

# Grundlagen Rechnernetze und Verteilte Systeme (GRNVS)

IN0010 – SoSe 2019

**Prof. Dr.-Ing. Georg Carle**

Dr.-Ing. Stephan Günther, Johannes Naab, Henning Stubbe

Lehrstuhl für Netzarchitekturen und Netzdienste  
Fakultät für Informatik  
Technische Universität München

Motivation

Multiplexing

Verbindungslose Übertragung

Verbindungsorientierte Übertragung

Network Address Translation (NAT)

Codedemos

Literaturangaben

## Motivation

### Aufgaben der Transportschicht

Multiplexing

Verbindungslose Übertragung

Verbindungsorientierte Übertragung

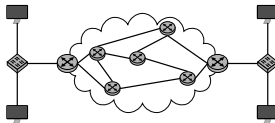
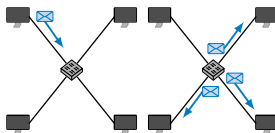
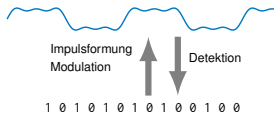
Network Address Translation (NAT)

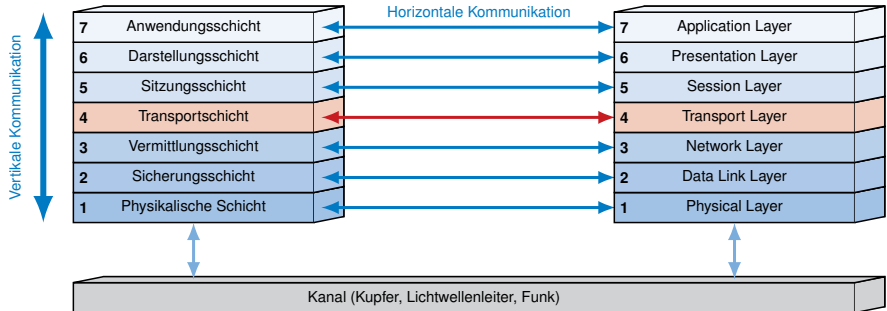
Codedemos

Literaturangaben

## 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 Host Ende-zu-Ende adressiert und Daten paketorientiert vermittelt werden (Schicht 3)



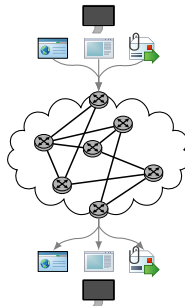


Die wesentlichen Aufgaben der Transportschicht sind

- **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

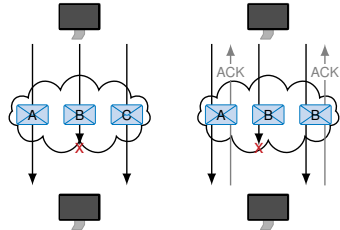


Die wesentlichen Aufgaben der Transportschicht sind

- **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

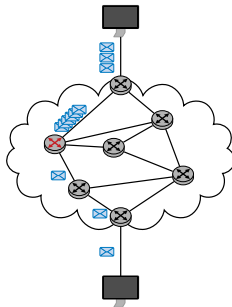


Die wesentlichen Aufgaben der Transportschicht sind

- 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 drohende Überlast im Netz
- Flusskontrolle (Flow Control)
  - Laststeuerung durch den Empfänger





Motivation

**Multiplexing**

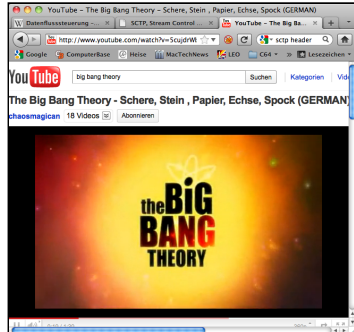
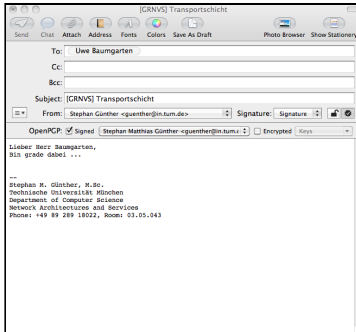
Verbindungslose Übertragung

Verbindungsorientierte Übertragung

Network Address Translation (NAT)

Codedemos

Literaturangaben



OSI Schicht 7 – 5

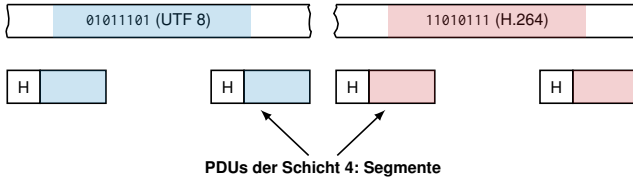
01011101 (UTF 8)

11010111 (H.264)

Transportschicht

## 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

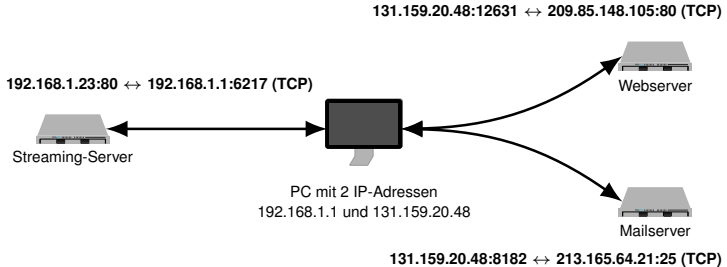
- einen Quellport und
- einen Zielport,

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

⇒ **5-Tupel** bestehend aus:

(SrcIPAddr, SrcPort, DstIPAddr, DstPort, Protocol)

## Beispiel:



- Portnummern sind bei den bekannten Transportprotokollen 16 bit lang.
- Betriebssysteme verwenden das 5-Tupel (IP-Adressen, Portnummern, Protokoll), um Anwendungen **Sockets** bereitzustellen.
- Eine Anwendung wiederum adressiert einen Socket mittels eines **File-Descriptors** (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()`).

Motivation

Multiplexing

Verbindungslose Übertragung

User Datagram Protocol (UDP)

Verbindungsorientierte Übertragung

Network Address Translation (NAT)

Codedemos

Literaturangaben

**Funktionsweise:** Header eines Transportprotokolls besteht mind. aus

- Quell- und Zielport 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 Zielport) 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)

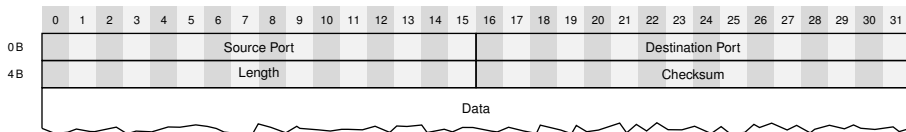
**Hinweise:**

- Verbindungslose POSIX-Sockets werden mittels des Präprozessormakros `SOCK_DGRAM` identifiziert.
- DGRAM steht dabei für **Datagram**, worunter man schlicht eine Nachricht bestimmter Länge versteht, die aus Sicht der Transportschicht als Einheit übertragen werden soll.

Das **User Datagram Protocol (UDP)** ist eines der beiden am häufigsten verwendeten Transportprotokolle im Internet. Es bietet

- ungesicherte und nachrichtenorientierte Übertragung
- bei geringem Overhead.

## UDP-Header:



- „Length“ gibt die Länge von Header und Daten in Vielfachen von Byte an.
- Die Prüfsumme erstreckt sich über Header und Daten.
  - Die Verwendung der UDP-Prüfsumme ist bei IPv4 optional, wird für IPv6 jedoch vorausgesetzt.
  - 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 der Prüfsumme dient). Er beinhaltet folgende Felder des IP-Headers: Quell- und Ziel-IP-Adresse, ein 8-Bit-Feld mit Nullen, Protocol-ID und Länge des UDP-Datagramms.

## 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:

- Keine Zusicherung irgendeiner Form von Dienstqualität (beliebig hohe Fehlerrate)
- Datagramme können out-of-order ausgeliefert werden (beispielsweise bei Verwendung mehrerer Pfade zu einem Ziel)
- Keine Flusskontrolle (schneller Sender kann langsamen Empfänger überfordern)
- Keine Staukontrollmechanismen (Überlast im Netz führt zu hohen Verlustraten)



## Wo wird UDP eingesetzt?

UDP wird überall dort eingesetzt, wo

- gelegentlicher Verlust von Datagrammen tolerierbar ist bzw. durch höhere Schichten wieder ausgeglichen wird oder
- ein zeitaufwendiger Verbindungsaufbau, wie er bei anderen Transportprotokollen benötigt wird, nicht tolerierbar ist.

## Beispiele:

- Namesauflösung mittels **Domain Name System (DNS)**
  - Mittels DNS werden „Webadressen“ wie `www.tum.de` in IP-Adressen übersetzt.
  - Bevor die Namensauflösung nicht abgeschlossen ist, kann auch keine Verbindung zum Ziel aufgebaut werden (→ Zeitverzögerung).
- Datenverkehr mit Echtzeitanforderungen
  - Mechanismen zur Fluss- und Staukontrolle können nicht-deterministische Latenzen einführen.
  - Zu spät ankommende Daten haben hier häufig keine Relevanz mehr.
- Google's **QUIC**
  - Experimentelles Protokoll zur Beschleunigung TLS 1.3 verschlüsselter Verbindungen.
  - Fehlende Mechanismen von UDP werden auf der Anwendungsschicht implementiert.

Motivation

Multiplexing

Verbindungslose Übertragung

Verbindungsorientierte Übertragung

- Sliding-Window-Verfahren

- Transmission Control Protocol (TCP)

- Fluss- und Staukontrolle bei TCP

Network Address Translation (NAT)

Codedemos

Literaturangaben

**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, ...).

**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, ...).

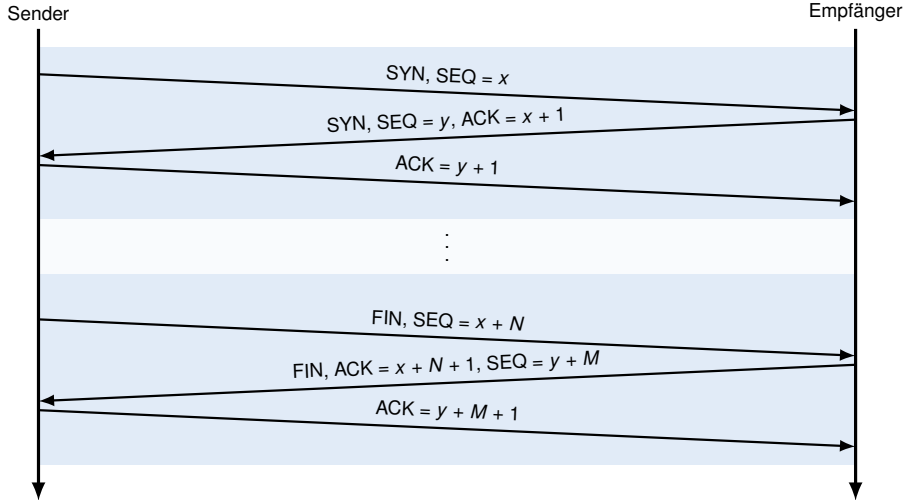
**Verbindungsphasen:**

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

**Vereinbarungen:** Wir gehen zunächst davon aus,

- dass stets ganze Segmente bestätigt werden und
- dass in einer Quittung das nächste erwartete Segment angegeben wird.

