FACHHOCHSCHULE
TECHNIKUM WIEN

Institut: Embedded Systems

Themengebiet: Embedded Networking

TCP/IP Client Server Architektur

Version 2.6, September 2011

Peter Balog, Thomas M. Galla, Franz Hollerer

Inhaltsverzeichnis

0. Übersicht

0.1. Lehrziele

Das primäre Lehrziel dieses Studienbriefes zum Thema **TCP/IP Client Server Architektur** ist die Vertiefung von theoretischem Wissen im Bereich der (Internet-) Kommunikation und die Erarbeitung von Kenntnissen und Fähigkeiten in der praktischen Programmierung von TCP/IP Applikationen mit der Programmiersprache C auf einer UNIX-basierten Plattform unter Verwendung des Socket-APIs.

Als sekundäres Lehrziel kann die Vertiefung im Bereich der Betriebssysteme und das praktische Anwenden der Systemprogrammierung an sich gesehen werden.

0.2. Lehrinhalt

Ausgehend von diesem Studienbrief, der, basierend auf einer komprimierten Zusammenfassung der relevanten TCP/IP Grundlagen, sehr konzeptionell sowohl die TCP/IP Client Server Architektur als auch die Programmierung von TCP/IP Applikationen betrachtet, werden in Folge problem- bzw. projektbasiert, unter Verwendung von Originaldokumentation (z.B. Manual Pages) und (Standard-) Literatur, die notwendigen Kenntnisse und Fähigkeiten selbständig erarbeitet, um das primäre Lehrziel zu erreichen.

0.3. Aufgaben, Übungen

Das Kapitel 5 enthält eine Sammlung lehrzielorientierter Fragen. Die Beantwortung sollte nach Durcharbeiten des Studienbriefes möglich sein. Fragen die mit einem * gekennzeichnet sind, können aus dem vorliegenden Studienbrief nicht direkt beantwortet werden. Mit entsprechenden Vorkenntnissen aus den Bereichen „Internetworking – TCP/IP Protokollfamilie“, „Programmieren in C“ und „Grundlagen von Betriebssystemen – Systemprogrammierung“ sollte die Beantwortung jedoch möglich sein.

0.4. Lernwegempfehlung

Arbeiten Sie den Studienbrief durch und versuchen Sie die anfallenden Aufgaben und die lehrzielorientierten Fragen zu lösen bzw. zu beantworten.

0.5. Literatur

[1] Brian „Beej“ Hall, „Beej’s Guide to Network Programming – Using Internet Sockets“, 1995-2009 (http://beej.us/guide/bgnet/)

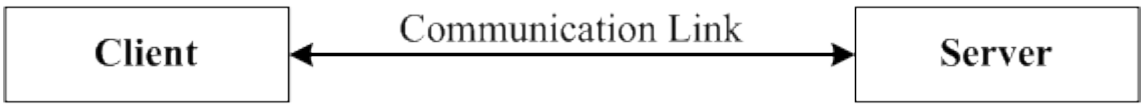
[2] W. Richard Stevens, „Advanced Programming in the UNIX Environment“, Addison Wesley, 1993, ISBN 0-201-56317-7

[3] W. Richard Stevens, „UNIX Network Programming - Volume 1, Networking APIs: Sockets and XTI), Prentice Hall, 1998, ISBN 0-13-490012-X

[4] relevante UNIX Manual Pages (siehe unter Selected SysCalls)

1. Client Server Architektur

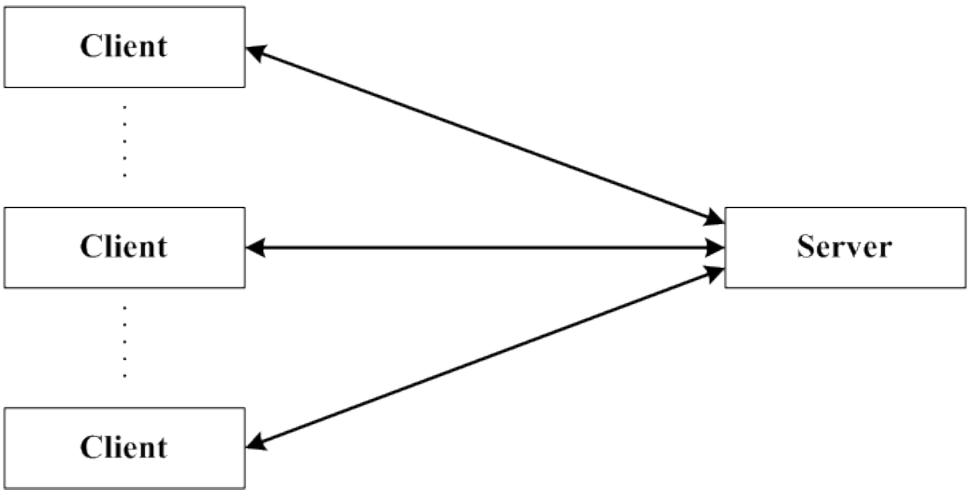
Üblicherweise lassen sich Netzwerk-Applikationen in die beiden Teile **Client** und **Server** strukturieren. Zwischen dem Client und dem Server besteht ein Kommunikationspfad.



Aus der täglichen Arbeit mit dem Internet sind uns zahlreiche Beispiele bekannt:

- der Web-Browser (Client) mit dem Web Server
- der FTP-Client mit dem FTP-Server

Normalerweise kommuniziert ein Client zu einer bestimmten Zeit nur mit einem Server. Für einen Server hingegen ist es nicht ungewöhnlich, dass er zu einer bestimmten Zeit mit mehreren Clients gleichzeitig eine aktive Verbindung hat.

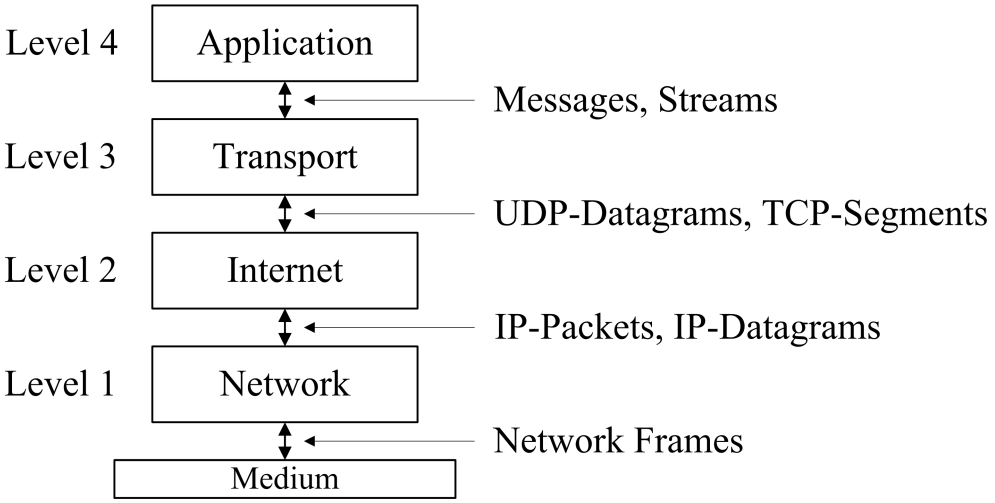


2. TCP/IP, Internet

Im Rahmen dieses Teils der Lehrveranstaltung beschäftigen wir uns speziell mit der Linux-basierten TCP/IP Client Server Programmierung. TCP (*Transmission Control Protocol*) und IP (*Internet Protocol*) sind die beiden wichtigsten bzw. zentralen Protokolle der **Internet-Protokollfamilie**. Dies ist der Grund dafür, dass die Bezeichnung **TCP/IP** üblicherweise als Synonym für die Internet-Protokollfamilie bzw. für die gesamte Thematik rund um das Internet verwendet wird. So spricht man z.B. bei „klassischem" NFS (*Network File System*) oft von einer TCP/IP-Applikationen, obwohl NFS eigentlich das UDP (*User Datagram Protocol*) anstelle des TCP auf der Transportschicht verwendet. Gleiches gilt für NTP (*Network Time Protocol*).

2.1. Schichtung der Internet Protokollfamilie

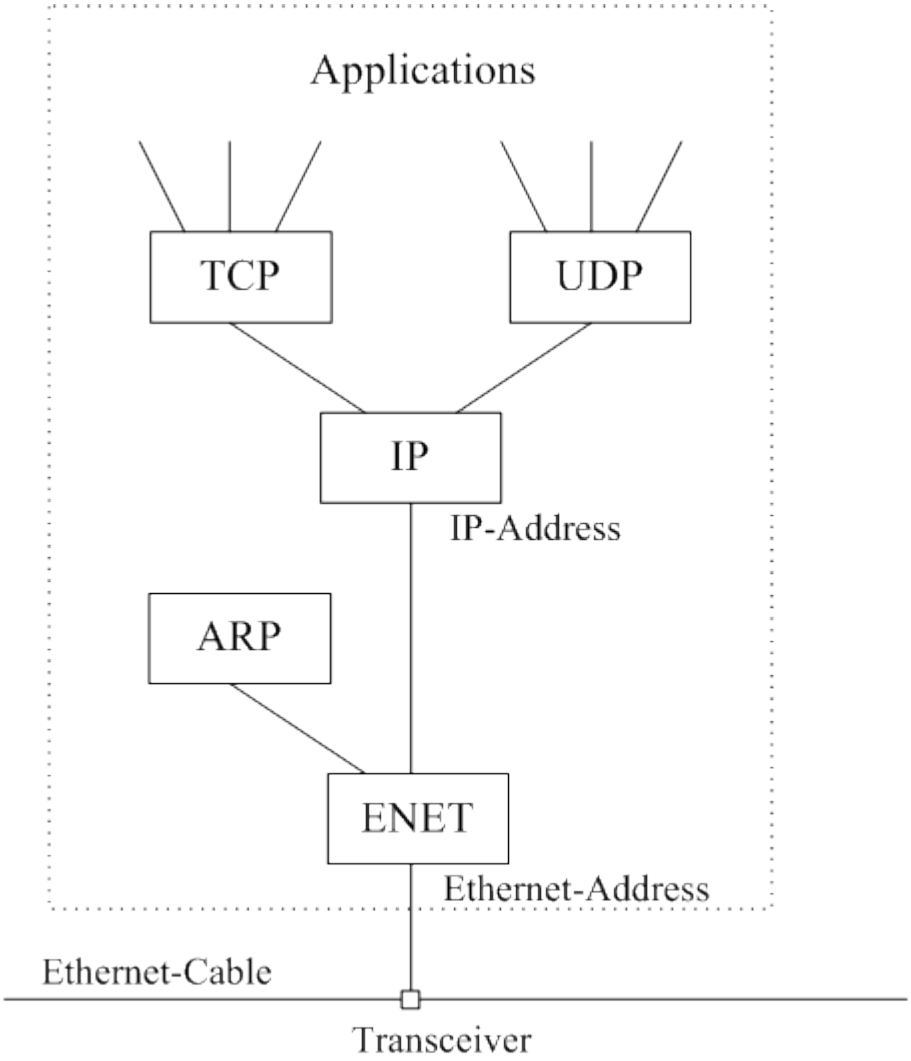
Das Internet basiert auf einem 4 Schichten Modell. Die Ebene 1 „Network" ist im Rahmen von Internet nicht definiert und stellt das „physikalische Netzwerk" (z.B. Ethernet) dar. Das IP-Protokoll liegt in der Ebene 2 während die Transportprotokolle TCP und UDP auf der Ebene 3 liegen.



