Computer Networks (2025-2026)

Part 3: Transport Layer

Jeroen Famaey

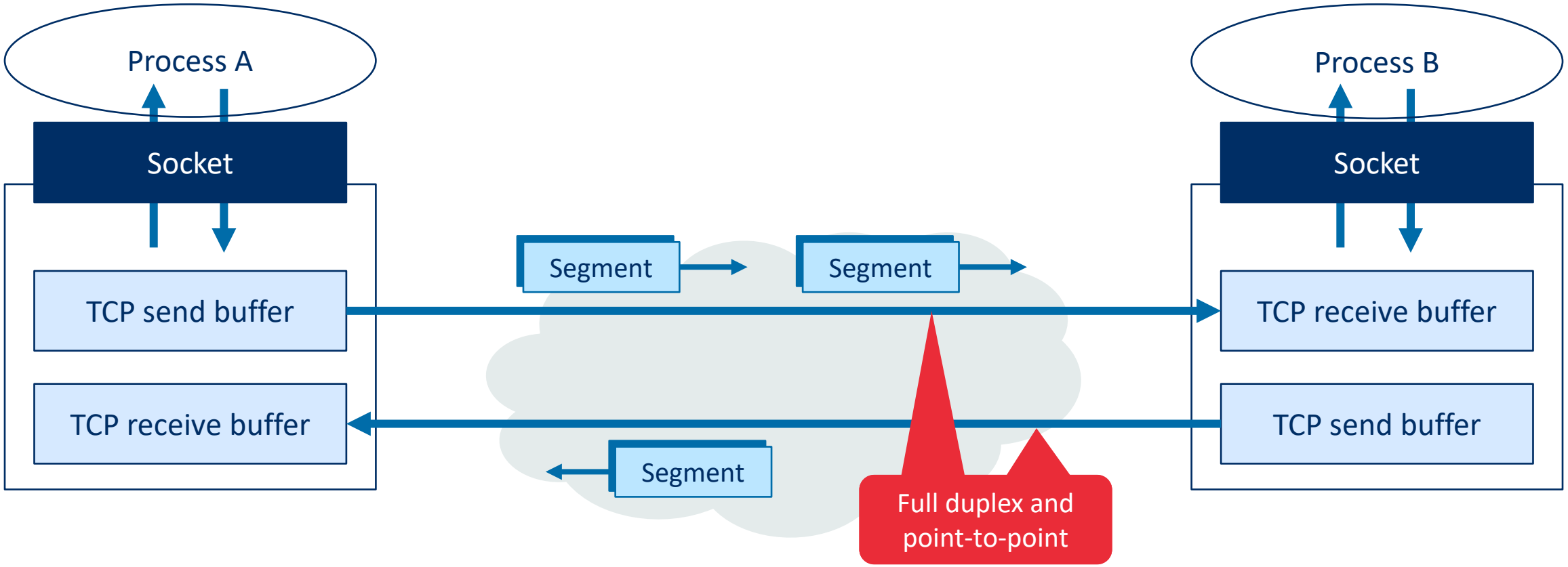
Andrei Belogaev

{jeroen.famaey, andrei.belogaev}@uantwerpen.be

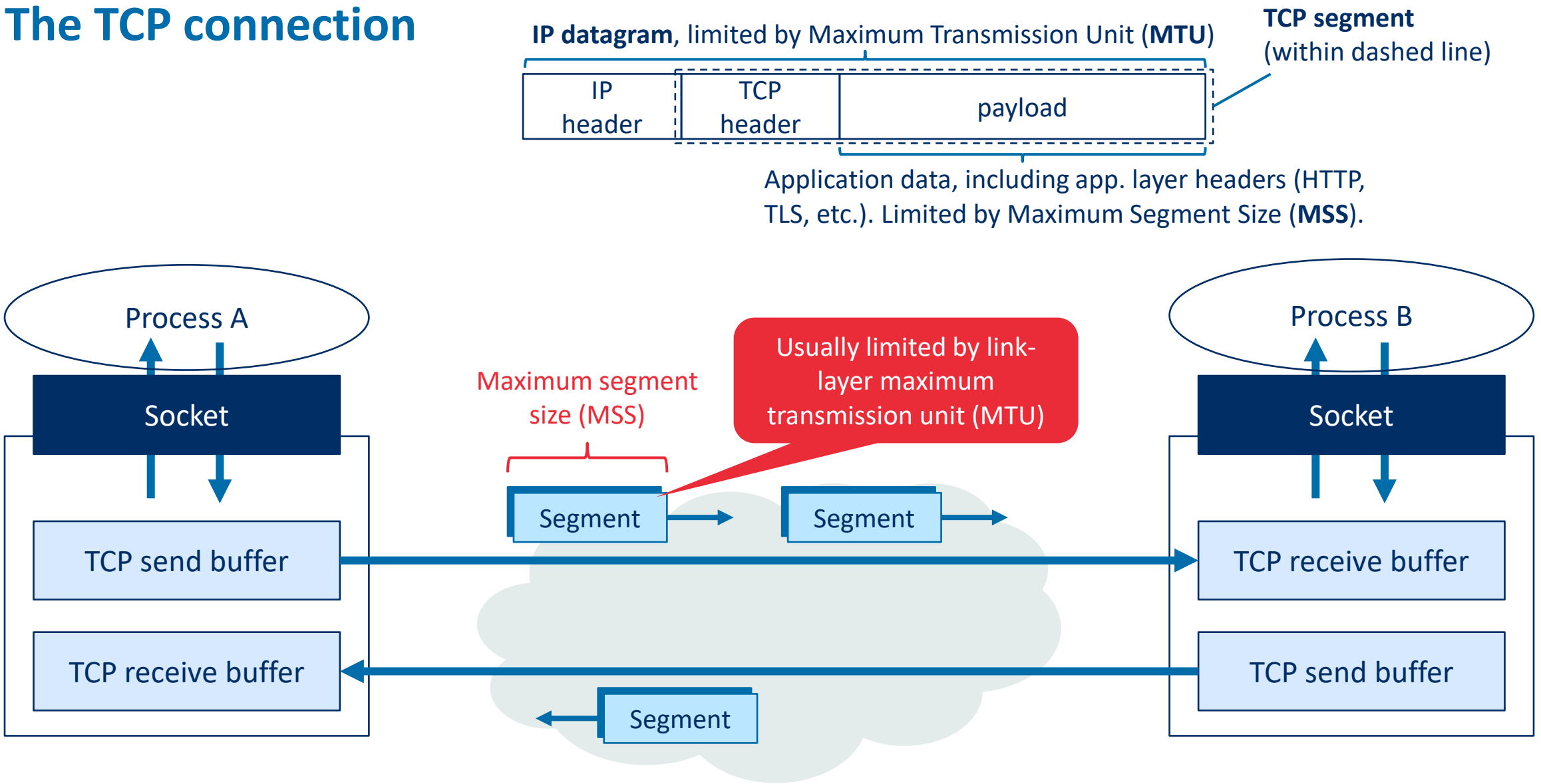
University of Antwerp
Faculty of Science

Transport Control Protocol (TCP)

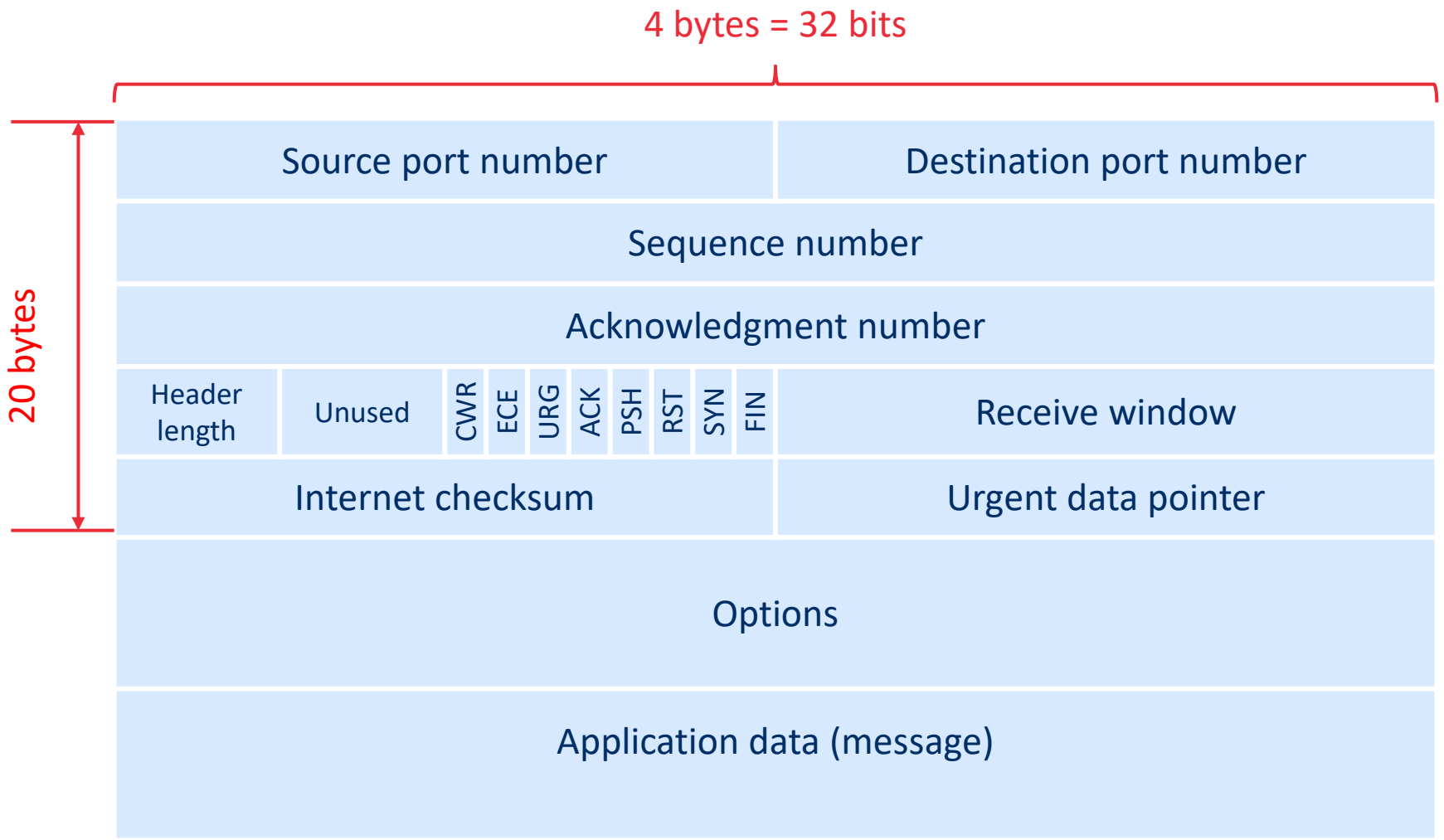
The TCP connection



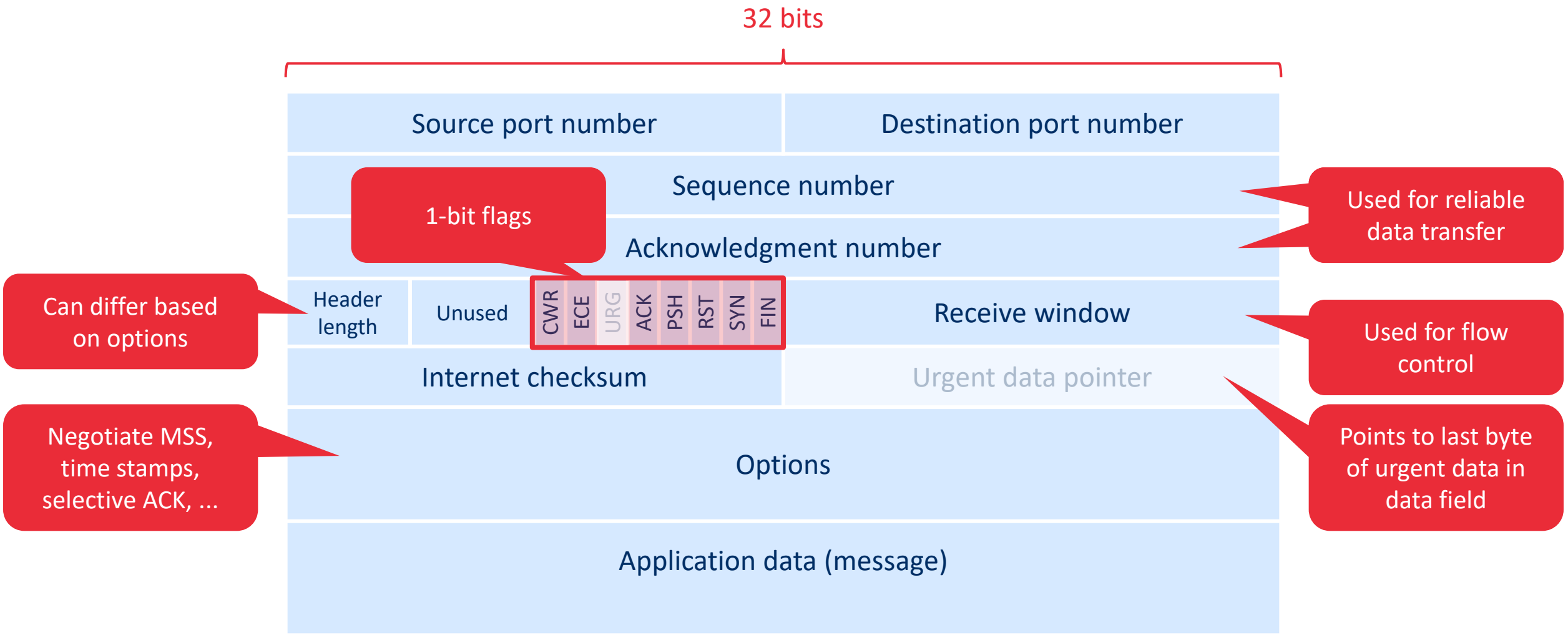
The TCP connection



TCP segment structure



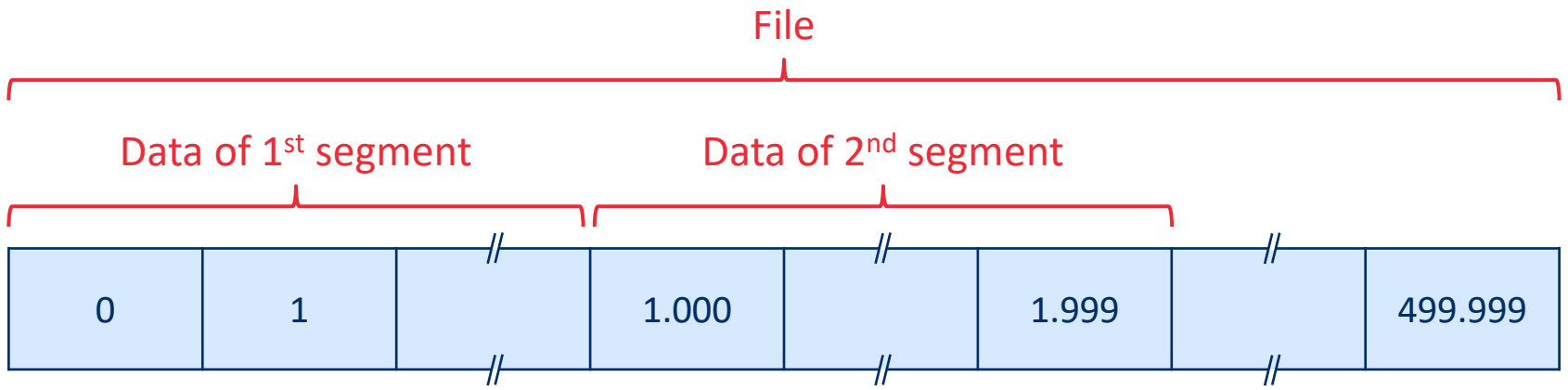
TCP segment structure



Sequence and acknowledgement numbers

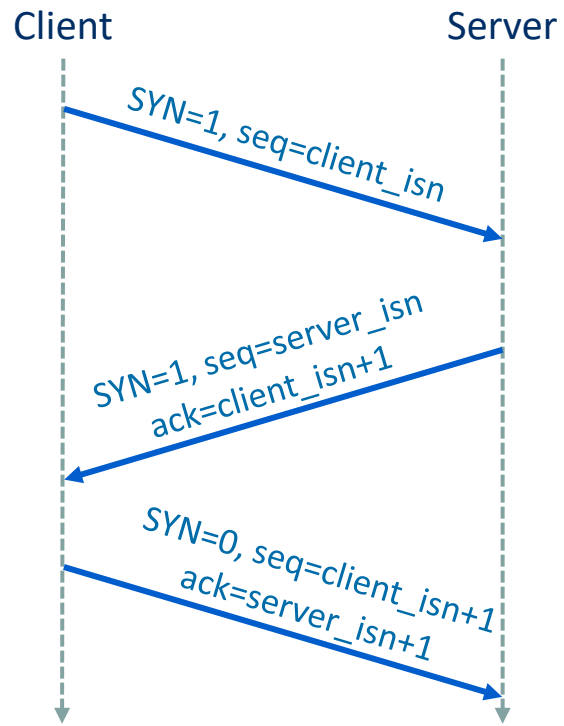
- TCP views data as an unstructured, but ordered, stream of bytes
- **Sequence number** of a segment refers to the position of its first byte in the stream and **NOT** the number of the segment itself
- **Acknowledgement number** is the next byte that receiver expects from the sender
  - TCP thus uses **cumulative acknowledgments**

