

Transportschicht (Fortsetzung TCP)

Nagle Algorithmus

TCP dient zur Übertragung eines kontinuierlichen Datenstroms. Werden die Zeichen von der Anwendungsschicht langsam eingeliefert, führt eine einfache Segmentierung zu Segmenten mit genau einem Byte Nutzdaten bei mindestens 40 Byte Header. Dies kann mit dem Nagle Algorithmus verhindert werden:

- ▶ Sind alle Daten im Sendefenster bestätigt, sende sofort das nächste Segment.
- ▶ Sind genug Daten zu senden (MSS erreicht), sende sofort ein weiteres Segment.
- ▶ Andernfalls warte, bis weitere Daten zu senden sind oder alle bereits gesendeten Segmente bestätigt sind.

Bemerkung: Benutzt die Gegenseite Delayed ACKs, kann der Nagle Algorithmus zu merklichen Verzögerungen der Übertragung führen!

TCP Optionen

Die Kodierung von TCP Optionen entspricht dem Verfahren im IP Header. Entsprechend sind sie wieder mit folgenden zwei Ausnahmen Type-Länge-Wert kodiert:

- 0 End of Options (1 Byte, kein Längensfeld)
- 1 No Operation (1 Byte, kein Längensfeld)
- 2 Maximum Segment Size (Länge 4, 2 Byte MSS)
- 3 Window Scale Factor, RFC1323 (Länge 3, 1 Byte Shift)
- 4 Selective ACK Permitted, RFC2018 (Länge 2)
- 5 Selective ACK, RFC2018
- 8 Timestamp, RFC1323 (Länge 10, 8 Byte für 2 Zähler)

Beispiel

SYN Paket eines Linux Hosts (nur TCP Rahmen):

Bytes	Interpretation
e65c 3039	Quellport 58972, Zielport 12345
d651 05e7	Initiale Sequenznummer 3595634151
0000 0000	ACK Sequenznummer 0
a002	Header Länge 40 Bytes, SYN
16d0	Window Size 5840 (mal Faktor)
4135 0000	Prüfsumme, URG nicht gesetzt
0204 05b4	MSS 1460 Bytes
0402	Selective ACK
080a 000e 715c 0000 0000	Timestamp TSval 946524
01	No Option
03 0306	Window Scale, Shift 6 (d.h. Faktor 64)

MSS Option, RFC793

Die MSS Option dient dazu, die Wahrscheinlichkeit zu senken, daß ein TCP Segment von der Vermittlungsschicht fragmentiert werden muß.

- ▶ Die MSS Option ist nur im SYN / SYN-ACK erlaubt.
- ▶ Üblicherweise berechnen Hosts die MSS als MTU des Interfaces, über das die Verbindung aufgebaut werden soll, abzüglich der minimalen Headerlänge (40 Bytes bei IPv4).
- ▶ Wird MSS nicht gesendet, wird der Default 536 angenommen.
- ▶ Beide Seiten verwenden das Minimum der eigenen MSS und der MSS der Gegenseite zur Segmentierung.
- ▶ Die Segmentierung kann bei bestehender Verbindung durch ICMP Destination Unreachable, Fragmentation Needed verkleinert werden (Path MTU Discovery).

Timestamp Option, RFC1323

- ▶ Die bisherige Zeit bis zur Antwort auf Segmente durch die Gegenseite ist die wichtigste Größe bei der Entscheidung, ob ein Segment erneut übertragen werden muß oder noch eine Antwort zu erwarten ist.
- ▶ Da es bei TCP kein einfaches Request-Response Modell gibt (vgl. ICMP Echo Request), sondern ein ACK mehrere Segmente bestätigen kann, werden geschätzte Antwortzeiten bei großen Empfangsfenstern schnell ungenau.
- ▶ Die Timestamp Option wird benutzt, wenn im SYN und SYN-ACK die Option gesetzt ist, also beide Hosts die Option verstehen.

Timestamp Algorithmus

- ▶ Die Option enthält 4 Byte TS Value (TSval) und 4 Byte TS Echo Reply (TSecr)
- ▶ TSecr ist nur gesetzt, wenn das ACK Flag gesetzt ist, sonst 0

Folgender Algorithmus wird von TCP verwendet, um TSecr zu bestimmen:

1. TS.Recent speichert den zu sendenden TSval, Last.ACK.sent speichert die ACK Sequenznummer des letzten gesendeten Segmentes.
2. Ist Last.ACK.sent innerhalb eines empfangenen Segmentes, setze TS.Recent auf den Zeitstempel im Segment.
3. Wird ein Segment mit Timestamp Option gesendet, setze TSecr auf TS.Recent.