In der Ebene 4 liegen die Internet-Applikationsprotokolle, wie z.B. FTP (File Transfer Protocol), Telnet (Terminal over Network), SMTP (Simple Mail Transport Protocol), SNMP (Simple Network Managment Protocol), und HTTP (Hypertext Transport Protocol).

2.2. Ein Knoten im Internet

Die folgende Abbildung zeigt, stark vereinfacht und schematisch, einen Internet-Knoten. Dabei handelt es sich z.B. um einen Linux-PC in einem Ethernet-LAN, welches selbst über einen Router (oder ganz allgemein ein Gateway) an das Internet angeschlossen ist.



Die Ebene 1 (Network) wird durch das Netzwerkmodul (Ethernet-Modul ENET) bedient. Darüber liegt in der Ebene 2 (Internet) das IP-Modul. Die Zuordnung zwischen der (logischen) Internet-Adresse und der (physikalischen) Ethernet-Adresse wird über das ARP-Modul (Address Resolution Protocol) bewerkstelligt. Auf der Ebene 3 liegen die beiden Transport Protokolle TCP und UDP. In der Ebene 4 liegen die Applikations-Module, die entweder auf UDP oder TCP aufsetzen.

Aufgabe 1:

Woher weiß das IP-Modul, ob die Daten für das TCP- oder das UDP- Modul bestimmt sind?

Aufgabe 2:

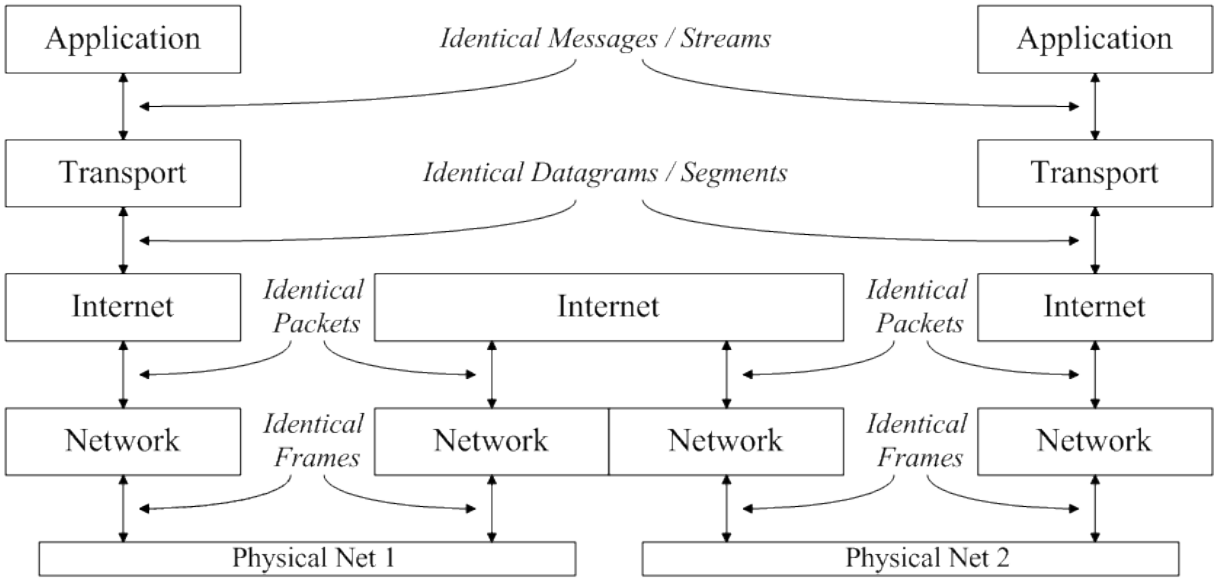
Woher weiß das entsprechende Transport-Modul zu welchem Applikations-Modul die Daten weitergeleitet werden müssen?

Aufgabe 3:

Nennen Sie einige Applikationen bzw. Applikationsprotokolle die auf TCP und einige die auf UDP aufsetzen?

2.3. Peer-to-Peer Kommunikation

Die Kommunikation zwischen 2 Knoten im Internet (gilt für jedes geschichtete Netzwerk) erfolgt Peer-to-Peer. D.h. die korrespondierenden Schichten „unterhalten“ sich miteinander und das Protokoll regelt die Art und Weise dieser Unterhaltung.



Die obige Abbildung illustriert dies und definiert überdies die verwendete Terminologie. Auf der Ebene der Applikationen werden **Messages** oder **Streams** ausgetauscht. TCP basierte Transporteinheiten werden als (**TCP-) Segmente** bezeichnet. Bei UDP handelt es sich um **UDP-Datagramme**. In der IP-Schicht spricht man von **IP-Datagrammen**[1] und in der Netzwerkschicht von **Rahmen** (*frames*). In jeder Schicht wird ein *Protocol-Header* hinzugefügt, der entsprechende Steuerinformationen für die korrespondierende Schicht des Kommunikationspartners enthält. Diesen Vorgang nennt man *Data Encapsulation*:

[1] Im Gegensatz dazu ist der Terminus **Paket** eine generische Bezeichnung für eine Einheit von Daten ist, die auf beliebigem Level im Netzwerkstack ausgetauscht wird.

2.4. Horizontaler und vertikaler Datenfluss

Während zwischen den korrespondierenden Schichten von zwei Knoten im Netzwerk der Datenfluss **horizontal** stattfindet, werden die Daten in einem Knoten **vertikal** von Schicht zu Schicht weitergeleitet. Die Schichten die durchlaufen werden bezeichnet man als **Protocol-Stack**. Der Protocol-Stack für eine FTP-Verbindung eines Internet-Knotens an einem Ethernet-LAN lautet somit FTP / TCP / IP / ENET. Der Protocol-Stack für NFS lautet für den selben Knoten NFS / RPC / UDP / IP / ENET.

Beim Senden werden die Daten von oben nach unten durch die Schichten gereicht; man spricht von einem **N-to-1 Multiplexer**.

Beim Empfangen werden die Daten von unten nach oben gereicht; man spricht von einem **1-to-N Demultiplexer**. Die Demultiplex-Information steht im jeweiligen Protocol-Header der unteren Schicht. Für einen Internet-Knoten in einem Ethernet-LAN und eine Telnet-Verbindung gilt:

- ENET: Type-Field im Ethernet-Frame = 0x0800 (weiter zu IP)
- IP: Protocol-Number im IP-Header = 6 (weiter zu TCP)
- TCP: Port-Number im TCP-Header = 23 (weiter zu Telnet)

Aufgabe 4:

Wo sind all diese Werte definiert?

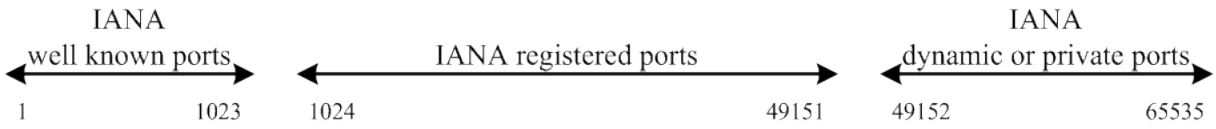
2.5. Ports und Sockets

Eine der zentralen Fragen lautet: *„Wie finden Client und Server zueinander?“*.

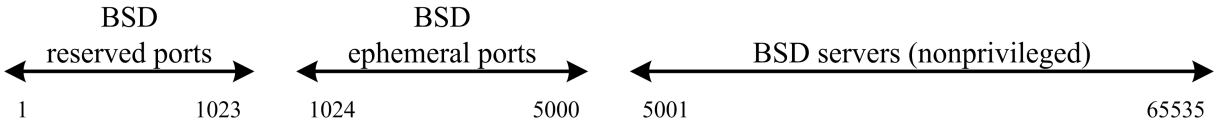
Sowohl der clientseitige als auch der serverseitige Teil einer Internet-Applikation verwenden entweder das TCP oder das UDP als Transportprotokoll oder setzen direkt (*raw* - roh) auf der IP-Schicht auf, wenn die Applikation keinen Transportdienst benötigt bzw. der Transportdienst integraler Bestandteil der Applikation ist. Bei Verwendung von UDP bzw. TCP dient die UDP- bzw. die TCP-Portnummer zur Identifikation (Demultiplexinformation) der Applikation. Bei Applikationen die direkt auf IP aufsetzen wird die IP-Protokoll-Nummer verwendet, welche dann häufig als *Raw-Portnumber* bezeichnet wird.