## Beispiel: Aufbau und Abbau einer Verbindung

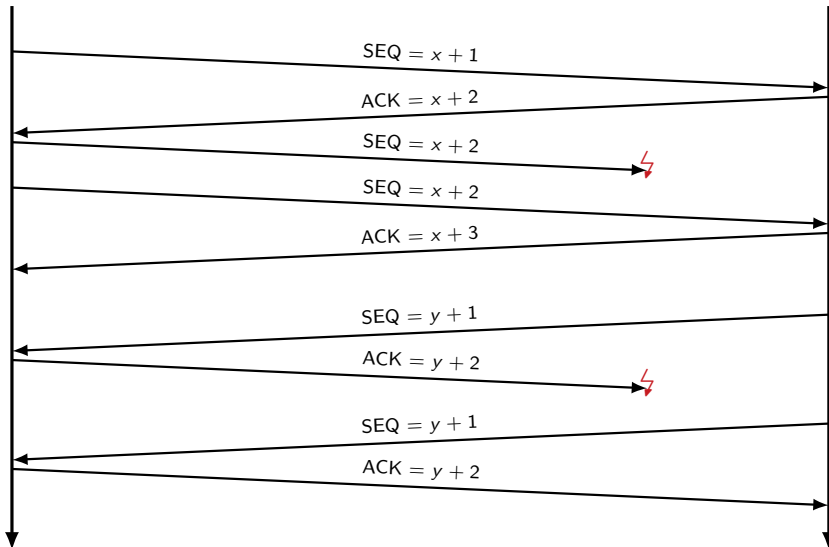


Diese Art des Verbindungsaufbaues bezeichnet man als **3-Way-Handshake**.

## Beispiel: Übertragungsphase

Sender

Empfänger



### 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 Umlaufverzögerung (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 dass der Sender auf eine Bestätigung warten muss.

### Vorteile:

- Zeit 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 an die verfügbare Datenrate auf dem Übertragungspfad zwischen Sender und Empfänger angepasst werden → [Staukontrolle](#)

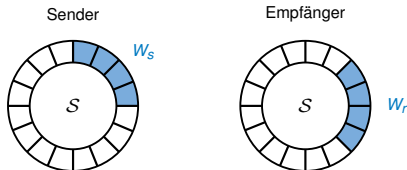
### Probleme:

- 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 bestätigten 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.






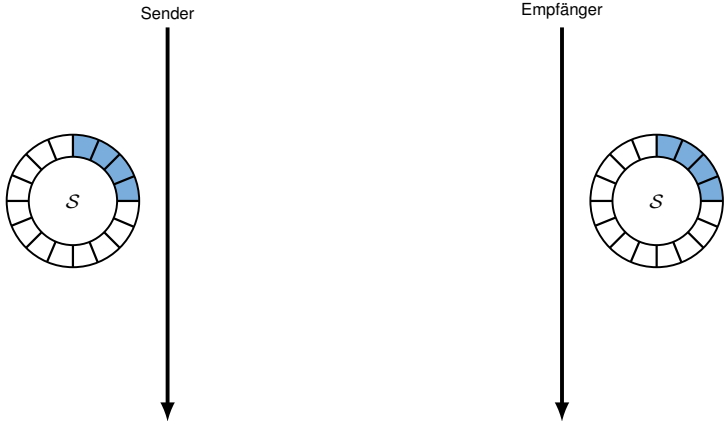
### Vereinbarungen:




- 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.

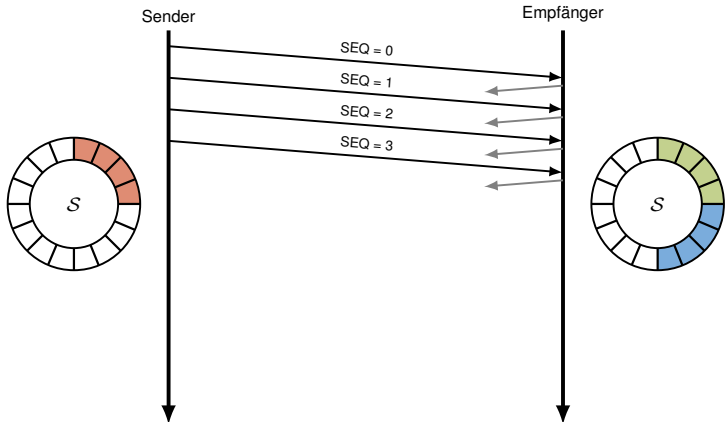
### Wichtig:

- 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.

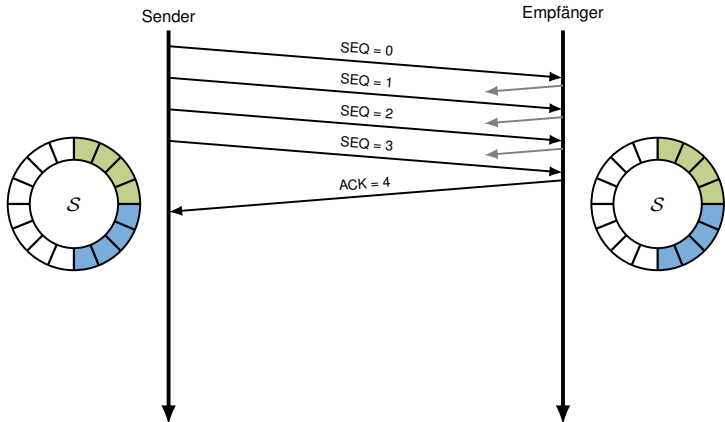
-  Sendefenster  $W_s$  bzw. Empfangsfenster  $W_r$
-  gesendet aber noch nicht bestätigt
-  gesendet und bestätigt/empfangen



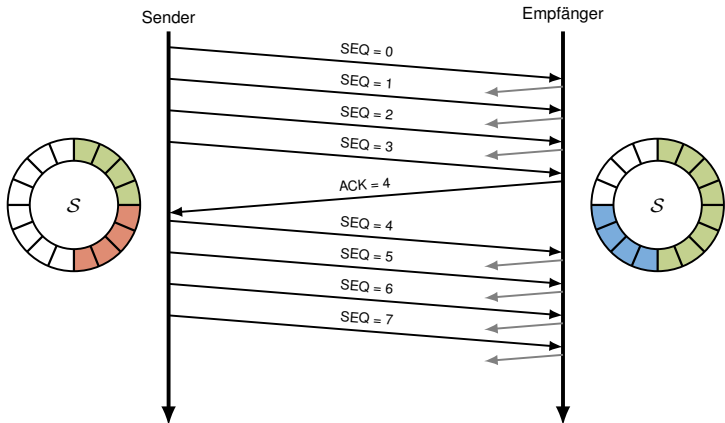
-  Sendefenster  $W_s$  bzw. Empfangsfenster  $W_r$
-  gesendet aber noch nicht bestätigt
-  gesendet und bestätigt/empfangen



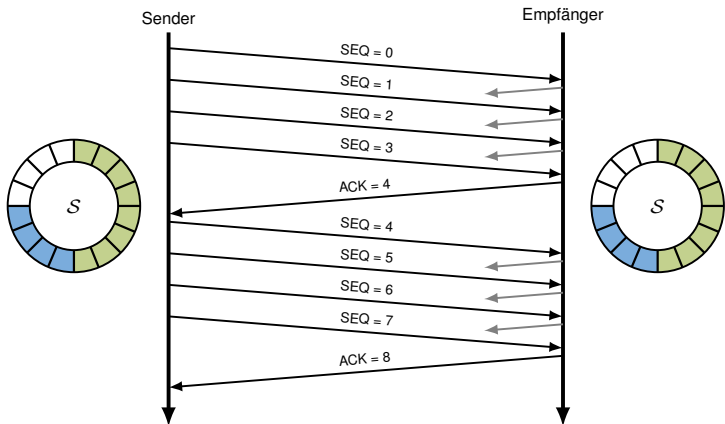
- Sendefenster  $W_s$  bzw. Empfangsfenster  $W_r$
- gesendet aber noch nicht bestätigt
- gesendet und bestätigt/empfangen



- 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 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

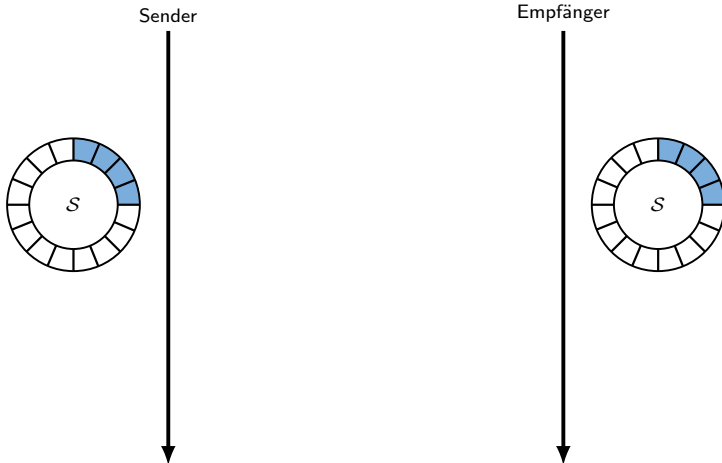
**Wichtig:**

- 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:** (siehe Übung)

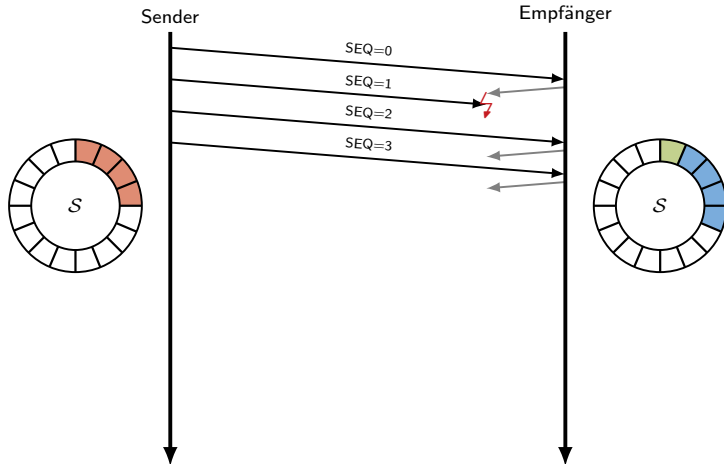
Wie groß darf das Sendefenster  $W_s$  in Abhängigkeit des Sequenznummernraums  $S$  höchstens gewählt werden, so dass die Verfahren funktionieren?