Window Scale Option, RFC1323

Bei schnellen Anbindungen mit hoher Latenzzeit ist das 64KByte Window nicht ausreichend, um hohen Datendurchsatz zu erzielen.

- ▶ Der 1 Byte Shift Count der Window Scale Option gibt an, um wie viele Bit der Wert im Window Feld des TCP Rahmens nach links verschoben werden muß, um den Wert in Byte zu erhalten.
- ▶ Die Window Scale Option wird nur genutzt, wenn beide Hosts im SYN bzw. SYN-ACK ihren Shift Count angeben.
- ▶ Der maximale Shift Count ist 14, d.h. das Window ist maximal $65535 \cdot 2^{14}$ Byte groß.

Selective ACK (SACK), RFC2018

Geht ein Segment bei der Übertragung verloren, müssen in der Regel die Daten vom letzten bestätigten Byte (ACK Sequenznummer) startend ggfs. in anderer Segmentierung erneut übertragen werden (vgl. Go-Back-n ARQ).

Die TCP Selective ACK Option erlaubt es, Bereiche im Datenstrom zu bestätigen und damit implizit fehlende Bereiche erneut anzufordern (vgl. Selective-Repeat ARQ).

SACK wird nur verwendet, wenn beide Hosts es im SYN / SYN-ACK anbieten, d.h. Option 4 SACK Permitted senden.

Selective ACK Format

Kind 5	$2 + 8n$
1st Block Start	
1st Block End	
2nd Block Start	
2nd Block End	
n-th Block Start	
n-th Block End	

Selective ACK Felder

- ▶ Im Selective ACK werden erfolgreich empfangene Segmente bestätigt.
- ▶ Jeder Block im Selective ACK ist zusammenhängend und isoliert, d.h. das Byte vor dem übertragenen Anfang des Blocks ist noch nicht bestätigt, genauso wie das Byte, das durch das Ende des Blocks identifiziert wird (halboffenes Intervall).
- ▶ Die maximale Anzahl Blöcke ist 4, falls keine weiteren Optionen (speziell Timestamp) verwendet werden.
- ▶ Selective ACK ist weit verbreitet (Windows98, Solaris 8).

Duplicate Selective ACK (DSACK, vgl. RFC2883)

Um dem Sender Rückmeldung über unnötig wiederholte Segmente zu geben, ist der Selective ACK in RFC2883 erweitert worden. Die Optionen und deren Format ist aber unverändert.

- ▶ Nach RFC2018 müssen die Blöcke im SACK disjunkt sein und die Sequenznummern hinter ($\text{mod}2^{32}$) der ACK Sequenznummer liegen. Diese Beschränkung wird aufgehoben.
- ▶ Ein Block kann nach RFC2883 wie in RFC2018 einen noch nicht bestätigten Bereich im Datenstrom beschreiben.
- ▶ Liegt ein Block vor der ACK Sequenznummer oder innerhalb eines anderen Blocks, bezeichnet er einen mehrfach empfangenen Abschnitt.

TCP Flußkontrolle (vgl. RFC3390)

Bei großen Sende- und Empfangsfenstern kann es leicht zur Überlastung des Netzes kommen. Deshalb werden vier Algorithmen zur Flußkontrolle angewendet (Van Jacobson):

1. TCP Slow Start
2. Congestion Avoidance
3. Fast Retransmits
4. Fast Recovery

Algorithmen 1 und 2 sind bei TCP verbindlich.

Prinzipiell existieren zwei möglich Engpässe

- ▶ Kapazität des Netzwerks
- ▶ Kapazität des Empfängers

TCP Slow Start / Congestion Avoidance

Sei MSS die Maximum Segment Size der Gegenseite, $WINS$ deren Windowsize und FS die Anzahl nicht bestätigter Bytes.

- ▶ Der Kontext einer Verbindung hält die Werte Congestion Window ($CWND$) und Slow Start Threshold (S), er sendet maximal $\min(CWND, WINS)$ Bytes ohne ACK.
- ▶ Anfangs ist $CWND$ maximal so groß wie $4 \cdot MSS$ (oft $CWND = MSS$), S z.B. $WINS$
- ▶ Ist $CWND \leq S$ und wird ein Segment bestätigt, setze $CWND_+ = MSS$ (Slow Start)
- ▶ Ist $CWND > S$, inkrementiere für jede Round Trip Time $CWND$ um maximal MSS (Congestion Avoidance)
- ▶ Gehen Segmente verloren (Timeout oder doppelter ACK), dann setze $S := \max(2 \cdot MSS, FS/2)$ und $CWND := MSS$.