Standardapplikationen wie FTP, Telnet, SMTP, etc. haben serverseitig „wohl definierte“ Portnummern, die sogenannten ***well known ports***, zugeordnet bekommen (siehe „Assigned Numbers“-RFC, z.B. http://www.ietf.org/rfc.html). Clientseitig wird beim Aufbau einer Verbindung zu einem Server eine lokale Portnummer (***local port***) vom Betriebssystem vergeben. Alle lokalen Portnummern eines Knoten werden vom Betriebssystem verwaltet. Die 16-Bit breiten Portnummern werden von der IANA (Internet Assigned Numbers Authority) strukturiert und reguliert:

- *Well known ports* (0 bis 1023) werden von der IANA reguliert.
- *Registered ports* (1024 bis 49151) werden von der IANA nicht reguliert, jedoch registriert und veröffentlicht (z.B. die X-Server-Ports 6000 bis 6063).
- *Dynamic or private ports* (49152 bis 65535) sind *ephemeral* und frei verfügbar.

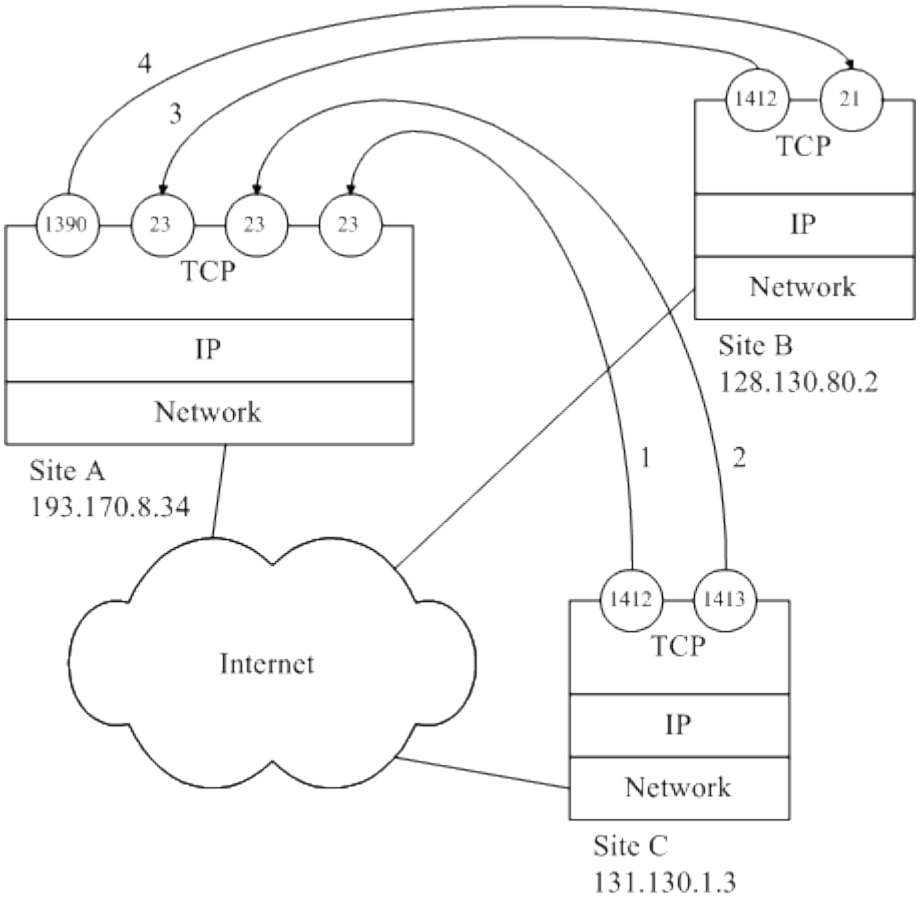


Die Zuordnung der IANA wurde stark von den BSD Portnummern beeinflusst, da diese in vielen Unix-Derivaten verwendet werden.



Die Kombination von **IP-Adresse : Portnummer** (z.B. 193.170.255.35:21) wird als **Socket** bezeichnet. Eine Client-Server Verbindung ist erst durch das **Socket-Paar** (Server-Socket, Client-Socket) eindeutig definiert (siehe Verbindungen 1 und 2 in nächster Abbildung). Der Grund liegt darin, dass auf einem Knoten 2 Clientprozesse die selbe Serverapplikation auf einem anderen Knoten ansprechen dürfen.

Connection	Client-InAddr	Client-Port	Server-InAddr	Server-Port
1	131.130.1.3	1412	193.170.8.34	23
2	131.130.1.3	1413	193.170.8.34	23



2.6. TCP vs. UDP

Eine weiter Frage lautet: *„Warum gibt es zwei Transportprotokolle (TCP, UDP)?“*.

TCP (Transmission Control Protcol) und UDP (User Datagram Protocol) stellen unterschiedliche Dienste auf der Transportschicht (Ebene 3 der Internetschichtung) zur Verfügung. TCP ermöglicht eine zuverlässige Übertragung von **Bytestreams** (eigentlich ein *stream of octetts*), während UDP eine verbindungslose, damit "unzuverlässige" Übertragung von **(UDP-) Datagrammen** gestattet.

Der Applikationsprogrammierer entscheidet sich für einen der beiden Transportdienste. Sollen über WAN-Verbindungen Dateien zuverlässig übertragen werden, dann stellt TCP sicher die geeignete Wahl dar (z.B. FTP).

Im schnellen, i.A. zuverlässigen LAN-Bereich wird für viele Dienste UDP die richtige Wahl sein (z.B. NFS). UDP hat den geringeren Overhead, was in einer niedrigeren Netzbelastung resultiert. Entscheidet man sich für UDP als Transportdienst, und benötigt man aber trotzdem ein gewisses Maß an Zuverlässigkeit, dann muss die Applikation selbst diese Zuverlässigkeit implementieren.

Verwendet man TCP auf der Transportebene, muss jedoch struturierte Daten übertragen, dann muss ebenfalls die Sendeapplikation dafür Sorge tragen, dass die Empfangsapplikation die notwendige Struktur vorfindet. – Diese Struktur kann auf Ebene der gesamten Requests, auch der Ebene eines einzelnen Records, sowie auf der Ebene jedes Feldes eines einzelnen Records notwendig sein.

Aub jeder dieser Ebenen kann das Aufbringen der Struktur auf eine der folgenden Arten geschehen:

- *Verwenden einer fixen vordefinierten Länge:* Die Anzahl der Bytes der Feldes, Records, oder des Gesamtrequests ist fix und sowohl dem Sender als auch dem Empfänger bekannt
- *Länge als definierter Teil des Feldes, Records, oder des Gesamtrequests:* Die Anzahl der Bytes eines Feldes, Records, oder des Gesamtrequests wird an einer definierten Position (z.B., in einem definierten Feld des Records, oder im ersten Byte eines Feldes) vom Sender abgelegt und vom Empfänger von dieser Position gelesen. – HTTP

(RFC 2616) verwendet z.B. diese Technik: Länge des Gesamtrequests wir in einem definierten Feld im HTTP Header mitgeschickt.

- *Expliziter Teminator:* Hierbei wird zwischen Sender und Empfänger ein definiertes Zeichen (bzw. eine definierte Sequenz von Zeichen) als Terminator eines Feldes, Records, oder des Gesamtrequest vereinbart. Für den Fall, dass diese Sequenz in den Nutzdaten zufällig vorkommt, so muss sie vom Sender „escaped“ werden, um nicht vom Empfänger fälschlich als Terminator interpretiert zu werden. Dies wird als *stuffing* bezeichnet. – SMTP (RFC 5321) verwendet z.B. diese Technik: Ein '.' auf einer eigene Zeile markiert das Ende des Datenfeldes.
- *End-of-file (EOF):* Wenn *genau ein Request* über die Verbindung ausgetauscht wird, so kann dessen Ende auch durch das (einseitige) Terminieren der Verbindung markiert werden, welches beim Empfänger als end-of-file (EOF) wahrgenommen wird.

2.7. TCP (Transmission Control Protocol)

Mit TCP wird der Applikation auf der Internet-Transportschicht ein zuverlässiger *End-to-End* Dienst geboten. Das verbindungsorientierte, oktettorientierte TCP setzt auf IP auf und wird auch, wegen seiner Flusskontrolle auf Oktettbasis, als ***reliable stream transport service*** bezeichnet. TCP stellt die folgenden Dienste zur Verfügung:

- **End-to-End Kontrolle**
  Das TCP gewährleistet eine korrekte Datenübertragung durch einen positiven Quittierungsmechanismus auf Oktettbasis. Verlorengegangene Daten werden nochmals übertragen.
- **Multiplexmechanismus**
  Über die Portadressierung erfolgt ein Multiplexen zu den Applikationen. Weiters wird eine Verbindung durch ihre beiden *Sockets* (IP-Adresse + Portnummer) identifiziert; deshalb können zwischen 2 Stationen parallele Verbindungen zu identen (meist *well-known*-) *Sockets* bestehen.
- **Das Verbindungsmanagement**
  umfasst den gesicherten Aufbau einer Verbindung (3-Way-Handshake), die gesicherte Aufrechterhaltung der (virtuellen) Verbindung während des gesamten Informationsaustausches sowie einen gesicherten Abbau (bzw. einen forcierten Abort) der Verbindung.
- **Datentransport**
  Der Datentransport kann nach erfolgreichem Aufbau der Verbindung bidirektional (*full-duplex*) erfolgen.
- **Flusskontrolle**
  Der Datenstrom wird durch Sequenz- und Bestätigungsnummern auf Oktettbasis zuverlässig übertragen. Mit einem *Window*-Mechanismus wird der Empfängerüberlauf verhindert.

- **Zeitüberwachung der Verbindung**

  Übertragene Daten müssen innerhalb einer definierten Zeit vom Empfänger quittiert werden. Erfolgt diese Quittierung nicht, werden die Daten erneut übertragen (*retransmission-mechanism*).

- **Reihenfolge**

  Client und Server tauschen vollkommen transparent Datenströme aus. TCP garantiert, dass die Daten in derselben Reihenfolge den Empfängerprozess erreichen, wie sie vom Senderprozess verschickt wurden.

