $\alpha = 0.875$
- Timeout value (RTO) =  $2 \times$  EstimatedRTT
- Employs **exponential backoff**
  - Every time RTO timer expires, set  $RTO \leftarrow 2 \cdot RTO$
  - (Up to maximum  $\geq 60$  sec)
  - Every time new measurement comes in (= successful original transmission), collapse RTO back to  $2 \times$  EstimatedRTT

# Karn/Partridge in action

Figure 5: Performance of an RFC793 retransmit timer



from Jacobson and Karels, SIGCOMM 1988

# Jacobson/Karels Algorithm

- Problem: need to better capture variability in RTT
  - Directly measure **deviation**
- Deviation = | SampleRTT – EstimatedRTT |
- EstimatedDeviation: exponential average of Deviation
- RTO = EstimatedRTT + 4 x EstimatedDeviation

# With Jacobson/Karels

Figure 5: Performance of an RFC793 retransmit timer

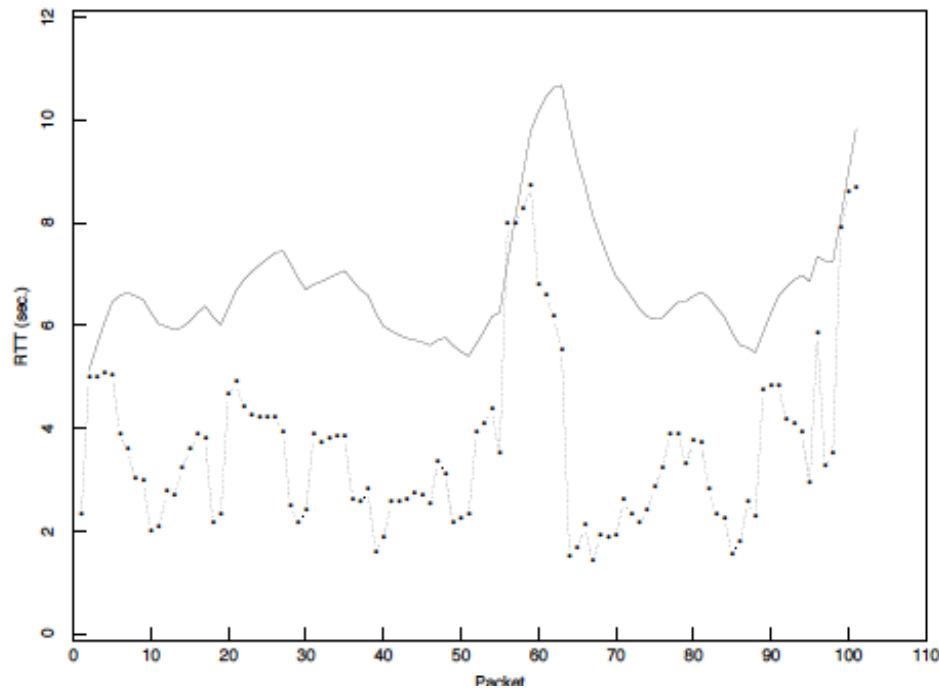
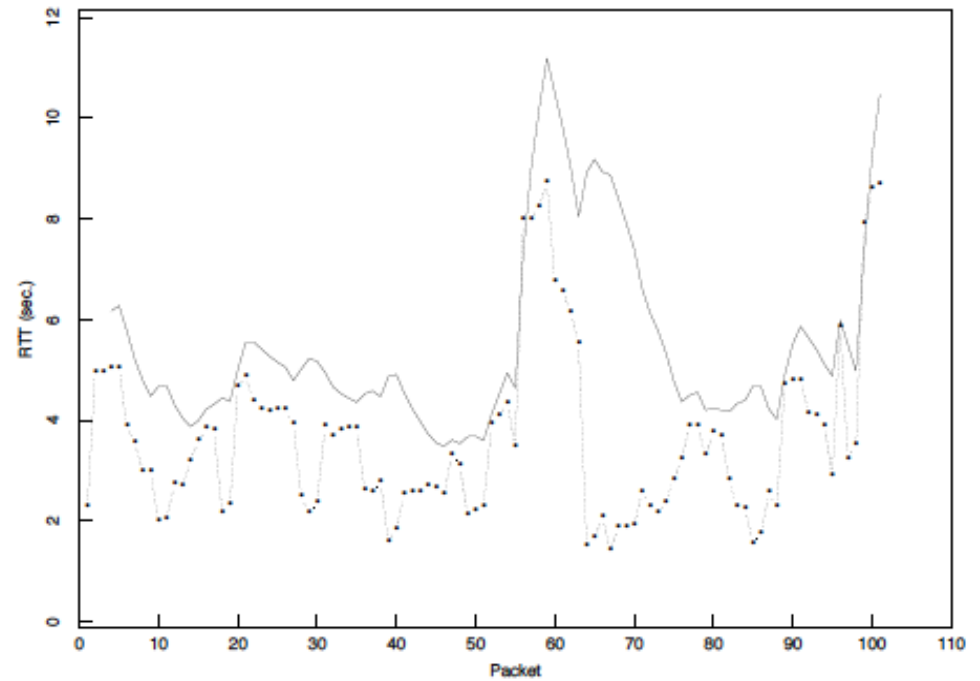


Figure 6: Performance of a Mean+Variance retransmit timer

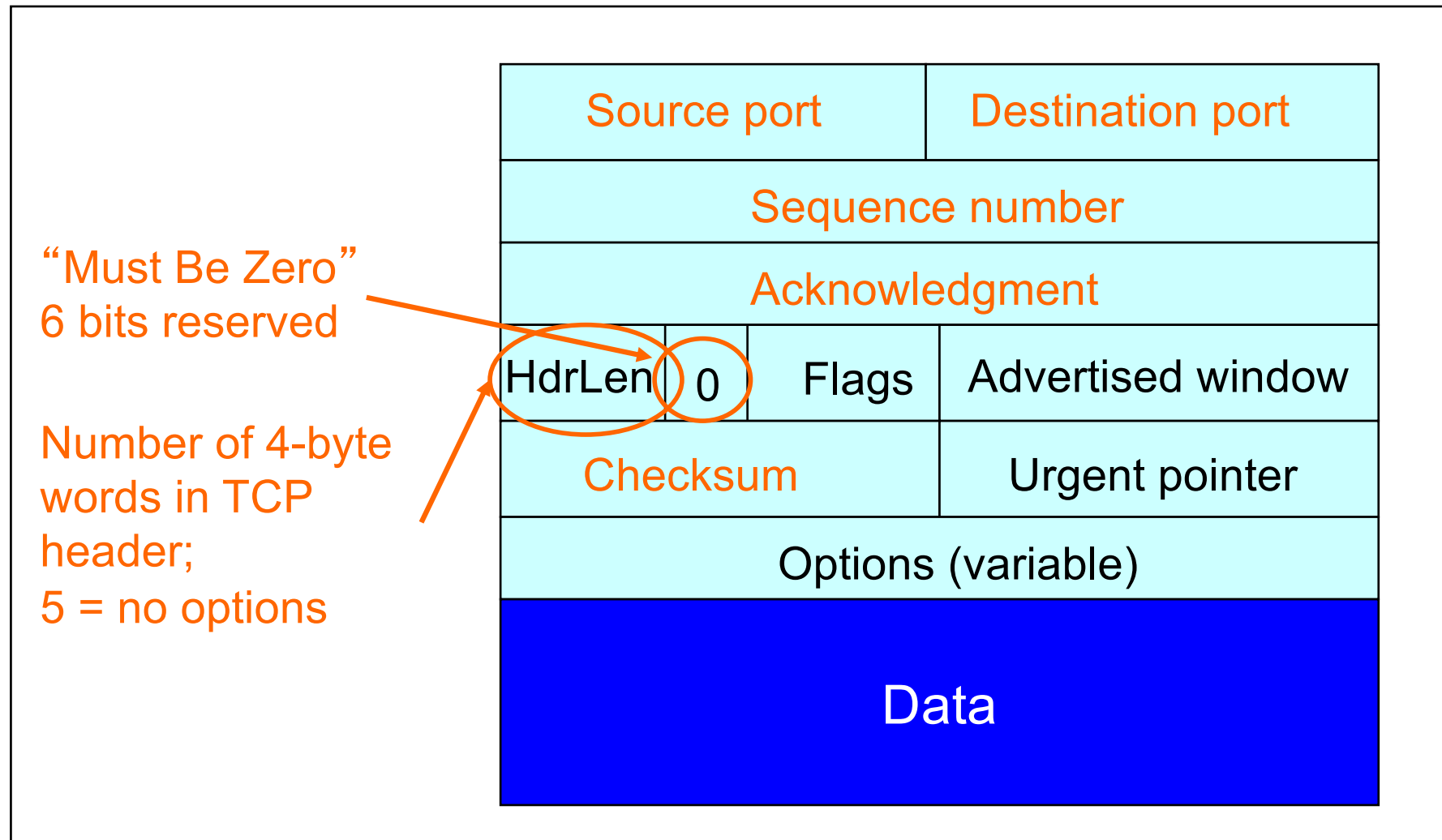


# What does TCP do?

Most of our previous ideas, but some key differences

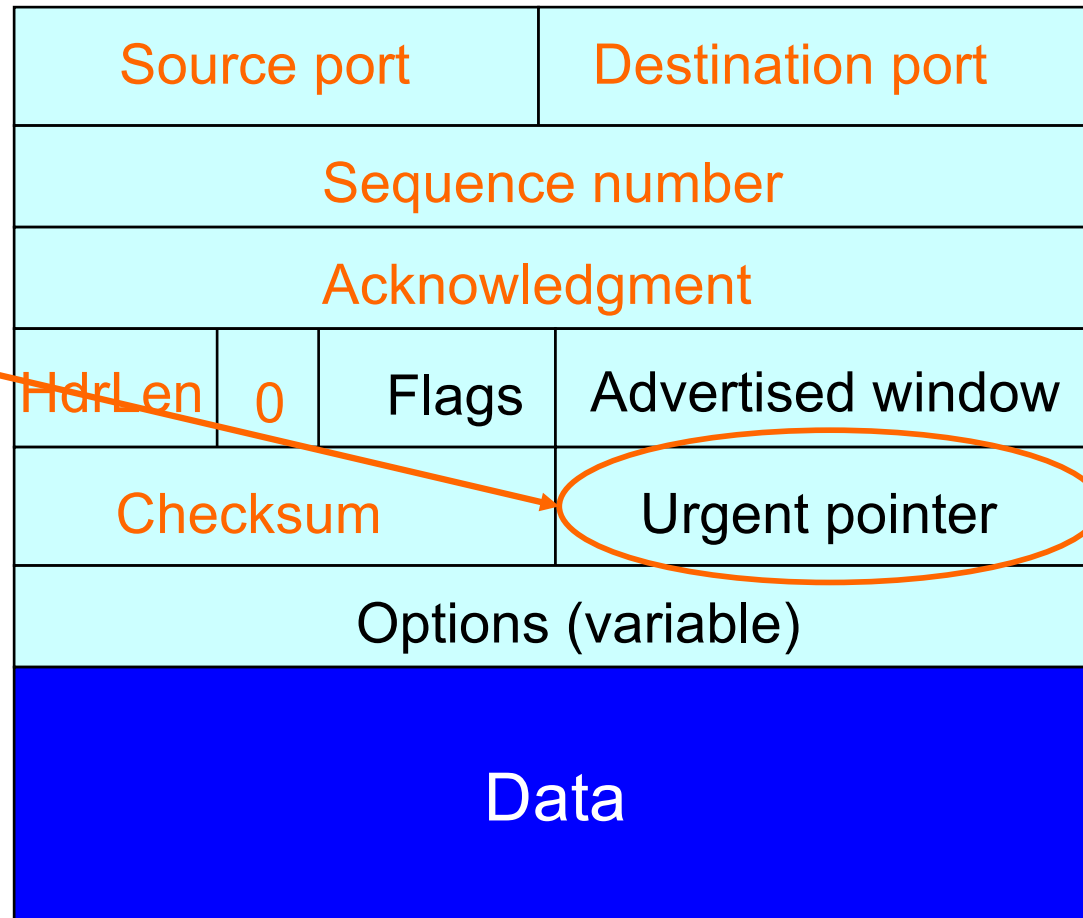
- Checksum
- Sequence numbers are byte offsets
- Receiver sends cumulative acknowledgements (like GBN)
- Receivers do not drop out-of-sequence packets (like SR)
- Introduces fast retransmit: optimization that uses duplicate ACKs to trigger early retransmission
- Sender maintains a single retransmission timer (like GBN) and retransmits on timeout

# TCP Header: What's left?

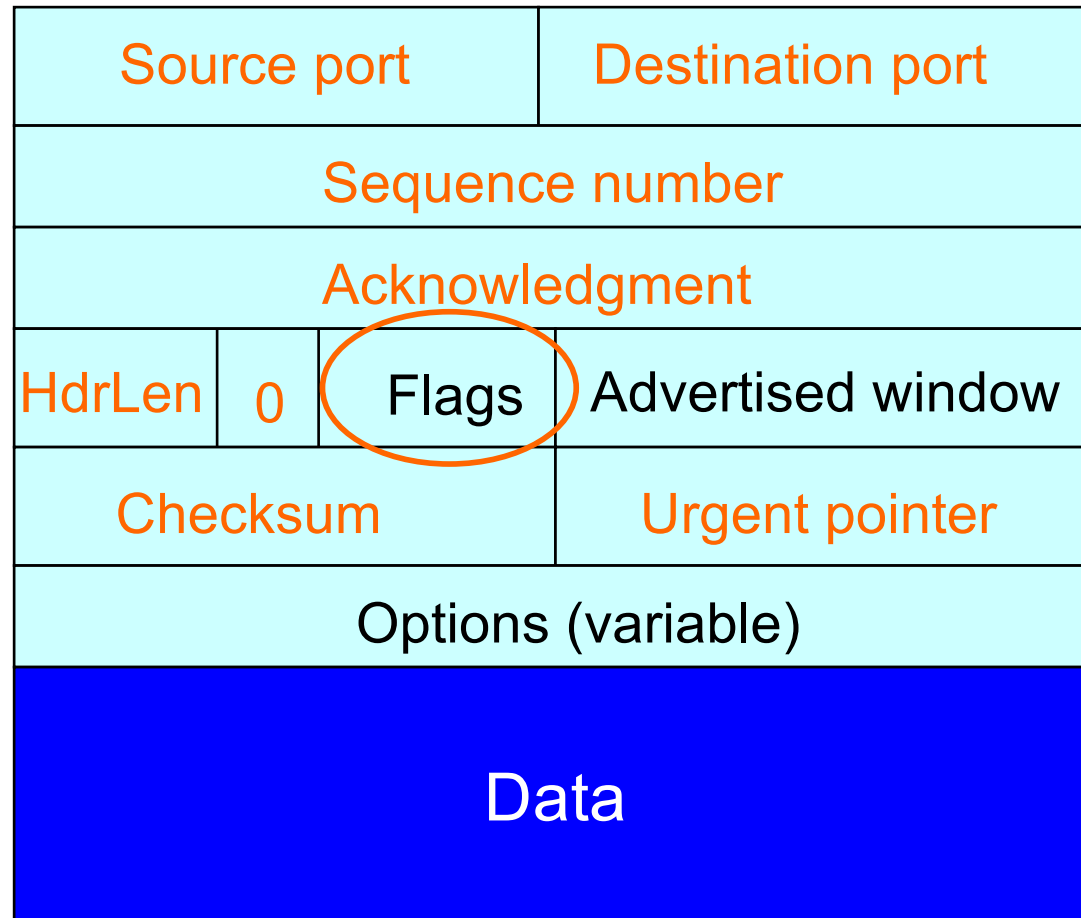


# TCP Header: What's left?

Used with **URG** flag to indicate urgent data (not discussed further)



# TCP Header: What's left?



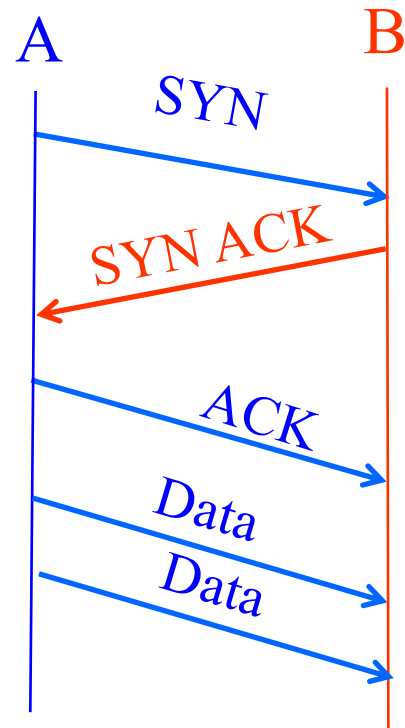


# TCP Connection Establishment and Initial Sequence Numbers

# Initial Sequence Number (ISN)

- Sequence number for the very first byte
- Why not just use ISN = 0?
- Practical issue
  - IP addresses and port #s uniquely identify a connection
  - Eventually, though, these port #s do get **used again**
  - ... small chance an old packet is **still in flight**
- TCP therefore **requires** changing ISN
- Hosts exchange ISNs when they establish a connection

