Lecture 11: Transport Layer
TCP again

COMP 332, Spring 2024
Victoria Manfredi

WESLEYAN
UNIVERSITY

Acknowledgements: materials adapted from Computer Networking: A Top Down Approach 7th edition: ©1996-2016, J.F Kurose and K.W. Ross, All Rights Reserved as well as from slides by Abraham Matta at Boston University, and some material from Computer Networks by Tannenbaum and Wetherall.

Today

Announcements

- homework 5 due Thursday at 11:59p
- Midterm is Wed after break (will talk more next class)

TCP

- overview
- reliable data transfer
- seq #s and ack #s
- timeouts
- reliable data transport
- connection management

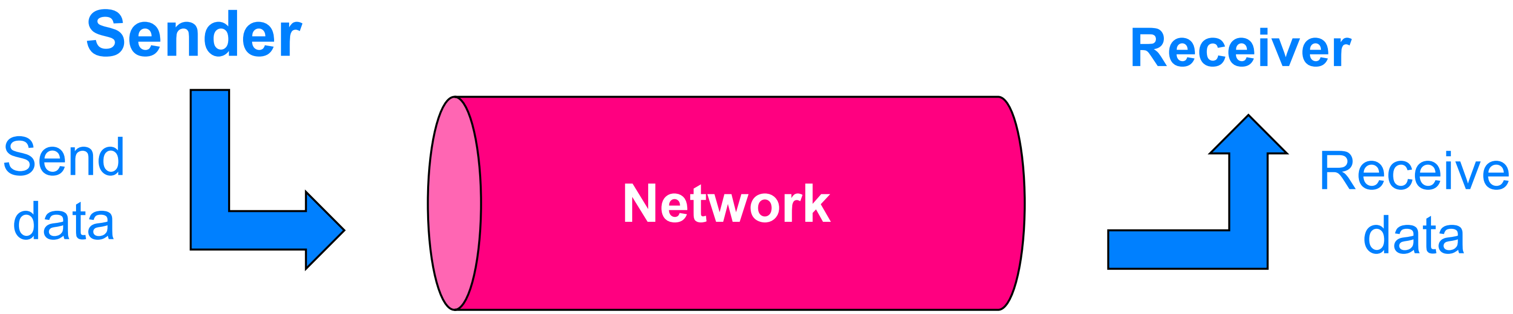
TCP
OVERVIEW

Transmission Control Protocol (TCP)

RFCs: 793,1122,1323, 2018, 2581

Main transport protocol used in Internet, provides

- mux/dmux: which packets go where
- connection-oriented, point-to-point
  - 2 hosts set up connection before exchanging data, tear down after
  - bidirectional data flow (full duplex)
- flow control: don't overwhelm receiver
- congestion control: don't overwhelm network
- reliable: resends lost packets, checks for and corrects errors
- in-order: buffers data until sequential chunk to pass up
- byte stream: no msg boundaries, data treated as stream



How does TCP provide these services?

Using many techniques we already talked about

Sliding window

- congestion and flow control determine window size
- seq #s are byte offsets

Cumulative ACKs but does not drop out-of-order packets

- **only one retransmission timer**
  - intuitively, associate with oldest unACKed packet
- **timeout period**
  - estimated from observations
- **fast retransmit**
  - 3 duplicate ACKs trigger early retransmit

TCP is not perfect but works pretty well!

TCP segment structure

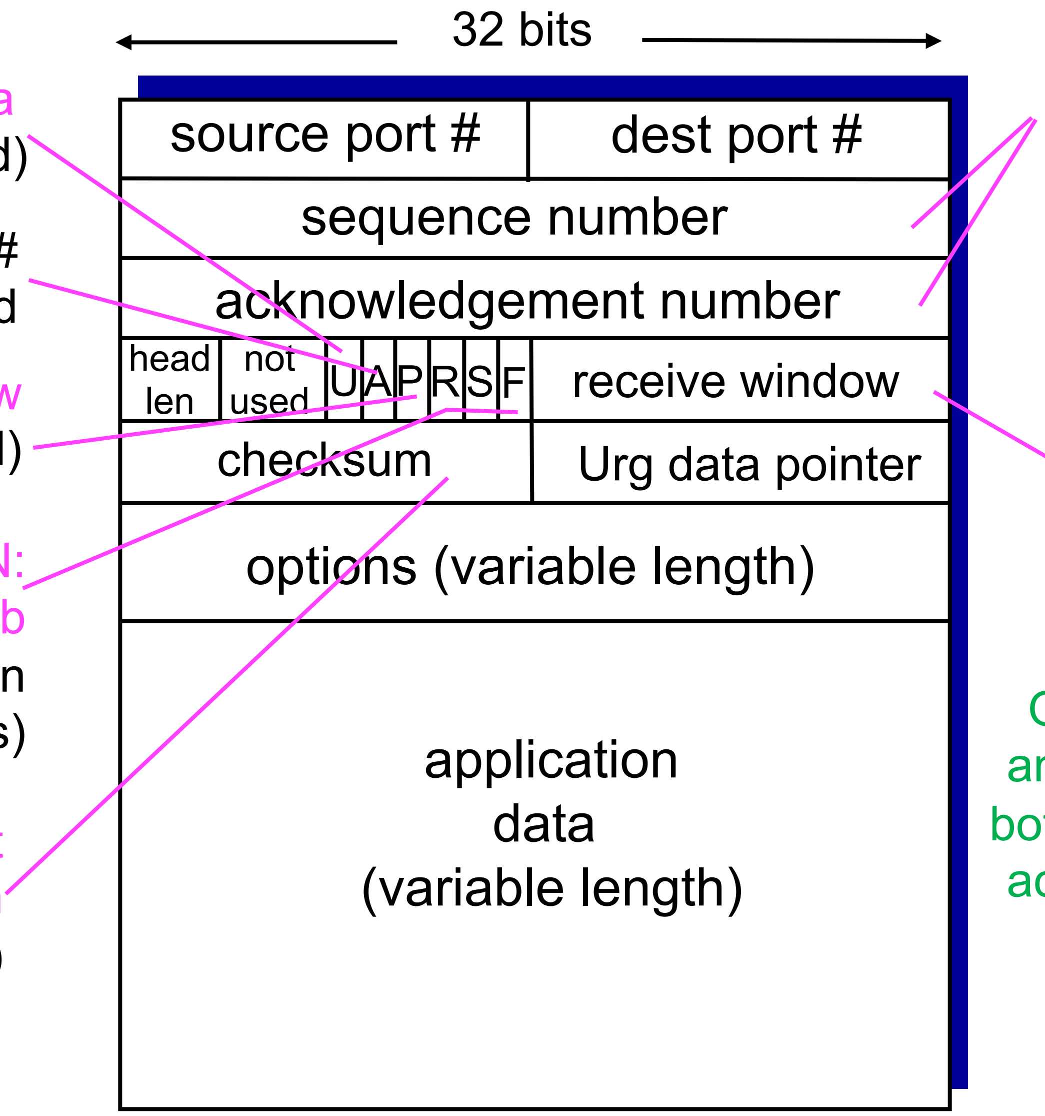
URG: urgent data
(generally not used)

ACK: ACK #
valid

PSH: push data now
(generally not used)

RST, SYN, FIN:
connection estab
(setup, teardown
commands)

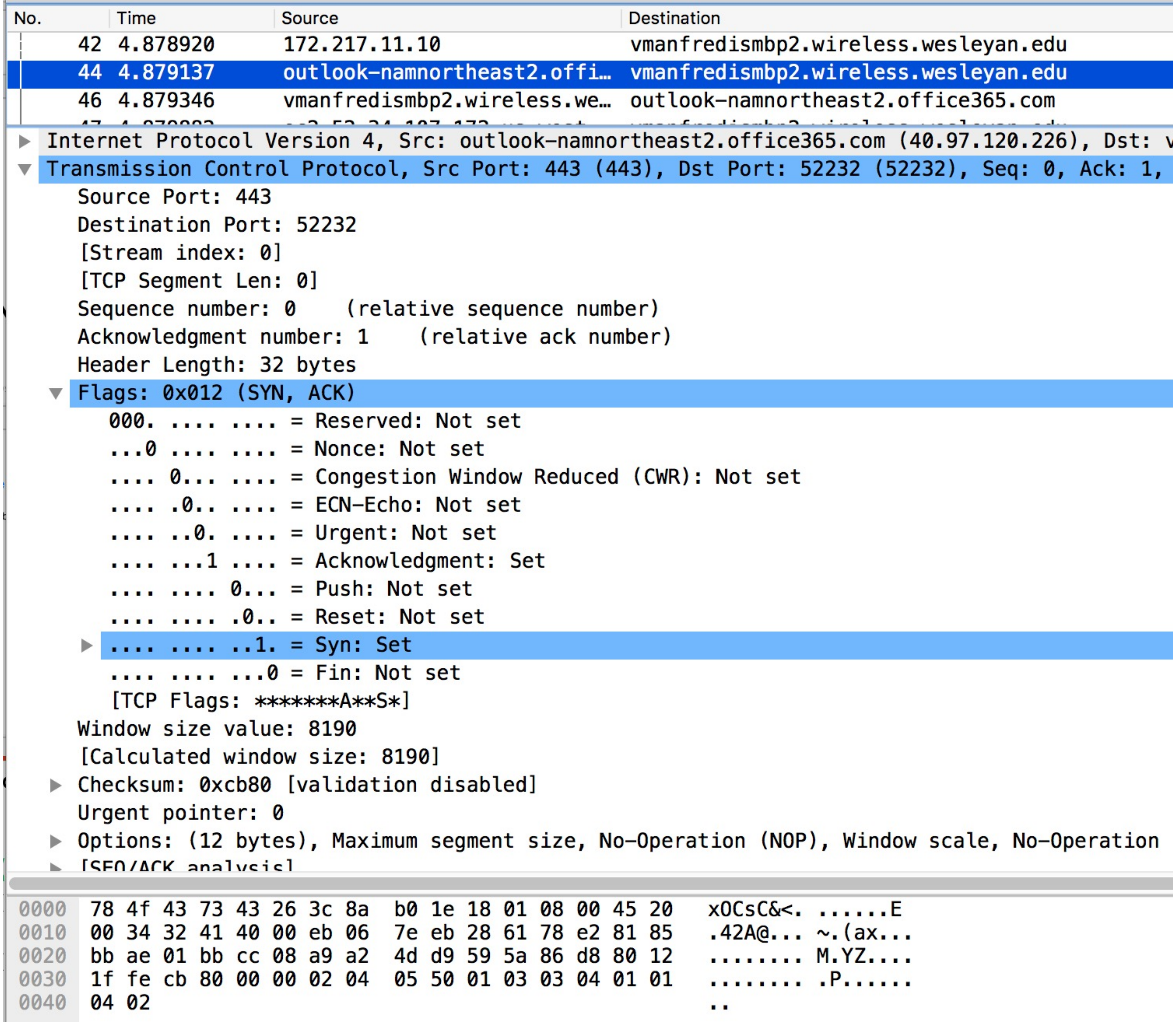
Internet
checksum
(as in UDP)



counting
by bytes
of data
(not segments!)

