

Grundlagen Rechnernetze und Verteilte Systeme

SoSe 2012

Kapitel 4: Transportschicht

Prof. Dr.-Ing. Georg Carle

Stephan M. Günther, M.Sc.

Nadine Herold, M.Sc.

Dipl.-Inf. Stephan Posselt

Fakultät für Informatik

Lehrstuhl für Netzarchitekturen und Netzdienste

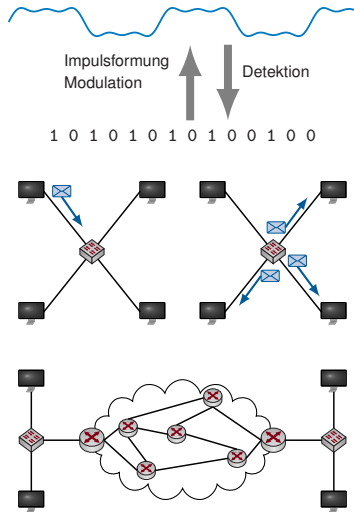
Technische Universität München

Worum geht es in diesem Kapitel?

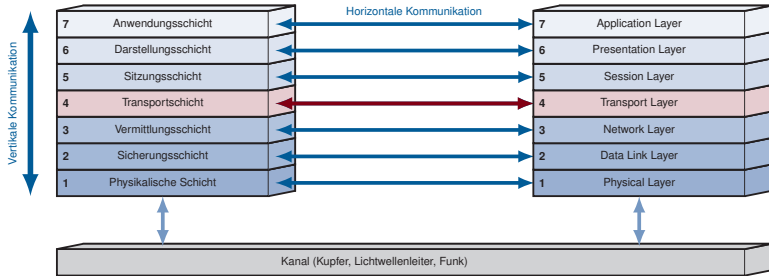
- 1 Motivation
- 2 Multiplexing
- 3 Verbindungslose Übertragung
- 4 Verbindungsorientierte Übertragung
- 5 Network Address Translation (NAT)

Wir haben bislang gesehen:

- ▶ Wie digitale Daten durch messbare Größen dargestellt, übertragen und rekonstruiert werden (Schicht 1)
- ▶ Wie der Zugriff auf das Übertragungsmedium gesteuert und der jeweilige Next-Hop adressiert wird (Schicht 2)
- ▶ Wie auf Basis logischer Adressen End-zu-End Verbindungen zwischen Sender und Empfänger hergestellt werden



Einordnung im ISO/OSI-Modell



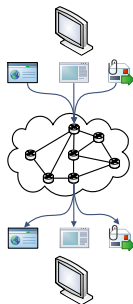
Aufgaben der Transportschicht

Die wesentlichen Aufgaben der Transportschicht sind

- ▶ **Multiplexing** von Datenströmen unterschiedlicher Anwendungen bzw. Anwendungsinstanzen,

Multiplexing:

- ▶ Segmentierung der Datenströme unterschiedlicher Anwendungen (Browser, Chat, Email, ...)
- ▶ Segmente werden in jeweils unabhängigen IP-Paketen zum Empfänger geroutet
- ▶ Empfänger muss die Segmente den einzelnen Datenströmen zuordnen und an die jeweilige Anwendung weiterreichen



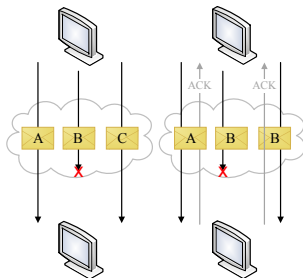
Aufgaben der Transportschicht

Die wesentlichen Aufgaben der Transportschicht sind

- ▶ **Multiplexing** von Datenströmen unterschiedlicher Anwendungen bzw. Anwendungsinstanzen,
- ▶ Bereitstellung **verbindungsloser** und **verbindungsorientierter** Transportmechanismen und

Transportdienste:

- ▶ Verbindungslos (**Best Effort**)
 - ▶ Segmente sind aus Sicht der Transportschicht voneinander unabhängig
 - ▶ Keine Sequenznummern, keine Übertragungswiederholung, keine Garantie der richtigen Reihenfolge
- ▶ Verbindungsorientiert
 - ▶ Übertragungswiederholung bei Fehlern
 - ▶ Garantie der richtigen Reihenfolge einzelner Segmente



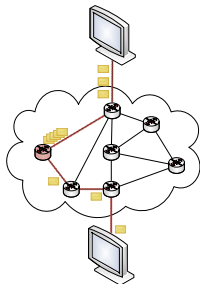
Aufgaben der Transportschicht

Die wesentlichen Aufgaben der Transportschicht sind

- ▶ **Multiplexing** von Datenströmen unterschiedlicher Anwendungen bzw. Anwendungsinstanzen,
- ▶ Bereitstellung **verbindungsloser** und **verbindungsorientierter** Transportmechanismen und
- ▶ Mechanismen zur **Stau-** und **Flusskontrolle**.

Stau- und Flusskontrolle:

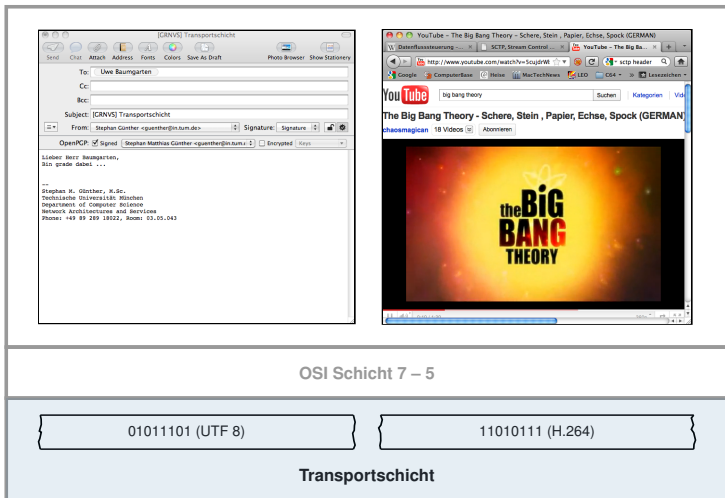
- ▶ **Staukontrolle (Congestion Control)**
 - ▶ Reaktion auf Überlast im Netz
- ▶ **Flusskontrolle (Flow Control)**
 - ▶ Laststeuerung durch den Empfänger



Übersicht

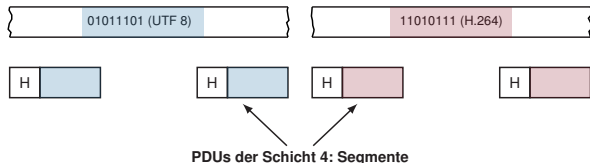
- 1 Motivation
- 2 Multiplexing**
- 3 Verbindungslose Übertragung
- 4 Verbindungsorientierte Übertragung
- 5 Network Address Translation (NAT)

Multiplexing



Auf der Transportschicht

- 1 werden die kodierten Datenströme in **Segmente** unterteilt und
- 2 jedes Segment mit einem Header versehen.

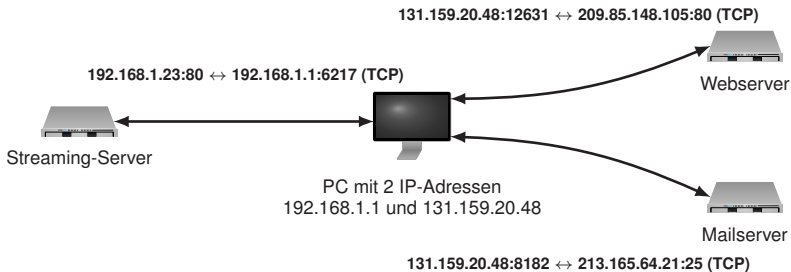


Ein solcher Header enthält jeweils mindestens

- ▶ einen **Quellport** und
- ▶ einen **Zielport**,

welche zusammen mit den IP-Adressen und dem verwendeten Transportprotokoll die Anwendung auf dem jeweiligen Host eindeutig identifizieren.

⇒ **5-Tupel (Protocol, SrcPort, DstPort, SrcIP, DstIP)**

Beispiel:


- ▶ Portnummern sind bei den bekannten Transportprotokollen 16 Bit lang
- ▶ Betriebssysteme verwenden das Quintupel (Portnummern, IP-Adressen, Protokoll), um Anwendungen **Sockets** bereitzustellen
- ▶ Eine Anwendung wiederum adressiert einen Socket mittels eines **File-Deskriptors** (ganzzahliger Wert)
- ▶ Verbindungsorientierte Sockets können nach dem Verbindungsaufbau sehr einfach genutzt werden, da der Empfänger bereits feststeht (Lesen und Schreiben mittels Systemaufrufen `read()` und `write()` möglich)
- ▶ Verbindungslose Sockets benötigen Adressangaben, an wen gesendet oder von wem empfangen werden soll (`sendto()` und `recvfrom()`)

Übersicht

- 1 Motivation
- 2 Multiplexing
- 3 Verbindungslose Übertragung**
- 4 Verbindungsorientierte Übertragung
- 5 Network Address Translation (NAT)

Verbindungslose Übertragung

Funktionsweise:

Header eines Transportprotokolls besteht mind. aus

- ▶ Quell- und Zielpport sowie
- ▶ einer Längenangabe der Nutzdaten.

Dies ermöglicht es einer Anwendung beim Senden für jedes einzelne Paket

- ▶ den Empfänger (IP-Adresse) und
- ▶ die empfangende Anwendung (Protokoll und Zielpport) anzugeben.

Probleme:

Da die Segmente unabhängig voneinander und aus Sicht der Transportschicht **zustandslos** versendet werden, kann nicht sichergestellt werden, dass

- ▶ Segmente den Empfänger erreichen (Pakete können verloren gehen) und
- ▶ der Empfänger die Segmente in der richtigen Reihenfolge erhält (Pakete werden unabhängig geroutet).

Folglich spricht man von einer **ungesicherten, verbindungslosen** oder **nachrichtenorientierten** Kommunikation. (Nicht zu verwechseln mit nachrichtenorientierter Übertragung auf Schicht 2)

Verbindungslose POSIX-Sockets werden mittels des Präprozessormakros `SOCK_DGRAM` identifiziert.

Case Study: User Datagram Protocol (UDP)

Das **User Datagram Protocol (UDP)** ist eines von zwei gebräuchlichen Transportprotokollen im Internet. Es bietet

- ▶ ungesicherte / nachrichtenorientierte Übertragung
- ▶ bei minimalen Overhead.

UDP-Header:



- ▶ „Length“ gibt die Länge von Header und Daten in Vielfachen von Byte an
- ▶ Die Checksumme erstreckt sich über Header und Daten
 - ▶ Die UDP-Checksumme ist optional
 - ▶ Wird sie nicht verwendet, wird das Feld auf 0 gesetzt
 - ▶ Wird sie verwendet, wird zur Berechnung ein **Pseudo-Header** genutzt (eine Art „Default-IP-Header“ der nur zur Berechnung dient aber nicht übertragen wird)

Vorteile von UDP:

- ▶ Geringer Overhead
- ▶ Keine Verzögerung durch Verbindungsaufbau oder Retransmits und Reordering von Segmenten
- ▶ Gut geeignet für Echtzeitanwendungen (Voice over IP, Online-Spiele) sofern gelegentlicher Paketverlust in Kauf genommen werden kann
- ▶ Keine Beeinflussung der Datenrate durch Fluss- und Staukontrollmechanismen (kann Vorteile haben, siehe Übung)

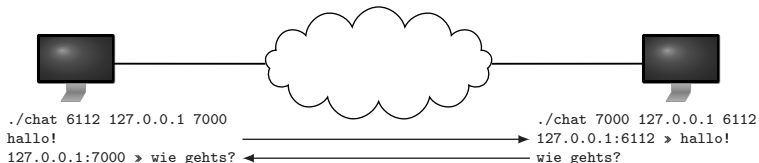
Nachteile von UDP:

- ▶ Es wird keinerlei Dienstqualität zugesichert (beliebig hohe Fehlerrate)
- ▶ Kein Reordering von Segmenten
- ▶ Keine Möglichkeit zur Flusskontrolle (schneller Sender kann langsamen Empfänger überfordern)
- ▶ Keine Staukontrollmechanismen (Überlast im Netz führt zu hohen Verlustraten)

Case Study: UDP-Chat

Was wir wollen:

- ▶ Eine Anwendung, die gleichzeitig als Client und Server arbeiten soll (P2P-Modell)
- ▶ Nur 1:1-Verbindungen, also keine Gruppen-Chats



Was wir brauchen: Einen Socket,

- ▶ auf dem ausgehende Nachrichten gesendet werden (an Ziel-IP und Ziel-Port) und
- ▶ der an die lokale(n) IP(s) und Portnummer gebunden wird, um Nachrichten empfangen zu können

Welche Sprache?

