

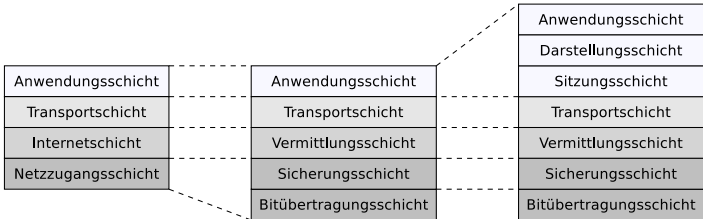
Transportschicht

- 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

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
- Transportprotokolle können die negativen Eigenschaften der Netze in die (positiven) Eigenschaften umwandeln, die für Interprozesskommunikation (Ende-zu-Ende-Kommunikation) nötig sind
- Die am häufigsten verwendeten Transportprotokolle:
 - **UDP**
 - **TCP**
- Adressierung erfolgt in der Transportschicht mit **Sockets**

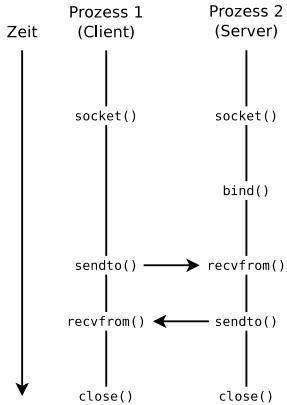
Ports (2/2)

- Die Portnummern sind in 3 Gruppen unterteilt:
 - 0 bis 1023 (*Well Known Ports*)
 - Diese sind Anwendungen fest zugeordnet und allgemein bekannt
 - 1024 bis 49151 (*Registered Ports*)
 - Anwendungsentwickler können sich Portnummern in diesem Bereich für eigene Anwendungen registrieren
 - 49152 bis 65535 (*Private Ports*)
 - Sind nicht registriert und können frei verwendet werden

- Verschiedene Anwendungen können im Betriebssystem gleichzeitig identische Portnummern verwenden, wenn Sie über unterschiedliche Transportprotokolle kommunizieren
- Zudem gibt es Anwendungen, die Kommunikation via TCP und UDP über eine einzige Portnummer realisieren
- Beispiel: Domain Name System – DNS (siehe Foliensatz 10)

- Die Well Known Ports und die Registered Ports werden durch die Internet Assigned Numbers Authority (IANA) vergeben
- Unter Linux/UNIX existiert die Datei /etc/services
 - Hier sind Anwendungen (Dienste) den Portnummern zugeordnet
- Unter Windows: %WINDIR%\system32\drivers\etc\services

Verbindungslose Kommunikation mit Sockets – UDP



• Client

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

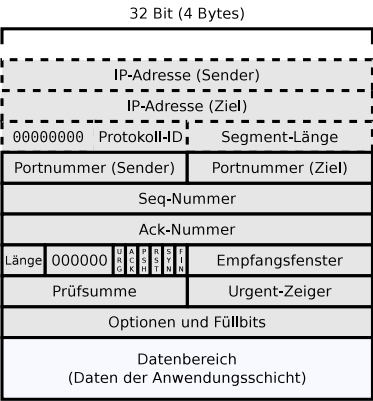
• Server

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

UDP-Sockets können als **blockierend** (Standard) und **nicht-blockierend** implementiert werden

- Wird sendto auf einen nicht blockierenden Socket ausgeführt, wird eine Fehlermeldung EAGAIN oder EWOULDBLOCK ausgegeben, wenn der Sendepuffer nicht über ausreichend freie Kapazität verfügt. Der Prozess wird dadurch aber nicht blockiert und kann zu einem späteren Zeitpunkt versuchen, Daten in den Sendepuffer zu schreiben
- Wird recvfrom auf einen nicht-blockierenden Socket ausgeführt, wird eine Fehlermeldung EAGAIN oder EWOULDBLOCK ausgegeben, wenn der Empfangspuffer keine Daten enthält. Der Prozess ist dadurch aber nicht blockiert und kann zu einem späteren Zeitpunkt versuchen, Daten aus dem Empfangspuffer zu lesen

Aufbau von TCP-Segmenten (5/5)



- 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. . .