- Example (MSS=1000 bytes, File = 500.000 bytes):

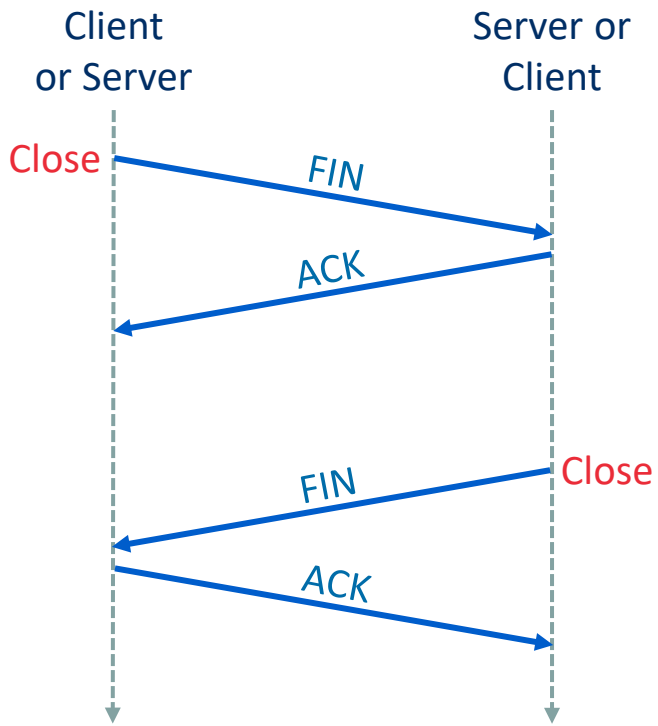


Connection management

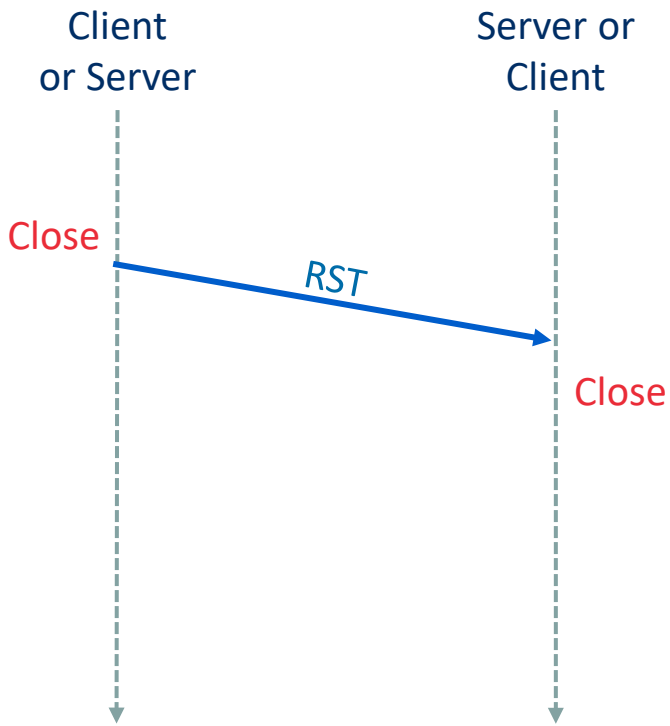
Connection establishment (three-way handshake)



Closing the TCP connection gracefully



Aborting the TCP connection



Estimating the round-trip time (RTT)

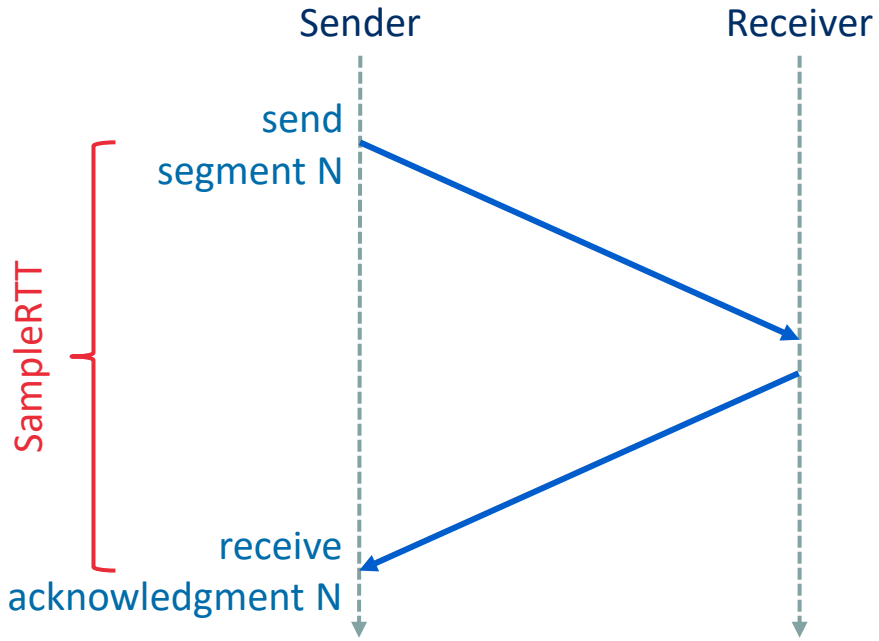
- The TCP retransmit timeout should be larger than the RTT
- How to estimate the RTT in a real network?
  - Measured as the time between sending a segment and receiving the corresponding acknowledgment (**SampleRTT**)
  - By default not calculated for segments that are retransmitted (an option exists to account for this, cf. next slide)
  - To reduce fluctuations, **EstimatedRTT** is an exponential weighted moving average (EWMA) of SampleRTT
  - The variability of the RTT is calculated as **DevRTT**

Recommendation: $\alpha = 0.125$

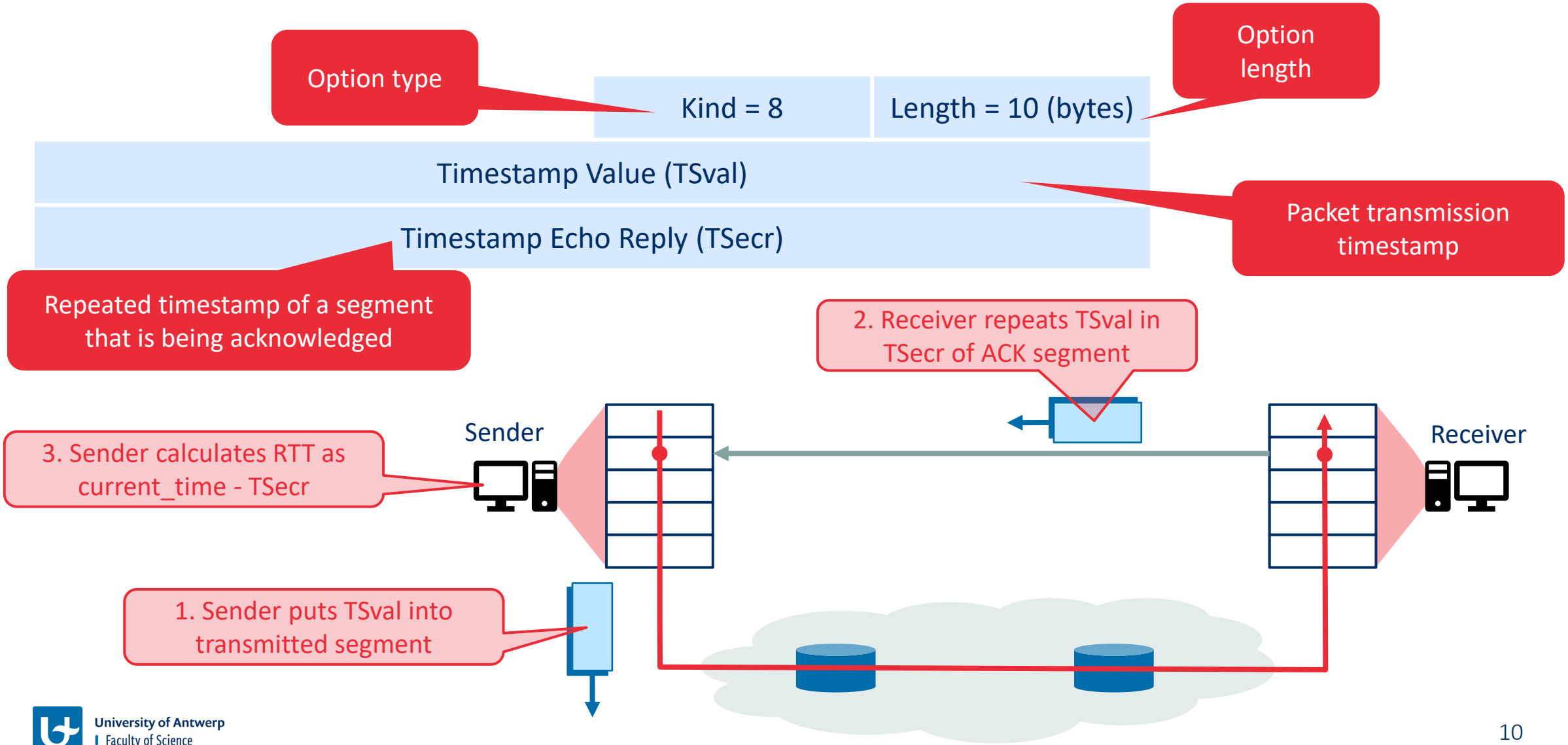
$$\text{EstimatedRTT} = (1 - \alpha) \cdot \text{EstimatedRTT} + \alpha \cdot \text{SampleRTT}$$

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

Recommendation: $\beta = 0.25$

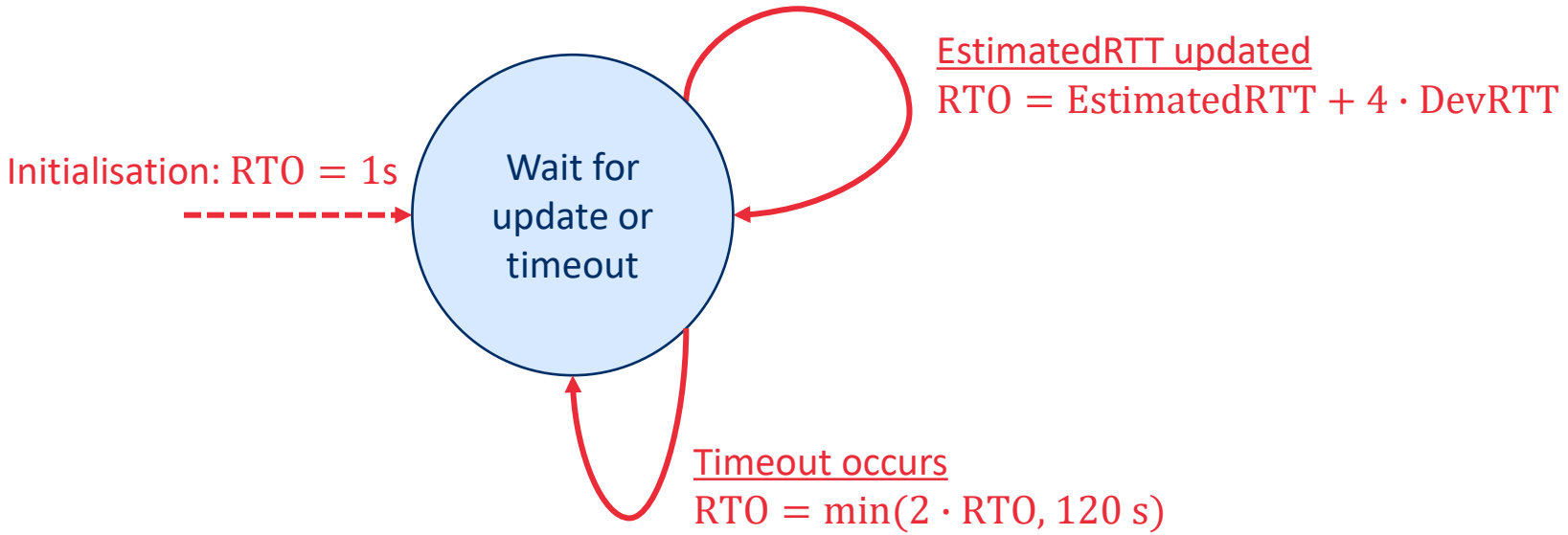


Estimating the round-trip time (RTT) using Timestamp Option

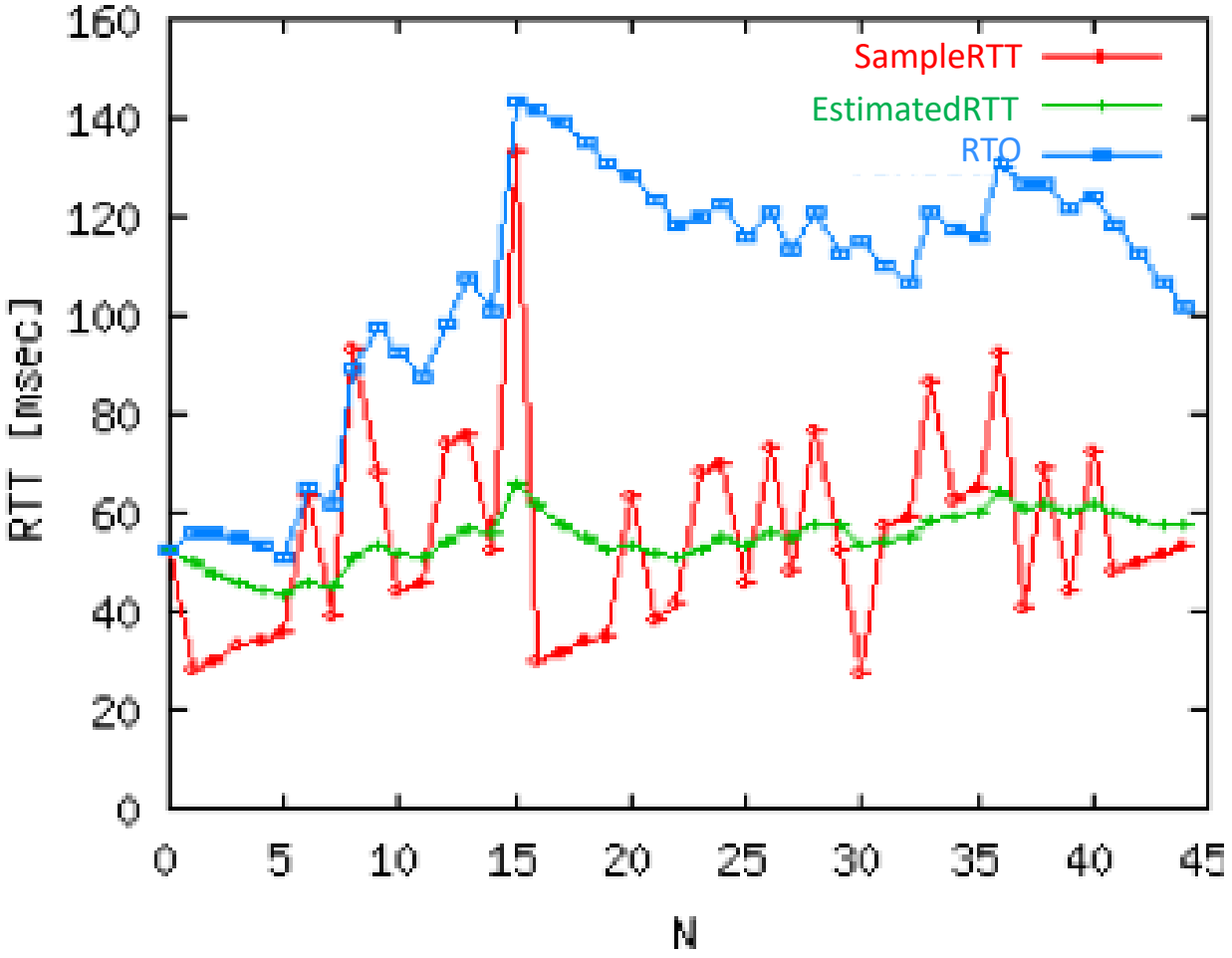


Managing the retransmission timeout (RTO)

