Computernetzwerke

Transportschicht

Sebastian Bauer

Wintersemester 2022/2023

Computer Engineering Curriculum

Hybrides Referenzmodell: Transportschicht

Adressierung in der Transportschicht

- Transportprotokolle (UDP und TCP) erweitern IP-Adresse um Port (16 bit-Zahl)
- Unterscheiden mehrere Verbindungen zwischen demselben Paar von Endpunkten
- Verfügbar für Anwendungen zusammen mit IP-Adresse über Sockets

Gruppen von Portnummern

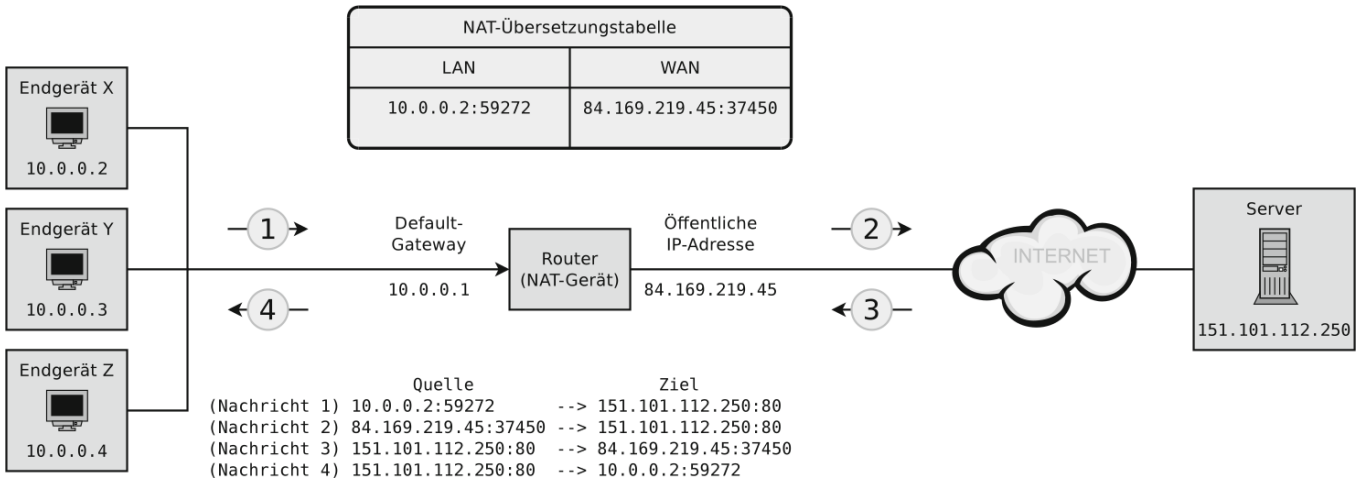
Name	**Portbereich**
Well Known Ports	0 bis 1023
Registered Ports	1024 bis 49 151
Private Ports	49 152 bis 65 535

Einige Standardports

Port	Name	Beschreibung
21	FTP	Dateitransfer
23	Telnet	Terminalemulation
25	SMTP	E-Mail-Versand
53	DNS	Auflösung Domainnamen in IP-Adressen
67	DHCP	Netzwerkkonfiguration
80	HTTP	Webserver
110	POP3	Zugriff auf E-Mails
143	IMAP4	Zugriff auf E-Mails
443	HTTPS	verschlüsselter Webserver
993	IMAPS	verschlüsselter Zugriff auf E-Mails
995	POPS	verschlüsselter Zugriff auf E-Mails

Portadressübersetzung

Ermöglicht Internetzugriff von Rechnern aus privaten Netzwerk



Netzwerkadressübersetzung (*network address translation*)

Netzwerkadressübersetzung (NAT) beschränkt sich auf die IP-Adresse, kennt also keine Ports. Der Begriff ist trotzdem häufig Synonym zu PAT.

Outline

1 UDP

2 TCP

3 Sockets

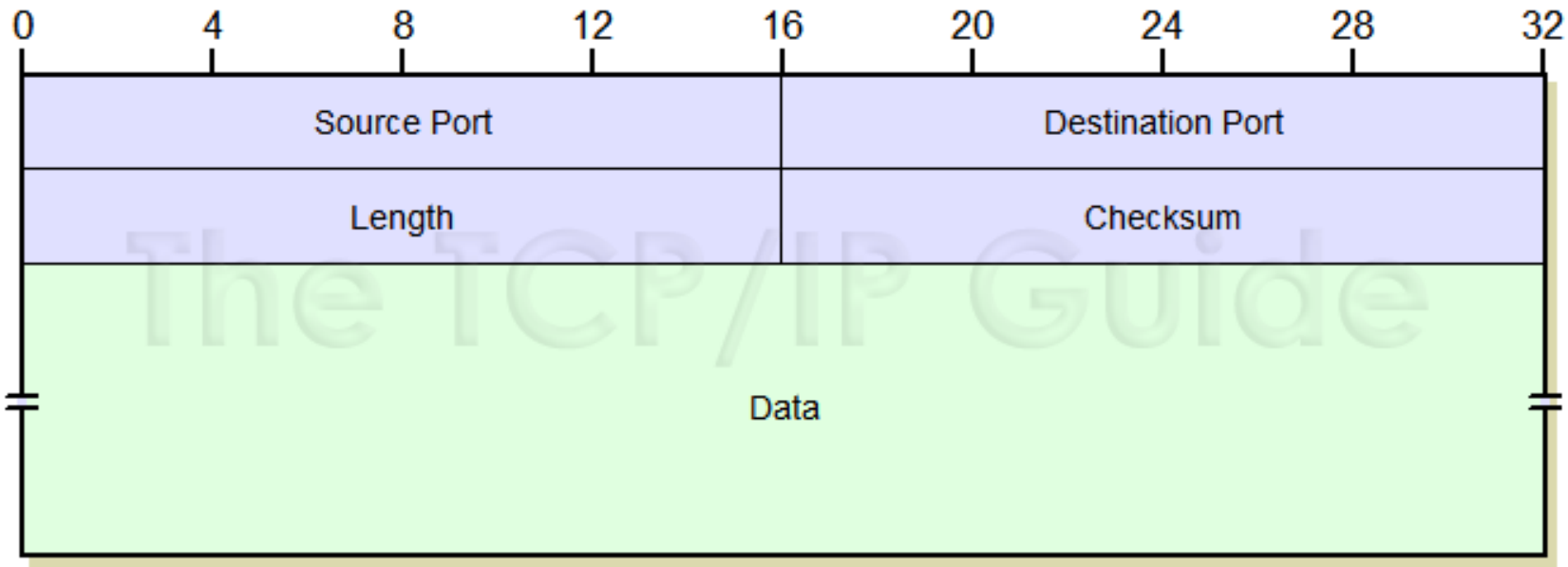
UDP

- Protokollelemente nennt an Datagramme
- Dünne Schicht über IP-Protokoll:
  - Verbindungslos
  - Keine Flusskontrolle

→ Segmente können verloren gehen (unzuverlässig)

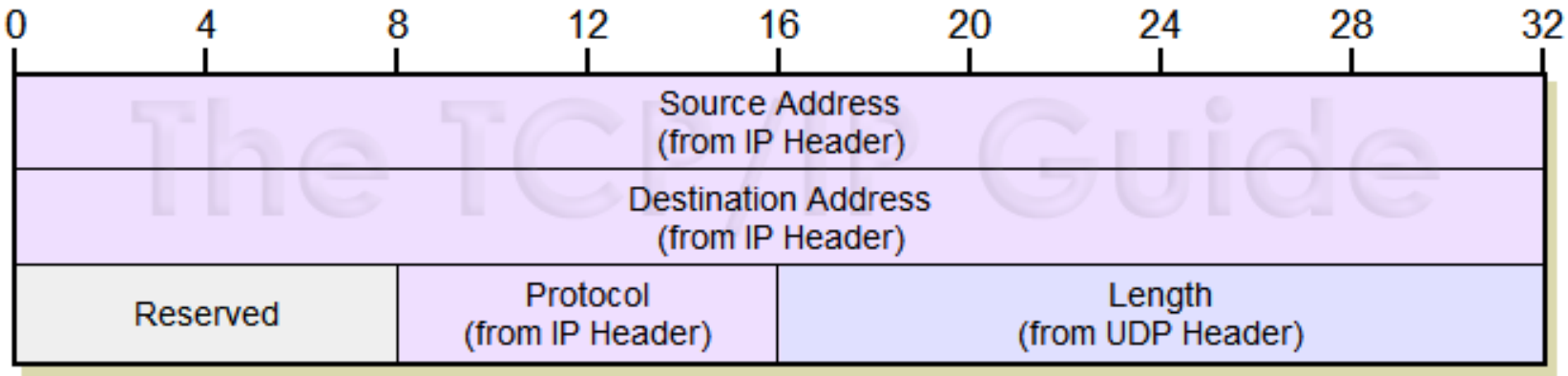
- Geeignet für Echtzeitkommunikation (z. B. Videotelefonie)
- Erhöhung der Zuverlässigkeit zu Fuß möglich (aber wieso nicht gleich TCP?)
- Multi- und Broadcast möglich

UDP-Header


0
4
8
12
16
20
24
28
32
Source Port
Destination Port
Length
Checksum
Data

UDP-Prüfsumme

- Wird über Header, Daten und sog. Pseudo-Header gebildet
- Pseudo-Header hat folgenden Inhalt:



- Berechnungsvorschrift: RFC 1071
- Einzige Möglichkeit Übertragung zu verifizieren

Datagramm empfangen

- Beim Empfang muss der Rechner folgendes machen:
  - IP-Adresse aus IP-Paket lesen
  - Bauen des Pseudoheaders (explizit oder implizit)
  - Prüfsumme berechnen (dabei alte merken und dann 0en)
  - Beide Prüfsummen vergleichen
- Wenn Prüfsummen identisch sind, dann hat Datagramm (wahrscheinlich) richtigen Rechner und richtigen Port erreicht
- Ziel ist es also auch zu ermitteln, ob Datagramm beim richtigen Empfänger angekommen ist.