# bytes
rcvr willing
to accept

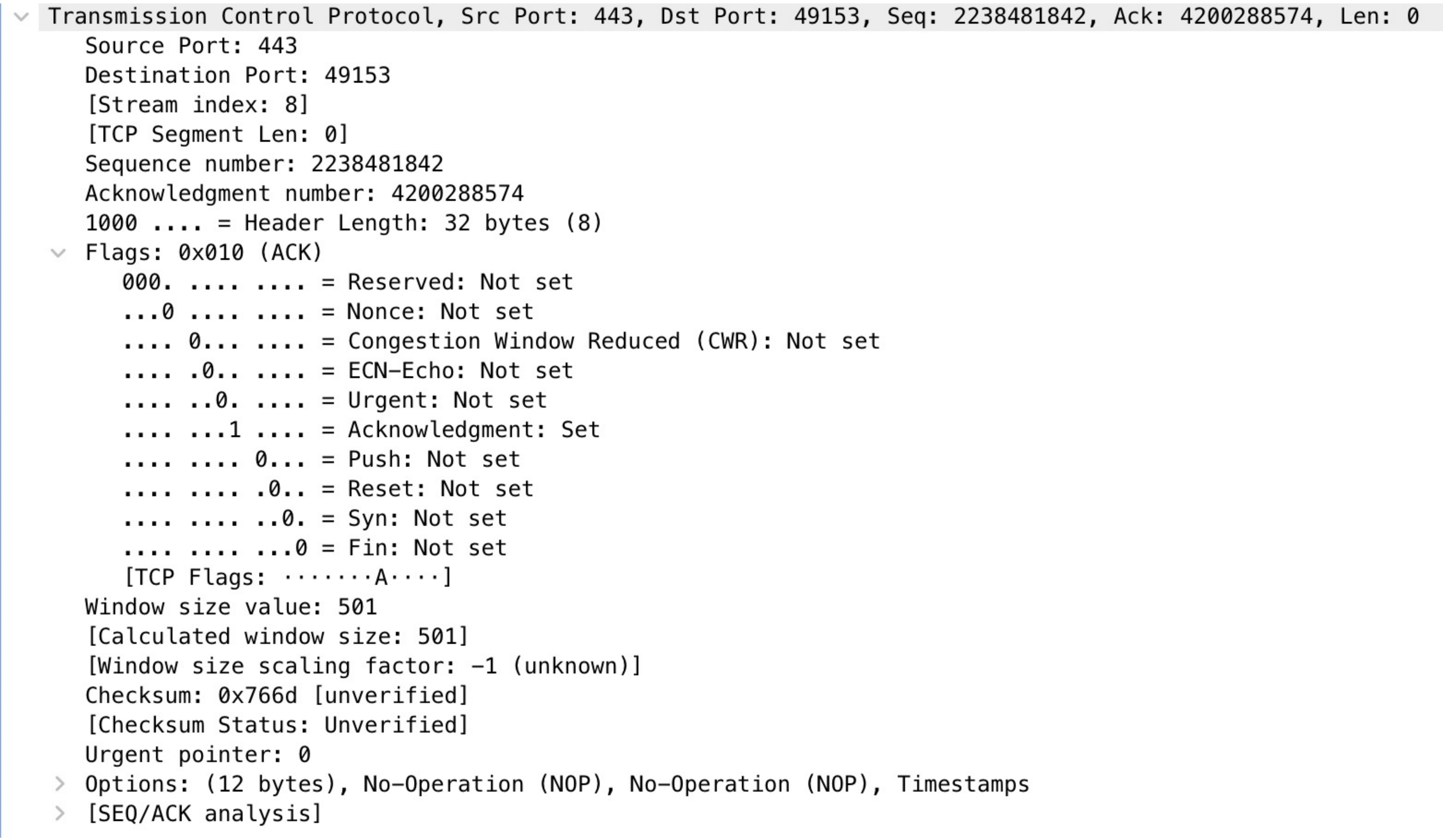
Q: Why both seq #
and ack #? Could be
both sending data and
acking received data



```
No.   Time        Source                          Destination
   42 4.878920    172.217.11.10                   vmanfredismbp2.wireless.wesleyan.edu
   44 4.879137    outlook-namnortheast2.offi…     vmanfredismbp2.wireless.wesleyan.edu
   46 4.879346    vmanfredismbp2.wireless.we…     outlook-namnortheast2.office365.com
   47 4.879882    ec2-52-24-107-172.us-west…      vmanfredismbp2.wireless.wesleyan.edu
```

```
▶ Internet Protocol Version 4, Src: outlook-namnortheast2.office365.com (40.97.120.226), Dst: v
▼ Transmission Control Protocol, Src Port: 443 (443), Dst Port: 52232 (52232), Seq: 0, Ack: 1,
    Source Port: 443
    Destination Port: 52232
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 0    (relative sequence number)
    Acknowledgment number: 1    (relative ack number)
    Header Length: 32 bytes
  ▼ Flags: 0x012 (SYN, ACK)
      000. .... .... = Reserved: Not set
      ...0 .... .... = Nonce: Not set
      .... 0... .... = Congestion Window Reduced (CWR): Not set
      .... .0.. .... = ECN-Echo: Not set
      .... ..0. .... = Urgent: Not set
      .... ...1 .... = Acknowledgment: Set
      .... .... 0... = Push: Not set
      .... .... .0.. = Reset: Not set
    ▶ .... .... ..1. = Syn: Set
      .... .... ...0 = Fin: Not set
      [TCP Flags: *******A**S*]
    Window size value: 8190
    [Calculated window size: 8190]
  ▶ Checksum: 0xcb80 [validation disabled]
    Urgent pointer: 0
  ▶ Options: (12 bytes), Maximum segment size, No-Operation (NOP), Window scale, No-Operation
  ▶ [SEQ/ACK analysis]
```

```
0000  78 4f 43 73 43 26 3c 8a  b0 1e 18 01 08 00 45 20   xOCsC&<. ......E
0010  00 34 32 41 40 00 eb 06  7e eb 28 61 78 e2 81 85   .42A@... ~.(ax...
0020  bb ae 01 bb cc 08 a9 a2  4d d9 59 5a 86 d8 80 12   ........ M.YZ....
0030  1f fe cb 80 00 00 02 04  05 50 01 03 03 04 01 01   ........ .P......
0040  04 02                                              ..
```



```
Transmission Control Protocol, Src Port: 443, Dst Port: 49153, Seq: 2238481842, Ack: 4200288574, Len: 0
   Source Port: 443
   Destination Port: 49153
   [Stream index: 8]
   [TCP Segment Len: 0]
   Sequence number: 2238481842
   Acknowledgment number: 4200288574
   1000 .... = Header Length: 32 bytes (8)
   Flags: 0x010 (ACK)
      000. .... .... = Reserved: Not set
      ...0 .... .... = Nonce: Not set
      .... 0... .... = Congestion Window Reduced (CWR): Not set
      .... .0.. .... = ECN-Echo: Not set
      .... ..0. .... = Urgent: Not set
      .... ...1 .... = Acknowledgment: Set
      .... .... 0... = Push: Not set
      .... .... .0.. = Reset: Not set
      .... .... ..0. = Syn: Not set
      .... .... ...0 = Fin: Not set
      [TCP Flags: ·······A····]
   Window size value: 501
   [Calculated window size: 501]
   [Window size scaling factor: -1 (unknown)]
   Checksum: 0x766d [unverified]
   [Checksum Status: Unverified]
   Urgent pointer: 0
   Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
   [SEQ/ACK analysis]
```

TCP

SEQ #S AND ACK #S

TCP seq. numbers, ACKs

Sequence #s

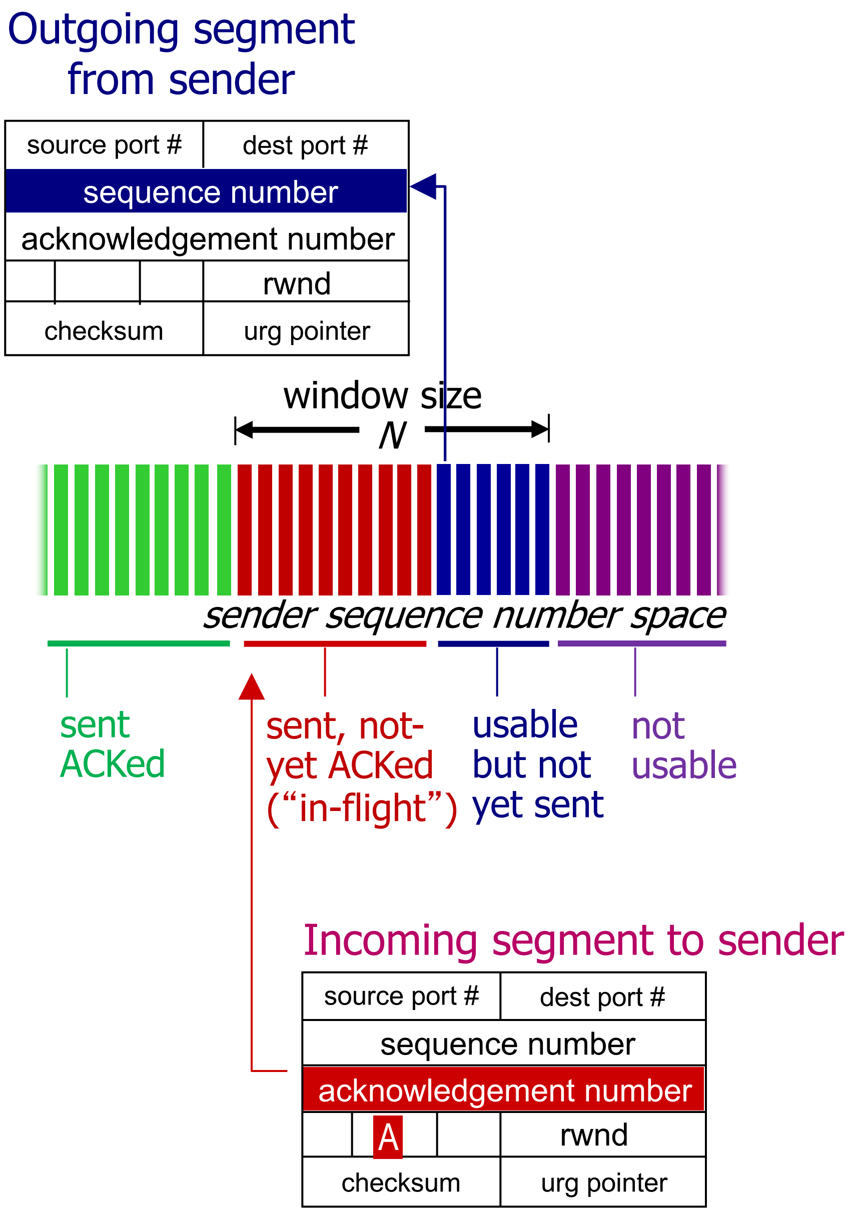
- byte stream # of first byte in segment's data

Acknowledgements

- seq # of next byte expected from other side
- cumulative ACK

Q: how does receiver handle out-of-order segments?