- RTO should be close to but larger than EstimatedRTT
  - Margin should be larger if the variability (DevRTT) is large
  - An initial RTO = 1 second is recommended
  - When a timeout occurs, the RTO is immediately doubled, but recalculated when EstimatedRTT is next updated



Example of sampleRTT, estimated RTT, and RTO over time



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

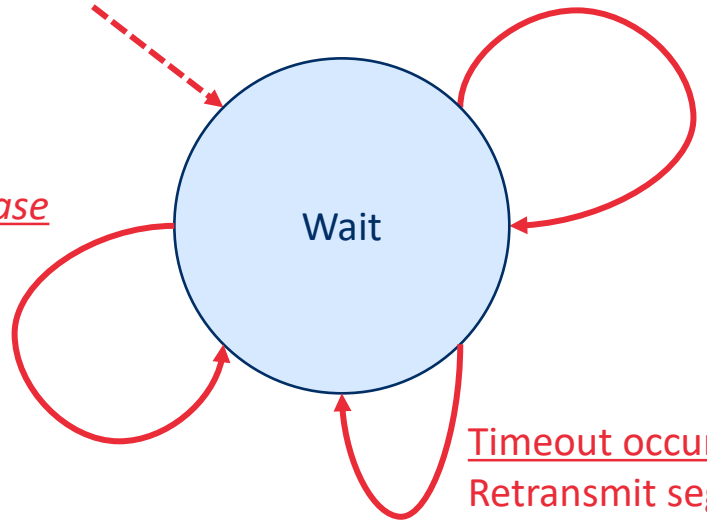
$$\text{DevRTT} = (1-\beta)\cdot \text{DevRTT} + \beta \cdot |\text{SampleRTT} - \text{EstimatedRTT}|$$

$$\text{RTO} = \min(\text{EstimatedRTT} + 4 \cdot \text{DevRTT}, 120s)$$

Retransmission timer management

- RFC 6298 recommends the TCP sender keeps a **single** retransmission timer
  - Associated with the oldest unacknowledged segment
  - Restarts when an acknowledgment for this segment is received
  - When the timer expires sender starts retransmission of the oldest unacknowledged segment

- Simplified TCP sender:



Data received from application above
Create TCP segment with sequence number *NextSeqNum*
Start retransmission timer (**if not running yet**)
Pass segment to IP
NextSeqNum = NextSeqNum + len(data)

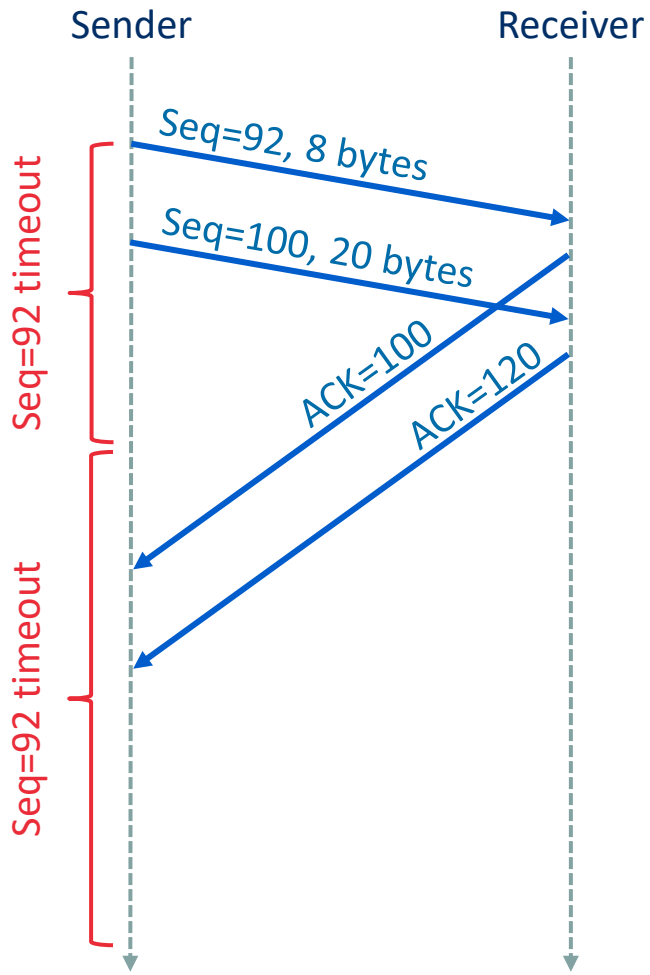
ACK(y) received, with y > *SendBase*
SendBase = y
restart timer (**if any not-yet-acknowledged data**)

Timeout occurs
Retransmit segment with sequence number *SendBase*
Double RTO value
restart timer

Exercise: Retransmission timeout in TCP

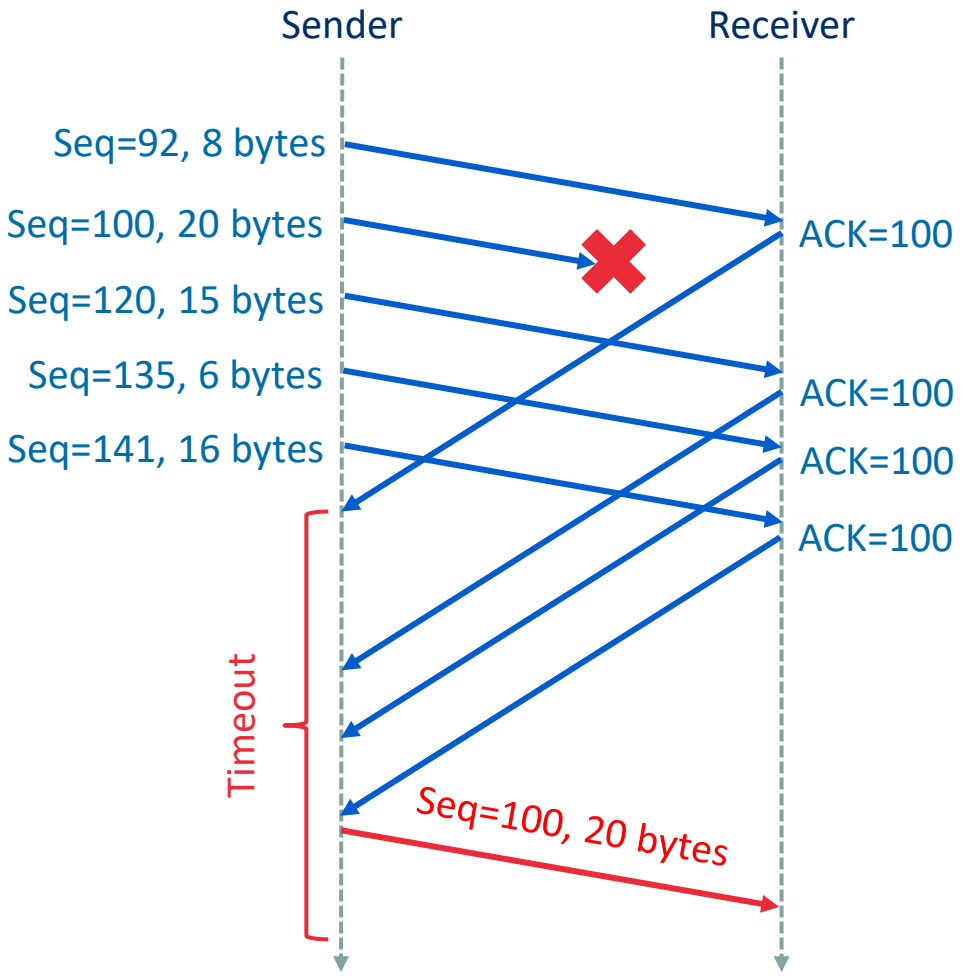
What happens after the first timeout of segment 92?

A. Only segment 92 will be resent
B. Segments 92 and 100 will be both be resent
C. Segment 92 will be resent, and 100 will only be resent after the next timeout
D. Neither segment will be resent

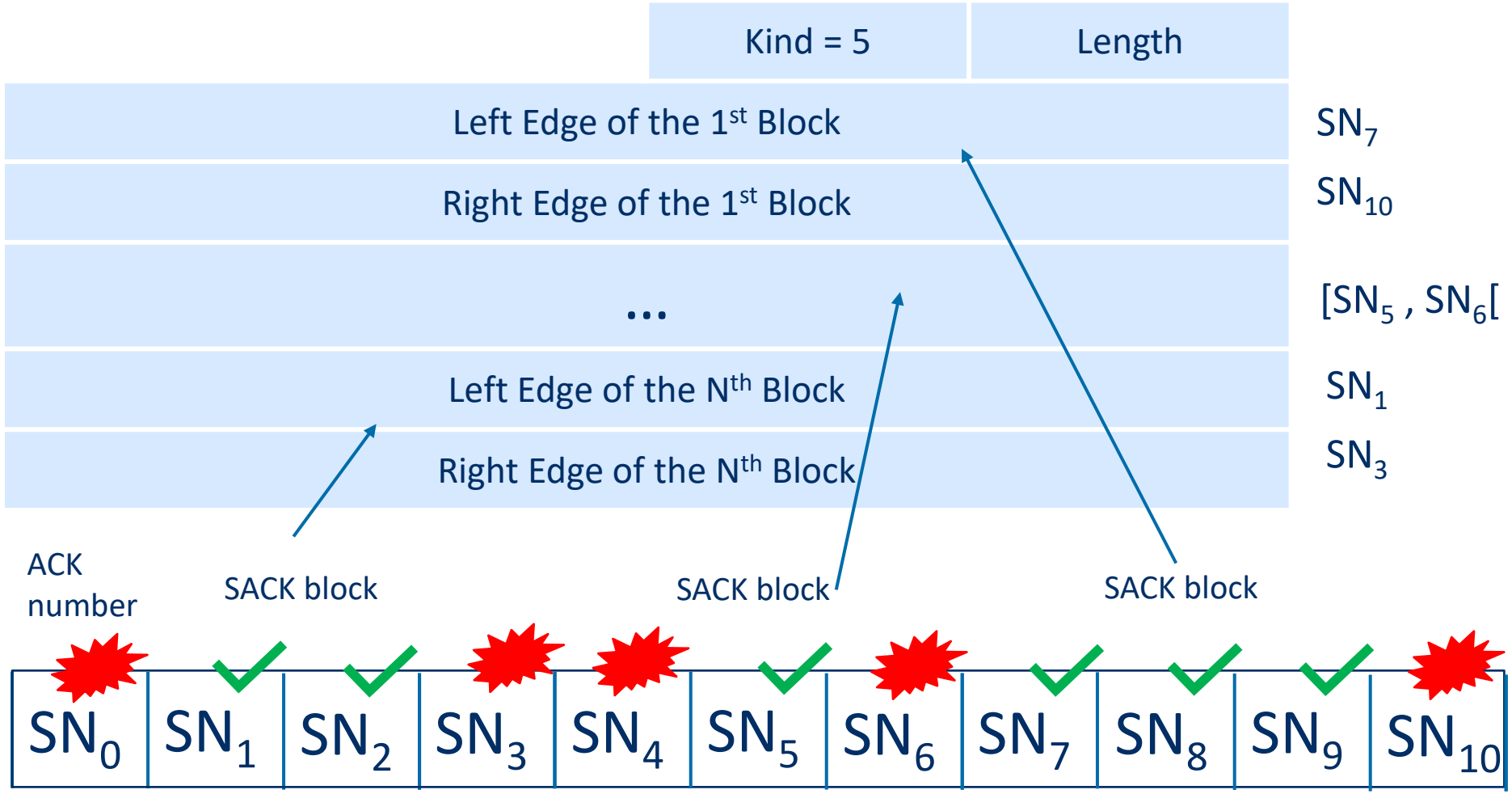


Fast retransmission

- The RTO value can be relatively long, especially for high-speed (e.g., Gigabit Ethernet) and high-RTT (e.g., satellites) networks.
- Lost packets can be detected earlier by counting **duplicate ACKs**, whose acknowledge number corresponds to already acknowledged segment.
- After **3 duplicate ACKs**, the oldest non-acknowledged segment is retransmitted.
- If a retransmission timeout occurs, it is an indication of significant problems or changes in the network.



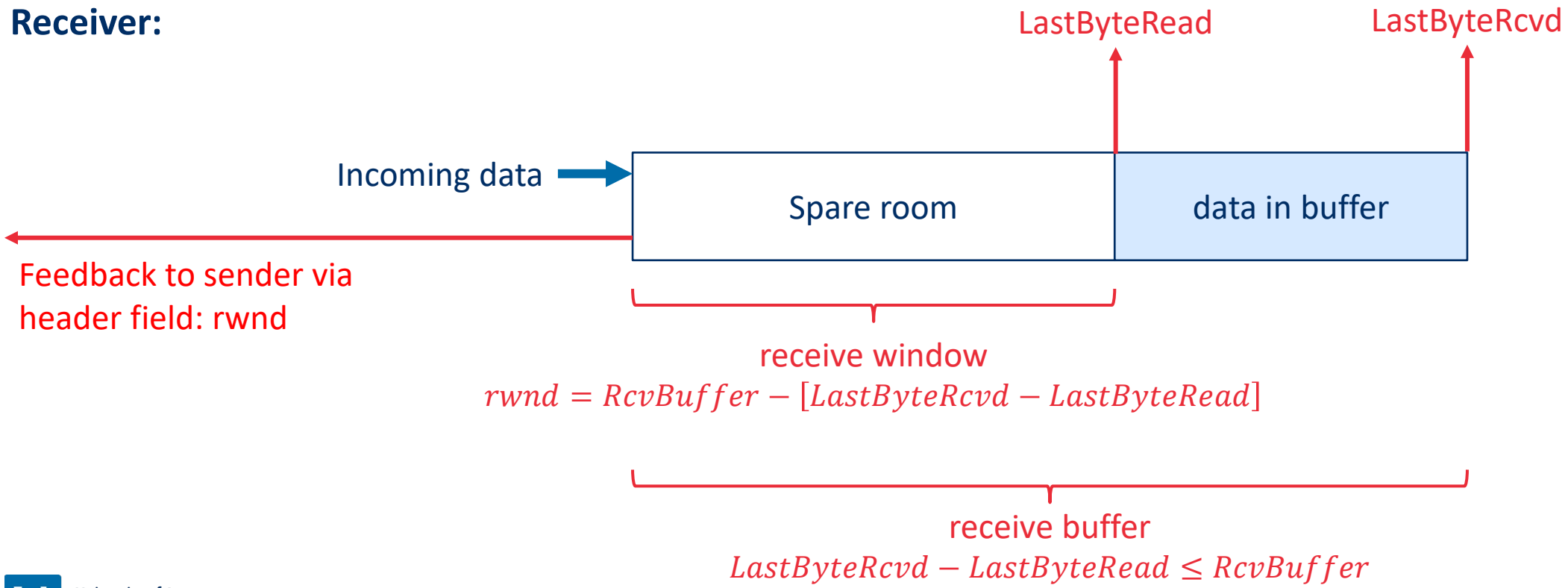
TCP Selective Acknowledgement (SACK) option



Flow control – Receiver

Ensures that the sender does not send data faster than the receiver application reads it

