Lecture 6:

The Transport Layer

Katerina Argyraki, EPFL

Outline (from last lecture)

- Interaction with application layer
  - UDP
  - TCP
- Reliable data delivery
  - Imaginary protocol
  - UDP & TCP at the next lecture

Outline (from last lecture)

- Interaction with application layer
  - UDP
  - TCP
- Reliable data delivery
  - Imaginary protocol
  - UDP & TCP at the next lecture

UDP: reliability elements

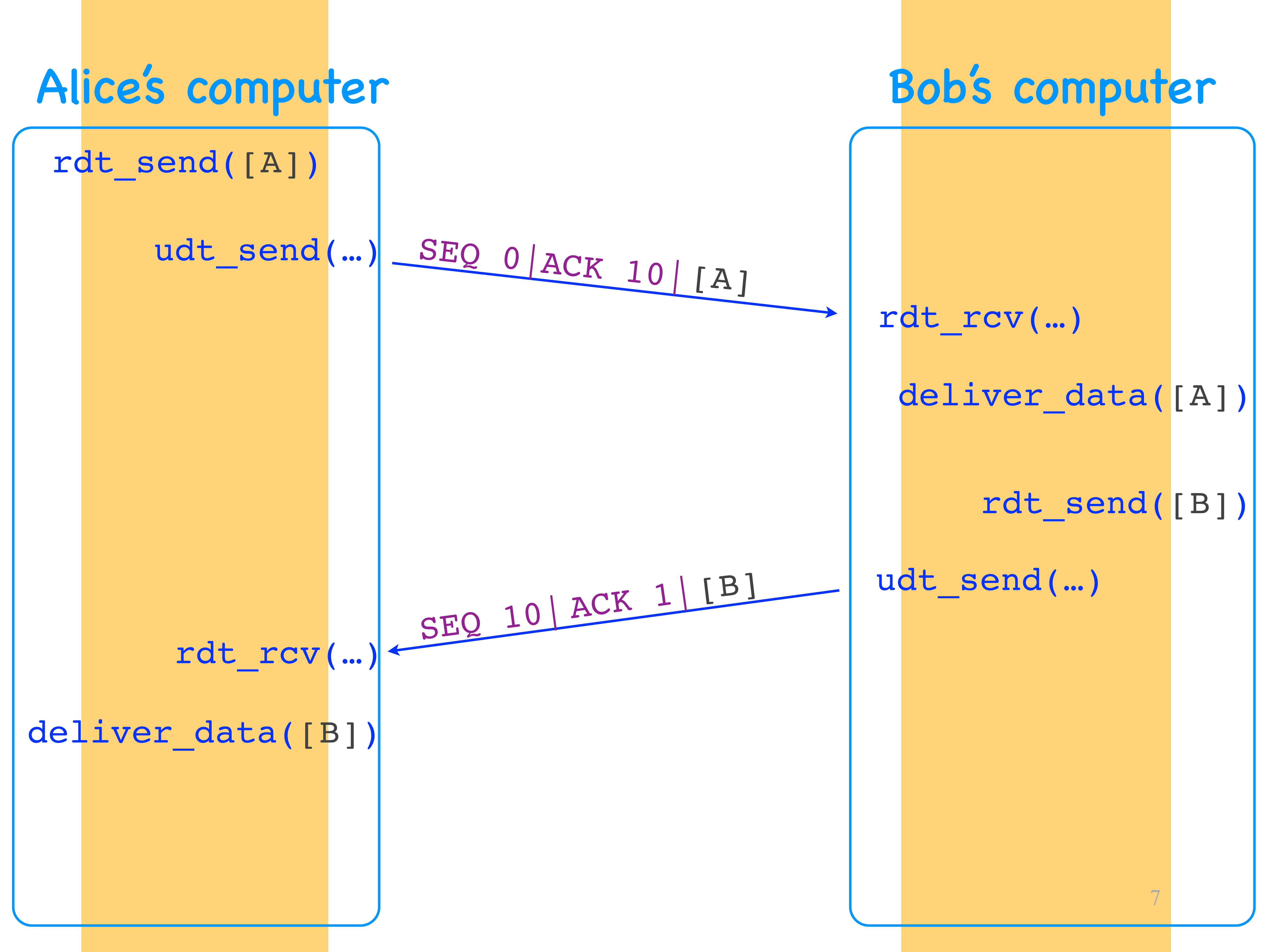
- UDP does not really offer reliable data delivery
- Checksums to detect corruption

TCP: reliability elements

- Checksums to detect corruption
- ACKs to signal successful reception
- SEQs to disambiguate segments
- Timeouts to detect loss
- Retransmissions to recover from corruption+loss

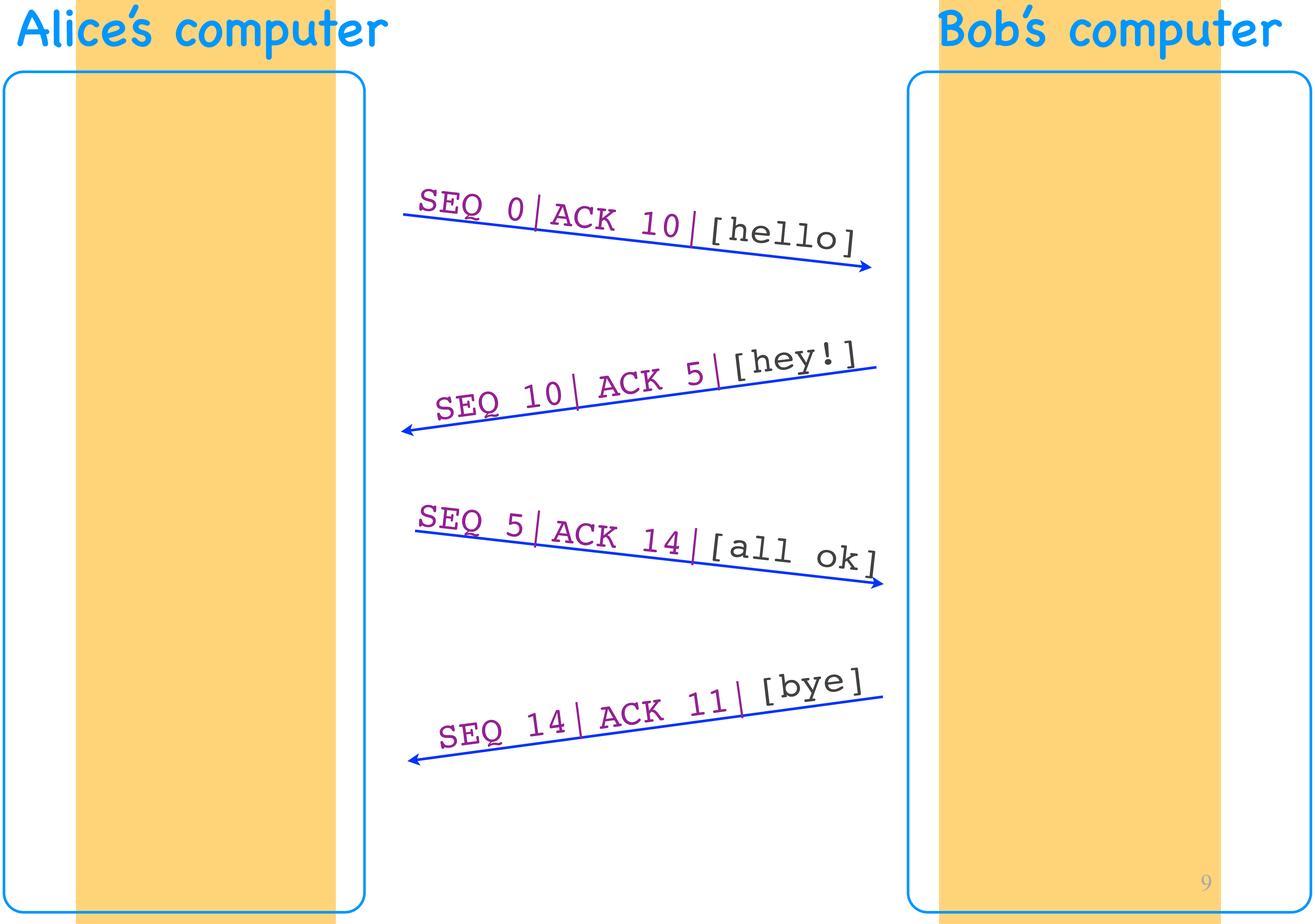
TCP: reliability elements

- Checksums to detect corruption
- ACKs to signal successful reception
- SEQs to disambiguate segments
- Timeouts to detect loss
- Retransmissions to recover from corruption+loss

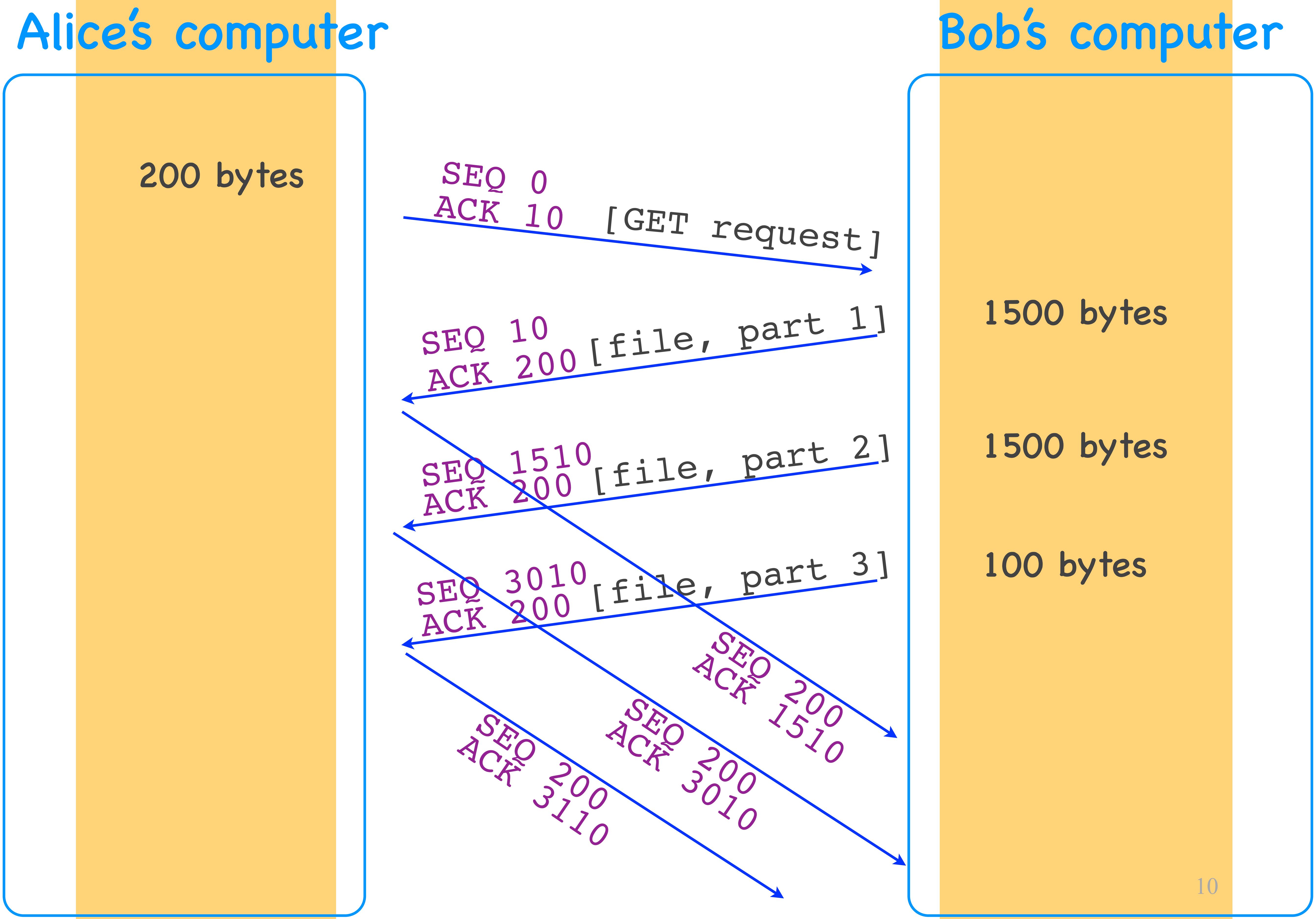

Alice's computer
Bob's computer
rdt_send([A])
udt_send(…)
SEQ 0|ACK 10|[A]
rdt_rcv(…)
deliver_data([A])
rdt_send([B])
udt_send(…)
SEQ 10|ACK 1|[B]
rdt_rcv(…)
deliver_data([B])

SEQs & ACKs

- Data bytes are implicitly numbered
- SEQ: # of the first data byte
- ACK: # of the next data byte that is expected (cumulative)
- Both always present,
  even if it appears unnecessary



Alice's computer
Bob's computer
SEQ 0 | ACK 10 | [hello]
SEQ 10 | ACK 5 | [hey!]
SEQ 5 | ACK 14 | [all ok]
SEQ 14 | ACK 11 | [bye]

