

Lecture 7:

The Transport Layer

Katerina Argyraki, EPFL

Outline (from last lecture)

- Process-to-process communication
 - UDP
 - TCP
- Reliable data delivery
 - Imaginary protocol
 - UDP
 - TCP

Outline (from last lecture)

- Process-to-process communication
 - UDP
 - TCP
- **Reliable data delivery**
 - Imaginary protocol
 - **UDP**
 - TCP

UDP: reliability elements

- **Checksums** to detect corruption

Alice's computer

`rdt_send()`

`udt_send()`

Bob's computer

`rdt_rcv()`

check for
corruption

`deliver_data()`

Outline (from last lecture)

- Process-to-process communication
 - UDP
 - TCP
- **Reliable data delivery**
 - Imaginary protocol
 - UDP
 - **TCP**

TCP: reliability elements

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

Alice's computer

rdt_send([A])

udt_send(...)

rdt_rcv(...)

deliver_data([B])

Bob's computer

rdt_rcv(...)

deliver_data([A])

rdt_send([B])

udt_send(...)

SEQ 0 | ACK 10 | [A]

SEQ 10 | ACK 1 | [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

200 bytes

Bob's computer

1500 bytes

1500 bytes

100 bytes

SEQ 0
ACK 10 [GET request]

SEQ 10
ACK 200 [file, part 1]

SEQ 1510
ACK 200 [file, part 2]

SEQ 3010
ACK 200 [file, part 3]

SEQ 200
ACK 1510

SEQ 200
ACK 3010

SEQ 200
ACK 3110

Simple things to remember

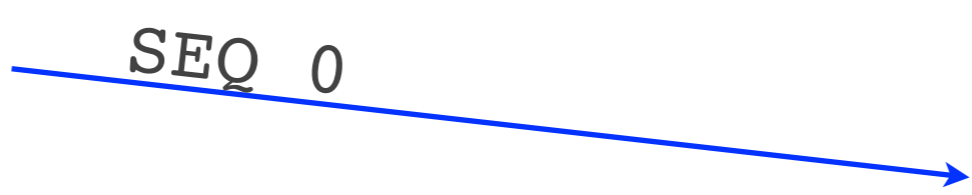
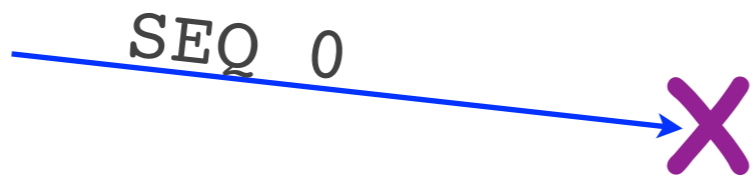
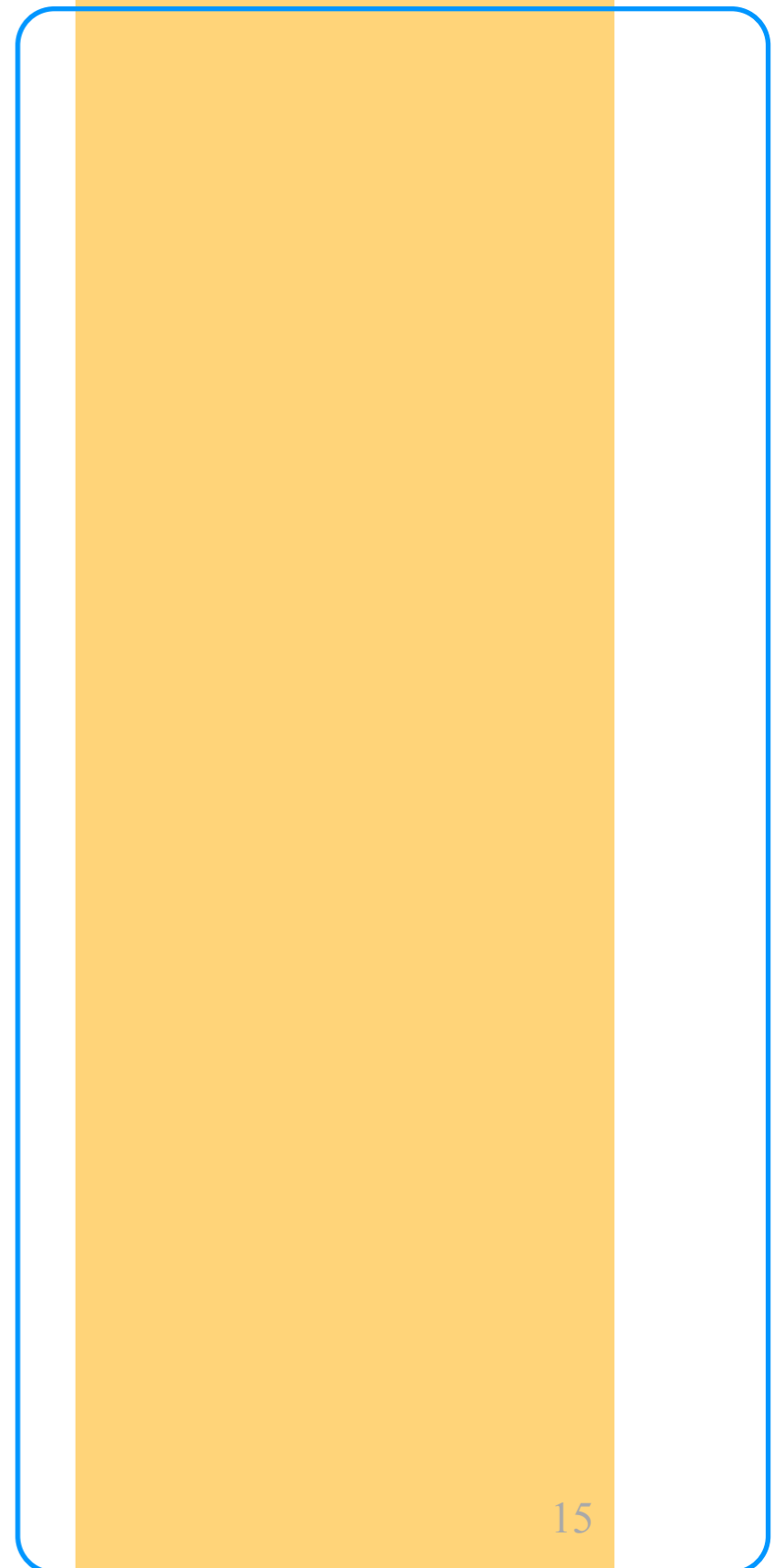
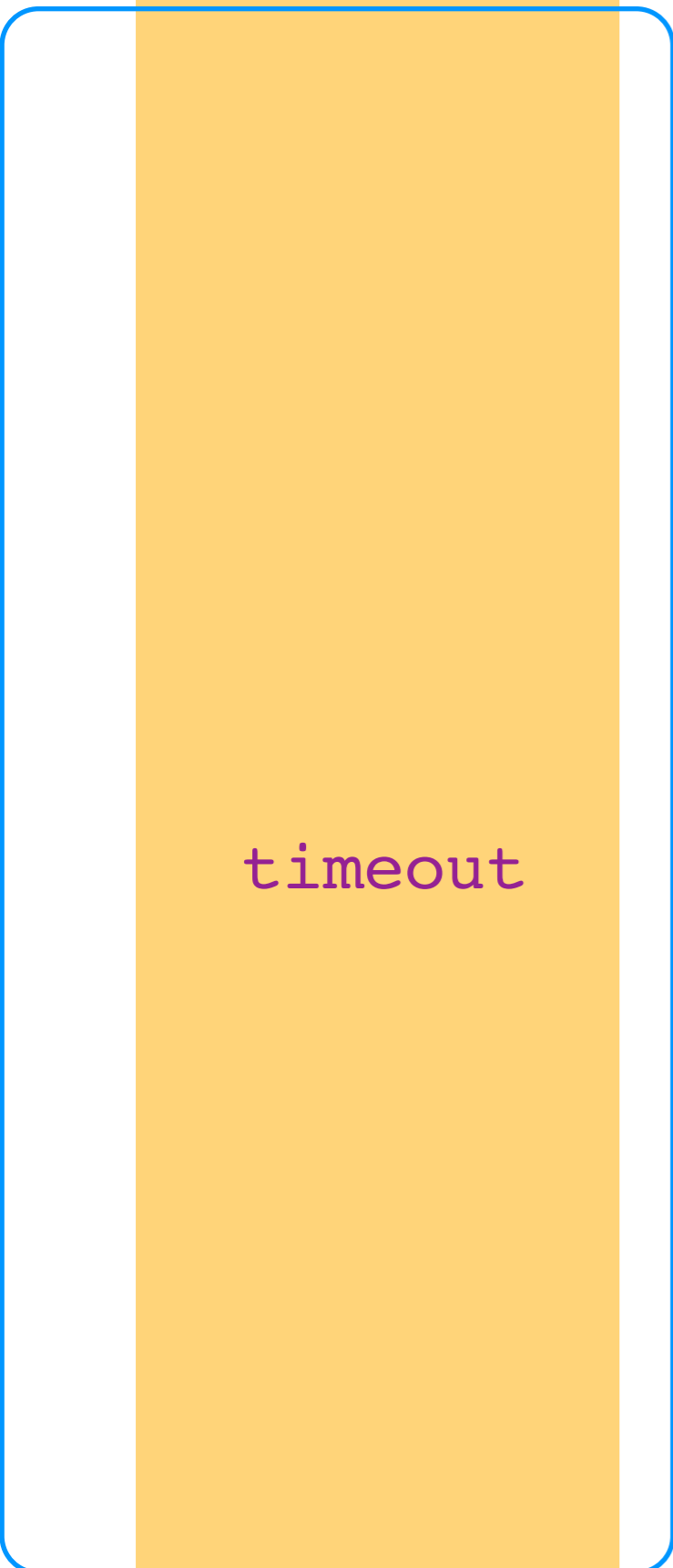
- A TCP connection may carry bidirectional communication
- A segment may or may not carry data (but it always carries a SEQ)
- There exists a maximum segment size (MSS), dictated by network properties

TCP: reliability elements

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

Alice's computer

Bob's computer



How long should the timeout be?

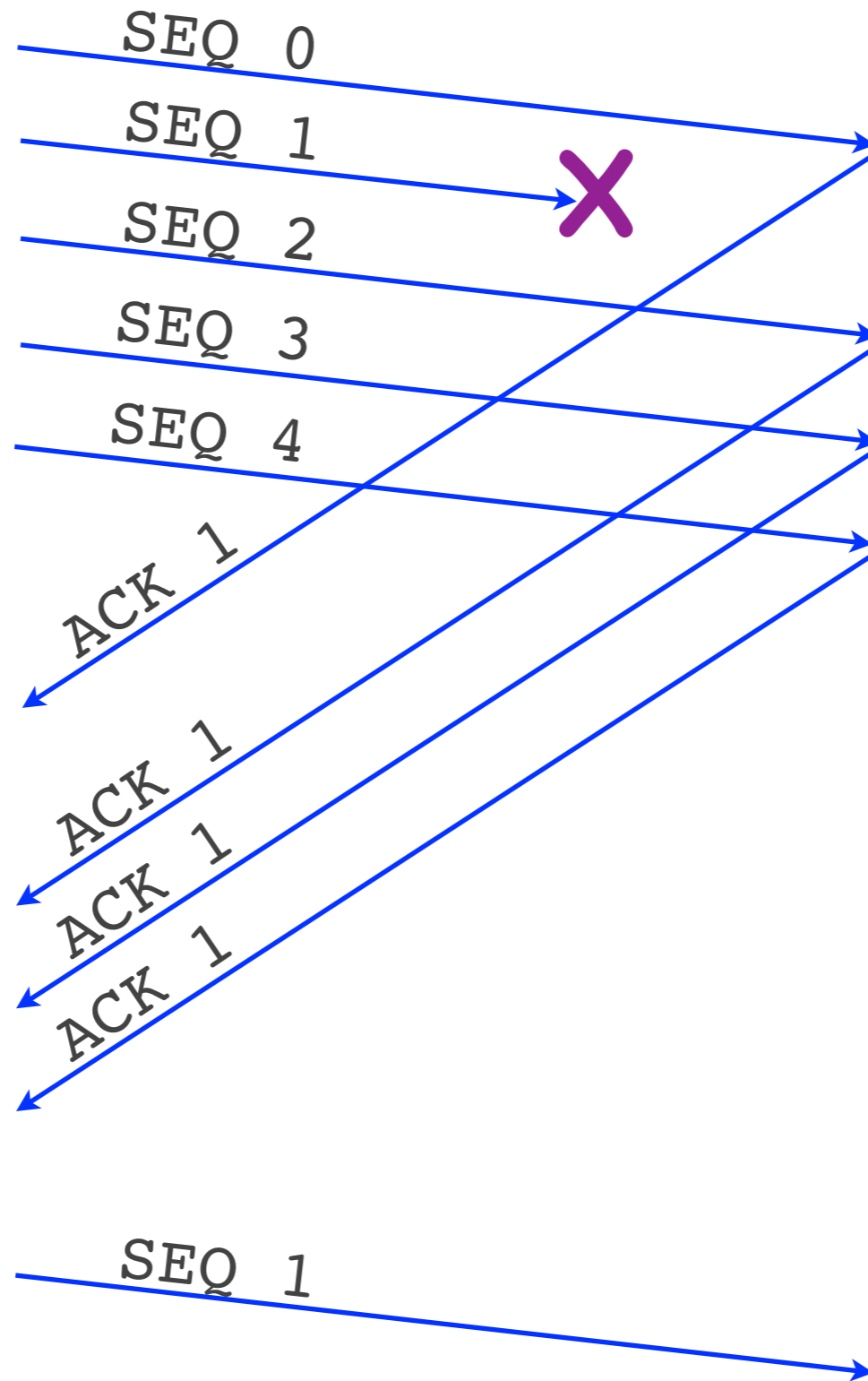
Timeout calculation

- EstimatedRTT =
 $0.875 \text{ EstimatedRTT} + 0.125 \text{ SampleRTT}$
- DevRTT = function(RTT variance)
- Timeout = EstimatedRTT + 4 DevRTT

