

9. Foliensatz Computernetze

Prof. Dr. Christian Baun

Frankfurt University of Applied Sciences
(1971–2014: Fachhochschule Frankfurt am Main)
Fachbereich Informatik und Ingenieurwissenschaften
christianbaun@fb2.fra-uas.de

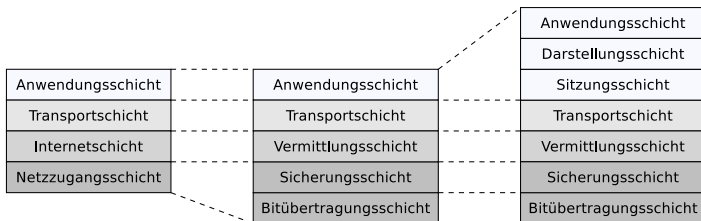
Transportschicht

- Aufgaben der Transportschicht (Transport Layer):
 - Enthält **Ende-zu-Ende-Protokolle** für Interprozesskommunikation
 - Adressierung der Prozesse mit **Portnummern**
 - Unterteilung der Daten der Anwendungsschicht in **Segmente**

TCP/IP-Referenzmodell

Hybrides Referenzmodell

OSI-Referenzmodell



Übungsblatt 5
wiederholt die für
die Lernziele
relevanten Inhalte
dieses Foliensatzes

- Geräte: Gateway
- Protokolle: TCP, UDP

Lernziele dieses Foliensatzes

- Transportschicht
 - Eigenschaften von Transportprotokollen
 - Adressierung in der Transportschicht
 - User Datagram Protocol (UDP)
 - Aufbau von UDP-Segmenten
 - Arbeitsweise
 - Transmission Control Protocol (TCP)
 - Aufbau von TCP-Segmenten
 - Arbeitsweise
 - Flusskontrolle (*Flow Control*)
 - Überlastkontrolle (*Congestion Control*)
 - Denial of Service-Attacken via SYN-Flood

Eigenschaften von Transportprotokollen

- Gewünschte Eigenschaften von Transportprotokollen sind u.a.
 - Garantierte Datenübertragung
 - Einhaltung der korrekten Reihenfolge der Daten
 - Unterstützung beliebig großer Datenübertragungen
 - Der Sender soll das Netzwerk nicht überlasten
 - Er soll in der Lage sein, den eigenen Datenfluss (die Übertragungsrate) anzupassen \implies Flusskontrolle
 - Der Empfänger soll das Sendeverhalten des Senders kontrollieren können, um Überlast beim Empfänger zu vermeiden \implies Überlastkontrolle
- Es sind also Transportprotokolle nötig, die die negativen Eigenschaften der Netze in die (positiven) Eigenschaften umwandeln, die von Transportprotokollen erwartet werden
- Die am häufigsten verwendeten Transportprotokolle:
 - **UDP**
 - **TCP**
- Adressierung erfolgt in der Transportschicht mit **Sockets**

Sockets

- **Sockets** sind die plattformunabhängige, standardisierte **Schnittstelle** zwischen der Implementierung der Netzwerkprotokolle im Betriebssystem und den Anwendungen
- **Ein Socket besteht aus einer Portnummer und einer IP-Adresse**
- Man unterscheidet zwischen Stream Sockets und Datagram Sockets
 - **Stream Sockets** verwendeten das verbindungsorientierte TCP
 - **Datagram Sockets** verwendeten das verbindungslose UDP

Werkzeug(e) zur Kontrolle offener Ports und Sockets unter...

- Linux/UNIX: netstat, lsof und nmap
- Windows: netstat

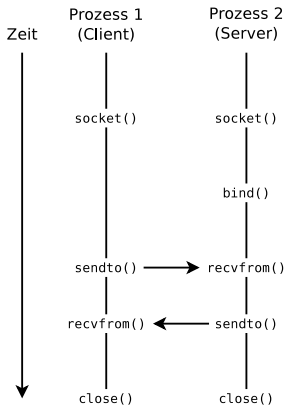
Alternativen zu Sockets in der Interprozesskommunikation (IPC)

Pipes, Message Queues und gemeinsamer Speicher (Shared Memory) \implies siehe Betriebssysteme-Vorlesung

User Datagram Protocol (UDP)

- **Verbindungsloses Transportprotokoll**
 - Datenübertragungen finden ohne vorherigen Verbindungsaufbau statt
- Einfacheres Protokoll als das verbindungsorientierte TCP
 - Nur für die Adressierung der Segmente zuständig
 - Es findet keine Sicherung der Datenübertragung statt
- Übertragungen werden nicht vom Empfänger beim Sender bestätigt
 - Segmente können bei der Übertragung verloren gehen
- Abhängig von der Anwendung, z.B. bei Videostreaming, ist das akzeptabel
 - Geht bei der Übertragung eines Videos via TCP ein Segment, also eine Bildinformation verloren, wird es neu angefordert
 - Es käme zu Aussetzern
 - Um das zu kompensieren, sind Wiedergabepuffer nötig
 - Speziell bei Videotelefonie versucht man aber die Puffer möglichst klein zu halten, weil diese zu Verzögerungen führen
 - Nutzt man UDP zur Übertragung eines Videos oder für Videotelefonie, geht beim Verlust eines Segments nur ein Bild verloren

Verbindungslose Kommunikation mit Sockets – UDP



- **Client**

- Socket erstellen (`socket`)
- Daten senden (`sendto`) und empfangen (`recvfrom`)
- Socket schließen (`close`)

- **Server**

- Socket erstellen (`socket`)
- Socket an einen Port binden (`bind`)
- Daten senden (`sendto`) und empfangen (`recvfrom`)
- Socket schließen (`close`)