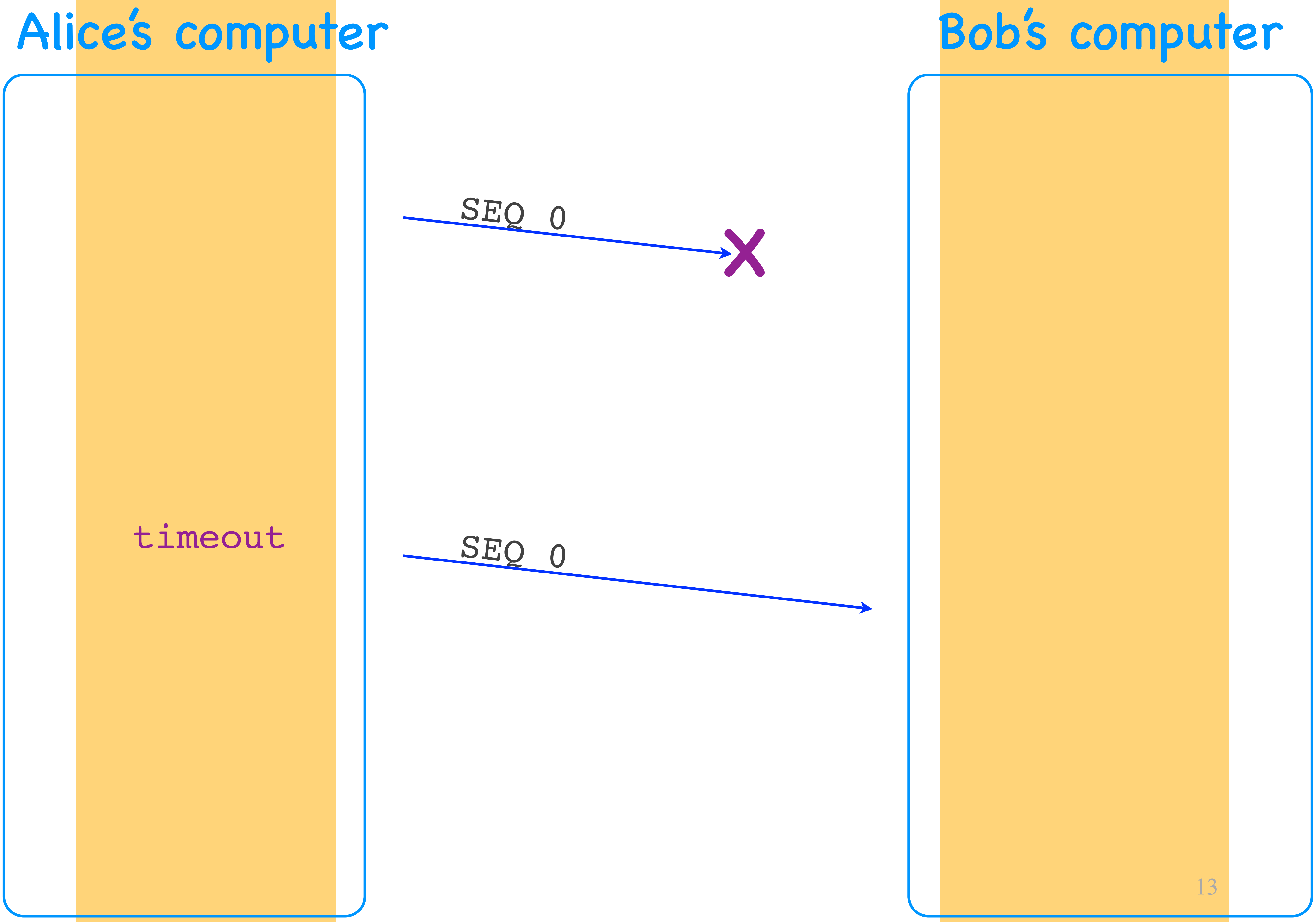

Alice's computer
Bob's computer
200 bytes
SEQ 0
ACK 10
[GET request]
1500 bytes
SEQ 10
ACK 200
[file, part 1]
SEQ 200
ACK 1510
1500 bytes
SEQ 1510
ACK 200
[file, part 2]
SEQ 200
ACK 3010
100 bytes
SEQ 3010
ACK 200
[file, part 3]
SEQ 200
ACK 3110

Simple things to remember

- A TCP connection may carry bidirectional communication
- A segment may or may not carry data (but it always carries a SEQ)
- There exists a maximum segment size (MSS), dictated by network properties

TCP: reliability elements

- Checksums to detect corruption
- ACKs to signal successful reception
- SEQs to disambiguate segments
- Timeouts to detect loss
- Retransmissions to recover from corruption+loss



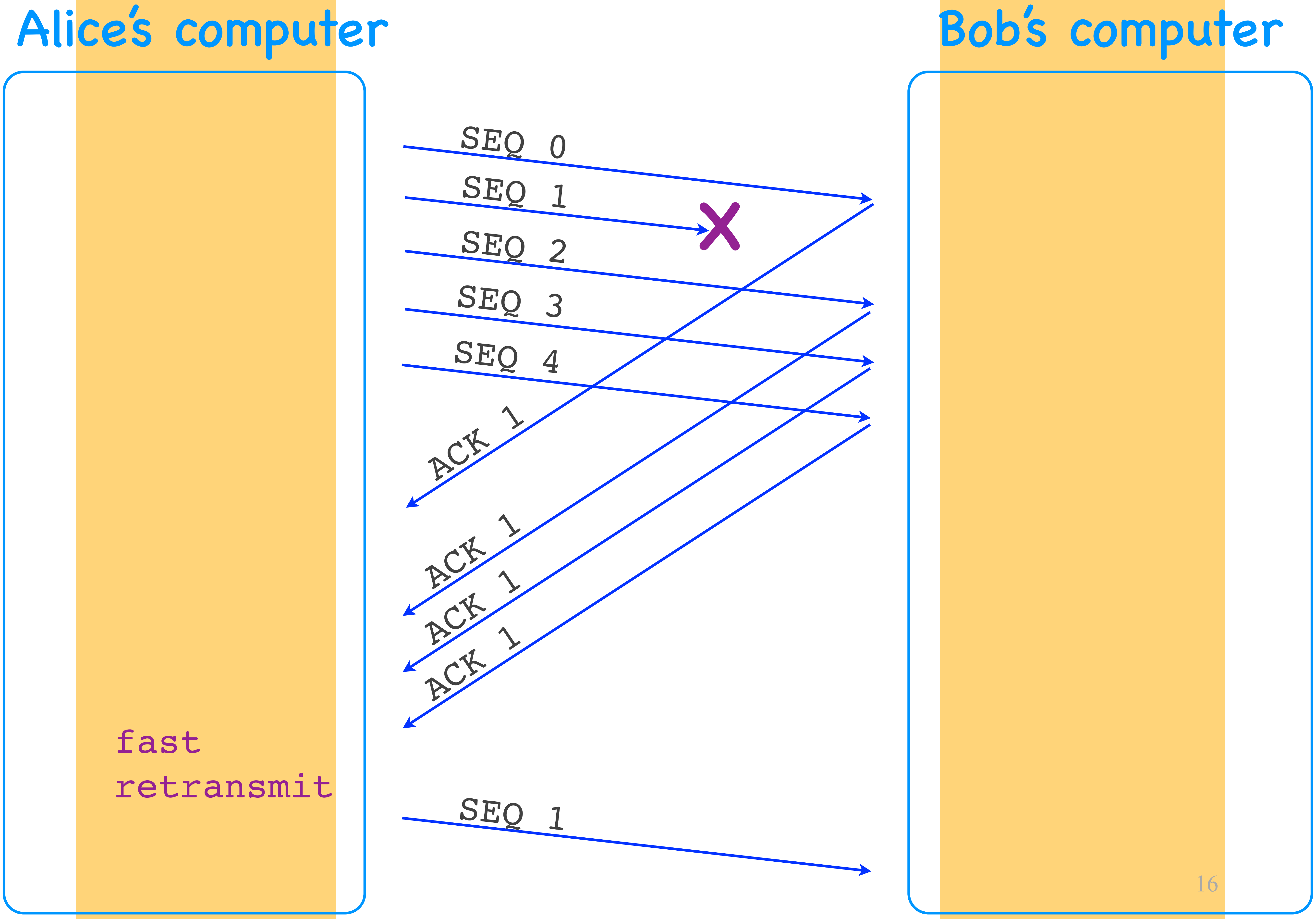
Alice's computer
Bob's computer
SEQ 0
timeout
SEQ 0

How long should the timeout be?

Timeout calculation

- EstimatedRTT =
  0.875 EstimatedRTT + 0.125 SampleRTT
- DevRTT = function(RTT variance)
- Timeout = EstimatedRTT + 4 DevRTT

Empirical, conservative prediction of RTT



Alice's computer
Bob's computer
SEQ 0
SEQ 1
SEQ 2
SEQ 3
SEQ 4
ACK 1
ACK 1
ACK 1
ACK 1
fast
retransmit
SEQ 1

Two retransmission triggers

- Timeout => retransmission of oldest unacknowledged segment

- 3 duplicate ACKs => fast retransmit of oldest unacknowledged segment
  - avoid unnecessary wait for timeout
  - 1 duplicate ACK not enough <= network may have reordered a data segment or duplicated an ACK

TCP: reliability elements

- Checksums to detect corruption
- ACKs to signal successful reception
- SEQs to disambiguate segments
- Timeouts to detect loss
- Retransmissions to recover from corruption+loss

Is TCP Go-back-N or SR?

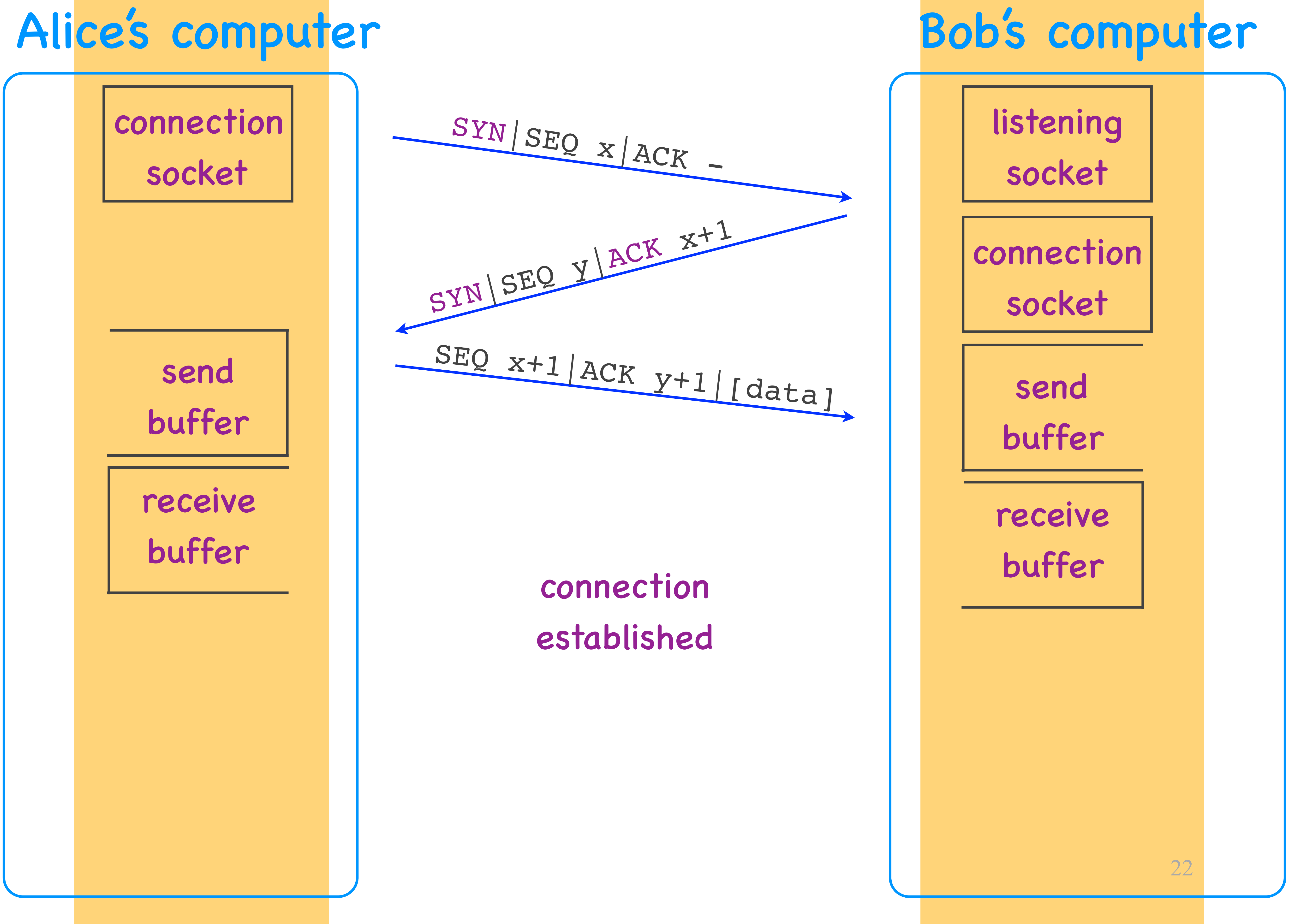
- Go-back-N: cumulative ACKs,
  retransmits multiple segments
- SR: selective ACKs,
  retransmits 1 segment on timeout
- TCP: cumulative ACKs,
  retransmits 1 segment => Go-back-N/SR mix

