

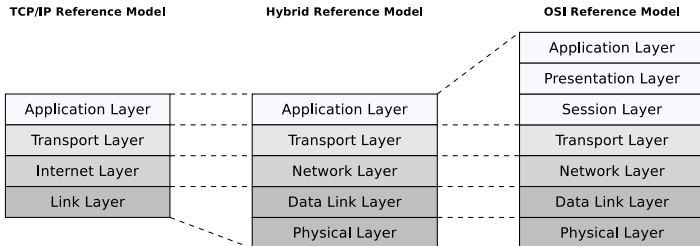


Prof. Dr. Christian Baun – 9th Slide Set Computer Networks – Frankfurt University of Applied Sciences – WS2425

- 2/50

# Transport Layer

- Functions of the Transport Layer
  - Contains **end-to-end protocols** for inter-process communication
  - In this layer, processes are addressed via **port numbers**
  - Application Layer data is split here into smaller **segments**



Exercise sheet 5 repeats the contents of this slide set which are relevant for these learning objectives

- Devices: Gateway
- Protocols: TCP, UDP

## Challenges for Transport Layer Protocols

- The Network Layer protocol IP works **connectionless**
  - IP packets are *routed* independently of each other to the destination site
  - Advantage: Little overhead
- Drawbacks from the Transport Layer perspective
  - IP packets can get **lost** or **discarded** because the TTL has expired
  - IP packets often arrive at the destination site in the **wrong order**
  - **Multiple copies** of IP packets arrive at the destination
- Reasons:
  - Large networks are not static  $\implies$  their infrastructure constantly changes
  - Transmission media can fail
  - The workload varies and therefore the networks' delay
- These problems are common in computer networks
  - Depending on the application, transport protocols need to compensate these drawbacks

# Characteristics of Transport Layer Protocols

- Desired characteristics of Transport Layer protocols include. . .
  - guaranteed data transmission
  - ensuring the correct delivery order
  - support for data transmissions of any size
  - the sender must not overload the network
    - It must be able to adjust its own data flow (data rate)  $\implies$  flow control
  - the receiver should be able to control the transmission rate of the sender  
 $\implies$  congestion control
- Transport Layer protocols can convert the networks' negative characteristics into the (positive) characteristics that are required for inter-process communication (end-to-end-communication)
- The most common used Transport Layer protocols:
  - **UDP**
  - **TCP**
- The addressing is realized via **sockets**

# Addressing in the Transport Layer

- Every application which uses TCP or UDP, has a **port number** assigned
  - It specifies which service is accessed
  - For TCP and UDP, the size of port numbers is 16 bits
    - Thus, the range of possible port numbers is from 0 to 65,535
- In principle, port numbers can be assigned as wished
  - Conventions exist, that specify which ports are used by common used applications

Port number	Service	Description
21	FTP	File transfer
22	SSH	Encrypted terminal emulation (secure shell)
23	Telnet	Terminal emulation for remote control of computers
25	SMTP	E-mail transfer
53	DNS	Resolution of domain names into IP addresses
67	DHCP	Assignment of the network configuration to clients
80	HTTP	Webserver
110	POP3	Client access to E-mail server
143	IMAP	Client access to E-mail server
443	HTTPS	Webserver (encrypted)
993	IMAPS	Client access to E-mail server (encrypted)
995	POP3S	Client access to E-mail server (encrypted)

- The table only contains a small selection of well-known port numbers

## Ports (2/2)

- The port numbers are divided into 3 groups:
    - 0 to 1023 (*Well Known Ports*)
      - These are permanently assigned to applications and commonly known
    - 1024 to 49151 (*Registered Ports*)
      - Application developers can register port numbers in this range for own applications
    - 49152 to 65535 (*Private Ports*)
      - These port numbers are not registered and can be used freely
  - Different applications can use identical port numbers inside an operating system at the same time, if they communicate via different transport protocols
  - In addition, some applications exist, which implement communication via TCP and UDP via a single port number
  - Example: Domain Name System – DNS (see slide set 10)
- 
- Well Known Ports and Registered Ports are assigned by the Internet Assigned Numbers Authority (IANA)
  - In Linux/UNIX systems, the configuration file `/etc/services` exists
    - Here, the applications (services) are mapped to specific port numbers
  - In Windows systems: `%WINDIR%\system32\drivers\etc\services`

# Sockets

- **Sockets** are the platform-independent, standardized **interface** between the implementation of the network protocols in the operating system and the applications
- **A socket consists of a port number and an IP address**
- Stream Sockets and datagram sockets exist
  - **Stream sockets** use the connection-oriented TCP
  - **Datagram sockets** use the connectionless UDP

## Tool(s) to monitor the open ports and sockets with...

- Linux/UNIX: netstat, lsof, ss (part of the iproute2 package), nmap
- Windows: netstat, tcpvcon, TCPView, PowerShell (Get-NetTCPConnection, Get-NetUDPEndpoint)

## Alternatives for sockets to implement inter-process communication (IPC)

Pipes, message queues and shared memory  $\implies$  see Operating Systems course

