Technische Universität Berlin

# Computer Networks

## Transport Layer

# Chapter

1. Introduction
2. Protocols
3. Application layer
4. Web services
5. Distributed hash tables
6. Time synchronization
7. Error control
8. **Transport layer**
   - **UDP / TCP**
   - **TCP flow and congestion control**
   - **Performance evaluation**
9. Network layer
10. Internet protocol
11. Data link layer
12. WLAN

Top-Down-Approach

| |
|---|
| Application Layer |
| Presentation Layer |
| Session Layer |
| Transport Layer |
| Network Layer |
| Data link Layer |
| Physical Layer |

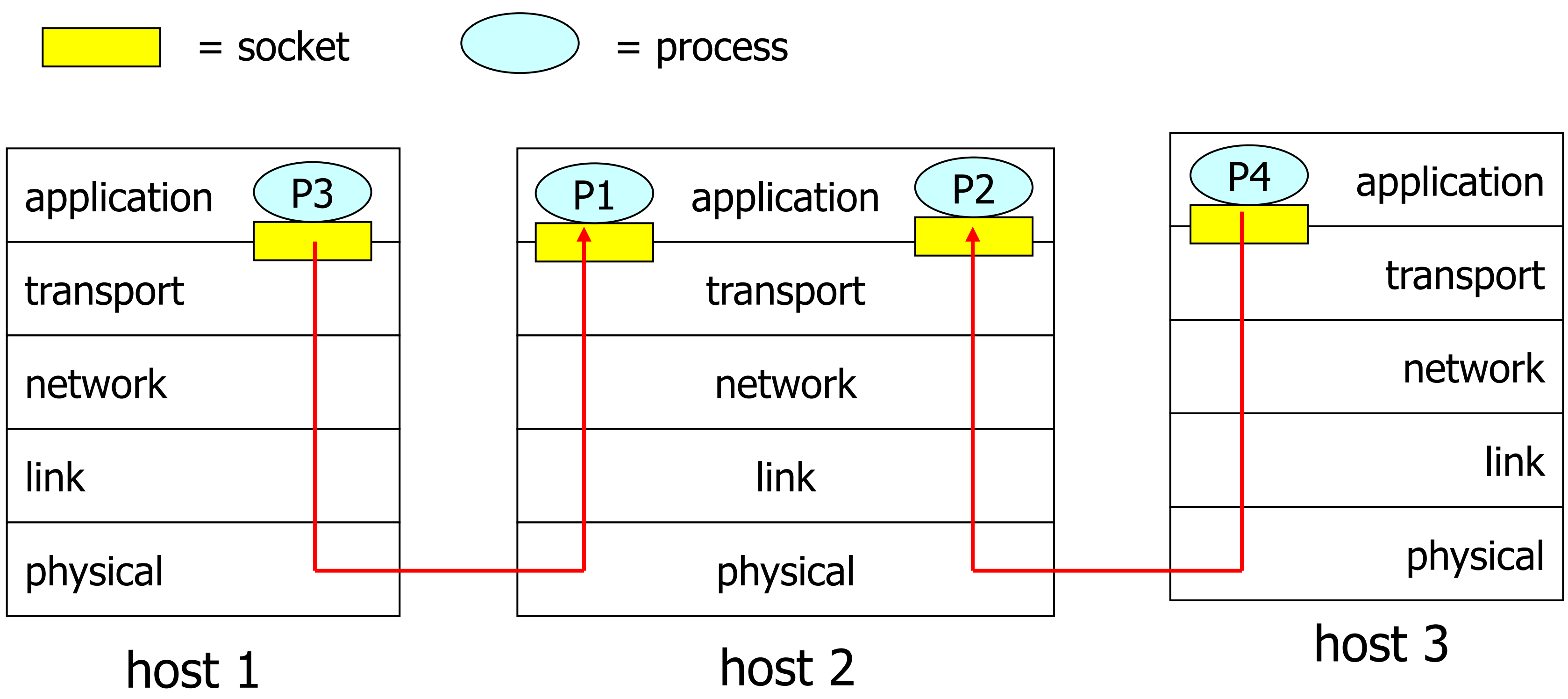
# Transport layer

# Transport layer

- Provides access from user processes to transport services: **transport layer interface**
- Packet loss is common in real networks, network layer does, by definition, not provide reliable communication
- Connection-oriented transport layer services may realize reliable communication
  - Objective is to realize reliable connection over an unreliable network
- In general: **abstraction from imperfections of the network**

# Socket interface

- Goal of transport layer: communication between user processes (running on different computers)



# Transport layer

- Possible services
  - Error control
  - In sequence transmission
  - Connection-less / connection-oriented communication
  - Flow and congestion control
  - Quality of service guarantees (e.g., data rate, delay, loss)

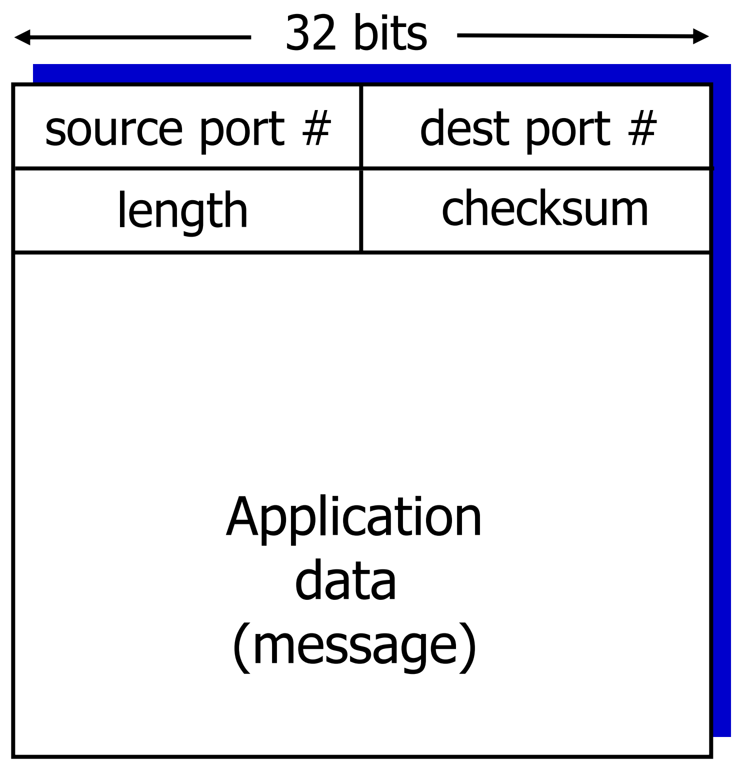
- User Datagram Protocol (UDP)
  - Connection-less, no flow or congestion control, no guarantee for in-sequence delivery
  - Interface for simple packet transmission over IP

- Transmission Control Protocol (TCP)
  - Connection-oriented, provides error, flow, and congestion control, no quality of service
  - Interface provides abstraction of a byte stream

# UDP

# UDP

- Segment:
  - Source port (16 Bit)
  - Destination port (16 Bit)
  - Length of entire segment (16 Bit)
  - Checksum (16 Bit)
    optional, $0000000000000000_2$
    means that the field is not used



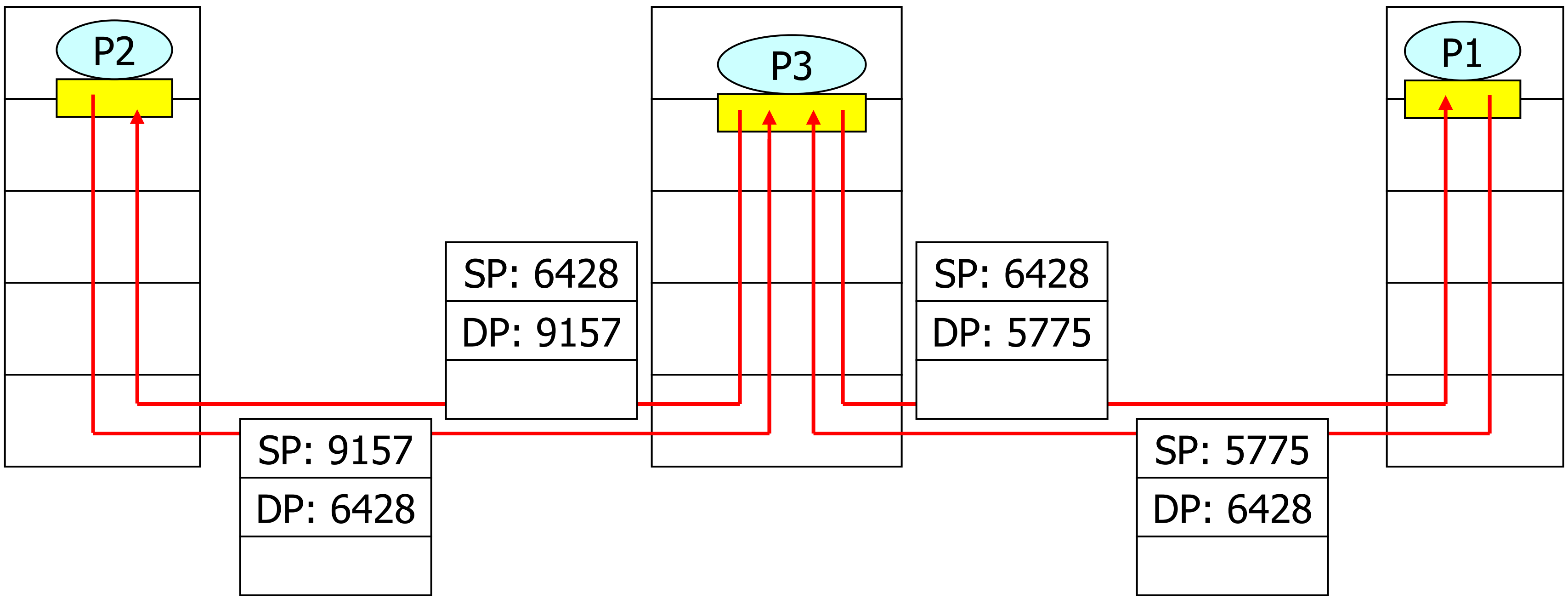

- Question: where is the source and destination IP address?

# UDP: Multiplexing and demultiplexing

- **Multiplexing** of segments from different application processes at source host
- **Demultiplexing** of segments to be delivered to specific application processes at destination host
- Identification using port numbers
  - Source port: selected by application process or operating system
  - Destination port: identifies target application
- An application process can use multiple such ports at the same time
- Implementation using **socket interface**

# UDP: Multiplexing and demultiplexing

- Example






Telecommunication Networks Group

# UDP: Checksum

- Checksum calculation
  - Segment is treated as array of 16-bit numbers
  - All these numbers are added in **ones' complement arithmetic**
    - You get -x from x by inverting all bits
    - The remainder is added to the result
  - The result is inverted → checksum
- Sender: checksum is calculated and added to the segment
- Receiver: checksum is calculated again and added to the received checksum
  - No error if result is $1111111111111111_2$ (ones' complement representation of 0)
- Single bit errors can be detected, but not doubles
  - There are, of course, better error detection mechanisms

# UDP: Checksum

Technische Universität Berlin

- Example

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| | | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| Remainder | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| Sum | | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| Checksum | | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |



Telecommunication Networks Group

# UDP: error probabilities

- Let the bit error probability be $p = 10^{-7}$
- Let the segment length be $N = 10^4$ bit
- Usual assumption: bit errors are independent (easy calculation but wrong in reality – burst errors)
- Now, probability of at least one bit error per segment:
  $1-(1-p)^N = 0{,}00099950 \approx 10^4 \cdot 10^{-7} = 10^{-3}$
- Probability of two bit errors per segment:
  - Number of bit pairs: $\sum_{i=1}^{N-1} i = (N-1) \cdot N/2 = (10^4 - 1) \cdot 10^4 / 2 \approx 10^8 / 2$
  - Probability that a specific pair is in error: $10^{-14}$
  - Probability that any bit pair is in error: $10^8 \cdot 10^{-14}/2 = 10^{-6}/2$
- Question: how long does it take to get a segment with two bit errors: (a) at 10 Mbps and b) at 10 Gbps?

