Lecture 12: Transport Layer
TCP again

COMP 332, Fall 2018

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 today at 11:59p
- Q: views on group projects?

Midterm in class on Wed. Oct. 17

- closed book, closed notes, covers through whatever we get through today
- short answer: how does UDP use ports #s to demultiplex incoming datagrams?
- long answer: given channel characteristics design a protocol

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

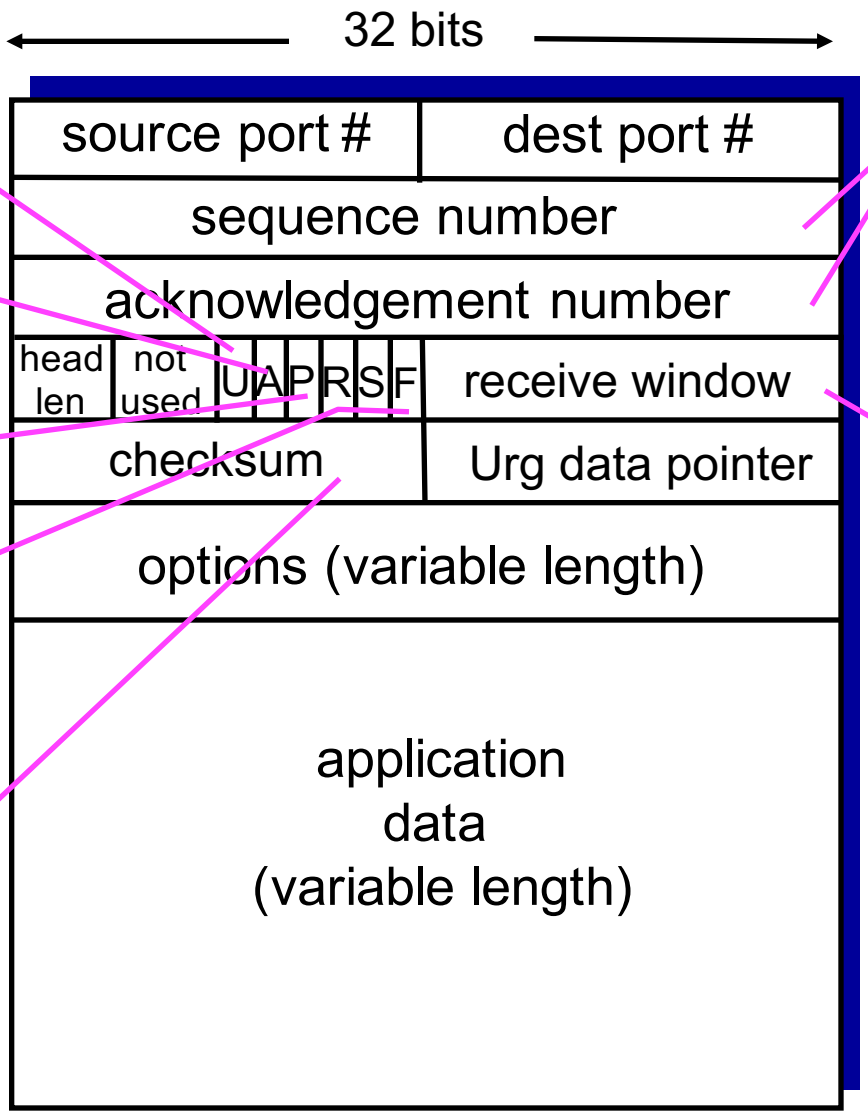
- congestion and flow control determine window size
- seq #s are byte offsets

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)

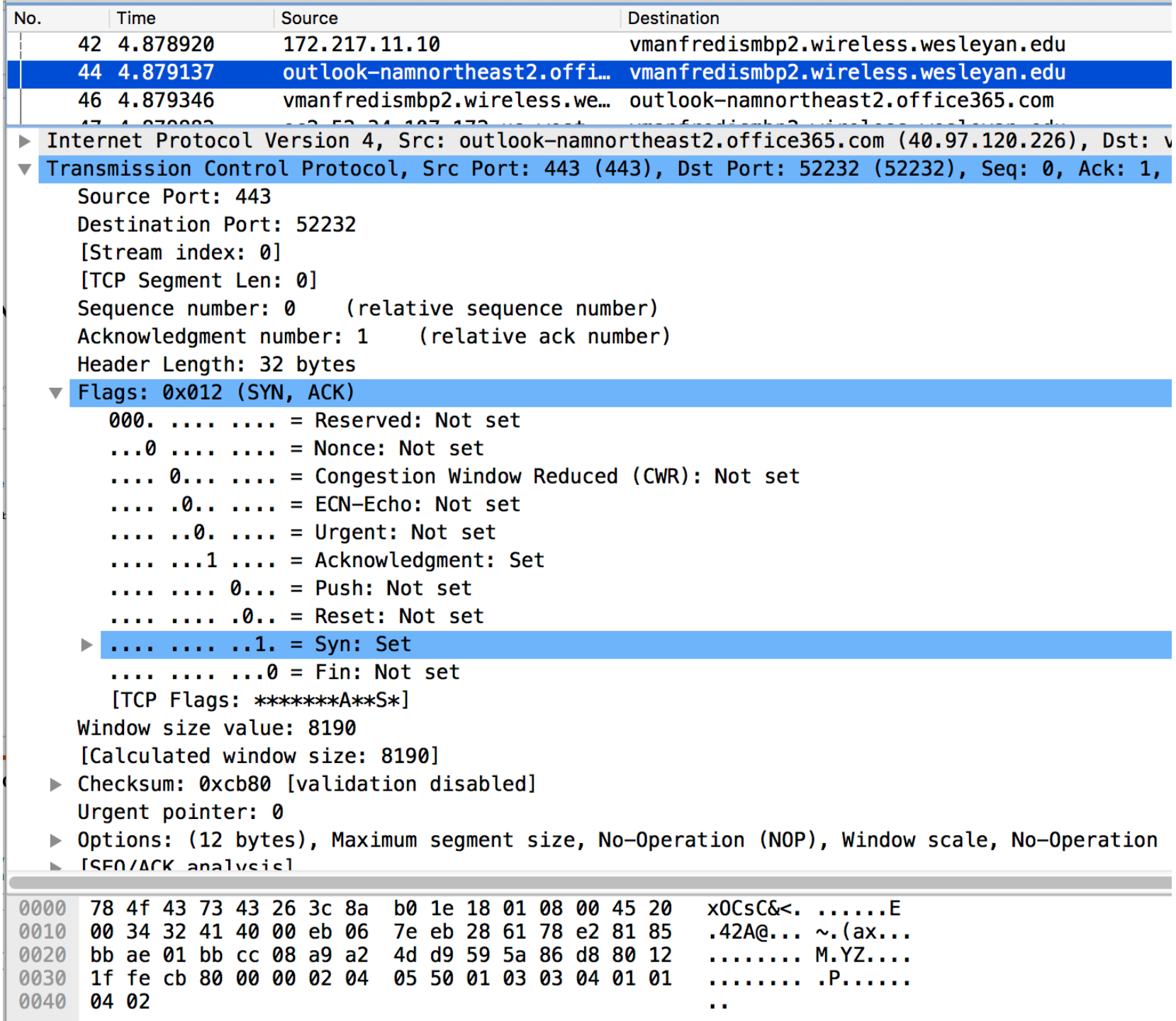
RST, SYN, FIN: connection estab (setup, teardown commands)

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
Internet Protocol Version 4, Src: outlook-namnortheast2.office365.com (40.97.120.226), Dst:
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 ..

