

## 5 Transport Layer



arrangement

### 5 Transport Layer

#### 5.1 Ports

#### 5.2 UDP – User Datagram Protocol

#### 5.3 TCP - Transmission Control Protocol

#### 5.4 Other Transport Layer Protocols

#### 5.5 Socket API

#### 5.6 Exercises - Transport Layer

#### 5.7 Summary - Transport Layer

The Network Layer enables end-to-end communication to take place between any two devices anywhere in the world. However, this communication is unreliable because data packets may be lost at the Network Layer. In order to compensate for this, the Transport Layer provides mechanisms to ensure the safe transmission of the data packets. This includes error control, flow control, and congestion control. In addition, communication is no longer between devices but between applications on the devices.

In the Internet, two protocols are used almost exclusively at the Transport Layer:

- User Datagram Protocol (UDP, RFC 768, August 1980) and
- Transmission Control Protocol (TCP, RFC 793, September 1981).

TCP implements all the required properties of the Transport Layer and is quite complex. UDP in contrast is a simple protocol and is as unreliable as IP. We will see that the choice of TCP or UDP as Transport Layer protocol depends on the requirements of an application.

The Transport Layer is only implemented on the end systems. Intermediate systems, in particular routers, do not have any knowledge about the Transport Layer.

### 5.1 Ports

The Transport Layer makes it possible to distinguish between applications on end systems, which requires an additional type of addressing. Consider the following situation: You use an end system to browse in the Internet. In the background you have also opened an e-mail program, which retrieves new e-mails from time to time. Some additional routine checks of programs might also be carried out to determine whether

updates are available. Let us now assume that an IP packet is sent from the Internet to your end system. Then it has to be decided whether this packet is part of an e-mail or whether it is part of a web page you are currently viewing. In practice, the so-called **port numbers** in the headers of TCP and UDP are used for the distinction.

The port numbers can be regarded as virtual interfaces on a computer. A distinction is made between a port number on the client side and a port number on the server side. The combination of a port number with the IP address of the device is called a **socket**.



In the online version an video is shown here.

Link to video : <http://www.youtube.com/embed/DuVZ5ki7QPs> 

#### Sockets and ports

In the protocol headers, 16-bit fields are contained for the port numbers. Among the port numbers there are differences between ranges.

**Well-known ports:** Port numbers 1 to 1023 are centrally assigned by IANA and can be used on most systems only by system processes or privileged users. These ports are used for the server side. The table shows selected examples of port numbers.

Application	Port	Explanation
FTP	20 / TCP	User data of the File Transfer Protocol (data transmission)
FTP	21 / TCP	Control data of the File Transfer Protocol (data transmission)
SSH	22 / TCP	Remote access with virtual terminal (Secure Shell)
Telnet	23 / TCP	Remote access with virtual terminal
SMTP	25 / TCP	Transmission of e-mails with the Simple Mail Transfer Protocol

DNS	53 / UDP+TCP	Resolution of domain names into IP addresses through the Domain Name System
DHCP	67 / UDP	Configuration of end systems via the Dynamic Host Configuration Protocol
DHCP	68 / UDP	Configuration of end systems via the Dynamic Host Configuration Protocol
HTTP	80 / TCP	Access to WWW pages with the Hypertext Transfer Protocol
POP	110 / TCP	Retrieval of e-mails with the Post Office Protocol
IMAP	143 / TCP	Retrieval of e-mails with the Internet Message Access Protocol

**Examples of well-known ports**

**Registered ports:** The port numbers 1024 to 49151 can be registered at IANA for applications. On most systems, they can be used by non-privileged applications. Use by clients is also possible.

**Example:**

SIP	5060 / UDP+TCP	Session Initiation Protocol (signaling for voice over IP)
-----	----------------	---

**Example: Registered ports**

**Dynamic and/or private ports:** No reservation for applications is provided for the port numbers 49152 to 65535. The ports are only used on the client side.


So while port numbers smaller than 1024 are typically used on the server side, port numbers larger than 1023 are dynamically selected on the client side. A UDP or TCP data stream can therefore be identified by five items:

- Client IP address
- Client port number
- Server IP address
- Server port number
- Transport Layer protocol used (TCP, UDP, ...)

If, for example, different clients retrieve web pages from a WWW server, only the client IP address and the client port number are different. It can even be the case that the distinction can be made only based on the client port number. This is actually often the case because web pages are often retrieved using simultaneously existing TCP connections. Within these TCP connections, the client IP address, server IP address and server port number are the same; only the client port number is different.



notice

A list of the most important ports is available on every computer (/etc/services under UNIX / LINUX or \windows\services under Windows). Since 2002, the port numbers are no longer published as RFCs, but can be found directly at [IANA](https://iana.org) .

## 5.2 UDP – User Datagram Protocol



arrangement

### 5.2 UDP – User Datagram Protocol

#### 5.2.1 UDP Header

#### 5.2.2 Application of UDP

UDP is a very simple protocol, which has only very few additional features in comparison to IP. It can be regarded as a kind of “IP on the Transport Layer.” Because UDP like IP is based on datagram transmission, UDP data units are called **datagrams**.

UDP does not try to improve the **missing reliability** of IP (see [Introduction to IPv4](#)). This means datagrams can be lost or be delivered to an application in the wrong order. The UDP header contains a checksum for bit error checking. If the checksum is not correct, the datagram is discarded. However, there is no retransmission.

UDP does **not** have **flow or congestion control**. This means that with UDP you can set a data rate and the UDP datagrams are then sent with this data rate. UDP does not notice what effect these datagrams have on other data streams in the network or whether the receiver might be overwhelmed by the amount of data delivered. The UDP data rate is limited only by how fast the application provides the data and the bit rate available at the attachment to the network. This has significant effects with regard to TCP data transmissions because TCP has a congestion control function and reduces the data rate if the network is overloaded. As a consequence, TCP data transfers can be completely suppressed by UDP. In this regard, you can call UDP **unfair** (see [Fairness](#)).

UDP also does not consider which MTU size limit applies at the Data Link Layer. This may require that UDP datagrams are fragmented at the IP layer. As already mentioned when discussing [fragmentation](#), this is unfavorable because fragmentation is inefficient.

UDP is **not connection-oriented**, so each UDP datagram is considered by itself. Because a single datagram is sent from one sender to one or more receivers, you can view the communication as simplex. Since multiple receivers are also possible, UDP supports multicast. Because UDP is not connection-oriented, there are no connection setup and decommissioning phases. Data can thus be sent immediately with UDP because a delay due to a connection setup phase does not occur.



In the online version an video is shown here.

Link to video : [http://www.youtube.com/embed/Ymn9dJu9\\_UU](http://www.youtube.com/embed/Ymn9dJu9_UU)

User Datagram Protocol



annotation

Since UDP is not connection-oriented, it is inappropriate to speak of a UDP connection if two end systems communicate with one another via UDP. It is better to call this a **UDP data stream**.

Since UDP has only a few features and no connection states must be maintained, a server can communicate with more clients simultaneously with UDP than with TCP.

### 5.2.1 UDP Header

The header is always 8 bytes long and is constructed as shown in the rollover element.



In the online version an rollover element is shown here.

#### UDP header

Begin printversion

0	8	16	24	31
Source port		Destination port		
Length		Checksum		

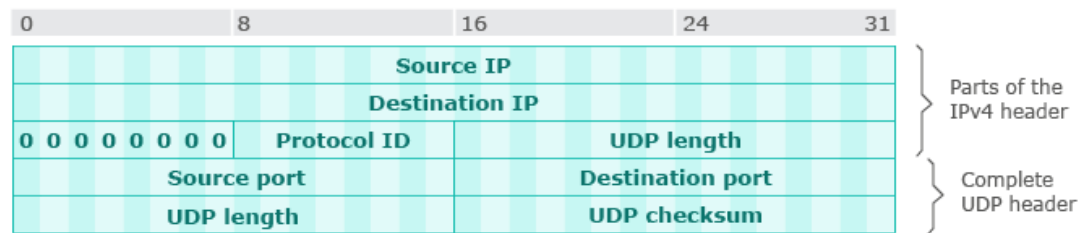
#### Destination port

Port numbers are used to identify application processes on the source and destination side.

field	explanation
Source port	Port numbers are used to identify application processes on the source and destination side.
Destination port	Port numbers are used to identify application processes on the source and destination side.
Length	The field indicates the length of the UDP header and the data (unit bytes).
Checksum	The checksum is optional and is calculated based on the Pseudo Header which comprises parts of the UDP and IP Header (reminder: the IP checksum is calculated based on the IP Header alone). The checksum should always be calculated, even though there are applications which do not calculate it due to performance reasons.

End printversion

The calculation of the checksum for the UDP header is not only based on the UDP header and the payload data. It also includes selected fields from the IP header. This area is called a **pseudo header**. In this case, the UDP identification, i.e. a 17 (binary coded), is in the protocol ID field.



Pseudo header for calculating the UDP checksum

## 5.2.2 Application of UDP

As indicated above, the lack of reliability is a critical issue for UDP use. Therefore, the question arises whether this protocol is useful for applications at all.

However, it is often used if communication with the counterpart can be reduced to a simple **request-response** mechanism. This avoids the overhead of TCP. A typical application of UDP is the Network Time Protocol, which queries the time of another computer. DNS (Domain Name System) is another protocol of this kind that requests the IP address from the DNS server for a given domain name.

Applications that transmit speech also use UDP. In this case, the language remains comprehensible as long as the percentage of lost datagrams does not become too high. The use of UDP is appropriate for **real-time requirements** of voice transmission.

## 5.3 TCP - Transmission Control Protocol



arrangement

### 5.3 TCP - Transmission Control Protocol

#### 5.3.1 TCP Header

#### 5.3.2 TCP Connections

#### 5.3.3 Error Control

#### 5.3.4 Flow Control

#### 5.3.5 Congestion Control


**TCP** is used for **reliable data transmission** between two end systems. Reliability means ensuring that no data is lost and all bit errors are corrected. In addition, data is delivered to the application in the correct order. Since the underlying IP does not have these properties, they must be implemented by TCP. The data units are referred to as **TCP segments**.

TCP implements a **flow control** so that the receiver does not get too much data that it can no longer accept. In addition, an algorithm for **congestion control** is used to protect devices on the network, such as routers and switches, from overloading. This algorithm automatically enables adaptation to the situation in the network. One consequence of this is that you do not have to set a bit rate for TCP. TCP continuously checks which data rate is currently available in the network and reduces or increases the data rate accordingly. TCP therefore behaves **fairly** against other data transmissions.

TCP is a **connection-oriented** protocol. This connection orientation, where the data units stand in relation to each other within a context, makes it possible to implement the features already mentioned. Accordingly, there is a connection **setup** phase (**three-way handshake**) at the beginning of a TCP connection and a decommissioning phase at the end of the connection (**four-way close**).



In the online version an video is shown here.

Link to video : <http://www.youtube.com/embed/AAvKXCwAgl0> 

**Transmission Control Protocol**

### 5.3.1 TCP Header

Before explaining the procedures implemented in TCP, the header is explained here.



In the online version an rollover element is shown here.

**TCP Header**

Begin printversion



0		8				16				24				31			
Source port								Destination port									
Sequence number																	
Acknowledgement number																	
Data offset		Re-served		N	C	E	U	A	P	R	S	F	Receive window				
				S	W	R	C	R	C	S	S	Y					
Checksum								Urgent pointer									
Options												Padding					
Data																	

<i>field</i>	<i>explanation</i>
<b>Source port</b>	Port numbers are used to identify application processes on the source and destination side.
<b>Destination port</b>	Port numbers are used to identify application processes on the source and destination side.
<b>Sequence number</b>	Indicates the number of first payload data byte. The only exception is inside the SYN segment during connection setup. The initial sequence number (ISN) is provided in this case. The first data byte's number is ISN + 1 later on.
<b>Acknowledgement number</b>	If the ACK bit is set, which is usually the case except of the SYN segment, then this field shows the sequence number of the next byte expected from the other side.
<b>Data offset</b>	Indicates where the payload data start. The value is indicated as the number of 32 bit words.
<b>Reserved</b>	Is not used. Has to be zero.
<b>Flags</b>	Nine single bits for connection control. The detailed meanings of the bits are:
<b>NS</b>	ECN-Nonce Concealment Protection (experimental, belongs to Explicit Congestion Notification mechanism)
<b>CWR</b>	Congestion Window Reduced (belongs to Explicit Congestion Notification mechanism)
<b>ECE</b>	Explicit Congestion Echo (belongs to Explicit Congestion Notification mechanism)
<b>URG</b>	Urgent Data Pointer is valid.
<b>ACK</b>	Acknowledgement Number is valid.
<b>PSH</b>	Data should be provided directly to the application, i.e. without buffering (Push).
<b>RST</b>	Connection reset
<b>SYN</b>	Connection setup request (by one party); the sequence number is initialized.
<b>FIN</b>	Connection decommissioning request (by one party)
<b>Receive window</b>	Number of free bytes in the receive buffer. The indicated value may have to be multiplied by a factor (window scaling). The window belongs to the flow control mechanism.
<b>Checksum</b>	To check for bit errors. It is calculated based on parts of the IP header (pseudo header), TCP header and data. If the checksum indicates bit errors on the receiver side, the segment is dropped.
<b>Urgent pointer</b>	Points to data with usual importance which are behind high priority data.
<b>Options</b>	There can be options at the end of the TCP header, in particular on connection setup. The length of the TCP header without options is 20 bytes.
<b>Padding</b>	If the length of the options cannot be divided by 32 bits, padding bits have to be added.
<b>Data</b>	Payload data from IP's point of view.

End printversion

An example of a TCP header is also provided here. The example values apply to the first segment during TCP connection setup.



example

### Trace of an TCP header

```

Transmission Control Protocol, Src Port: 3792 (3792), Dst Port: discard (9), Seq:
2134411119, Ack: 0, Len: 0
Source port: 3792 (3792)
Destination port: discard (9)
Sequence number: 2134411119
Header length: 28 bytes
Flags: 0x0002 (SYN)
0... ....
.0.. ....
..0. .... = Urgent: Not set
...0 .... = Acknowledgment: Not set
.... 0... = Push: Not set
.... .0.. = Reset: Not set
.... ..1. = Syn: Set
.... ...0 = Fin: Not set
Window size: 16384
Checksum: 0xb77e (correct)
Options: (8 bytes)
Maximum segment size: 1460 bytes
NOP
NOP
SACK permitted

```

## 5.3.2 TCP Connections



arrangement

5.3.2 [TCP Connections](#)

5.3.2.1 [Connection Setup](#)

5.3.2.2 [Connection Decommissioning](#)

5.3.2.3 [Connection State Diagram](#)

5.3.2.4 [Netstat](#)

As already mentioned, TCP is a connection-oriented protocol. This section therefore deals with connection setup and decommissioning as well as possible states of TCP connections at the client and the server. The netstat tool, which can be used to display the states of TCP connections, is presented at the end.