Outline

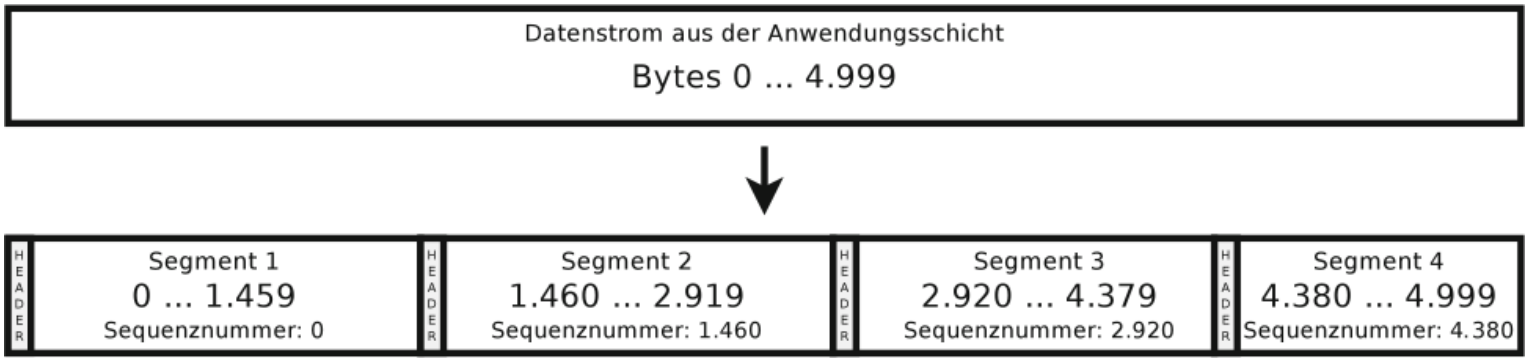
1 UDP

2 TCP

3 Sockets

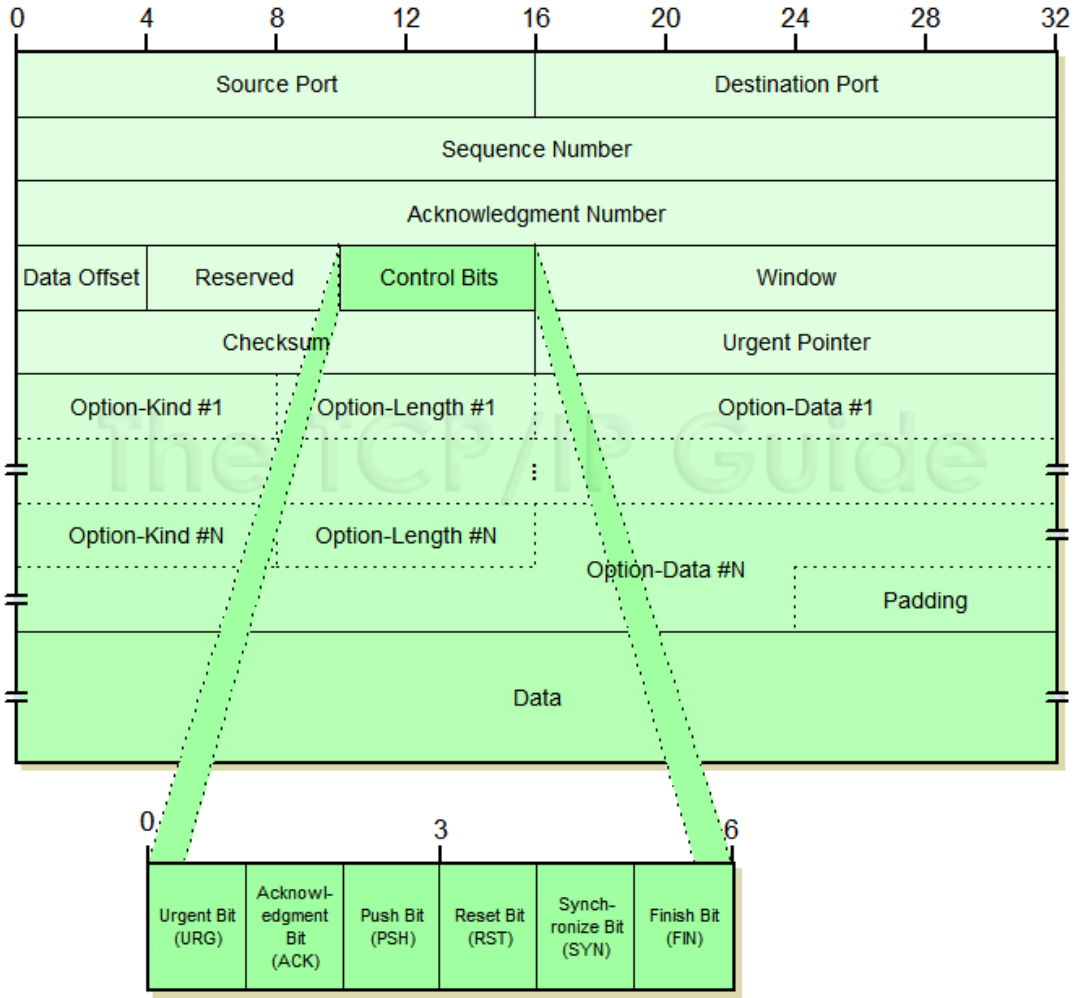
TCP

- Verbindungsorientiert und zuverlässig
- Ähnlich wie Dateien (öffnen, lesen, schreiben, schließen)
- Nutzdaten sind Datenstrom



- Elementares TCP: RFC 793
- Übersicht zu RFCs: RFC 7414

TCP-Header



siehe http://www.tcpipguide.com/free/t_TCPMessageSegmentFormat.htm

Sequenz- und Acknummern: relative (zu SYN) Positionen im Bytestrom

Prüfsumme schließt Pseudo-Header ein.

TCP-Control-Bits

URG Daten sollen sofort vom Empfänger verarbeitet werden. Ähnlich den Interrupts.

ACK Ack-Nummer ist gültig → Empfang wird bestätigt

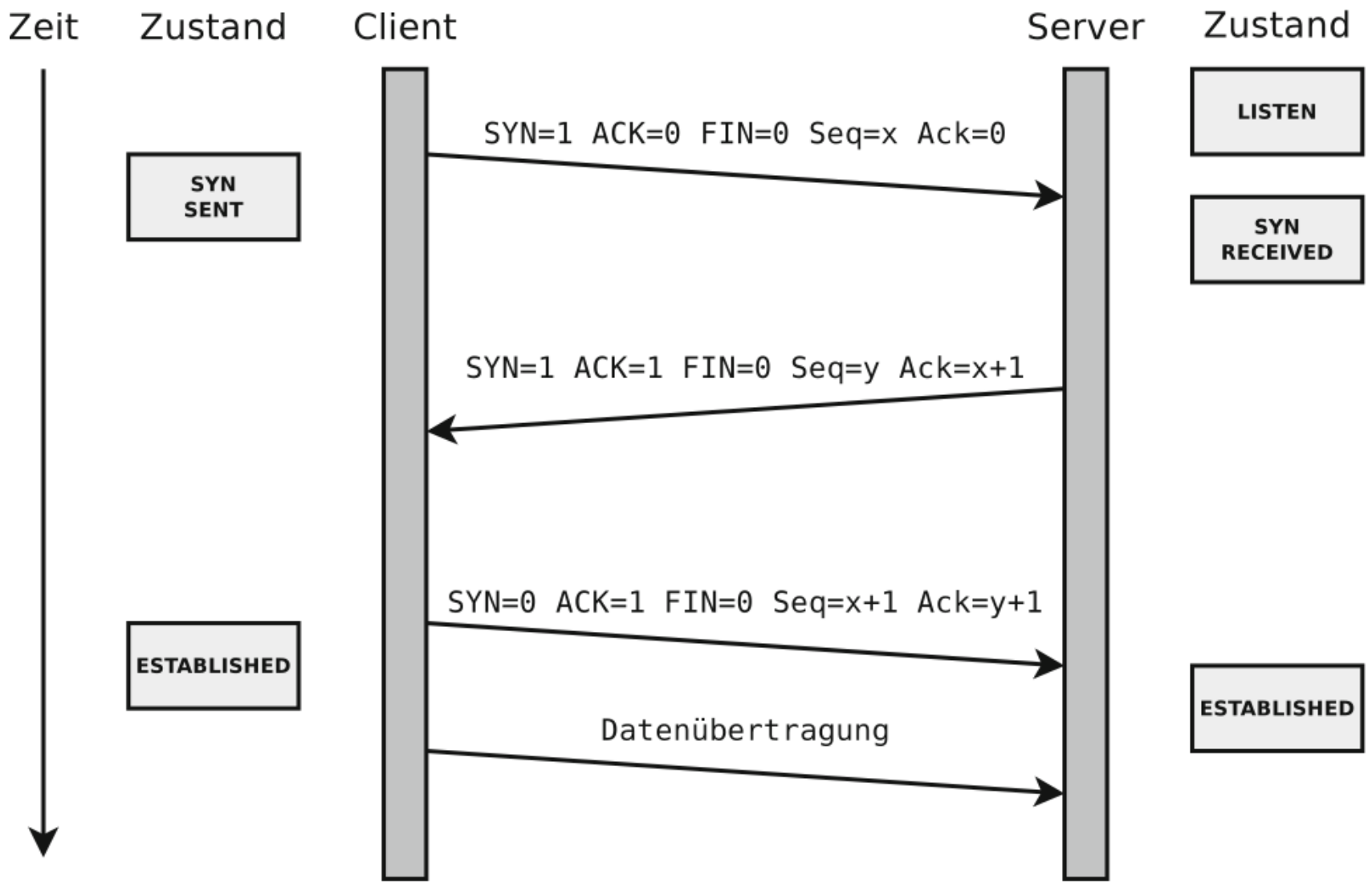
PSH Segment soll sofort zur nächsten Schicht geleitet werden (ohne zu warten, bis Puffer voll)