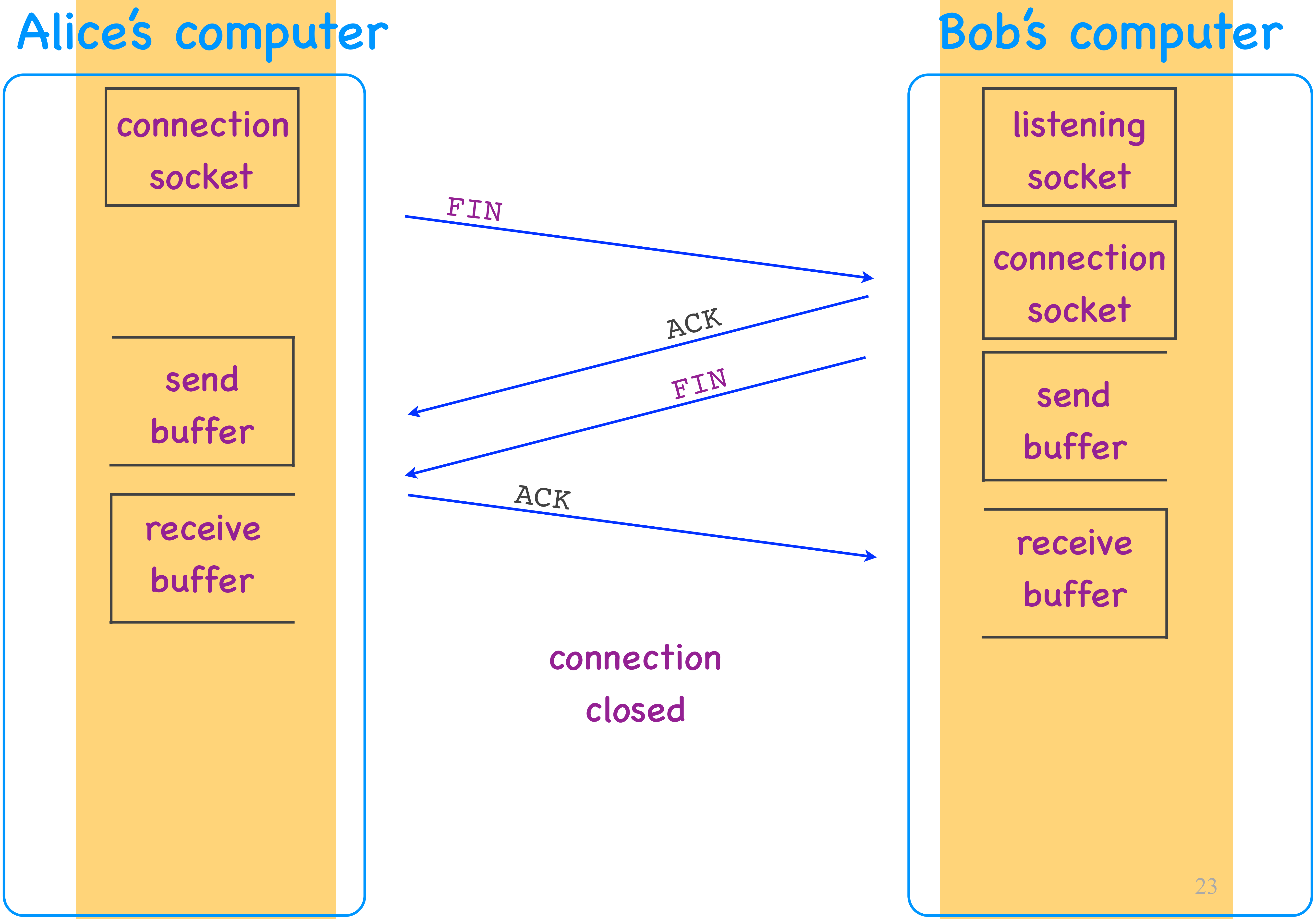
TCP elements

- Connection setup and teardown
- Connection hijacking
- Connection setup (SYN) flooding
- Flow control
- Congestion control

TCP elements

- Connection setup and teardown
- Connection hijacking
- Connection setup (SYN) flooding
- Flow control
- Congestion control


Alice's computer
connection socket
send buffer
receive buffer
Bob's computer
listening socket
connection socket
send buffer
receive buffer
SYN | SEQ x | ACK –
SYN | SEQ y | ACK x+1
SEQ x+1 | ACK y+1 | [data]
connection established



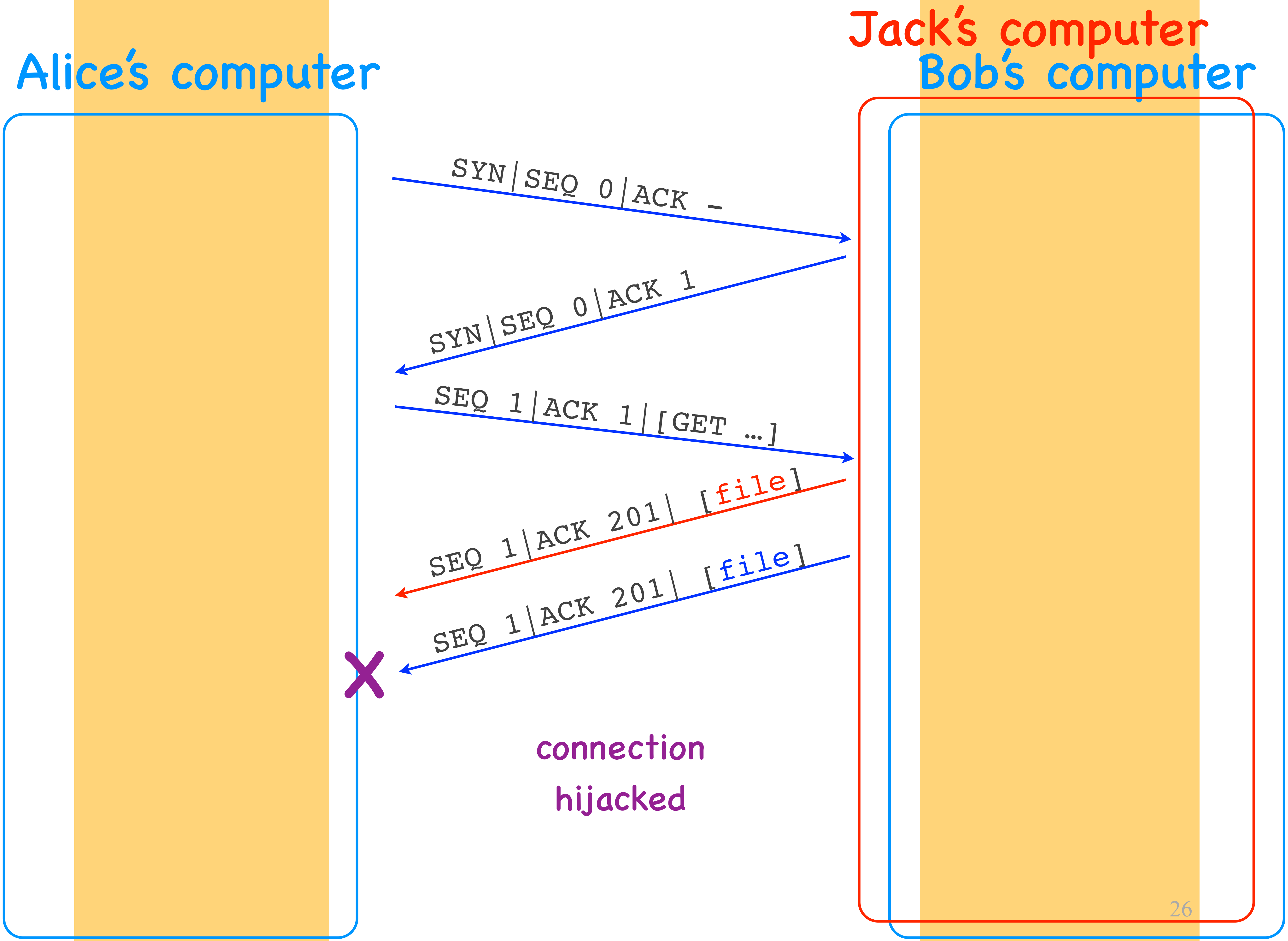
Alice's computer
connection socket
send buffer
receive buffer
Bob's computer
listening socket
connection socket
send buffer
receive buffer
FIN
ACK
FIN
ACK
connection closed

Connection setup

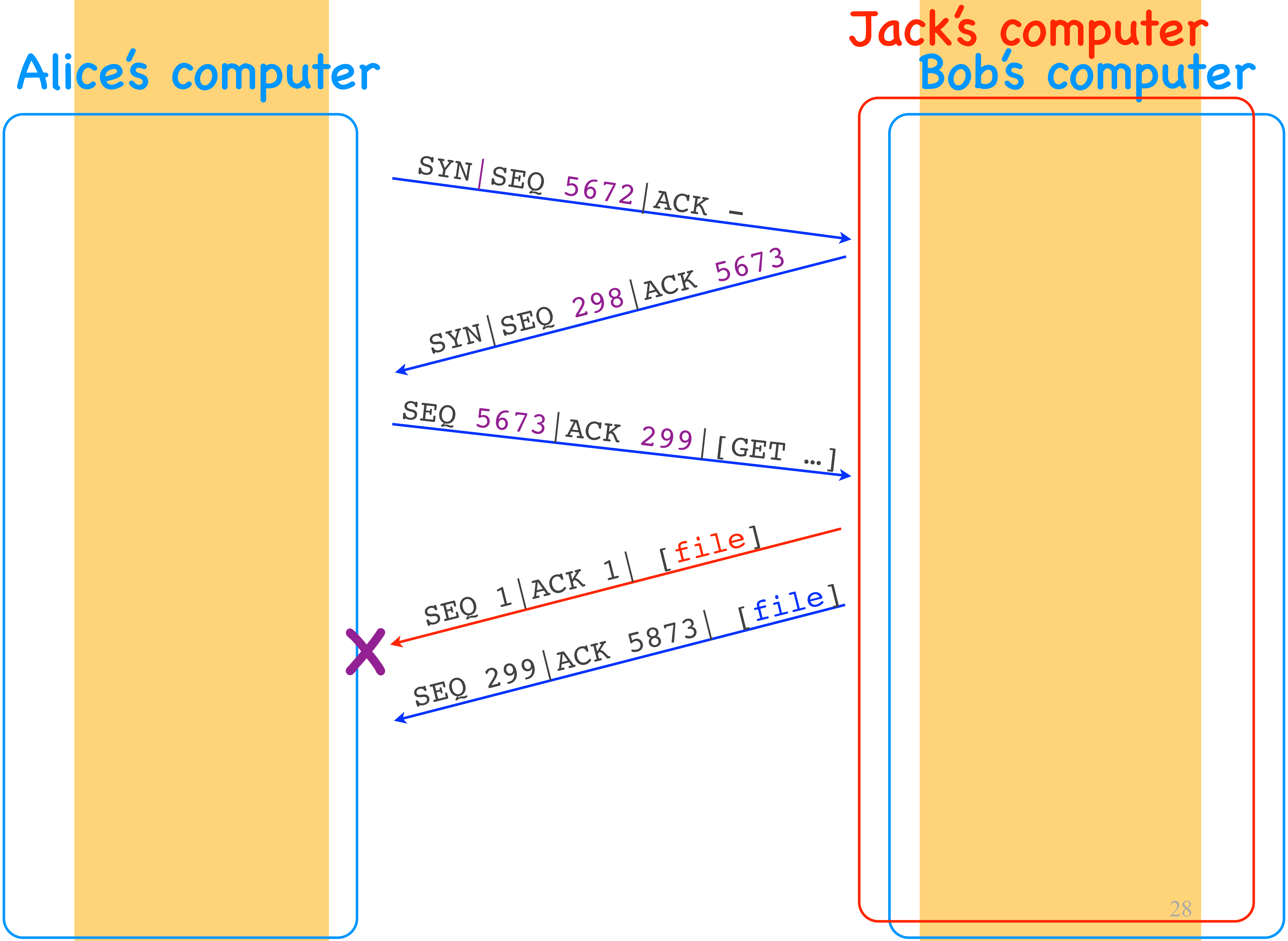
- 3-way handshake
  - "TCP client": end-system initiating the handshake
  - "TCP server": the other end-system
- First 2 segments carry a SYN flag
  - 1-bit field in the TCP header
- "TCP connection" = resources (sockets, buffers…) allocated for communication

TCP elements

- Connection setup and teardown
- Connection hijacking
- Connection setup (SYN) flooding
- Flow control
- Congestion control


Jack's computer
Alice's computer
Bob's computer
SYN|SEQ 0|ACK -
SYN|SEQ 0|ACK 1
SEQ 1|ACK 1|[GET …]
SEQ 1|ACK 201| [file]
SEQ 1|ACK 201| [file]
connection
hijacked

How to prevent connection hijacking?



