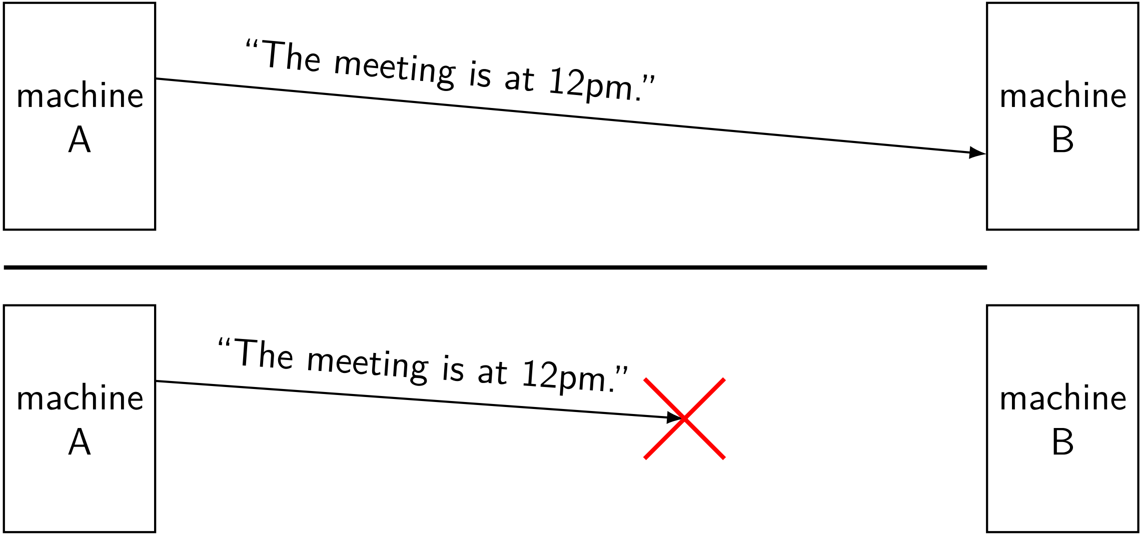
changelog

6 Sep 2024: add explicit note re: ACK up to X being inclusive first time it appears

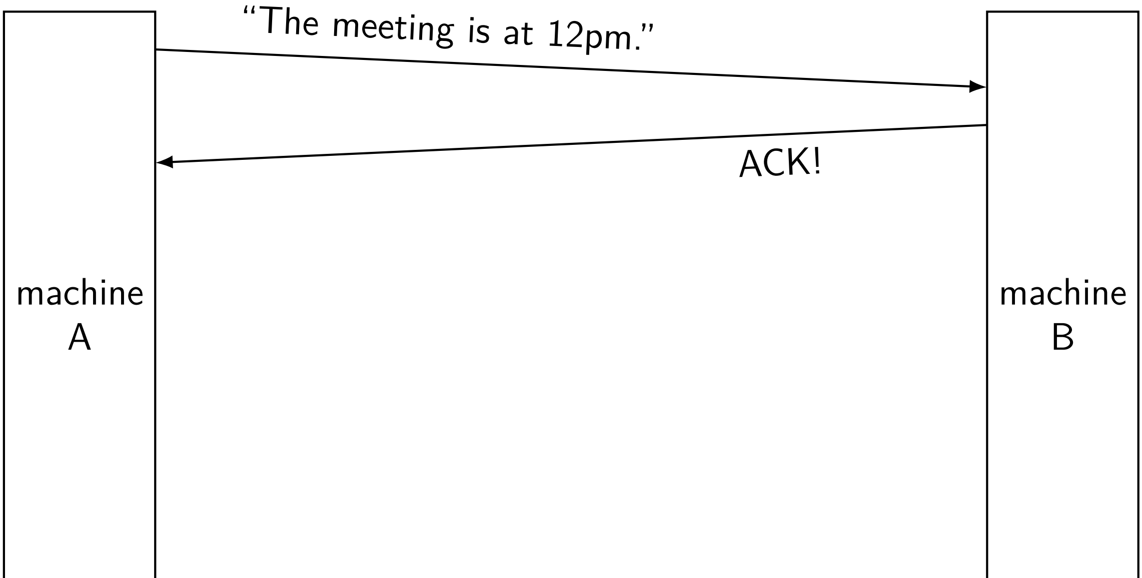
10 Sep 2024: be more clear that ACK number is 1+last byte sequence number

10 Sep 2024: correct discussion of window size variation to discuss burstiness, which means that the maximum possible latency may not be where the throughput collapse happens

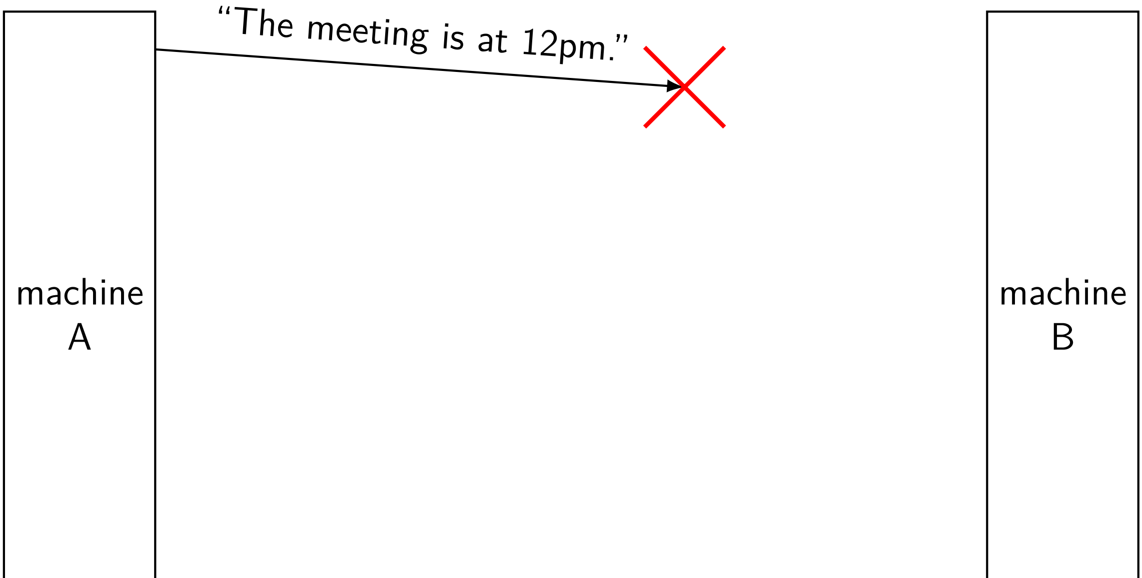
dealing with network message lost



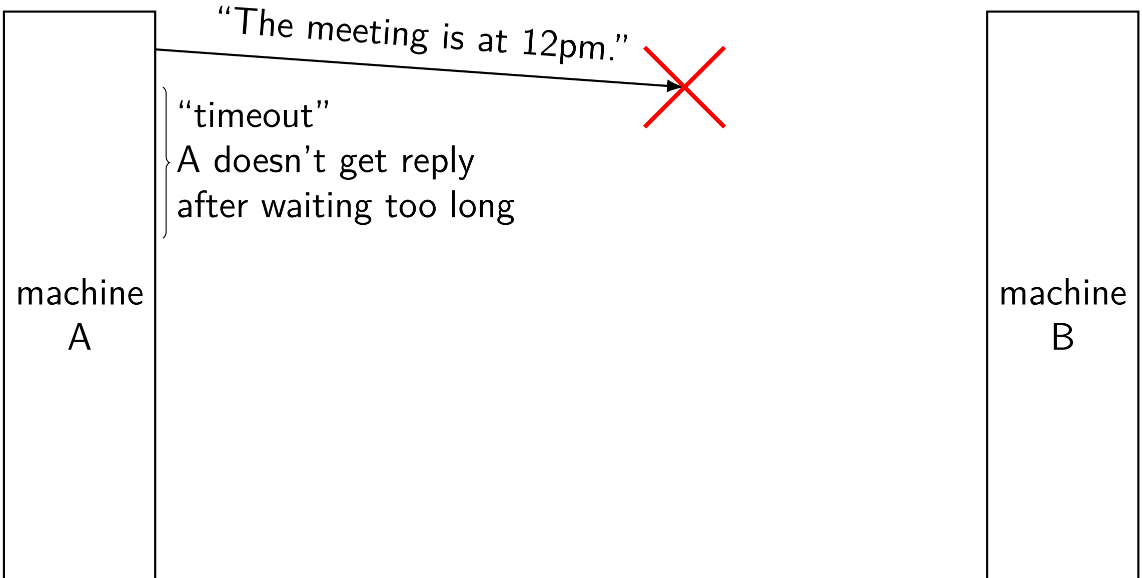
handling lost message: acknowledgements



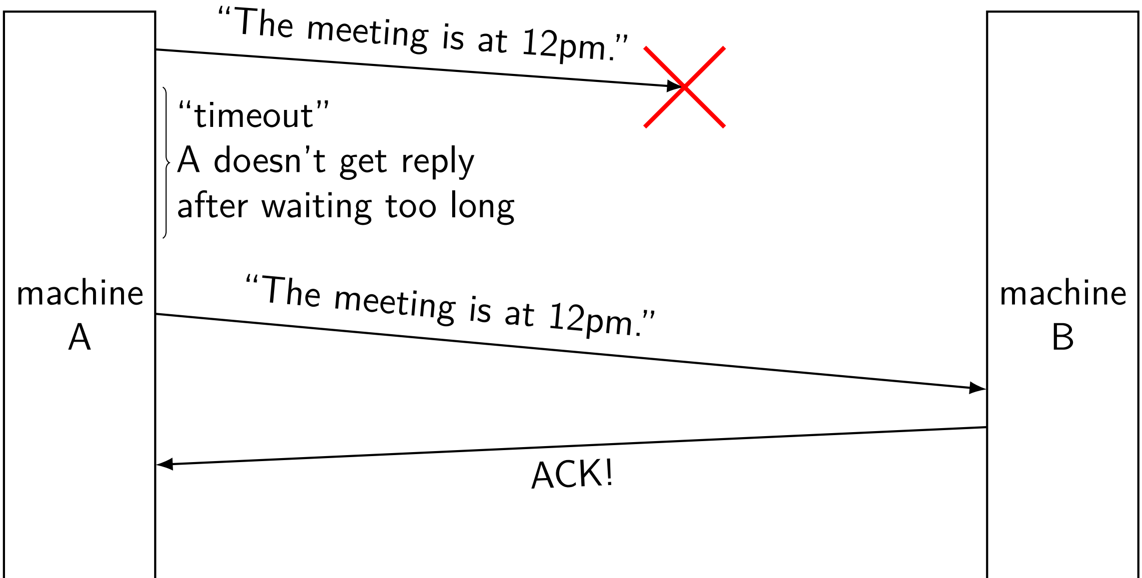
handling lost message



handling lost message



handling lost message



protocol so far

on sender: until ACK received:

- (re)send frame of data
- wait fixed amount of time for ACK

on receiver: continuously:

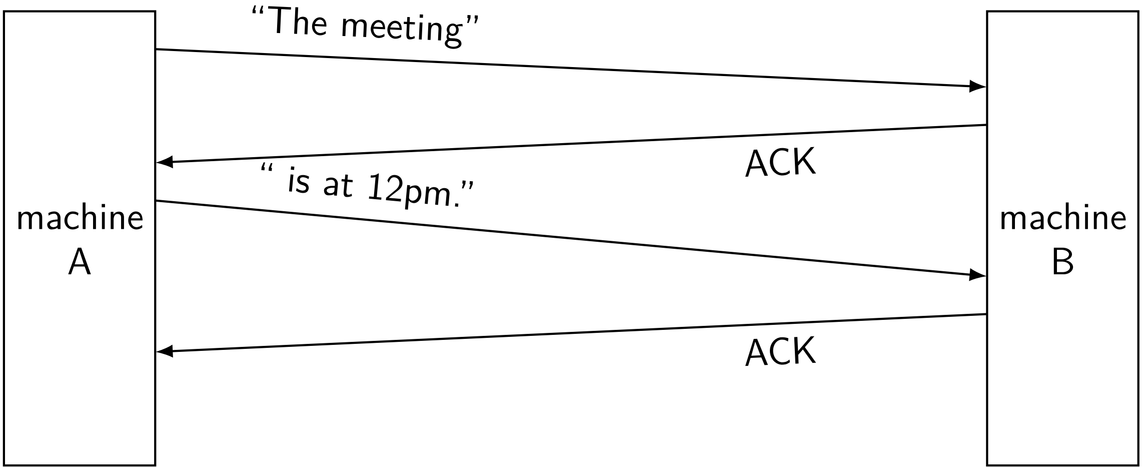
- wait for frame of data
- send ACK back

problem

really want to send multiple frames

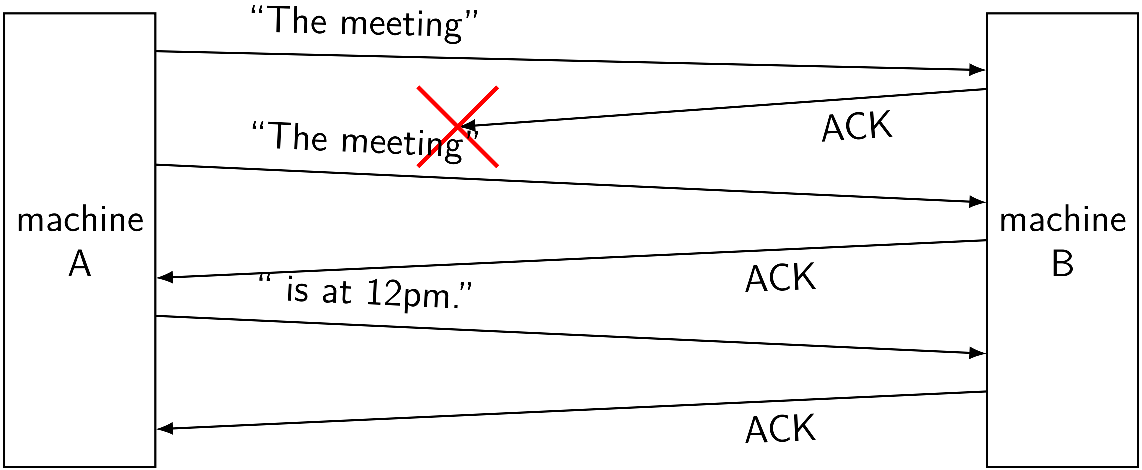
example: data split in multiple pieces

splitting messages: try 1

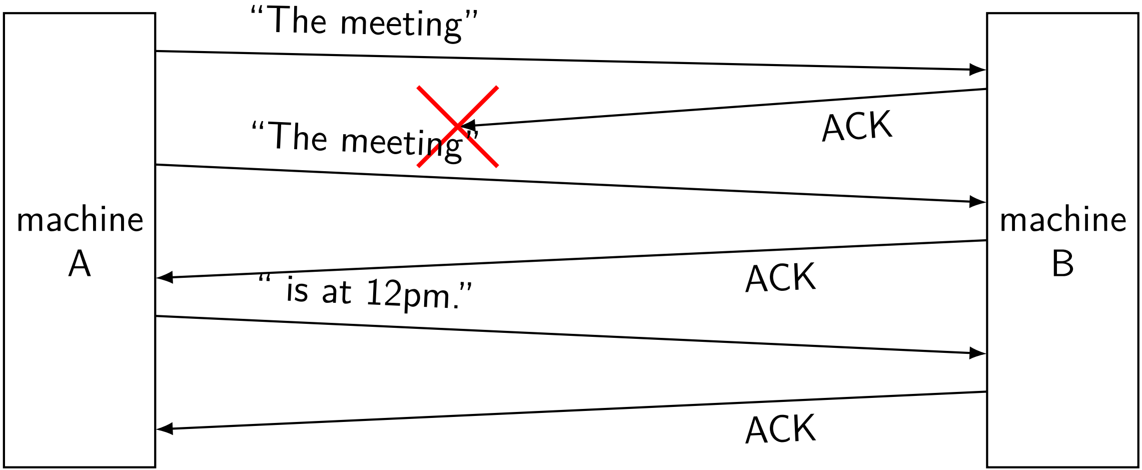


reconstructed message:
The meeting is at 12pm.

splitting messages: try 1 — problem 1



splitting messages: try 1 — problem 1



reconstructed message:
The meetingThe meeting is at 12pm.

exercise: other problems?

sending 'The meeting', 'is at 12pm'

what would be received for each of these scenarios?

1. message (instead of acknowledgment) is lost
2. first message from machine A is delayed a long time by network
3. acknowledgment of second message lost instead of first

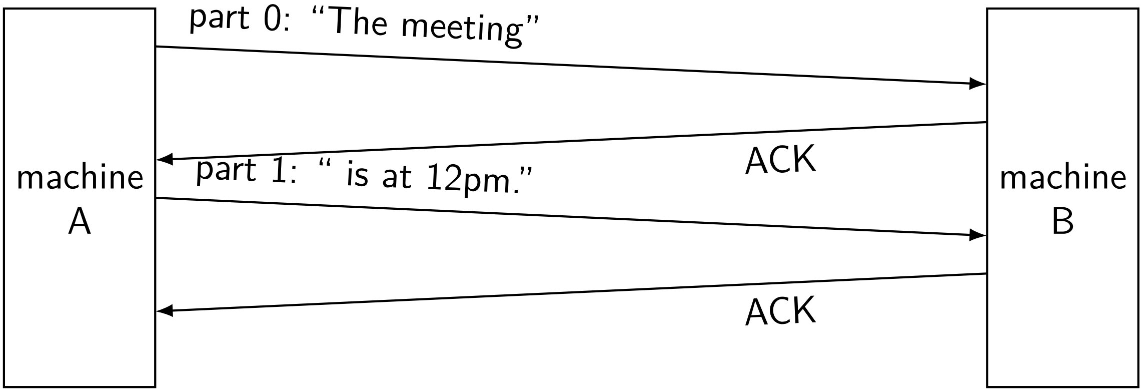
aside: message delays

long message delays not possible with direct link

but are possible with:

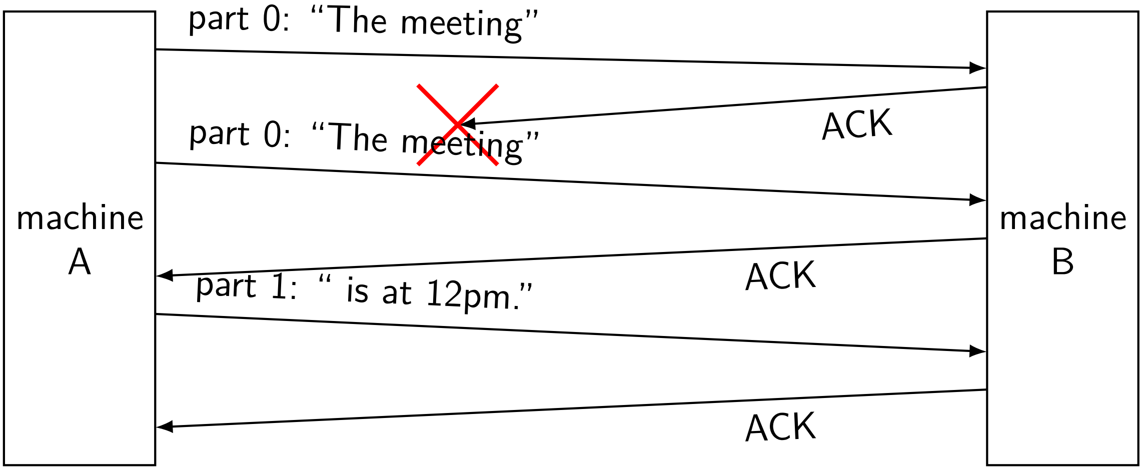
- multiple paths from A to B
- doing this kind of acknowledgment + resending hop-by-hop

splitting messages: try 2



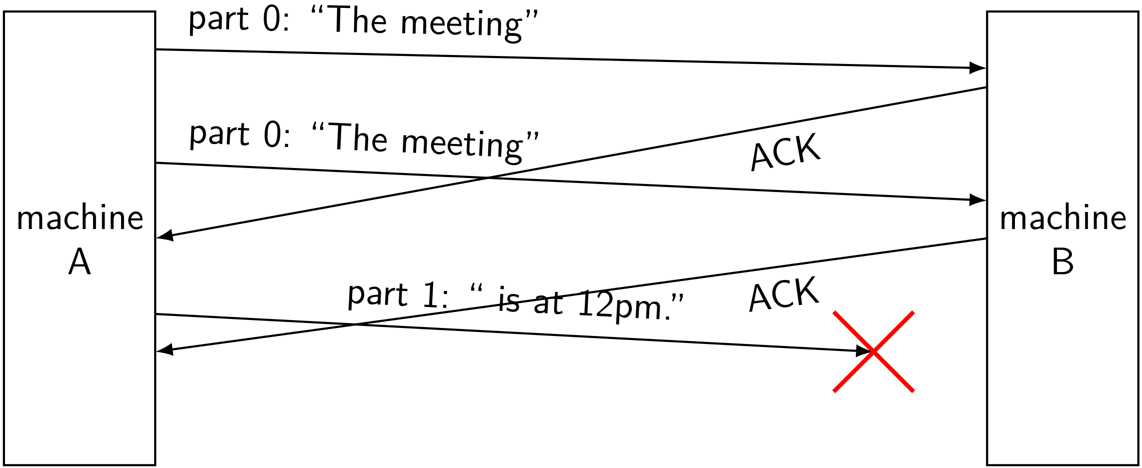
reconstructed message:
The meeting is at 12pm.