TCP

SEQ #S AND ACK #S

TCP seq. numbers, ACKs

Sequence #s

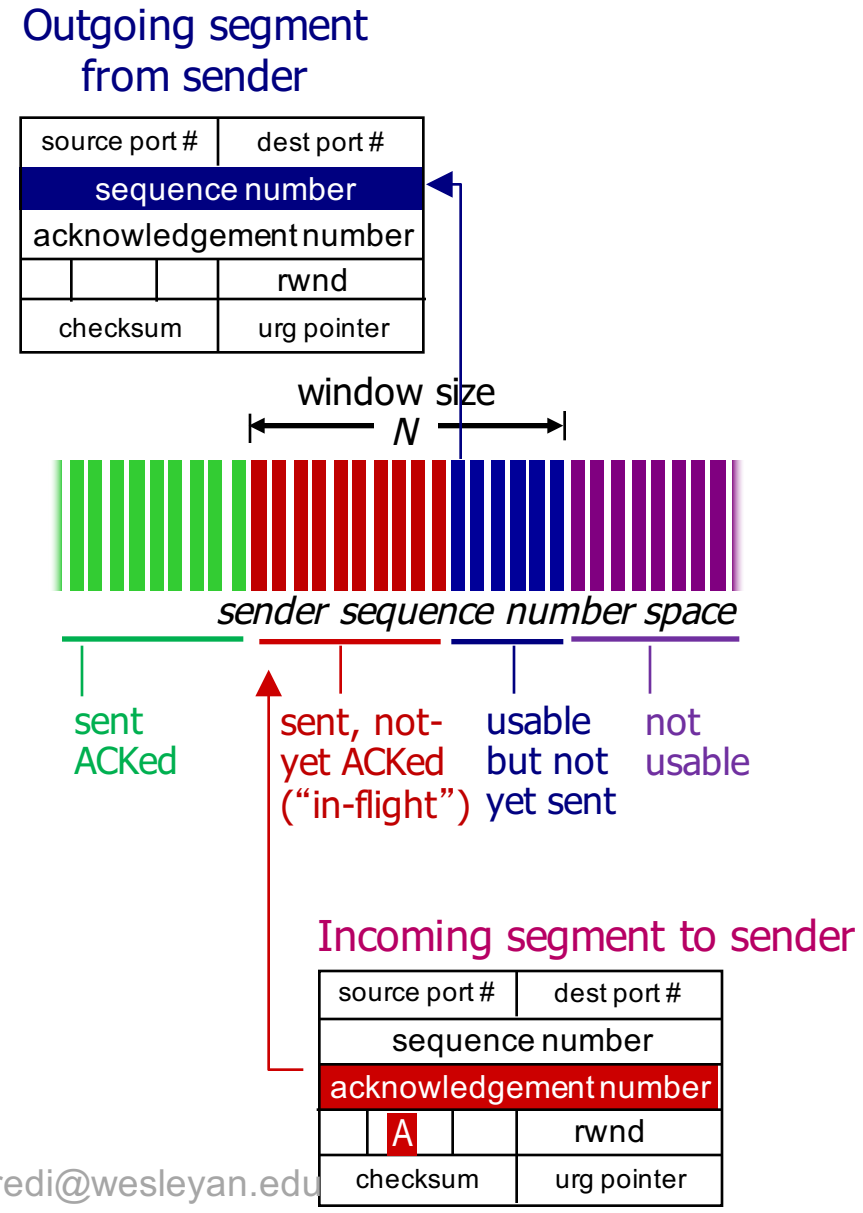
- byte stream # of first byte in segment's data