- **Spezialfunktionen**

  Über die beiden Spezialfunktionen *PUSH* und *URGENT* können Vorrangdaten (meist Befehle die die Verbindung selbst betreffen) gekennzeichnet werden. *PUSH* erzwingt die sofortige Datenübergabe an die nächsthöhere bzw. nächstniedrigere Schicht. Ein *URGENT* Segment zeigt dem Empfänger dringliche Daten an (o*ut of band*); es liegt jedoch am Empfänger, wie diese Information genutzt wird.

- **Fehlerbehandlung**

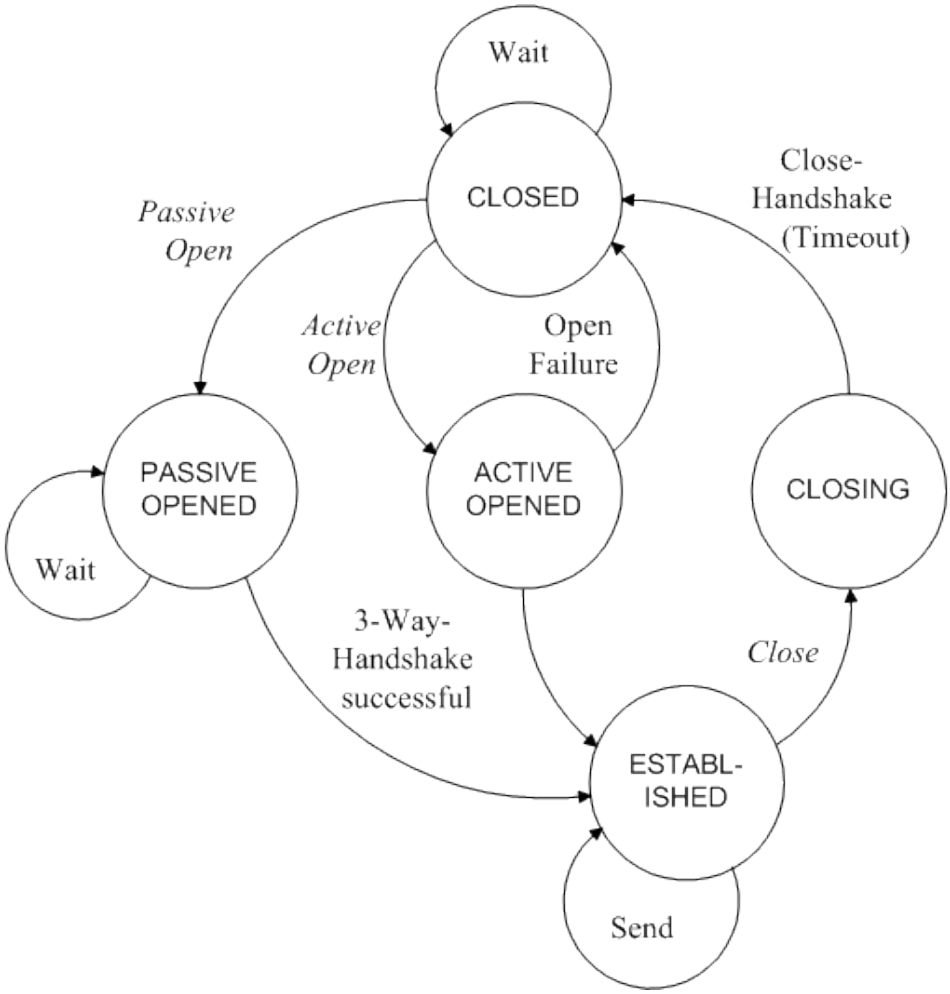
  Wird ein empfangenes Segment als fehlerhaft erkannt, wird es verworfen. Die Daten werden daher nicht positiv quittiert; es erfolgt eine automatische *retransmission*.

TCP

Reliable Stream Service
Virtual Circuit (connection-oriented service)
3-Way Handshake (to establish a connection)
Full Duplex
Flowcontrol, Sequence Numbers, Sliding Window
Positive Acknowledge with Retransmission (PAR)

2.8. Die TCP-State-Machine

Der folgende Zustandsübergangsgraph (*state transition diagram*) zeigt vereinfacht die Zustände eines TCP-Moduls. Die *kursiv* geschriebenen Übergangsbedingungen sind die entsprechenden *service requests* einer Applikation, die TCP verwendet.



Sowohl Client- als auch Serverapplikation starten im CLOSED Zustand. Eine Server-Applikation (z.B. Telnet-Server) fordert vom TCP-Modul ein ***Passive Open*** mit der entsprechenden *Source-Port*-Nummer (in diesem Fall *well known port* 23) an. Das *Destination-Port (bzw. Socket)* bleibt unspezifiziert. Die Server-Applikation landet im PASSIVE_OPENED Zustand und wartet auf eine Anforderung eines Clients.

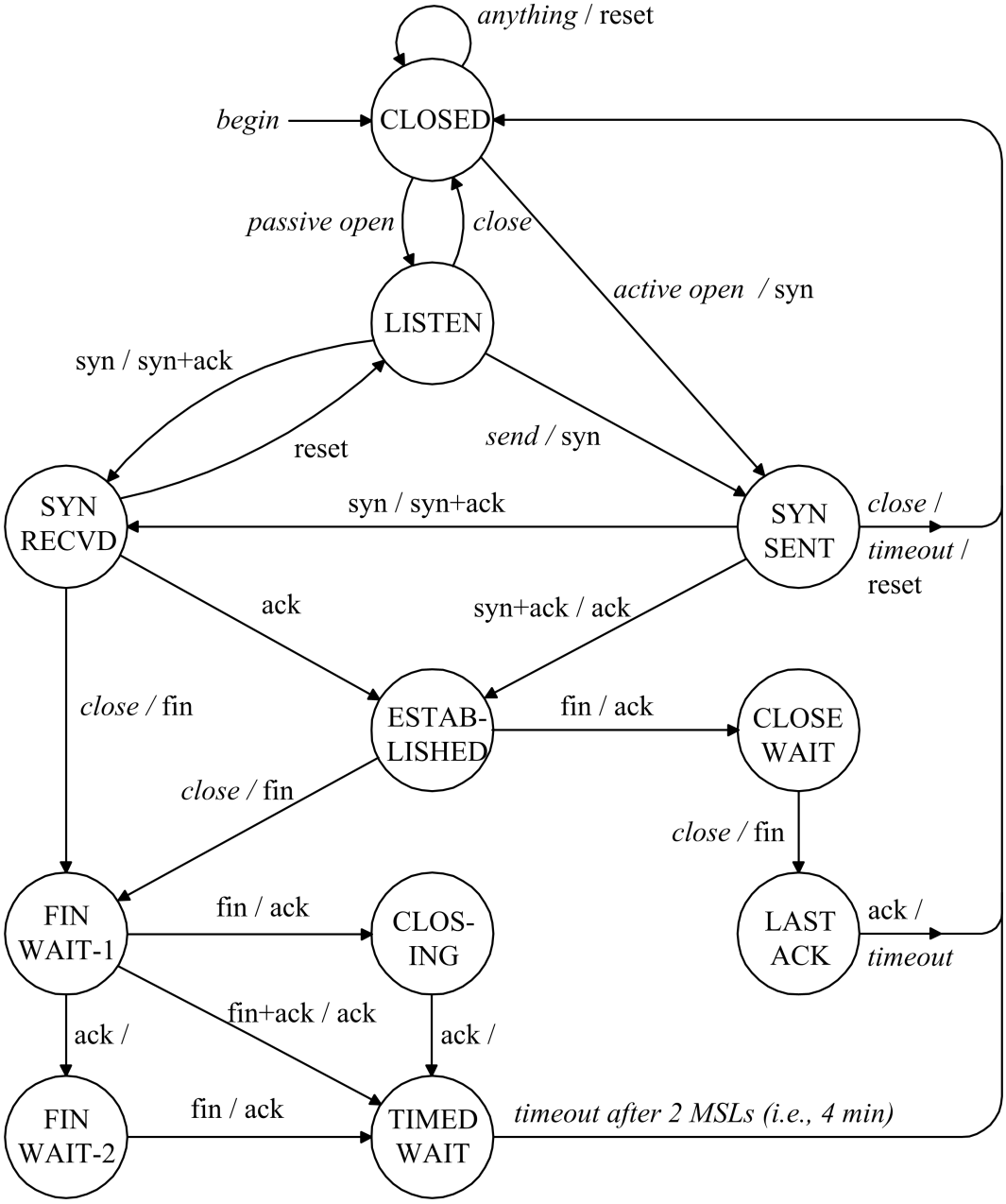
Eine Client-Applikation fordert vom TCP-Modul ein ***Active Open*** an, und spezifiziert dabei den *Destination-Socket*. Die Source-Port-Nummer ist die nächste im (Betriebs-) System frei verfügbare (*local port*). Der Client landet im fehlerfreien Fall im ACTIVE_OPENED

Zustand. Zwischen TCP-Modul des Servers und TCP-Modul des Clients findet nun der Verbindungsaufbau (*3 way handshake*, siehe weiter unten) statt.

Nach erfolgreichem Verbindungsaufbau befinden sich sowohl der Server als auch der Client im Zustand ESTABLISHED. Nun können Daten (vollduplex, d.h. in beide Richtungen) ausgetauscht werden.

Entweder der Server oder der Client fordert einen Verbindungsabbau an (*close request*). Im Zustand CLOSING sind alle Aktivitäten zusammengefasst gedacht die notwendig sind um sowohl den Client als auch den Server in den Ausgangszustand (CLOSED) zu bringen.

Für qualitative Überlegungen genügt i. A. dieses sehr stark vereinfachte Zustandsdiagramm. Will man jedoch z.B. den Verbindungsaufbau im Detail analysieren und verstehen so benötigt man das TCP-STD in seiner detaillierten Form. Die nächste Abbildung zeigt dieses in einer *Mealy*-Notation,- d.h. über den Transitionen stehen die Eingangsbedingungen und, getrennt von einem Schrägstrich, die Ausgangswerte. Kursiv geschrieben sind wieder die *service-requests* (z.B. SysCalls) und „normal“ geschrieben sind die Flags des TCP-Headers (siehe Lehrveranstaltung „Telekommunikation“).