splitting messages: try 2 — missed ack



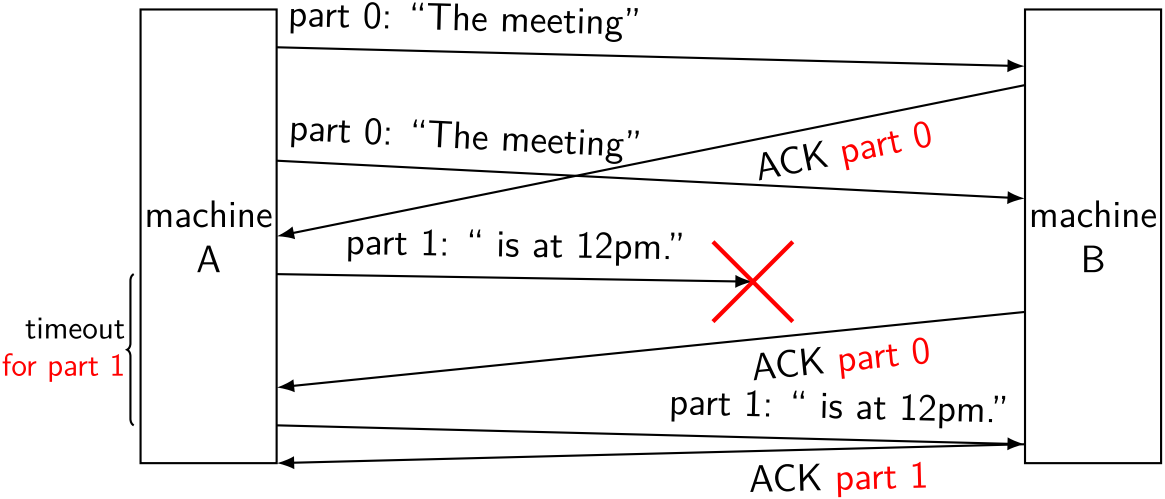
reconstructed message:
The meeting is at 12pm.

splitting messages: try 2 — problem



A thinks: part 0 + part 1 acknowleged!

splitting messages: version 3



sequence numbers

call the ‘part’ label *sequence number*

for now: sequence number = message (or *segment*) number

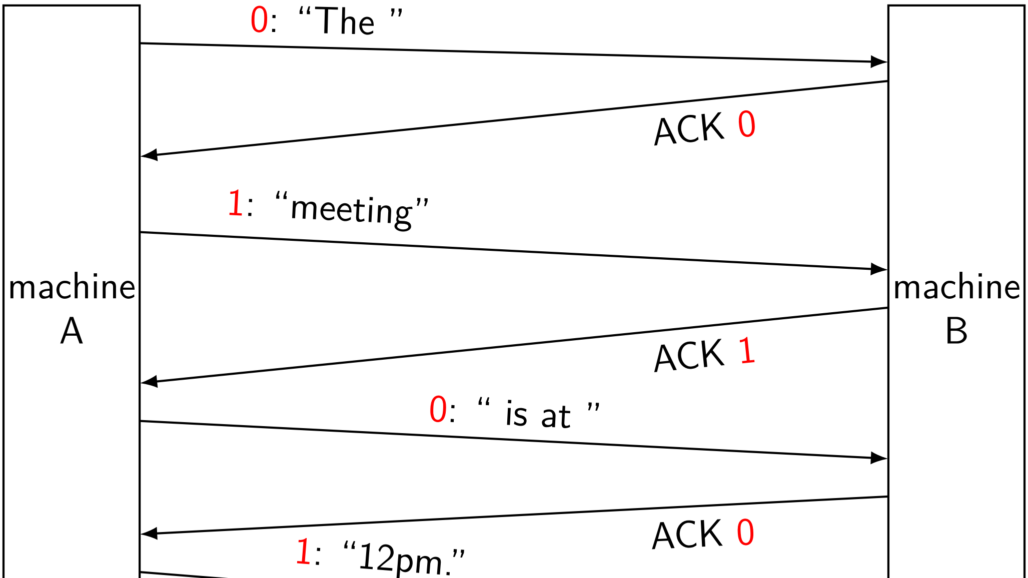
in TCP: sequence number = byte number

important question: how large can they get?

if we never reuse them — infinite!

so *really* want to reuse them

1-bit sequence number



‘stop and wait’

machine A is only sending one thing at a time

never start sending next thing until after sending previous thing

stop-and-wait exercise (receive, 1)

machine B receives $\boxed{\texttt{0: X}}$

machine B sends $\boxed{\text{ACK } \texttt{0}}$

machine B receives $\boxed{\texttt{0: X}}$

what should machine B do now?

A. send $\boxed{\text{ACK } \texttt{0}}$ B. send $\boxed{\text{ACK } \texttt{1}}$ C. send nothing

stop-and-wait exercise (receive, 2)

machine B receives `0: X`

machine B sends `ACK 0`

machine B receives `1: X`

what should machine B do now?

A. send `ACK 0` B. send `ACK 1` C. send nothing

stop-and-wait exercise (receive, 3)

machine B receives `0: X`

machine B sends `ACK 0`

machine B receives `1: Y`

machine B sends `ACK 1`

machine B receives `0: X`

what should machine B do now?

A. send `ACK 0` B. send `ACK 1` C. send nothing

stop-and-wait exercise (send, 1)

A trying to send 'X', then 'Y', then 'Z'

machine A sends `0: X`

machine A sends `0: X`

machine A receives `ACK 0`

machine A sends `1: Y`

machine A receives `ACK 0`

what should machine A do now?

A. send `0: X` again B. send `1: Y` again

C. send `0: Z` D. something else

stop-and-wait exercise (send, 2)

A trying to send 'X', then 'Y', then 'Z'

machine A sends $\boxed{\texttt{0: X}}$

machine A sends $\boxed{\texttt{0: X}}$

machine A receives $\boxed{\text{ACK } \texttt{0}}$

machine A sends $\boxed{\texttt{1: Y}}$

machine A receives $\boxed{\text{ACK } \texttt{1}}$

what should machine A do now?

A. send $\boxed{\texttt{0: X}}$ again B. send $\boxed{\texttt{1: Y}}$ again

C. send $\boxed{\texttt{0: Z}}$ D. something else

stop-and-wait issues

two issues with stop-and-wait:

doesn't use close to full capacity of network

not clear how to set timeouts

looking at metrics

several important metrics we'll care about

(both for this and future topics)

looking at metrics

several important metrics we'll care about

(both for this and future topics)

throughput and *bandwidth* ($\sim$ how much capacity used/available)

latency and *round-trip time* (RTT) ($\sim$ what timeouts needed)

jitter ($\sim$ safety margin for timeouts)

bandwidth / throughput

bandwidth / data rate: maximum rate we can send per unit time

- most commonly measuring the speed of a link

1 gigabit/second = transmit 1 bit / nanosecond

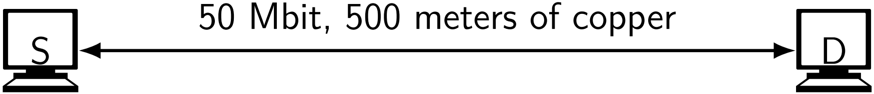
throughput: acheived rate per unit time

- often lower than total bandwidth because of losses
- (we'll give several examples throughout the semester)

latency (1)

latency: time for message: SOURCE $\rightarrow$ DEST

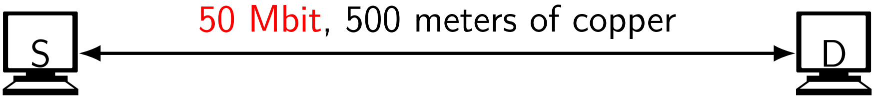
example: 1000 bit message from S to D:



latency (1)

latency: time for message: SOURCE $\rightarrow$ DEST

example: 1000 bit message from S to D:



one bit sent each $1/50$M second $= 0.02\ \mu$s

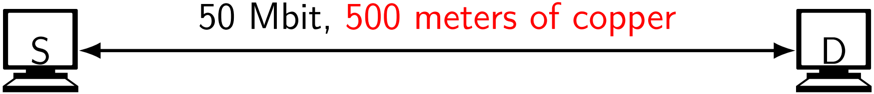
1000 bits take $0.02 \times 1000 = 20\ \mu$s to sent

"transmission delay"

latency (1)

latency: time for message: SOURCE $\rightarrow$ DEST

example: 1000 bit message from S to D:



one bit sent each $1/50$M second $= 0.02\ \mu$s

1000 bits take $0.02 \times 1000 = 20\ \mu$s to sent

“transmission delay”

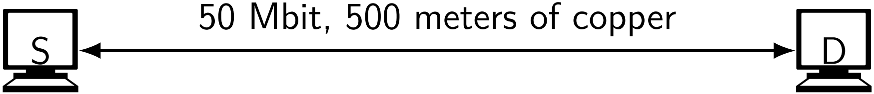
$+\ 2.2$ microseconds for bit to go down cable $(2.3 \times 10^8$ m/s$)$

“propogation delay”

latency (1)

latency: time for message: SOURCE $\rightarrow$ DEST

example: 1000 bit message from S to D:



one bit sent each $1/50\text{M}$ second $=$ 0.02 μs

1000 bits take $0.02 \times 1000 = 20$ μs to sent

"transmission delay"

$+\ 2.2$ microseconds for bit to go down cable (2.3×10^8 m/s)

"propogation delay"

total latency of about 22.2 μs

latency (1, ex)

exercise: latency for 20000 bit message from S to D

assume speed of signal through fiber of 2.0×10^8 m/s

latency (1, ex)

exercise: latency for 20000 bit message from S to D

assume speed of signal through fiber of 2.0×10^8 m/s

latency (1, ex)

exercise: latency for 20000 bit message from S to D

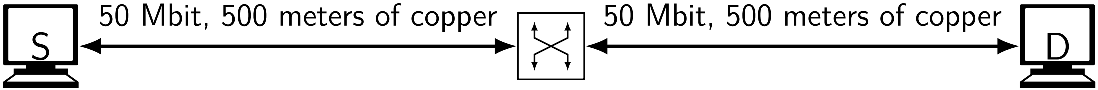
assume speed of signal through fiber of 2.0×10^8 m/s

latency (2)

example: 1000 bit packet from S to D

assume when message is received:

- 5 other 1000-bit packets in queue; no extra bits between packets
- no other switch processing time



S to switch, switch to D: 22.2 μs (transmit+propogate delay)

within switch: wait $20 \times 5 = 100$ μs for 5 other packets ($20\mu s = 1$ packet transmit delay)

- “queueing delay”

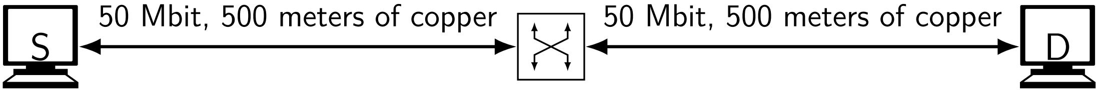
total latency: $22.2 + 100 + 22.2 = 144.4$ microseconds

latency (2)

example: 1000 bit packet from S to D

assume when message is received:

- 5 other 1000-bit packets in queue; no extra bits between packets
- no other switch processing time



S to switch, switch to D: 22.2 μs (transmit+propogate delay)

within switch: wait $20 \times 5 = 100$ μs for 5 other packets ($20\mu s$ = 1 packet transmit delay)

- "queueing delay"

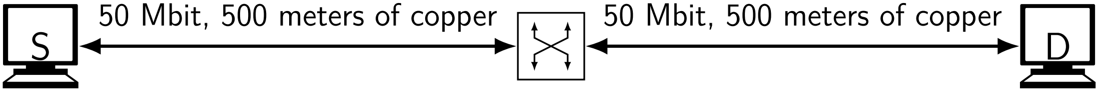
total latency: $22.2 + 100 + 22.2 = 144.4$ microseconds

latency (2)

example: 1000 bit packet from S to D

assume when message is received:

- 5 other 1000-bit packets in queue; no extra bits between packets
- no other switch processing time



S to switch, switch to D: 22.2 μs (transmit+propogate delay)

within switch: wait $20 \times 5 = 100$ μs for 5 other packets ($20\mu s = 1$ packet transmit delay)

- "queueing delay"

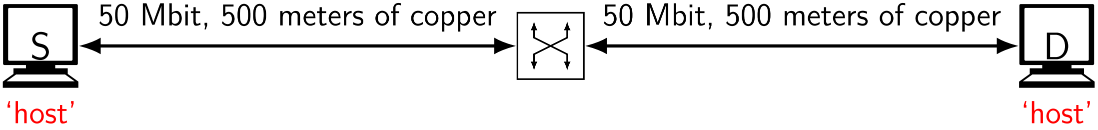
total latency: $22.2 + 100 + 22.2 = 144.4$ microseconds

latency (2)

example: 1000 bit packet from S to D

assume when message is received:

- 5 other 1000-bit packets in queue; no extra bits between packets
- no other switch processing time



S to switch, switch to D: 22.2 μs (transmit+propogate delay)

within switch: wait $20 \times 5 = 100$ μs for 5 other packets ($20\mu s = 1$ packet transmit delay)

- "queueing delay"

round trip time

round-trip-time (RTT): time for message:
SOURCE $\rightarrow$ DEST $\rightarrow$ SOURCE

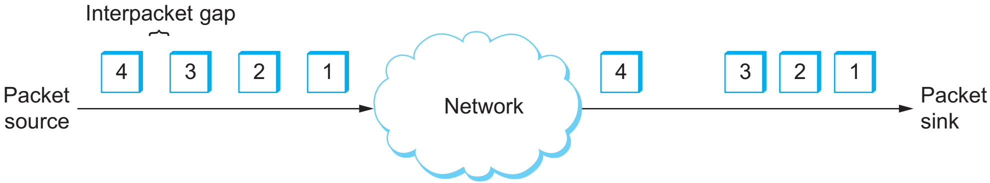
much easier to measure than one-way latency

typically how we'll set latency

jitter

variation in latency

most commonly from changing queuing delays



measuring round-trip time (1a)

```
charles@reisst14$ ping 1.1.1.1
PING 1.1.1.1 (1.1.1.1) 56(84) bytes of data.
64 bytes from 1.1.1.1: icmp_seq=1 ttl=52 time=13.8 ms
64 bytes from 1.1.1.1: icmp_seq=2 ttl=52 time=15.0 ms
64 bytes from 1.1.1.1: icmp_seq=3 ttl=52 time=12.5 ms
64 bytes from 1.1.1.1: icmp_seq=4 ttl=52 time=12.3 ms
64 bytes from 1.1.1.1: icmp_seq=5 ttl=52 time=13.5 ms
64 bytes from 1.1.1.1: icmp_seq=6 ttl=52 time=12.5 ms
64 bytes from 1.1.1.1: icmp_seq=7 ttl=52 time=13.3 ms
64 bytes from 1.1.1.1: icmp_seq=8 ttl=52 time=13.2 ms
64 bytes from 1.1.1.1: icmp_seq=9 ttl=52 time=13.3 ms
64 bytes from 1.1.1.1: icmp_seq=10 ttl=52 time=14.1 ms
^C
--- 1.1.1.1 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9014ms
rtt min/avg/max/mdev = 12.273/13.343/15.024/0.786 ms
```

measuring round-trip-time (1b)

No.	Time	Source	Destination	Protocol	Length	Info
17	2.766137597	172.25.188.87	1.1.1.1	ICMP	98	Echo (ping) request id=0x0002, seq=1/256, ttl=
18	2.779916793	1.1.1.1	172.25.188.87	ICMP	98	Echo (ping) reply id=0x0002, seq=1/256, ttl=
28	3.768363948	172.25.188.87	1.1.1.1	ICMP	98	Echo (ping) request id=0x0002, seq=2/512, ttl=
29	3.783346606	1.1.1.1	172.25.188.87	ICMP	98	Echo (ping) reply id=0x0002, seq=2/512, ttl=
59	4.769791044	172.25.188.87	1.1.1.1	ICMP	98	Echo (ping) request id=0x0002, seq=3/768, ttl=
60	4.782213735	1.1.1.1	172.25.188.87	ICMP	98	Echo (ping) reply id=0x0002, seq=3/768, ttl=
73	5.771827936	172.25.188.87	1.1.1.1	ICMP	98	Echo (ping) request id=0x0002, seq=4/1024, ttl
74	5.784055865	1.1.1.1	172.25.188.87	ICMP	98	Echo (ping) reply id=0x0002, seq=4/1024, ttl
79	6.773358205	172.25.188.87	1.1.1.1	ICMP	98	Echo (ping) request id=0x0002, seq=5/1280, ttl
80	6.786831460	1.1.1.1	172.25.188.87	ICMP	98	Echo (ping) reply id=0x0002, seq=5/1280, ttl
81	7.775177274	172.25.188.87	1.1.1.1	ICMP	98	Echo (ping) request id=0x0002, seq=6/1536, ttl
82	7.787654160	1.1.1.1	172.25.188.87	ICMP	98	Echo (ping) reply id=0x0002, seq=6/1536, ttl
85	8.776273952	172.25.188.87	1.1.1.1	ICMP	98	Echo (ping) request id=0x0002, seq=7/1792, ttl
86	8.789562086	1.1.1.1	172.25.188.87	ICMP	98	Echo (ping) reply id=0x0002, seq=7/1792, ttl
93	9.777262659	172.25.188.87	1.1.1.1	ICMP	98	Echo (ping) request id=0x0002, seq=8/2048, ttl
94	9.790425188	1.1.1.1	172.25.188.87	ICMP	98	Echo (ping) reply id=0x0002, seq=8/2048, ttl
110	10.778251280	172.25.188.87	1.1.1.1	ICMP	98	Echo (ping) request id=0x0002, seq=9/2304, ttl
119	10.791477471	1.1.1.1	172.25.188.87	ICMP	98	Echo (ping) reply id=0x0002, seq=9/2304, ttl
120	11.779834642	172.25.188.87	1.1.1.1	ICMP	98	Echo (ping) request id=0x0002, seq=10/2560, tt
121	11.793858285	1.1.1.1	172.25.188.87	ICMP	98	Echo (ping) reply id=0x0002, seq=10/2560, tt

measuring round-trip-time (1c)

```
▸ Frame 17: 98 bytes on wire (784 bits), 98 bytes captured (784
▸ Ethernet II, Src: f4:6d:3f:d3:64:59 (f4:6d:3f:d3:64:59), Dst:
▸ Internet Protocol Version 4, Src: 172.25.188.87, Dst: 1.1.1.1
▾ Internet Control Message Protocol
      Type: 8 (Echo (ping) request)
      Code: 0
      Checksum: 0xfa68 [correct]
      [Checksum Status: Good]
      Identifier (BE): 2 (0x0002)
      Identifier (LE): 512 (0x0200)
      Sequence Number (BE): 1 (0x0001)
      Sequence Number (LE): 256 (0x0100)
      [Response frame: 18]
      Timestamp from icmp data: Sep  2, 2024 12:59:20.000000000 E
      [Timestamp from icmp data (relative): 0.093073545 seconds]
    ▾ Data (48 bytes)
          Data: 7f6b0100000000001011121314151617181 91a1b1c1d1e1f20
          [Length: 48]
```

non-ICMP pings (1)

```
HPING www (enp0s31f6 128.143.67.8): NO FLAGS are set, 40 headers + 0 data
len=46 ip=128.143.67.8 ttl=63 DF id=0 sport=0 flags=RA seq=0 win=0 rtt=3.
len=46 ip=128.143.67.8 ttl=63 DF id=0 sport=0 flags=RA seq=1 win=0 rtt=3.
len=46 ip=128.143.67.8 ttl=63 DF id=0 sport=0 flags=RA seq=2 win=0 rtt=7.
len=46 ip=128.143.67.8 ttl=63 DF id=0 sport=0 flags=RA seq=3 win=0 rtt=6.
len=46 ip=128.143.67.8 ttl=63 DF id=0 sport=0 flags=RA seq=4 win=0 rtt=6.
len=46 ip=128.143.67.8 ttl=63 DF id=0 sport=0 flags=RA seq=5 win=0 rtt=6.
len=46 ip=128.143.67.8 ttl=63 DF id=0 sport=0 flags=RA seq=6 win=0 rtt=5.
len=46 ip=128.143.67.8 ttl=63 DF id=0 sport=0 flags=RA seq=7 win=0 rtt=5.
len=46 ip=128.143.67.8 ttl=63 DF id=0 sport=0 flags=RA seq=8 win=0 rtt=5.
len=46 ip=128.143.67.8 ttl=63 DF id=0 sport=0 flags=RA seq=9 win=0 rtt=4.
^C
--- www hping statistic ---
10 packets transmitted, 10 packets received, 0% packet loss
round-trip min/avg/max = 3.2/5.4/7.1 ms
```

non-ICMP pings (2)