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

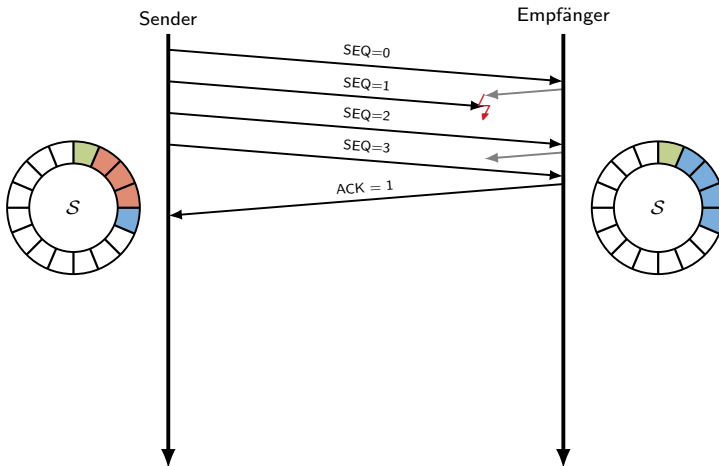




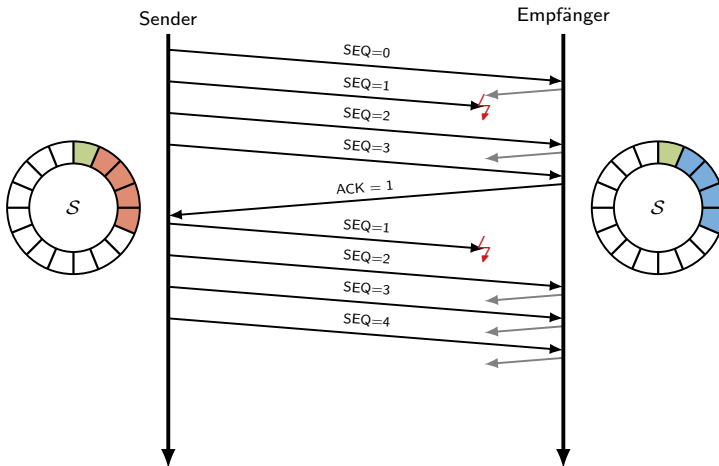
**Go-Back-N:**  $N = 16$ ,  $w_s = 4$ ,  $w_r = 4$



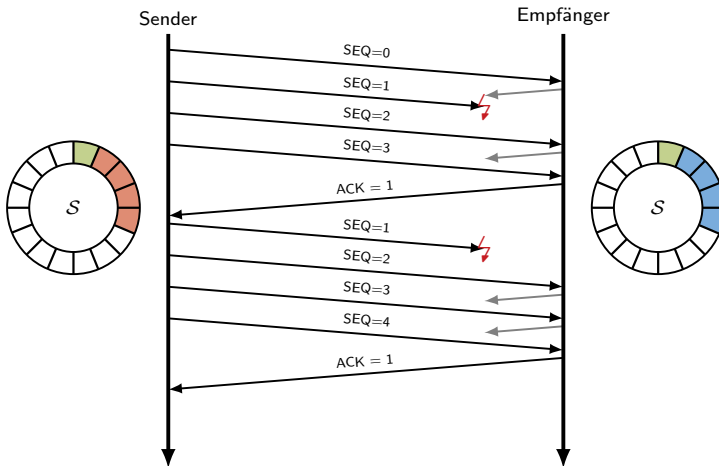
**Go-Back-N:**  $N = 16$ ,  $w_s = 4$ ,  $w_r = 4$



**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**

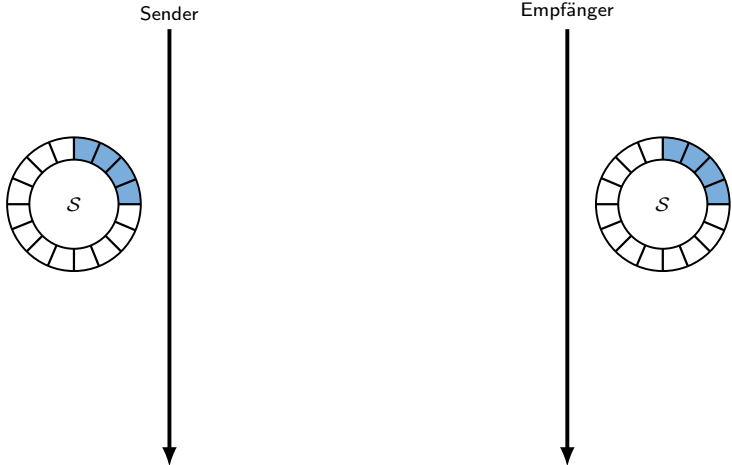
- Da der Empfänger stets nur das nächste erwartete Segment akzeptiert, reicht ein Empfangsfenster der Größe  $w_r = 1$  prinzipiell aus. Unabhängig davon muss für praktische Implementierungen ein ausreichend großer Empfangspuffer verfügbar sein.
- Bei einem Sequenznummernraum der Kardinalität  $N$  muss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

$$w_s \leq N - 1.$$

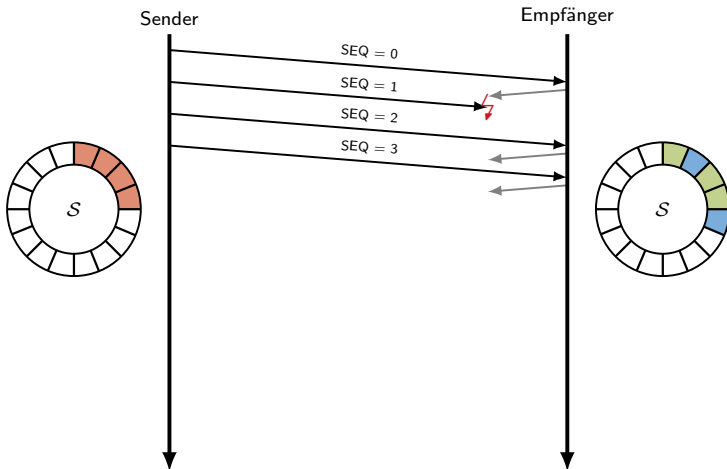
Andernfalls kann es zu Verwechslungen kommen (s. Übung).

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

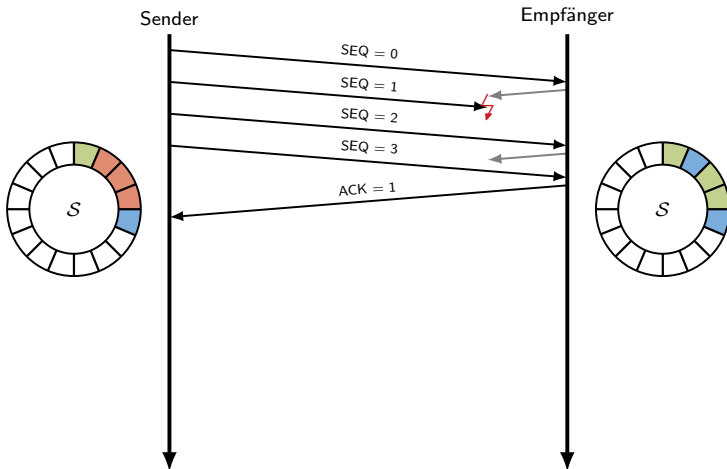
**Selective Repeat:**  $N = 16$ ,  $w_s = 4$ ,  $w_r = 4$



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

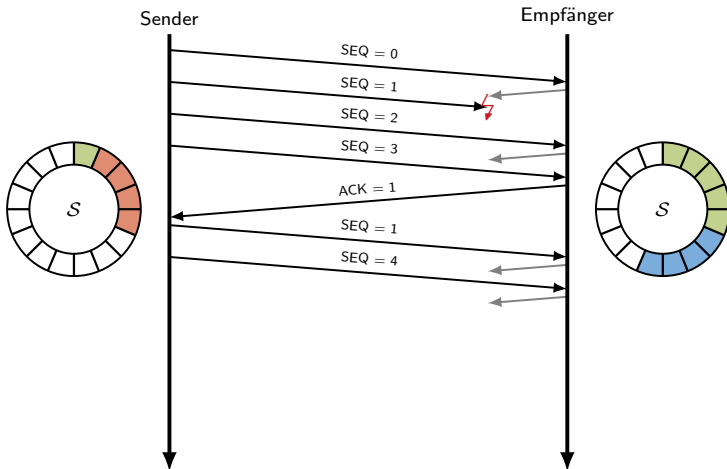


**Selective Repeat:**  $N = 16$ ,  $w_s = 4$ ,  $w_r = 4$

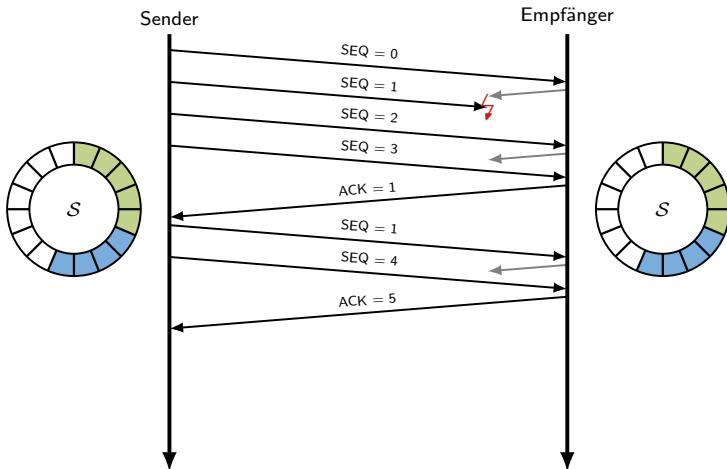




**Selective Repeat:**  $N = 16$ ,  $w_s = 4$ ,  $w_r = 4$



**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

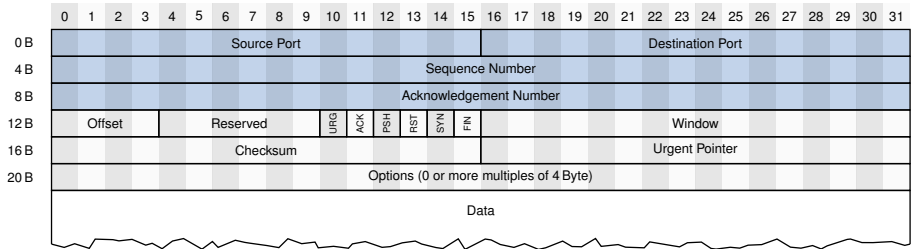
- Bei einer Umsetzung dieser Konzepte benötigt insbesondere der Empfänger einen **Empfangspuffer**, dessen Größe an die Sende- und Empfangsfenster angepasst ist.
- Für praktische Anwendungen werden die Größen von  $W_s$  und  $W_r$  dynamisch angepasst (siehe Case Study zu TCP), wodurch Algorithmen zur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

## Transmission Control Protocol (TCP)

Das **Transmission Control Protocol (TCP)** ist das dominierende Transportprotokoll im Internet (rund 90 % des Datenverkehrs im Internet [1]). Es bietet

- gesicherte/stromorientierte Übertragung mittels Sliding-Window und Selective Repeat sowie
- Mechanismen zur Fluss- und Staukontrolle.

### TCP-Header:



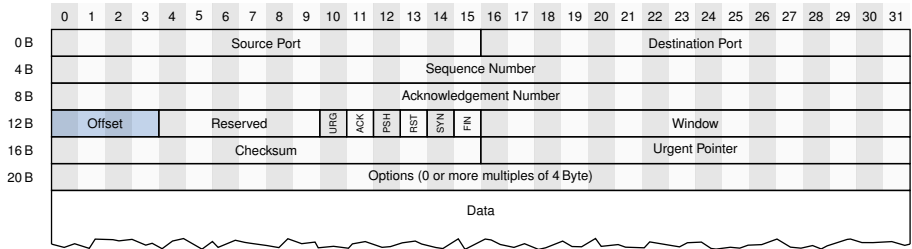
- **Quell-** und **Zielpor**t werden analog zu UDP verwendet.
- **Sequenz-** und **Bestätigungsnummer** dienen der gesicherten Übertragung. Es werden bei TCP **nicht** ganze Segmente sondern einzelne Bytes bestätigt (stromorientierte Übertragung).

## Transmission Control Protocol (TCP)

Das **Transmission Control Protocol (TCP)** ist das dominierende Transportprotokoll im Internet (rund 90 % des Datenverkehrs im Internet [1]). Es bietet

- gesicherte/stromorientierte Übertragung mittels Sliding-Window und Selective Repeat sowie
- Mechanismen zur Fluss- und Staukontrolle.