- TCP spec doesn't say
- up to implementer

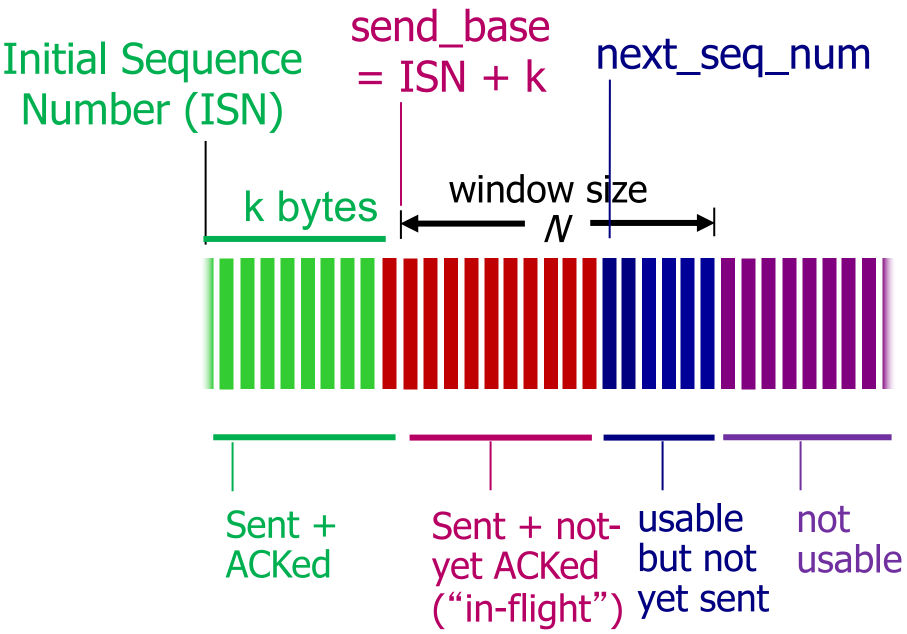


TCP ACKs

Cumulative ACKs (but different than in Go-Back-N)

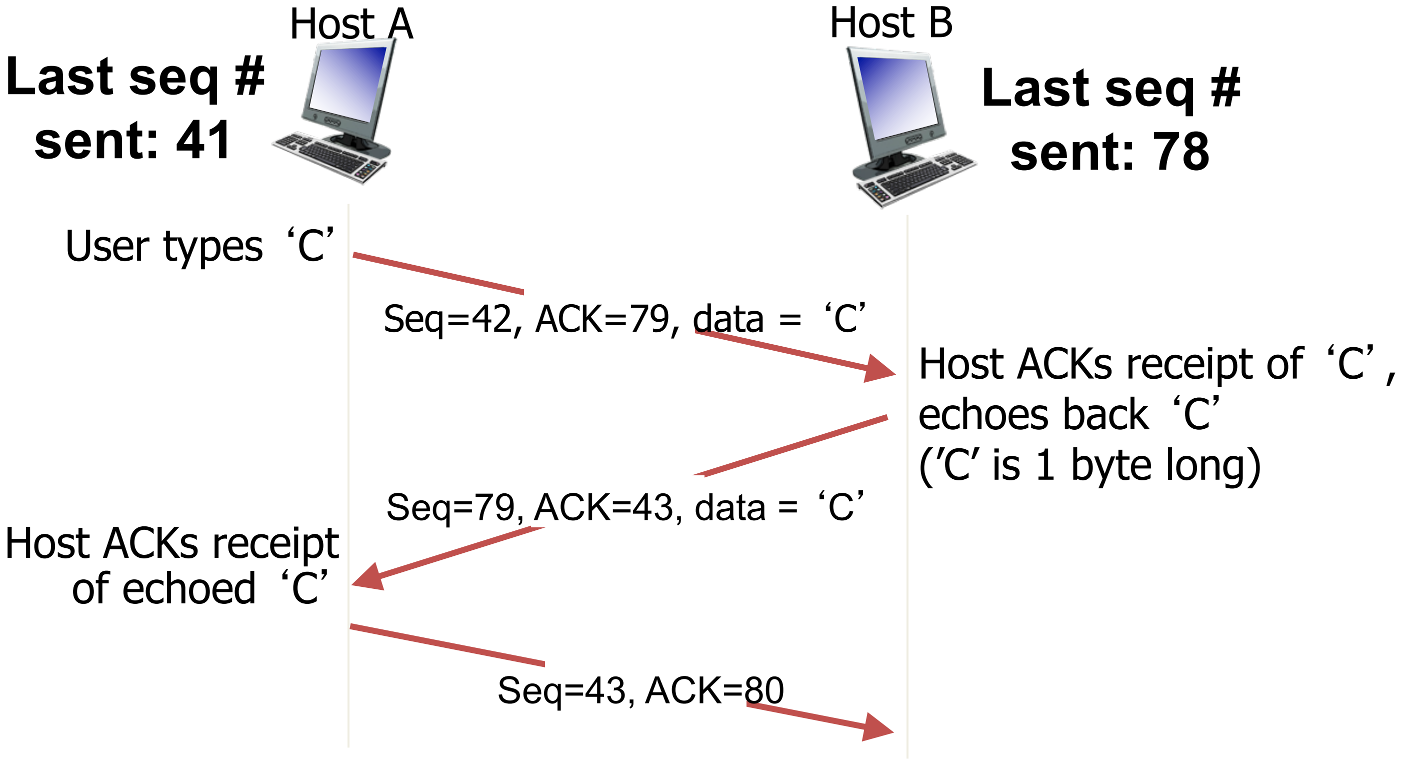
- ACKs what receiver expects next, not last packet received
  - implicitly also ACKs everything up to sequence number received
- only 1 retransmission timer (for first pkt in window)
  - sender retransmits only first pkt in window if no ack when timer expires

Sequence #s are not sequential: counting bytes not packets



TCP seq. numbers, ACKs

Sequence numbers are synchronized during connection set-up



Host A

Last seq # sent: 41

Host B

Last seq # sent: 78

User types ‘C’

Seq=42, ACK=79, data = ‘C’

Host ACKs receipt of ‘C’, echoes back ‘C’ (’C’ is 1 byte long)

Seq=79, ACK=43, data = ‘C’

Host ACKs receipt of echoed ‘C’

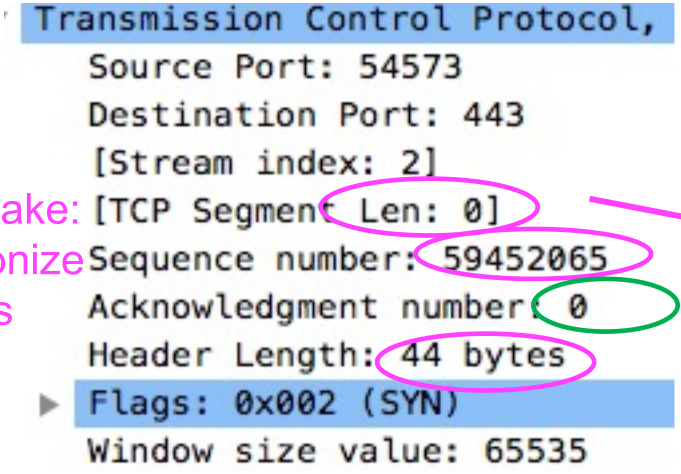
Seq=43, ACK=80

Simple nc scenario

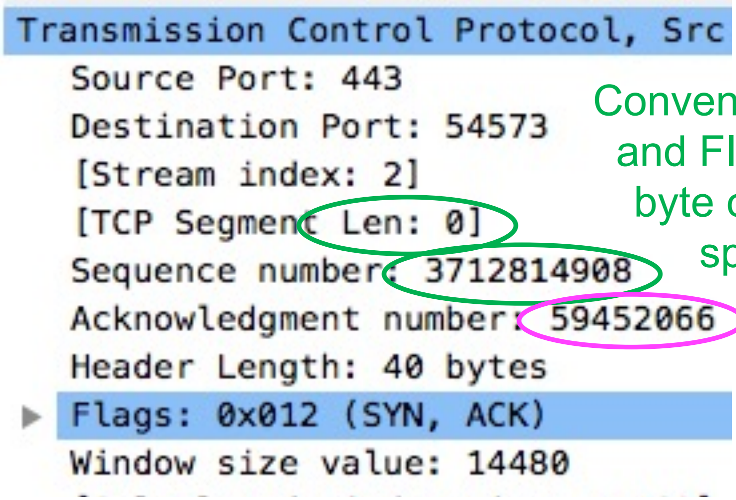
Host 1

Host 2

Handshake: Synchronize ISNs



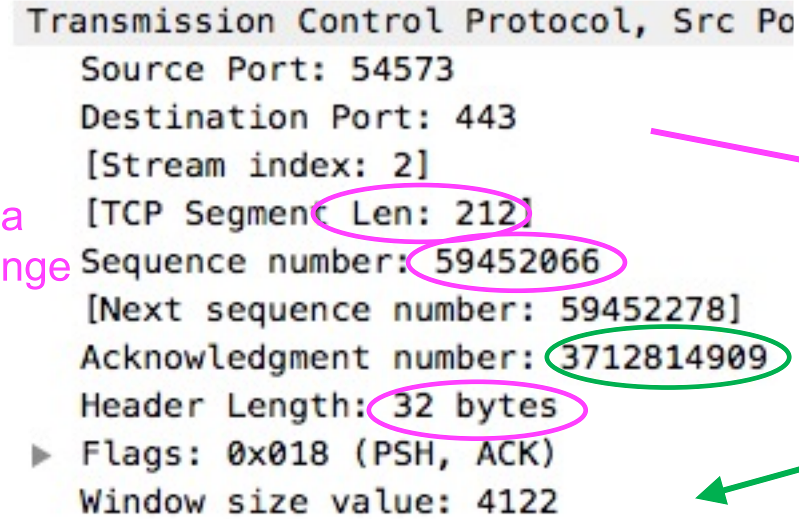
```
Transmission Control Protocol,
    Source Port: 54573
    Destination Port: 443
    [Stream index: 2]
    [TCP Segment Len: 0]
    Sequence number: 59452065
    Acknowledgment number: 0
    Header Length: 44 bytes
  ▶ Flags: 0x002 (SYN)
    Window size value: 65535
```