- ▶ C natürlich (;

Wichtige structs

```
struct sockaddr_in {
    __kernel_sa_family_t sin_family; /* Address family */
    __be16                sin_port;  /* Port number */
    struct in_addr        sin_addr;  /* Internet address */

    /* Pad to size of 'struct sockaddr'. */
    unsigned char        __pad[__SOCK_SIZE__ - sizeof(short int) -
                                sizeof(unsigned short int) - sizeof(struct in_addr)];
};

struct in_addr {
    __be32  s_addr;
};
```

Code für unser Programm:

```
struct sockaddr_in local;
local.sin_family    = AF_INET;
local.sin_port     = htons(LOCALPORT); // anwendungsspezifischer Port
local.sin_addr.s_addr = INADDR_ANY;    // empfangen von allen Adressen/allen Adaptern
```

Sockets

- ▶ Aus Sicht des Betriebssystems ist ein Socket nichts weiter als ein [File-Deskriptor](#).
- ▶ Sockets stellen die Schnittstelle zwischen einem Programm (unserer Chatanwendung) und dem Betriebssystem dar.

Ein Socket für unser Programm:

```
int sd;
if ( 0 > (sd=socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP)) ) {
    fprintf(stderr,"socket() failed: %d - %s\n",errno,strerror(errno));
    exit(1);
}
```

Der Socket muss noch eine Adresse bekommen:

```
if ( 0 > bind(sd,(struct sockaddr *)&local,sizeof(local)) ) {
    fprintf(stderr,"bind() failed: %d - %s\n",errno,strerror(errno));
    exit(1);
}
```

Wie merkt unser Programm, wenn neue Daten ankommen?

Hier gibt es 3 Möglichkeiten:

- ▶ Einfach ein `read()` auf den Socket
 - ▶ `read()` blockiert, solange bis etwas kommt
 - ▶ Mit einem einzelnen Prozess bzw. Thread können wir so nur einen einzigen Socket überwachen
 - ▶ Unser Programm würde nicht einmal auf Tastatureingaben reagieren
- ▶ Der scheinbar komplizierte Weg über `select()`
 - ▶ Wir packen alle File-Deskriptoren, die überwacht werden sollen, in ein Set
 - ▶ Wir übergeben `select()` dieses Set
 - ▶ Sobald etwas passiert, gibt uns `select()` ein Set mit genau den File-Deskriptoren zurück, die bereit sind
- ▶ Event-Loop-basiert, z.B. mit "epoll" (wird hier nicht behandelt)

Ein `select()` für unser Programm:

```
fd_set rfds,rfd;
FD_ZERO(&rfds);
FD_SET(STDIN_FILENO,&rfds);
FD_SET(sd,&rfds);
maxfd = MAX(sd,STDIN_FILENO);

while(1) {
    rfd = rfd;
    if ( 0 > select(maxfd+1,&rfd,NULL,NULL,NULL) ) {
        fprintf(stderr,"select() failed: %d - %s\n",errno,strerror(errno));
        exit(1);
    }
    (...)
}
```

Empfangen von Daten

- ▶ Sobald etwas interessantes passiert, wird `select()` uns das sagen.
- ▶ Wir müssen feststellen, welcher der File-Deskriptoren bereit ist.
- ▶ Im Fall der Standardeingabe (STDIN) können wir mit `gets()` (oder etwas besserem) einfach die Eingabe Lesen
- ▶ Wenn der File-Deskriptor des Sockets bereit ist, könnten wir `read()` verwenden, erfahren dann aber nie, wer uns was geschickt hat.
- ▶ Besser wir nutzen `recvfrom()`: Hier können wir ein `struct sockaddr_in` übergeben, in das uns `recvfrom()` reinschreibt, von wem wir etwas empfangen haben.

Ein `recvfrom()` für unser Programm:

```
while(1) {
    (...)
    if ( FD_ISSET(sd,&rfd) ) {
        if ( 0 > (len=recvfrom(sd,buffer,BUFFLEN-1,0,(struct sockaddr *)&from,&slen)) ) {
            fprintf(stderr,"recvfrom() failed: %d - %s\n",errno,strerror(errno));
            exit(1);
        }
        fprintf(stdout,"%s:%d >> %s\n",inet_ntoa(from.sin_addr),ntohs(from.sin_port),buffer);
    }
    (...)
}
```

Senden von Daten

- ▶ Um Daten zu Senden, müssen wir `sendto()` nutzen.
- ▶ Diesem kann man ein `struct sockaddr_in` übergeben, in dem steht, wer der Empfänger sein soll.
- ▶ Ein einfaches `write()` funktioniert nicht, da das Betriebssystem dann nicht weiß, an wen es die Daten senden soll.

Ein `sendto()` für unser Programm:

```
while(1) {
  (...)
  if ( FD_ISSET(STDIN_FILENO,&rfd) ) {
    if( NULL == (s=gets(buffer)) )
      continue;
    if ( 0 > (len=sendto(sd,buffer,MIN(strlen(buffer)+1,BUFSIZE),0,(struct sockaddr *)&remote,sizeof(
      fprintf(stderr,"sendto() failed: %d - %s\n",errno,strerror(errno));
      exit(1);
    }
  }
  (...)
}
```

Übersicht

- 1 Motivation
- 2 Multiplexing
- 3 Verbindungslose Übertragung
- 4 Verbindungsorientierte Übertragung**
- 5 Network Address Translation (NAT)

Verbindungsorientierte Übertragung

Grundlegende Idee: Linear durchnummerierte Segmente mittels **Sequenznummern** im Protokollheader

Sequenznummern ermöglichen insbesondere

- ▶ **Bestätigung** erfolgreich übertragener Segmente,
- ▶ **Identifikation** fehlender Segmente,
- ▶ **erneutes Anfordern** fehlender Segmente und
- ▶ **Zusammensetzen** der Segmente in der **richtigen Reihenfolge**.

Probleme: Sender und Empfänger müssen

- ▶ sich zunächst synchronisieren (Austausch der initialen Sequenznummern) und
- ▶ Zustand halten (aktuelle Sequenznummer, bereits bestätigte Segmente, ...).

Verbindungsorientierte Übertragung

Grundlegende Idee: Linear durchnummerierte Segmente mittels **Sequenznummern** im Protokollheader

Sequenznummern ermöglichen insbesondere

- ▶ **Bestätigung** erfolgreich übertragener Segmente,
- ▶ **Identifikation** fehlender Segmente,
- ▶ **erneutes Anfordern** fehlender Segmente und
- ▶ **Zusammensetzen** der Segmente in der **richtigen Reihenfolge**.

Probleme: Sender und Empfänger müssen

- ▶ sich zunächst synchronisieren (Austausch der initialen Sequenznummern) und
- ▶ Zustand halten (aktuelle Sequenznummer, bereits bestätigte Segmente, ...).

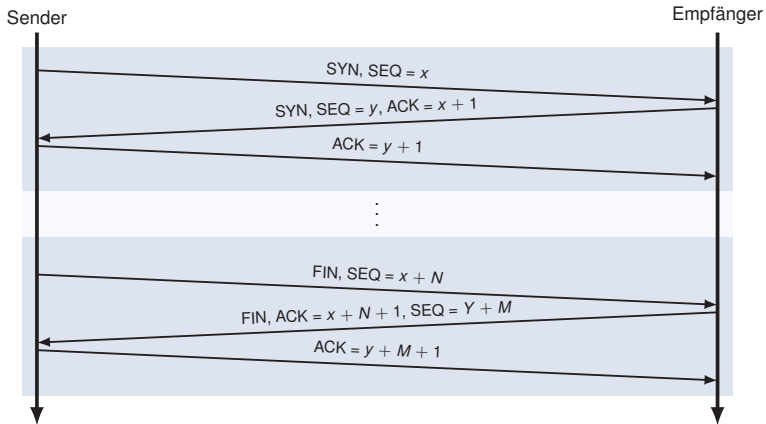
Verbindungsphasen:

- 1 Verbindungsaufbau (Handshake)**
- 2 Datenübertragung**
- 3 Verbindungsabbau (Teardown)**

Vereinbarungen: Wir gehen zunächst davon aus,

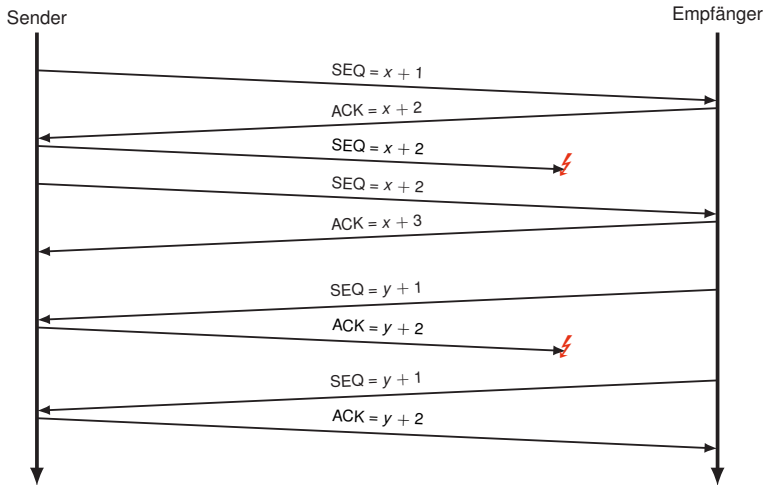
- ▶ dass stets ganze Segmente bestätigt werden und
- ▶ dass stets das nächste erwartete Segment bestätigt wird.

Beispiel: Aufbau und Abbau einer Verbindung



Diese Art des Verbindungsaufbaues bezeichnet man als **3-Way-Handshake**.

Beispiel: Übertragungsphase



Sliding-Window Verfahren

Bislang:

- ▶ Im vorherigen Beispiel hat der Sender stets nur ein Segment gesendet und dann auf eine Bestätigung gewartet
- ▶ Dieses Verfahren ist ineffizient, da abhängig von der **Round Trip Time (RTT)** zwischen Sender und Empfänger viel Bandbreite ungenutzt bleibt („Stop and Wait“-Verfahren)

Idee: Teile dem Sender mit, wie viele Segmente **nach** dem letzten bestätigten Segment auf einmal übertragen werden dürfen, ohne auf eine Bestätigung warten zu müssen.

Vorteile:

- ▶ Zeitlücken zwischen dem Absenden eines Segments und dem Eintreffen einer Bestätigung kann effizienter genutzt werden
- ▶ Durch die Aushandlung dieser **Fenstergrößen** kann der Empfänger die Datenrate steuern → **Flusskontrolle**
- ▶ Durch algorithmische Anpassung der Fenstergröße kann die Datenrate dem Übertragungspfad zwischen Sender und Empfänger angepasst werden → **Staukontrolle**

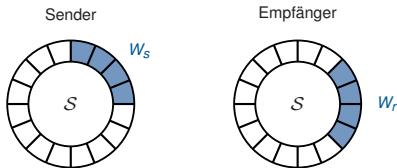
Nachteile:

- ▶ Sender und Empfänger müssen mehr Zustand halten
(Was wurde bereits empfangen? Was wird als nächstes erwartet?)
- ▶ Der Sequenznummernraum ist endlich → Wie werden Missverständnisse verhindert?

Zur Notation:

- ▶ Sender und Empfänger haben denselben Sequenznummernraum $\mathcal{S} = \{0, 1, 2, \dots, N - 1\}$.

Beispiel: $N = 16$:






- ▶ Sendefenster (**Send Window**) $W_s \subset \mathcal{S}$, $|W_s| = w_s$:
Es dürfen w_s Segmente nach dem letzten bestätigtem Segment auf einmal gesendet werden.
- ▶ Empfangsfenster (**Receive Window**) $W_r \subset \mathcal{S}$, $|W_r| = w_r$:
Sequenznummern der Segmente, die als nächstes akzeptiert werden.
- ▶ Sende- und Empfangsfenster „verschieben“ und überlappen sich während des Datenaustauschs.