TCP Slow Start / Congestion Avoidance (2)

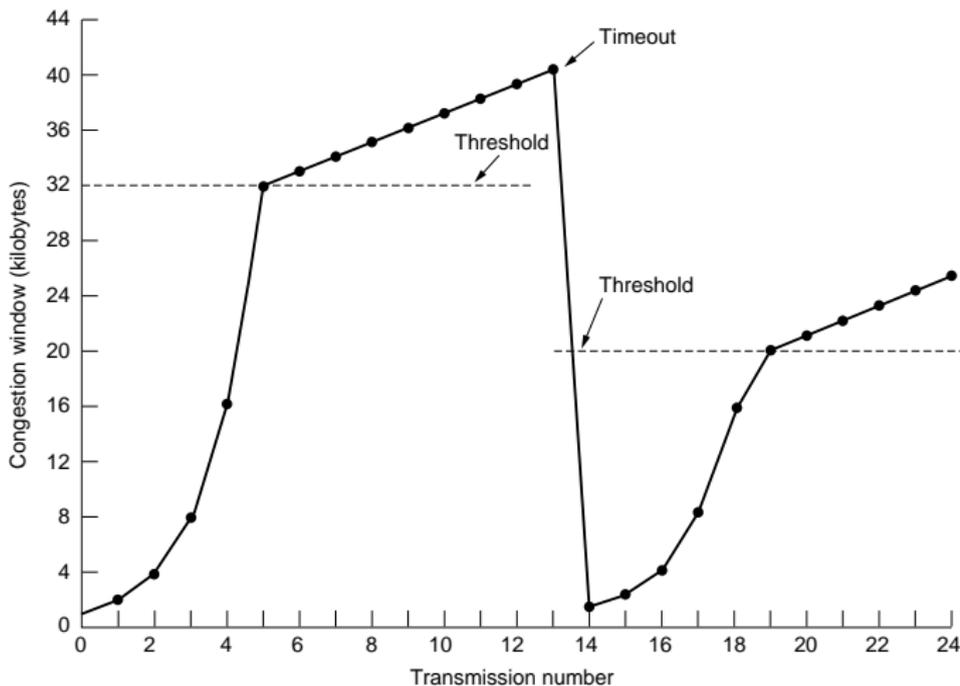
Slow Start

- ▶ Start mit sehr kleinem *CWND*
- ▶ exponentielles Wachstum von *CWND*, Verdopplung bei jeder Übertragung
- ▶ *CWND* wächst bis Slow Start Threshold *S* erreicht

Congestion Avoidance

- ▶ Wenn Threshold *S* mit Slow Start erreicht wurde, langsames lineares Wachstum bis entweder
- ▶ die Window Size *WINS* des Empfängers erreicht wurde
→ Empfängerkapazität erreicht, *CWND* bleibt konstant
- ▶ oder Fehler auftreten (Timeout, doppeltes ACK)
→ Netzwerkkapazität erreicht, *CWND* wird zurückgesetzt

TCP Slow Start / Congestion Avoidance (3)



(c) Tanenbaum, Computer Networks

Fast Retransmit / Fast Recovery

Bei großen Empfangsfenstern wird der Sender in der Regel durch doppelte ACK auf verlorene Segmente aufmerksam. Diese können auch durch Umordnung der Segmente entstanden sein.