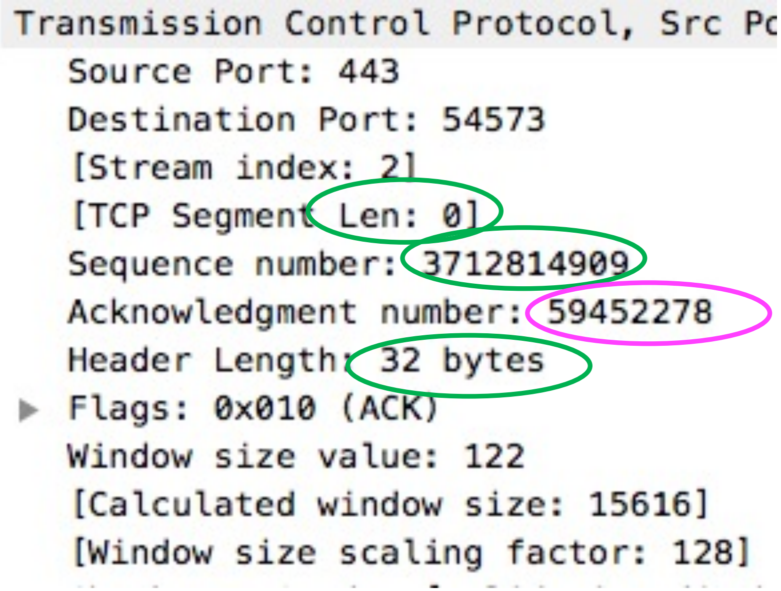
```
Transmission Control Protocol, Src
    Source Port: 443
    Destination Port: 54573
    [Stream index: 2]
    [TCP Segment Len: 0]
    Sequence number: 3712814908
    Acknowledgment number: 59452066
    Header Length: 40 bytes
  ▶ Flags: 0x012 (SYN, ACK)
    Window size value: 14480
```

Convention: SYN and FIN take 1 byte of seq # space

Data exchange



```
Transmission Control Protocol, Src Po
    Source Port: 54573
    Destination Port: 443
    [Stream index: 2]
    [TCP Segment Len: 212]
    Sequence number: 59452066
    [Next sequence number: 59452278]
    Acknowledgment number: 3712814909
    Header Length: 32 bytes
  ▶ Flags: 0x018 (PSH, ACK)
    Window size value: 4122
```



```
Transmission Control Protocol, Src Po
    Source Port: 443
    Destination Port: 54573
    [Stream index: 2]
    [TCP Segment Len: 0]
    Sequence number: 3712814909
    Acknowledgment number: 59452278
    Header Length: 32 bytes
  ▶ Flags: 0x010 (ACK)
    Window size value: 122
    [Calculated window size: 15616]
    [Window size scaling factor: 128]
```

What are seq and ack #s in next segment from receiver?

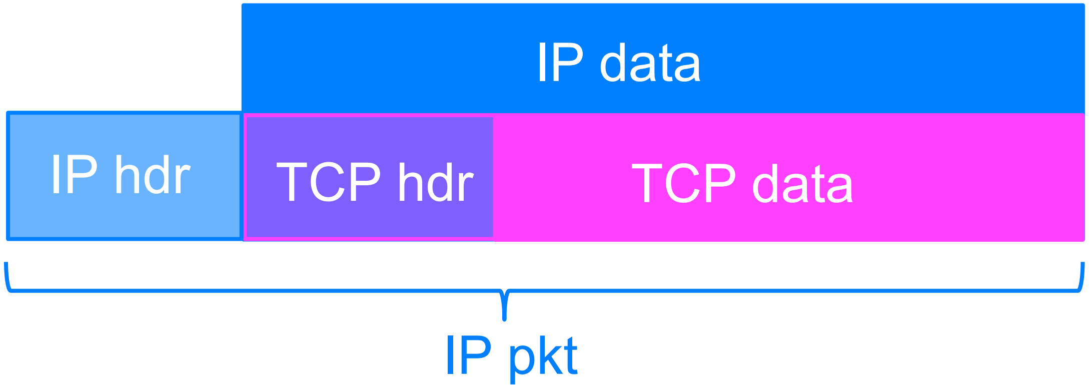
Segment size

Max length of IP packet in bytes

- MTU: Maximum Transmission Unit
- 1500 bytes if Ethernet used as link layer protocol

Max length of TCP data in bytes

- MSS: Maximum Segment Size
- MSS = MTU – IP hdr – TCP hdr
  - TCP header >= 20bytes



TCP segment sent when either it is full (meets MSS) or not full but timeout occurs

TCP
TIMEOUTS

TCP timeout

Q: how to set TCP timeout value?

Longer than RTT (ideally proportional)

- but RTT varies ….

Too short

- premature timeout
- unnecessary retransmissions

Too long

- slow reaction to segment loss

How to estimate RTT

SampleRTT

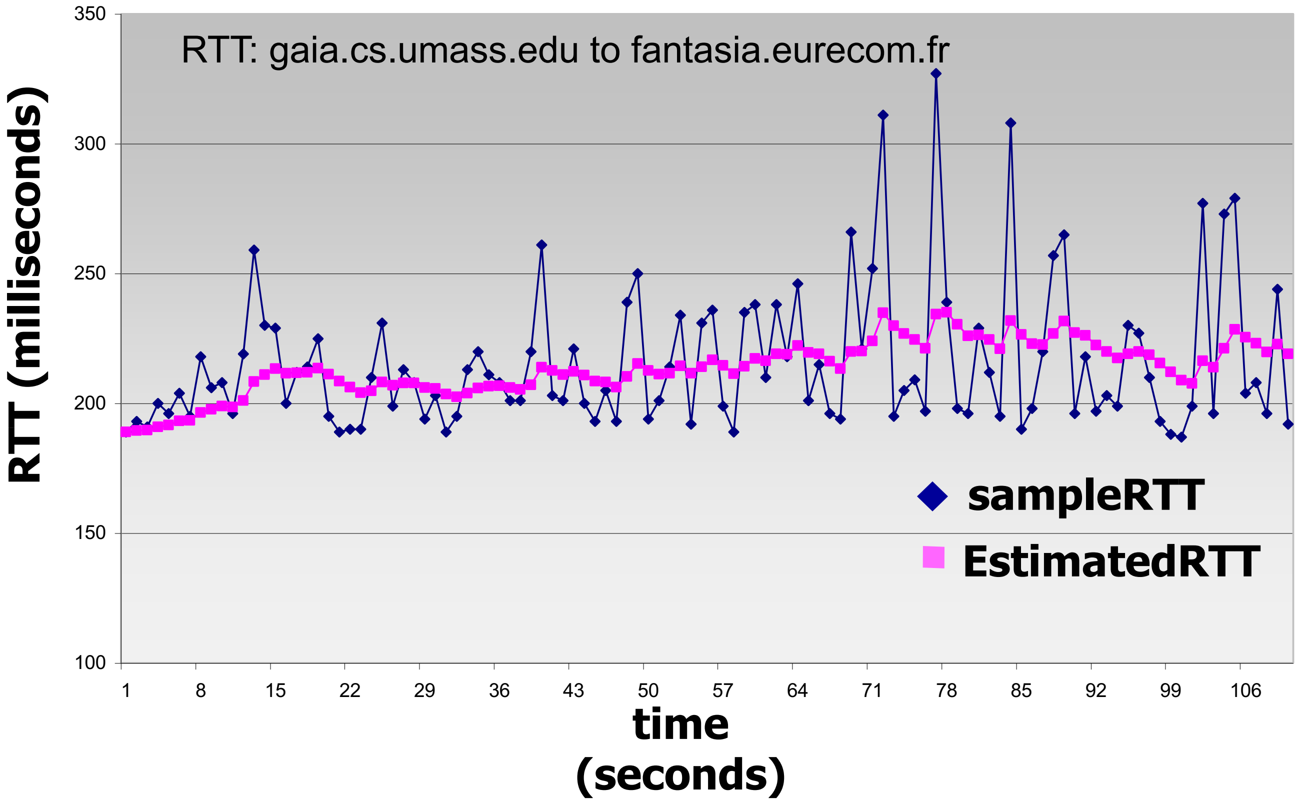
- time from segment transmission to ACK reception
- ignore retransmissions
  - since problems associating retransmitted ACK with right pkt
  - will vary: use average of several measurements

EstimatedRTT

- exponential weighted moving average of sampleRTTs
- influence of past sample decreases exponentially fast
- typical value: $\alpha = 0.125$

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

Variation in RTT



Q: How to handle variation in RTT?

– timeout interval should be ≥ EstimatedRTT
  - because of variation of RTT values
  - large variation in EstimatedRTT ⇒ larger safety margin

Handling variation in RTT

Estimate SampleRTT deviation from EstimatedRTT

$$\text{DevRTT} = (1-\beta)*\text{DevRTT} + \beta*|\text{SampleRTT}-\text{EstimatedRTT}|$$

(typically, $\beta = 0.25$)

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4*\text{DevRTT}$$

“safety margin”

If timeout occurs: timeout interval doubled to prevent premature timeout for subsequent segments

TCP

RELIABLE DATA TRANSFER

TCP reliable data transfer

TCP creates rdt service on top of IP's unreliable service

- pipelined segments
- cumulative acks
- single retransmission timer

Retransmissions triggered by

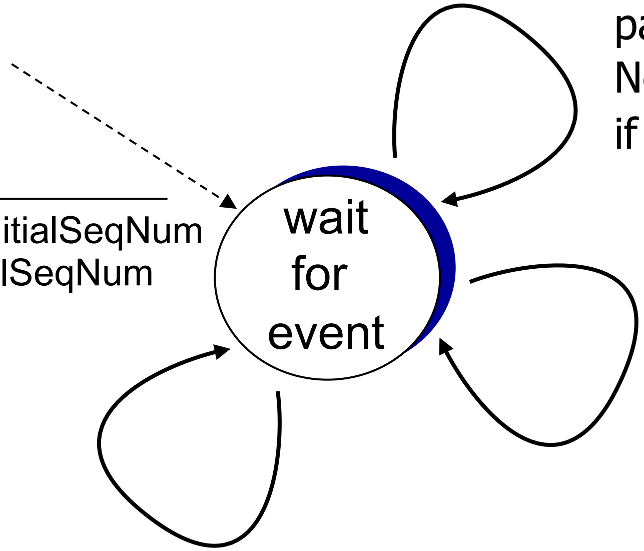
- timeout events
- duplicate ACKs

Let's initially consider simplified TCP sender

- ignore duplicate acks
- ignore flow control, congestion control

TCP sender (simplified)

Seq # is byte-stream # of first data byte in segment. Timer is for oldest unacked segment



Λ

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum
```

data received from application above

```
create segment, seq. #: NextSeqNum
pass segment to IP (i.e., “send”)
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
    start timer
```

timeout