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	128.143.71.27	128.143.67.8	TCP	54	1385 → 0 [<None>] Seq=1 Win=512 Len=0
2	0.000228953	128.143.67.8	128.143.71.27	TCP	60	0 → 1385 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
3	1.000254996	128.143.71.27	128.143.67.8	TCP	54	1386 → 0 [<None>] Seq=1 Win=512 Len=0
4	1.000555183	128.143.67.8	128.143.71.27	TCP	60	0 → 1386 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
5	2.000547949	128.143.71.27	128.143.67.8	TCP	54	1387 → 0 [<None>] Seq=1 Win=512 Len=0
6	2.000861315	128.143.67.8	128.143.71.27	TCP	60	0 → 1387 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
7	3.000765651	128.143.71.27	128.143.67.8	TCP	54	1388 → 0 [<None>] Seq=1 Win=512 Len=0
8	3.000975736	128.143.67.8	128.143.71.27	TCP	60	0 → 1388 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
9	4.000919160	128.143.71.27	128.143.67.8	TCP	54	1389 → 0 [<None>] Seq=1 Win=512 Len=0
10	4.001230544	128.143.67.8	128.143.71.27	TCP	60	0 → 1389 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
11	5.001235561	128.143.71.27	128.143.67.8	TCP	54	1390 → 0 [<None>] Seq=1 Win=512 Len=0
12	5.001548682	128.143.67.8	128.143.71.27	TCP	60	0 → 1390 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
13	6.001564520	128.143.71.27	128.143.67.8	TCP	54	1391 → 0 [<None>] Seq=1 Win=512 Len=0
14	6.001886342	128.143.67.8	128.143.71.27	TCP	60	0 → 1391 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
15	7.001812043	128.143.71.27	128.143.67.8	TCP	54	1392 → 0 [<None>] Seq=1 Win=512 Len=0
16	7.002151624	128.143.67.8	128.143.71.27	TCP	60	0 → 1392 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
17	8.002159545	128.143.71.27	128.143.67.8	TCP	54	1393 → 0 [<None>] Seq=1 Win=512 Len=0
18	8.002492960	128.143.67.8	128.143.71.27	TCP	60	0 → 1393 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
19	9.002512076	128.143.71.27	128.143.67.8	TCP	54	1394 → 0 [<None>] Seq=1 Win=512 Len=0
20	9.002826074	128.143.67.8	128.143.71.27	TCP	60	0 → 1394 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
21	10.002865037	128.143.71.27	128.143.67.8	TCP	54	1395 → 0 [<None>] Seq=1 Win=512 Len=0
22	10.003183918	128.143.67.8	128.143.71.27	TCP	60	0 → 1395 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0

measuring throughput?

```
$ scp test.dat portal.cs.virginia.edu:test.dat
test.dat                                 100%   32MB  23.0MB/s   00:01
$ scp portal.cs.virginia.edu:test.dat .
test.dat                                 100%   32MB  28.2MB/s   00:01
```

(but might be measuring disk speed instead)

also more specialized tools like `iperf`

- require program to run on both ends

measuring throughput

```
$ iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size:  128 KByte (default)
------------------------------------------------------------
[  1] local 128.143.71.87 port 5001 connected with 128.143.71.27 port 5476
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-10.0147 sec  1.09 GBytes   934 Mbits/sec
```

—

```
$ iperf -c kytos02 | tee iperf.out
------------------------------------------------------------
Client connecting to kytos02, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  1] local 128.143.71.27 port 54760 connected with 128.143.71.87 port 500
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-10.0256 sec  1.09 GBytes   933 Mbits/sec
```

measuring transmission delay?

```
PING www.cs.virginia.edu (128.143.67.8) 1400(1428) bytes of data.
--- www.cs.virginia.edu ping statistics ---
1000 packets transmitted, 1000 received, 0% packet loss, time 50638ms
rtt min/avg/max/mdev = 0.319/0.461/1.222/0.039 ms
$ ping -s 16 www -i 0.05 -c 1000 -q
PING www.cs.virginia.edu (128.143.67.8) 16(44) bytes of data.
--- www.cs.virginia.edu ping statistics ---
1000 packets transmitted, 1000 received, 0% packet loss, time 50995ms
rtt min/avg/max/mdev = 0.156/0.345/1.539/0.068 ms
```

approx. $0.461 - 0.345 = 0.116$ ms delay for $1400 - 16$ extra bytes

with two links in each direction = approx $\frac{0.116}{4} = 0.029$ ms/link

$\frac{1400 - 16\text{byte}}{0.029\text{ms}} \approx 50$ Mbit/sec (does not match Gigabit ethernet)

probably other processing time besides sending on links, though

stop-and-wait performance

stop-and-wait protocol

assuming no packets lost/corrupted

about one packet per round-trip time

example: local ethernet

my home wired network: 0.6 ms round trip time

typical packet has about 1400 bytes = 11200 bits of data

throughput with stop-and-wait:
$11200\text{b}/0.6\text{ms} \approx 19000\text{b}/\text{ms} = 19\,000\,000\text{b}/\text{s} = 19\text{Mbit}/\text{s}$

available bandwidth is about 1 Gbit/s

example: local ethernet

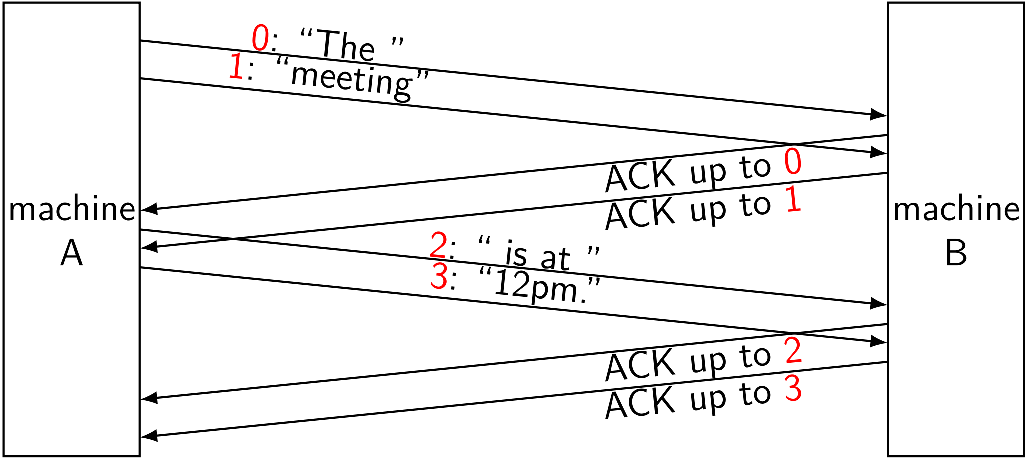
my home wired network: 0.6 ms round trip time

typical packet has about 1400 bytes $=$ 11200 bits of data

throughput with stop-and-wait:
$11200\text{b}/0.6\text{ms} \approx 19000\text{b}/\text{ms} = 19\ 000\ 000\text{b}/\text{s} = 19\text{Mbit}/\text{s}$

available bandwidth is about 1 Gbit/s

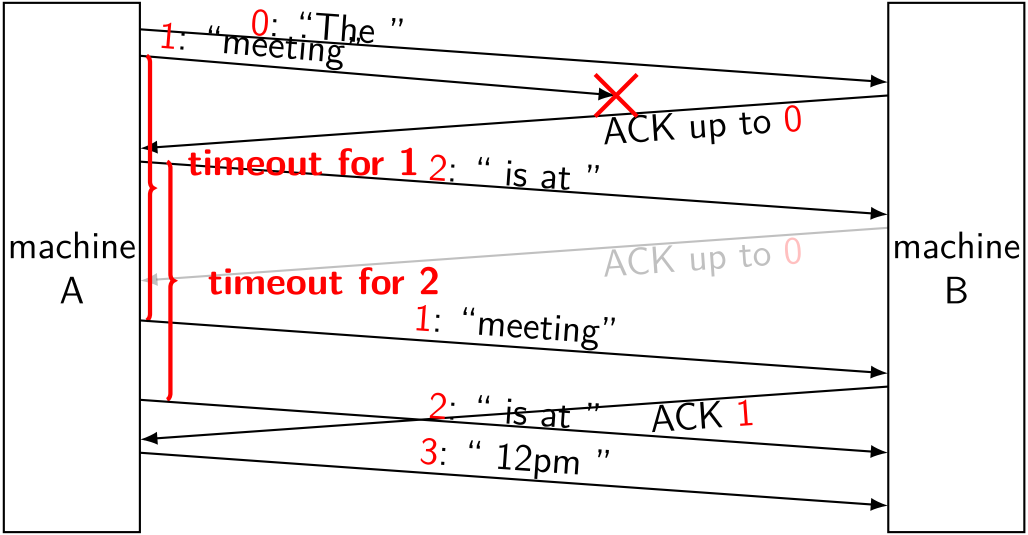
sending two at a time



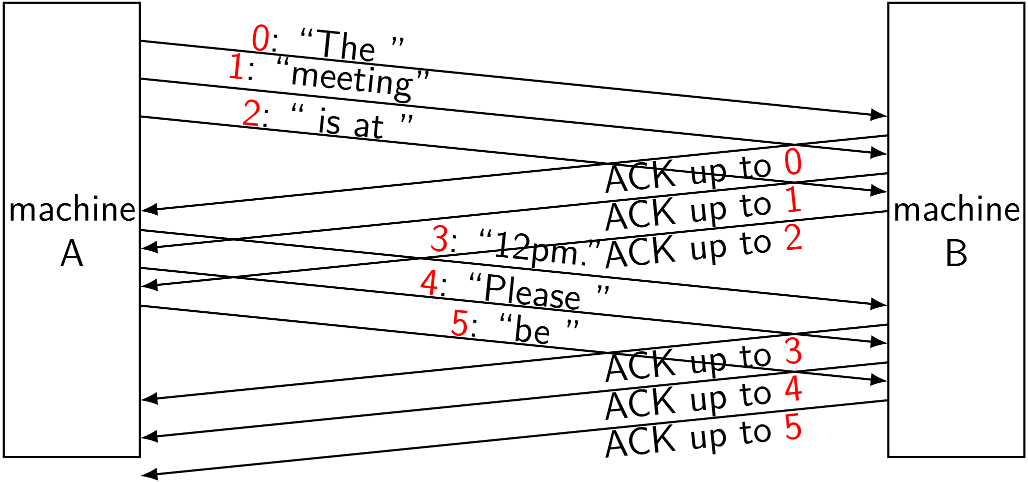
(ACK up to X = ACK X and everything before it)

key idea: always have two in flight

timeouts per message



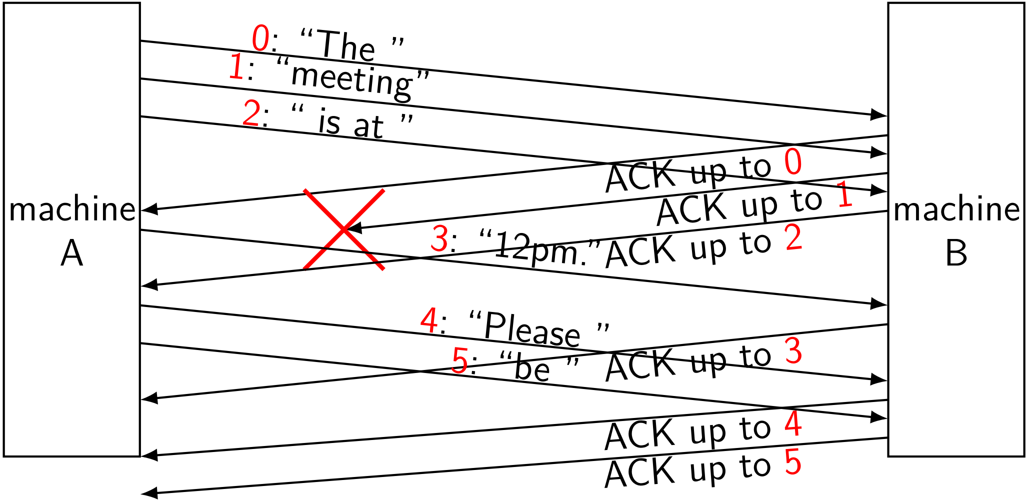
sending three at a time



choose "window size" to have in flight

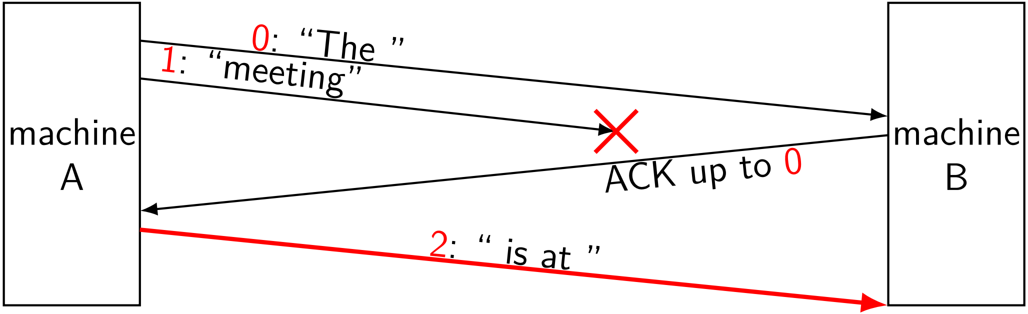
send when previous acknowledged

lost ACKs?



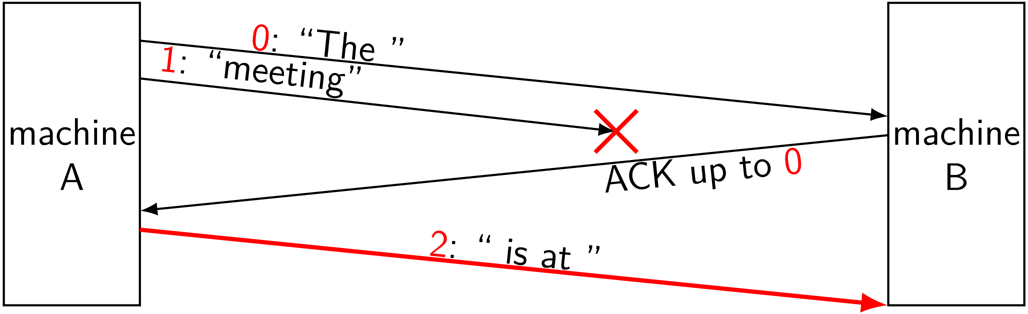
machine A
machine B
0: "The "
1: "meeting"
2: " is at "
ACK up to 0
ACK up to 1
3: "12pm."
ACK up to 2
4: "Please "
5: "be "
ACK up to 3
ACK up to 4
ACK up to 5

missing messages?



question: what should receiver do with sequence number 2?

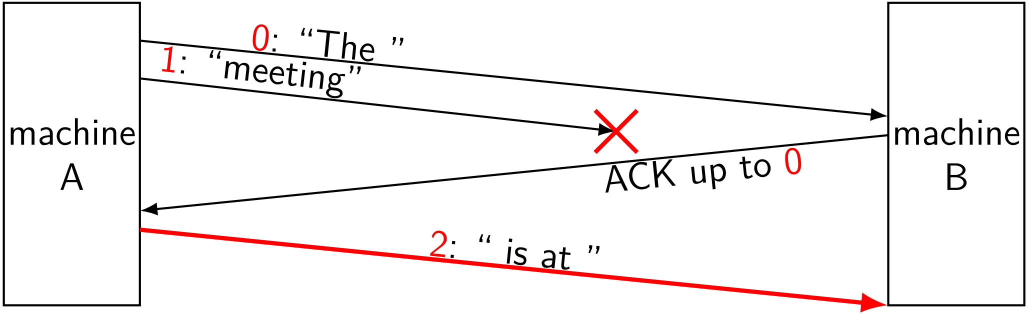
missing messages?



question: what should receiver do with sequence number 2?

one idea: ignore it?

missing messages?

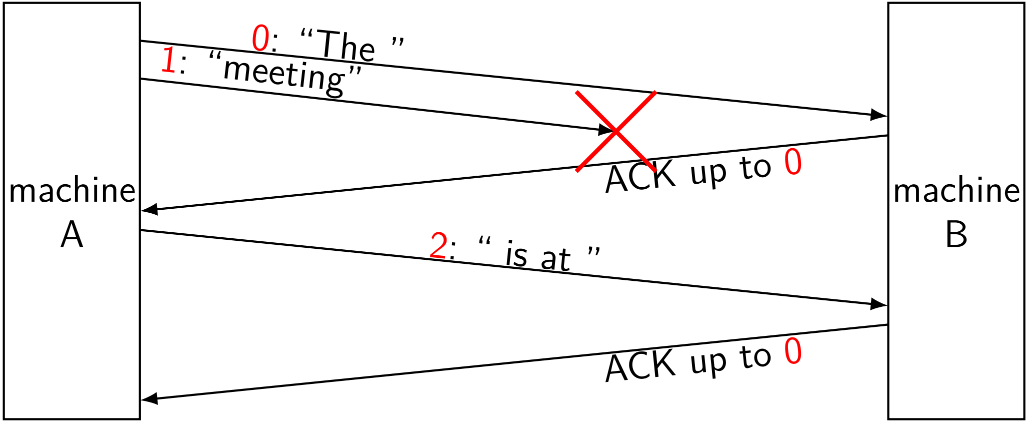


question: what should receiver do with sequence number 2?

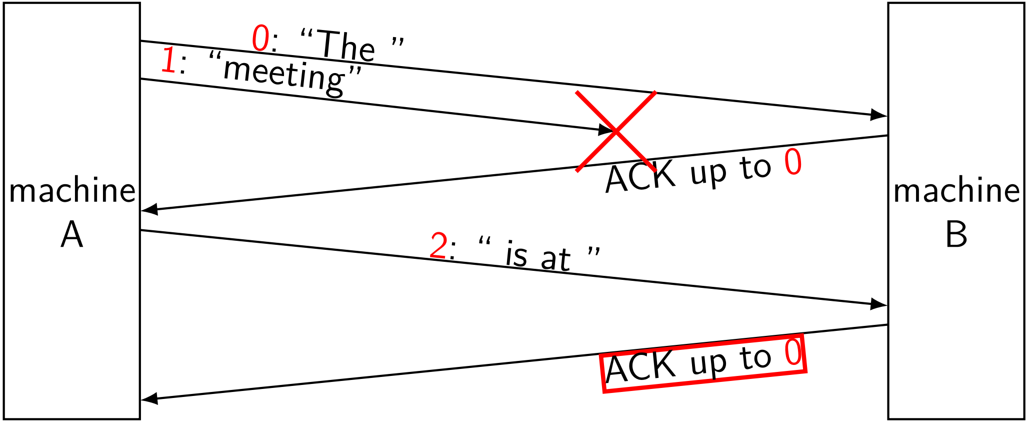
one idea: ignore it?

better idea: send something back to sender

better idea: always ACK



better idea: always ACK



only ACK x if everything up to and including x received

intuition: ACK tells sender where to start sending more

fast retransmit

if large window + data packet 2 is lost, then sender will see

ACK 0, ACK 1, ACK 1, ACK 1, ACK 1, ACK 1

duplicate ACKs indicate missing packet 2

shouldn't wait for timeout

fast retransmit

if large window + data packet 2 is lost, then sender will see

ACK 0, ACK 1, ACK 1, ACK 1, ACK 1, ACK 1

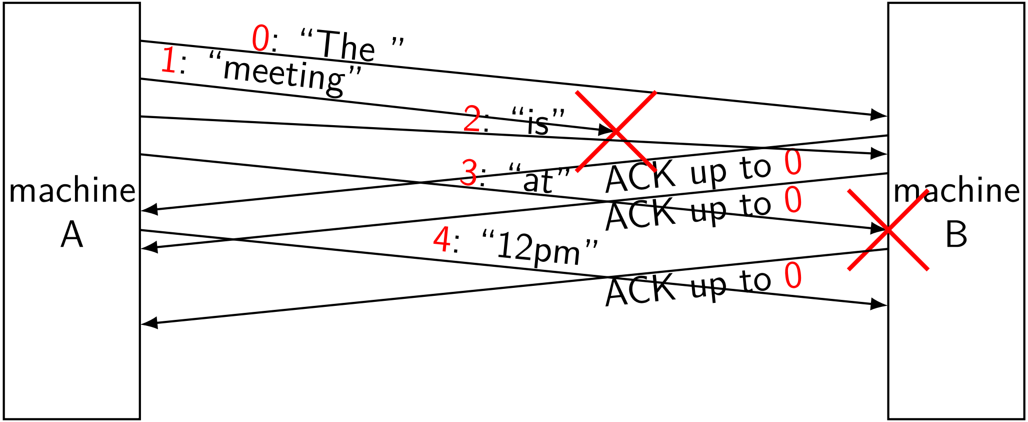
duplicate ACKs indicate missing packet 2

shouldn't wait for timeout

$\rightarrow$ TCP heuristic: retransmit immediately after $\sim$3 duplicate ACKs

- not 1 duplicate ACK to tolerate some reordering
- also some other details (we'll talk later)