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

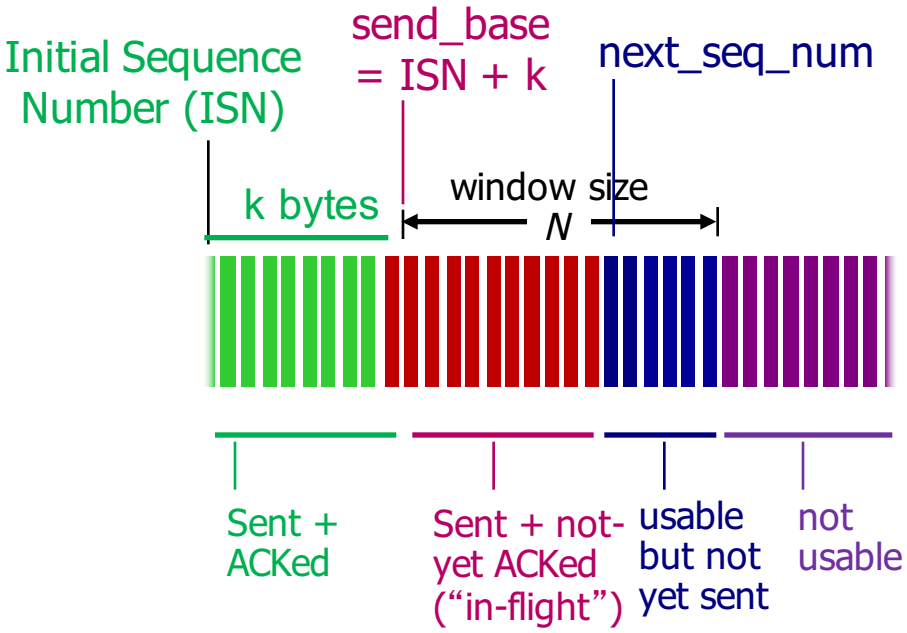


TCP ACKs

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

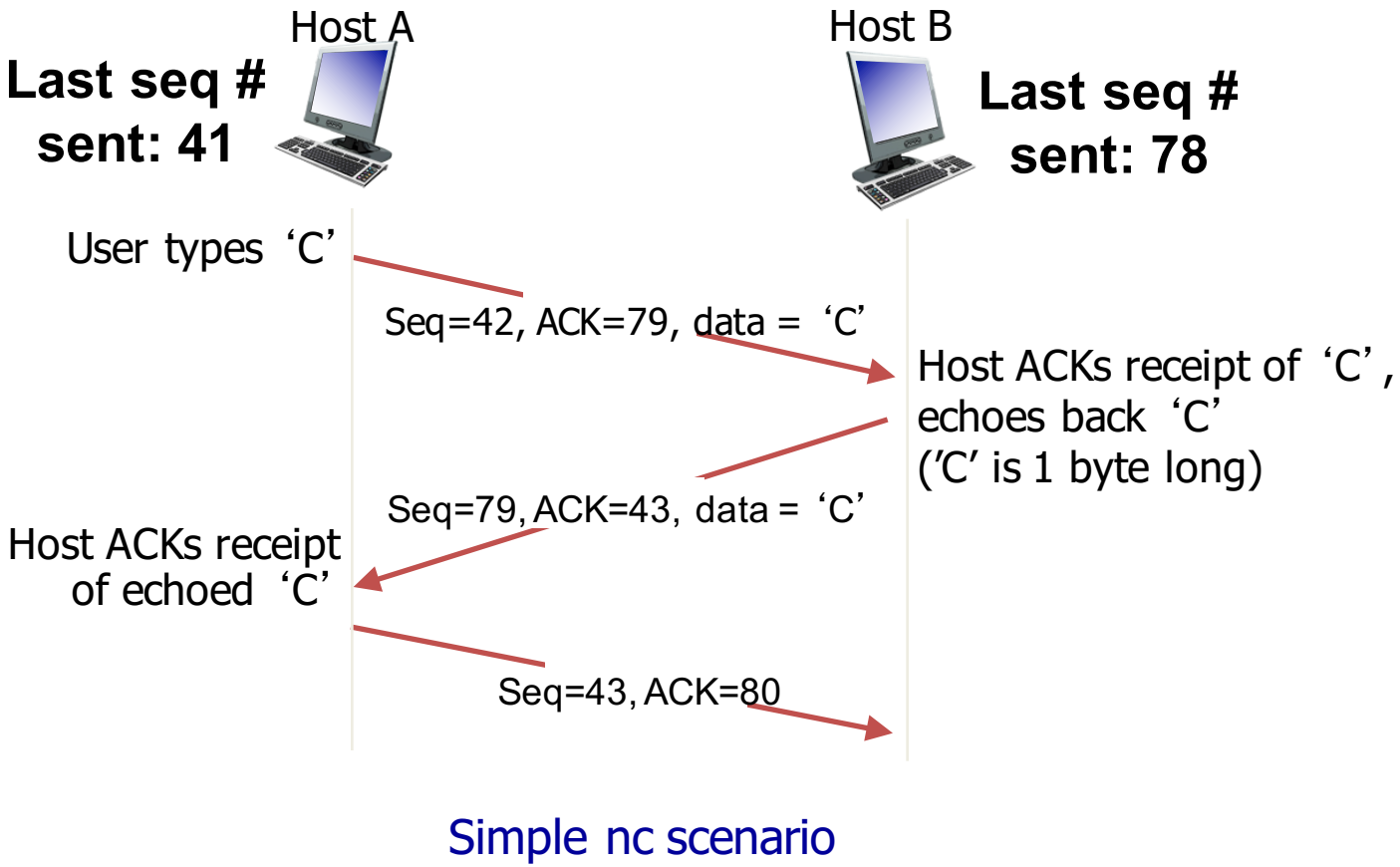
- ACKs what receiver expects next, not last packet received
  - implicitly also ACKs everything up to sequence number received
- only 1 retransmission timer (for first pkt in window)
  - sender retransmits only first pkt in window if no ack when timer expires

Sequence #s are not sequential: counting bytes not packets



TCP seq. numbers, ACKs

Sequence numbers are synchronized during connection set-up



Simple nc scenario

Host 1

```
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
```

Handshake: Synchronize ISNs

Host 2

```
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

```
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
```

Data exchange

```
Transmission Control Protocol, Src Pc
    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?

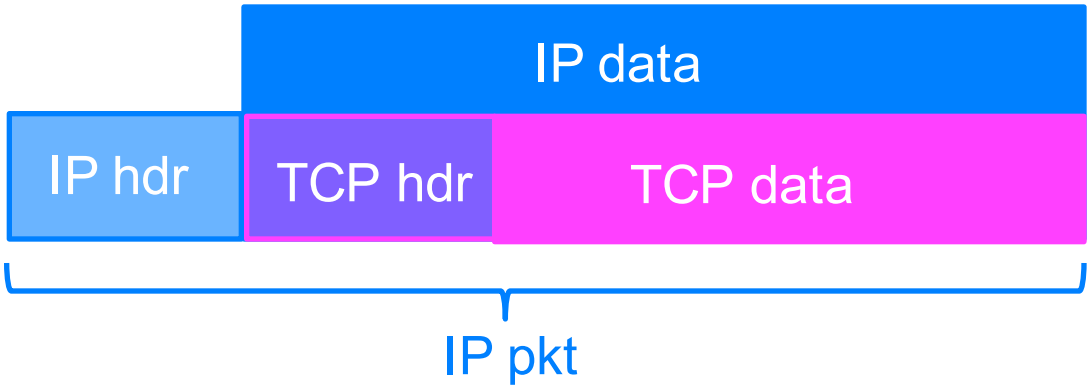
Segment size

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

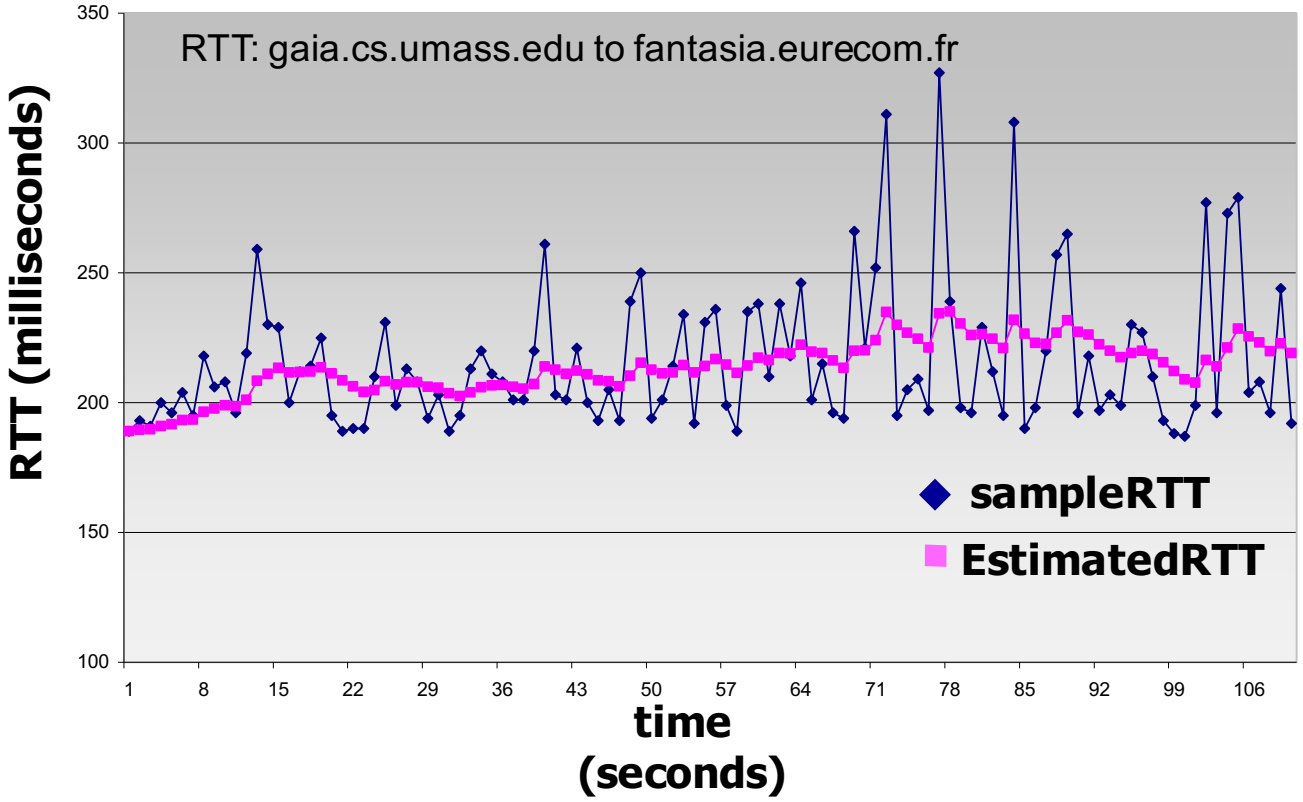
- time from segment transmission to ACK reception
- ignore retransmissions
  - since problems associating retransmitted ACK with right pkt
  - will vary: use average of several measurements