RST Verbindung soll zurückgesetzt werden

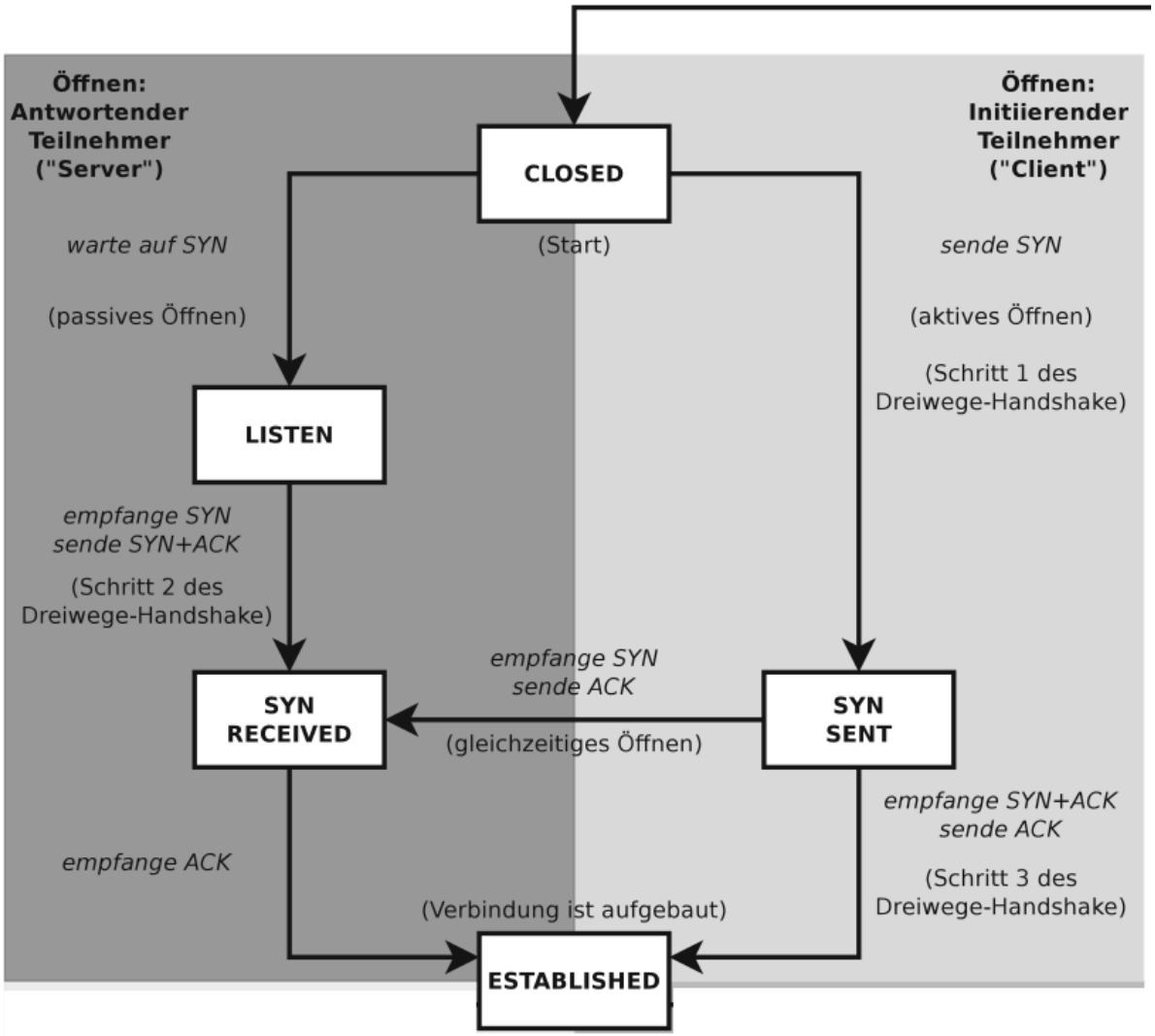
SYN Synchronisierung der Sequenznummern (beim Verbindungsaufbau)

FIN Verbindung soll geschlossen werden → Sender des Segments schickt keine weiteren Daten mehr (Gegenpart kann dies aber noch tun)

