CS2105 Introduction to Computer Networks

Lecture 5

UDP and TCP

10 September 2018

Midterm Test

- Monday 8 October 2018
  - Week 8
  - Time: 2:15pm – 3:15pm
  - Venue: MPSH 2
- MRQs
  - Multiple Response Question
  - More fun than MCQ
  - Partial score → higher random expectation
- Short Qns?

PREVIOUS LECTURE

Designing Reliable Protocols

Network layer service is unreliable.

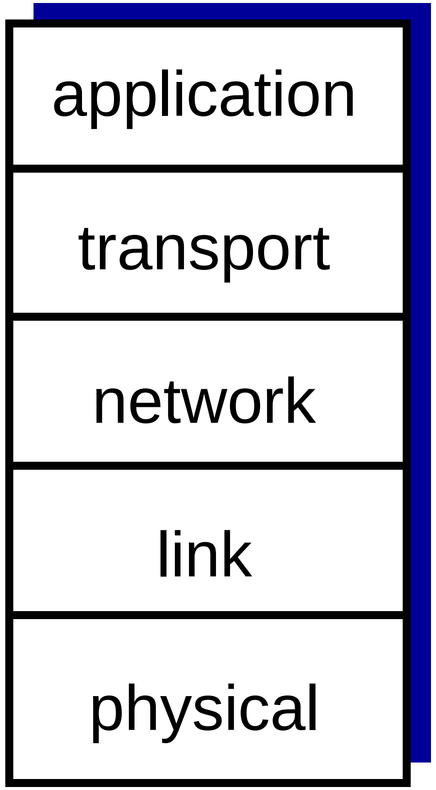
- Packets may be lost or corrupted during transmission.

A reliable transport protocol should

- ensure that packets are received by receiver in good order.

Scenario: one sender, one receiver

- Sender sends data packets to receiver
- Receiver feeds back to sender

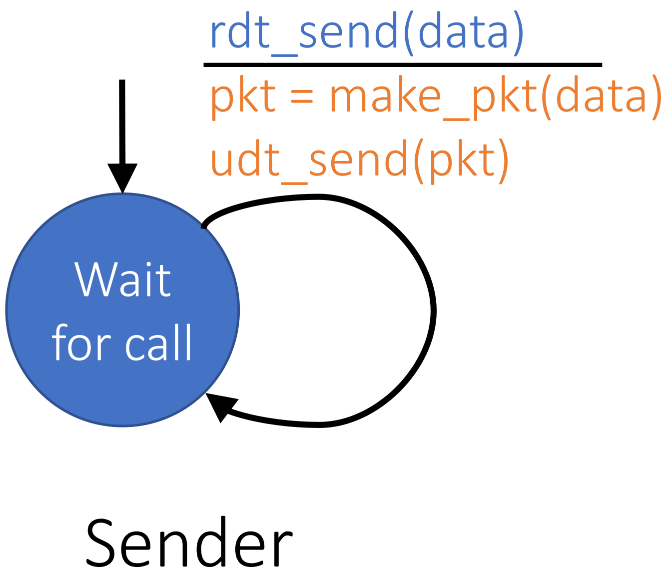


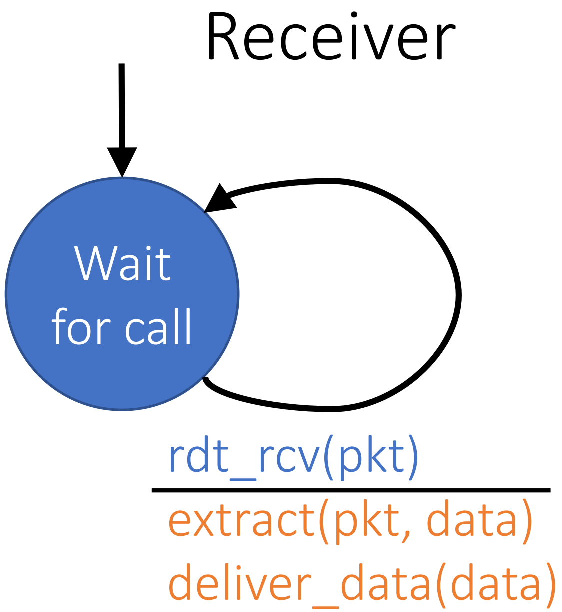
rdt 1.0: Reliable Channel

Assume underlying channel is perfectly reliable.

No error checking is needed.

- Sender sends data into underlying (perfect) channel
- Receiver reads data from underlying (perfect) channel





rdt 2.0: Channel with Bit Errors

PREVIOUS LECTURE

Assumption:

- underlying channel may flip bits in packets
- other than that, the channel is perfect

Receiver may use checksum to detect bit errors

Question: how to recover from bit errors?

- Acknowledgements (ACKs): receiver explicitly tells sender that packet received is OK.
- Negative acknowledgements (NAKs): receiver explicitly tells sender that packet has errors.
- Sender retransmits packet on receipt of NAK.

PREVIOUS LECTURE

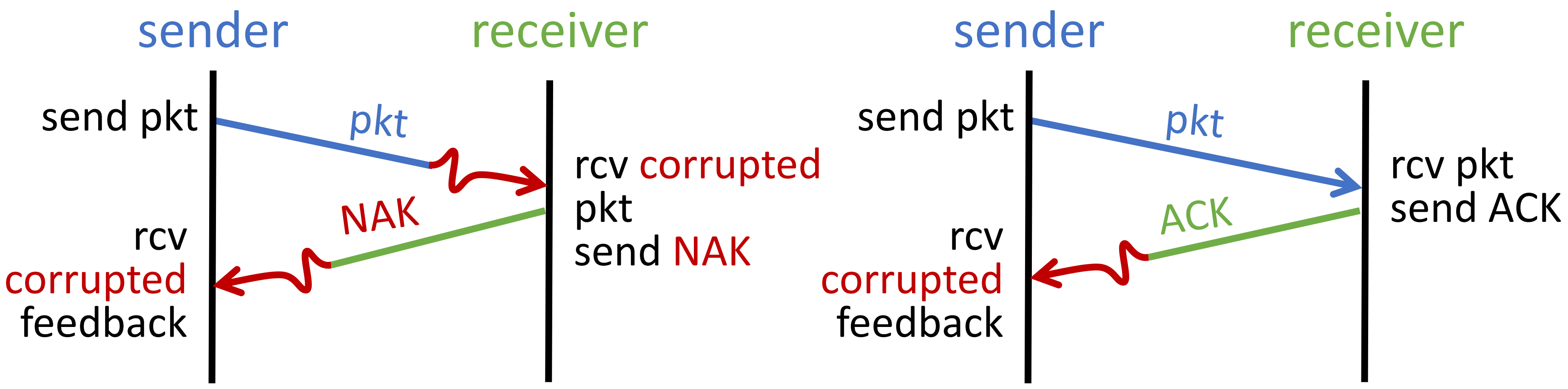
rdt 2.0 has a Fatal Flaw!

What happens if ACK/NAK is corrupted?

- Sender doesn't know what happened at receiver!

So what should the sender do?

- Sender just retransmits when receives garbled ACK or NAK.
- Question: does this work?



a) corrupted NAK

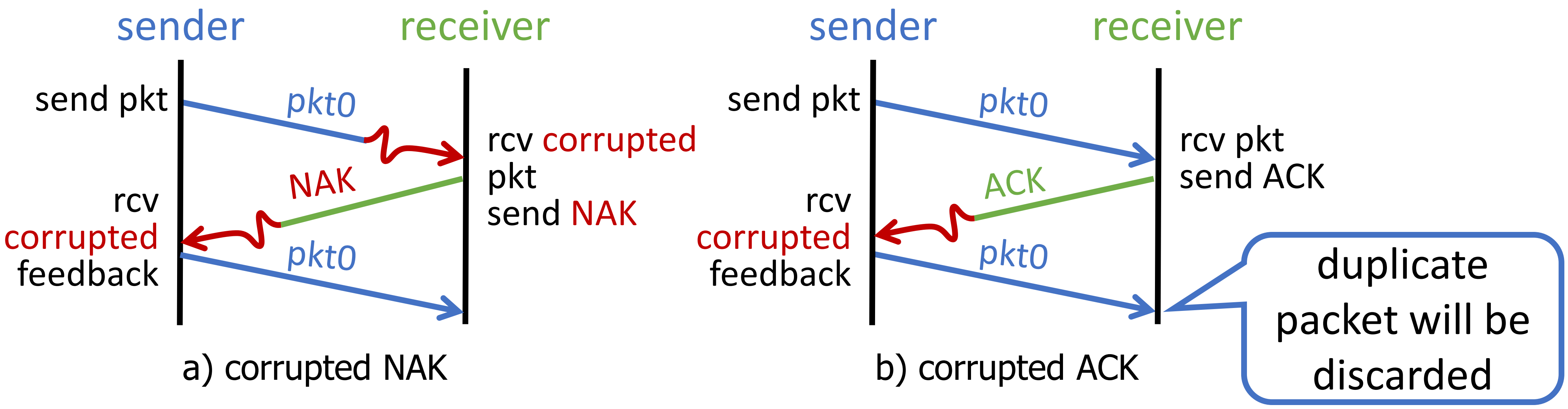
b) corrupted ACK

rdt 2.1: Add a sequence number

To handle duplicates:

- Sender retransmits current packet if ACK/NAK is garbled.
- Sender adds sequence number to each packet.
- Receiver discards (doesn't deliver up) duplicate packet.

This gives rise to protocol rdt 2.1.



rdt 2.2: a NAK-free Protocol

Same assumption and functionality as rdt 2.1, but use ACKs only.

Instead of sending NAK, receiver sends ACK for the last packet received correctly.

- Now receiver must explicitly include seq. # of the packet being ACKed.

Duplicate ACKs at sender results in same action as NAK: retransmit current pkt

rdt 3.0: Channel with Errors and Loss

PREVIOUS LECTURE

Assumption: underlying channel

- may flip bits in packets
- may lose packets
- may incur arbitrarily long packet delay
- but will not re-order packets

To handle packet loss:

- Sender waits “reasonable” amount of time for ACK.
- Sender retransmits if no ACK is received till timeout.

Sender relies on timeout/retransmission to deal with both packet corruption and packet loss.

Learning Outcomes

After this class, you are expected to:

- know the simplicity of UDP and the service it provides.
- know how to calculate the checksum of a packet.
- know the operation of the components of TCP
  - sequence number
  - acknowledgement number
  - retransmission,
  - receiver window
  - connection setup and termination

Chapter 3: Roadmap

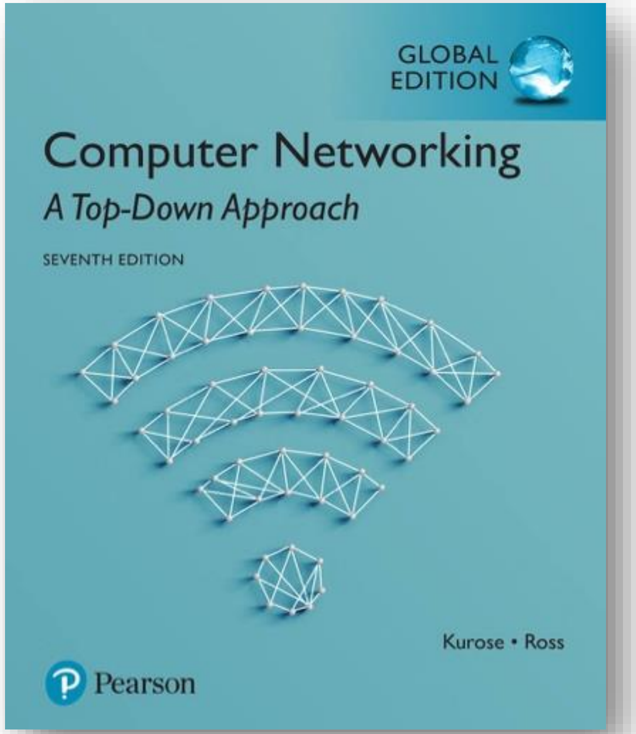
3.1 Transport-layer services

3.2 Multiplexing and de-multiplexing

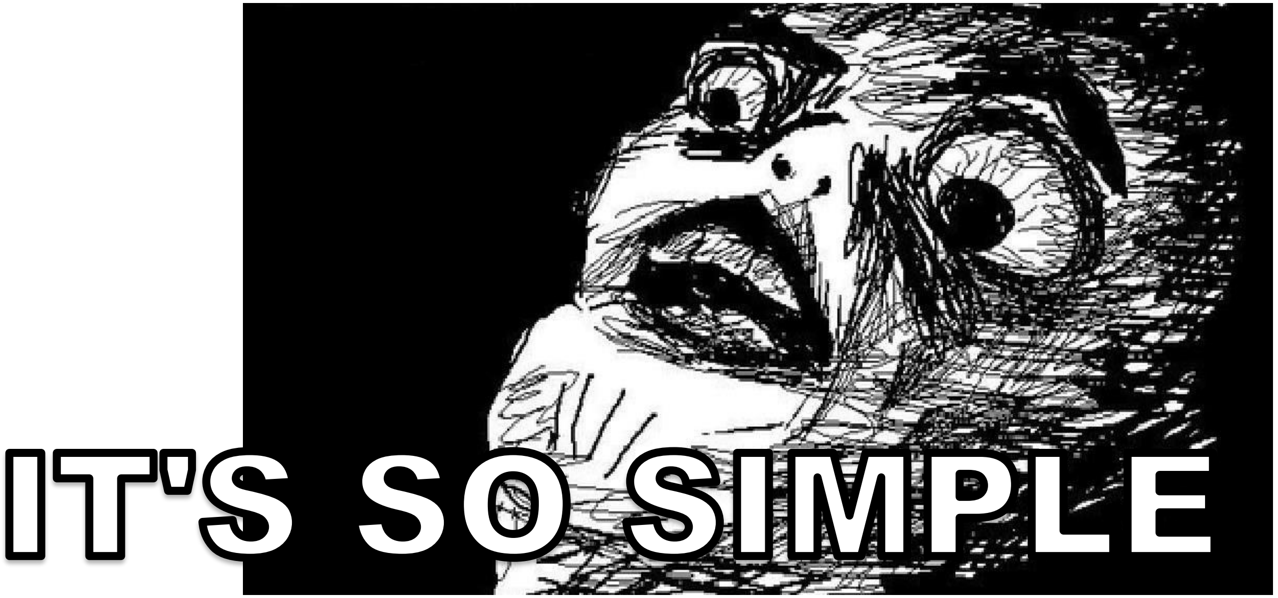
3.3 Connectionless transport: UDP

3.4 Principles of reliable data transfer

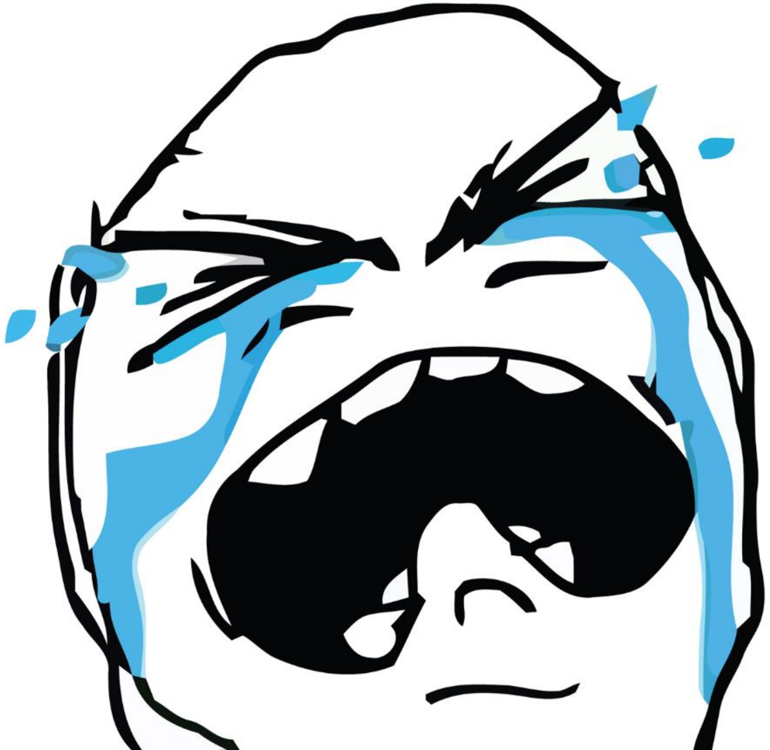
3.5 Connection-oriented transport: TCP



You can't imagine how simple UDP is.