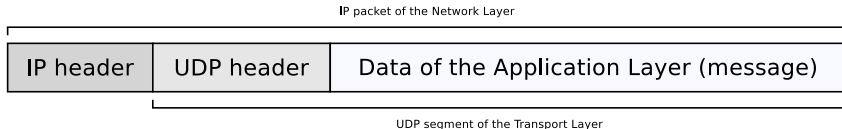


# User Datagram Protocol (UDP)

- **Connectionless Transport Layer protocol**
  - Transmissions take place without previous connection establishment
- More simple protocol in contrast to the connection-oriented TCP
  - Only responsible for addressing of the segments
  - Does not secure the data transmission
- The receiver does not acknowledge transmissions at the sender
  - Segments can get lost during transmission
- Depending on the application (e.g. video streaming) this is accepted
  - If a TCP segment (and therefore some image information) gets lost during the transmission of a video, it is requested again
    - A drawback would be dropouts
  - To compensate for this, a buffer at the receiver site is required
    - Especially video telephony software tries to keep the buffer as small as possible because they cause delays
  - If UDP is used for video transmission or video telephony, the only consequence of losing a segment is losing an image

# User Datagram Protocol (UDP)

- Maximum size of an UDP segment: 65,535 Bytes
  - Reason: The size of the **length** field inside the UDP header, which contains the segment length, is 16 bits
    - The maximum representable number with 16 bits is 65,535
  - UDP segments of this size are transmitted in fragments by IP



UDP standard: RFC 768 from 1980  
<http://tools.ietf.org/rfc/rfc768.txt>

# Structure of UDP Segments

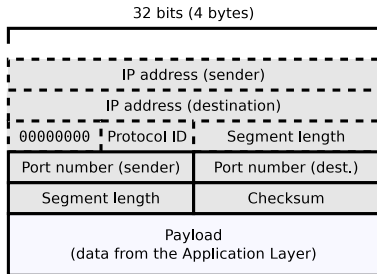
- The UDP header consists of 4 fields, each 16 bits in size
    - **Port number (sender)**
      - The field can stay empty (value 0), if no response is required
    - **Port number (destination)**
    - **Length** of the complete segment (without pseudo-header)
    - **Checksum** of the complete segment (including pseudo-header)
  - A pseudo-header is created, which includes the IP addresses of sender and destination, as well as some Network Layer information
    - Protocol ID of UDP = 17
  - The pseudo-header is not transmitted
    - But it is used for the checksum calculation
- The diagram illustrates the structure of a UDP segment. It is divided into two main parts: a pseudo-header and the segment itself.

**Pseudo-header (32 bits / 4 bytes):** This section is used for checksum calculation but is not transmitted. It contains:

  - IP address (sender)
  - IP address (destination)
  - Protocol ID (represented by the value 00000000)
  - Segment length

**Segment:** This is the actual data being transmitted. It contains:

  - Port number (sender)
  - Port number (destination)
  - Segment length
  - Checksum
  - Payload (data from the Application)

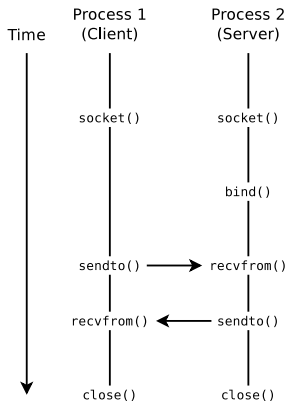


Remember NAT from slide set 8...

If a NAT device (Router) is used, this routing device also needs to recalculate the checksums in UDP segments when doing IP address translations



## Connectionless Communication via Sockets – UDP



- **Client**

- Create socket (`socket`)
- Send (`sendto`) and receive data (`recvfrom`)
- Close socket (`close`)

- **Server**

- Create socket (`socket`)
- Bind socket to a port (`bind`)
- Send (`sendto`) and receive data (`recvfrom`)
- Close socket (`close`)

UDP sockets can be implemented as **blocking** (standard) and **non-blocking**

- `sendto` applied to a non-blocking socket causes an error message `EAGAIN` or `EWouldBlock` if the send buffer has no sufficient free capacity but the process is not blocked. The process can later try to write data into the send buffer
- `recvfrom` applied to a non-blocking socket causes an error message `EAGAIN` or `EWouldBlock` if the receive buffer has no data to be fetched but the process is not blocked. The process can later try to fetch data from the receive buffer

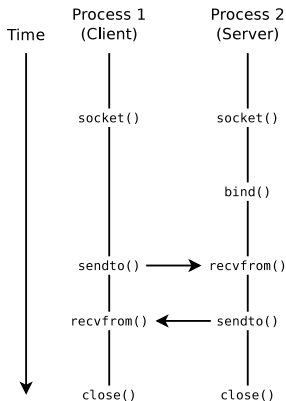
## Sockets via UDP – Example (Server)

```

1  #!/usr/bin/env python
2  # Server: Receives a message via UDP
3
4  import socket                                # Import module socket
5
6  # For all interfaces of the host
7  HOST = ''                                    # '' = all interfaces
8  PORT = 50000                                # Port number of server
9
10 # Create socket and return socket descriptor
11 sd = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
12
13 try:
14     sd.bind((HOST, PORT))                    # Bind socket to port
15     while True:
16         # Receive data
17         data = sd.recvfrom(1024)
18         # Print out received data
19         print 'Received:', repr(data)
20 finally:
21     sd.close()                                # Close socket

```

```
$ python udp_server.py
```

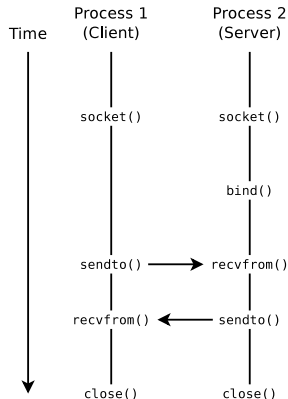


## Sockets via UDP – Example (Client)

```
1 #!/usr/bin/env python
2 # Client: Sends a message via UDP
3
4 import socket                # Import module socket
5
6 HOST = 'localhost'          # Hostname of Server
7 PORT = 50000                 # Port number of Server
8 MESSAGE = 'Hello World'     # Message
9
10 # Create socket and return socket descriptor
11 sd = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
12
13 # Send message to socket
14 sd.sendto(MESSAGE, (HOST, PORT))
15
16 sd.close()                   # Close socket
```

```
$ python udp_client.py
```

```
$ python udp_server.py
Received: ('Hello World', ('127.0.0.1', 39834))
```