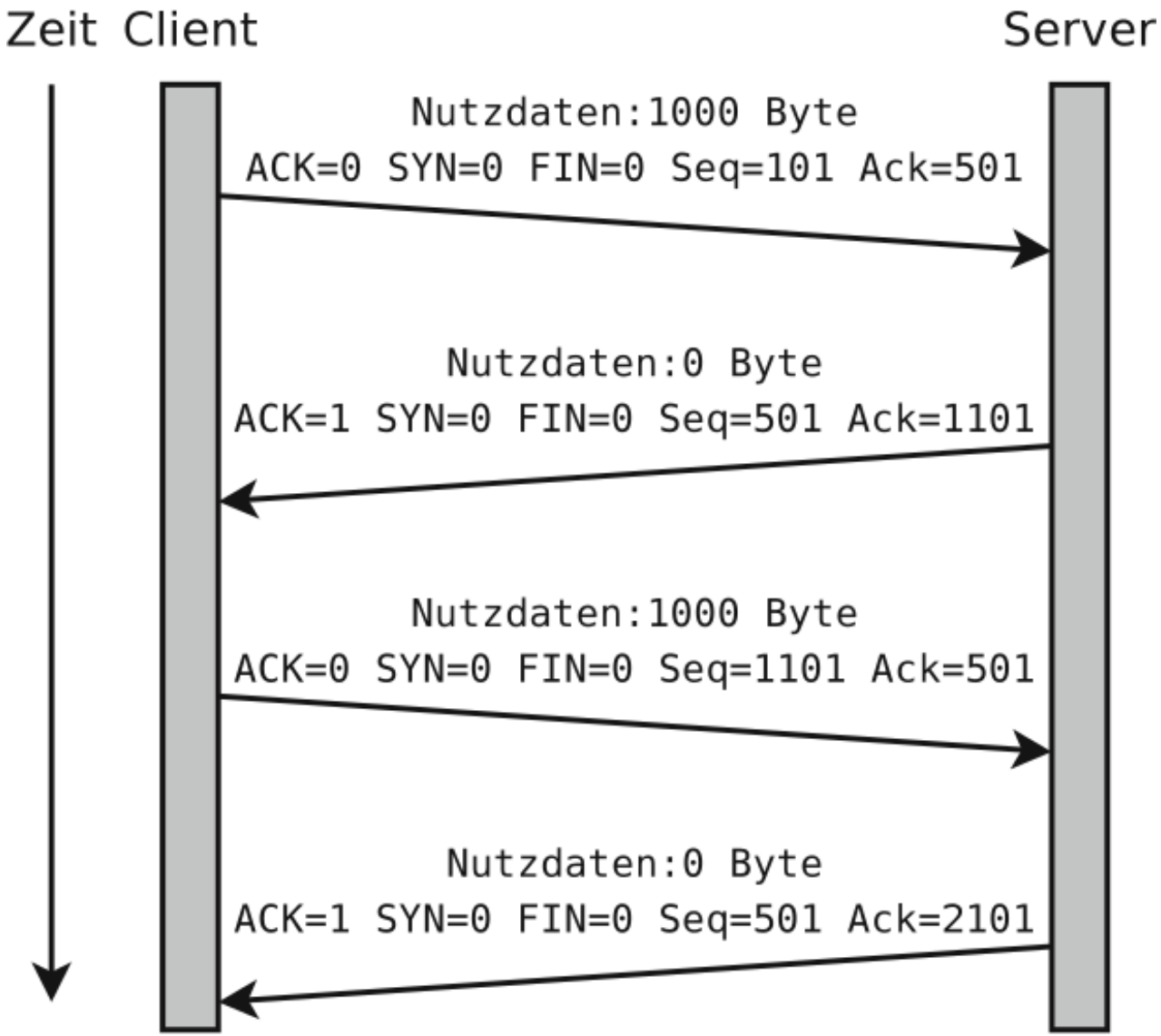
Verbindungsaufbau – Dreiwege-Handshake



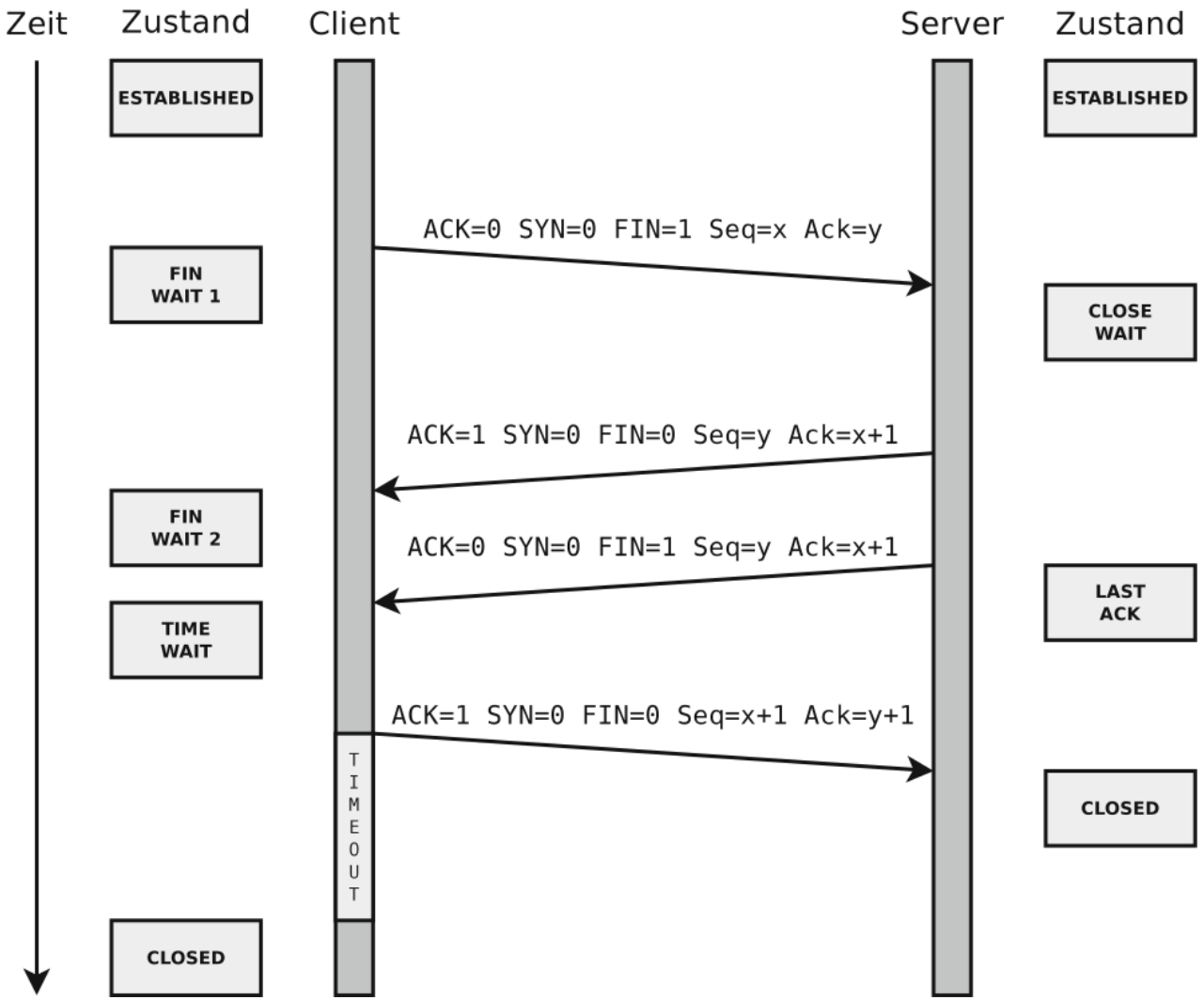
Verbindungsaufbau – Zustandsdiagramm



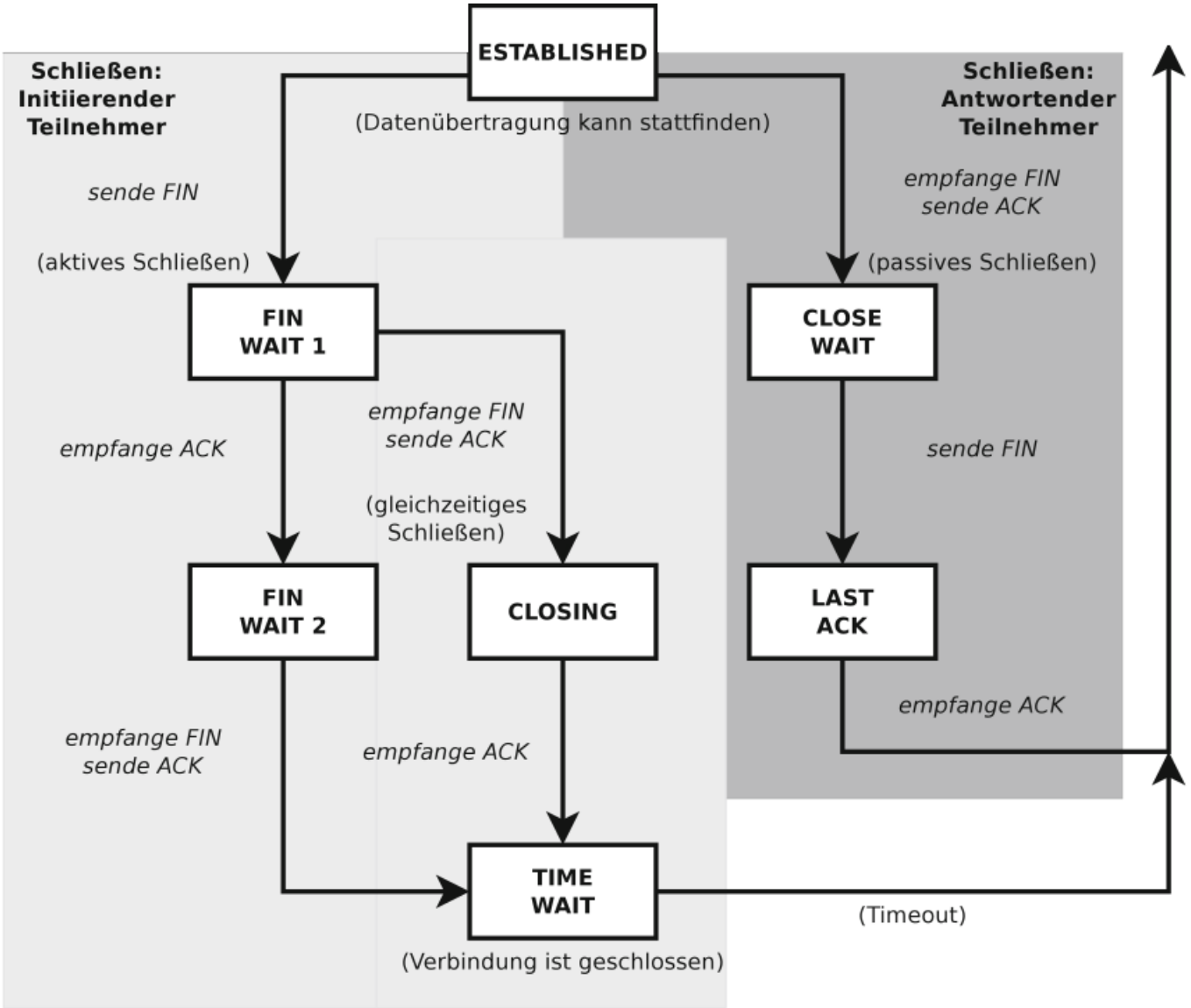
Datenübertragung



Verbindungsabbau



Verbindungsabbau – Zustandsdiagramm



Multi- und Broadcast?

Multi- und Broadcast

- Ist Multi- oder Broadcast möglich?

Multi- und Broadcast?