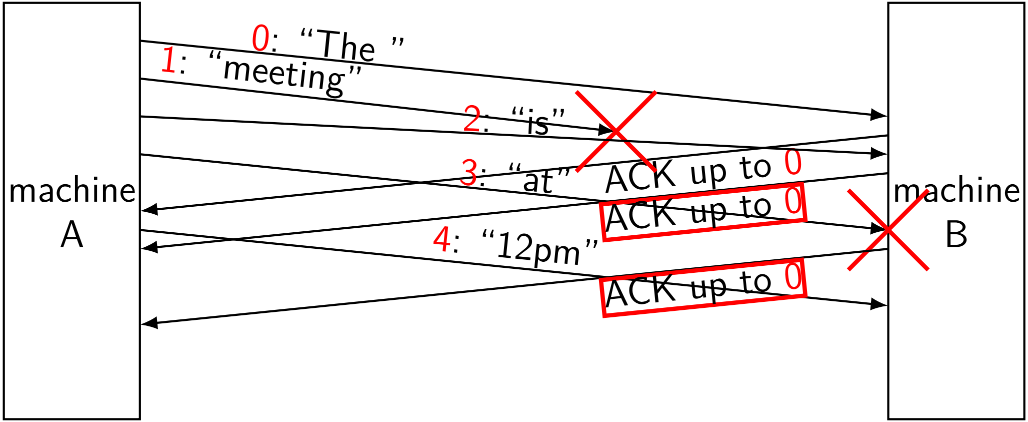
multiple missing



duplicate ACK heuristic will quickly resend 1, but not 3

would like to supply better information

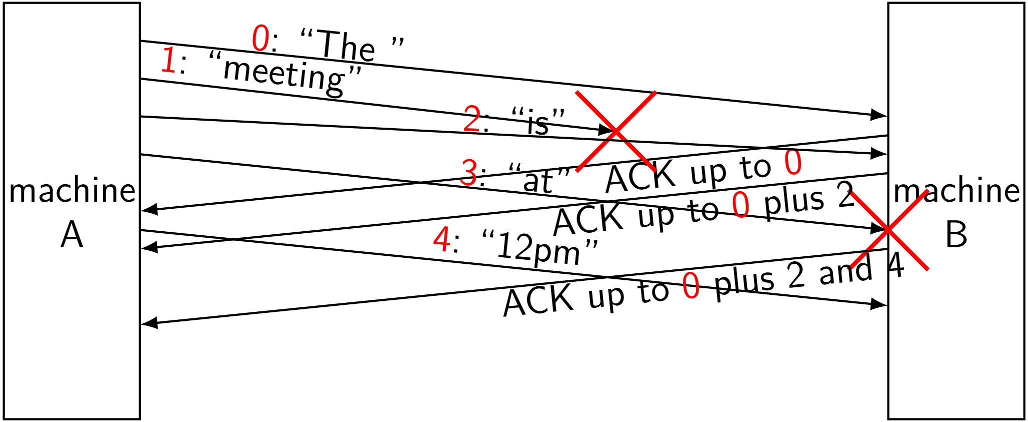
multiple missing



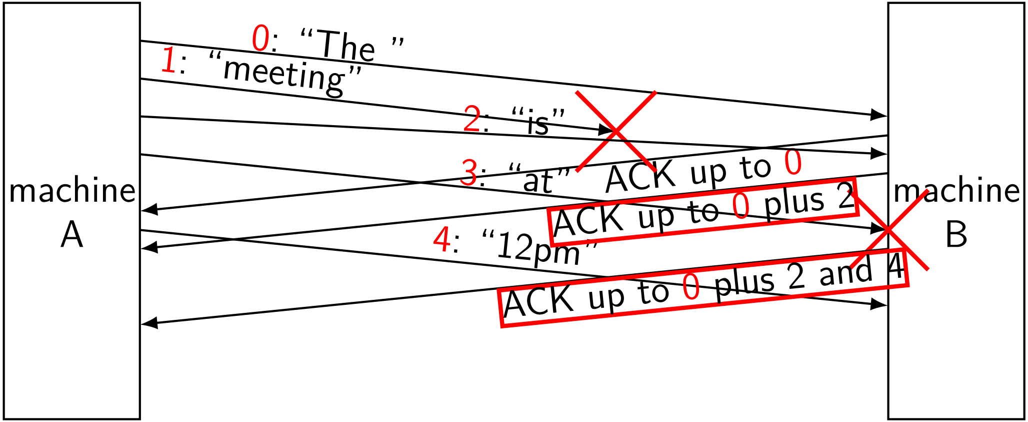
duplicate ACK heuristic will quickly resend 1, but not 3

would like to supply better information

selective acknowledgments



selective acknowledgments



selective acknowledgments in TCP

optional feature (“extension”) described in RFC 2018

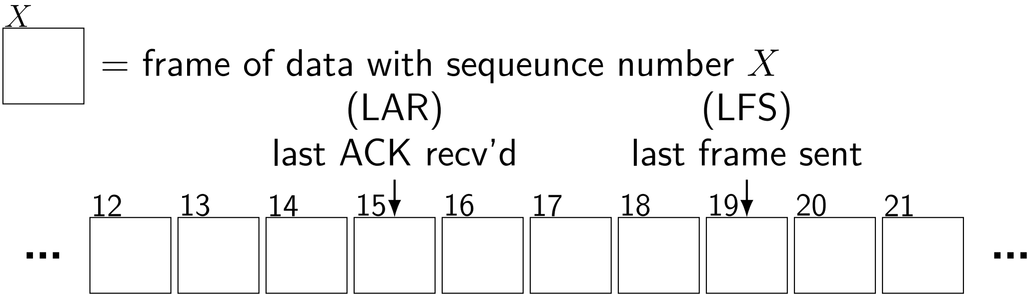
send list of ranges received

typically room for 3 ranges

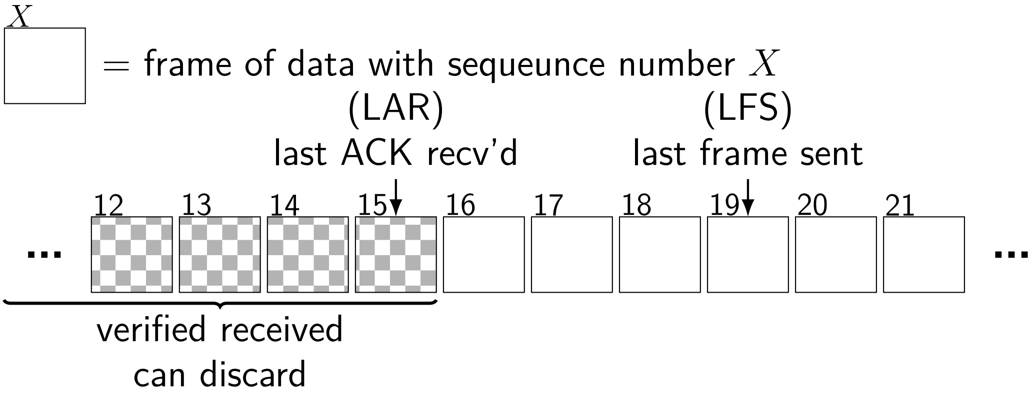
if more than 3 ranges to report, then:

- include range with most recently received frame
- include other ranges until sent three times

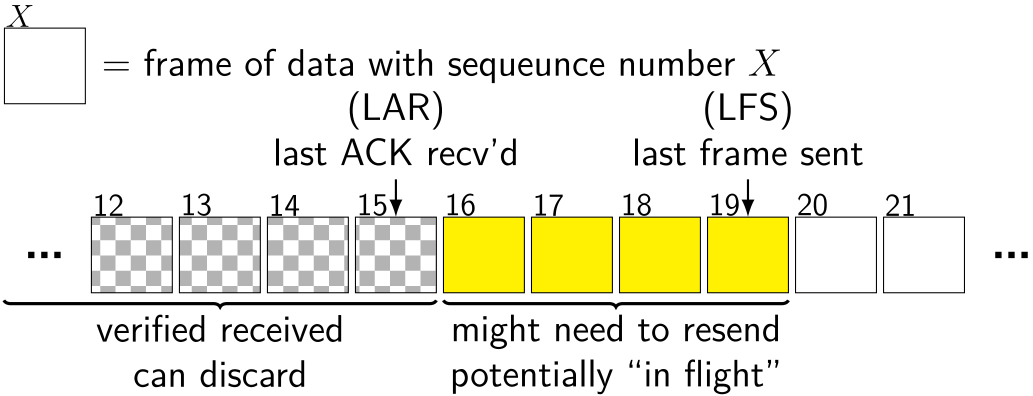
sender window tracking



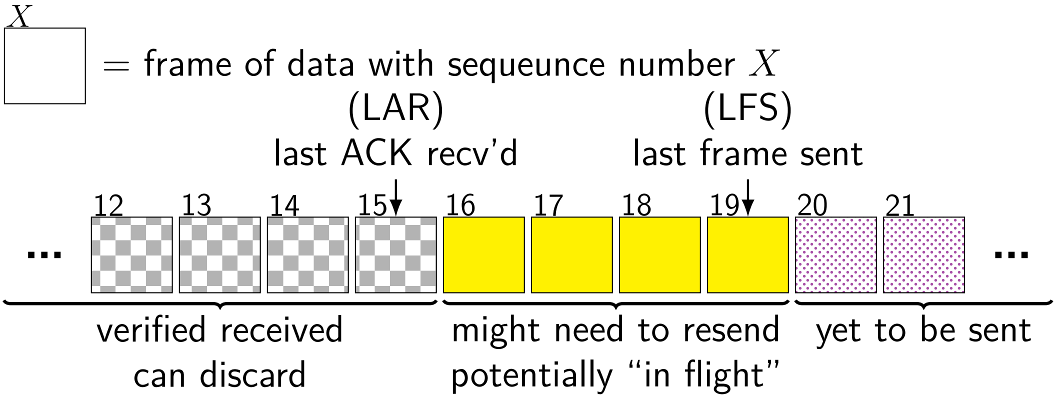
sender window tracking



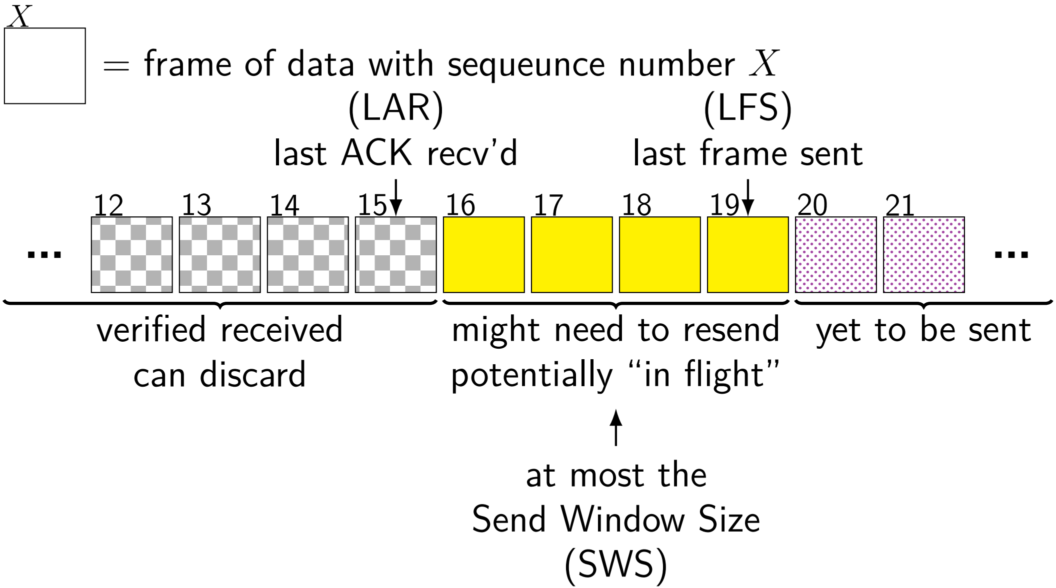
sender window tracking



sender window tracking



sender window tracking



exercise 1: out-of-bounds ACK

last ACK recv'd (LAR)	10
last frame sent (LFS)	15
send window size (SWS)	5

what probably happened if we receive an ACK for...

9? 10? 13? 16?

A. only possible if network reorders frames

B. only possible from undetected frame corruption

C. lost ACK for frame ≤ 10

D. lost ACK for frame > 10

E. lost frame 11

F. resent frame from timeout

exercise 2: sender logic

last ACK recv'd (LAR)	10
last frame sent (LFS)	15
send window size (SWS)	5

In this case, there's a timeout that will trigger frame 13 to be resent. If still active, this timeout should be cancelled upon …

A. receiving ACK 12 B. receiving ACK 13
C. receiving ACK 14 D. sending frame 16

exercise 3a: new data

last ACK recv'd (LAR)	4
last frame sent (LFS)	6
send window size (SWS)	5

if we compute a new frame of data with sequence number 7 to eventually send, we should

A. send it now, advancing LFS
B. wait until we get an ACK for 5 or 6 to send it
C. wait until we get an ACK for 6 to send it
D. wait until the frame with sequence number 6 is resent to send it D. so

exercise 3b: new data

last ACK recv'd (LAR)	4
last frame sent (LFS)	8
send window size (SWS)	4

if we compute a new frame of data with sequence number 9 to eventually send, we should

A. send it now, advancing LFS
B. wait until we get an ACK for 5 or 6 to send it
C. wait until we get an ACK for 6 to send it
D. decline to accept the data because we will never be able to send it
E. something else

sender logic summarized

track variables:

- LFS (last frame sent)
- LAR (last ACK recv'd)
- SWS (send window size)

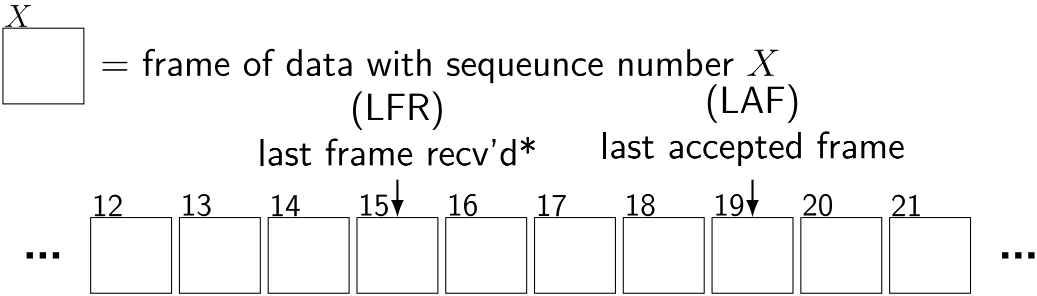
when receiving ACK $LAR < X \leq LFS$:

- LAR $\leftarrow X$
- clear any timers to resend frames $\leq X$

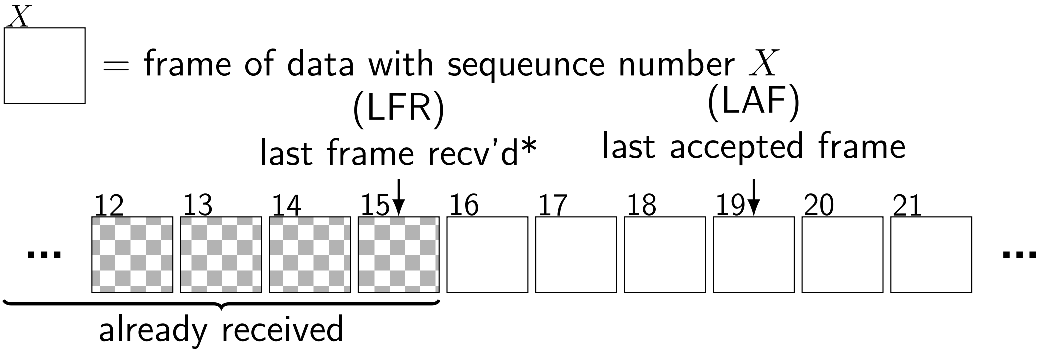
whenever SWS [send window size] $>$ LFS - LAR *and* data for frame LFS $+$ 1 is available:

- send frame LFS $+$ 1
- set timer to resend frame LFS $+$ 1
- LFS $\leftarrow$ LFS $+$ 1

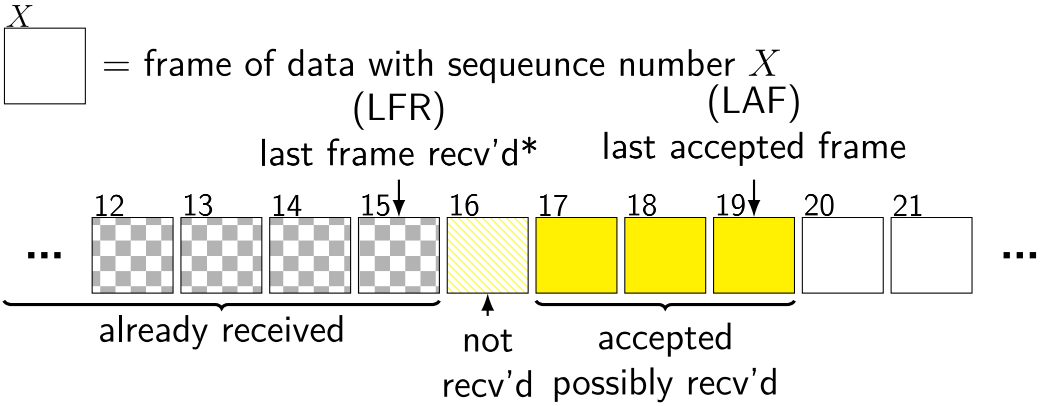
receiver window tracking



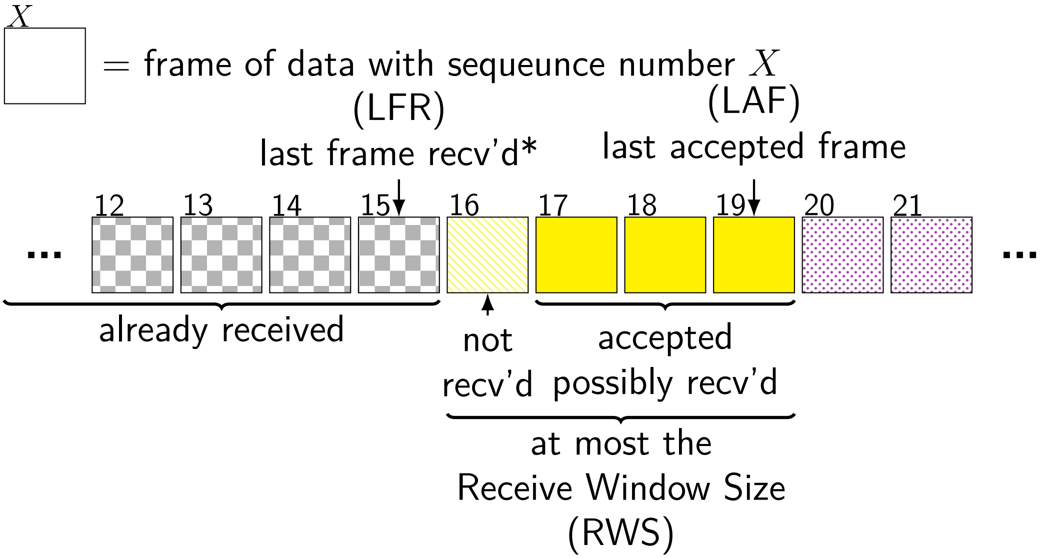
receiver window tracking



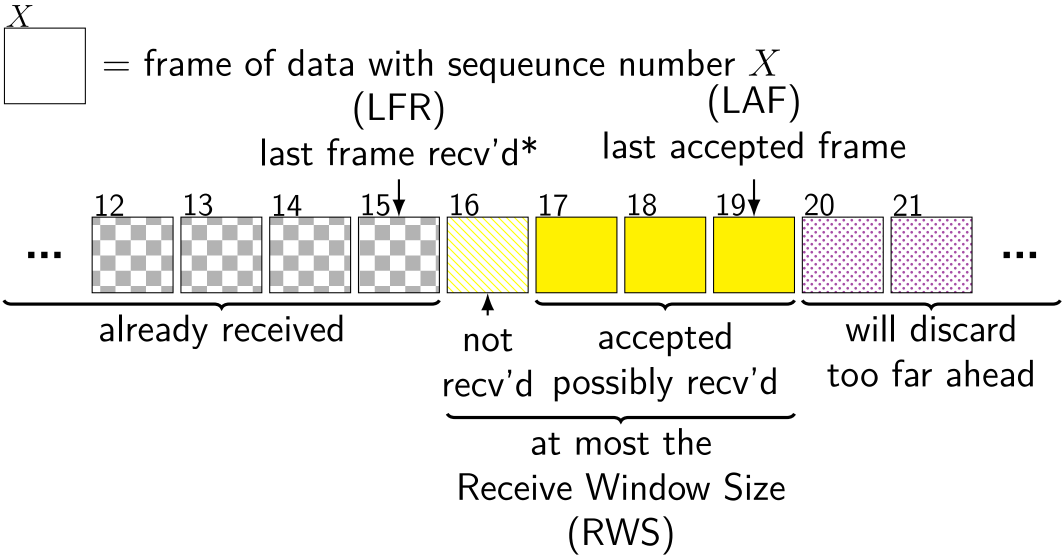
receiver window tracking



receiver window tracking



receiver window tracking



receiver logic summarized

track variables:

- LFR (last frame recv'd) — excludes frames after a missing frame
- LAF (last accepted frame)
- RWS (receive window size)

when receiving frame $LFR < X \leq LAF$:

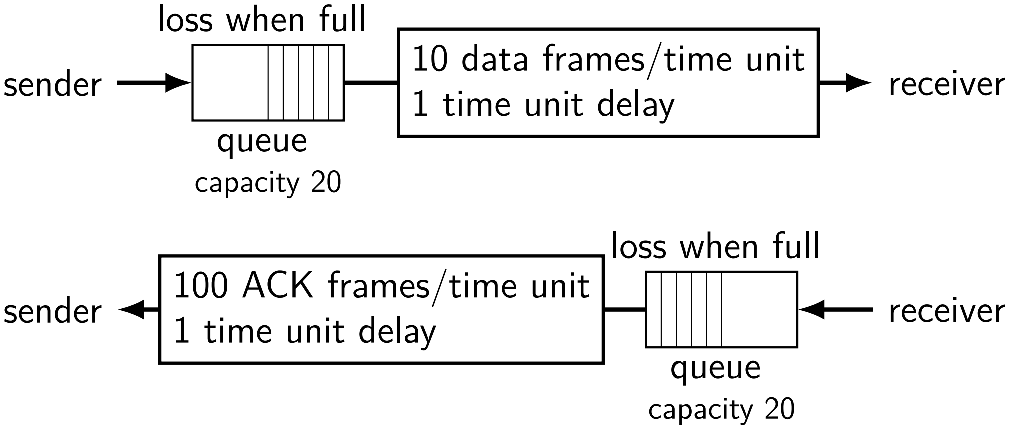
LFR $\leftarrow$ (first missing frame after LFR) - 1

- only advances if X $=$ LFR
- could advance by more than one if frames previously out of order