# Transmission Control Protocol (TCP)

- **Connection-oriented Transport Layer protocol**
- Makes connections via IP reliable in a way, which is desired or required for many applications
- Ensures that segments reach their destination **completely** and in the **correct order**
  - Lost or unacknowledged TCP segments are requested by the receiver at the sender and sent again
- TCP connections are opened and closed like files
  - Equal to files, the position in the data stream is precisely specified

TCP standard: RFC 793 from 1981  
<http://tools.ietf.org/rfc/rfc793.txt>



# Sequence Numbers in TCP

- TCP treats payload as an unstructured, but ordered data stream
- **Sequence numbers** are used for numbering the bytes in the data stream
  - The sequence number of a segment is the position of the segment's first byte in the data stream
- Example
  - The sender splits the Application Layer data stream into segments
    - Length of data stream: 5,000 bytes
    - MSS: 1,460 bytes



## Some key figures...

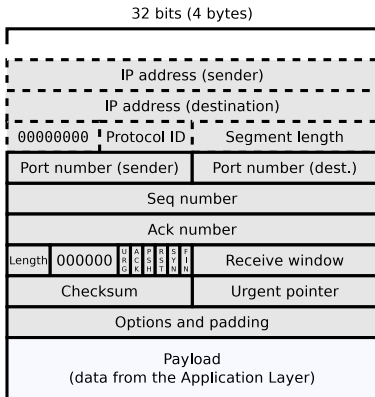
**Maximum Transfer Unit (MTU):** Maximum size of the IP packets

MTU of Ethernet = 1,500 bytes, MTU of PPPoE (e.g. DSL) = 1,492 bytes

**Maximum Segment Size (MSS):** Maximum segment size

MSS = MTU - 40 bytes for IPv4 header and TCP header

## Structure of TCP Segments (1/5)



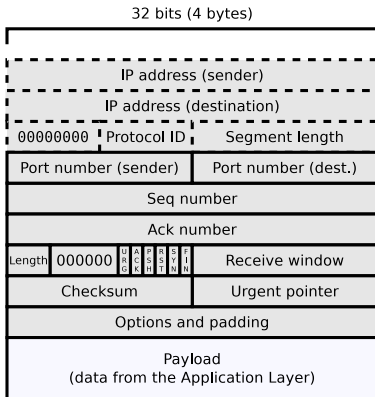
- A TCP segment can contain a maximum of 64 kB payload (data of the Application Layer)
  - Usually, segments are smaller ( $\leq 1500$  bytes for Ethernet)
- The header of TCP segments is more complex compared with UDP segments

## Overhead

- Size of the TCP header (without the options field): just 20 bytes
- Size of the IP header (without the options field): also just 20 bytes

⇒ The overhead caused by the TCP and IP headers is small for an IP packet with a size of several kB

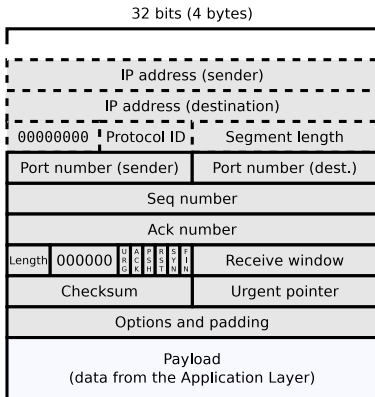
## Structure of TCP Segments (2/5)



- One field contains the port number of the sender process
- Another field contains the port number of the process which is expected to receive the segment
- **Seq number** contains the sequence number of the current segment
- **Ack number** contains the sequence number of the next expected segment

- The **length** field specifies the size of the TCP header in 32-bit words to tell the receiver where the payload starts in the segment
  - The field is required, because the field *options and padding* can have a variable length (a multiple of 32 bits)

## Structure of TCP Segments (3/5)



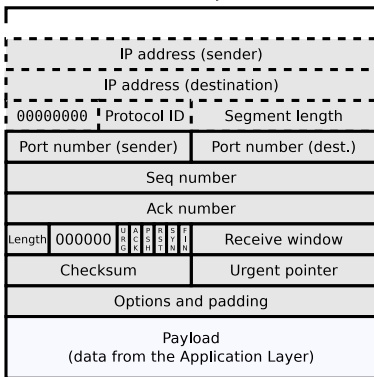
- The field 000000 is 6 bits long and not used
  - It contains per default value zero
- The 6 fields with a size of 1 bit each, are required for connection establishment, data exchange and connection termination
  - The functionality of these fields is described with the assumption that the fields contain the value 1  $\implies$  it is set

**URG** (Urgent) is not discussed in this course

- **ACK** (Acknowledge)
  - Specifies that the acknowledgement number in **Ack number** is valid
  - It is also used to acknowledge the receive of segments

## Structure of TCP Segments (4/5)

32 bits (4 bytes)