But the complexity of TCP will make you cry



Layering

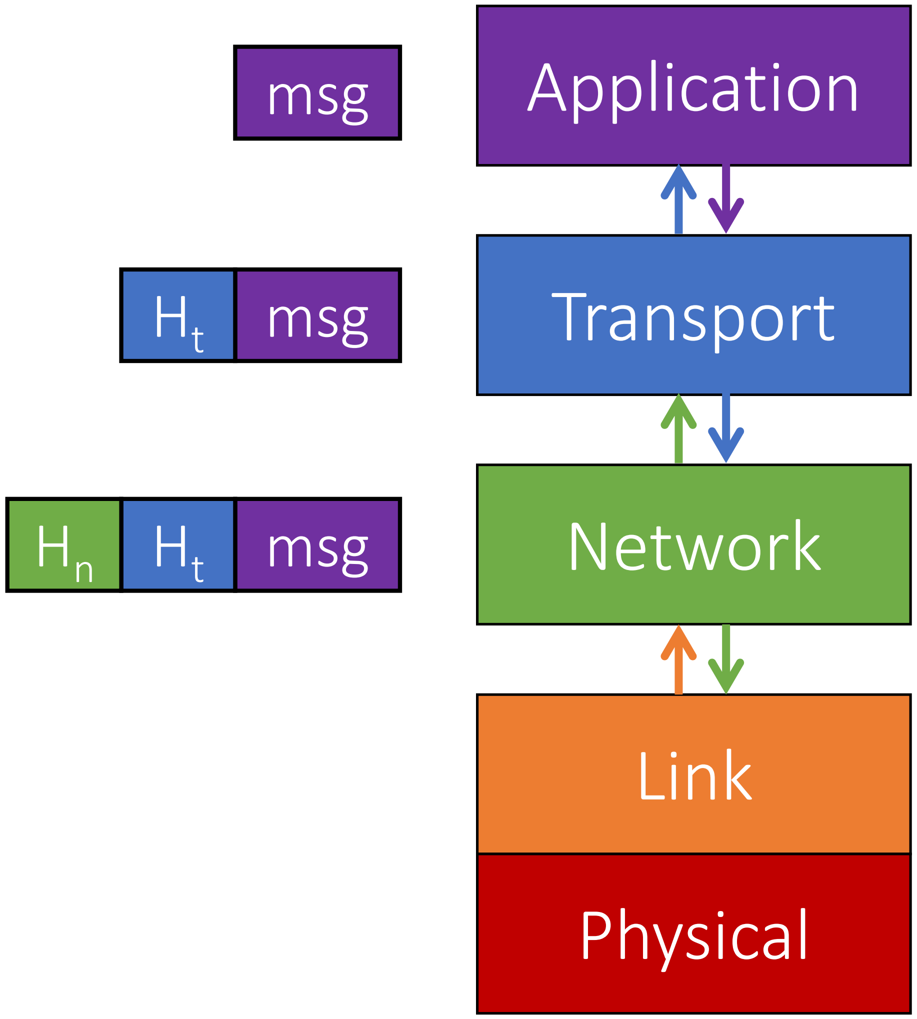
message

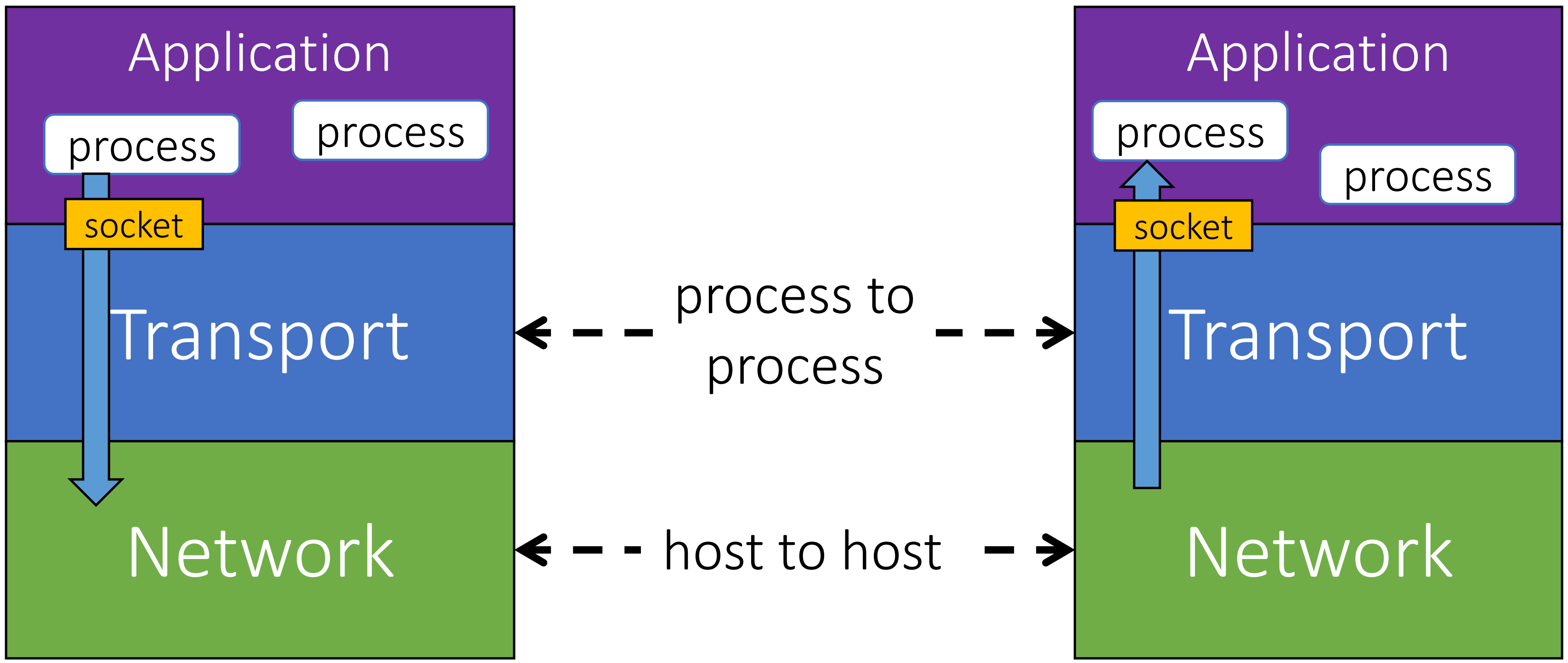
segment

H_t contains source and destination port

datagram

H_n contains source and destination IP address




Application
process
process
socket
Transport
Network
process to
process
host to host
Application
process
process
socket
Transport
Network

User Datagram Protocol

[RFC 768]

Internet protocols are described in documents known as Request for Comments (RFC)

http://www.itef.org/rfc/rfc768.txt is only 3 pages

UDP

adds very little service on top of IP:

- Connectionless multiplexing / de-multiplexing
- Checksum

transmission is unreliable

- Often used by streaming multimedia apps (loss tolerant & rate sensitive)

To achieve reliable transmission over UDP

- Application implements error detection and recovery mechanisms!

Connectionless De-multiplexing

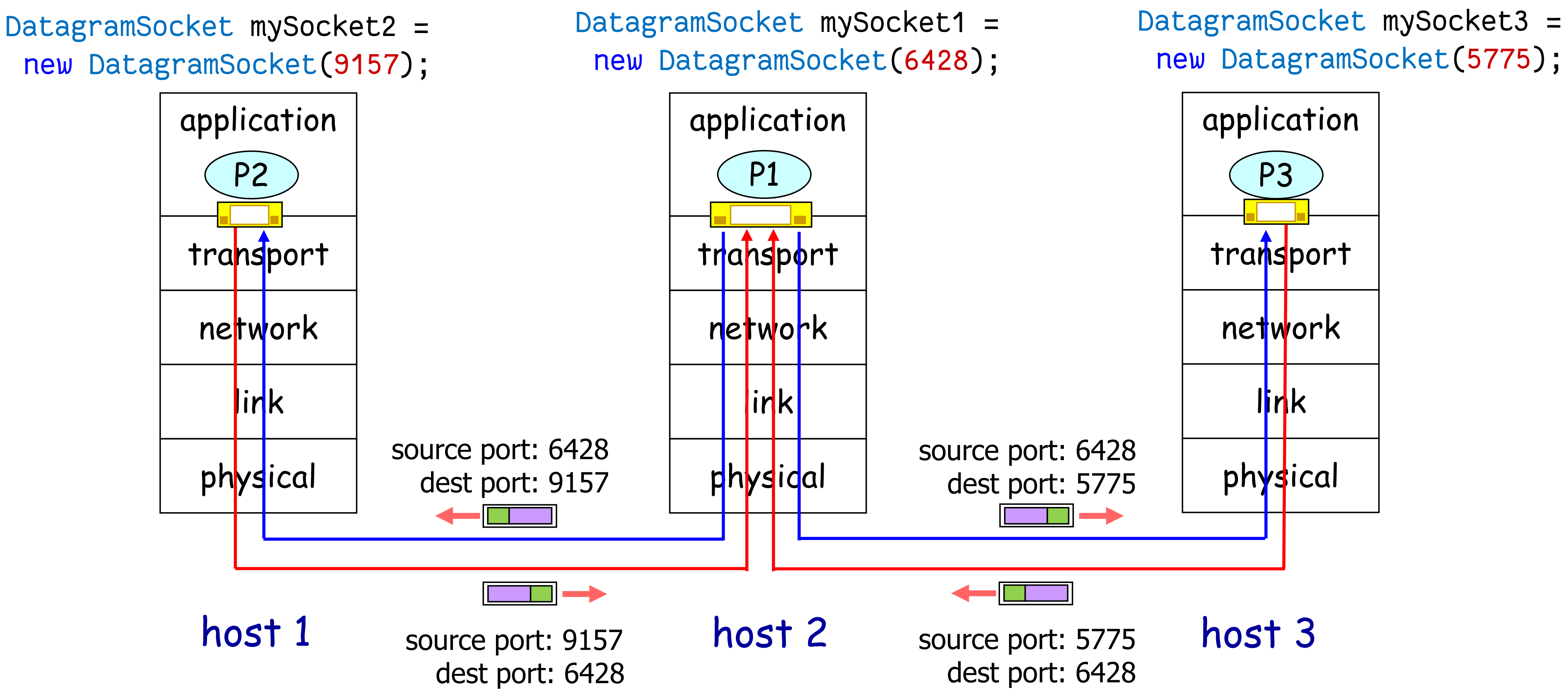
UDP sender:

- Creates a socket with local port #
- When creating a datagram to send to UDP socket, sender must specify destination IP address and port #

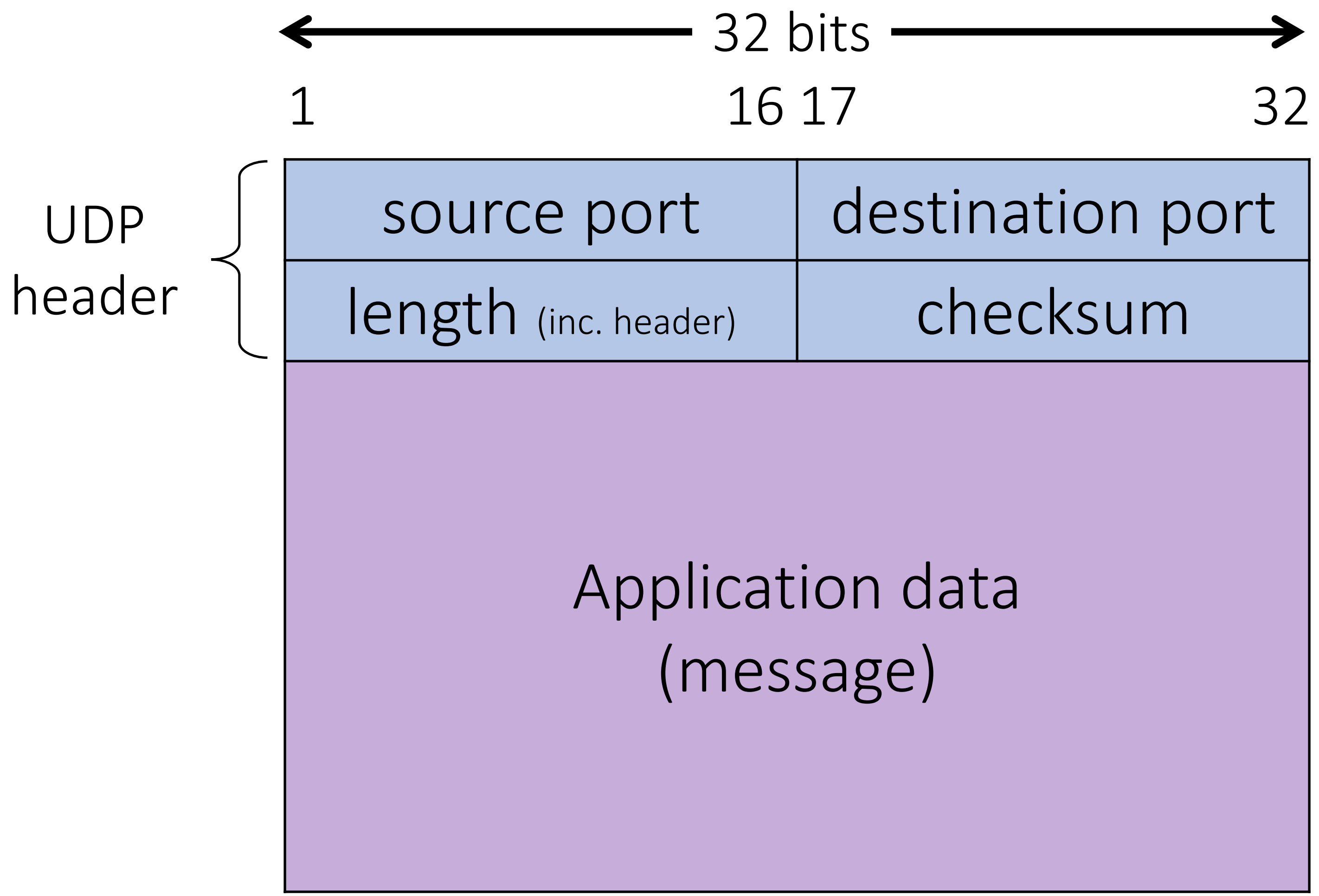
When UDP receiver receives a UDP segment:

- Checks destination port # in segment
- Directs UDP segment to the socket with that port #
- IP datagrams (from different sources) with the same destination port # will be directed to the same UDP socket at destination

Connectionless De-multiplexing



UDP segment structure



Why use UDP?

