Grundlagen Rechnernetze und Verteilte Systeme (GRNVS)

IN0010 – SoSe 2025

Prof. Dr.-Ing. Georg Carle

Manuel Simon, Markus Sosnowski, Stefan Lachnit, Lorenz Lehle, Christian Dietze, Prof. Dr.-Ing. Stephan Günther

Lehrstuhl für Netzarchitekturen und Netzdienste
School for Computation, Information and Technology
Technische Universität München

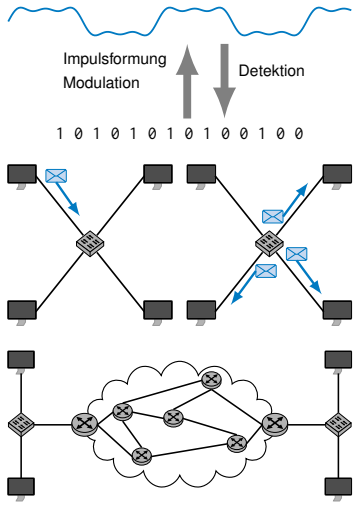
Kapitel 4: Transportschicht

Kapitel 4: Transportschicht

Motivation

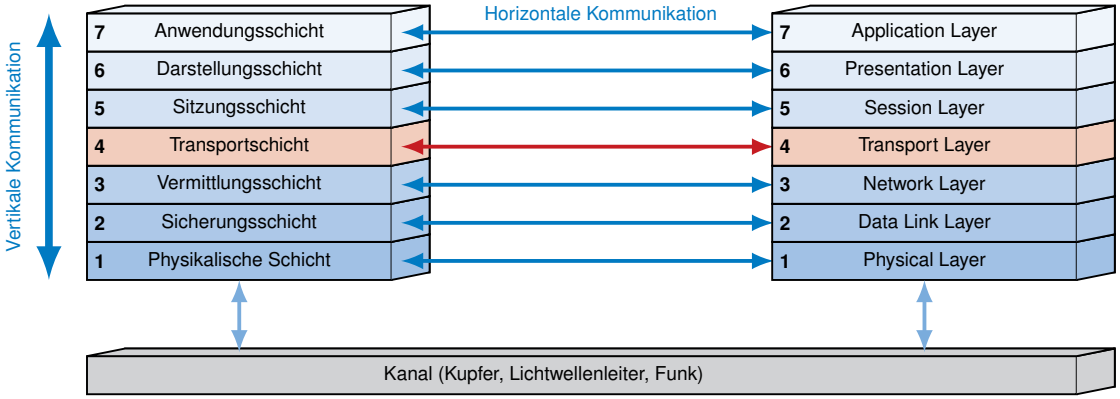
Wir haben bislang gesehen:

- 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)



Motivation

Einordnung im ISO/OSI-Modell



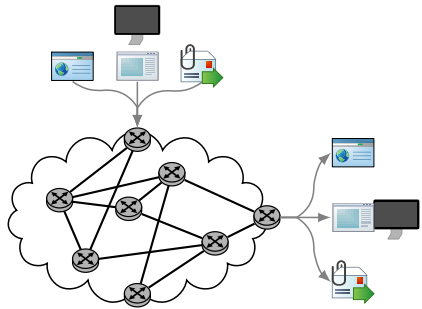
Aufgaben der Transportschicht

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



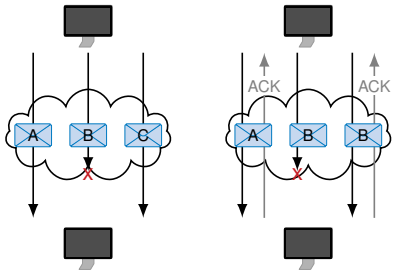
Aufgaben der Transportschicht

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



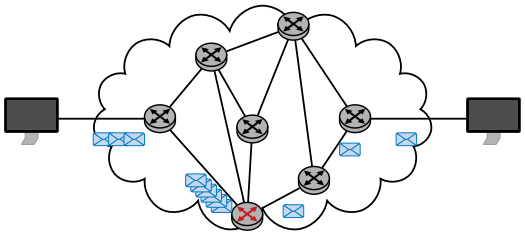
Aufgaben der Transportschicht

Die wesentlichen Aufgaben der Transportschicht sind

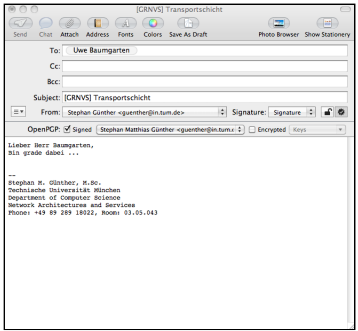
- Multiplexing von Datenströmen unterschiedlicher Anwendungen bzw. Anwendungsinstanzen,
- Bereitstellung verbindungsloser und verbindungsorientierter Transportmechanismen und
- Mechanismen zur Stau- und Flusskontrolle.

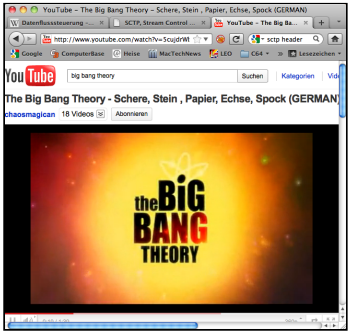
Stau- und Flusskontrolle:

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



Kapitel 4: Transportschicht





OSI Schicht 7 – 5

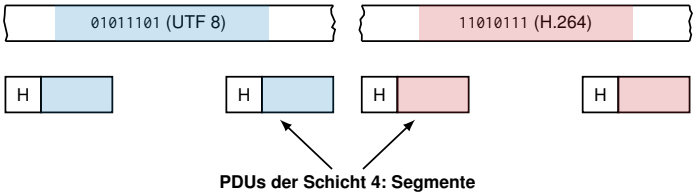
01011101 (UTF 8)	11010111 (H.264)

Transportschicht

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

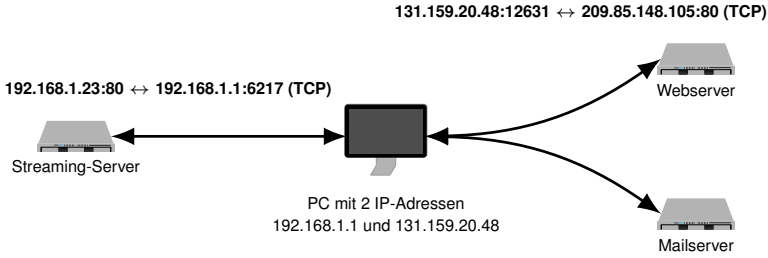
- einen Quellport und
- einen Zielport,

welche zusammen mit den Absender- bzw. Empfänger-IP und dem verwendeten Transportprotokoll die Anwendung auf am Sender bzw. Empfänger jeweils 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-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()`).

Kapitel 4: Transportschicht

Verbindungslose Übertragung

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 nachrichtenorienterten Kommunikation. (Nicht zu verwechseln mit nachrichtenorientierter Übertragung auf Schicht 2)

Hinweise:

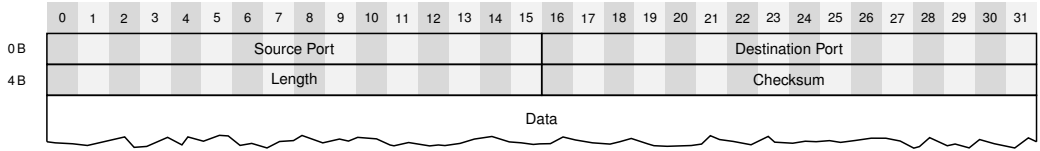
- Verbindungslose POSIX-Sockets werden mittels des Präprozessormakros `SOCK_DGRAM` identifiziert.
- `DGRAM` steht dabei für Datagram, worunter man schlicht eine Nachricht bestimmter Länge versteht, die aus Sicht der Transportschicht als Einheit übertragen werden soll.

User Datagram Protocol (UDP)

Das User Datagram Protcol (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 langes Feld mit Nullen, Protocol-ID und Länge des UDP-Datagramms.

User Datagram Protocol (UDP)

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 ($\rightarrow$ 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
  - Protokoll zur Beschleunigung TLS 1.3 verschlüsselter Verbindungen.
  - Fehlende Mechanismen von UDP werden auf der Anwendungsschicht implementiert.
  - Als Folge müssen Anwendung QUIC direkt implementieren (oder User-Space Libraries verwenden). Es gibt keine direkte Implementierung von QUIC im Kernel / Betriebssystem.
  - Insofern ist es nicht richtig, bei QUIC von einem Transportprotokoll im Sinne des ISO / OSI-Modells zu sprechen.