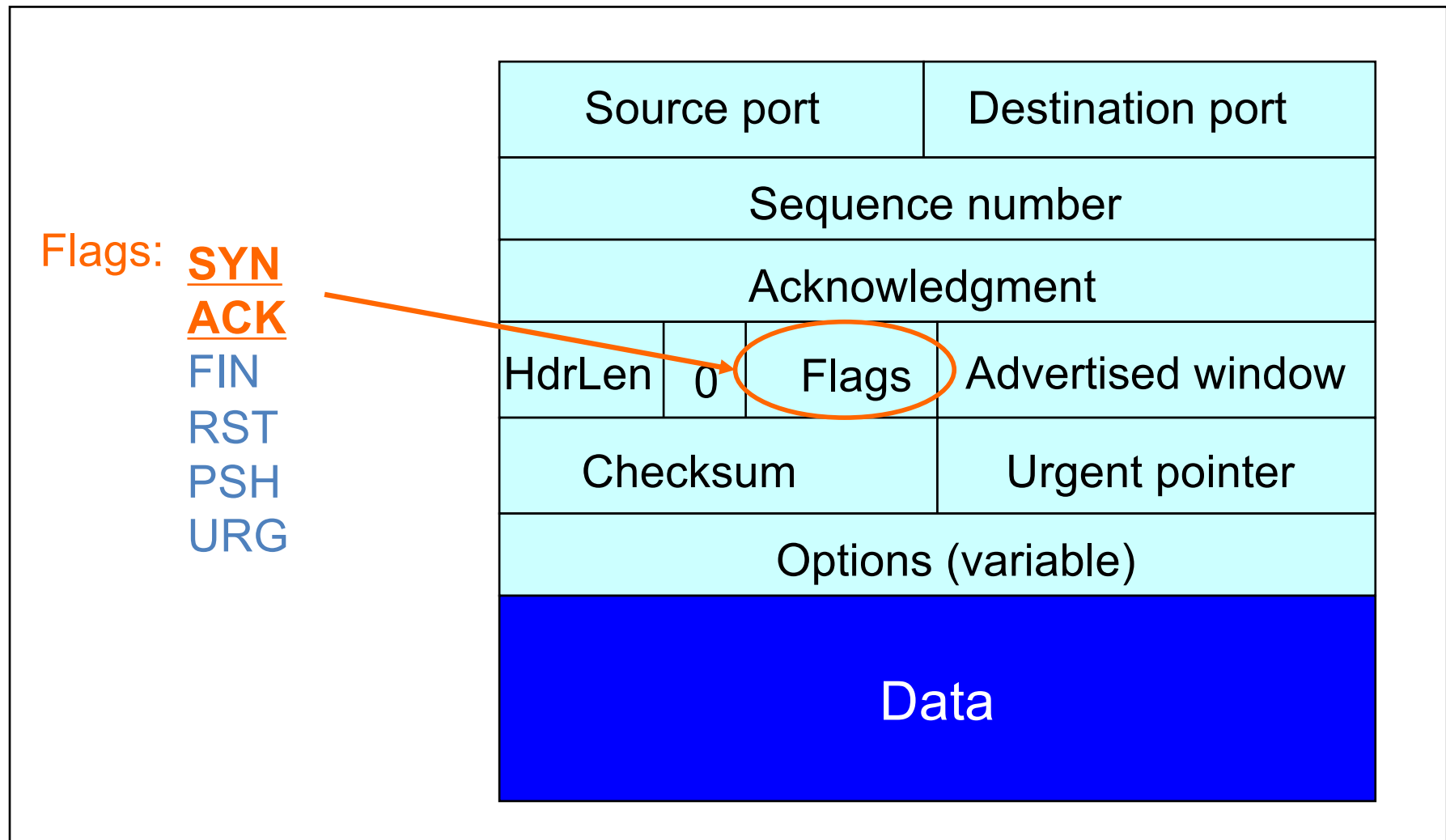
# Establishing a TCP Connection



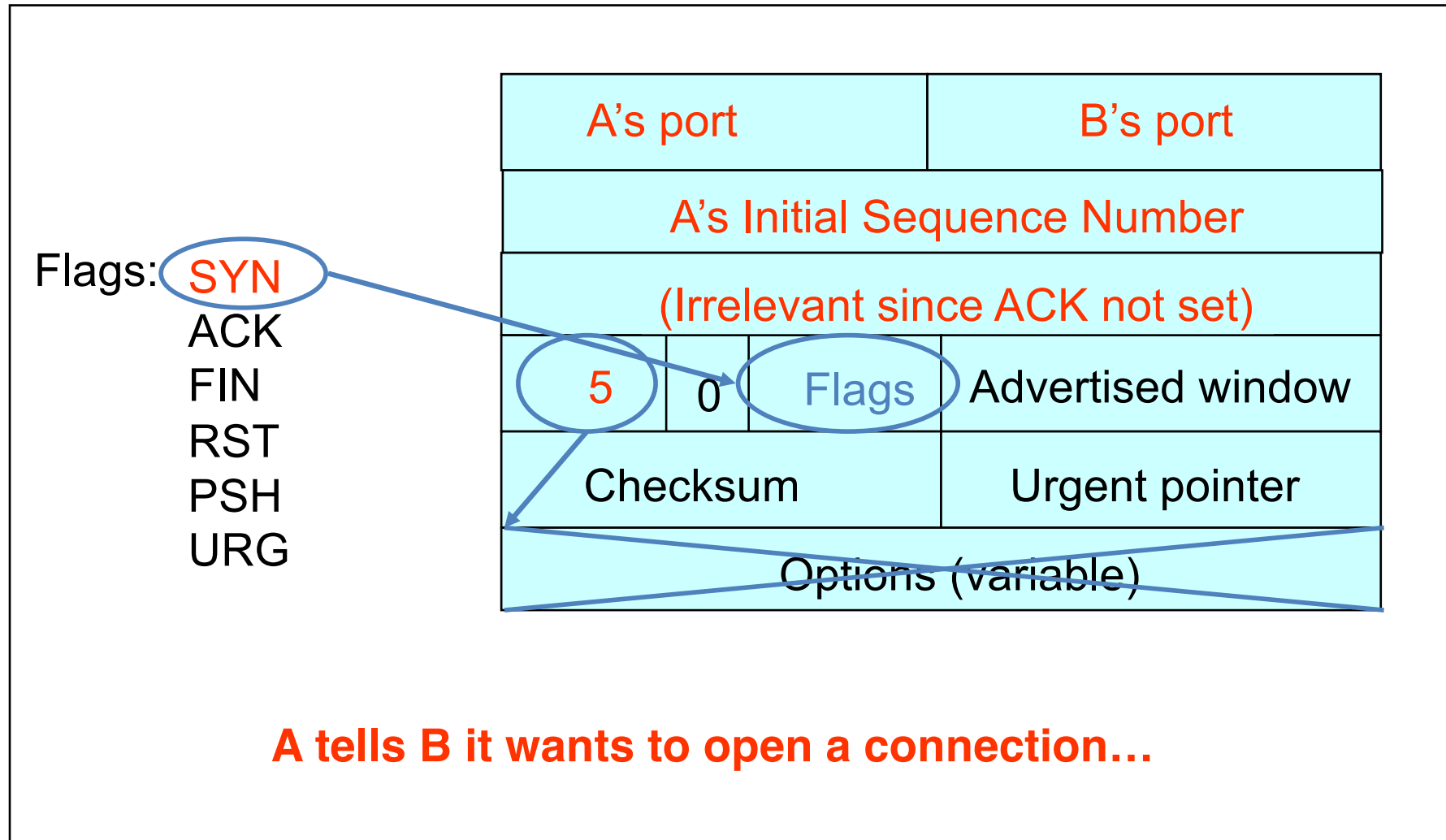
Each host tells its ISN to the other host.

- Three-way handshake to establish connection
  - Host A sends a **SYN** (open; “synchronize sequence numbers”) to host B
  - Host B returns a SYN acknowledgment (**SYN ACK**)
  - Host A sends an **ACK** to acknowledge the SYN ACK