No connection set-up

- Reduce delay

No connection state at sender or receiver

- Need less resources

Small header size

- Less overhead

No congestion control

- Can blast as fast as desired

Checksum

Several different checksum algorithm

- Cyclic Redundancy Check (CRC)
- Message Digest v5 (MD5)
- Secure Hash Algorithm 1 (SHA-1)
- Secure Hash Algorithm 2 (SHA-2)
- UDP/TCP Checksum (RFC 768)

UDP Checksum

Purpose: to detect errors (single bit flips) in the transmitted segment.

Sender

Compute checksum value

- $f(P_s) = c_s$

Include checksum value into UDP checksum field

Receiver

Compute checksum value

- $f(P_r) = c_r$

Compare if computed checksum equals checksum field value:

- NO - error detected
- YES - no error detected (but really no error?)

Computing Checksum

1. Split segment into 16-bit integers (checksum initially 0)

```
1110 0110 0110 0110
1101 0101 0101 0101
```

Computing Checksum

1. Split segment into 16-bit integers
2. Add next integer with wrap around carry

carry

wrap-around carry

```
  1110 0110 0110 0110
  1101 0101 0101 0101
1 1011 1011 1011 1011
                    1
  1011 1011 1011 1100
```

Computing Checksum

1. Split segment into 16-bit integers
2. Add next integer with wrap around carry
3. Compute 1's complement

```
      1110 0110 0110 0110
      1101 0101 0101 0101
    1 1011 1011 1011 1011
    ---------------------
                        1
sum:      1011 1011 1011 1100
checksum: 0100 0100 0100 0011
```

Intuition

Using base10 integers

- Ensure no change to a sequence of numbers:

$$27, 48, 13, 73, 52$$

Compute the sum

- $27 + 48 + 13 + 73 + 52 = 186$

But what if you only have 2 digits?

- Truncate? 86
- Wrap around? $86 + 1 = 87$

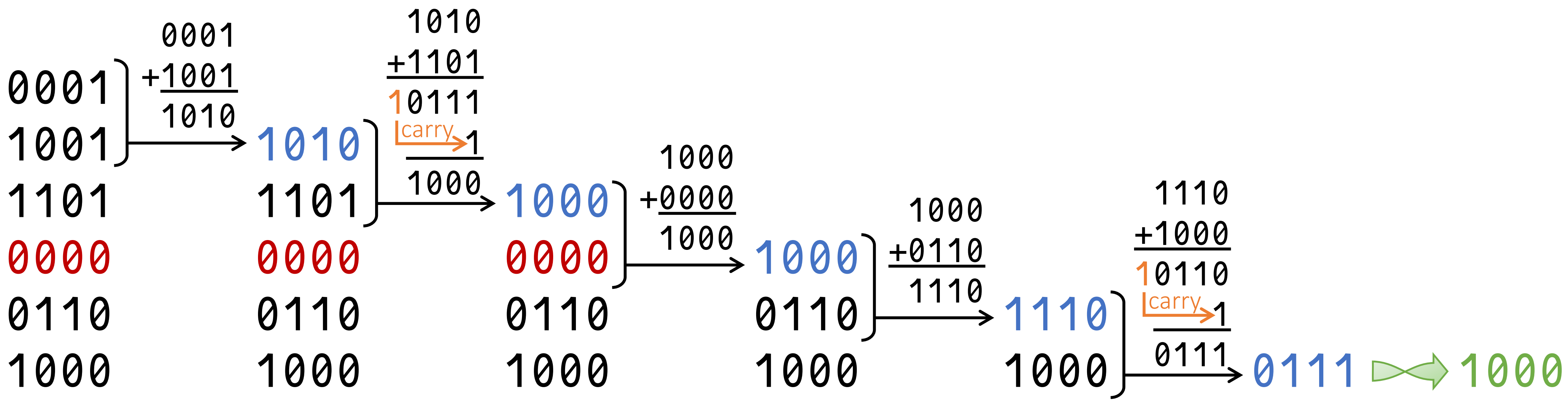
Take compliment?

- $87 \rightarrow 12$

Example

Using 4-bit integers for simplicity

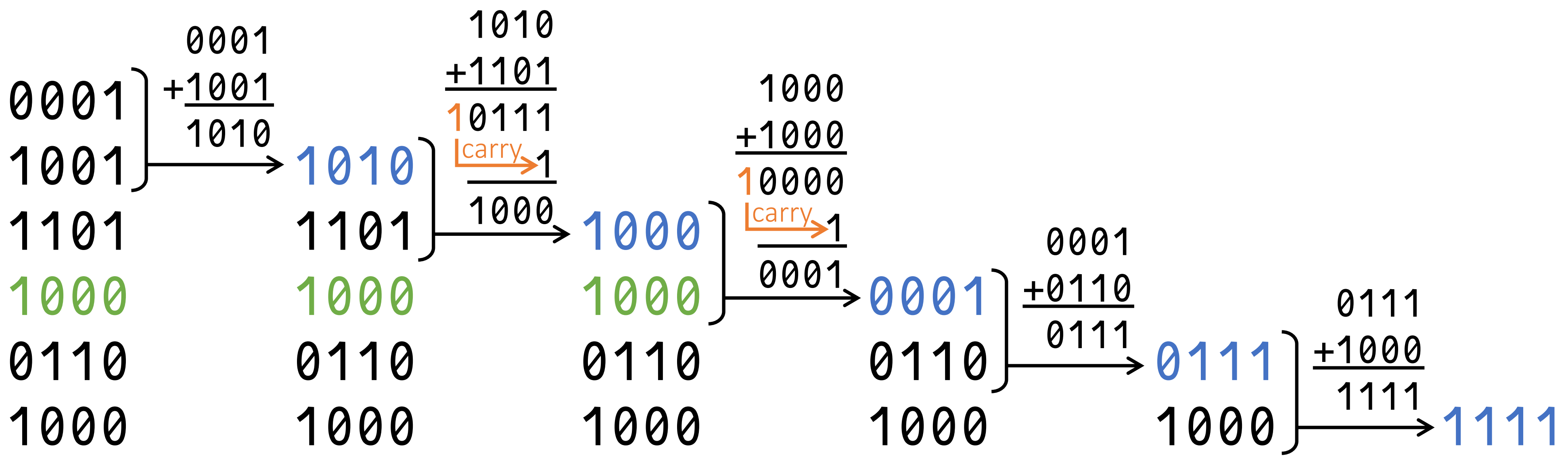
- Sender computes checksum for this message



Example

Using 4-bit integers for simplicity

- Receiver verifies checksum for this message



Chapter 3: Roadmap

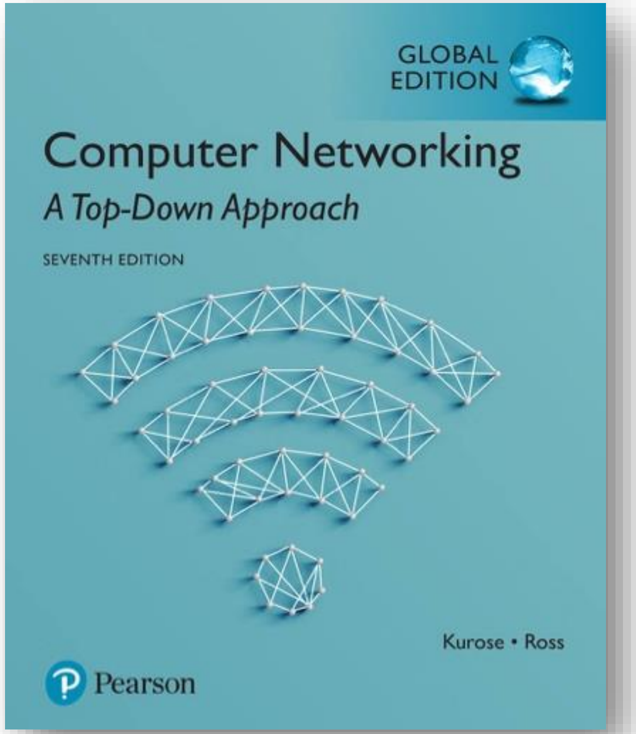
3.1 Transport-layer services

3.2 Multiplexing and de-multiplexing

3.3 Connectionless transport: UDP

3.4 Principles of reliable data transfer

3.5 Connection-oriented transport: TCP



Transmission Control Protocol

[RFC 793, RFC 1122, RFC 1323, RFC 2018, RFC 2581, …]

We only scratch the surface of TCP in CS2105.
More will be covered in CS3103.

TCP Overview

Connection-oriented

- handshaking (exchange of control messages) before sending app data

Reliable, in-order byte steam

- Application passes data to TCP and TCP forms packets in view of MSS (maximum segment size)
- (For UDP, app forms packets: `DatagramPacket`)

Flow control and congestion control

- Not discussed!

Re-cap

	Go-Back-N	Selective Repeat
ACK	cumulative	selective
out-of-order	ignore	buffer
retransmit	all unACK	one unACK
timer	oldest unACK	one per unACK

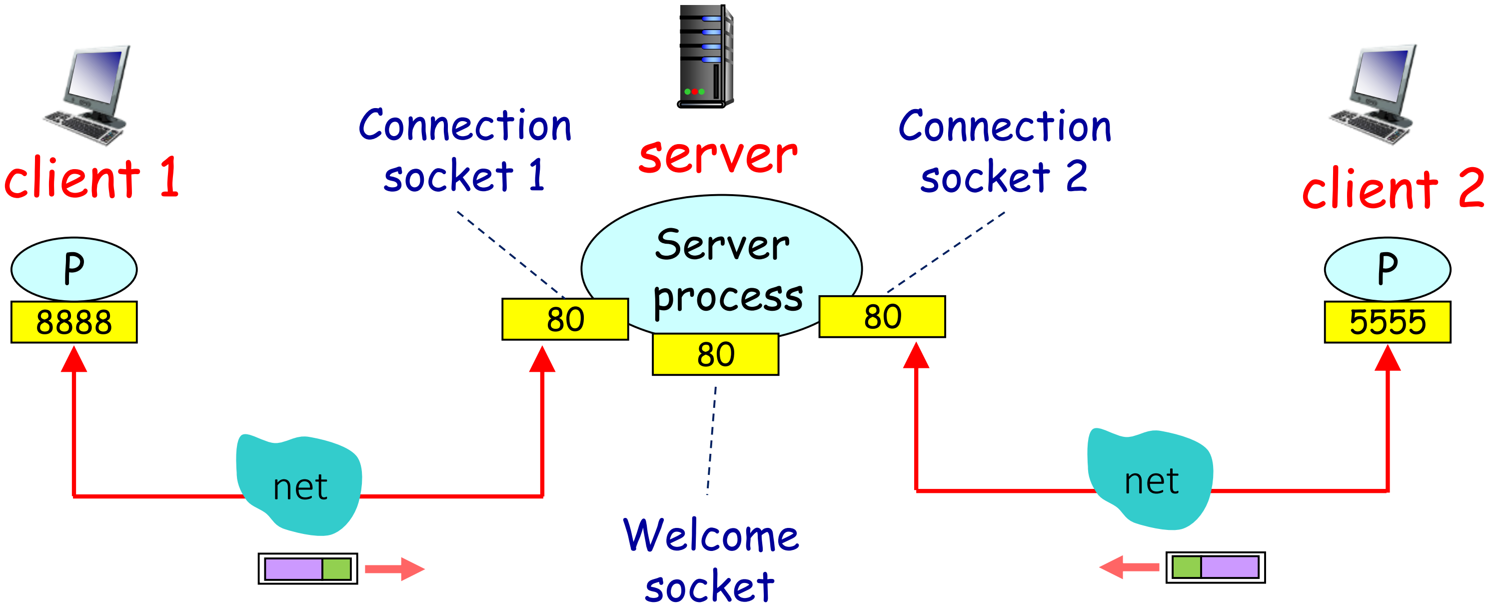
Which is TCP?