Kapitel 4: Transportschicht

Verbindungsorientierte Übertragung

Grundlegende Idee: Linear durchnummeriere 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 durchnummeriere 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:

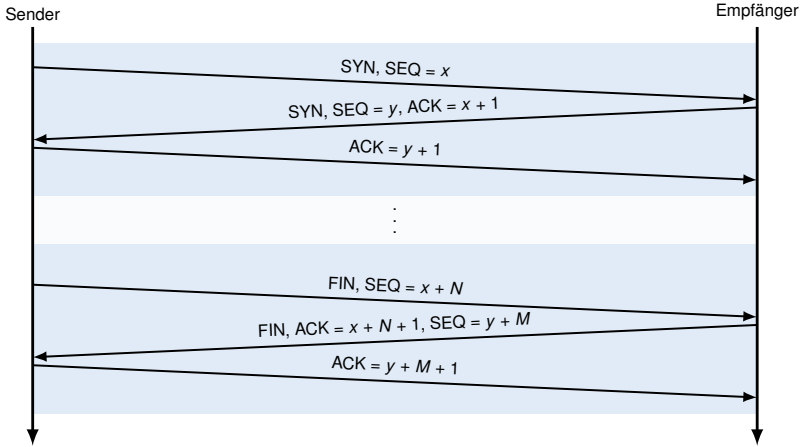
1. Verbindungsaufbau (Handshake)
2. Datenübertragung
3. Verbindungsabbau (Teardown)

Vereinbarungen: Wir gehen zunächst davon aus,

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

Verbindungsorientierte Übertragung

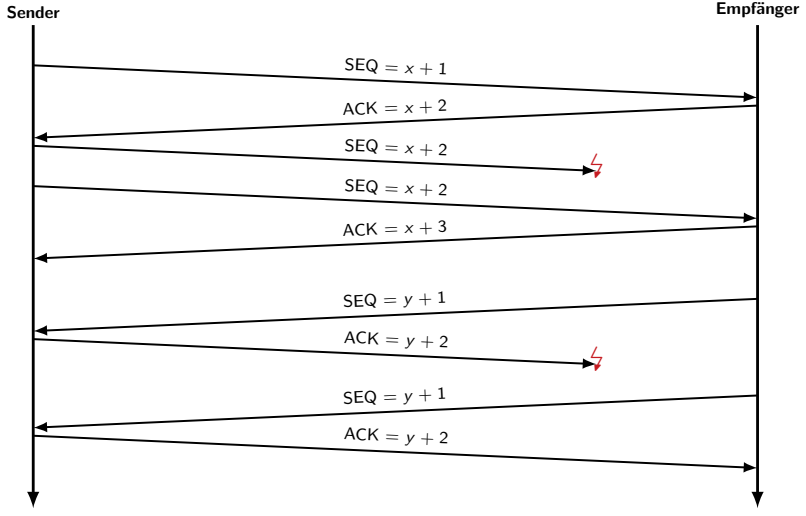
Beispiel: Aufbau und Abbau einer Verbindung



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

Verbindungsorientierte Übertragung

Beispiel: Übertragungsphase



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

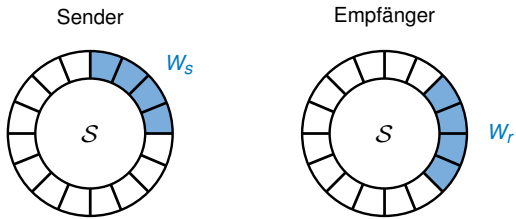
Probleme:

- 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 $\mathcal{S} = \{0,1,2,\ldots,N-1\}$.

  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.

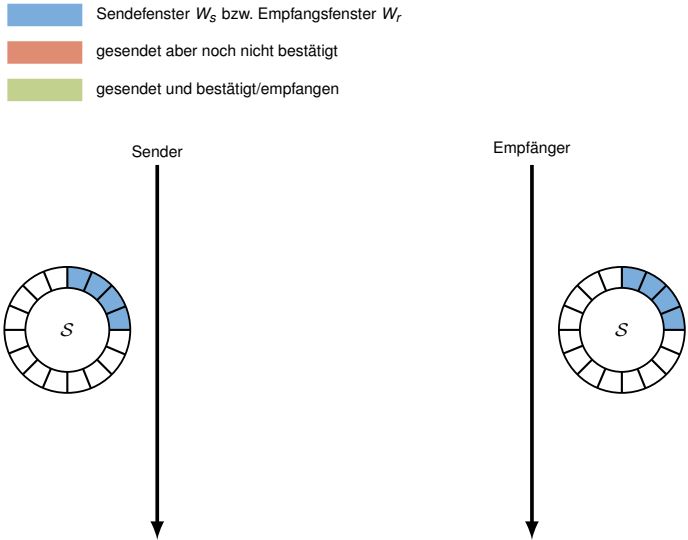
Vereinbarungen:

- 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.

Sliding-Window-Verfahren

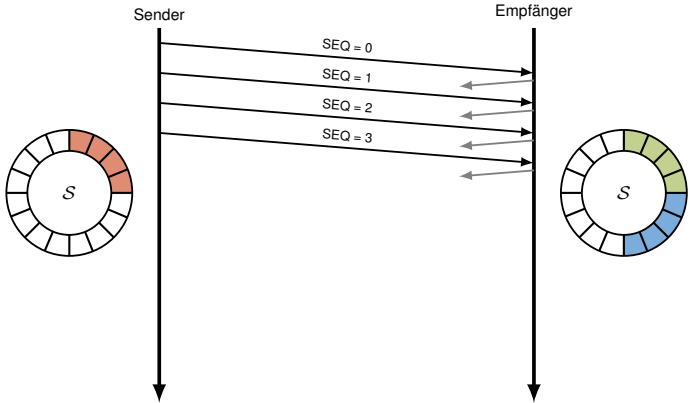


Sliding-Window-Verfahren

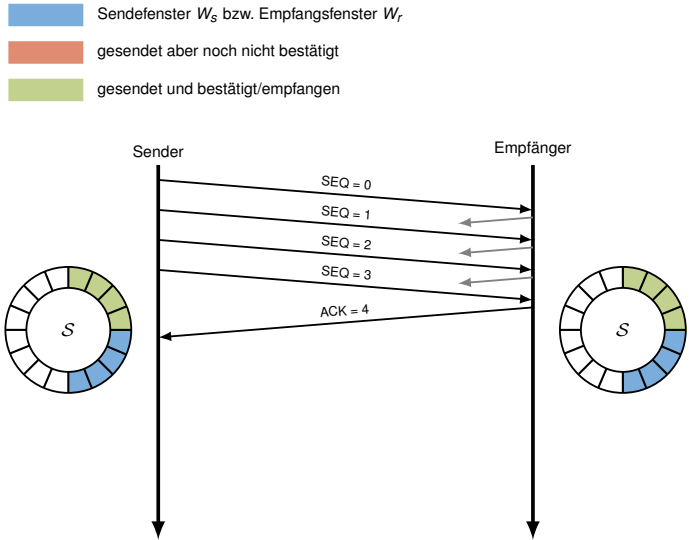
Sendefenster W_s bzw. Empfangsfenster W_r

gesendet aber noch nicht bestätigt

gesendet und bestätigt/empfangen



Sliding-Window-Verfahren

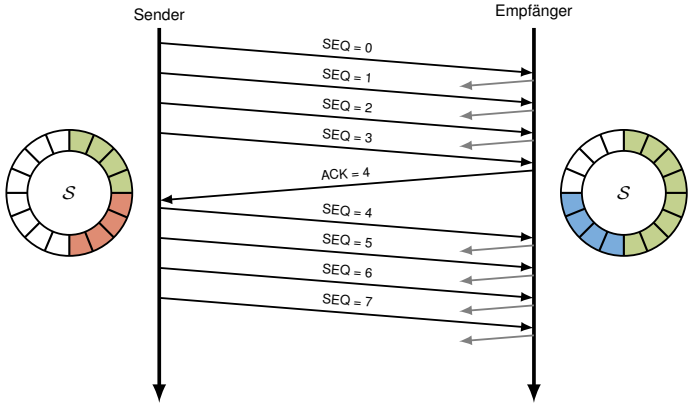


Sliding-Window-Verfahren

Sendefenster W_s bzw. Empfangsfenster W_r

gesendet aber noch nicht bestätigt

gesendet und bestätigt/empfangen

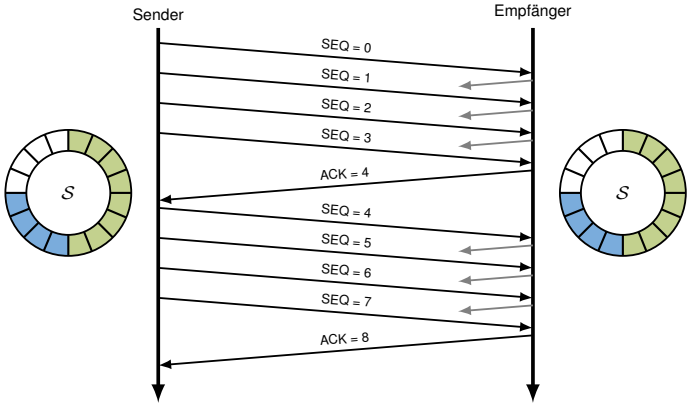


Sliding-Window-Verfahren

Sendefenster W_s bzw. Empfangsfenster W_r

gesendet aber noch nicht bestätigt

gesendet und bestätigt/empfangen



Sliding-Window-Verfahren

Neues Problem: Wie wird mit Segmentverlusten umgegangen?