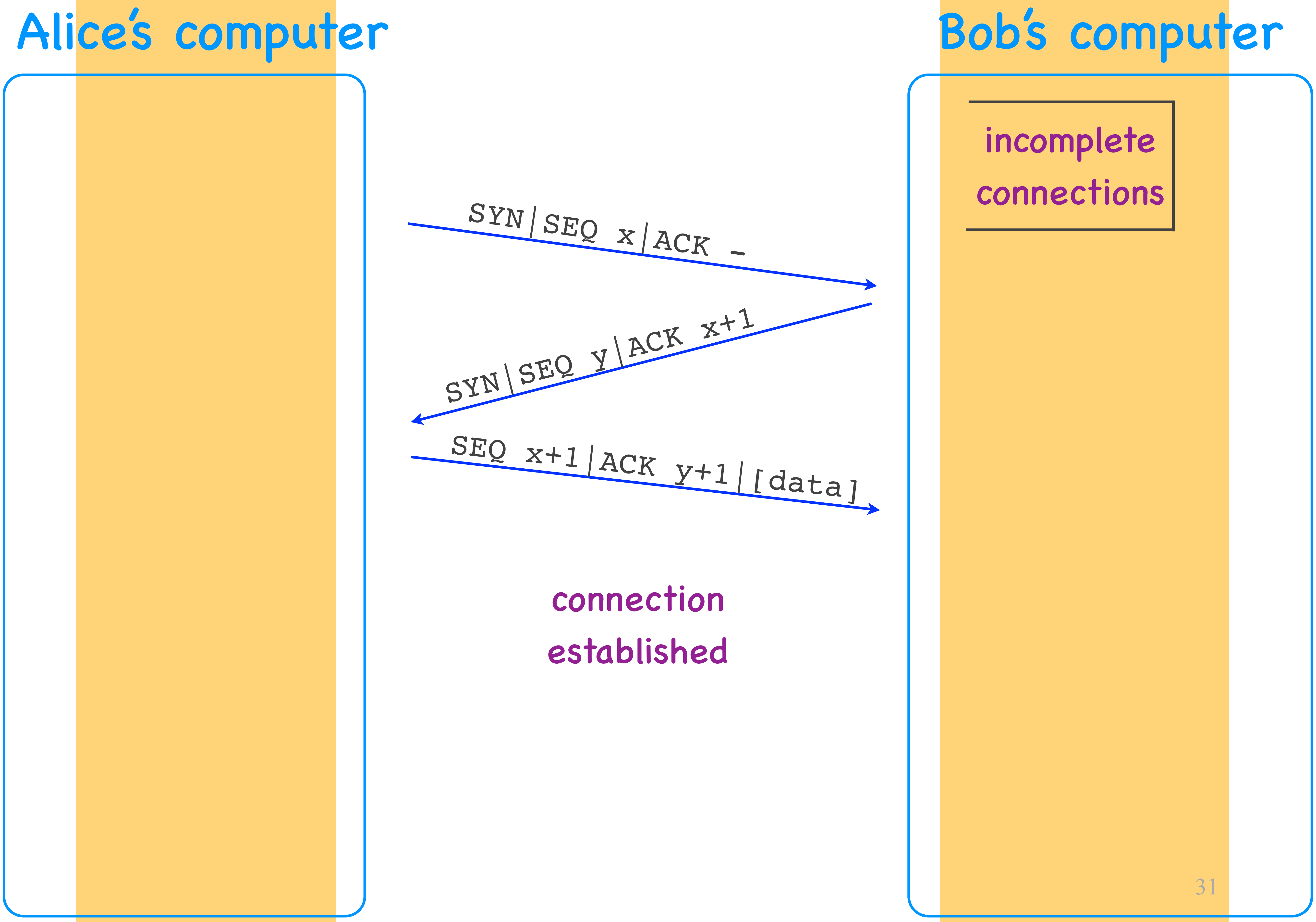
Jack's computer
Alice's computer
Bob's computer
SYN|SEQ 5672|ACK –
SYN|SEQ 298|ACK 5673
SEQ 5673|ACK 299|[GET …]
SEQ 1|ACK 1| [file]
SEQ 299|ACK 5873| [file]

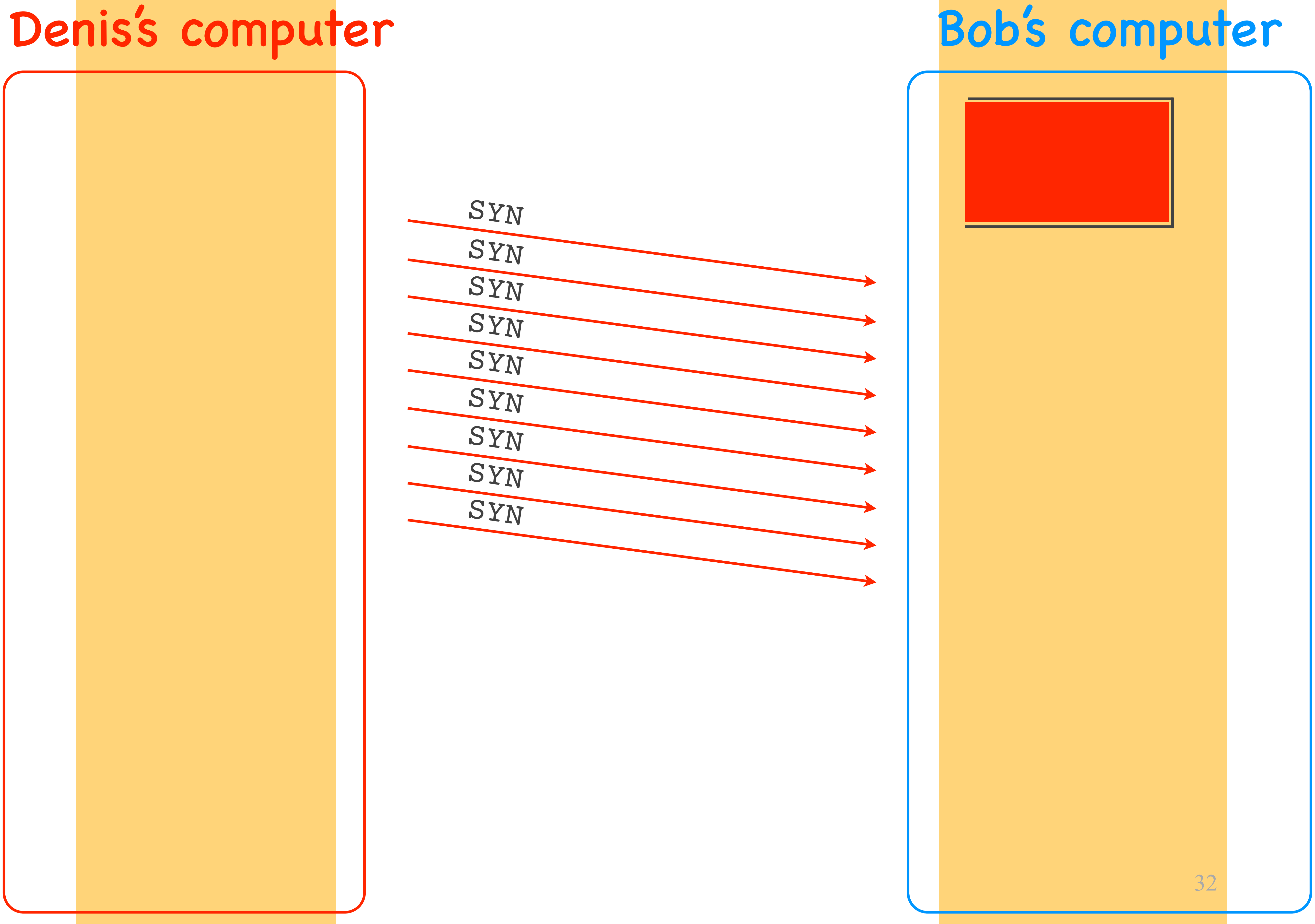
Connection hijacking

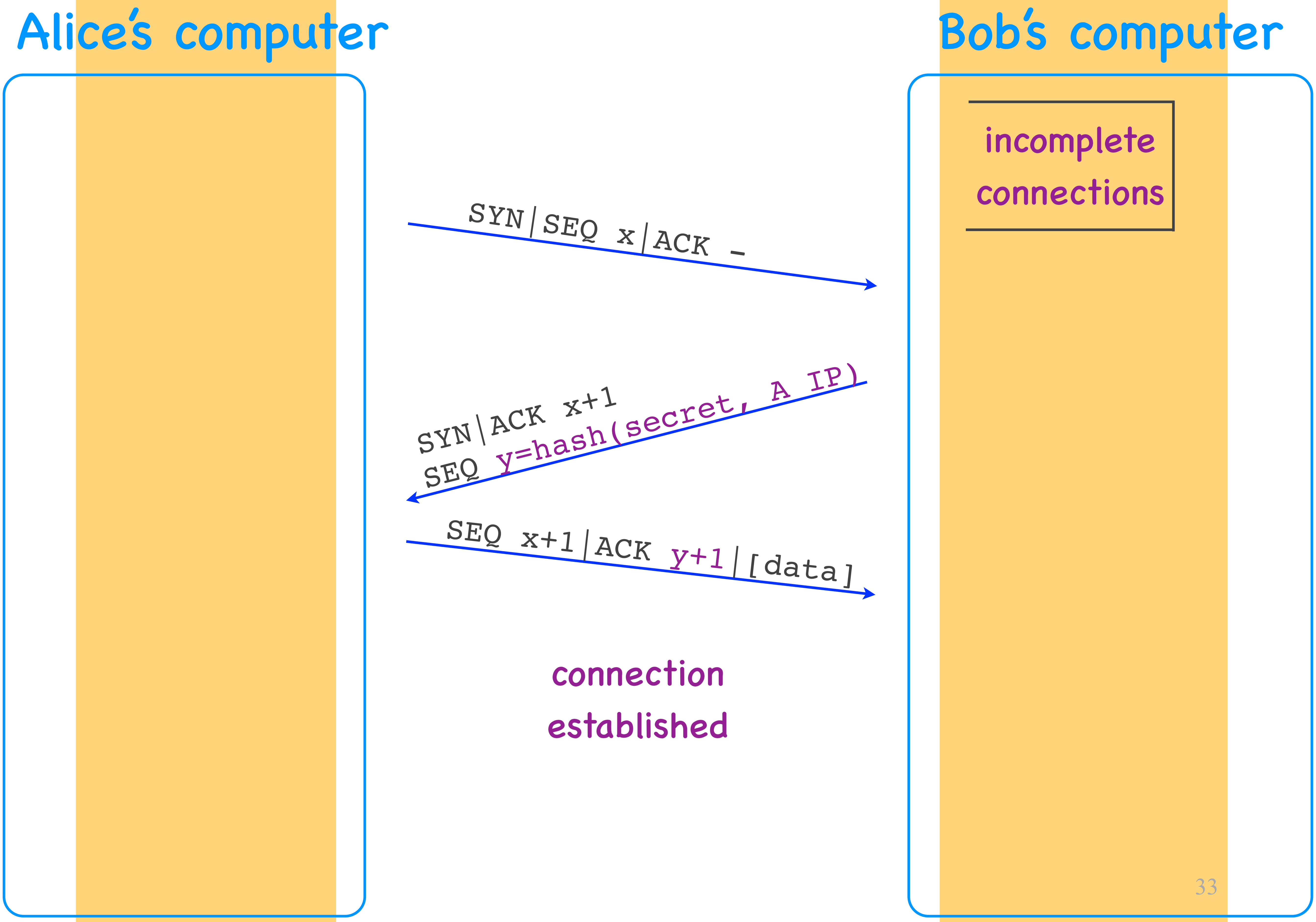
- Attacker impersonates TCP server (or client)
  - sends segment that appears to be coming from the impersonated end-system
- Approach: fake valid segment
  - if the TCP header predictable
- Solution: make TCP header (SEQs) unpredictable