# UDP: pseudo header

- In reality, it is a bit more complicated… UDP uses a **pseudo header**
  - It contains source and destination IP address, protocol number (17 for UDP), segment length
  - Sender initialized checksum with 0, prepares a pseudo header, and calculates the checksum over the segment including the pseudo header
  - This checksum is used
  - Receiver gets IP address information and prepares again a pseudo header, initializes checksum with 0, and calculates the checksum
- Advantage: the checksum also detects errors in the IP addresses
- Disadvantage: abstraction in the strict layering is violated (even though only at the end systems)

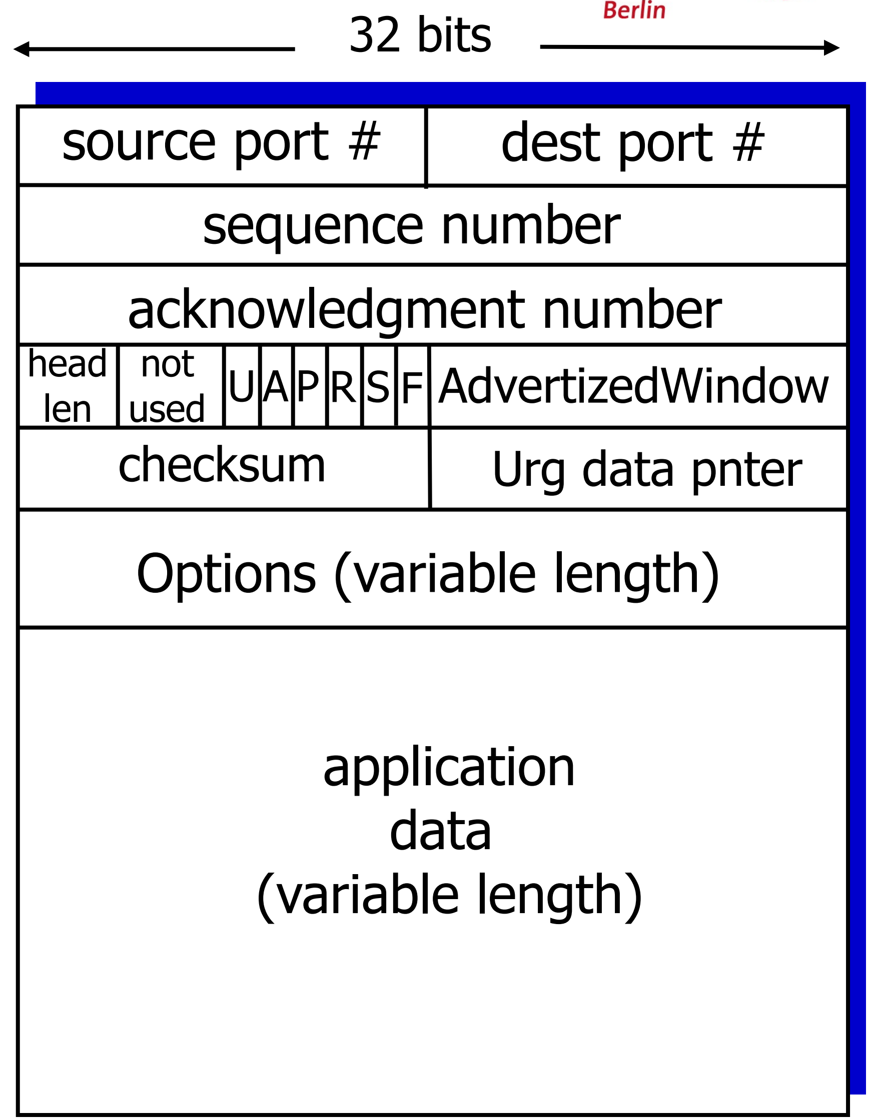
# TCP

# TCP

- Transmission Control Protocol
  - Mostly used transport protocol on the internet
  - RFCs 793, 1122, 1323, 2018, 2581
  - **Point to point**: one sender, one receiver
  - **In-order** delivered byte stream
  - Window-based **error control**
  - **Full duplex**: two independent byte streams in opposite direction
  - **Connection oriented**: connection needs to be established and teared down
  - **Flow control**: mechanism to avoid overloading the receiver
  - **Congestion control**: mechanism to avoid overloading the network beyond capacity

# TCP: Segment format

- sequence number: position of the first byte of the segment in the byte stream
- ack. number: number of the next expected byte in the byte stream
- Control flags:
  - URG (urgent pointer)
  - ACK (acknowledgement)
  - PSH (push segment)
  - RST (reset connection)
  - SYN (synchronize connection)
  - FIN (terminate connection)
- AdvertizedWindow: window size for flow control
- checksum: checksum as in UDP

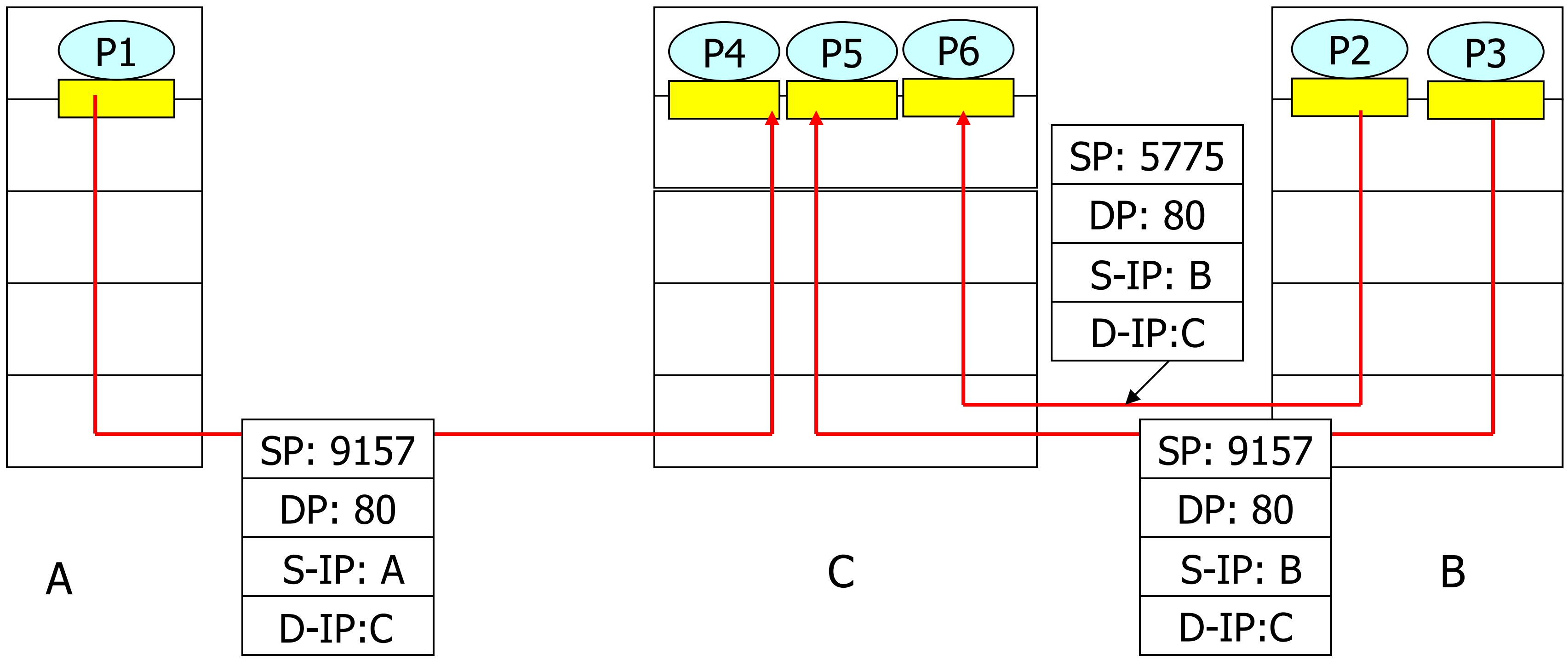


# TCP

- Multiplexing and demultiplexing
  - TCP connection characterized by 4-tupel
    - Source IP address
    - Destination IP address
    - Source port
    - Destination port
  - Thus, one port can serve multiple TCP connections (e.g., 80 for http)
- Pseudo header
  - As in UDP, including checksum

# TCP

- Multiplexing and demultiplexing, example:



# TCP: Error Control

- Mix of Go-Back-N and Selective Repeat (and some new aspects)
  - **Buffer on sender and receiver side**
  - **One timer**
  - **Cumulative ACKs**
  - Sequence and ACK numbers count bytes in byte stream
    - **Sequence number:** position of **first byte** of the segment in byte stream
    - **ACK number:** position of **next exected byte** in byte stream

- Many implementation variants, we concentrate on the widely used TCP Reno; more in the diverse RFCs

# TCP: Error Control

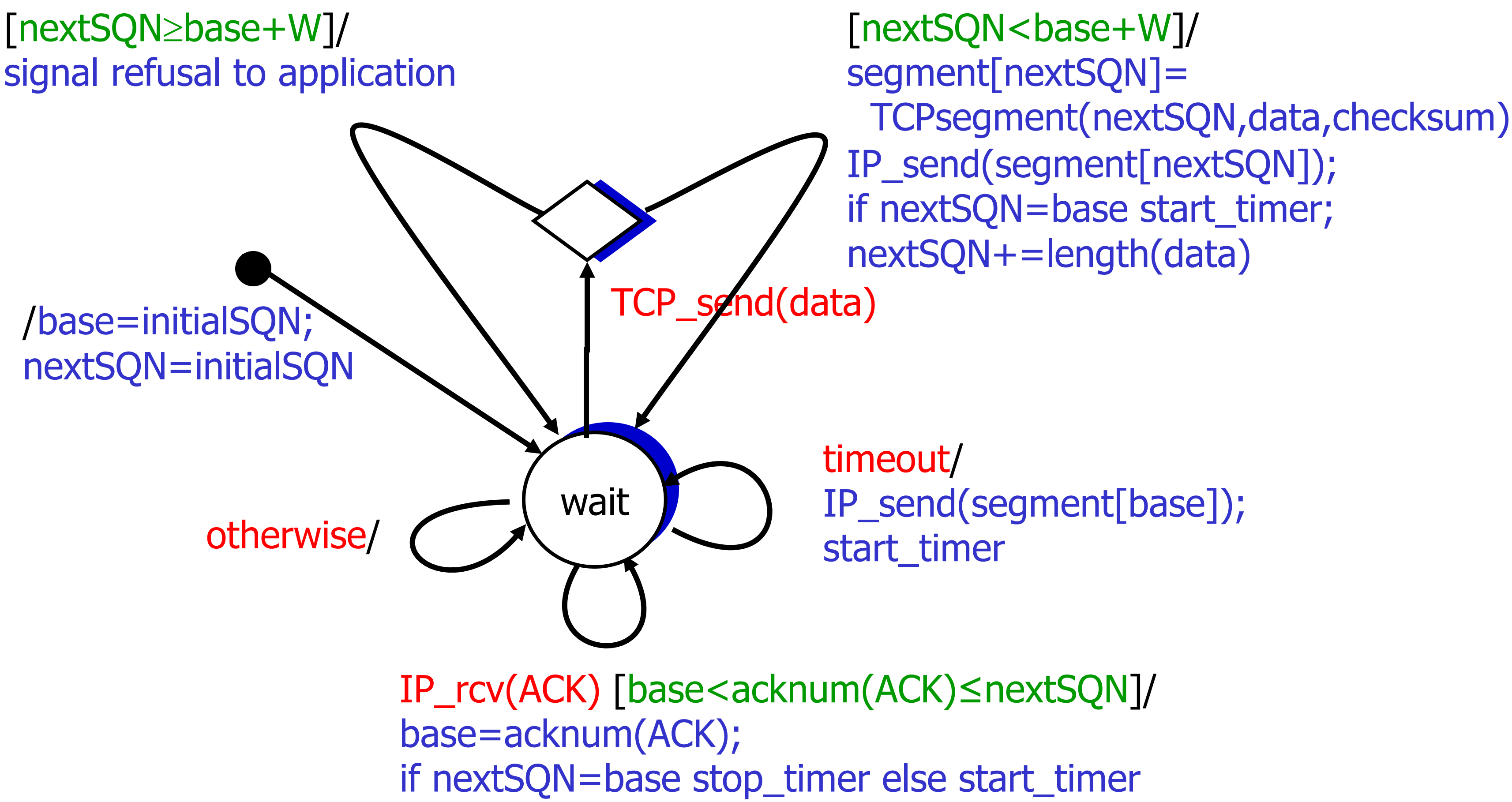
- Big picture
  - Sender may send multiple segments before receiving the ACK (up to a maximum number of bytes as defined by flow and congestion control)
  - After sending of the first segment, a timer is started
  - All not yet acknowledged segments are buffered
  - When timer expires, the **first not yet acknowledged segment is retransmitted**
  - Receiver sends **cumulative ACKs** indicating the position of the first not yet received byte
  - Window on sender and receiver is always moved to the next gap

# TCP: Error Control

- Window at sender and receiver

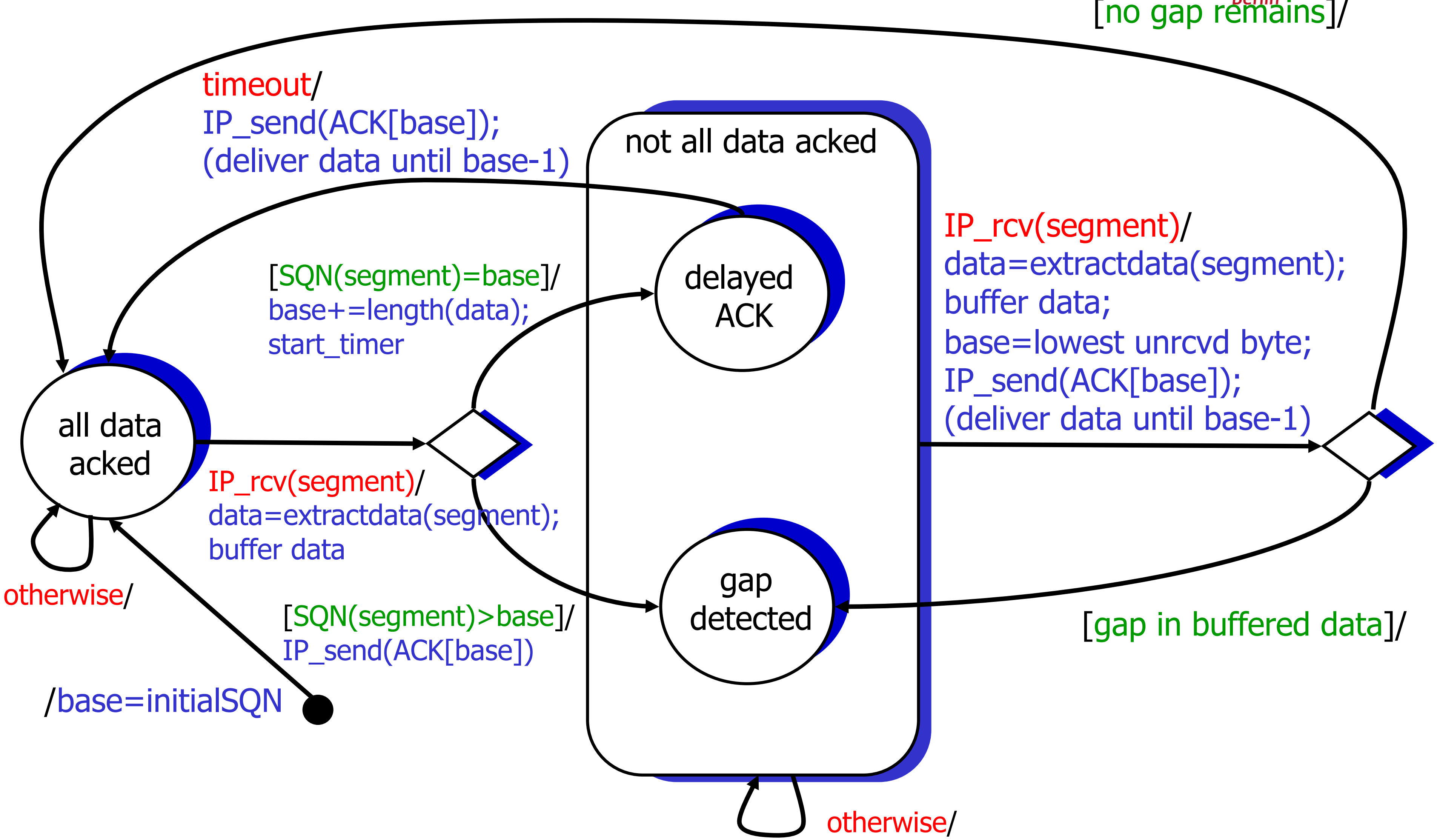

  - base: first byte of window
  - base+W: first byte outside window
  - nextSQN: first byte of next segment
  - Window at sender contains transmitted but not yet acknowledged and not yet sent packets
  - Window at receiver contains received packets, gaps, and space for not yet received packets