```
retransmit not-yet-acked segment
            with smallest seq. #
start timer
```

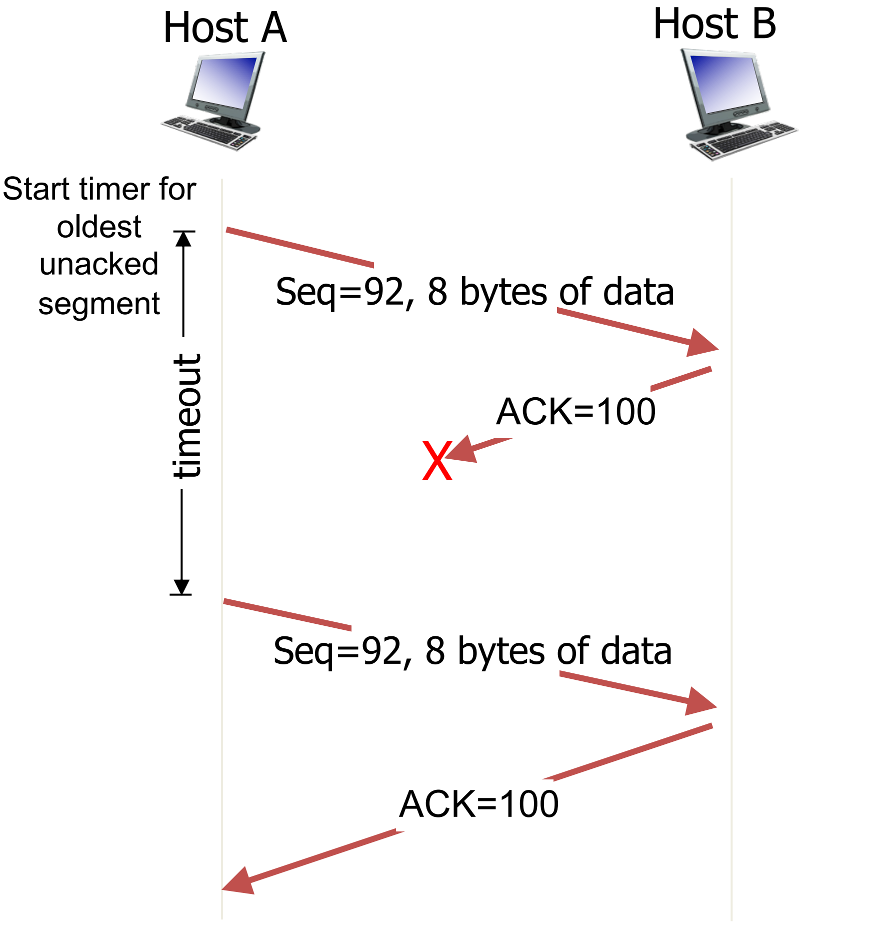
Retransmit first segment in window, restart timer

ACK received, with ACK field value y

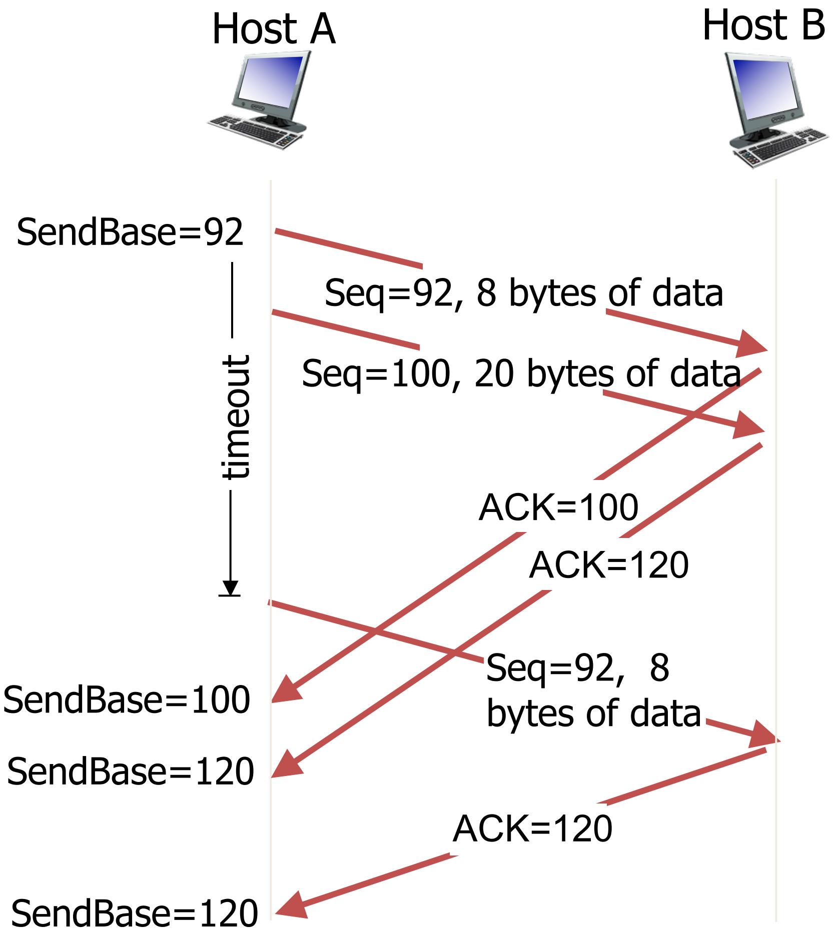
```
if (y > SendBase) {
   SendBase = y
   /* SendBase–1: last cumulatively ACKed byte */
   if (there are currently not-yet-acked segments)
       start timer
     else stop timer
 }
```

If acks previously unacked segments, update what is known to be ACKed, start timer if still unacked segments

TCP: retransmission scenarios

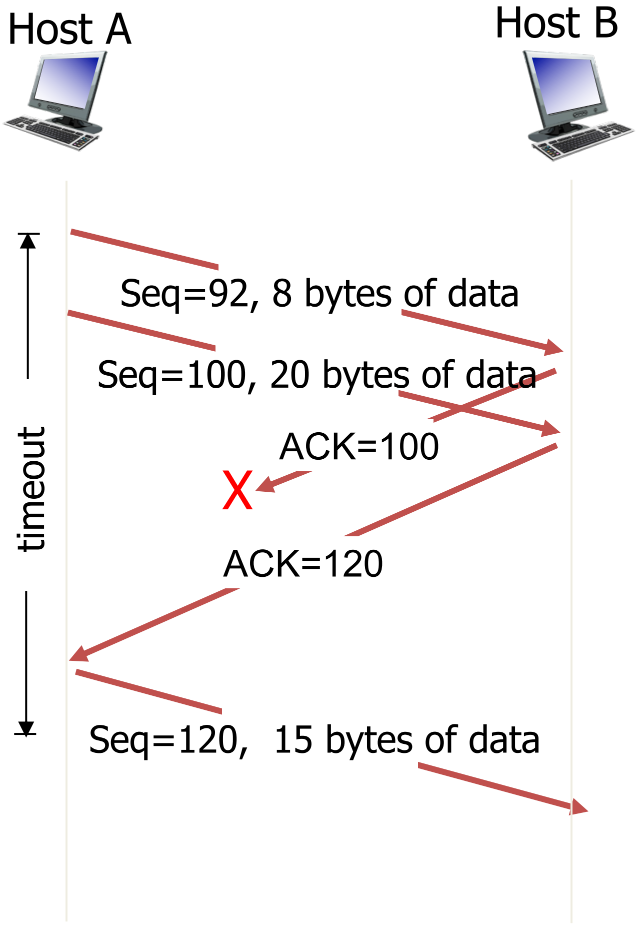


lost ACK scenario



premature timeout

TCP: retransmission scenarios



cumulative ACK

Duplicate ACKs

Time-out period often relatively long

- long delay before resending lost packet

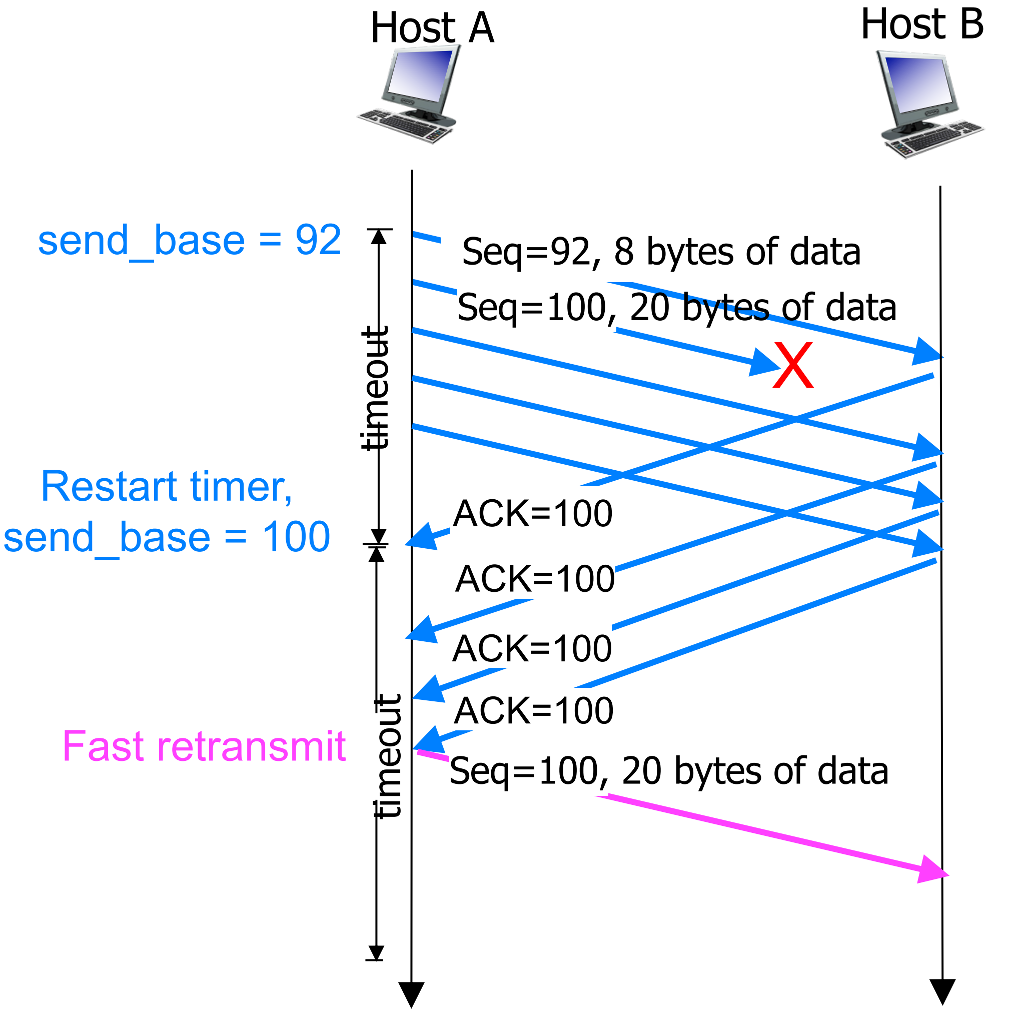
Duplicate ACKs indicate isolated loss

- rather than congestion causing many losses
  - sender often sends many segments back-to-back
  - if segment is lost, likely many duplicate ACKs
  - ACKs being received indicates some packets received at destination since ACK sent for every packet: so not congestion

TCP fast retransmit

- if sender receives 3 ACKs for same data (triple duplicate ACKs)
  - resend unacked segment with smallest seq #
- Q: why 3?
  - pkts may just have been reordered otherwise
  - likely that unacked segment lost, so don’t wait for timeout

TCP fast retransmit



Host A

Host B

send_base = 92