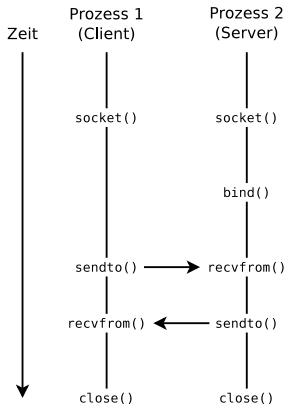
Sockets via UDP – Beispiel (Server)

```

1 #!/usr/bin/env python
2 # -*- coding: iso-8859-15 -*-
3 # Server: Empfängt eine Nachricht via UDP
4
5 # Modul socket importieren
6 import socket
7
8 # Stellvertretend für alle Schnittstellen des Hosts
9 # '' = alle Schnittstellen
10 HOST = ''
11 # Portnummer des Servers
12 PORT = 50000
13
14 # Socket erzeugen und Socket-Deskriptor zurückliefern
15 sd = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
16
17 try:
18     sd.bind((HOST, PORT))                # Socket an Port
19                                         binden
20     while True:
21         data = sd.recvfrom(1024)        # Daten empfangen
22         # Empfangene Daten ausgeben
23         print 'Received:', repr(data)
24 finally:
25     sd.close()                          # Socket schließen

```

```
$ python udp_server.py
```



Sockets via UDP – Beispiel (Client)

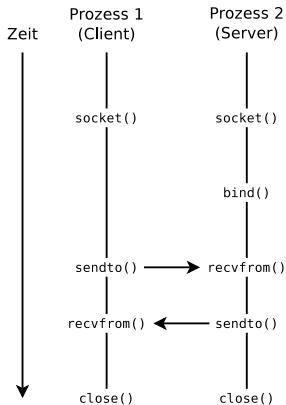
```

1 #!/usr/bin/env python
2 # -*- coding: iso-8859-15 -*-
3 # Client: Schickt eine Nachricht via UDP
4
5 import socket                # Modul socket importieren
6
7 HOST = 'localhost'          # Hostname des Servers
8 PORT = 50000                 # Portnummer des Servers
9 MESSAGE = 'Hello World'     # Nachricht
10
11 # Socket erzeugen und Socket-Deskriptor zurückliefern
12 sd = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
13
14 # Nachricht an Socket senden
15 sd.sendto(MESSAGE, (HOST, PORT))
16
17 sd.close()                  # Socket schließen

```

```
$ python udp_client.py
```

```
$ python udp_server.py
Received: ('Hello World', ('127.0.0.1', 39834))
```



Transmission Control Protocol (TCP)

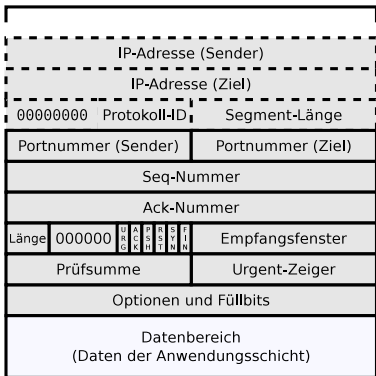
- **Verbindungsorientiertes Transportprotokoll**
- Erweitert das Vermittlungsprotokoll IP um die Zuverlässigkeit, die für viele Anwendungen gewünscht bzw. nötig ist
- Garantiert, dass Segmente **vollständig** und in der **korrekten Reihenfolge** ihr Ziel erreichen
 - Verlorene oder nicht bestätigte TCP-Segmente sendet der Sender erneut
- Eine TCP-Verbindung wird wie eine Datei geöffnet und geschlossen
 - Genau wie bei einer Datei wird die Position im Datenstrom exakt angegeben

TCP-Standard: RFC 793 von 1981

<http://tools.ietf.org/rfc/rfc793.txt>

Aufbau von TCP-Segmenten (1/5)

32 Bit (4 Bytes)



- Ein TCP-Segment kann maximal 64 kB Nutzdaten (Daten der Anwendungsschicht) enthalten
 - Üblich sind kleinere Segmente (≤ 1500 Bytes bei Ethernet)
- Der Header von TCP-Segmenten ist komplexer im Vergleich zu UDP-Segmenten

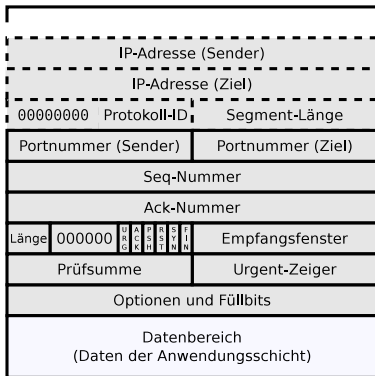
Overhead

- Größe des TCP-Headers (ohne das Optionsfeld): nur 20 Bytes
- Größe des IP-Headers (ohne das Optionsfeld): auch nur 20 Bytes

⇒ Der Overhead, den die TCP- und IP-Header verursachen, ist bei einer IP-Paketgröße von mehreren kB gering

Aufbau von TCP-Segmenten (2/5)

32 Bit (4 Bytes)

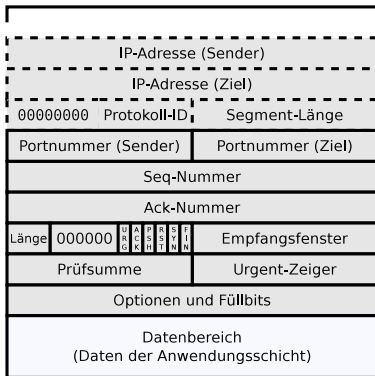


- Ein Datenfeld enthält die Portnummer des sendenden Prozesses
- Ein weiteres Datenfeld enthält die Portnummer des Prozesses, der das Segment empfangen soll
- **Seq-Nummer** enthält die Folgenummer (Sequenznummer) des aktuellen Segments
- **Ack-Nummer** enthält die Folgenummer des nächsten erwarteten Segments

- **Länge** enthält die Länge des TCP-Headers in 32-Bit-Worten, damit der Empfänger weiß, wo die Nutzdaten im TCP-Segment anfangen
 - Dieses Feld ist nötig, weil das Feld *Optionen und Füllbits* eine variable Länge (Vielfaches von 32 Bits) haben kann