### TCP-Header:



### (Data) Offset

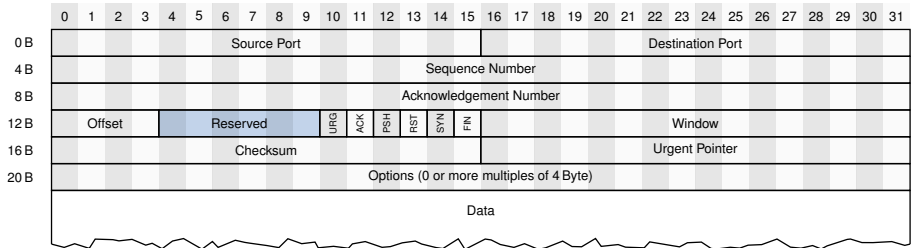
- Gibt die Länge des TCP-Headers in Vielfachen von 4 B an.
- Der TCP-Header hat variable Länge (Optionen, vgl. IPv4-Header).

## Transmission Control Protocol (TCP)

Das **Transmission Control Protocol (TCP)** ist das dominierende Transportprotokoll im Internet (rund 90 % des Datenverkehrs im Internet [1]). Es bietet

- gesicherte/stromorientierte Übertragung mittels Sliding-Window und Selective Repeat sowie
- Mechanismen zur Fluss- und Staukontrolle.

### TCP-Header:



### Reserved

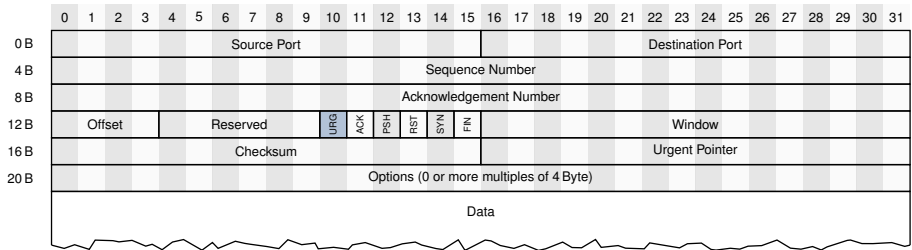
- Hat in bisherigen TCP-Versionen keine Verwendung. Muss auf 0 gesetzt werden, so dass zukünftige TCP-Versionen bei Bedarf das Feld nutzen können.

## Transmission Control Protocol (TCP)

Das **Transmission Control Protocol (TCP)** ist das dominierende Transportprotokoll im Internet (rund 90 % des Datenverkehrs im Internet [1]). Es bietet

- gesicherte/stromorientierte Übertragung mittels Sliding-Window und Selective Repeat sowie
- Mechanismen zur Fluss- und Staukontrolle.

### TCP-Header:



### Flag URG („urgent“) (selten verwendet)

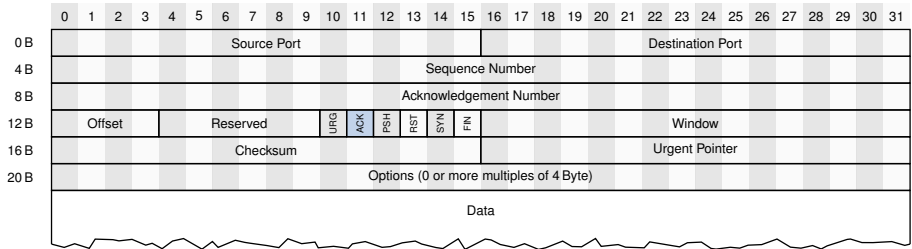
- Ist das Flag gesetzt, werden die Daten im aktuellen TCP-Segment beginnend mit dem ersten Byte bis zu der Stelle, an die das Feld **Urgent Pointer** zeigt, sofort an höhere Schichten weitergeleitet.

## Transmission Control Protocol (TCP)

Das **Transmission Control Protocol (TCP)** ist das dominierende Transportprotokoll im Internet (rund 90 % des Datenverkehrs im Internet [1]). Es bietet

- gesicherte/stromorientierte Übertragung mittels Sliding-Window und Selective Repeat sowie
- Mechanismen zur 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.

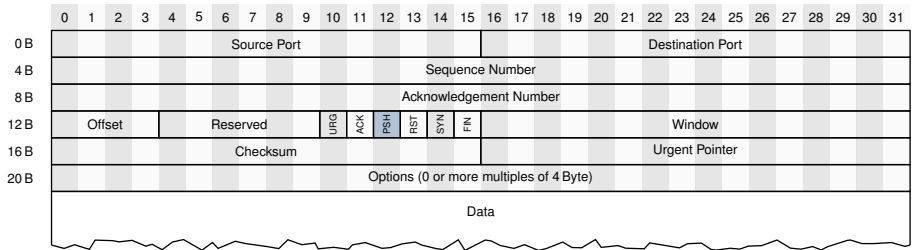


## Transmission Control Protocol (TCP)

Das **Transmission Control Protocol (TCP)** ist das dominierende Transportprotokoll im Internet (rund 90 % des Datenverkehrs im Internet [1]). Es bietet

- gesicherte/stromorientierte Übertragung mittels Sliding-Window und Selective Repeat sowie
- Mechanismen zur Fluss- und Staukontrolle.

### TCP-Header:



### Flag PSH („push“)

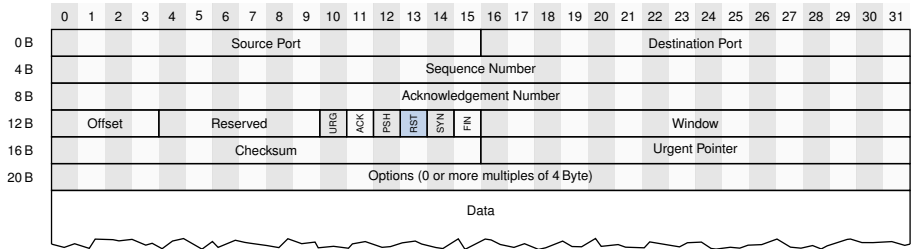
- Ist das Flag gesetzt, werden sende- und empfangsseitige Puffer des TCP-Stacks umgangen.
- Sinnvoll für interaktive Anwendungen (z. B. **Telnet**-Verbindungen).

## Transmission Control Protocol (TCP)

Das **Transmission Control Protocol (TCP)** ist das dominierende Transportprotokoll im Internet (rund 90 % des Datenverkehrs im Internet [1]). Es bietet

- gesicherte/stromorientierte Übertragung mittels Sliding-Window und Selective Repeat sowie
- Mechanismen zur Fluss- und Staukontrolle.

### TCP-Header:



### Flag RST („reset“)

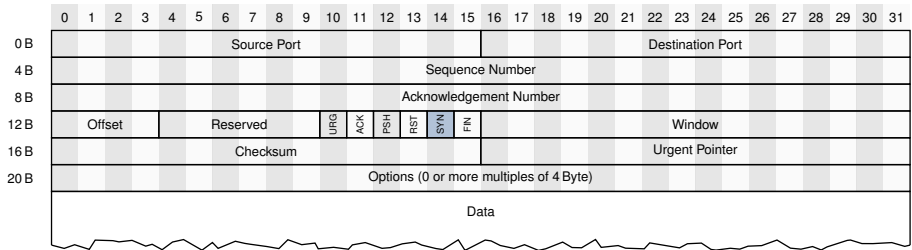
- Dient dem Abbruch einer TCP-Verbindung ohne ordnungsgemäßen Verbindungsabbau.

## Transmission Control Protocol (TCP)

Das **Transmission Control Protocol (TCP)** ist das dominierende Transportprotokoll im Internet (rund 90 % des Datenverkehrs im Internet [1]). Es bietet

- gesicherte/stromorientierte Übertragung mittels Sliding-Window und Selective Repeat sowie
- Mechanismen zur Fluss- und Staukontrolle.

### TCP-Header:



### Flag SYN („synchronization“)

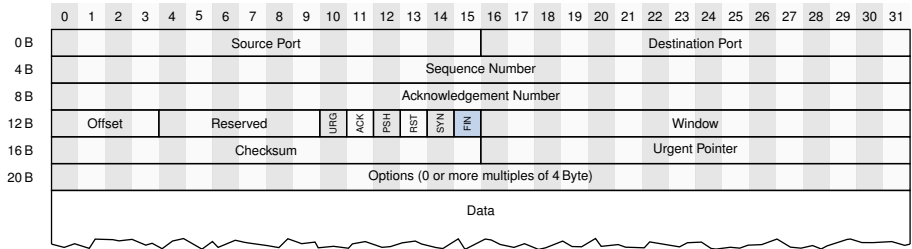
- Ist das Flag gesetzt, handelt es sich um ein Segment, welches zum Verbindungsaufbau gehört (initialer Austausch von Sequenznummern).
- Ein gesetztes SYN-Flag inkrementiert Sequenz- und Bestätigungsnummern um 1 obwohl keine Nutzdaten transportiert werden.

## Transmission Control Protocol (TCP)

Das **Transmission Control Protocol (TCP)** ist das dominierende Transportprotokoll im Internet (rund 90 % des Datenverkehrs im Internet [1]). Es bietet

- gesicherte/stromorientierte Übertragung mittels Sliding-Window und Selective Repeat sowie
- Mechanismen zur Fluss- und Staukontrolle.

### TCP-Header:



### Flag FIN („finish“)

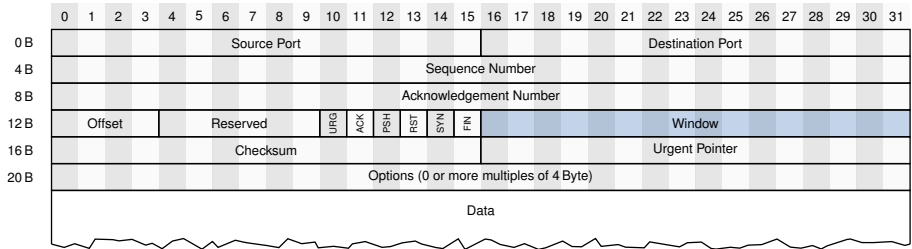
- Ist das Flag gesetzt, handelt es sich um ein Segment, welches zum Verbindungsabbau gehört.
- Ein gesetztes FIN-Flag inkrementiert Sequenz- und Bestätigungsnummern um 1 obwohl keine Nutzdaten transportiert werden.

## Transmission Control Protocol (TCP)

Das **Transmission Control Protocol (TCP)** ist das dominierende Transportprotokoll im Internet (rund 90 % des Datenverkehrs im Internet [1]). Es bietet

- gesicherte/stromorientierte Übertragung mittels Sliding-Window und Selective Repeat sowie
- Mechanismen zur Fluss- und Staukontrolle.

### TCP-Header:



### Receive Window

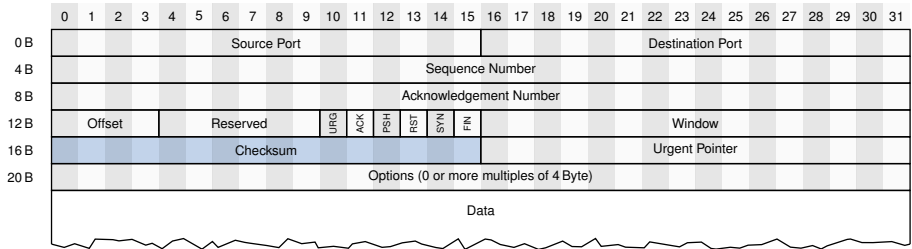
- Größe des aktuellen Empfangsfensters  $W_r$  in Byte.
- Ermöglicht es dem Empfänger, die Datenrate des Senders zu drosseln.

## Transmission Control Protocol (TCP)

Das **Transmission Control Protocol (TCP)** ist das dominierende Transportprotokoll im Internet (rund 90 % des Datenverkehrs im Internet [1]). Es bietet

- gesicherte/stromorientierte Übertragung mittels Sliding-Window und Selective Repeat sowie
- Mechanismen zur Fluss- und Staukontrolle.