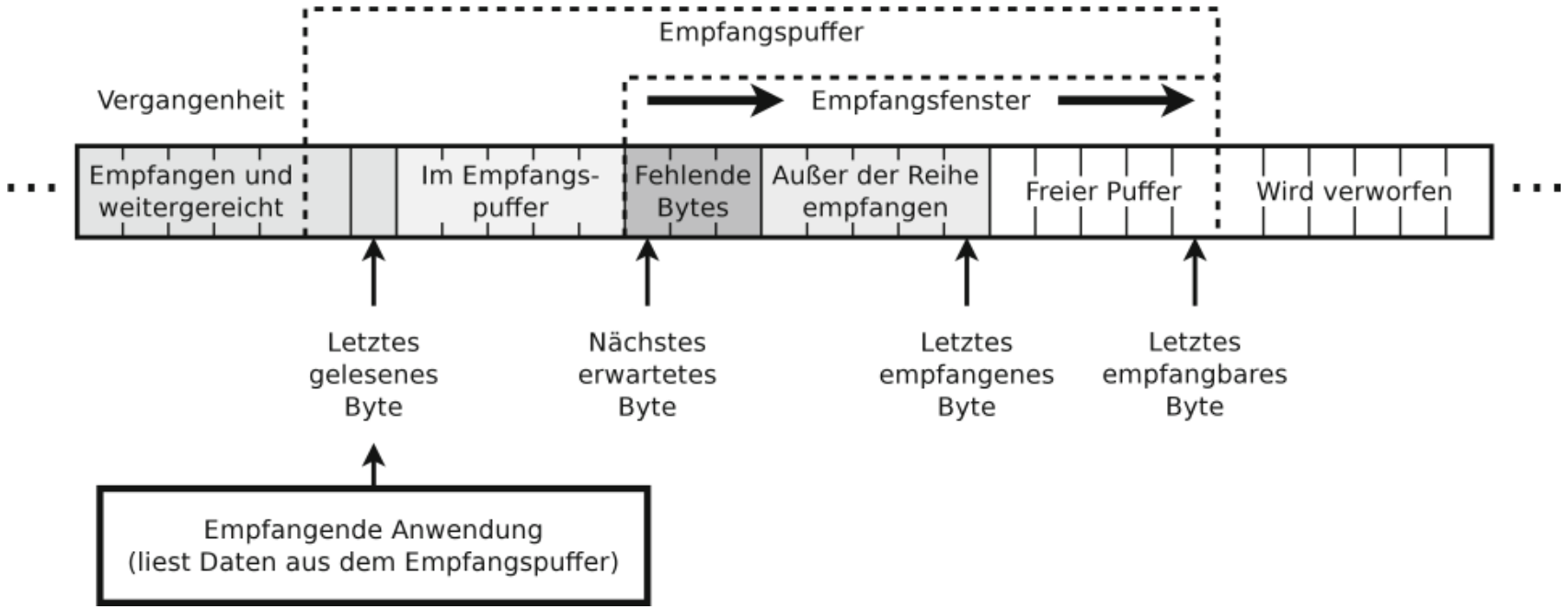
Multi- und Broadcast

- Ist Multi- oder Broadcast möglich?
- Wie lassen sich z. B. Chatrooms mit TCP realisieren?

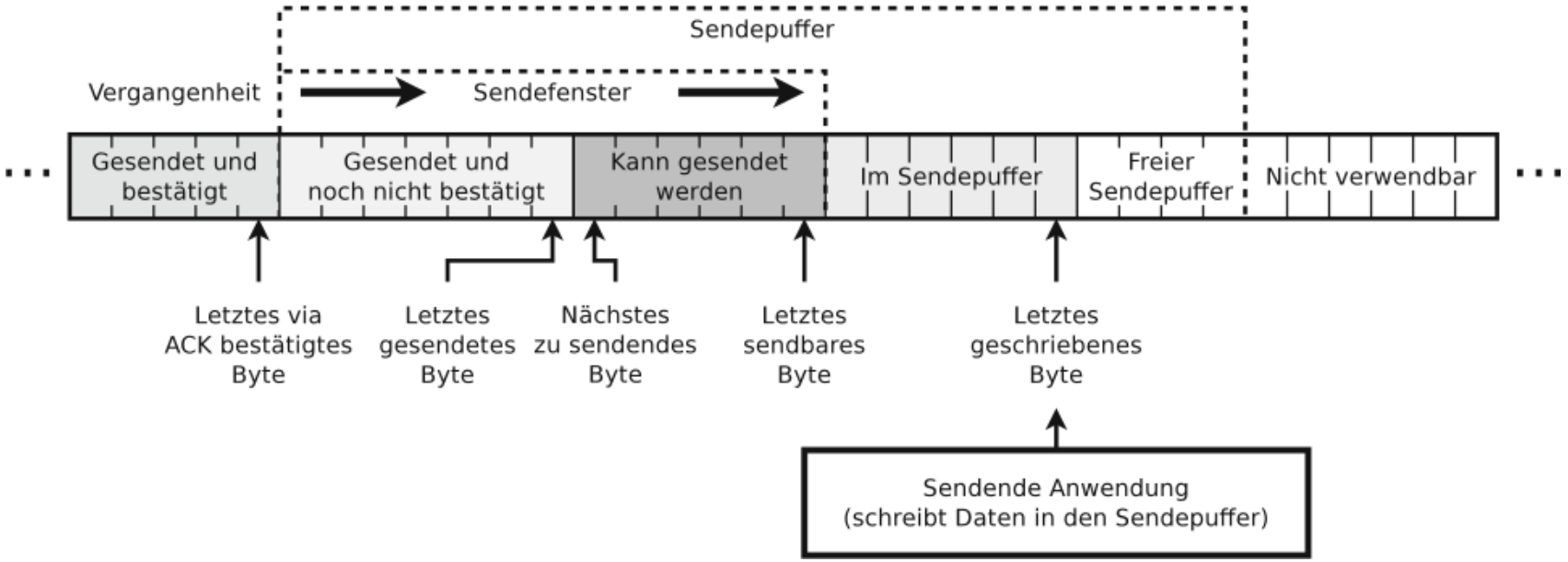
Flusskontrolle

- Schiebefensterverfahren über Sende- und Empfangspuffer
- Kumulative Bestätigungen erlaubt
- Puffert nicht in Reihenfolge ankommende Segmente
- Bei Zeitüberschreitung Neuübertagung aller Segmente im Sendepuffer
- Empfänger teilt Sender Größe des Empfangspuffers mit
- Sender verwaltet Empfangsfenster

Empfangspuffer



Sendepuffer



Überlastkontrolle (engl. *congenstion control*)

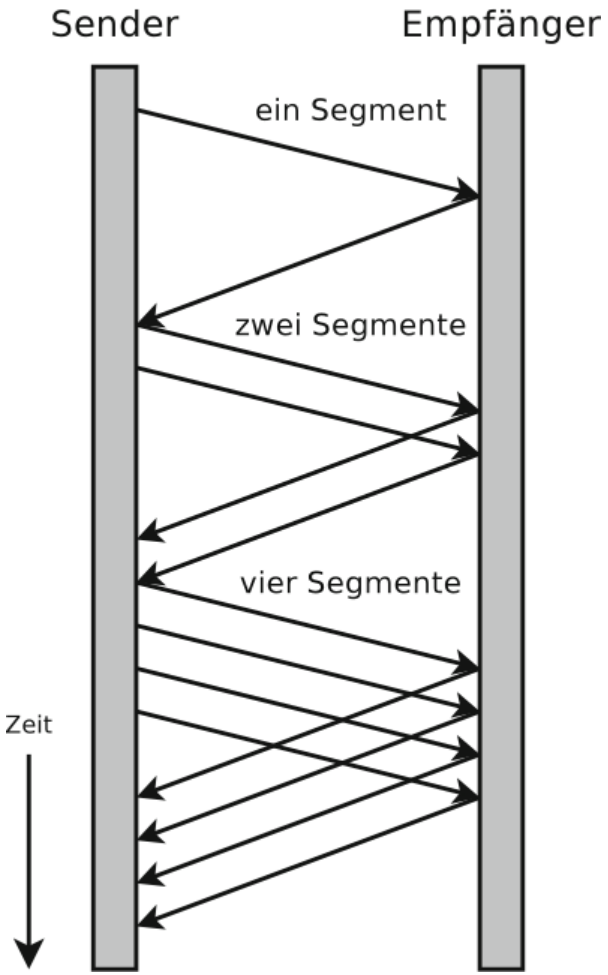
- Flusskontrolle löst Überlastung beim Empfänger
- Was ist mit Überlastung anderer Teilnehmer? (Router)
- Anzeichen für Überlastungen:
  - Paketverluste durch Pufferüberläufe in Routern
  - Lange Wartezeiten durch volle Warteschlangen
  - Häufige Übertragungswiederholungen bzw. Timeouts
- TCP sieht hierfür *Überlastkontrolle* vor
  - Ziel auch hier: Datenrate reduzieren
  - Sender verwaltet weiteres Fenster: *Überlastungsfenster* mit 64 KiB (mehr mit sog. *TCP Window Scale Option*)