Aufbau von TCP-Segmenten (3/5)

32 Bit (4 Bytes)



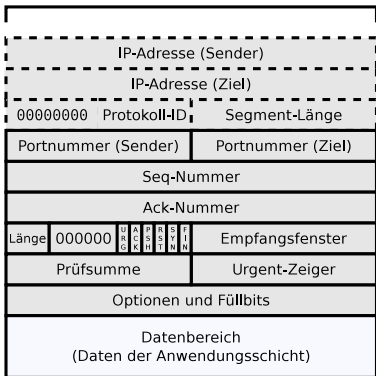
- Das Datenfeld 000000 ist 6 Bits groß und wird nicht verwendet
 - Es hat den Wert Null
- Die 6 je 1 Bit großen Datenfelder sind für Verbindungsaufbau, Datenaustausch und Verbindungsabbau nötig
 - Im folgenden sind die Funktionen dieser Datenfelder jeweils so beschrieben, das sie den Wert 1 haben, also *gesetzt* sind

URG (Urgent) wird in dieser Vorlesung nicht behandelt

- **ACK** (Acknowledge)
 - Gibt an, dass die Bestätigungsnummer im Datenfeld **Ack-Nummer** gültig ist
 - Es wird also verwendet, um den Empfang von Segmenten zu bestätigen

Aufbau von TCP-Segmenten (4/5)

32 Bit (4 Bytes)



PSH (Push) wird in dieser Vorlesung nicht behandelt

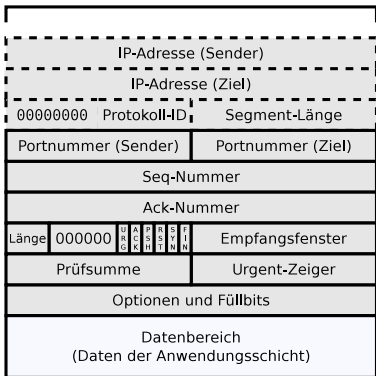
RST (Reset) wird in dieser Vorlesung nicht behandelt

- **SYN** (Synchronize)
 - Weist die Synchronisation der Sequenznummern an
 - Das initiiert den Verbindungsaufbau
- **FIN** (Finish)
 - Weist den Verbindungsabbau an und gibt an, dass der Sender keine Nutzdaten mehr schicken wird

- **Empfangsfenster** enthält die Anzahl freier Bytes im Empfangsfensters des Senders zur Flusskontrolle

Aufbau von TCP-Segmenten (5/5)

32 Bit (4 Bytes)



- Genau wie bei UDP existiert auch für jedes TCP-Segment ein Pseudo-Header, der nicht übertragen wird
 - Dessen Datenfelder gehen aber inklusive regulärem TCP-Header und Nutzdaten in die Berechnung der **Prüfsumme** mit ein
 - Die **Protokoll-ID** von TCP ist die 6

Der **Urgent-Zeiger** wird in dieser Vorlesung nicht behandelt

Das Feld **Optionen und Füllbits** muss ein Vielfaches von 32 Bits groß sein und wird in dieser Vorlesung nicht behandelt

Erinnern Sie sich an NAT aus Foliensatz 8...

Wird ein NAT-Gerät (Router) verwendet, muss dieses Gerät auch die Prüfsummen in TCP-Segmenten neu berechnen, wenn es die IP-Adressen ersetzt

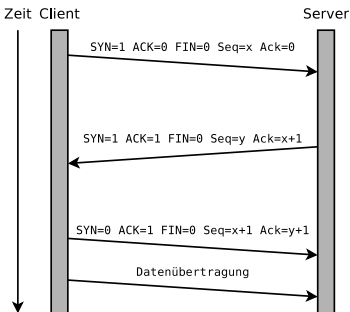
Arbeitsweise von TCP

Sie wissen bereits...

- Jedes Segment hat eine eindeutige Folgenummer (**Sequenznummer**)
- Die Sequenznummer eines Segments ist die Position des ersten Bytes des Segments im Bytestrom
- Anhand der Sequenznummer kann der Empfänger...
 - die Reihenfolge der Segmente korrigieren
 - doppelt angekommene Segmente aussortieren
- Die Länge eines Segments ist aus dem IP-Header bekannt
 - So werden Lücken im Datenstrom entdeckt und der Empfänger kann verlorene Segmente neu anfordern
- Beim Öffnen einer Verbindung (**Dreizege-Handshake**) tauschen beide Kommunikationspartner in drei Schritten Kontrollinformationen aus
 - So ist garantiert, dass der jeweilige Partner existiert und Daten annimmt

TCP-Verbindungsaufbau (Dreiwege-Handshake)

- Der Server wartet passiv auf eine ankommende Verbindung
- Client sendet ein Segment mit $SYN=1$ und fordert damit zur Synchronisation der Folgenummern auf
 \Rightarrow *Synchronize*
 - Server sendet als Bestätigung ein Segment mit $ACK=1$ und fordert mit $SYN=1$ seinerseits zur Synchronisation der Folgenummern auf
 \Rightarrow *Synchronize Acknowledge*
 - Client bestätigt mit einem Segment mit $ACK=1$ und die Verbindung steht
 \Rightarrow *Acknowledge*
-
- Die Anfangs-Sequenznummern (x und y) werden zufällig bestimmt
 - Beim Verbindungsaufbau werden keine Nutzdaten ausgetauscht

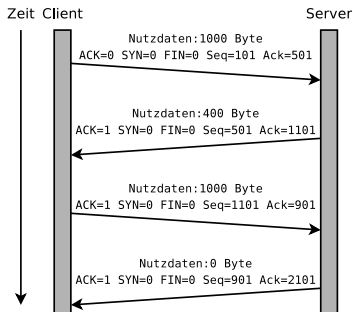


TCP-Datenübertragung

Um eine Datenübertragung zu zeigen, sind für die **Seq-Number** (Folgenummer aktuelles Segment) und die **Ack-Number** (Folgenummer nächstes erwartetes Segment) konkrete Werte nötig