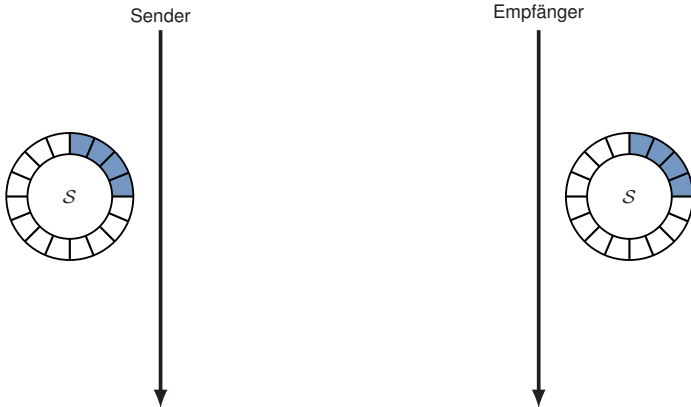
Vereinbarungen:




- ▶ Eine Bestätigung $ACK = m + 1$ bestätigt alle Segmente mit $SEQ \leq m$. Dies wird als **kumulative Bestätigung** bezeichnet.
- ▶ Gewöhnlich löst **jedes erfolgreich empfangene** Segment das Senden einer Bestätigung aus, wobei stets das **nächste erwartete** Segment bestätigt wird. Dies wird als **Forward Acknowledgement** bezeichnet.

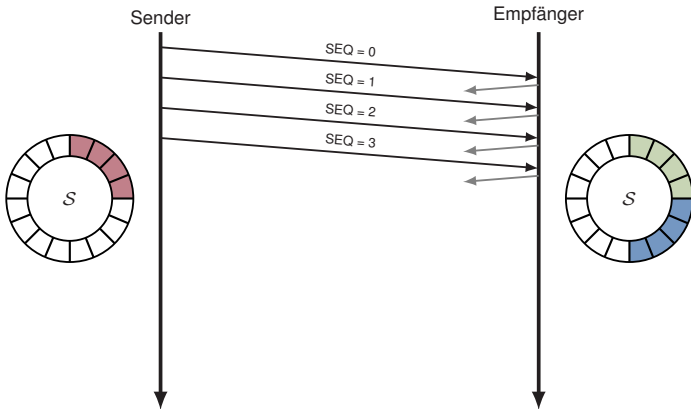
Wichtig:




- ▶ In den folgenden Grafiken sind die meisten Bestätigungen zwecks Übersichtlichkeit nur angedeutet (graue Pfeile).
- ▶ Die Auswirkungen auf Sende- und Empfangsfenster beziehen sich nur auf den Erhalt der schwarz eingezeichneten Bestätigungen.
- ▶ Dies ist äquivalent zur Annahme, dass die angedeuteten Bestätigungen verloren gehen.

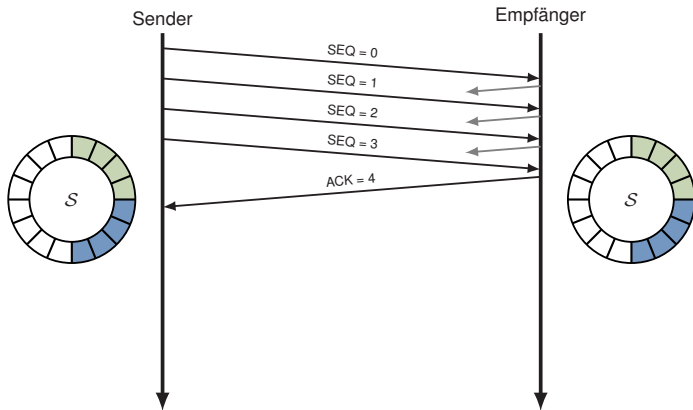
-  Sendefenster W_s bzw. Empfangsfenster W_r
-  gesendet aber noch nicht bestätigt
-  gesendet und bestätigt / empfangen



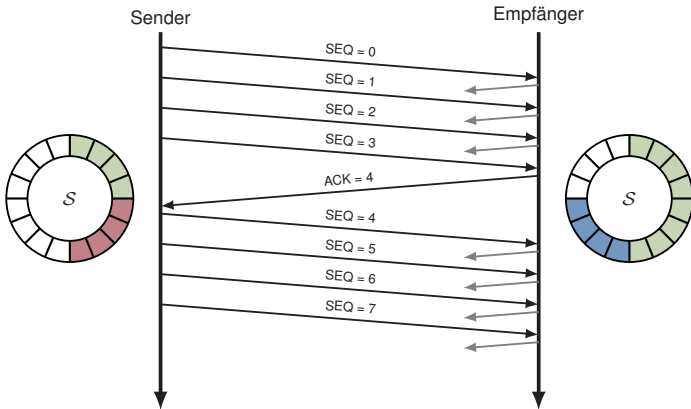
-  Sendefenster W_s bzw. Empfangsfenster W_r
-  gesendet aber noch nicht bestätigt
-  gesendet und bestätigt / empfangen






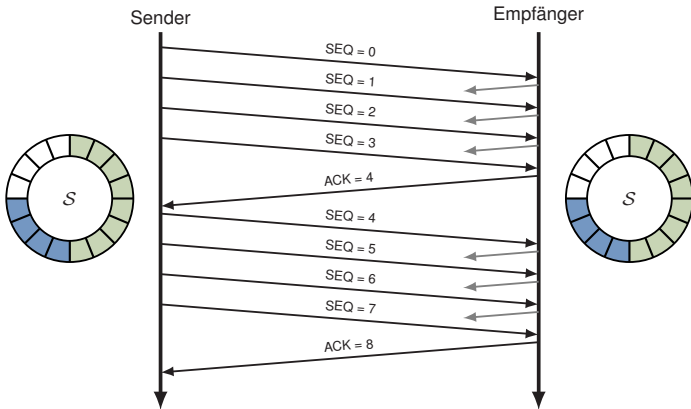
-  Sendefenster W_S bzw. Empfangsfenster W_r
-  gesendet aber noch nicht bestätigt
-  gesendet und bestätigt / empfangen



- Sendefenster W_S bzw. Empfangsfenster W_r
- gesendet aber noch nicht bestätigt
- gesendet und bestätigt / empfangen



-  Sendefenster W_S bzw. Empfangsfenster W_r
-  gesendet aber noch nicht bestätigt
-  gesendet und bestätigt / empfangen



Neues Problem: Wie wird jetzt mit Segmentverlusten umgegangen?

Zwei Möglichkeiten:

1 Go-Back-N

- ▶ Akzeptiere stets nur die nächste erwartete Sequenznummer
- ▶ Alle anderen Segmente werden verworfen

2 Selective-Repeat

- ▶ Akzeptiere alle Sequenznummern, die in das aktuelle Empfangsfenster fallen
- ▶ Diese müssen gepuffert werden, bis fehlende Segmente erneut übertragen wurden

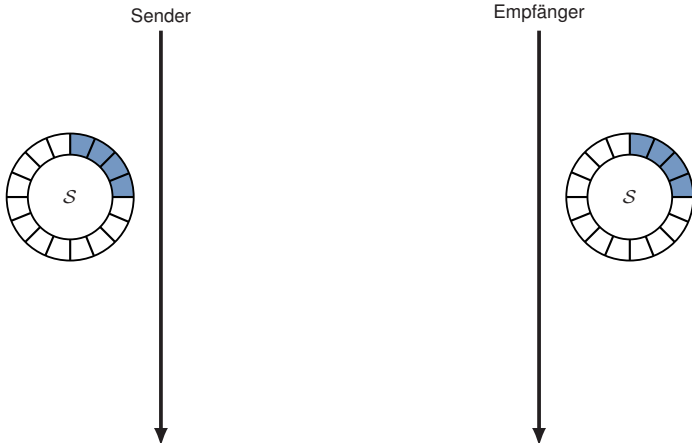
Wichtig:

- ▶ In beiden Fällen muss der Sequenznummernraum so gewählt werden, dass wiederholte Segmente eindeutig von neuen Segmenten unterschieden werden können.
- ▶ Andernfalls würde es zu Verwechslungen kommen
→ Auslieferung von Duplikaten an höhere Schichten, keine korrekte Reihenfolge

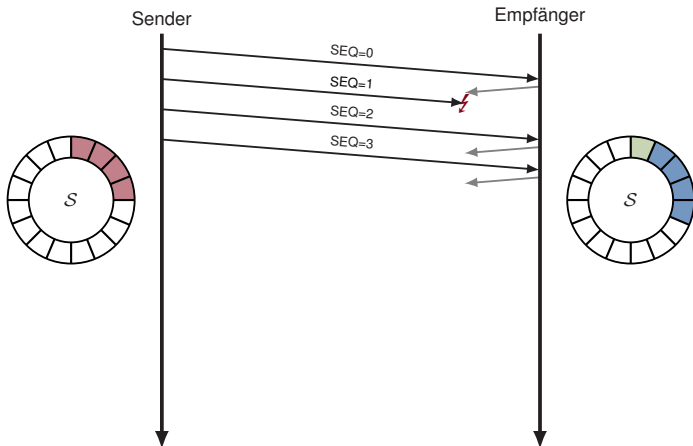
Frage: (s. Übung)

Wie groß darf das Sendefenster W_s in Abhängigkeit des Sequenznummernraums S höchstens gewählt werden, so dass die Verfahren funktionieren?

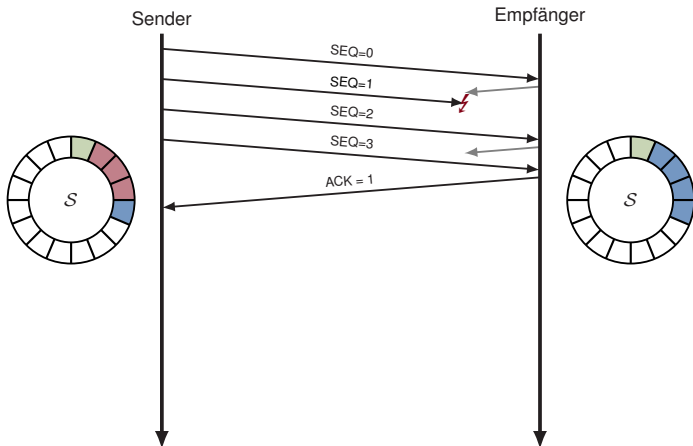
Go-Back-N: $N = 16$, $w_s = 4$, $w_r = 4$



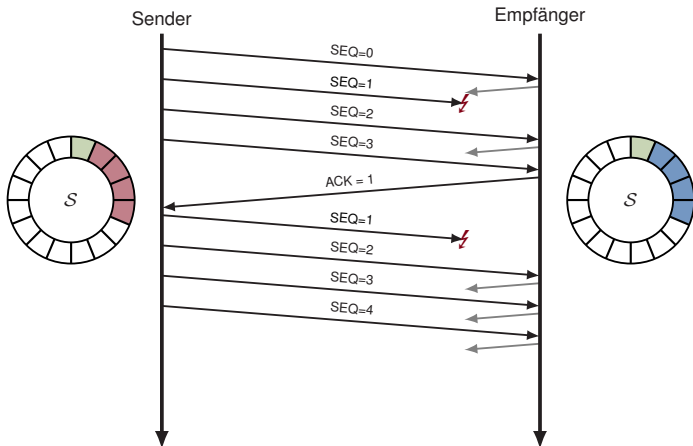
Go-Back-N: $N = 16$, $w_s = 4$, $w_r = 4$



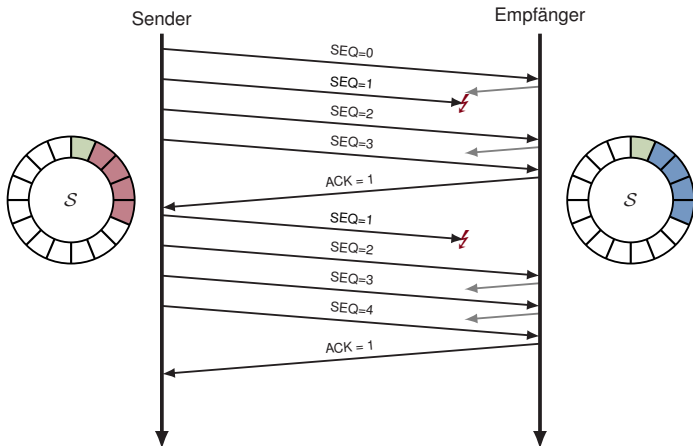
Go-Back-N: $N = 16$, $w_s = 4$, $w_r = 4$