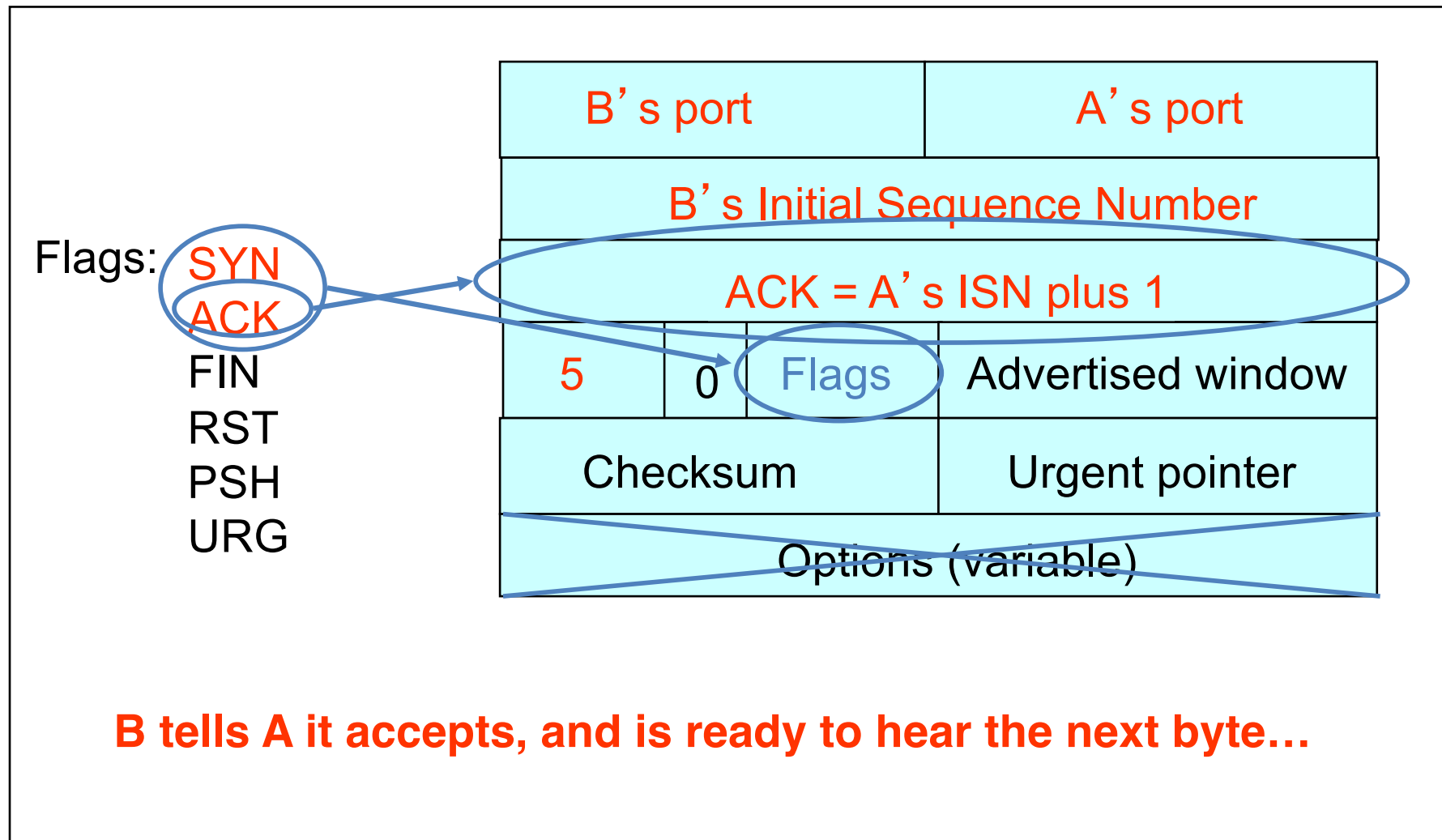
# TCP Header



# Step 1: A's Initial SYN Packet



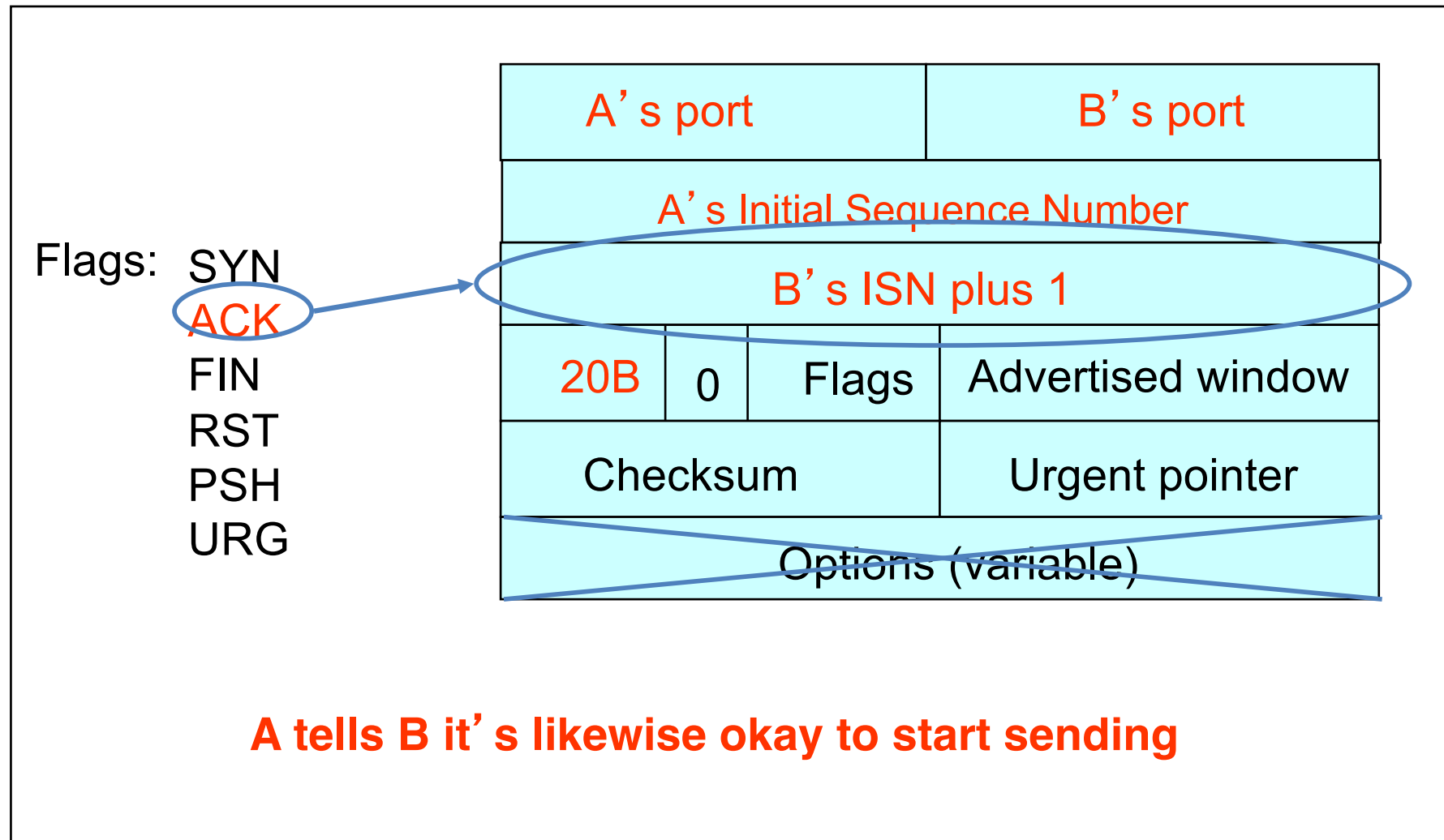
# Step 2: B's SYN-ACK Packet



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

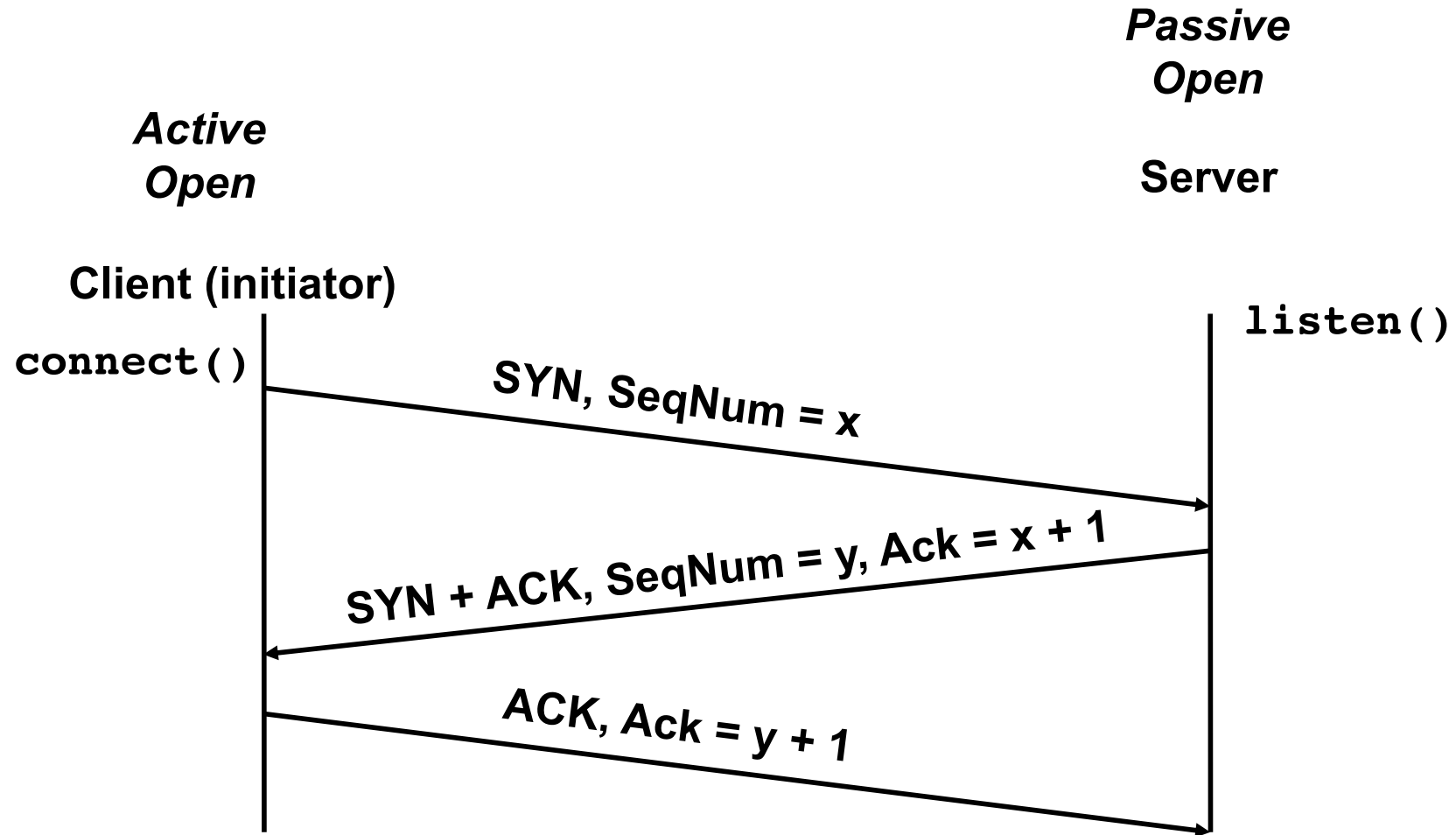
**... upon receiving this packet, A can start sending data**

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



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

# Timing Diagram: 3-Way Handshaking



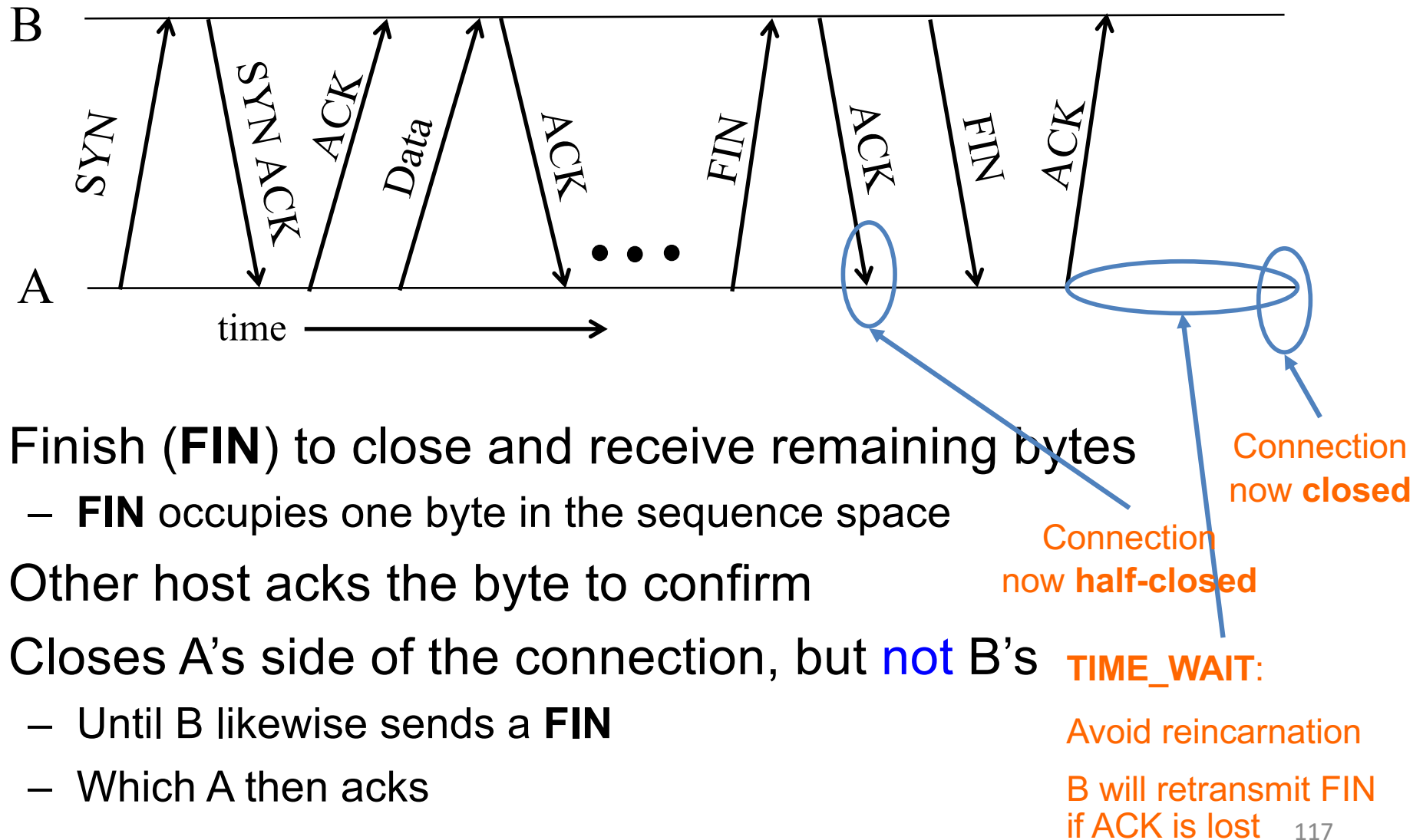


# What if the SYN Packet Gets Lost?

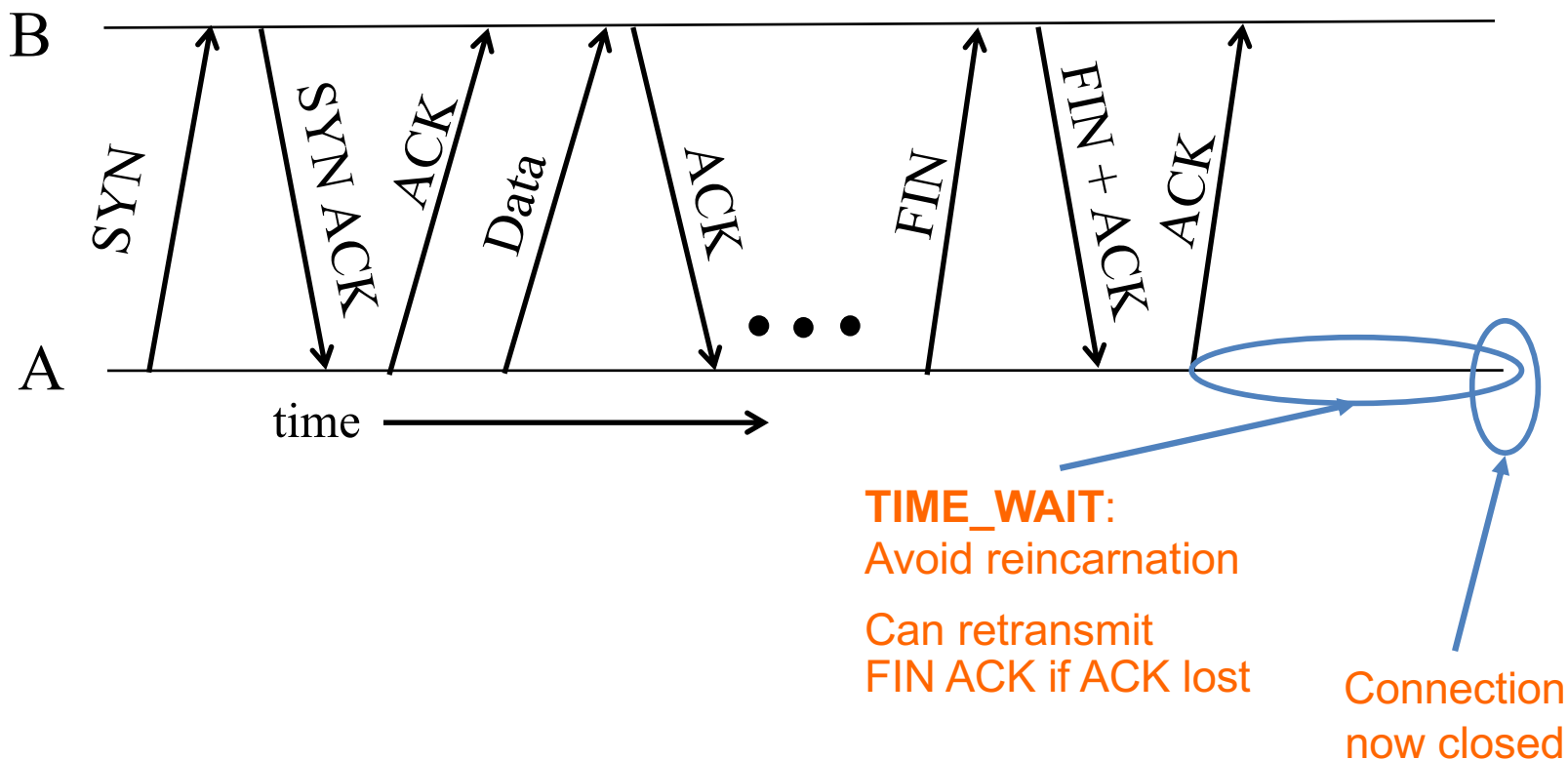
- Suppose the SYN packet gets lost
  - Packet is lost inside the network, or:
  - Server **discards** the packet (e.g., it's too busy)
- Eventually, no SYN-ACK arrives
  - Sender sets a **timer** and **waits** for the SYN-ACK
  - ... and retransmits the SYN if needed
- How should the TCP sender set the timer?
  - Sender has **no idea** how far away the receiver is
  - Hard to guess a reasonable length of time to wait
  - **SHOULD** (RFCs 1122 & 2988) use default of **3 seconds**
    - Some implementations instead use 6 seconds

# Tearing Down the Connection

# Normal Termination, One Side At A Time

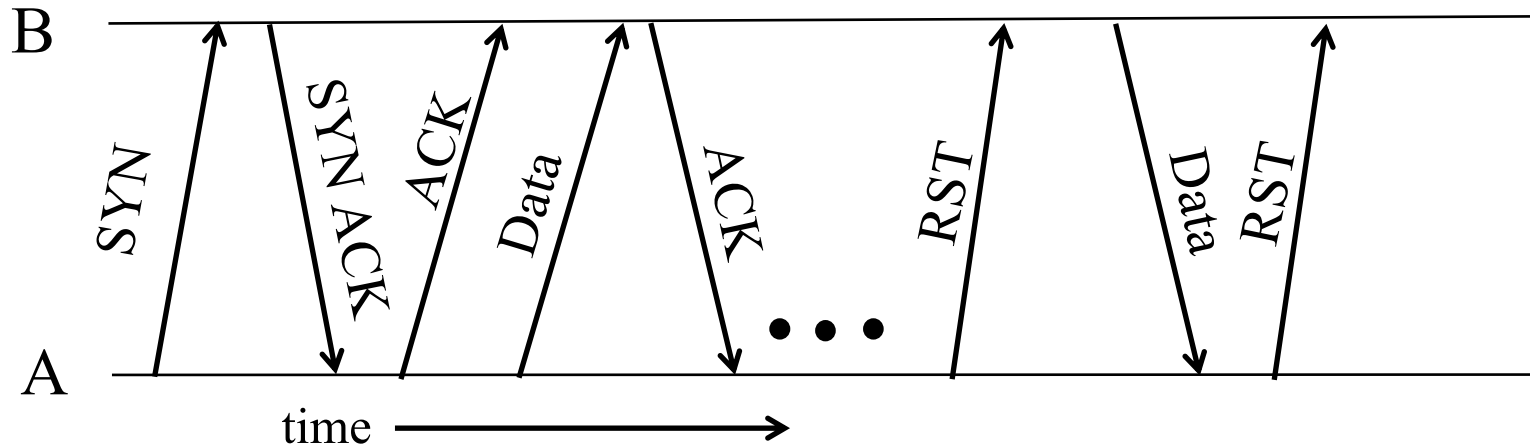


# Normal Termination, Both Together



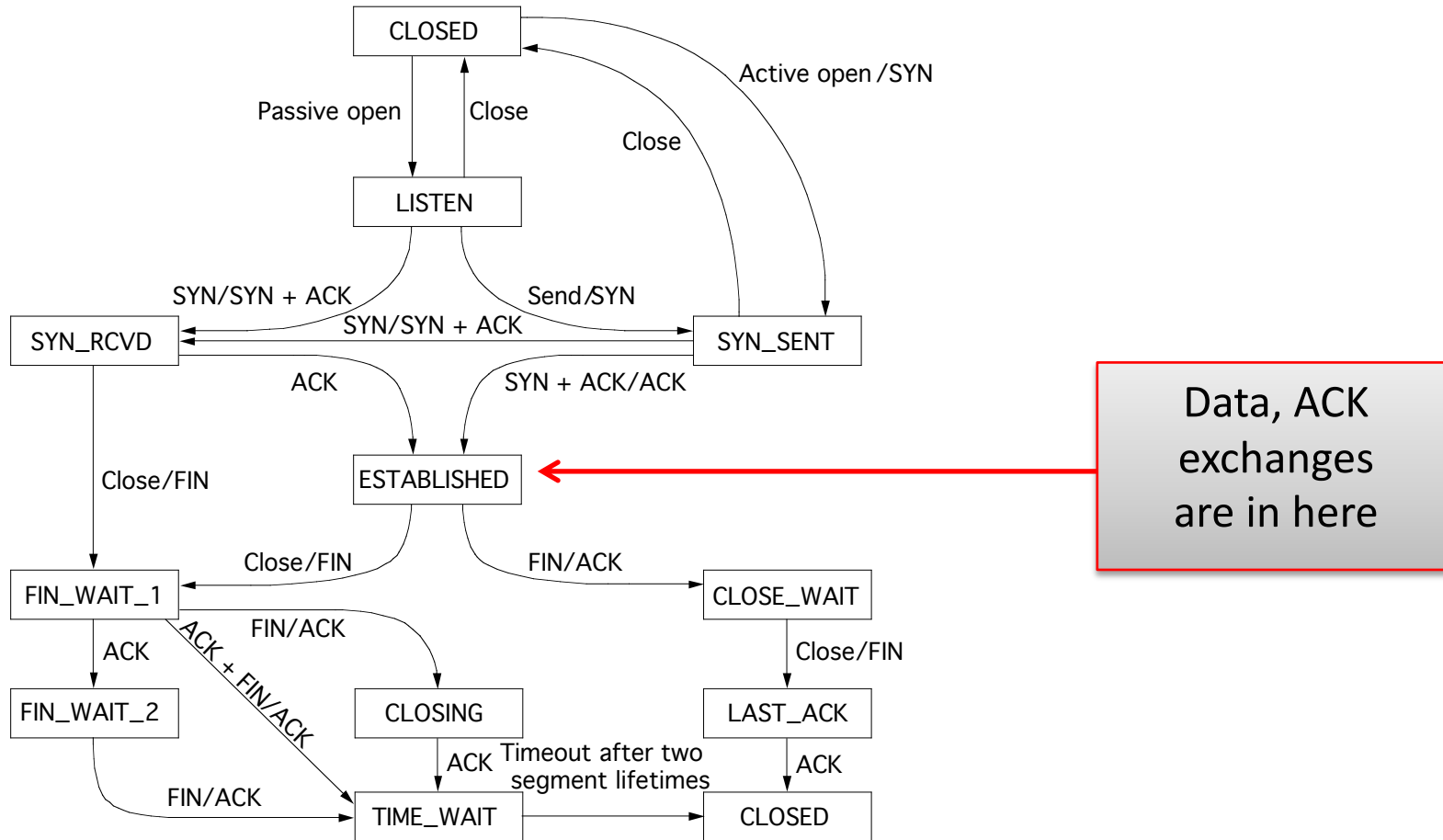
- Same as before, but B sets **FIN** with their ack of A's **FIN**