- Im Beispiel ist zu Beginn des Dreizeige-Handshake die Folgenummer des Clients $x=100$ und die des Servers $y=500$
- Nach Abschluss des Dreizeige-Handshake: $x=101$ und $y=501$

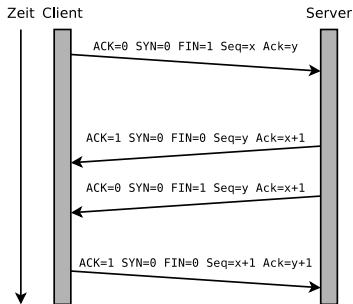
- 1 Client überträgt 1000 Byte Nutzdaten
- 2 Server bestätigt mit $ACK=1$ die empfangenen Nutzdaten und fordert mit der Ack-Number 1101 das nächste Segment an. Im gleichen Segment überträgt der Server 400 Bytes Nutzdaten
- 3 Client überträgt weitere 1000 Byte Nutzdaten. Zudem bestätigt er den Empfang der Nutzdaten mit $ACK=1$ und fordert mit der Ack-Number 901 das nächste Segment an
- 4 Server bestätigt mit $ACK=1$ die empfangenen Nutzdaten und fordert mit der Ack-Number 2101 das nächste Segment an



TCP-Verbindungsabbau

- Der Verbindungsabbau ist dem Verbindungsaufbau ähnlich
- Statt des SYN-Bit kommt das FIN-Bit zum Einsatz, das anzeigt, dass keine Nutzdaten mehr vom Sender kommen

- 1 Client sendet den Abbauwunsch mit FIN=1
- 2 Server sendet eine Bestätigung mit ACK=1
- 3 Server sendet den Abbauwunsch mit FIN=1
- 4 Client sendet eine Bestätigung mit ACK=1



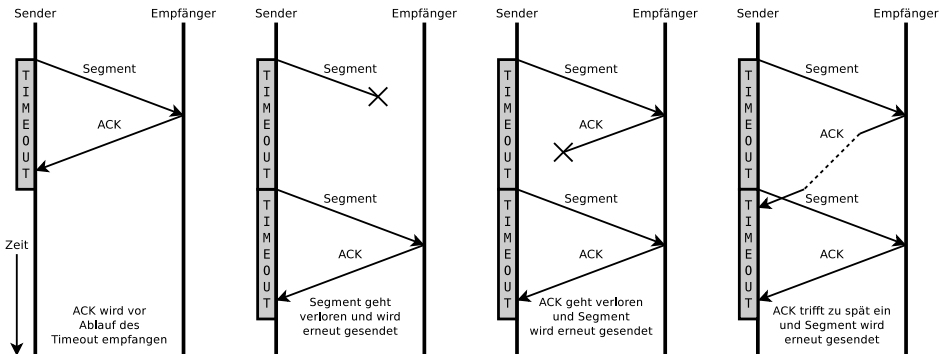
- Beim Verbindungsabbau werden keine Nutzdaten ausgetauscht

Zuverlässige Übertragung durch Flusskontrolle (*Flow Control*)

- Via Flusskontrolle steuert der Empfänger die **Sendegeschwindigkeit des Senders** dynamisch und stellt so und die **Vollständigkeit der Datenübertragung** sicher
 - Langsame Empfänger sollen nicht mit Daten überschüttet werden
 - Dadurch würden Daten verloren gehen
 - Während der Übertragung verlorene Daten werden erneut gesendet
- Vorgehensweise: **Sendewiederholungen**, wenn diese nötig sind
- Grundlegende Mechanismen:
 - **Bestätigungen** (Acknowledgements, ACK) als Feedback bzw. Quittung
 - **Zeitschranken** (Timeouts)
- Konzepte zur Flusskontrolle:
 - **Stop-and-Wait**
 - **Schiebefenster** (Sliding-Window)

Stop-and-Wait

- Nach dem Senden eines Segments wartet der Sender auf ein ACK
 - Kommt in einer bestimmten Zeit kein ACK an \Rightarrow Timeout
 - Timeout \Rightarrow Segment wird erneut gesendet



- Nachteil: Geringer Durchsatz gegenüber der Leitungskapazität

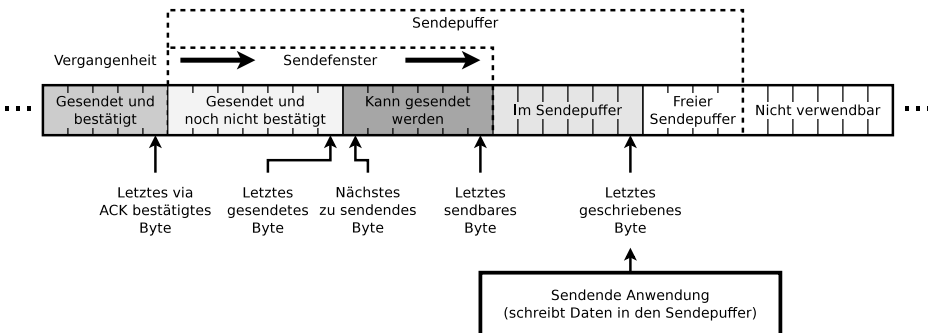
Das **Trivial File Transfer Protocol** (RFC 783) arbeitet nach dem Prinzip Stop-and-Wait

Schiebefenster (Sliding-Window)

- Ein **Fenster** ermöglicht dem Sender die Übertragung einer bestimmten Menge Segmente, bevor eine Bestätigung (Quittung) erwartet wird
 - Beim Eintreffen einer Bestätigung wird das Sendefenster verschoben und der Sender kann weitere Segmente aussenden
 - Der Empfänger kann mehrere Segmente auf einmal bestätigen
⇒ **kumulative Acknowledgements**
 - Beim Timeout übermittelt der Sender alle Segmente im Fenster neu
 - Er sendet also alles ab der letzten unbestätigten Sequenznummer erneut
- Ziel: Leitungs- und Empfangskapazität besser auslasten