Empirical, conservative prediction of RTT

Alice's computer

Bob's computer



fast
retransmit

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 corruption+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

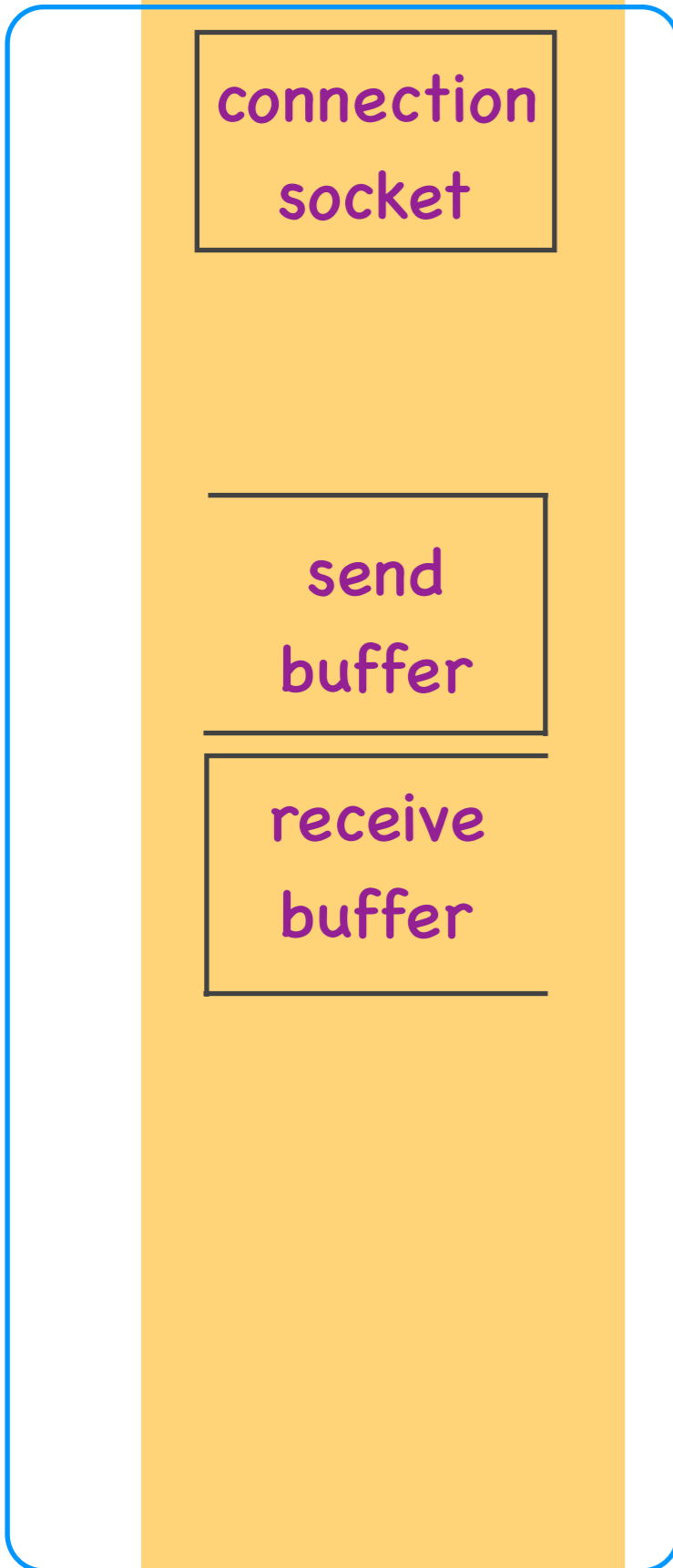
TCP elements

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

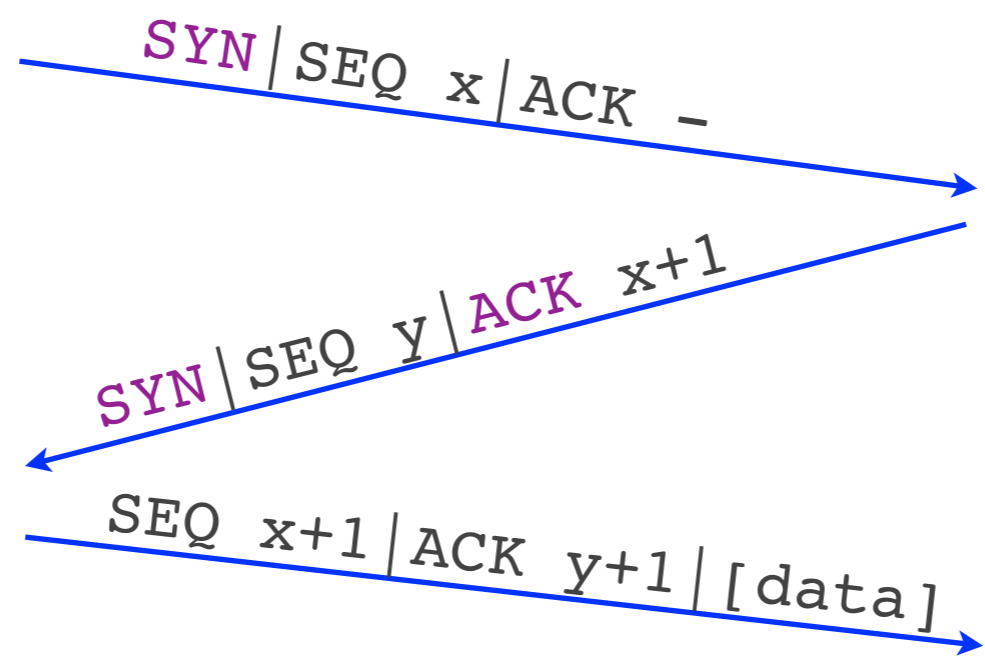
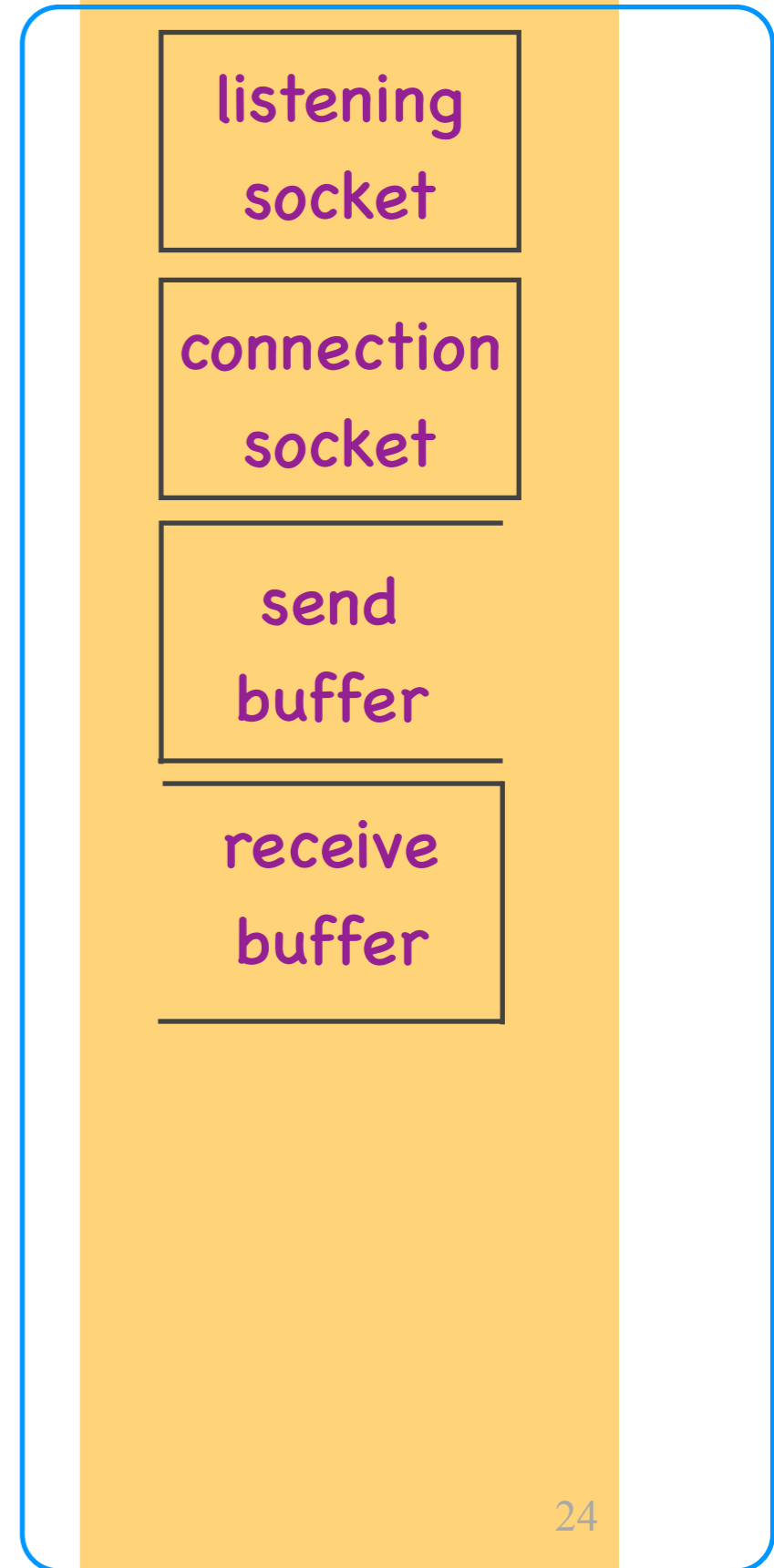
TCP elements