### TCP-Header:



### Checksum

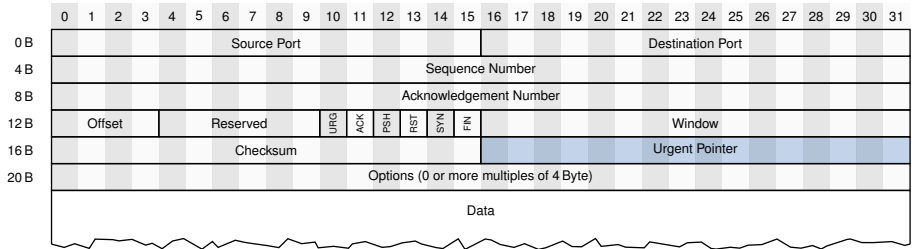
- Prüfsumme über Header und Daten.
- Wie bei UDP wird zur Berechnung ein **Pseudo-Header** verwendet.

## Transmission Control Protocol (TCP)

Das **Transmission Control Protocol (TCP)** ist das dominierende Transportprotokoll im Internet (rund 90 % des Datenverkehrs im Internet [1]). Es bietet

- gesicherte/stromorientierte Übertragung mittels Sliding-Window und Selective Repeat sowie
- Mechanismen zur Fluss- und Staukontrolle.

### TCP-Header:



### Urgent Pointer (selten verwendet)

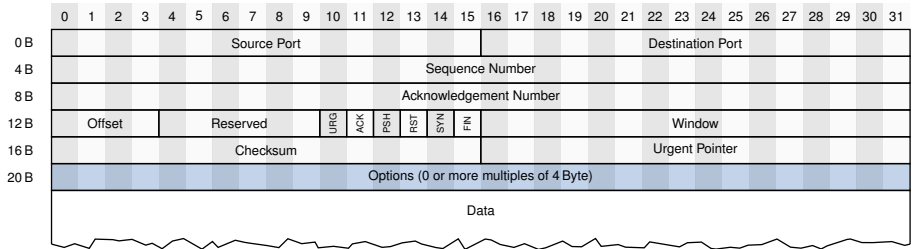
- Gibt das Ende der „Urgent-Daten“ an, welche unmittelbar nach dem Header beginnen und bei gesetztem URG-Flag sofort an höhere Schichten weitergereicht werden sollen.

## Transmission Control Protocol (TCP)

Das **Transmission Control Protocol (TCP)** ist das dominierende Transportprotokoll im Internet (rund 90 % des Datenverkehrs im Internet [1]). Es bietet

- gesicherte/stromorientierte Übertragung mittels Sliding-Window und Selective Repeat sowie
- Mechanismen zur 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<sup>1</sup>

## Beispiele:

- MSS bei FastEthernet
  - MTU beträgt 1500 B.
  - Davon entfallen 20 B auf den IPv4-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 langer 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.

<sup>1</sup> Das ist in der Praxis natürlich nicht immer möglich, da auf Schicht 4 im Allgemeinen unbekannt ist, welches Protokoll auf Schicht 3 verwendet wird, ob Optionen/Extension Header verwendet werden oder es auf noch eine zusätzliche Encapsulation zwischen Schicht 3 und Schicht 2 gibt (z.B. PPPoE bei DSL-Verbindungen).

### 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 Übertragungsrate des Senders gedrosselt werden, z. B. wenn sich der Empfangspuffer des Empfängers füllt.

### 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 Übertragungsrate 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 grundsätzlich 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  $(1/w_c)$  MSS vergrößert, d. h. nach Bestätigung eines vollständigen Staukontrollfensters um genau eine MSS.
- Ein vollständiges Fenster kann frühestens nach 1 RTT bestätigt sein.
- Dies führt zu **linearem Wachstum** des Staukontrollfensters in der RTT.

## TCP-Varianten:

- Wir betrachten hier eine auf das Wesentliche reduzierte Implementierung von TCP, die auf **TCP Reno** basiert.
- Die einzelnen TCP-Version (**Tahoe, Reno, New Reno, Cubic, ...**) unterscheiden sich in Details, sind aber alle zueinander kompatibel.
- Linux verwendet derzeit **TCP Cubic**, welches das Congestion Window schneller anwachsen lässt als andere TCP-Varianten.

Die folgende Beschreibung bezieht sich auf eine vereinfachte Implementierung von **TCP Reno**:

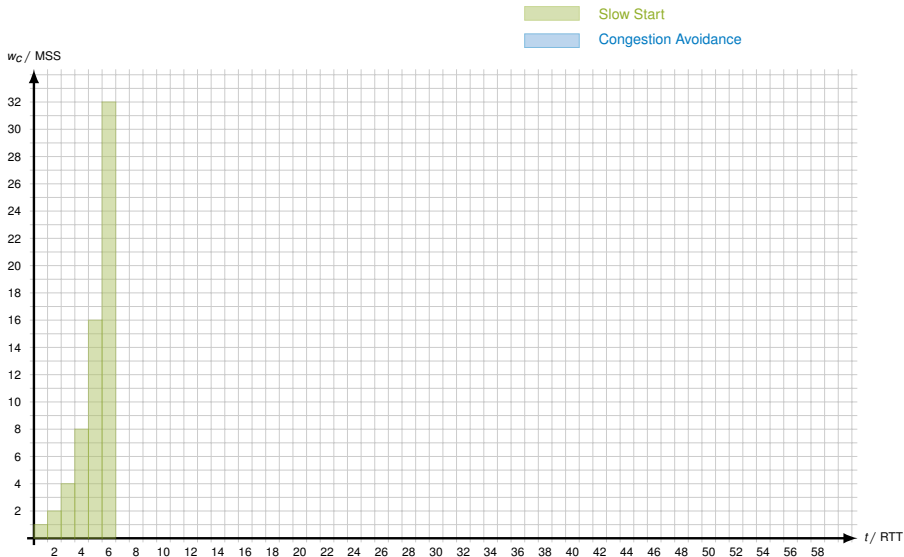
## 1. 3 duplizierte Bestätigungen (Duplicate ACKs)

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

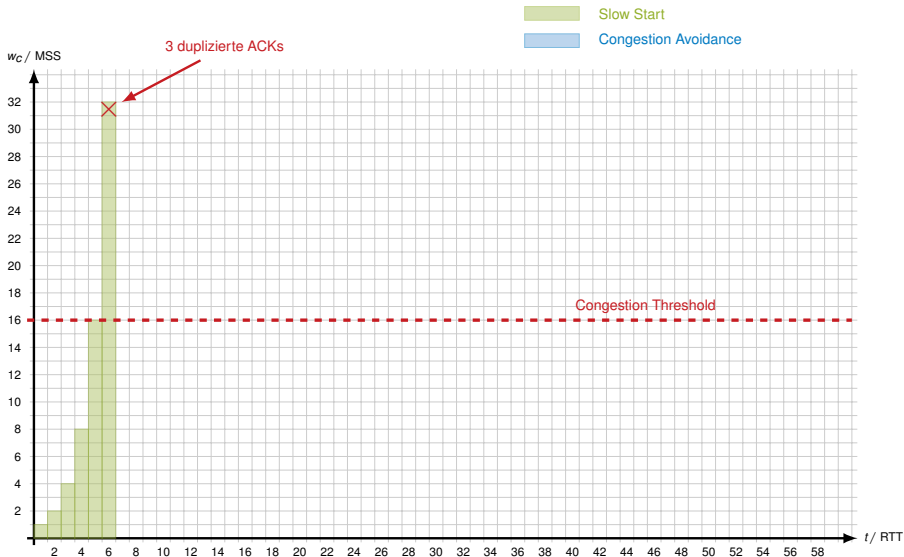
## 2. Timeout

- Setze den Schwellwert für die Stauvermeidung auf  $w_c/2$ .
  - Setze  $w_c = 1$  MSS.
  - Beginne mit einem neuen Slow-Start.
- 
- Der Vorgänger **TCP-Tahoe** unterscheidet z. B. nicht zwischen diesen beiden Fällen und führt immer Fall 2 aus.
  - Grundsätzlich sind alle TCP-Versionen kompatibel zueinander, allerdings können sich die unterschiedlichen Staukontrollverfahren gegenseitig nachteilig beeinflussen.

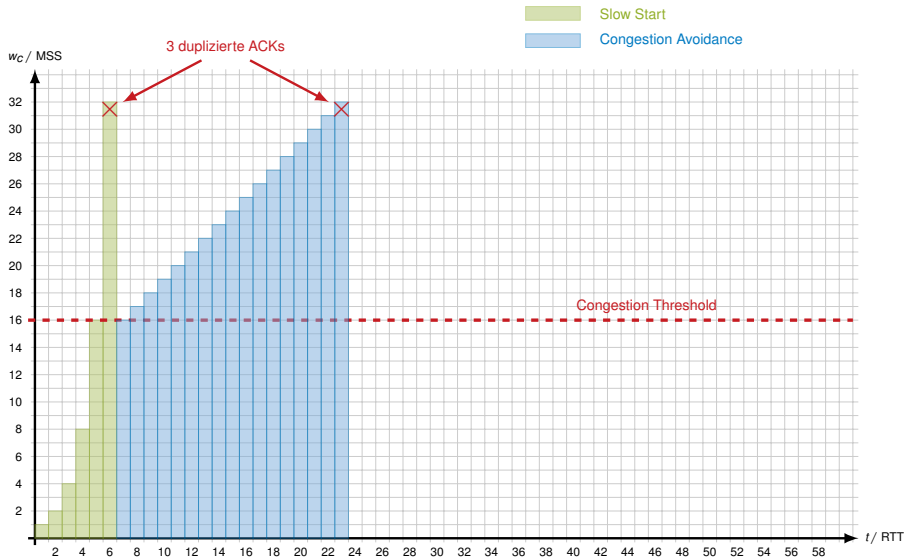
## Beispiel: TCP-Reno (mit einigen Vereinfachungen)