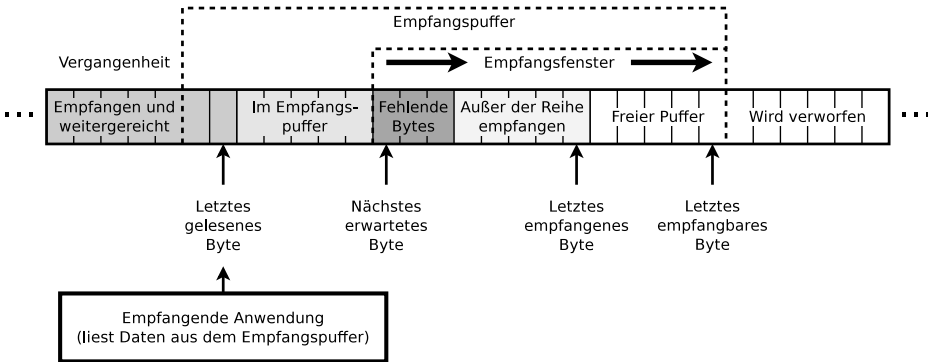
Schiebefenster – Vorgehensweise: Sender

- Der Sendepuffer enthält Daten der Anwendungsschicht, die...
 - bereits gesendet, aber noch nicht bestätigt wurden
 - bereits vorliegen, aber noch nicht gesendet wurden



Schiebefenster – Vorgehensweise: Empfänger

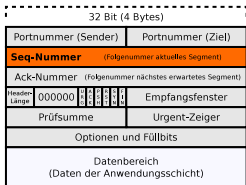
- Der Empfangspuffer enthält Daten für die Anwendungsschicht, die...
 - in der korrekten Reihenfolge vorliegen, aber noch nicht gelesen wurden
 - außer der Reihe angekommen sind



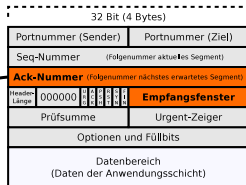
- Der Empfänger gibt dem Sender an, wie groß sein Empfangsfenster ist
 - Wichtig, um einen Pufferüberlauf zu vermeiden!

TCP-Flusskontrolle

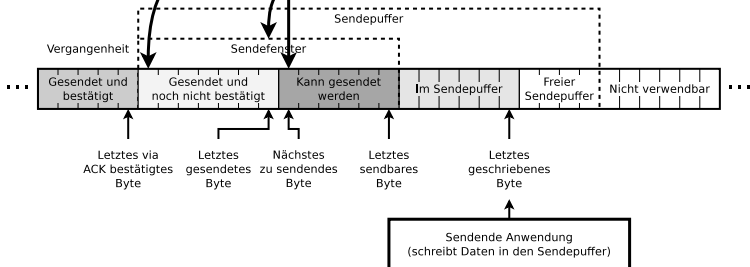
zu sendendes Segment



empfangenes Segment

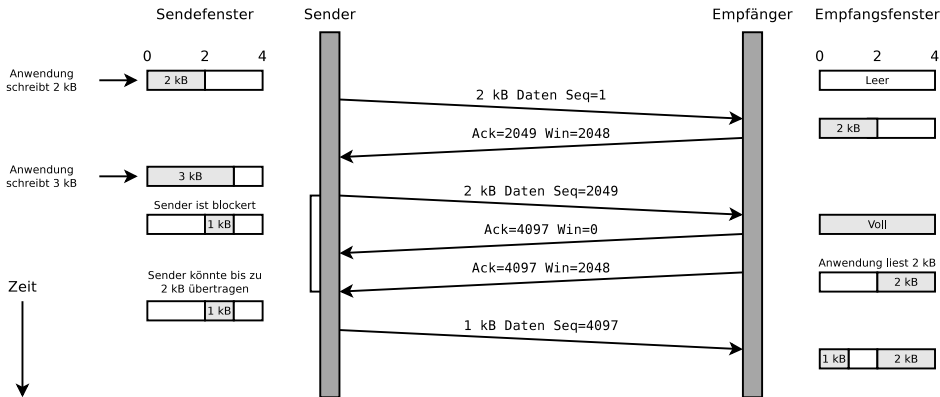


Empfänger teilt hier mit, wie viele Bytes er empfangen kann



Beispiel zur Flusskontrolle bei TCP

- Empfänger informiert in jedem Segment über das freie Empfangsfenster
- Ist das Empfangsfenster voll, ist der Sender blockiert, bis er vom Empfänger erfährt, dass im Empfangsfenster freier Speicher ist
- Wird Kapazität im Empfangsfenster frei \Rightarrow **Fensteraktualisierung**



Silly Window Syndrom

- Gefahr des **Silly Window Syndrom**, bei dem sehr viele kleine Segmente geschickt werden, was den Protokoll-Overhead vergrößert
 - Szenario:
 - Ein überlasteter Empfänger mit vollständig gefülltem Empfangspuffer
 - Sobald die Anwendung wenige Bytes (z.B. 1 Byte) aus dem Empfangspuffer gelesen hat, sendet der Empfänger ein Segment mit der Größe des freien Empfangspuffers
 - Der Sender sendet dadurch ein Segment mit lediglich 1 Byte Nutzdaten
 - Overhead: Mindestens 40 Bytes für die TCP/IP-Header jedes IP-Pakets (Nötig sind: 1 Segment mit den Nutzdaten, 1 Segment für die Bestätigung und eventuell noch ein Segment nur für die Fensteraktualisierung)
 - Lösungsansatz: **Silly Window Syndrom Avoidance**
 - Der Empfänger benachrichtigt den Sender erst über freie Empfangskapazität, wenn der Empfangspuffer mindestens zu 25% leer ist oder ein Segment mit der Größe MSS empfangen werden kann