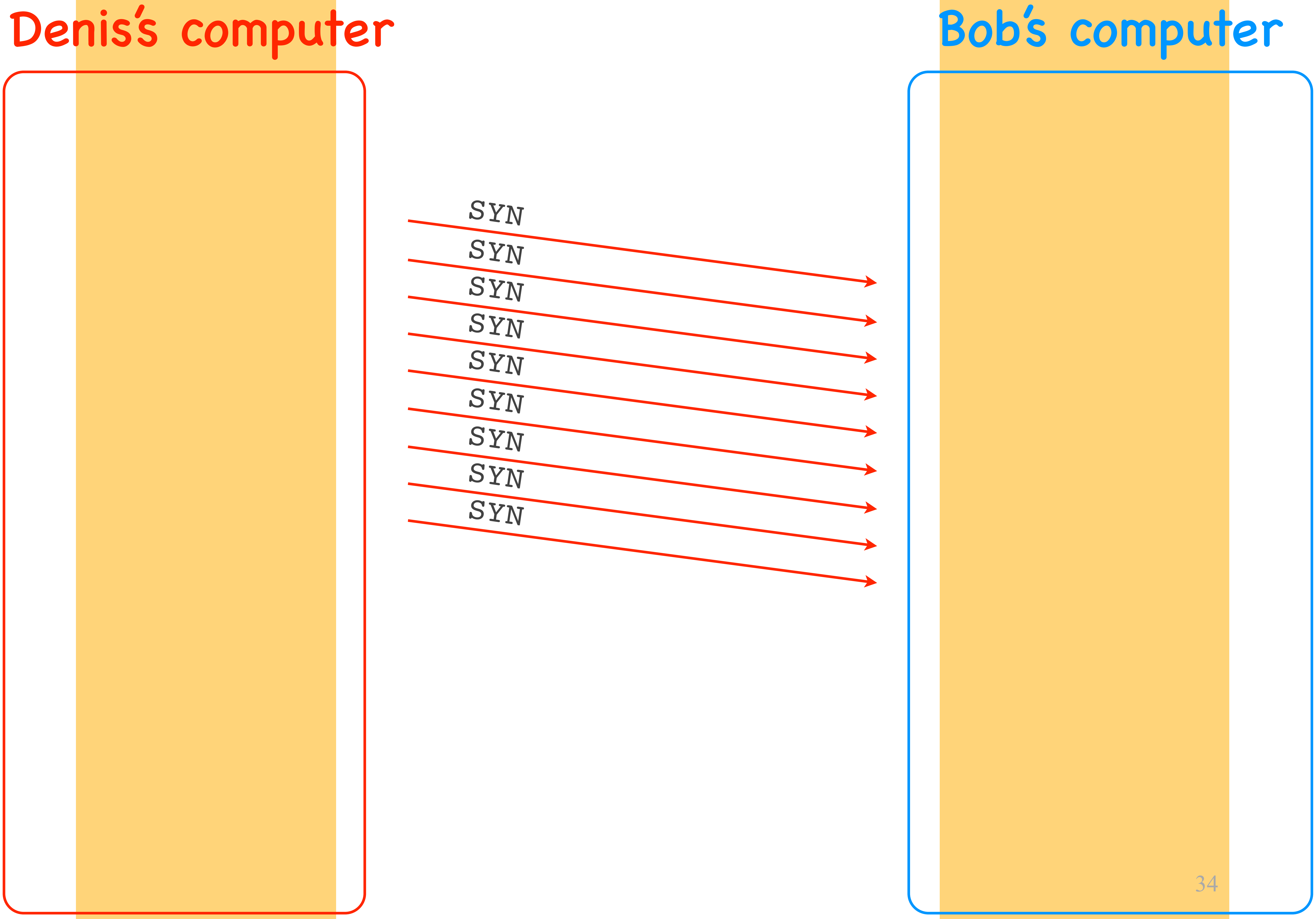
TCP elements

- Connection setup and teardown
- Connection hijacking
- Connection setup (SYN) flooding
- Flow control
- Congestion control


Alice's computer
Bob's computer
incomplete connections
SYN|SEQ x|ACK -
SYN|SEQ y|ACK x+1
SEQ x+1|ACK y+1|[data]
connection established


Denis's computer
Bob's computer
SYN
SYN
SYN
SYN
SYN
SYN
SYN
SYN
SYN


Alice's computer
Bob's computer
incomplete connections
SYN|SEQ x|ACK -
SYN|ACK x+1
SEQ y=hash(secret, A IP)
SEQ x+1|ACK y+1|[data]
connection established

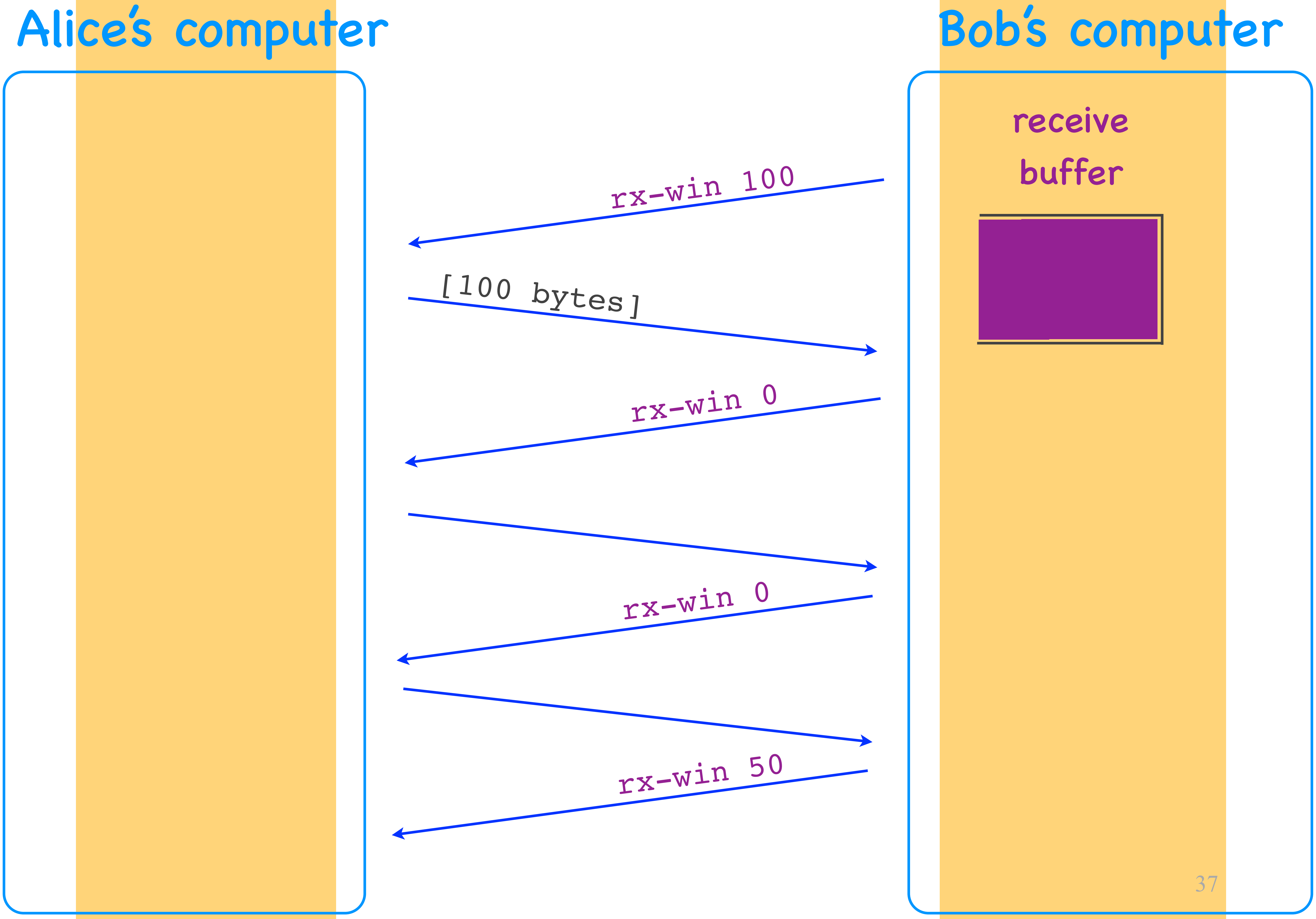

Denis's computer
Bob's computer
SYN
SYN
SYN
SYN
SYN
SYN
SYN
SYN
SYN

SYN flooding

- Attacker exhausts buffer for incomplete connections
  - sends lots of connection setup requests
- Problem: one small resource affects all TCP communication
- Solution: remove the resource
  - pass the state to the TCP client

TCP elements

- Connection setup and teardown
- Connection hijacking
- Connection setup (SYN) flooding
- Flow control
- Congestion control


Alice's computer
Bob's computer
receive
buffer
rx-win 100
[100 bytes]
rx-win 0
rx-win 0
rx-win 50

Flow control

- Goal: not overwhelm the receiver
  - not send at a rate that the receiver cannot handle

- How: "receiver window"
  - spare room in receiver's rx buffer
  - receiver communicates it to sender as TCP header field

TCP elements

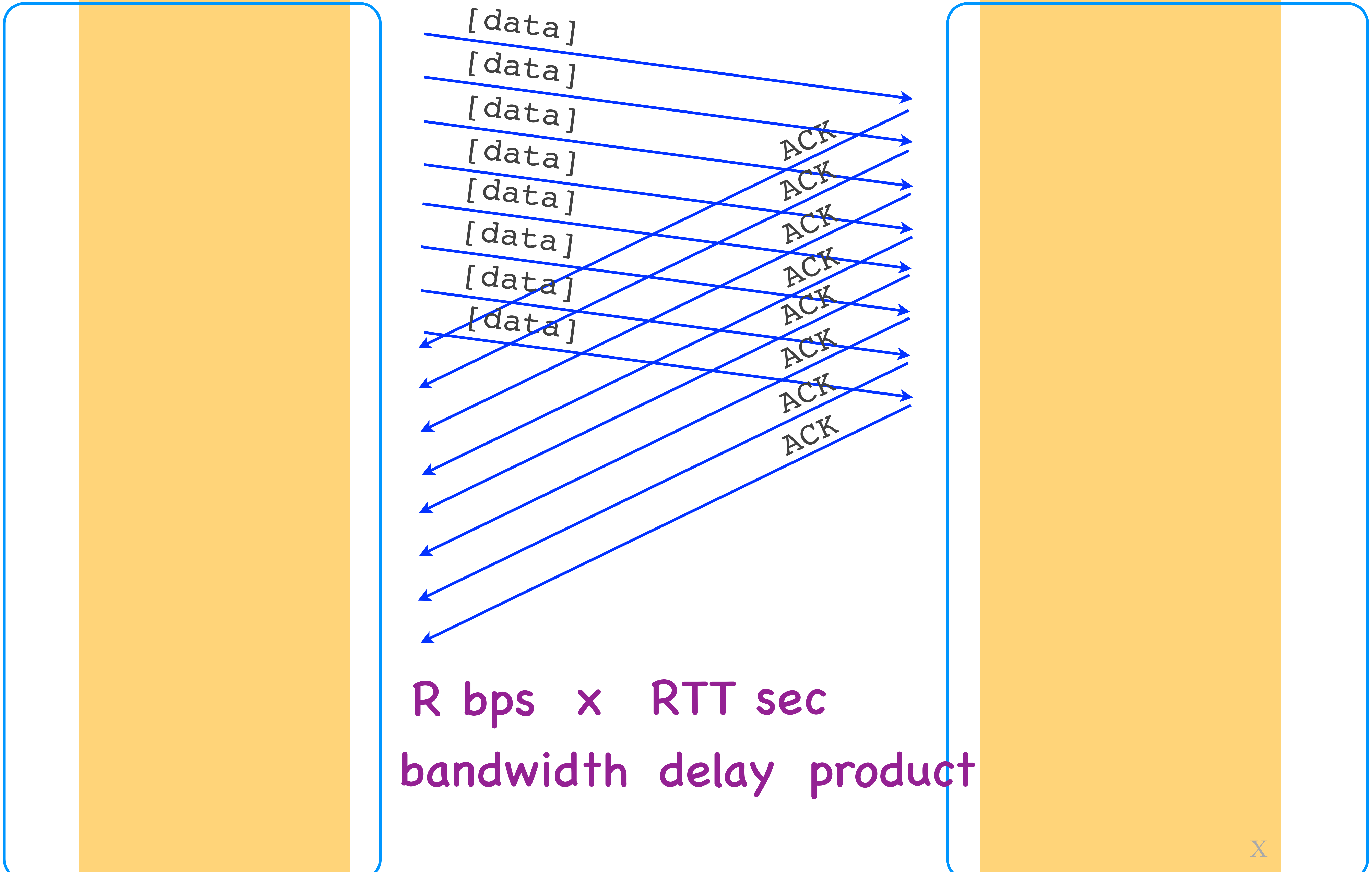
- Connection setup and teardown
- Connection hijacking
- Connection setup (SYN) flooding
- Flow control
- Congestion control

Congestion control

- Goal: not overwhelm the network
  - not send at a rate that the would create network congestion
- How: "congestion window"
  - number of unacknowledged bytes that the sender can transmit without creating congestion
  - sender estimates it on its own