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

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

(typically, $\beta = 0.25$)

$$TimeoutInterval = EstimatedRTT + 4*DevRTT$$

"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

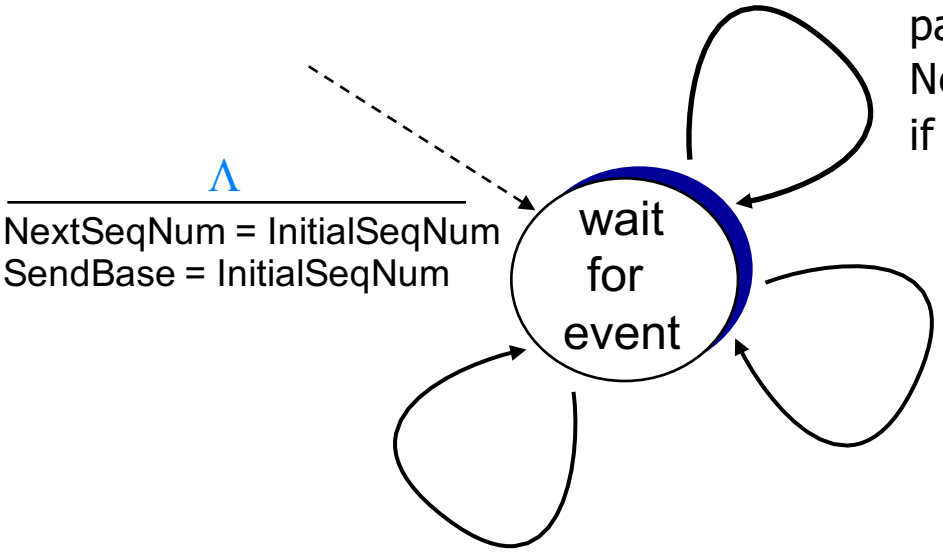
- timeout events
- duplicate ACKs

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



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

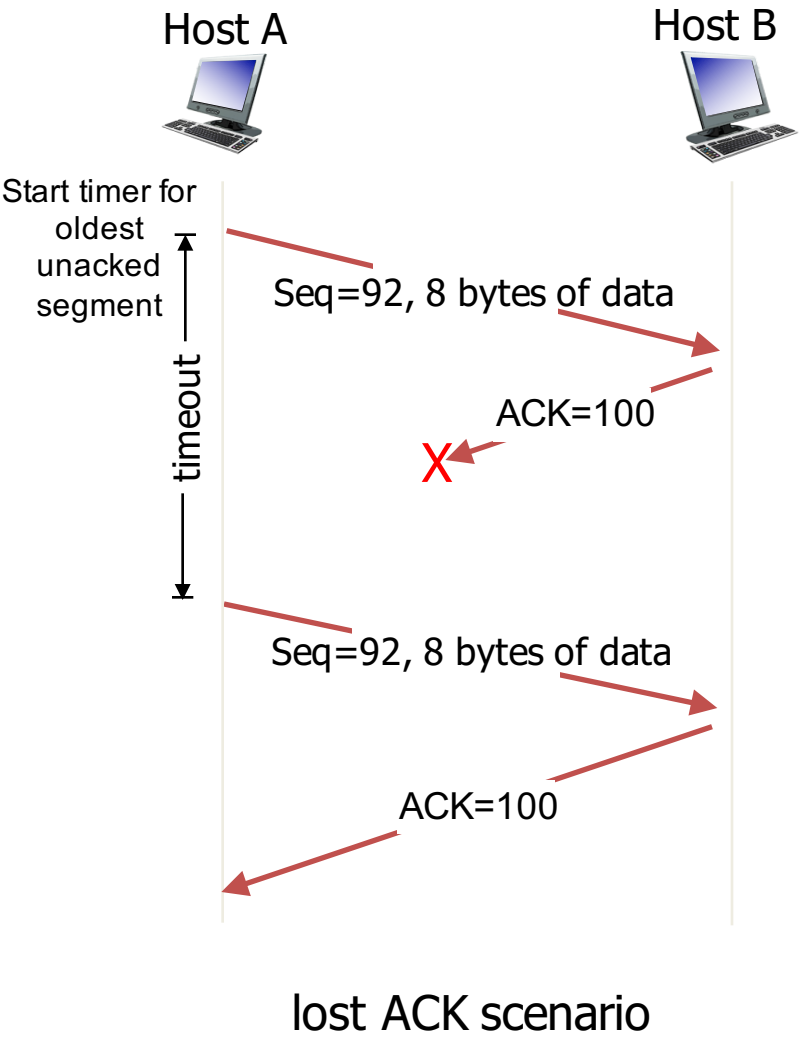
Retransmit first segment in window, restart timer

ACK received, with ACK field value y

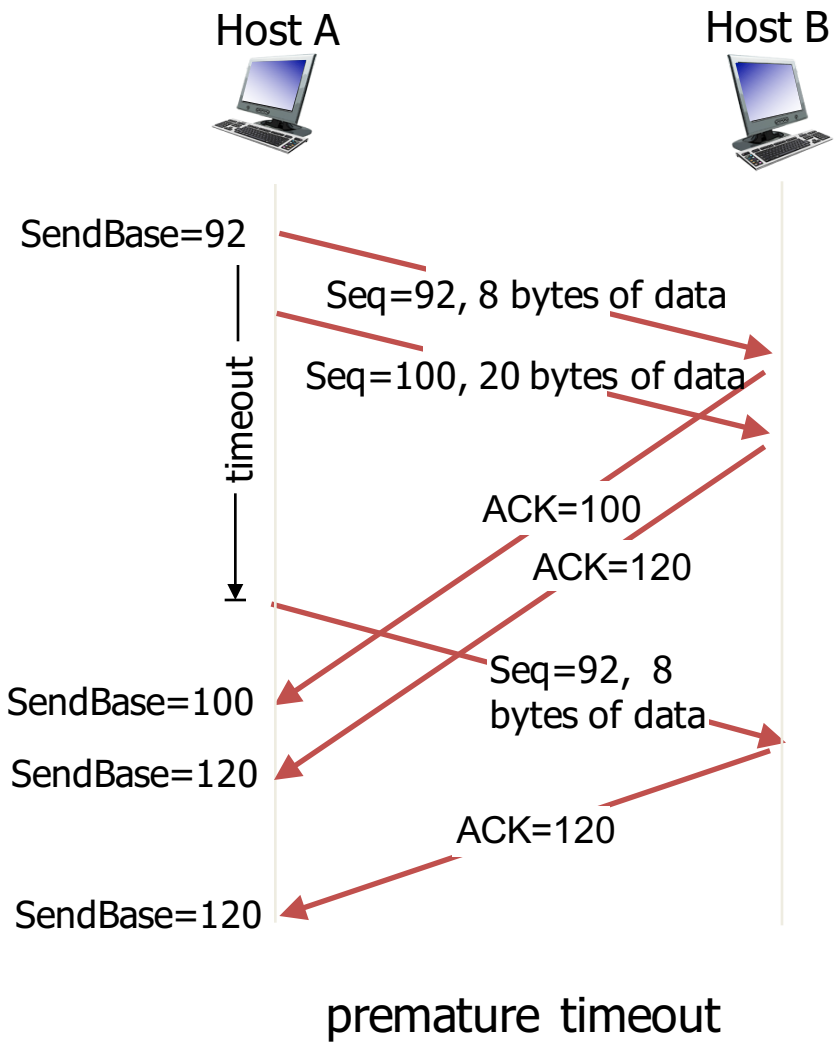
```
if (y > SendBase) {
    SendBase = y
    /* SendBase–1: last cumulatively ACKed byte */
    if (there are currently not-yet-acked segments)
        start timer
    else stop timer
    }
```