Gründe für Überlastung

- Mögliche Ursachen für Überlastungen:

- ① **Empfängerkapazität**

- Der Empfänger kann die empfangen Daten nicht schnell genug verarbeiten und darum ist sein Empfangspuffer voll
 - Bereits gelöst durch die **Flusskontrolle**

- ② **Netzkapazität**

- Wird ein Computernetz über seine Kapazität hinaus beansprucht, kommt es zu Überlastungen \Rightarrow **Congestion Control**
 - Einzige hilfreiche Reaktion bei Überlastungen: **Datenrate reduzieren**
 - TCP versucht Überlastungen durch dynamische Veränderungen der Fenstergröße zu vermeiden \Rightarrow **Dynamisches Sliding Window**

- Es gibt nicht *die eine* Lösung für beide Ursachen

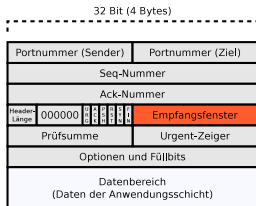
- Beide Ursachen werden getrennt angegangen

Anzeichen für Überlastungen der Netzkapazität

- Paketverluste durch Pufferüberläufe in Routern
- Lange Wartezeiten durch volle Warteschlangen in Routern
- Häufige Übertragungswiederholungen wegen Timeout oder Paket-/Segmentverlust

Lösungsansatz gegen Überlastung

- Der Sender verwaltet 2 Fenster
 - Advertised Receive Window** (*Empfangsfenster*)
 - Vermeidet Überlast beim Empfänger
 - Wird vom Empfänger angeboten (*advertised*)
 - Congestion Window** (*Überlastungsfenster*)
 - Vermeidet Überlastung des Netzes
 - Legt der Sender fest
- Das Minimum beider Fenster ist die maximale Anzahl Bytes, die der Sender übertragen kann
 - Beispiel:
 - Kann der Empfänger zum Beispiel gemäß seinem Empfangsfenster 20 kB empfangen, aber der Sender erkennt, dass bei mehr als 12 kB das Netz verstopft, dann sendet er nur 12 kB.
- Woher weiß der Sender wie leistungsfähig das Netz ist?
⇒ Wie ermittelt der Sender die Größe des Überlastungsfensters?

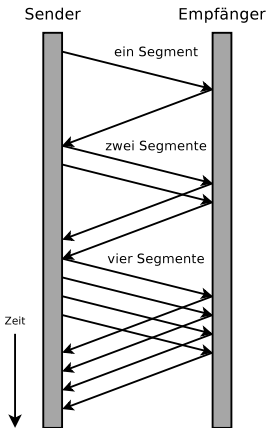


Größe des Überlastungsfensters festlegen

Sie wissen bereits...

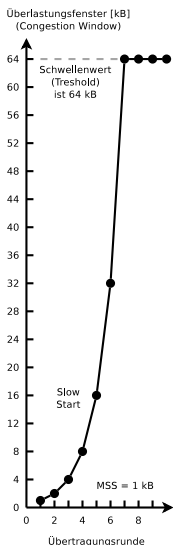
- Der Sender kann genau sagen, wie groß das Empfangsfenster ist
 - Grund: Der Empfänger teilt es ihm mit jedem Segment mit
-
- Problem für den Sender: **Wie groß ist das Überlastungsfenster?**
 - Der Sender weiß zu keiner Zeit sicher, wie leistungsfähig das Netz ist
 - Die Leistungsfähigkeit der Netze ist nicht statisch
 - Sie hängt u.a. von der Auslastung und von Netzstörungen ab
 - Lösungsweg: Der Sender muss sich an das Maximum dessen, was das Netzwerk übertragen kann, **herantasten**

Überlastungsfenster festlegen – Verbindungsaufbau



- Beim Verbindungsaufbau initialisiert der Sender das Überlastungsfenster auf die maximale Segmentgröße (MSS)
- Vorgehensweise:
 - 1 Segment mit der Größe MSS senden
 - Wird Empfang des Segments vor dem Timeout bestätigt, wird das Überlastungsfenster verdoppelt
 - 2 Segmente mit der Größe MSS senden
 - Wird der Empfang beider Segmente vor dem Timeout bestätigt, wird das Überlastungsfenster erneut verdoppelt
 - usw.

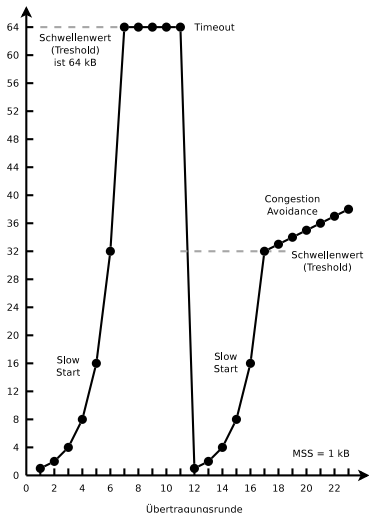
Überlastungsfenster festlegen – Slow Start



- Das Überlastungsfenster wächst exponentiell bis. . .
 - das vom Empfänger festgelegte Empfangsfenster erreicht ist
 - oder der **Schwellenwert** (Threshold) erreicht ist
 - oder es zum Timeout kommt
- Die exponentielle Wachstumsphase heißt **Slow Start**
 - Grund: Die niedrige Senderate des Senders am Anfang
- Hat das Überlastungsfenster die Größe des Empfangsfensters erreicht, wächst es nicht weiter
- Der Schwellenwert ist am Anfang der Übertragung 2^{16} Byte = 64 kB, damit er zu Beginn keine Rolle spielt
 - Das Empfangsfenster ist maximal $2^{16} - 1$ Bytes groß
 - Ist durch die Größe des Datenfelds **Empfangsfenster** im TCP-Header festgelegt