You should be able to answer after this lecture

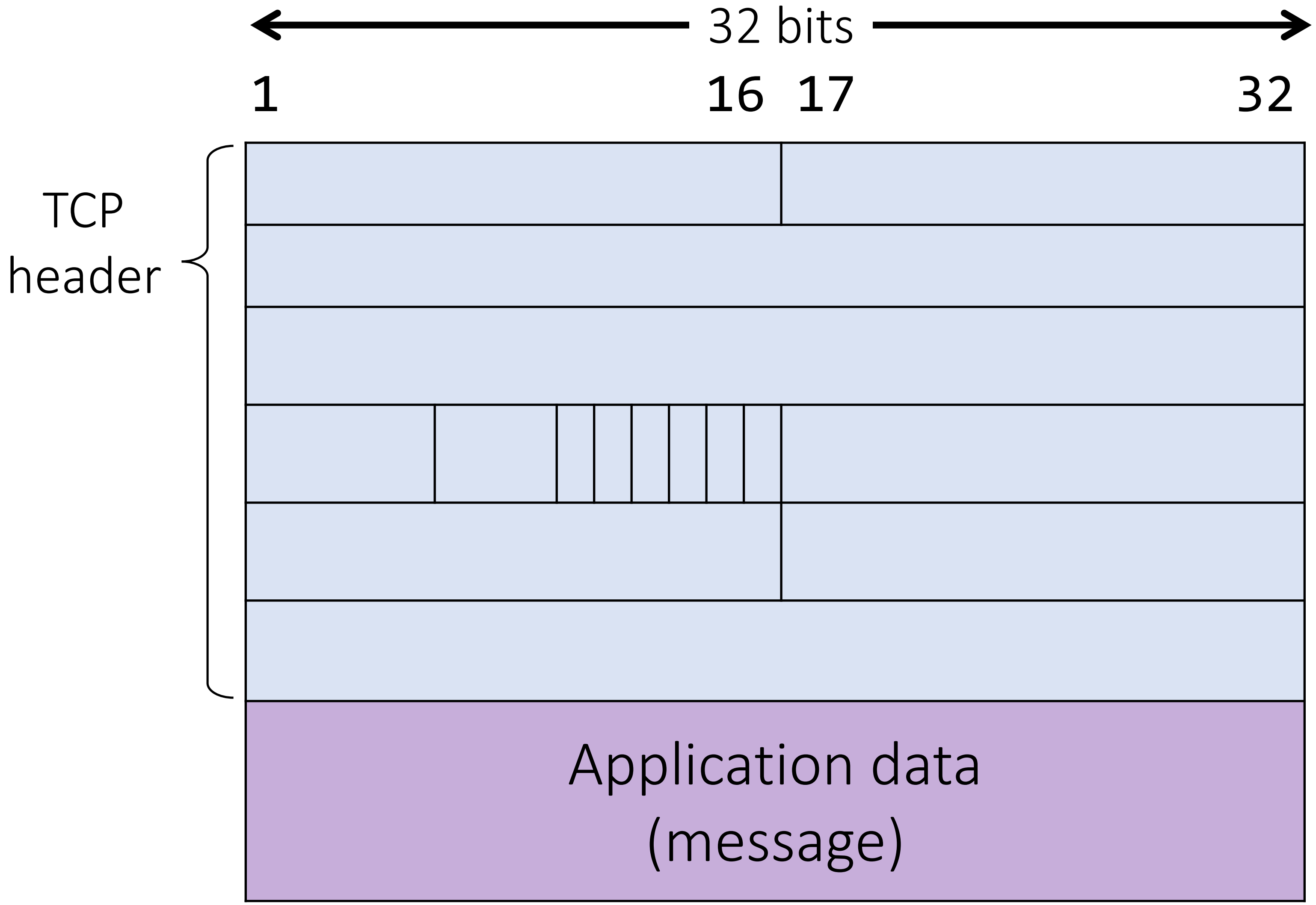
Connection-oriented De-mux

A TCP connection (socket) is identified by 4-tuple:

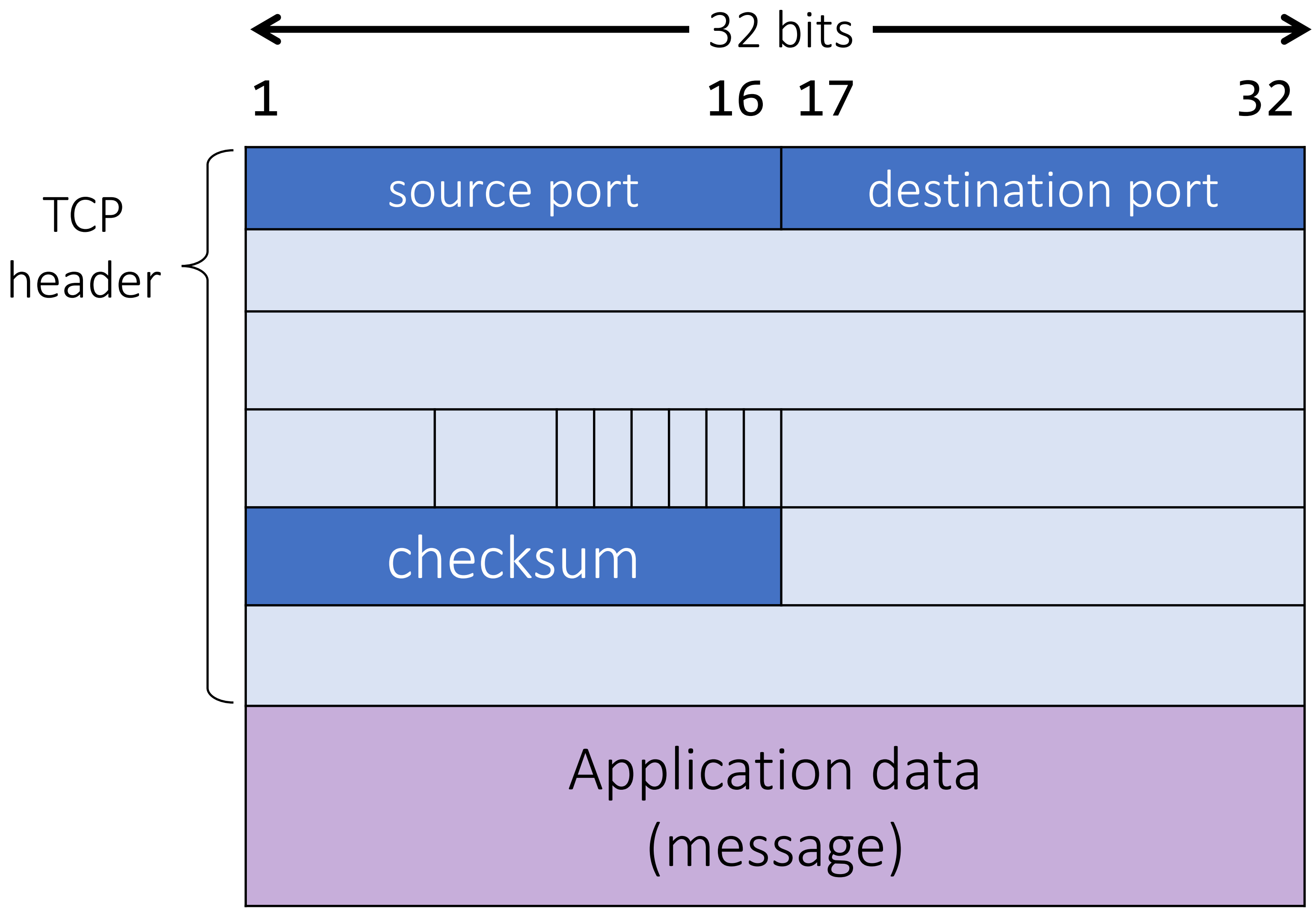
- (srcIPAddr, srcPort, destIPAddr, destPort)
- Receiver uses all four values to direct a segment to the appropriate socket.