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

TCP: retransmission scenarios

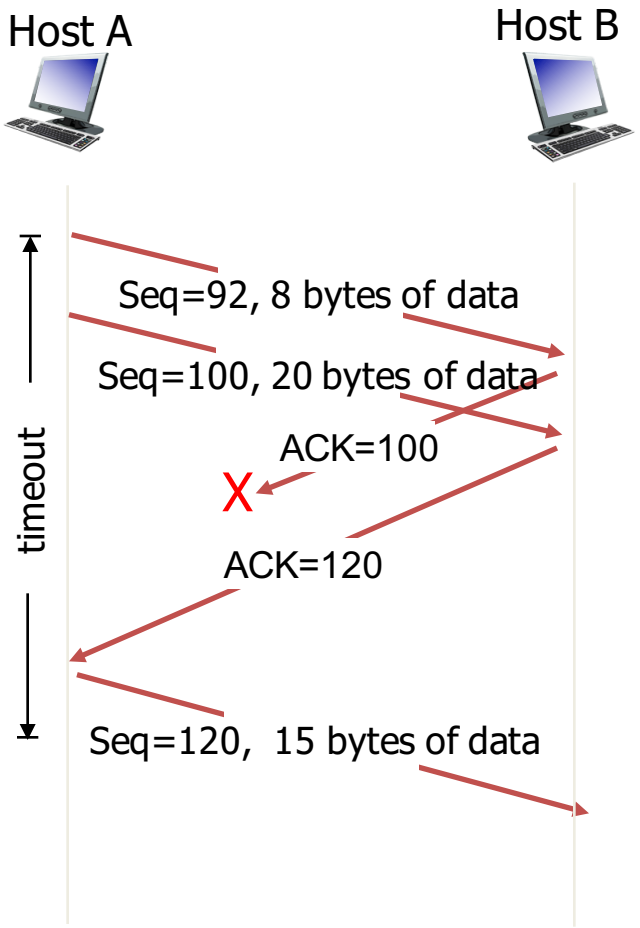


lost ACK scenario



premature timeout

TCP: retransmission scenarios



cumulative ACK

Duplicate ACKs

Time-out period often relatively long

- long delay before resending lost packet

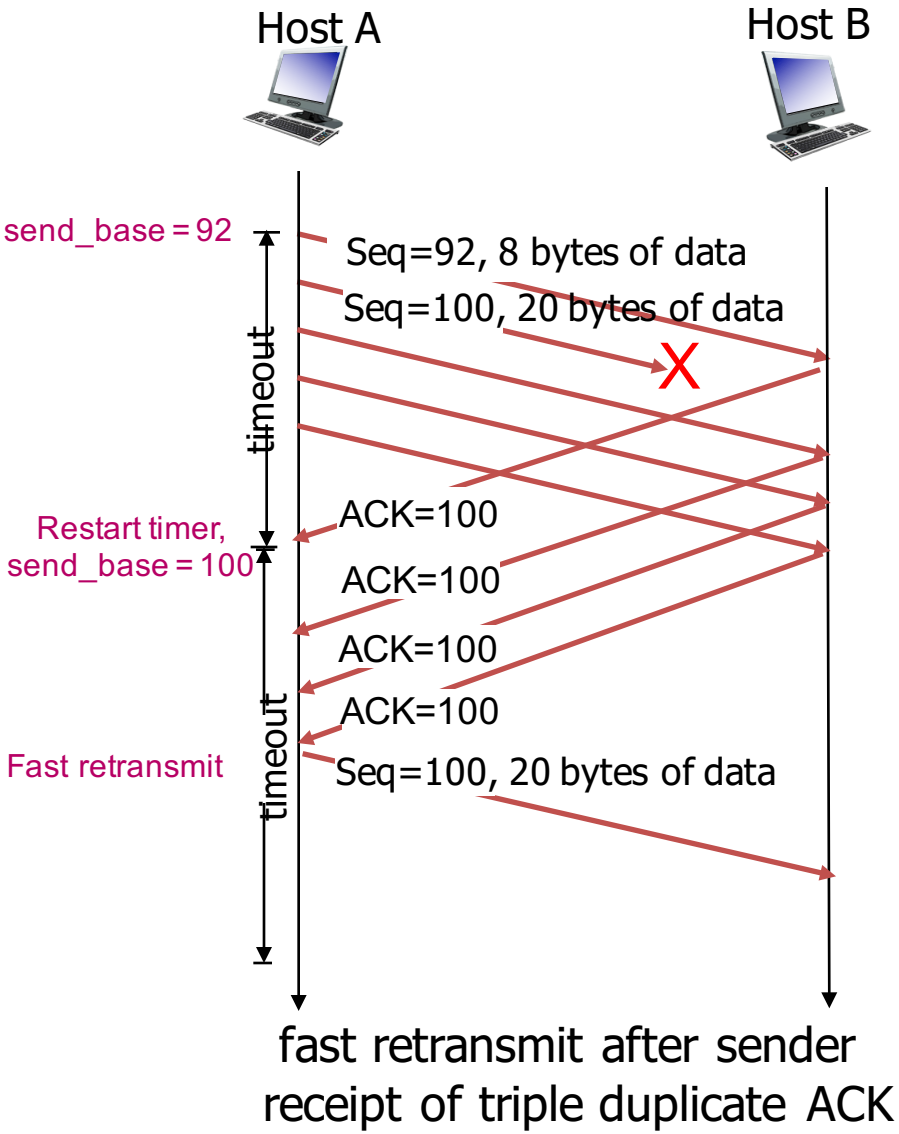
Duplicate ACKs

- 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 #
- Why 3?
  - pkts may just have been reordered otherwise
  - likely that unacked segment lost, so don't wait for timeout