Alice's computer

Bob's computer



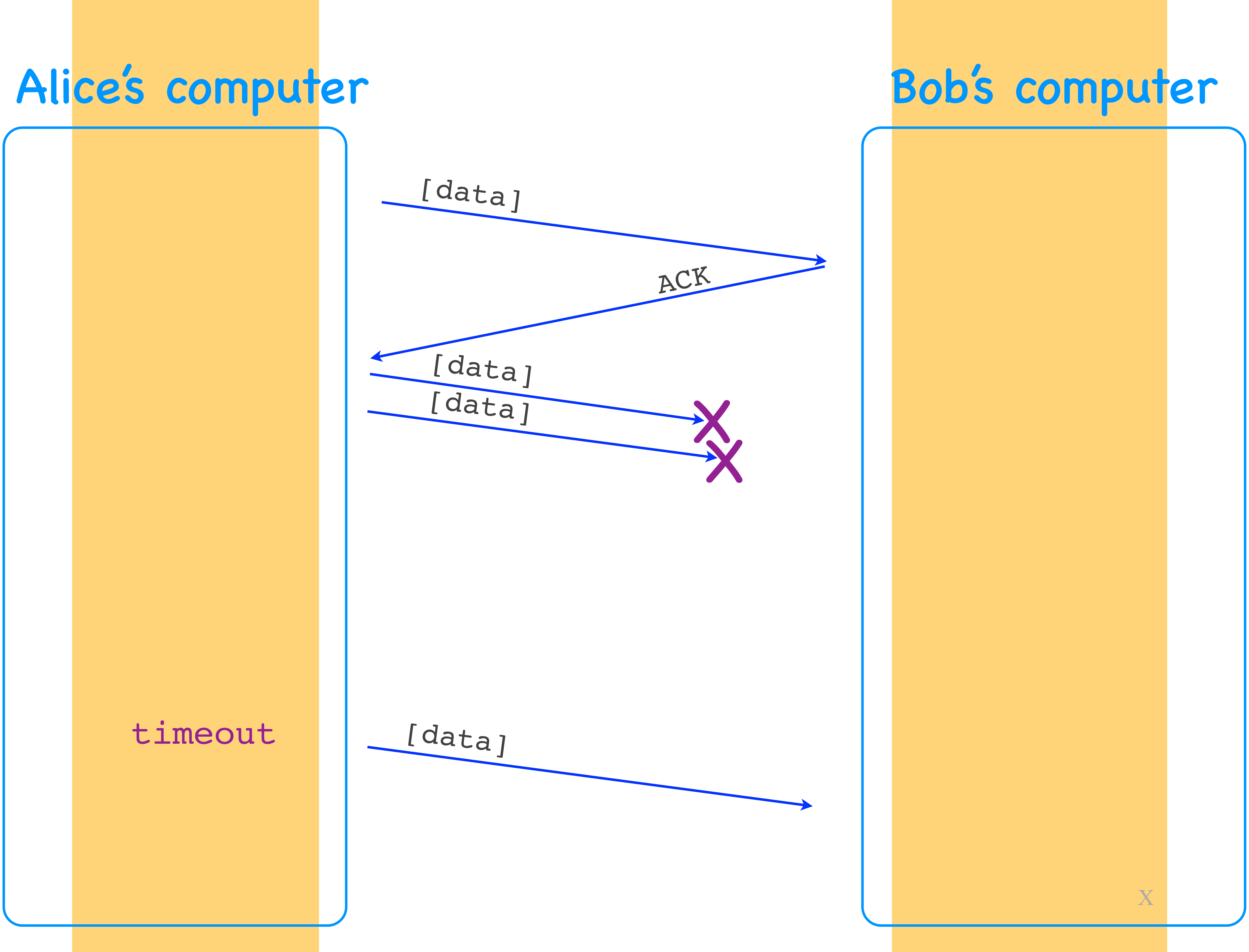
R bps x RTT sec

bandwidth delay product

X

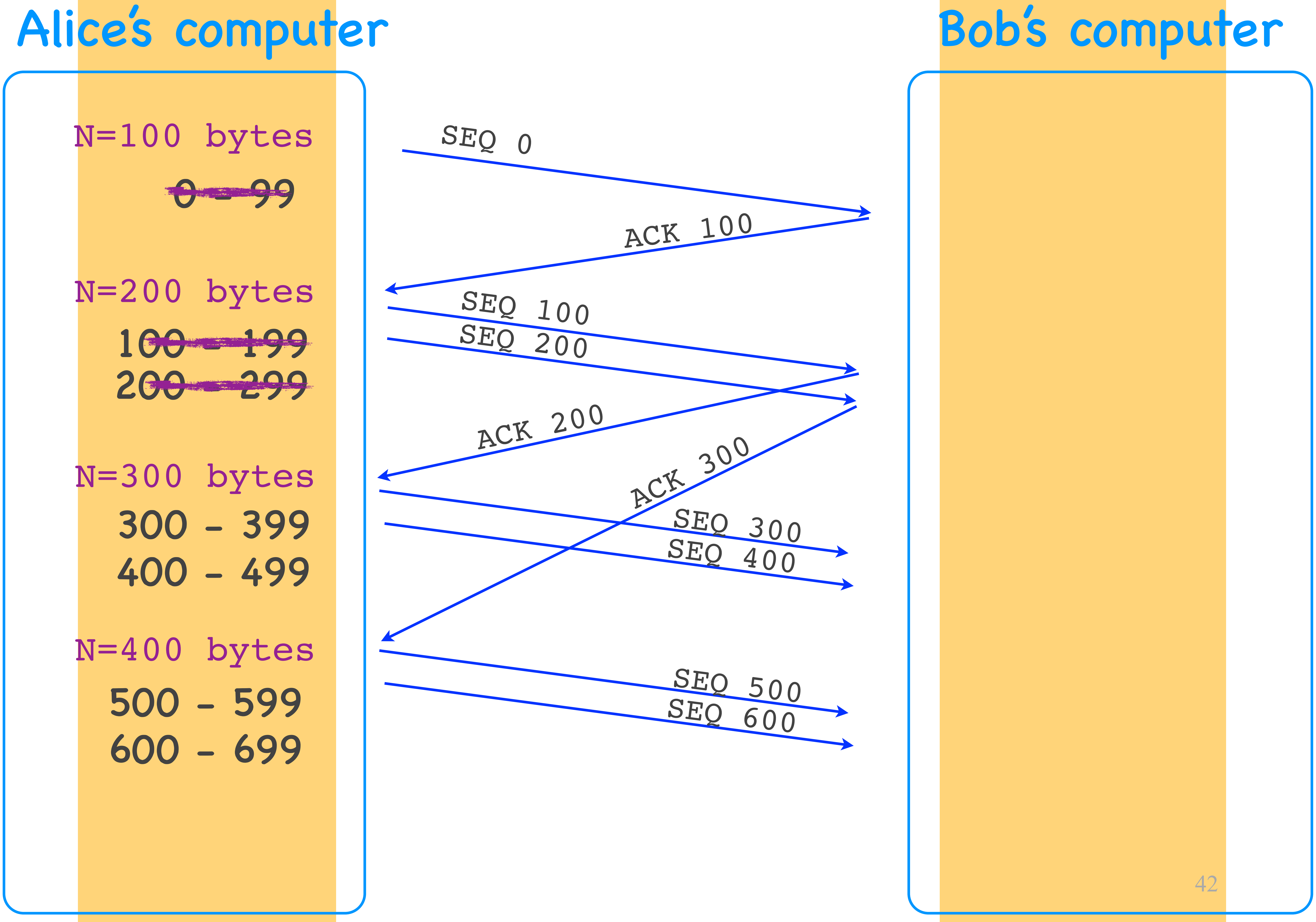
Bandwidth-delay product

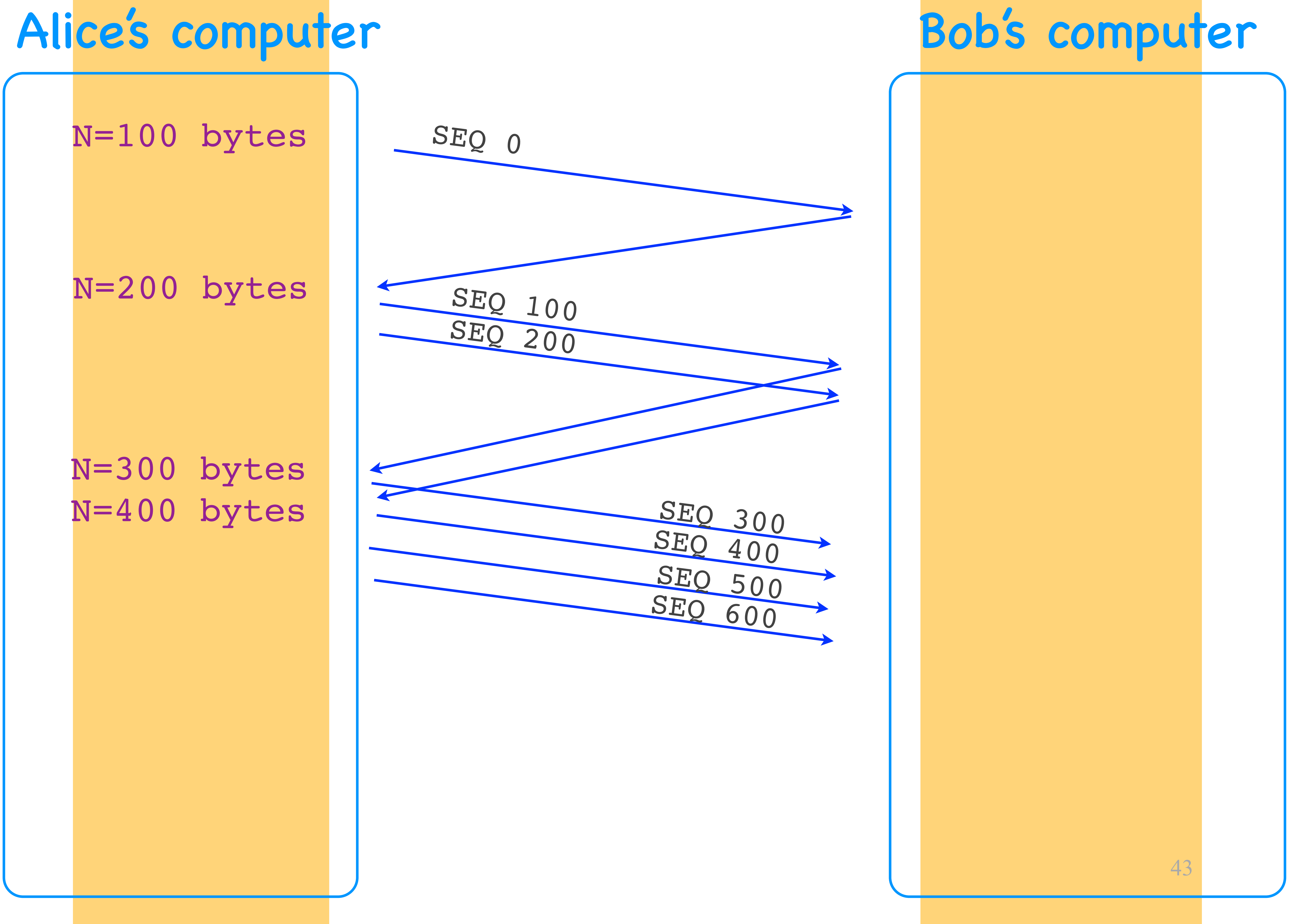
- Max amount of traffic that the sender can transmit until it gets the first ACK
- = the maximum congestion window size that makes sense