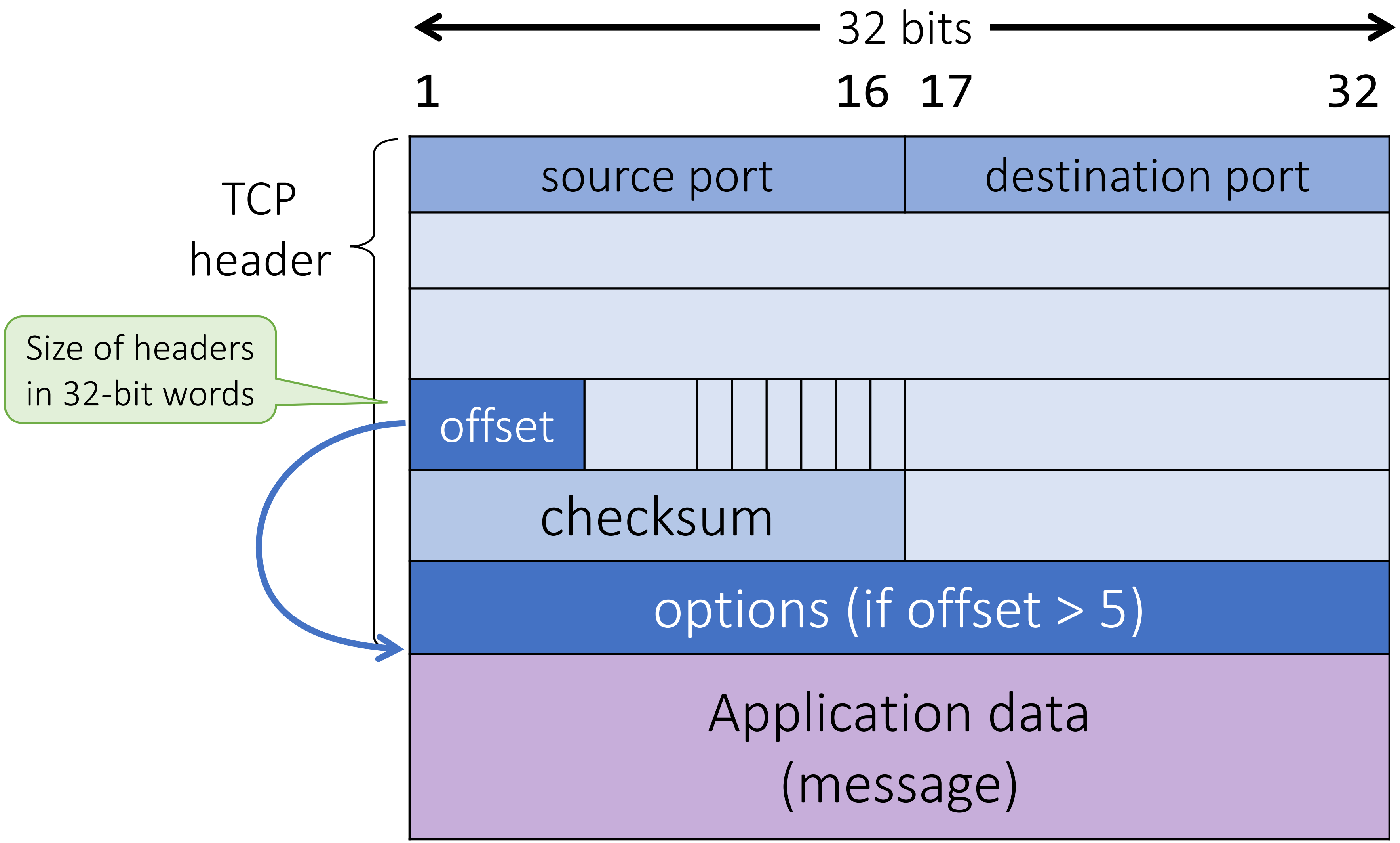
TCP segment structure



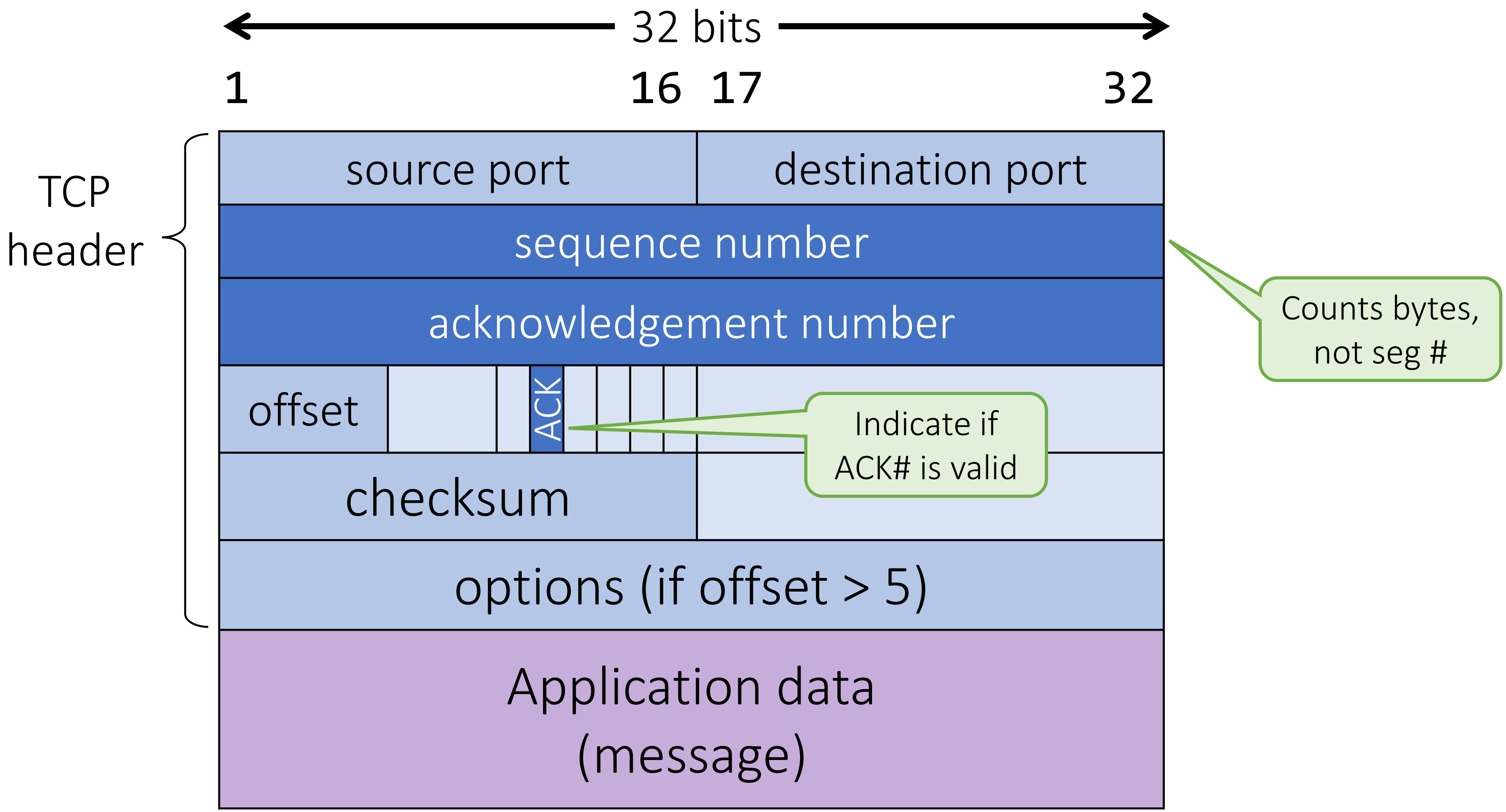
TCP segment structure



TCP segment structure



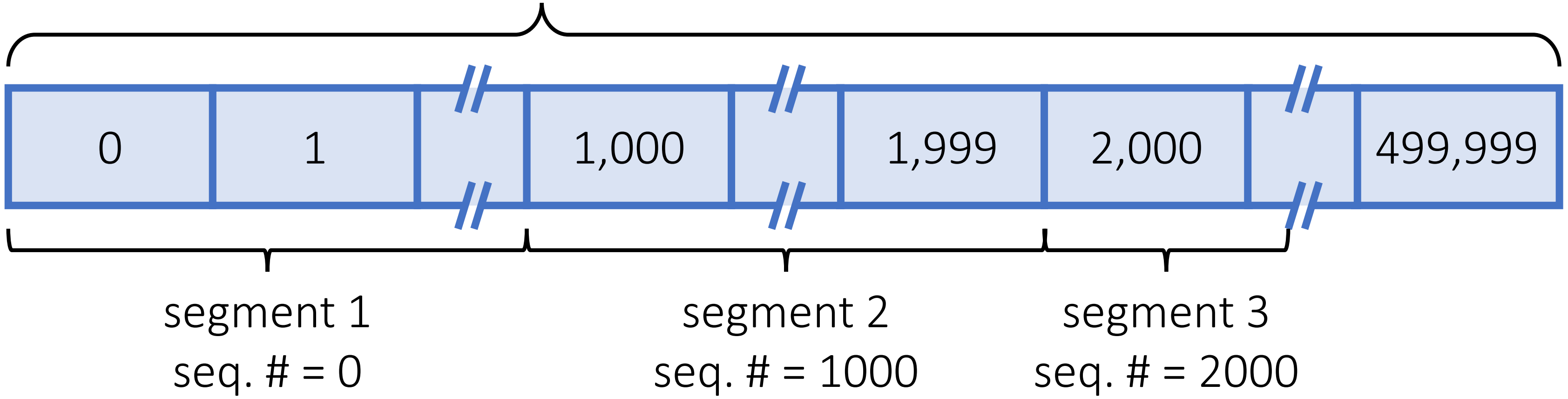
TCP Sequence numbers



TCP Sequence Number

"byte number" of the first byte of data in the segment

Example: 500,000-byte file with 1,000-byte segments

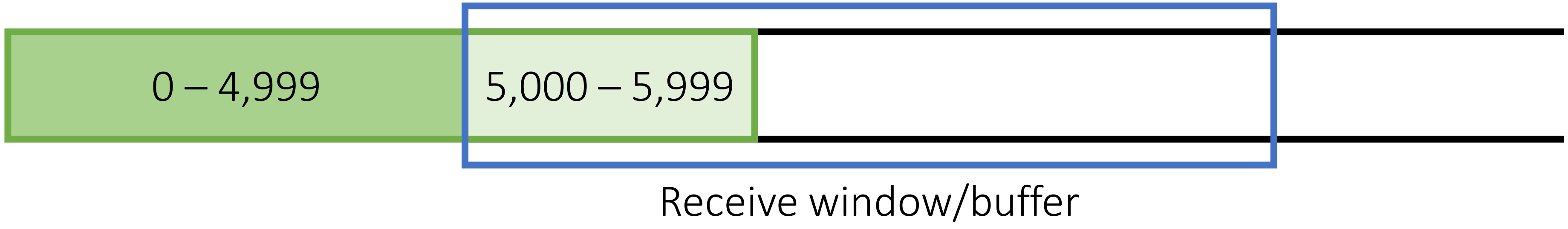


TCP Acknowledgement Number

Seq. # of the **next byte** of data expected

TCP only ACKs up to the missing byte (cumulative ACK)

Receives 1,000 byte seq. # 5,000



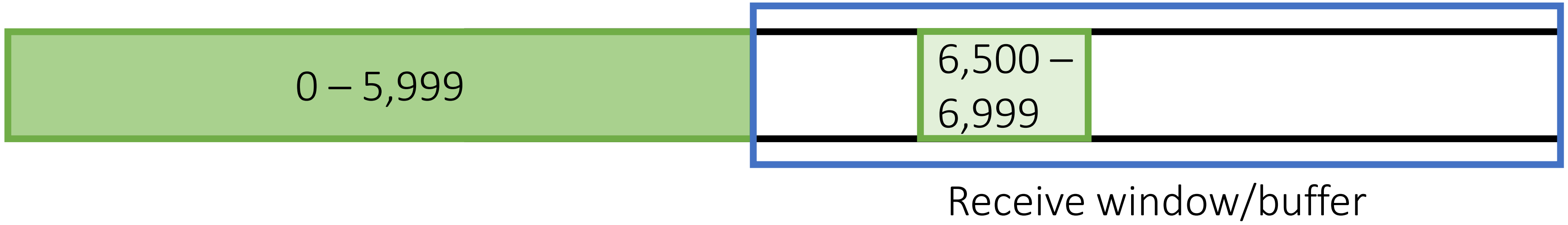
Sends ACK # 6,000

TCP Acknowledgement Number

Seq. # of the next byte of data expected

TCP only ACKs up to the missing byte (cumulative ACK)

Receives 500 byte seq. # 6,500



Sends ACK # 6,000

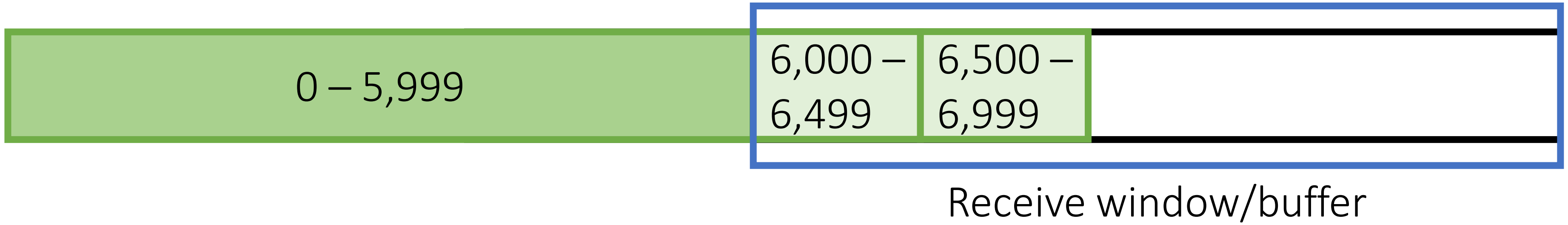
Note: TCP specs does not say how out-of-order segments should be handled

TCP Acknowledgement Number

Seq. # of the <u>next byte</u> of data expected

TCP only ACKs up to the missing byte (cumulative ACK)

Receives 500 byte seq. # 6,000



Sends ACK # 7,000

TCP Acknowledgement Number

Seq. # of the <u>next byte</u> of data expected

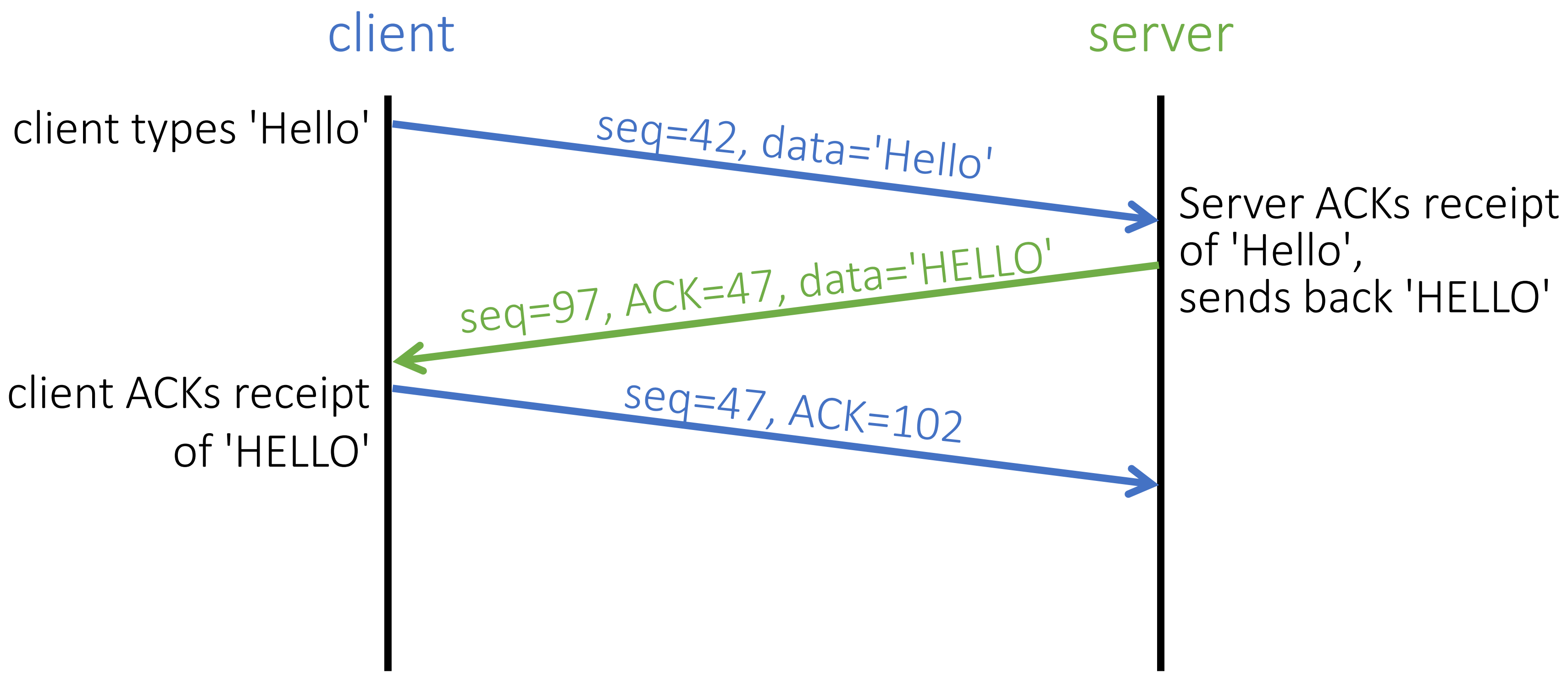
TCP only ACKs up to the missing byte (cumulative ACK)

Receives 500 byte seq. # 6,000

Sends ACK # 7,000

TCP Echo Server

TCP is full duplex, i.e., bi-directional data flow



ACKs are "piggybacked" on data segment

Question: How large should a segment be?

Maximum Segment Size (MSS)

typically derived from link-layer's MTU

TCP Sender Events (simplified)

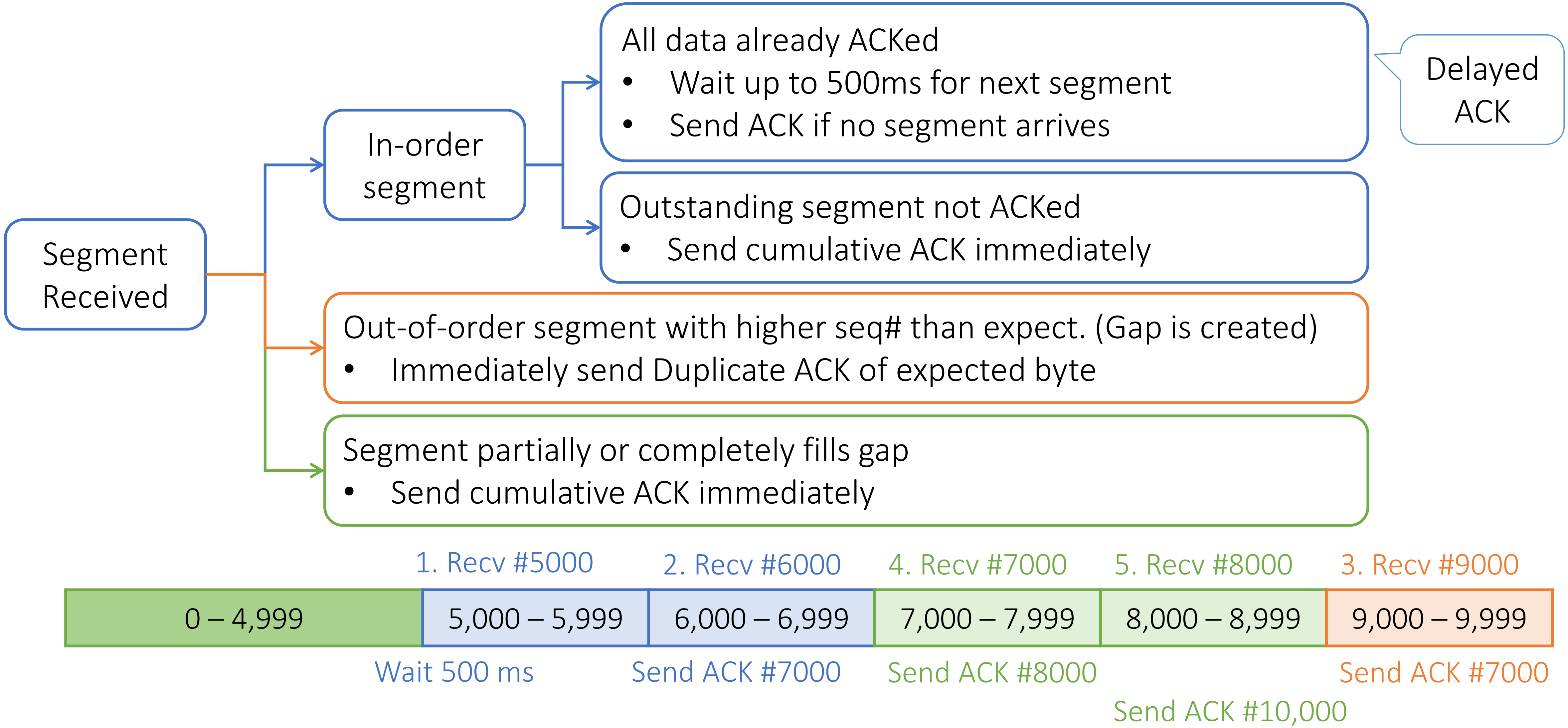
```
loop(forever)
  switch(event)
    event: data received from application
        create TCP segment with nextSeqNum
        if (timer not currently running)
            start timer
        pass segment to IP
        nextSeqNum += length(data)

    event: timer timeout
      retransmit unacknowledged segment with smallest seq num
      start timer

    event: ACK received, with ACK num #y
      if (y > sendBase)
          sendBase = y
          if (there are still unacknowledged segments)
              start timer
```