LAS $\leftarrow$ LFR $+$ RWS

- only advances if X $=$ LFR

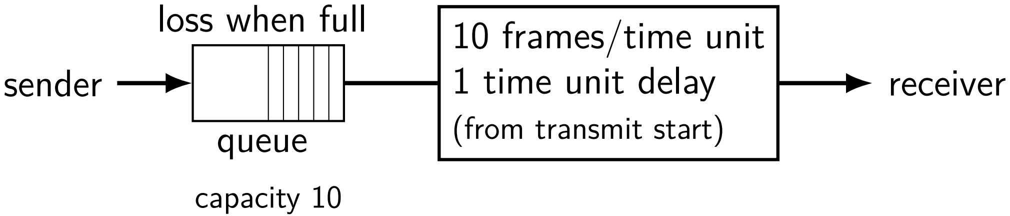
simple network model



simulator from upcoming assignment

command line `--delay 1 --bandwidth-forward 10 --bandwidth-backward 100 --buffer 30`

exercise: forward latency

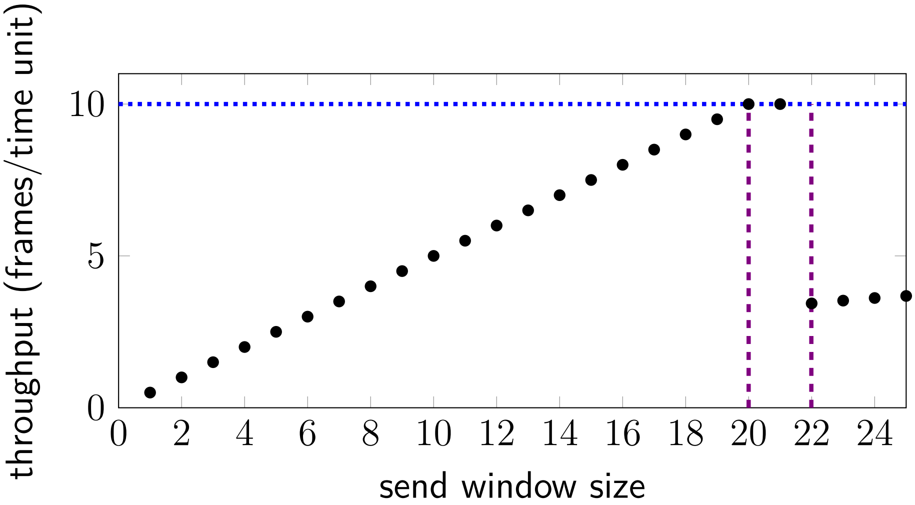


minimum latency = 1 time unit

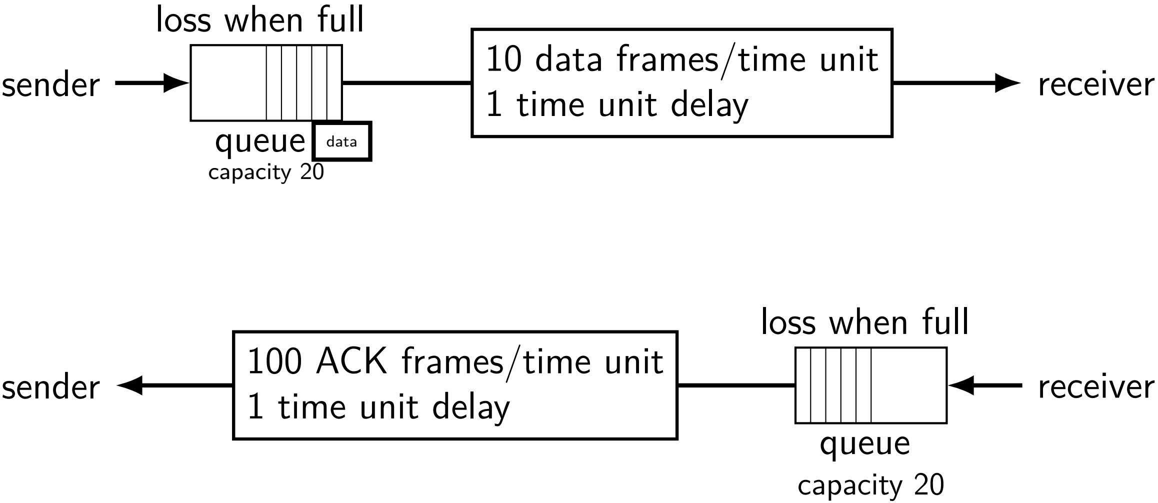
exercise: maximum latency?

A. 1 time unit
B. 1.1 time unit
C. 1.2 time unit
C. 1.4 time unit
D. 1.9 time unit
E. 2.0 time unit
F. 2.1 time unit
G. something else

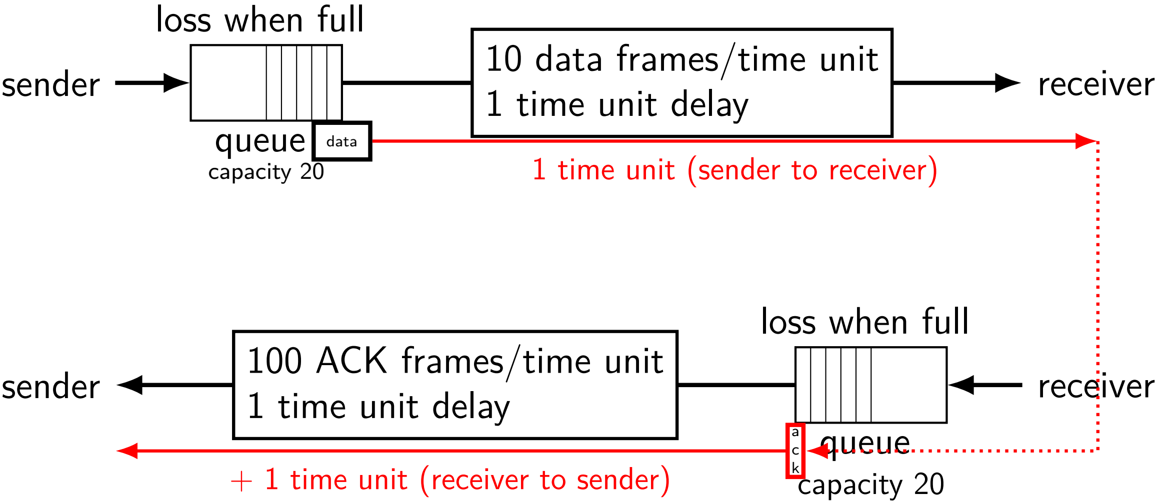
throughput and window size



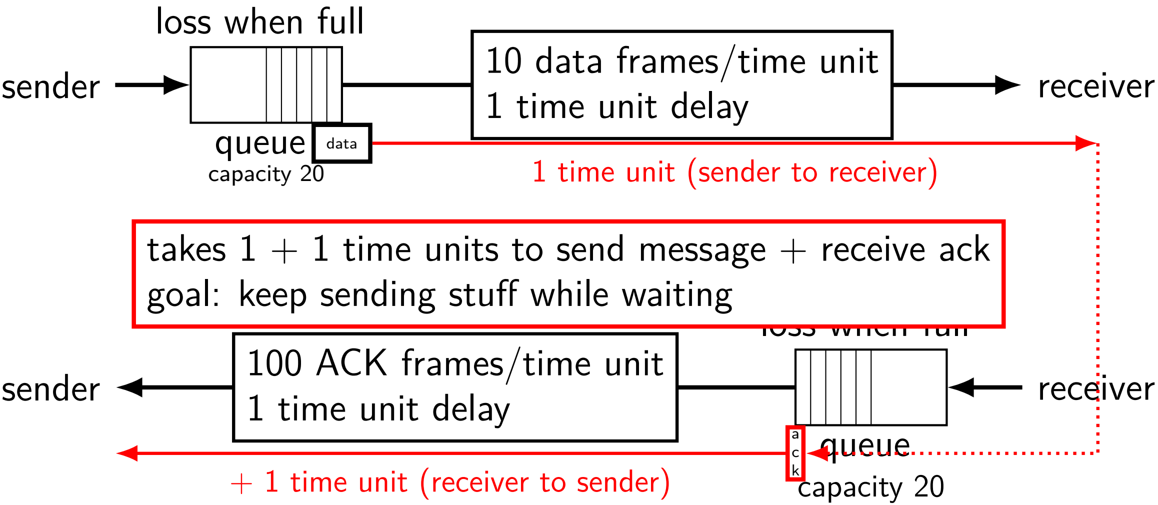
packet transit time



packet transit time



packet transit time



takes 1 + 1 time units to send message + receive ack

goal: keep sending stuff while waiting

filling the pipe

round-trip time of 2 time units

from send data to receive ACK (assuming no queuing delay)

can send 10 data frames per time unit

$=$ can send 20 data frames while waiting for ACK

filling the pipe

round-trip time of 2 time units

from send data to receive ACK (assuming no queuing delay)

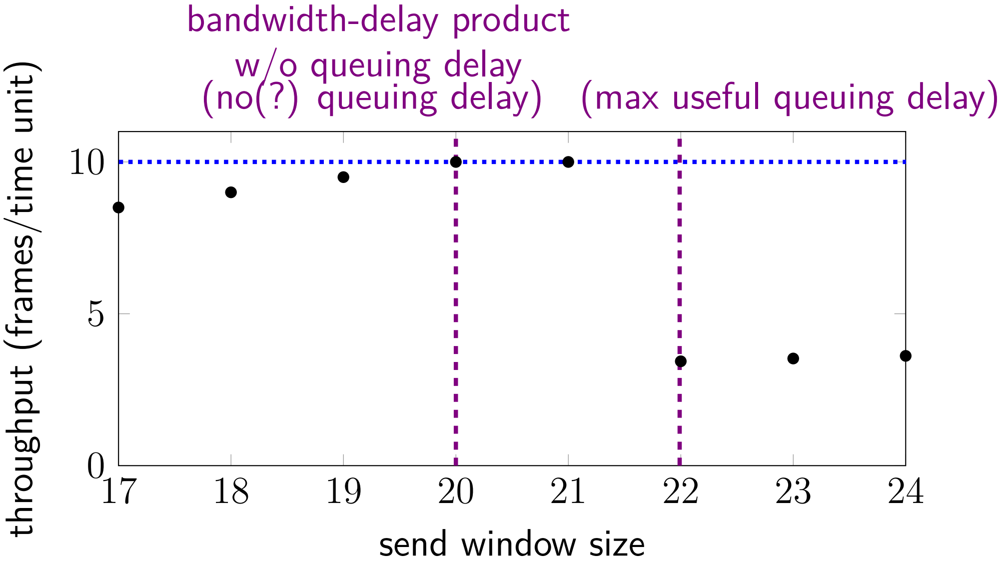
can send 10 data frames per time unit

$=$ can send 20 data frames while waiting for ACK

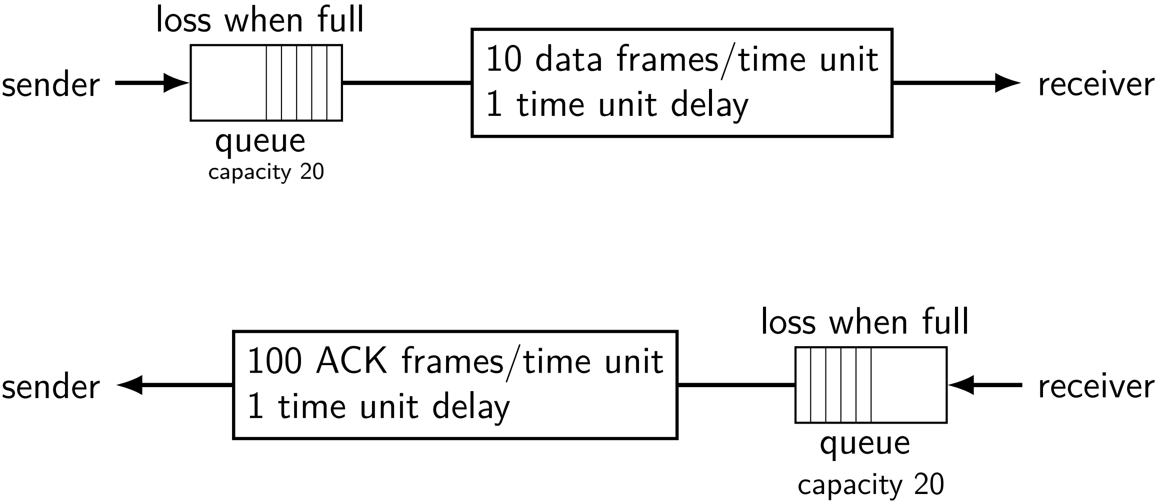
“bandwidth-delay product”

10/time unit (banwidth) times 2 time unit (RTT $=$ delay)

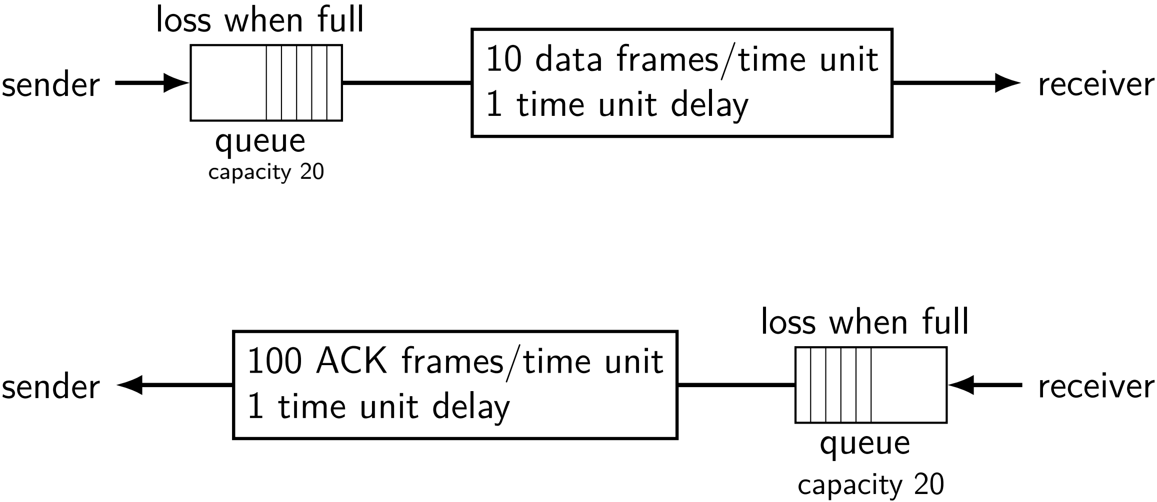
throughput and window size (detail)



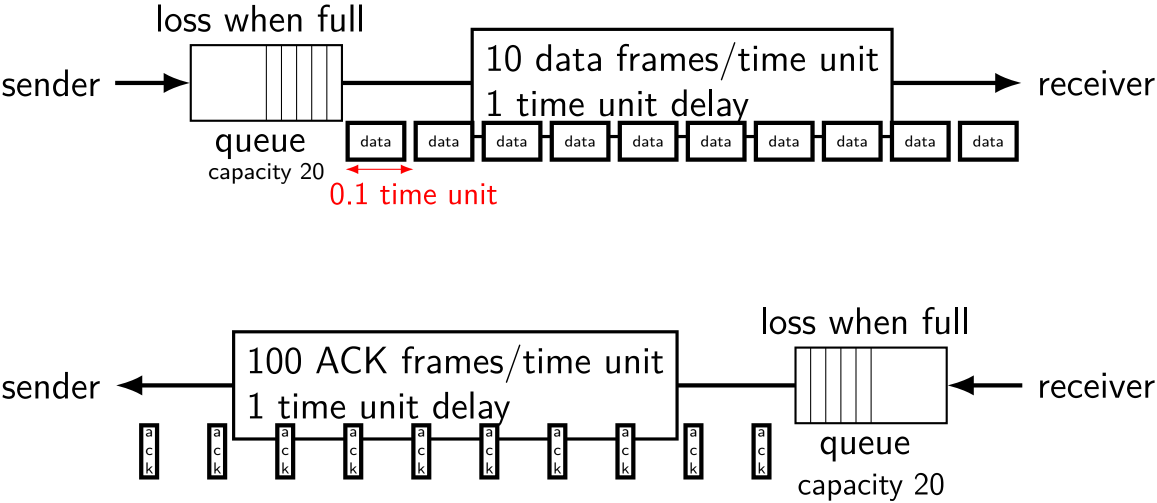
filling the pipe



filling the pipe



filling the pipe



on bursts

max possible queuing delay suggests window size of 30

approx. 3 time units times 10

problem: “bursts” temporarily exceed queue size

achievable average queue size not that high

sender could moderate by “pacing” packets

sliding windows used to solve...

flow control

keep sender from getting too far ahead of receiver
...by having window sizes set correctly
how? receiver tells sender what window size is okay

congestion control

keep network from being overloaded
(while making good use of available bandwidth)
...by having window sizes set correctly
how? it's complicated — big topic later

sequence number wraparound

protocol so far requires arbitrarily large sequence numbers

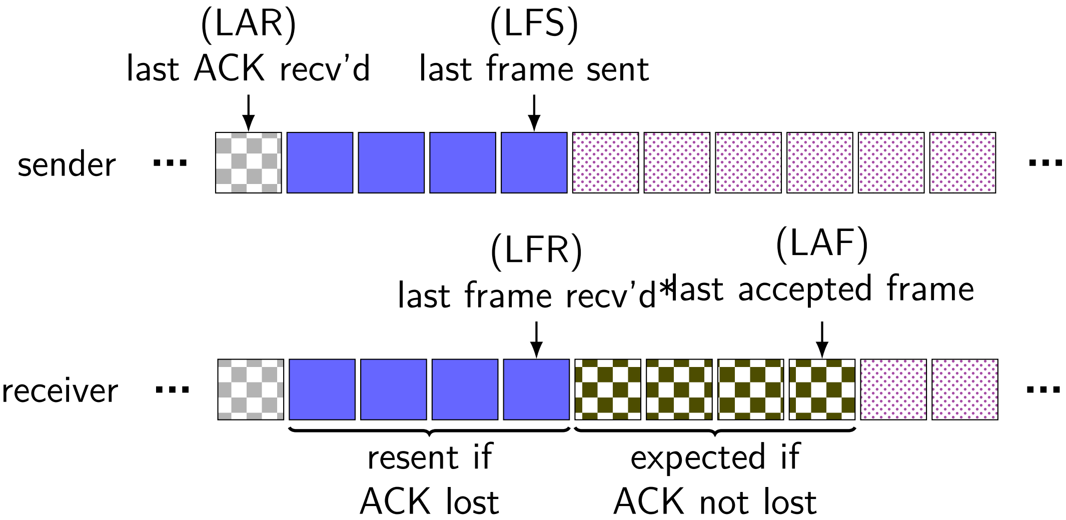
- doing $<$ and $>$ checks on sequence number, so they need to increase

would like to use smaller sequence numbers

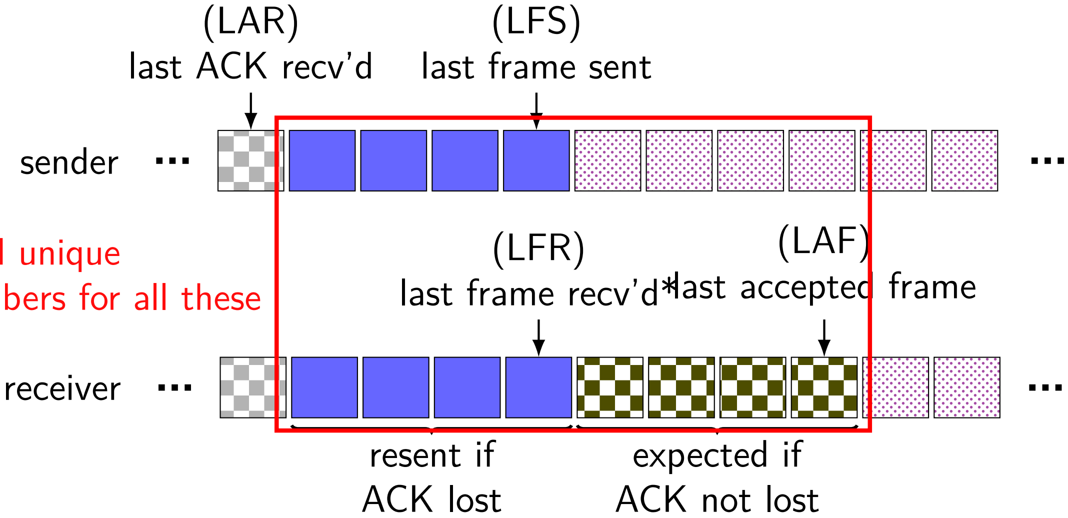
- think: transferring multi-gigabyte file

question: what goes wrong when we reuse sequeunce numbers?