# TCP sender

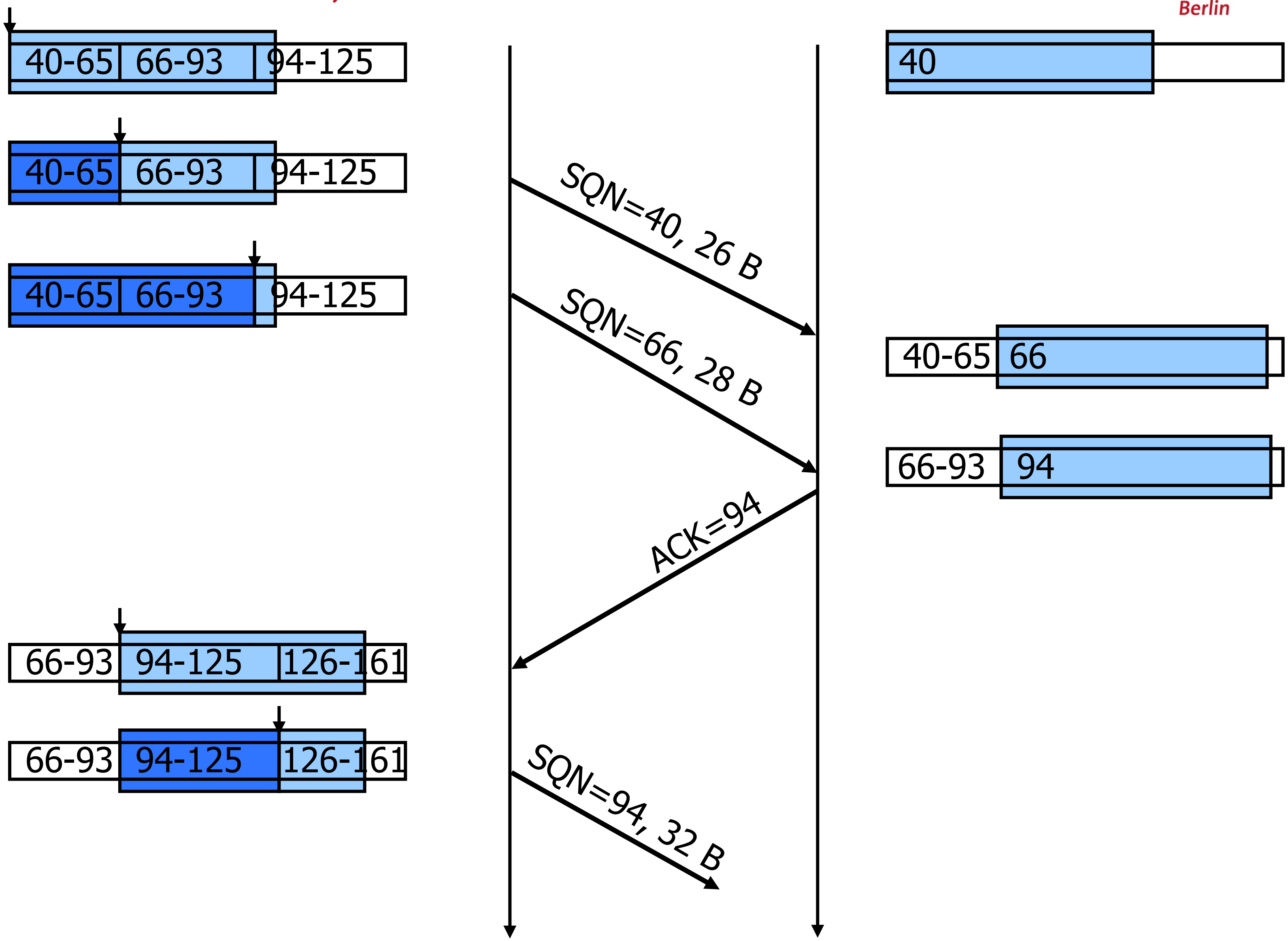


[nextSQN≥base+W]/
signal refusal to application
[nextSQN<base+W]/
segment[nextSQN]=
TCPsegment(nextSQN,data,checksum);
IP_send(segment[nextSQN]);
if nextSQN=base start_timer;
nextSQN+=length(data)
TCP_send(data)
/base=initialSQN;
nextSQN=initialSQN
wait
otherwise/
timeout/
IP_send(segment[base]);
start_timer
IP_rcv(ACK) [base<acknum(ACK)≤nextSQN]/
base=acknum(ACK);
if nextSQN=base stop_timer else start_timer

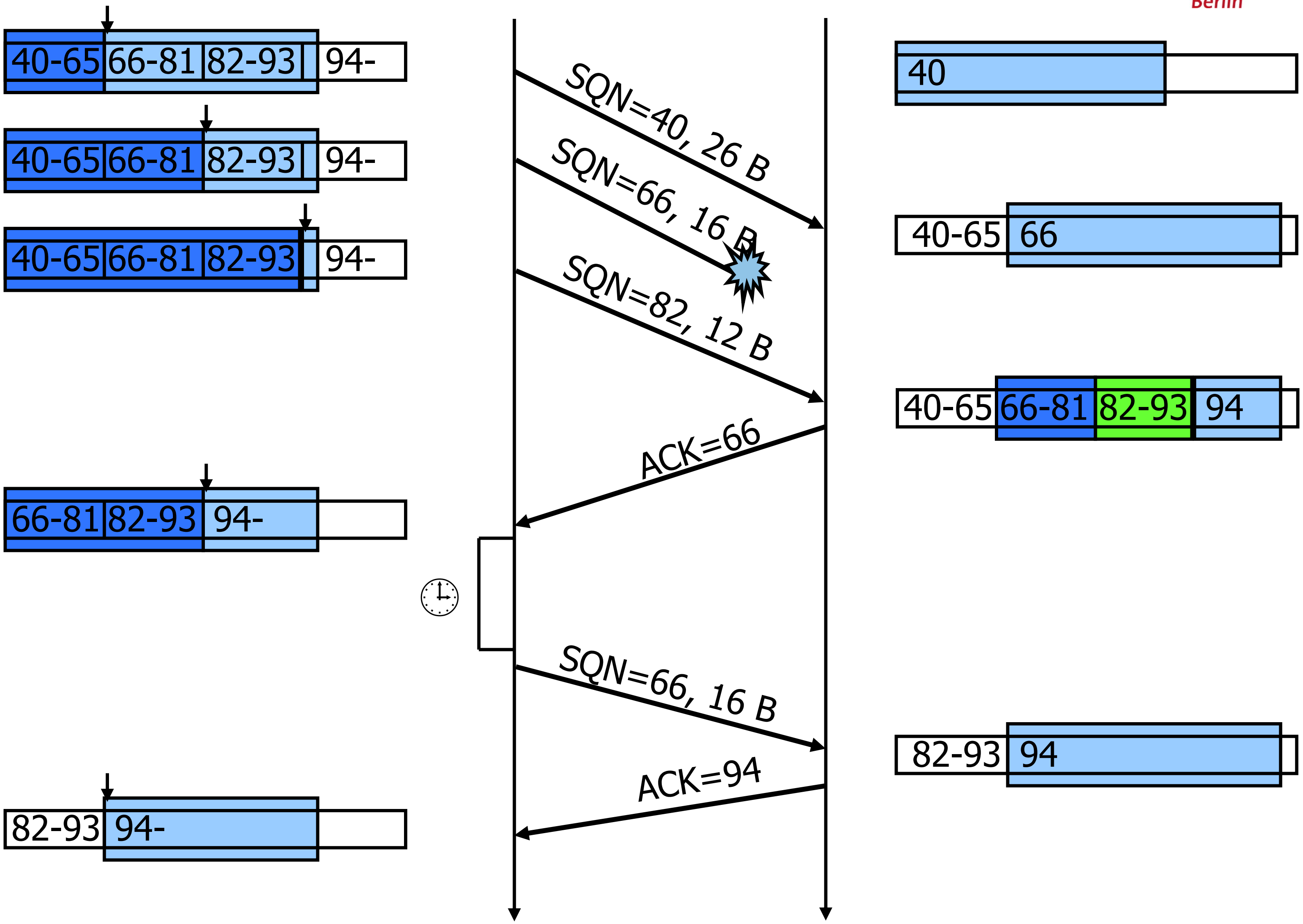
# TCP receiver



# TCP: Error Control, normal execution



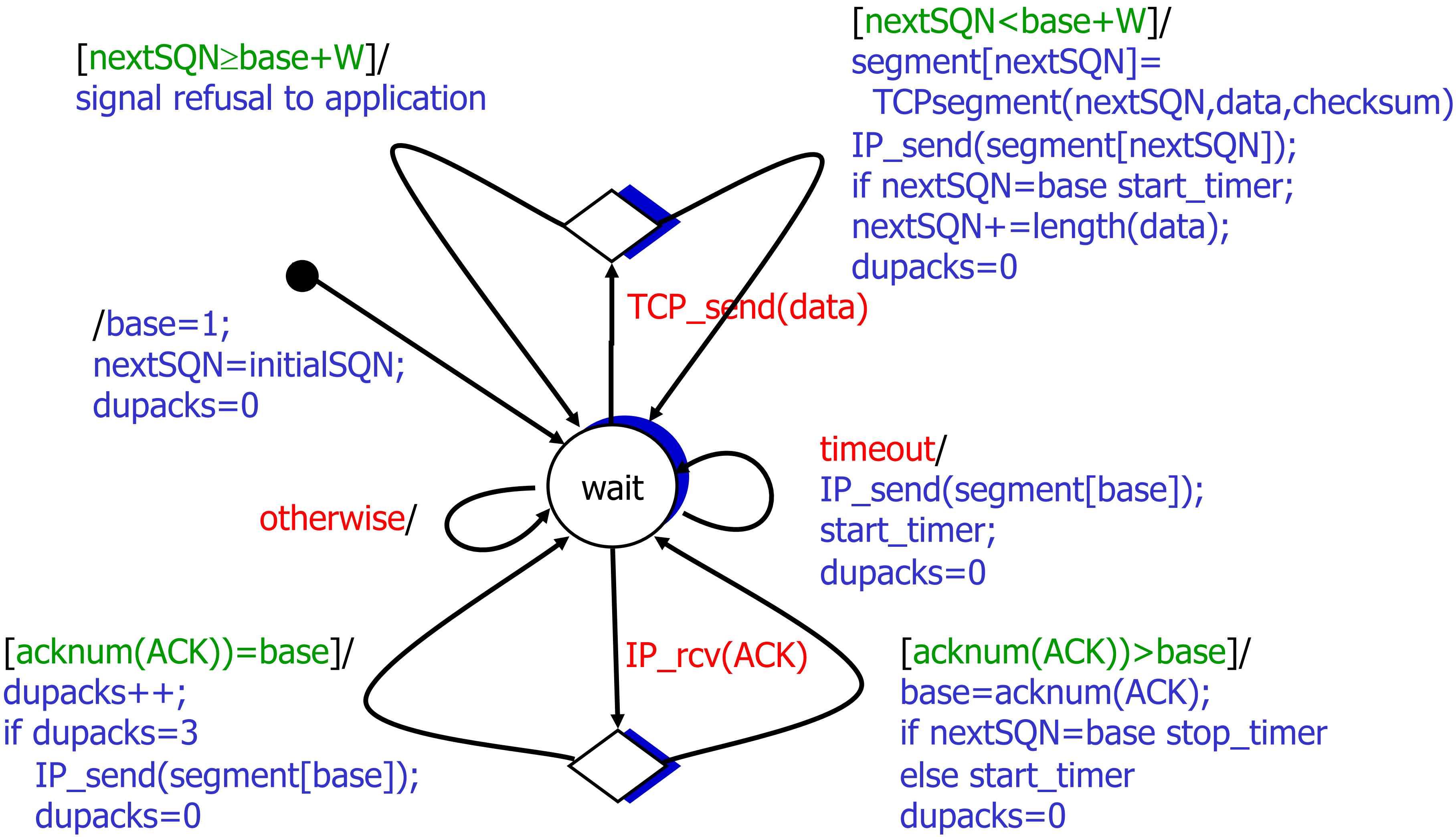
# TCP: Error Control, packet loss



# TCP: Error Control

- Fast retransmit
  - For large bandwidth-delay-product, it takes a long time until a lost packet will be discovered; even longer for multiple lost packets
  - ACKs containing the same ACK number are called **duplicated ACKs**
  - This is an early indicator for a lost segment
  - **Fast retransmit** is initiated after receiving 3 duplicated ACKs (i.e., 4 ACKs with the same ACK number); this triggers a retransmission of the likely lost segment
  - Integration in our state charts (counter for **dupacks** is required)