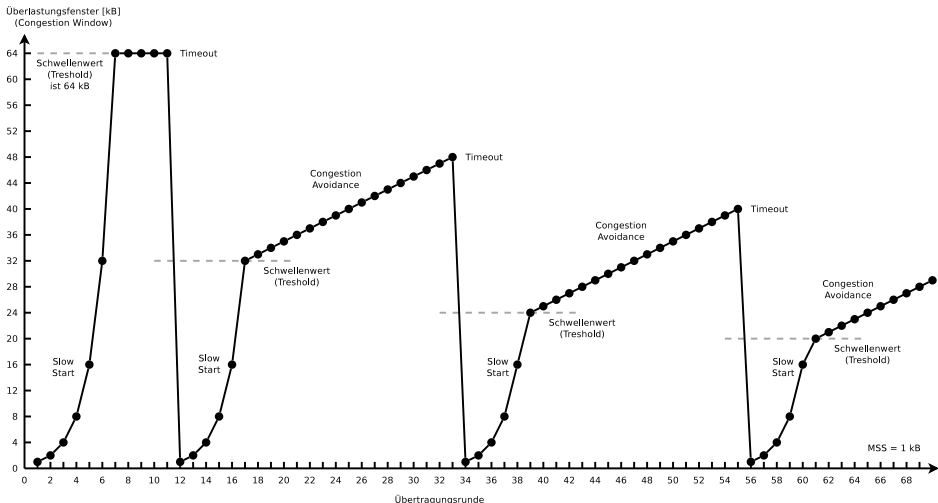
Überlastungsfenster festlegen – Congestion Avoidance

Überlastungsfenster [kB]
(Congestion Window)



- Kommt es zum Timeout, wird...
 - der Schwellenwert auf die Hälfte des Überlastungsfensters gesetzt
 - und das Überlastungsfenster auf die Größe 1 MSS reduziert
- Es folgt erneut die Phase Slow Start
 - Wird der Schwellenwert erreicht, wächst das Überlastungsfenster linear bis...
 - das vom Empfänger festgelegte Empfangsfenster erreicht ist
 - oder es zum Timeout kommt
- Die Phase des linearen Wachstums heißt **Congestion Avoidance**

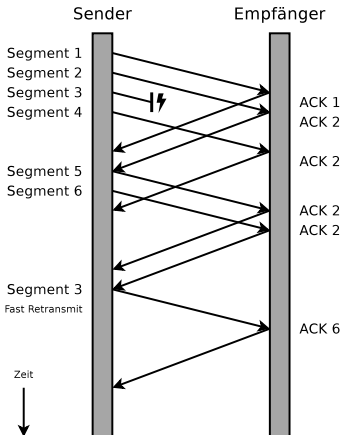
Mögliche Fortführung des Beispiels



Gründe für einen Timeout und sinnvolles Vorgehen

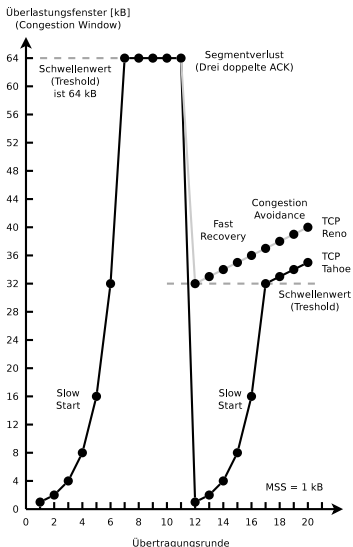
- Ein **Timeout** kann verschiedene Gründe haben
 - Überlast (\implies Verzögerung)
 - Verlust der Sendung
 - Verlust der Bestätigung (ACK)
- Nicht nur Verzögerungen durch Überlast, sondern auch jedes Verlustereignis reduziert das Überlastungsfenster auf 1 MSS
 - Entspricht dem Vorgehen der veralteten TCP-Version *Tahoe* (1988)
- Modernere TCP-Versionen unterscheiden zwischen...
 - Timeout wegen Netzüberlast
 - und **mehrfachem Eintreffen von Bestätigungen** (ACKs) wegen Verlustereignis

Fast Retransmit



- Geht ein Segment verloren, entsteht im Datenstrom beim Empfänger eine *Lücke*
 - Der Empfänger sendet bei jedem weiteren nach dieser Lücke empfangenen Segment ein ACK für das Segment vor dem verlorenen Segment
- Beim Segmentverlust ist eine Reduzierung des Überlastungsfensters auf 1 MSS unnötig
 - Grund: Für einen Segmentverlust ist nicht zwingend Überlastung verantwortlich
- TCP *Reno* (1990) sendet nach **dreimaligem Empfang eines doppelten ACK** das verlorene Segment neu
 ⇒ **Fast Retransmit**

Fast Recovery



- TCP *Reno* vermeidet auch die Phase Slow Start nach dreimaligem Empfang eines doppelten ACK
 ⇒ **Fast Recovery**
- Das Überlastungsfenster wird nach dreimaligem Empfang eines doppelten ACK direkt auf den Schwellenwert gesetzt
 - Das Überlastungsfenster wächst mit jeder bestätigten Übertragung linear. . .
 - bis das vom Empfänger festgelegte Empfangsfenster erreicht ist
 - oder es zum Timeout kommt

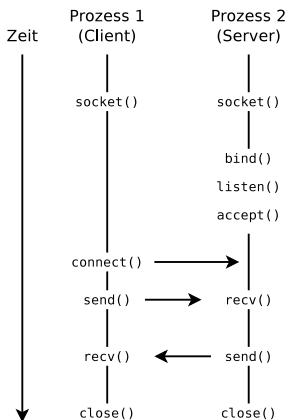
Additive Increase / Multiplicative Decrease (AIMD)