Zwei Möglichkeiten:

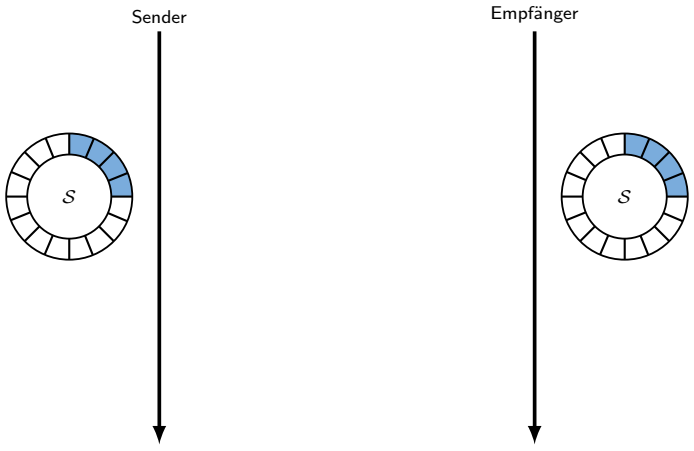
1. Go-Back-N
   - Akzeptiere stets nur die nächste erwartete Sequenznummer
   - Alle anderen Segmente werden verworfen
2. Selective-Repeat
   - Akzeptiere alle Sequenznummern, die in das aktuelle Empfangsfenster fallen
   - Diese müssen gepuffert werden, bis fehlende Segmente erneut übertragen wurden

Wichtig:

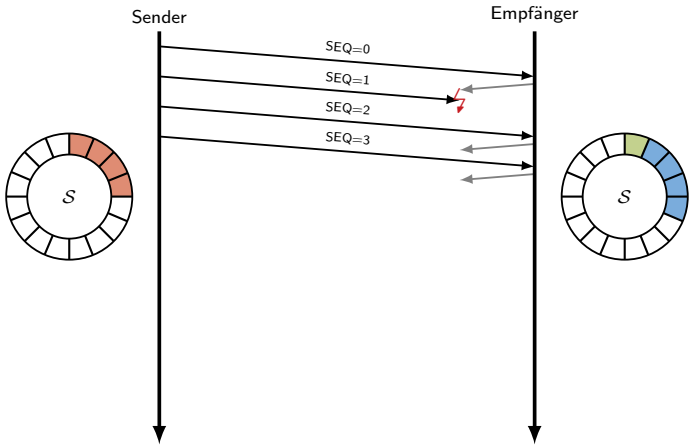
- In beiden Fällen muss der Sequenznummernraum so gewählt werden, dass wiederholte Segmente eindeutig von neuen Segmenten unterschieden werden können.
- Andernfalls würde es zu Verwechslungen kommen
  $\rightarrow$ Auslieferung von Duplikaten an höhere Schichten, keine korrekte Reihenfolge.

Frage: Wie groß darf das Sendefenster W_s in Abhängigkeit des Sequenznummernraums $\mathcal{S}$ höchstens gewählt werden, so dass die Verfahren funktionieren? (siehe Übung)

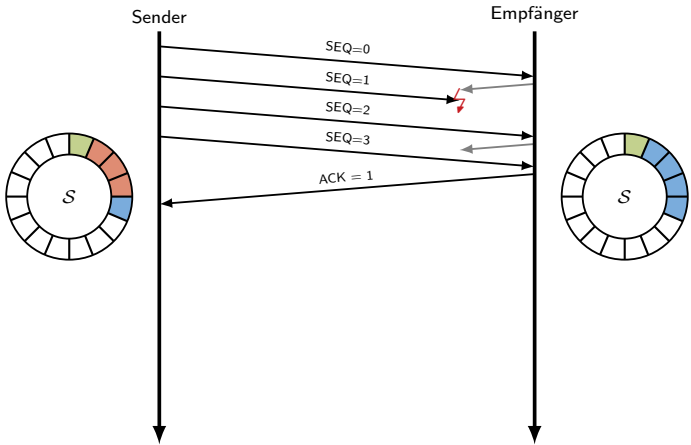
Go-Back-N: $N = 16$, $w_s = 4$, $w_r = 4$



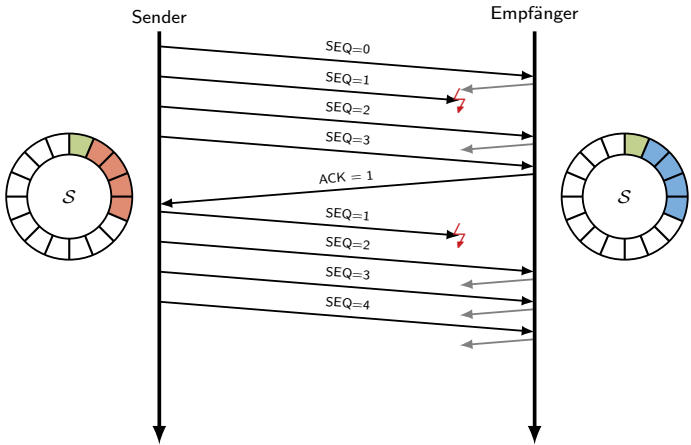
Go-Back-N: $N = 16$, $w_s = 4$, $w_r = 4$



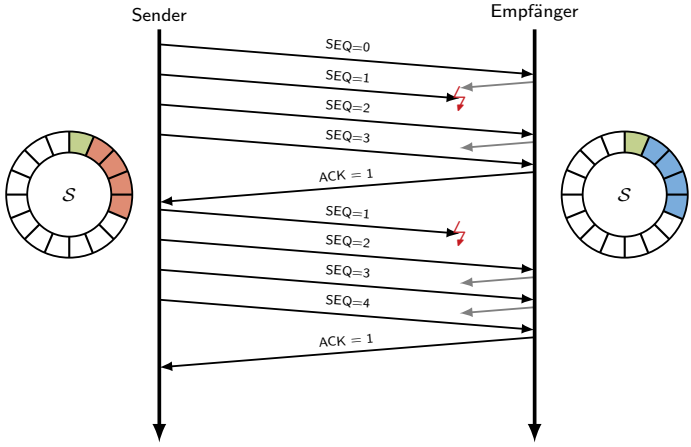
Go-Back-N: $N = 16$, $w_s = 4$, $w_r = 4$



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



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



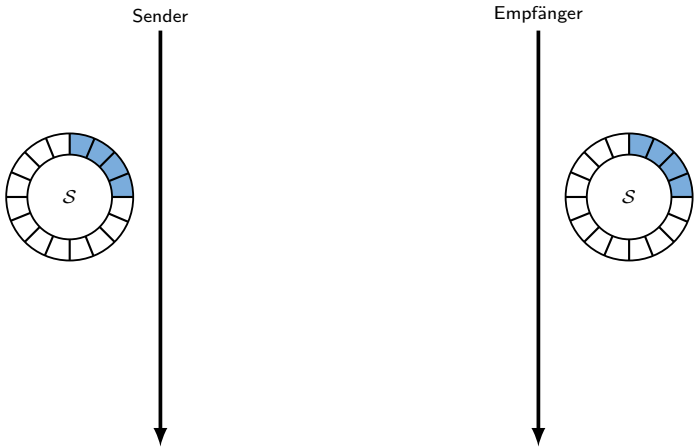
Anmerkungen zu Go-Back-N

- 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:

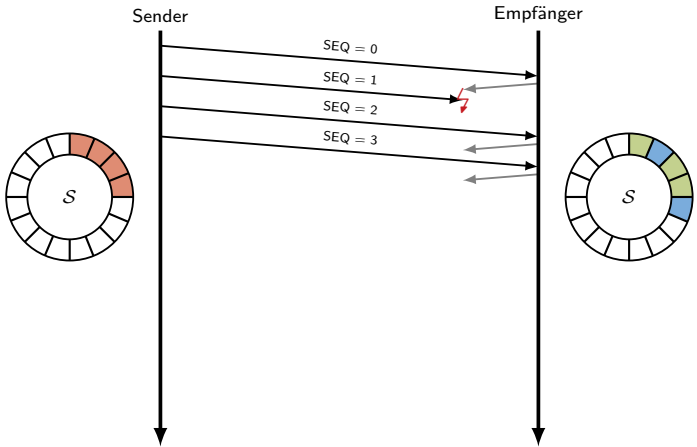
  $$w_s \leq N - 1.$$

  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.

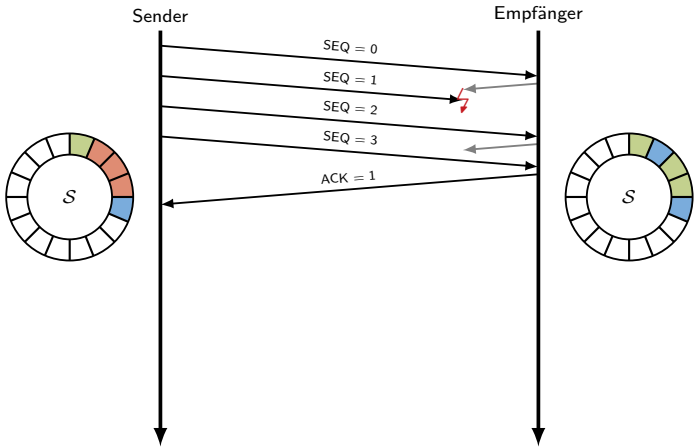
Selective Repeat: $N = 16$, $w_s = 4$, $w_r = 4$



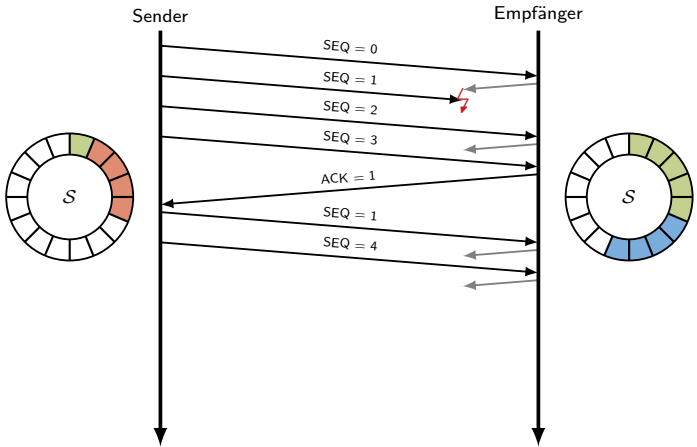
Selective Repeat: $N = 16$, $w_s = 4$, $w_r = 4$