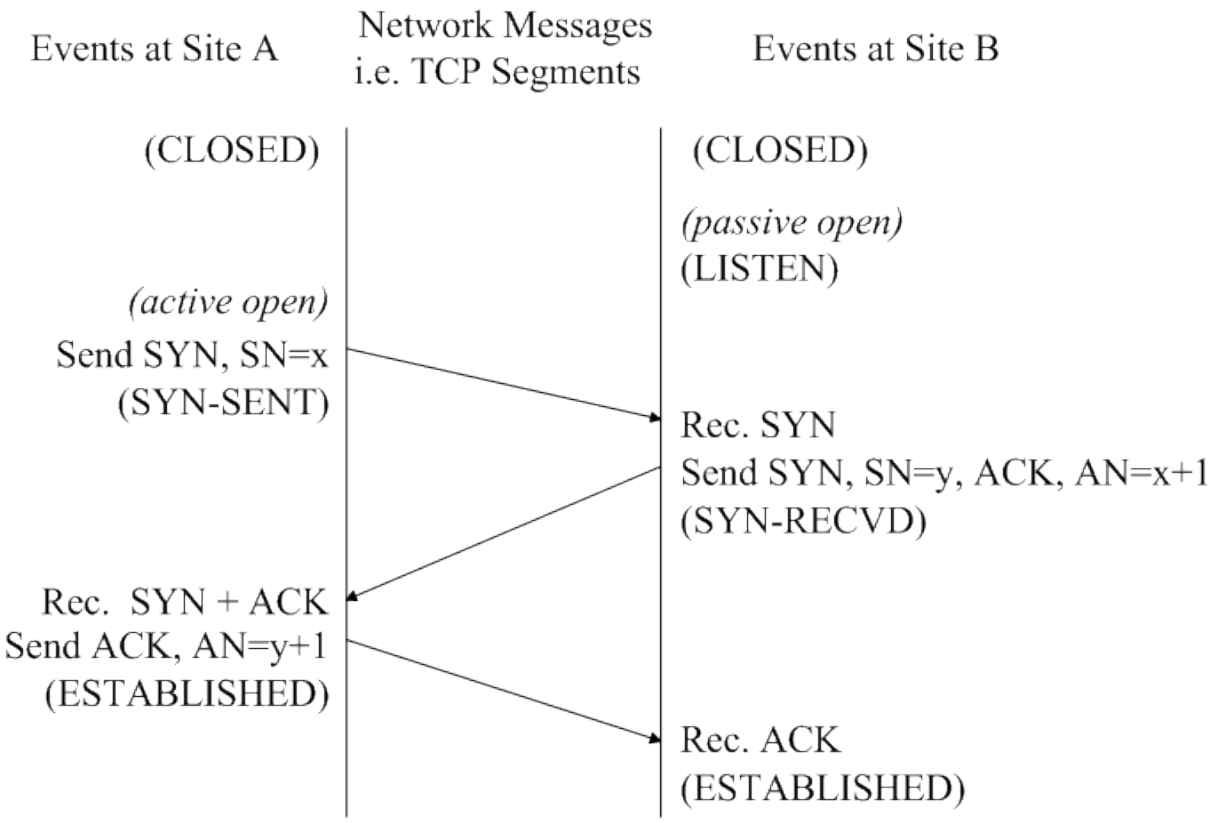

anything / reset
begin
CLOSED
passive open
close
active open / syn
LISTEN
syn / syn+ack
reset
send / syn
SYN RECVD
syn / syn+ack
SYN SENT
close /
timeout /
reset
ack
syn+ack / ack
close / fin
ESTAB-LISHED
fin / ack
CLOSE WAIT
close / fin
close / fin
FIN WAIT-1
fin / ack
CLOS-ING
LAST ACK
ack /
timeout
fin+ack / ack
ack /
ack /
FIN WAIT-2
fin / ack
TIMED WAIT
timeout after 2 MSLs (i.e., 4 min)

2.9. TCP Datenfluss

Bei einer TCP/IP Client Server Kommunikation „läuft“ sowohl im Client als auch im Server eine TCP-State-Machine. Um den client- und serverseitigen zeitlichen Ablauf der Zustände besser verstehen zu können, wird im Folgenden die Übertragung der TCP-Segmente zwischen Client und Server zeitlich visualisiert (Interaktionsdiagramme). Weiters wird in diesem Kapitel der PAR-Mechanismus (*positive acknowledge with retransmission*) und die elementare Flusskontrolle (s*liding window mechanism*) vorgestellt.

2.9.1. TCP Verbindungsaufbau (*3 way handshake*)

1. Station A will eine TCP-Verbindung zu Station B aufbauen und sendet dazu ein TCP-Segment mit aktivem SYN und der *Initial Sequence Number* x (ISN) im Sequence Number (SN) Feld des TCP-Headers.

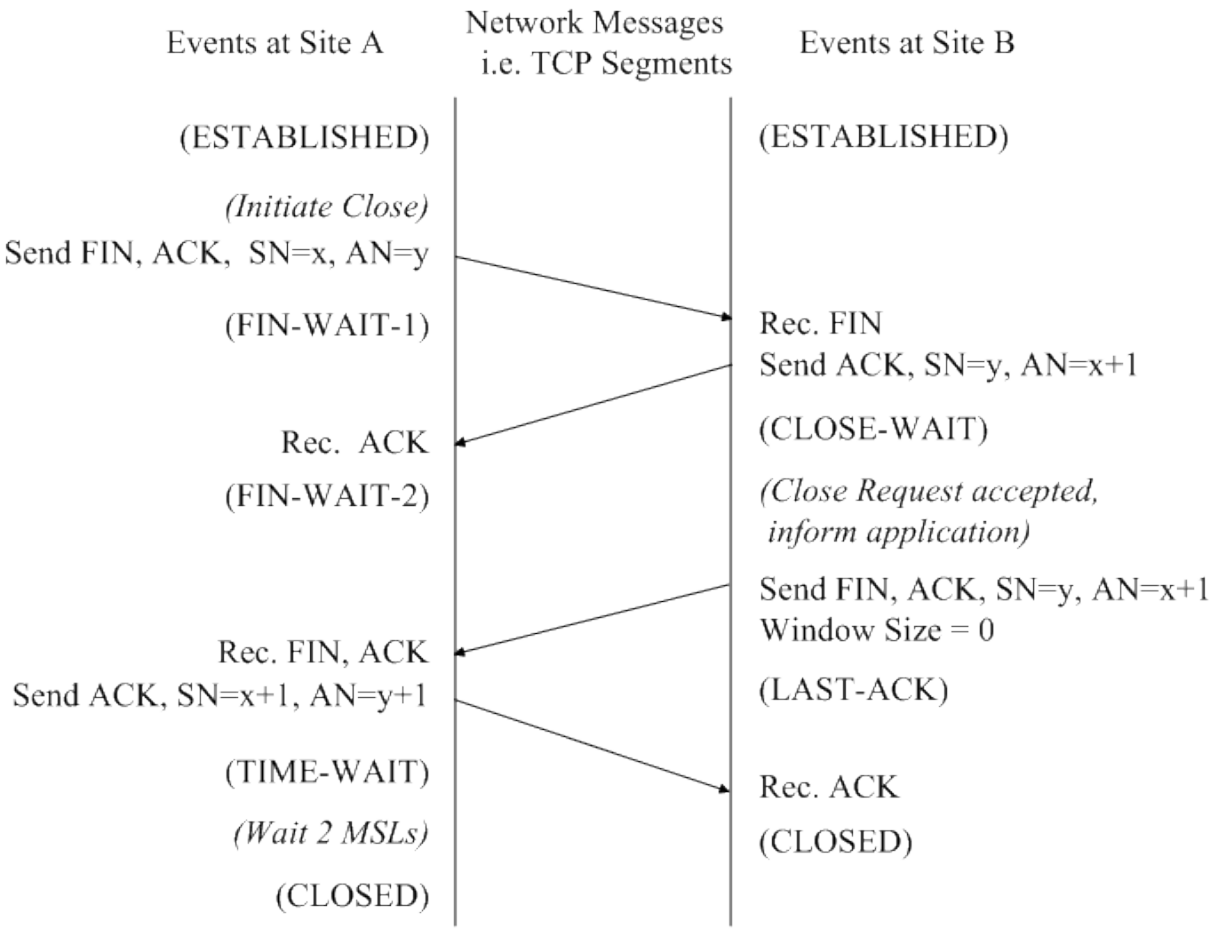


2. Station B empfängt diesen SYN-Request und antwortet ihrerseits mit der Bestätigung der ISN im Acknowledge Number (AN) Feld durch den Wert x+1 und gesetztem ACK-Flag. Im selben Segment übermittelt Station B an die Station A ihre ISN y im Sequence Number Feld und setzt das SYN Flag.
3. Station A empfängt ein Segment mit ACK und SYN gesetzt. ACK bedeutet, dass Station B den Verbindungswunsch inkl. ISN akzeptiert. SYN zeigt an, dass Station B im Sequence Number Feld ihre ISN übermittelt. Station A quittiert dies mit einem ACK Segment; im Acknowledge Number Feld steht y+1 als Bestätigung der ISN von Station B. Nachdem Station B diesen ACK empfangen hat, ist die virtuelle, voll-duplex Verbindung zwischen den Stationen A und B aufgebaut. Die Datenoktettnummerierung von A nach B beginnt mit x+1 und die von B nach A mit y+1.

Neben dem Synchronisieren der ISNs werden beim Verbindungsaufbau im allgemeinen zusätzlich die *Window*-Größe und die Maximale Segmentlänge (MSS *Maximum Segment Size*) ausgetauscht.

2.9.2. Der Verbindungsabbau

Nachdem die Datenübertragung beendet ist, wird die Verbindung abgebaut. Dabei ist es auch möglich, dass beide Stationen gleichzeitig die Verbindung abzubauen wünschen (vollduplex). Die folgende Abbildung zeigt einen geordneten Abbau der Verbindung.