In the online version an video is shown here.

Link to video : <http://www.youtube.com/embed/g9kSbjBmQi8>

TCP Connection Setup and Close

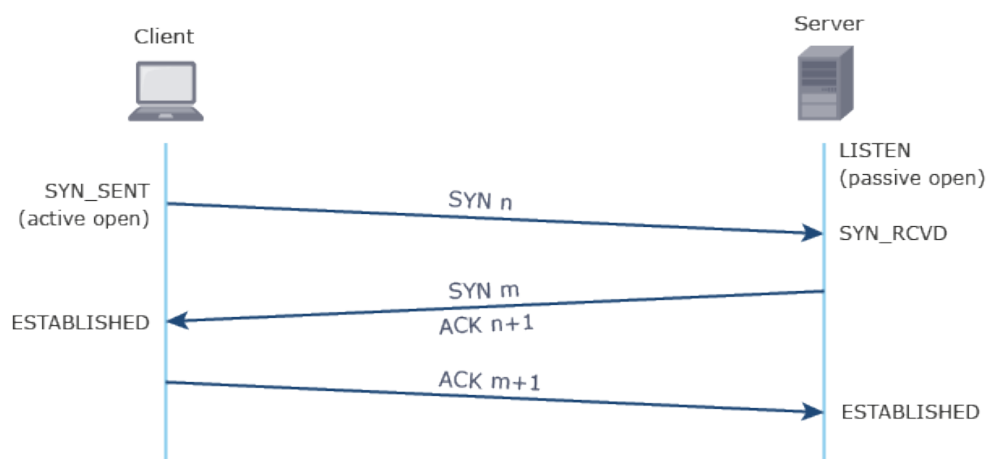
### 5.3.2.1 Connection Setup

The connection setup, referred to as the “**three-way handshake**”, happens in three steps. They are called according to the set flag bits.

1. **SYN**: The client sends a SYN segment (SYN bit is set in the header) with an initial sequence number (client ISN).
2. **SYN-ACK**: The server responds with a SYN segment containing a server ISN. The segment of the client is positively acknowledged so the ACK bit is also set. The number in the acknowledgment field is the client ISN + 1.
3. **ACK**: The client confirms the segment of the server, so the ACK bit is set. The acknowledgment number is server ISN + 1. In this segment the SYN bit is not set.

This establishes the connection. Data can then be transmitted simultaneously in both directions (i.e. duplex).

The ISN is taken from a time generator, which is incremented approximately every 4 ms. Therefore an overflow happens approximately every 4 hours 55 minutes. Since the maximum segment lifetime (MSL) is significantly lower, we can safely assume that the ISN is unique.





### Connection setup, 'three-way handshake'

During the connection setup, the **maximum segment size** can be transmitted as a TCP option in the SYN segment. This specifies the maximum size of a data block that can be sent. TCP takes the MTU of the physical interface as the basis to derive the MSS. Both sides can specify different values for the MSS. If the MSS is not specified, it is set to 536.

The following animation shows a connection setup between a client and a server for illustration purposes.

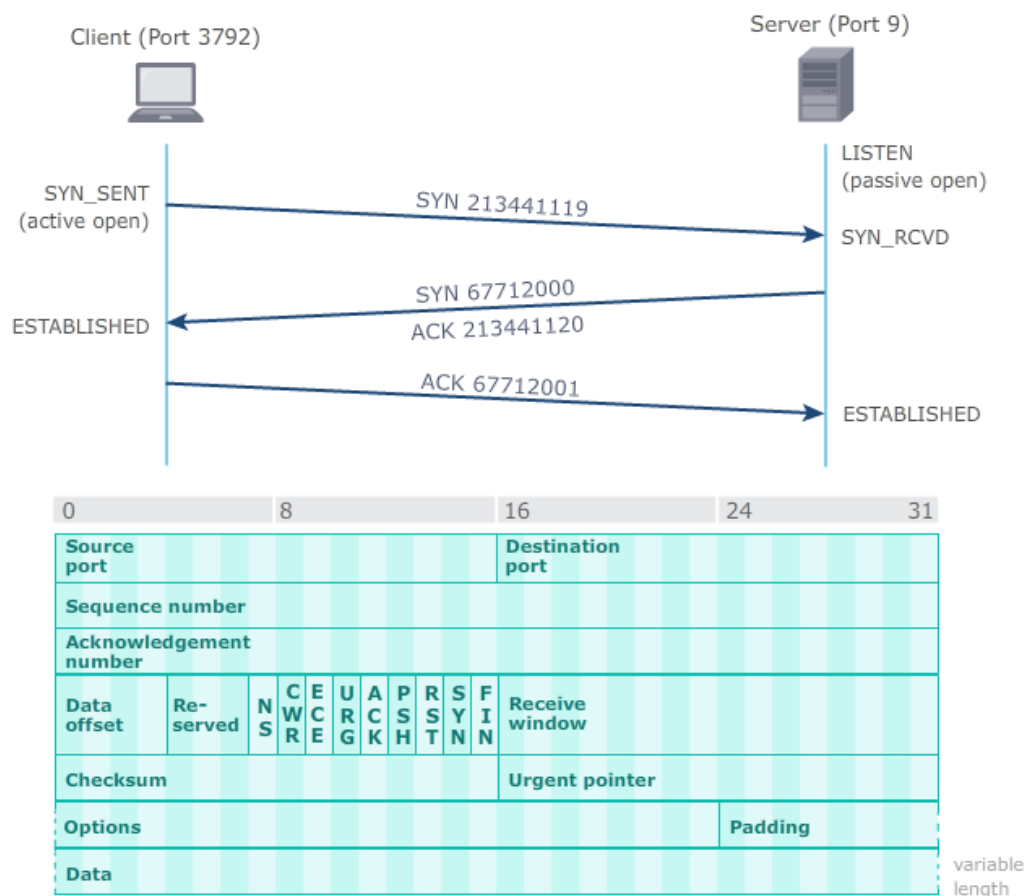


In the online version an animation is shown here.

```
<loop_video source="tcp_verbindungsaufbau_pro_animated.mp4" width="698"
height="570"> </loop_animation>
```

### TCP connection setup

Begin printversion



To set up a connection to the server, the client sends a SYN segment with an initial sequence number. Its internal state is then the active open state. The TCP Header contains information about source port, destination port, initial sequence number, window size and maximum segment size. The SYN flag is set. Also additional fields such as checksum have contents which are not shown to avoid confusion.

The server's state is LISTEN which is also called passive open. It receives the SYN segment and changes its state to SYN\_RCVD. The server replies and indicates source and destination port in a SYN segment which also contains its initial sequence number. In the same segment it confirms the SYN segment from the client with an acknowledgement number. The acknowledgement number is the sequence number from the received SYN segment increased by one. The flags SYN and ACK are set. In addition, the window size and the maximum segment size are given.

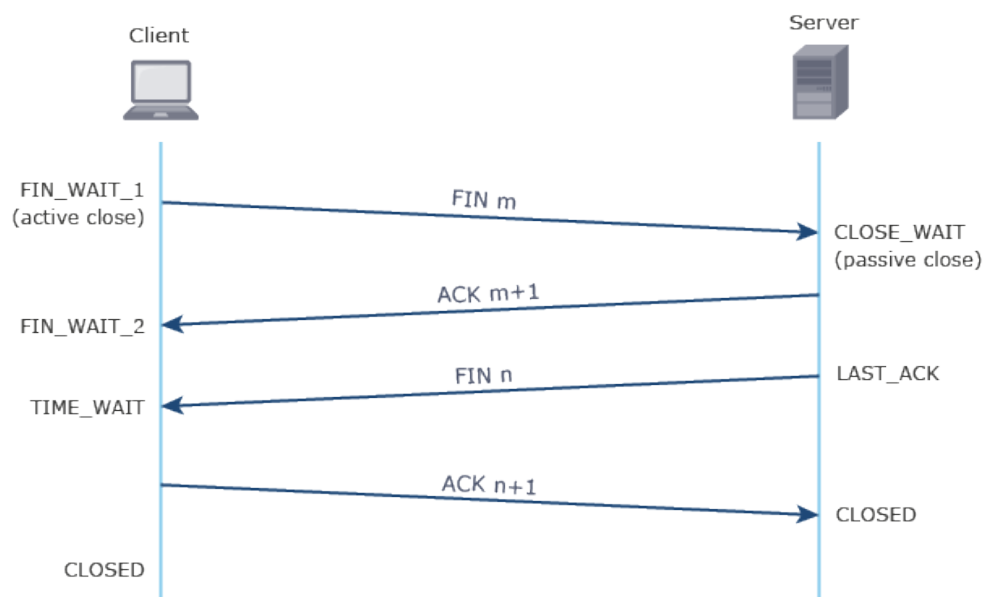
The client gets the ACK from the server and changes its state to the ESTABLISHED state. The client confirms the SYN from the server with an ACK. Its number is increased by one in comparison to the sequence number from the received SYN-ACK segment. Its own sequence number is the initial sequence number increased by one. The ACK flag is set.

The server receives the ACK and changes its state to ESTABLISHED. The connection can now to be used for data exchange in both directions.

End printversion

### 5.3.2.2 Connection Decommissioning

Since TCP operates in full duplex mode, both transmission channels can be closed independently from each other. If one side sends a FIN segment, it indicates that it does no longer want to send data. The other side confirms the reception of the FIN segment with an ACK segment and can afterwards continue to send data. Once all data has been transmitted, this side also sends a FIN segment, which in turn is acknowledged with a final ACK by the first side. The TCP connection is completely closed then.



#### Usual TCP connection decommissioning

The following animation provides an illustration of a TCP connection commissioning.

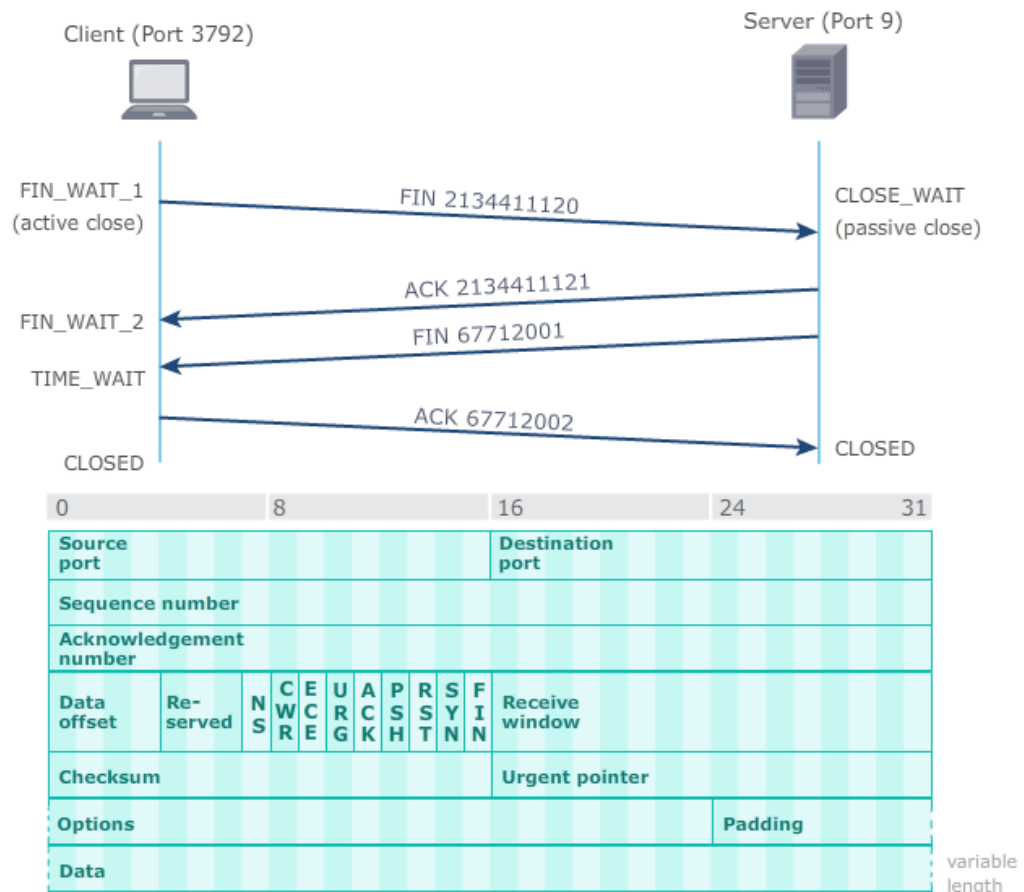


In the online version an animation is shown here.

```
<loop_video source="tcp_verbindungsabbau_pro_ani_en.mp4" width="698"
height="570"> </loop_animation>
```

**TCP connection decommissioning**

Begin printversion



In the following example the client wants to terminate a TCP connection to the server because it does not want to send further data to it. For doing so, the client sends a FIN segment with the current sequence number and changes its state towards the FIN\_WAIT\_1 state (active close). In addition to the sequence number, the source port and destination port are provided in the TCP header. The FIN flag is set. Also the other header fields such as the window size are filled with appropriate values which is not shown to make it easier to understand. The server receives the FIN segment and changes its state to CLOSE\_WAIT (passive close).

The server confirms the FIN segment by an ACK segment. Its acknowledgement number is the sequence number of the received FIN segment increased by one. The acknowledgement number, source port, destination port and additional values are given in the TCP header. The ACK flag is set. The client receives the ACK segment and changes its state to the FIN\_WAIT\_2 state.

If the server has sent all remaining data, it also sends a FIN segment with an appropriate sequence number which is contained in the TCP header. The FIN flag is set. Once the client has received this FIN segment, it changes its state to TIME\_WAIT.



At the end, the client confirms the server's FIN segment with an ACK segment. The acknowledgement number in the segment is the sequence number of the received FIN segment increased by one. The acknowledgement number, source port and destination port are contained in the TCP header. The ACK flag is set. The server gets the ACK segment and changes its state to the CLOSED state. The TCP connection is terminated. The server can release its socket now.