# Abrupt Termination

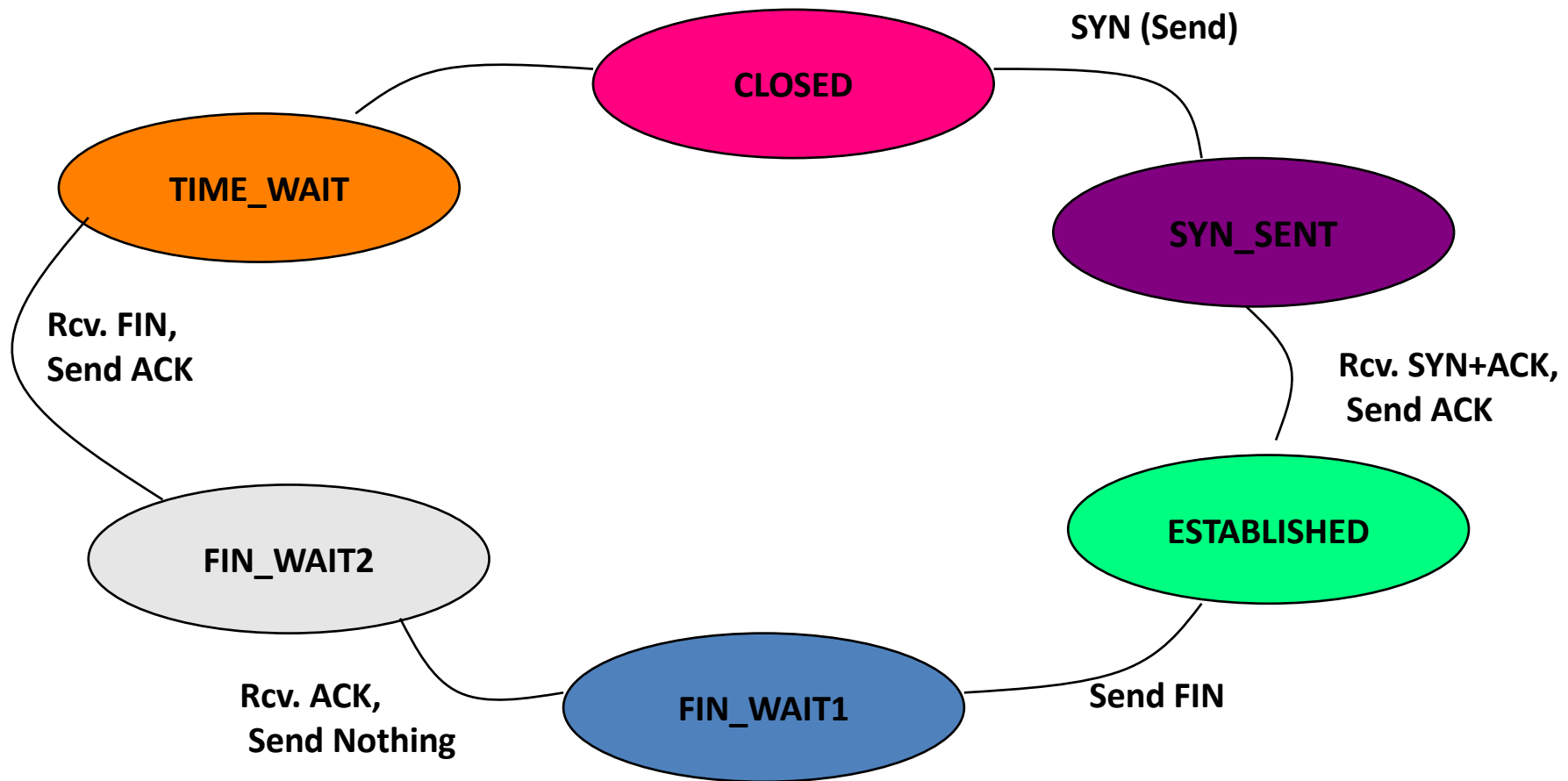


- A sends a RESET (**RST**) to B
  - E.g., because application process on A **crashed**
- **That's it**
  - B does **not** ack the **RST**
  - Thus, **RST** is **not** delivered **reliably**
  - And: any data in flight is **lost**
  - But: if B sends anything more, will elicit **another RST**

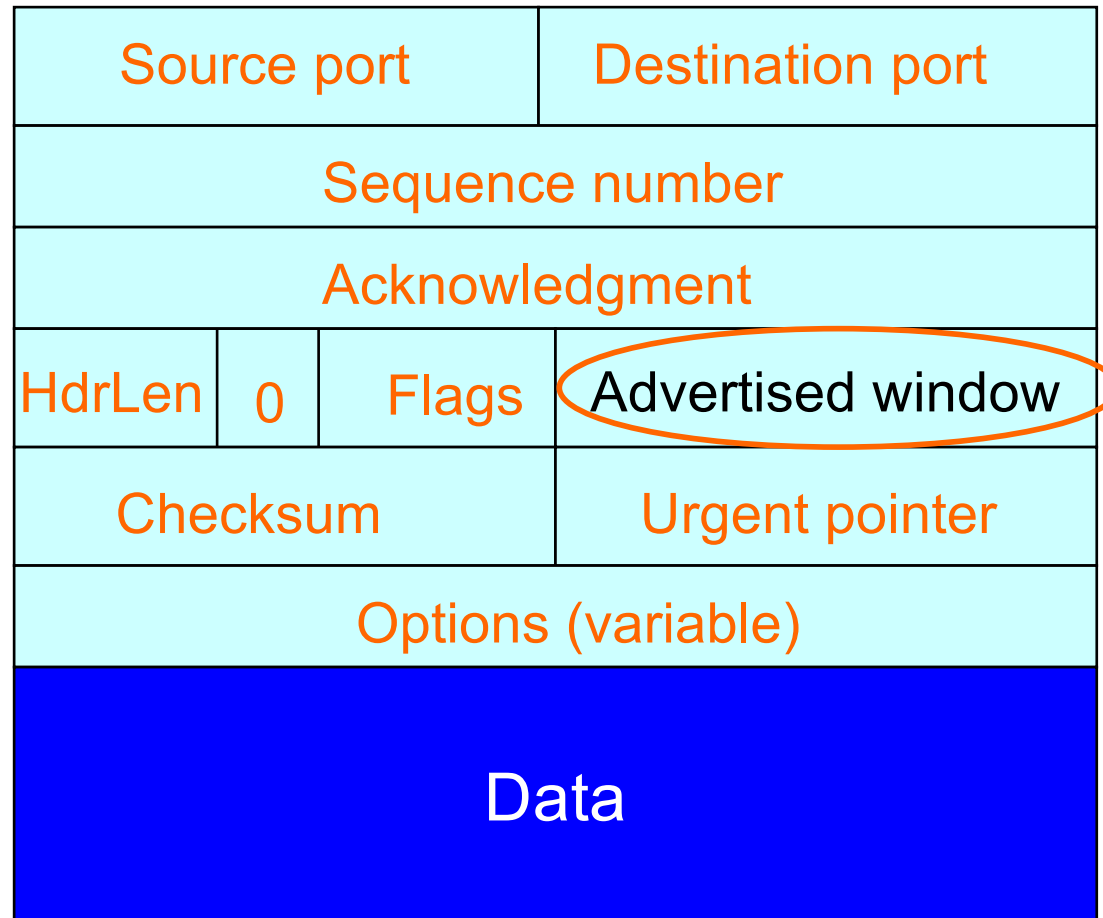
# TCP State Transitions



# An Simpler View of the Client Side



# TCP Header



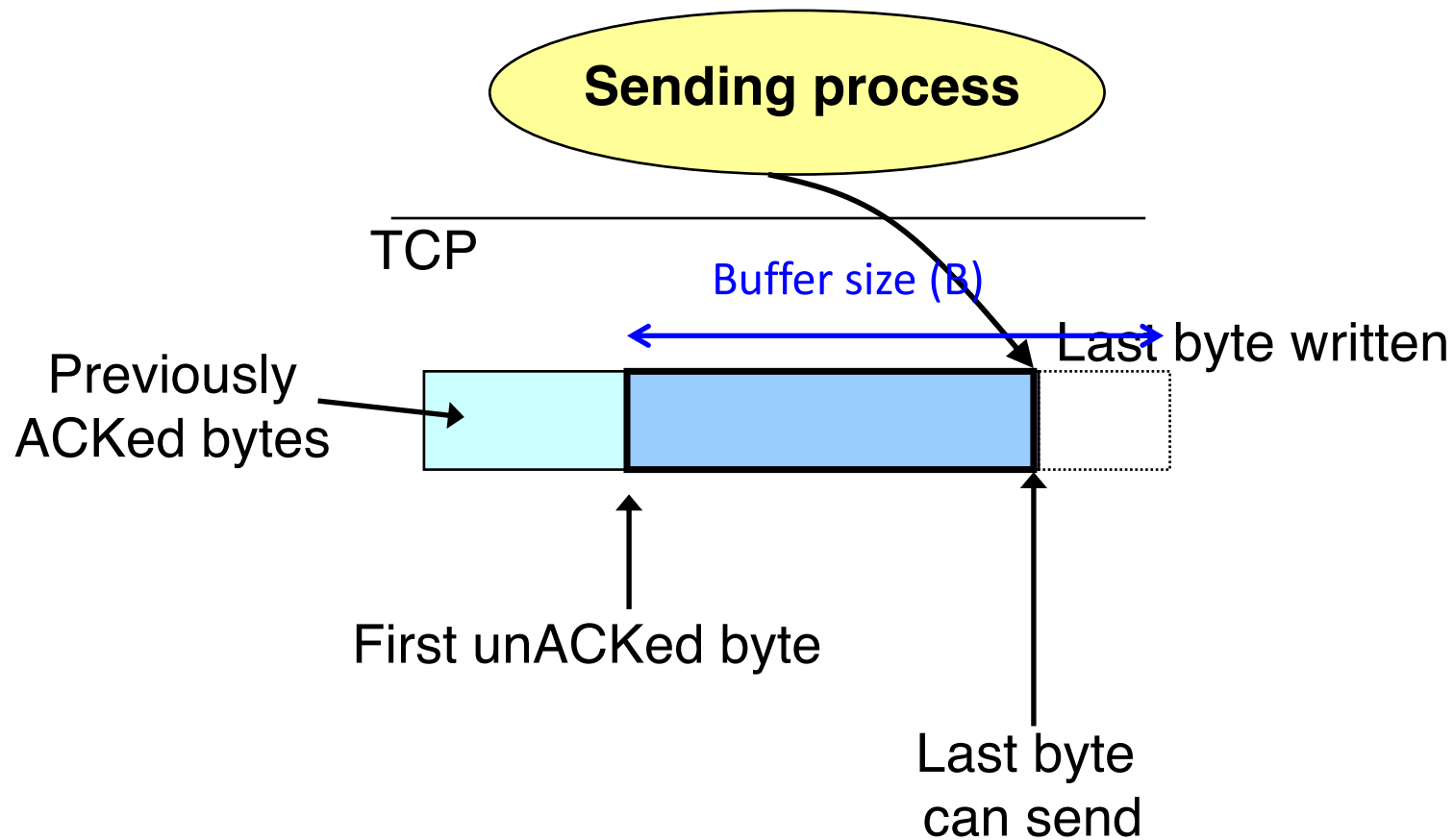


- What does TCP do?
  - ARQ windowing, set-up, tear-down
- Flow Control in TCP

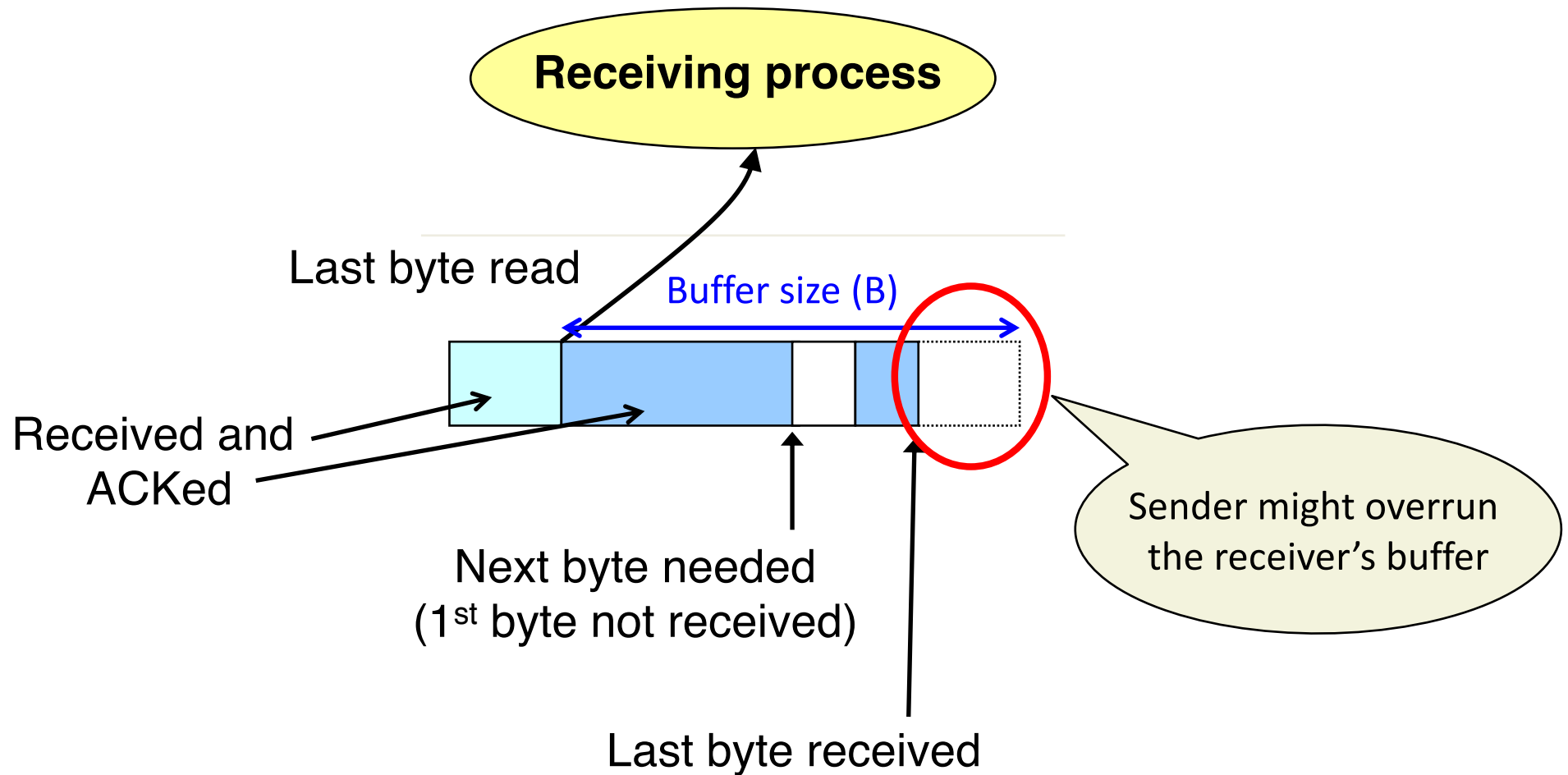
# Recap: Sliding Window (so far)