- AIMD ist das Prinzip/Konzept der Überlastkontrolle bei TCP
 - **Rasche Reduzierung** des Überlastungsfensters nach Timeout oder Verlustereignis und **langsames (lineares) Anwachsen** des Überlastungsfensters
- Grund für **aggressive Senkung** und **konservative Erhöhung** des Überlastungsfensters:
 - Die Folgen eines zu großen Überlastungsfensters sind schlimmer als die eines zu kleinen Fensters
 - Ist das Fenster zu klein, bleibt verfügbare Bandbreite ungenutzt
 - Ist das Fenster zu groß, gehen Segmente verloren und müssen erneut übertragen werden
 - Das vergrößert die Überlastung des Netzes noch mehr!
- Möglichst rasch muss der Zustand der Überlastung verlassen werden
 - Darum wird die Größe des Überlastungsfensters deutlich reduziert

Zusammenfassung zu Flusskontrolle und Überlastkontrolle

- Mit **Flusskontrolle** versucht TCP die Bandbreite eines verbindungslosen Netzes (\implies IP) effizient zu nutzen
 - Schiebefenster beim Sender (**Sendefenster**) und Empfänger (**Empfangsfenster**) dienen als Puffer zum Senden und Empfangen
 - Der Empfänger kontrolliert das Sendeverhalten des Senders
- Gründe für Überlastungen: **Empfangskapazität** und **Netzkapazität**
 - Empfangsfenster vermeidet Überlast beim Empfänger
 - Überlastungsfenster vermeidet Überlastung des Netzes
 - Effektiv verwendetes Fenster = Minimum beider Fenster
- Versuch der Maximierung der Netzauslastung und der schnellen Reaktion bei Überlastungsanzeichen
 - Prinzip des **Additive Increase / Multiplicative Decrease** (AIMD)

Verbindungsorientierte Kommunikation mit Sockets – TCP



• Client

- Socket erstellen (`socket`)
- Client mit Server-Socket verbinden (`connect`)
- Daten senden (`send`) und empfangen (`recv`)
- Socket schließen (`close`)

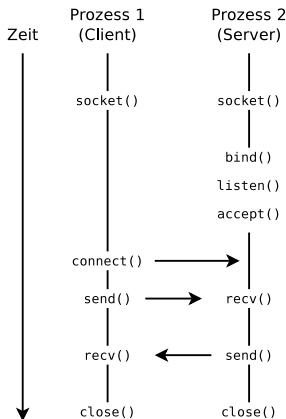
• Server

- Socket erstellen (`socket`)
- Socket an einen Port binden (`bind`)
- Socket empfangsbereit machen (`listen`)
 - Richtete eine Warteschlange für Verbindungen mit Clients ein
- Server akzeptiert Verbindungen (`accept`)
- Daten senden (`send`) und empfangen (`recv`)
- Socket schließen (`close`)

Sockets via TCP – Beispiel (Server)

```

1  #!/usr/bin/env python
2  # -*- coding: iso-8859-15 -*-
3  # Echo Server via TCP
4  import socket                # Modul socket importieren
5  HOST = ''                    # '' = alle Schnittstellen
6  PORT = 50007                 # Portnummer des Servers
7
8  # Socket erzeugen und Socket-Deskriptor zurückliefern
9  sd = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
10 # Socket an Port binden
11 sd.bind((HOST, PORT))
12 # Socket empfangsbereit machen
13 # Max. Anzahl Verbindungen = 1
14 sd.listen(1)
15 # Socket akzeptiert Verbindungen
16 conn, addr = sd.accept()
17
18 print 'Connected by', addr
19
20 while 1:                     # Endlosschleife
21     data = conn.recv(1024)    # Daten empfangen
22     if not data: break        # Endlosschleife abbrechen
23     # Empfangene Daten zurücksenden
24     conn.send(data)
25
26 sd.close()                   # Socket schließen
    
```



```
$ python tcp_server.py
```

Sockets via TCP – Beispiel (Client)

```

1  #!/usr/bin/env python
2  # -*- coding: iso-8859-15 -*-
3  # Echo Client via TCP
4  # Modul socket importieren
5  import socket
6
7  HOST = 'localhost'           # Hostname des Servers
8  PORT = 50007                 # Portnummer des Servers
9
10 # Socket erzeugen und Socket-Deskriptor zurückliefern
11 sd = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
12 # Mit Server-Socket verbinden
13 sd.connect((HOST, PORT))
14
15 sd.send('Hello, world')      # Daten senden
16 data = sd.recv(1024)        # Daten empfangen
17 sd.close()                  # Socket schließen
18
19 # Empfangene Daten ausgeben
20 print 'Empfangen:', repr(data)

```

```

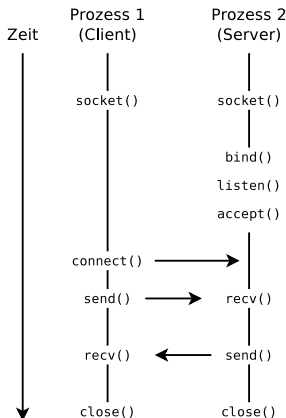
$ python tcp_client.py
Empfangen: 'Hello, world'

```

```

$ python tcp_server.py
Connected by ('127.0.0.1', 49898)

```



Denial of Service-Attacken via SYN-Flood

- Ziel des Angriffs: Dienste oder Server unerreichbar machen
- Ein Client sendet viele Verbindungsanfragen (**SYN**), antwortet aber nicht auf die Bestätigungen (**SYN ACK**) des Servers mit **ACK**
- Der Server wartet einige Zeit auf die Bestätigung des Clients
 - Es könnten ja Netzwerkprobleme die Bestätigung verzögern
 - Während dieser Zeit werden die Client-Adresse und der Status der unvollständigen Verbindung im Speicher des Netzwerkstacks gehalten
- Durch das Fluten des Servers mit Verbindungsanfragen wird die Tabelle mit den TCP-Verbindungen im Netzwerkstack komplett gefüllt
⇒ Der Server kann keine neuen Verbindungen mehr aufbauen
- Der Speicherverbrauch auf dem Server kann so groß werden, dass der Hauptspeicher komplett gefüllt wird und der Server abstürzt
- Gegenmaßnahme: Echtzeitanalyse des Netzwerks durch intelligente Firewalls