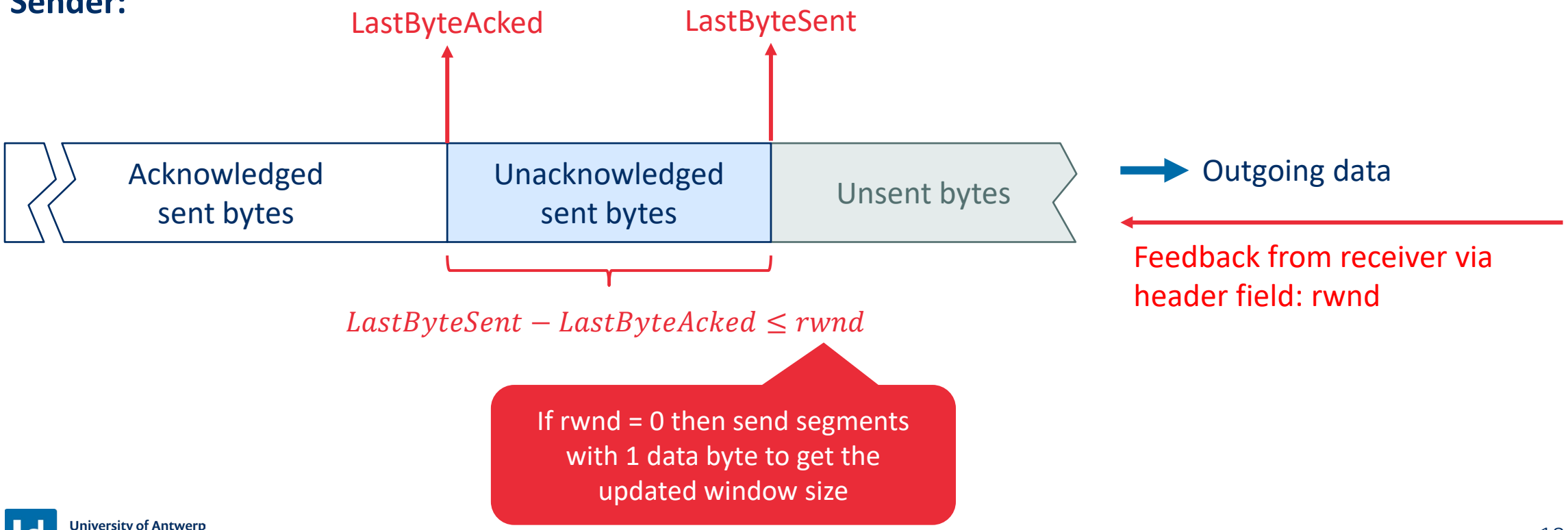
Receiver:



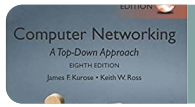
Flow control – Sender

Ensures that the sender does not send data faster than the receiver application reads it

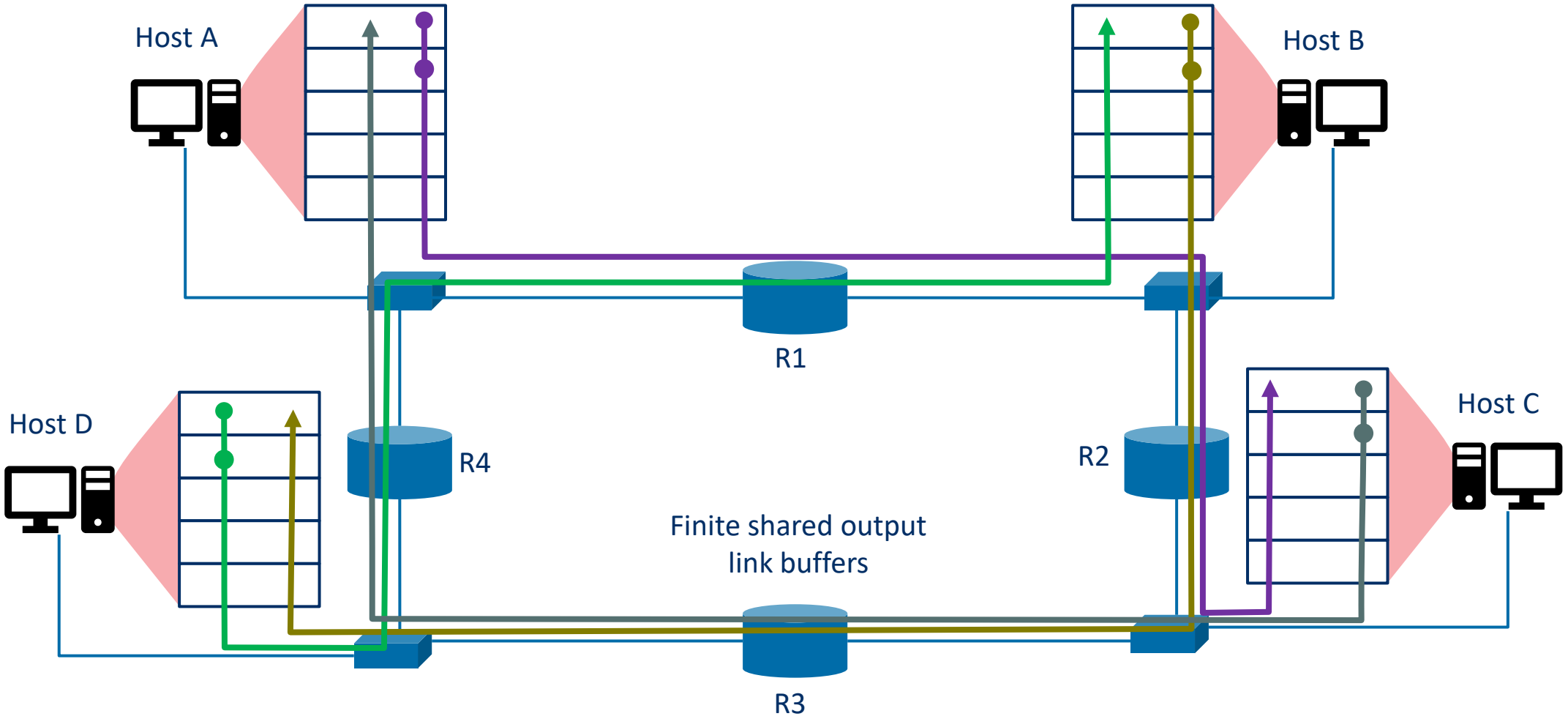
Sender:



TCP congestion control



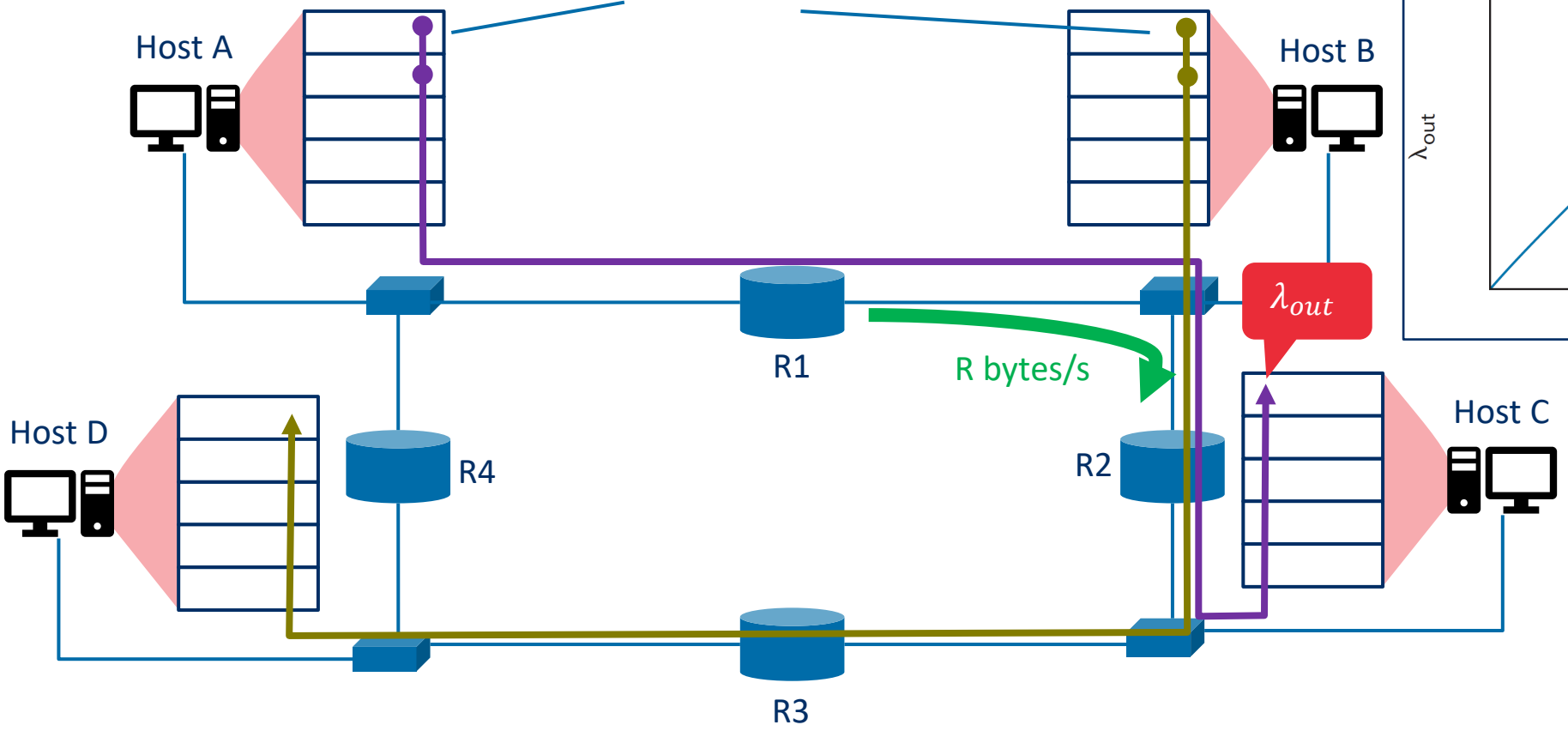
Understanding congestion

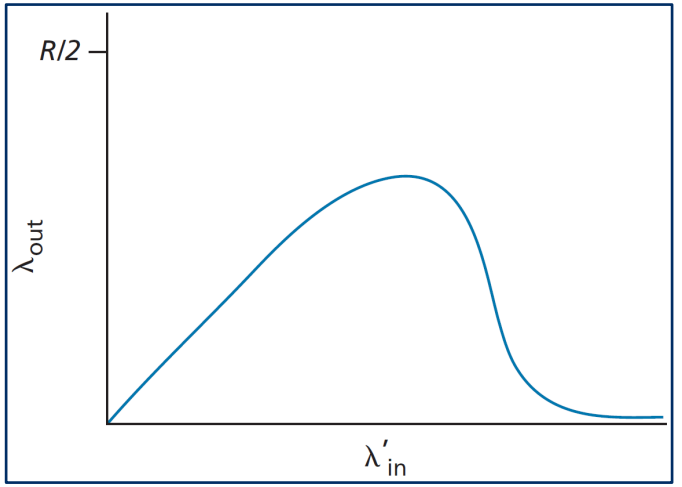


Understanding congestion

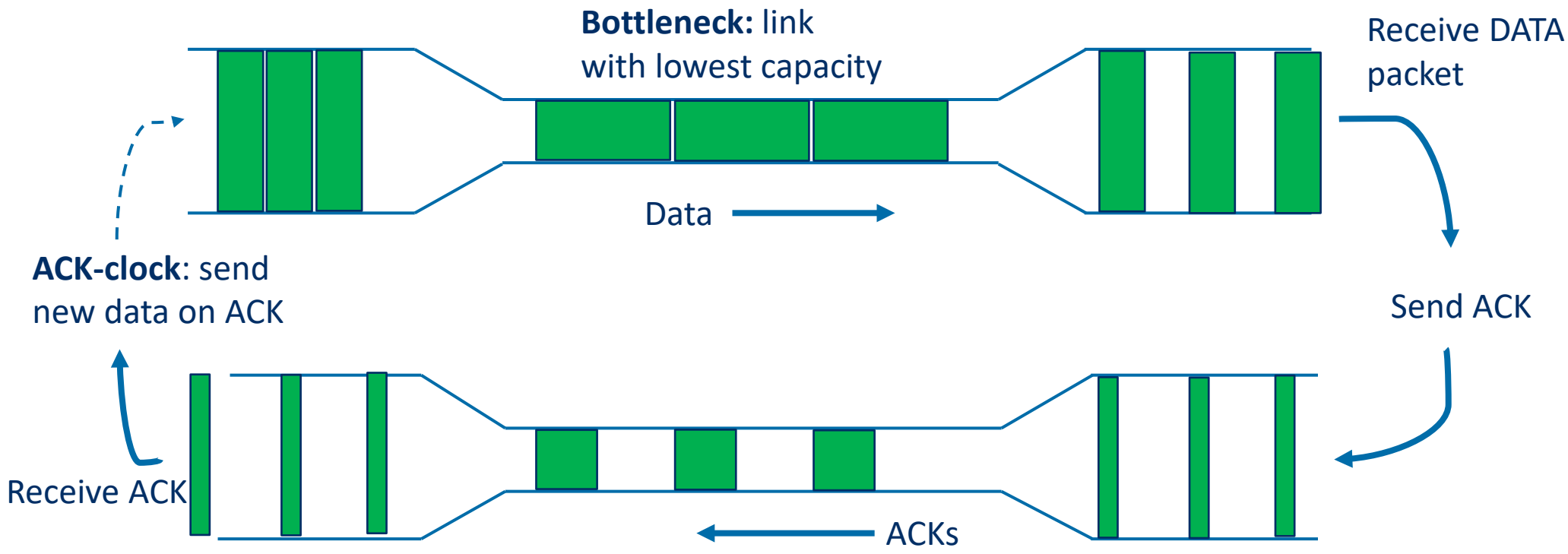
λ_{in} bytes/s: original data

λ'_{in} bytes/s: original & retransmitted data

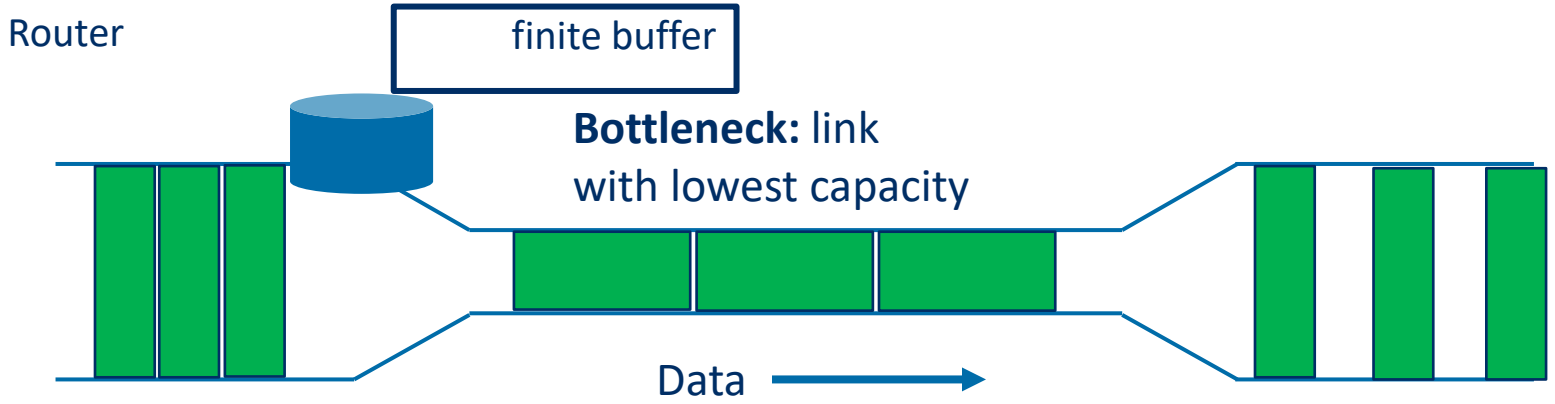




Understanding congestion: bottleneck



Optimal pipelining for single data flow



Optimal amount of data “in flight”, i.e., sent but not acknowledged, is determined by:

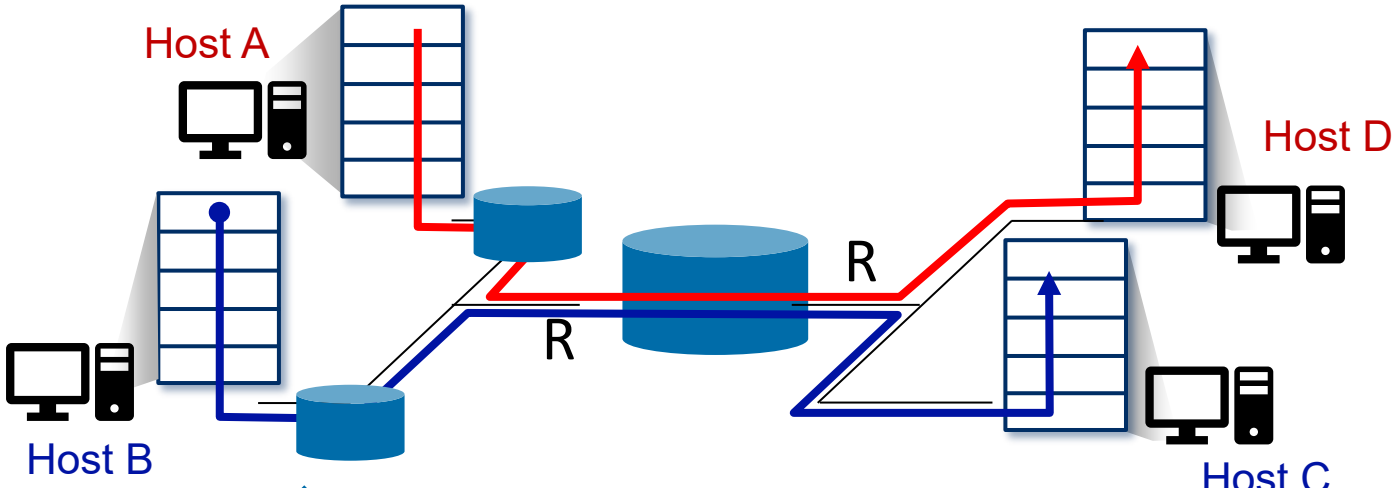
- Bottleneck link capacity C bytes/s;
- Round-trip time RTT s.