Go-Back-N: $N = 16$, $w_s = 4$, $w_r = 4$



Go-Back-N: $N = 16$, $w_s = 4$, $w_r = 4$



Anmerkungen zu Go-Back-N

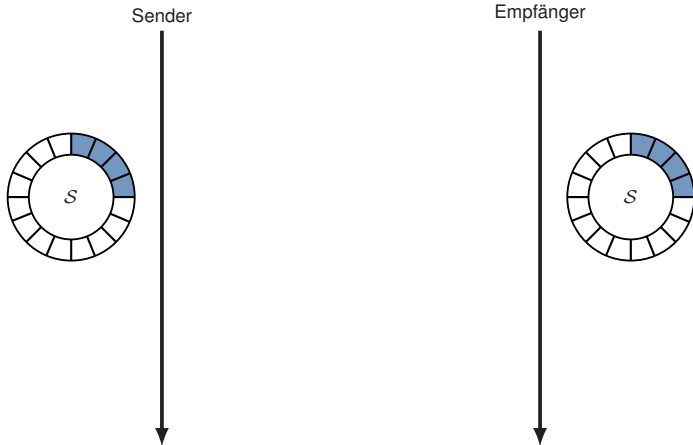
- ▶ Da der Empfänger stets nur das nächste erwartete Segment akzeptiert, reicht ein Empfangsfenster der Größe $w_r = 1$ prinzipiell aus. Unabhängig davon muss für praktische Implementierungen ein ausreichend großer Empfangspuffer verfügbar sein.
- ▶ Bei einem Sequenznummernraum der Kardinalität N muss für das Sendefenster stets gelten:

$$w_s \leq N - 1.$$

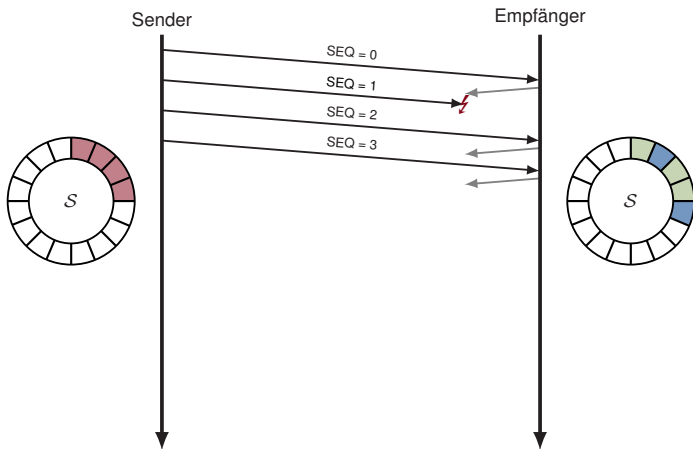
Andernfalls kann es zu Verwechslungen kommen (s. Übung).

- ▶ Das Verwerfen erfolgreich übertragener aber nicht in der erwarteten Reihenfolge eintreffender Segmente macht das Verfahren einfach zu implementieren aber weniger effizient.

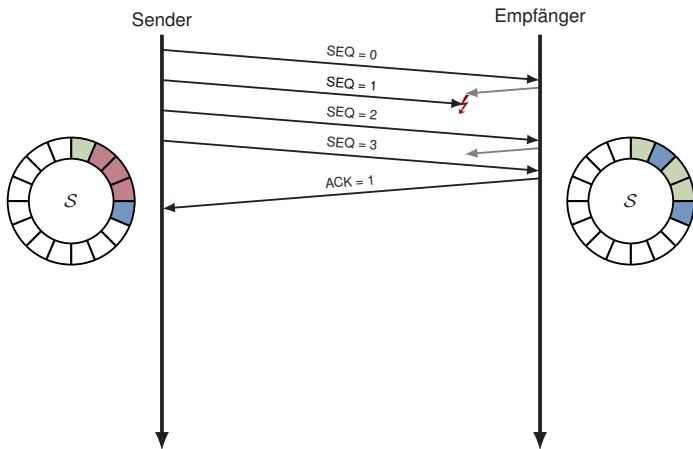
Selective Repeat: $N = 16$, $w_s = 4$, $w_r = 4$



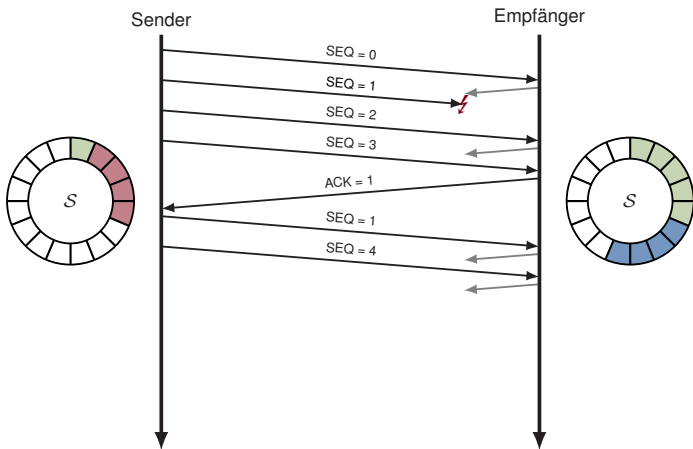
Selective Repeat: $N = 16$, $w_s = 4$, $w_r = 4$



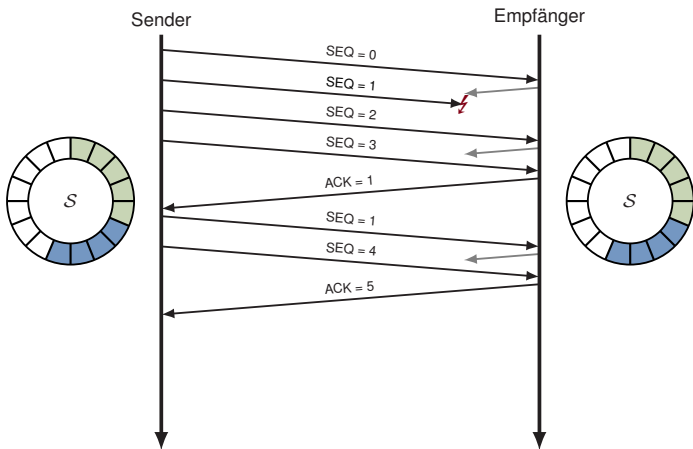
Selective Repeat: $N = 16$, $w_s = 4$, $w_r = 4$



Selective Repeat: $N = 16$, $w_s = 4$, $w_r = 4$



Selective Repeat: $N = 16$, $w_s = 4$, $w_r = 4$



Anmerkungen zu Selective Repeat

- ▶ Wählt man $w_r = 1$ und w_s unabhängig von w_r , so degeneriert Selective Repeat zu Go-Back-N.
- ▶ Bei einem Sequenznummernraum der Kardinalität N muss für das Sendefenster stets gelten:

$$w_s \leq \left\lfloor \frac{N}{2} \right\rfloor.$$

Andernfalls kann es zu Verwechslungen kommen (s. Übung).

Allgemeine Anmerkungen

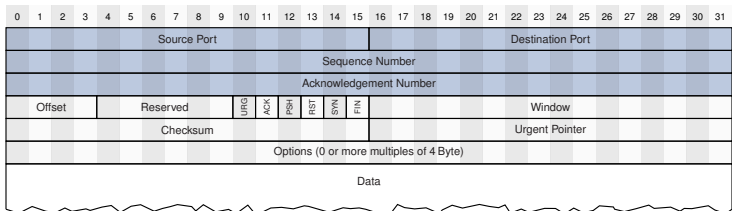
- ▶ Bei einer Umsetzung dieser Konzepte benötigt insbesondere der Empfänger einen **Empfangspuffer**, dessen Größe an die Sende- und Empfangsfenster angepasst ist.
- ▶ Für praktische Anwendungen werden die Größen von W_s und W_r dynamisch angepasst (siehe Case Study zu TCP), wodurch Algorithmen zur **Staukontrolle** und **Flusskontrolle** auf Schicht 4 ermöglicht werden.

Case Study: Transmission Control Protocol (TCP)

Das **Transmission Control Protocol (TCP)** ist das dominante Transportprotokoll im Internet. Es bietet

- ▶ gesicherte / stromorientierte Übertragung sowie
- ▶ Mechanismen für Fluss- und Staukontrolle.

TCP-Header:



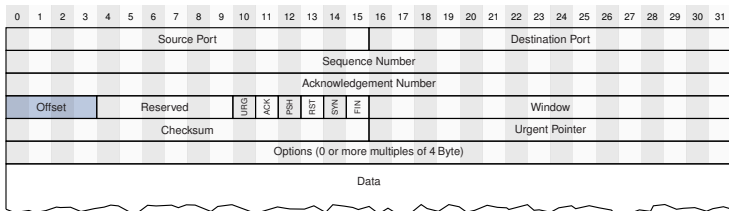
- ▶ **Quell-** und **Zielport** werden analog zu UDP verwendet.
- ▶ **Sequenz-** und **Bestätigungsnummer** dienen der gesicherten Übertragung. Es werden bei TCP **nicht** ganze Segmente sondern einzelne Byte bestätigt (stromorientierte Übertragung).

Case Study: Transmission Control Protocol (TCP)

Das **Transmission Control Protocol (TCP)** ist das dominante Transportprotokoll im Internet. Es bietet

- ▶ gesicherte / stromorientierte Übertragung sowie
- ▶ Mechanismen für Fluss- und Staukontrolle.

TCP-Header:



(Data) Offset

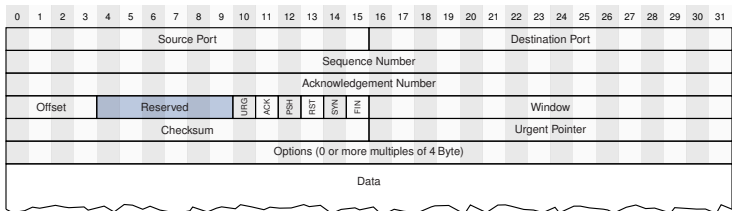
- ▶ Gibt die Länge des TCP-Headers in Vielfachen von 4 Byte an.
- ▶ Der TCP-Header hat variable Länge (Optionen, vgl. IP-Header).

Case Study: Transmission Control Protocol (TCP)

Das **Transmission Control Protocol (TCP)** ist das dominante Transportprotokoll im Internet. Es bietet

- ▶ gesicherte / stromorientierte Übertragung sowie
- ▶ Mechanismen für Fluss- und Staukontrolle.

TCP-Header:



Reserved

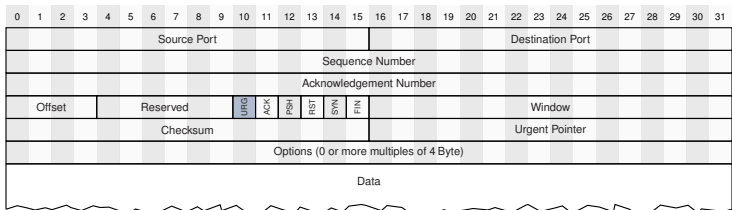
- ▶ Keine Verwendung, wird gewöhnlich auf Null gesetzt.

Case Study: Transmission Control Protocol (TCP)

Das **Transmission Control Protocol (TCP)** ist das dominante Transportprotokoll im Internet. Es bietet

- ▶ gesicherte / stromorientierte Übertragung sowie
- ▶ Mechanismen für Fluss- und Staukontrolle.

TCP-Header:



Flag URG („urgent“) (selten verwendet)

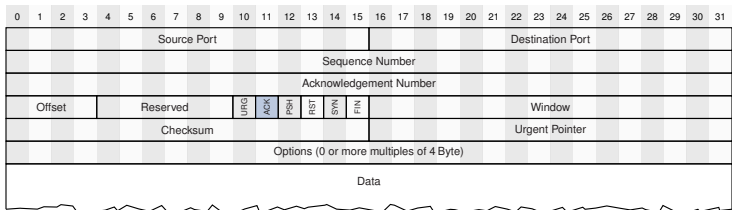
- ▶ Ist das Flag gesetzt, werden die Daten im aktuellen TCP-Segment beginnend mit dem ersten Byte bis zu der Stelle, an die das Feld **Urgent Pointer** zeigt, sofort an höhere Schichten weitergeleitet.