## Beispiel: TCP-Reno (mit einigen Vereinfachungen)

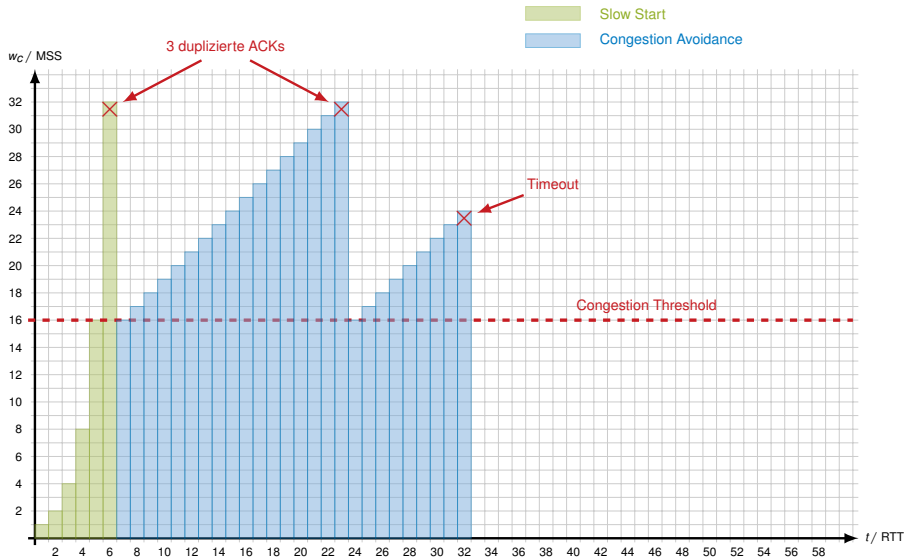


## Beispiel: TCP-Reno (mit einigen Vereinfachungen)

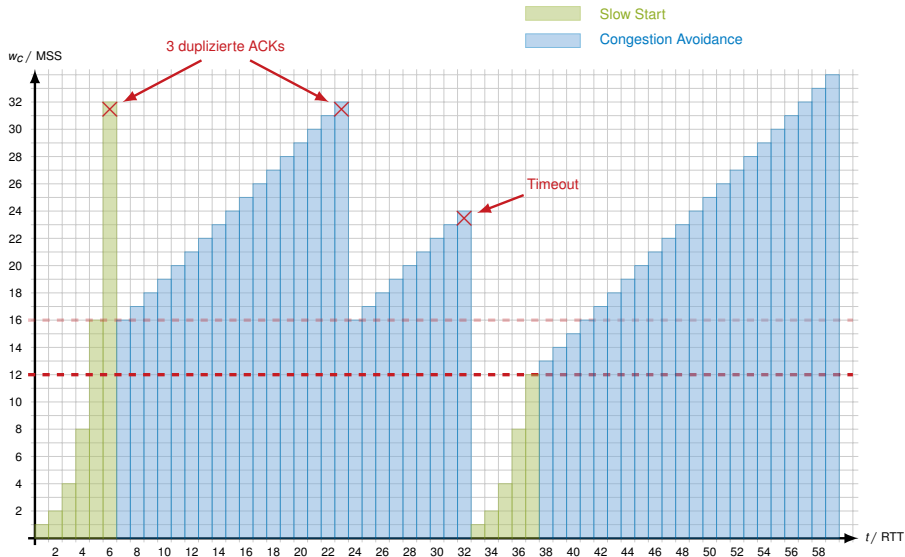




## Beispiel: TCP-Reno (mit einigen Vereinfachungen)



## Beispiel: TCP-Reno (mit einigen Vereinfachungen)



### Anmerkungen

TCP ermöglicht gesicherte Verbindungen. Die Protokollmechanismen zur Fehlerkontrolle wurden im Hinblick auf **Überlast im Netz** entwickelt.

- TCP interpretiert von Verlust von Paketen (Daten und Bestätigungen) stets als eine Folge einer **Überlastsituation** (und nicht als Folge von Bitfehlern einer unzuverlässigen Übertragung).
- In der Folge reduziert TCP die Datenrate.
- Handelt es sich bei den Paketverlusten jedoch um die Folge von Bitfehlern, 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, überfordert.

⇒ 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 bzw. anzustreben sind.
- Bei Bedarf müssen zusätzliche Bestätigungsverfahren auf Schicht 2 zum Einsatz kommen, um dies zu gewährleisten (z. B. IEEE 802.11).

Motivation

Multiplexing

Verbindungslose Übertragung

Verbindungsorientierte Übertragung

Network Address Translation (NAT)

Codedemos

Literaturangaben

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 hauptsächlich verwendeten IPv4-Adressen sehr knapp sind.

**Frage:** Müssen IP-Adressen immer **eindeutig** sein?

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 hauptsächlich verwendeten IPv4-Adressen sehr knapp sind.

**Frage:** Müssen IP-Adressen immer **eindeutig** sein?

**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.

- $N \leq M$ : Die Übersetzung geschieht statisch oder dynamisch indem jeder privaten IP-Adresse mind. eine öffentliche IP-Adresse zugeordnet wird.
- $N > M$ : 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. bei einem privaten DSL-Anschluss.

## Was sind private IP-Adressen?

**Private IP-Adressen** sind spezielle Adressbereiche, welche

- zur privaten Nutzung ohne vorherige Registrierung freigegeben sind,
- deswegen in unterschiedlichen Netzen vorkommen können,
- aus diesem Grund nicht eindeutig und 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 sind:

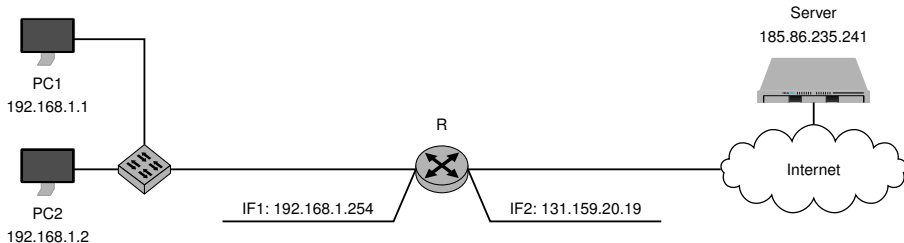
- 10.0.0.0/8
- 172.16.0.0/12
- 169.254.0.0/16
- 192.168.0.0/16

Der Bereich 169.254.0.0/16 wird 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?

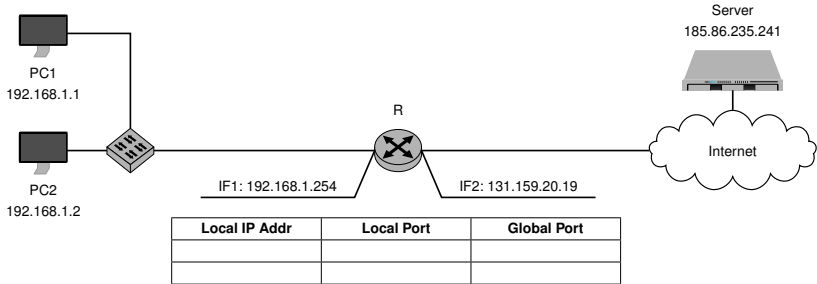
Üblicherweise übernehmen Router die Netzwerkadressübersetzung:



- 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.

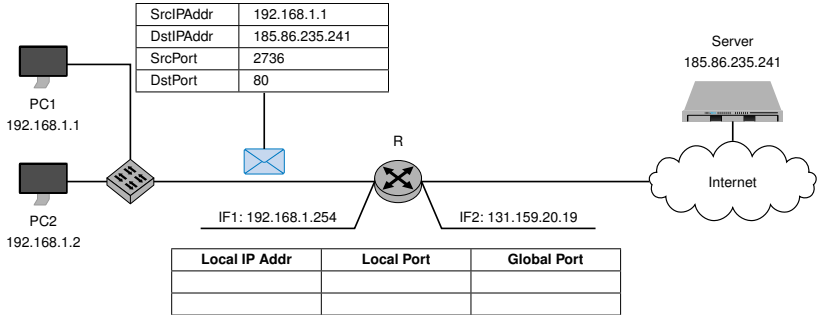


PC1 greift auf eine Webseite zu, welche auf dem Server mit der IP-Adresse 185.86.235.241 liegt:



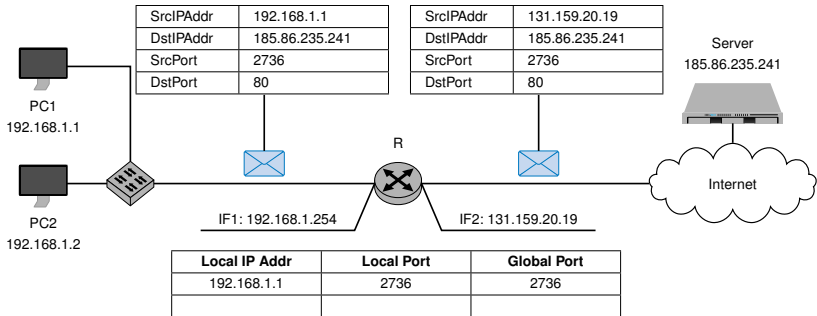
- Die NAT-Tabelle von R sei zu Beginn leer.

PC1 greift auf eine Webseite zu, welche auf dem Server mit der IP-Adresse 185.86.235.241 liegt:



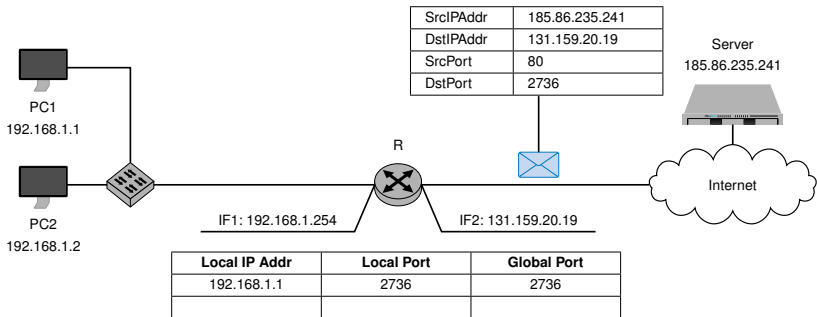
- Die NAT-Tabelle von R sei zu Beginn leer.
- PC1 sendet ein Paket (TCP SYN) an den Server:
  - PC1 verwendet seine private IP-Adresse als Absenderadresse
  - Der Quellport wird von PC1 zufällig im Bereich [1024,65535] gewählt (sog. [Ephemeral Ports](#))
  - Der Zielport ist durch das Application Layer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

PC1 greift auf eine Webseite zu, welche auf dem Server mit der IP-Adresse 185.86.235.241 liegt:



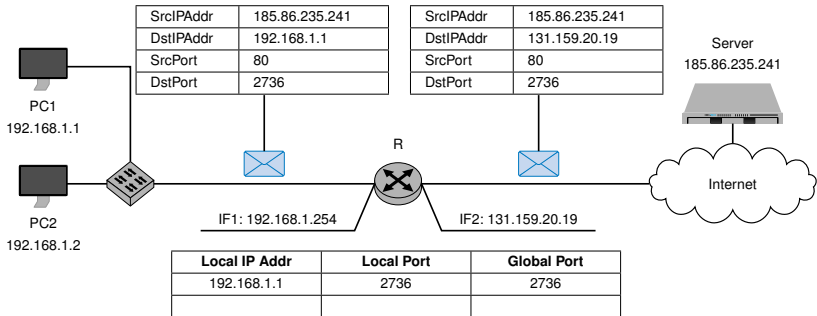
- Die NAT-Tabelle von R sei zu Beginn leer.
- PC1 sendet ein Paket (TCP SYN) an den Server:
  - PC1 verwendet seine private IP-Adresse als Absenderadresse
  - Der Quellport wird von PC1 zufällig im Bereich [1024,65535] gewählt (sog. [Ephemeral Ports](#))
  - Der Zielport ist durch das Application Layer 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 führt, wird dieser beibehalten
  - R erzeugt einen neuen Eintrag in seiner NAT-Tabelle, welche die Änderungen an dem Paket dokumentieren

Antwort vom Server an PC1



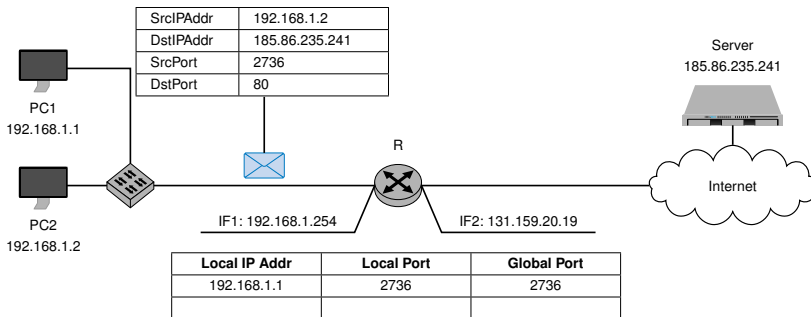
- Der Server generiert eine Antwort:
  - Der Server weiß nichts von der Adressübersetzung und hält R für PC1.
  - Die Empfängeradresse ist daher die öffentliche IP-Adresse von R, der Zielpport der von R übersetzte Quellport aus der vorherigen Nachricht.