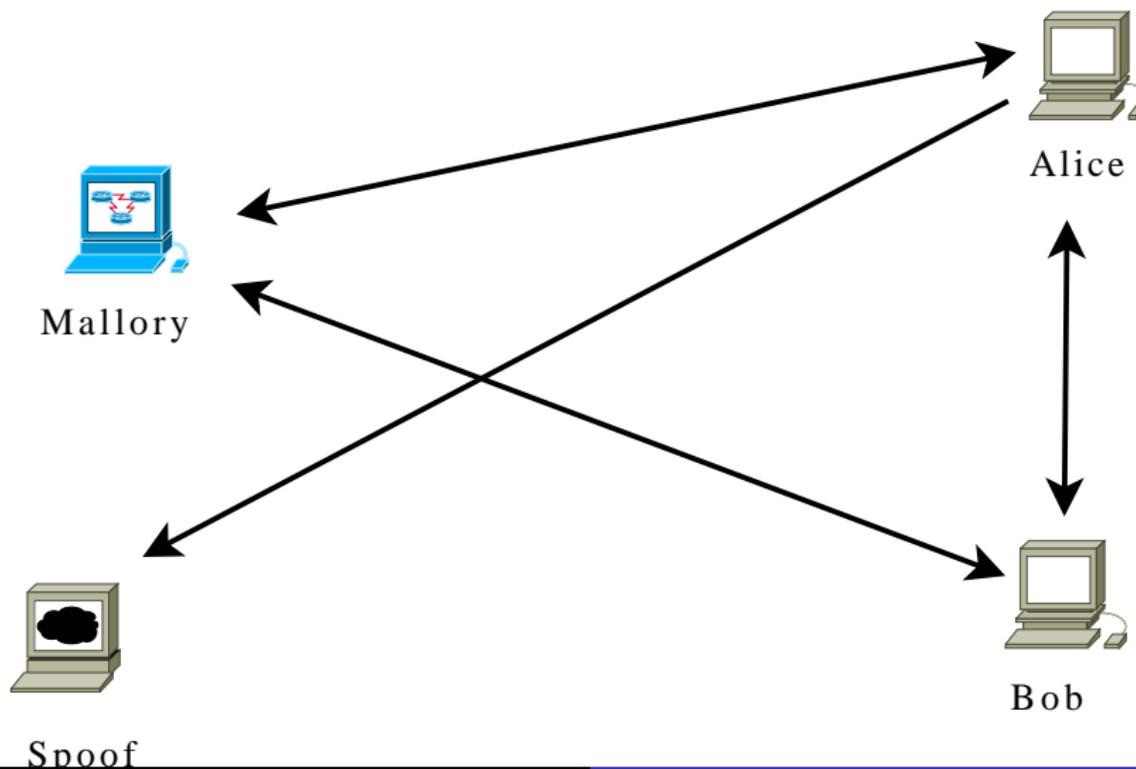
- ▶ Wird der dritte ACK mit gleicher ACK Sequenznummer empfangen, setze $S := \min(WINS, CWND)/2$
Sende das (vermutlich) fehlende Segment
Setze $CWDN := S + 3 \cdot MSS$
- ▶ Für jedes weitere ACK mit derselben Sequenznummer, setze $CWND_+ = 1$ und sende ein weiteres Segment.
- ▶ Beim nächsten neuen ACK setze $CWND := S$

Die Mitnick Attacke

Einleitung

- ▶ Angriff von Kevin Mitnick auf Rechner von Tsutomu Shimomura Weihnachten 1994.
- ▶ Tsutomu Shimomura war Spezialist für Rechnersicherheit bei SUN Microsystems
- ▶ John Markoff, damals Journalist der New York Times hat den Angriff bekannt gemacht.
- ▶ Die Details zum Angriff sind z.B. hier dokumentiert
<http://www.gulker.com/ra/hack/>

Beteiligte Systeme



Finger und Stealth SYN Scan

- ▶ **mallory#** finger -l @alice
Liste der Benutzer auf **alice**
- ▶ **mallory#** finger -l @bob
Liste der Benutzer auf **bob**
- ▶ **mallory#** finger -l [2]root@bob
Details einer Session des Benutzers **root** auf **bob**, z.B.:
Last login Thu Aug 5 09:55 (PDT) on ttyx1 from **alice**
- ▶ 1 mallory.51916 > alice.login: S
2 alice.login > mallory.51916: S ack
Test, ob auf **alice** der `rlogin` Dienst aktiviert ist

TCP Sequenznummern

- ▶ 1 mallory.1000 > bob.shell: S 1382726990(0)
2 bob.shell > mallory.1000: S 2021824000(0) ack 1382726991
3 mallory.1000 > bob.shell: R 1382726991(0)
- ▶ 1 mallory.999 > bob.shell: S 1382726991(0)
2 bob.shell > mallory.999: S 2021952000(0) ack 1382726992
3 mallory.999 > bob.shell: R 1382726992(0)
- ▶ 1 mallory.998 > bob.shell: S 1382726992(0)
2 bob.shell > mallory.998: S 2022080000(0) ack 1382726993
3 mallory.998 > bob.shell: R 1382726993(0)
- ▶ Die initialen Sequenznummern von **bob** wachsen pro Verbindungsaufbau um 128000

SYN Flood Angriff auf **alice.login**

- ▶ 1 spoof.600 > alice.login: S 1382726960(0)
- 2 alice.login > spoof.600: S 1022085000(0) ack 1382726961
- 3 spoof.601 > alice.login: S 1382726961(0)
- 4 alice.login > spoof.601: S 1022086000(0) ack 1382726962

...