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

Alice's computer

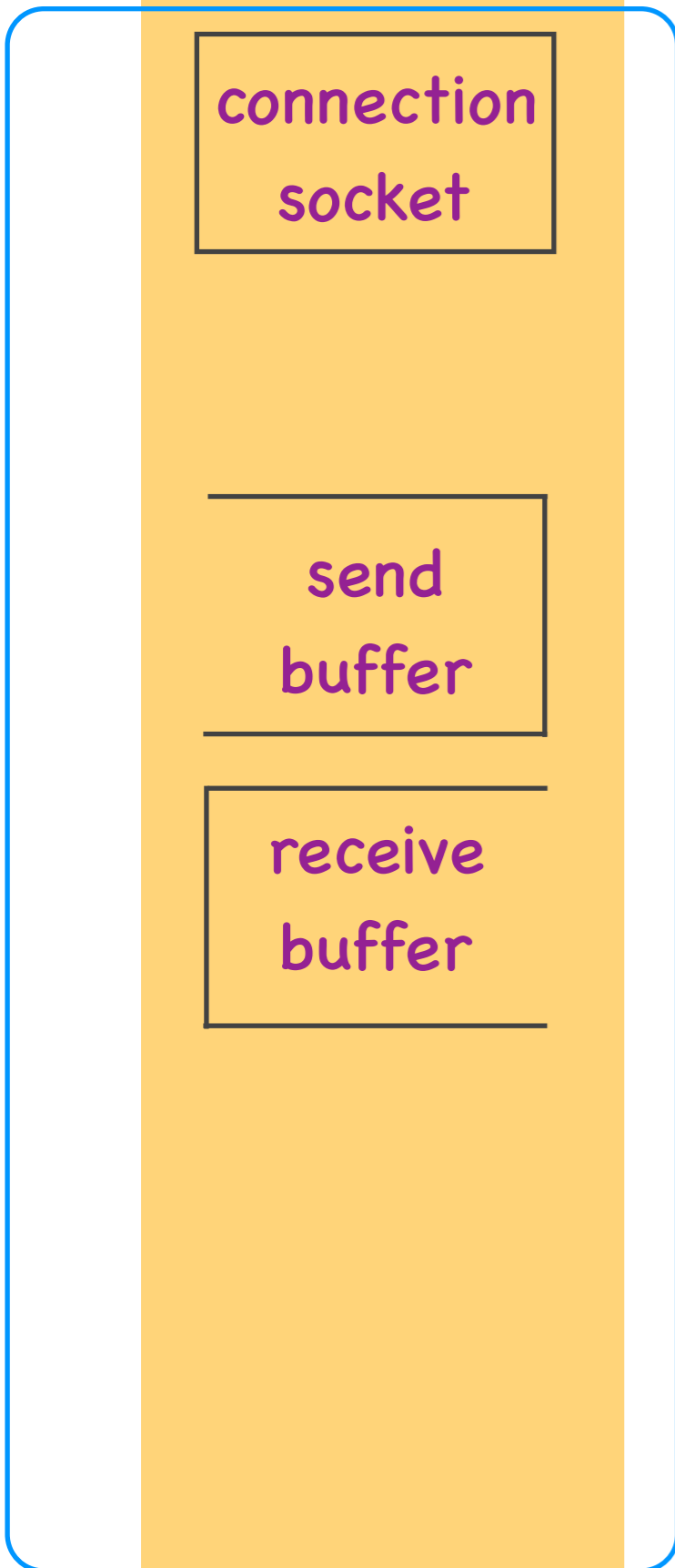


Bob's computer

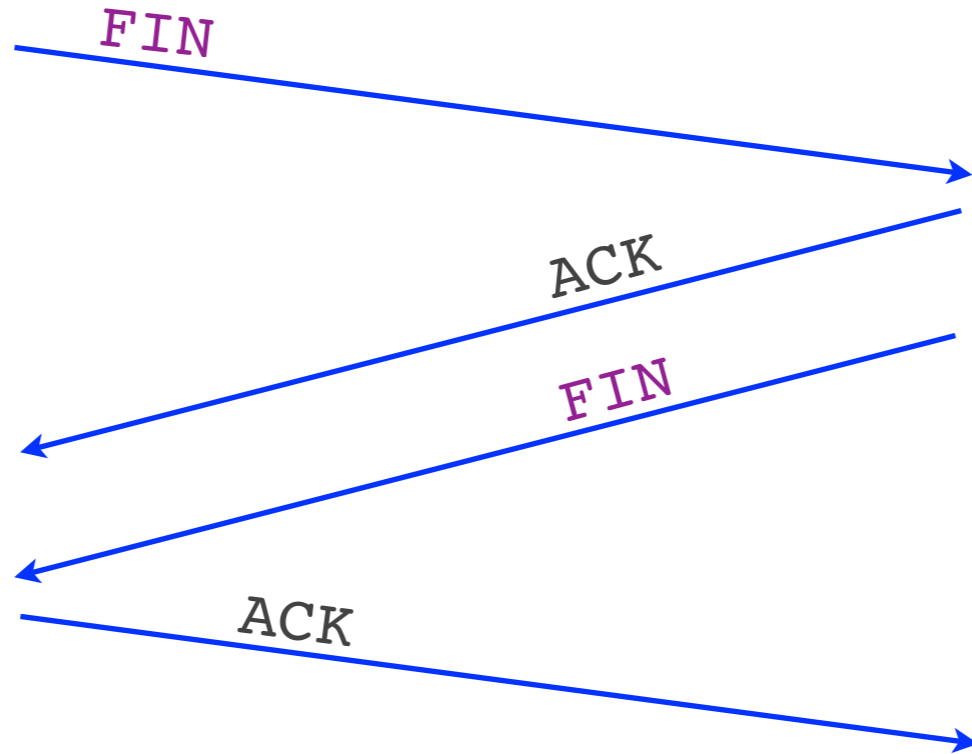
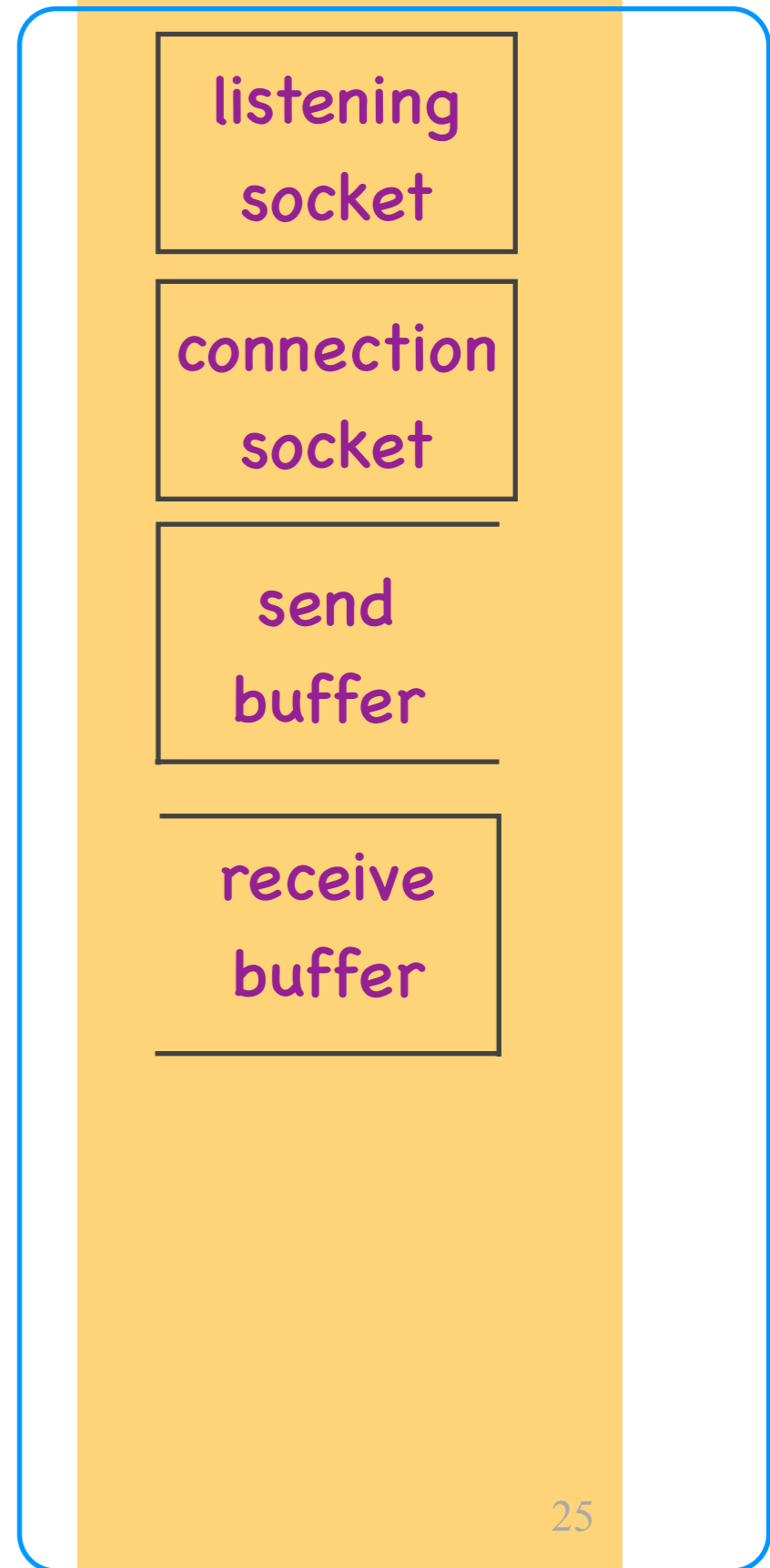


connection established

Alice's computer



Bob's computer



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

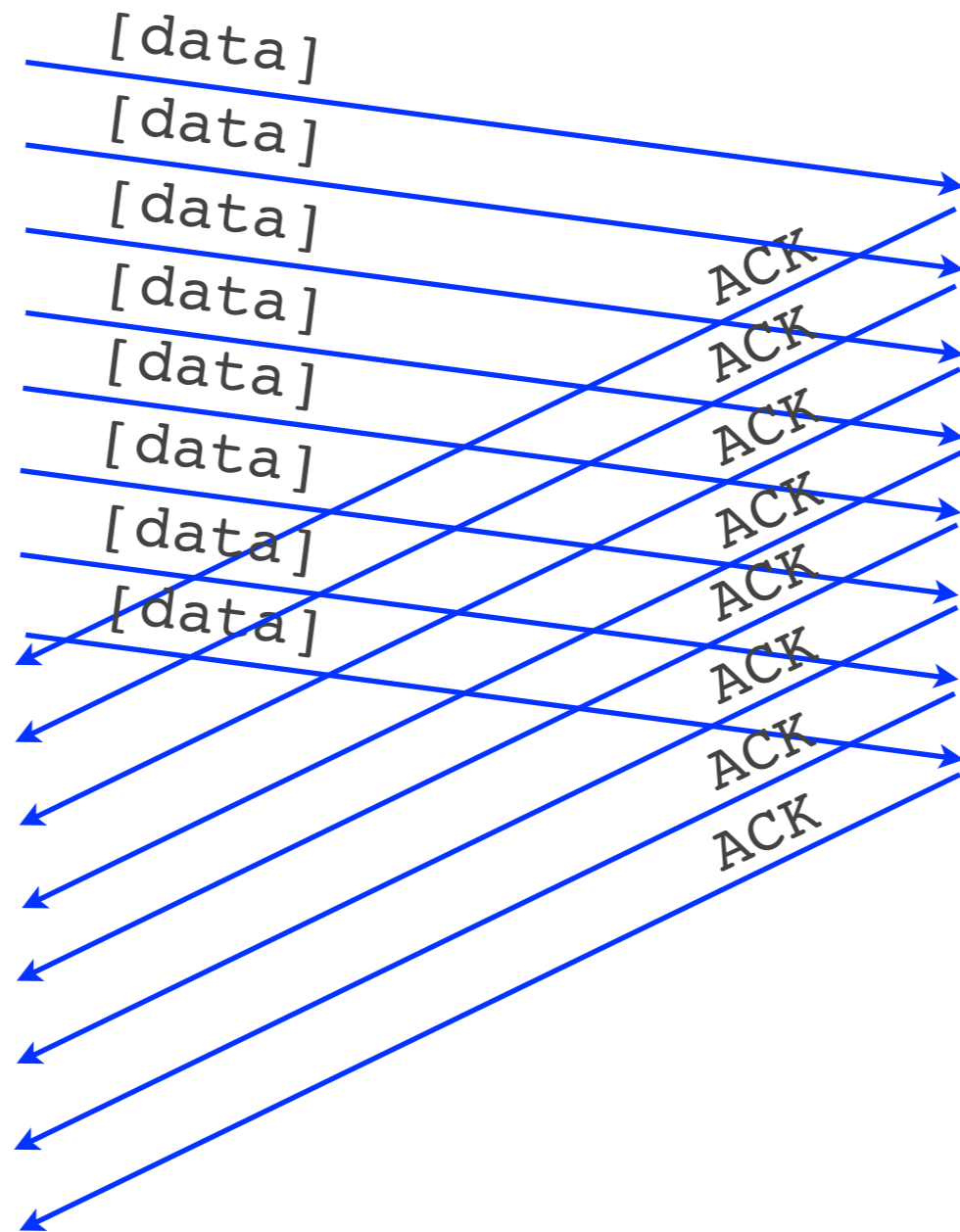
- Connection setup and teardown
- **Connection hijacking**
- Connection setup (SYN) flooding
- Flow control
- Congestion control

Congestion control

- Goal: not overwhelm the **network**
 - not send at a rate that the network would create 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