sender/receiver desync: missing ACKs

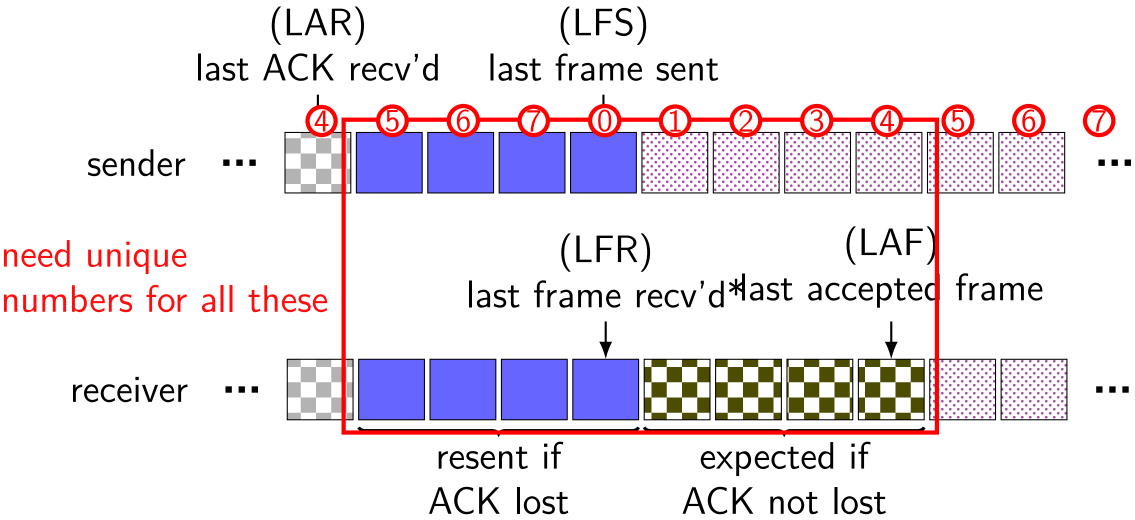


sender/receiver desync: missing ACKs

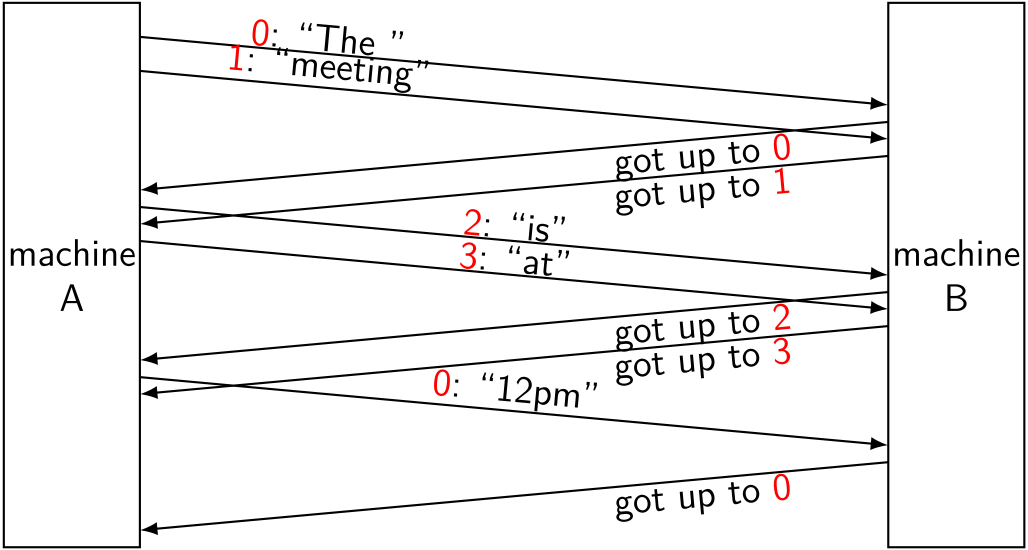


need unique numbers for all these

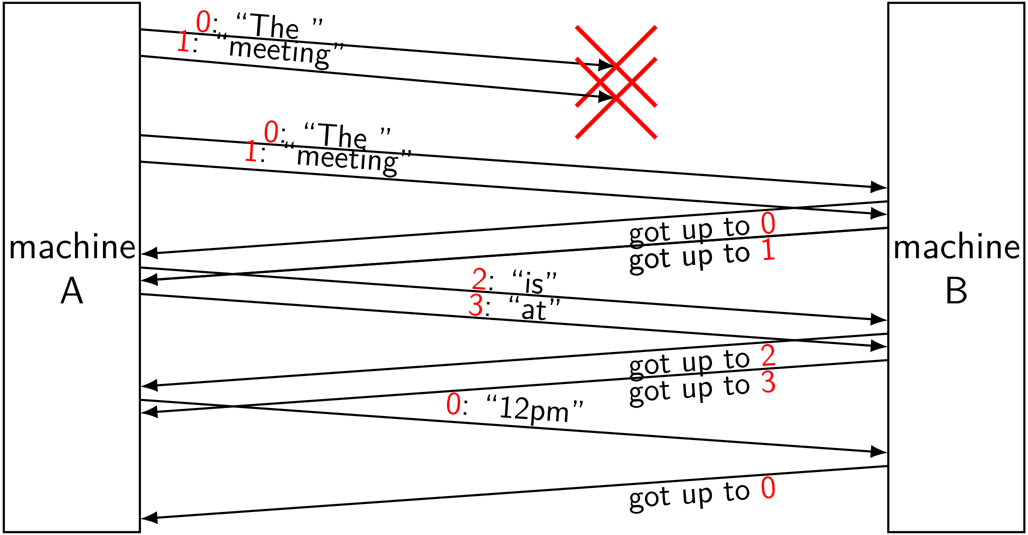
sender/receiver desync: missing ACKs



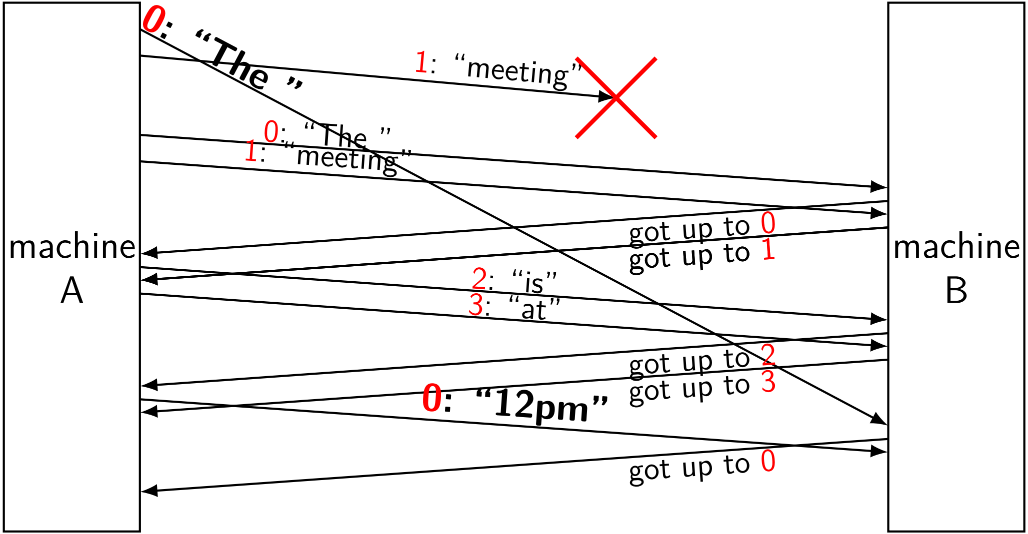
wraparound


machine A
machine B
0: "The "
1: "meeting"
got up to 0
got up to 1
2: "is"
3: "at"
got up to 2
got up to 3
0: "12pm"
got up to 0

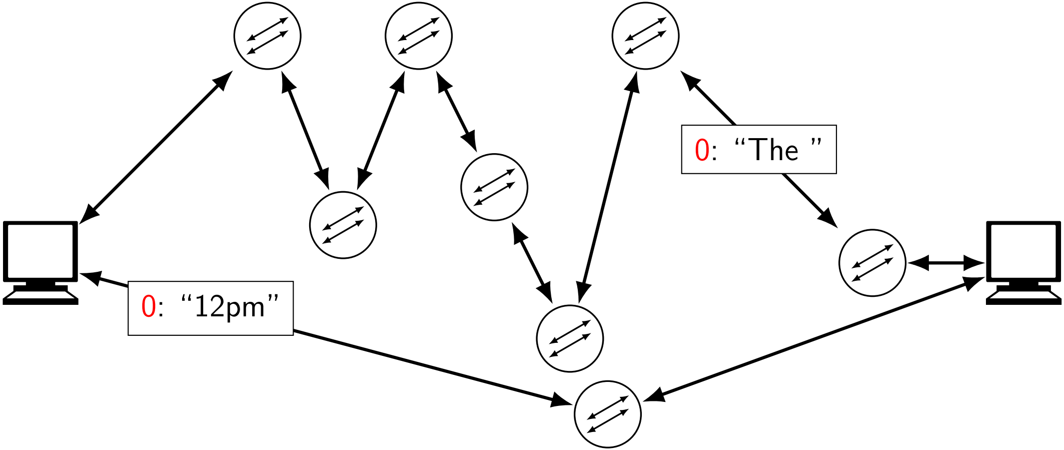
loss and resend?



very bad reordering



possible reason



sequence numbers in practice

TCP tries to assume 120 second "maximum segment lifetime"

segment = TCP's name for a packet

original TCP used 32-bit sequence number identifying *byte* number (not segment number)

problem: means wraparound happens on modern (Gigibit+) links in seconds!

sequence numbers in practice

TCP tries to assume 120 second "maximum segment lifetime"

segment = TCP's name for a packet

original TCP used 32-bit sequence number identifying *byte* number (not segment number)

problem: means wraparound happens on modern (Gigibit+) links in seconds!

workaround: add *additional* 32-bit timestamp field

used to detect/discard duplicates

can also be used to set timeouts and/or window sizes

TCP

transmission control protocol (TCP)

implements reliable streams of bytes

similar mechanism to what we've described

TCP extras/differences

bidirectional —

separate sequence numbers in each direction
can combine data (from A to B) with acknowledgment (from B to A)

sequence numbers are byte numbers —

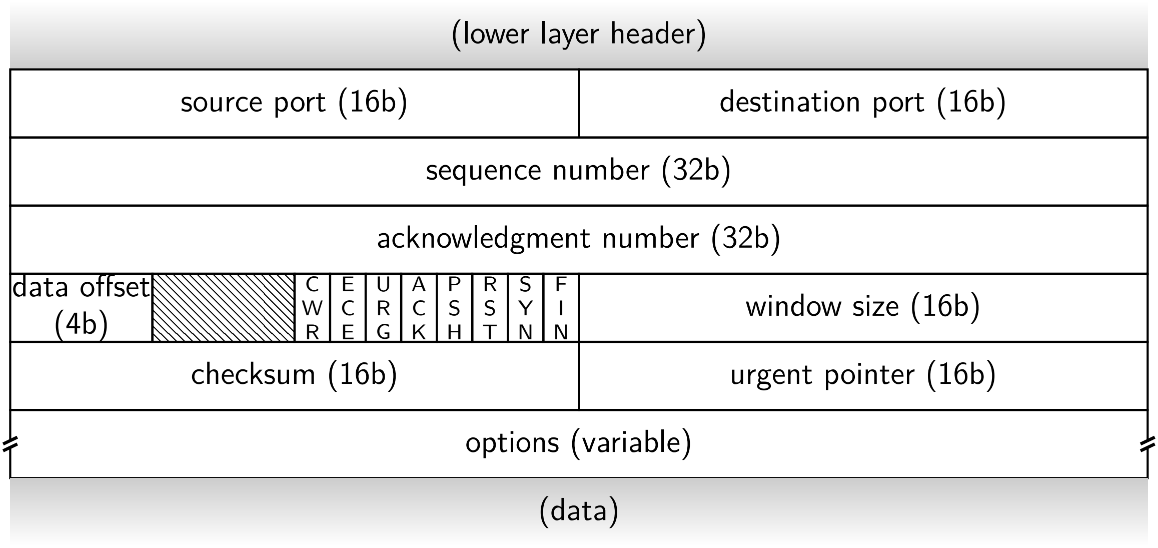
can retransmit data in different sized packets
sequence numbers = index of *first byte* sent
acknowledgment numbers = 1 + index of *last byte* acknowledged

dynamic/variable window sizes

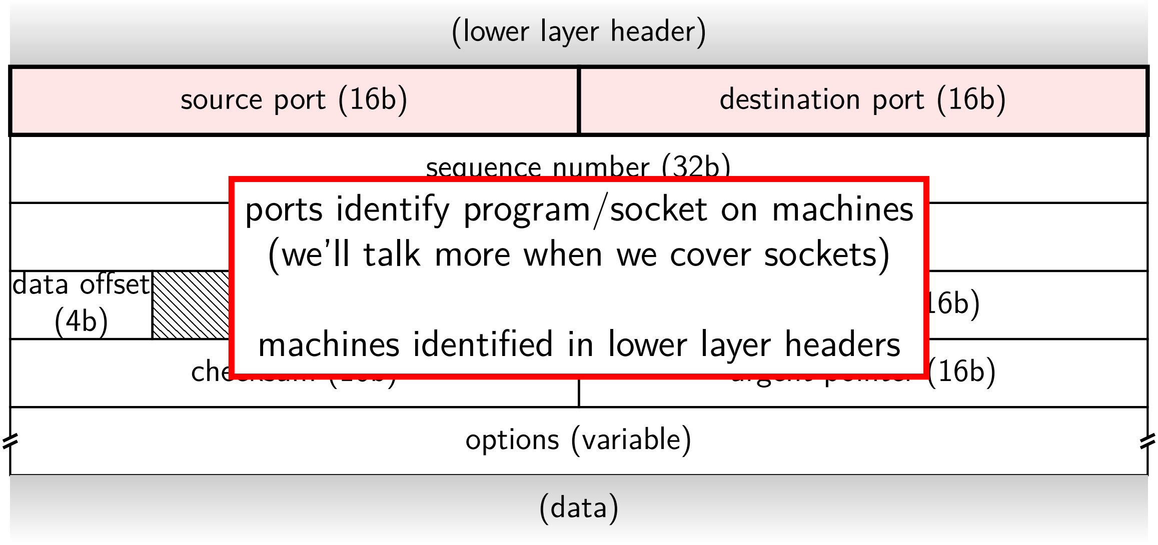
we'll discuss strategies later

offical name for packets = *segments*

TCP segment format



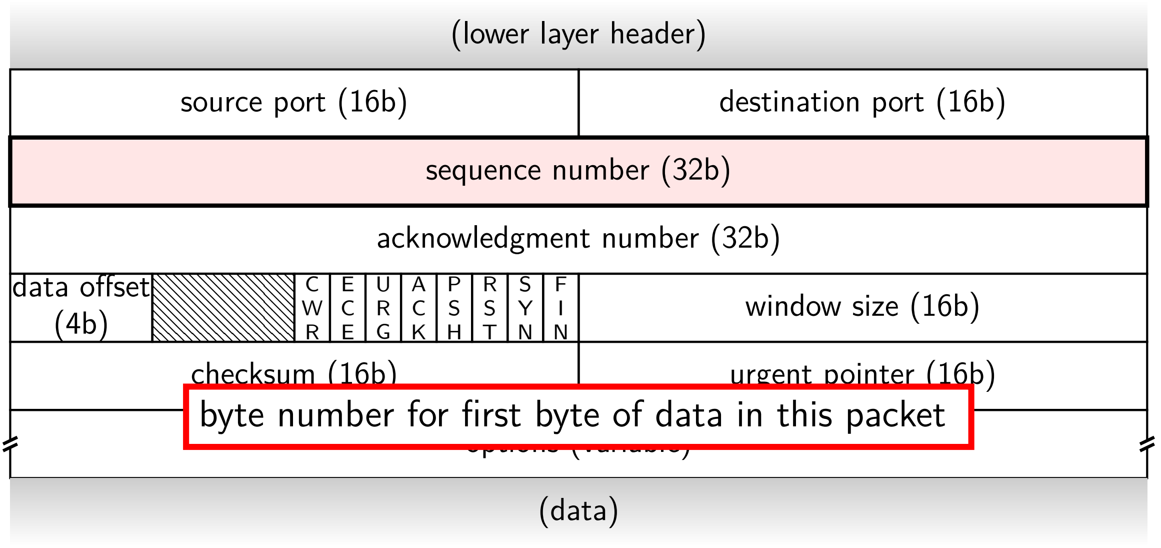
TCP segment format



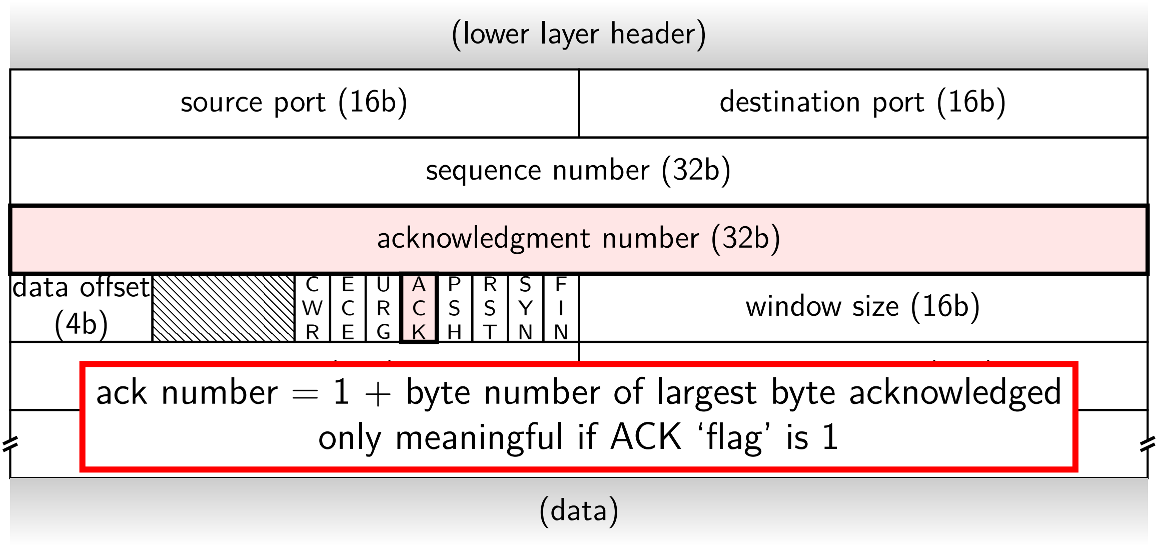
ports identify program/socket on machines
(we'll talk more when we cover sockets)