- Both sender & receiver maintain a **window**
- **Left edge** of window:
  - Sender: beginning of **unacknowledged** data
  - Receiver: beginning of **undelivered** data
- Right edge: Left edge + *constant*
  - constant only limited by buffer size in the transport layer

# Sliding Window at Sender (so far)



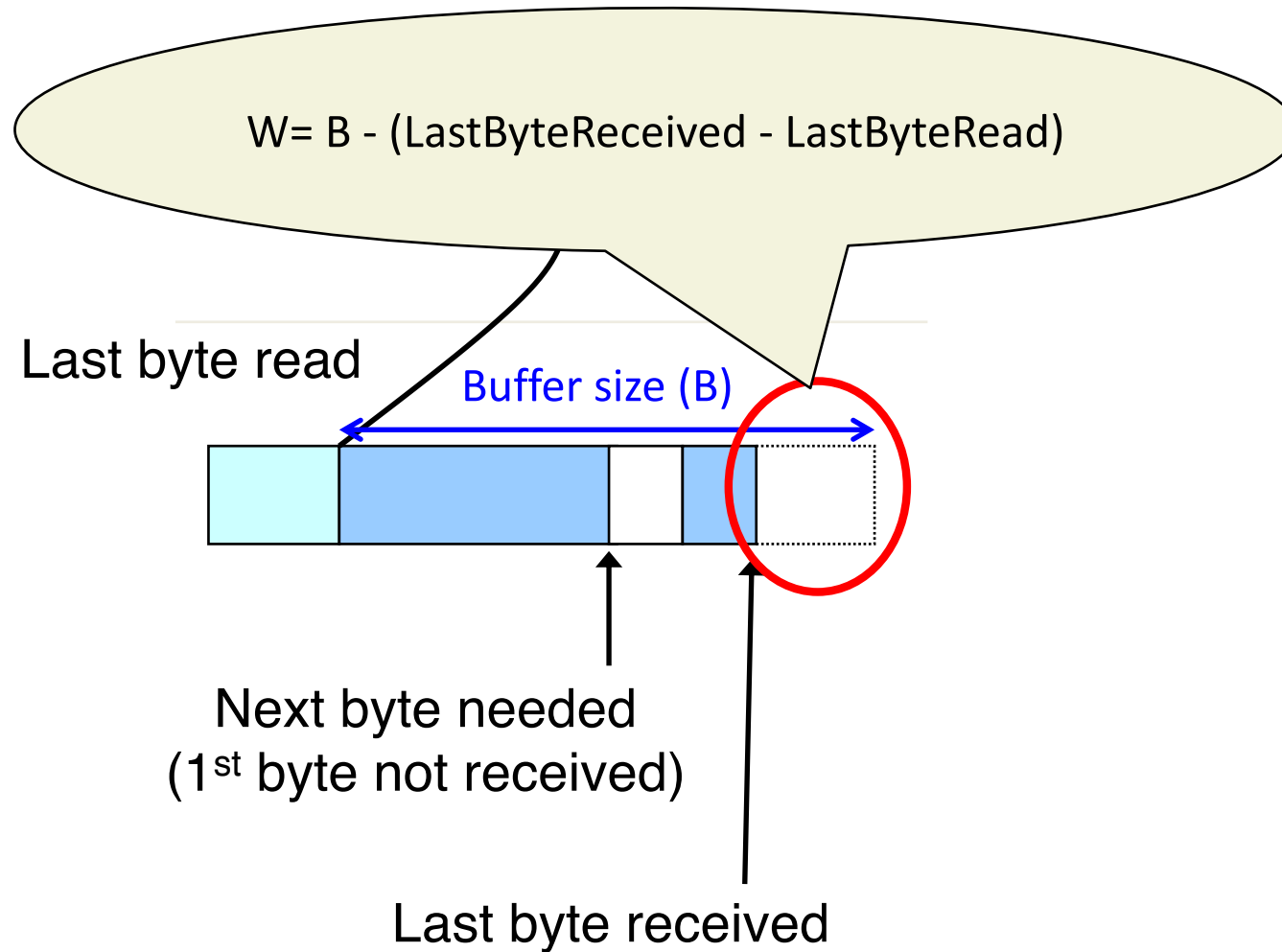
# Sliding Window at Receiver (so far)



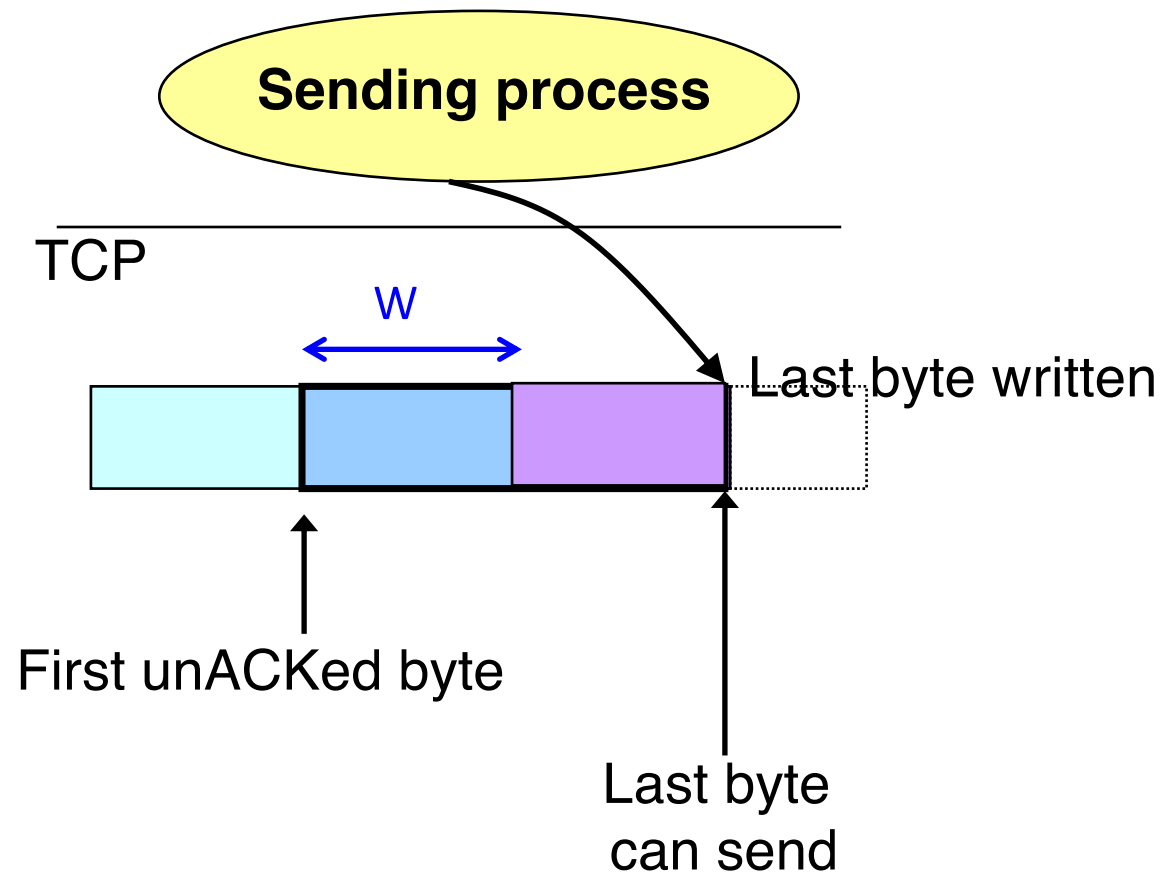
# Solution: Advertised Window (Flow Control)

- Receiver uses an “Advertised Window” ( $W$ ) to prevent sender from overflowing its window
  - Receiver indicates value of  $W$  in ACKs
  - Sender limits number of bytes it can have in flight  $\leq W$

# Sliding Window at Receiver



# Sliding Window at Sender (so far)



# Sliding Window w/ Flow Control

- Sender: window **advances** when new data ack'd
- Receiver: window advances as receiving process **consumes** data
- Receiver **advertises** to the sender where the receiver window currently ends (“righthand edge”)
  - Sender agrees not to exceed this amount



# Advertised Window Limits Rate

- Sender can send no faster than  $W/RTT$  bytes/sec
- Receiver only advertises more space when it has consumed old arriving data
- In original TCP design, that was the **sole** protocol mechanism controlling sender's rate
- What's missing?

# TCP

- The concepts underlying TCP are simple
  - acknowledgments (feedback)
  - timers
  - sliding windows
  - buffer management
  - sequence numbers

- What does TCP do?
  - ARQ windowing, set-up, tear-down
- Flow Control in TCP
- Congestion Control in TCP

## We have seen:

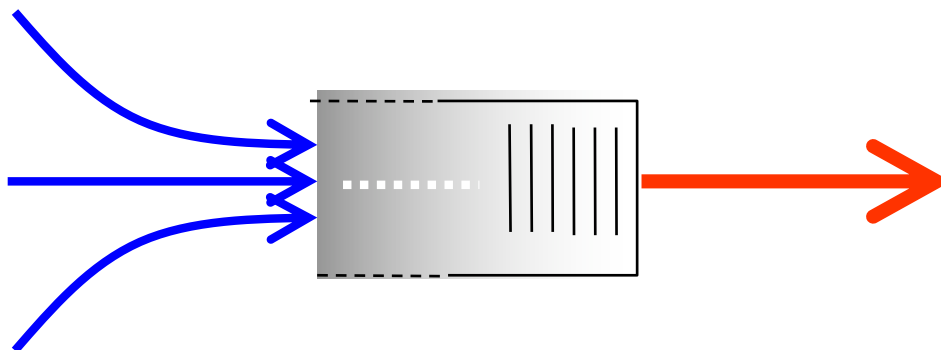
- **Flow control**: adjusting the sending rate to keep from overwhelming a slow *receiver*

## Now lets attend...

- **Congestion control**: adjusting the sending rate to keep from overloading the *network*

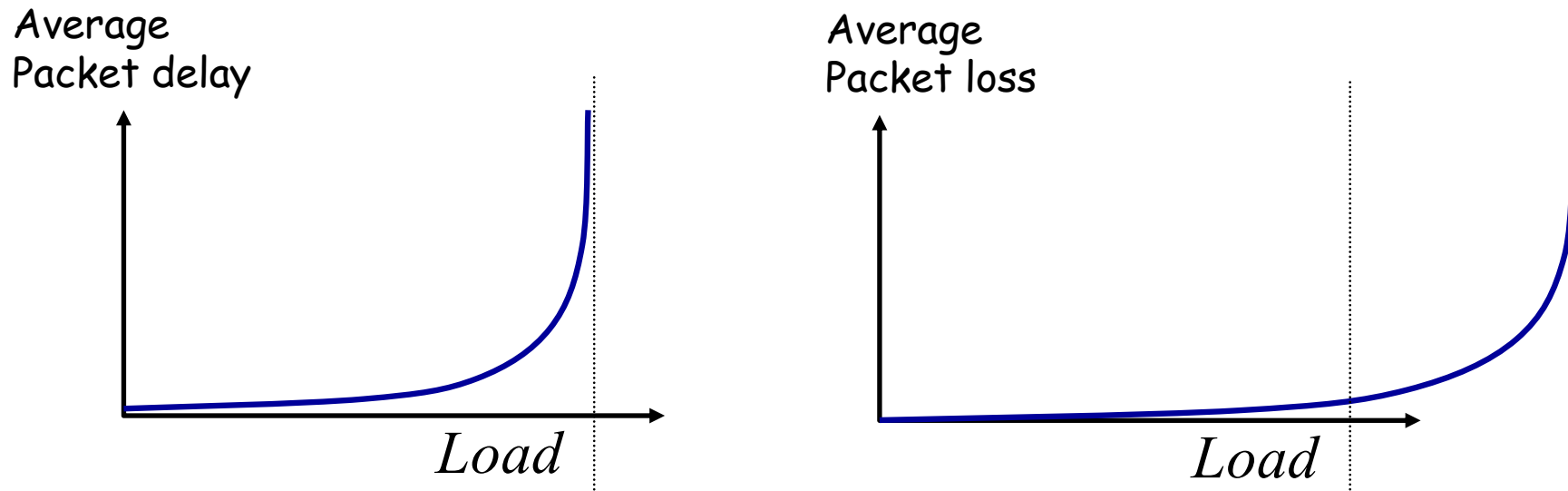
# Statistical Multiplexing → Congestion

- If two packets arrive at the same time
  - A router can only transmit one
  - ... and either buffers or drops the other
- If many packets arrive in a short period of time
  - The router cannot keep up with the arriving traffic
  - ... **delays** traffic, and the buffer may eventually **overflow**
- Internet traffic is **bursty**



# Congestion is undesirable

Typical **queuing system** with bursty arrivals



**Must balance utilization versus delay and loss**

# Who Takes Care of Congestion?

- Network? End hosts? Both?
- TCP's approach:
  - **End hosts** adjust sending rate
  - Based on **implicit feedback** from network
- Not the only approach
  - A consequence of history rather than planning

# Some History: TCP in the 1980s

- Sending rate only limited by flow control
  - Packet drops → senders (repeatedly!) retransmit a full window's worth of packets
- Led to “congestion collapse” starting Oct. 1986
  - Throughput on the NSF network dropped from 32Kbits/s to 40bits/sec
- “Fixed” by Van Jacobson's development of TCP's congestion control (CC) algorithms



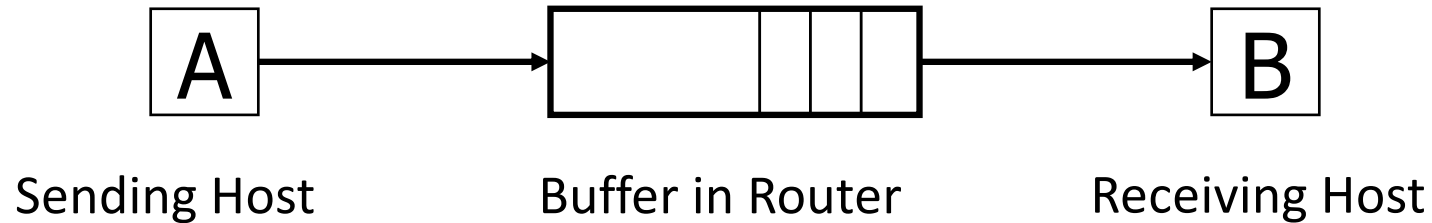
# Jacobson's Approach

- Extend TCP's existing window-based protocol but adapt the window size in response to congestion
  - required no upgrades to routers or applications!
  - patch of a few lines of code to TCP implementations
- A pragmatic and effective solution
  - but many other approaches exist
- Extensively improved on since
  - topic now sees less activity in ISP contexts
  - but is making a comeback in datacenter environments

# Three Issues to Consider

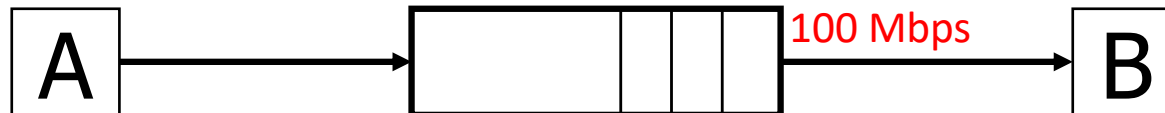
- Discovering the available (bottleneck) bandwidth
- Adjusting to variations in bandwidth
- Sharing bandwidth between flows

# Abstract View



- Ignore internal structure of router and model it as having a single queue for a particular input-output pair

# Discovering available bandwidth



- Pick sending rate to match bottleneck bandwidth
  - Without any *a priori* knowledge
  - Could be gigabit link, could be a modem