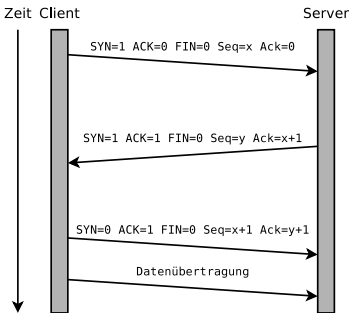
- Jedes Segment hat eine eindeutige Folgenummer (**Sequenznummer**)
- Die Sequenznummer eines Segments ist die Position des ersten Bytes des Segments im Bytestrom

- 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 (**Dreibege-Handshake**) tauschen beide Kommunikationspartner in drei Schritten Kontrollinformationen aus
 - So ist garantiert, dass der jeweilige Partner existiert und Daten annimmt

TCP-Verbindungsaufbau (Dreizege-Handshake)

- Der Server wartet passiv auf eine ankommende Verbindung

- 1 Client sendet ein Segment mit SYN=1 und fordert damit zur Synchronisation der Folgenummern auf
⇒ *Synchronize*
- 2 Server sendet als Bestätigung ein Segment mit ACK=1 und fordert mit SYN=1 seinerseits zur Synchronisation der Folgenummern auf
⇒ *Synchronize Acknowledge*
- 3 Client bestätigt mit einem Segment mit ACK=1 und die Verbindung steht
⇒ *Acknowledge*



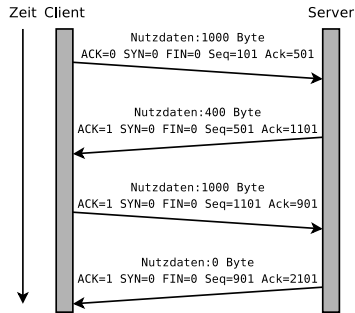
- Die Anfangs-Sequenznummern (x und y) werden zufällig bestimmt
- Beim Verbindungsaufbau werden keine Nutzdaten ausgetauscht!

TCP-Datenübertragung

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

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

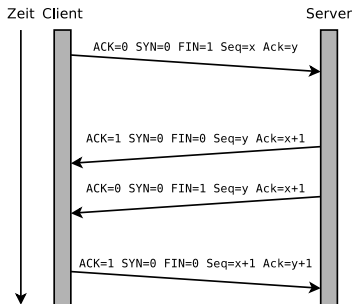
- 1 Client überträgt 1000 Byte Nutzdaten
- 2 Server bestätigt mit ACK=1 die empfangenen Nutzdaten und fordert mit der Ack-Nummer 1101 das nächste Segment an. Im gleichen Segment überträgt der Server 400 Bytes Nutzdaten
- 3 Client überträgt weitere 1000 Byte Nutzdaten. Zudem bestätigt er den Empfang der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



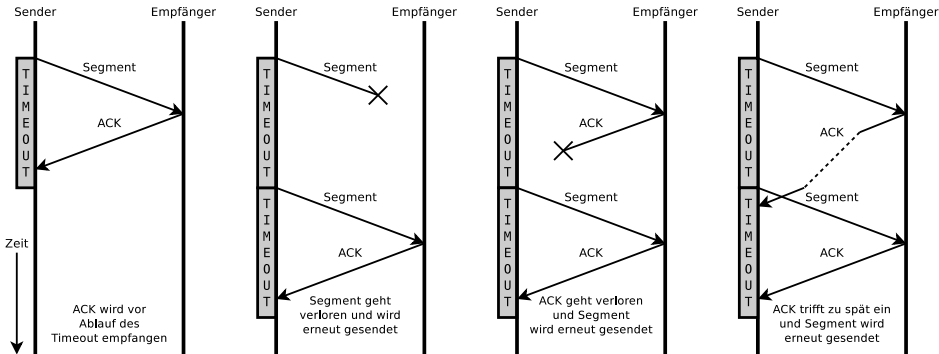
- Beim Verbindungsabbau werden keine Nutzdaten ausgetauscht

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 \implies Timeout
 - Timeout \implies Segment wird erneut gesendet



- **Nachteil:** Geringer Durchsatz gegenüber der Leitungskapazität

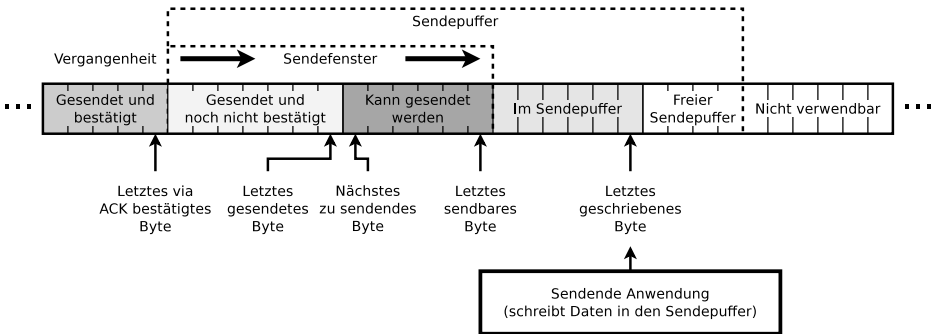
Das **Trivial File Transfer Protocol** (RFC 783) arbeitet nach dem Prinzip Stop-and-Wait