Sender only keeps one timer

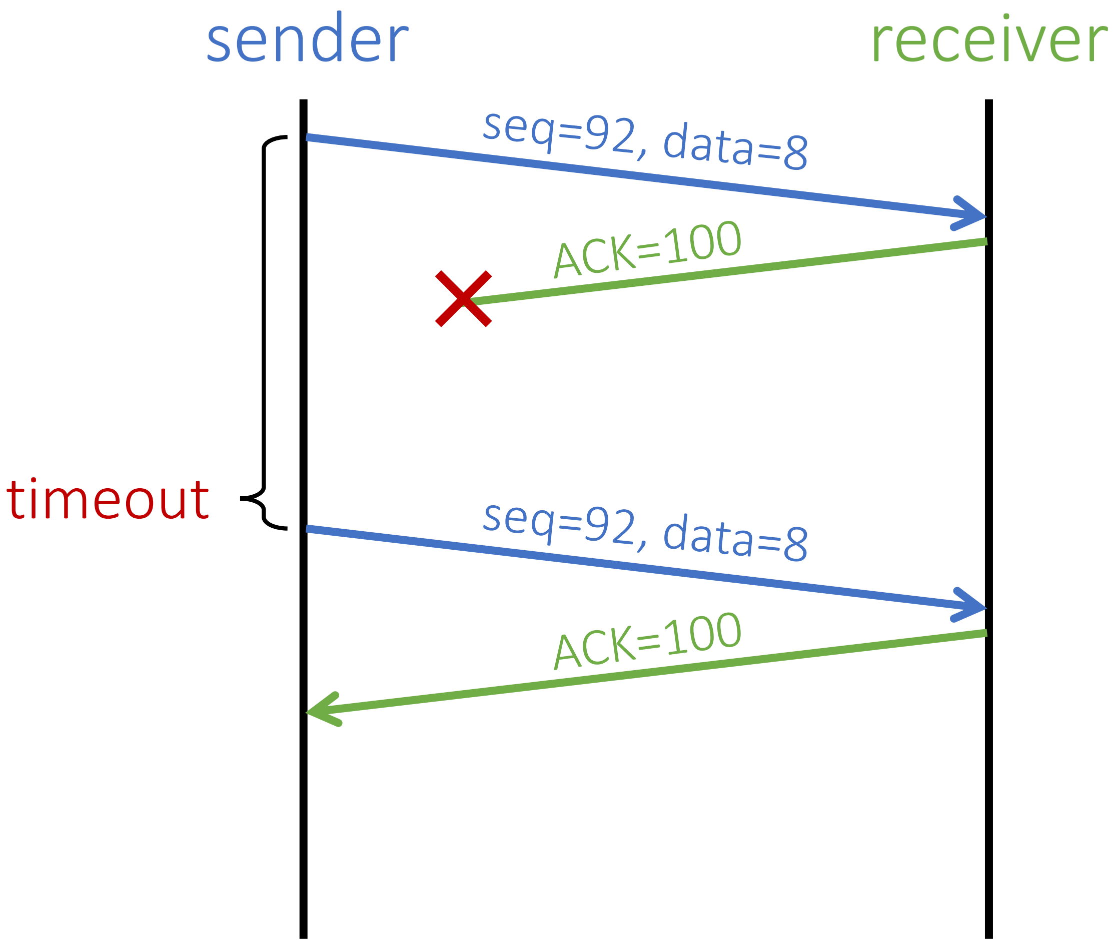
Retransmit only oldest unACK segment

Cumulative ACK

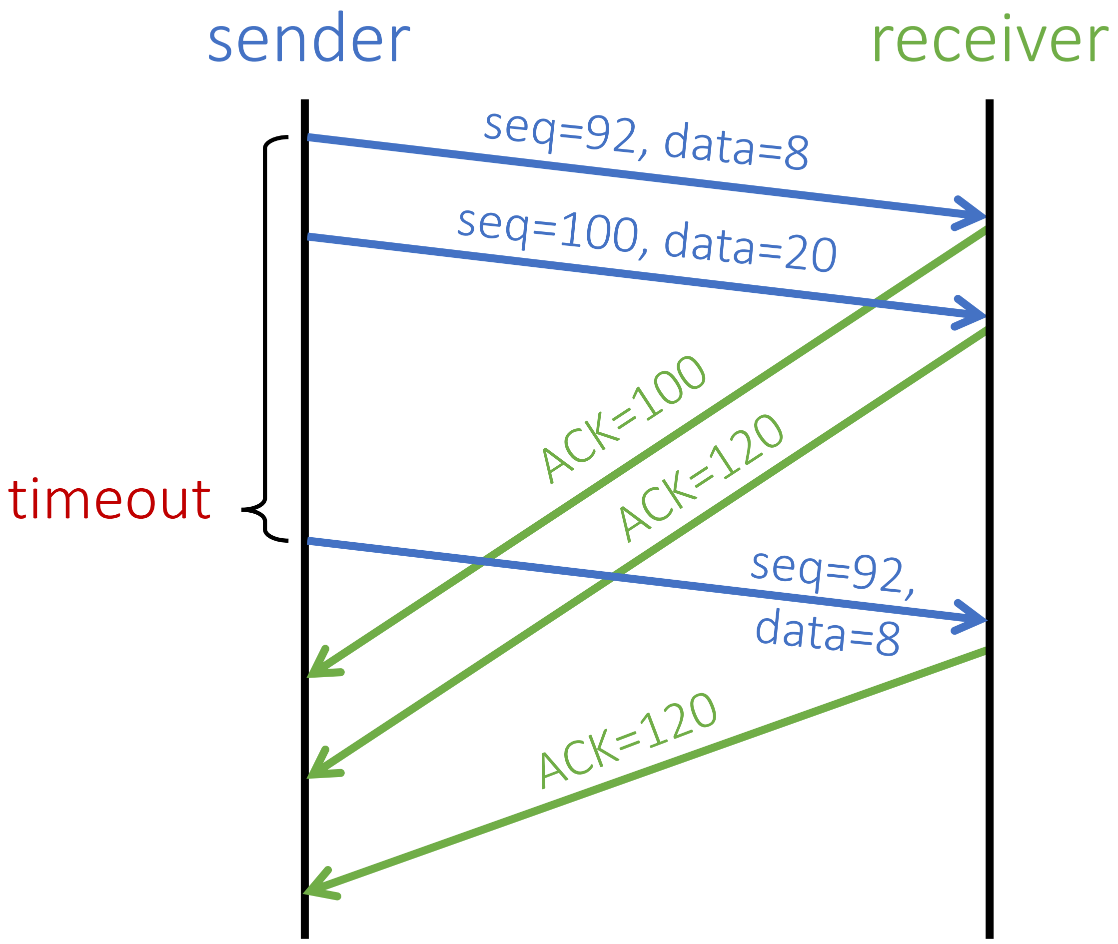
TCP Receiver Events



TCP Timeout/Retransmission

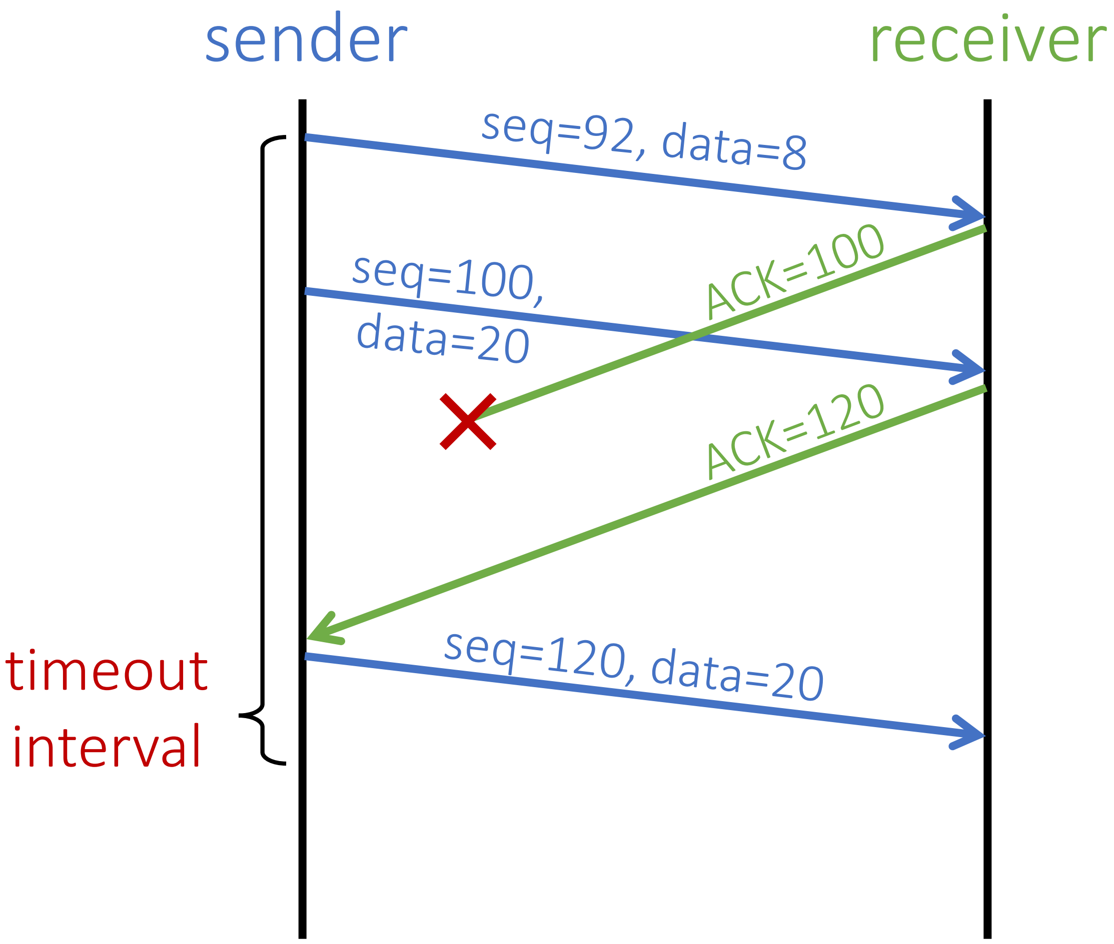


Lost ACK



Premature timeout

TCP Timeout/Retransmission



Cumulative ACK prevents rxmt

TCP Timeout Value

Too short?

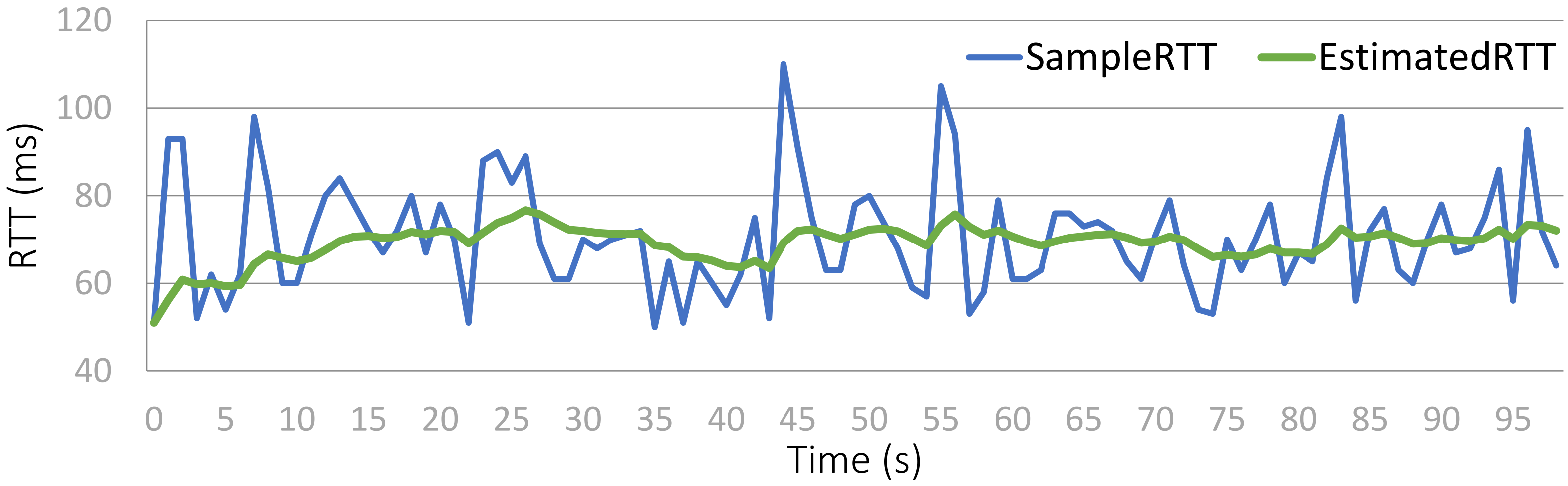
- Premature timeout and unnecessary retransmissions.

Too long?

- Slow reaction to loss

Key point: Timeout > RTT

- But RTT can vary!



Estimating RTT

Take SampleRTT (RTT_s)

- *Once every RTT*

Compute EstimatedRTT (RTT_ε)

- $RTT_\varepsilon = (1 - \alpha) \cdot RTT_\varepsilon + \alpha \cdot RTT_s$
- typical value of $\alpha = \frac{1}{8}$

Exponential Weighted Moving Average (EWMA)

Setting Retransmisson Time Out (RTO)

Compute Deviation of RTT

- $RTT_{dev} = (1 - \beta) \cdot RTT_{dev} + \beta \cdot |RTT_s - RTT_\varepsilon|$
- typical value of $\beta = \frac{1}{4}$

RTO Interval is set to

$$RTT_\varepsilon + 4 \times RTT_{dev}$$

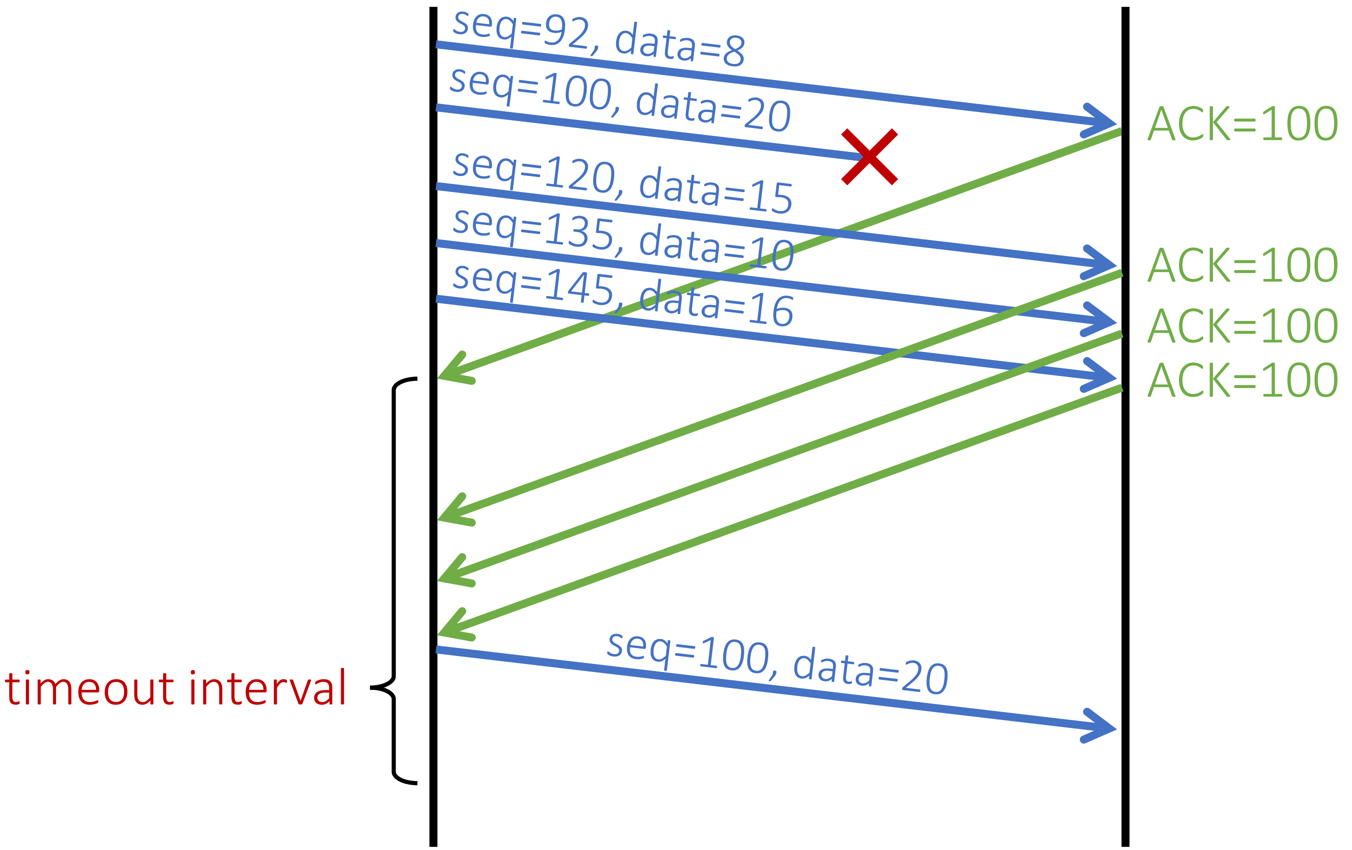
estimated RTT

“safety margin”