# Adjusting to variations in bandwidth

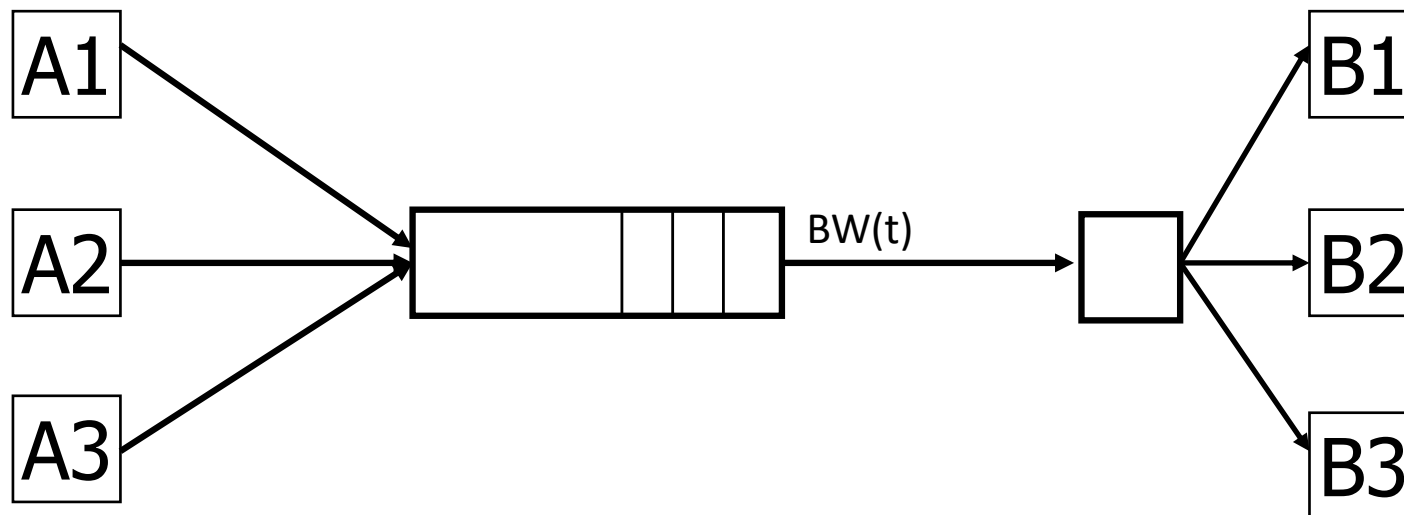


- Adjust rate to match **instantaneous** bandwidth
  - Assuming you have rough idea of bandwidth

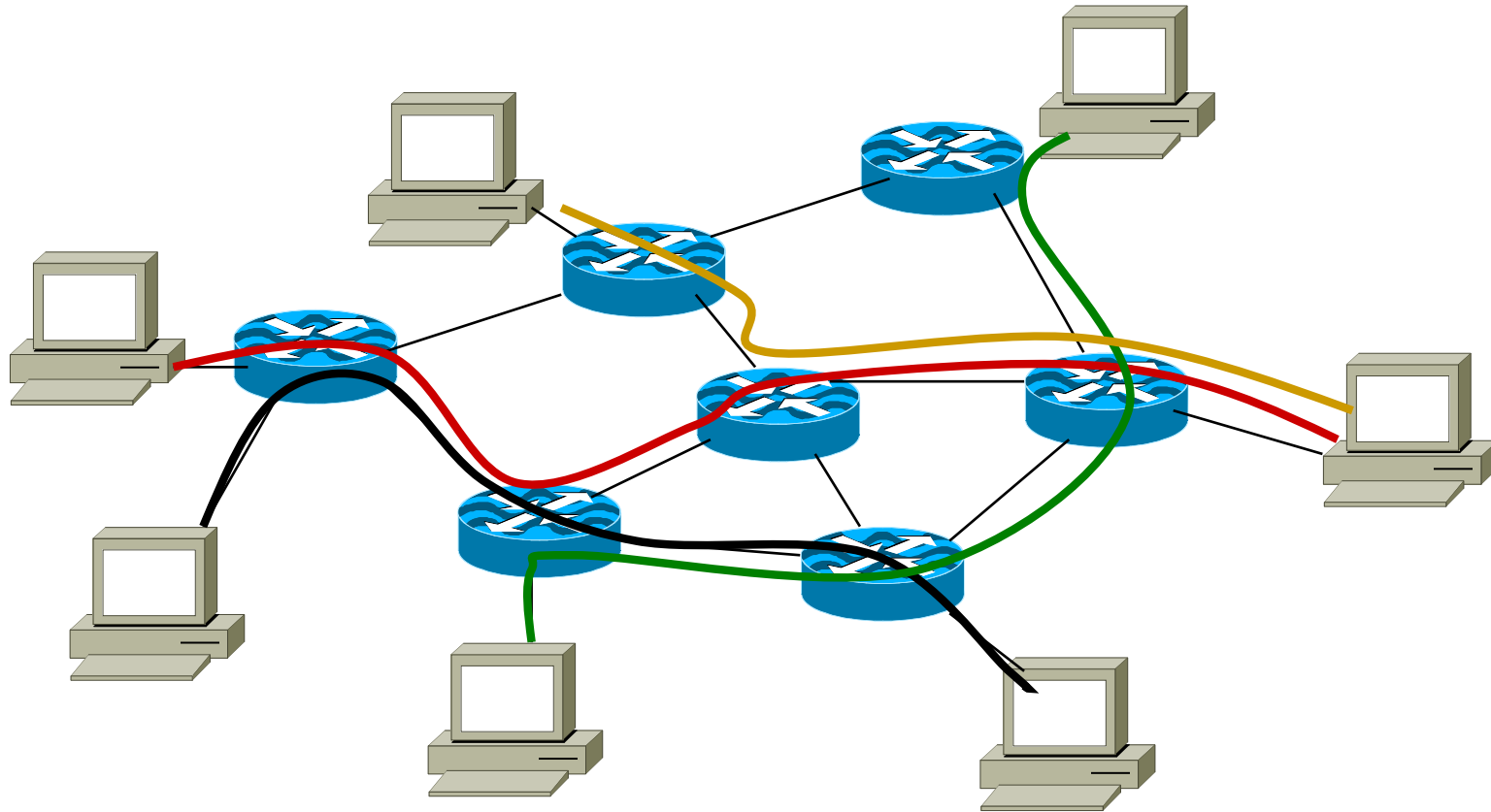
# Multiple flows and sharing bandwidth

Two Issues:

- Adjust total sending rate to match bandwidth
- Allocation of bandwidth between flows



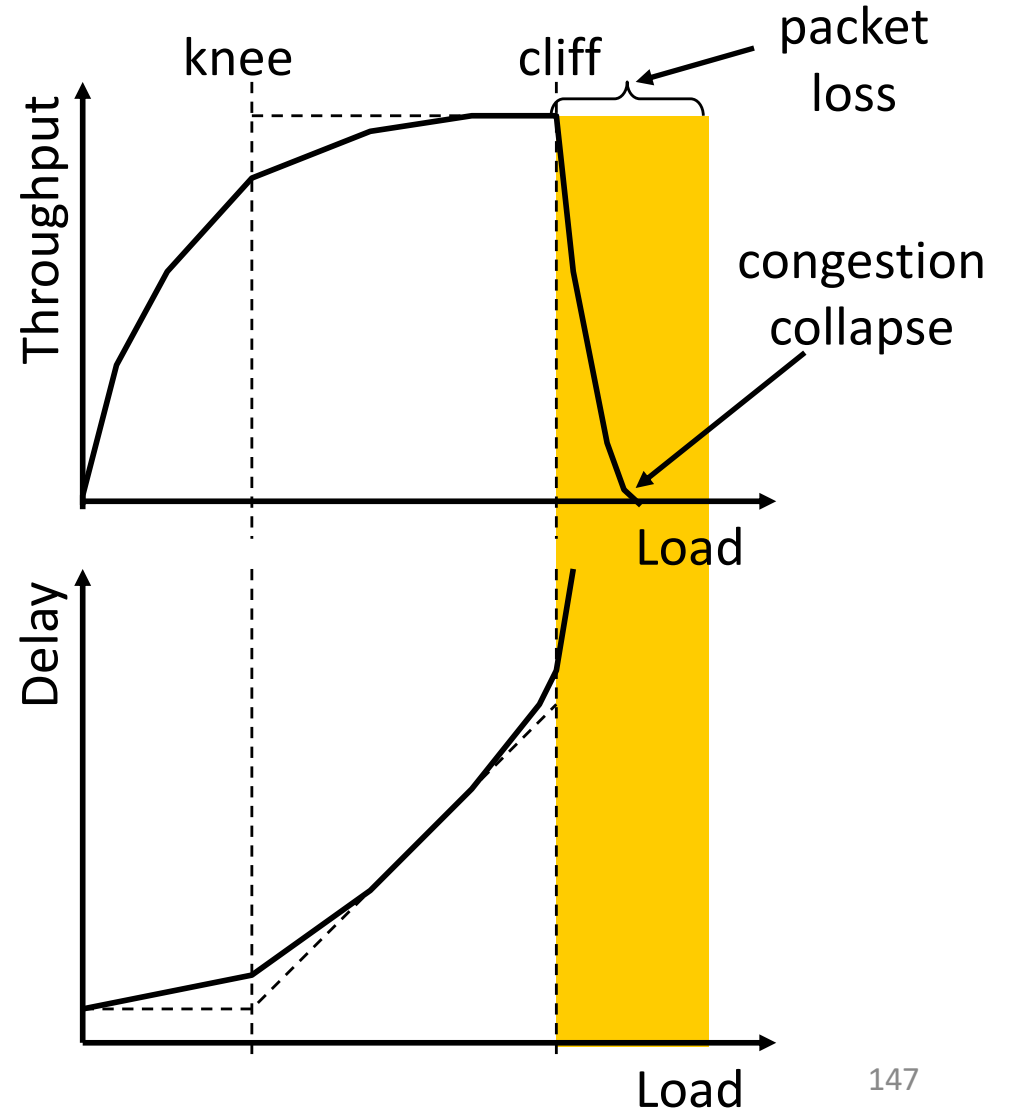
# Reality



Congestion control is a resource allocation problem involving many flows, many links, and complicated global dynamics

# View from a single flow

- Knee – point after which
  - Throughput increases slowly
  - Delay increases fast
- Cliff – point after which
  - Throughput starts to drop to zero (congestion collapse)
  - Delay approaches infinity





# General Approaches

- (0) Send without care
  - Many packet drops

# General Approaches

(0) Send without care

(1) Reservations

- Pre-arrange bandwidth allocations
- Requires negotiation before sending packets
- Low utilization

# General Approaches

(0) Send without care

(1) Reservations

(2) Pricing

- Don't drop packets for the high-bidders
- Requires payment model

# General Approaches

(0) Send without care

(1) Reservations

(2) Pricing

(3) Dynamic Adjustment

- Hosts probe network; infer level of congestion; adjust
- Network reports congestion level to hosts; hosts adjust
- Combinations of the above
- Simple to implement but suboptimal, messy dynamics

# General Approaches

- (0) Send without care
- (1) Reservations
- (2) Pricing
- (3) Dynamic Adjustment

## All three techniques have their place

- *Generality* of dynamic adjustment has proven powerful
- Doesn't presume business model, traffic characteristics, application requirements; does assume good citizenship

# TCP's Approach in a Nutshell

- TCP connection has window
  - Controls number of packets in flight
- Sending rate:  $\sim \text{Window}/\text{RTT}$
- Vary window size to control sending rate

# All These Windows...

- Congestion Window: **CWND**
  - How many bytes can be sent without overflowing routers
  - Computed by the sender using congestion control algorithm
- Flow control window: **AdvertisedWindow (RWND)**
  - How many bytes can be sent without overflowing receiver's buffers
  - Determined by the receiver and reported to the sender
- Sender-side window = **minimum**{**CWND**,**RWND**}
  - Assume for this material that  $RWND \gg CWND$

# Note

- This lecture will talk about CWND in units of MSS
  - (Recall MSS: Maximum Segment Size, the amount of payload data in a TCP packet)
  - This is only for pedagogical purposes
- **In reality this is a LIE:** Real implementations maintain CWND in bytes



# Two Basic Questions

- How does the sender detect congestion?
- How does the sender adjust its sending rate?
  - To address three issues
    - Finding available bottleneck bandwidth
    - Adjusting to bandwidth variations
    - Sharing bandwidth

# Detecting Congestion

- Packet delays
  - Tricky: noisy signal (delay often varies considerably)
- Router tell endhosts they're congested
- Packet loss
  - Fail-safe signal that TCP already has to detect
  - Complication: non-congestive loss (checksum errors)
- Two indicators of packet loss
  - No ACK after certain time interval: **timeout**
  - Multiple **duplicate ACKs**

# Not All Losses the Same

- Duplicate ACKs: isolated loss
  - Still getting ACKs
- Timeout: much more serious
  - Not enough dupacks
  - Must have suffered several losses
- We will adjust rate differently for each case

# Rate Adjustment

- Basic structure:
  - Upon receipt of ACK (of new data): increase rate
  - Upon detection of loss: decrease rate
- How we increase/decrease the rate depends on the phase of congestion control we're in:
  - Discovering available bottleneck bandwidth vs.
  - Adjusting to bandwidth variations

# Bandwidth Discovery with Slow Start

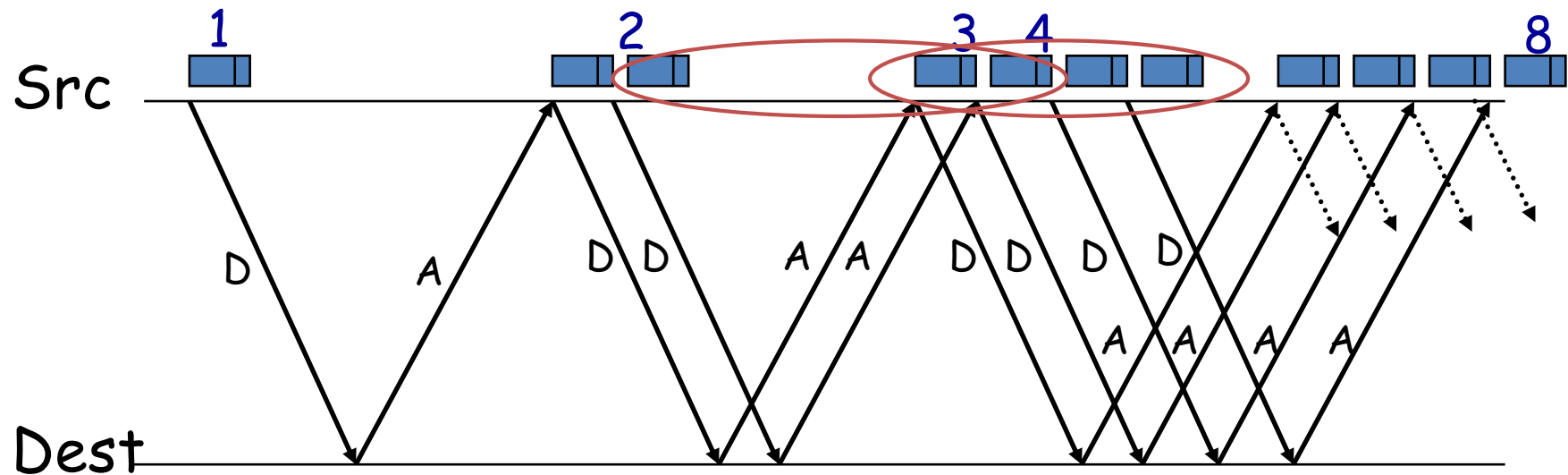
- Goal: estimate available bandwidth
  - start slow (for safety)
  - but ramp up quickly (for efficiency)
- Consider
  - RTT = 100ms, MSS=1000bytes
  - Window size to fill 1Mbps of BW = 12.5 packets
  - Window size to fill 1Gbps = 12,500 packets
  - Either is possible!