After the elapse of the TIME\_WAIT state the socket can also be released on the client side. The client changes its state to CLOSED.

End printversion

During the connection decommissioning, the side that sent the last ACK goes into the **TIME\_WAIT state** before the connection is completely terminated.



annotation

Why is this waiting time required?

It must be taken into account here that so far an error-free sequence of segments has been shown. However, the procedure must be able to cope with segment losses at any point.

If the last ACK is lost, the last FIN segment is repeated. If the side that sent the last ACK has already been completely terminated and all resources have been released, it is unclear where this FIN segment is to be delivered. It would be even more serious if in the interim period - before the repeated FIN segment was sent - a new connection with the same port number was set up. This new connection would then incorrectly receive the repeated FIN segment, which cannot be handled by it.

Therefore, the side that sent the last ACK waits as long as needed so that a repeated FIN segment can arrive. It must therefore wait as long as a segment may still be transmitted in the network. This time is called the maximum segment lifetime (MSL). Since both the last ACK segment and a repeated FIN segment have a maximum lifetime MSL, the waiting time is set to  $2 \times \text{MSL}$ . RFC 793 specifies MSL as 2 minutes, so  **$2 \times \text{MSL}$**  means a wait time of 4 minutes. From today's point of view, this waiting time is quite long and can, for example, be reduced to 30 or 60 seconds.



annotation

This waiting time also means that the port number cannot be used again during this period. Since clients, that use their ports (port numbers > 1023) dynamically, in most cases send the last ACK, this does not pose a problem for the application because the next unused port is simply taken. However, if a server sends the last ACK, it must take special precautions; otherwise, the service would not be available for up to 4 minutes (in this case, there is the `SO_REUSEADDR` option in the socket library; see [Socket API Overview](#)).

### 5.3.2.3 Connection State Diagram

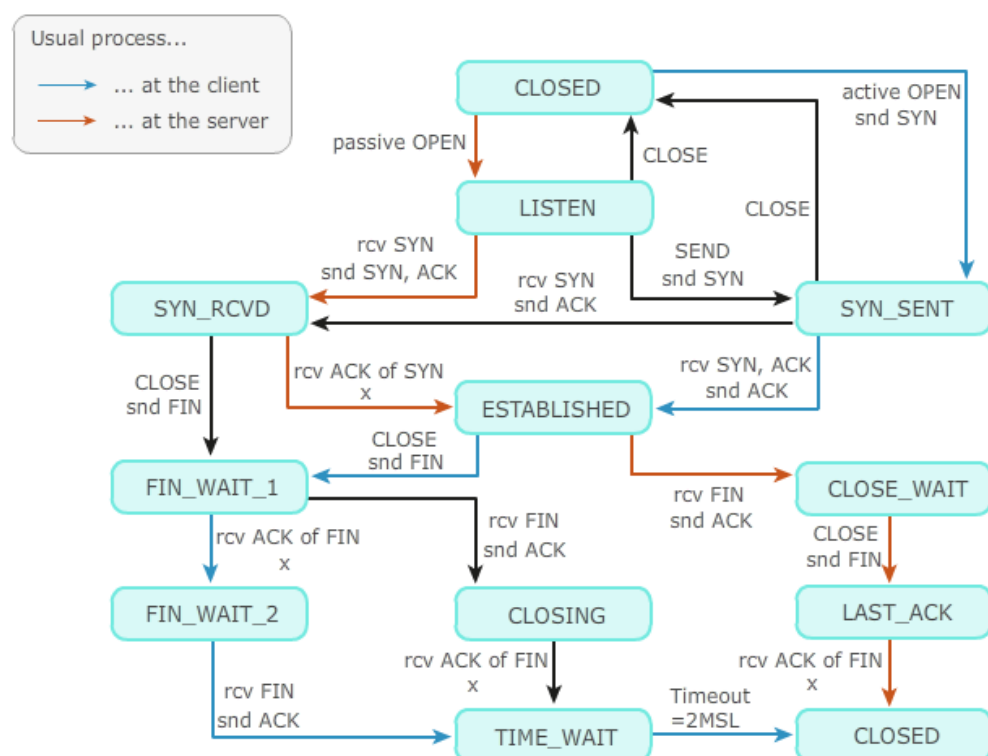
The various states that TCP can have during a connection are shown in the **connection state diagram**. Some of the states were already shown as part of the sections on connection setup and decommissioning. The CLOSED state is fictive. In this state, there is no connection and no resources are reserved.



In the online version an animation is shown here.

Connection state diagram

Begin printversion



Here we see the TCP state transition diagram as defined in RFC 793. The boxes represent the states which TCP can have during the connection setup and decommissioning. The blue arrows indicate the usual sequence of states on the client side. The red arrows show the usual state sequence on the server side. In addition to the state transitions which are explained in two additional videos about the connection setup and decommissioning, additional state transitions are possible. They are not explained further, but are displayed in the diagram with black arrows.

The 11 states are shown by the command line tool “netstat”. The state CLOSED is, however, not visible because “netstat” only considers existing connections. The states LISTEN, ESTABLISHED, and TIME\_WAIT are often visible in “netstat” because these states can last for a long time.

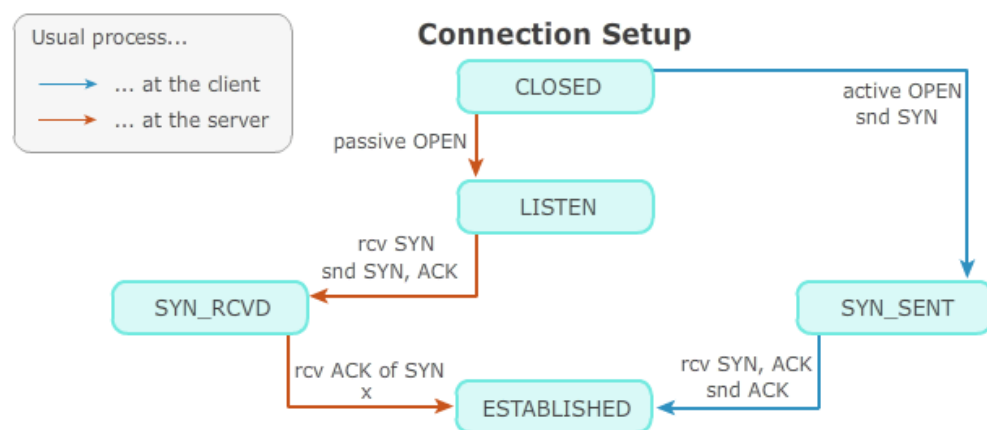
End printversion



In the online version an animation is shown here.

#### Connection setup

Begin printversion



Prior to opening a TCP connection the passive server side has to be activated. The server does a transition from the CLOSED into the LISTEN state without sending or receiving data. This state transition is called passive OPEN. The server waits in the LISTEN state that the client wants to set up a connection. The client starts a connection to the server by sending a SYN segment and changes its state from the CLOSED to the SYN\_SENT state. The server receives the SYN segment. It confirms it with an ACK segment which is combined with an own SYN segment and then changes its state from LISTEN to

SYN\_RCVD. The client receives the SYN-ACK segment and confirms the received SYN with an own ACK and changes its state from SYN\_SENT towards ESTABLISHED. The server gets the ACK and changes its state from SYN\_RCVD to ESTABLISHED. Now the connection is established and data can be exchanged.

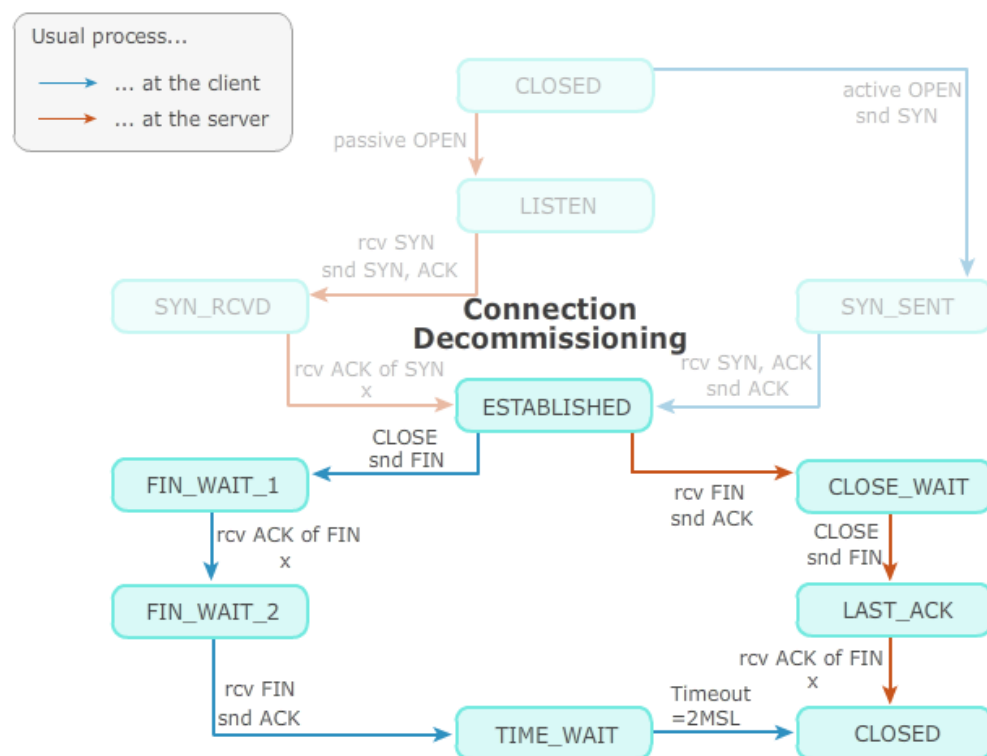
End printversion



In the online version an animation is shown here.

#### Connection decommissioning

Begin printversion



The connection decommissioning is done in two steps. At first, one party tells the other that the connection should be closed. Then the second party also announces that the connection should be terminated. In principle both steps are independent, but usually they directly follow each other.

*1st step:* If one party does not want to send more data, e.g. the client after completely receiving a requested file, it sends a FIN segment and changes its state from ESTABLISHED to FIN\_WAIT\_1. The other party confirms the received FIN with an ACK and changes its state from ESTABLISHED to CLOSE\_WAIT. The first party receives the

ACK and changes its state from `FIN_WAIT_1` to `FIN_WAIT_2`. Now the second party is still allowed to send data. Usually the second step to close the connection is done directly afterwards.

*2nd step:* The other party sends a FIN segment and changes its state from `CLOSE_WAIT` to `LAST_ACK`. The first party confirms the received FIN with an ACK and changes its state from `FIN_WAIT_2` to `TIME_WAIT`. It is changed to `CLOSED` after a 2 MSL timeout. In this transition all resources are released. The other side gets the ACK and changes its state from `LAST_ACK` to `CLOSED`. This transition also implies a release of all resources.

End printversion

With the netstat tool, which is described in the next section, the states of TCP connections of your own computer can be displayed.



task

### Task: TCP states I

Can the `CLOSED` state be displayed with netstat?

#### Solution

No, it is fictive and does not use any resources.

### Task: TCP states II

Why are the `LISTEN` and `ESTABLISHED` states frequently visible? Why is the `TIME_WAIT` state shown often?

#### Solution

`LISTEN`: Passive server sockets are in the `LISTEN` state and always visible (24/7 operation).

`ESTABLISHED`: This state is visible as long as a connection is active.

`TIME_WAIT`: One side of the connection remains available 1-4 minutes after the connection has been terminated.

### 5.3.2.4 Netstat

Netstat (network statistics) displays various information about network communication. Netstat does not send and receive packets. Nevertheless, the status of the opened sockets is displayed, i.e., states like LISTEN, ESTABLISHED, etc. for TCP connections. With netstat, you can therefore quickly get an overview of the connections that a host has to the Internet.

Netstat is available on all UNIX and Windows computers and can be accessed under Windows in the DOS window.

You can also use a freeware version of netstat with a graphical user interface, which can be found under the name [TCPView](#)  at Microsoft.

In addition to displaying the TCP connection states, netstat also has other functions.

- With “netstat -r” you can see the local routing table.
- With “netstat -s” you can look at statistics for communication via different protocols. It provides you with information, for example, about how often packets are discarded.



practice

With Linux you can find also information about current TCP connections with the command “ss -i”. It displays information about the congestion control algorithm, the RTT, the MSS and window sizes (congestion control, flow control).

### 5.3.3 Error Control



arrangement

#### 5.3.3 Error Control

##### 5.3.3.1 Determination of Retransmission Timeouts

##### 5.3.3.2 Acknowledgment Mechanisms

##### 5.3.3.3 Fast Retransmit

##### 5.3.3.4 Selective Acknowledgments

The error control mechanisms of TCP are there to ensure that **all data** is received correctly. To do this, the loss of segments and the occurrence of faulty segments have to be recognized. The segments must then be retransmitted. To identify whether segments

have been received correctly, TCP uses positive acknowledgments. If something is received incorrectly, which is determined by checksum comparison, no feedback is sent.

In addition, the **correct sequence** of data must be maintained when delivering data to the application. The sequence has to be restored by the protocol especially when segments have been lost and retransmitted.

Overall, you should note that many of the mechanisms that are used for TCP error control are already known to you from the Data Link Layer (see Error Detection and Error Correction). The important difference, however, is that TCP has an end-to-end error control. This takes place only in the original source and final destination and not on intermediate systems. Error control on the Data Link Layer, on the other hand, takes place on a link-by-link basis, e.g. between two routers.

### 5.3.3.1 Determination of Retransmission Timeouts

When a TCP entity sends a segment, the receiver must acknowledge its reception. If the acknowledgment does not occur, TCP transmits the segment again. One question that arises is how long TCP should wait for an acknowledgment. This waiting period is called **retransmission timeout (RTO)**.

The length of the timeout can obviously not be defined in a static manner, but must be adapted to the network path: For paths with a short runtime, the acknowledgments arrive more quickly, so the selected timeout can be shorter; the timeout will be longer for paths with a longer runtime. The timeout should also be adapted to the variability of the acknowledgments: If the acknowledgments happen to be faster at times and slower at other times, the timeout should increase, so as not to create too many superfluous repeats.



In the online version an video is shown here.