Zusammenspiel Überlast- und Flusskontrolle

Aufgabe

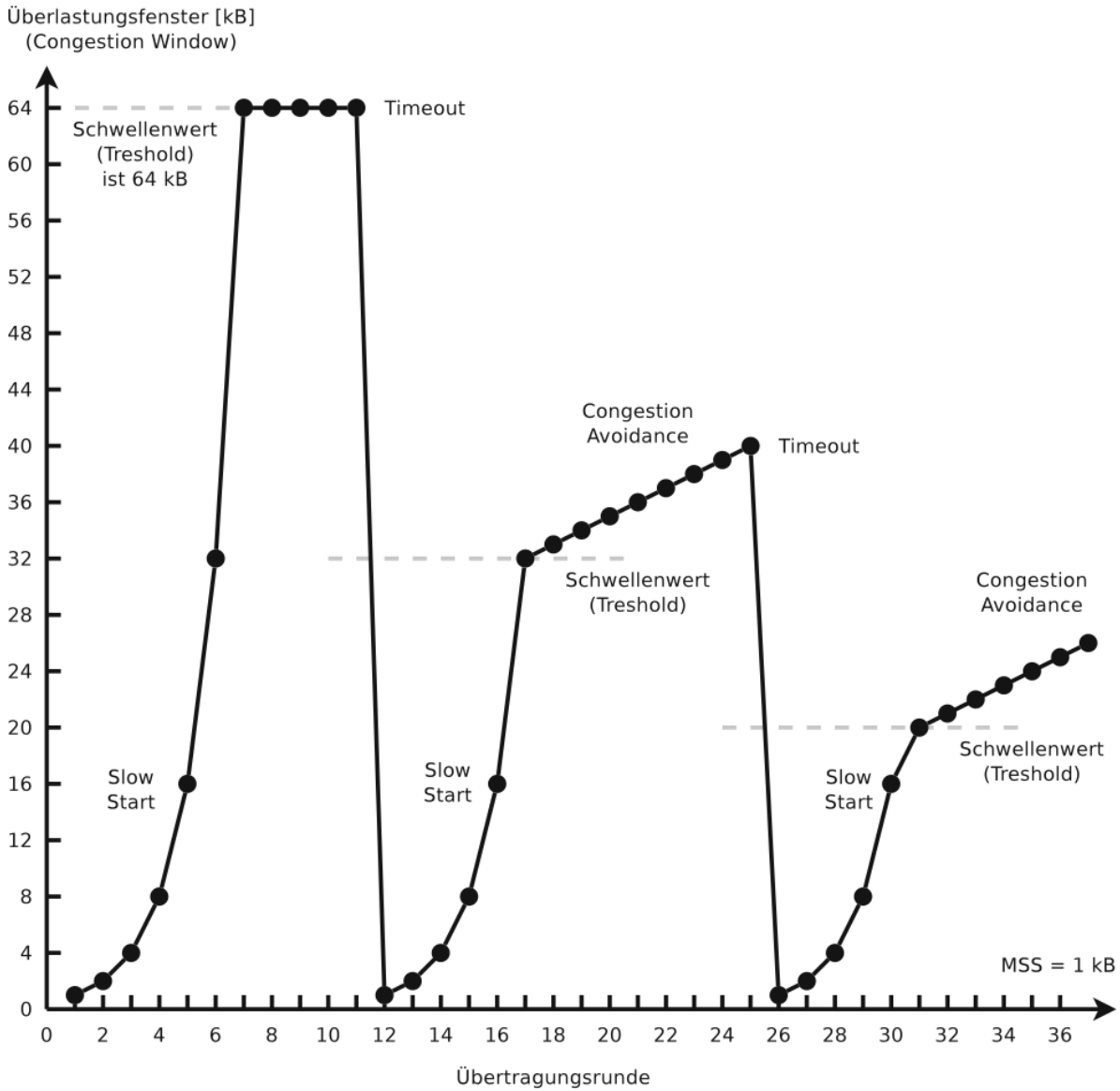
Ein Empfänger teilt mit, dass er nur 20 KiB Daten empfangen kann. Der Sender weiß, dass bei 12 KiB das Netz verstopft. Wie viele Daten sendet er?

Größe des Überlastungsfenster: Slow-Start-Algo



1. Erst ein Segment senden, dann zwei
2. Weiter Verdoppeln bis bestimmter Schwellwert (zu Beginn Größe des Empfangsfenster) erreicht ist
3. Dann lineare Vergrößerung (*congestion avoidance*)
4. Passiert irgendwo ein Timeout:
   - Halbierung des Schwellwerts
   - Überlastungsfenster = 1 Seg.
   - Zurück zu 2.
   - Verfahren Teil von *TCP Tahoe*

Slow-Start-Beispiel



TCP Reno (Fast-Retransmit und -Recovery)

- Timeout kann andere Gründe als Überlast haben (welche?)

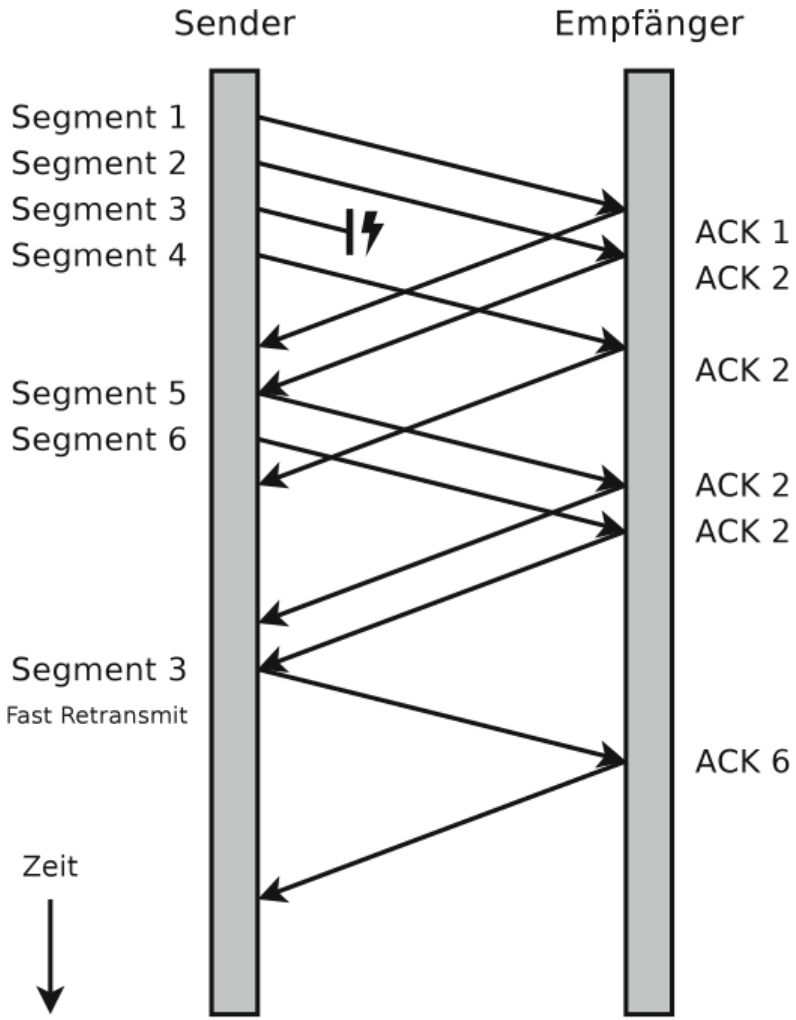
TCP Reno (Fast-Retransmit und -Recovery)

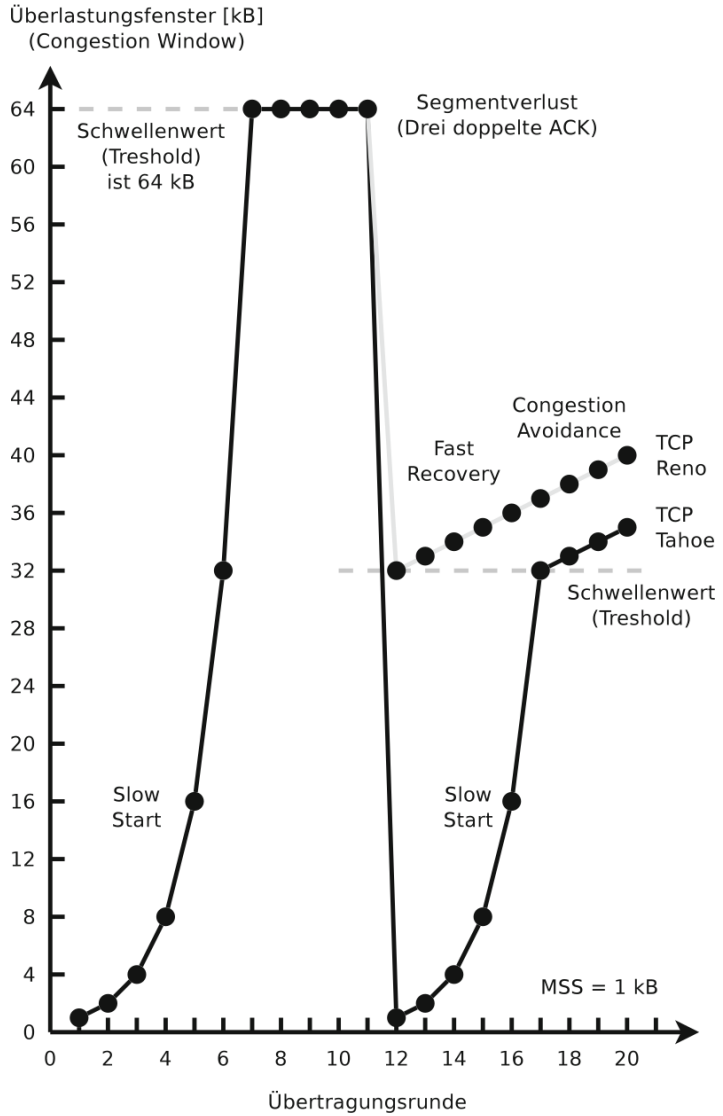
- Timeout kann andere Gründe als Überlast haben (welche?)
- Geht Segment verloren, entsteht beim Empfänger Lücke
- Empfänger sendet bei jedem weiteren empfangenen Segment ACK zum Segment vor dem Verlust
- Netz ist damit ggf. nicht überlastet

→ *TCP Reno*-Algorithmus nach dreimaligen Erhalt eines doppelten ACKs:

- Sendet das verlorene Segment neu (warten ggf. nicht bis auf Timeout)
- Setzt Überlastungsfenster auf Schwellwert

Fast-Retransmit und -Recovery Beispiel





Weitere Verfahren

Es existieren eine Vielzahl modernerer Verfahren (siehe https://en.wikipedia.org/wiki/TCP_congestion_control):

- TCP Vegas
- TCP New Reno
- TCP Hybla
- TCP BIC
- TCP CUBIC
- Agile-SD TCP
- ...