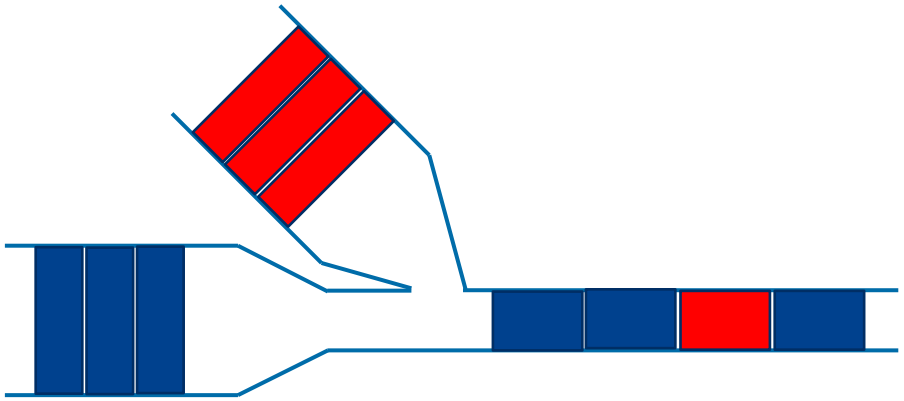
To fully fill the bottleneck link with packets “back-to-back” **without queueing,** the sender should pipeline $C \cdot RTT$ bytes. This number is called **bandwidth-delay product (BDP).**

When the sender starts transmitting more than BDP data, the queueing will start happening. When the router’s buffer will become full, this router will start dropping packets.*

*In practice, it will start dropping packets earlier to avoid bursty drops.

Fairness for multiple data flows





Multiple connections sharing the same bottleneck link should use this link:

- **efficiently**, i.e., with maximum utilization;
- **fairly**, i.e., take equal shares of its capacity.

Summary: goals of congestion control

- Throttle (i.e., decrease transmission rate) senders in the face of network congestion
- Efficiently utilize the capacity of the bottleneck to maximize throughput
- Fairly share the bottleneck link between multiple connections

Two main approaches to congestion control

End-to-end congestion control

- Presence of congestion is inferred by end systems based on observed network behaviour
- Approach used by TCP
  - TCP packet loss is used to identify congestion*
  - TCP decreases window size accordingly

*In practice, packet loss can be caused not only by network congestion, but due to different other reasons. Notably, in wireless networks (such as Wi-Fi) many packet losses happen due to high variability of the wireless medium and interference.

Network-assisted congestion control

- Routers provide explicit feedback to the sender and/or receiver regarding congestion state
- Can be a single bit or more detailed feedback
- Revisited in relatively new standard L4S (RFC 9330) actively pushed by Nokia.
- Used by IBM SNA, DECnet, ATM, ...
- Supported by TCP, but many “middleboxes” (routers, firewalls, etc.) do not support it.

TCP congestion control

- TCP limits the rate at which it sends traffic as a function of perceived network congestion
- This approach raises several questions
  - How does a TCP sender limit its sending rate?
  - How does a TCP sender perceive that there is congestion?
  - What algorithm should the sender use to adapt its send rate as a function of congestion?
- Consists of 3 parts
  - Slow start
  - Congestion avoidance
  - Fast retransmit & fast recovery

Sender uses congestion window (cwnd)

$$LastByteSent - LastByteAcked \leq \min(cwnd, rwnd)$$

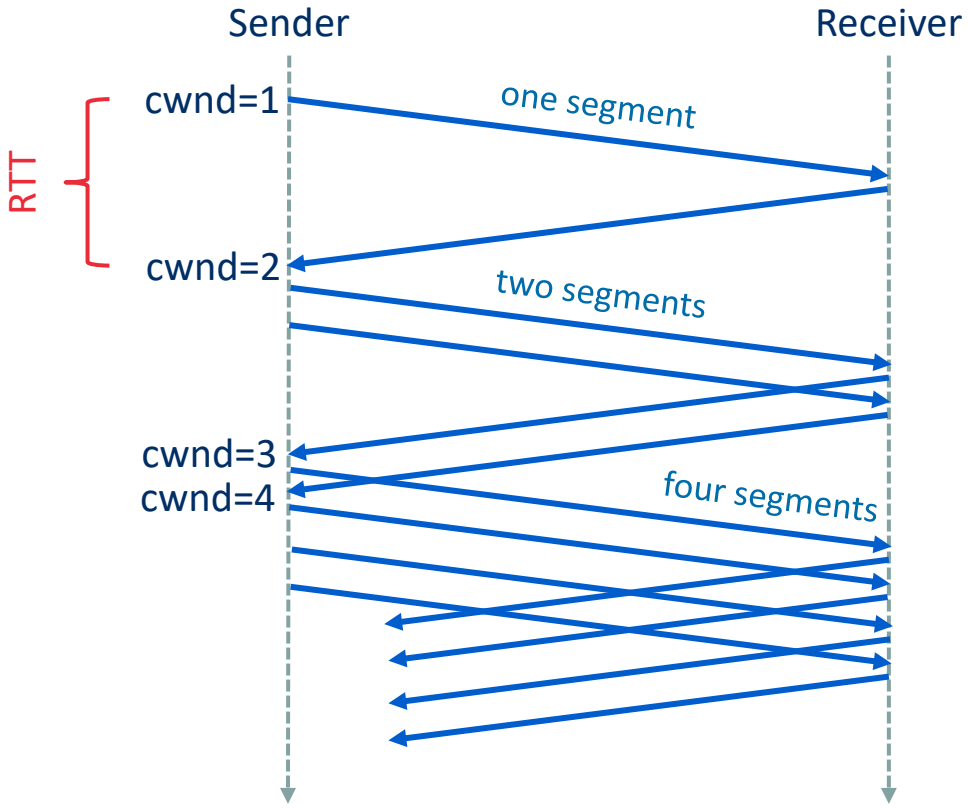
Timeout (severe problem) or three duplicate ACKs (congestion just started)

Decrease cwnd when segments are lost. Decrease stronger in case of timeout.

Increase cwnd when ACKs are received

State 1: Slow start

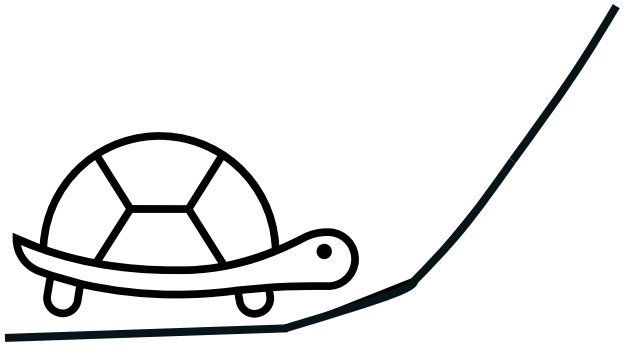
- *cwnd* is initialized to $init\_cwnd$ MSS and increased by 1 MSS on every acknowledged MSS
  - "vanilla" slow start: $init\_cwnd = 1$ MSS
  - RFC 5681: $init\_cwnd \sim 4000$ bytes (3 MSS for MSS = 1460 bytes)
  - RFC 6928 (Linux default): $init\_cwnd = 10$ MSS
- When $cwnd \geq ssthresh$ (slow start threshold), TCP transitions into **congestion avoidance mode**
- Initial value of ssthresh is Infinity (= no threshold)



How long it will take to reach 1 Gbps speed?

$$\frac{(500 \cdot 8)bits \cdot 10\ MSS \cdot 2^{N_{RTT}}}{200 \cdot 10^{-3}s} = 10^9\ bits/s$$

$$N_{RTT} = \log_2 \frac{10^6 \cdot 200}{40000} = \log_2 5000 \approx 12.29\ RTTs$$



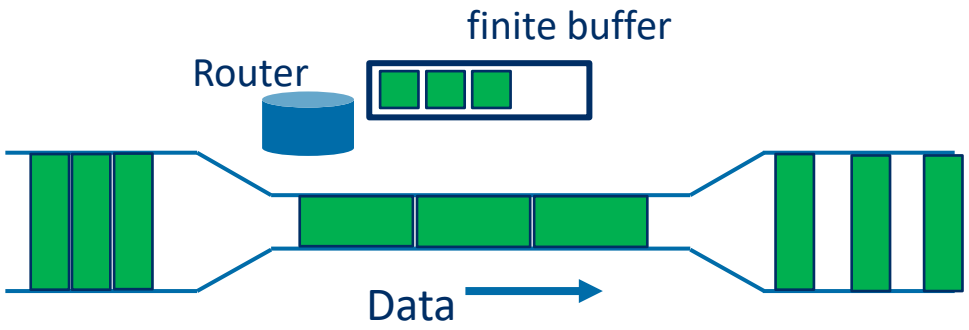
It will take $\sim 12.29 \cdot 200 \cdot 10^{-3}s \approx 2.5\ s$ to reach 1 Gbps. This is a really **slow** start.

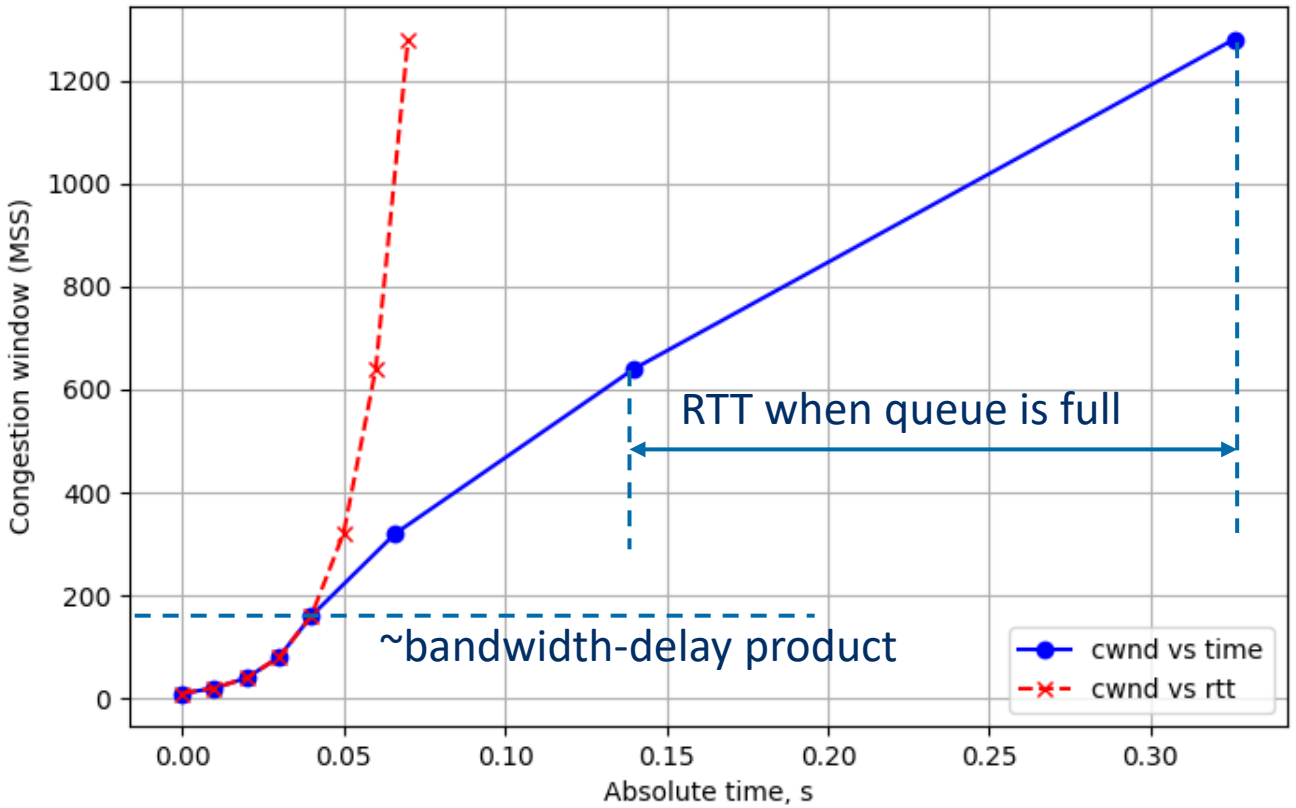
Typically, it is faster: MSS = 1460 bytes, Internet RTT ~20-50 ms. This results in 7-8 RTTs.

State 1: Slow start

cwnd is increased twice in every RTT.

However, in absolute time scale, RTT starts increasing when *cwnd* reaches bandwidth-delay product.