Link to video : <http://www.youtube.com/embed/-ynMxWtIFek>

**TCP Timeouts**

Over the years, various algorithms to determine the retransmission timeout have been tested. The current state is described in RFC 6298 and uses the following parameters and formulas:



## Parameters for determining the retransmission timeout (RFC 6298)

Begin printversion

Algorithm	Explanation	Remark
RTT	Round Trip Time	is measured between the transmission of a segment and the reception of the acknowledgment segment for it
SRTT	Smoothed Round Trip Time	smoothed estimated RTT
RTTVAR	Round Trip Time Variation	indication to which extent the RTT values are fluctuating
G	Granularity	granularity of used clock which is relatively rough (between 100 and 500 ms)
RTO	Retransmission Timeout	if no acknowledgment is received within this waiting time, a retransmission is performed

Parameters for determining the retransmission timeout (RFC 6298 [↗](#))

End printversion

The initialization is performed as follows.

$$\text{SRTT} \leftarrow \text{RTT}$$

$$\text{RTTVAR} \leftarrow \text{RTT}/2$$

$$\text{RTO} \leftarrow \text{SRTT} + \max(G, 4 \cdot \text{RTTVAR})$$

In the subsequent RTT measurements, the following applies:

$$(1) \text{RTTVAR} \leftarrow (1 - \beta) \cdot \text{RTTVAR} + \beta \cdot |\text{SRTT} - \text{RTT}|$$

$$(2) \text{SRTT} \leftarrow (1 - \alpha) \cdot \text{SRTT} + \alpha \cdot \text{RTT}$$

where  $\alpha = 1/8$  und  $\beta = 1/4$

$$(3) \text{RTO} \leftarrow \text{SRTT} + \max(G, 4 \cdot \text{RTTVAR})$$



annotation

What do these formulas mean?

First, an estimate for the next RTT value is needed (abbreviated as SRTT, see formula 2). This value is determined as a moving average. 7/8 of it comes from the previous estimate and the current value only contributes with a factor of 1/8. The estimated value is therefore relatively insensitive to random fluctuations of the RTT. With an estimate




for RTT, however, you do not have the RTO value because you do not want to make a retransmission in the case of small delays compared to the estimation.

You could always add a predefined safeguard time, but the implementation is more sophisticated. The safeguard time is selected according to how strongly the measured values fluctuate. For this reason, the variance of the measured values is estimated by the parameter RTTVAR (see formula 1) with a further moving average. From this value, the safeguard time, which is relevant when the RTO is set in formula 3, is obtained. The parameter G when setting the RTO is only for the unlikely event that RTTVAR should be 0.

According to the RFC, RTO values less than one second should be rounded up to one second.



practice


In practice, the time of one second, which is called the minimum in RFC, is relatively long. In Linux, significantly lower values are common and adjustable (see [answer in a Linux forum](#) .

If a segment is retransmitted, the RTO must then be doubled. On further retransmission, this continues until the maximum of 60 s is reached. How often segments are retransmitted before the connection is disconnected depends on the implementation.

It is important not to use acknowledgments of retransmitted segments to calculate the RTO, since it cannot be decided which retransmitted segment the acknowledgment refers to and therefore the calculation would be wrong. This is known as **Karn's algorithm**.



practice


The practical implementation of the RTO calculation within Linux is presented in a [blog entry](#) .

### 5.3.3.2 Acknowledgment Mechanisms

When a new TCP segment is received by the destination side, the question arises as to how the recipient should acknowledge it. In this case, it has to be distinguished between different situations, which differ according to which TCP segments have previously been received.

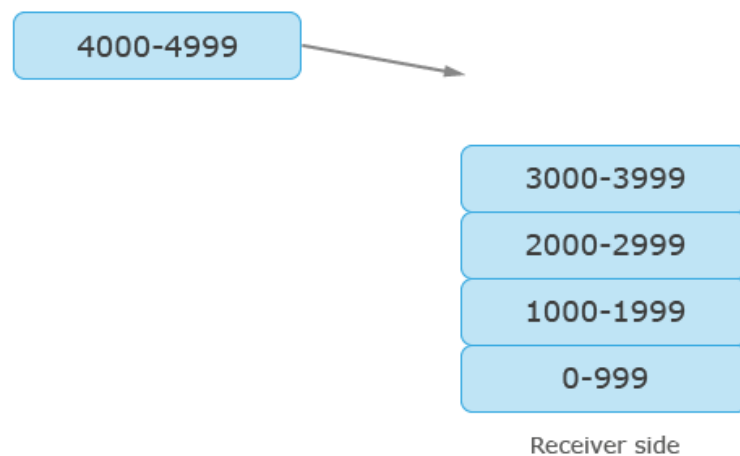


In the online version an video is shown here.

Link to video : <http://www.youtube.com/embed/Zbkx651w21E> 

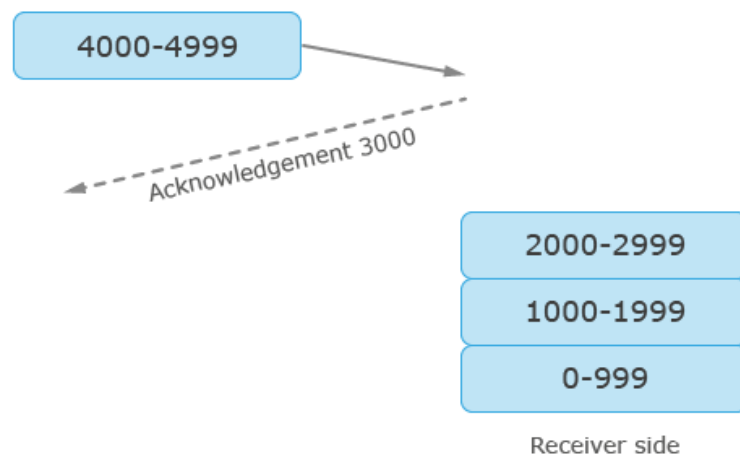
#### Acknowledgment mechanisms

In the first situation, the currently received segment fits to the expectations of the receiver side in terms of the sequence number. For example, the acknowledgment number in the previous TCP segment from the receiver side may have been 4000, so it would now fit if the bytes 4000 to 4999 were received in a segment (see figure below). You could assume that the receipt is acknowledged immediately. However, an optimization has been introduced at this point so that the receiver still waits a little bit to see whether further segments are arriving (**delayed acknowledgment**) shortly afterwards. Then it could send only one segment for acknowledgment, which means it would have to transfer less data. Such a type of summarizing acknowledgment is called **cumulative acknowledgment**. However, the receiver side will also send an acknowledgment after a short time if no further segments arrive.



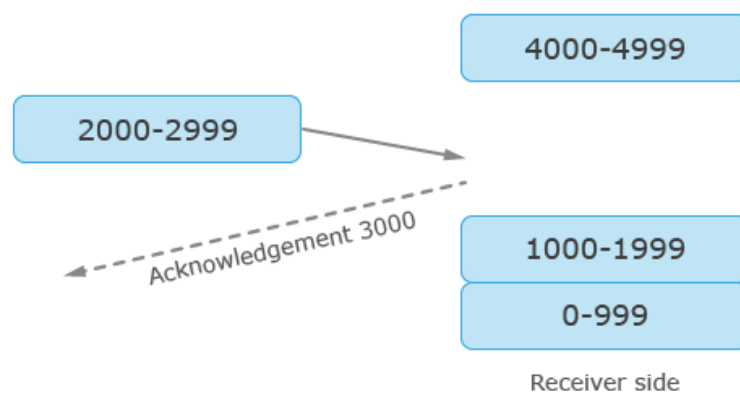
#### Segment with the expected sequence number arrives

In the second situation, a segment arrives whose sequence number does not match to the expected value; it is too high. In this case, the previous acknowledgment is repeated immediately (in the example in the figure below, 3000 is entered in the acknowledgment number field). This is called a **duplicate acknowledgment**. This acknowledgment refers to the point up to which all segments had arrived correctly. The fact that this acknowledgment is sent immediately has to do with the Fast Retransmit mechanism, which is explained in the following section.



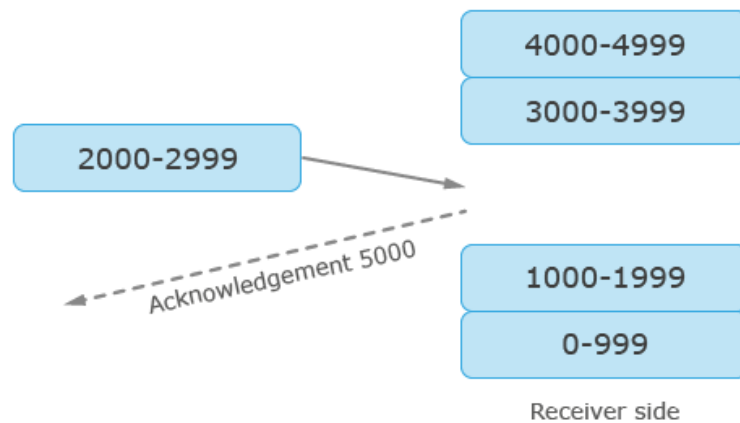
#### Segment with a sequence number that is too high arrives

In the third case, a segment whose sequence number fits to the expectation is received, but prior to this segments already have been received out of order. Despite of the reception of this segment, a gap remains among the segments that have already been received. In this case, an acknowledgment is sent immediately. This is intended to prevent a possible retransmission of the segment, since it arrived late relative to the segments that had already been received out of order.



#### Segment with expected sequence number arrives; out-of-order segments had already arrived prior to this; gap continues to exist

The fourth case is similar to the third case, except that the currently received segment closes a gap. A cumulative acknowledgment is then sent immediately. Its acknowledgment number then includes previously received segments.

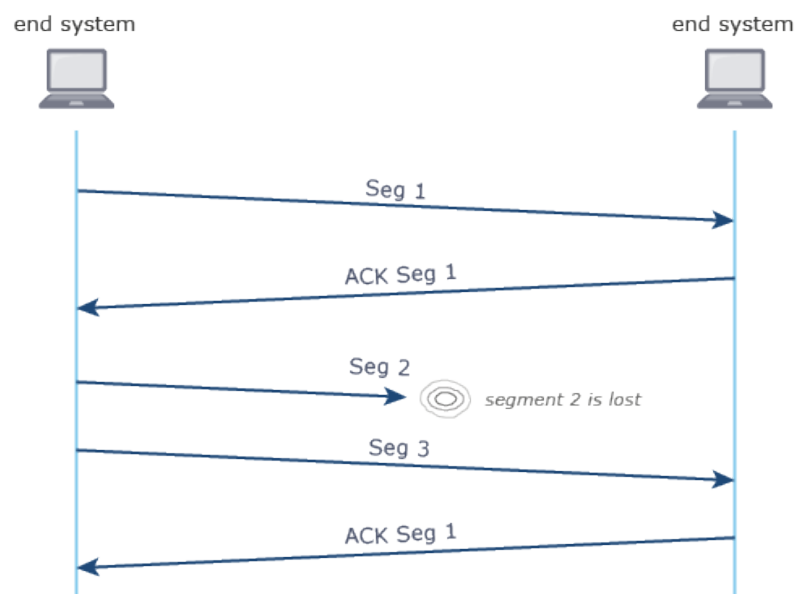


Segment with expected sequence number arrives; out-of-order segments had already arrived prior to this; gap is now closed

If segments with a too low sequence number arrive (subsequently), they are simply discarded.

### 5.3.3.3 Fast Retransmit

If a segment is lost but subsequent segments successfully arrive at the receiver, this results in **duplicate acknowledgments** (see figure below). The acknowledgment refers several times to a sequence number that is expected next.



Duplicate ACKs

At this point there is an improvement option called fast retransmit (RFC 5681). If 3 duplicate ACKs (i.e. 4 identical ACKs) are received consecutively, the loss of the segment that contained the next expected bytes is assumed. The lost segment is then sent again without waiting for the retransmission timeout, which is between 1 s and 60 s according to RFC 6298. This algorithm allows errors in the transmission to be compensated for more quickly.

### 5.3.3.4 Selective Acknowledgments

The difference between the methods go-back-N and selective retransmission has already been discussed in the general discussion of error-protection mechanisms in the Data Link Layer (see the [Credit Method and Sliding Window Techniques](#)).

TCP basically uses the Go-back-N method, but there are some possibilities for modification. A frequently activated option is **selective acknowledgments** (RFC 2018). In this case, the sequence numbers of data received out of order are transmitted to the sender in the TCP options field (usually three connected ranges that have already been received can be given here). The sender can then send just the missing data and does not send any superfluous data that the receiver has already received before.

## 5.3.4 Flow Control



arrangement

### 5.3.4 Flow Control

#### 5.3.4.1 [Sliding Window Use in TCP Flow Control](#)

#### 5.3.4.2 [Zero Window Probe](#)

#### 5.3.4.3 [Nagle Algorithm](#)

TCP flow control is used to prevent that a receiver is flooded with too much data. It may be that a receiver cannot process data internally as quickly as they are delivered via the network. If this situation exists for a short period of time, it can be compensated for by a receive buffer. However, if it persists for a long time, then data would be transmitted unnecessarily over the network since they could not be accepted by the receiver and would be lost there. Therefore, a feedback mechanism is implemented with flow control, which allows the receiver to continuously inform the sender how much data it is allowed to send.



In the online version an video is shown here.

Link to video : <http://www.youtube.com/embed/SNuU1HBLICI>

**Flow control**

### 5.3.4.1 Sliding Window Use in TCP Flow Control

In order to achieve a high data rate, the sliding window technique is used for TCP (see Credit Method and Sliding Window Techniques). With this technique, typically some TCP segments are sent following each other, which are then confirmed in the best case only with a single acknowledgment. The question in this context is how many segments may be sent one after the other.

In addition to Congestion Control, flow control also leads to a limitation of the data rate on the sender side. The aim of flow control is to ensure that the receiver is not overloaded. The receiver has a buffer, in which it stores data received via the network, which are fetched from an application. This buffer is called the **receive buffer**. The buffer may be filled to some extent with data while the rest of it is empty. This free space is called the **receive window**. The idea is that with a TCP header field of the same name the other end always indicates how much free space is still available. The sender may not transmit more data than the free space that is available.



**Receive buffer and receive window**

The field in the TCP header has 16 bits so the maximum value is 65535. However, at the beginning of a TCP connection, you can negotiate a parameter called **window scaling** so that the value represents a multiple of one byte. For example, a multiplication by 256 can be negotiated. This is often used at high bit rates when large buffers are used (see Congestion Control for Large High Bit Rate Networks).