# TCP sender with fast retransmit



# TCP: Error Control

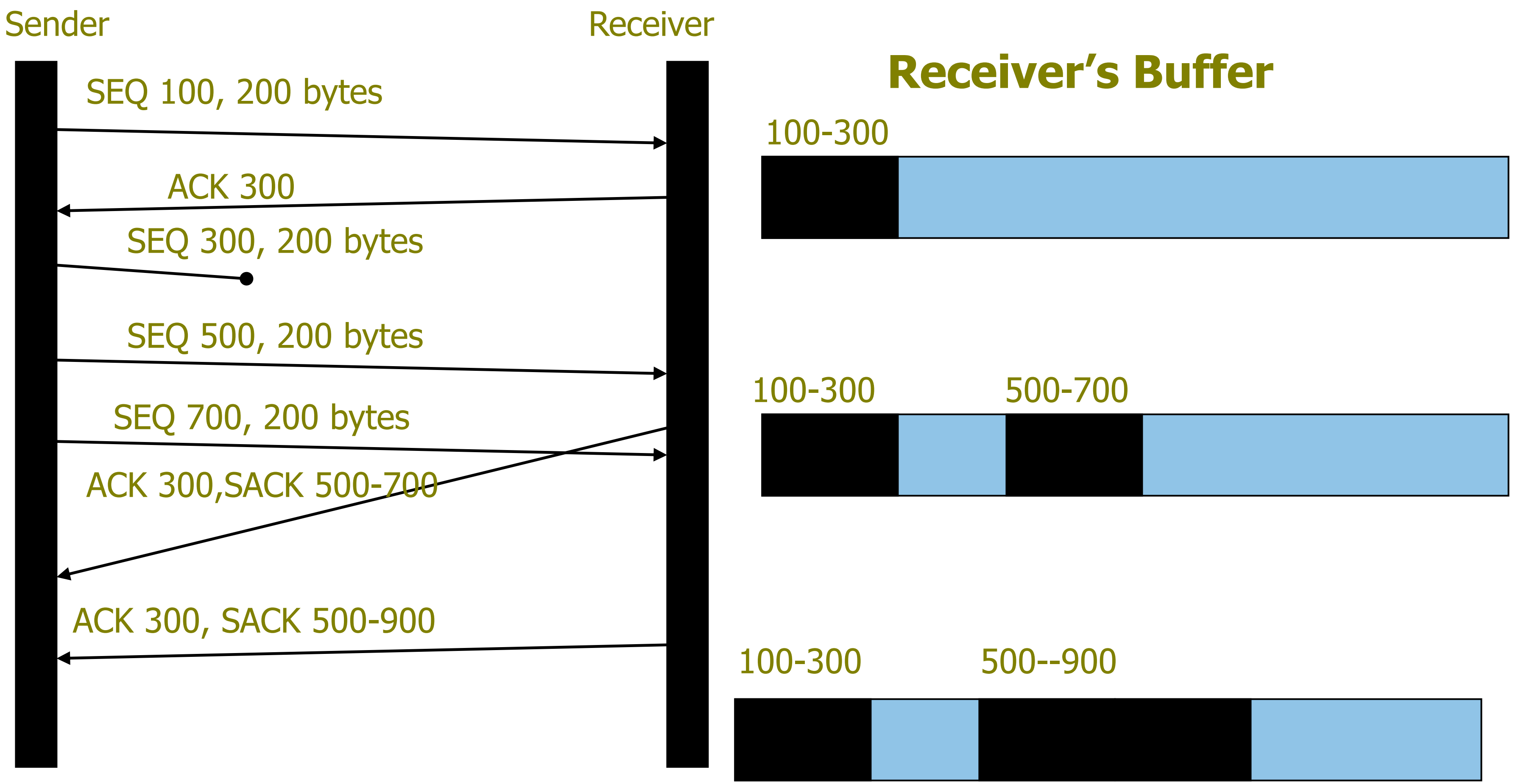
- Comments
  - TCP is **full duplex**: two logical connections
  - ACKs are **piggybacked**: segments in the opposite direction carry ACKs
  - **Delayed ACK** is supposed to reduce the number of ACKs
- Open problems
  - **Multiple packets lost in the window have catastrophic effect** on throughput; sender needs to wait for every lost packet one round trip time
  - Solution: TCP extension **selective acknowledgements (SACK)** using the TCP options field

# Selective Acknowledgements (SACK)

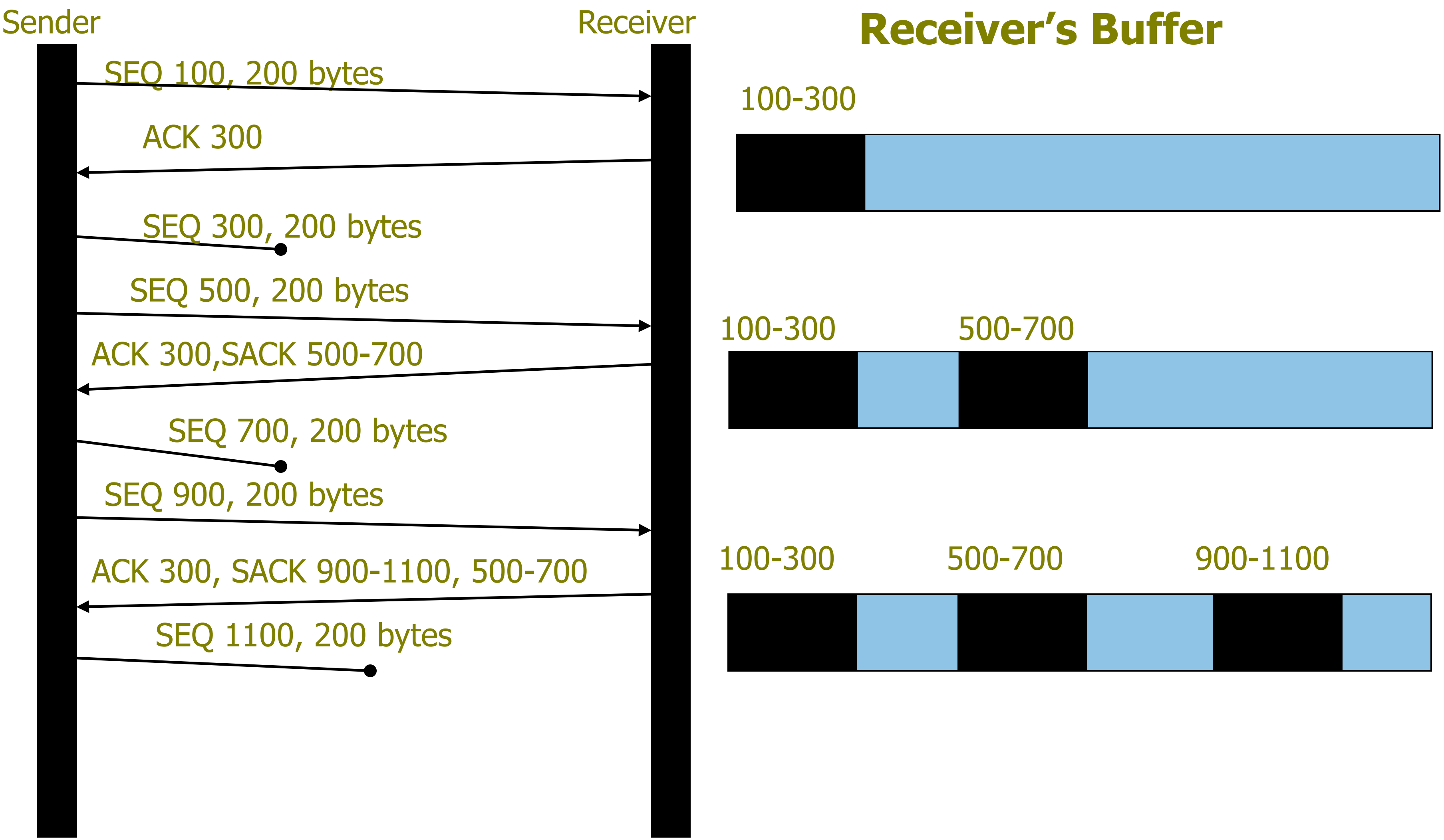
- Basic idea
  - Selective acknowledgments (SACK) inform sender about individual packets that have been received but cannot be acknowledged cumulatively
  - Sender does not need to retransmit these (after timeout)

- Rules
  - Normal ACK remain unchanged (compatibility)
  - SACKs are signaled for the first packet out of order
- Receiver
  - Sends as many SACKs as possible (limited by space in TCP header)
- Sender
  - Initiates retransmissions

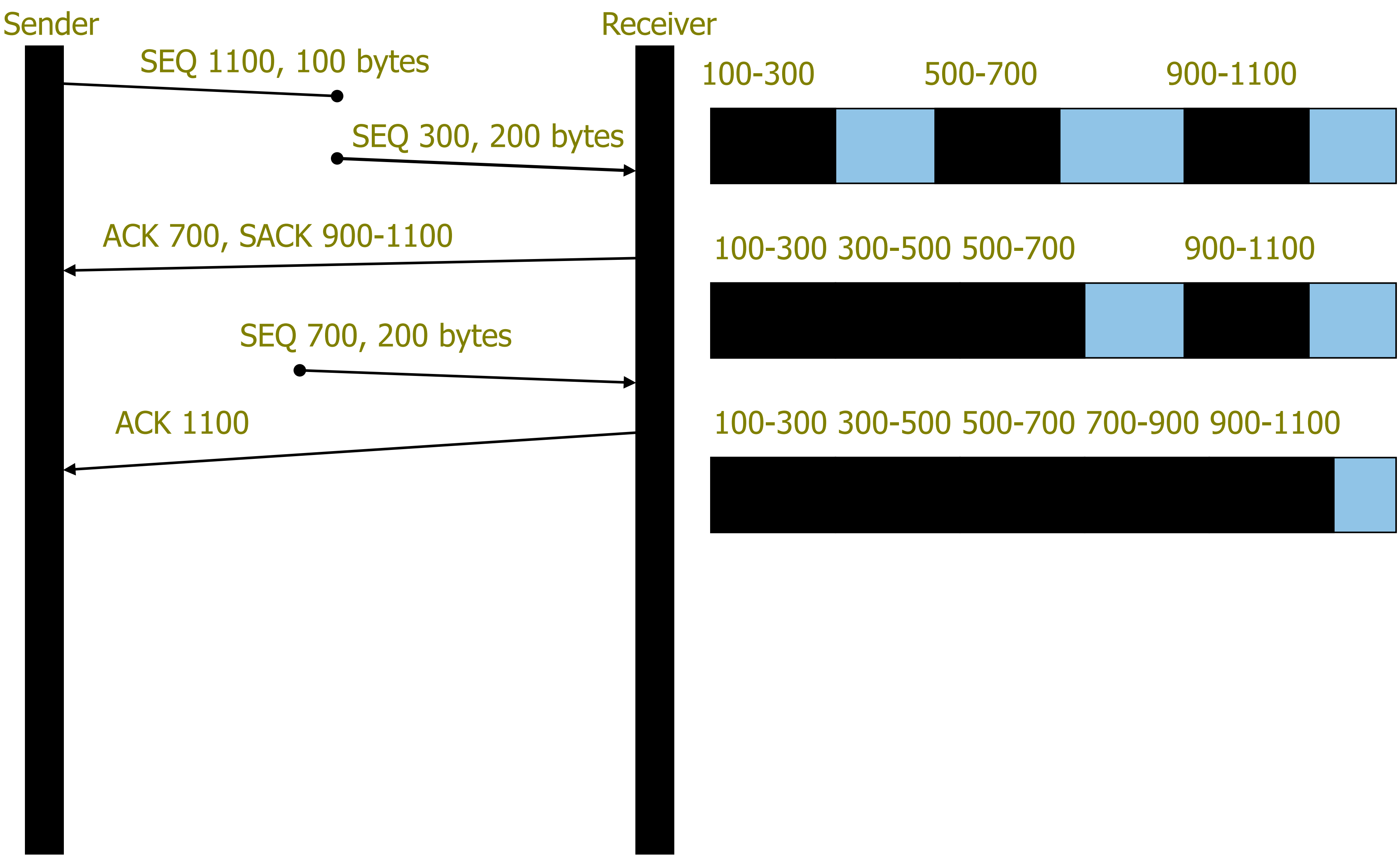
# SACK Example



# SACK Example



# SACK Example



# TCP Error Control

- Sequence number space
  - Sequence numbers: 32 bit, i.e., $2^{32}$ sequence numbers
  - Requirement for sliding window protocols: $2^{32}$>>2 $2^{16}$
    - Sequence number vs. window size
  - Sequence numbers overflow
    - At 10 Mbps: 57 minutes
    - At 1 Gbps: 34 seconds
  - For high bit rates, this is getting qui8te short
  - TCP extension uses time stamps in options to distinguish segments