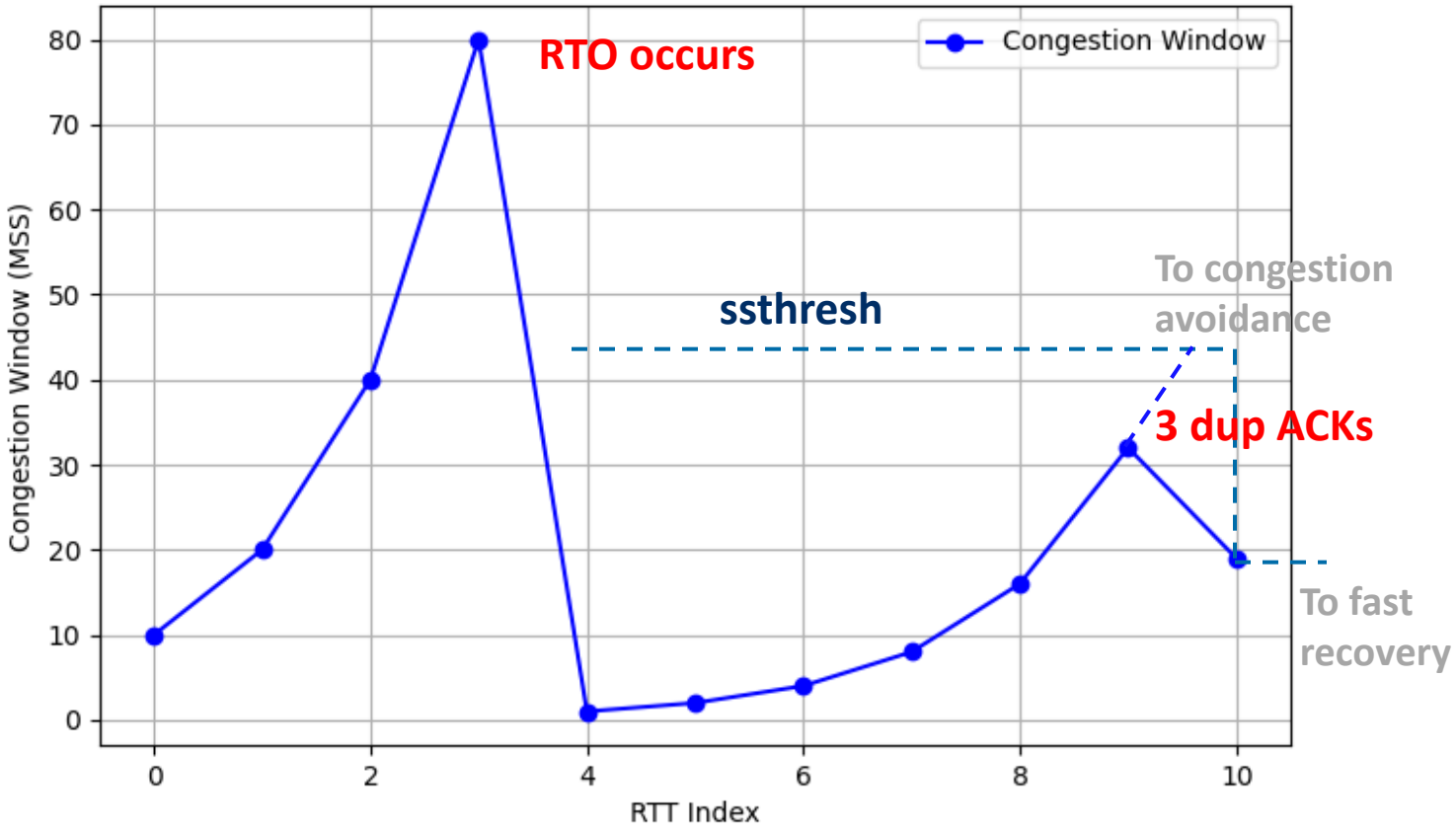




State 1: Slow start

- When a **timeout** occurs
  - ssthresh is set to cwnd / 2
  - cwnd is reset to 1 MSS

- When 3 duplicate ACKs are received TCP transitions into **fast recovery** state
  - ssthresh is set to cwnd / 2
  - cwnd is set to (cwnd / 2 + 3 MSS)

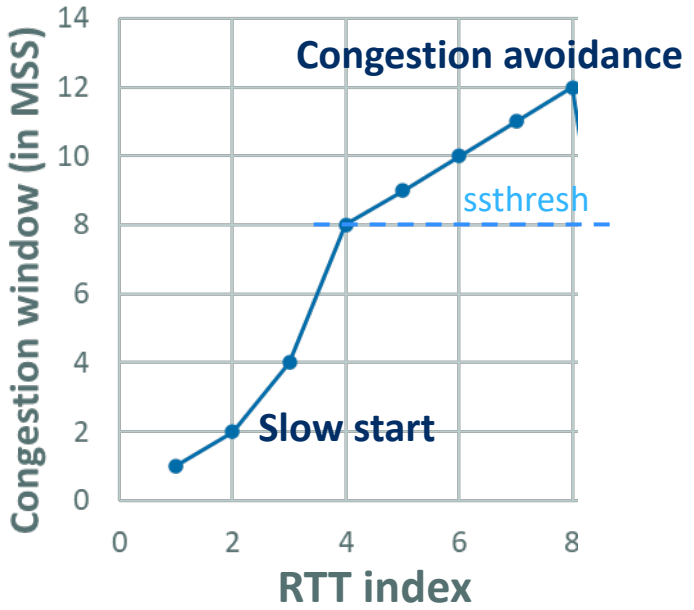


State 2: Congestion avoidance [TCP Reno, Additive Increase Multiplicative Decrease (AIMD)]

- Starts when cwnd reaches *ssthresh* (half of cwnd value when congestion was last encountered)

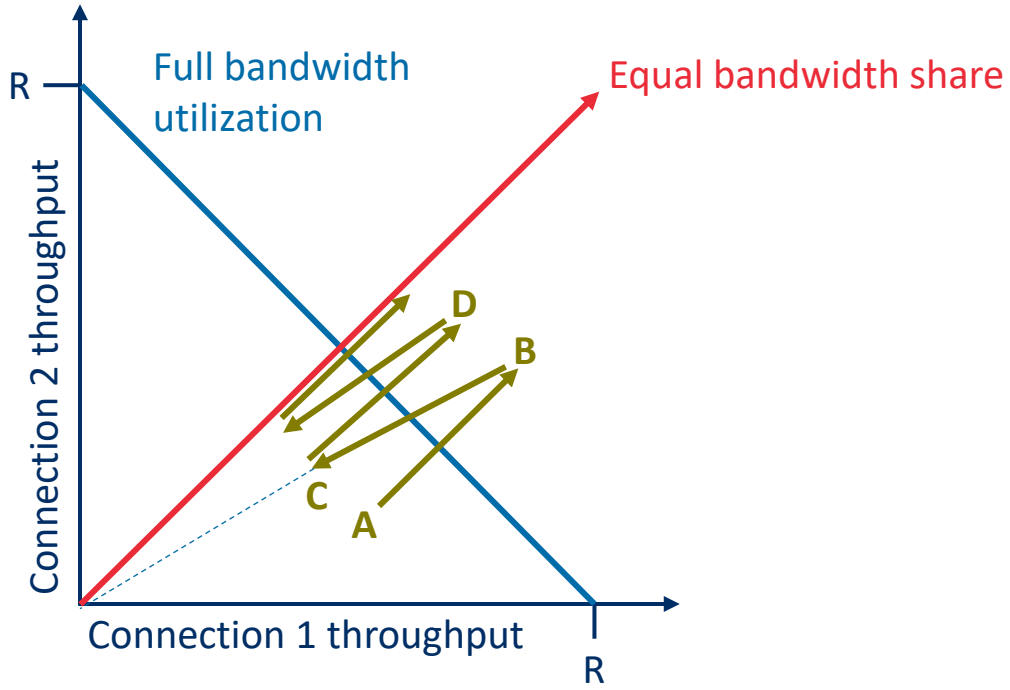
- TCP only increases cwnd by 1 MSS every RTT (instead of doubling it as in slow start mode)
  - Common approach: When new ACK arrives, $cwnd = cwnd + 1/cwnd$. The idea is to increase $cwnd$ by 1 after $cwnd$ ACKs are received (i.e., after ~1 RTT).
  - If $cwnd$ is measured in bytes, the formula transforms into $cwnd\_bytes = cwnd\_bytes + MSS^2/cwnd\_bytes$

- TCP goes to:
  - slow start state when a timeout occurs ($ssthresh = \frac{cwnd}{2};\ cwnd = 1$);
  - fast recovery after 3 duplicate ACKs ($ssthresh = \frac{cwnd}{2};\ cwnd = ssthresh + 3\ MSS$).



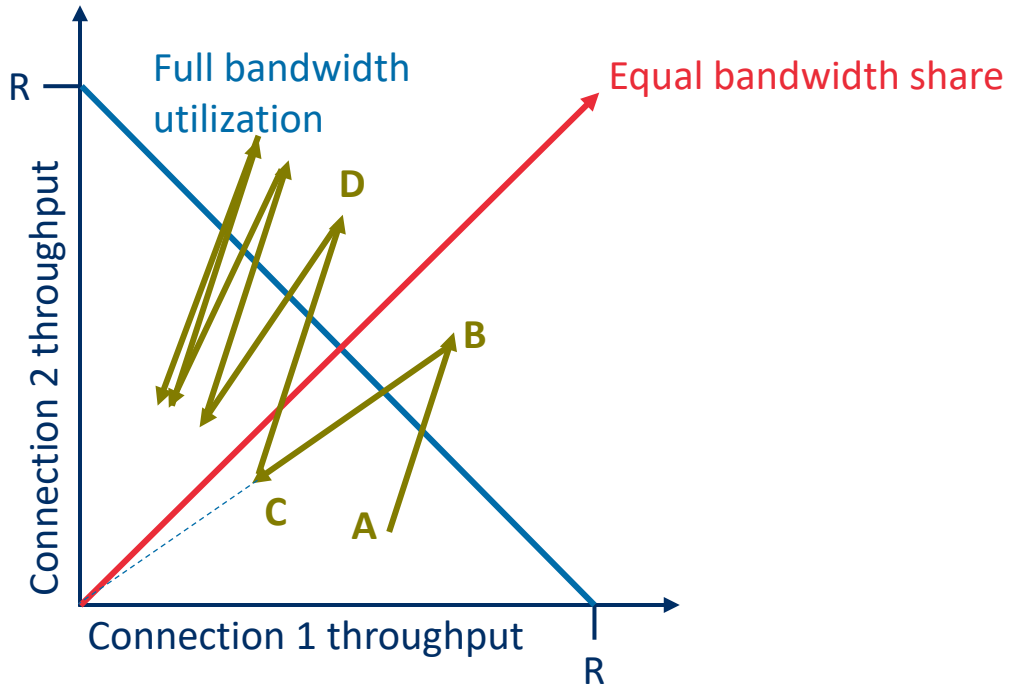
Fairness of TCP Reno's AIMD approach

- A congestion control mechanism is **fair** if K connections sharing a bottleneck link with transmission rate R bps, each achieve an average transmission rate of approximately R/K.
- The TCP Reno's AIMD algorithm has been shown to be fair in certain cases.
- Let's consider an example of a link with transmission rate R shared by 2 TCP connections **with equal RTTs**

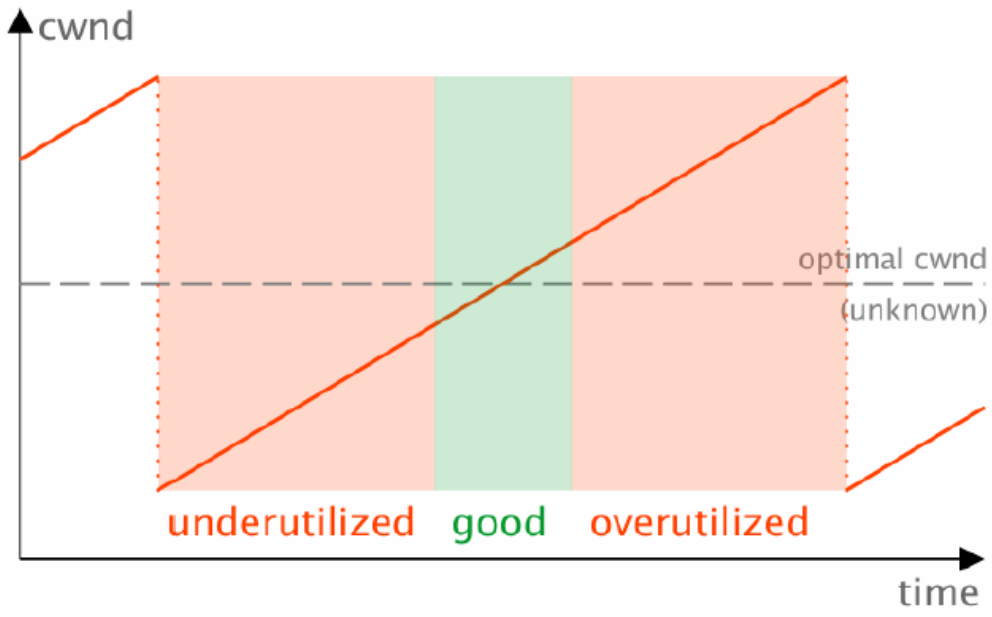


RTT unfairness of TCP Reno's AIMD approach

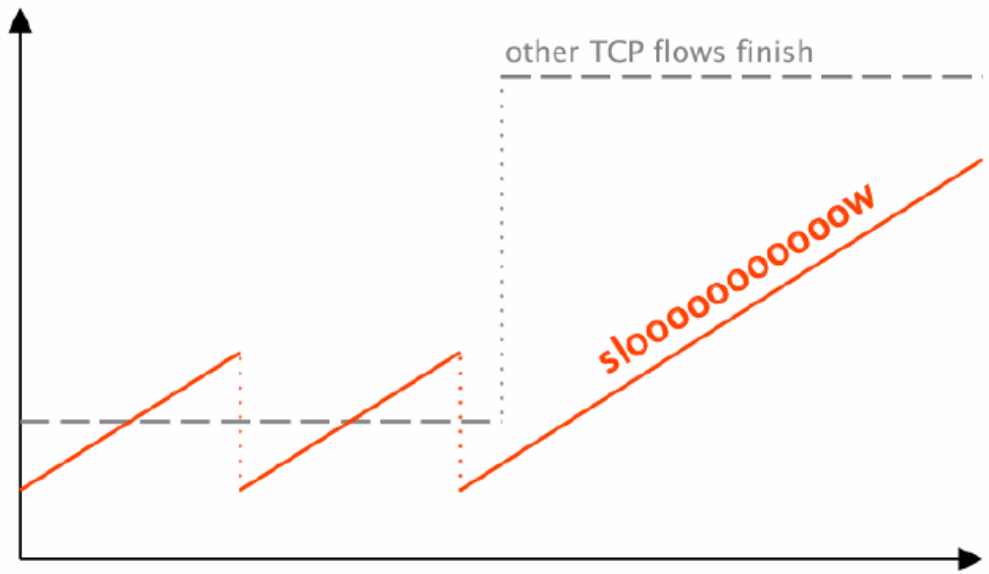
- A congestion control mechanism is **fair** if K connections sharing a bottleneck link with transmission rate R bps, each achieve an average transmission rate of approximately R/K.
- The TCP Reno's AIMD algorithm relies on ACK clocks. The connection with smaller RTT will increase CWND faster. Besides, throughput formula (approx. cwnd/RTT) has RTT in denominator.
- Let's consider an example of a link with transmission rate R shared by 2 TCP connections **with non-equal RTTs** ($RTT_1 > RTT_2$).