- 15 spoof.607 > alice.login: S 1382726967
- 16 alice.login > spoof.607: S 1022092000(0) ack 1382726968

Bis hier wurden von **alice** SYN ACK erzeugt

- ▶ 17 spoof.608 > alice.login: S 1382726968(0)
- 18 spoof.609 > alice.login: S 1382726969(0)

Die Listen-Queue von **alice** ist voll, SYN ACK Pakete werden verworfen, sofern sie nicht ein gespeichertes SYN beantworten.

IP Spoofing Angriff auf bob.login

- ▶ **mallory** schickt ein SYN Paket mit falscher Quell-IP an **bob**
1 alice.login > bob.shell: S 1382727010:1382727010(0)
- ▶ **bob** antwortet **alice** 2 bob.shell > alice.login: S 2022208000(0)
ack 1382727011
- ▶ **alice** verwirft den SYN ACK, sendet kein RST
- ▶ **mallory** sieht den SYN ACK nicht, kann den Inhalt aber vorhersagen
- ▶ **mallory** sendet ACK, damit ist die Verbindung ESTABLISHED
3 alice.login > bob.shell: . ack 2022208001
- ▶ **mallory** sendet Schadcode
Daten: ..root.root.echo + + » /.rhosts
5 alice.login > bob.shell: F 1382727044(0) ack 2022208001

Konsequenzen

- ▶ `rlogin -l root bob` funktioniert ohne Authentifizierung aus dem Internet
- ▶ Kevin Mitnick wurde später zu 46 Monaten Haft verurteilt
- ▶ Moderne Betriebssysteme benutzen zufällige initiale Sequenznummern
- ▶ `rlogin/rsh/rexec` sind durch `ssh` weitgehend abgelöst
- ▶ SYN Flooding bleibt weiter ein Problem, SYN Cookies nur ein Behelf

User Datagram Protocol (UDP)

Grundlagen

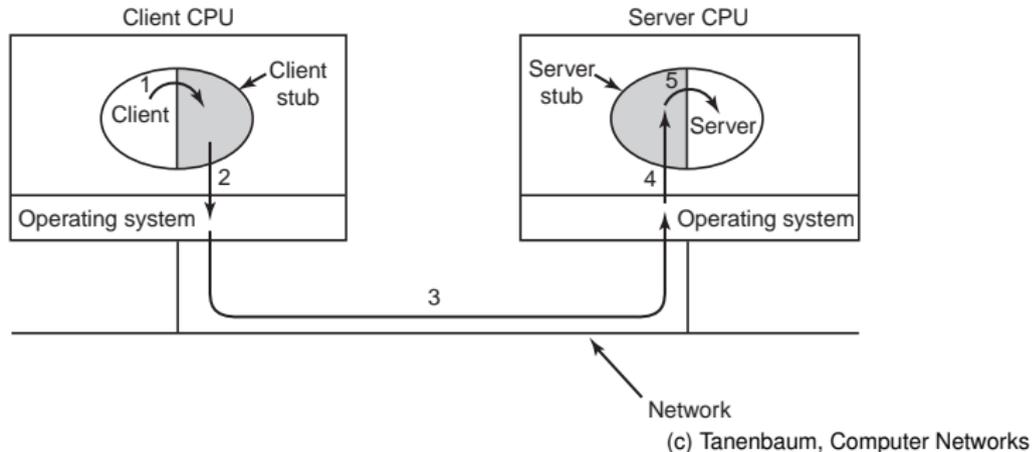
- ▶ Wert des Protokoll Feldes im IP Header ist 17 (vgl. RFC1700)
- ▶ Jedes UDP Datagramm erzeugt genau ein (möglicherweise fragmentiertes) IP Paket
- ▶ Die relevante Zusatzinformation durch UDP sind die Portnummern, die den Zweck des IP Pakets spezifizieren
- ▶ ungesicherter, verbindungsloser Transport (vgl. TCP: gesichert, verbindungsorientiert)
- ▶ UDP unterstützt Unicast und Multicast
- ▶ Daten sind bei UDP immer eine Folge von 8 Bit Zeichen