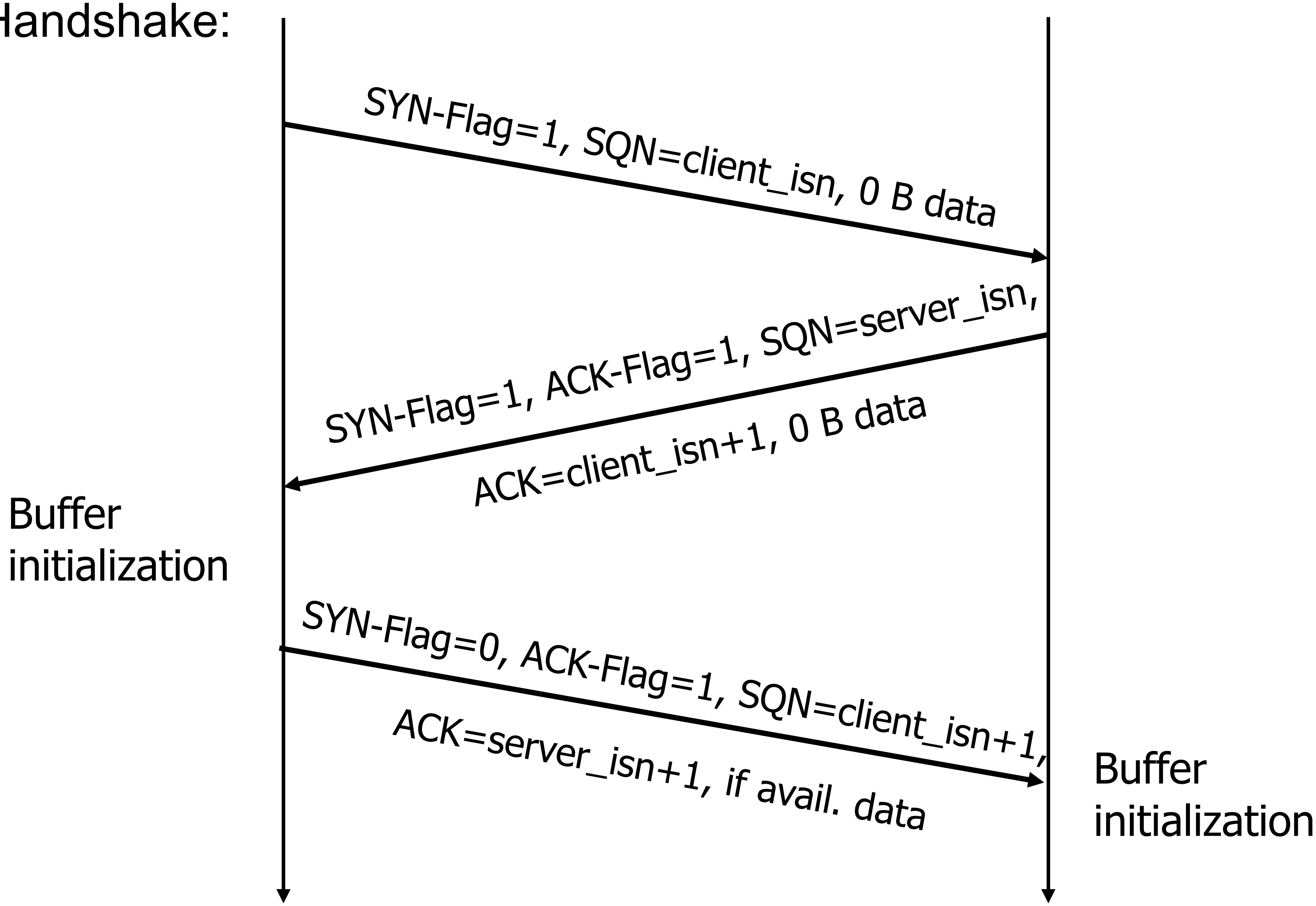
# Connection establishment

# TCP: Connection Setup and Tear Down

- Connection established using **3-way-handshake**
  - **SYN-Segment**: Client sends segment with SYN-Flag=1, random initial Client-SQN (client_isn), no data
  - **SYNACK-Segment**: Server sends segment with SYN-Flag=ACK-Flag=1, random initial Server-SQN (server_isn), ACK=client_isn+1, no data; all buffers are initialized
  - **ACK-Segment**: Client sends segment with ACK-Flag=1; SQN=client_isn+1, ACK=server_isn+1 and potentially data; all buffers are initialized

# TCP: Connection Setup and Tear Down

- 3-Way-Handshake:

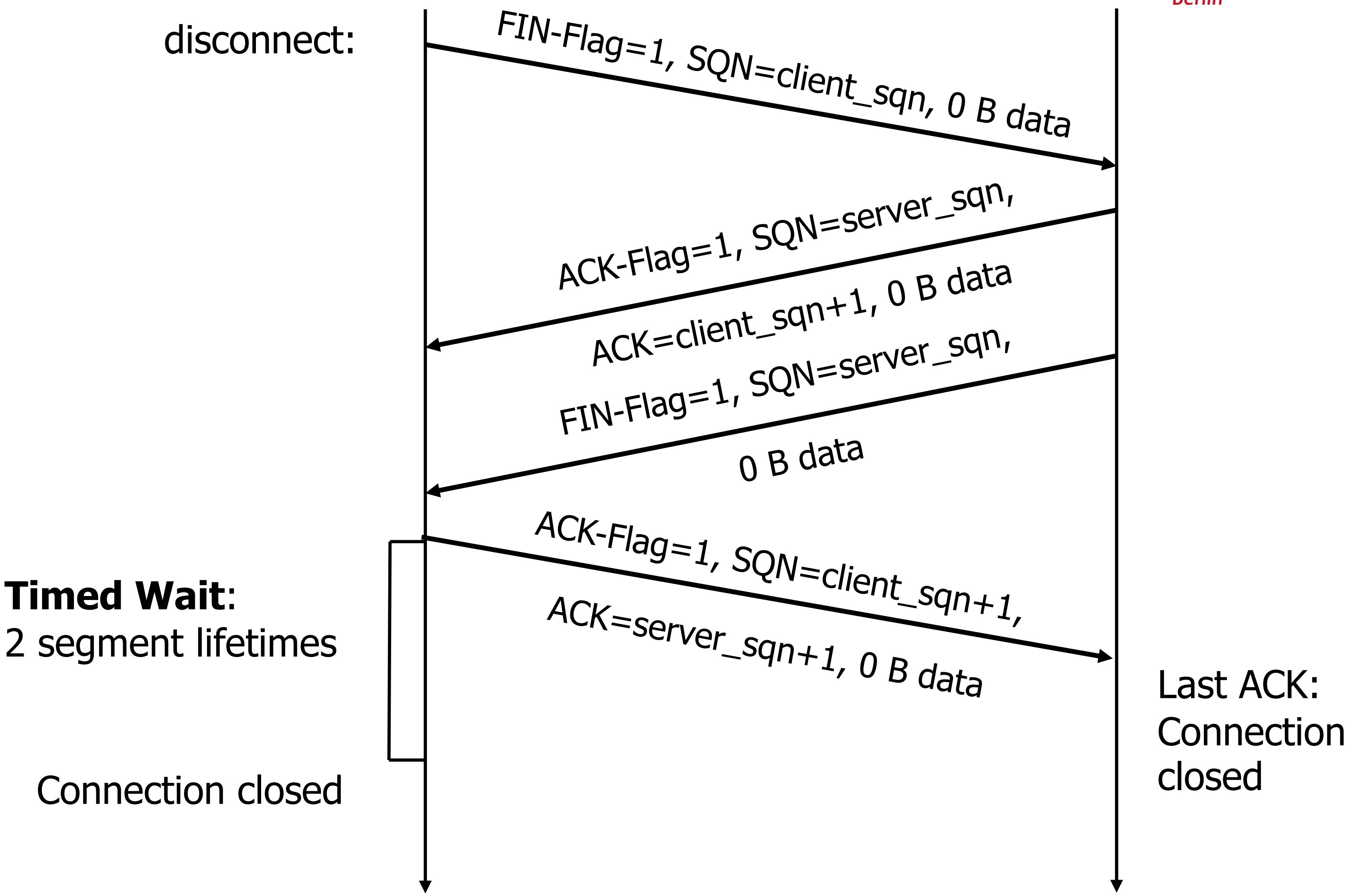


# TCP: Connection Setup and Tear Down

- Sequence numbers in SYN and FIN segments
  - Segments with SYN-Flag=1 or FIN-Flag=1 must not carry data; SQN needs to be incremented by one to explicitly acknowledge these segments

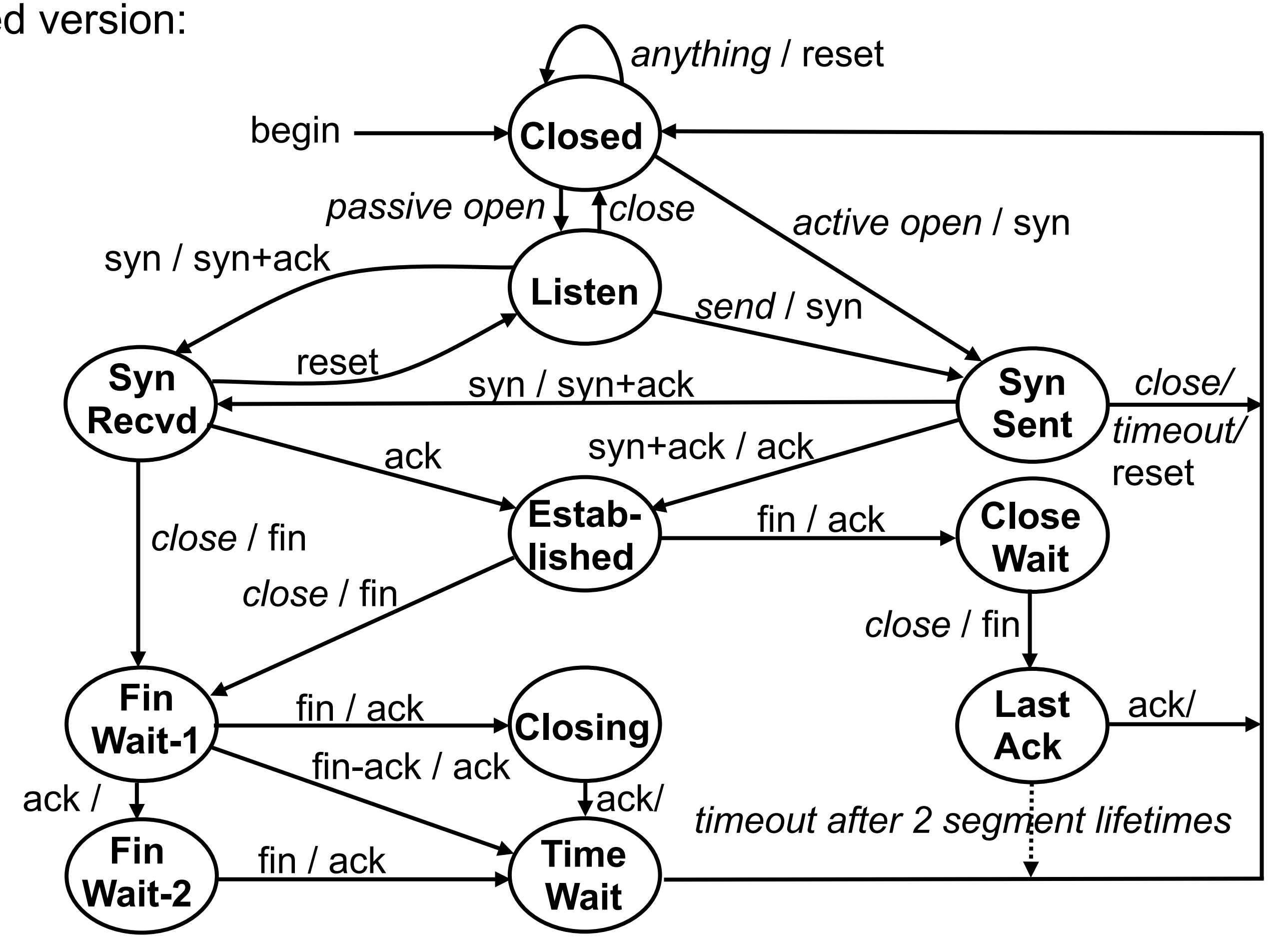
- Disconnection
  - **Both sides can initiate disconnect** by sending a segment with FIN-Flag=1
  - Needs to be acknowledged
  - Both sides need to close each half of the connection
  - After tear down, no data can be sent anymore but reception is still possible
  - **Timed wait**: the party that initiated the connection tear down needs to wait for another 2 segment lifetimes to receive old segments (and to protect a new TCP connection re-using the same port numbers)

# TCP: Connection Setup and Tear Down



# TCP: Connection Management

- State machine first defined in RFC 793
- Improved version:



# RTT Estimation

# RTT Estimation

- Timeout for retransmissions
  - Sender needs to set a timeout
  - ACK can be received after one RTT at the earliest
  - If timeout is too small, there will be unnecessary retransmissions
  - If timeout is too large, there will be unnecessary waiting times
  - Obviously, the RTT is dynamic and depends on the current situation

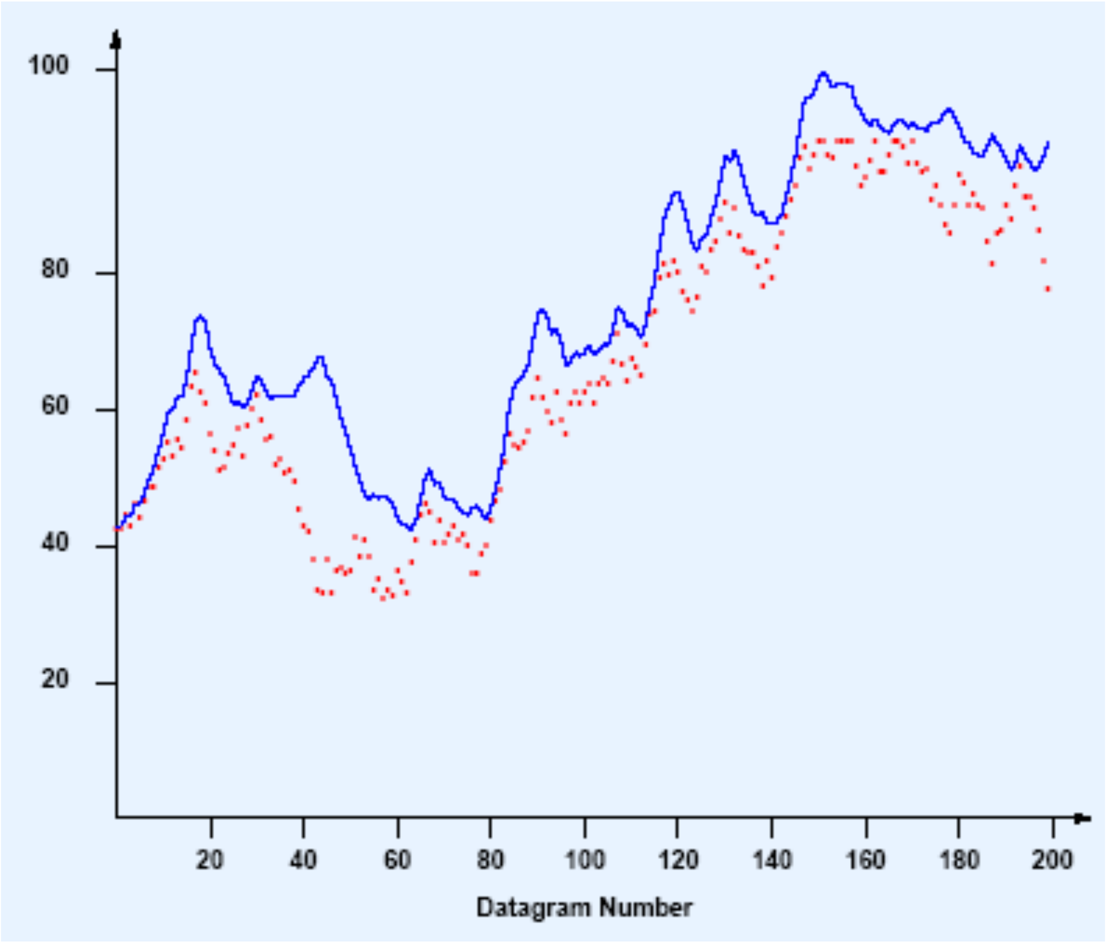
- TCP RTT estimation
  - **Time stamp** for data segments and ACKs,
    difference = measurement of the current RTT
  - **Average** and **deviation** from multiple measurements help to derive the timeout
  - Measurements for retransmitted segments must be discarded