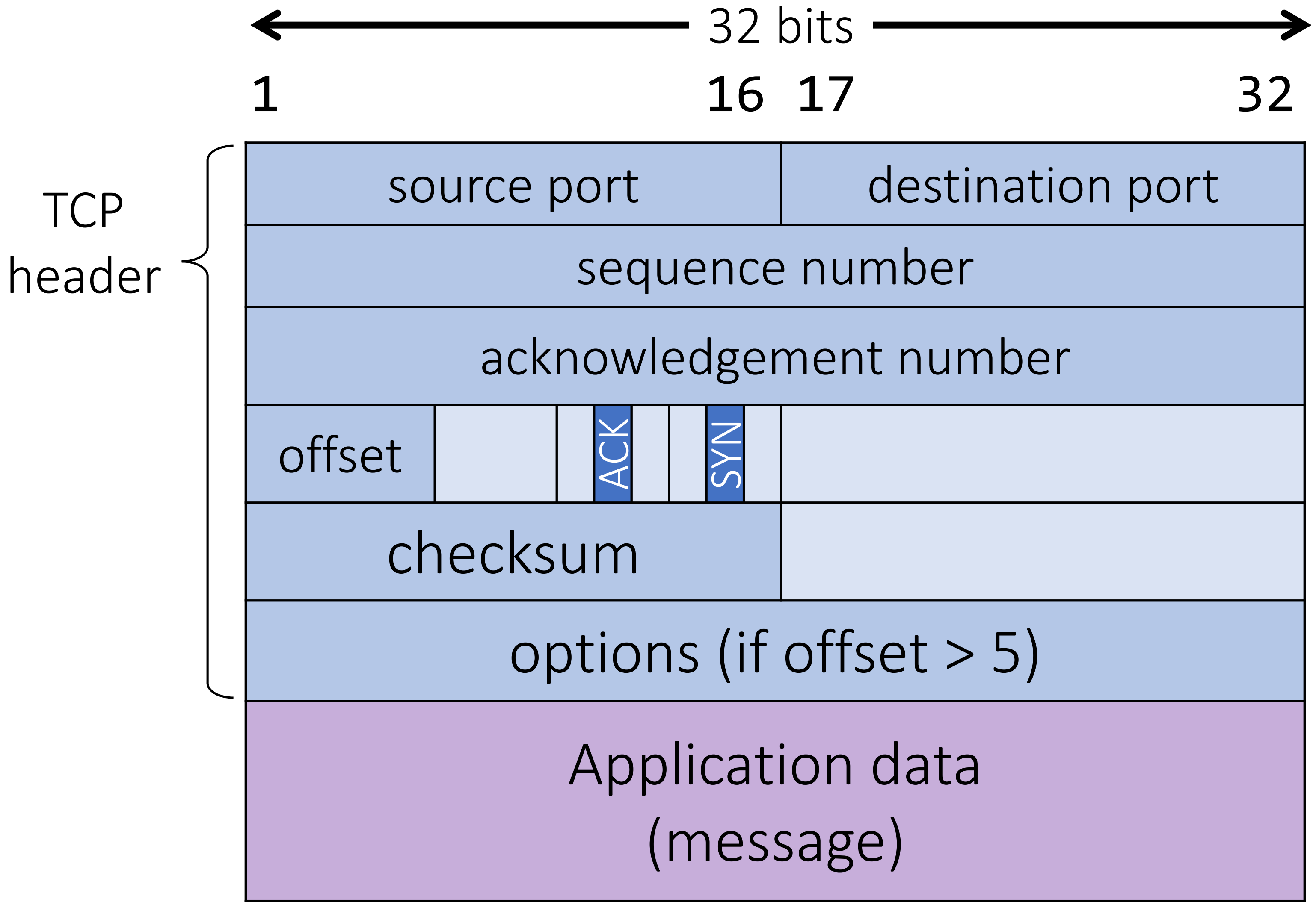
TCP Fast Retransmission [RFC 2001]