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

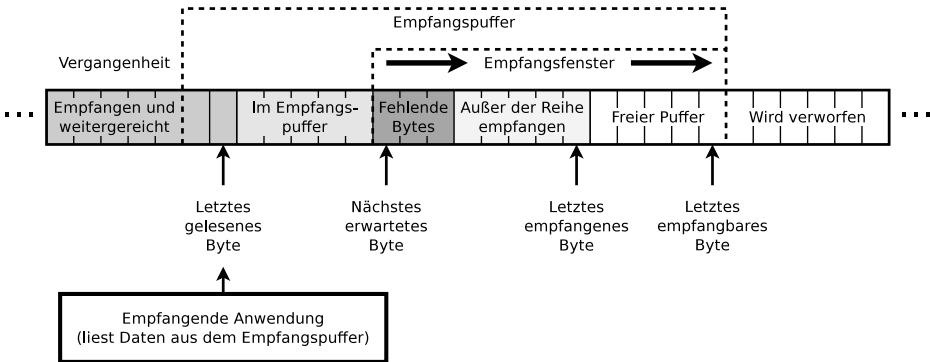
Schiebefenster – Vorgehensweise: Sender

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



Schiebefenster – Vorgehensweise: Empfänger

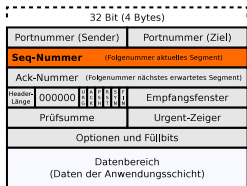
- Der Empfangspuffer enthält Daten für die Anwendungsschicht, die...
 - in der korrekten Reihenfolge vorliegen, aber noch nicht gelesen wurden
 - außer der Reihe angekommen sind



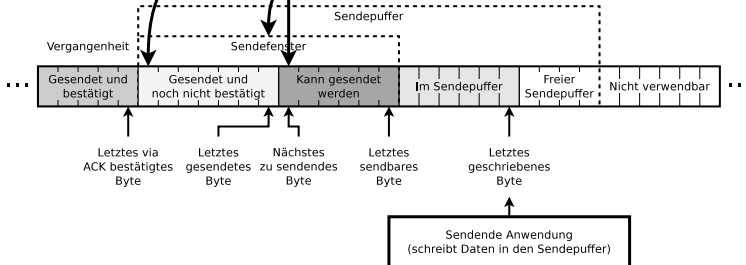
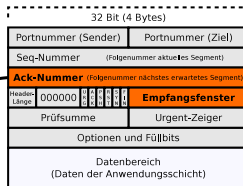
- Der Empfänger gibt dem Sender an, wie groß sein Empfangsfenster ist
 - Wichtig, um einen Pufferüberlauf zu vermeiden!

TCP-Flusskontrolle

zu sendendes Segment



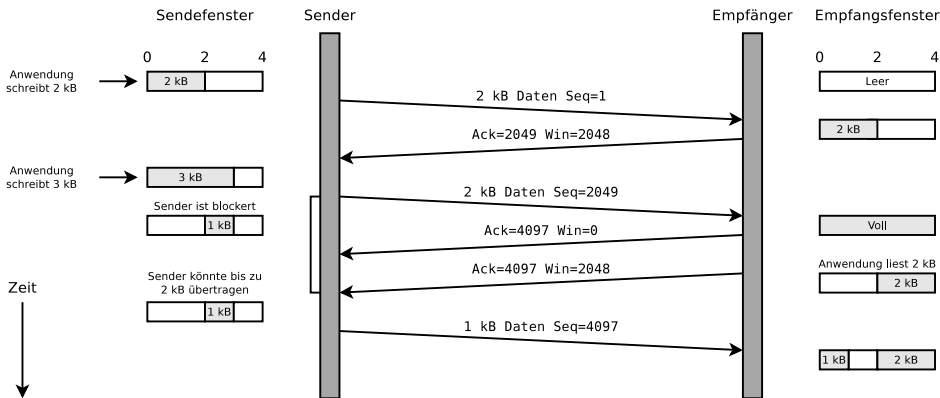
empfangenes Segment



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 \implies **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 \implies **Congestion Control**
 - Einzig hilfreiche Reaktion bei Überlastungen: **Datenrate reduzieren**
 - TCP versucht Überlastungen durch dynamische Veränderungen der Fenstergröße zu vermeiden \implies **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