Anwendungsgebiete

- ▶ Schnittstelle zum IP Protokoll
- ▶ Demultiplexing von Übertragung auf der IP Schnittstelle auf verschiedene Ports
- ▶ UDP bietet keine Flußkontrolle, Fehlerkorrektur, Retransmission
- ▶ ermöglicht daher aber höheren Schichten ungefilterten Zugriff auf das IP Protokoll, sodaß höhere Schichten, z.B. die Anwendungsschicht, eigene Verfahren implementieren können
- ▶ Nutzung für
 - ▶ real-time Multimedia Anwendung (Streaming,...)
(z.B. Real-Time Protocol, RTP)
 - ▶ kurze Client-Server Requests (z.B. DNS, Domain Name Service)
(z.B. Remote Procedure Call, RPC)

Remote Procedure Call (RPC)

- ▶ ermöglicht Funktionsaufrufe auf anderen Rechnern
- ▶ identische Syntax wie auf lokalem Rechner
- ▶ problematisch sind übergebene Variablen (z.B. Pointer)



UDP Rahmen

Source Port	Destination Port
UDP Length	UDP Checksum
UDP Data	

Felder im UDP Header

- ▶ **Source Port:** Nummer, mit der die sendende Applikation vom Host identifiziert werden kann.
- ▶ **Destination Port:** Nummer, mit der die empfangende Applikation vom Host identifiziert werden kann.
- ▶ **UDP Length:** 16 Bit Länge des gesamten Datagramms in Bytes
- ▶ **Checksum:** 16 Bit Prüfsumme des gesamten Datagramms zuzüglich eines Pseudoheaders

UDP Prüfsummenberechnung

Dem UDP Datagramm wird ein Pseudoheader vorangestellt, bei ungerader Anzahl Bytes mit einer 0 aufgefüllt und dann das für IP üblicher Verfahren angewendet. Ist das Ergebnis 0, wird `0xFFFF` eingesetzt, ist das Feld 0, wurde es nicht berechnet.

Source IP Address		
Destination IP Address		
0	Protocol	UDP Length
Source Port		Destination Port
UDP Length		UDP Checksum
(Padded) UDP Data		

Größe von UDP Datagrammen

- ▶ Der UDP Header erlaubt 65507 Bytes Daten (8 Byte UDP Header, 20 Byte IP Header)
- ▶ Ein Host muß IP Pakete von mindestens 576 Byte empfangen können (RFC791), d.h. mindestens 548 Byte UDP Daten.
- ▶ Liest eine Anwendung erfolgreich empfangene Daten, kann das Datagramm gekürzt werden.
- ▶ Der Zwischenspeicher kann von der Anwendungsschicht vergrößert oder verkleinert werden.

Die meisten UDP basierten Protokolle gehen davon aus, daß maximal 512 Byte Daten pro Datagramm übertragen und verarbeitet werden können.

Berkeley Sockets (BSD Socket API)

Standard

- ▶ Entstanden an der Berkeley University of California (Berkeley Software Distribution, BSD) (Application Programming Interface, API)
- ▶ Veröffentlicht als Programmierschnittstelle des BSD4.2 (1983)
- ▶ Inzwischen standardisiert als IEEE Std. 1003.1-2004, als Single Unix Specification der Open Group und als ISO/IEC 9945:2002
- ▶ Die standardisierte C Schnittstelle ist aus verschiedenen Programmiersprachen nutzbar.

Umfang

Durch den Standard werden Schnittstellen beschrieben, die folgende Aspekte des Zugriffs auf die Transportschicht regeln:

- ▶ Adressierung, Adressumwandlung
- ▶ Verbindungsaufbau/Verbindungsabbau
- ▶ Konfiguration der Interna einer Verbindung
- ▶ Konfiguration der möglichen Zugriffsmethoden
- ▶ Datentransfer

Adressierung

Die BSD Socket API bietet Datenstrukturen und Konvertierungsfunktionen für Adressen verschiedener Formate:

- ▶ `inet_ntop`, `inet_pton`: Umwandlung von IP Adressen in binäre Darstellung (IPv4 und IPv6)
- ▶ `getaddrinfo`: Umwandlung eines symbolischen Namens in Adresse(n) der Transportschicht
- ▶ `getpeername`: Liefert die Adresse der Gegenseite einer Verbindung