Timeout is often relatively long

– If 3 Duplicate ACKs are received, resend segment immediately



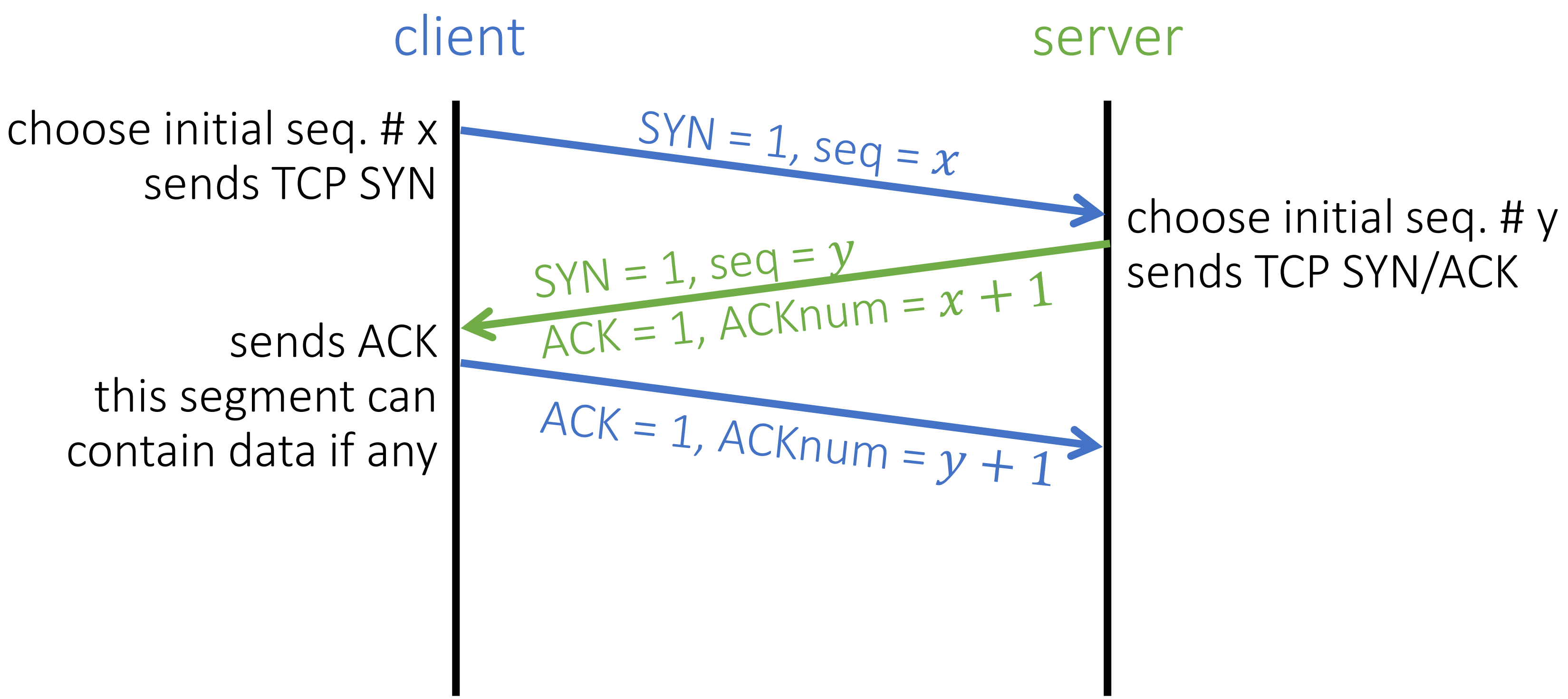
TCP Connection Establishment



TCP Connection Establishment

Established using a 3-way handshake

- Agree on connection and exchange parameters



Half-open Connections

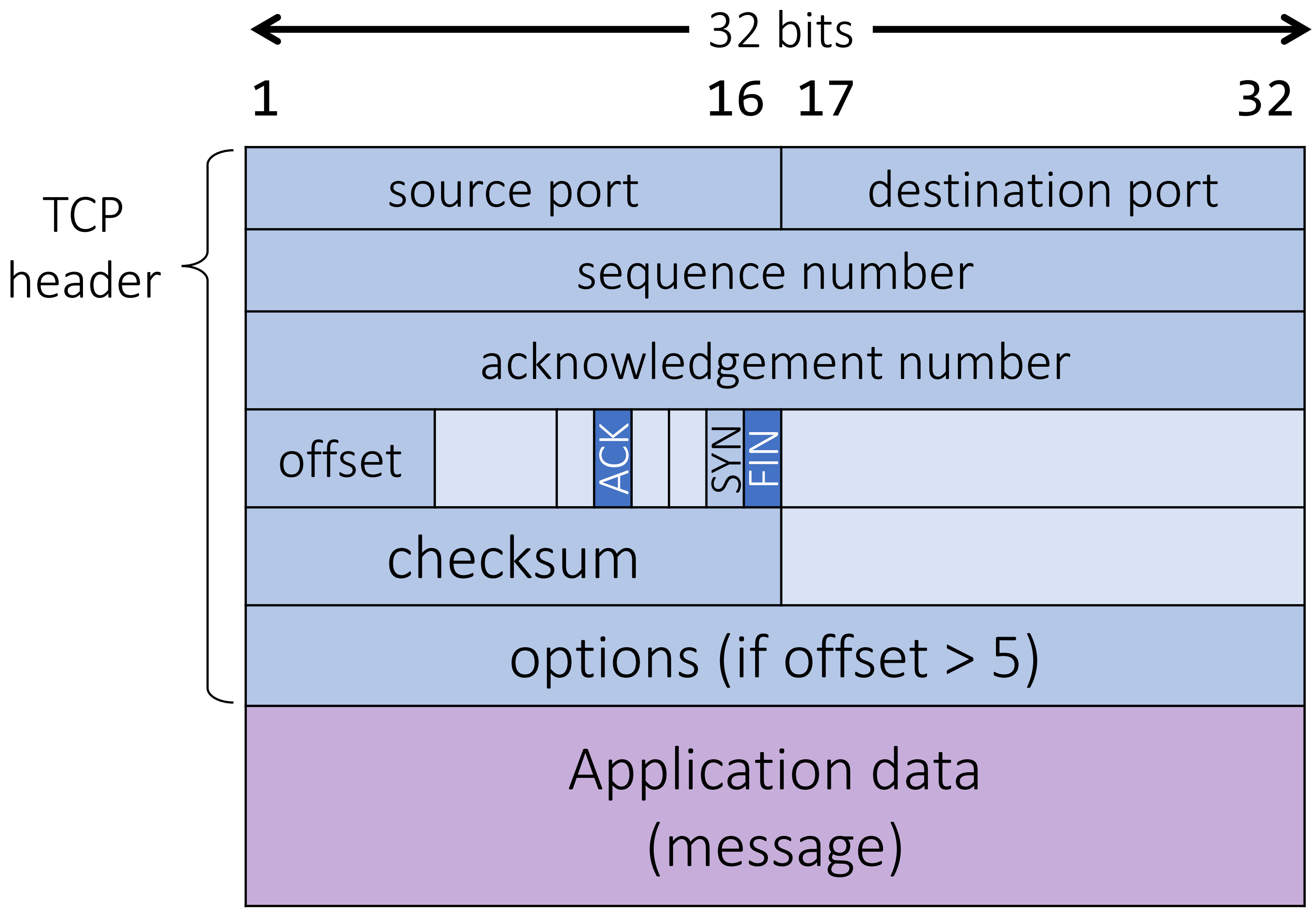
SYN Flooding

- DoS style attack by sending SYN

SYN/ACK Flooding

- DoS style to overwhelm network

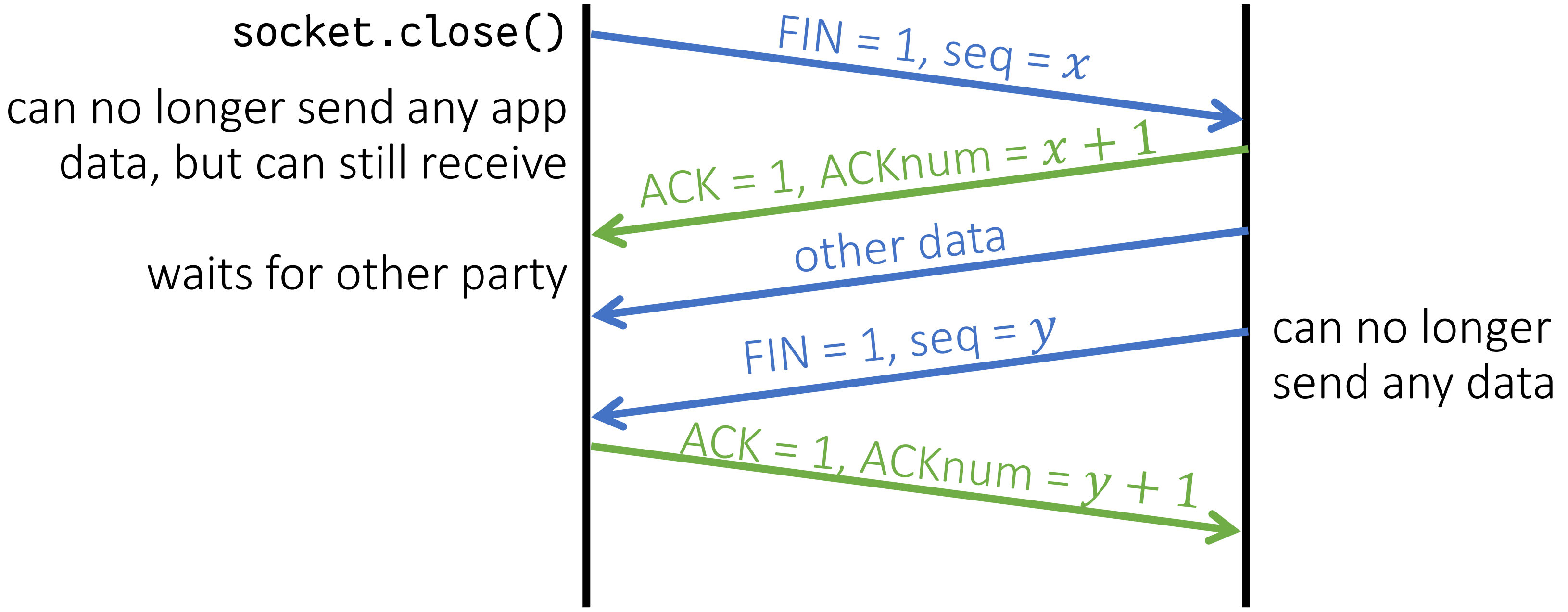
TCP Closing Connection



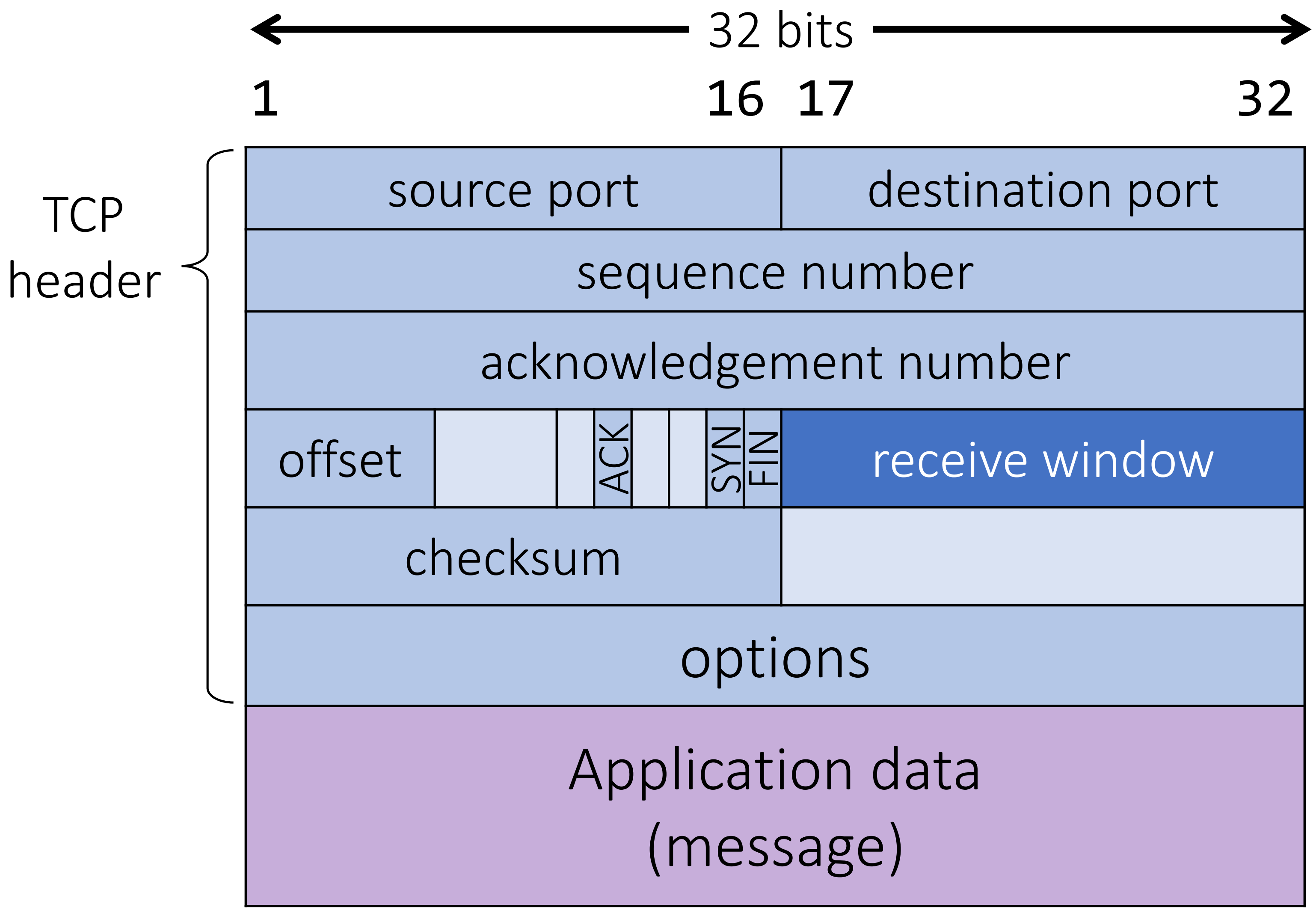
TCP Closing Connection

Each side closes their own side of the connection

- Send segments with FIN bit set
- No more sending of data after FIN, but can still receive

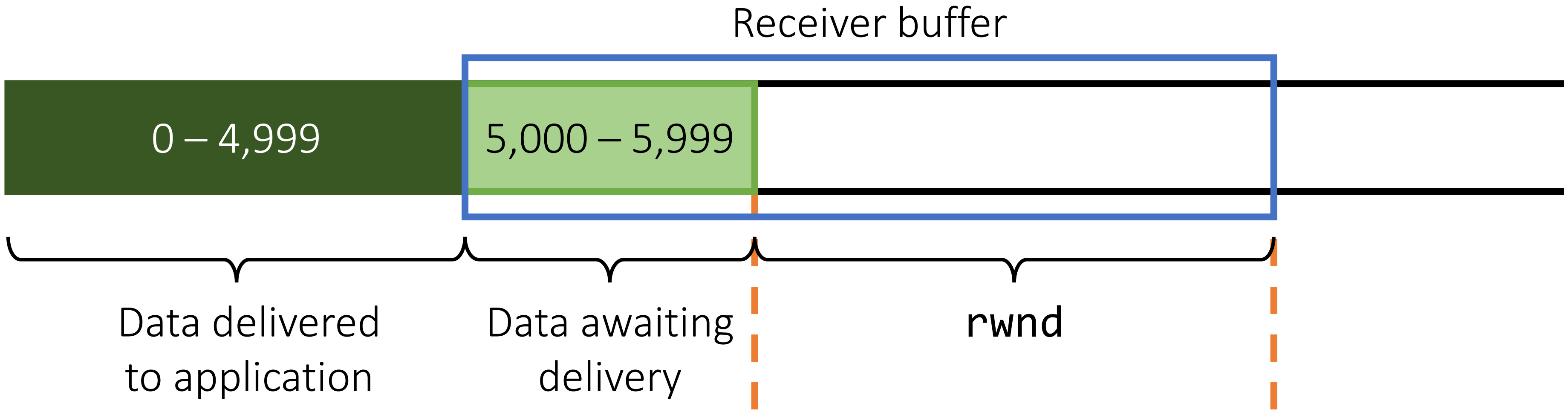


TCP Flow Control

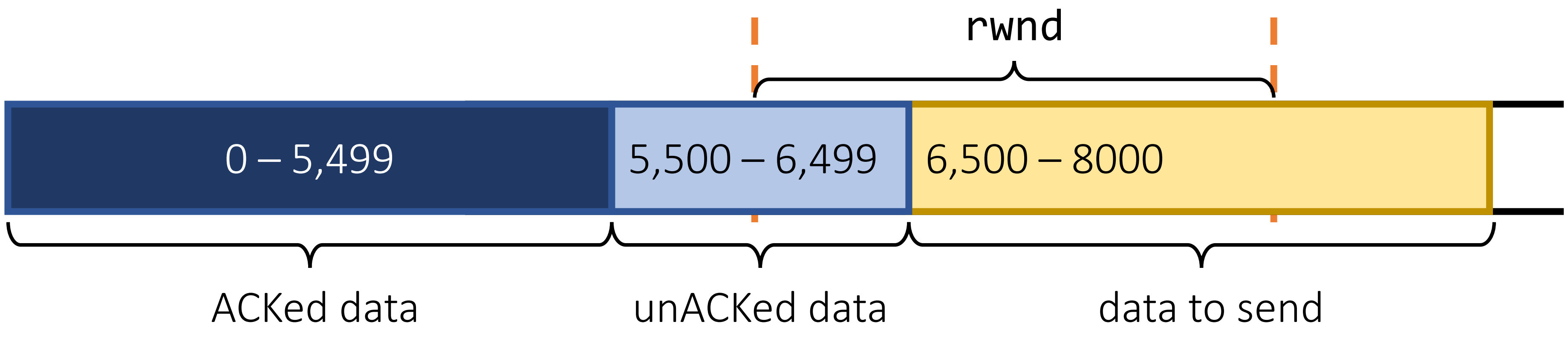


TCP Flow Control

Receiver buffers data to application

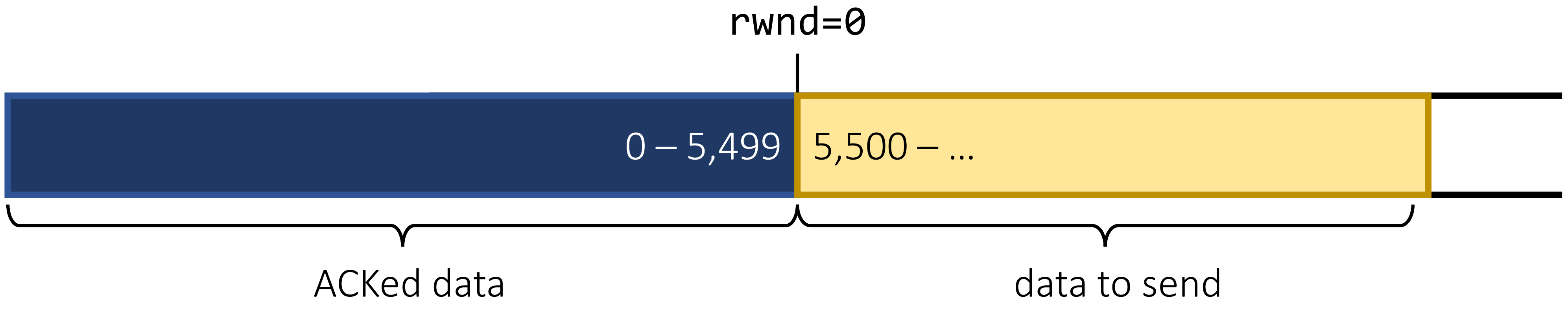


Tells sender how much data it can send



TCP Flow Control

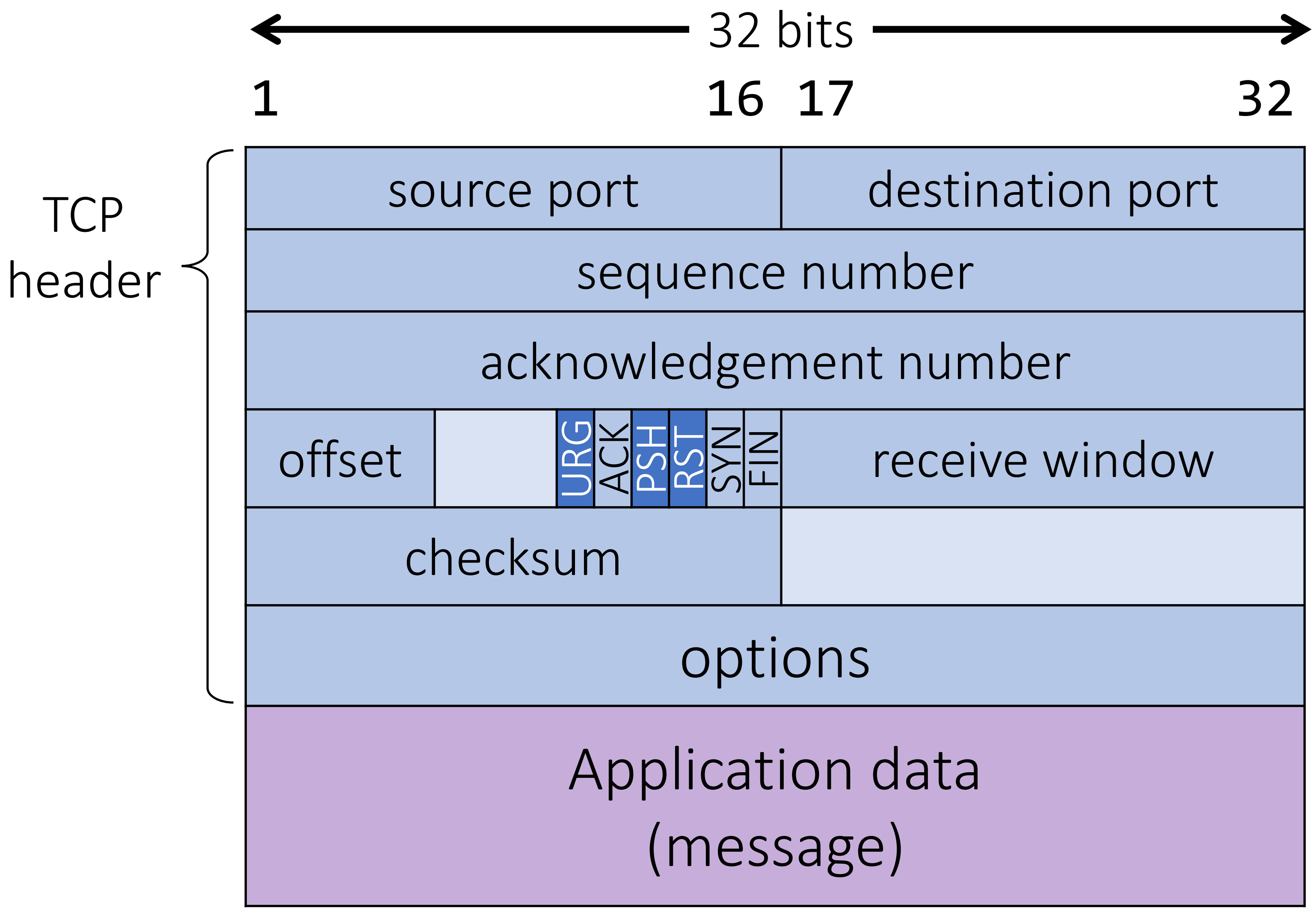
What if rwnd is 0, i.e., full?



How does the sender knows when it empties?

- Sender sends a 0-data segment
- a.k.a zero-window probe

TCP segment structure



What we did not cover

TCP congestion control (Chapter 3.6 & 3.7)

- Be polite and send less if network is congested

Other TCP options

- e.g., SACK, Timestamp

Other TCP techniques

- e.g., FACK, DSACK

Summary

Connectionless: UDP

- Segment structure
- Computing Checksum

Connection-oriented: TCP

- Segment structure
- Reliable data transfer
- Setting and updating RTO
- Establishing and closing connection
- Flow control