## Antwort vom Server an PC1



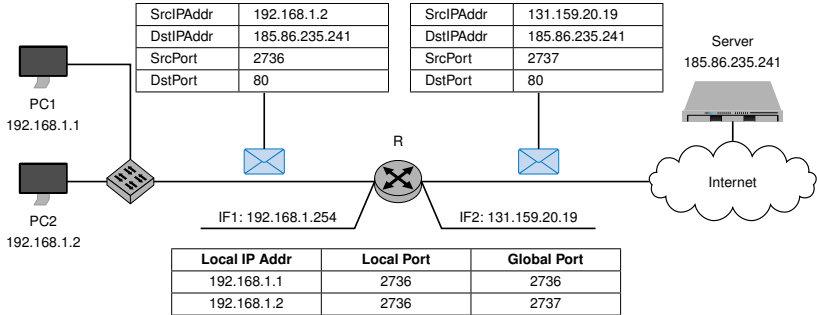
- Der Server generiert eine Antwort:
  - Der Server weiß nichts von der Adressübersetzung und hält R für PC1.
  - Die Empfängeradresse ist daher die öffentliche IP-Adresse von R, der Zielport der von R übersetzte Quellport aus der vorherigen Nachricht.
- R macht die Adressübersetzung rückgängig:
  - In der NAT-Tabelle wird nach der Zielportnummer in der Spalte Global Port gesucht, dieser in Local Port zurückübersetzt und die Ziel-IP-Adresse 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-Adresse und Ziel-Port
- Das verwendete Protokoll (TCP, UDP)
- Die eigene globale IP-Adresse (sinnvoll, wenn ein Router mehr als eine globale IP-Adresse 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-Adresse 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 (siehe „Masterkurs Rechnernetze“):

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



## Allgemeine Anmerkungen

- Ist NAT eine Firewall [2]<sup>1</sup>?
  - **Nein!**
  - Restriktive NAT-Varianten bieten zwar insofern einen grundlegenden Schutz, da sie eingehende Verbindungen ohne vorherigen Verbindungsaufbau aus dem lokalen Netz heraus **nicht** erlauben, dies sollte aber nicht mit den Funktionen einer Firewall verwechselt werden.
  - Eine darüber hinausgehende Filterung (wie es bei einer Firewall der Fall wäre) findet nicht statt.

---

<sup>1</sup> [Firewalls](#) erlauben es, eingehenden und ausgehenden Datenverkehr anhand von IP-Adressen, Portnummern sowie vorherigen Ereignissen („Stateful Firewalls“) zu filtern. Diese Funktionen werden häufig von Routern übernommen. In manchen Fällen wird auch der Inhalt einzelner Nachrichten überprüft („Deep Packet Inspection“).

## Allgemeine Anmerkungen

- Ist NAT eine Firewall [2]<sup>1</sup>?
  - **Nein!**
  - Restriktive NAT-Varianten bieten zwar insofern einen grundlegenden Schutz, da sie eingehende Verbindungen ohne vorherigen Verbindungsaufbau aus dem lokalen Netz heraus **nicht** erlauben, dies sollte aber nicht mit den Funktionen einer Firewall verwechselt werden.
  - Eine darüber hinausgehende Filterung (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).

---

<sup>1</sup> [Firewalls](#) erlauben es, eingehenden und ausgehenden Datenverkehr anhand von IP-Adressen, Portnummern sowie vorherigen Ereignissen („Stateful Firewalls“) zu filtern. Diese Funktionen werden häufig von Routern übernommen. In manchen Fällen wird auch der Inhalt einzelner Nachrichten überprüft („Deep Packet Inspection“).

## Allgemeine Anmerkungen

- Ist NAT eine Firewall [2]<sup>1</sup>?
  - **Nein!**
  - Restriktive NAT-Varianten bieten zwar insofern einen grundlegenden Schutz, da sie eingehende Verbindungen ohne vorherigen Verbindungsaufbau aus dem lokalen Netz heraus **nicht** erlauben, dies sollte aber nicht mit den Funktionen einer Firewall verwechselt werden.
  - Eine darüber hinausgehende Filterung (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).

---

<sup>1</sup> Firewalls erlauben es, eingehenden und ausgehenden Datenverkehr anhand von IP-Adressen, Portnummern sowie vorherigen Ereignissen („Stateful Firewalls“) zu filtern. Diese Funktionen werden häufig von Routern übernommen. In manchen Fällen wird auch der Inhalt einzelner Nachrichten überprüft („Deep Packet Inspection“).

## Allgemeine Anmerkungen

- Ist NAT eine Firewall [2]<sup>1</sup>?
  - **Nein!**
  - Restriktive NAT-Varianten bieten zwar insofern einen grundlegenden Schutz, da sie eingehende Verbindungen ohne vorherigen Verbindungsaufbau aus dem lokalen Netz heraus **nicht** erlauben, dies sollte aber nicht mit den Funktionen einer Firewall verwechselt werden.
  - Eine darüber hinausgehende Filterung (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.

<sup>1</sup> **Firewalls** erlauben es, eingehenden und ausgehenden Datenverkehr anhand von IP-Adressen, Portnummern sowie vorherigen Ereignissen („Stateful Firewalls“) zu filtern. Diese Funktionen werden häufig von Routern übernommen. In manchen Fällen wird auch der Inhalt einzelner Nachrichten überprüft („Deep Packet Inspection“).

## 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.

**Problem:** Traceroute funktioniert mit manchen Virtualisierungslösungen nicht, z. B. wenn ältere Versionen der Virtualbox-NAT-Implementierung verwendet werden.

- Traceroute basiert auf ICMP-TTL-Exceeded-Nachrichten.
- Diese Nachrichten haben (anders als ein ICMP Echo Reply) keine ICMP-ID.
- Das liegt daran, dass jedes beliebige IP-Paket (nicht zwangsläufig ein ICMP-Echo-Request) ein ICMP-TTL-Exceeded auslösen kann und dieses (wie im Fall eines TCP-Pakets) natürlich keine ICMP-ID besitzt.
- Stattdessen trägt der 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 im Fall eines TTL-Exceeded in diesen ersten 8 Byte nach der ICMP-ID eines Echo-Requests oder aber nach den Portnummern eines Transportprotokolls suchen, um die Übersetzung rückgängig machen zu können.
- Genau diese Rückübersetzung führen ältere Versionen der NAT-Implementierung von Virtualbox nicht durch.

## NAT und IPv6

- NAT kann auch für IPv6 verwendet werden.



## NAT und IPv6

- NAT kann auch für IPv6 verwendet werden.

### Präfix-Übersetzung

- NAT für IPv6 besitzt spezifische Probleme und Herausforderungen. RFC 6296 spezifiziert IPv6-to-IPv6 Network Prefix Translation.
- Dabei wird ein 1:1-Mapping von Adressen erzeugt.
  - Dies wäre auch bei IPv4 im begrenzten Umfang möglich (sofern einer Organisation ausreichend öffentliche Adressen zur Verfügung stehen),
  - aber meist wenig sinnvoll.
- Damit können **Unique-Local Unicast-Adressen** (fc00::/7, also **private** IPv6-Adressen) in global gültige Adressen übersetzt werden.
- Die Übersetzung erfolgt auf Präfixen:
  - Ein interne Präfix fd01:0203:0405::/48 wird z. B. auf das globale Präfix 2001:db8:0001::/48 abgebildet.
- Dabei werden keine Layer-4-Merkmale (Ports, Identifier) verwendet.
- Die Übersetzung erfolgt, abgesehen von der Konfiguration der Adresspräfixe, zustandslos. Es wird keine NAT-Tabelle benötigt.
- Um zu verhindern, dass die Layer 4 Checksums aufgrund der Adressübersetzung modifiziert werden müssen, kann die Adressübersetzung so gewählt werden, dass die ursprüngliche Prüfsummen weiterhin stimmen.

## NAT und IPv6

- NAT kann auch für IPv6 verwendet werden.

### Präfix-Übersetzung

- NAT für IPv6 besitzt spezifische Probleme und Herausforderungen. RFC 6296 spezifiziert IPv6-to-IPv6 Network Prefix Translation.
- Dabei wird ein 1:1-Mapping von Adressen erzeugt.
  - Dies wäre auch bei IPv4 im begrenzten Umfang möglich (sofern einer Organisation ausreichend öffentliche Adressen zur Verfügung stehen),
  - aber meist wenig sinnvoll.
- Damit können **Unique-Local Unicast-Adressen** (fc00::/7, also **private** IPv6-Adressen) in global gültige Adressen übersetzt werden.
- Die Übersetzung erfolgt auf Präfixen:
  - Ein interne Präfix fd01:0203:0405::/48 wird z. B. auf das globale Präfix 2001:db8:0001::/48 abgebildet.
- Dabei werden keine Layer-4-Merkmale (Ports, Identifier) verwendet.
- Die Übersetzung erfolgt, abgesehen von der Konfiguration der Adresspräfixe, zustandslos. Es wird keine NAT-Tabelle benötigt.
- Um zu verhindern, dass die Layer 4 Checksums aufgrund der Adressübersetzung modifiziert werden müssen, kann die Adressübersetzung so gewählt werden, dass die ursprüngliche Prüfsummen weiterhin stimmen.

### Einsatz von NAT bei IPv6

- Ein häufiger Grund für NAT (die Adressknappheit bei IPv4) ist bei IPv6 aber nicht gegeben.

Motivation

Multiplexing

Verbindungslose Übertragung

Verbindungsorientierte Übertragung

Network Address Translation (NAT)

**Codedemos**

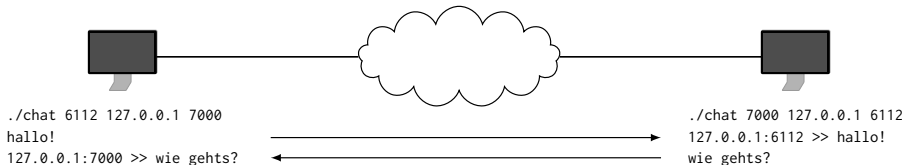
UDP-Chat

Case Study: TCP-Relay-Chat

Literaturangaben

**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 (;

**Download**

- <https://grnvs.net/codedemos/udpchat> (oder <https://grnvs.net/codedemos/udpchat.tar.xz>)
- make

## Wichtige structs

- struct sockaddr\_in: Enthält Adressfamilie (AF\_INET oder AF\_INET6), Portnummer und IP-Adresse

```
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: Repräsentiert eine IPv4-Adresse in Network Byte Order

```
struct in_addr {
    __be32  s_addr;
};
```

- struct sockaddr: Adress-Struktur, die den Typ des Sockets offen lässt (generalisiert sockaddr\_in und sockaddr\_un)

```
struct sockaddr {
    sa_family_t    sa_family; /* address family, AF_xxx */
    char           sa_data[14]; /* 14 bytes of protocol address */
};
```

## Code für unser Programm:

Initialisieren der sockaddr structs

```
struct sockaddr_in local;  
local.sin_family    = AF_INET;           // Adressfamilie  
local.sin_port      = htons(port);       // anwendungsspezifischer Port  
local.sin_addr.s_addr = INADDR_ANY;      // empfang von allen Adressen
```

- AF\_INET spezifiziert IPv4 (Alternativen sind z. B. AF\_INET6 und AF\_UNIX)
- Die 16 bit lange Portnummer, auf der eingehende Pakete erwartet werden, muss in Network Byte Order angegeben werden (daher htons() wie „host to network short“)
- INADDR\_ANY entspricht der unspezifizierten IPv4 Adresse 0.0.0.0, d. h. Daten werden auf allen IP-Adressen bzw. Interfaces erwartet

## Sockets

- Aus Sicht des Betriebssystems ist ein Socket nichts weiter als ein [Filedescriptor](#), d. h. ein Integer.
- Sockets stellen die Schnittstelle zwischen einem Programm (unserer Chatanwendung) und dem Betriebssystem dar.

### Ein Socket für unser Programm:

- `socket()` erzeugt einen neuen Socket vom angegebenen Typ:
  - `AF_INET` spezifiziert einen IPv4 Socket.
  - `SOCK_DGRAM` gibt an, dass es ein datagram-oriented Socket sein soll.
  - `IPPROTO_UDP` gibt das Transportprotokoll an.
- Rückgabewert ist der Socketdescriptor oder -1 bei einem Fehler.