TCP fast retransmit



fast retransmit after sender receipt of triple duplicate ACK

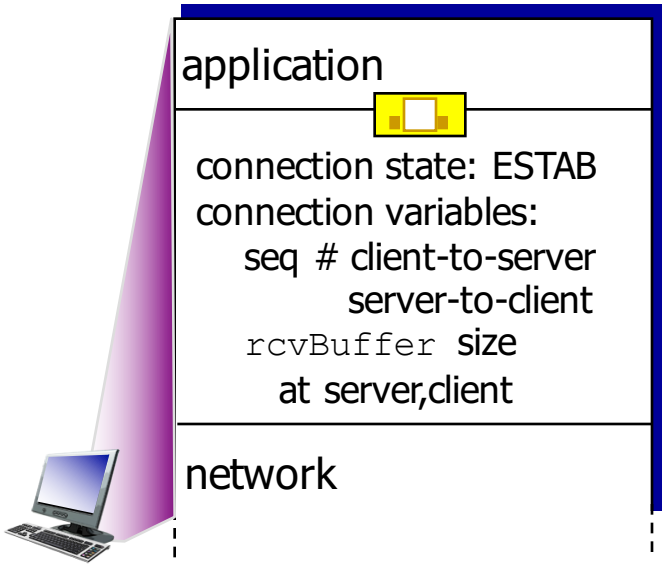
TCP

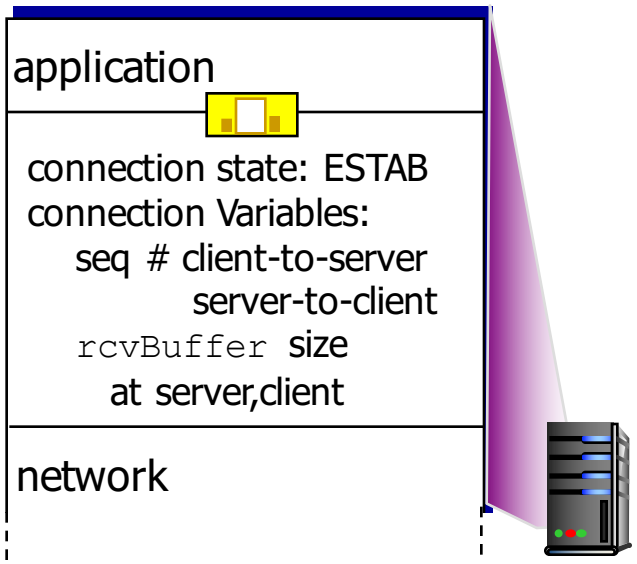
CONNECTION MANAGEMENT

Connection Management

Before exchanging data, sender/receiver handshake

- establish connection and connection parameters
  - each knowing the other willing to establish connection
- tear down connection when done



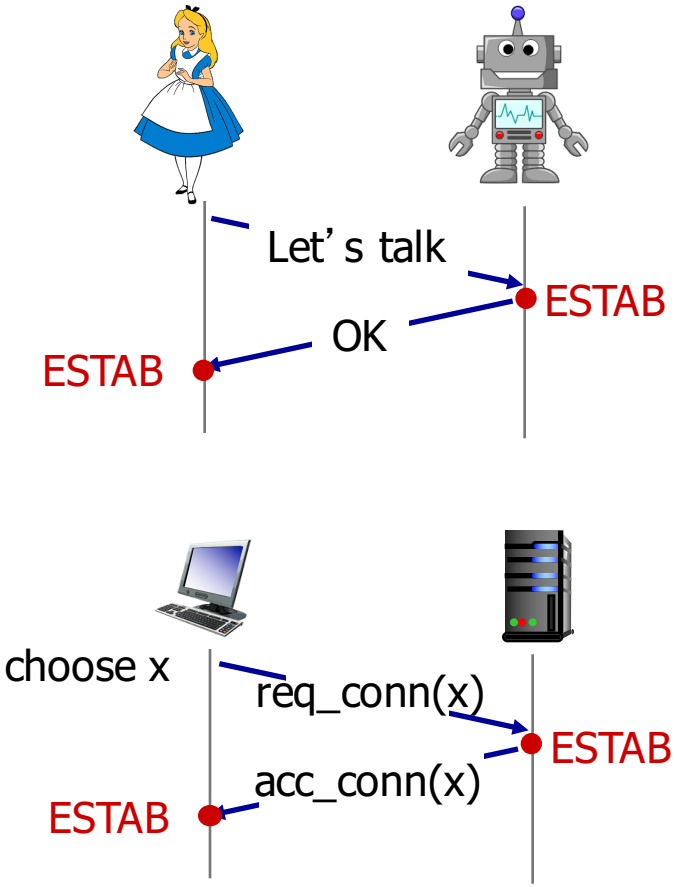


sock = sock.connect((host, port))

conn, addr = server_sock.accept()

Agreeing to establish a connection

2-way handshake:

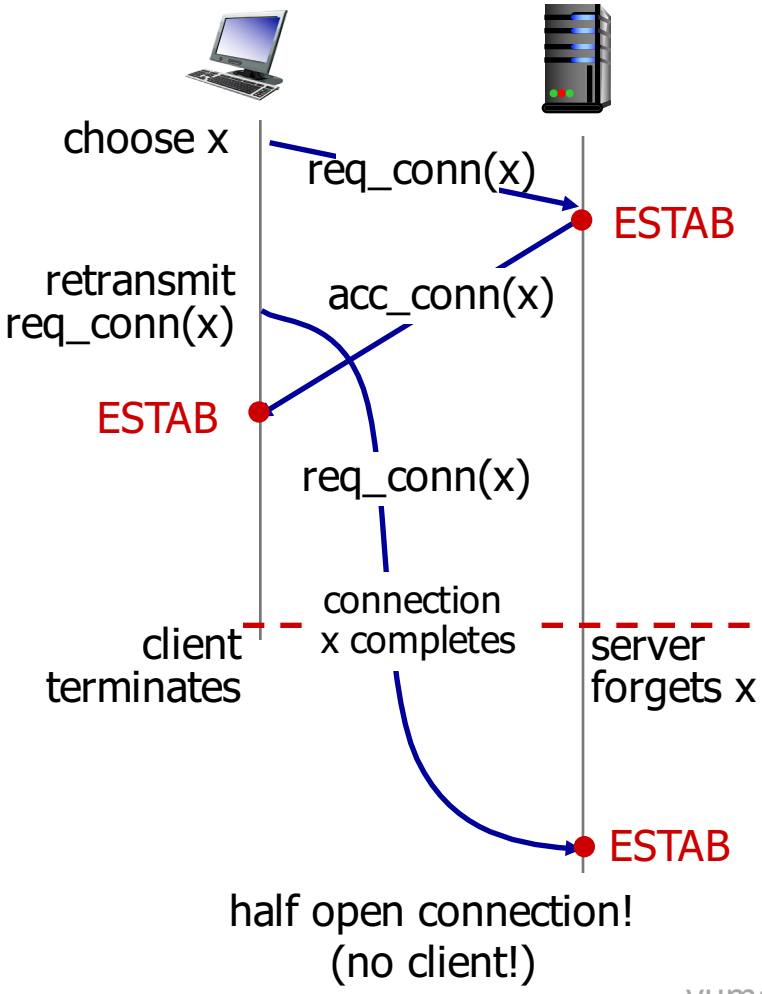


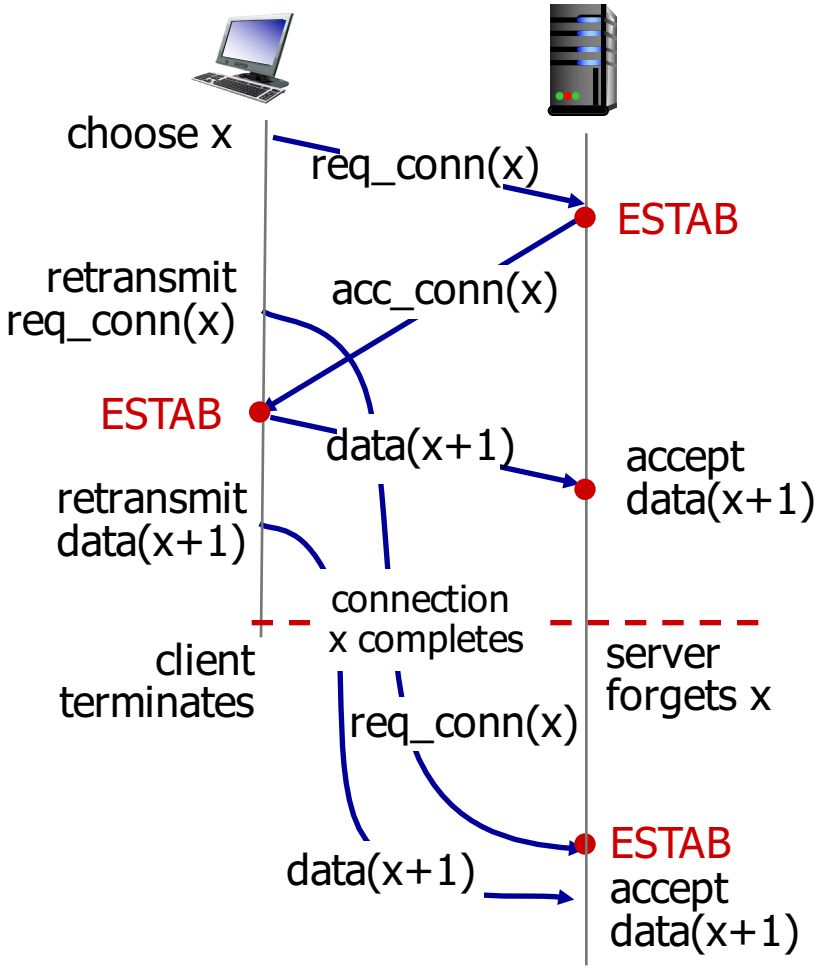
Q: will 2-way handshake always work in network?

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

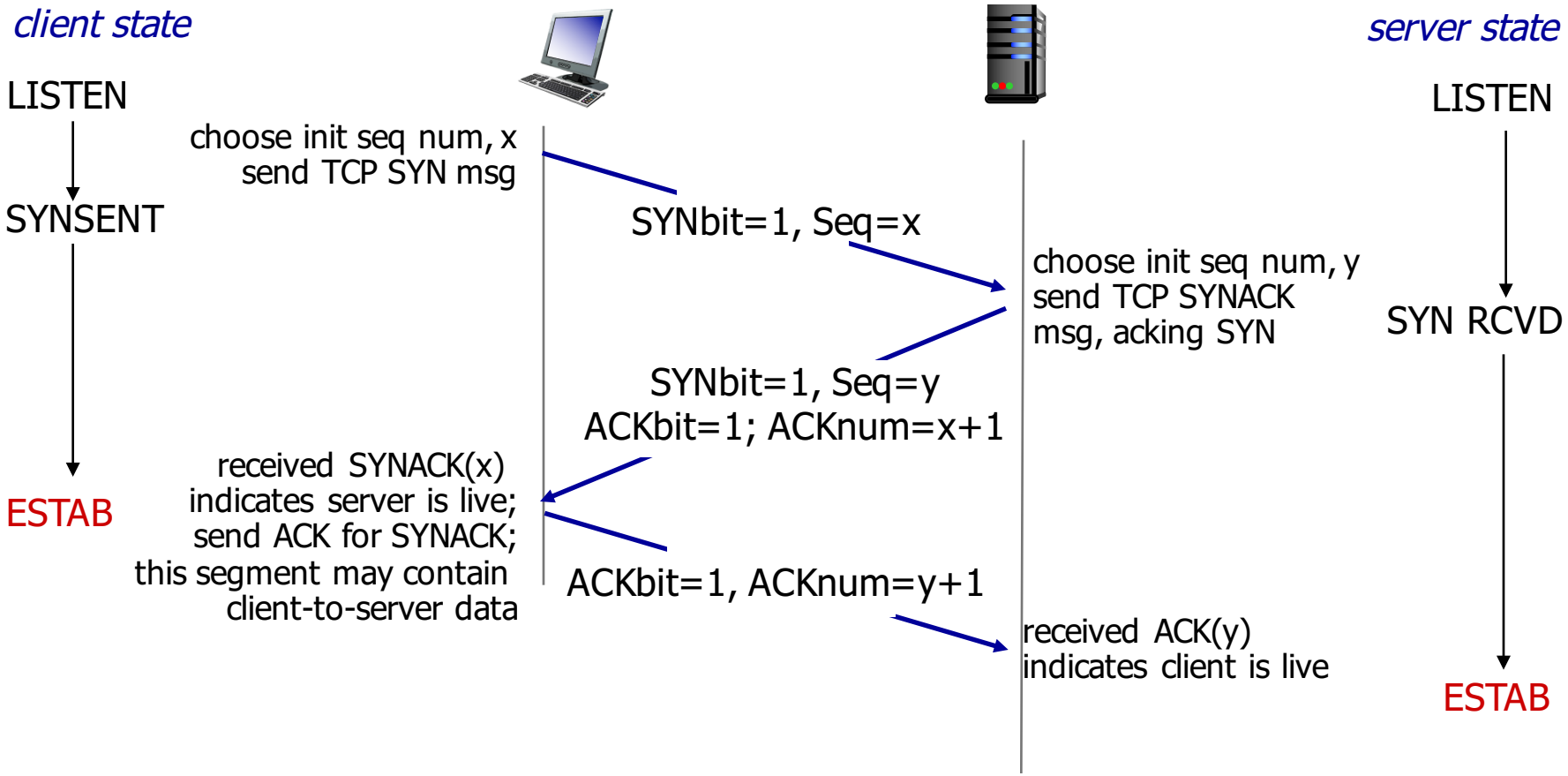
Agreeing to establish a connection

2-way handshake failure scenarios:

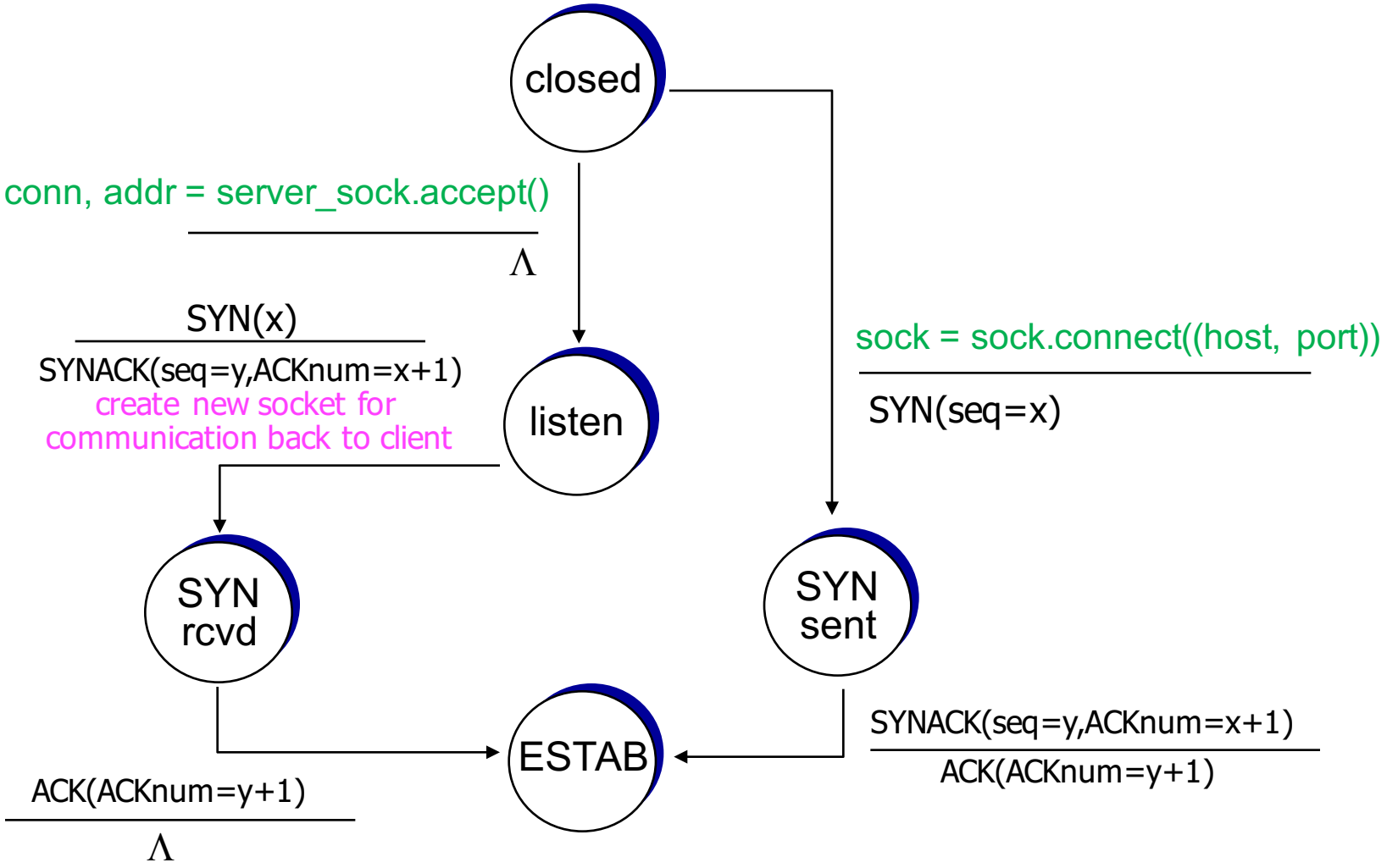




TCP 3-way handshake



TCP 3-way handshake: FSM



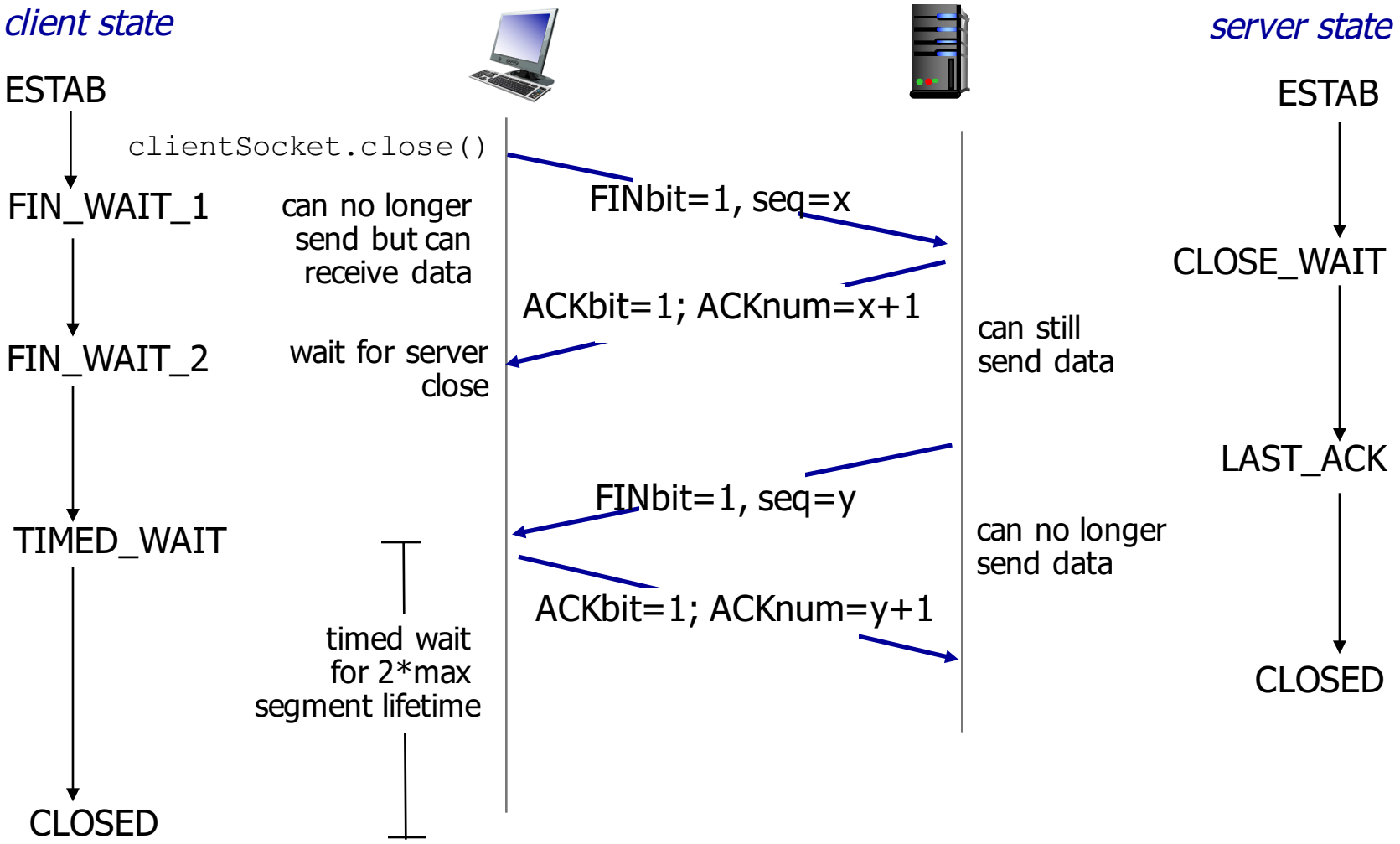
Look at the state of tcp connections

```
> 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 close connection: 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