TCP Reno is inefficient for fast networks



It is not efficient



It is not fast enough

(from materials of ETH course “Advanced Topics in Communication Networks”, Fall-2025)

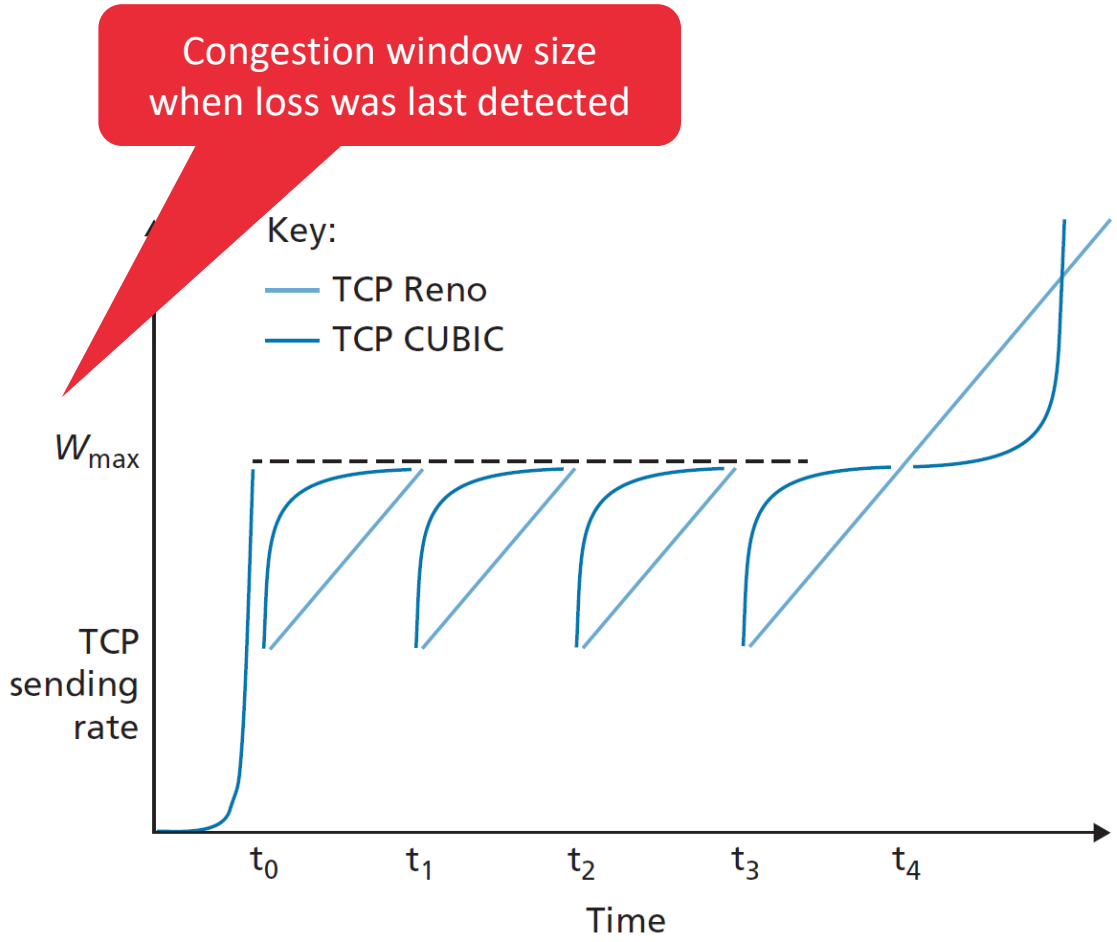
State 2: Congestion avoidance [TCP CUBIC, default in many OS]

TCP Reno

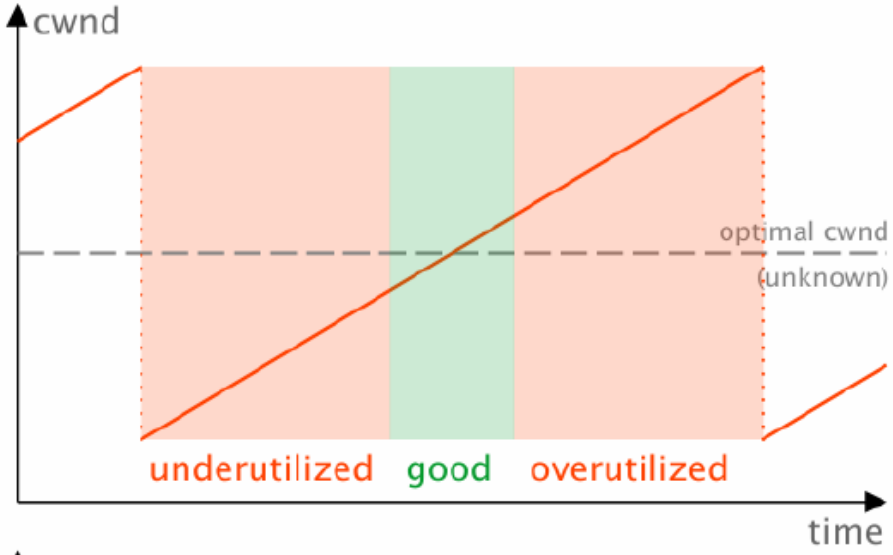
- Additive-increase, multiplicative-decrease (AIMD)
- Slowly increases cwnd after packet loss
- Window increases by $\frac{MSS^2}{cwnd}$ bytes on every ACK

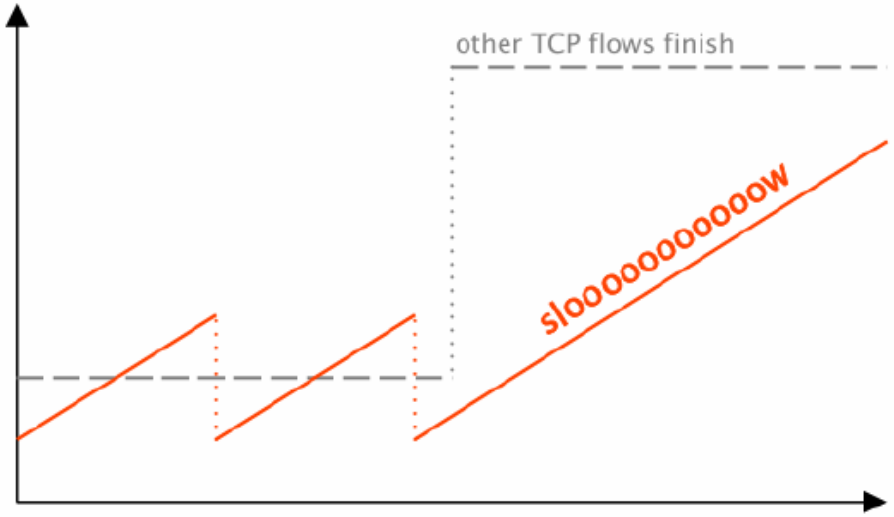
TCP CUBIC

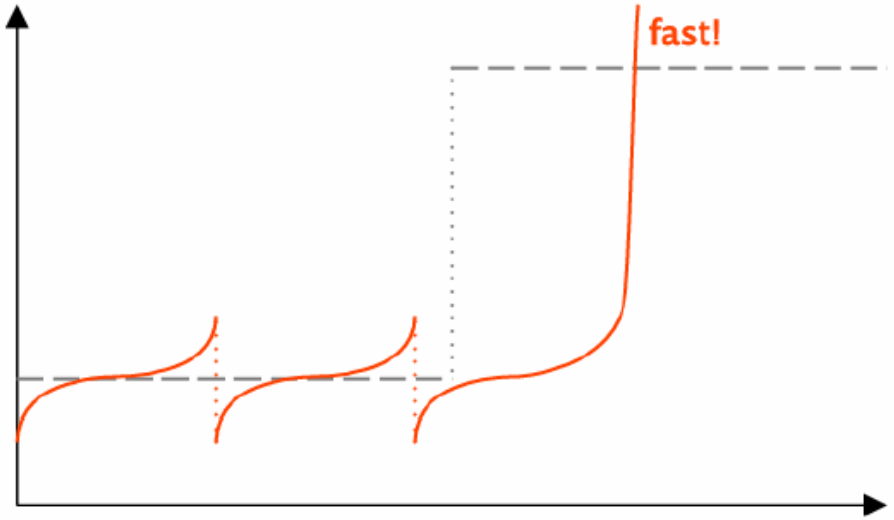
- Quickly ramps up sending rate to get close to pre-loss sending rate
- Probes cautiously afterwards
- Window increase function depends on the time elapsed since congestion event, not on ACKs arrival rate => better RTT-fairness.



TCP Cubic is more efficient for fast networks



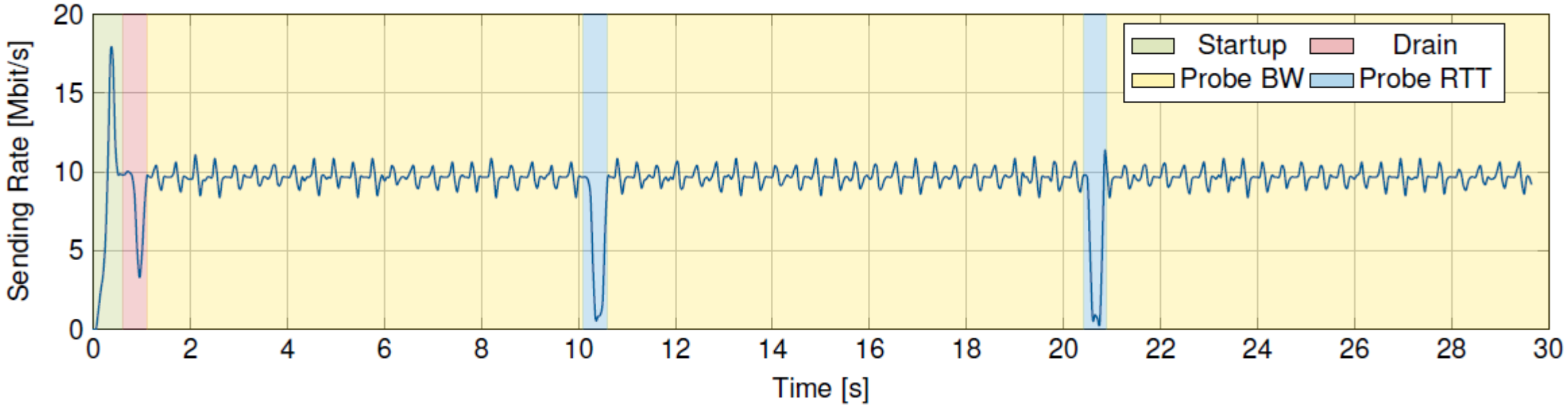




(from materials of ETH course "Advanced Topics in Communication Networks", Fall-2025)

TCP BBR (Bottleneck Bandwidth and RTT) by Google

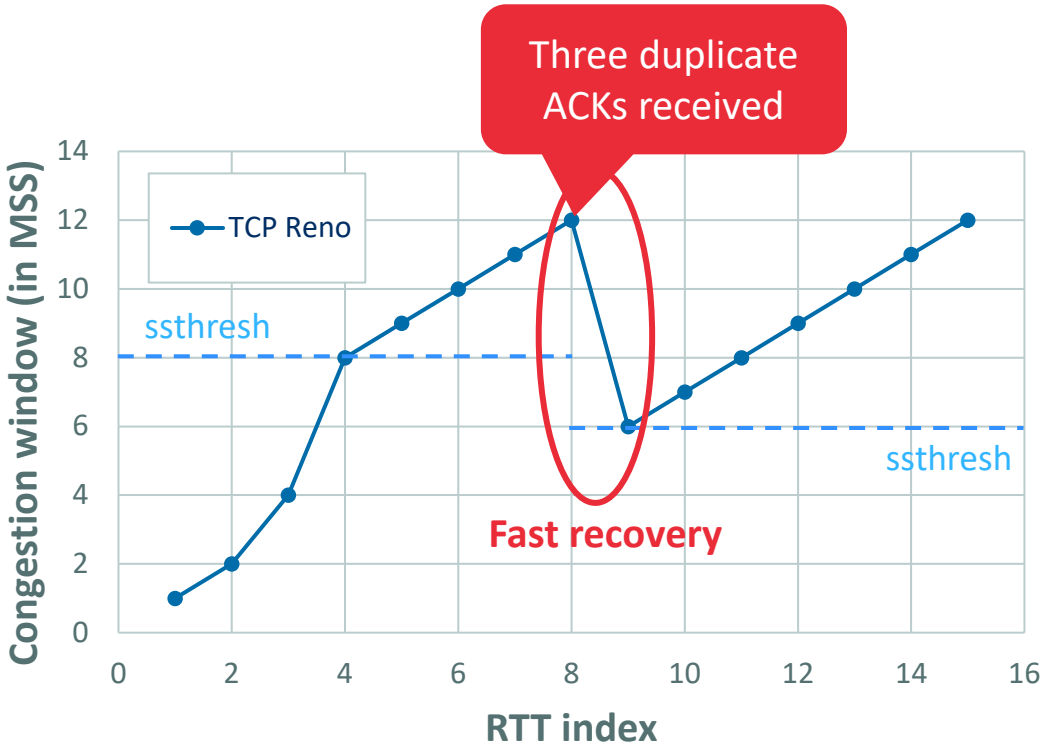
- Presented by Google in 2016. Still under development, currently BBRv3 is the latest version.
- Available in Linux since kernel v4.9. Used in Google's and Youtube servers, as well as in Google's backbone network.
- Continuously monitors the network to estimate bandwidth-delay product (BDP) and operate most of the time near the optimal point (cwnd = BDP).



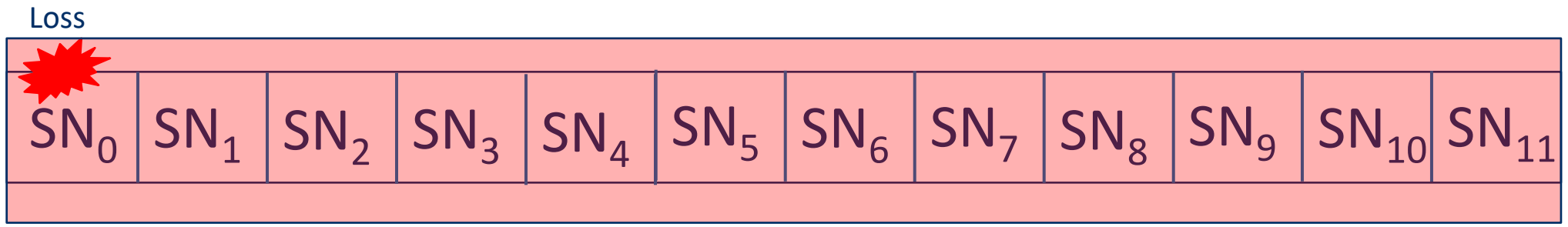
(from materials of TUM course "Advanced Computer Networking", 2025-2026)

State 3: Fast recovery

- Fast recovery starts when 3 duplicate ACKs are received during slow start or congestion avoidance
  - ssthresh is set to cwnd / 2
  - $cwnd = ssthresh + 3 \cdot MSS$
- On each next duplicate ACK $cwnd$ is increased by 1 MSS (window inflation)
- When a **timeout** occurs, TCP goes to slow start (setting cwnd to 1 MSS and ssthresh to cwnd / 2)
- When an **ACK arrives** for the missing segment, TCP goes back to congestion avoidance. $cwnd$ is reset to ssthresh (window deflation)



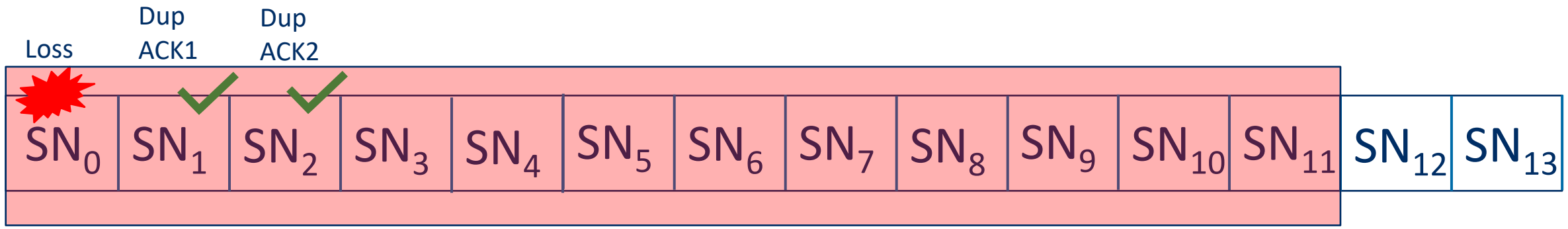
State 3: Fast recovery



TCP segment with sequence number SN_0 is lost.

Current congestion window size *cwnd* = 12 MSS

State 3: Fast recovery

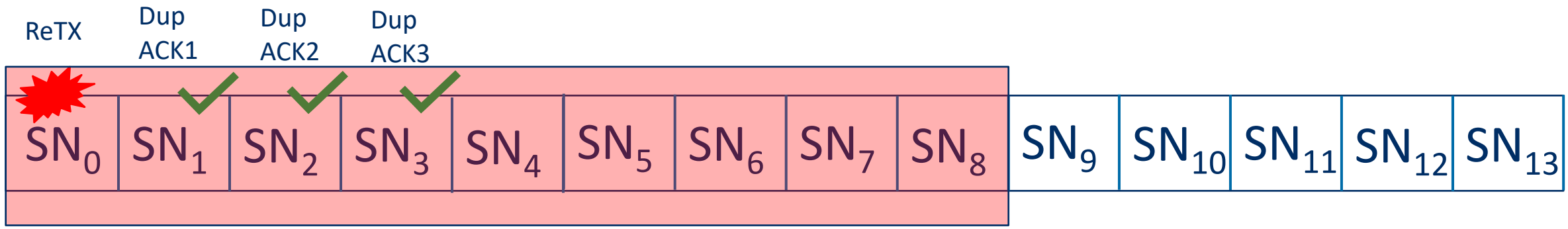


First two duplicate ACKs arrived.

ACKs confirm that some data segments have been received => two data segments has been removed from pipe. TCP transmits 2 segments to substitute these segments (limited transmit) [RFC 3042].