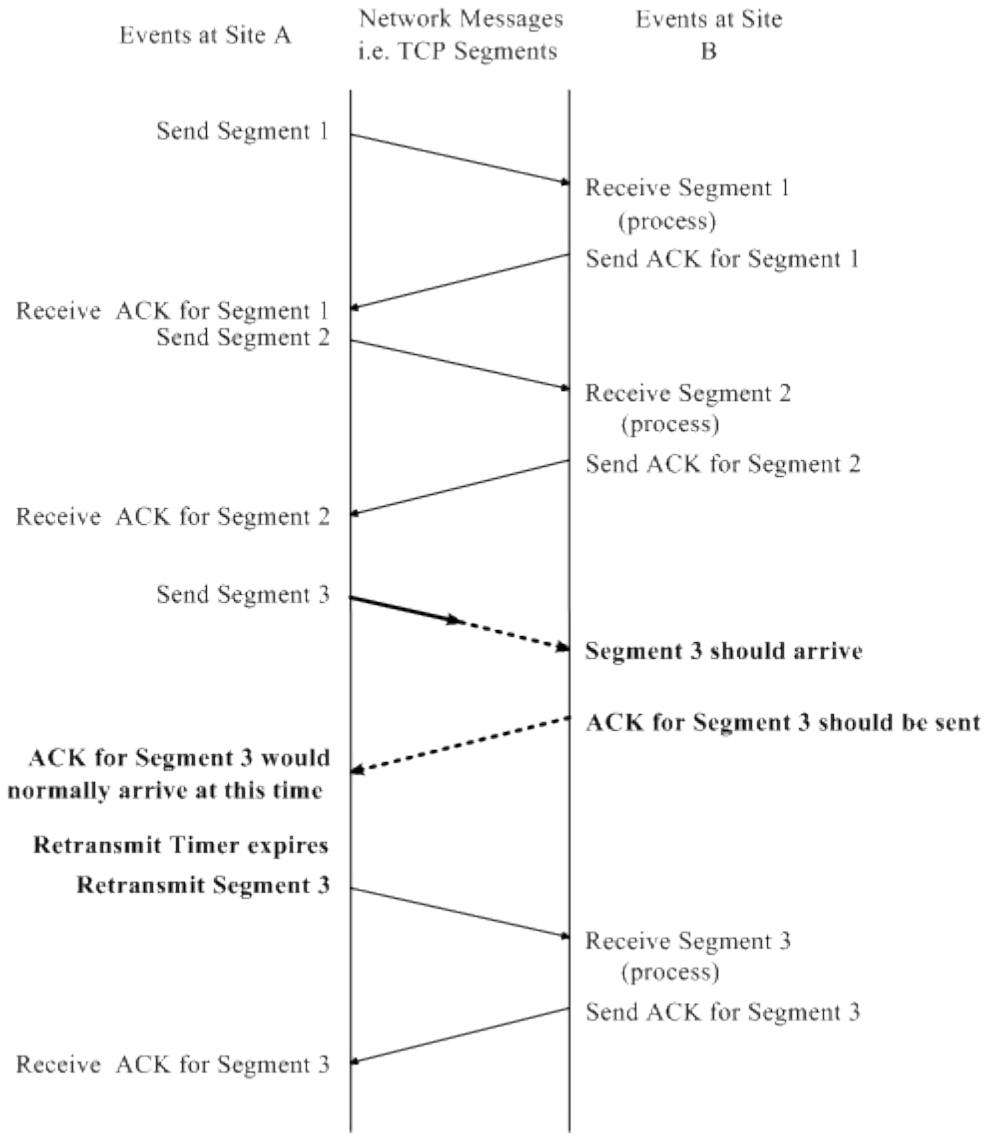


Eine Verbindung kann auch durch ein rücksetzen (gesetztes RST-Flag) abgebrochen werden (*Abort*). Das Segment mit RST gesetzt wird vom Empfänger als gültig erkannt, wenn darin die Sequenz-Nummer mit der eigenen, letzten Acknowledge-Nummer übereinstimmt. Bei einem Abort wird keine Rücksicht auf eventuelle Restdaten im Empfangspuffer (und im Netz) genommen.

Generell wird eine Verbindung rückgesetzt, wenn eine Station ein Segment empfängt, das sie nicht erwartet hat. Das trifft für jedes Segment zu, das nach einem *System-Crash* eintrifft.

2.9.3. Positive Acknowledge and Retransmission

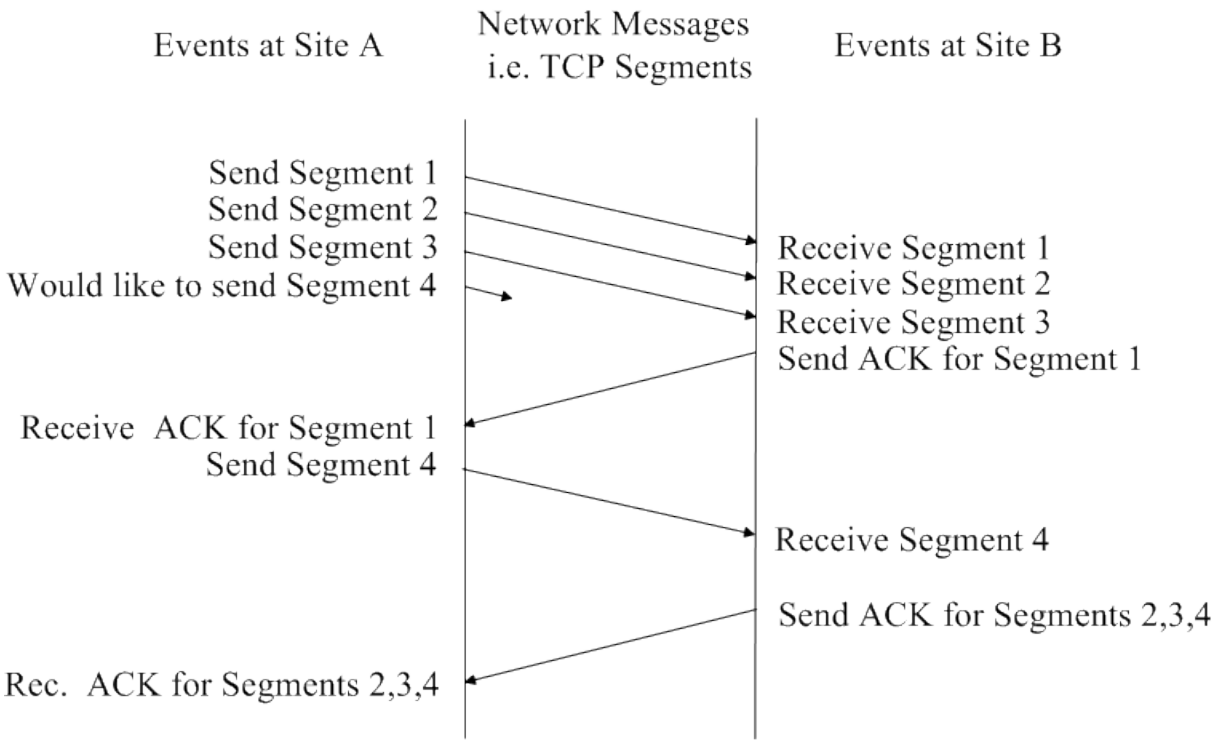
Das folgende Interaktionsdiagramm bedarf keiner weiteren Erklärung...



2.9.4. Die Flussteuerung (Sliding Window Mechanism)

Der einfache *Positive Acknowledge* Mechanismus ist nicht effektiv, weil der Sender nach dem Versenden eines Segments auf eine positve Quittierung warten muss. Außerdem kostet es Netzwerkbandbreite, da auf jedes Sendepaket ein Quittierungspaket folgen muss.

Der *Sliding-Window* Mechanismus erlaubt dem Sender soviel Segmente ohne Quittierung zu versenden, wie der Empfänger in der Lage ist aufzunehmen. Da die Nummerierung auf Oktettebene erfolgt, ist es möglich, dass der Empfänger mehrere gesendete Segmente auf einmal quittiert.



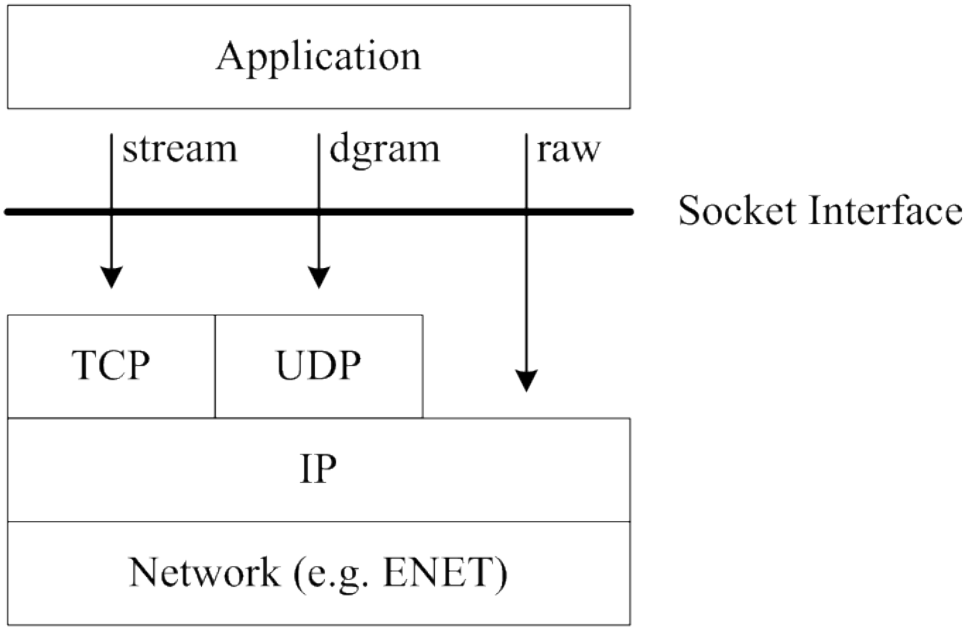
Dieser Mechanismus gilt natürlich nicht für jegliche Art von *out of band* Segmente (z.B. *Urgent*-Segmente).