Seq=92, 8 bytes of data

Seq=100, 20 bytes of data

X

timeout

Restart timer,
send_base = 100

ACK=100

ACK=100

ACK=100

ACK=100

Fast retransmit

timeout

Seq=100, 20 bytes of data

fast retransmit after sender
receipt of triple duplicate ACK

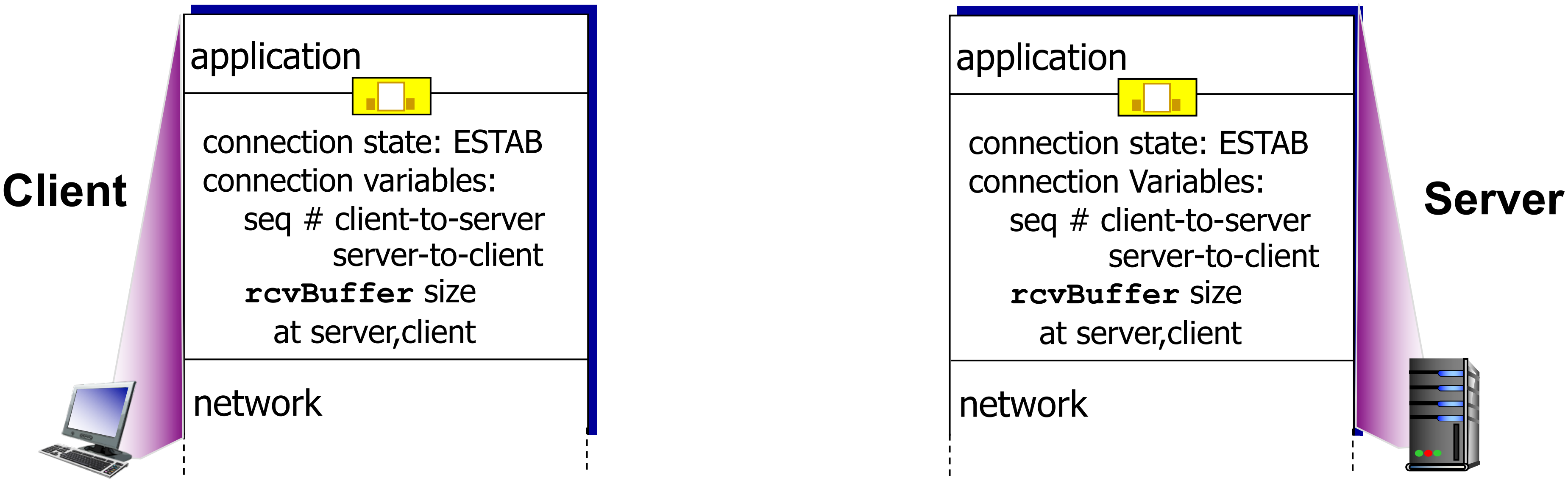
TCP

CONNECTION MANAGEMENT

Connection Management

Before exchanging data, sender/receiver handshake

- establish connection and connection parameters
- tear down connection when done



Client

application

connection state: ESTAB
connection variables:
seq # client-to-server
server-to-client
rcvBuffer size
at server,client

network

Server

application

connection state: ESTAB
connection Variables:
seq # client-to-server
server-to-client
rcvBuffer size
at server,client

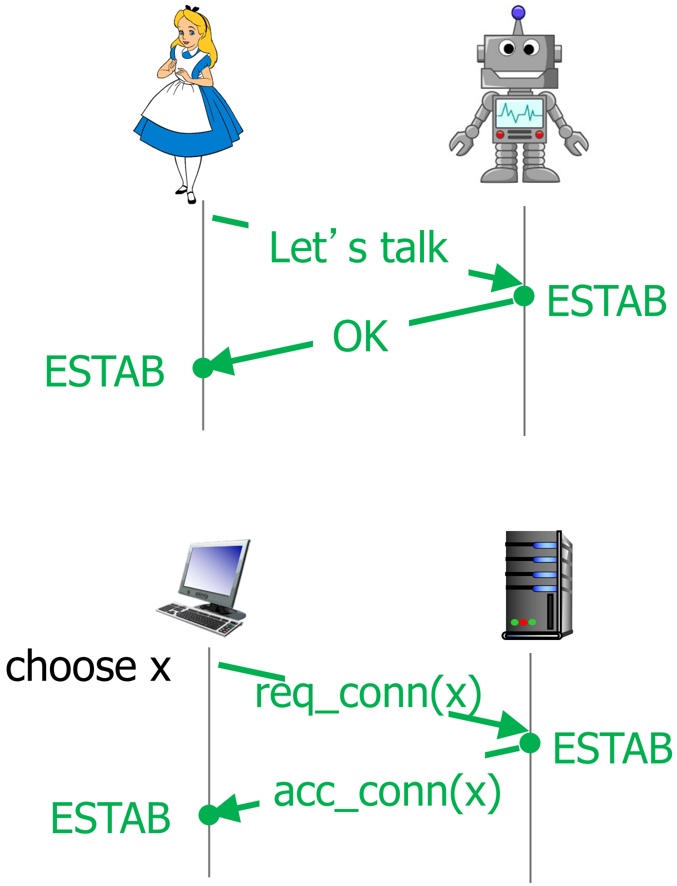
network

sock = sock.connect((host, port))

conn, addr = server_sock.accept()

Agreeing to establish a connection

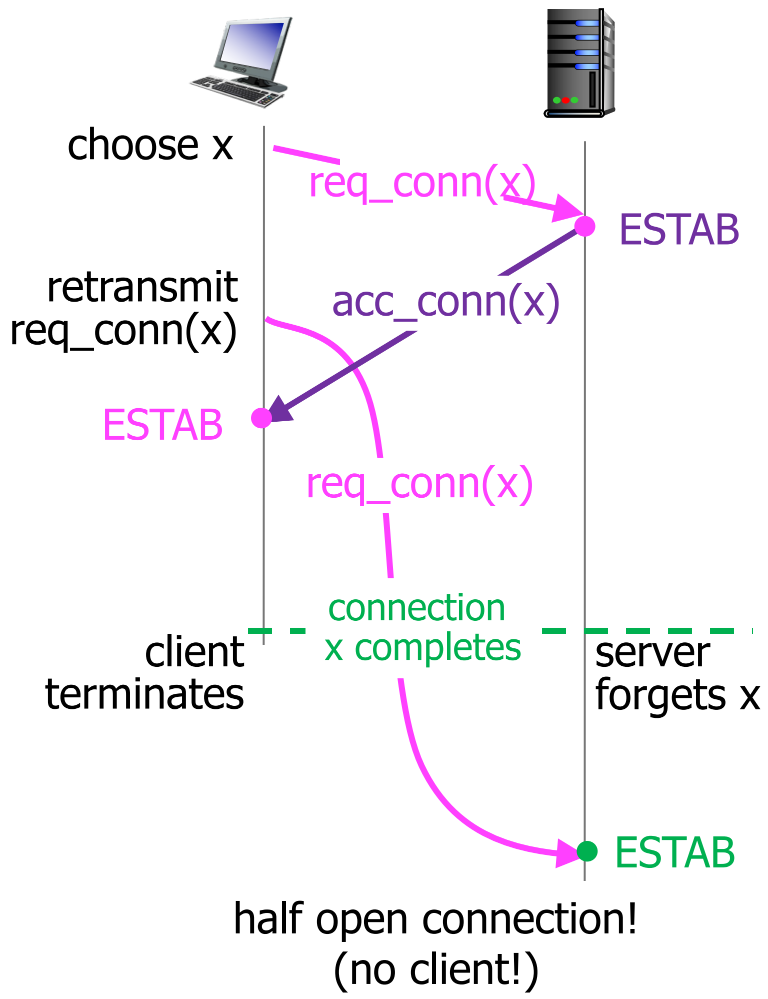
2-way handshake:

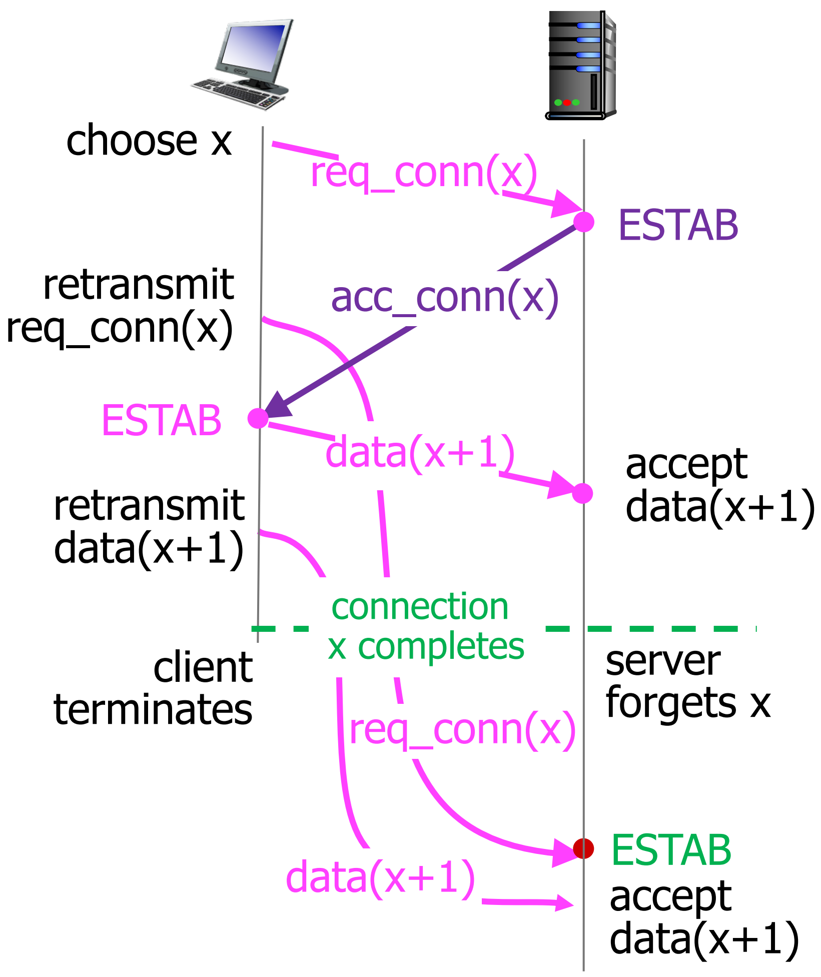


Q: will 2-way handshake always work in network?

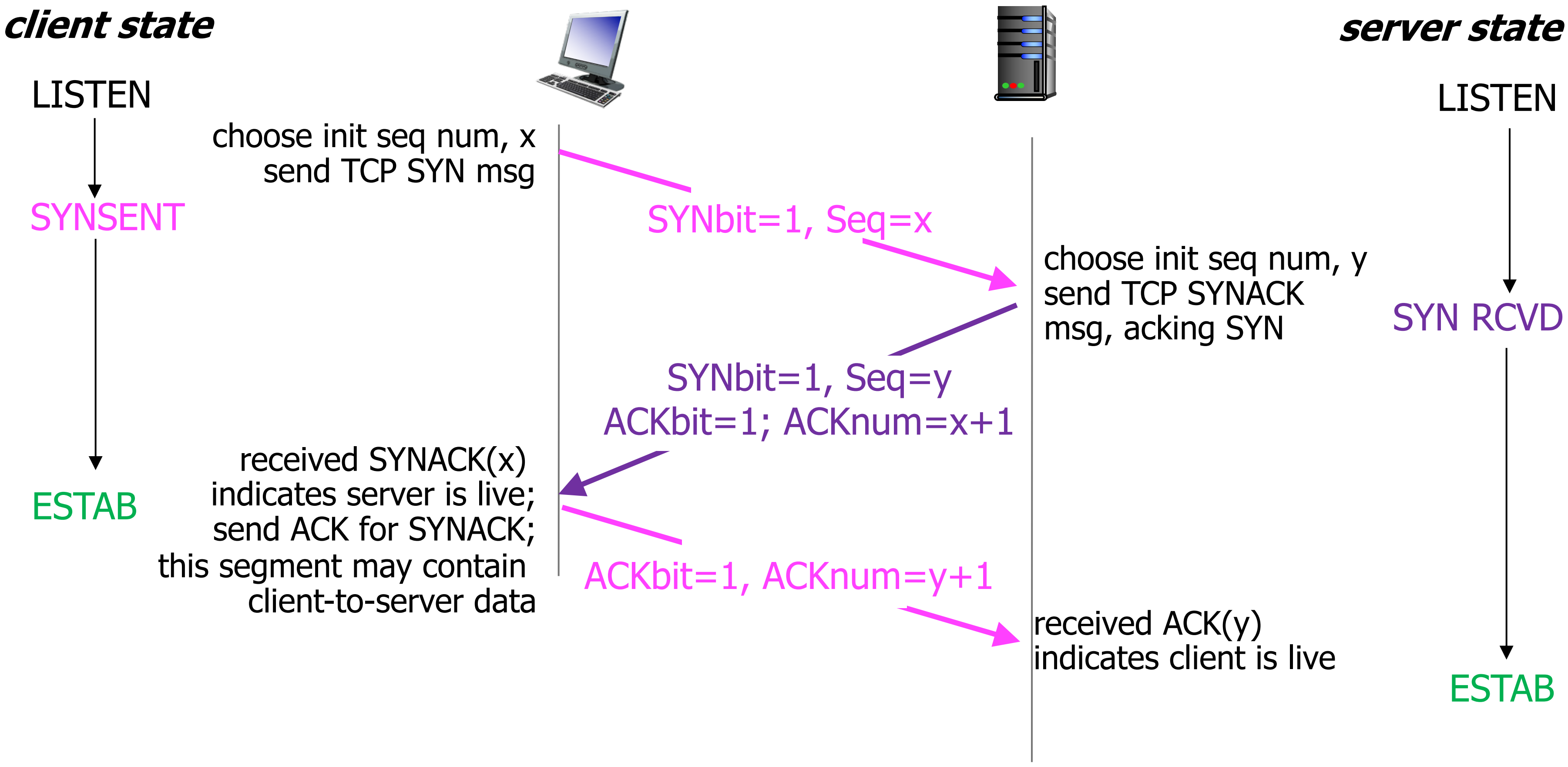
- variable delays
- retransmitted messages
  - e.g. req_conn(x)) due to message loss
- message reordering
- can’t see other side

2-way handshake failure scenarios

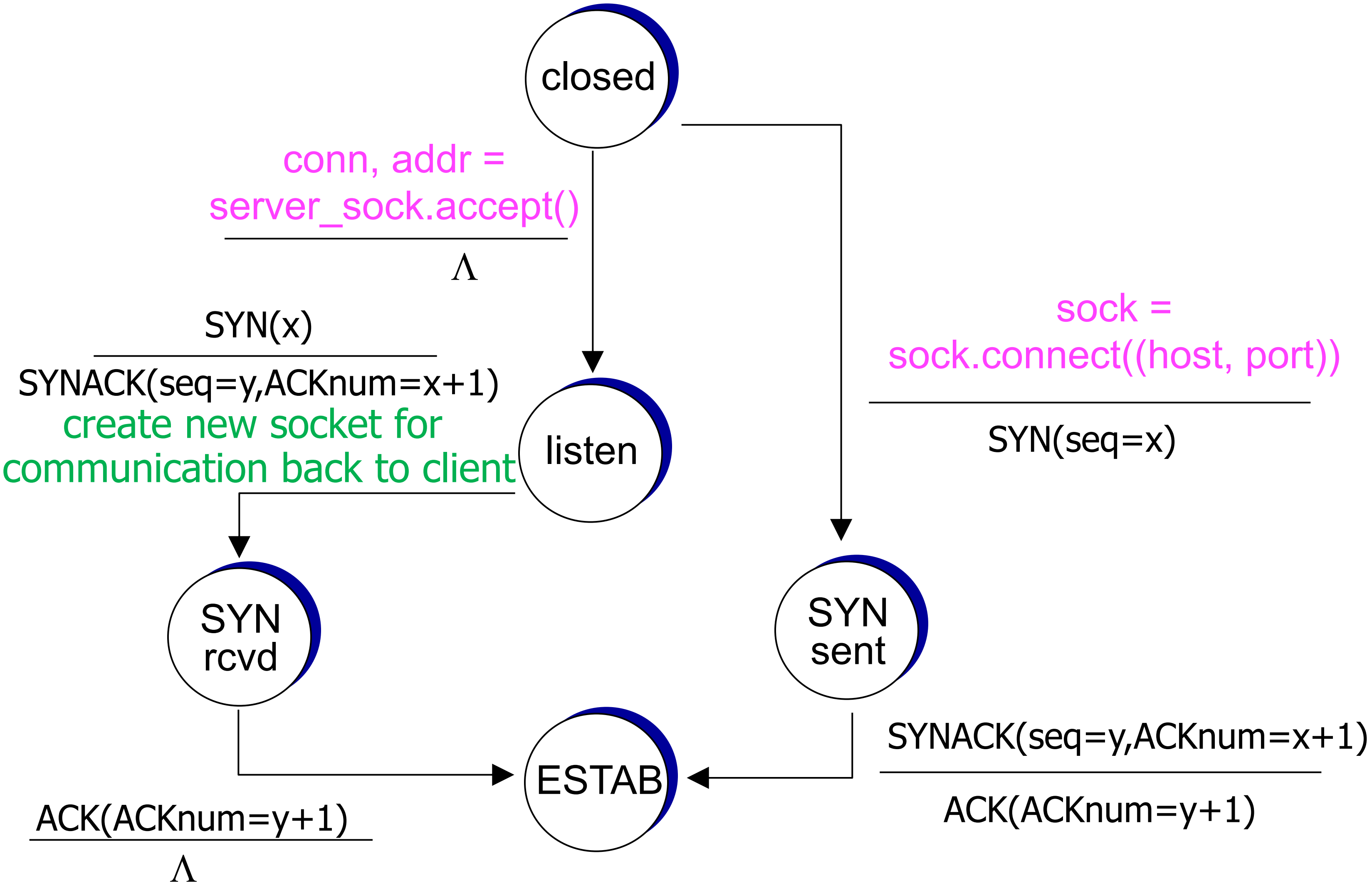




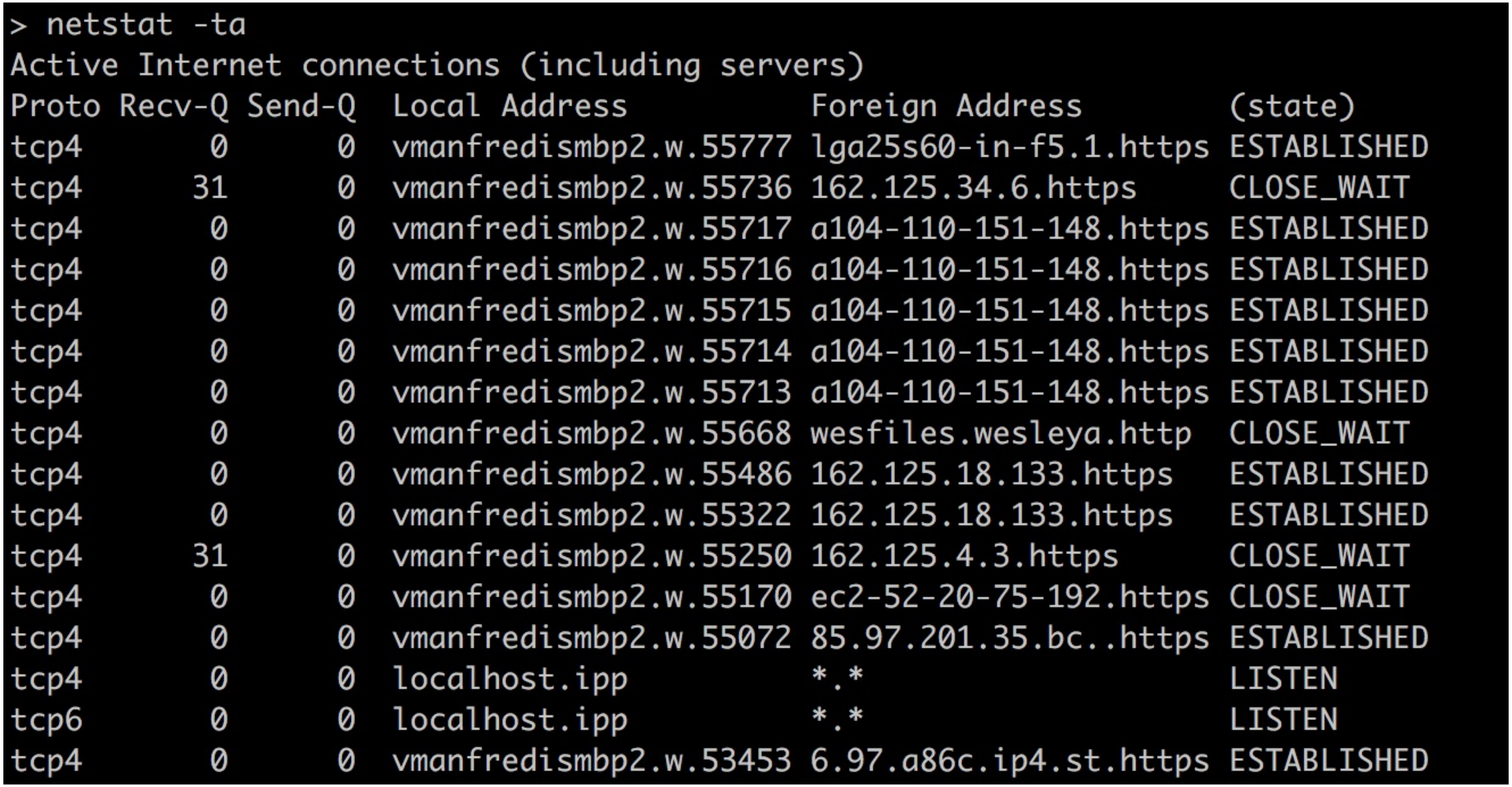
TCP 3-way handshake



TCP 3-way handshake: FSM



Look at the state of tcp connections

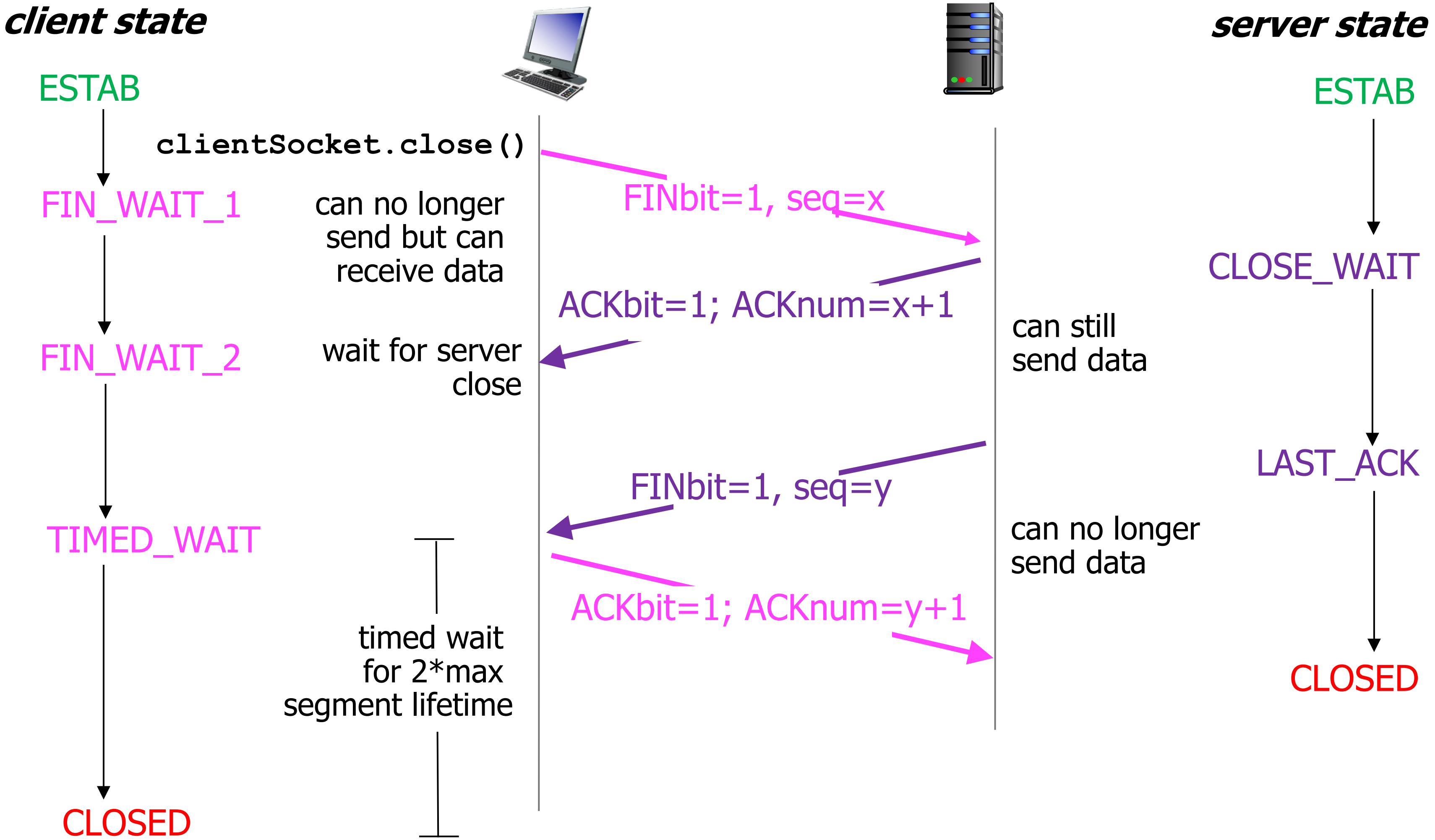