Segments with sequence numbers SN_{12} and SN_{13} are transmitted.

State 3: Fast recovery

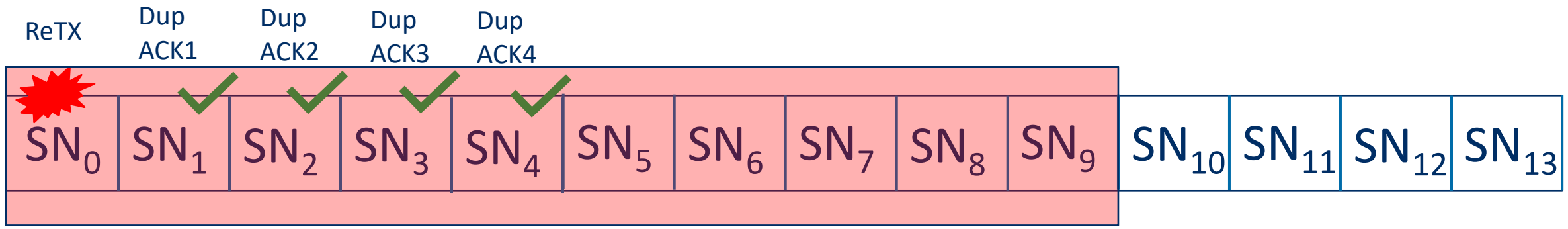


Third duplicate ACK arrived.

Fast retransmit is performed. TCP retransmits segment with sequence number $SN_{0.}$

Fast recovery starts. Slow start threshold is set to *ssthresh* = 6 MSS, congestion window is set to *cwnd* = *ssthresh* + 3 MSS = 9 MSS.

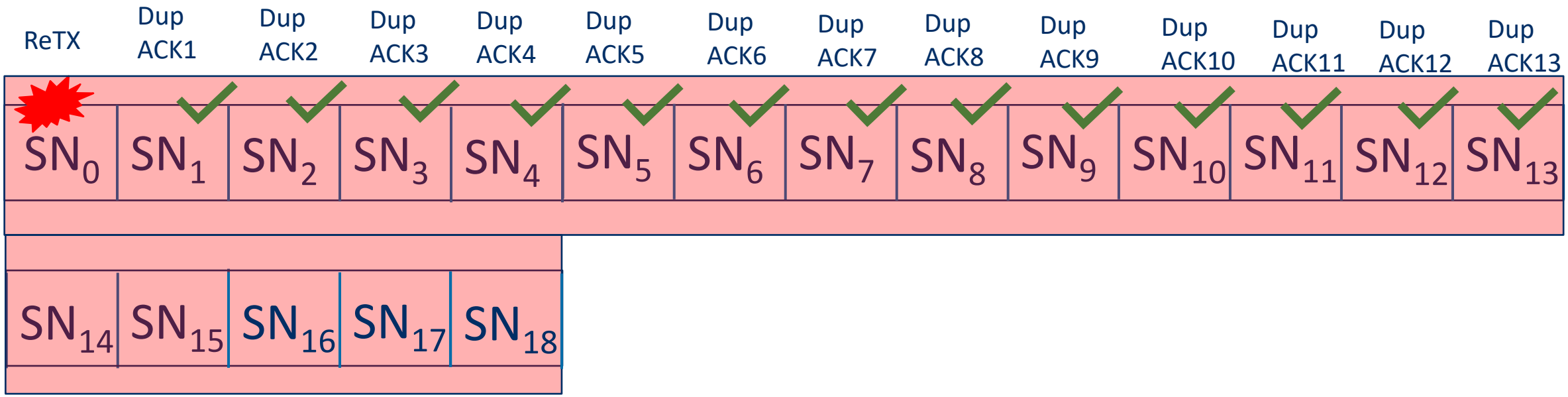
State 3: Fast recovery



Fourth duplicate ACK arrived.

One more segment left the pipe => TCP “inflates” window to compensate (TCP conservation law). Since segment with SN_9 has been already sent, nothing will be sent.

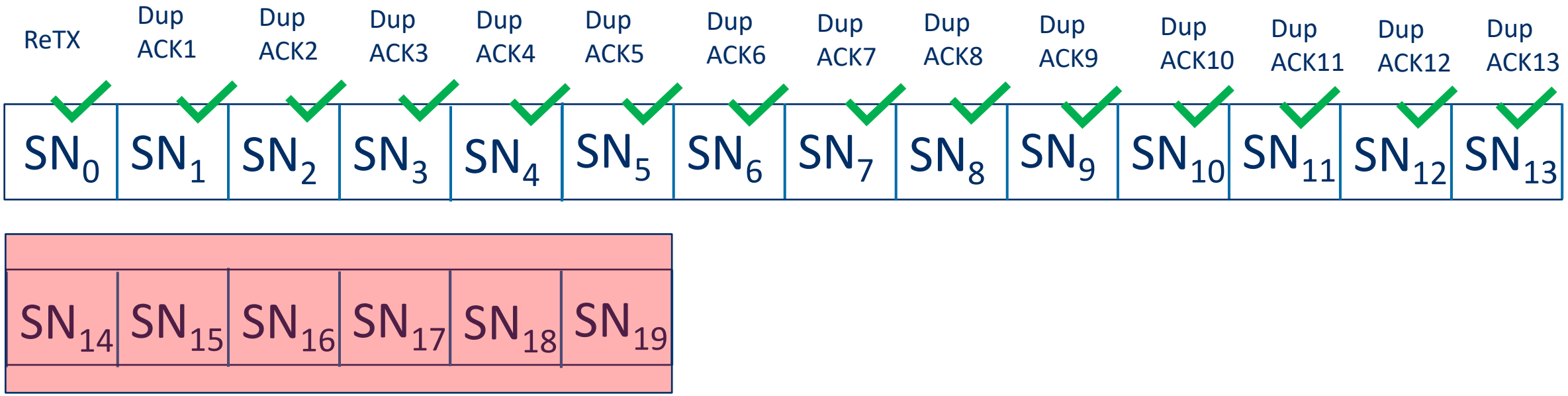
State 3: Fast recovery



13th duplicate ACK arrived.

TCP continue “inflating” window. New segments with sequence numbers SN_{14}, SN_{15}, SN_{16}, SN_{17}, and SN_{18} are sent.

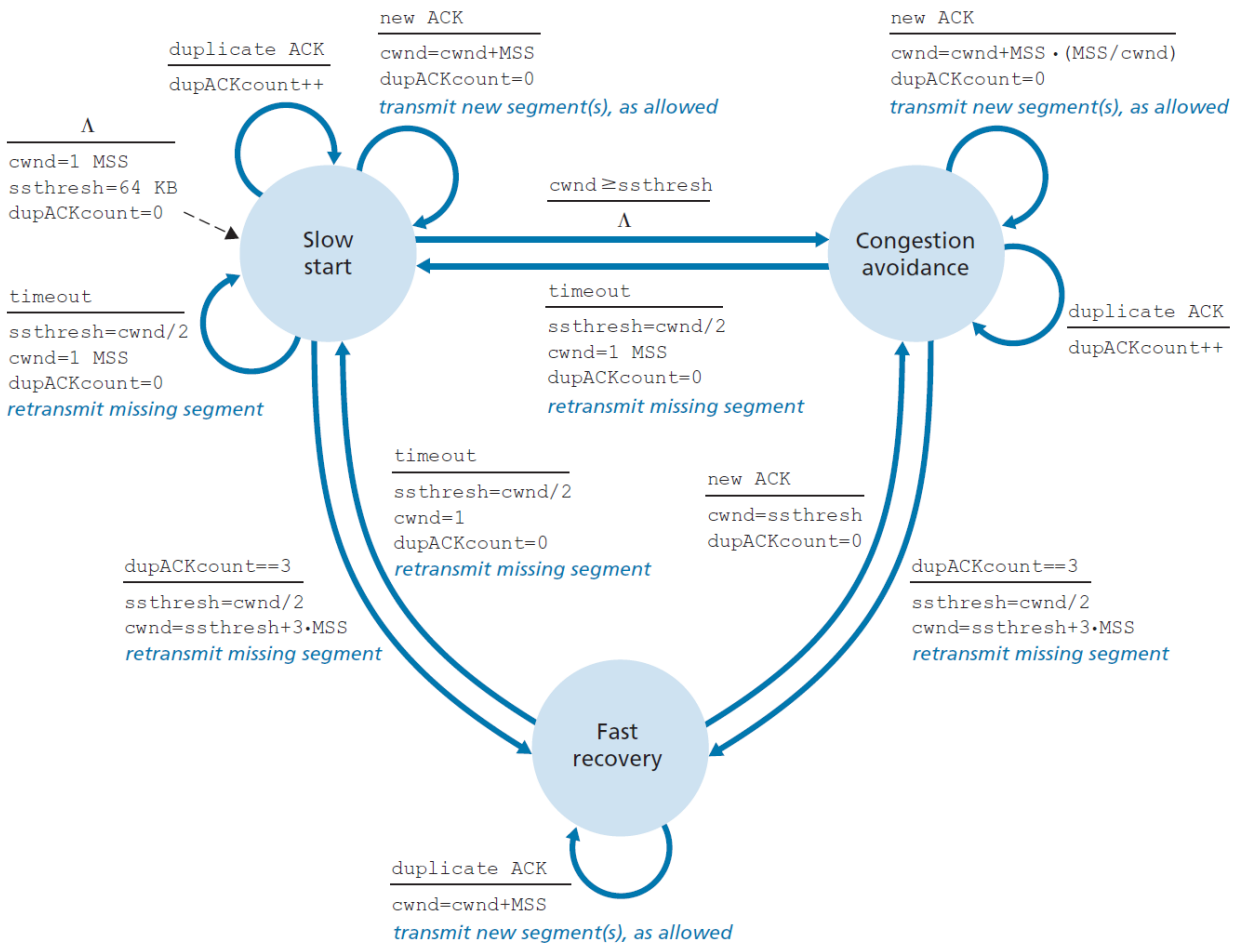
State 3: Fast recovery



Acknowledgement for TCP segment with sequence number SN_0 finally arrives. Fast recovery is finished.

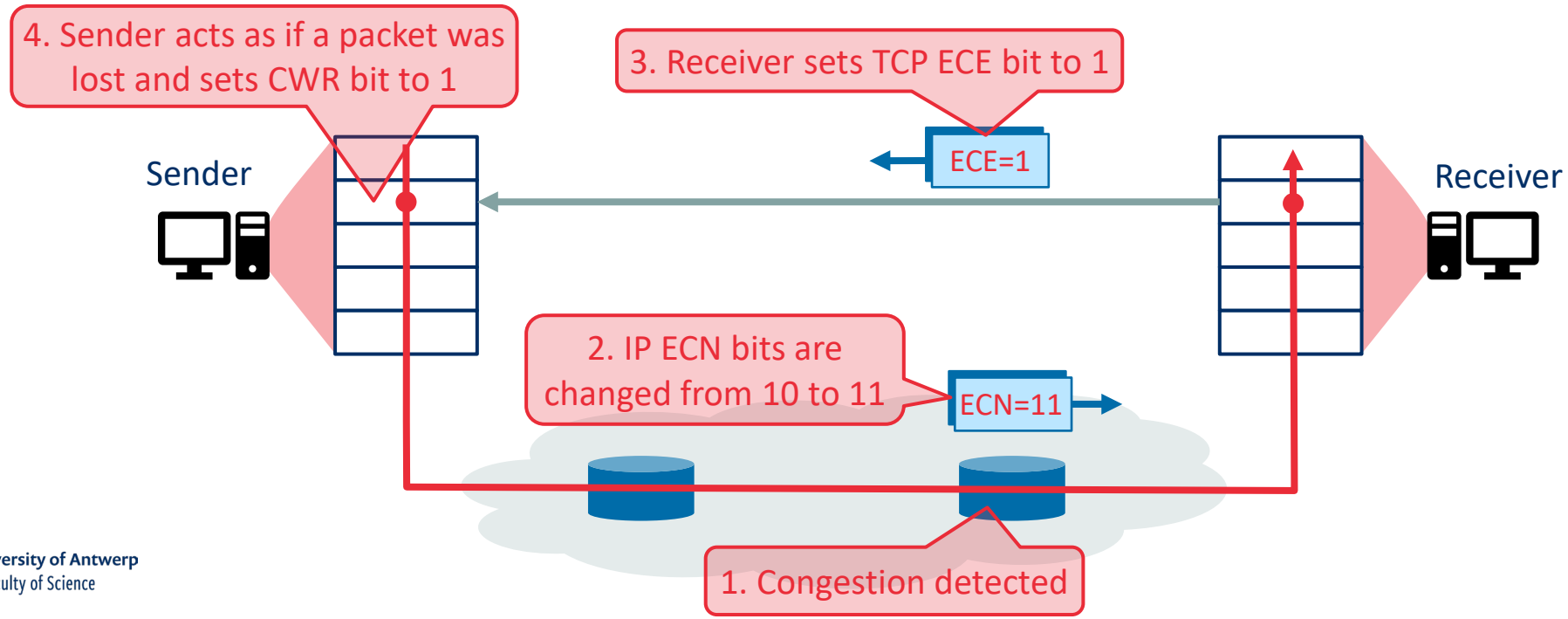
TCP deflates congestion window. The new value is: *cwnd* = *ssthresh* = 6 MSS. Segment with sequence number SN_{19} is sent. TCP congestion avoidance start.

Summary: FSM description of TCP congestion control

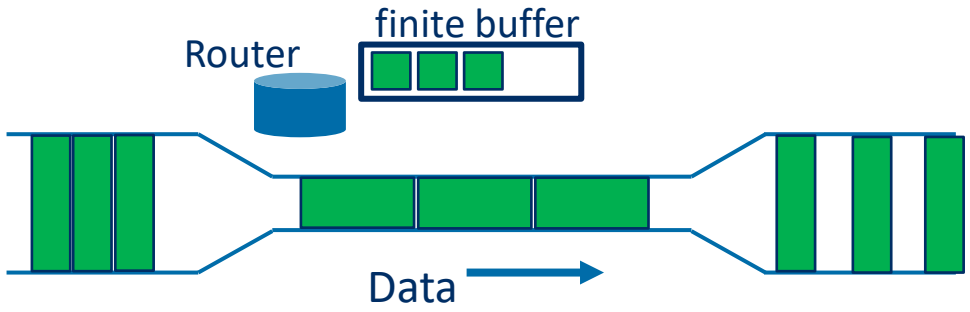


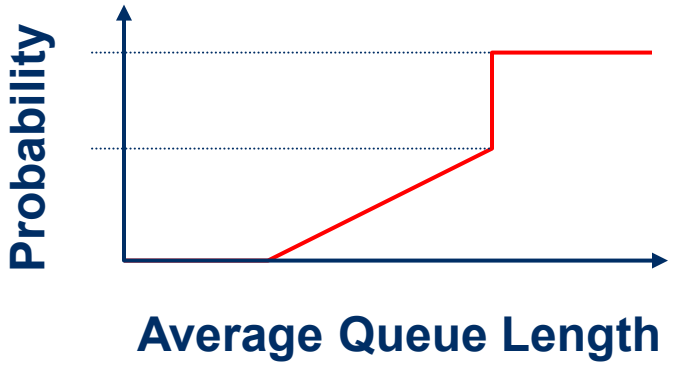
Explicit congestion notification (ECN) [RFC 3168]

- Uses 2 bits in the **IP** header
  - ECN-capable transport (ECT) bit is set to 1 if the sender and receiver support ECN
  - Congestion Experienced (CE) bit is set to 1 if an intermediary router experiences congestion
- What qualifies as congestion is left to the equipment vendor and/or network operator to define
- If the CE bit is 1 on a received TCP packet, the receiver will set the TCP ECE bit to 1 in its ACK
- If the sender receives an ACK with ECE=1 then it acts as if a packet was lost and sets CWR=1 in next segment



Active queue management





- When the buffer of the router becomes full, it is already too late to start dropping the packets.
  - long delay because of waiting full queue of packets;
  - butch-losses because multiple packets arrive to the fully occupied buffer
- Active queue management addresses this issue. The idea is to drop packets earlier to throttle TCP senders
- Example: *Random Early Detection (RED)* is designed to drop packets randomly. The probability of dropping increases with the queue length.