# RTT Estimation

- RTT estimation
  - Every measurement results in one SampleRTT
  - **Moving average (Exponentially Weighted Moving Average, EWMA)**: EstimatedRTT = (1-$\alpha$) x EstimatedRTT + $\alpha$ x SampleRTT
  - For large $\alpha$, the estimated RTT is very sensitive to fast variations, for small $\alpha$, the estimated RTT is more stable, typical value: $\alpha$ = 0.125
- Mean variation
  - Again, measured as moving average (similar to standard deviation)
  - DevRTT = (1-$\beta$) x DevRTT + $\beta$ x |SampleRTT-EstimatedRTT|
  - Typical value: $\beta$ = 0.25
- Timeout
  - Based on estimated RTT and estimated variation:
  - TimeoutInterval = EstimatedRTT + 4 x DevRTT
  - Timeout Backoff: if timeout is triggered, it will be doubled until a new SampleRTT is available

# RTT Estimation

- Example






Technische Universität Berlin

Telecommunication Networks Group

# Flow and Congestion Control

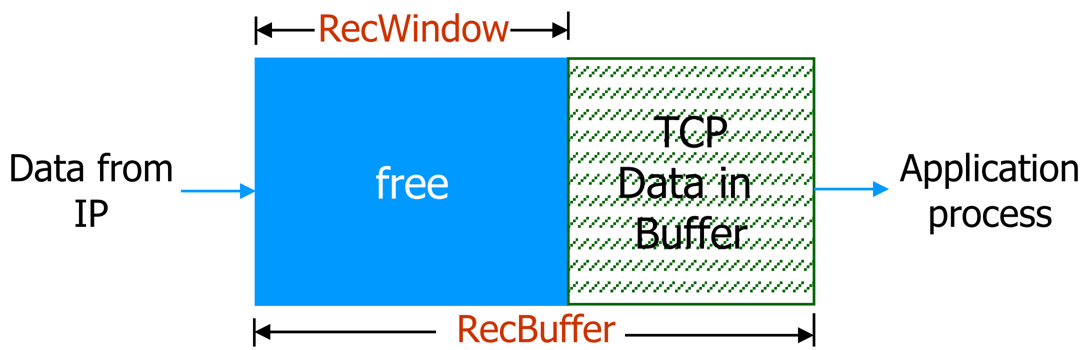
# Flow and Congestion Control

- Flow Control
  - Mechanism to enable the **receiver** to control the rate at the sender to avoid overload at the receiver
  - Explicit by notifying the sender about the current **window size**

- Congestion Control
  - Mechanism to enable the **network** to control the rate at the sender to avoid overload of the network (data rates, buffers)
  - Can be **explicit** but mostly uses **implicit** signals

# Principles of TCP Flow Control

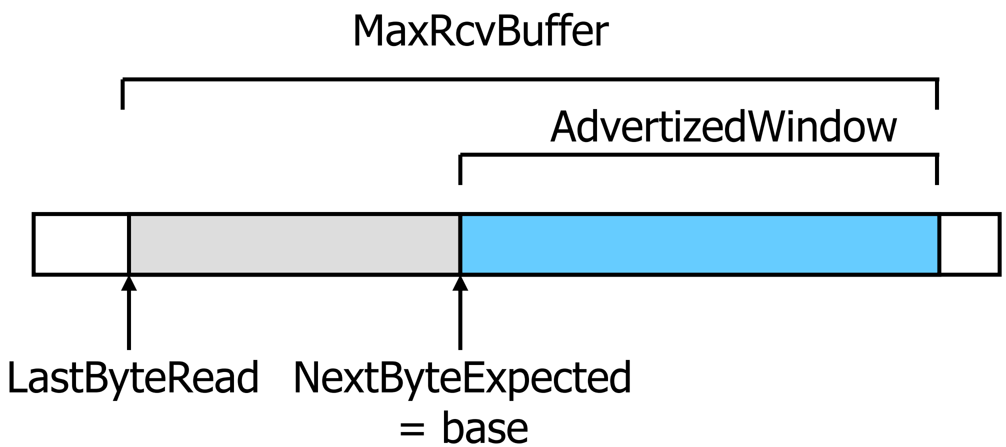
- The receiver has a buffer, IP adds newly received data to the buffer, application reads from buffer to free buffer space
- The amount of currently available free buffer is signaled to the sender




- The sender has a buffer, the application writes new data to the buffer, IP reads from the buffer and sends it to the receiver, buffer space is released when acknowledged
- The application is blocked if buffer is full
- This way, the receiver limits the max. amount of data to be transmitted until acknowledged

# TCP Flow Control

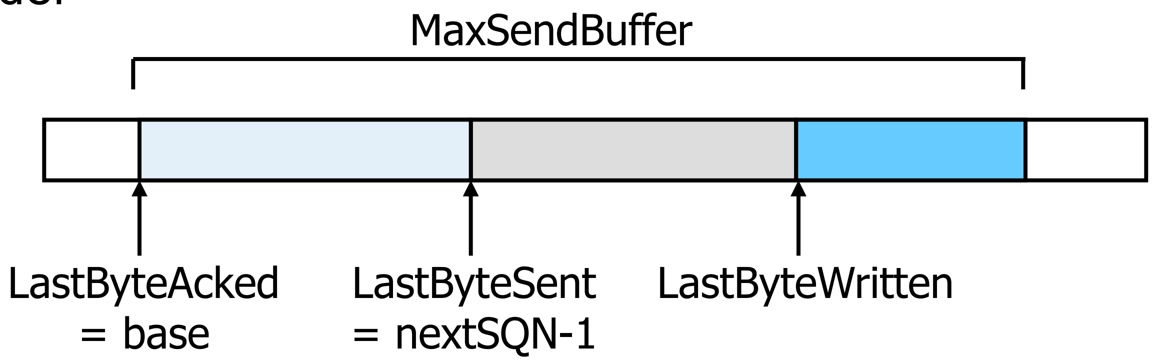
- Buffer at receiver




  - LastByteRead: last byte that was delivered to the application
  - NextByteExpected: next expected byte
  - MaxRcvBuffer: total available buffer space
  - **AdvertizedWindow** = MaxRcvBuffer - ((NextByteExpected - 1) - LastByteRead): available buffer space, will be advertised to sender

# TCP Flow Control

- Buffer at sender



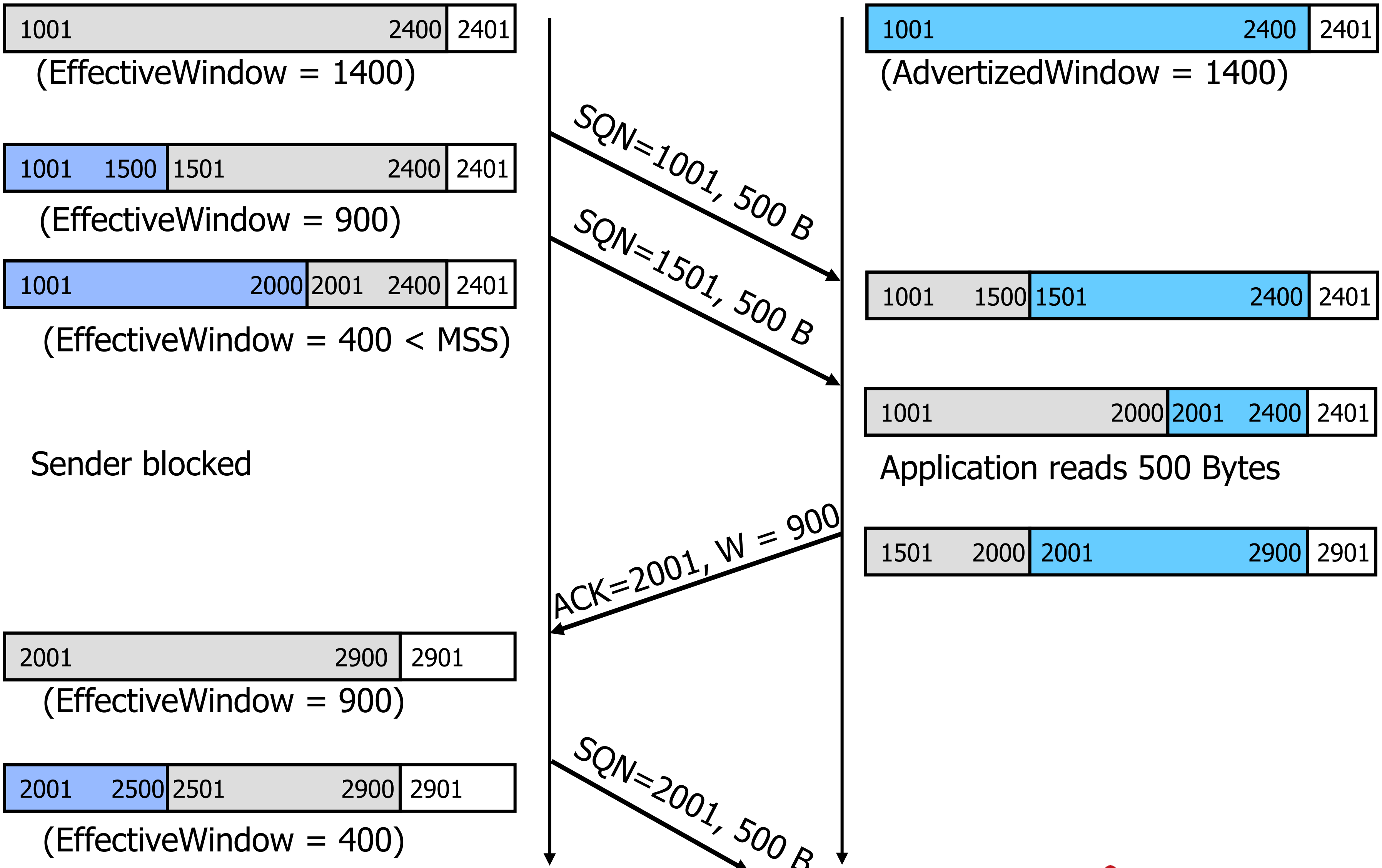

  - LastByteAcked: last acknowledged byte
  - LastByteSent: last sent byte
  - LastByteWritten: last byte written by application
  - MaxSendBuffer: total available buffer space
  - **EffectiveWindow** = AdvertizedWindow - (LastByteSent-LastByteAcked)
  - Sender only sends if EffectiveWindow > 0
  - Application only writes if LastByteWritten - LastByteAcked ≤ MaxSendBuffer

# TCP Flow Control

- Comments on TCP flow control
  - Initially, AdvertizedWindow is to be set as large as possible
  - As soon as AdvertizedWindow = 0, **probing segments** of size 1 will be sent, otherwise no ACKs will be triggered that inform the sender about larger AdvertizedWindow
  - Avoiding the **Silly Window Syndrome**
    - Segments with only few payload data are very inefficient
    - If buffer is full, only small segments are sent and circulate between sender and receiver, i.e., they stay in the system
  - Solution 1: periodically try larger segments
  - Solution 2: wait until AdvertizedWindow is large enough again
    - **MSS (Maximum Segment Size)**, Default is 536 Byte
    - When receiver indicated AdvertizedWindow = 0, it waits until it can announce AdvertizedWindow ≥ MSS