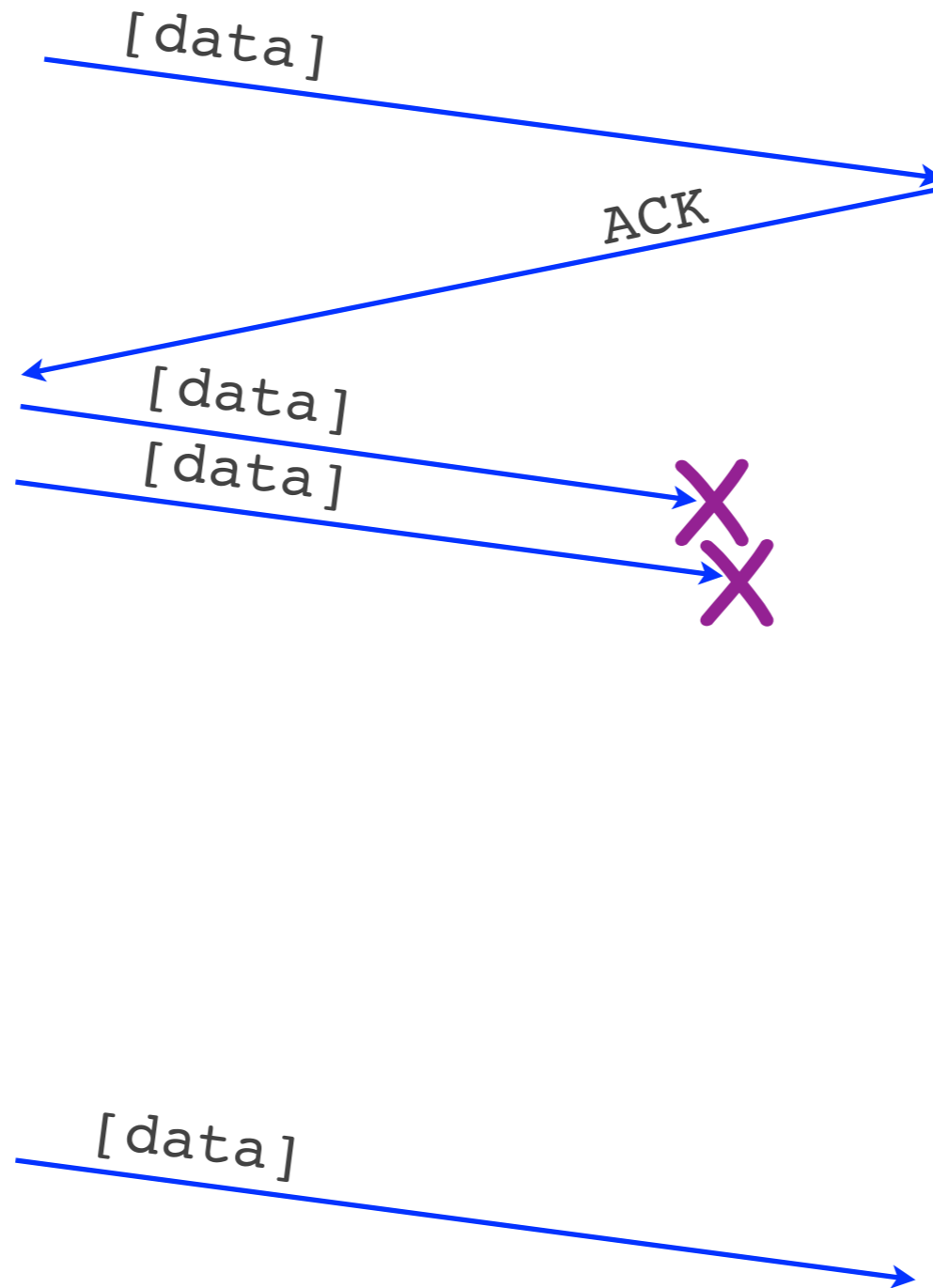
$R \text{ bps} \times \text{RTT sec}$
bandwidth delay product