```
> netstat -ta
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp4       0      0  vmanfredismbp2.w.55777 lga25s60-in-f5.1.https ESTABLISHED
tcp4      31      0  vmanfredismbp2.w.55736 162.125.34.6.https     CLOSE_WAIT
tcp4       0      0  vmanfredismbp2.w.55717 a104-110-151-148.https ESTABLISHED
tcp4       0      0  vmanfredismbp2.w.55716 a104-110-151-148.https ESTABLISHED
tcp4       0      0  vmanfredismbp2.w.55715 a104-110-151-148.https ESTABLISHED
tcp4       0      0  vmanfredismbp2.w.55714 a104-110-151-148.https ESTABLISHED
tcp4       0      0  vmanfredismbp2.w.55713 a104-110-151-148.https ESTABLISHED
tcp4       0      0  vmanfredismbp2.w.55668 wesfiles.wesleya.http  CLOSE_WAIT
tcp4       0      0  vmanfredismbp2.w.55486 162.125.18.133.https   ESTABLISHED
tcp4       0      0  vmanfredismbp2.w.55322 162.125.18.133.https   ESTABLISHED
tcp4      31      0  vmanfredismbp2.w.55250 162.125.4.3.https      CLOSE_WAIT
tcp4       0      0  vmanfredismbp2.w.55170 ec2-52-20-75-192.https CLOSE_WAIT
tcp4       0      0  vmanfredismbp2.w.55072 85.97.201.35.bc..https ESTABLISHED
tcp4       0      0  localhost.ipp          *.*                    LISTEN
tcp6       0      0  localhost.ipp          *.*                    LISTEN
tcp4       0      0  vmanfredismbp2.w.53453 6.97.a86c.ip4.st.https ESTABLISHED
```

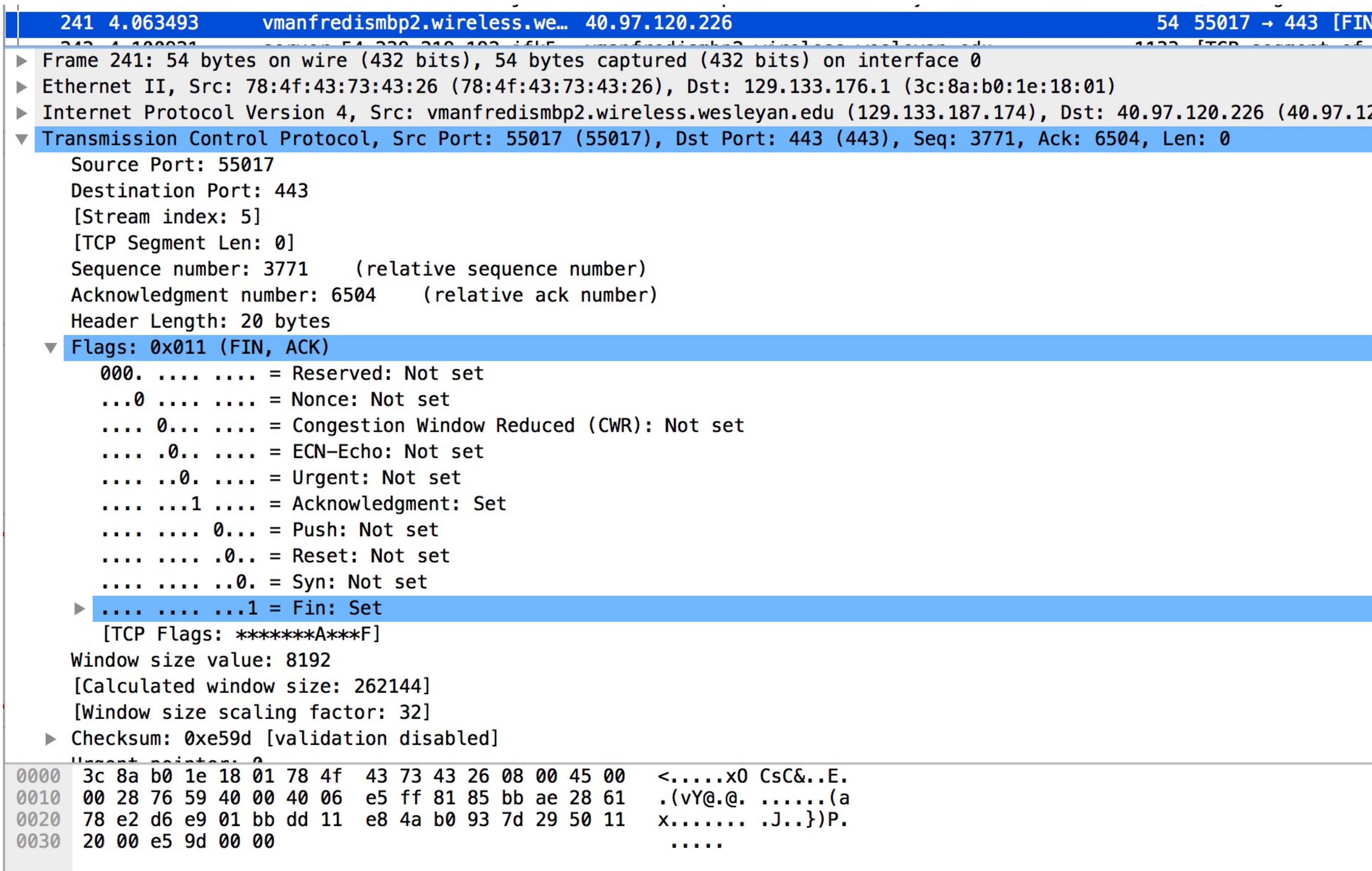
TCP: politely closing a connection

Client, server each sends TCP segment with FIN bit = 1

- respond to received FIN with ACK (ACK can be combined with own FIN)



FIN segment in Wireshark