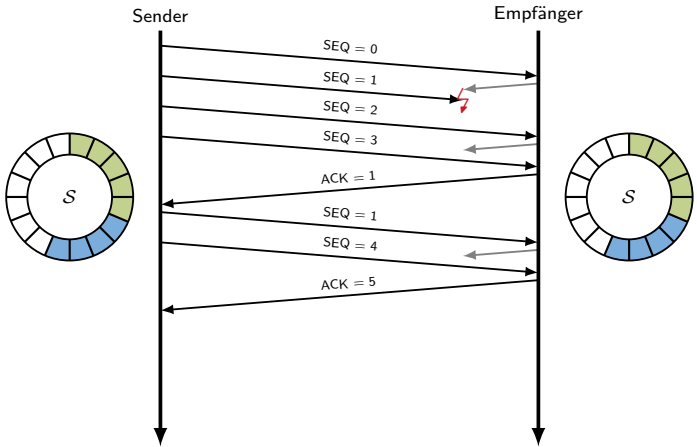
Selective Repeat: $N = 16$, $w_s = 4$, $w_r = 4$



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

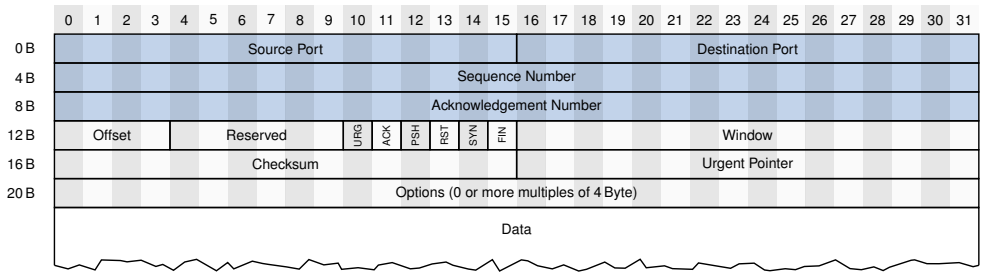
- Bei einer Umsetzung dieser Konzepte benötigt insbesondere der Empfänger einen Empfangspuffer, dessen Größe an die Sende- und Empfangsfenster angepasst ist.
- Für praktische Anwendungen werden die Größen von W_s und W_r dynamisch angepasst (siehe Case Study zu TCP), wodurch Algorithmen zur Staukontrolle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

Transmission Control Protocol (TCP)

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:



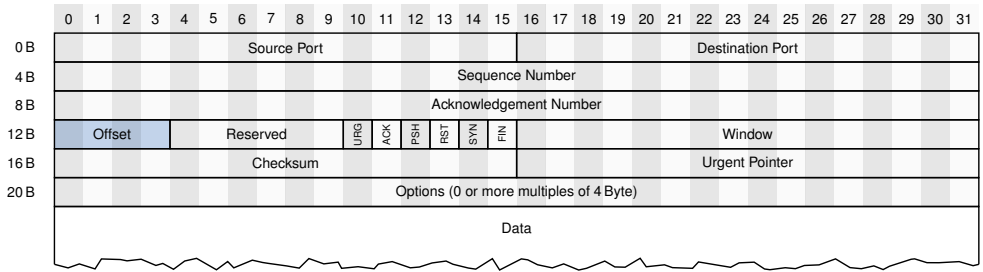
- Quell- und Zielport werden analog zu UDP verwendet.
- Sequenz- und Bestätigungsnummer dienen der gesicherten Übertragung. Es werden bei TCP nicht ganze Segmente sondern einzelne Bytes bestätigt (stromorientierte Übertragung).

Transmission Control Protocol (TCP)

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

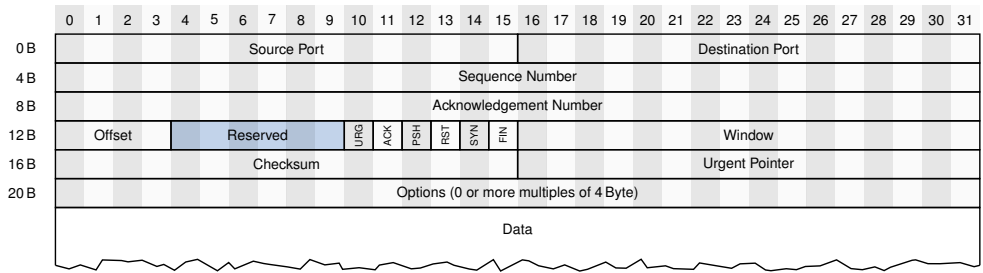
- Gibt die Länge des TCP-Headers in Vielfachen von 4 B an.
- Der TCP-Header hat variable Länge (Optionen, vgl. IPv4-Header).

Transmission Control Protocol (TCP)

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

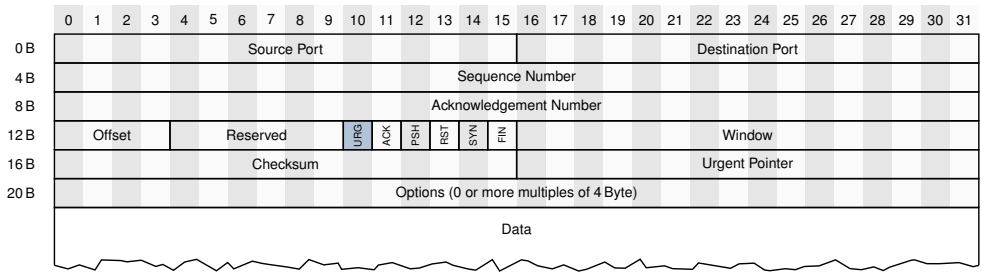
- Hat in bisherigen TCP-Versionen keine Verwendung. Muss auf 0 gesetzt werden, so dass zukünftige TCP-Versionen bei Bedarf das Feld nutzen können.

Transmission Control Protocol (TCP)

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)

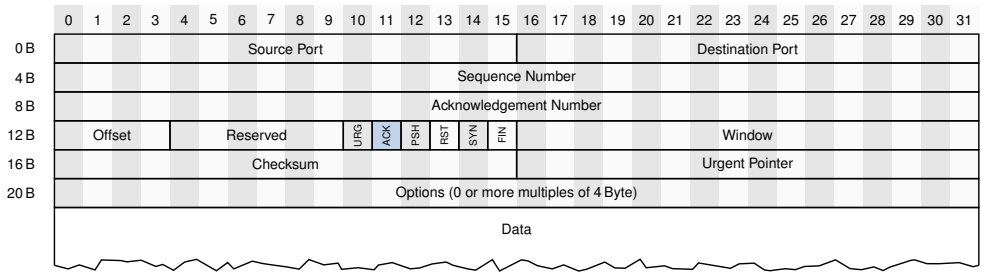
- Ist das Flag gesetzt, werden die Daten im aktuellen TCP-Segment beginnend mit dem ersten Byte bis zu der Stelle, an die das Feld Urgent Pointer zeigt, sofort an höhere Schichten weitergeleitet.

Transmission Control Protocol (TCP)

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“)

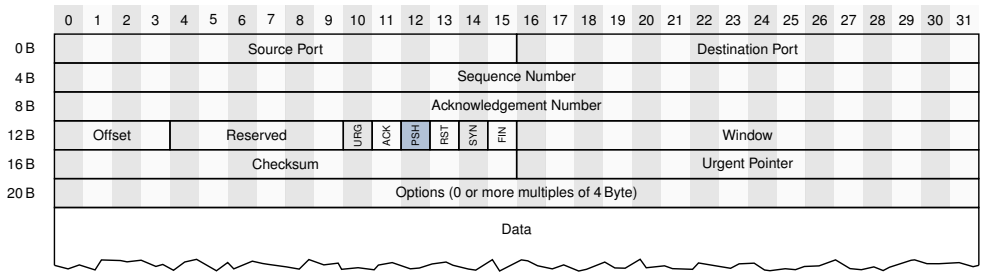
- Ist das Flag gesetzt, handelt es sich um eine Empfangsbestätigung.
- Bestätigungen können bei TCP auch „huckepack“ (engl. piggy backing) übertragen werden, d. h. es werden gleichzeitig Nutzdaten von *A* nach *B* übertragen und ein zuvor von *B* nach *A* gesendetes Segment bestätigt.
- Die Acknowledgement-Number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

Transmission Control Protocol (TCP)

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“)

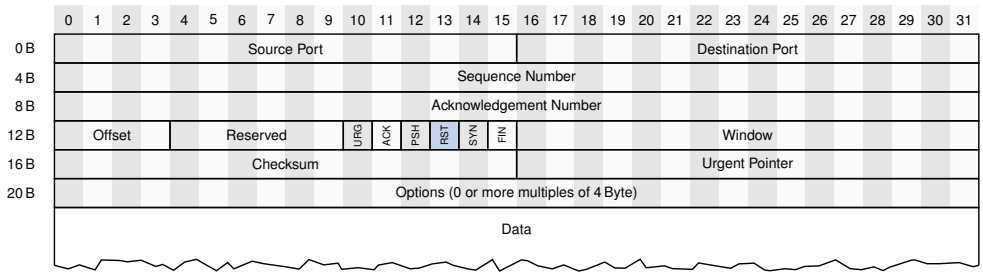
- Ist das Flag gesetzt, werden sende- und empfangsseitige Puffer des TCP-Stacks umgangen.
- Sinnvoll für interaktive Anwendungen (z. B. Telnet-Verbindungen).