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



timeout

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

N=100 bytes

~~0 - 99~~

N=200 bytes

~~100 - 199~~

~~200 - 299~~

N=300 bytes

300 - 399

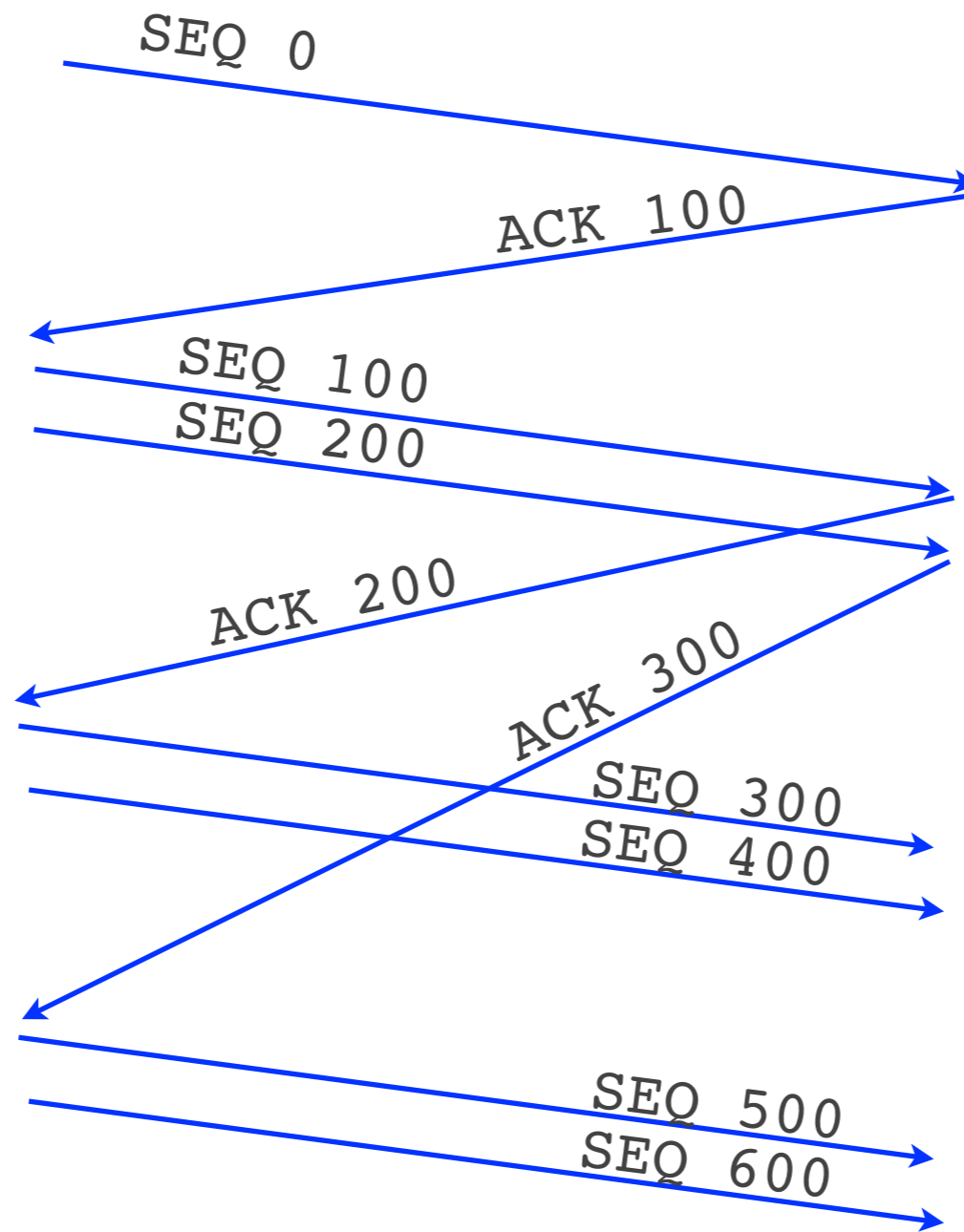
400 - 499

N=400 bytes

500 - 599

600 - 699

Bob's computer



Alice's computer

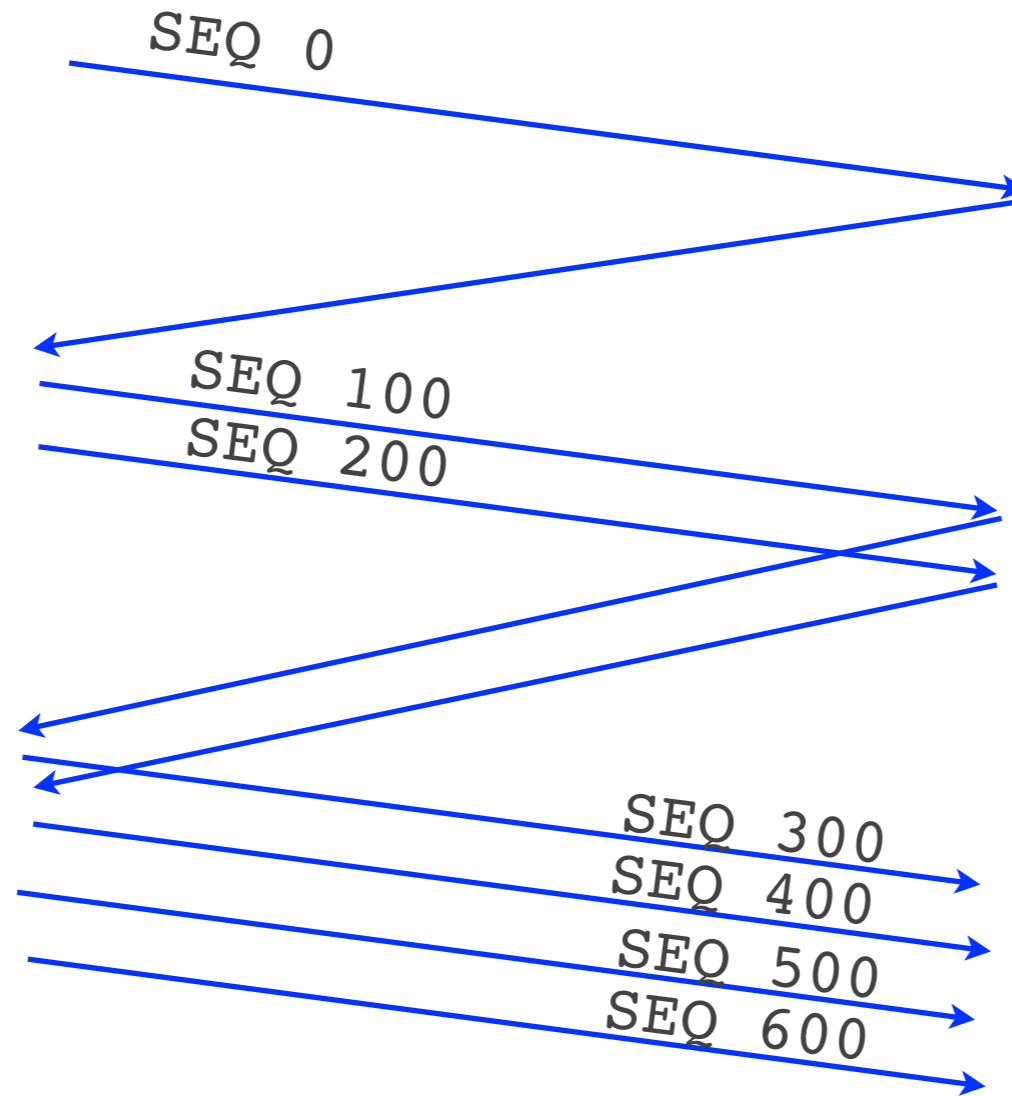
N=100 bytes

N=200 bytes

N=300 bytes

N=400 bytes

Bob's computer



Increase window size

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

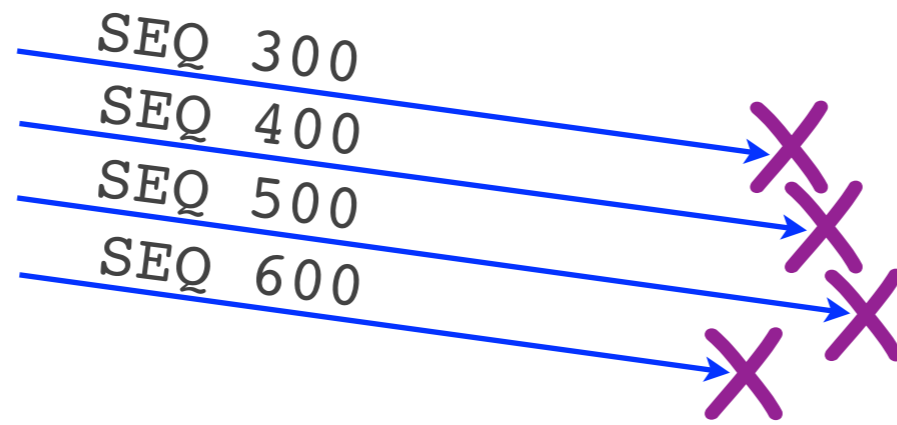
Alice's computer

N=400 bytes

timeout

ssthresh=
200 bytes

N=100 bytes



Bob's computer

Alice's computer

N=100 bytes

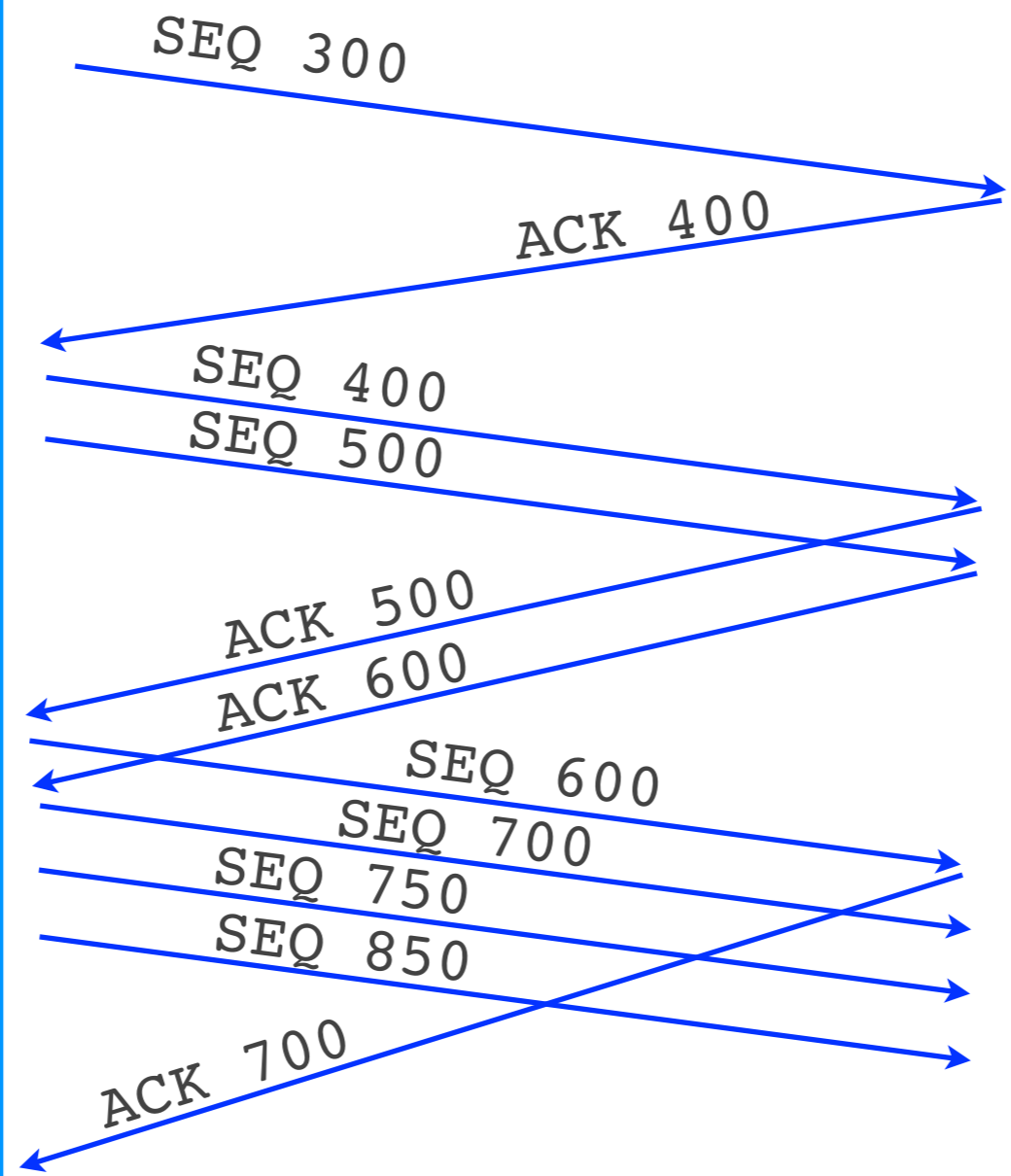
N=200 bytes

N=250 bytes

N=290 bytes

N=324 bytes

Bob's computer



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

Alice's computer