① **Advertised Receive Window** (*Empfangsfenster*)

- Vermeidet Überlast beim Empfänger
- Wird vom Empfänger angeboten (*advertised*)

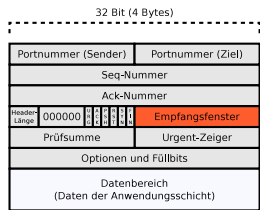
② **Congestion Window** (*Überlastungsfenster*)

- 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?

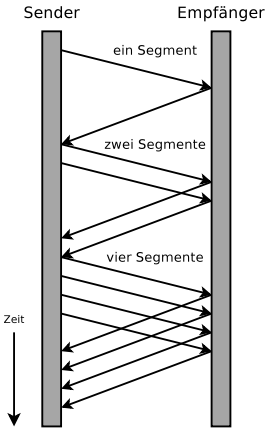
Größe des Überlastungsfensters festlegen

Sie wissen bereits...

- Der Sender kann genau sagen, wie groß das Empfangsfenster ist
- Grund: Der Empfänger teilt es ihm mit jedem Segment mit

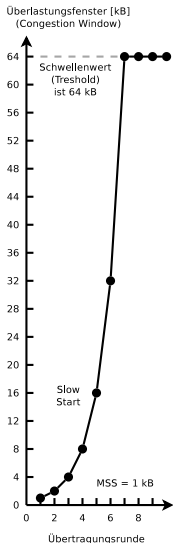
- Problem für den Sender: **Wie groß ist das Überlastungsfenster?**
 - Der Sender weiß zu keiner Zeit sicher, wie leistungsfähig das Netz ist
 - Die Leistungsfähigkeit der Netze ist nicht statisch
 - Sie hängt u.a. von der Auslastung und von Netzstörungen ab
- Lösungsweg: Der Sender muss sich an das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

Überlastungsfenster festlegen – Verbindungsaufbau



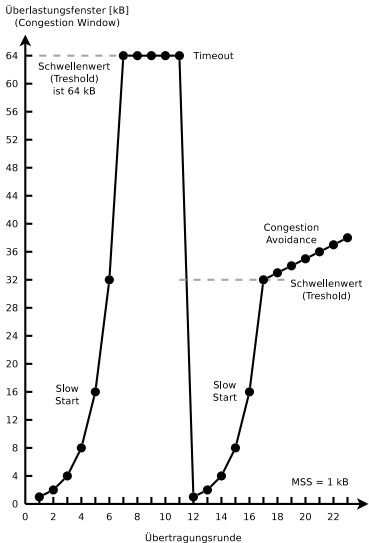
- Beim Verbindungsaufbau initialisiert der Sender das Überlastungsfenster auf die maximale Segmentgröße (MSS)
- Vorgehensweise:
 - 1 Segment mit der Größe MSS senden
 - Wird Empfang des Segments vor dem Timeout bestätigt, wird das Überlastungsfenster verdoppelt
 - 2 Segmente mit der Größe MSS senden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

Überlastungsfenster festlegen – Slow Start



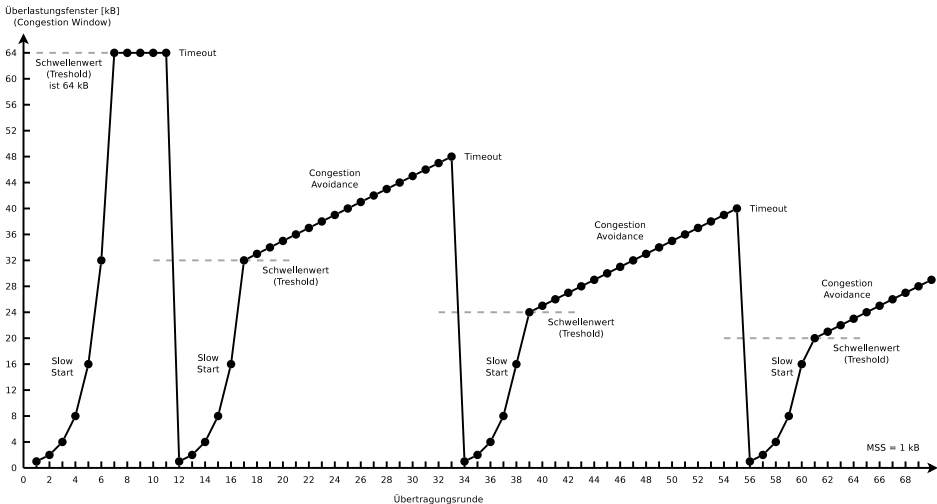
- Das Überlastungsfenster wächst exponentiell bis. . .
 - das vom Empfänger festgelegte Empfangsfenster erreicht ist
 - oder der **Schwellenwert** (Threshold) erreicht ist
 - oder es zum Timeout kommt
- Die exponentielle Wachstumsphase heißt **Slow Start**
 - Grund: Die niedrige Senderate des Senders 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