Thema ist Gegenstand aktueller Forschung!

Schwächen von TCP

- Datenübertragung eines Datenstrom blockiert bei Fehler bis zur Lösung

$\Rightarrow$ Blockiert auch weitere Transfers auf demselben Datenstrom

HTTP/2

Bei HTTP/2 wird ein Datenstrom genutzt, um mehrere Dateien zu übertragen. Was bedeutet das für den Abruf einer Webseite, wenn ein Fehler bei der Übertragung auftritt?

Outline

1 UDP

2 TCP

3 Sockets

Sockets

- Standardisierte Schnittstelle zw. Anwendungen und Transportschicht (aber auch tiefer liegende Schichten)
- Programm fordert Socket vom Betriebssystem an
- Betriebssystem verwaltet Sockets und Verbindungsinfos
- Eindeutig identifiziert durch
  - eine Rechneradresse (z. B. IPv4-Adresse),
  - ein Protokoll (z. B. TCP oder UDP) und
  - bei TCP und UDP durch eine Portnummer
- Sockets erweitern UNIX I/O-Funktionalität
  - Dateideskriptoren für Netzwerkverbindungen
  - Erweiterte Lese- und Schreibsystemaufrufe

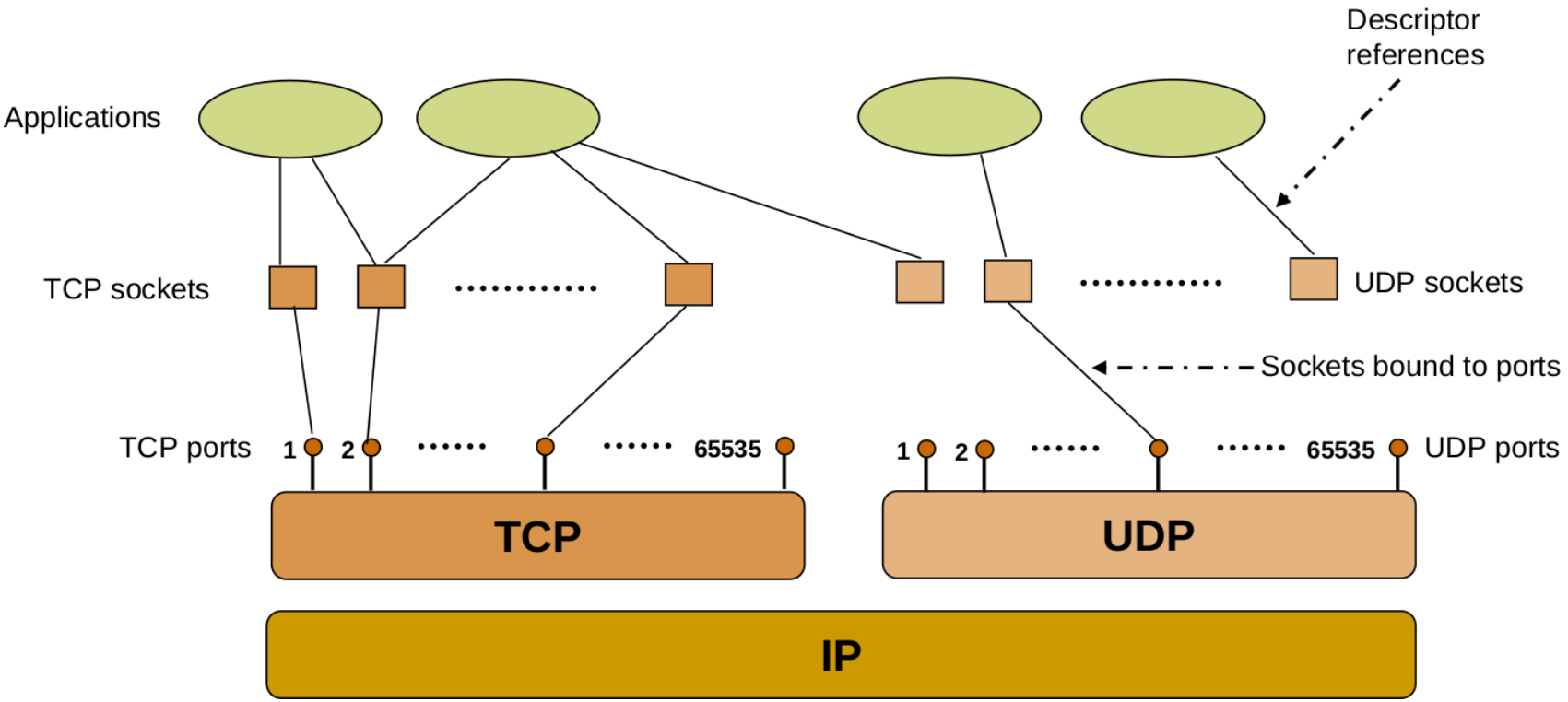
Internet-spezifische Sockets der Transportschicht

UDP und TCP werden durch zwei verschiedene Sockettypen abgebildet:

Streams (TCP): Verlässlicher, verbindungsorientierter Bytestrom-Dienst.
Typname: `SOCK_STREAM`

Datagrams (UDP): Nicht zuverlässiger, verbindungsloser Dienst mit Paketen maximaler Größe (64 KiB).
Typname: `SOCK_DGRAM`

Socketsübersicht



Von https://www.csd.uoc.gr/~hy556/material/tutorials/cs556-3rd-tutorial.pdf

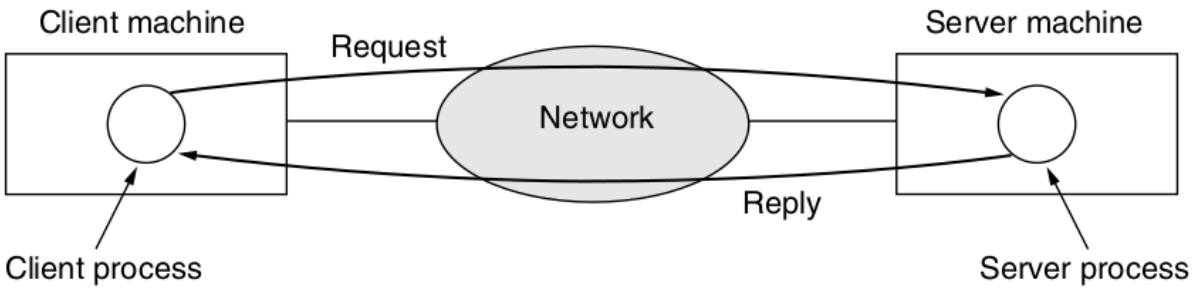
Client-Server-Kommunikation

Server:

- Wartet passiv auf Anfragen von Client
- Antwortet Clienten
- passiver Socket

Client:

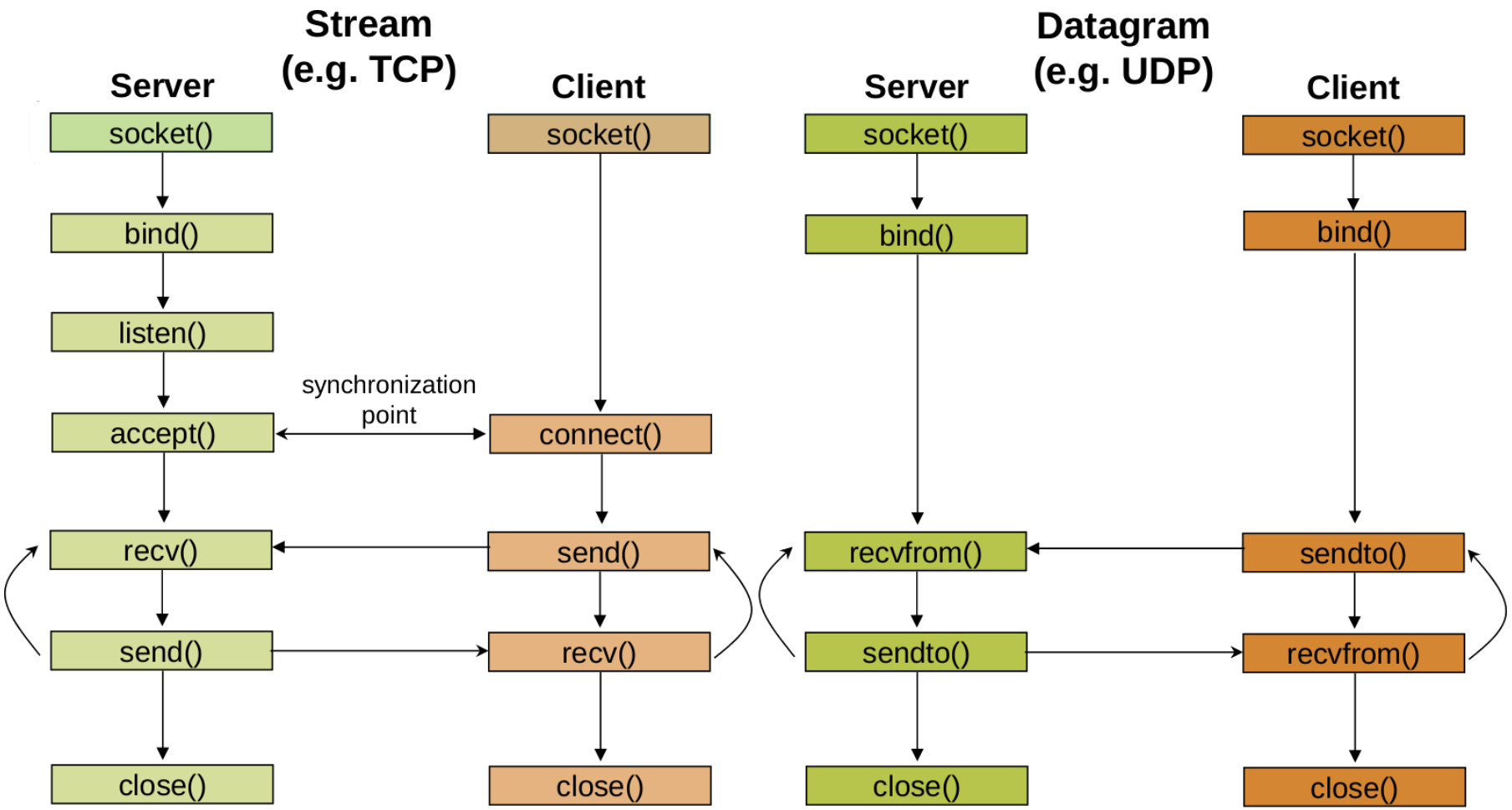
- Initiiert Kommunikation mit Server
- Muss Adresse und Port des Servers kennen
- aktiver Socket