Alice's computer
Bob's computer
[data]
ACK
[data]
[data]
timeout
[data]
X

Self-clocking

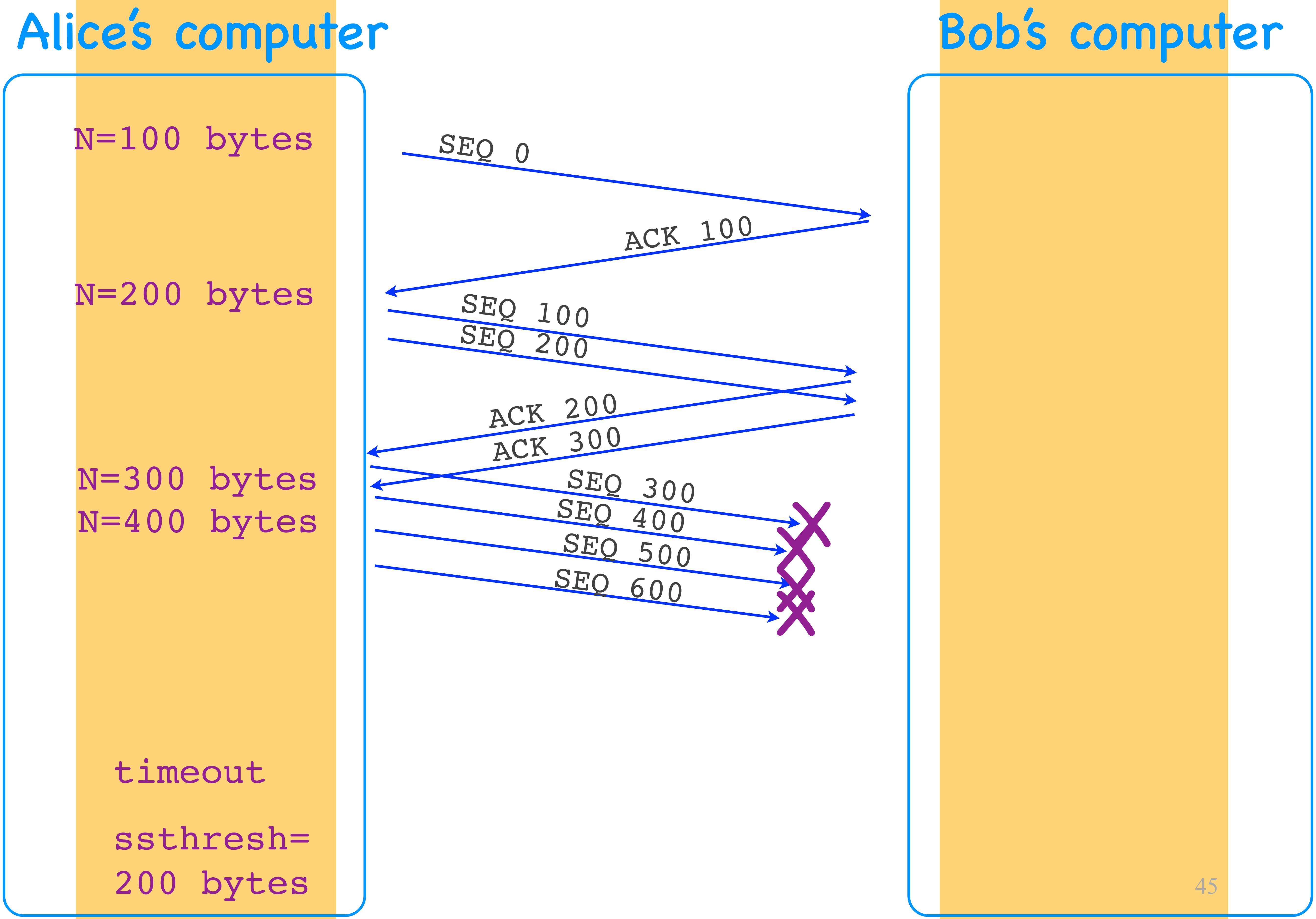
- Sender guesses the "right" congestion window based on the ACKs
- ACK = no congestion, increase window
- No ACK = congestion, decrease window

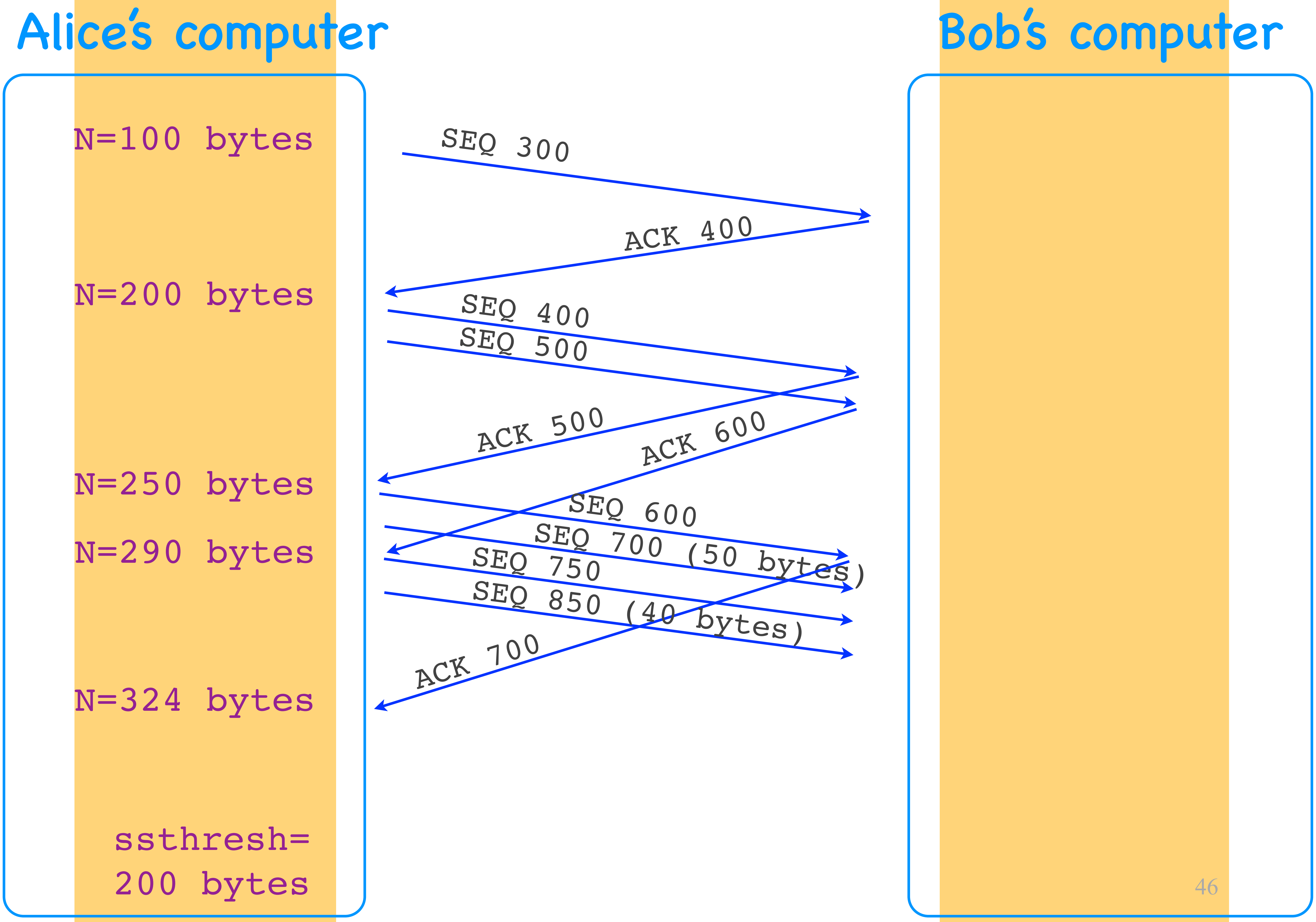

Alice's computer
Bob's computer
N=100 bytes
0 - 99
N=200 bytes
100 - 199
200 - 299
N=300 bytes
300 - 399
400 - 499
N=400 bytes
500 - 599
600 - 699
SEQ 0
ACK 100
SEQ 100
SEQ 200
ACK 200
ACK 300
SEQ 300
SEQ 400
SEQ 500
SEQ 600


Alice's computer
Bob's computer
N=100 bytes
N=200 bytes
N=300 bytes
N=400 bytes
SEQ 0
SEQ 100
SEQ 200
SEQ 300
SEQ 400
SEQ 500
SEQ 600

Increase window size

- Exponentially
  - by 1 MSS for every ACKed segment
  - = window doubles every RTT
  - when we do not expect congestion


Alice's computer
Bob's computer
N=100 bytes
SEQ 0
ACK 100
N=200 bytes
SEQ 100
SEQ 200
ACK 200
ACK 300
N=300 bytes
N=400 bytes
SEQ 300
SEQ 400
SEQ 500
SEQ 600
timeout
ssthresh=
200 bytes


Alice's computer
Bob's computer
N=100 bytes
SEQ 300
ACK 400
N=200 bytes
SEQ 400
SEQ 500
ACK 500
ACK 600
N=250 bytes
SEQ 600
SEQ 700 (50 bytes)
N=290 bytes
SEQ 750
SEQ 850 (40 bytes)
ACK 700
N=324 bytes
ssthresh=
200 bytes

Increase window size

- Exponentially
  - by 1 MSS for every ACKed segment
  - = window doubles every RTT
  - when we do not expect congestion
- Linearly
  - by MSS*MSS/N for every ACKed segment
  - = by 1 MSS every RTT
  - when we expect congestion

Goal: increase N by MSS bytes per RTT

Alice sends N unack-ed bytes per RTT

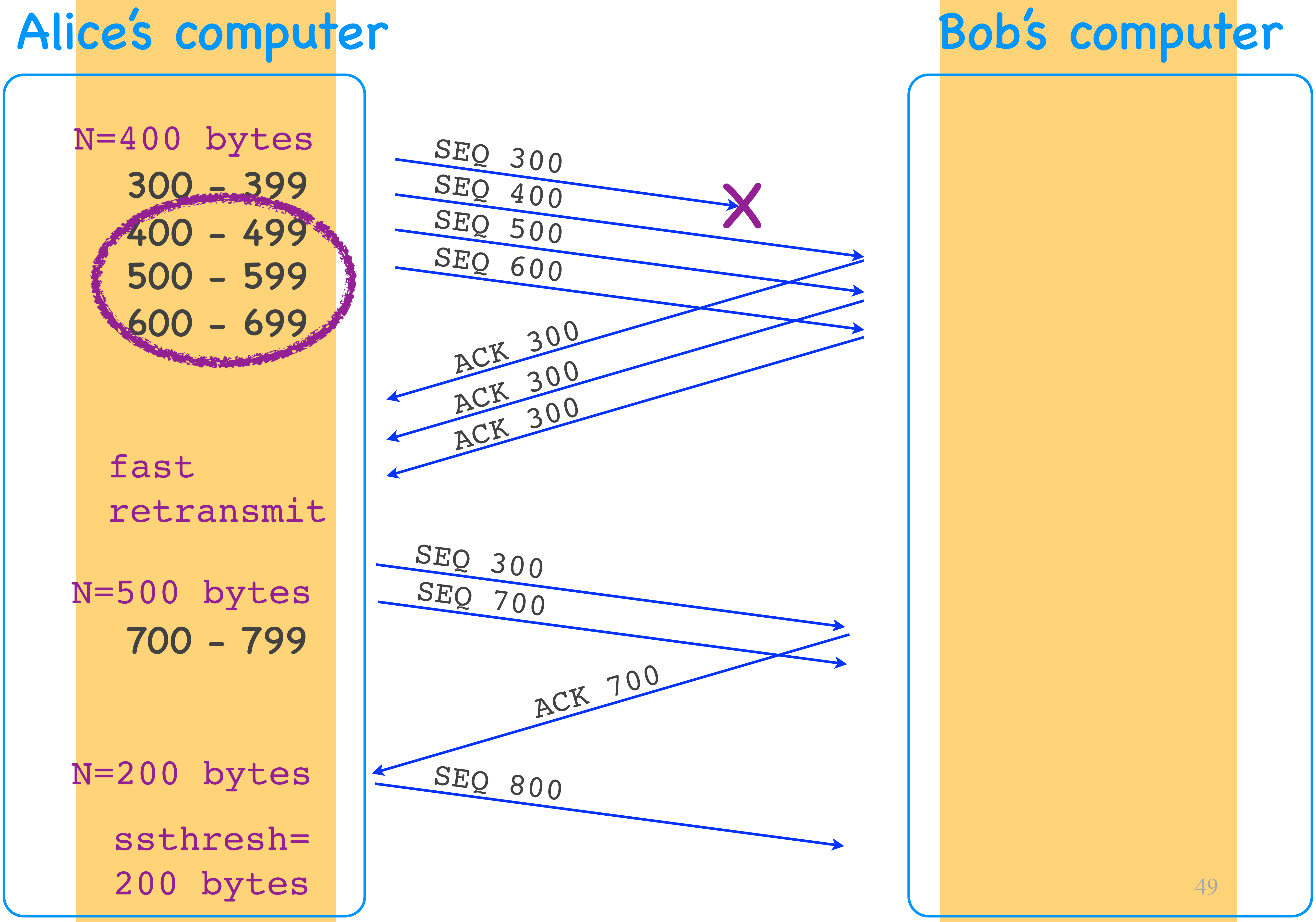
$$= \frac{N}{MSS} \text{ data segments per RTT}$$