Transmission Control Protocol (TCP)

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“)

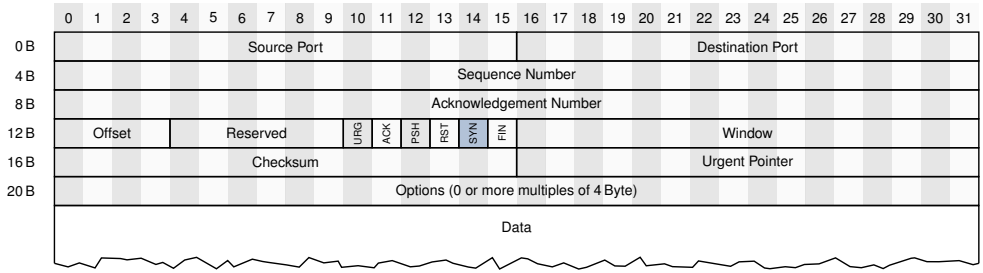
- Dient dem Abbruch einer TCP-Verbindung ohne ordnungsgemäßen Verbindungsabbau.

Transmission Control Protocol (TCP)

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“)

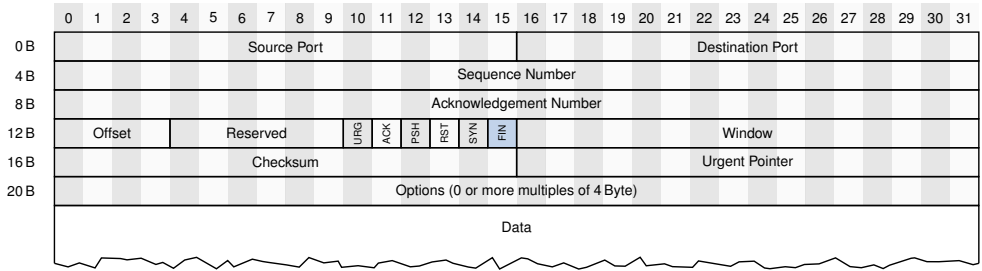
- Ist das Flag gesetzt, handelt es sich um ein Segment, welches zum Verbindungsaufbau gehört (initialer Austausch von Sequenznummern).
- Ein gesetztes SYN-Flag inkrementiert Sequenz- und Bestätigungsnummern um 1 obwohl keine Nutzdaten transportiert werden.

Transmission Control Protocol (TCP)

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“)

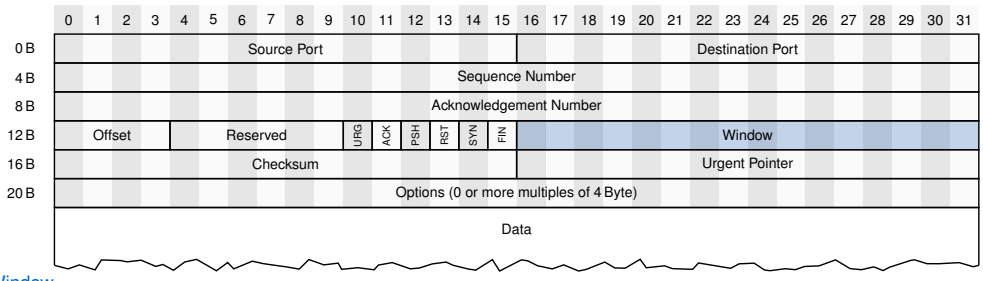
- Ist das Flag gesetzt, handelt es sich um ein Segment, welches zum Verbindungsabbau gehört.
- Ein gesetztes FIN-Flag inkrementiert Sequenz- und Bestätigungsnummern um 1 obwohl keine Nutzdaten transportiert werden.

Transmission Control Protocol (TCP)

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

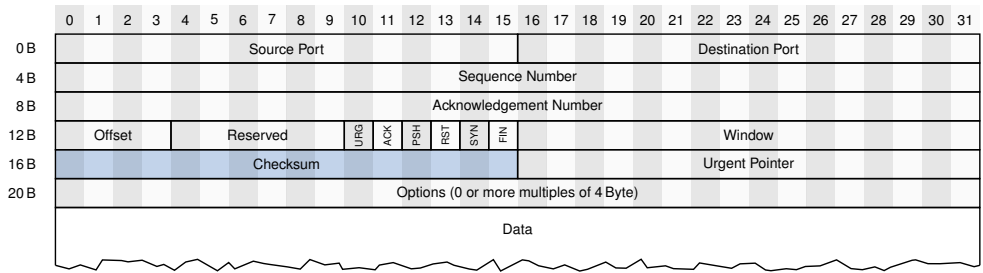
- Größe des aktuellen Empfangsfensters W_r in Byte.
- Ermöglicht es dem Empfänger, die Datenrate des Senders zu drosseln.

Transmission Control Protocol (TCP)

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

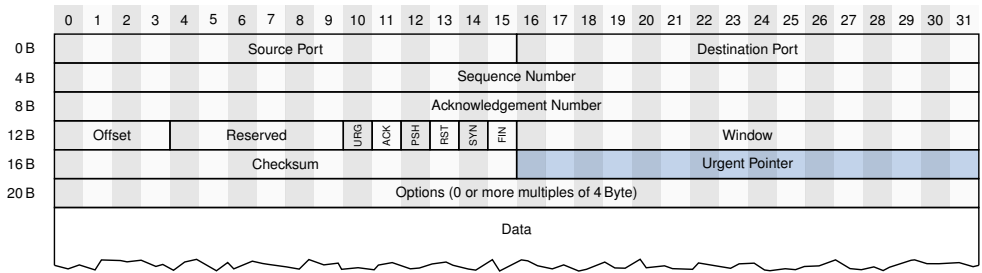
- Prüfsumme über Header und Daten.
- Wie bei UDP wird zur Berechnung ein Pseudo-Header verwendet.

Transmission Control Protocol (TCP)

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)

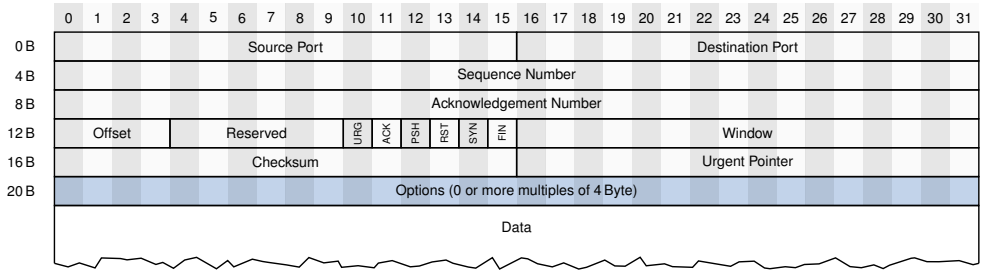
- Gibt das Ende der „Urgent-Daten“ an, welche unmittelbar nach dem Header beginnen und bei gesetztem URG-Flag sofort an höhere Schichten weitergereicht werden sollen.

Transmission Control Protocol (TCP)

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[1]

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.

[1] 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.

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 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 bestätige 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:

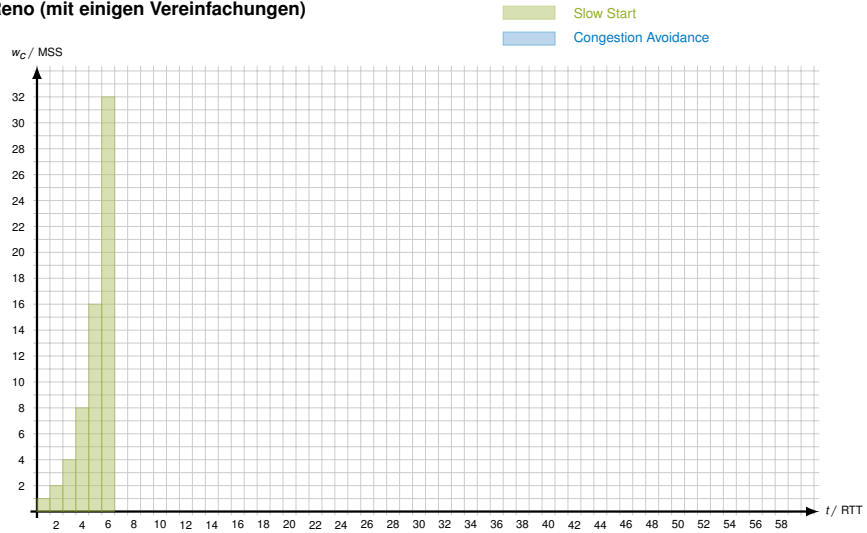
- Wir betrachten hier eine auf das Wesentliche reduzierte Implementierung von TCP, die auf TCP *Reno* basiert.
- Die einzelnen TCP-Version (*Tahoe, Reno, New Reno, Cubic, . . .*) unterscheiden sich in Details, sind aber alle zueinander kompatibel.
- Linux verwendet derzeit TCP *Cubic*, welches das Congestion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