```
int sd;
if (0 > (sd=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP))) {
    perror("socket() failed");
    exit(1);
}
```

### Der Socket muss noch eine Adresse bekommen:

- `bind()` assoziiert einen Filedescriptor mit den zugehörigen Adressinformationen.
- Der Cast in `struct sockaddr` ist deswegen notwendig, da `bind()` nicht nur mit `struct sockaddr_in` zurechtkommt.
- Rückgabewert ist 0 bei Erfolg und -1 bei Fehlern.

```
if (0 > bind(sd, (struct sockaddr *)&local, sizeof(local))) {
    perror("bind() failed");
    exit(1);
}
```

## Was geht mit so einem Socket?

- Der so erstellte Socket kann zum Empfangen aber auch zum Senden von Daten verwendet werden.
- Beim Senden wird die im struct `sockaddr` angegebene Portnummer als Quellport verwendet.
- Die Absender-IP entscheidet sich auf Basis des zum Senden verwendeten Interfaces, falls mittels `INADDR_ANY` beliebige Adressen spezifiziert wurden.

## Wie merkt unser Programm, wenn neue Daten ankommen?

Hier gibt es 3 Möglichkeiten:

- Einfach ein `read()` auf dem Socket:
  - `read()` blockiert, solange bis etwas kommt.
  - Mit einem einzelnen Prozess bzw. Thread können wir so nur einen einzigen Socket überwachen.
  - Unser Programm könnte nicht einmal Daten empfangen und (mehr oder weniger) gleichzeitig auf Tastatureingaben reagieren.
- Nutzung von `select()` oder `pselect()`:
  - Wir packen alle Filedescriptors, die überwacht werden sollen, in ein `fd_set`.
  - Wir übergeben `select()` dieses Set.
  - Sobald etwas passiert, modifiziert `select()` das übergebene `fd_set`, so dass es genau die Filedescriptors enthält, die bereit geworden sind.
  - Rückgabewert von `select()` ist die Anzahl bereitgewordener Filedescriptors oder -1 bei einem Fehler.
- Bei einer großen Anzahl von Filedescriptors wird `select()` ggf. ineffizient. Hier bietet sich dann `epoll()` an (hier nicht weiter 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);

for (;;) {
    rfd = rfd;
    if (0 > select(maxfd+1, &rfd, NULL, NULL, NULL)) {
        perror("select() failed");
        exit(1);
    }
    (...)
}
```

- rfd ist das „read file descriptor set“, welches alle Filedeskriptoren beinhaltet, die select() überwachen soll.
- maxfd ist der numerisch größte Filedeskriptor, den select() überwachen soll.
- select() selbst gibt als Rückgabewert die Anzahl der bereit gewordenen Filedeskriptoren zurück (oder -1 bei einem Fehler).
- Das übergebene Deskriptorset wird dabei modifiziert, so dass es nur noch die aktiven Filedeskriptoren enthält (weswegen rfd zu Beginn der Schleife immer wieder hergestellt werden muss).

## Empfangen von Daten

- Sobald etwas Interessantes passiert, wird `select()` uns das sagen.
- Wir müssen feststellen, welcher der Filedeskriptoren bereit ist.
- Im Fall der Standardeingabe (STDIN) können wir mit `fgets()` einfach die Eingabe lesen.
- Wenn der Filedeskriptor des Sockets bereit ist, könnten wir `read()` oder `recv()` verwenden. Dann werden wir aber bei verbindungslosen Transportprotokollen wie UDP nie erfahren, wer uns etwas 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:

```
for (;;) {
    (...)
    if (FD_ISSET(sd,&rfd)) {
        len = recvfrom(sd,buffer,BUFFLEN-1,0,(struct sockaddr*)&from, &slen);
        if (0 > len)
            perror("recvfrom() failed");
            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 mit verbindungslosen Protokollen `sendto()` nutzen.
- Diesem muss 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:

```
for (;;) {
    (...)
    if (FD_ISSET(STDIN_FILENO, &rfd)) {
        if (NULL == (s=fgets(buffer, BUFFLEN, stdin)))
            continue;

        len = sendto(sd, buffer, strlen(buffer), 0,
                     (struct sockaddr *)&remote, sizeof(remote));

        if (0 > len)
            perror("sendto() failed");
            exit(1);
    }
}
(...)
```

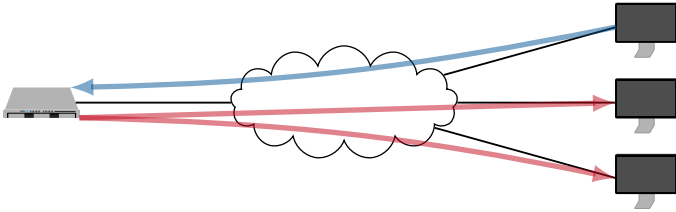
*Für alle, die mit der Indentation der Codebeispiele unglücklich sind, sei der [Linux kernel coding style](#) Pflichtlektüre!*

## Und jetzt?

1. Zuerst probieren wir jetzt den schon zum Download verfügbaren UDP-Chat aus:
  - Er ermöglicht nur 1:1 Chats.
  - Fremde können (sofern Zieladresse und -port bekannt sind), einem der beiden Chatpartner ebenfalls Nachrichten Senden.
  - Eine Antwort an den „Störer“ wird aber nicht möglich sein.
2. Danach werden wir den 1:1 Chat zu einem UDP-basierten Internet Relay Chat erweitern:
  - Es sollen dann N:N Chats möglich sein.
  - Das modifizierte Programm soll als Server dienen und mit den unmodifizierten UDP-Chats zusammenarbeiten.

### 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 (;

#### Download:

- [https://grnvs.net/codedemos/tcpchat\\_client](https://grnvs.net/codedemos/tcpchat_client)  
(oder [https://grnvs.net/codedemos/tcpchat\\_client.tar.xz](https://grnvs.net/codedemos/tcpchat_client.tar.xz))
- `make`

**Der Client** sieht ähnlich aus, wie unser UDP-Chat:

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

if (0 > (sd=socket(AF_INET, SOCK_STREAM, IPPROTO_TCP))) {
    perror("socket() failed");
    exit(1);
}

if (0 > connect(sd, (struct sockaddr *)&remote, sizeof(remote))) {
    perror("connect() failed");
    exit(1);
}
```

- Socket-Typ ist jetzt SOCK\_STREAM und das Protokoll IPPROTO\_TCP.
- 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:

```
(...)
for(;;) {
    rfd = rfd;

    if (0 > select(maxfd+1,&rfd,NULL,NULL,NULL)) {
        perror("select() failed");
        exit(1);
    }

    if (FD_ISSET(STDIN_FILENO,&rfd)) {
        if (NULL == (s = fgets(buffer, sizeof(buffer)-1, stdin)))
            continue;
        if (0 >= (len=send(sd,buffer,MIN(strlen(buffer)+1,BUFFLEN),0))) {
            perror("send() failed");
            exit(-1);
        }
    }
    if (FD_ISSET(sd,&rfd)) {
        if (0 >= (len=recv(sd,buffer,BUFFLEN-1,0))) { (...) }
        fprintf(stdout,">> %s\n",buffer);
    }
}
```

Da Sender und Empfänger bekannt sind, reicht ein `send()` bzw. `recv()` statt `sendto()` oder `recvfrom()`.

### Der Server

```
if (0 > (sd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP))) {  
    perror("socket() failed");  
    exit(-1);  
}  
  
local.sin_family      = AF_INET;  
local.sin_port        = htons(port);  
local.sin_addr.s_addr = INADDR_ANY;  
  
if (0 > bind(sd, (struct sockaddr *)&local, sizeof(local))) {  
    perror("bind() failed");  
    exit(-1);  
}  
  
if (0 > listen(sd, MAXCLIENTS)) {  
    perror("listen() failed");  
    exit(-1);  
}
```

Neu ist der Aufruf von `listen()`, welcher den Socket als **passiv** markiert. Dies bedeutet, dass über diesen Socket **keine** Daten versendet oder empfangen werden sondern stattdessen eingehende Verbindungen auf diesem Socket erwartet werden.



## Case Study: TCP-Relay-Chat

Verbindet sich ein Client, muss die Verbindung akzeptiert werden:

```
for (;;) {
    rfd = rfd;

    if (0 > select(maxfd+1, &rfd, NULL, NULL, NULL)) {
        perror("select() failed");
        exit(-1);
    }

    if (FD_ISSET(sd, &rfd)) {
        if (0 > (csd = accept(sd, (struct sockaddr *)&safrom, &slen))) {
            perror("accept() failed");
            exit(-1);
        }
        cl_add(&cl, csd, safrom); // 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. Hat man den richtigen Socket gefunden, kann man mittels `recv()` und `send()` empfangen und senden:

```
for (;;) {  
    (...)   
    len = recv(client.sd, inbuff, BUFFLEN, 0);  
    (...)   
    len = send(client.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.

Mit diesen Codefragmenten lässt sich relativ schnell

- ein einfacher TCP-basierter Relay-Chat implementieren, der
- im Funktionsumfang dem gestern in der Vorlesung implementierten UDP-basierten Relay-Chat entspricht.

### Frage:

- Welche Schwächen / Probleme des gestrigen Chats, werden wir nicht mehr haben?
- Welche neuen Probleme werden wir jetzt bekommen?

Auf <https://grnvs.net/codedemos> steht eine Serverimplementierung des TCP-Chats zur Verfügung, der interessante weitere Funktionen wie

- Flood-Protection (Spammer werden rausgeworfen) und
- zeitbegrenztes Banning von Clients basierend auf deren IP-Adresse unterstützt.

### **Ist das relevant für die Klausur? JA!**

Zwar werden wir nicht „auf Papier programmieren“, Sie sollten aber wissen

- worin die Unterschiede der Transportprotokolle bestehen,
- welche Syscalls (und in welcher Reihenfolge) zum öffnen der entsprechenden Ports bzw. Verbindungen notwendig sind,
- was diese Syscalls tun,
- worin die Probleme unserer beiden Beispielprogramme bestehen und
- wie diese prinzipiell anzugehen sind.

Insbesondere die in den Vorlesungen aufgetretenen Eigenheiten beider Programme (die eine Folge der jeweils verwendeten Protokolle darstellen) sind relevant.

Motivation

Multiplexing

Verbindungslose Übertragung

Verbindungsorientierte Übertragung

Network Address Translation (NAT)

Codedemos

Literaturangaben

- [1] Analyzing UDP Usage in Internet Traffic, 2009.  
<http://www.caida.org/research/traffic-analysis/tcpudpratio/>.
- [2] L. Zhang.  
A Retrospective View of NAT.  
*IETF Journal*, 3(2), 2007.  
<http://www.internetsociety.org/articles/retrospective-view-nat>.