MaxSendBuffer=MaxRcvBuffer=1400 Bytes
MSS = 500 Bytes

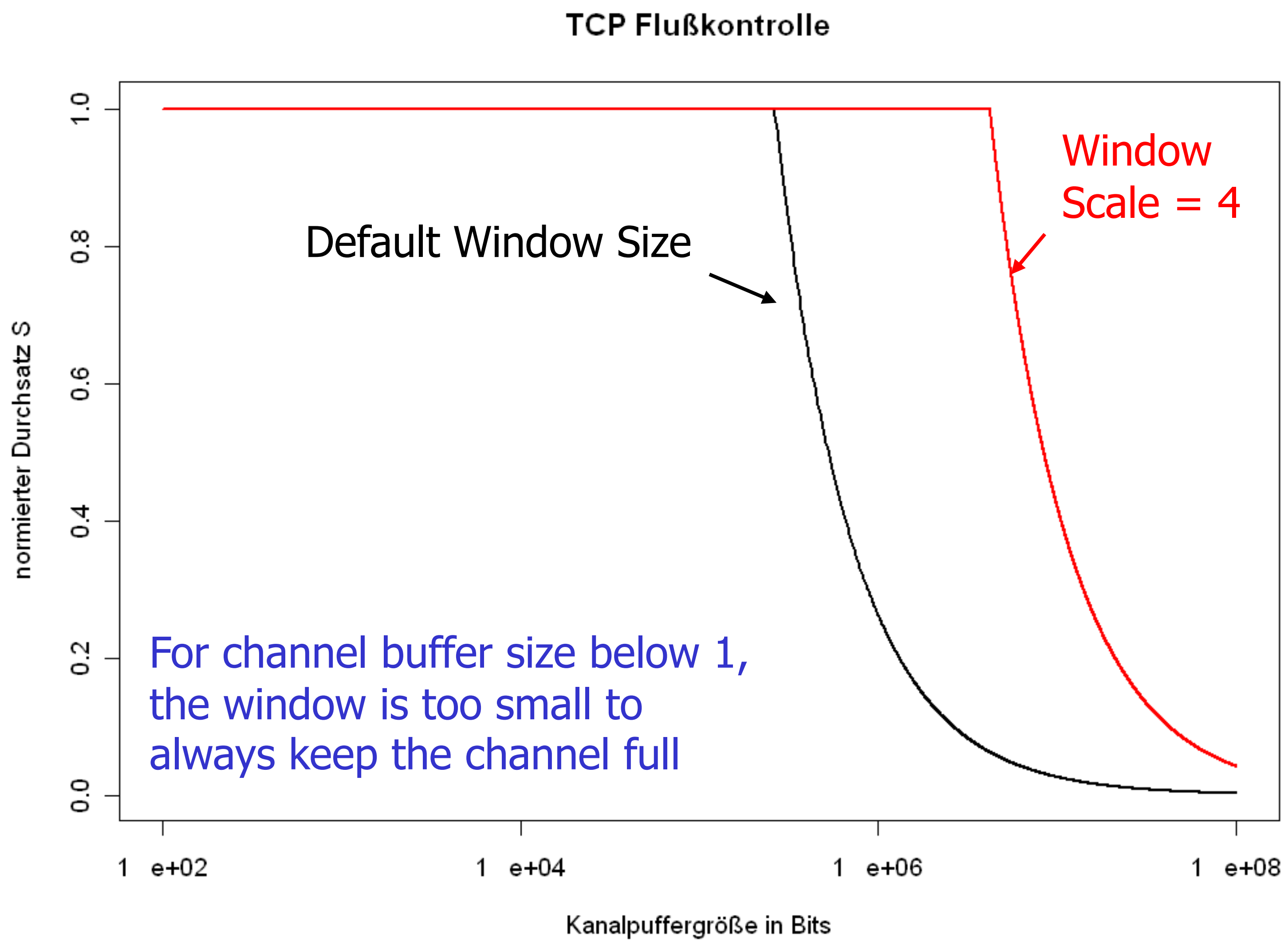
# TCP Flow Control: Example



# TCP Flow Control

- Is the window large enough?
  - Possibility of ambiguities due to re-used segment numbers?
    - AdvertizedWindow is 16 Bit, thus, the window covers $2^{16}$ Byte
    - Fundamental condition for window protocols is fulfilled: $2^{32} >> 2 \cdot 2^{16}$
  - Time until overflow of sequence numbers
    - At 10 Mbps: 57 minutes
    - At 1 Gbps: 34 seconds
  - So, how to keep the pipe full?
    - We will see (next slide) that if the bandwidth-delay-product is very large, the maximum window is not sufficient
    - TCP introduced a **Window-Scale-Faktor** $F \leq 14$ in the TCP options field (RFC 1323), now the window size is AdvertizedWindow $\cdot\ 2^F$
    - Note: it still holds $2^{32} > 2 \cdot 2^{30}$

# TCP Flow Control



# TCP Congestion Control

- Overview
  - TCP sender used received ACKs to determine the best send rate
  - Sender uses the **CongestionWindow**, which is combined with flow control information to set the used **EffectiveWindow**:
    - MaxWindow = Min(CongestionWindow, AdvertizedWindow)
    - EffectiveWindow = MaxWindow - (LastByteSent-LastByteAcked)
  - The data rate will be about CongestionWindow/RTT
  - By increasing the CongestionWindows, the sender increases the data rate – in order to get close to the maximum as supported by the network
  - For every lost segment (identified through 3 duplicated ACKs or the timeout), the CongestionWindows, and thus the data rate, will be reduced

# TCP Congestion Control

- TCP uses 3 mechanisms
  - **Slow Start**: when starting a connection, the sender increases the CongestionWindow starting with one MSS exponentially until it experiences a lost segment (indicated by 3 duplicated ACKs)
  - Then it changes to **AIMD (Additive Increase, Multiplicative Decrease)**: the CongestionWindow is halved and then linearly increased until again 3 duplicated ACKs are received
  - Then again AIMD ...
  - **Conversative reaction** after timeout: now slow start again until half of the last CongestionWindows, then AIMD

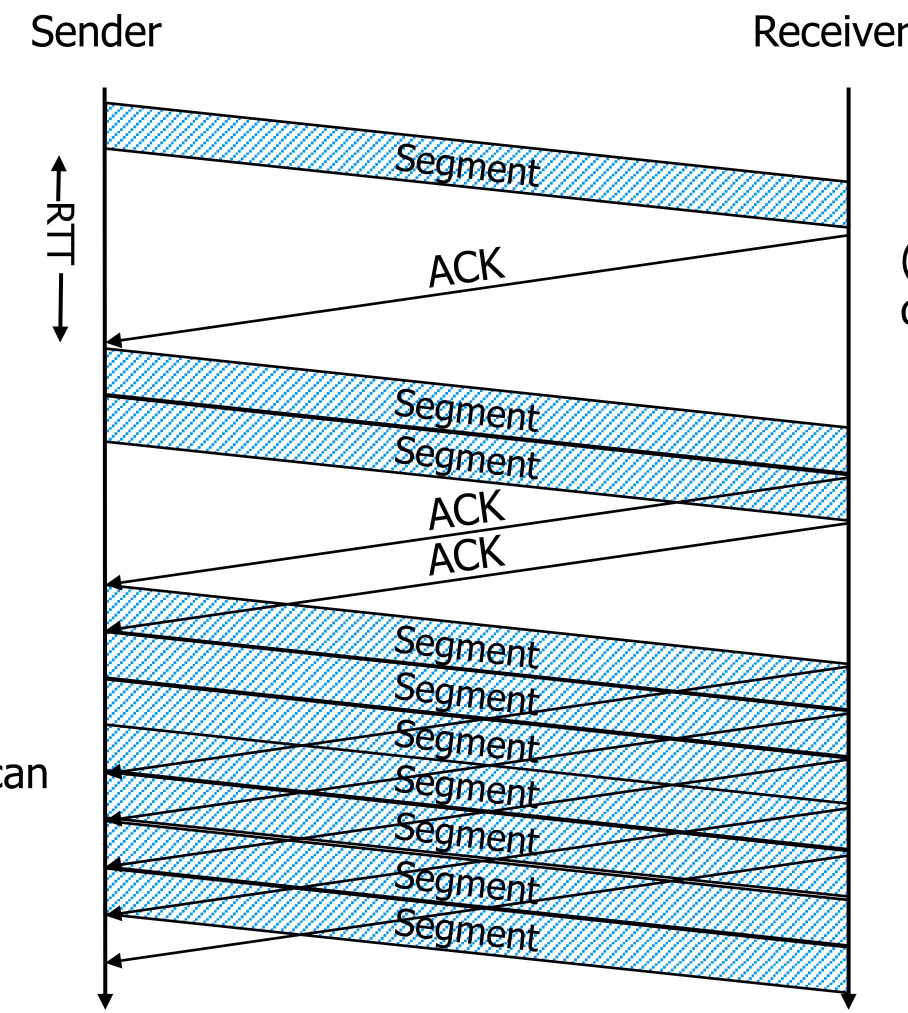
# TCP Congestion Control

- More details:
  - Slow Start
    - Set CongestionWindow = MSS
    - For every ACKs: CongestionWindow *= 2
      (this realizes exponential increase)
    - Until threshold, then additive increase (threshold is initially intinite)
  - After receiving 3 duplicated ACKs:
    - Multiplicative decrease:
      Threshold = CongestionWindow/2; CongestionWindow /= 2
    - Additive increase: for every ACK CongestionWindow += α
      - α ≈ MSS: this realizes linear increase by one MSS per RTT
  - After timeout
    - Threshold = CongestionWindow/2; CongestionWindow = MSS

# TCP Congestion Control

- Slow start example

CongestionWindow = 1 MSS




(we ignore the delayed ACK here)

CongestionWindow = 2 MSS

CongestionWindow = 4 MSS

(the pipe is filled now, sender can send without waiting time)

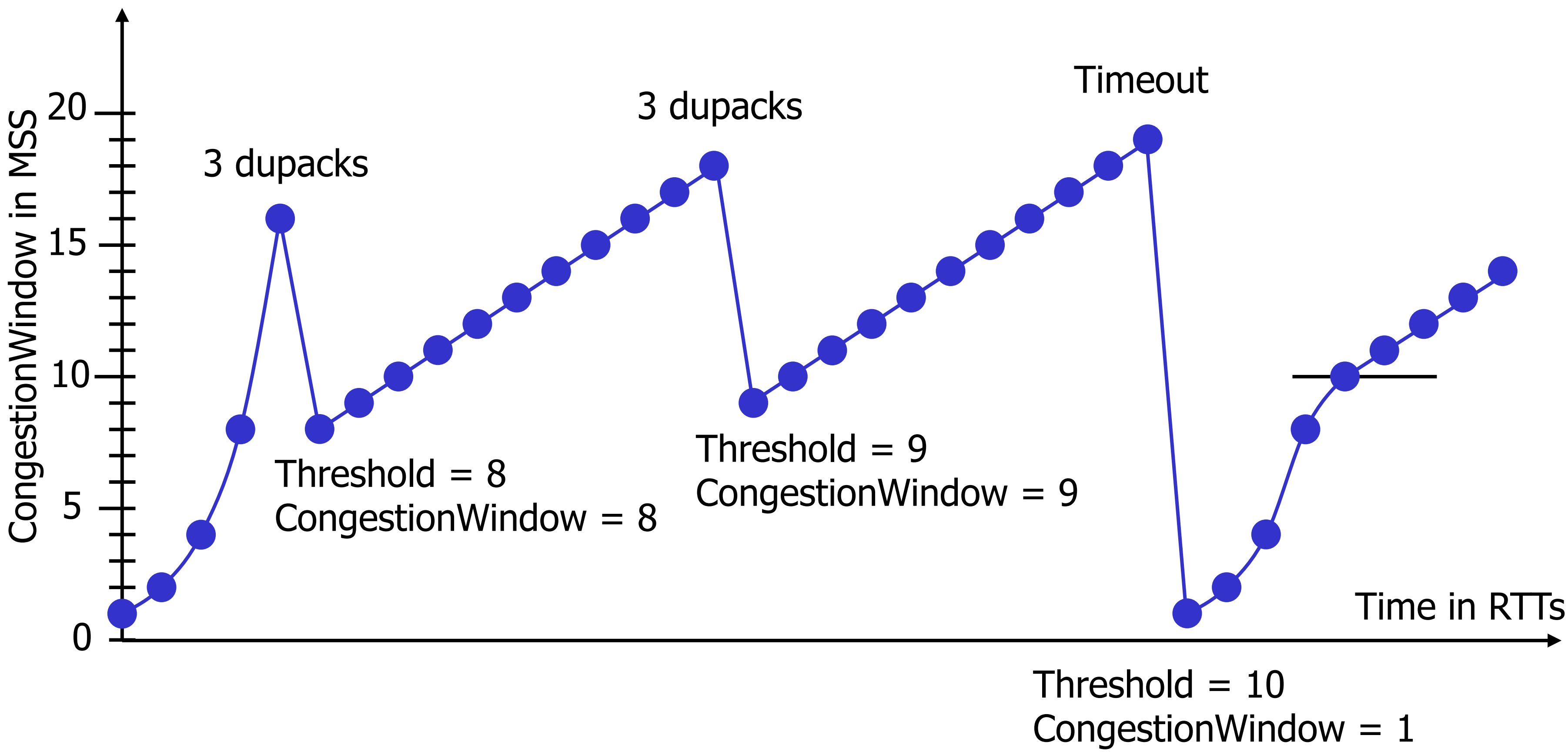
CongestionWindow = 8 MSS



Telecommunication Networks Group

# TCP Congestion Control

- Example

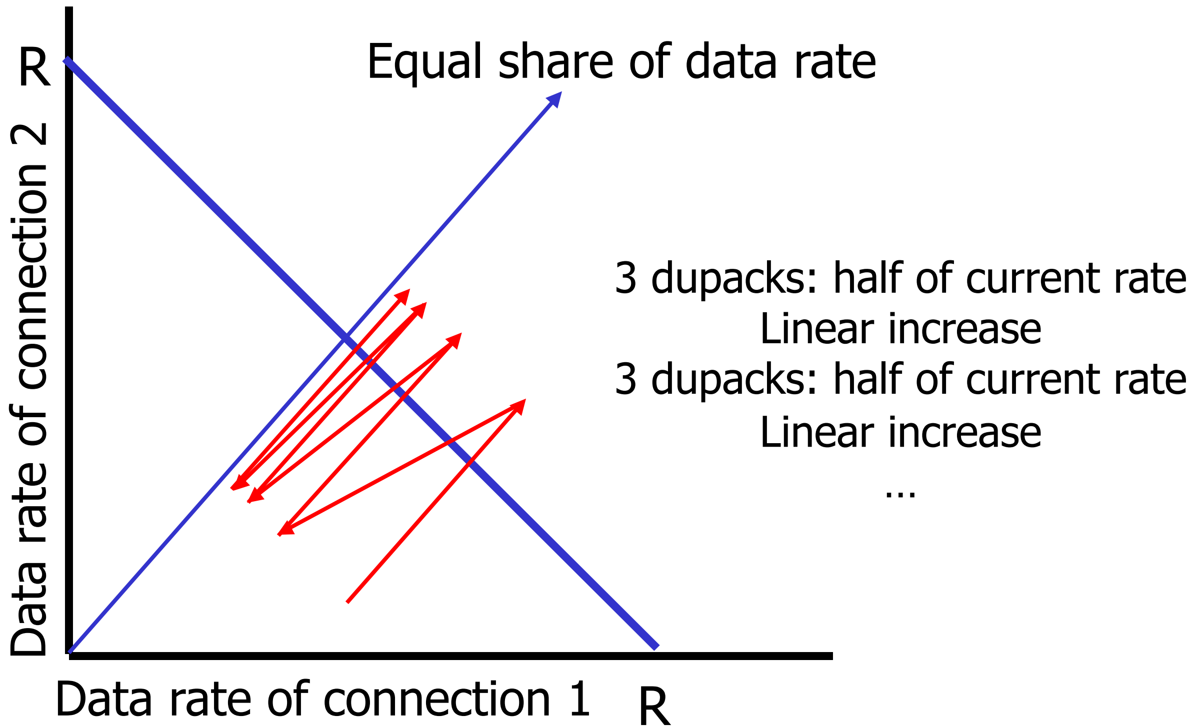


# TCP Congestion Control

- Resulting throughput
  - Assumption: only AIMD, for long lasting connections, slow start may be ignored
  - CongestionWindow will oscillate between max window W and half of it W/2
  - Data rate will be between W/RTT and ½ W/RTT
  - On average ¾ W/RTT