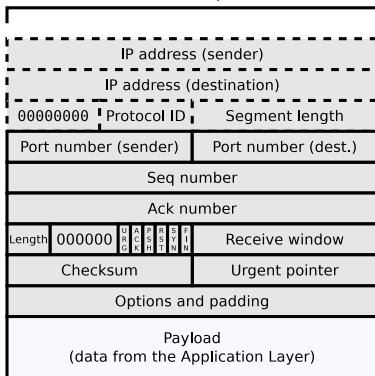
**PSH** (Push) is not discussed in this course

**RST** (Reset) is not discussed in this course

- **SYN** (Synchronize)
  - Requests the synchronization of the sequence numbers
  - That initiates the connection establishment
- **FIN** (Finish)
  - Requests the connection termination and indicates that the sender will not send any more payload

# Structure of TCP Segments (5/5)

32 bits (4 bytes)



- Just as with UDP, for each TCP segment, a pseudo header exists, which is not transmitted
  - But the pseudo header fields are used together with the regular TCP header fields and the payload to calculate the **checksum**
  - Protocol ID** of TCP = 6

The **urgent pointer** is not discussed in this course

The field **options and padding** must be a multiple of 32 bits and is not discussed in this course

Remember NAT from slide set 8...

If a NAT device (Router) is used, this routing device also needs to recalculate the checksums in TCP segments when doing IP address translations

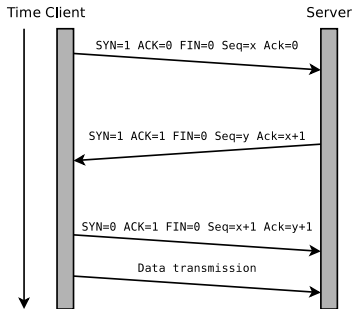
# Functioning of TCP

## You already know...

- Each segment has a unique **sequence number**
- The sequence number of a segment is the position of the segment's first byte in the data stream
- The sequence number enables the receiver to...
  - correct the order of the segments
  - sort out segments, which arrived twice
- The length of a segment is known from the IP header
  - This way, missing bytes in the data stream are discovered and the receiver can request lost segments
- To establish a connection, TCP uses a **three-way handshake**, where both communication partners exchange control information in three steps
  - This ensures that the communication partner exists and can accept data transmissions

# TCP Connection Establishment (three-way Handshake)

- The server waits passively for an incoming connection
- ❶ Client sends a segment with  $\text{SYN}=1$  as a request to synchronize the sequence numbers  
 $\Rightarrow$  *Synchronize*
- ❷ Server sends as confirmation a segment with  $\text{ACK}=1$  and requests with  $\text{SYN}=1$  to synchronize the sequence numbers too  
 $\Rightarrow$  *Synchronize Acknowledge*
- ❸ Client confirms with a segment with  $\text{ACK}=1$  that the connection is established  
 $\Rightarrow$  *Acknowledge*
- The initial sequence numbers ( $x$  and  $y$ ) are determined randomly
- No payload is exchanged during connection establishment!



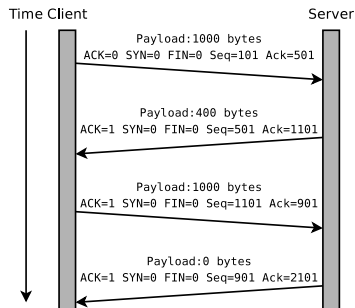


# TCP Data Transmission

To demonstrate a data transmission, **Seq number** (sequence number of the current segment) and **Ack number** (sequence number of the next expected segment) need particular values

- In our example at the beginning of the three-way handshake, the client's sequence number is  $x=100$  and the server's sequence number is  $y=500$
- After completion of the three-way handshake:  $x=101$  and  $y=501$

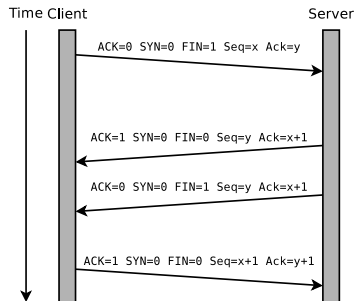
- 1 The client transmits 1000 bytes payload
- 2 Server acknowledges with  $ACK=1$  the received payload and requests with the Ack number 1101 the next segment. In the same segment, the server transfers 400 bytes of payload
- 3 The client transmits another 1000 bytes payload. And it acknowledges the received payload with the ACK bit set and requests with the Ack number 901 the next segment
- 4 Server acknowledges with  $ACK=1$  the received payload and requests with the Ack number 2101 the next segment



# TCP Connection Termination

- Connection termination is similar to the connection establishment
- Instead of the SYN bit set, the FIN bit is used to indicate that the sender will not transmit any more payload

- 1 The client sends the request for connection termination with  $\text{FIN}=1$
- 2 The server sends an acknowledgment with  $\text{ACK}=1$
- 3 The server sends the request for connection termination with  $\text{FIN}=1$
- 4 The client sends an acknowledgment with  $\text{ACK}=1$



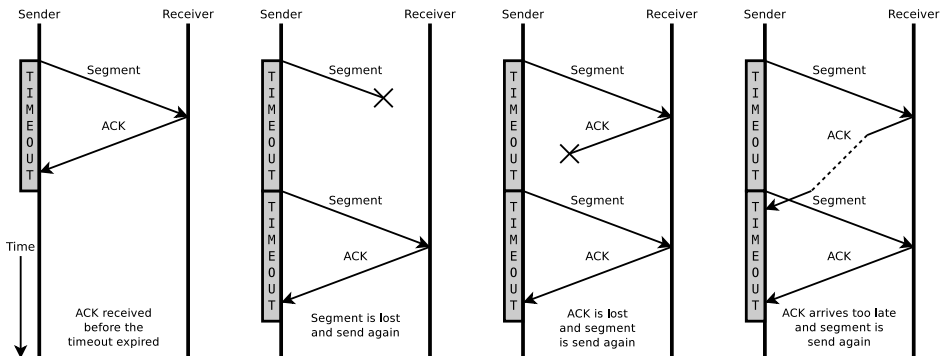
- No payload is exchanged during connection termination!