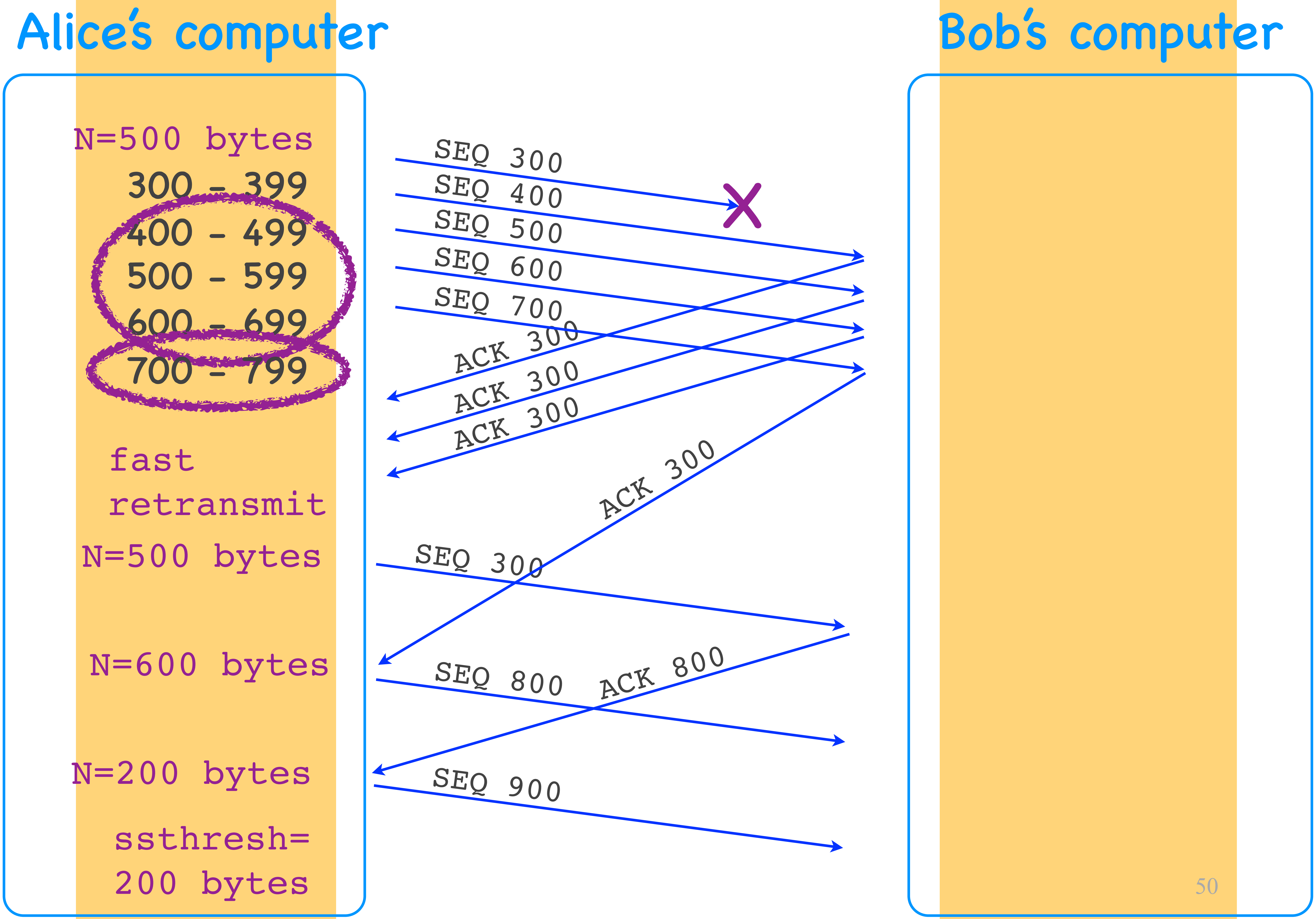
She expects $\frac{N}{MSS}$ ACKs per RTT

$$\frac{N}{MSS} * \frac{MSS*MSS}{N} \text{ bytes} = MSS \text{ bytes}$$

Basic algorithm (Tahoe)

- Set window to 1 MSS,
  increase exponentially
- On timeout, reset window to 1 MSS,
  set ssthresh to last window/2
- On reaching ssthresh,
  transition to linear increase

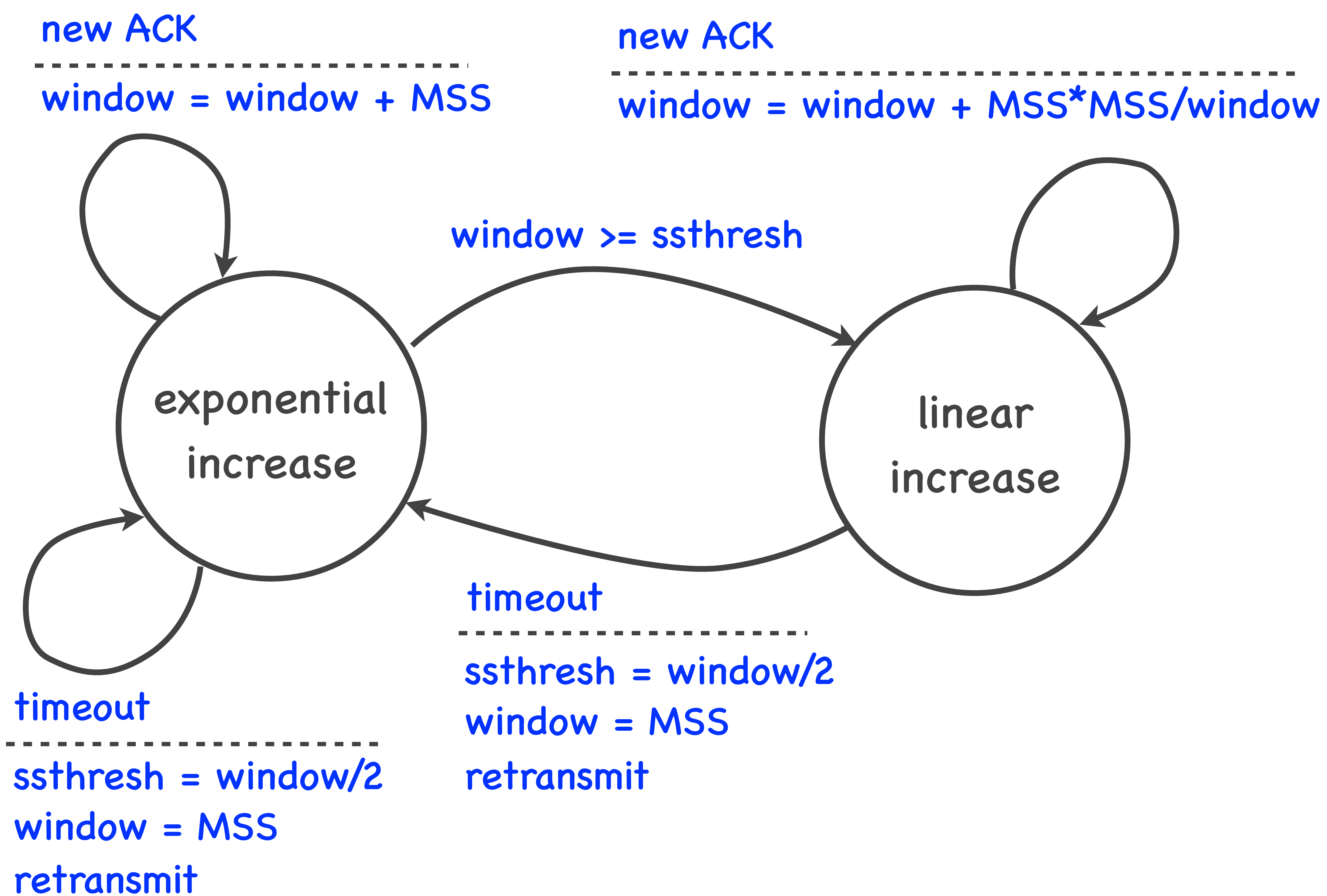

Alice's computer
Bob's computer
N=400 bytes
300 - 399
400 - 499
500 - 599
600 - 699
SEQ 300
SEQ 400
SEQ 500
SEQ 600
ACK 300
ACK 300
ACK 300
fast
retransmit
N=500 bytes
700 - 799
SEQ 300
SEQ 700
ACK 700
N=200 bytes
SEQ 800
ssthresh=
200 bytes

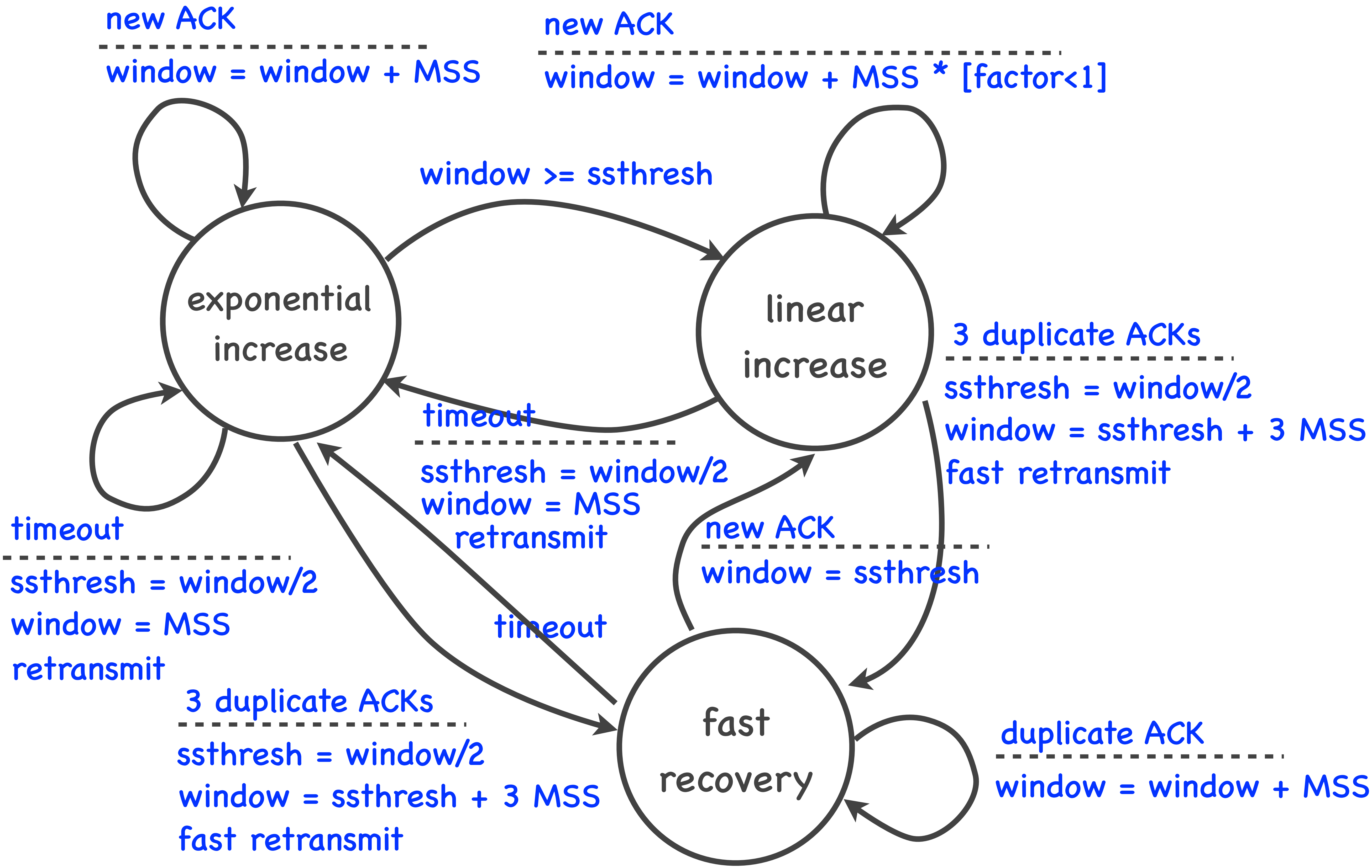


Alice's computer
Bob's computer
N=500 bytes
300 - 399
400 - 499
500 - 599
600 - 699
700 - 799
SEQ 300
SEQ 400
SEQ 500
SEQ 600
SEQ 700
ACK 300
ACK 300
ACK 300
ACK 300
fast retransmit
N=500 bytes
SEQ 300
N=600 bytes
SEQ 800
ACK 800
N=200 bytes
SEQ 900
ssthresh= 200 bytes

Basic algorithm (Reno)

- Set window to 1 MSS, increase exponentially
- On timeout, reset window to 1 MSS, set ssthresh to last window/2, retransmit
- On receiving 3 duplicate ACKs, set window to ssthresh (+inflation), retransmit
- On reaching ssthresh transition to linear increase


new ACK
window = window + MSS
new ACK
window = window + MSS*MSS/window
window >= ssthresh
exponential increase
linear increase
timeout
ssthresh = window/2
window = MSS
retransmit
timeout
ssthresh = window/2
window = MSS
retransmit



new ACK
window = window + MSS
new ACK
window = window + MSS * [factor<1]
window >= ssthresh
exponential increase
linear increase
3 duplicate ACKs
ssthresh = window/2
window = ssthresh + 3 MSS
fast retransmit
timeout
ssthresh = window/2
window = MSS
retransmit
new ACK
window = ssthresh
timeout
ssthresh = window/2
window = MSS
retransmit
timeout
3 duplicate ACKs
ssthresh = window/2
window = ssthresh + 3 MSS
fast retransmit
fast recovery
duplicate ACK
window = window + MSS

TCP terminology

- Exponential increase = slow start
  - it's called slow, because it starts from a small window; but it's not really slow, the window increases exponentially
- Linear increase = congestion avoidance
  - this term does make sense; it means that TCP expects congestion, so it increases the window more cautiously

Flow + congestion control

- Goal: not overwhelm receiver or network
- How: sender window
  - sender learns receiver window from receiver
  - sender computes congestion window on its own
  - Sender window = min{ receiver w, congestion w }