3. TCP/IP Client Server Programmierung

In diesem Kapitel wird ein konzeptioneller Überblick über die (Linux bzw. BSD) Socket-Programmierung für TCP/IP Client Server Applikationen gegeben. Es wird jedoch nicht auf die Details der Programmierung (Parameter der System Calls, detaillierter Aufbau der Datenstrukturen, etc.) eingegangen. Dafür stehen die folgenden Studieneinheiten (FUV + UE) zur Verfügung, wo dieses spezifische, praktische Know How mit Hilfe von Original-unterlagen, wie z.B. man-pages, und Sekundärliteratur erarbeitet werden muss.

3.1. Das Socket-Interface

Das Socket-Interface ist ein API (***Application Programming Interface***) welches mit dem Unix-System 4.2BSD im Jahre 1983 veröffentlicht wurde. Es ist heute für viele (Multitasking-) Betriebssysteme verfügbar. Das Socket-Interface bietet einer Internet-applikation die Möglichkeit über TCP (*stream socket*), UDP (*datagram socket*) oder direkt über IP (*raw socket*) die Verbindung zwischen Client und Server aufzubauen. Das Socket-Interface liegt zwischen Ebene 3 und 4 (bzw. zwischen 2 und 4 für *raw-sockets*) im vier-schichtigen Kommunikationsmodell des Internets.



Die Unterstützung von Netzwerken allgemein und vom Internet im Speziellen ist i.A. Teil des Betriebssystems. Das Modul eines Betriebssystems welches die Kommunikation übers Internet ermöglicht und das Socket-API zur Verfügung stellt wird oft als **TCP/IP-Stack** bezeichnet. Der TCP/IP- oder Internet-Stack eines Betriebssystems stellt üblicherweise neben dem API auch Servicefunktionen und Internetapplikationen zur Verfügung.

Die wichtigsten ***System Calls (SysCalls)*** (nicht nur Socket-API !) für Internetapplikationen sind:

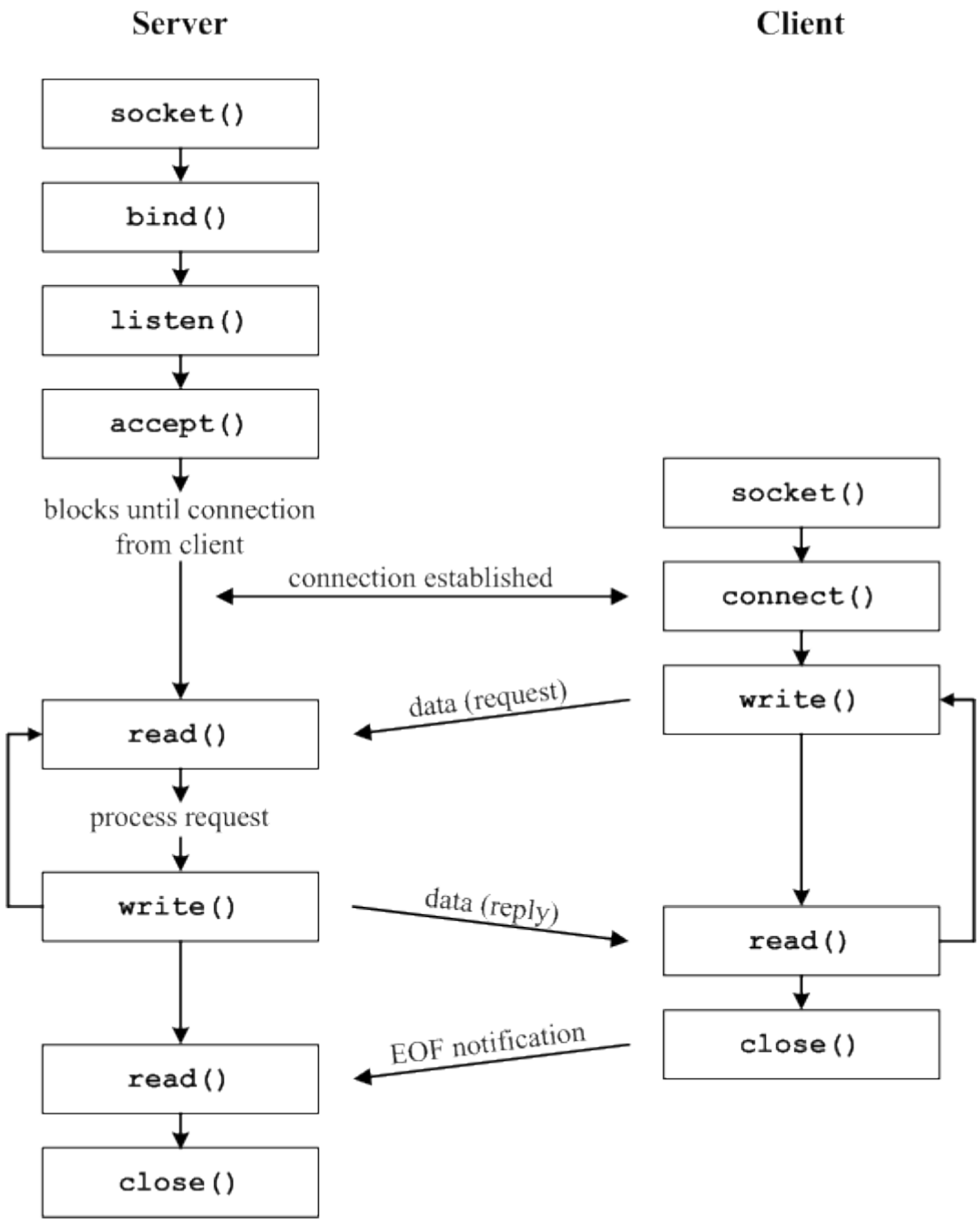
- **`socket(), bind(), listen(), accept(), connect()`**
- **`getaddrinfo(), inet_pton(), inet_ntop(), htons(), htonl(), ntohs(), ntohl(), ...`**
- **`getsockname(), getpeername()`**
- **`read(), write(), close(), shutdown(), fcntl(), select()`**
- **`fork(), exec(), sigaction(), waitpid(), ...`**

Die wichtigsten Servicefunktionen im Zusammenhang mit Netzwerk und Internet sind:

- **`ifconfig, arp, netstat, tcpdump`**
- **`traceroute, nslookup`**

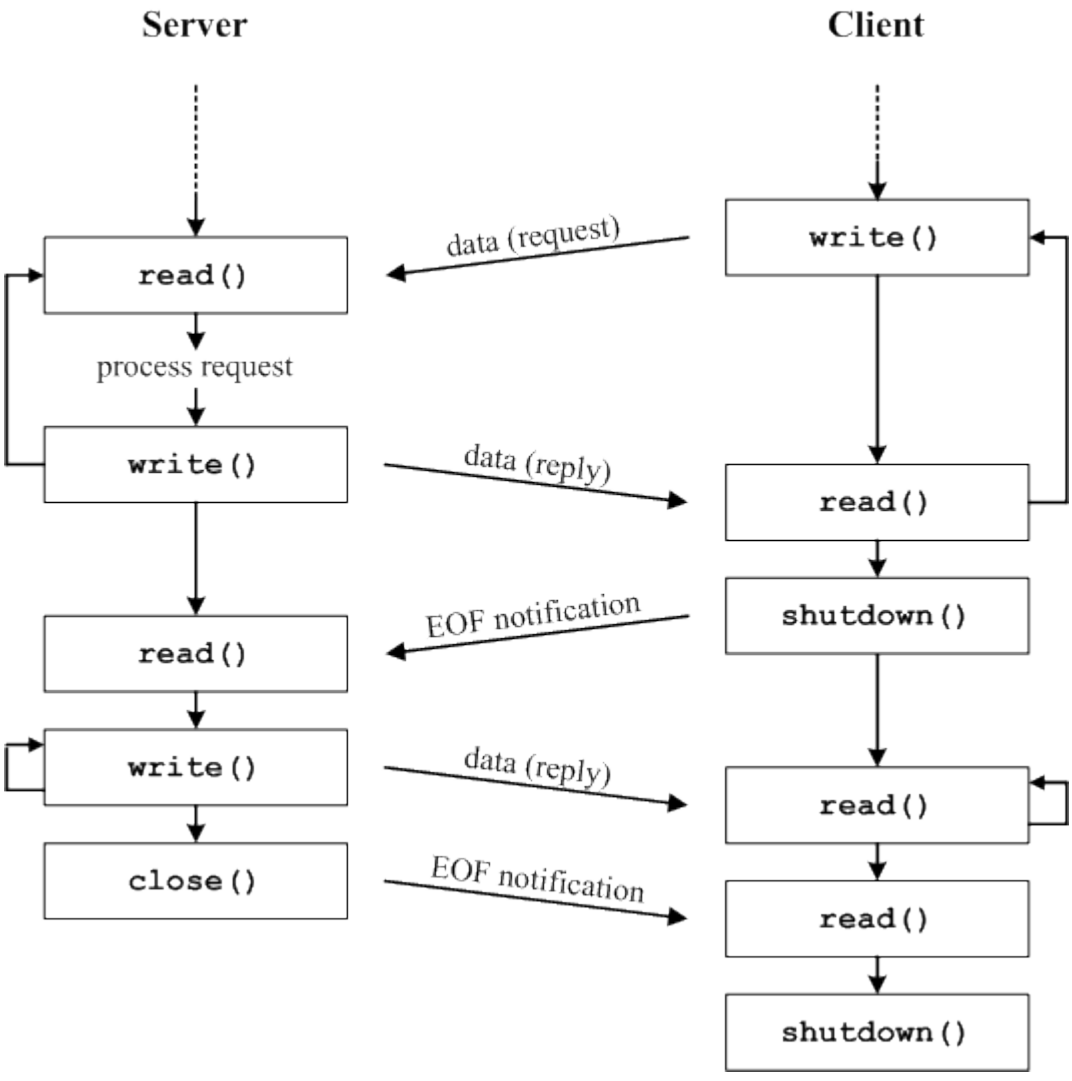
3.2. TCP-Server, TCP-Client

Die folgende Abbildung zeigt qualitativ die Abfolge der wesentlichsten System Calls für einen TCP-Server und einen TCP-Client. Der dargestellte Server kann nur eine Client-Verbindung zu einer Zeit bearbeiten. Man bezeichnet solch einen Server als *simple-server* oder *single-threaded-server*.