# Reliable Transmission through Flow Control

- Via Flow control, the receiver controls the **transmission speed** of the sender dynamically, and this way ensures the **completeness of the data transmission**
  - Receivers with a low performance should not be flooded with data they can not process fast enough
    - As result, data would be lost
  - During transmission, lost data is transmitted again
- Procedure: **Transmission retries**, when they are required
- Basic mechanisms:
  - **Acknowledgements** (ACK) as feedback (receipt)
  - **Timeouts**
- Concepts for flow control:
  - **Stop-and-Wait**
  - **Sliding Window**

# Stop-and-Wait

- After transmitting a segment, the sender waits for an ACK
  - If no ACK arrives in a certain time  $\Rightarrow$  timeout
  - Timeout  $\Rightarrow$  segment is sent again



- Drawback: Lesser throughput compared to the transmission-line capacitance

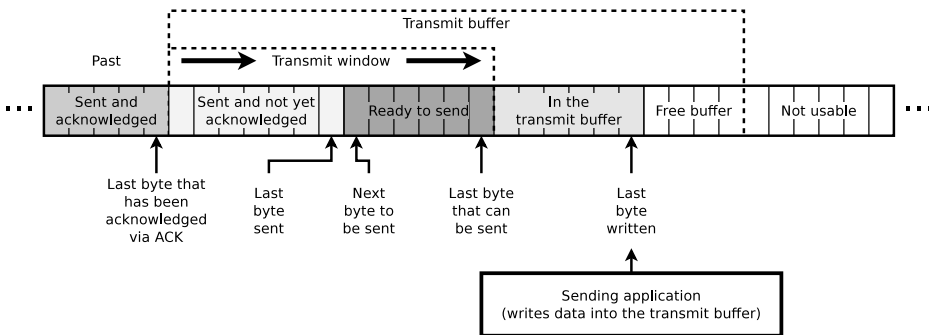
The **Trivial File Transfer Protocol** (RFC 783) operates according to the Stop-and-Wait principle

# Sliding Window

- A **window** allows the sender to transmit a certain number of segments before an acknowledgment is expected
  - Upon arrival of an acknowledgment, the transmit window is moved, and the sender can send further segments
    - The receiver can acknowledge several segments at once  
⇒ **cumulative acknowledgments**
  - If a timeout occurs, the sender transmits all segments in the window again
    - The sender sends everything again beginning from the last not acknowledged sequence number
- Objective: Better utilization of the line capacity and receiver capacity

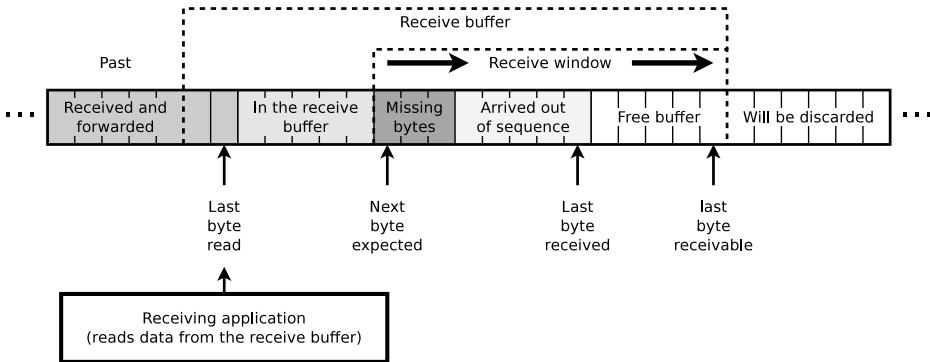
# Sliding Window – Method: Sender

- The transmit buffer contains data of the Application Layer, which...
  - has already been sent but not yet confirmed
  - is ready to be sent, but has not been sent up to now



## Sliding Window – Method: Receiver

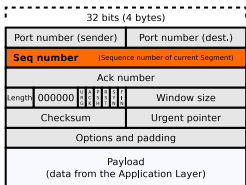
- The receive buffer contains data for the Application Layer, which...
  - is in the correct order, but has not been read
  - has been received out of sequence



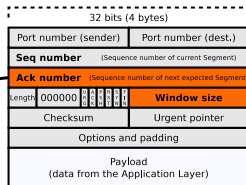
- The receiver informs the sender about the size of its receive window
  - This is important to avoid a buffer overflow!