```
 241 4.063493    vmanfredismbp2.wireless.we…  40.97.120.226                              54  55017 → 443 [FIN
▶ Frame 241: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) on interface 0
▶ Ethernet II, Src: 78:4f:43:73:43:26 (78:4f:43:73:43:26), Dst: 129.133.176.1 (3c:8a:b0:1e:18:01)
▶ Internet Protocol Version 4, Src: vmanfredismbp2.wireless.wesleyan.edu (129.133.187.174), Dst: 40.97.120.226 (40.97.12
▼ Transmission Control Protocol, Src Port: 55017 (55017), Dst Port: 443 (443), Seq: 3771, Ack: 6504, Len: 0
    Source Port: 55017
    Destination Port: 443
    [Stream index: 5]
    [TCP Segment Len: 0]
    Sequence number: 3771    (relative sequence number)
    Acknowledgment number: 6504    (relative ack number)
    Header Length: 20 bytes
  ▼ Flags: 0x011 (FIN, ACK)
      000. .... .... = Reserved: Not set
      ...0 .... .... = Nonce: Not set
      .... 0... .... = Congestion Window Reduced (CWR): Not set
      .... .0.. .... = ECN-Echo: Not set
      .... ..0. .... = Urgent: Not set
      .... ...1 .... = Acknowledgment: Set
      .... .... 0... = Push: Not set
      .... .... .0.. = Reset: Not set
      .... .... ..0. = Syn: Not set
    ▶ .... .... ...1 = Fin: Set
      [TCP Flags: *******A***F]
    Window size value: 8192
    [Calculated window size: 262144]
    [Window size scaling factor: 32]
  ▶ Checksum: 0xe59d [validation disabled]
    Urgent pointer: 0
0000  3c 8a b0 1e 18 01 78 4f  43 73 43 26 08 00 45 00   <.....xO CsC&..E.
0010  00 28 76 59 40 00 40 06  e5 ff 81 85 bb ae 28 61   .(vY@.@. ......(a
0020  78 e2 d6 e9 01 bb dd 11  e8 4a b0 93 7d 29 50 11   x....... .J..})P.
0030  20 00 e5 9d 00 00                                   .....
```