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

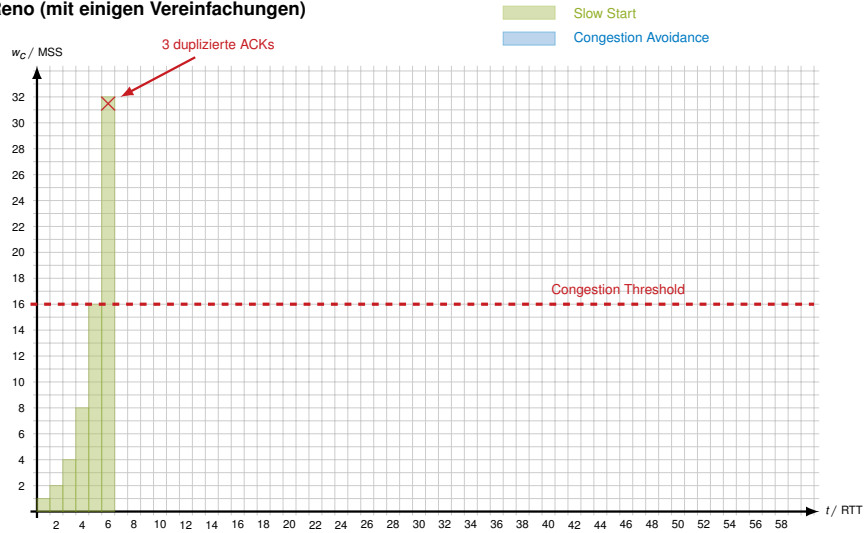
1. 3 duplizierte Bestätigungen (Duplicate ACKs)
   - Setze den Schwellwert für die Stauvermeidung auf $w_c/2$.
   - Reduziere W_c auf die Größe dieses Schwellwerts.
   - Beginne mit der Stauvermeidungsphase.
2. Timeout
   - Setze den Schwellwert für die Stauvermeidung auf $w_c/2$.
   - Setze $w_c = 1$ 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.

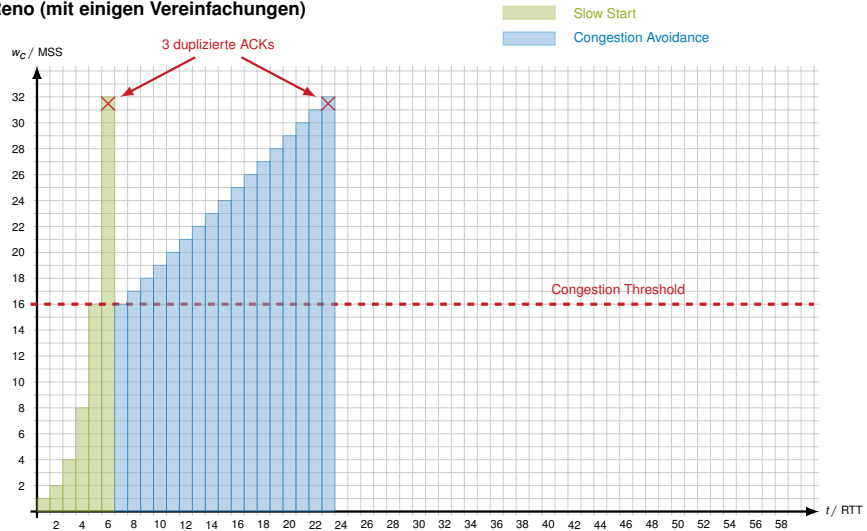
Beispiel: TCP-Reno (mit einigen Vereinfachungen)



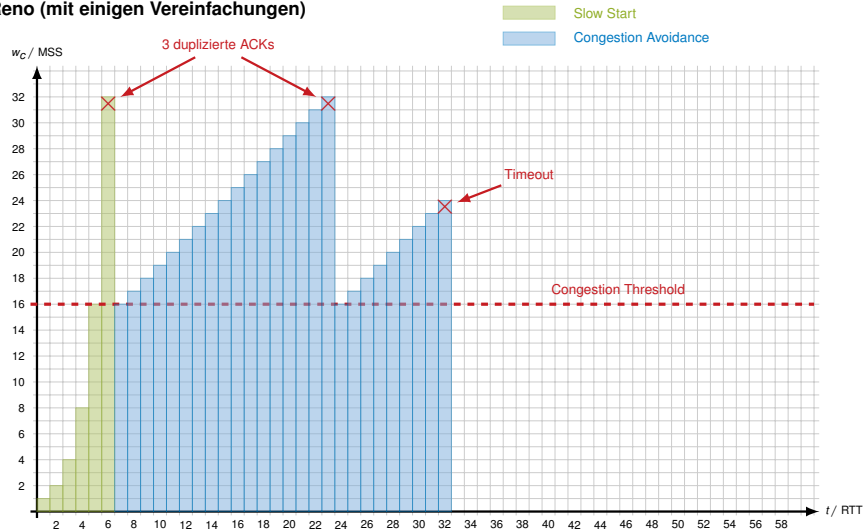
Beispiel: TCP-Reno (mit einigen Vereinfachungen)



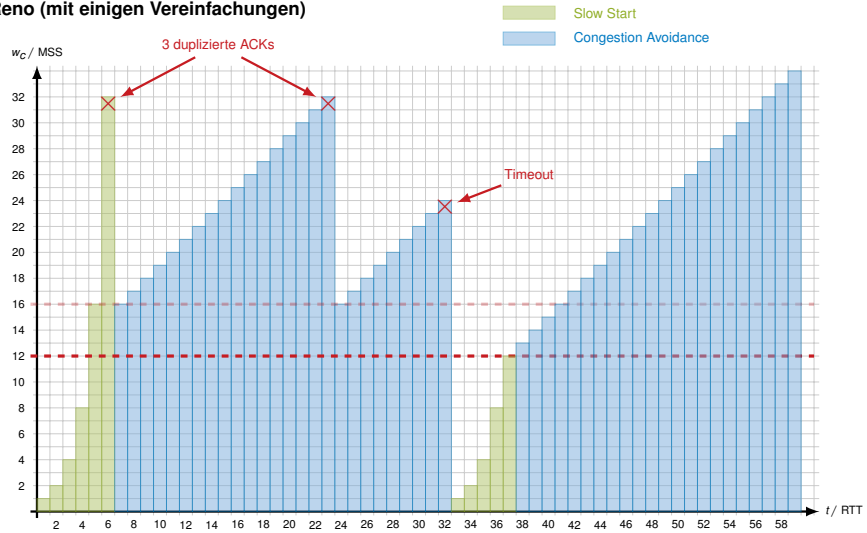
Beispiel: TCP-Reno (mit einigen Vereinfachungen)



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 den Verlust von Segmenten (Daten und Bestätigungen) stets als eine Folge einer Überlastsituation im Netzwerk (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 bzw. 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.

$\Rightarrow$ 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).

Kapitel 4: Transportschicht

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 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 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 zur Übersetzung von $N \geq 1$ auf $M \geq 1$ andere IP-Adressen. Bei IPv4 ist der weitaus häufigste Anwendungsfall die Abbildung von N privaten (nicht öffentlichen) auf M öffentliche (global eindeutige) IP-Adressen:

- $N \leq M$: 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 weder eindeutig noch zur Ende-zu-Ende-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 bei IPv4 sind:

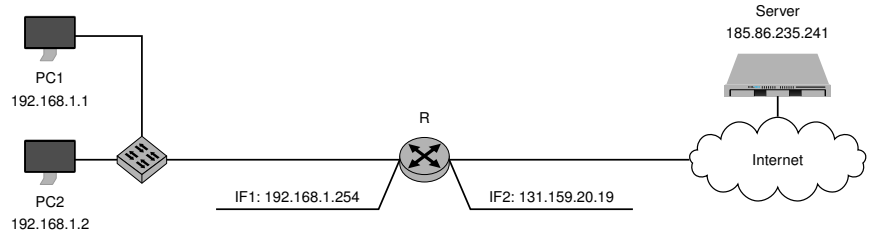
- 10.0.0.0 / 8
- 172.16.0.0 / 12
- 169.254.0.0 / 16
- 192.168.0.0 / 16

Der Bereich 169.254.0.0 / 16 wird zur automatischen Adressvergabe (Automatic Private IP Adressing) 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?

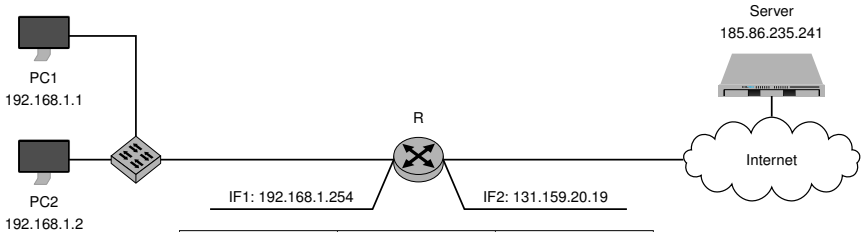
Üblicherweise übernehmen Router die Netzwerkadressübersetzung:



- 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.