# TCP Flow Control

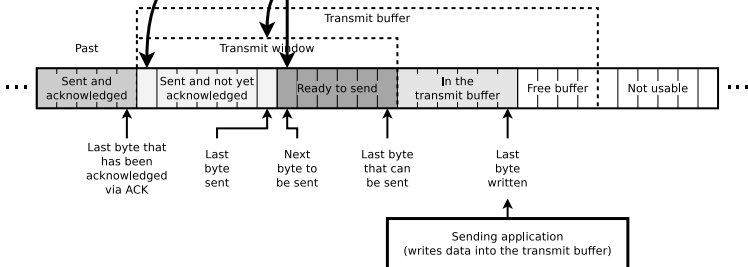
**Segment to be sent**



**Received segment**



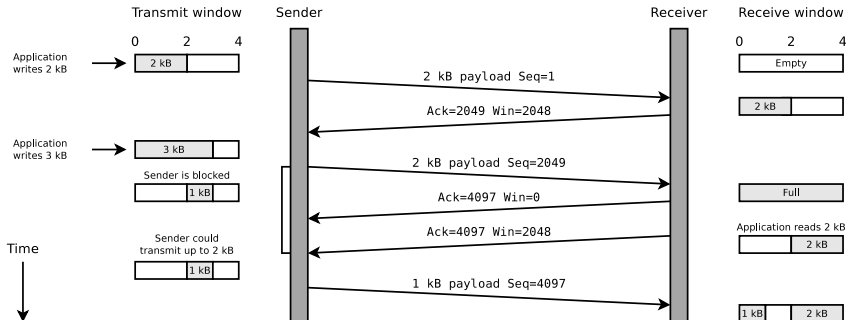
Receiver informs about the number of bytes it is able to receive





# Example of Flow Control in TCP

- The receiver informs the sender in every segment, how much free storage capacity its receive window has
- If the receive window has no free capacity, the sender is blocked until it gets informed by the receiver that free storage capacity exists
- If storage capacity in the receive window becomes free  $\Rightarrow$  **A segment with the current free storage capacity is sent**



# Silly Window Syndrome

- The **Silly window syndrome** is a problem where a large number of segments is send, which increases the protocol overhead
  - Scenario
    - A receiver is overloaded and its receive buffer is completely filled
    - Once the application has read a few bytes (e.g. 1 byte) from the receive buffer, the receiver sends a segment with the free storage capacity of the receive buffer
    - For this reason, the sender transmits a segment, which contains just 1 byte payload
    - Overhead: At least 40 bytes for the TCP/IP headers of each IP packet (Required are: 1 segment with the payload, 1 segment for the acknowledgement and eventually another segment which notifies about the current free storage capacity in the receive window)
  - Solution: **Silly window syndrome avoidance**
    - The receiver does not notify the sender about free storage capacity in the receive window before 25% of the receive buffer is free or a segment of size MSS can be received

# Reasons why Congestion occurs

- Possible reasons for the occurrence of congestion:
  - ① **Receiver capacity**
    - The receiver can not process the received data fast enough and therefore its receive buffer becomes full
    - Already solved by **flow control**
  - ② **Network capacity**
    - Congestion (overload) occurs when the utilization of a computer network exceeds its capacity  $\Rightarrow$  **congestion control**
    - Only useful reaction to congestion: **Reduce the data rate**
    - TCP tries to avoid congestion by changing the window size dynamically  $\Rightarrow$  **dynamic sliding window**
- *The one solution*, which solves both causes does not exist
  - Both causes are addressed separately

## Signs of congestion of the network

- Packet losses due to buffer overflows in Routers
- Long waiting times due to full queues in Routers
- Frequent retransmissions due to timeout or packet-/segment loss

# Approach to avoid Congestion

- The sender maintains 2 windows

## ① Advertised Receive Window

- Avoids congestion of the receiver
- Offered (*advertised*) by the receiver

## ② Congestion Window

- Avoids congestion of the network
- Determined by the sender

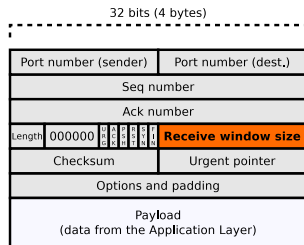
- The minimum of both windows is the maximum number of bytes, the sender can transmit

- Example:

- If the receive window of the receiver has a free storage capacity of 20 kB, but the sender recognizes that a network congestion occurs when more than 12 kB are sent, it transmits only 12 kB

- How does the sender know the capacitance of the network?

⇒ how does the sender determine the size of the congestion window?

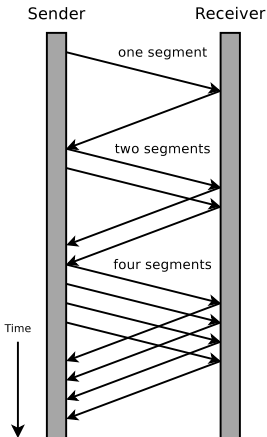


# Determine the Size of the Congestion Window

## You already know...

- The sender can precisely specify the size of the receive window
- Reason: The receiver informs the sender with every segment, about the free storage capacity of its receive window
- Challenge for the sender: **What is the size of the congestion window?**
  - The sender never knows the capacity of the network for sure
  - The capacity of computers networks is not static
    - It depends, among others, on the network utilization and the occurrence of network faults
- Solution: The sender must **incrementally** try to identify the network capacity

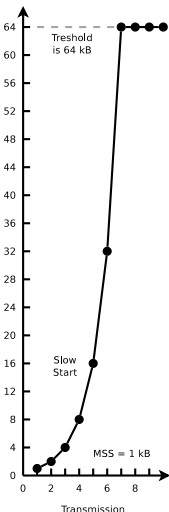
## Determine the Congestion Window Size – Connection Establishment



- During connection establishment, the sender initializes the congestion window to maximum segment size (MSS)
- Method:
  - 1 segment of size MSS is sent
    - If the segment is acknowledged before the timeout expires, the congestion window is doubled
  - 2 segments of size MSS are sent
    - If both segments are acknowledged before the timeout expires, the congestion window is doubled again
  - ...