N=100 bytes

N=200 bytes

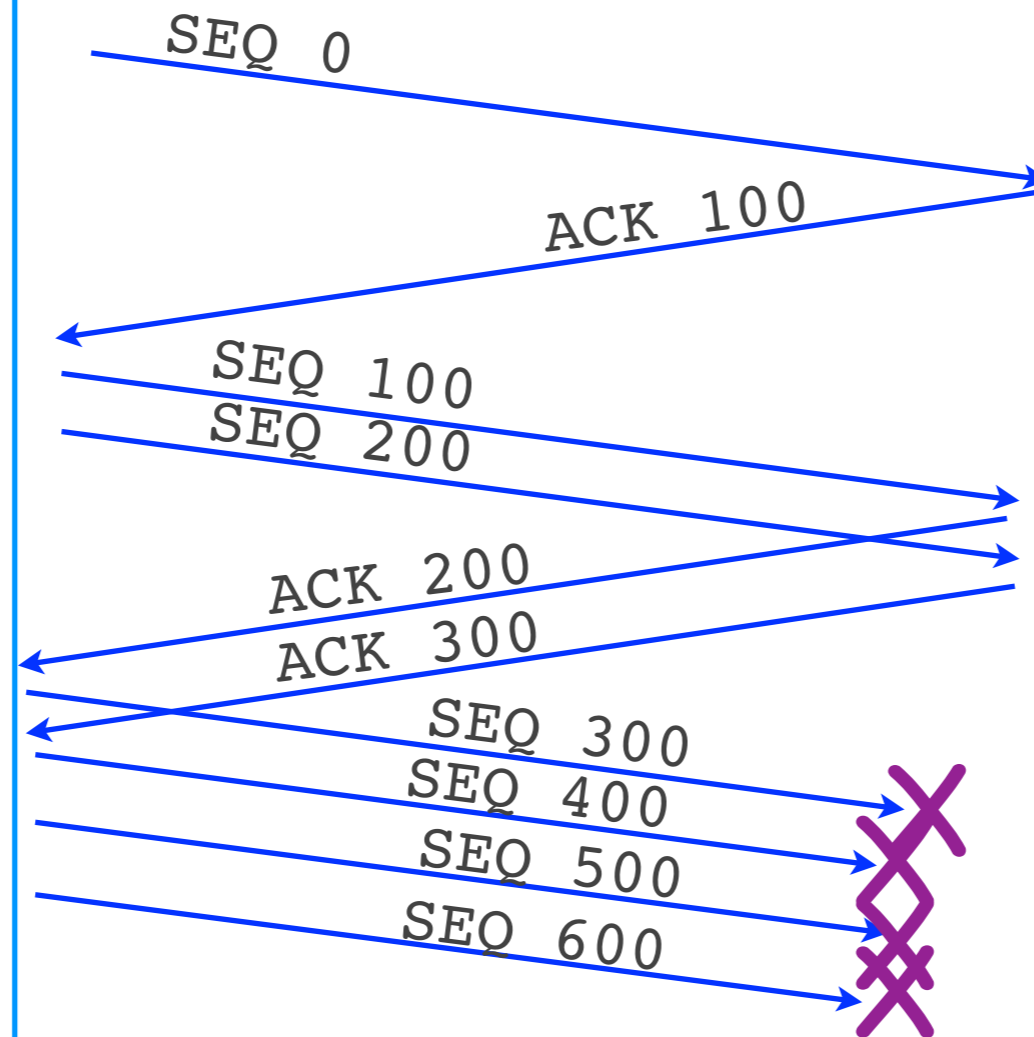
N=300 bytes

N=400 bytes

timeout

ssthresh=
200 bytes

Bob's computer



Alice's computer

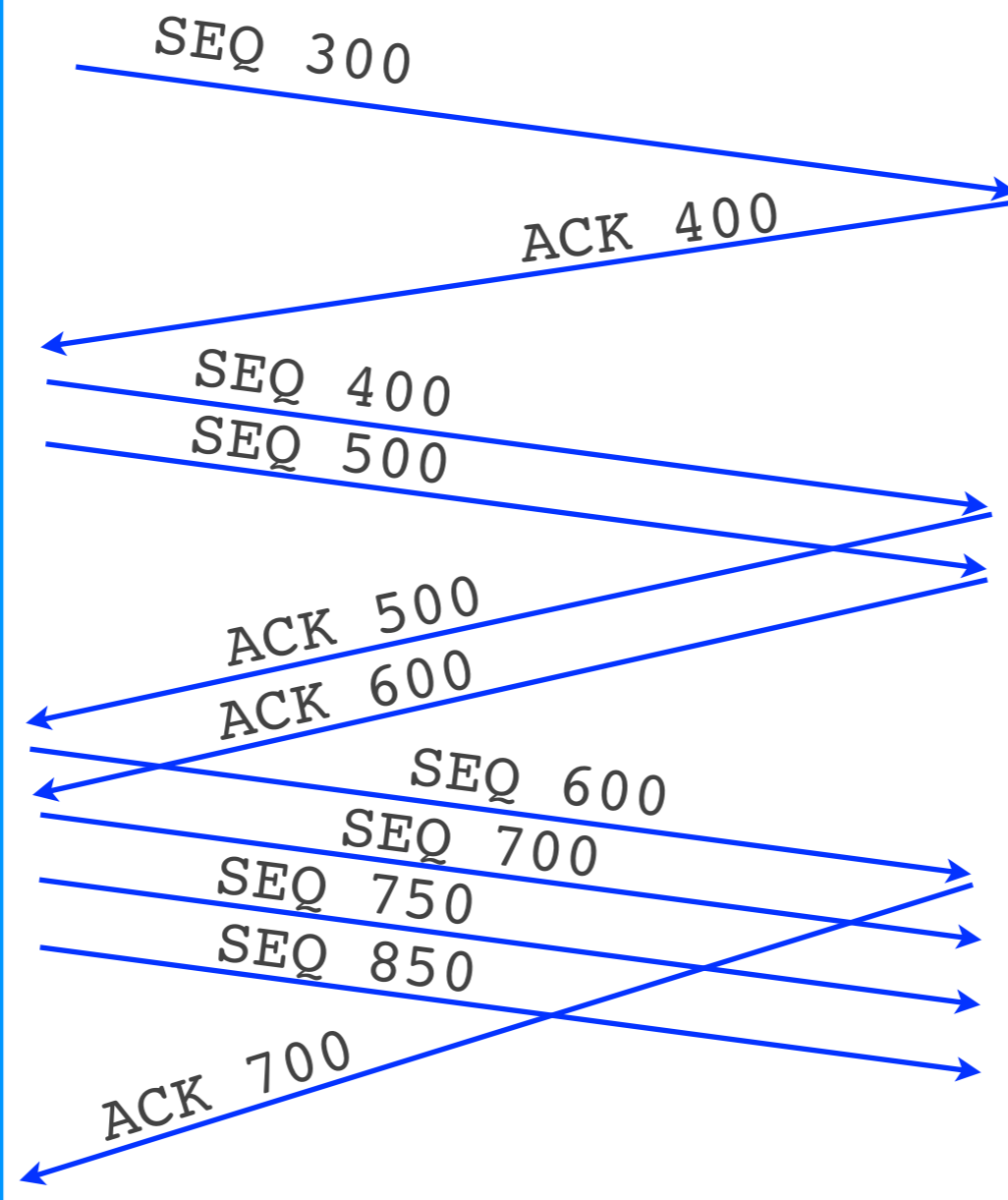
N=100 bytes

N=200 bytes

N=250 bytes

N=290 bytes

N=324 bytes



Bob's computer

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

N=400 bytes

300 - 399

400 - 499

500 - 599

600 - 699

fast

retransmit

N=500 bytes

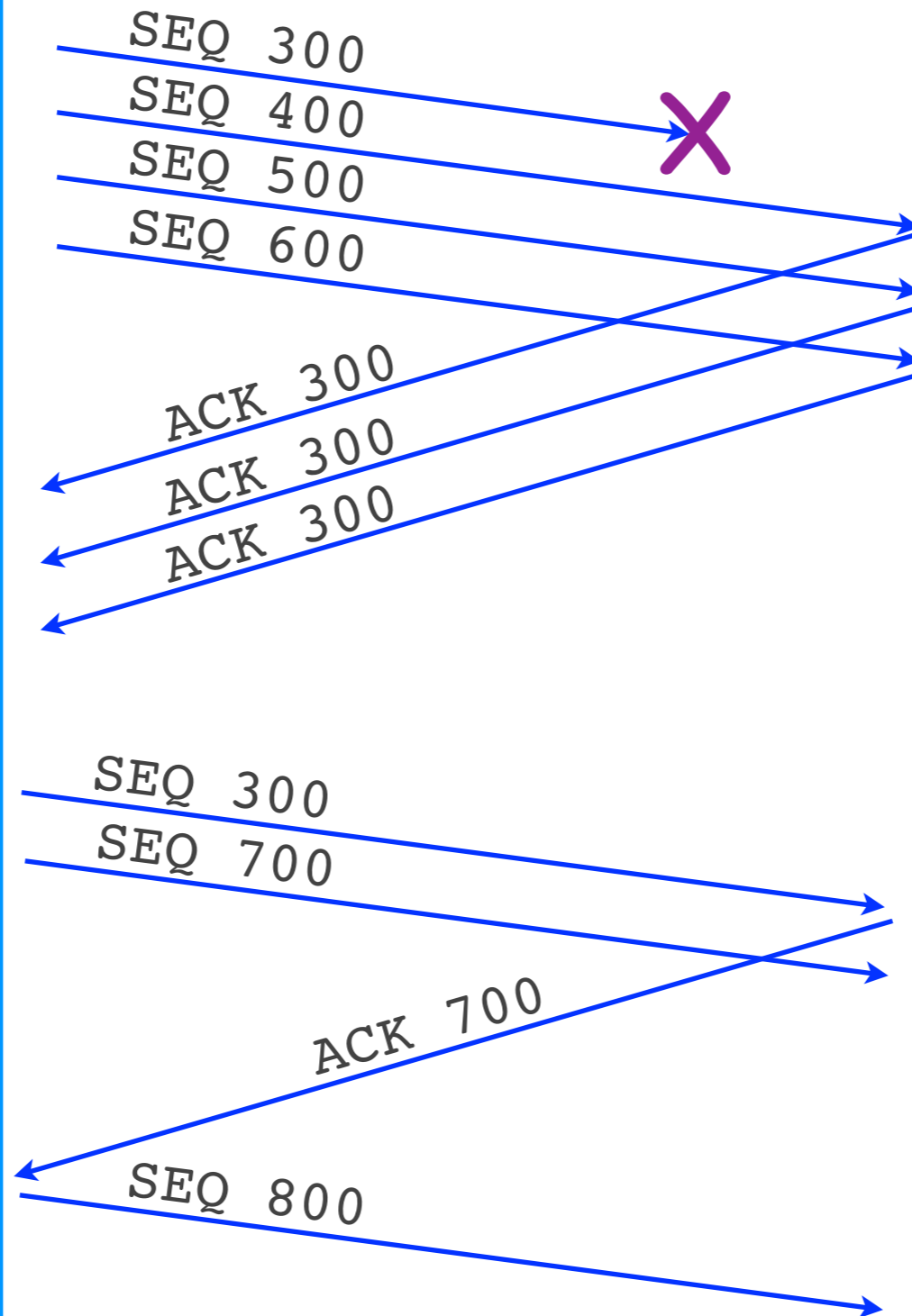
700 - 799

N=200 bytes

ssthresh=

200 bytes

Bob's computer



Basic algorithm (Reno)

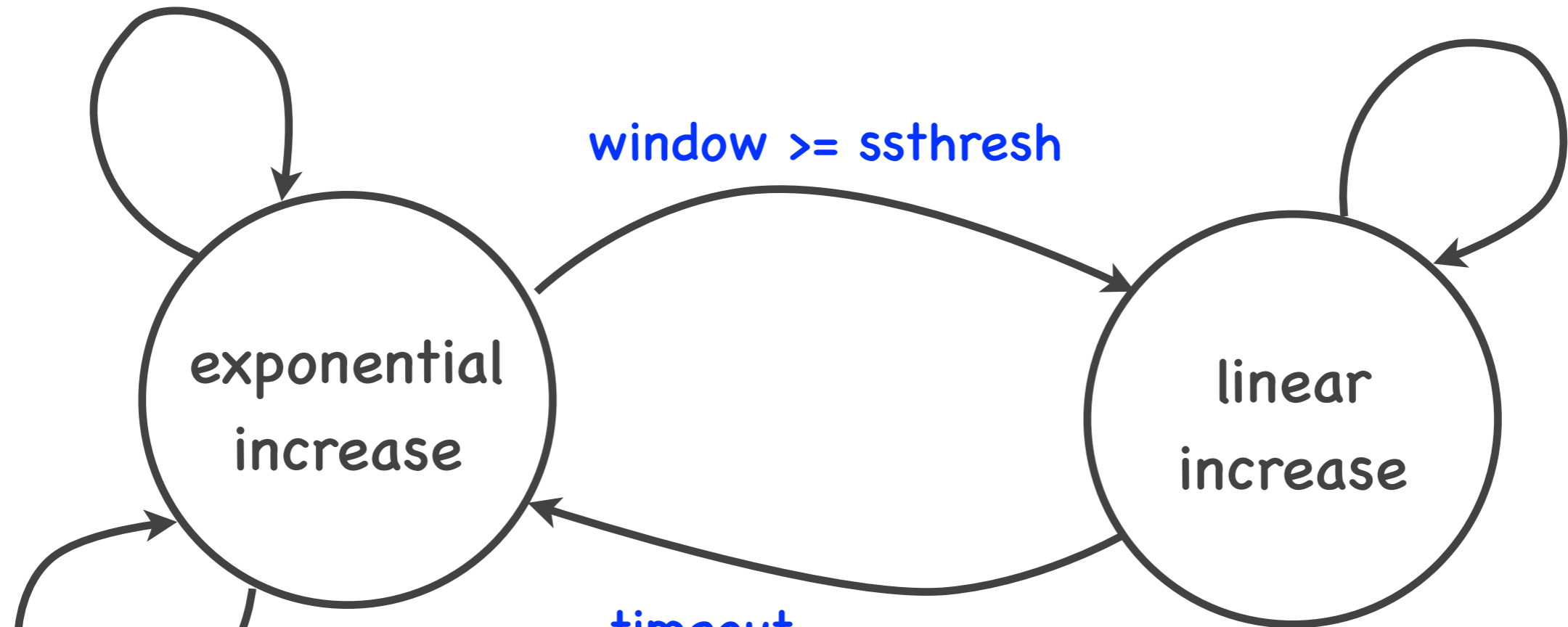
- Set window to 1 MSS, increase exponentially
- On timeout, reset window to 1 MSS, set ssthresh to last window/2, retransmit
- On receiving 3 duplicate ACKs, set window to ssthresh (+inflation), retransmit
- On reaching ssthresh transition to linear increase