Network Address Translation (NAT)

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

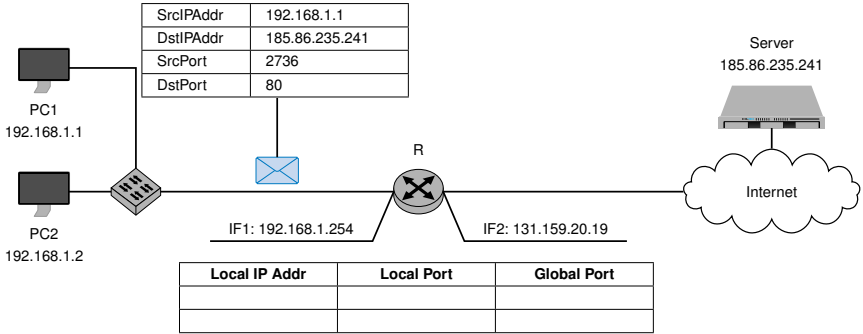


Local IP Addr	Local Port	Global Port

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

Network Address Translation (NAT)

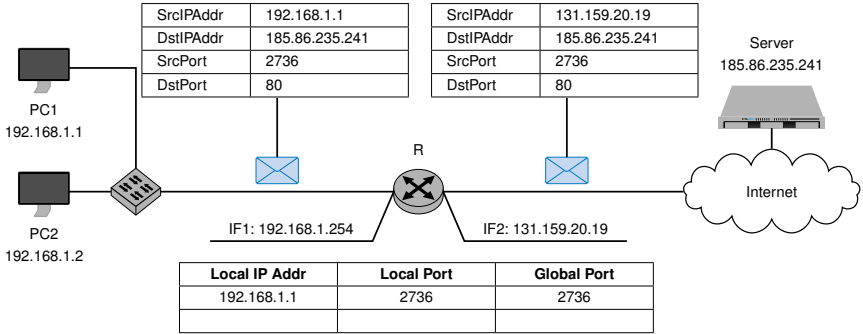
PC1 greift auf eine Webseite zu, welche auf dem Server mit der IP-Adresse 185.86.235.241 liegt:



- 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 Protokoll auf Schickt 7 vorgegeben (80 = HTTP)

Network Address Translation (NAT)

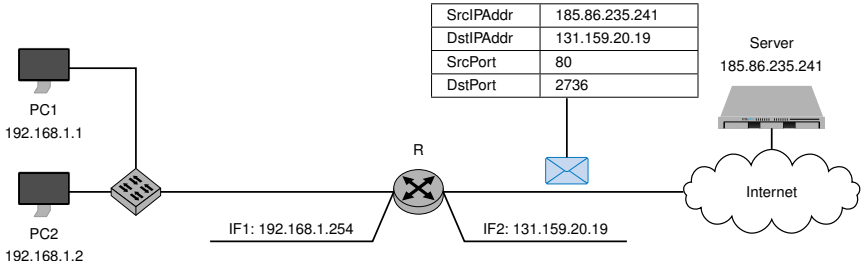
PC1 greift auf eine Webseite zu, welche auf dem Server mit der IP-Adresse 185.86.235.241 liegt:



- 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 Protokoll auf Schickt 7 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

Network Address Translation (NAT)

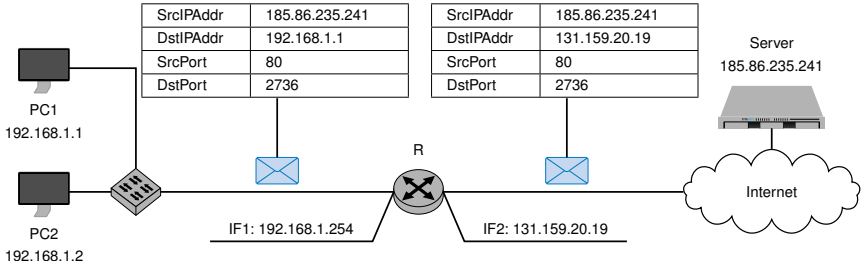
Antwort vom Server an PC1



- 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.

Network Address Translation (NAT)

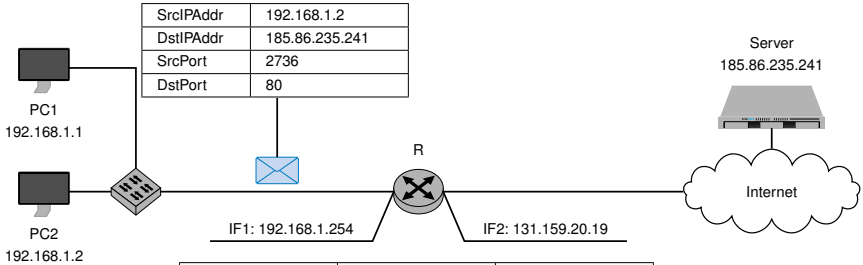
Antwort vom Server an PC1



- 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.

Network Address Translation (NAT)

PC2 greift nun ebenfalls auf den Server zu:

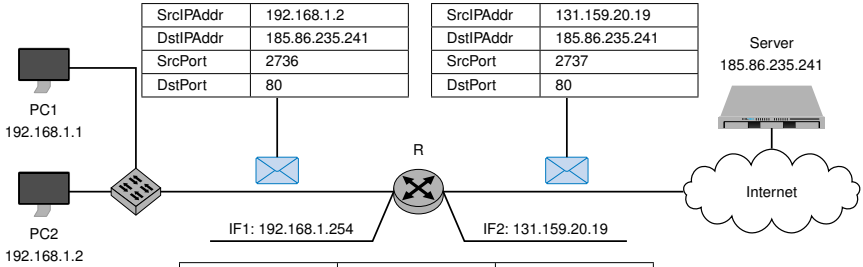


Local IP Addr	Local Port	Global Port
192.168.1.1	2736	2736

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

Network Address Translation (NAT)

PC2 greift nun ebenfalls auf den Server zu:



Local IP Addr	Local Port	Global Port
192.168.1.1	2736	2736
192.168.1.2	2736	2737

- 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.

Network Address Translation (NAT)

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:

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

Allgemeine Anmerkungen

- Ist NAT eine Firewall [2][1]?
  - **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.

[1] 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][1]?
  - **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 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).

[1] 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][1]?
  - **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 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).

[1] 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][1]?
  - **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 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.

[1] 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 Übersetzungs rückgängig machen zu können.
- Genau diese Rückübersetzung führen ältere Versionen der NAT-Implemnentierung von Virtualbox nicht durch.

NAT und IPv6

- NAT kann auch für IPv6 verwendet werden.

Network Address Translation (NAT)

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.

Network Address Translation (NAT)

NAT und IPv6

- NAT kann auch für IPv6 verwendet werden.

Präfix-Übersetzung

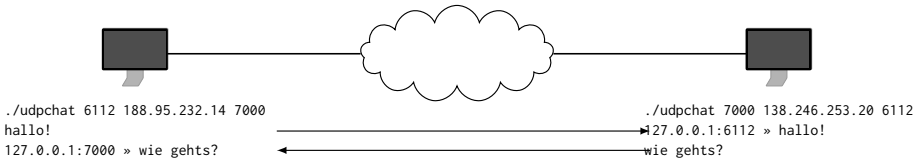
- NAT für IPv6 besitzt spezifische Probleme und Herausforderungen. RFC 6296 spezifiziert IPv6-to-IPv6 Network Prefix Translation.
- Dabei wird ein 1:1-Mapping von Adressen erzeugt.
  - Dies wäre auch bei IPv4 im begrenzten Umfang möglich (sofern einer Organisation ausreichend öffentliche Adressen zur 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.

Was wir wollen:

- Ein kleines Python-Skript, das gleichzeitig als Client und Server arbeiten soll (P2P-Modell)
- Nur 1:1-Verbindungen, also keine Gruppen-Chats (vorerst)



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

Case Study: UDP-Chat

`socket()`[1]

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

POSIX Socket API: `socket()`

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

int socket(int domain, int type, int protocol);
```

Parameter `domain` ($\widehat{\approx}$ Layer 3 Protokoll)
Spezifiziert die Protokollfamilie, die mit dem Socket benutzt werden soll.

`AF_INET`	Für IPv4 Verbindungen
`AF_INET6`	Für IPv6 Verbindungen
`AF_UNIX`	Lokale Kommunikation über UNIX Domänen Sockets

Parameter `type` ($\widehat{\approx}$ Layer 4 Protokoll)
Spezifiziert die Art der Kommunikation, welche verwendet werden soll.

`SOCK_STREAM`	sequentielle, zuverlässige, verbindungsorientierte Zweiweg-Kommunikation
`SOCK_DGRAM`	verbindungslose, unzuverlässige Kommunikation

Parameter `protocol`

- Spezifiziert das konkrete Protokoll, das verwendet werden soll.
- Da es normalerweise nur ein Protokoll pro `domain` und `type` gibt, wird hier in der Regel 0 eingetragen.

[1] Siehe man page `socket(2)`

Case Study: UDP-Chat

`bind()`[2]

- Assoziiert einen Socket mit einer Adresse
- Sollte aufgerufen werden, wenn man mit einem Socket eingehende Pakete entgegennehmen möchte

POSIX Socket API: `bind()`

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

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Parameter `sockfd`

Der File Deskriptor des Sockets

Parameter `addr`

- Adresse, die mit dem Socket assoziiert werden soll.
- Spezifiziert in protokollspezifischer Datenstruktur, dessen Länge in `addrlen` angegeben ist.
- Spezielle Adresswerte:

`INADDR_ANY`	Assoziiert den Socket mit allen passenden Adressen
`INADDR_LOOPBACK`	Assoziiert den Socket mit der Loopback-Adresse $\longrightarrow$ Nur lokal auf dem Gerät erreichbar

[2] Siehe man page `bind(1)`

Case Study: UDP-Chat

sendto()[3]

- Sendet binäre Nutzdaten an die in den Parametern spezifizierte Zieladresse
- Wird in der Regel nur für UDP Verbindungen benutzt

POSIX Socket API: sendto()

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

ssize_t sendto(int sockfd, const void buf[.len], size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);
```

Parameter `sockfd` Der File Deskriptor des Sockets

Parameter `buff` und Paramter `len` Die binären Nutzdaten der Länge `len`, welche versandt werden sollen.

Parameter `flags` Zusätzliche Optionen. Siehe `man 2 sendto`

Parameter `dest_addr`

- Adresse, an die die Nachricht verschickt werden soll
- Spezifiziert in protokollspezifischer Datenstruktur, dessen Länge in `addrlen` angegeben ist.