Case Study: Transmission Control Protocol (TCP)

Das **Transmission Control Protocol (TCP)** ist das dominante Transportprotokoll im Internet. Es bietet

- ▶ gesicherte / stromorientierte Übertragung sowie
- ▶ Mechanismen für Fluss- und Staukontrolle.

TCP-Header:



Flag ACK („acknowledgement“)

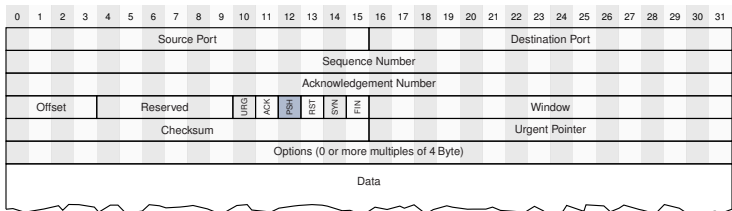
- ▶ Ist das Flag gesetzt, handelt es sich um eine Empfangsbestätigung.
- ▶ Bestätigungen können bei TCP auch „huckepack“ (engl. **piggy backing**) übertragen werden, d. h. es werden gleichzeitig Nutzdaten von *A* nach *B* übertragen und ein zuvor von *B* nach *A* gesendetes Segment bestätigt.
- ▶ Die Acknowledgement-Number gibt bei TCP stets **das nächste erwartete Byte** an.

Case Study: Transmission Control Protocol (TCP)

Das **Transmission Control Protocol (TCP)** ist das dominante Transportprotokoll im Internet. Es bietet

- ▶ gesicherte / stromorientierte Übertragung sowie
- ▶ Mechanismen für Fluss- und Staukontrolle.

TCP-Header:



Flag PSH („push“)

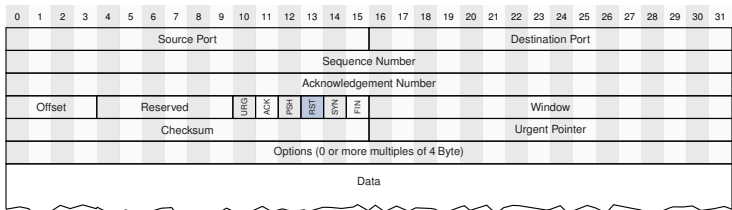
- ▶ Ist das Flag gesetzt, werden sende- und empfangsseitige Puffer des TCP-Stacks umgangen.
- ▶ Sinnvoll für interaktive Anwendungen (z. B. **Telnet**-Verbindungen → siehe Kapitel 6).

Case Study: Transmission Control Protocol (TCP)

Das **Transmission Control Protocol (TCP)** ist das dominante Transportprotokoll im Internet. Es bietet

- ▶ gesicherte / stromorientierte Übertragung sowie
- ▶ Mechanismen für Fluss- und Staukontrolle.

TCP-Header:



Flag RST („reset“)

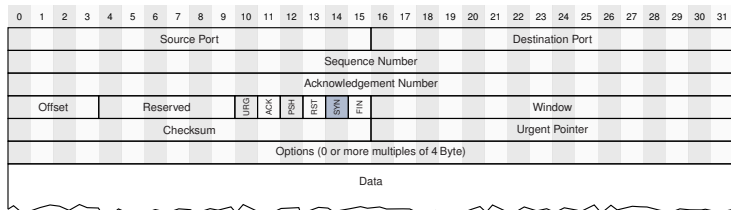
- ▶ Dient dem Abbruch einer TCP-Verbindung ohne ordnungsgemäßen Verbindungsabbau.

Case Study: Transmission Control Protocol (TCP)

Das **Transmission Control Protocol (TCP)** ist das dominante Transportprotokoll im Internet. Es bietet

- ▶ gesicherte / stromorientierte Übertragung sowie
- ▶ Mechanismen für Fluss- und Staukontrolle.

TCP-Header:



Flag SYN („synchronization“)

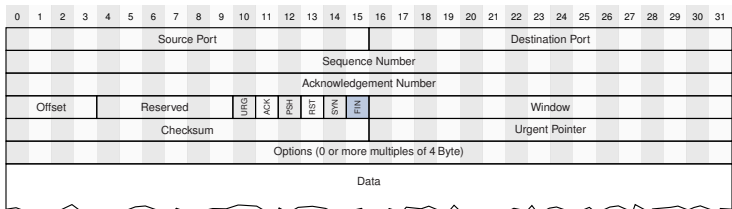
- ▶ Ist das Flag gesetzt, handelt es sich um ein Segment, welches zum Verbindungsaufbau gehört (initialer Austausch von Sequenznummern).
- ▶ Ein gesetztes SYN-Flag inkrementiert Sequenz- und Bestätigungsnummern um 1 obwohl keine Nutzdaten transportiert werden.

Case Study: Transmission Control Protocol (TCP)

Das **Transmission Control Protocol (TCP)** ist das dominante Transportprotokoll im Internet. Es bietet

- ▶ gesicherte / stromorientierte Übertragung sowie
- ▶ Mechanismen für Fluss- und Staukontrolle.

TCP-Header:



Flag FIN („finish“)

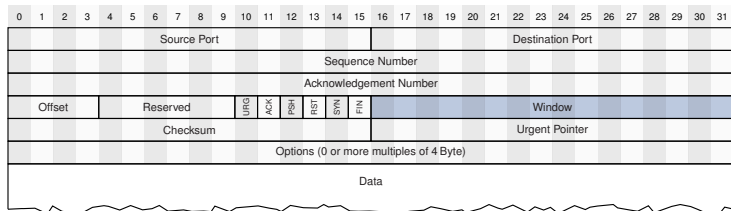
- ▶ Ist das Flag gesetzt, handelt es sich um ein Segment, welches zum Verbindungsabbau gehört.
- ▶ Ein gesetztes FIN-Flag inkrementiert Sequenz- und Bestätigungsnummern um 1 obwohl keine Nutzdaten transportiert werden.

Case Study: Transmission Control Protocol (TCP)

Das **Transmission Control Protocol (TCP)** ist das dominante Transportprotokoll im Internet. Es bietet

- ▶ gesicherte / stromorientierte Übertragung sowie
- ▶ Mechanismen für Fluss- und Staukontrolle.

TCP-Header:



Receive Window

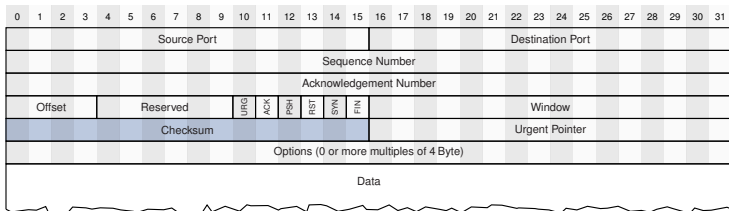
- ▶ Größe des aktuellen Empfangsfensters W_r in Byte.
- ▶ Ermöglicht es dem Sender, die Datenrate des Senders zu drosseln.

Case Study: Transmission Control Protocol (TCP)

Das [Transmission Control Protocol \(TCP\)](#) ist das dominante Transportprotokoll im Internet. Es bietet

- ▶ gesicherte / stromorientierte Übertragung sowie
- ▶ Mechanismen für Fluss- und Staukontrolle.

TCP-Header:



Checksum

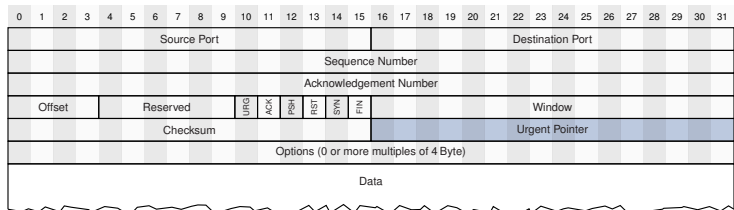
- ▶ Checksumme über Header und Daten.
- ▶ Wie bei UDP wird zur Berechnung ein [Pseudo-Header](#) verwendet.

Case Study: Transmission Control Protocol (TCP)

Das **Transmission Control Protocol (TCP)** ist das dominante Transportprotokoll im Internet. Es bietet

- ▶ gesicherte / stromorientierte Übertragung sowie
- ▶ Mechanismen für Fluss- und Staukontrolle.

TCP-Header:



Urgent Pointer (selten verwendet)

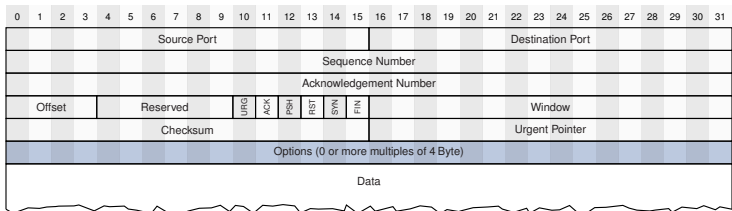
- ▶ Gibt das Ende der „Urgent-Daten“ an, welche unmittelbar nach dem Header beginnen und bei gesetztem URG-Flag sofort an höhere Schichten weitergereicht werden sollen.

Case Study: Transmission Control Protocol (TCP)

Das **Transmission Control Protocol (TCP)** ist das dominante Transportprotokoll im Internet. Es bietet

- ▶ gesicherte / stromorientierte Übertragung sowie
- ▶ Mechanismen für Fluss- und Staukontrolle.

TCP-Header:



Options

- ▶ Zusätzliche Optionen, z. B. **Window Scaling** (s. Übung), selektive Bestätigungen oder Angabe der **Maximum Segment Size (MSS)**.

Anmerkungen zur MSS

- ▶ Die MSS gibt die maximale Größe eines TCP-Segments (Nutzdaten ohne TCP-Header) an.
- ▶ Zum Vergleich gibt die MTU (Maximum Transfer Unit) die maximale Größe der Nutzdaten aus Sicht von Schicht 2 an (alles einschließlich des IP-Headers).
- ▶ In der Praxis sollte die MSS so gewählt werden, dass keine IP-Fragmentierung beim Senden notwendig ist.

Beispiele:

- ▶ MSS bei FastEthernet
 - ▶ MTU beträgt 1500 B
 - ▶ Davon entfallen 20 B auf den IP-Header und weitere 20 B auf den TCP-Header (sofern keine Optionen verwendet werden)
 - ▶ Die sinnvolle MSS beträgt demnach 1460 B.
- ▶ DSL-Verbindungen
 - ▶ Zwischen Ethernet- und IP-Header wird ein 8 B PPPoE-Header eingefügt
 - ▶ Demzufolge sollte die MSS auf 1452 B reduziert werden
- ▶ VPN-Verbindungen
 - ▶ Abhängig vom eingesetzten Verschlüsselungsverfahren sind weitere Header notwendig
 - ▶ Die sinnvolle MSS ist hier nicht immer offensichtlich

Fluss- und Staukontrolle bei TCP

TCP-Flusskontrolle

Ziel der **Flusskontrolle** ist es, Überlastsituationen beim Empfänger zu vermeiden. Dies wird erreicht, indem der Empfänger eine Maximalgröße für das Sendefenster des Senders vorgibt.

- ▶ Empfänger teilt dem Sender über das Feld **Receive Window** im TCP-Header die aktuelle Größe des Empfangsfensters W_r mit.
- ▶ Der Sender interpretiert diesen Wert als die maximale Anzahl an Byte, die ohne Abwarten einer Bestätigung übertragen werden dürfen.
- ▶ Durch Herabsetzen des Wertes kann die Übertragungsrates des Senders gedrosselt werden, z. B. wenn sich der Empfangspuffer des Empfängers füllt.

TCP-Staukontrolle

Ziel der **Staukontrolle** ist es, Überlastsituationen im Netz zu vermeiden. Dazu muss der Sender Engpässe im Netz erkennen und die Größe des Sendefensters entsprechend anpassen.