new ACK

$$\text{window} = \text{window} + \text{MSS}$$

new ACK

$$\text{window} = \text{window} + \text{MSS} * \text{MSS} / \text{window}$$



window >= ssthresh

exponential
increase

linear
increase

timeout

$$\text{ssthresh} = \text{window} / 2$$

$$\text{window} = \text{MSS}$$

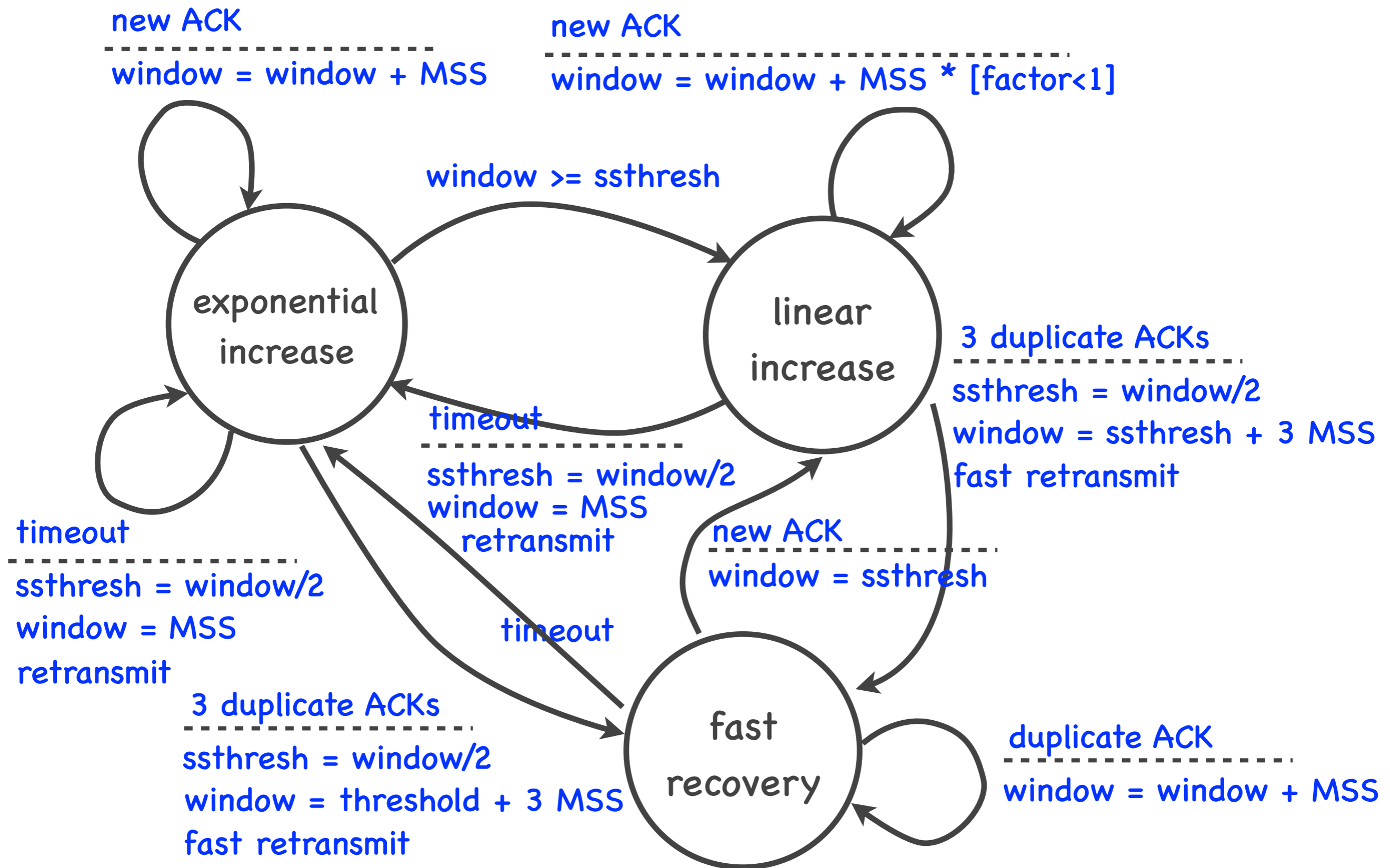
retransmit

timeout

$$\text{ssthresh} = \text{window} / 2$$

$$\text{window} = \text{MSS}$$

retransmit

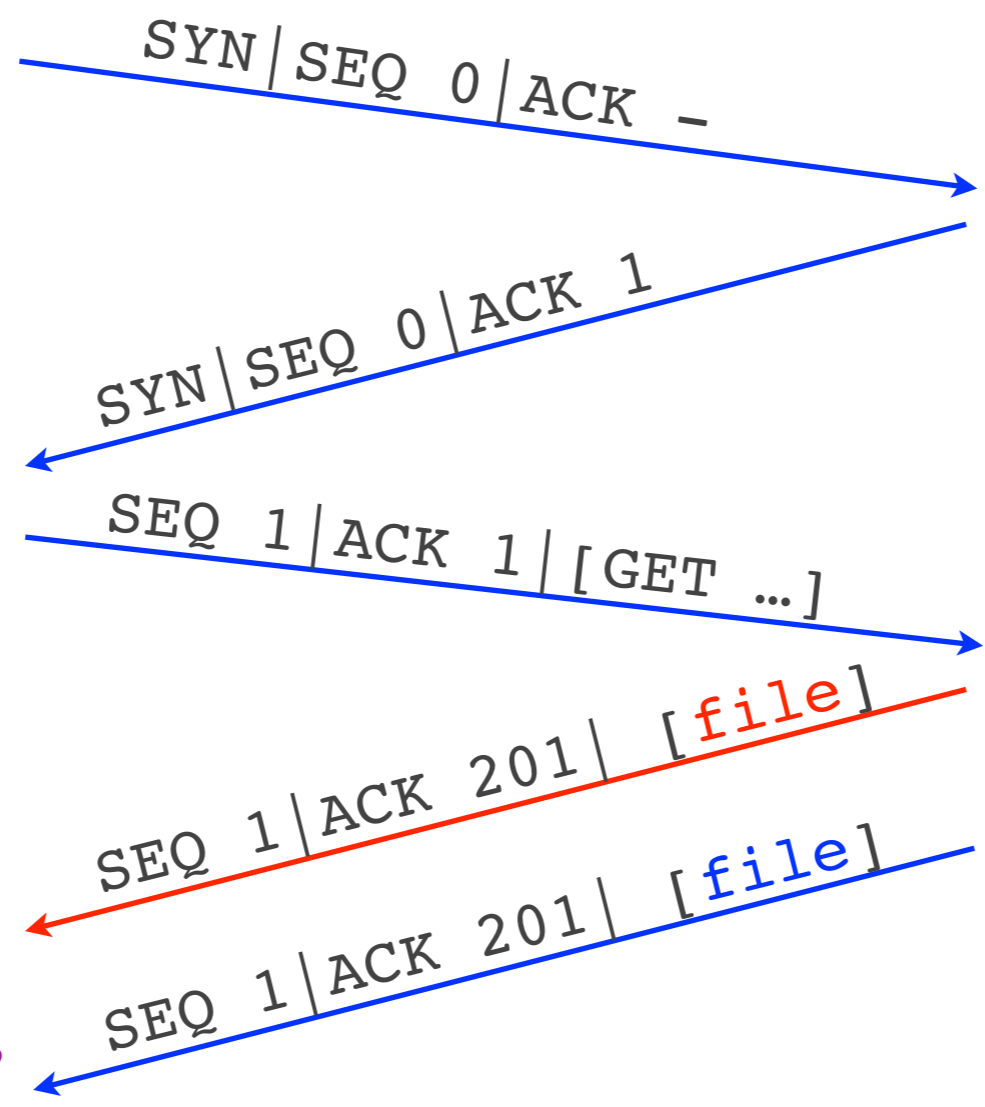


TCP terminology

- 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

Alice's computer

Jack's computer
Bob's computer



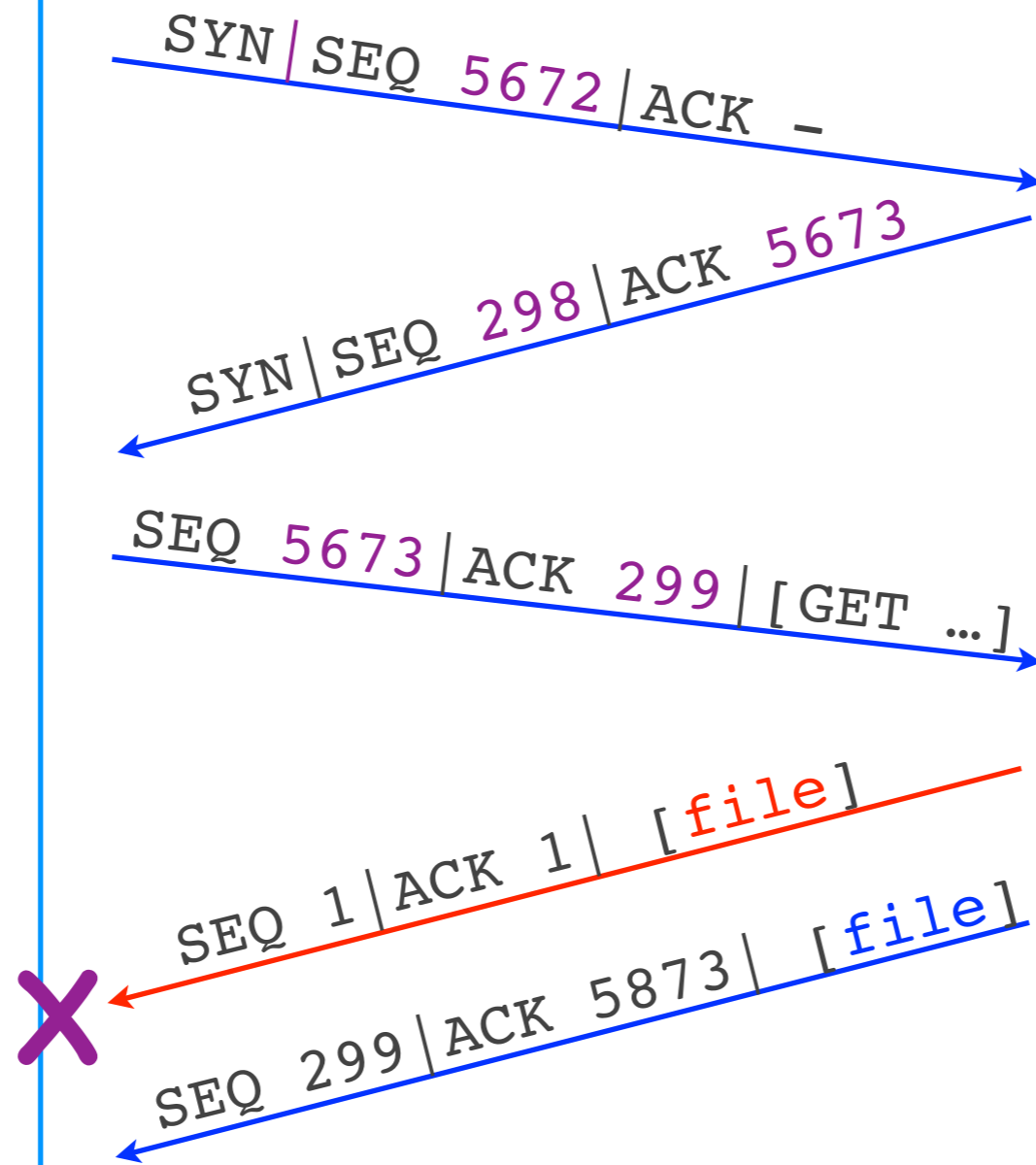
X

connection
hijacked

How to prevent connection hijacking?

Alice's computer

Jack's computer
Bob's computer



Connection hijacking

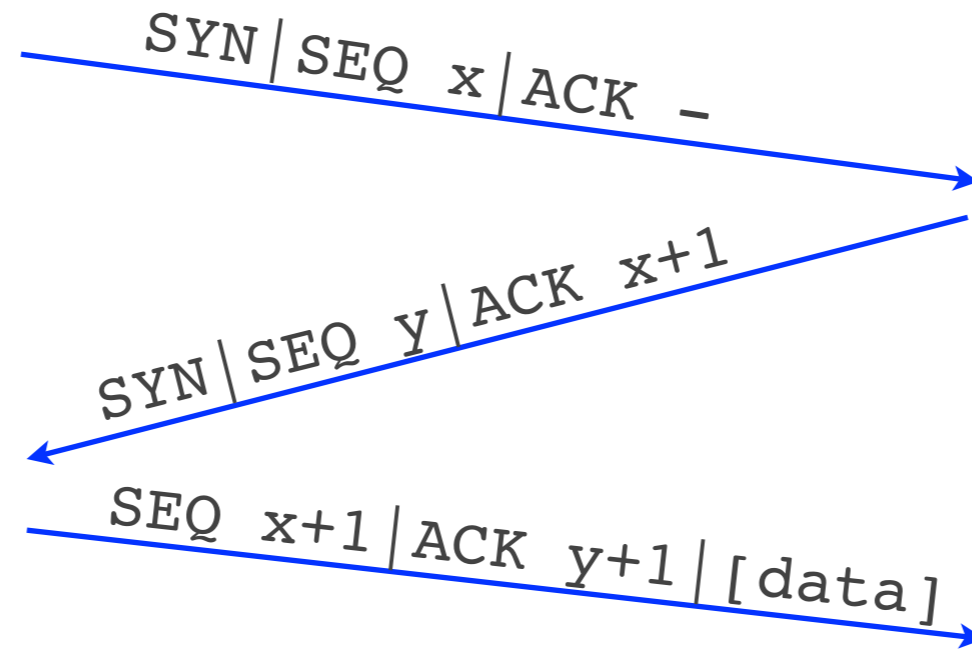
- Attacker **impersonates** TCP server (or client)
 - sends segment that appears to be coming from the impersonated end-system
- Approach: fake valid segment
 - if the TCP header predictable
- Solution: make TCP header (SEQs) **unpredictable**