Verbindungsaufbau/Verbindungsabbau

- ▶ `socket`: Belegt Resource (sog. Socket) für den Zugriff auf die Verbindung
- ▶ `bind`: Setzt Quelladresse der Verbindung
- ▶ `connect`: Setzt Zieladresse und baut ggfs. Verbindung auf (vgl. active open)
- ▶ `listen`, `accept`: Läßt Socket auf Verbindungsaufbau warten (vgl. passive open)
- ▶ `shutdown`: Schließt eine Richtung einer Verbindung
- ▶ `close`: Schließt eine Verbindung

Datentransfer

- ▶ `select`, `poll`: Testet, ob ein Ereignis auf dem Socket wartet.
- ▶ `send`, `sendto`, `sendmsg`: Sende Daten, abhängig vom gewählten Transport
- ▶ `recv`, `recvfrom`, `recvmsg`: Empfange Daten, abhängig vom gewählten Transport

Konfiguration

Die Konfiguration der Vermittlungsschicht und Transportschicht erfolgt über `setsockopt` und `fcntl`. Es folgt eine Auswahl der portablen Optionen:

- ▶ `SO_BROADCAST`: Erlaube Broadcasts bei UDP
- ▶ `SO_ERROR`: Lese Fehler mit `getsockopt`
- ▶ `SO_KEEPALIVE`: Schalte TCP Keepalive ein
- ▶ `SO_LINGER`: Warte, bis alle Daten bei einem `close` oder `shutdown` erfolgreich übertragen sind
- ▶ `SO_RCVBUF`: Größe des Empfangspuffers
- ▶ `SO_RCVLOWAT`: Mindestmenge an Daten, die zur Anwendung hochgereicht werden
- ▶ `SO_RCVTIMEO`: Timeout beim Lesen von Daten aus dem Empfangspuffer

Konfiguration (2)

- ▶ `SO_REUSEADDR`: Verwende Adresse neu, ohne 2 MSL Timeout abzuwarten
- ▶ `SO_SNDBUF`: Größe des Sendepuffers
- ▶ `SO_SNDLOWAT`: Minimale Datenmenge bei Sendeoperationen
- ▶ `SO_SNDTIMEO`: Timeout beim Senden
- ▶ `SO_TYPE`: Abfrage des Verbindungstyp/Protokolls

- ▶ `TCP_MAXSEG`: Setze MSS
- ▶ `TCP_NODELAY`: Schaltet Nagle Algorithmus aus
- ▶ `O_NONBLOCK`: Aufrufe auf dem Socket blockieren nicht mehr.

Client

```
from socket import *
from ReadUntilEOF import *

c_so = socket(AF_INET, SOCK_STREAM)
c_so.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
c_so.bind( ( "0.0.0.0", 7777 ) )

c_so.connect( ( "localhost", 6666 ) )

c_so.sendall("client request line 1\n")
c_so.sendall("client request line 2\n")
c_so.shutdown(SHUT_WR)

print readUntilEOF(c_so)

c_so.close()
```

Server

```
from socket import *
from ReadUntilEOF import *

a_so = socket(AF_INET, SOCK_STREAM)
a_so.bind( ( "localhost", 6666 ) )
a_so.listen(10)

s_so, addr = a_so.accept()
a_so.close()

print "Connection from ", addr
print readUntilEOF(s_so)
s_so.sendall("server response line 1\n")
s_so.sendall("server response line 2\n")
s_so.close()
```

Ablauf

- ▶ S: `a_so = socket(AF_INET, SOCK_STREAM)`
 - ▶ Erzeuge Socket für TCP Verbindung
- ▶ S: `a_so.bind(("localhost", 6666))`
 - ▶ Registriere Socket mit Port 6666 und Quell-IP 127.0.0.1
- ▶ S: `a_so.listen(10)`
 - ▶ Socket wechselt in den Zustand LISTEN
 - ▶ 10 Warteplätze für eingehende Verbindungen
- ▶ S: `s_so, addr = a_so.accept()`
 - ▶ Applikation wartet, bis eine vollständig aufgebaute Verbindung besteht

Ablauf

- ▶ C: `c_so = socket(AF_INET, SOCK_STREAM)`
 - ▶ Datenstruktur für TCP Verbindung wird angelegt
 - ▶ IP Header Version ist 4
 - ▶ IP Header Protocol ist 6 (TCP)
- ▶ C:
 - `c_so.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)`
 - ▶ Socket im Zustand `TIME_WAIT` kann benutzt werden
- ▶ C: `c_so.bind(("0.0.0.0", 7777))`
 - ▶ IP Header Source IP wird anhand der Routing Tabelle ermittelt
 - ▶ TCP Header Source Port ist 7777
 - ▶ Socket ist im Zustand `CLOSED`