# Fairness of TCP Congestion Control

- Scenario
  - 2 TCP connections share the date rate R of a channel
  - Fairness: each connection should get R/2
  - Again, we ignore slow start for long lasting connections



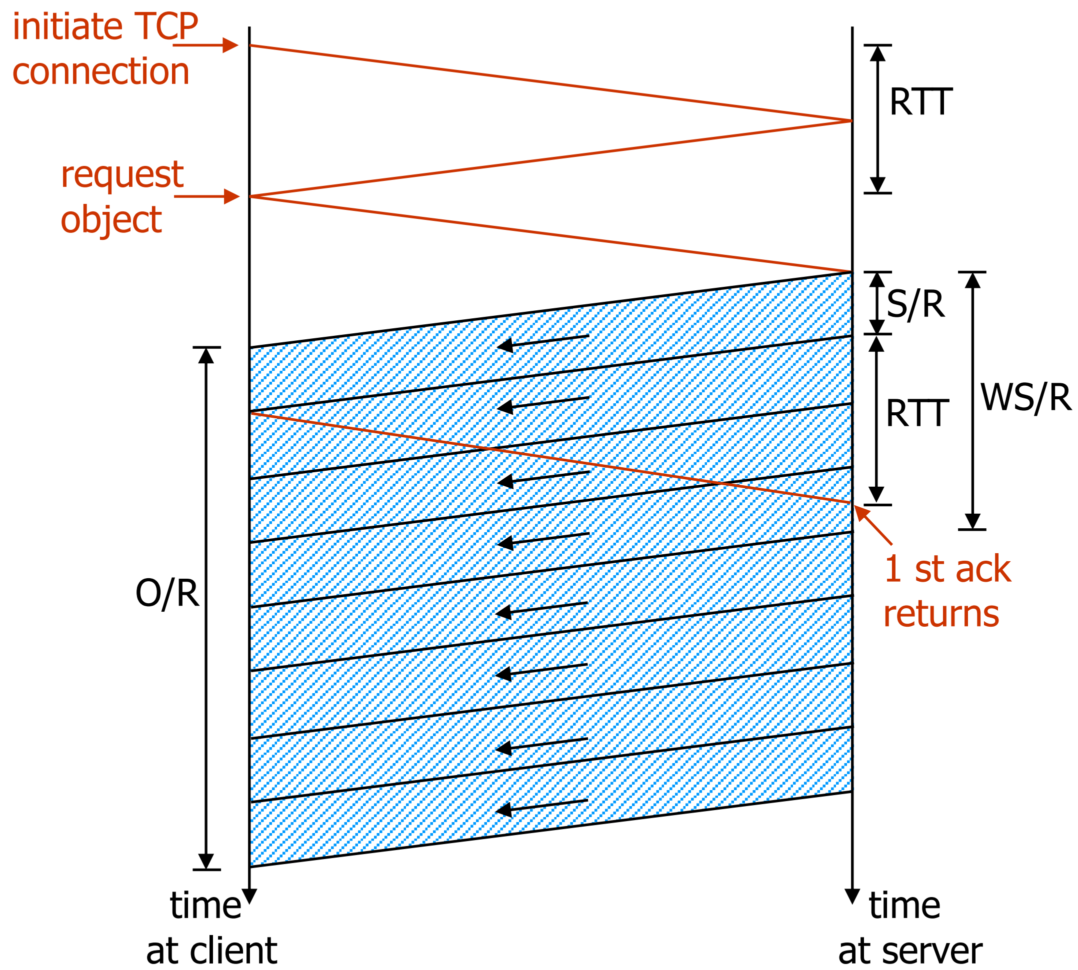
# Performance Analysis

# TCP Performance Analysis

- How long does it take to transfer an object via TCP
  - Depends on object size, data rate, propagation delay, delays due to protocol mechanisms, …
  - Particularly slow start may have a significant influence
  - Assumptions
    - No bit errors, no packet loss, constant data rate and delays, ACKs are sent instantaneously, no protocol processing time, window size of flow control is always sufficiently large
  - Notation
    - S: MSS in Bit
    - O: object size in Bit
    - R: date rate in Bit/s
    - RTT: round trip time in s
    - W: window size in MSS
  - We start looking at a fixed window size, then we study the impact of slow start

# Fixed window size

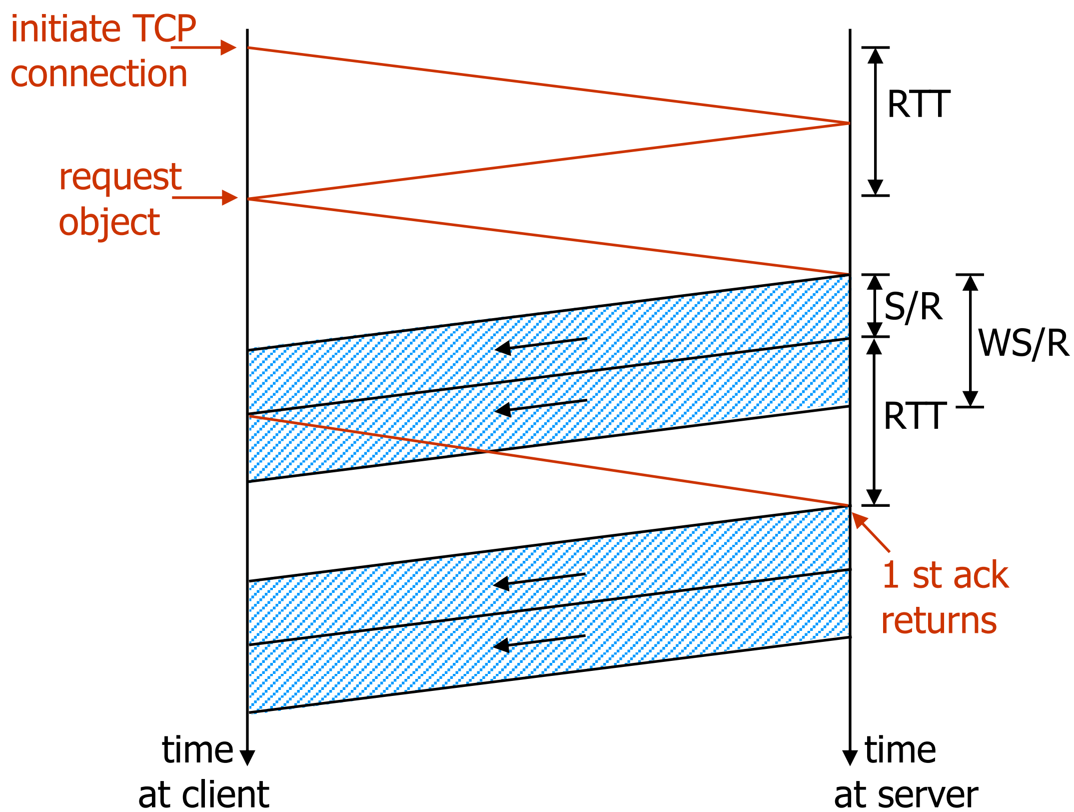
- First case
  - Window sufficiently large: WS/R > RTT + S/R
  - Delay = **2RTT + O/R**



# Fixed window size

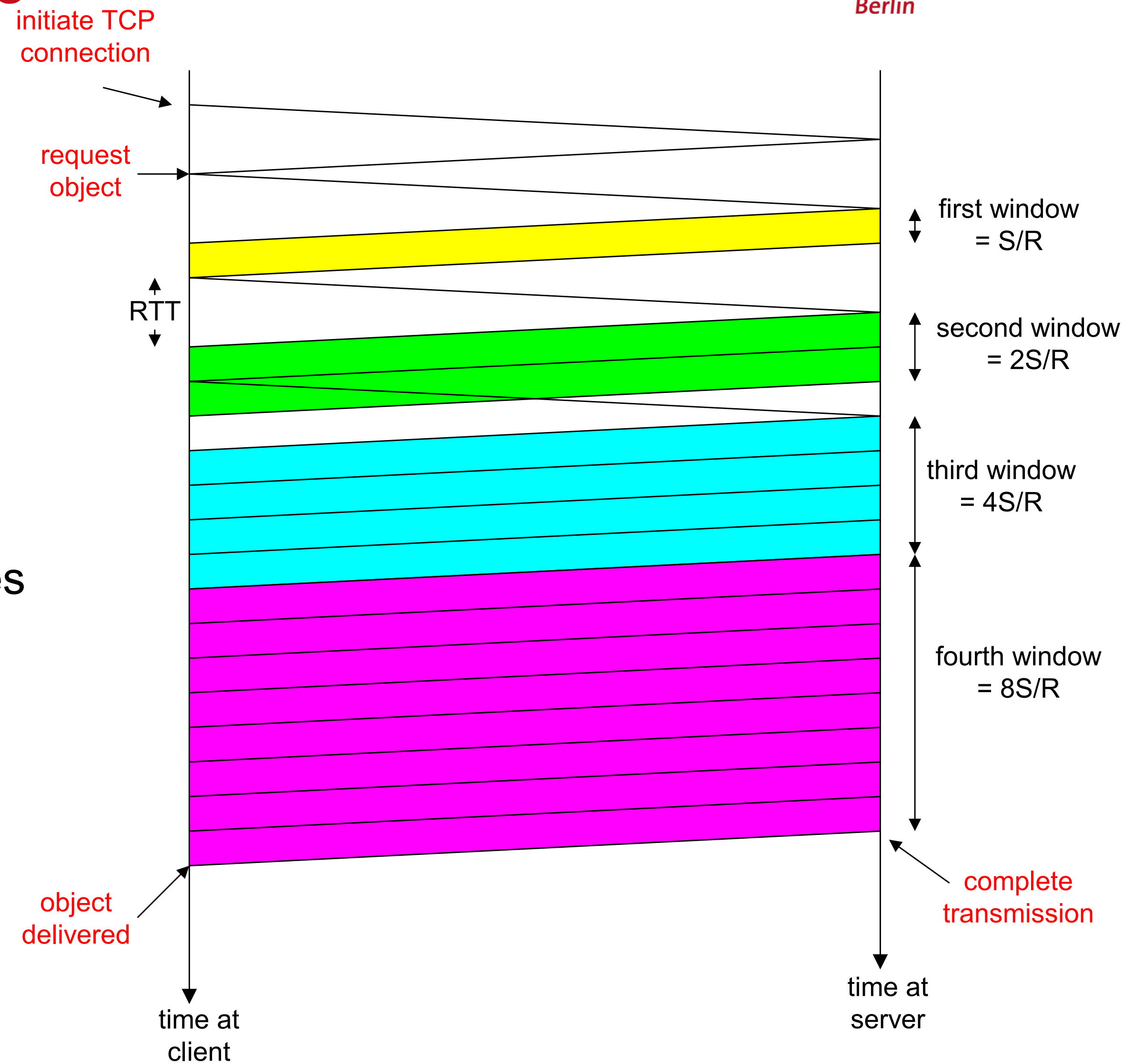
- Second case
  - Window too small:
    WS/R < RTT + S/R
  - Delay =
    **2RTT + O/R + (K-1)[S/R + RTT - WS/R]**
  - K is the number of windows to transmit the object:

$$K = \left\lceil \frac{O}{WS} \right\rceil$$



# Window grows according to slow start

- Growing window
  - 2RTT for connection setup
  - O/R for sending of O
  - O/S segments
  - K windows
  - P slow start waiting times
  - Example:
    - O/S = 15
    - K = 4
    - P = 2



# Window grows according to slow start

- Delay = 2 RTT + O/R + slow start waiting times
- Slow start waiting time in k-th window
  - $2^{k-1}$ S/R = transmission time in k-th window
  - S/R + RTT = time until first ACK
  - max[S/R + RTT - $2^{k-1}$ S/R, 0] = waiting time in k-th window
- Using P = number of slow start waiting times:

$$Delay = \frac{O}{R} + 2RTT + \sum_{k=1}^{P} waiting\ time\ k$$

$$= \frac{O}{R} + 2RTT + \sum_{k=1}^{P} \left[\frac{S}{R} + RTT - 2^{k-1}\frac{S}{R}\right]$$

$$= \frac{O}{R} + 2RTT + P\left[RTT + \frac{S}{R}\right] - (2^{P} - 1)\frac{S}{R}$$

# Window grows according to slow start

- Deriving the number of slow start waiting times
  - K = number of windows needed for the object

$$= \min\{k : 2^0 S + 2^1 S + \cdots + 2^{k-1} S \geq O\} = \min\{k : 2^0 + 2^1 + \cdots + 2^{k-1} \geq O / S\}$$

$$= \min\{k : 2^k - 1 \geq \frac{O}{S}\} = \min\{k : k \geq \log_2(\frac{O}{S} + 1)\} = \left\lceil \log_2(\frac{O}{S} + 1) \right\rceil$$

  - Q = number of slow start waiting times for an infinitely large object

$$= \max_k \left( \frac{S}{R} + RTT - 2^{k-1} \frac{S}{R} \geq 0 \right) = \max_k \left( 2^{k-1} \leq 1 + \frac{RTT}{S / R} \right) = \left\lfloor \log_2 \left( 1 + \frac{RTT}{S / R} \right) \right\rfloor + 1$$

  - Now, P = min(Q, K-1)

# Window grows according to slow start

- Result:

$$delay = \frac{O}{R} + 2RTT + P\left[RTT + \frac{S}{R}\right] - (2^P - 1)\frac{S}{R}$$

  - Equation contains the product of P and RTT, thus, if RTT is large and/or many slow start waiting times are experienced, the delay increases significantly