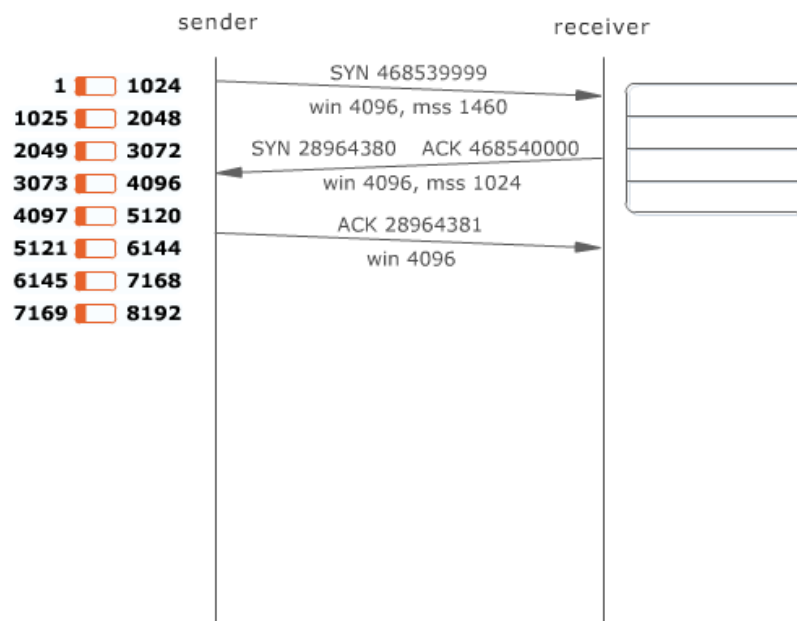
In the extreme case, it is possible to send so much data that the buffer is completely full (**receive window = 0**) because the receiver no longer can keep up with processing the data. Then the sender has no choice but to wait for the receiver to have free space in the buffer after the transmitted data has been processed. The processing of the data may, for example, mean to store the received data on the hard disk.



In the online version an animation is shown here.

#### Sliding window mechanism

Begin printversion



In this example the sender wants to send a data packet with the size 8192 bytes to the receiver. For doing so, it sends a SYN segment first to set up the connection. It indicates its window size, i.e. the size of its receive buffer, in the TCP Header. In this case the size is 4096 bytes. In addition, it provides his maximum segment size (MSS) as part of the TCP header. It is 1460 bytes in the current example.

The receiver confirms the SYN with an ACK, sends a SYN on its own and provides its window size, i.e. the size of the receive buffer as 4096 bytes and the maximum segment size as 1024 bytes.

---

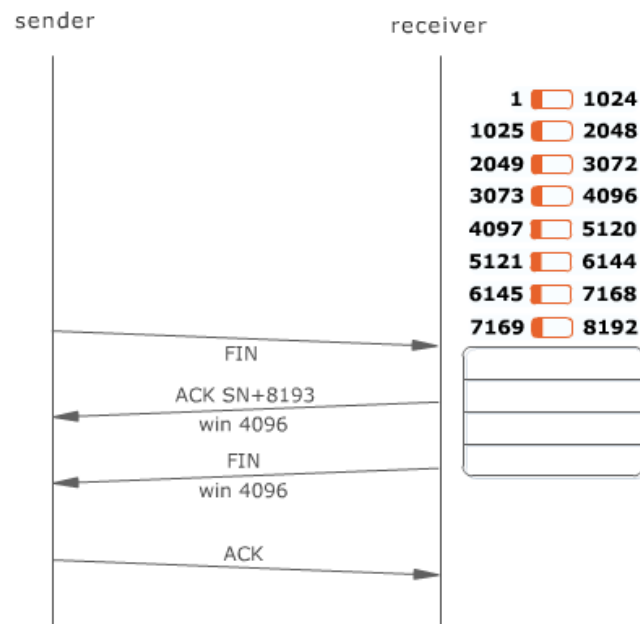
Let us assume for simplicity that the sender always sends segments with 1024 bytes; it is then going to send eight segments. It confirms the SYN segment of the receiver using an ACK. The connection is established. Due to the window size indication as 4096 bytes and the maximum segment size 1024 bytes the sender can send four segments with 1024 bytes each. Then it has to wait for a confirmation. The receive buffer is now completely full, i.e. the window is closed: all data have not been retrieved from the receive buffer yet. Therefore, the window size is set to zero, i.e.  $\text{win} = 0$ . The sender is not allowed to send any segments right now. The receiver confirms the received segments by sending an ACK with an acknowledgement number which is increased by 4097 bytes. In this way the receiver indicates that it has received 4096 bytes and now expects the byte 4097.

Once the receiver has processed the received segments, it opens his window again and shifts it. In this example, it has already processed three segments and sends an ACK to the sender. It confirms again the already received segments and indicates that there are 3072 bytes of free space in the buffer, i.e.  $\text{win} = 3072$ .

The sender sends the three next segments. The data have not been read from the receive buffer. Therefore, the window size is zero.

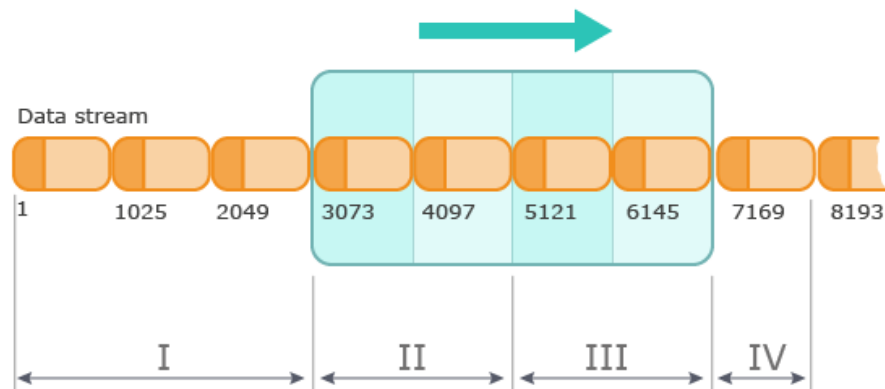
The receiver confirms them with an ACK. After processing two segments, the receiver opens its window again, sends again an ACK to the sender and provides its window size as 2048 bytes. The sender sends the last segment. The FIN flag is set in the TCP header. In this way the sender indicates that it wants to terminate the connection once the receiver has received and confirmed the last segment. The receiver confirms the last segment with an ACK. Afterwards, it sends a FIN segment as well. The window size is set accordingly. The last FIN segment is confirmed with an ACK. The TCP connection is closed in this way.





End printversion

In the following figure, the data stream is shown again horizontally from the sender's perspective to illustrate the sliding window technique:



#### Sliding window mechanism

- **I:** Segments that have been sent by the sender and have already been acknowledged by the receiver. The data acknowledged with ACK move the left edge of the window to the right; the size of the current window is indicated in the ACK segment with the window field.

- **II:** Segments that have been sent by the sender and have not yet been acknowledged by the receiver.
- **III:** Segments that may still be sent without waiting for an acknowledgment from the receiver.
- **IV:** Segments outside the send window, which may not be sent yet.



websource

The University of Innsbruck provides an [animation](#) about this topic.

### 5.3.4.2 Zero Window Probe

If the buffer is completely filled at the receiver side, a deadlock situation can arise. The receiver informs in an acknowledgment that the receive window is zero. So the sender cannot send any further data. The sender therefore waits. However, the receiver always only responds with an acknowledgment to new data received from the sender (a usual one-way data transfer is assumed, e.g. copying a file from A to B). If it does not receive any new data, it will also wait. So both sides wait.

TCP uses a timer (**persist timer**) to resolve this situation. This is used to send a so-called zero window probe, which is supposed to cause the side that indicated the zero window to send an ACK segment with the current window size. The window size can either continue to be zero, or there may be free space again after data has been fetched by the application.

The **zero window probes** (RFC 1122, RFC 6429) are sent until either there is free space available again or the application is terminated. A backoff algorithm is used so that the probes are sent at exponentially longer time intervals.

### 5.3.4.3 Nagle Algorithm

An unfavorable situation with respect to the efficiency of TCP exists when segments with few user data are often transmitted. This is called “**silly window syndrome**” (SWS). It can occur for two reasons:

1. The receiver indicates after a zero window that there is now some space in the receive buffer, in order to receive some (but little) data. It does not wait until more space is available.

2. The sender transmits many small segments, rather than waiting for further data to send a larger segment.

The Nagle algorithm (RFC 1122) prevents the SWS. When applying the algorithm, data is first collected so that segments which are as full as possible are transmitted (Maximum Segment Size user data).

This algorithm, however, leads to a delay, which is not desirable for interactive applications. For example, keystrokes could be transferred to another computer via ssh. Of course, it would be unfavorable in this case if you had to type until a segment is completely full. The use of the technique can be prevented in these cases with the TCP\_NODELAY option.

## 5.3.5 Congestion Control



arrangement

### 5.3.5 Congestion Control

#### 5.3.5.1 Maximum Segment Size

#### 5.3.5.2 Additive Increase / Multiplicative Decrease

#### 5.3.5.3 Slow Start

#### 5.3.5.4 TCP Tahoe and Reno

#### 5.3.5.5 Further Congestion Control Algorithms

#### 5.3.5.6 Congestion Control for Large High Bit Rate Networks

#### 5.3.5.7 Explicit Congestion Notification

#### 5.3.5.8 Fairness

The congestion control of TCP is intended to protect the network against permanent overload. In contrast to flow control, it is not about the overloading of end systems, but an overloading of network components through which too much data is transferred.

The algorithms for the TCP congestion control are described in RFC 5681. An important contribution to the basic principles was made by Van Jacobson (see [paper from 1988](#) ). In the following sections, the relevant mechanisms are explained step by step.



In the online version an video is shown here.

Link to video : [http://www.youtube.com/embed/w\\_gmr7f8WJE](http://www.youtube.com/embed/w_gmr7f8WJE)

**Congestion control**

### 5.3.5.1 Maximum Segment Size

The **maximum segment size** (MSS) is an important basic size of TCP, which is negotiated in the TCP connection setup. This variable is used to determine how many payload data are to be transmitted within a TCP segment. This determination could be made independent from the conditions on the Data Link Layer, especially since these layers are not adjacent according the OSI model. However, for performance reasons, it is useful to deviate from this in order to avoid inefficient fragmentation through the Internet Protocol.

Fragmentation occurs if IP packets are larger than the MTU at the Data Link Layer. As you know it already, TCP segments are transmitted in IP packets, so that you have to pay attention to the size conditions for the TCP segments. The MSS is therefore selected as MTU minus the sizes of IP header and TCP header. In case of IPv4, the header size is 20 bytes, and in case of IPv6, the header size is 40 bytes. The TCP header has 20 bytes. This specification therefore optimizes the MTU use and prevents fragmentation.



task

#### Task: MSS for Ethernet

How large is the MSS for Ethernet and IPv4?

##### Solution

In case of Ethernet II, the calculation is:

$$\begin{aligned} \text{MSS} &= 1500 \text{ bytes (Ethernet II MTU)} - 20 \text{ bytes (IP header)} - 20 \text{ bytes (TCP header)} \\ &= 1460 \text{ bytes} \end{aligned}$$

In case of IEEE Ethernet, the calculation is:

$$\begin{aligned} \text{MSS} &= 1492 \text{ bytes (IEEE Ethernet MTU)} - 20 \text{ bytes (IP header)} - 20 \text{ bytes (TCP header)} \\ &= 1452 \text{ bytes} \end{aligned}$$

### 5.3.5.2 Additive Increase / Multiplicative Decrease

The mechanisms of TCP congestion control are introduced successively in the following. The underlying model is the **additive increase / multiplicative decrease** model (AIMD, see [paper](#) ).

The **additive increase** part means that an end system increases the data rate linearly. In an example situation, the system sends a number  $n$  of TCP segments, each containing a data set according to the MSS, as fast as possible. If all segments are confirmed, which should be the case approximately after the RTT if the transfer is successful, the data rate is increased by one MSS. This means that afterwards  $n + 1$  segments are sent shortly after each other. As long as there are no losses, the data rate is always increased again by one MSS.

If, however, all or part of the acknowledgments are missing, TCP assumes that there is congestion in the network. TCP thus assumes that intermediate systems in the network, i.e. routers or switches, are overloaded, and that therefore segments in the intermediate systems have been discarded. This assumption is true for fixed networks in most cases, but segments may also be lost due to other reasons, e.g. if bit errors have occurred. In wireless transmission, this can occur more frequently, which means that the assumption of congestion is unfavorable.

TCP reacts to the lack of acknowledgments with **multiplicative decrease**, which means a significant reduction in the data rate. The data rate is bisected. The bisection of the data rate is carried out because the intermediate systems in the network are significantly relieved in this way. The buffers are completely filled with packets waiting for their forwarding. Therefore, the possibility should be provided to transfer on the queued packets. Afterwards, TCP increases the data rate linearly again.

So you can see that you do not get a constant data rate; rather, a permanent data rate testing takes place. The data rate is increased linearly until losses occur, then it is significantly reduced, and then it is linearly increased again until a reduction is necessary again after losses, etc.

The number of bytes that can currently be transmitted with directly successive segments is called the **congestion window**. It must also be noted at this point that the described Flow Control also exists parallel, which determines the receive window. Both thus limit the data rate of the sender, who must comply to the minimum of the two values.

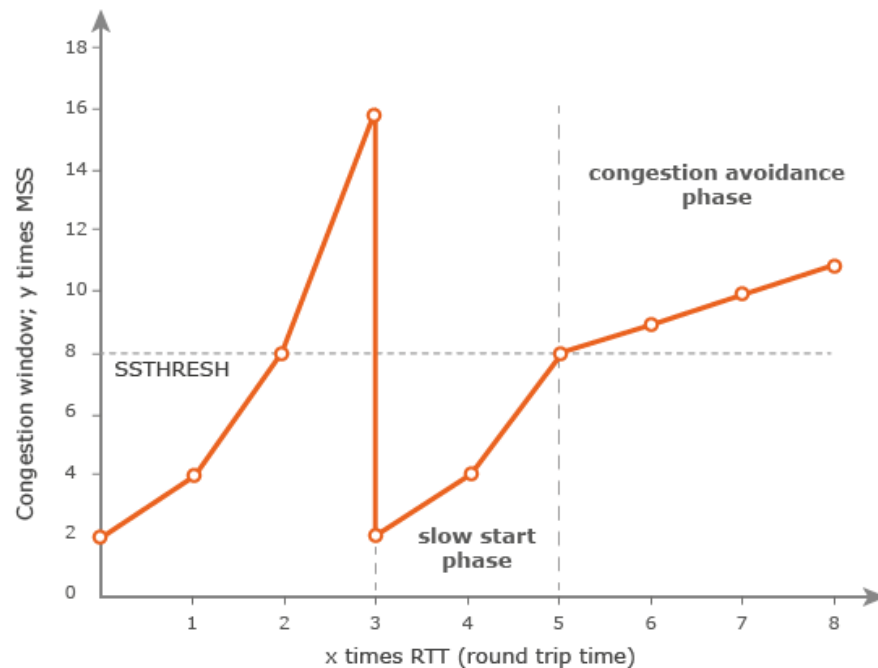
### 5.3.5.3 Slow Start

The AIMD method has an important disadvantage at the beginning of TCP connections. If a relatively high bit rate is possible in the network, then it takes a relatively long time with the linear increase of the data rate until a high bit rate is reached. This is particularly unfavorable in view of the fact that many TCP connections exist only for a short period of time, which is in particular the case when web pages are retrieved.