Ablauf

- ▶ C: `c_so.connect(("localhost", 6666))`
 - ▶ IP Header Source IP ist 127.0.0.1
 - ▶ TCP Header Destination Port ist 6666
 - ▶ Client sendet
IP 127.0.0.1.7777 > 127.0.0.1.6666: S
2476749965:2476749965(0)
 - ▶ Socket ist im Zustand SYN_SENT
- ▶ S: Empfängt SYN
 - ▶ Daten werden in der Listen Queue gespeichert
 - ▶ Server antwortet
IP 127.0.0.1.6666 > 127.0.0.1.7777: S
2976505067:2976505067(0) ack 2476749966
 - ▶ Socket ist im Zustand SYN_RCVD

Ablauf

- ▶ C: Empfängt SYN-ACK
 - ▶ Prüft (Host,Port) Paar und Sequenznummer
 - ▶ Client antwortet
 IP 127.0.0.1.7777 > 127.0.0.1.6666: . ack 2976505068
 - ▶ Socket ist im Zustand ESTABLISHED
- ▶ S: Empfängt ACK
 - ▶ Holt Verbindungsdaten aus der Warteschlange
 - ▶ Socket ist im Zustand ESTABLISHED
 - ▶ `s_so, addr = a_so.accept()` liefert Filedescriptor

Ablauf

- ▶ `S: print "Connection from ", addr`
 - ▶ Gibt Source IP und Source Port aus
 - ▶ Die Daten könnten auch über `s_so.getpeername()` ausgelesen werden.
- ▶ `S: print readUntilEOF(s_so)`
 - ▶ Liest Daten vom Filedescriptor, bis EOF signalisiert wird

Ablauf

- ▶ C: `c_so.sendall("client request line 1\n")`
 - ▶ Schreibt 22 Bytes in den Puffer des Sockets
 - ▶ Client sendet
IP 127.0.0.1.7777 > 127.0.0.1.6666: P
2476749966:2476749988(22) ack 2976505068
- ▶ C: `c_so.sendall("client request line 2\n")`
 - ▶ Schreibt 22 Bytes in den Puffer des Sockets
 - ▶ Client sendet
IP 127.0.0.1.7777 > 127.0.0.1.6666: P
2476749988:2476750010(22) ack 2976505068
- ▶ C: `c_so.shutdown(SHUT_WR)`
 - ▶ Client sendet
IP 127.0.0.1.7777 > 127.0.0.1.6666: F
2476750010:2476750010(0) ack 2976505068
 - ▶ Socket ist im Zustand `FIN_WAIT_1`

Ablauf

- ▶ S: Empfängt "client request line 1\n"
 - ▶ Bestätigt mit
IP 127.0.0.1.6666 > 127.0.0.1.7777: . ack 2476749988
 - ▶ Applikation wartet weiter auf Daten
- ▶ S: Empfängt "client request line 2\n"
 - ▶ Bestätigt mit
IP 127.0.0.1.6666 > 127.0.0.1.7777: . ack 2476750010
 - ▶ Applikation wartet weiter auf Daten
- ▶ S: Empfängt FIN
 - ▶ Socket ist jetzt im Zustand CLOSE_WAIT
 - ▶ Applikation liest EOF vom Filedeskriptor
 - ▶ `s_so.sendall("server response line 1\n")`
 - ▶ Server sendet
IP 127.0.0.1.6666 > 127.0.0.1.7777: P
2976505068:2976505091(23) ack 2476750011

Ablauf

- ▶ S: `c_so.sendall("""server response line 2\n""")`
 - ▶ Schreibt 23 Bytes in den Puffer des Sockets
 - ▶ Server sendet
 IP 127.0.0.1.6666 > 127.0.0.1.7777: P
 2976505091:2976505114(23) ack 2476750011
- ▶ C: empfängt beide Zeilen
 - ▶ Bestätigt mit
 IP 127.0.0.1.7777 > 127.0.0.1.6666: . ack 2976505114
- ▶ S: `s_so.close()`
 - ▶ Server sendet
 IP 127.0.0.1.6666 > 127.0.0.1.7777: F
 2976505114:2976505114(0) ack 2476750011
- ▶ C: Empfängt FIN
 - ▶ Client bestätigt:
 IP 127.0.0.1.7777 > 127.0.0.1.6666: . ack 2976505115