## Determine the Congestion Window Size – Slow Start

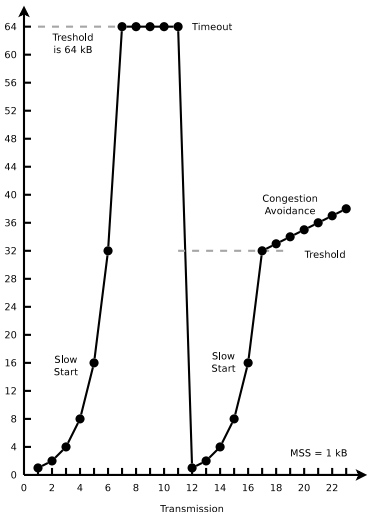
Congestion Window [kB]



- The congestion window grows exponentially until...
  - the size of the receive window is reached, which has been determined by the sender
  - or the **threshold** is reached
  - or a **timeout** expires
- The exponential growth phase is called **slow start**
  - Reason: The low transmission rate of the sender at the beginning
- If the congestion window reaches the size of the receive window, it stops growing
- At the beginning of the transmission, the threshold value is  $2^{16}$  bytes = 64 kB, so that it plays no role at the beginning
  - Maximum size of the receive window:  $2^{16} - 1$  bytes
    - This is determined by the size of the field **window size** in the TCP header

# Determine the Congestion Window Size – Congestion Avoidance

Congestion Window [kB]

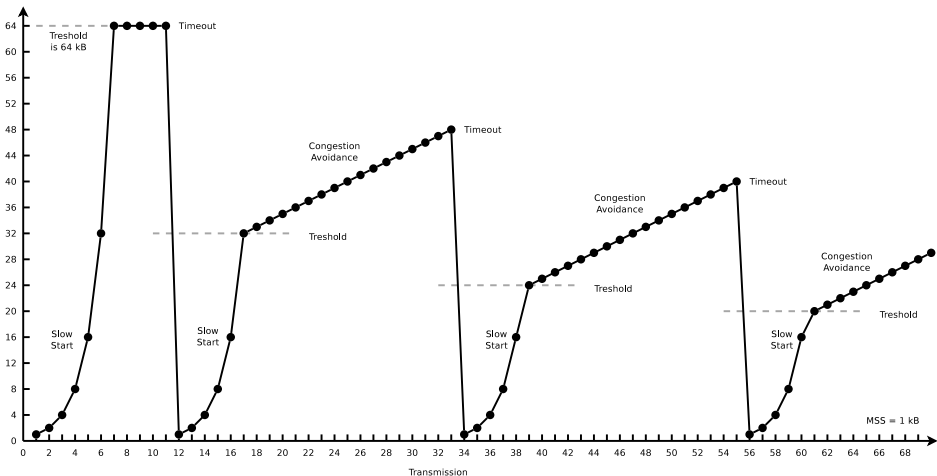


- If a timeout expires, . . .
  - the threshold value is set to half the congestion window
  - and the size of the congestion window is reduced to the size 1 MSS
- Then, once again the slow start phase follows
  - If the threshold value is reached, the congestion window grows linear, . . .
    - until the size of the receive window is reached, which is determined by the receiver
    - or until a timeout expires
- The linear growth phase is called **congestion avoidance**



# Possible Continuation of the Example

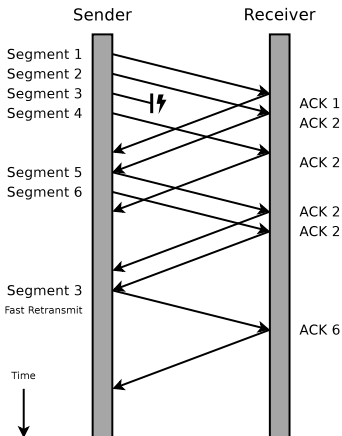
Congestion Window [kB]



# Reasons why a Timeout expires and reasonable Proceeding

- An expired **timeout** can have different reasons
  - Congestion ( $\implies$  delay)
  - Loss of a transmission
  - Loss of an acknowledgment (ACK)
- Not only delays due to congestion, but also each loss event reduces the congestion window to size 1 MSS
  - The obsolete TCP version *Tahoe* (1988) works this way
- Modern TCP versions differ between...
  - expired timeout caused by congestion of the network
  - and **multiple arrival of acknowledgments** (ACKs) caused by loss event

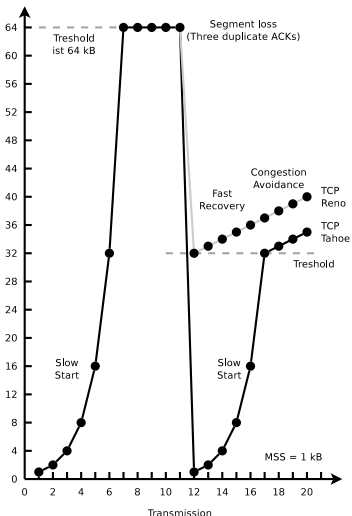
# Fast Retransmit



- A lost segment causes a *gap* in the data stream at receiver site
  - The receiver sends for every additional received segment an ACK for the segment before (the lost segment!)
- If a segment gets lost, a reduction of the congestion window to value 1 MSS is not necessary
  - Reason: A segment loss is not caused by congestion in any case
- If 3 duplicate ACKs arrive, TCP *Reno* (1990) sends the lost segment again  
⇒ **fast retransmit**

# Fast Recovery

Congestion Window [kB]



- TCP *Reno* also avoids the slow start phase if 3 duplicate ACKs arrive  
⇒ **fast recovery**
- If 3 duplicate ACKs arrive, the congestion window is set directly to the threshold value
  - The congestion window grows linear with every acknowledged transmission, . . .
    - until the size of the receive window is reached, which is specified by the receiver
    - or until a timeout expires