Therefore the idea of the **slow start** is to initially increase **the data rate exponentially** until a predefined threshold `ssthresh` (slow start threshold) is reached or until losses occur. At the beginning of the slow start, the congestion window (`cwnd`) is often set to 2 MSS, which means that two segments can be sent at once without waiting for an acknowledgment (other values are also possible depending on the MSS, see RFC 5681). Each confirmed segment increases `cwnd` by 1 MSS, i.e., if both segments are acknowledged `cwnd` is set to 4 MSS. Now 4 segments can be sent at once. When all four segments have been acknowledged, `cwnd` is set to 8 MSS, and so on. This exponential increase causes TCP to approach the possible bit rate very quickly. If the threshold (`ssthresh`) is reached or exceeded, `cwnd` is only increased linearly by a maximum of 1 segment per round trip time. This behavior is called **congestion avoidance**. `Ssthresh` can be set arbitrarily high at the beginning of the data transfer - it is usually set to the window offered by the receiver, e.g. 32 Kbytes.

If an overload occurs, the threshold (slow start threshold) is obviously too high and not suitable for the current conditions of the network. Therefore, the threshold is recalculated if there is an overload. The new threshold is calculated the last values minus half of the number of unacknowledged data segments. Then, a slow start is performed again, i.e. the method is used not only at the beginning of a TCP connection, but also repeatedly after losses.

Let us assume that overload occurs after 16 segments have been sent at once and none of these segments has been acknowledged. Then, the threshold `ssthresh` is set to 8 MSS and `cwnd` is reduced to 1 MSS. After this, a slow start is performed again, and `cwnd` is subsequently increased to 2, 4 and 8. Then, above the threshold, `cwnd` is only increased linearly to 9, 10, ..., until an overload is detected again. Since the process repeats, there is a **sawtooth curve** when the transmission rate of TCP is plotted over time.



“Slow start” and “congestion avoidance” phases



important

The bite rate of TCP is permanently changing and adapted to the current transmission capacity of the network. TCP can therefore adapt itself to different conditions in the network.

### 5.3.5.4 TCP Tahoe and Reno

In the previous discussion, it has not yet been explained exactly how an overload in the network can be noticed. There can be two situations.

- **Retransmission timeout:** The transmitted segments are no longer acknowledged, since, for example, a router has dropped them because of overload.
- **Duplicate acknowledgments:** In a router, a segment is dropped because of overload, or the segment does not arrive correctly due to bit errors. The next segment and perhaps further segments are again provided to the receiver. Then the receiver can only acknowledge the segment that it received before the lost segment. This leads to duplicate acknowledgments (see [Fast Retransmit](#)).

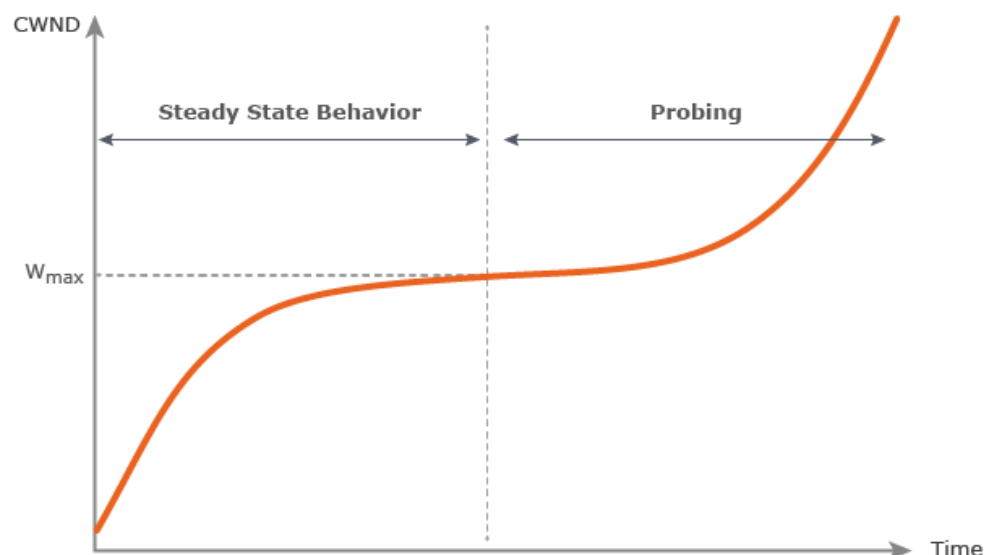
In the original variant of TCP Tahoe, both situations are handled the same way. The *cwin* is set to 1 MSS and a Slow Start is carried out, with the *ssthresh* corresponding to the bisected previous *cwin*. After the *ssthresh* is reached, the data rate is increased linearly again.

In TCP Reno, however, situations are differentiated. When several duplicate ACKs are received, the sender can assume that the data rate with which it uses to send segments is actually almost appropriate – only one segment has been lost after all. It is therefore not reasonable to carry out a slow start. Instead, the *cwin* is bisected, and then the data rate is again increased linearly. This behavior is called **fast recovery** (RFC 5681).

### 5.3.5.5 Further Congestion Control Algorithms

After the definition of TCP Tahoe (1988) and TCP Reno (1990), a series of other algorithms for congestion control was established in the 1990s and 2000s. The development over time is presented in [Wikipedia](#). An [overview article](#) provides a systematic classification of the methods.

An important current example is the algorithm [CUBIC](#), which is the standard algorithm for Linux and which should also be used for QUIC (see [Other Transport Layer Protocols](#)). The algorithm got its name from a cubic function ( $x^3$ ), which describes its behavior.



Data rate of the CUBIC algorithm



The figure shows the increase of the data rate as long as no losses occur. If losses occur, the increase starts again with adjusted parameters. The data rate initially rises relatively quickly but then approaches a plateau region more slowly. The idea is that the data rate can stay in a relatively high range longer without having to leave it soon after loss. Only after a longer time the data rate increased more aggressively again. This can be useful if capacities have become free again in a network because other transmissions have been terminated.



practice

With Linux, you can easily find out which congestion control mechanism is used. This is contained in the file `/proc/sys/net/ipv4/tcp_congestion_control`. If you want to use a different congestion control algorithm, you can simply write its name into the file. It may need to be loaded first. You can find out if this is the case with the following file: `/proc/sys/net/ipv4/tcp_available_congestion_control`. For more information (available kernel modules, reloading core modules), see this [blog entry](#). For Windows, the algorithms Compound TCP, DataCenter TCP, or NewReno are available (see the “congestion provider” parameter in the [command Set-NetTCPSetting](#)); NewReno (RFC 6582) is the default for clients (see also the [Wikipedia article about Compound TCP](#)).

A further development direction, which is however only briefly mentioned here, is [multipath TCP](#). A TCP stream is to be transmitted here in parallel over several paths in which individual TCP connections exist. This can be interesting, for example, if an end system is simultaneously connected to the Internet via Ethernet and WLAN. The advantage is not only a higher data rate, but also a better robustness in case of problems with a network connection.

## 5.3.5.6 Congestion Control for Large High Bit Rate Networks




arrangement


### 5.3.5.6 Congestion Control for Large High Bit Rate Networks

#### 5.3.5.6.1 In Depth: Bit Rate and Latency

#### 5.3.5.6.2 Sequence Number Overflow

When using the TCP congestion control, there is a particular challenge for networks with a high bit rate and a large distance between the sender and the receiver. This applies, for example, to scientific networks, where nowadays data rates of 10 or 100 Gbit/s are usual. Due to high delays, there is the difficulty that you know only after the RTT whether the data arrived correctly, and then a decision to increase or reduce the data rate can be made. You therefore cannot adapt quickly to changing conditions in the network. A reduction in the data rate after loss can lead to a consequence that the network's capacity is not used in the best way for an extended period of time. For example, the New Zealand research network REANNZ is therefore working on a packet loss rate very close to zero (see [REANNZ website](#) ) .

Another point is that you need large buffers on the sender and the receiver side because there are many data on the line in such networks (see [In Depth: Bit Rate and Latency](#)). Otherwise, it could happen on the sender side that acknowledgments for many data arrive, but there are not enough new data available in the buffer for the further transmission. In the same way, however, also small sizes of receive buffers slow down the data rate since the sender cannot transmit more data than there is free space in the buffer.

At a low RTT, this problem does not occur because acknowledgments are obtained quickly. This can lead to false assessments by users who think wide area networks are responsible for poor performance. For example, you have tested an application in a local area network, where a high TCP data rate has been reached, and then you release it for general use over wide area networks. In this case, the TCP data rate can be low, something for which the wide area network seems to be responsible. However, this conclusion is (almost always) wrong because it is typically due to insufficient buffer sizes (see [TCP tuning website from ESnet](#) ) . Another reason is often low bit rates for network access.

#### 5.3.5.6.1 In Depth: Bit Rate and Latency

The relationships are to be investigated again here in more depth. We consider two questions related to transmission lines.

- How long does it take to transfer a certain amount of data?
- How many bits can be transferred on one line at a time?

**How long does it take to transfer a certain amount of data?** This question is not easy to answer since different factors must be taken into account, which are referred to collectively as **latency**:

- **Propagation delay**


The signals propagate with light speed in the transmission medium. In a vacuum, this is about 300,000 km/s, in a fiber-optic cable about 200,000 km/s, and in a copper cable about 240,000 km/s. If we transmit a signal within a company (1 km) on a fiber-optic link, this obviously takes significantly less than 1 ms until it arrives at the receiver; across the whole of Germany (1,000 km), it takes 5 ms; a signal to Australia (20,000 km) requires 100 ms. A signal via satellite to Australia (72,000 km) requires 240 ms. The speed of light is greater in a vacuum than in a fiber-optic cable - but the distance to the satellite is much higher. Please note that the one-way delays have been indicated here and not the RTT.

- **Transmission delay for data transmission**

When few data are transferred, it is faster than if many data are transferred. Transferring a data amount of 1 Mbyte (i.e. 8 Mbit) over an ISDN line with a bandwidth of 64 kbit/s takes  $8,000,000/64,000$  s, i.e. 125 s. The same data quantity over a line with a bit rate of 1 Gbit/s takes only 8 ms; At a bit rate of 10 Gbit/s, it only takes 0.8 ms. These calculations are significantly simplified because the data must be divided into many TCP segments between which there are inevitable time gaps. Effects such as those of error control and congestion control must also be considered.

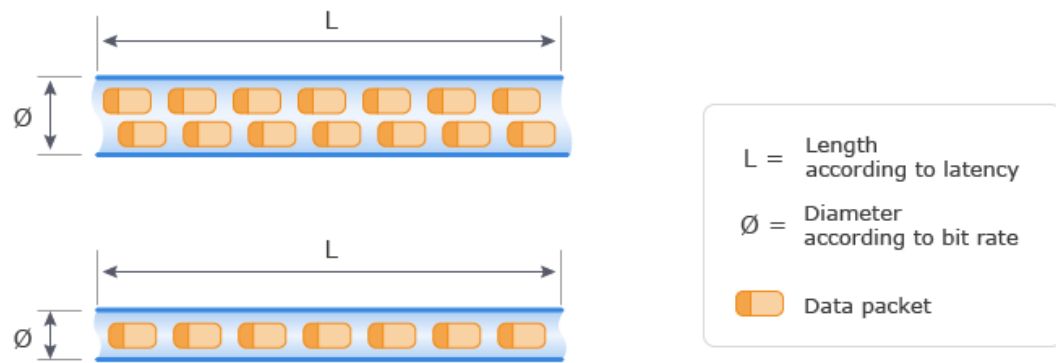
- **System-related delay**

The signals are delayed in the computer of the sender and receiver by internal processing. On their way, they are routed through several routers, switches, or other intermediate systems, thereby delaying transmission. The system-dependent delay will not be discussed further below.

The total delay (latency) is composed of these three factors. The relationship between propagation delay and transmission delay is presented in an [animation from the University of Innsbruck](#) .

#### **How many bits can be transferred on one line at a time?**

A line can be thought of as a hollow tube whose length is the delay and whose diameter is the bit rate. The following figure shows two lines with the same delay (length of the line) but different bit rates (diameter of the line):



#### Two lines with different bit rates

The line is optimally utilized if there are no transmission gaps between the packets. The product of bit rate and delay is the measure of the **capacity** of a line. It shows the maximum bits that can be transmitted simultaneously in the line. The term **bandwidth delay product** has unfortunately been established in the literature, even though it concerns bit rate (unit bit/s) and not bandwidth (unit Hz).

Senders and receivers should be able to make optimum use of the available capacity. As already mentioned, sufficiently large buffer sizes are required on both sides.



#### Examples of bit rates, RTT and bandwidth delay product

Begin printversion

Kind of link	bit rate	RTT	bit rate $\times$ delay
ISDN	64 kbit/s	30 ms	240 bytes
2 Mbit/s line	2 Mbit/s	60 ms	15,000 bytes
2 Mbit/s satellite link	2 Mbit/s	500 ms	125,000 bytes
10 Mbit/s Ethernet local	10 Mbit/s	3 ms	3,750 bytes
1 Gbit/s line	1 Gbit/s	30 ms	3,750,000 bytes
10 Gbit/s line	10 Gbit/s	30 ms	37,500,000 bytes



#### Examples of bit rates, RTT and bandwidth delay product

End printversion



task

**Task: Bit rate and delay I**

For a data transfer from Germany to the USA, a one-way delay of 100 ms must be considered, which results from the run time and the delay time in the routers and computers.

You have installed a 1 Gbit/s line and usually transfer large files with a length of 1 MByte (= 8 Mbit). How long does it take until such a file completely arrives at the receiver?