# “Slow Start” Phase

- Sender starts at a slow rate but increases **exponentially** until first loss
- Start with a small congestion window
  - Initially,  $CWND = 1$
  - So, initial sending rate is  $MSS/RTT$
- Double the  $CWND$  for each  $RTT$  with no loss

# Slow Start in Action

- For each RTT: double CWND
- Simpler implementation: for each ACK, CWND += 1



# Adjusting to Varying Bandwidth

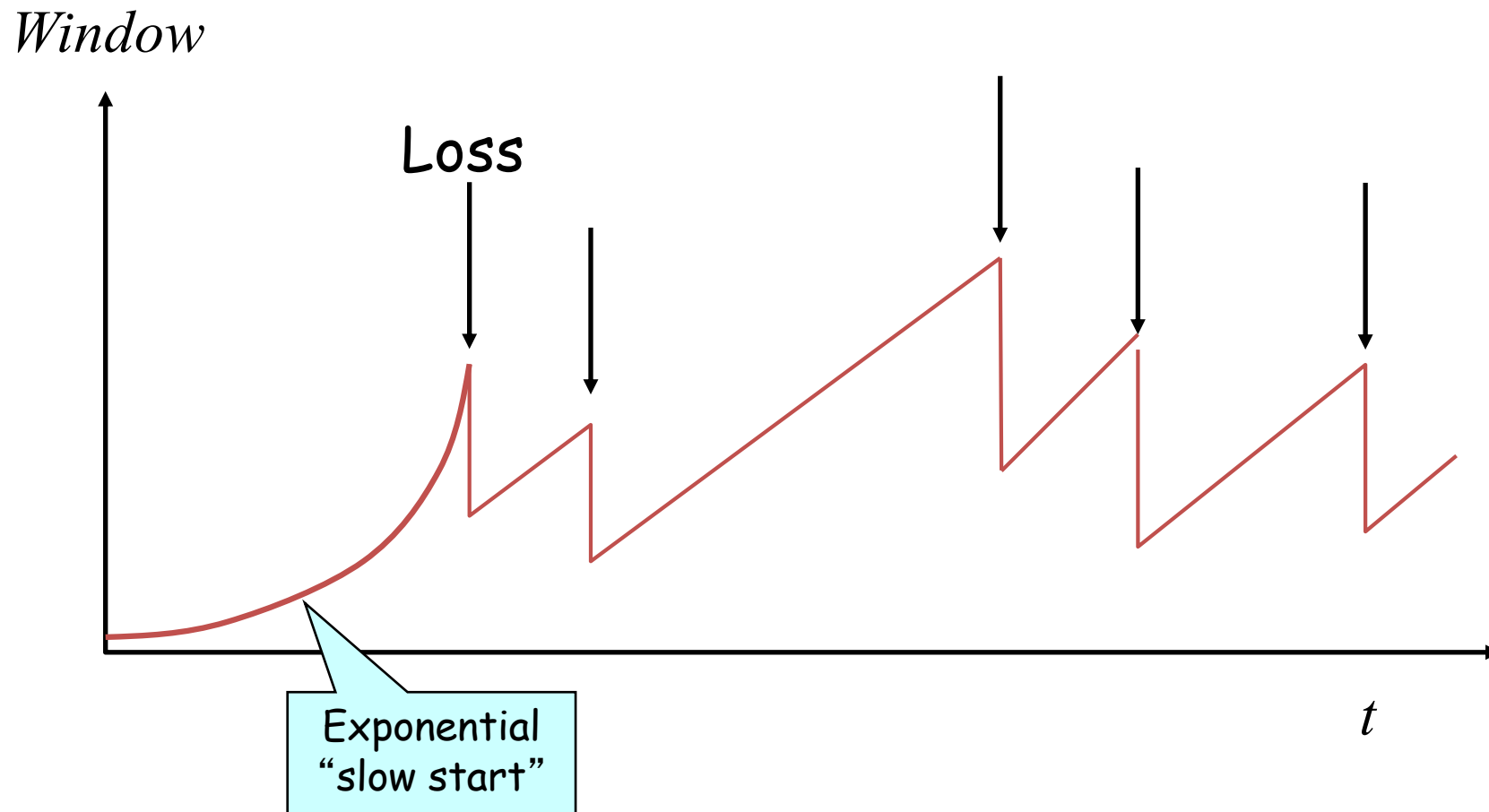
- Slow start gave an estimate of available bandwidth
- Now, want to track variations in this available bandwidth, oscillating around its current value
  - Repeated probing (rate increase) and backoff (rate decrease)
- TCP uses: “Additive Increase Multiplicative Decrease” (AIMD)
  - We’ll see why shortly...



# AIMD

- Additive increase
  - Window grows by one MSS for every RTT with no loss
  - For each successful RTT,  $CWND = CWND + 1$
  - Simple implementation:
    - for each ACK,  $CWND = CWND + 1/CWND$
- Multiplicative decrease
  - On loss of packet, divide congestion window in **half**
  - On loss,  $CWND = CWND/2$

# Leads to the TCP “Sawtooth”



# Slow-Start vs. AIMD

- When does a sender stop Slow-Start and start Additive Increase?
- Introduce a “slow start threshold” (**ssthresh**)
  - Initialized to a large value
  - On timeout,  $ssthresh = CWND/2$
- When  $CWND = ssthresh$ , sender switches from slow-start to AIMD-style increase

- What does TCP do?
  - ARQ windowing, set-up, tear-down
- Flow Control in TCP
- Congestion Control in TCP
  - AIMD

- What does TCP do?
  - ARQ windowing, set-up, tear-down
- Flow Control in TCP
- Congestion Control in TCP
  - AIMD, Fast-Recovery

# One Final Phase: Fast Recovery

- The problem: congestion avoidance too slow in recovering from an isolated loss

# Example (in units of MSS, not bytes)

- Consider a TCP connection with:
  - CWND=10 packets
  - Last ACK was for packet # 101
    - i.e., receiver expecting next packet to have seq. no. 101
- 10 packets [101, 102, 103,..., 110] are in flight
  - Packet 101 is dropped
  - What ACKs do they generate?
  - And how does the sender respond?

# The problem – A timeline

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)
- ACK 101 (due to 104) cwnd=10 dupACK#3 (no xmit)
- RETRANSMIT 101 ssthresh=5 cwnd= 5
- ACK 101 (due to 105) cwnd=5 + 1/5 (no xmit)
- ACK 101 (due to 106) cwnd=5 + 2/5 (no xmit)
- ACK 101 (due to 107) cwnd=5 + 3/5 (no xmit)
- ACK 101 (due to 108) cwnd=5 + 4/5 (no xmit)
- ACK 101 (due to 109) cwnd=5 + 5/5 (no xmit)
- ACK 101 (due to 110) cwnd=6 + 1/5 (no xmit)
- ACK 111 (due to 101) ← only now can we transmit new packets
- Plus no packets in flight so ACK “clocking” (to increase CWND) stalls for another RTT



# Solution: Fast Recovery

Idea: Grant the sender temporary “credit” for each dupACK so as to keep packets in flight

- If dupACKcount = 3
  - ssthresh = cwnd/2
  - cwnd = ssthresh + 3
- While in fast recovery
  - cwnd = cwnd + 1 for each additional duplicate ACK
- Exit fast recovery after receiving new ACK
  - set cwnd = ssthresh

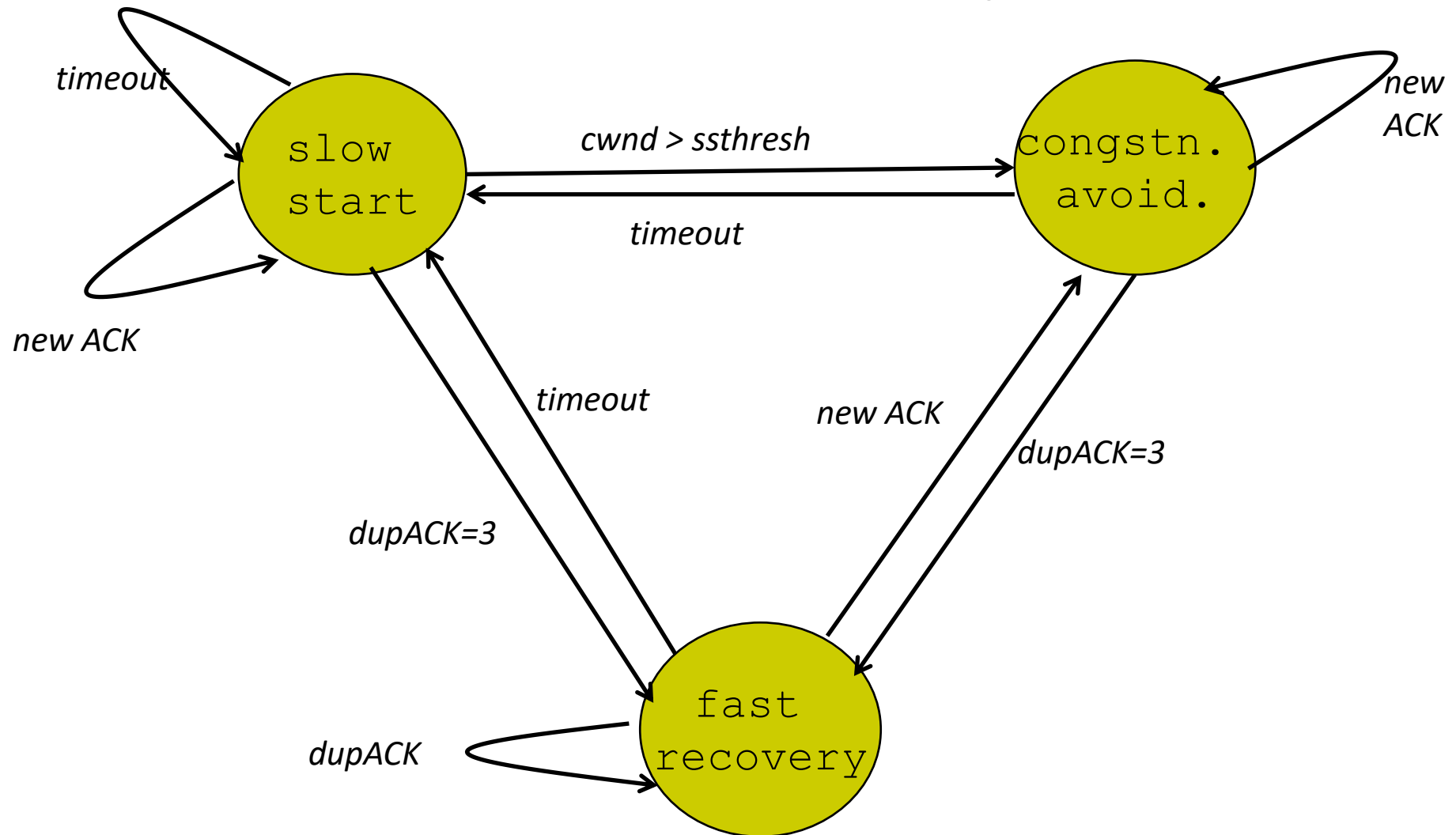
# Example

- Consider a TCP connection with:
  - CWND=10 packets
  - Last ACK was for packet # 101
    - i.e., receiver expecting next packet to have seq. no. 101
- 10 packets [101, 102, 103,..., 110] are in flight
  - Packet 101 is dropped

# Timeline

- ACK 101 (due to 102) cwnd=10 dup#1
- ACK 101 (due to 103) cwnd=10 dup#2
- ACK 101 (due to 104) cwnd=10 dup#3
- REXMIT 101 ssthresh=5 cwnd= 8 (5+3)
- ACK 101 (due to 105) cwnd= 9 (no xmit)
- ACK 101 (due to 106) cwnd=10 (no xmit)
- ACK 101 (due to 107) cwnd=11 (xmit 111)
- ACK 101 (due to 108) cwnd=12 (xmit 112)
- ACK 101 (due to 109) cwnd=13 (xmit 113)
- ACK 101 (due to 110) cwnd=14 (xmit 114)
- ACK 111 (due to 101) cwnd = 5 (xmit 115) ← exiting fast recovery
- Packets 111-114 already in flight
- ACK 112 (due to 111) cwnd =  $5 + 1/5$  ← back in congestion avoidance

# Putting it all together: The TCP State Machine (partial)



- How are *ssthresh*, *CWND* and *dupACKcount* updated for each event that causes a state transition?

# TCP Flavors

- TCP-Tahoe
  - $\text{cwnd} = 1$  on triple dupACK
- TCP-Reno
  - $\text{cwnd} = 1$  on timeout
  - $\text{cwnd} = \text{cwnd}/2$  on triple dupack
- TCP-newReno
  - TCP-Reno + improved fast recovery
- TCP-SACK
  - incorporates selective acknowledgements

- What does TCP do?
  - ARQ windowing, set-up, tear-down
- Flow Control in TCP
- Congestion Control in TCP
  - AIMD, Fast-Recovery, Throughput

# TCP Flavors

- TCP-Tahoe
  - $CWND = 1$  on triple dupACK
- TCP-Reno
  - $CWND = 1$  on timeout
  - $CWND = CWND/2$  on triple dupack
- TCP-newReno
  - TCP-Reno + improved fast recovery
- TCP-SACK
  - incorporates selective acknowledgements



Our default assumption

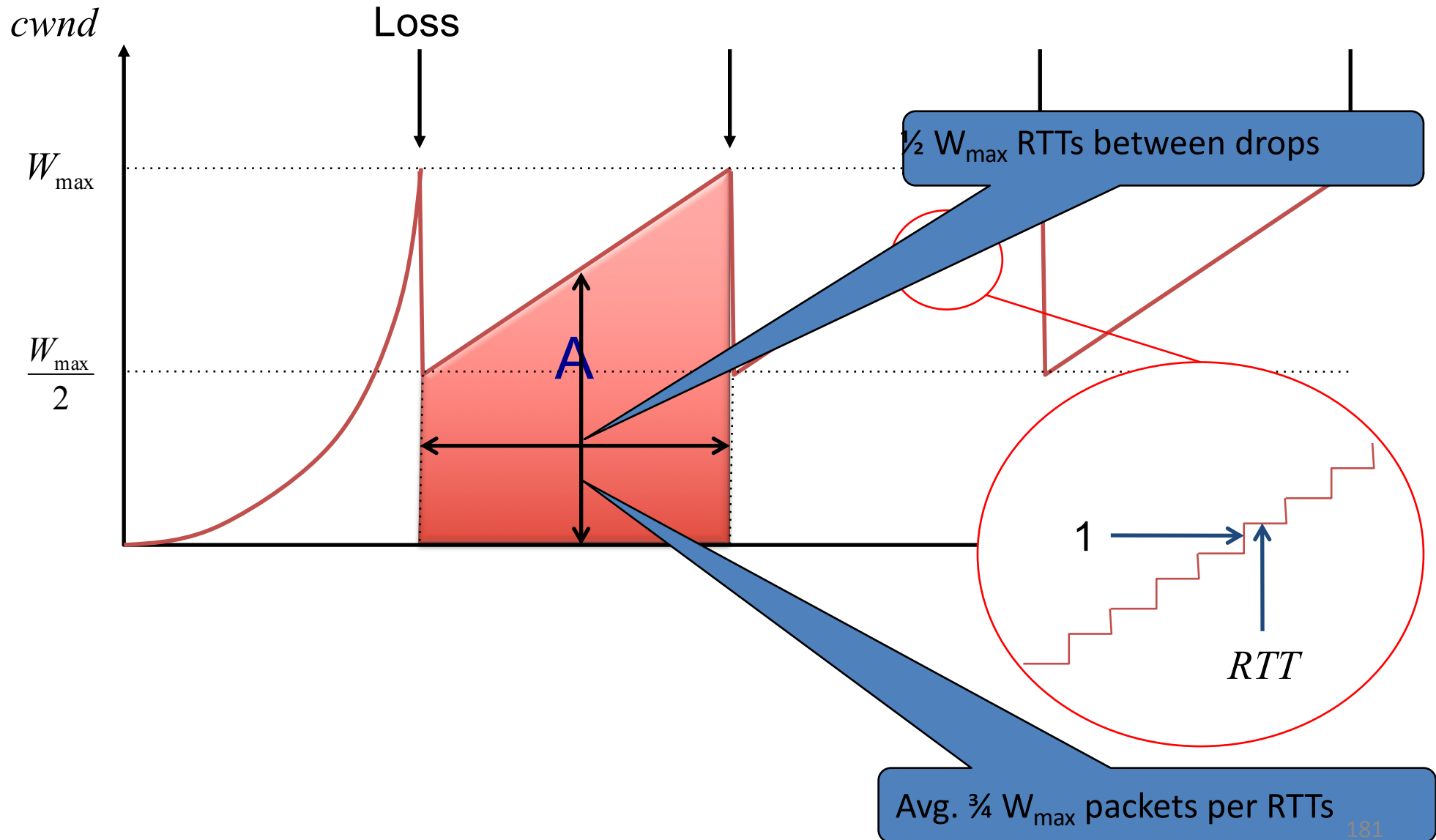
# Interoperability

- How can all these algorithms coexist? Don't we need a single, uniform standard?
- What happens if I'm using Reno and you are using Tahoe, and we try to communicate?