Zu diesem Zweck wird beim Sender zusätzlich ein **Staukontrollfenster** (engl. **Congestion Window**) W_c eingeführt, dessen Größe wir mit w_c bezeichnen:

- ▶ W_c wird vergrößert, solange Daten verlustfrei übertragen werden
- ▶ W_c wird verkleinert, wenn Verluste auftreten
- ▶ Für das tatsächliche Sendefenster gilt stets $w_s = \min\{w_c, w_r\}$

TCP-Staukontrolle

Man unterscheidet bei TCP zwischen zwei Phasen der Staukontrolle:

1 Slow-Start:

Für jedes bestätigte Segment wird W_c um eine MSS vergrößert. Dies führt zu **exponentiellem Wachstum** des Staukontrollfensters bis ein Schwellwert (engl. **Congestion Threshold**) erreicht ist. Danach wird mit der Congestion-Avoidance-Phase fortgefahren.

2 Congestion Avoidance:

Für jedes bestätigte Segment wird W_c lediglich um $(1/w_c) \cdot MSS$ vergrößert, d. h. nach Bestätigung eines vollständigen Staukontrollfensters um genau eine MSS. Dies führt zu **linearem Wachstum** des Staukontrollfensters.

Es gibt mehrere Varianten von TCP, welche sich hinsichtlich ihres Verhaltens beim Auftreten von Segmentverlusten unterscheiden. Im Folgenden beziehen wir uns auf „TCP-Reno“:

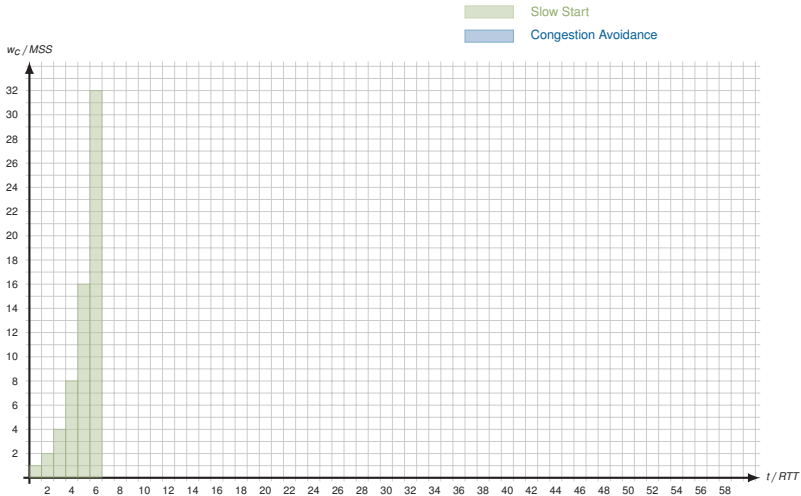
1 3 doppelte Bestätigungen (Duplicate ACKs)

- ▶ Setze den Schwellwert für die Stauvermeidung auf die $w_c/2$
- ▶ Reduziere W_c auf die Größe dieses Schwellwerts
- ▶ Beginne mit der Stauvermeidungsphase

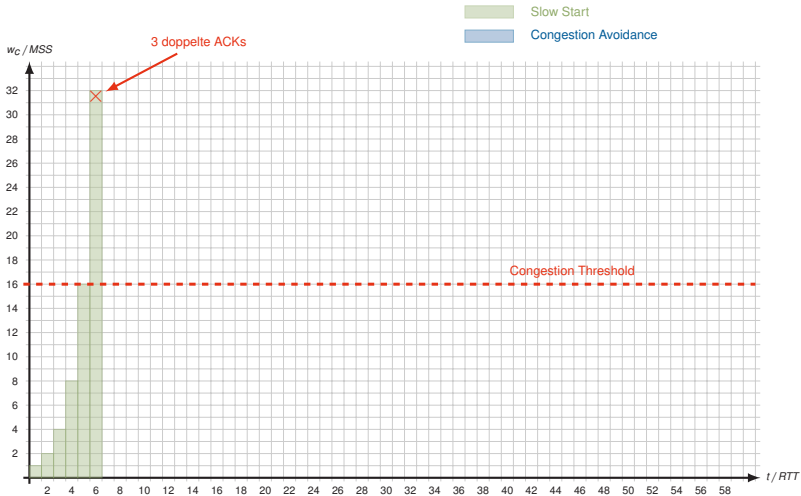
2 Timeout

- ▶ Setze den Schwellwert für die Stauvermeidung auf die $w_c/2$
 - ▶ Setze $w_c = 1$
 - ▶ Beginne mit einem neuen Slow-Start
-
- ▶ Der Vorgänger **TCP-Tahoe** unterscheidet nicht zwischen diesen beiden Fällen und führt immer Fall 2 aus.
 - ▶ Es gibt eine Reihe weiterer TCP-Versionen, die sich insbesondere hinsichtlich ihres Staukontrollverhaltens unterscheiden.
 - ▶ Grundsätzlich sind alle TCP-Versionen kompatibel zueinander, allerdings können sich die unterschiedlichen Staukontrollverfahren gegenseitig nachteilig beeinflussen.

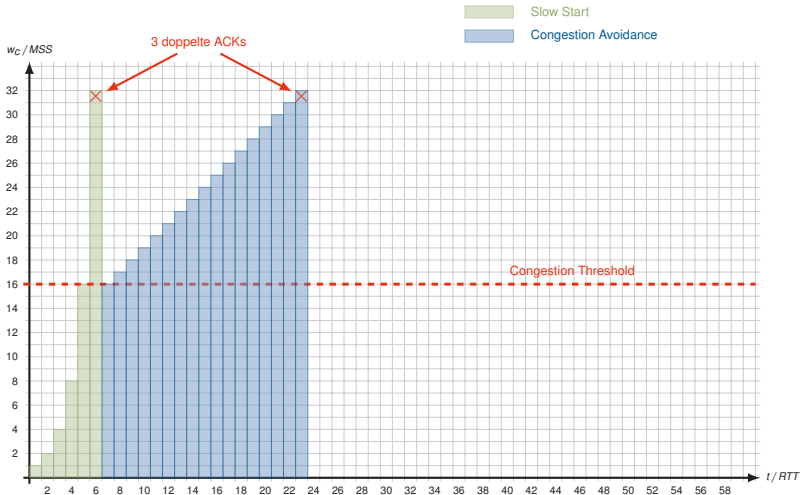
Beispiel: TCP-Reno



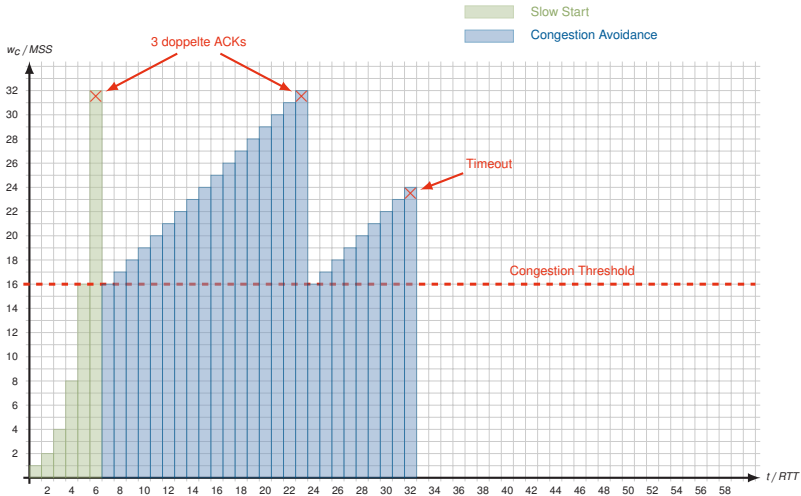
Beispiel: TCP-Reno



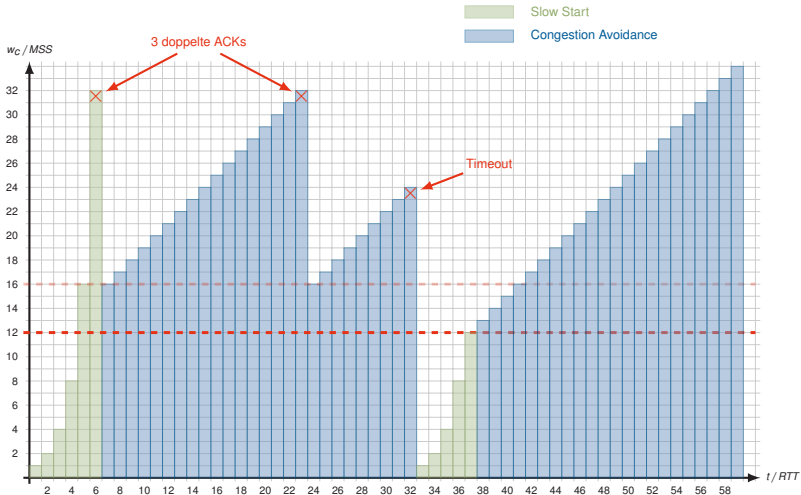
Beispiel: TCP-Reno



Beispiel: TCP-Reno



Beispiel: TCP-Reno



Anmerkungen

Obwohl TCP gesicherte Verbindungen ermöglicht, dient es **nicht der Kompensierung eines unzuverlässigen Data Link Layers!**

- ▶ TCP interpretiert von Verlust von Paketen (Daten und Bestätigungen) stets als eine Folge einer **Überlastsituation**.
- ▶ In der Folge reduziert TCP die Datenrate.
- ▶ Handelt es sich bei den Paketverlusten jedoch um die Folge von Bitfehlern oder Kollision, so wird die Datenrate unnötiger Weise gedrosselt.
- ▶ Durch die ständige Halbierung der Datenrate oder neue Slow-Starts kann das Sendefenster nicht mehr auf sinnvolle Größen anwachsen.
- ▶ In der Praxis ist TCP bereits mit 1 % Paketverlust, der nicht auf Überlast zurückzuführen ist, bereits vollständig überforder!

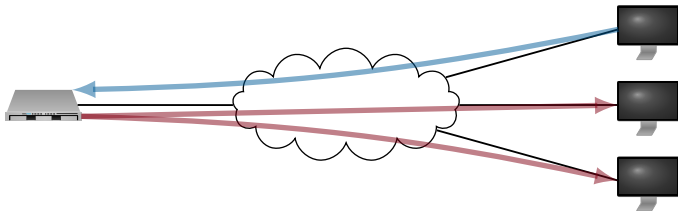
⇒ Die Schichten 1 – 3 müssen eine für TCP „ausreichend geringe“ Paketfehlerrate bereitstellen

- ▶ In der Praxis bedeutet dies, dass Verlustwahrscheinlichkeiten in der Größenordnung von 10^{-3} und niedriger notwendig sind.
- ▶ Bei Bedarf müssen zusätzliche Bestätigungsverfahren auf Schicht 2 zum Einsatz kommen, um dies zu gewährleisten (z. B. IEEE 802.11).

Case Study: TCP-Relay-Chat

Was wir wollen:

- ▶ Einen Server, welcher N Clientverbindungen gleichzeitig unterstützt
- ▶ Sendet ein Client eine Nachricht an den Server, soll diese an alle anderen Clients weitergeleitet werden
- ▶ Dies entspricht einem Chatroom
- ▶ Server und Client sind nun zwei unterschiedliche Programme



Welche Sprache?

- ▶ C natürlich (;

Der Server

- ▶ Sieht ähnlich aus wie unser UDP-Chat ...

```

if ( 0 > (sd=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP)) ) {
    fprintf(stderr,"socket() failed: %d-%s\n",errno,strerror(errno));
    exit(1);
}

local.sin_family    = AF_INET;
local.sin_port      = htons(local_port);
local.sin_addr.s_addr = INADDR_ANY;

if ( 0 > bind(sd,(struct sockaddr *)&local,sizeof(local)) ) {
    fprintf(stderr,"bind() failed: %d-%s\n",errno,strerror(errno));
    exit(1);
}

if ( 0 > listen(sd,MAXCLIENTS) ) {
    fprintf(stderr,"listen() failed: %d-%s\n",errno,strerror(errno));
    exit(1);
}
    
```

Neu sind

- ▶ der Socket-Typ `SOCK_STREAM` und das Protokoll `IPPROTO_TCP` sowie
- ▶ der Aufruf von `listen()`, welcher den Socket als **passiv** markiert.

Letzteres bedeutet, dass über diesen Socket **keine** Daten versendet oder empfangen werden sondern stattdessen eingehende Verbindungen auf diesem Socket erwartet werden.