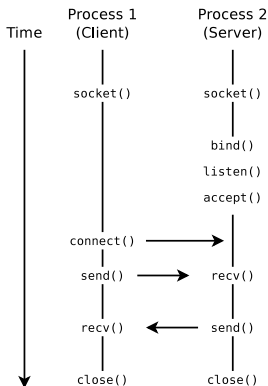
# Additive Increase / Multiplicative Decrease (AIMD)

- The concept of TCP congestion control is called AIMD
  - It stands for **rapid reduction** of the congestion window after a timeout expired or a loss event occurred and **slow (linear) increase** of the congestion window
- Reason for **aggressive reduction** and **conservative increase** of the congestion window:
  - The consequences of a congestion window which is too large in size are worse than for a window which is too small
  - If the window is too small in size, available bandwidth remains unused
  - If the window is too large in size, segments will get lost and must be transmitted again
    - This increases the congestion of the network even more!
- The state of congestion must be left as quick as possible
  - Therefore, the size of the congestion window is reduced significantly

# Summary of Flow Control and Congestion Control

- By using **flow control**, TCP tries to use the available bandwidth of a connectionless network ( $\implies$  IP) efficiently
  - Sliding windows at sender site (**transmit window**) and receiver site (**receive window**) are used as buffers for sending and receiving
  - The receiver controls the transmission behavior of the sender
- Reasons why congestion happens: **receiver capacity** and **network capacity**
  - The receive window avoids congestion of the receiver
  - The congestion window avoids congestion of the network
  - Actual available (used) window = minimum of both windows
- Attempt to maximize the network utilization and react rapidly to indications for congestion
  - Principle of **Additive Increase / Multiplicative Decrease** (AIMD)

# Connection-oriented Communication via Sockets – TCP



TCP sockets can also be implemented in a **blocking** (default) or **non-blocking** manner. This affects the behavior of the functions `connect`, `accept`, `send` and `recv`

## • Client

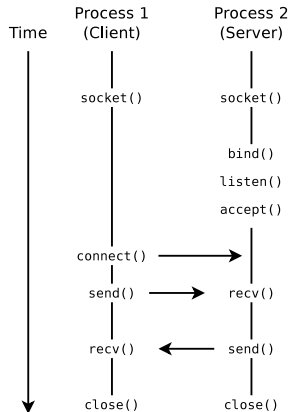
- Create socket (`socket`)
- Connect client with server socket (`connect`)
- Send (`send`) and receive data (`recv`)
- Close socket (`close`)

## • Server

- Create socket (`socket`)
- Bind socket to a port (`bind`)
- Make socket ready to receive (`listen`)
  - Set up a queue for connection requests. Specifies the number of connection requests, which can be stored in the queue
- Server accepts connections (`accept`)
  - Fetch the first connection request from the queue
- Send (`send`) and receive data (`recv`)
- Close socket (`close`)

# Sockets via TCP – Example (Server)

```
1 #!/usr/bin/env python
2 # Echo Server via TCP
3 import socket                # Import module socket
4
5 HOST = ''                    # '' = all interfaces
6 PORT = 50007                  # Port number of server
7
8 # Create socket and return socket descriptor
9 sd = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
10 # Bind socket to port
11 sd.bind((HOST, PORT))
12 # Make socket ready to receive
13 # Max. number of connections = 1
14 sd.listen(1)
15 # Socket accepts connections
16 conn, addr = sd.accept()
17
18 print 'Connected by', addr
19
20 while 1:                      # Infinite loop
21     data = conn.recv(1024)    # Receive data
22     if not data: break        # Break infinite loop
23     conn.send(data)           # Send back received data
24
25 sd.close()                    # Close socket
```



```
$ python tcp_server.py
```



## Sockets via TCP – Example (Client)

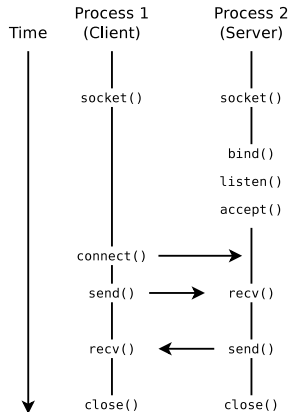
```

1 #!/usr/bin/env python
2 # Echo Client via TCP
3
4 import socket                # Import module socket
5
6 HOST = 'localhost'          # Hostname of Server
7 PORT = 50007                 # Port number of server
8
9 # Create socket and return socket descriptor
10 sd = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
11 # Connect with server socket
12 sd.connect((HOST, PORT))
13
14 sd.send('Hello, world')      # Send data
15 data = sd.recv(1024)         # Receive data
16 sd.close()                   # Close socket
17
18 # Print out received data
19 print 'Empfangen:', repr(data)

```

```
$ python tcp_client.py
Empfangen: 'Hello, world'
```

```
$ python tcp_server.py
Connected by ('127.0.0.1', 49898)
```



# Denial-of-Service Attacks via SYN Flood

- Target: Making services or servers inaccessible
- A client sends multiple connection requests (**SYN**), but does not respond to the acknowledgments (**SYN ACK**) of the server via **ACK**
- The server waits some time for the acknowledgment of the client
  - The confirmation delay could be caused by a network issue
  - During this period, the address of the client and the status of incomplete connection are stored in the memory of the network stack
- By flooding the server with connection requests, the table, which stores the TCP connections in the network stack is completely filled  
⇒ the server can no longer establish new connections
- The memory consumption at the server may become so large that the main memory gets completely filled and the server crashes
- Countermeasure: Real-time analysis of the network by intelligent firewalls