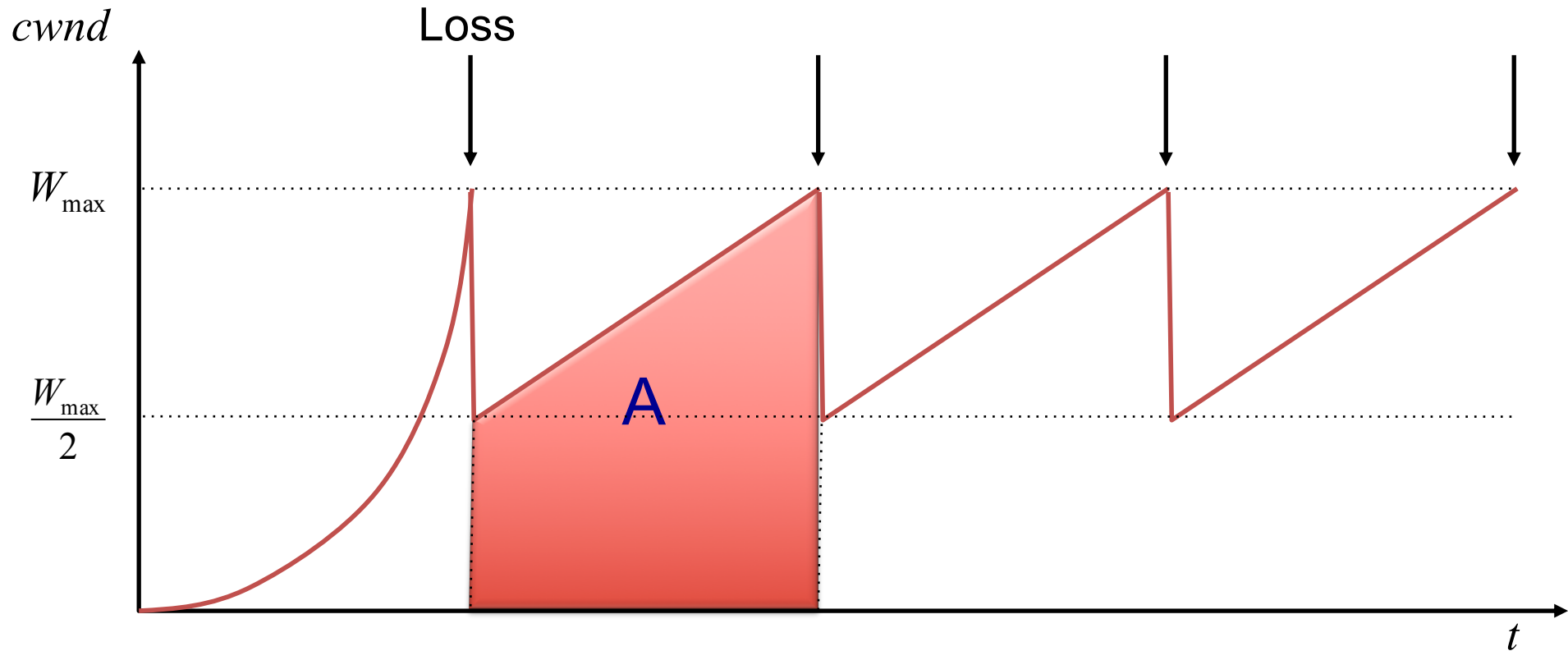


# TCP Throughput Equation

# A Simple Model for TCP Throughput



# A Simple Model for TCP Throughput



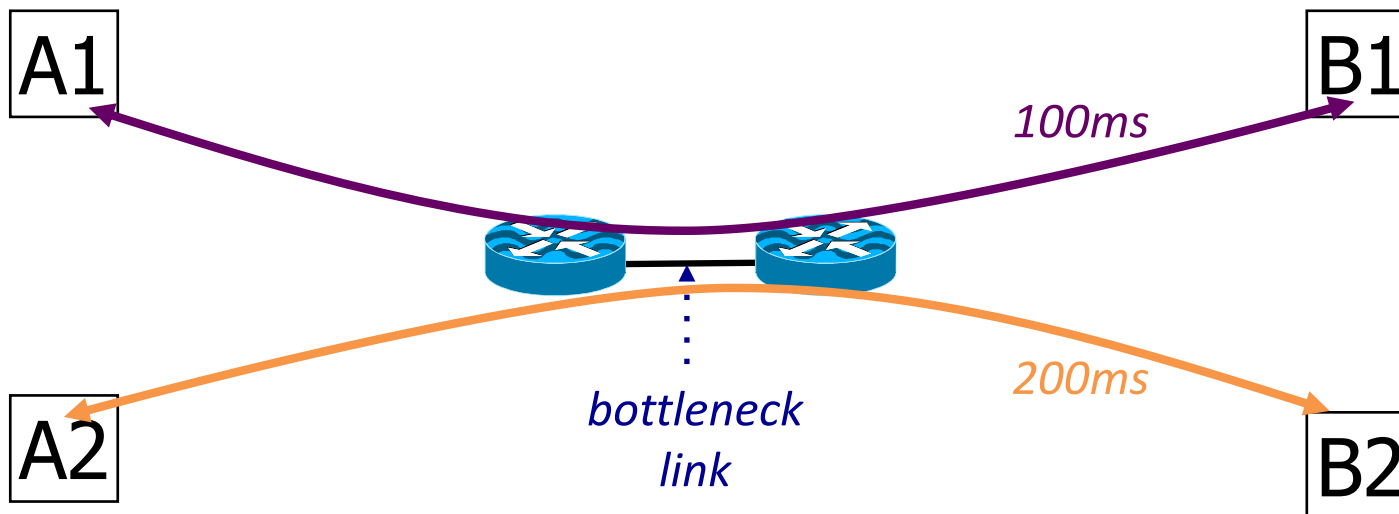
Packet drop rate,  $p = 1 / A$ , where  $A = \frac{3}{8} W_{\max}^2$

$$\text{Throughput, } B = \frac{A}{\left(\frac{W_{\max}}{2}\right) RTT} = \sqrt{\frac{3}{2}} \frac{1}{RTT \sqrt{p}}$$

# Implications (1): Different RTTs

$$\text{Throughput} = \sqrt{\frac{3}{2}} \frac{1}{RTT \sqrt{p}}$$

- Flows get throughput inversely proportional to RTT
- TCP unfair in the face of heterogeneous RTTs!



## Implications (2): High Speed TCP

$$\text{Throughput} = \sqrt{\frac{3}{2}} \frac{1}{RTT \sqrt{p}}$$

- Assume  $RTT = 100\text{ms}$ ,  $MSS=1500\text{bytes}$
- What value of  $p$  is required to reach 100Gbps throughput
  - $\sim 2 \times 10^{-12}$
- How long between drops?
  - $\sim 16.6$  hours
- How much data has been sent in this time?
  - $\sim 6$  petabits
- These are not practical numbers!

# Adapting TCP to High Speed

- Once past a threshold speed, increase CWND faster
  - A proposed standard [Floyd'03]: once speed is past some threshold, change equation to  $p^{-.8}$  rather than  $p^{-.5}$
  - Let the additive constant in AIMD depend on CWND
- Other approaches?
  - Multiple simultaneous connections (hack but works today)
  - Router-assisted approaches (will see shortly)

# Implications (3): *Rate*-based CC

$$\text{Throughput} = \sqrt{\frac{3}{2}} \frac{1}{RTT \sqrt{p}}$$

- TCP throughput is “choppy”
  - repeated swings between  $W/2$  to  $W$
- Some apps would prefer sending at a steady rate
  - e.g., streaming apps
- **A solution: “Equation-Based Congestion Control”**
  - ditch TCP’s increase/decrease rules and just follow the equation
  - measure drop percentage  $p$ , and set rate accordingly
- Following the TCP equation ensures we’re “TCP friendly”
  - i.e., use no more than TCP does in similar setting

# Recap: TCP problems

- Misled by non-congestion losses
- Fills up queues leading to high delays
- Short flows complete before discovering available capacity
- AIMD impractical for high speed links
- Sawtooth discovery too choppy for some apps
- Unfair under heterogeneous RTTs
- Tight coupling with reliability mechanisms
- Endhosts can cheat

Routers tell endpoints if they're congested

Routers tell endpoints what rate to send at

Routers enforce fair sharing

Could fix many of these with some help from routers!



# Router-Assisted Congestion Control

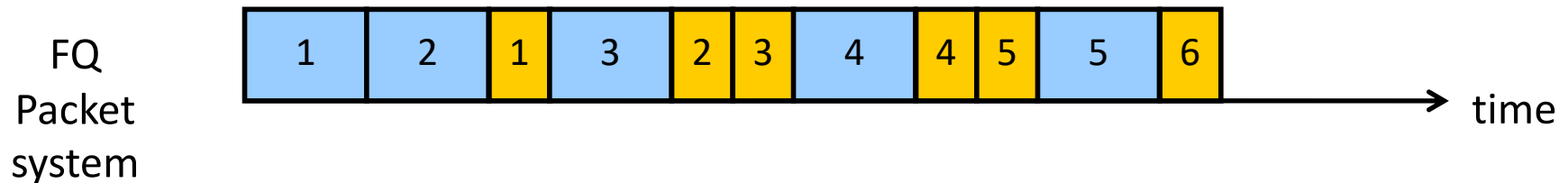
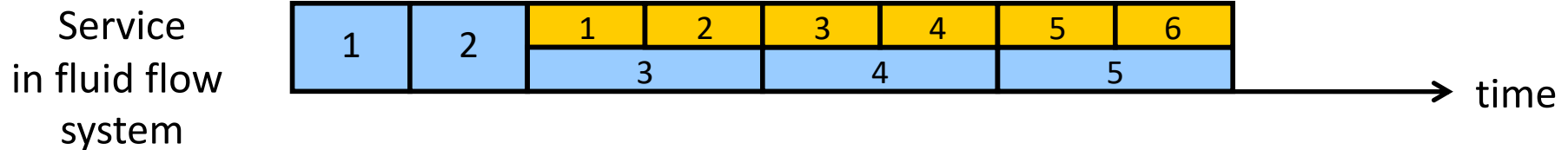
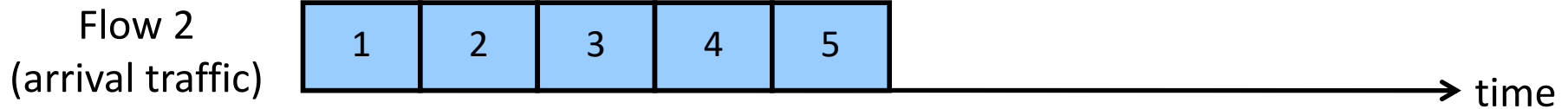
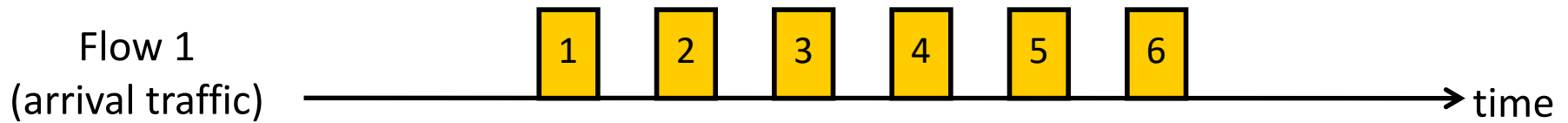
- Three tasks for CC:
  - Isolation/fairness
  - Adjustment\*
  - Detecting congestion

\* This may be *automatic* eg loss-response of TCP

# Fair Queuing (FQ)

- For each packet, compute the time at which the last bit of a packet would have left the router *if* flows are served bit-by-bit
- Then serve packets in the increasing order of their deadlines

# Example



# Fair Queuing (FQ)

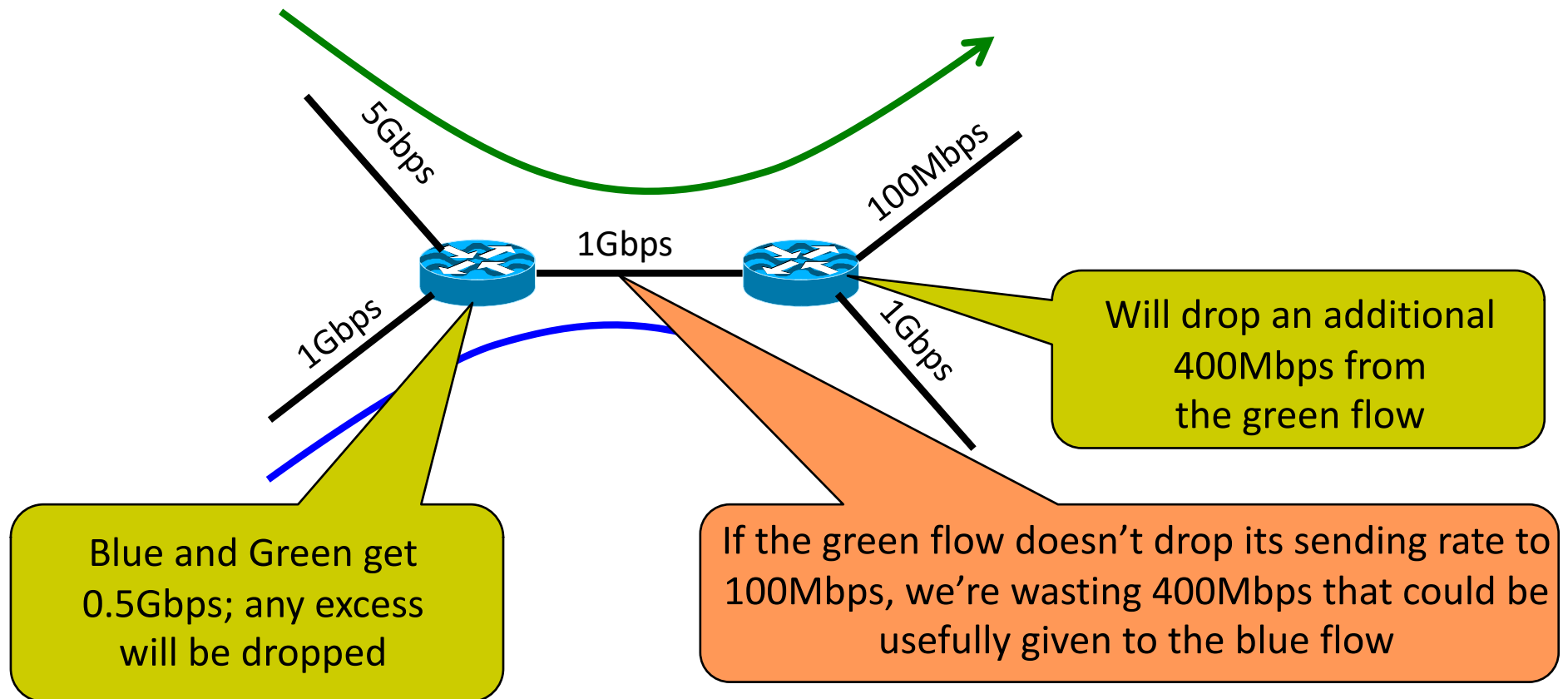
- Think of it as an implementation of round-robin generalized to the case where not all packets are equal sized
- **Weighted** fair queuing (WFQ): assign different flows different shares
- Today, some form of WFQ implemented in almost all routers
  - Not the case in the 1980-90s, when CC was being developed
  - Mostly used to isolate traffic at larger granularities (e.g., per-prefix)

# FQ vs. FIFO

- FQ advantages:
  - Isolation: cheating flows don't benefit
  - Bandwidth share does not depend on RTT
  - Flows can pick any rate adjustment scheme they want
  
- Disadvantages:
  - More complex than FIFO: per flow queue/state, additional per-packet book-keeping

# FQ in the big picture

- FQ does not eliminate congestion → it just manages the congestion



# Explicit Congestion Notification (ECN)

- Single bit in packet header; set by congested routers
  - If data packet has bit set, then ACK has ECN bit set
- Many options for when routers set the bit
  - tradeoff between (link) utilization and (packet) delay
- Congestion semantics can be exactly like that of drop
  - I.e., endhost reacts as though it saw a drop
- Advantages:
  - Don't confuse corruption with congestion; recovery w/ rate adjustment
  - Can serve as an early indicator of congestion to avoid delays
  - Easy (easier) to incrementally deploy
    - defined as extension to TCP/IP in RFC 3168 (uses diffserv bits in the IP header)

# TCP in detail

- What does TCP do?
  - ARQ windowing, set-up, tear-down
- Flow Control in TCP
- Congestion Control in TCP
  - AIMD, Fast-Recovery, Throughput
- Limitations of TCP Congestion Control
- Router-assisted Congestion Control (eg ECN)



# Recap

- TCP:
  - somewhat hacky
  - but practical/deployable
  - good enough to have raised the bar for the deployment of new, more optimal, approaches
  - though the needs of datacenters might change the status quos
- Beyond TCP (discussed in Topic 6):
  - QUIC / application-aware transport layers