Verbindet sich ein Client, muss die Verbindung akzeptiert werden:

```
while(1) {
    rfd = rfd;

    if ( 0 > select(maxfd+1,&rfd,NULL,NULL,NULL) ) {
        fprintf(stderr,"select() failed: %d-%s\n",errno,strerror(errno));
        exit(1);
    }

    if ( FD_ISSET(sd,&rfd) ) {
        if ( 0 > (csd=accept(sd,(struct sockaddr *)&client,&slen)) ) {
            fprintf(stderr,"accept() failed: %d-%s\n",errno,strerror(errno));
            exit(1);
        }
        cl_add(&clients,csd,client); // Client in ne Liste schieben
        FD_SET(csd,&rfd);
        maxfd = MAX(maxfd,csd);
    }
    (...)
}
```

- ▶ `select()` reagiert, wenn auf dem Server-Socket `sd` eine Verbindung angezeigt wird
- ▶ `accept()` akzeptiert die Verbindung und erzeugt einen **neuen** Socket, der die Verbindung zum Client repräsentiert
- ▶ Der Server-Socket bleibt aktiv – es könnte ja noch ein Client kommen
- ▶ Der neue Socket muss natürlich in das Set der zu überwachenden File-Deskriptoren, falls uns der Client mal was schickt

Wenn uns nun ein Client was schickt, müssen wir

- ▶ den richtigen Socket raussuchen,
- ▶ die Daten vom Client empfangen und anschließend
- ▶ die empfangene Nachricht an alle anderen Clients weiterleiten.

Die Clients verwaltet man sinnvoller Weise in einer Liste (etwas ekelhaft mit C). Hat man den richtigen Socket gefunden, kann man mittels `recv()` und `send()` empfangen und senden:

```
while(1) {  
    (...)  
    len = recv(cl.sd, inbuff, BUFSIZE, 0);  
    (...)  
    len = send(cl.sd, outbuff, strlen(outbuff), 0);  
    (...)  
}
```

- ▶ Prinzipiell kann man hier anstelle von `recv()` und `send()` auch die Syscalls `read()` und `write()` nutzen, da im Gegensatz zum verbindungslosen UDP Sender und Empfänger schon feststehen
- ▶ `recv()` und `send()` sind aber zu bevorzugen, da hier bestimmte Ausnahmen (z. B. Verbindung unterbrochen) sinnvoll signalisiert werden

Der Client

Der Client ist viel leichter:

```
remote.sin_family    = AF_INET;
remote.sin_port      = htons(SERVERPORT);
remote.sin_addr.s_addr = htonl(SERVERIP);

if ( 0 > (sd=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP)) ) {
    fprintf(stderr,"socket() failed: %d-%s\n",errno,strerror(errno));
    exit(1);
}

if ( 0 > connect(sd,(struct sockaddr *)&remote,sizeof(remote)) ) {
    fprintf(stderr,"connect() failed: %d-%s\n",errno,strerror(errno));
    exit(1);
}
```

- ▶ Sofern wir als Client unseren Absenderport nicht angeben wollen (sondern uns vom Betriebssystem einen zuweisen lassen wollen), können wir auf ein `bind()` verzichten.
- ▶ Wir brauchen jetzt aber in jedem Fall ein `connect()`, um uns mit dem Server zu verbinden
- ▶ `connect()` muss natürlich gesagt werden, mit wem wir uns verbinden wollen (→ `struct sockaddr_in`)

Fast geschafft, es fehlt nur noch das Senden und Empfangen von Nachrichten:

```
(...)  
while(1) {  
    rfd = rfd;  
  
    if ( 0 > select(maxfd+1,&rfd,NULL,NULL,NULL) ) {  
        fprintf(stderr,"select() failed: %d-%s\n",errno,strerror(errno));  
        exit(1);  
    }  
  
    if ( FD_ISSET(STDIN_FILENO,&rfd) ) {  
        if( NULL == (s=gets(buffer)) )  
            continue;  
        if ( 0 >= (len=send(sd,buffer,MIN(strlen(buffer)+1,BUFFLEN),0)) ) { (...) }  
    }  
    if ( FD_ISSET(sd,&rfd) ) {  
        if ( 0 >= (len=recv(sd,buffer,BUFFLEN-1,0)) ) { (...) }  
        fprintf(stdout,">>%s\n",buffer);  
    }  
}
```

Jetzt wird chattet:

- ▶ Ladet Euch den Client runter: `svn update → ./pub/tcpclient/`
- ▶ Ein Server läuft auf `grnvs000.net.in.tum.de` Port 6112
- ▶ Der erste, der den Server abschießt, kriegt ein Bier!

Übersicht

- 1 Motivation
- 2 Multiplexing
- 3 Verbindungslose Übertragung
- 4 Verbindungsorientierte Übertragung
- 5 Network Address Translation (NAT)**

Network Address Translation (NAT)

In Kapitel 3 haben wir gelernt, dass

- ▶ IP-Adressen zur End-zu-End-Adressierung verwendet werden,
- ▶ aus diesem Grund global eindeutig sind und
- ▶ speziell die heute noch immer hauptsächlich verwendeten IPv4-Adressen sehr knapp sind.

Frage: Müssen IP-Adressen immer eindeutig sein?

Network Address Translation (NAT)

In Kapitel 3 haben wir gelernt, dass

- ▶ IP-Adressen zur End-zu-End-Adressierung verwendet werden,
- ▶ aus diesem Grund global eindeutig sind und
- ▶ speziell die heute noch immer hauptsächlich verwendeten IPv4-Adressen sehr knapp sind.

Antwort: Nein, IP-Adressen müssen nicht eindeutig sein, wenn

- ▶ keine Kommunikation mit im Internet befindlichen Hosts möglich sein muss **oder**
- ▶ die nicht eindeutigen **privaten IP-Adressen** auf geeignete Weise in **öffentliche Adressen** übersetzt werden.

Definition: NAT

Als **Network Address Translation (NAT)** bezeichnet man allgemein Techniken, welche es ermöglichen, N **private** (nicht global eindeutige) IP-Adressen auf M **globale** (weltweit eindeutige) IP-Adressen abzubilden.

- ▶ $N > M$: In diesem Fall wird eine öffentliche IP-Adresse von mehreren Computer gleichzeitig genutzt. Eine eindeutige Unterscheidung kann mittels **Port-Multiplexing** erreicht werden. Der häufigste Fall ist $M = 1$, z. B. ein privater DSL-Anschluss.
- ▶ $N \leq M$: Die Übersetzung geschieht statisch oder dynamisch indem jeder privaten IP-Adresse mind. eine öffentliche IP-Adresse zugeordnet wird.

Was sind private IP-Adressen?

Private IP-Adressen sind Adressen aus speziellen Adressbereichen, welche

- ▶ zur privaten Nutzung ohne vorherige Registrierung freigegeben sind,
- ▶ deswegen in unterschiedlichen Netzen vorkommen können,
- ▶ aus diesem Grund weder eindeutig noch zur End-Zu-End-Adressierung zwischen öffentlich erreichbaren Netzen geeignet sind und
- ▶ daher IP-Pakete mit privaten Empfänger-Adressen von Routern im Internet nicht weitergeleitet werden (oder werden sollten).

Die privaten Adressbereiche gemäß **RFC 5735** sind:

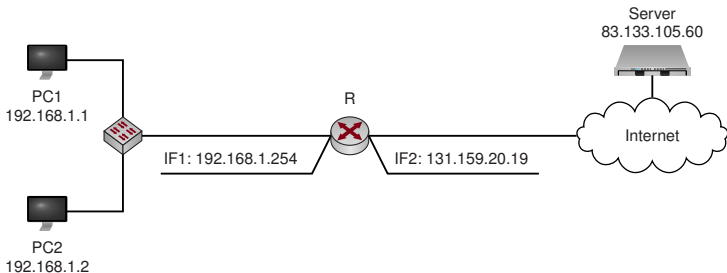
- ▶ 10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16

Der Bereich 169.254.0.0/16 wird nach **RFC 3927** zur **automatischen Adressvergabe (Automatic Private IP Addressing)** genutzt:

- ▶ Startet ein Computer ohne statisch vergebene Adresse, versucht dieser, einen DHCP-Server zu erreichen.
- ▶ Kann kein DHCP-Server gefunden werden, vergibt das Betriebssystem eine zufällig gewählte Adresse aus diesem Adressblock.
- ▶ Schlägt anschließend die ARP-Auflösung zu dieser Adresse fehl, wird angenommen, dass diese Adresse im lokalen Subnetz noch nicht verwendet wird. Andernfalls wird eine andere Adresse gewählt und der Vorgang wiederholt.

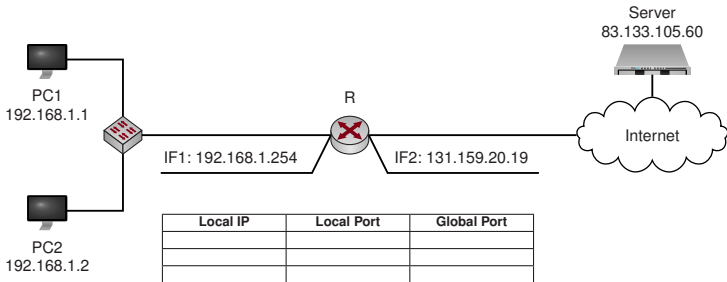
Wie funktioniert NAT im Detail?

Im Allgemeinen übernehmen Router die Netzwerkadressübersetzung:



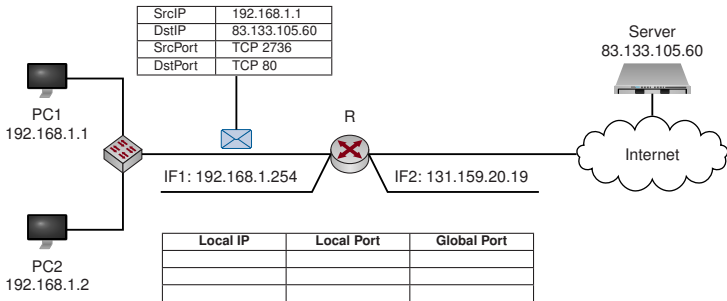
- ▶ PC1, PC2 und R können mittels privater IP-Adressen im Subnetz 192.168.1.0/24 miteinander kommunizieren.
- ▶ R ist über seine öffentliche Adresse 131.159.20.19 global erreichbar.
- ▶ PC1 und PC2 können wegen ihrer privaten Adressen nicht direkt mit anderen Hosts im Internet kommunizieren.
- ▶ Hosts im Internet können ebensowenig PC1 oder PC2 erreichen – selbst dann, wenn sie wissen, dass sich PC1 und PC2 hinter R befinden und die globale Adresse von R bekannt ist.

PC1 greift auf eine Webseite zu, welche auf dem Server mit der Adresse 83.133.105.60 liegt:



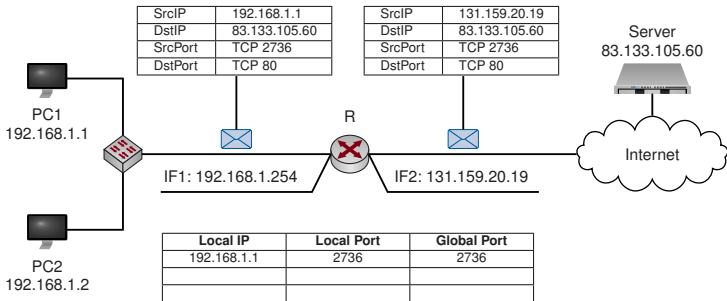
- ▶ Die NAT-Tabelle von R sei zu Beginn leer.