- 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**

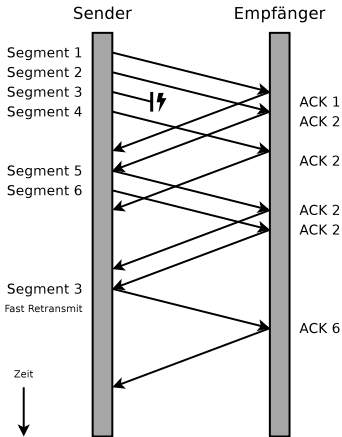
Mögliche Fortführung des Beispiels



Gründe für einen Timeout und sinnvolles Vorgehen

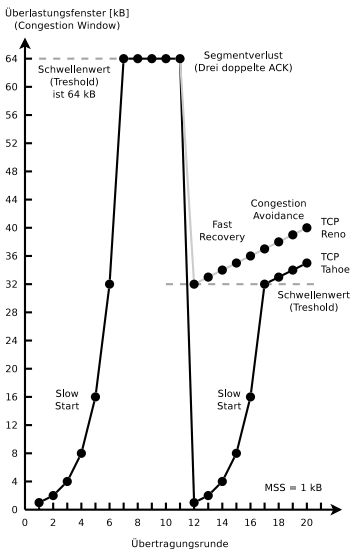
- Ein **Timeout** kann verschiedene Gründe haben
 - Überlast (\implies Verzögerung)
 - Verlust der Sendung
 - Verlust der Bestätigung (ACK)
- Nicht nur Verzögerungen durch Überlast, sondern auch jedes Verlustereignis reduziert das Überlastungsfenster auf 1 MSS
 - Entspricht dem Vorgehen der veralteten 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

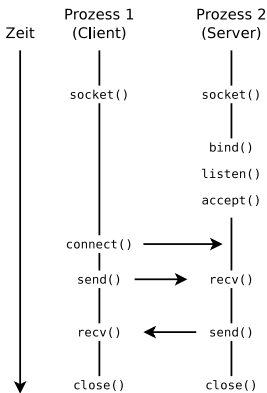
Additive Increase / Multiplicative Decrease (AIMD)

- 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



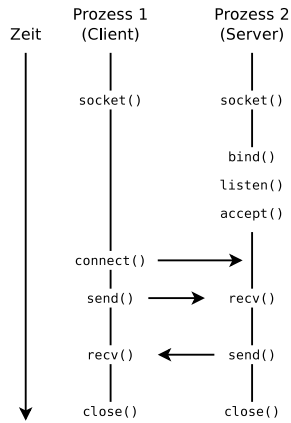
Auch TCP-Sockets können **blockierend** (Standard) und **nicht-blockierend** implementiert werden. Dadurch handelt sich das Verhalten der Funktionen `connect`, `accept`, `send` und `rcv`

- **Client**
 - Socket erstellen (`socket`)
 - Client mit Server-Socket verbinden (`connect`)
 - Daten senden (`send`) und empfangen (`rcv`)
 - Socket schließen (`close`)
- **Server**
 - Socket erstellen (`socket`)
 - Socket an einen Port binden (`bind`)
 - Socket empfangsbereit machen (`listen`)
 - Warteschlange für Verbindungsanfragen einrichten. Definiert wie viele Verbindungsanfragen gepuffert werden können
 - Verbindungsanforderung akzeptieren (`accept`)
 - Erste Verbindungsanforderung aus der Warteschlange holen
 - Daten senden (`send`) und empfangen (`rcv`)
 - Socket schließen (`close`)

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 un erreichbar 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