Funktionen für Sockets (BSD)

Primitive	**Bedeutung**
Socket	Neuen Kommunkationsendpunkt erstellen
Bind	Lokale Adresse mit Socket verknüpfen
Listen	Socket auf Wartemodus stellen
Accept	Auf Verbindung warten und akzeptieren
Connect	Verbindung zu einem Endpunkt herstellen
Send	Daten über Verbindung senden
Receive	Daten über Verbindung empfangen
Close	Verbindung schließen

Client-Server mit Sockets



Von https://www.csd.uoc.gr/~hy556/material/tutorials/cs556-3rd-tutorial.pdf

Socket erstellen

```
int sockid = socket(family, type, protocol);
```

- sockid: Deskriptor des erstellten Sockets (wie Dateihandle)
- familiy: Integer mit z. B. folgenden symbolischen Werten
  - `AF_INET`, IPv4-Protokollfamilie
  - `AF_INET6`, IPv6-Protokollfamilie
- type: Integer mit z. B. `SOCK_STREAM` und `SOCK_DGRAM`
- protocol: Integer für das Protokoll (meistens 0, d. h. Vorgabeprotokoll)

Funktion `socket()` erstellt Socket nur, er muss noch verbunden werden. Siehe auch `man socket`.

Adressierung der Sockets bzgl. Transportschicht

- „Generischer“ Datentyp ist

```
struct sockaddr {
  unsigned short sa_family;
  char sa_data[14];
}
```

- Für `AF_INET` ist dieser:

```
struct in_addr {
  unsigned long s_addr; /* IP-Adresse, Network-order */
};
struct sockaddr_in {
  unsigned short sin_family; /* AF_INET */
  unsigned short sin_port;   /* Port */
  struct in_addr sin_addr;   /* IP-Address */
};
```

- Für `AF_INET6` heißt er `struct sockaddr_in6` (siehe `man ipv6`).

Einem Socket eine Adresse zuweisen

```
int status = bind(sockid, &addrport, size);
```

- sockid: Socketdeskriptor
- addrport: `struct sockaddr` bzw. `struct sockaddr_in` spezifiziert lokale Adresse (Netzwerkinterface) und Port auf der Verbindungensanfragen ankommen. Wird `INADDR_ANY` für `sin_addr` benutzt, wird der Socket an alle lokalen Netzwerkinterfaces gebunden.
- size: Größe von addrport in Bytes.
- status: 0 falls ok, bei Fehler -1.

Socket in Listen-Modus schalten

Damit gebundener Socket auf eingehende Verbindungen horcht, muss er in den *listen*-Modus gesetzt werden:

```
int status = listen(sockid, backlog);
```

- sockid: Socketdeskriptor
- backlog: Länge der Warteschlange der noch nicht akzeptierten Verbindungen
- status: 0 falls ok, bei Fehler −1.

Beispiel – Socket mit Port 8080 verknüpfen

```
#include <sys/types.h>
#include <sys/socket.h>

/* ... */

int sockid;
struct sockaddr_in addrport;

socketid = socket(AF_INET, SOCK_STREAM, 0);
addrport.sin_family = AF_INET;
addrport.sin_port = htons(8080);
addrport.sin_addr.s_addr = htonl(INADDR_ANY);

if (bind(sockid, (struct sockaddr *)&addrport,
                        sizeof(addrport)) == -1)
{
   /* Handle error */
}

listen(sockid, 5);
```

Einkommende Verbindung akzeptieren

```
int cid = accept(sockid,&caddrport,&caddrportl);
```

- sockid: Socketdeskriptor
- caddrport: Adresse des Clienten ist nach Rückkehr hier abgelegt
- caddrportl: Größe von caddrport in Bytes. Ist nach Rückkehr ggf. verändert.
- cid: Socketdeskriptor der akzeptierten Verbindung. Kommunikation mit Clienten erfolgt nur über diesen Socket.
- Aufruf ist per Vorgabe *blockierend* (wartet bis eine Verbindungsanfrage kommt). Dieses Verhalten ist mit Hilfe von Socketoptionen anpassbar.

Verbindung aufbauen (Client)

```
int status = connect(sockid, &raddrport,
                     raddrportl);
```

- sockid: Socketdeskriptor
- raddrport: Endpunkt, mit dem sich verbunden werden soll
- raddrportl: Größe von raddrport
- status: 0 falls ok, bei Fehler -1.
- Aufruf ist vorgabemäßig *blockierend*

Möchte man Domainnamen verwenden, kann z. B. `gethostbyname()` genutzt werden, um raddrport zu befüllen.

Daten austauschen

```
int count = send(sockid, buf, len, flags);
```

- buf: Zeiger auf Puffer mit zu übertragenden Daten
- len: Anzahl der zu sendenden Bytes

```
int count = recv(sockid, buf, len, flags);
```

- buf: Zeiger auf Puffer, in dem empfangene Daten abgelegt werden
- len: Anzahl der zu empfangenden Bytes

Gemeinsame Parameter:

- sockid: Deskriptor für verbundenen Socket (`accept()` oder `connect()`)
- flags: Zusätzliche Flags, z. B. `MSG_DONTWAIT` für nicht blockierenden Modus

Socket schließen

Wenn Socket nicht mehr benutzt wird → schließen

```
int status = close(sockid);
```

- sockid: Deskriptor des Sockets, der geschlossen werden soll.
- status: 0 falls ok, bei Fehler -1.

Für TCP-Sockets: Verbindungsabbau wird eingeleitet (FIN-Segment wird gesendet). Assoziierter Port steht anderen Anwendungen wieder zur Verfügung.

Beispiel: Einfachster Web-Server, Teil 1

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <netinet/in.h>
4  #include <err.h>
5
6  char response[] = "HTTP/1.1 200 OK\r\n"
7  "Content-Type: text/html; charset=UTF-8\r\n\r\n"
8  "<!DOCTYPE html><html><head><title>Hello</title><style>"
9  "body { background-color: #111 }"
10 "h1 { font-size:4cm; text-align: center; color: black;text-shadow: 0 0 2mm red}"
11 "</style></head>"
12 "<body><h1>Hello, world!</h1></body></html>\r\n";
```

Beispiel: Einfachster Web-Server, Teil 2

```
14  int main()
15  {
16    int client_fd;
17    struct sockaddr_in svr_addr, cli_addr;
18    socklen_t sin_len = sizeof(cli_addr);
19
20    int sock = socket(AF_INET, SOCK_STREAM, 0);
21    if (sock < 0)
22      err(1, "can't open socket");
23
24    svr_addr.sin_family = AF_INET;
25    svr_addr.sin_addr.s_addr = INADDR_ANY;
26    svr_addr.sin_port = htons(8080);
27    if (bind(sock, (struct sockaddr *)&svr_addr, sizeof(svr_addr)) == -1) {
28      close(sock);
29      err(1, "Can't bind");
30    }
31
32    listen(sock, 5);
33    while (1) {
34      if ((client_fd = accept(sock, (struct sockaddr *)&cli_addr, &sin_len)) == -1) {
35        perror("Can't accept");
36        continue;
37      }
38
39      if (write(client_fd, response, sizeof(response) - 1) != sizeof(response) -1 ) {
40        perror("Can't write");
41      }
42      close(client_fd);
43    }
44  }
```

Beispiel: Einfachster Web-Server, Teil 3

```
$ gcc -O3 -Wall webserver.c -o webserver
$ ./webserver
```