machines identified in lower layer headers

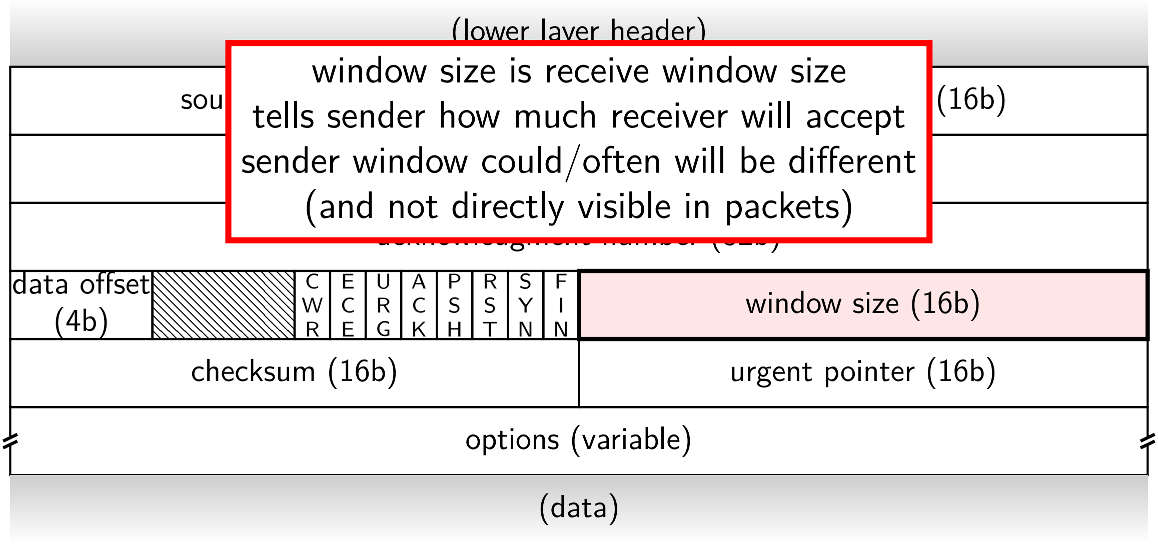
TCP segment format



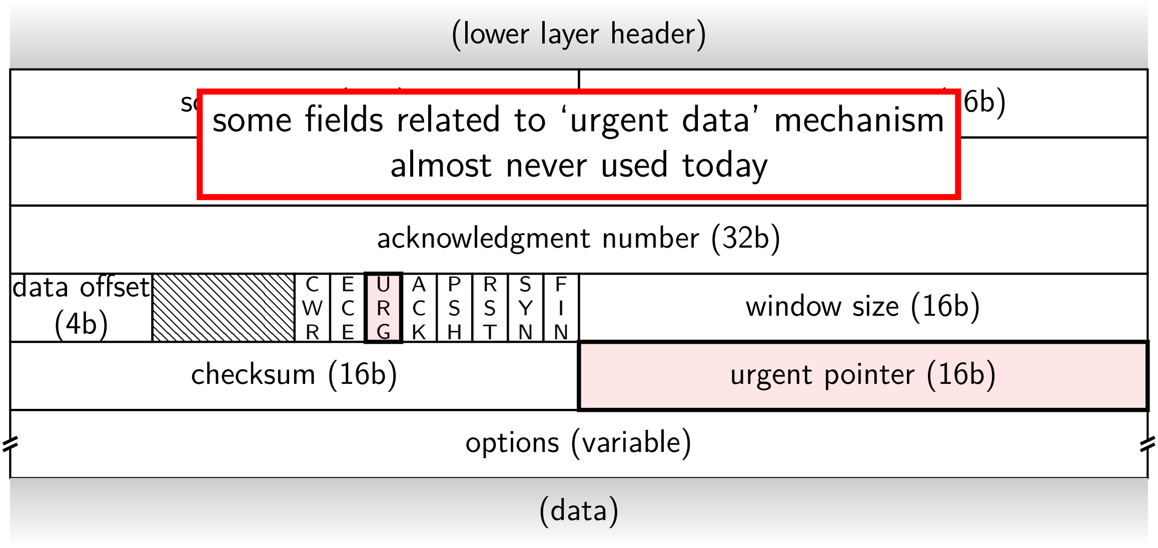
TCP segment format



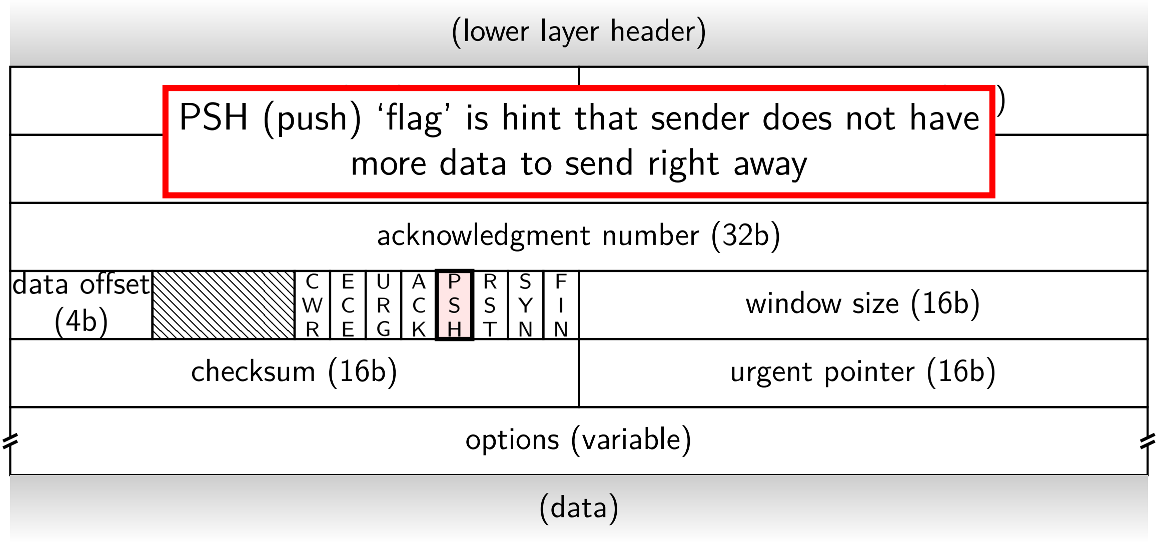
TCP segment format



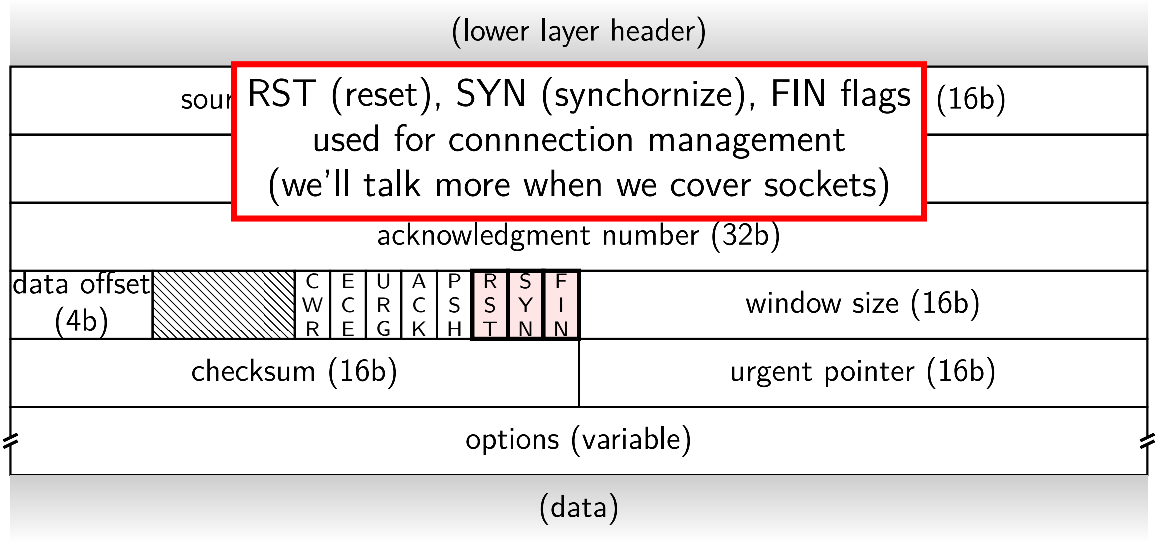
TCP segment format



TCP segment format



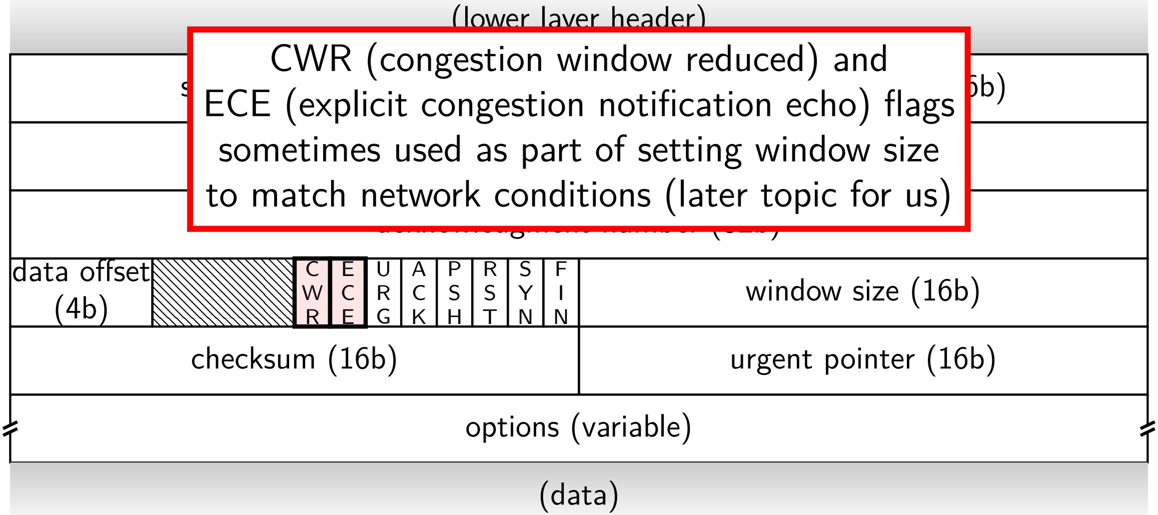
TCP segment format



RST (reset), SYN (synchornize), FIN flags used for connnection management (we'll talk more when we cover sockets)

TCP segment format

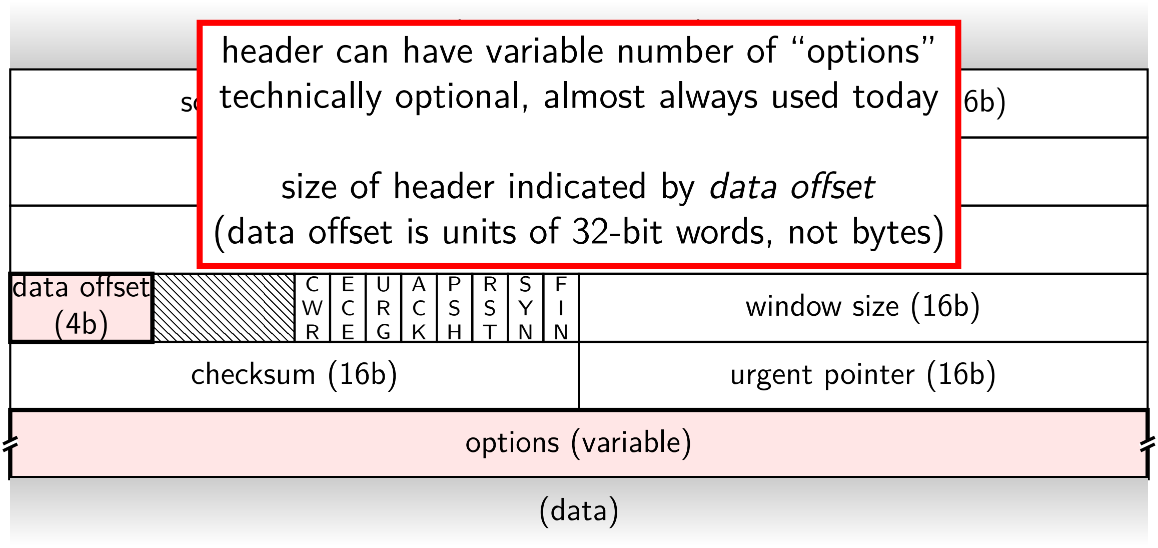
CWR (congestion window reduced) and
ECE (explicit congestion notification echo) flags
sometimes used as part of setting window size
to match network conditions (later topic for us)



TCP segment format

header can have variable number of “options”
technically optional, almost always used today

size of header indicated by *data offset*
(data offset is units of 32-bit words, not bytes)



exercise: maximum throughput

let's say we have a receiver window size of 65535 bytes

and a round-trip time of 100 ms

if we want to avoid sending data the receiver will reject as outside its window, maximum throughput?

A. around 32kbyte/sec
B. around 64kbyte/sec
C. around 128kbyte/sec
D. around 320kbyte/sec
E. around 640kbyte/sec
F. around 1280kbyte/sec
G. something else

selected TCP options

window size scale factor

- allow receiver window sizes greater than 64k
- needed to get reasonable bandwidth on modern networks

timestamps

- allow figuring out round trip time to estimate timeout
- extend 32-bit sequence number, which is too small for multi-gigabit networks

selective acknowledgements

- allow providing information about ‘holes’ in received data
- example: I got bytes 1–5000, 6000–7000, 8000–9000
- without it would only say 5000

selected TCP option formats

kind (8b)	length (8b)	option data

window scale option:

3	3	shift count

timestamps:

8	10	sender TS
echoed TS		

selected TCP option formats

kind (8b)	length (8b)	option data

window scale option:

3	3	shift count

timestamps:

8	10	sender TS
echoed TS		

length field permits skipping unrecognized options

selected TCP option formats

kind (8b)	length (8b)	option data

window scale option:

3	3	shift count

timestamps:

8	10	sender TS
ech		

unique kind codes for each option
list of valid codes maintained by IANA
(Internet Assigned Numbers Authority)

selected TCP option formats

kind (8b)	length (8b)	option data

window scale option:

3	3	shift count

timestamps:

8	TS
ech	

sent only in connection setup
only takes effect if both sides send it
sets amount to left-shift
all window size fields by

selected TCP option formats

kind (8b)	length (8b)	option data

window scale option:

3	3	shift count

timestamps:

8	10	sender TS
echoed TS		

selected TCP option formats

kind (8b)	

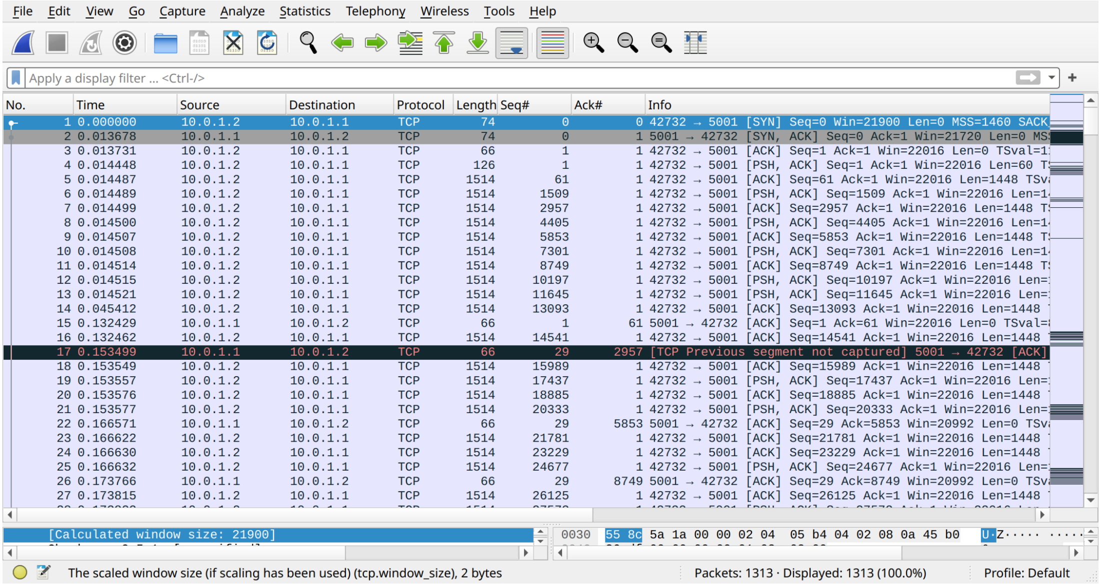
echoed timestamp —
only valid in ACK messages
copy of timestamp sent in message ACK'd
allows computing round-trip-time

3		shift count

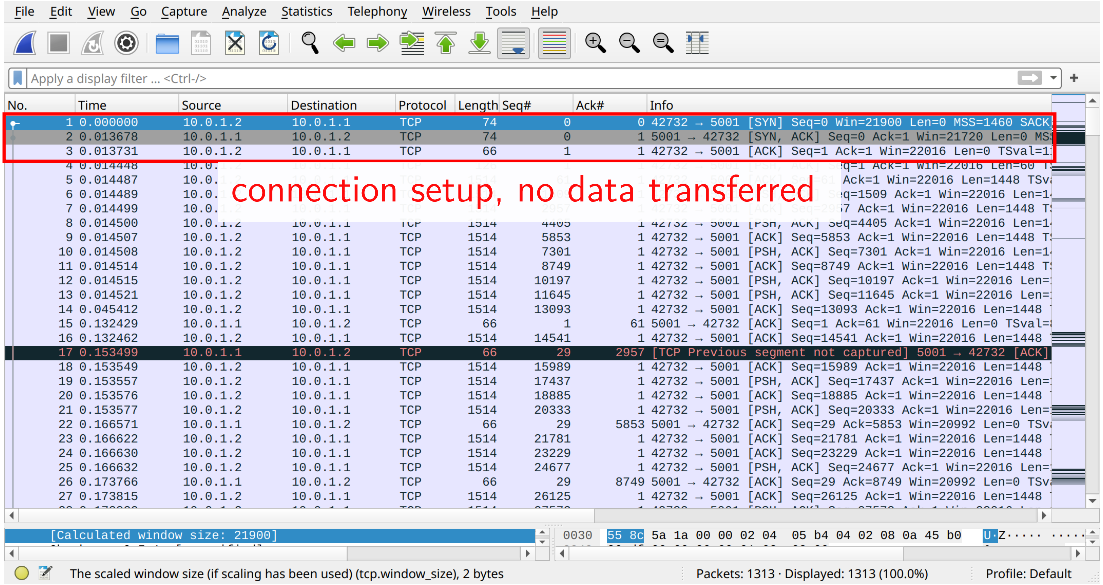
timestamps:

8	10	sender TS
echoed TS		

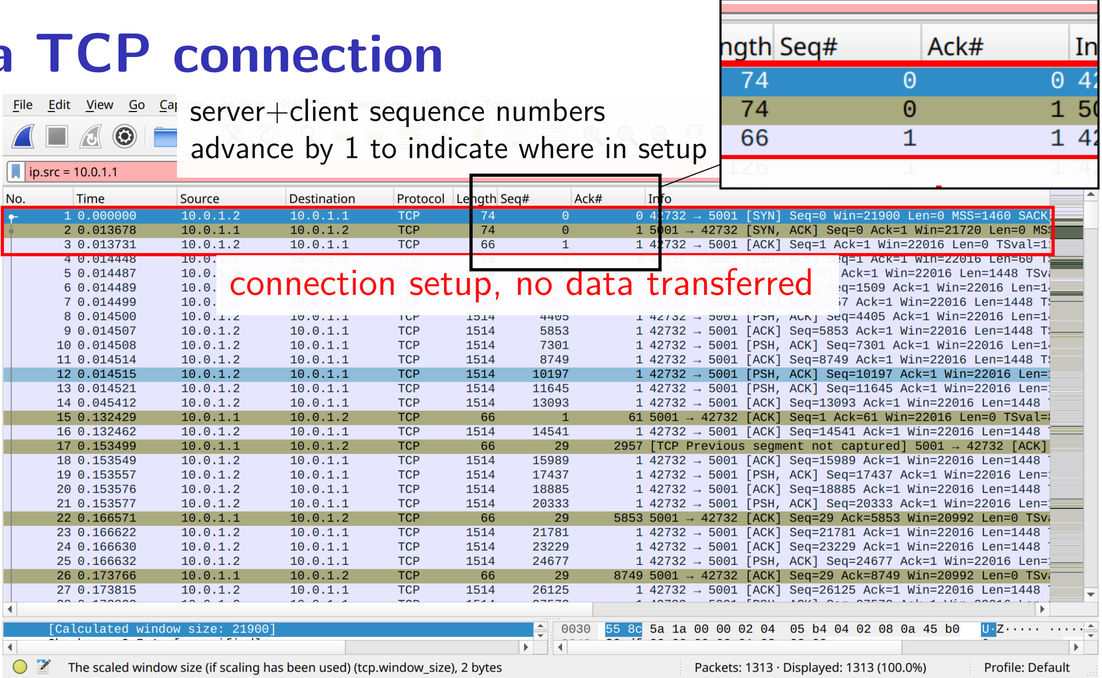
a TCP connection



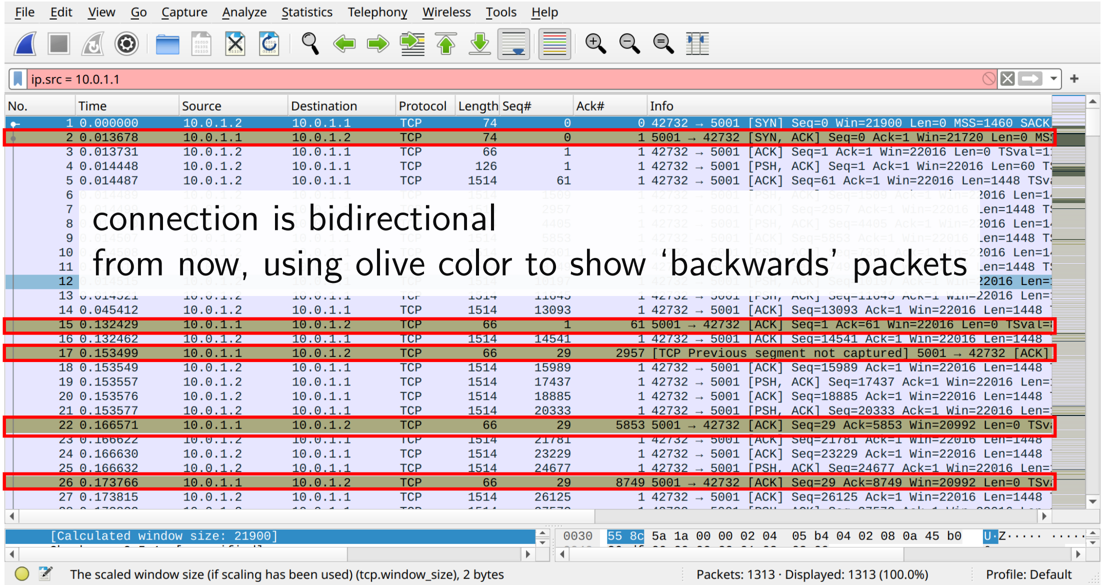
a TCP connection



a TCP connection



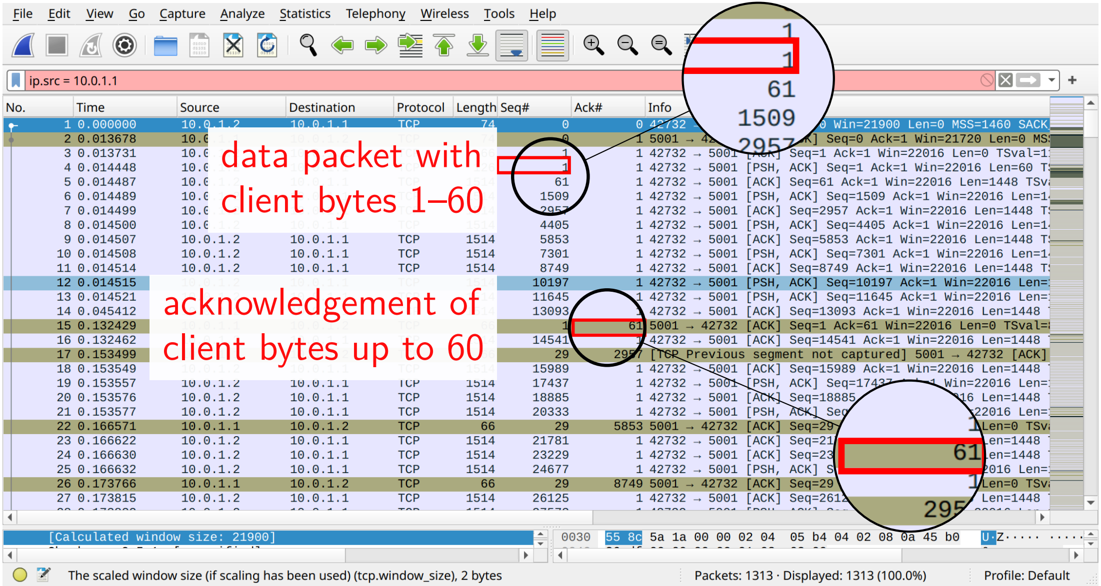
a TCP connection



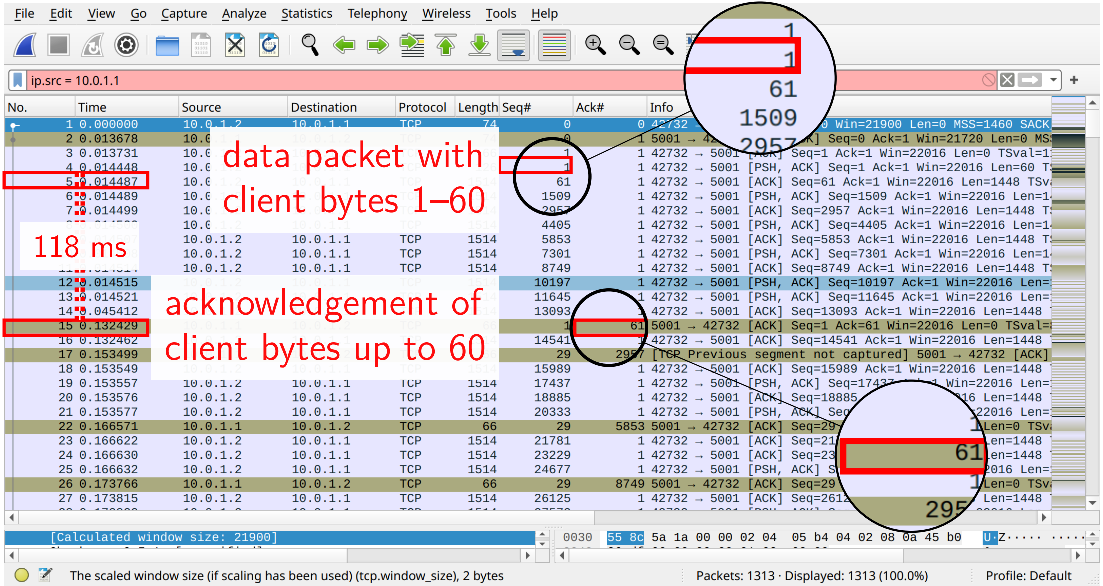
connection is bidirectional

from now, using olive color to show 'backwards' packets

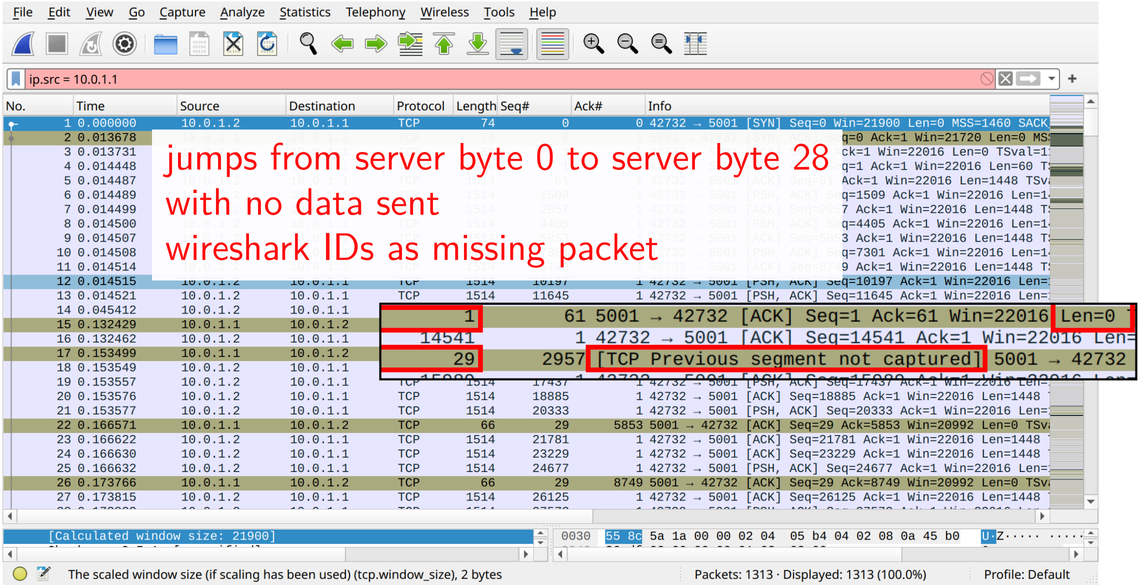
a TCP connection



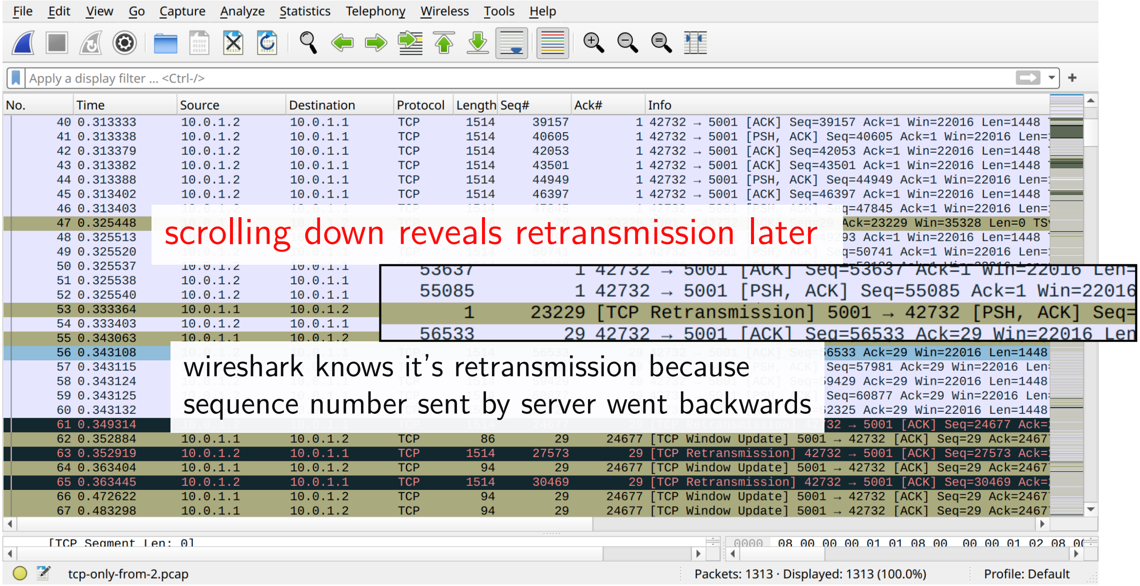
a TCP connection



a TCP connection



a TCP connection



scrolling down reveals retransmission later

wireshark knows it's retransmission because sequence number sent by server went backwards

first data packet

```
Frame 4: 126 bytes on wire (1008 bits), 126 bytes captured (1008 bits)
Ethernet II, Src: 08:00:00:00:01:02 (08:00:00:00:01:02), Dst: 08:00:00:00:01:01 (08:00:00:00
Internet Protocol Version 4, Src: 10.0.1.2, Dst: 10.0.1.1
Transmission Control Protocol, Src Port: 42732, Dst Port: 5001, Seq: 1, Ack: 1, Len: 60
    Source Port: 42732
    Destination Port: 5001
    [Stream index: 0]
    [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 60]
    Sequence Number: 1    (relative sequence number)
    Sequence Number (raw): 3465579712
    [Next Sequence Number: 61    (relative sequence number)]
    Acknowledgment Number: 1    (relative ack number)
    Acknowledgment number (raw): 3771659014
    1000 .... = Header Length: 32 bytes (8)
    Flags: 0x018 (PSH, ACK)
    Window: 43
    [Calculated window size: 22016]
    [Window size scaling factor: 512]
    Checksum: 0x4173 [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
    Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
        TCP Option - No-Operation (NOP)
        TCP Option - No-Operation (NOP)
        TCP Option - Timestamps
```

```
0000  08 0
0010  00 7
0020  01 0
0030  00 2
0040  43 e
0050  00 0
0060  00 0
0070  00 0
```

first data packet

```
▸ Frame 4: 126 bytes on wire (1008 bits), 126 bytes captured (1008 bits)      0000  08 0
▸ Ethernet II, Src: 08:00:00:00:01:02 (08:00:00:00:01:02), Dst: 08:00:00:00:01:01 (08:00:00:00   0010  00 7
▸ Internet Protocol Version 4, Src: 10.0.1.2, Dst: 10.0.1.1                    0020  01 0
▾ Transmission Control Protocol, Src Port: 42732, Dst Port: 5001, Seq: 1, Ack: 1, Len: 60   0030  00 2
    Source Port: 42732                                                         0040  43 e
    Destination Port: 5001                                                     0050  00 0
    [Stream index: 0]                                                          0060  00 0
  ▸ [Conversation complet                                                      0070  00 0
    [TCP Segment Len: 60]
    Sequence Number: 1
    Sequence Number (raw)
    [Next Sequence Number: 61    (relative sequence number)]
    Acknowledgment Number: 1    (relative ack number)
    Acknowledgment number (raw): 3771659014
    1000 .... = Header Length: 32 bytes (8)
  ▸ Flags: 0x018 (PSH, ACK)
    Window: 43
    [Calculated window size: 22016]
    [Window size scaling factor: 512]
    Checksum: 0x4173 [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
  ▾ Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
    ▸ TCP Option - No-Operation (NOP)
    ▸ TCP Option - No-Operation (NOP)
    ▸ TCP Option - Timestamps
```

not actually part of header

computed using length from lower layer

first data packet

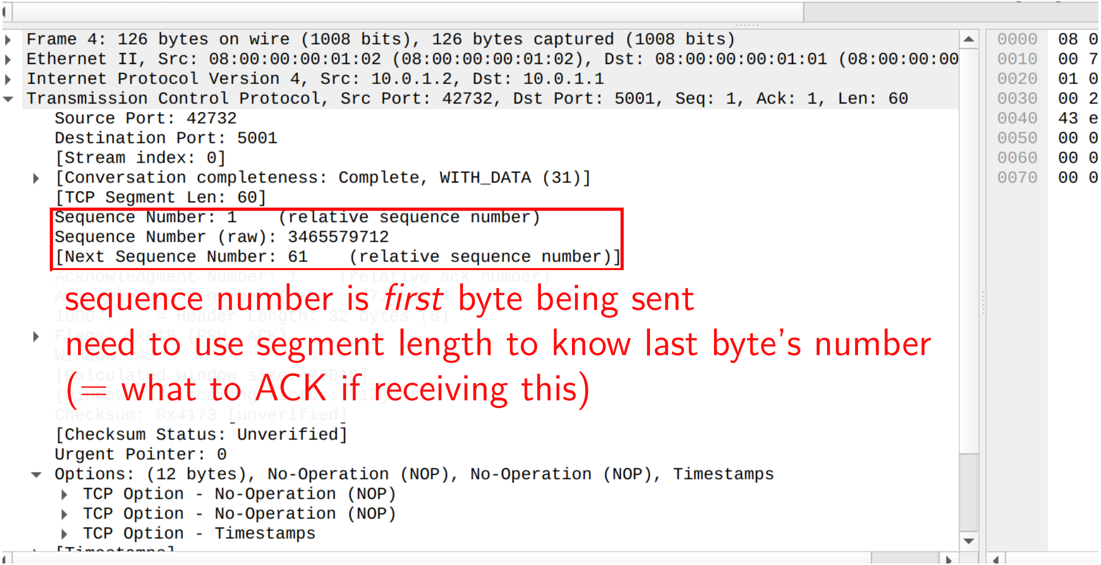
```
▸ Frame 4: 126 bytes on wire (1008 bits), 126 bytes captured (1008 bits)
▸ Ethernet II, Src: 08:00:00:00:01:02 (08:00:00:00:01:02), Dst: 08:00:00:00:01:01 (08:00:00:00
▸ Internet Protocol Version 4, Src: 10.0.1.2, Dst: 10.0.1.1
▾ Transmission Control Protocol, Src Port: 42732, Dst Port: 5001, Seq: 1, Ack: 1, Len: 60
     Source Port: 42732
     Destination Port: 5001
     [Stream index: 0]
   ▸ [Conversation completeness: Complete, WITH_DATA (31)]
     [TCP Segment Len: 60]
     Sequence Number: 1    (relative sequence number)
     Sequence Number (raw): 3465579712
     [Next Sequence Number: 61    (relative sequence number)]
     Acknowledgment number (raw): 3771659014
   ▸ Flags: 0x018 (PSH, ACK)
     Window: 43
     [Calculated window size: 22016]
     [Window size scaling factor: 512]
     Checksum: 0x4173 [unverified]
     [Checksum Status: Unverified]
     Urgent Pointer: 0
   ▾ Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
      ▸ TCP Option - No-Operation (NOP)
      ▸ TCP Option - No-Operation (NOP)
      ▸ TCP Option - Timestamps
```

```
0000  08 0
0010  00 7
0020  01 0
0030  00 2
0040  43 e
0050  00 0
0060  00 0
0070  00 0
```

sequence numbers in header don't start at 0

wireshark converts to 0-based indices

first data packet



sequence number is *first* byte being sent

need to use segment length to know last byte's number

(= what to ACK if receiving this)

first data packet

```
Frame 4: 126 bytes on wire (1008 bits), 126 bytes captured (1008 bits)
Ethernet II, Src: 08:00:00:00:01:02 (08:00:00:00:01:02), Dst: 08:00:00:00:01:01 (08:00:00:00
Internet Protocol Version 4, Src: 10.0.1.2, Dst: 10.0.1.1
Transmission Control Protocol, Src Port: 42732, Dst Port: 5001, Seq: 1, Ack: 1, Len: 60
    Source Port: 42732
    Destination Port: 5001
    [Stream index: 0]
    [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 60]
    Sequence Number: 1    (relative sequence number)
    Sequence Number (raw): 3465579712
    [Next Sequence Number: 61    (relative sequence number)]
    Acknowledgment Number: 1    (relative ack number)
    Acknowledgment number (raw): 3771659014
    Checksum: 0x4173 [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
    Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
        TCP Option - No-Operation (NOP)
        TCP Option - No-Operation (NOP)
        TCP Option - Timestamps
```

0000	08 0
0010	00 7
0020	01 0
0030	00 2
0040	43 e
0050	00 0
0060	00 0
0070	00 0

ack number indicates received start-of-connection stuff and nothing else (in case server sent something)

first data packet

```
Frame 4: 126 bytes on wire (1008 bits), 126 bytes captured (1008 bits)
Ethernet II, Src: 08:00:00:00:01:02 (08:00:00:00:01:02), Dst: 08:00:00:00:01:01 (08:00:00:00
Internet Protocol Version 4, Src: 10.0.1.2, Dst: 10.0.1.1
Transmission Control Protocol, Src Port: 42732, Dst Port: 5001, Seq: 1, Ack: 1, Len: 60
    Source Port: 42732
    Destination Port: 5001
    [Stream index: 0]
    [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 60]
    Sequence Number: 1    (relative sequence number)
    Sequence Number (raw): 3465579712
    [Next Sequence Number: 61    (relative sequence number)]
    Acknowledgment Number: 1    (relative ack number)
    Acknowledgment number (raw): 3771659014
    1000 .... = Header Length: 32 bytes (8)
    Flags: 0x018 (PSH, ACK)
    Window: 43
    Urgent Pointer: 0
    Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
        TCP Option - No-Operation (NOP)
        TCP Option - No-Operation (NOP)
        TCP Option - Timestamps
```

0000	08 0
0010	00 7
0020	01 0
0030	00 2
0040	43 e
0050	00 0
0060	00 0
0070	00 0

PSH = no more data right now

ACK = acknowledgment number is valid

first data packet

```
▸ Frame 4: 126 bytes on wire (1008 bits), 126 bytes captured (1008 bits)
▸ Ethernet II, Src: 08:00:00:00:01:02 (08:00:00:00:01:02), Dst: 08:00:00:00:01:01 (08:00:00:00
▸ Internet Protocol Version 4, Src: 10.0.1.2, Dst: 10.0.1.1
▾ Transmission Control Protocol, Src Port: 42732, Dst Port: 5001, Seq: 1, Ack: 1, Len: 60
     Source Port: 42732
     Destination Port: 5001
     [Stream index: 0]
   ▸ [Conversation completeness: Complete, WITH_DATA (31)]
     [TCP Segment Len: 60]
     Sequence Number: 1    (relative sequence number)
     Sequence Number (raw): 3465579712
     [Next Sequence Number: 61    (relative sequence number)]
     Acknowledgment Number: 1    (relative ack number)
     Acknowledgment number (raw): 3771659014
     1000 .... = Header Length: 32 bytes (8)
   ▸ Flags: 0x018 (PSH, ACK)
     Window: 43
     [Calculated window size: 22016]
     [Window size scaling factor: 512]
     Checksum: 0x4173 [unverified]
     [Checksum Status: Unverified]
     Urgent Pointer: 0
   ▾ Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
      ▸ TCP Option - No-Operation (NOP)
      ▸ TCP Option - No-Operation (NOP)
      ▸ TCP Option - Timestamps
     [Timestamps]
```

```
0000  08 0
0010  00 7
0020  01 0
0030  00 2
0040  43 e
0050  00 0
0060  00 0
0070  00 0
```

window scaling option in use
(scaling factor only sent in connection setup)

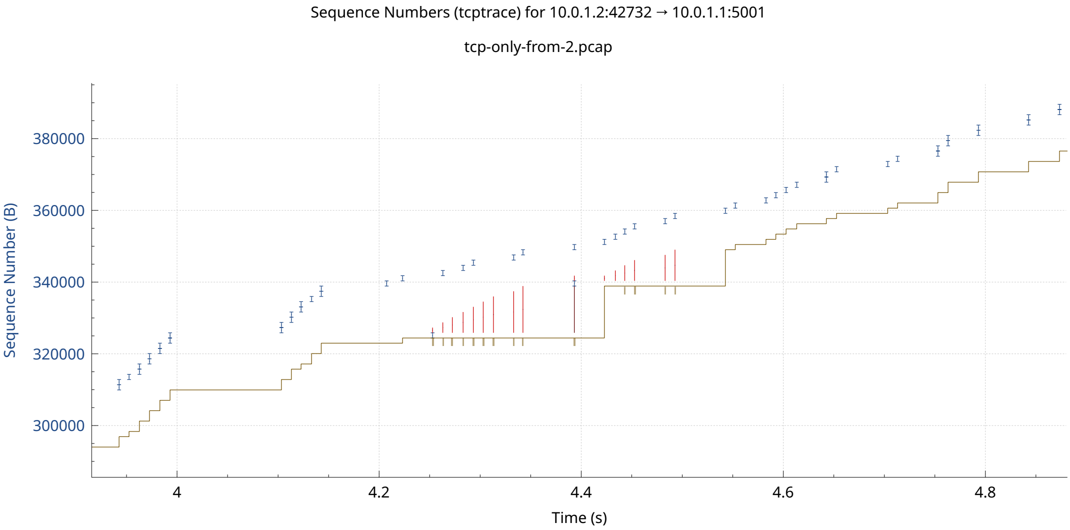
first data packet

```
Frame 4: 126 bytes on wire (1008 bits), 126 bytes captured (1008 bits)
Ethernet II, Src: 08:00:00:00:01:02 (08:00:00:00:01:02), Dst: 08:00:00:00:01:01 (08:00:00:00
Internet Protocol Version 4, Src: 10.0.1.2, Dst: 10.0.1.1
Transmission Control Protocol, Src Port: 42732, Dst Port: 5001, Seq: 1, Ack: 1, Len: 60
    Source Port: 42732
    Destination Port: 5001
    [Stream index: 0]
    [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 60]
    Sequence Number: 1    (relative sequence number)
    Sequence Number (raw): 3465579712
    [Next Sequence Number: 61    (relative sequence number)]
    Acknowledgment Number: 1    (relative ack number)
    Acknowledgment number (raw): 3771659014
    1000 .... = Header Length: 32 bytes (8)
    Flags: 0x018 (PSH, ACK)
    Window: 43
    [Calculated window size: 22016]
    [Window size scaling factor: 512]
    Urgent Pointer: 0
    Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
        TCP Option - No-Operation (NOP)
        TCP Option - No-Operation (NOP)
        TCP Option - Timestamps
```

```
0000  08 0
0010  00 7
0020  01 0
0030  00 2
0040  43 e
0050  00 0
0060  00 0
0070  00 0
```

no-operation options used to make TCP header size multiple of 4

sequence numbers graph



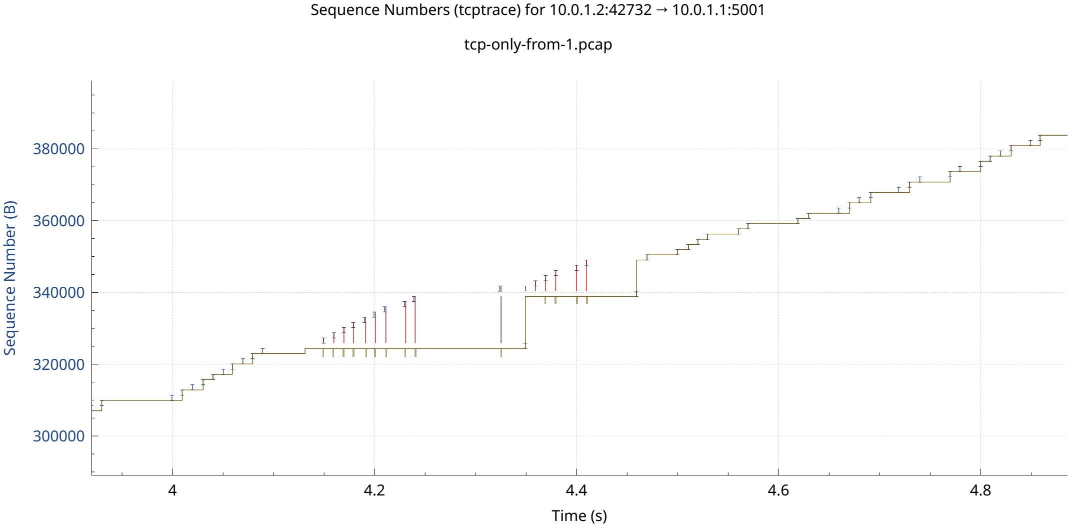
reading thigs graph

bottom line = last ack number

notches on bottom line = duplicate acks

red lines = selective ACK info

diff. timing in opposite direction



backup slides