Rückgabewert Spezifiziert die Anzahl an Bytes, die gesendet wurden.

[3] Siehe man page `sendto(2)`

Case Study: UDP-Chat

`recvfrom()`[4]

- Empfängt binäre Nutzdaten von einem Kommunikationspartner
- Wird in der Regel für UDP Verbindungen benutzt

POSIX Socket API: `recvfrom()`

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

ssize_t recvfrom(int sockfd, void buf[.len], size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
```

Parameter `sockfd` Der File Deskriptor des Sockets

Parameter `buff` und Paramter `len` Ein Pointer auf einen Puffer der Länge `len`, in welchen die empfangenen Daten angelegt werden sollen.

Parameter `flags` Zusätzliche Optionen. Siehe `man 2 recvfrom`

Parameter `src_addr` and `addrlen` Adresse, von der die Daten empfangen wurden, wobie `addrlen` die Größe der Addressdatenstruktur angibt.

Rückgabewert Spezifiziert die Anzahl an Bytes, die empfangen wurden.

[4] Siehe man page `recvfrom(2)`

Case Study: UDP-Chat

`close()`[5]

- Schließt den Dateideskriptor (Socket).
- Bei `SOCK_STREAM` Sockets wird die Verbindung geschlossen.

POSIX Socket API: `close()`

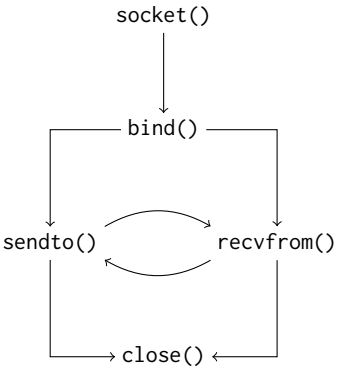
```
#include <unistd.h>

int close(int fd);
```

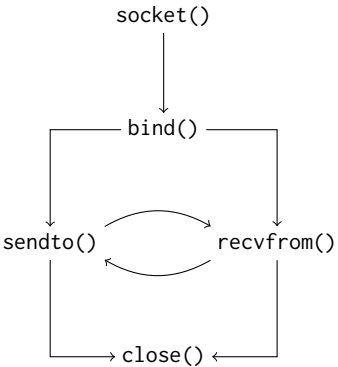
Parameter `fd` Der File Deskriptor des Sockets

[5] Siehe man page `close(2)`

Case Study: UDP-Chat

Überblick UDP Socket Kommunikation





Wie merkt unser Programm, wenn neue Daten ankommen?
Hier gibt es 3 Möglichkeiten:

- Einfach ein `recvfrom()` auf dem Socket:
  - `recvfrom()` 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()`:
  - 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.

Case Study: UDP-Chat

`select()`[6]

- Wartet darauf, dass ein Dateideskriptor bereit zum lesen oder schreiben ist oder ein Fehler aufgetreten ist
- Ineffizient bei großen Mengen an Dateideskriptoren, die überwacht werden sollen
- Begrenzt auf Dateideskriptoren mit Zahlenwert unter 1024

POSIX Socket API: `select()`

```
#include <sys/select.h>

typedef /* ... */ fd_set;

int select(int nfds, fd_set *readfds,
           fd_set *writefds,
           fd_set *exceptfds,
           struct timeval *timeout);

void FD_CLR(int fd, fd_set *set);
int  FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);
```

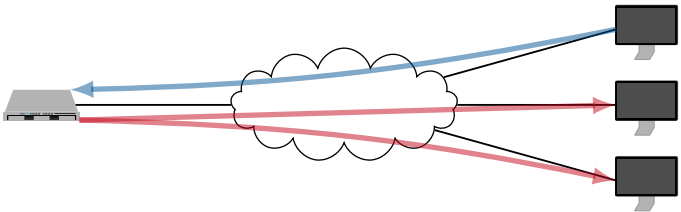
[6] Siehe `man 2 select`

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 erweiteren:
   - 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 in jedem Fall zwei unterschiedliche Programme



Download:
Der Chat (Client und Server) wird nach der Vorlesung unter `https://grnvs.net/codedemos` zum Download bereitgestellt.
Während der Vorlesung nutzen Sie als Client am besten `telnet <Ziel-IP> <Port>` wie über Moodle angekündigt.

Case Study: TCP-Relay-Chat

`listen()`[7]

- Markiert einen Socket als passiven Socket
- Eingehende Verbindungsanfragen werden nach dem Funktionsaufruf vom Betriebssystem in einer Warteschlange abgelegt
- Verbindungen in dieser Warteschlange können über den passiven Socket angenommen werden (siehe `accept`)

POSIX Socket API: `listen()`

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

int listen(int sockfd, int backlog);
```

Parameter `sockfd`

Der File Deskriptor des Sockets.

Parameter `backlog`

Gibt die maximale Größe der Warteschlange für eingehende Verbindungsanfragen an.

[7] Siehe man page `listen(2)`

Case Study: TCP-Relay-Chat

`accept()`[8]

- Akzeptiert eine Verbindungsanfrage aus der Warteschlange des Betriebssystems
- Blockiert, bis eine Verbindung aufgebaut wurde

POSIX Socket API: `accept()`

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

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

Parameter `sockfd`

Der File Deskriptor des passiven Sockets.

Parameter `addr` und `addrlen`

In diesen Pointern werden die Adressinformationen des neu verbundenen Kommunikationspartners abgelegt.

Rückgabewert Der Socket Dateideskriptor der aufgebauten Verbindung mit dem neuen Client.

[8] Siehe man page `accept(2)`

Case Study: TCP-Relay-Chat

`connect()`[9]

- Die Adresse des Kommunikationspartners des Sockets wird auf die gegebenen Adressinformationen gesetzt
- Im Fall von `SOCK_STREAM` wird der Verbindungsaufbau durchgeführt
- Im Fall von `SOCK_DGRAM` wird die Standardremoteadresse gesetzt (mehrfach änderbar)

POSIX Socket API: `connect()`

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

int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Parameter `sockfd`

Der Socket Dateideskriptor, welchem die Kommunikationspartneraddressinformationen assoziiert werden sollen.

Parameter `addr` und `addrlen`

Die Adressinformationen des Kommunikationspartners

[9] Siehe man page `connect(2)`

Case Study: TCP-Relay-Chat

`send()`[10]

- Sendet binäre Nutzdaten an den verbundenen Kommunikationspartner.
- Kommunikationspartner muss vorher durch `connect()` gesetzt worden sein.

POSIX Socket API: `send()`

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

ssize_t send(int sockfd, const void buf[.len], size_t len, int flags);
```

Parameter `sockfd`

Der File Deskriptor des Sockets

Parameter `buff` und Paramter `len`

Die binären Nutzdaten der Länge `len`, welche versendet werden sollen.

Parameter `flags`

Zusätzliche Optionen. Siehe `man 2 send`

Rückgabewert Spezifiziert die Anzahl an Bytes, die gesendet wurden.

[10] Siehe man page `send(2)`

Case Study: TCP-Relay-Chat

`recv()`[11]

- Empfängt binäre Nutzdaten vom spezifizierten Kommunikationspartner.
- Kommunikationspartner muss vorher durch `connect()` gesetzt worden sein.

POSIX Socket API: `recv()`

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

ssize_t recv(int sockfd, void buf[.len], size_t len, int flags);
```

Parameter `sockfd`

Der File Deskriptor des Sockets

Parameter `buff` und Paramter `len`

Ein Pointer auf einen Puffer der Länge `len`, in welchen die empfangenen Daten angelegt werden sollen.

Parameter `flags`

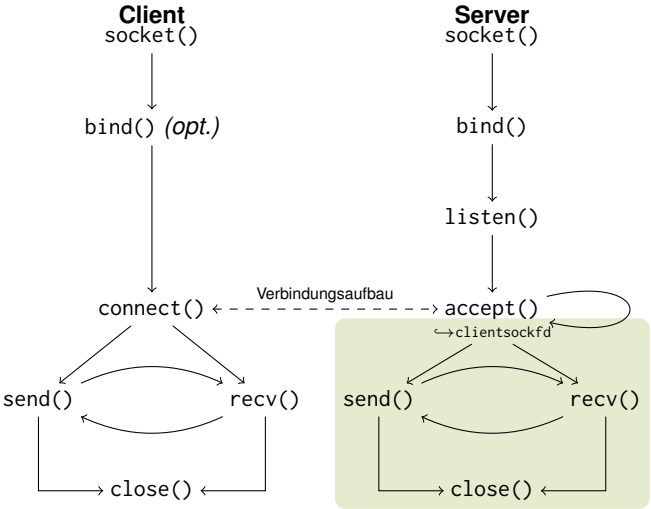
Zusätzliche Optionen. Siehe `man 2 recvfrom`

Rückgabewert Spezifiziert die Anzahl an Bytes, die empfangen wurden.

[11] Siehe man page `recv(2)`

Case Study: TCP-Relay-Chat

Überblick TCP Socket Kommunikation



Case Study: TCP-Relay-Chat

Mit diesen Socket API Funkionen lässt sich relativ schnell

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

Frage:

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

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 Socket API Funktionen (und in welcher Reihenfolge) zum öffnen der entsprechenden Ports bzw. Verbindungen notwendig sind,
- was diese Socket API Funktionen tun (d. h. etwas mehr sagen können, als „`select()` akzeptiert die Verbindung“...),
- 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.

Kapitel 4: Transportschicht

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.