TCP elements

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

Alice's computer

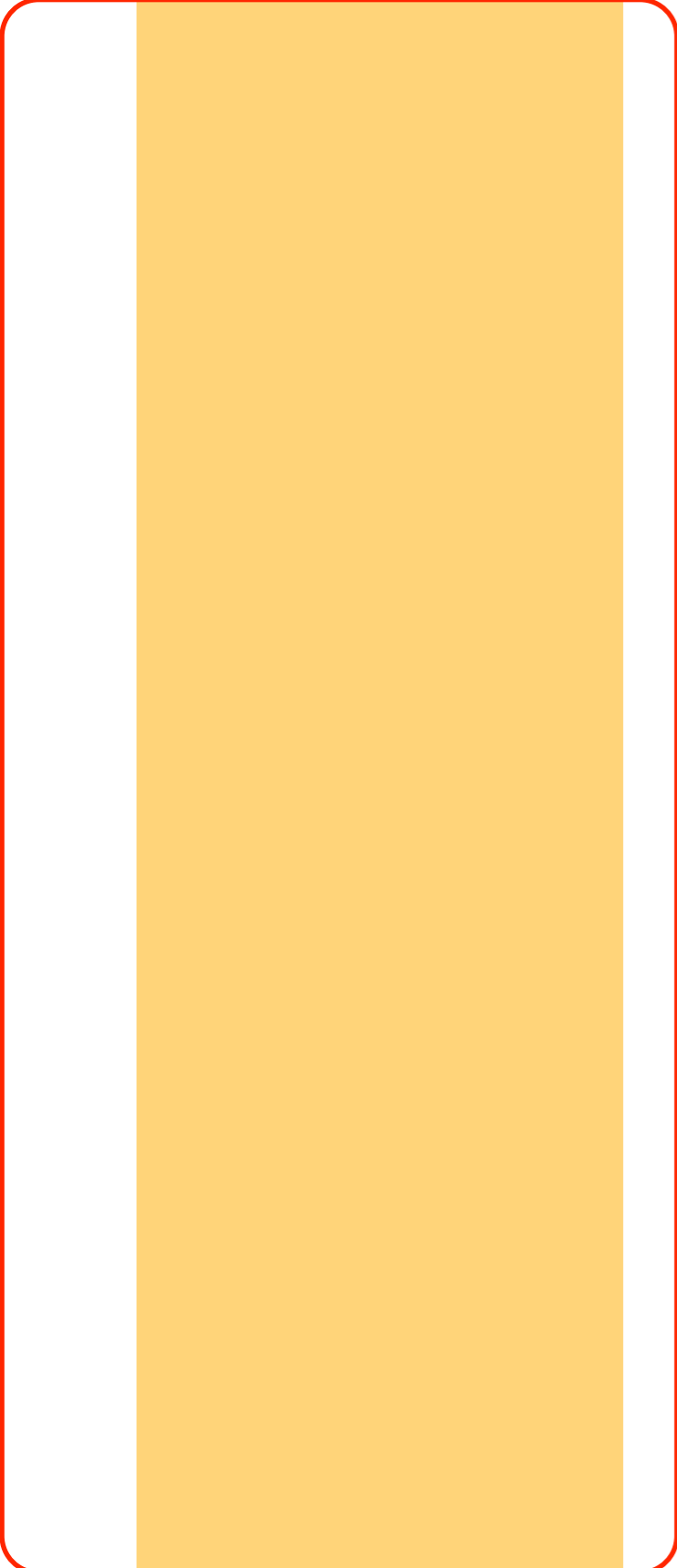
Bob's computer



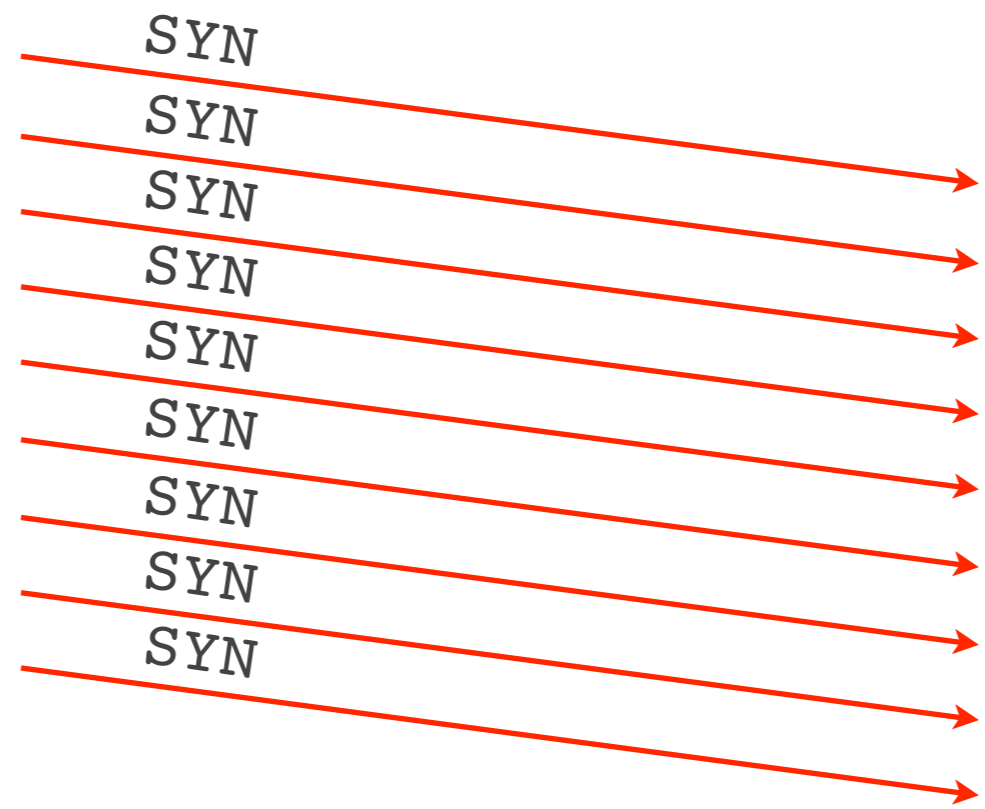
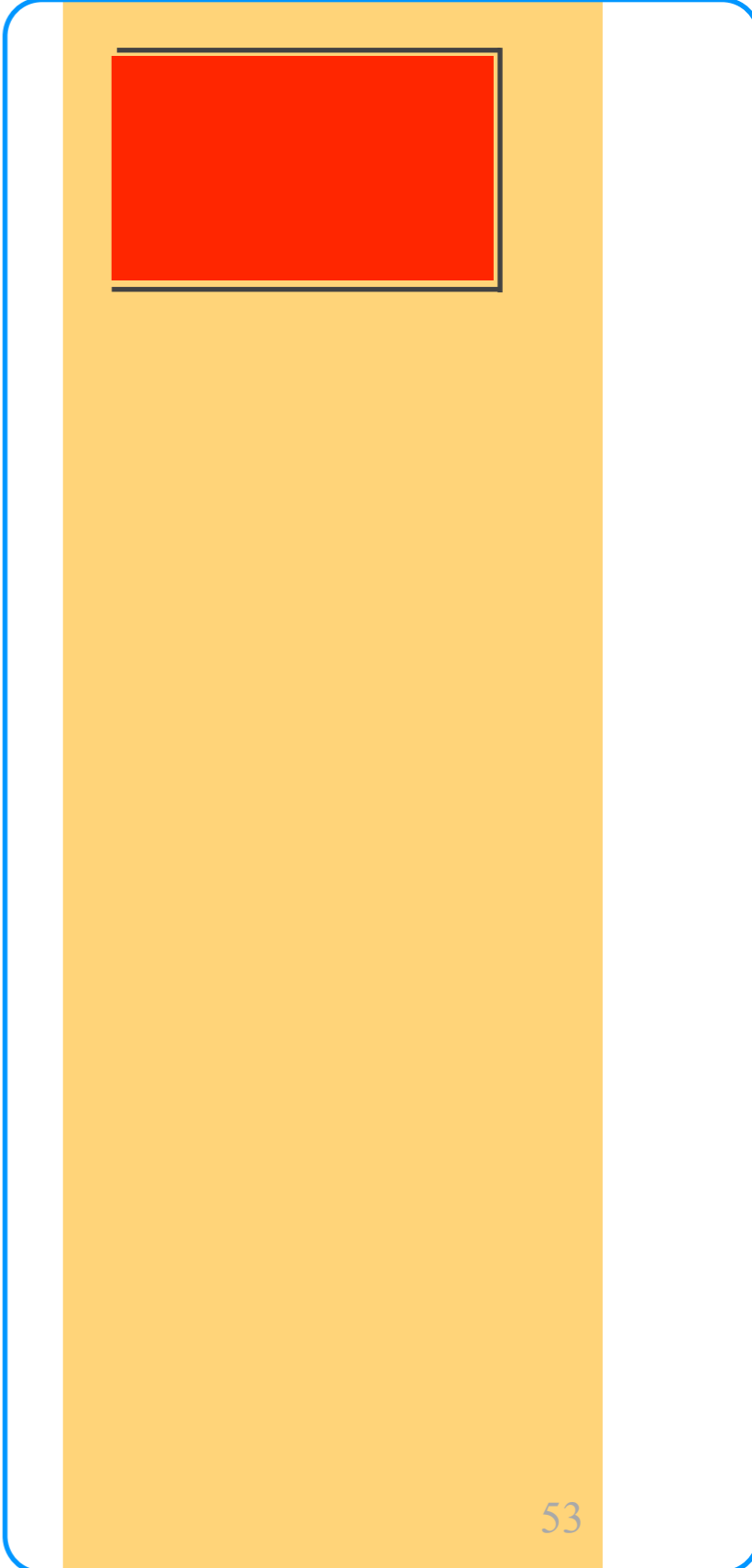
incomplete connections

connection established

Denis's computer



Bob's computer



Alice's computer

Bob's computer

SYN | SEQ x | ACK -

SYN | ACK $x+1$
SEQ $y = \text{hash}(\text{secret}, A \text{ IP})$

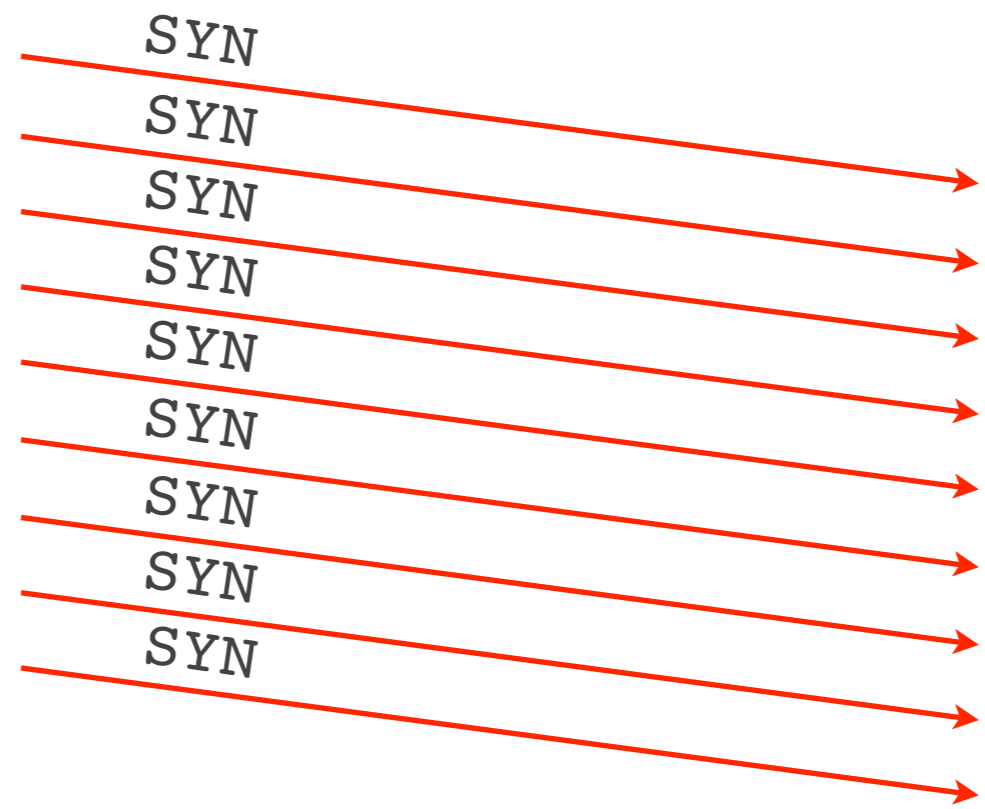
SEQ $x+1$ | ACK $y+1$ | [data]

incomplete
connections

connection
established

Denis's computer

Bob's computer



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

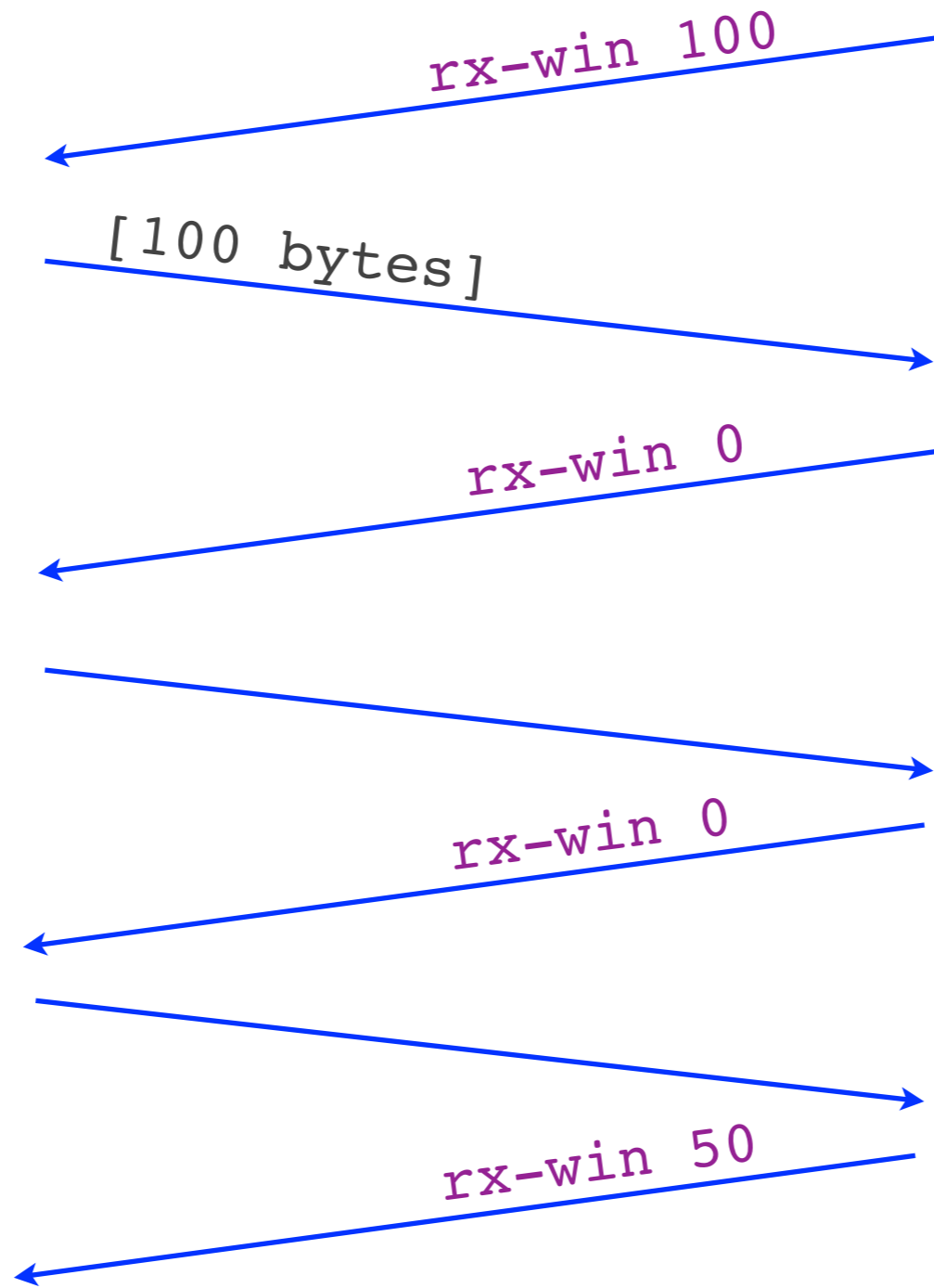
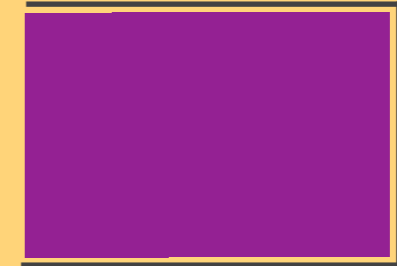
TCP elements

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

Alice's computer

Bob's computer

receive
buffer



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**