PC1 greift auf eine Webseite zu, welche auf dem Server mit der Adresse 83.133.105.60 liegt:



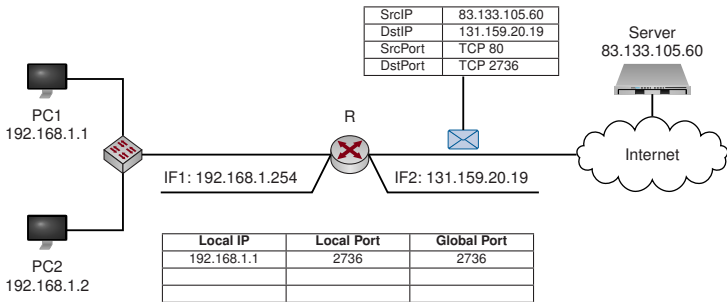
- ▶ Die NAT-Tabelle von R sei zu Beginn leer.
- ▶ PC1 sendet ein Paket (TCP SYN) an den Server:
 - ▶ PC1 verwendet seine private IP-Adresse als Absenderadresse
 - ▶ Der Quellport wird von PC1 zufällig im Bereich [1024, 65535] gewählt (sog. [Ephemeral Ports](#))
 - ▶ Der Zielport ist durch das Application Layer Protocol vorgegeben (80 = HTTP)

PC1 greift auf eine Webseite zu, welche auf dem Server mit der Adresse 83.133.105.60 liegt:



- ▶ Die NAT-Tabelle von R sei zu Beginn leer.
- ▶ PC1 sendet ein Paket (TCP SYN) an den Server:
 - ▶ PC1 verwendet seine private IP-Adresse als Absenderadresse
 - ▶ Der Quellport wird von PC1 zufällig im Bereich [1024, 65535] gewählt (sog. [Ephemeral Ports](#))
 - ▶ Der Zielport ist durch das Application Layer Protocol vorgegeben (80 = HTTP)
- ▶ Adressübersetzung an R:
 - ▶ R tauscht die Absenderadresse durch seine eigene globale Adresse aus
 - ▶ Sofern der Quellport nicht zu einer Kollision in der NAT-Tabelle führen würde, wird dieser beibehalten (andernfalls wird dieser ebenfalls ausgetauscht)
 - ▶ R erzeugt einen neuen Eintrag in seiner NAT-Tabelle, welche die Änderungen an dem Paket dokumentieren

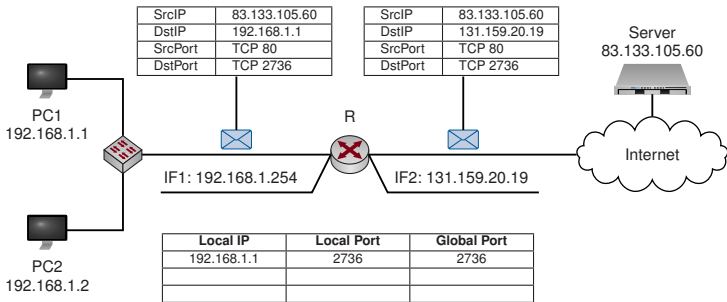
Antwort vom Server an PC1:



► Der Server generiert eine Antwort:

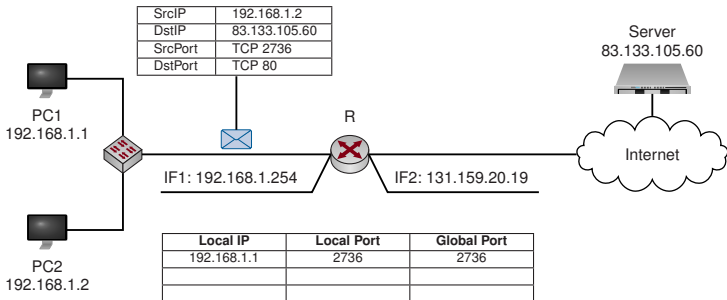
- Der Server weiß nichts von der Adressübersetzung und hält R für PC1
- Die Empfängeradresse ist daher die öffentliche IP von R, der Zielport der von R übersetzte Quellport aus der vorherigen Nachricht

Antwort vom Server an PC1:



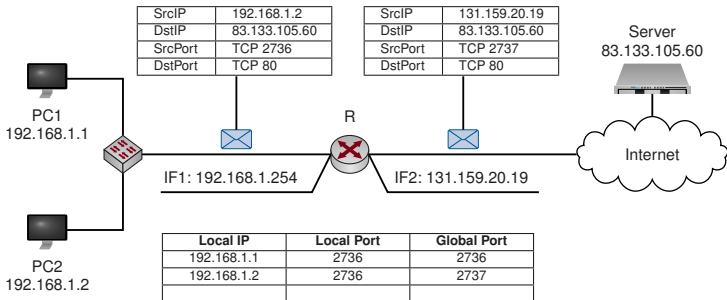
- ▶ Der Server generiert eine Antwort:
 - ▶ Der Server weiß nichts von der Adressübersetzung und hält R für PC1
 - ▶ Die Empfängeradresse ist daher die öffentliche IP von R, der Zielport der von R übersetzte Quellport aus der vorherigen Nachricht
- ▶ R macht die Adressübersetzung rückgängig
 - ▶ In der NAT-Tabelle wird nach der Zielportnummer in der Spalte Global Port gesucht, dieser in Local Port zurückübersetzt und die Ziel-IP des Pakets gegen die private IP-Adresse von PC1 ausgetauscht
 - ▶ Das so modifizierte Paket wird an PC1 weitergeleitet
 - ▶ Wie der Server weiß auch PC1 nichts von der Adressübersetzung

PC2 greift nun ebenfalls auf den Server zu:



- ▶ PC2 sendet ebenfalls ein Paket (TCP SYN) an den Server:
 - ▶ Rein zufällig wählt PC2 denselben Quell-Port wie PC1 (Portnummer 2736)

PC2 greift nun ebenfalls auf den Server zu:



- ▶ PC2 sendet ebenfalls ein Paket (TCP SYN) an den Server:
 - ▶ Rein zufällig wählt PC2 denselben Quell-Port wie PC1 (Portnummer 2736)
- ▶ Adressübersetzung an R:
 - ▶ R bemerkt, dass es bereits einen zu PC1 gehörenden Eintrag für den lokalen Port 2736 gibt
 - ▶ R erzeugt einen neuen Eintrag in der NAT-Tabelle, wobei für den globalen Port ein zufälliger Wert gewählt wird (z. B. der ursprüngliche Port von PC2 + 1)
 - ▶ Das Paket von PC2 wird entsprechend modifiziert und an den Server weitergeleitet
- ▶ Aus Sicht des Servers hat der „Computer“ R einfach zwei TCP-Verbindungen aufgebaut.

Ein Router könnte in die NAT-Tabelle zusätzliche Informationen aufnehmen:

- ▶ Ziel-IP und Ziel-Port
- ▶ Das verwendete Protokoll (TCP, UDP)
- ▶ Die eigene globale IP (sinnvoll, wenn ein Router mehr als eine globale IP besitzt)

In Abhängigkeit der gespeicherten Informationen unterscheidet man unterschiedliche Typen von NAT. Die eben diskutierte Variante (zzgl. eines Vermerks des Protokolls in der NAT-Tabelle) bezeichnet man als **Full Cone NAT**.

Eigenschaften von Full Cone NAT:

- ▶ Bei eingehenden Verbindungen findet keine Prüfung der Absender-IP oder des Absender-Ports statt, da die NAT-Tabelle nur den Ziel-Port und die zugehörige IP-Adresse bzw. Portnummer im lokalen Netz enthält.
- ▶ Existiert also einmal ein Eintrag in der NAT-Tabelle, so ist ein interner Host aus dem Internet über diesen Eintrag auch für jeden erreichbar, der ein TCP- bzw. UDP-Paket an die richtige Portnummer sendet.

Andere NAT-Varianten:

- ▶ Port Restricted NAT
- ▶ Address Restricted NAT
- ▶ Port and Address Restricted NAT
- ▶ Symmetric NAT

Allgemeine Anmerkungen

- ▶ Schützt ein NAT vor Angriffen aus dem Internet?
 - ▶ Teilweise.
 - ▶ Restriktive NAT-Varianten bieten insofern einen grundlegenden Schutz, da sie eingehende Verbindungen ohne vorherigen Verbindungsaufbau aus dem lokalen Netz heraus erlauben.
 - ▶ Eine darüber hinausgehende Filterung von Verbindungen (wie es bei einer Firewall der Fall wäre) findet nicht statt.
- ▶ Wie viele Einträge kann eine NAT-Tabelle fassen?
 - ▶ Im einfachsten Fall (Full Cone NAT) beträgt die theoretische Maximalgrenze ca. 2^{16} pro Transportprotokoll (TCP und UDP) und pro globaler IP-Adresse.
 - ▶ Bei komplexeren NAT-Typen sind durch die Aufnahme der Ziel-Ports mehr Kombinationen möglich.
 - ▶ In der Praxis ist die Größe durch die Fähigkeiten des Routers beschränkt (einige 1000 Mappings).
- ▶ Werden Mappings aus der NAT-Tabelle wieder gelöscht?
 - ▶ Dynamisch erzeugte Mappings werden nach einer gewissen Inaktivitätszeit gelöscht.
 - ▶ U. U. entfernt ein NAT-fähiger Router auch Mappings sofort, wenn er einen TCP-Verbindungsabbau erkennt (implementierungsabhängig).
- ▶ Können Einträge in der NAT-Tabelle auch von Hand erzeugt werden
 - ▶ Ja, diesen Vorgang nennt man [Port Forwarding](#).
 - ▶ Auf diese Weise wird es möglich, hinter einem NAT einen auf einem bestimmten Port öffentlich erreichbaren Server zu betreiben.

NAT und ICMP

- ▶ NAT verwendet Portnummern des Transportprotokolls
- ▶ Was ist, wenn das Transportprotokoll keine Portnummern hat oder IP-Pakete ohne TCP-/UDP-Header verschickt werden, z. B. ICMP?

NAT und ICMP

- ▶ NAT verwendet Portnummern des Transportprotokolls
- ▶ Was ist, wenn das Transportprotokoll keine Portnummern hat oder IP-Pakete ohne TCP-/UDP-Header verschickt werden, z. B. ICMP?

Antwort: Die ICMP-ID kann anstelle der Portnummern genutzt werden.

NAT und ICMP

- ▶ NAT verwendet Portnummern des Transportprotokolls
- ▶ Was ist, wenn das Transportprotokoll keine Portnummern hat oder IP-Pakete ohne TCP-/UDP-Header verschickt werden, z. B. ICMP?

Antwort: Die ICMP-ID kann anstelle der Portnummern genutzt werden.

Aus Assignment 2: Traceroute funktioniert nicht, wenn die Virtualbox-VM NAT nutzt. Warum?

- ▶ Traceroute basiert auf ICMP-TTL-Exceeded
- ▶ Diese Nachrichten nutzen (anders als ein ICMP-Echo-Reply) keine ICMP-ID.
- ▶ Das liegt einfach schon daran, dass jedes beliebige Paket (nicht nur ein Echo-Request) das TTL-Exceeded ausgelöst haben könnte und dieses (wie im Fall eines TCP-Pakets) natürlich keine ICMP-ID besitzt.
- ▶ Stattdessen trägt das Time-Exceeded den vollständigen IP-Header und die ersten 8 Byte der Payload des Pakets, welches den Time-Exceeded ausgelöst hat.
- ▶ Eine NAT-Implementierung müsste nun (genau wie Sie das im Assignment getan haben / hätten tun sollen) im Fall eines TTL-Exceeded in diesen ersten 8 Byte nach der ICMP-ID eines Echo-Requests oder nach den Portnummern eines Transportprotokolls suchen, um die Übersetzung rückgängig machen zu können.
- ▶ Genau das tut die NAT-Implementierung von Virtualbox aber nicht.

Zusammenfassung

In diesem Kapitel haben wir gelernt, wie

- ▶ Anwendungen auf einen Host voneinander unterschieden bzw. adressiert werden können (**Portnummern**),
- ▶ welche Transportprotokolle häufig genutzt werden (**UDP, TCP**),
- ▶ wie zuverlässige Datenübertragung mittels Bestätigungsverfahren funktioniert (**Go-Back-N, Selective Repeat**)
- ▶ wie Fluss- und Staukontrollemechanismen funktionieren,
- ▶ was private IP-Adressen sind und
- ▶ wie sich mehrere Hosts eine öffentliche IP-Adresse teilen können (**NAT**).

Unter anderem für die Endterm wichtig:

- ▶ Wie funktioniert das Zusammenspiel öffentlicher und privater Netzwerke?
- ▶ Gegeben eine Netztopologie mit NAT-fähigem Router sollten Sie in der Lage sein, an bestimmten Punkten im Netzwerk IP-Adressen und Portnummern von Nachrichten anzugeben.
- ▶ Ebenso sollten Sie in der Lage sein, einige weitere interessante Headerfelder der vorherigen Kapitel für solche Nachrichten angeben zu können, z. B. TTL, Quell-MAC, Ziel-MAC, etc.
- ▶ Ein Blick auf die Altklausuren könnte sinnvoll sein...



Literaturhinweise und Quellenangaben I