You have installed a new 10-Gbit/s line at a high cost and hope that now everything will be much faster. How long does it take until the data is received by the receiver?

**Solution**

1 Gbit/s line:  $100 \text{ ms} + 8 \text{ ms} = 108 \text{ ms}$

10 Gbit/s line:  $100 \text{ ms} + 0.8 \text{ ms} = 100.8 \text{ ms}$

So you can see that 10 times the data rate only leads to less than 10 percent reduction in the transmission time. The inevitable one-time delay therefore has an important influence.

**Task: Bit rate and delay II**

Consider the same task in local area network, where the delay time is 1 ms.

**Solution**

1 Gbit/s:  $1 \text{ ms} + 8 \text{ ms} = 9 \text{ ms}$

10 Gbit/s:  $1 \text{ ms} + 0.8 \text{ ms} = 1.8 \text{ ms}$

In the local area network, the increase of the bit rate is considerably more useful since the one-time delay is short.

**5.3.5.6.2 Sequence Number Overflow**

32 bits are available for the sequence numbers, which was regarded as completely sufficient for the bit rates that were possible at the beginning of the specification of TCP.

However, the following table shows that it can now take less than 1 s for the sequence numbers to repeat.



#### Examples of bit rates, RTT and bit-rate delay product

Begin printversion

Kind of link	bit rate	transfer time of $2^{32}$ bytes
ISDN	64 kbit/s	~ 6 days
2 Mbit/s line	2 Mbit/s	~ 5 hours
10 Mbit/s Ethernet	10 Mbit/s	~ 1 hour
1 Gbit/s link	1 Gbit/s	~ 43 s
10 Gbit/s link	10 Gbit/s	~ 0.43 s

Examples of bit rates, RTT and bit-rate delay product

End printversion

To avoid confusion, the TCP option **timestamp** (current specification in RFC 7323) was introduced, which can be agreed during the TCP connection setup. The segments are then sent with a timestamp in order to be able to reliably recognize the overflow.

### 5.3.5.7 Explicit Congestion Notification

As we have learned so far, congestion is detected indirectly by TCP. If segments are lost, TCP assumes that this was due to an overload of network components. However, this does not have to be correct since segments can fail to arrive correctly due to bit errors, and the acknowledgment is not sent then. This is a common situation, especially in case of wireless transmissions using Wireless LAN or mobile radio technologies. So if there is no congestion, then it is inappropriate if the data rate is reduced. Therefore, it would be better to be able to recognize a congestion situation with more certainty. In addition, it is desirable to reduce data rates earlier in a congestion situation so that buffer overflows do not occur.

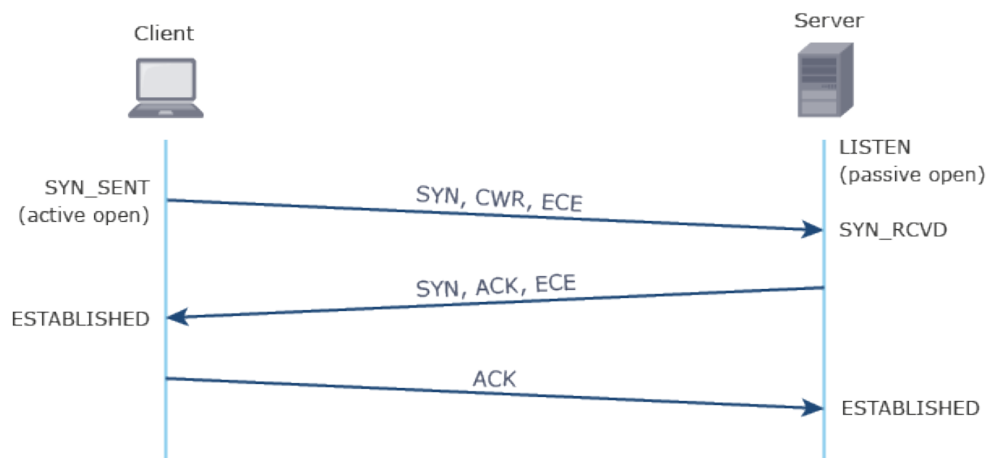
With **ECN (Explicit Congestion Notification)** (RFC 3168), a method has been developed that addresses exactly this issue. A router must recognize in time that an overload situation is going to occur, and it must explicitly inform the affected end systems. The senders then reduce their transmission bit rates so that no packets need to be dropped. However, this method assumes that routers can actively manage their queues and do not just react after the queues are full (so-called **active queue**

**management**, RFC 7567). For this purpose, the **RED (Random Early Deletion)** method is often used in the router, but there are also various other approaches.



reflection

A router triggers a sender to reduce its data rate through a congestion message.



#### TCP connection setup with

In the TCP connection setup, the receiver is notified of the ability to respond to ECN by setting the CWR and ECE flag in the SYN segment. The receiver confirms its ability by setting the ECE flag in the SYN/ACK segment. This means that both communication partners know that they can process congestion messages from the router.

The following animation shows the steps in this process:

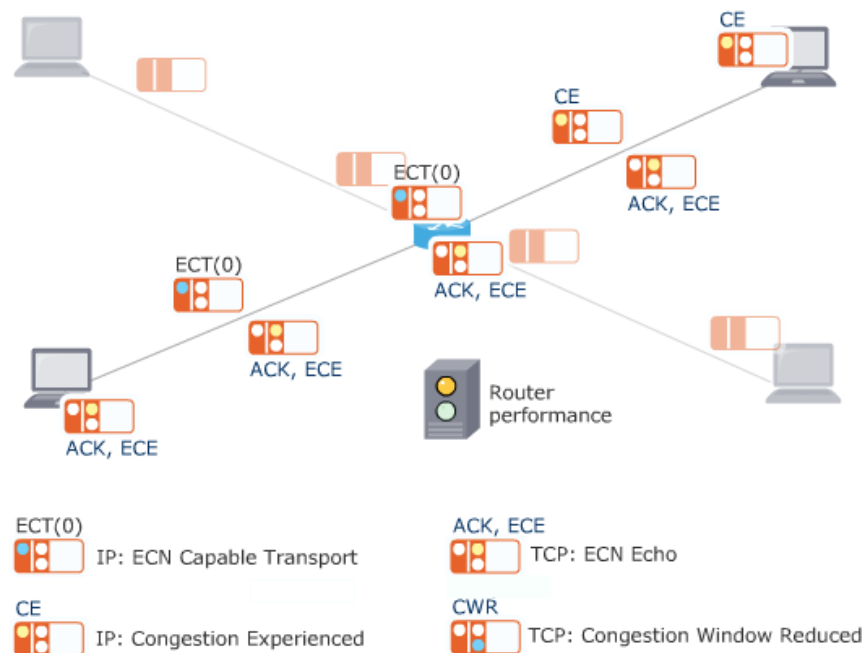


In the online version an animation is shown here.

```
<loop_video source="ueberlastvermeidung_ecn_pro_ani_en.mp4" width="698"
height="460"> </loop_animation>
```

**Overload prevention through ECN**

Begin printversion



It is indicated in the IP headers that routers should send notifications about evolving congestion: ECT zero is set. If there is an evolving congestion, the routers send the indication “congestion experienced” in the IP header to the receiver.

Consequently, the receiver sends the ECN echo flag back to the sender in the following TCP ACK segments to inform the sender that the router has identified evolving congestion.

If the sender gets such segments, it reduces its bit rate and indicates this in the TCP header by setting the “congestion window reduced” flag. The receiver continues to send TCP headers with the ECN echo flag until it receives a TCP header with the “congestion window reduced” flag.


End printversion

When the connection has been established, the sender sets the ECN field in the IP datagram to the value ECT (0), (ECN capable transport). If an overload evolves in the router, the router sets CE (congestion experienced) in the IP datagram. If the receiver has received an IP packet with CE, this information must be transmitted to the sender of the packet. For this purpose the ECE (ECN echo) flag is set in the following TCP-ACK segments. The receiver bisects its congestion window (cwnd), reduces the threshold (sssthresh) and then sends a segment with the CWR (congestion window reduced) flag.





annotation

When considering the relevance of this method in reality, the following circumstances have to be taken into account. Today's wide area networks have a lot of capacity and can be upgraded relatively easy, even in the case of bottlenecks, thanks to technologies such as [DWDM](#) . Nowadays, bit rates are limited by network access (e.g. DSL connections), so the use of ECN in provider networks is not really useful.

### 5.3.5.8 Fairness

If you look at the effects of TCP congestion control on multiple TCP connections, you can see that the connections get a roughly equal data rate after some time. To understand this, you should consider two situations.

- The network is not fully utilized: If there is still free capacity in the network, the bit rates are increased for all TCP connections. Viewed in absolute terms, each TCP connection's bit rate is increased approximately to the same extent. If, for example, you had two TCP connections with 20 Mbit/s and 60 Mbit/s, then these could have bit rates of 40 Mbit/s and 80 Mbit/s respectively after some time. That is, both have increased their data rate by 20 Mbit/s.
- The network is overloaded: When the data rates are increased, then the point where the network is overloaded and data units are lost is reached quickly after full capacity utilization. The data rates are then bisected in the known manner, whereby a connection with a high data rate must accept a larger reduction of the bit rate in absolute terms. If we continue the previous example, the bit rate of the first connection after the bit rate reduction is 20 Mbit/s and the of the second connection is 40 Mbit/s. We can still continue the example and assume a uniform increase so that the bit rates are increased by 30 Mbit/s for each connection. Then the resulting bit rates would be 50 Mbit/s and 70 Mbit/s. After a further bisection, the bit rates would then be 25 Mbit/s and 35 Mbit/s. So you see that the data rates converge more and more.



annotation

The given explanation is certainly simplified because it fits primarily to the basic algorithm AIMD and does not refer to more modern variants. In addition, the segment losses do not have to affect both connections in an equal manner. If, for example, a segment loss affects one TCP connection and it reduces its data rate, then more capacity

could be available for the other TCP connection. Segment loss related to it then occurs later.

With respect to TCP fairness, it is important to understand what fairness is about. Fairness means that TCP connections get roughly the same data rates. However, this does not result in fairness at the level of the end systems. For example, if you are performing a download via one TCP connection while another computer with which the first computer is sharing Internet access is downloading a website via six parallel TCP connections, then the second computer will get about 6/7 of the available bit rate. This case is very relevant in practice since web pages are frequently downloaded via parallel TCP connections. Parallel TCP connections are therefore often considered as a way to increase the bit rate.



In the online version an video is shown here.

Link to video : <http://www.youtube.com/embed/EB2Ko2Sq8nY>

#### Fairness

UDP does not have a congestion control mechanism. This means that bit rates are set for UDP streams and the data are then transferred to the network using these data rates. UDP does not worry about possible data loss. The consequence of this behavior is that UDP can completely displace TCP. If UDP streams still leave remaining capacities, then TCP connections share them fairly among each other.

## 5.4 Other Transport Layer Protocols

In addition to the protocols TCP and UDP, there are other Transport Layer protocols. Even though new Transport Layer protocols only have to be implemented by end systems according to the previous explanations in the chapter, there is in practice a major problem with so-called middle boxes. This term refers to intermediate systems in the network such as firewalls, which also have effects on the Transport Layer. Their rules are designed for TCP, UDP and ICMP, and typically discard other protocols at this layer. Therefore, commercial applications will not rely on an alternative Transport Layer protocol if many users cannot use the application then. The **Stream Control Transmission Protocol** (SCTP, RFC 4960), which is similar to TCP, but does not take care of sequence preservation, therefore did not become very important. The same is true for the DCCP protocol, which is similar to UDP but contains congestion control mechanisms.

Therefore, Google went another way in developing its own Transport Layer protocol called QUIC (**Quick UDP Internet Connections**), which is in the meantime developed further at [IETF](#). This protocol, which is intended in particular to accelerate the delivery of web pages in combination with [HTTP/2](#), is based on UDP. QUIC takes up ideas from some other protocols and integrates [Transport Layer Security](#) in particular. Even though the Google Chrome browser and Google's servers already implement the protocol, the development is still work-in-progress (see [development document](#)).

## 5.5 Socket API



arrangement

### 5.5 [Socket API](#)

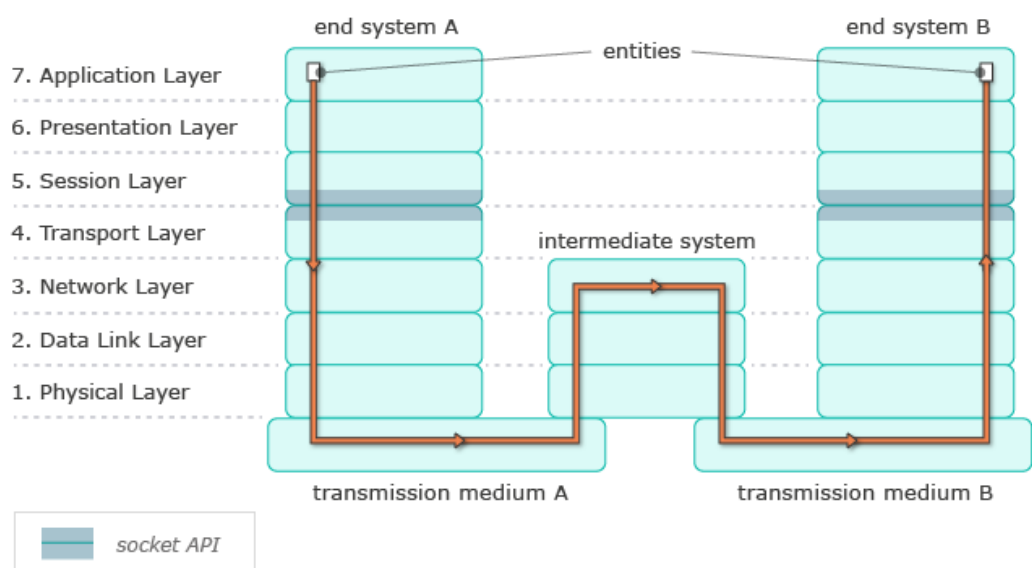
#### 5.5.1 [Connectionless Application](#)

#### 5.5.2 [Connection-Oriented Application](#)

#### 5.5.3 [Blocking Calls](#)

#### 5.5.4 [Socket API Overview](#)

Application programs access the protocols of the Transport Layer (TCP and UDP) via the **Socket API** (application programming interface). This interface is the link between the applications (layers 5, 6 and 7 in the OSI Model) and the communication protocols of layers 1 to 4 of the OSI Model that are integrated in the operating system kernel.





### Socket API

The Socket API is based on the original **Berkeley sockets** and is used in most applications today. Although the Socket API comes from the BSD/UNIX environment, it was ported exactly to Windows computers and can be used there with the same calls as on UNIX.

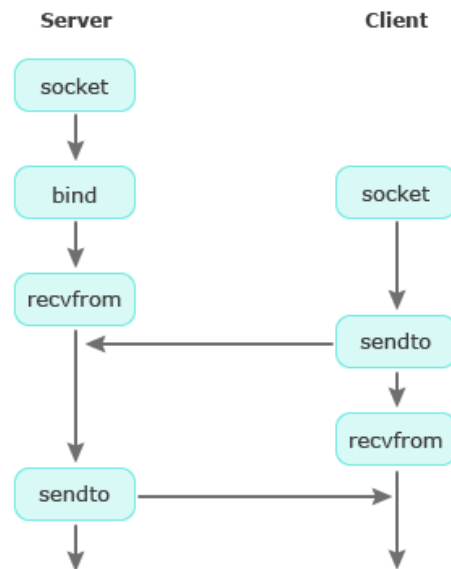
A **socket** describes a **connection endpoint** in the user program. The operating system provides resources to a socket, such as the transmit and receive buffers. Communication on the Internet always takes place between a client and a server, which must be indicated with the socket. A server waits for a request from a client. Therefore, a server must always be active before a client. Typically, servers run 24 hours a day and 7 days a week in so-called 24/7 operation, although this does not have to be the case. However, a server must be started before it can be addressed by a client.

There are three different types of sockets on the Internet:

- Connectionless sockets (**datagram sockets**). **UDP** is used here.
- Connection-oriented sockets (**stream sockets**). **TCP** is used here.
- Raw sockets, which do not use a Transport Layer protocol and instead directly use Network Layer protocols such as IP or ICMP. These sockets are rarely used and will not be described further here.

## 5.5.1 Connectionless Application

The following figure shows the calls to the Socket API for a connectionless communication with UDP.



**Calls to the socket API for a connectionless communication with UDP**

The server opens a datagram socket to use UDP. It assigns a port number to the socket (**bind**). This will in general be a well-known port number from the range 1 - 1023, e.g. port number 69 for a TFTP server.

Then a read operation (**recvfrom**) is performed to see if data has been received from a client. Since there is no information about the client in the case of a connectionless communication, this information (IP address and port number of the other side) must be supplied by the user program during each read operation.

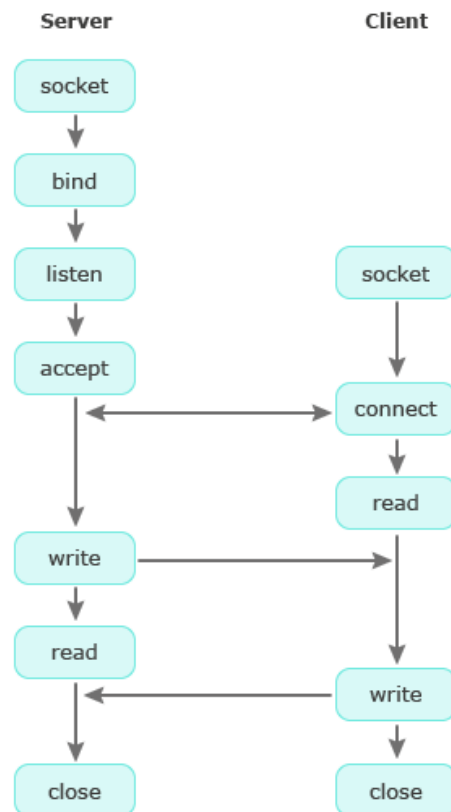
The client opens a datagram socket for UDP and can then immediately send data to the other side (**sendto**). For doing so, it must specify the IP address and the port number of the receiver.

What port number does the client socket get? The above figure is slightly simplified. Thus, a **bind** call can be made between the socket and **sendto** calls, with which a short-lived, unused port number greater than 1023 is assigned to the socket. If the **bind** call is not performed, the operating system automatically assigns an unused port to the client socket – this is the usual case.

Since there are no connections related to the use of UDP, connections also do not have to be decommissioned. However, unused UDP sockets should be closed with the **close** call to release resources in the operating system.

### 5.5.2 Connection-Oriented Application

The following figure shows the calls to the socket API for a connection-oriented communication with TCP.



**Calls to the Socket API for a connection-oriented communication with TCP**

The server opens a **stream socket** to use TCP. It assigns a port number to the socket (**bind**). This will in general be a well-known port number from the range 1 - 1023, e.g. port number 80 for a WWW server. The socket is then marked as passive with the **listen** call. Afterwards, the server is ready to accept requests from the client. The **accept** call waits until a connection request is received from the client. Then another socket is opened automatically, which is used for the communication with the client. The passive socket continues to exist and can now serve another connection request from another client. This allows several simultaneous connections to be served by a single server. An FTP server typically operates in this way.

The client opens a stream socket for TCP and then tries to connect to the server (**connect** call). What port number does the client socket receive? The above figure is slightly simplified. Thus, a **bind** call can be made between the socket and connect call, with which an unused port number greater than 1023 is assigned to the socket. If the bind call is

not performed, the operating system automatically assigns an unused port to the client socket – this is the usual case.

After the connection setup (TCP three way handshake) data can be exchanged. Which party first sends data or receives data depends on the application program. The above example could be an FTP connection. After the connection is established, the server first sends a welcome message and awaits the client user name and password.

For a TCP connection, simple read and write calls can be used (**read**, **write**, **recv**, **send**), which do not transfer address information (IP address and port number) to the application program. This information has already been communicated to the other side during the connection setup and cannot be changed during a connection.



To terminate a connection, **close** calls are used.

### 5.5.3 Blocking Calls

By default, all calls to the socket API that wait for data from the network - such as `recvfrom`, `read`, `recv`, `accept` - are blocked until data is received from the other side (so-called **blocking mode**). The application program can therefore do nothing else but wait during this time. In particular, it may happen that the connection is interrupted: An excavator cuts the line for example. Then the other side would just be waiting for nothing. To avoid this error situation, sockets are usually used in the so-called **non-blocking mode** (`ioctlsocket` call with parameter **FIONBIO**: file input/output non-blocking I/O). If a read command is now called and no data is present, an error number (**EWOULDBLOCK**) is passed to the application program, which then does not wait for data but can perform other tasks.

### 5.5.4 Socket API Overview

Besides the already described important socket API calls, there are still a number of other calls, some of which are self-explanatory.

If you want to work with sockets in connection with Python, you should look at the slides on the [companion website](#)  for the book by Kurose/Ross (*KuRo14* ). At the end of the slide set about the application layer, there is code for the implementation of UDP/TCP sockets.

**Socket functions**

\* These calls can block

Begin printversion

Function	Description
<b>accept()*</b>	A connection request is confirmed and assigned to a newly generated socket. The original sockets returns to the listen state.
<b>bind()</b>	Assignment of a port number to a socket
<b>closesocket()</b>	Closing of a socket. The socket is blocked on using the SO_LINGER option.
<b>connect()</b>	Connection request for the indicated socket
getpeername()	IP address and port number of the communication partner
getsockname()	Own IP address and port number
getsockopt()	Read socket options
htonl()	Converting from host byte order towards network byte order (32 bits)
htons()	Converting from host byte order towards network byte order (16 bits)
inet_addr()	Converting from Internet standard dotted notation into a 32 bit IP address
inet_ntoa()	Converting from IP address into Internet standard dotted notation "a.b.c.d"
ioctlsocket()	Configuration of special parameters, e.g. FIONBIO
<b>listen()</b>	Waiting for incoming connection requests
ntohl()	Converting from network byte order towards host byte order (32 bits)
ntohs()	Converting from network byte order towards host byte order (16 bits)
<b>recv()*</b>	Read data using a TCP socket
<b>recvfrom()*</b>	Read data using a UDP socket (or TCP socket)
<b>select()*</b>	Asynchronous I/O multiplexing
<b>send()*</b>	write to TCP socket
<b>sendto()*</b>	write to UDP socket (or TCP socket)
setsockopt()	Setting of socket options, e.g. buffer sizes
shutdown()	Termination of read or write direction of a socket connection
<b>socket()</b>	Generation of a socket

**Socket functions**  
 \* These calls can block


End printversion

**Other socket functions**

\* These calls can block

Begin printversion



Function	Description
gethostbyaddr()*	Search for a host name based on an IP address
gethostname()	Search for the name of the own host
gethostbyname()*	Search for an IP address based on a host name
getprotobyname()*	Search for a protocol number based on a protocol name
getprotobynumber()*	Search for a protocol name based on a protocol number
getservbyname()*	Search for a service number based on a service name
getservbyport()*	Search for a service name based on a service number
 <b>Other socket functions</b> * These calls can block	

End printversion




#### Socket options

The socket options are read with `getsockopt ()` and altered with `setsockopt ()`.

Begin printversion

Value	Meaning	Default
SO_ACCEPTION	Socket is in listen mode	FALSE until listen() is called
SO_BROADCAST	Socket can transfer broadcast messages	FALSE
SO_DEBUG	Debugging is enabled	FALSE
SO_DONTLINGER	If true, the SO_LINGER option is not possible	TRUE
SO_DONTROUTE	Routing not allowed	FALSE
SO_ERROR	Read and correct error status	0
SO_KEEPALIVE	Keep alive packets are sent	FALSE
SO_LINGER	Provides the current linger options	1 on/off is 0
SO_OOBINLINE	Out-of-band-data is received as part of the usual data stream	FALSE
SO_RCVBUF	Buffer size of the receiver	Depends on implementation
SO_REUSEADDR	The same socket address can be used again	FALSE
SO_SNDBUF	Buffer size of the sender	Depends on implementation
SO_TYPE	Socket type: SOCK_STREAM or SOCK_DGRAM or SOCK_RAW	Is set on socket call
TCP_NODELAY	The Nagle algorithm is turned of on sender side	Depends on implementation


**Socket options**  
 The socket options are read with getsockopt () and altered with setsockopt ().

End printversion


## 5.6 Exercises - Transport Layer





task

### Tasks for beginners

#### Task 1:

Download the [Advanced IP Scanner](#)  tool. Do not change the default settings and perform a scan of your subnet. Use Wireshark to record the scan.

#### Task 2:

Examine the ARD [Mediathek](#)  (media center in English) and [livestream](#) . ARD is the first German public TV program. Use Wireshark to find out which Transport Layer protocols are used. Please note that videos usually generate a large amount of data.

Remark: If the results do not match to your expectations, make your own thoughts why it is different.

**Task 3:**

Examine whether there are differences between the browsers on video provisioning. Watch YouTube videos with Google Chrome and Firefox and check the Transport Layer protocol use via Wireshark.

**Task 4:**

Visit the [speedguide.net](https://www.speedguide.net) test page and check which TCP parameters of a test connection are examined.

***Tasks for advanced learners*****Task 1:**

Use **netstat** on the Windows command line and look at its options (`netstat -h`). Examine the network connections (command without parameters), statistics (`-s`), and routing table (`-r`).

Remark: Please note that the options are a bit different in other operation systems even though the functionality of the tool is basically the same.

**Task 2:**

Carry out a test with the [Network Diagnostic Toolkit](#). Consider the TCP parameter details that the server provides in the “advanced” tab.

**Task 3:**

A given Wireshark file contains a recorded TCP connection. Examine whether the TCP connection setup and close is as expected. Consider also the parameter negotiation during the connection setup.

**Task 4:**

For the following task, you will receive two small Java programs that represent a TCP server and a TCP client. You can run these programs on two computers and then set the parameters in an appropriate manner to check the flow control of TCP.

The setting of the parameters can be tricky. For this reason, you can also get a Wireshark record, which clearly shows the effects of flow control.

**Task 5:**

You can do the following task in two ways. If you do not have the option to set up the configuration as described below, then look at the Wireshark files with the performed experiments about TCP congestion control.

However, you will learn more if you perform the tests on your own. To do this, you need three computers, which you connect via a switch (linking them via WLAN would also work as well as a configuration of the network via virtual machines). Download the Linux distribution [WANem](#), which is a wide area network emulator. The middle computer is booted to run WANem. WANem can be used to influence data transmission. Bit rate limitations can be configured and artificial delays as well as packet losses can be generated. The two other computers must then be configured so that their communication runs over the WANem computer (usually they would communicate directly with one another). To achieve this, the routing must be changed. In Windows, the “route add” command, which requires administrator privileges in the DOS window, is used. So, for example, you write on the command line “route add 192.168.0.3 mask 255.255.255.255 192.168.0.2”. This sends data traffic to the IP address 192.168.0.3 to the “router” 192.168.0.2 (see also [instructions in the Internet](#)). 192.168.0.2 is the IP address of the WANem computer in this example. On the two end systems, one can then either install [iperf3](#) or its graphical interface [JPerf](#) to perform throughput tests between the computers with different parameter settings.

If you want to artificially influence the network transmission on your own computer, you can use a tool called [Clumsy](#).

## 5.7 Summary - Transport Layer


In this chapter, the most common Internet protocols used in the Transport Layer were described in detail, namely UDP and TCP. The following table shows a comparison of the features of the two protocols.



### Comparison of the features of TCP and UDP

Begin printversion

Feature	UDP	TCP
Error control	Detection of bit errors	Detection and correction of bit errors and segment losses
Maintaining sequence order	Not guaranteed	Guaranteed
Flow control	Not available	Available
Congestion control	Not available	Available
Segmentation	Unknown	Is carried out
Connection orientation	No	Yes

 **Comparison of the features of TCP and UDP**

End printversion



annotation


If features are missing in UDP, it is possible to add them at the Application Layer. This is done, for example, with DNS, where one waits for appropriate answers to requests and, if necessary, repeats the requests.

In anticipation of the next chapter, you can already find an overview of applications using TCP or UDP.



**Use of TCP and UDP by applications**

Begin printversion

Application	Transport Layer protocol	Reason
WWW	TCP	Reliability
E-Mail (SMTP, POP, IMAP)	TCP	Reliability
Telnet/SSH	TCP	Reliability
DNS	Usually UDP	Overhead of TCP, requests may be repeated; TCP for larger responses
DHCP	UDP	Overhead of TCP, requests may be repeated
VoIP (RTP)	Usually UDP	TCP retransmissions not useful; data rate reduction of TCP in case of losses unfavorable
 Use of TCP and UDP by applications		

End printversion