In dieser Abbildung fordert der Client einen Verbindungsabbau durch Aufruf des **`close()`** SysCalls an. - Dies führt dazu, dass der Server beim Versuch mittels **`read()`** zu lesen, den Returnwert 0 von **`read()`** erhält (nachdem alle Daten vom Server gelesen wurden), welcher die end-of-file (EOF) Notifikation darstellt.

Alternativ kann der Client auch nur die eine Client→Server Richtung der Vollduplexverbindung durch den Aufruf von **`shutdown()`** mit **`SHUT_WR`** als zweitem Parameter schließen, was ebenso zu einer EOF Notifikation beim Server führt (siehe folgende Abbildung).

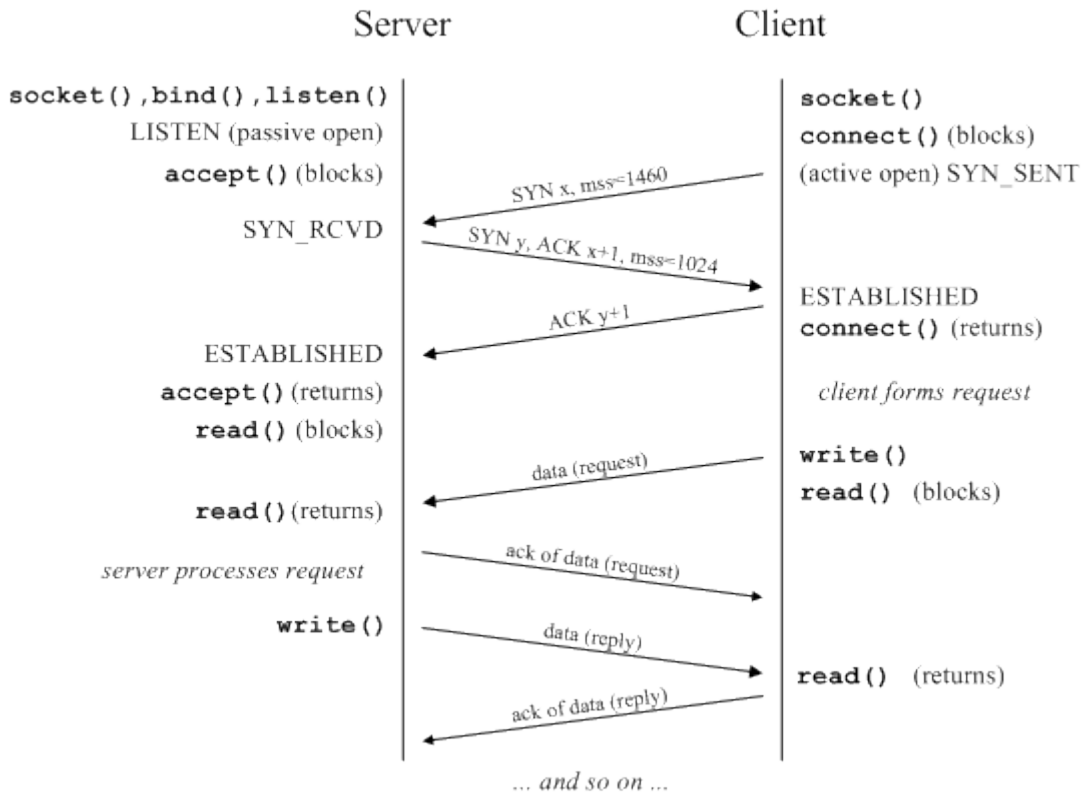


Allerdings kann in diesem Falle der Server noch Daten an den Client schicken (z.B., noch ausständige Replies), welche der Client seinerseits mittels **`read()`** wiederum lesen kann. Erst

wenn der Server seinerseits die noch offen Hälfte (Server→Client) der Vollduplexverbindung schließt (durch Aufruf von **`close()`** oder **`shutdown()`** mit **`SHUT_WR`** oder **`SHUT_RDWR`** als zweitem Parameter), erhält der Client die EOF Notifikation beim Aufruf von **`read()`** und sollte seinerseits ein **`shutdown()`** mit **`SHUT_RD`** als zweitem Parameter durchführen.

3.3. TCP Datenfluss

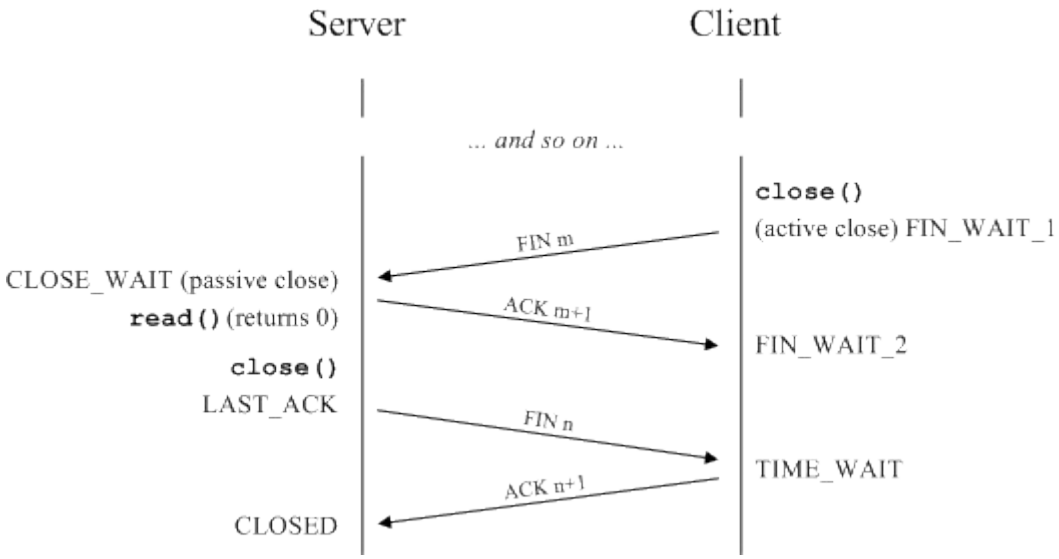
Die folgenden Interaktionsdiagramme zeigen den zeitlichen Ablauf der TCP-Segmente, die *System Calls* und die client- und serverseitigen Zustände der *TCP-State-Machines*. Das folgende Diagramm veranschaulicht den Verbindungsaufbau, sowie den Datenaustausch mittels der SysCalls **`read()`**[2] und **`write()`**.



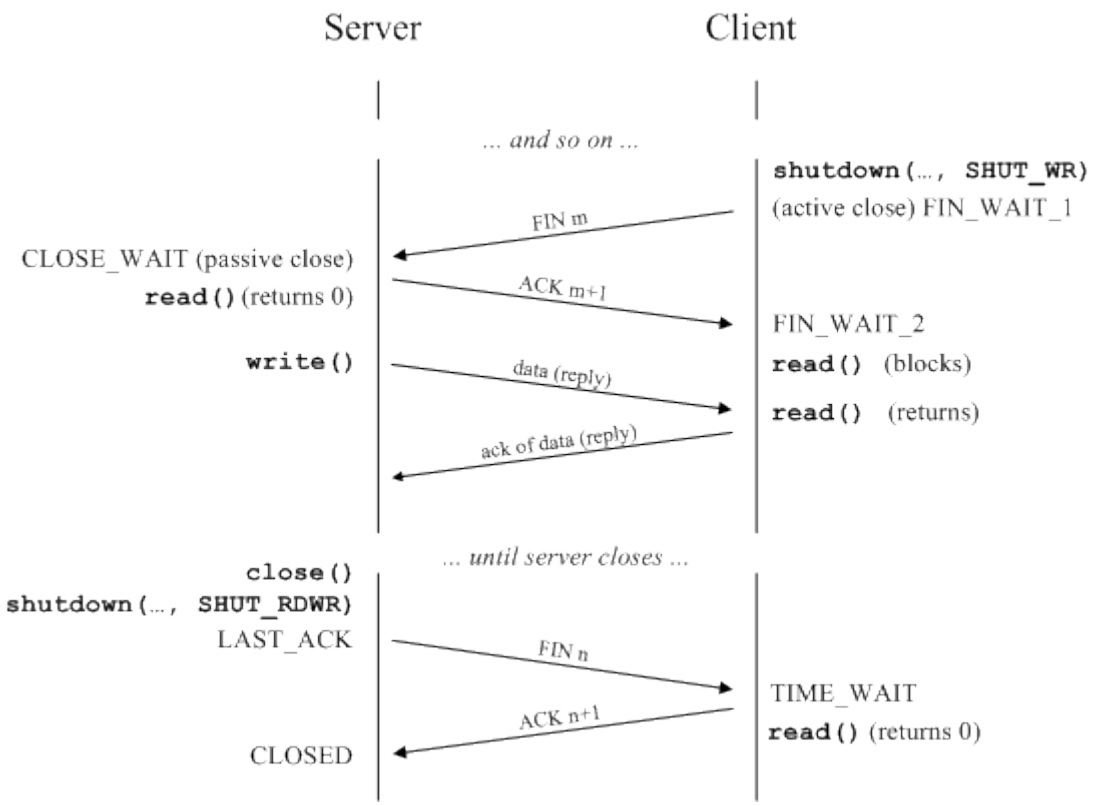
Sobald der Client keinen weiteren Request mehr an den Server stellen will, fordert der Client einen Verbindungsabbau durch den Aufruf des **`close()`** Systemcalls an. Dadurch werden **beide** Richtungen der Vollduplexverbindung abgebaut, d.h., nach einem (erfolgreichen) Aufruf von **`close()`** kann der Client weder auf den Socket schreiben, noch davon lesen.

[2]**ACHTUNG:** Es ist nicht garantiert, dass der **`read()`** SysCall immer die gewünschte Anzahl an Bytes zurückliefert (siehe Manual Page), auch nicht im fehlerfreien Fall.

Das folgende Diagramm zeigt den Verbindungsabbau mittels **`close()`**.



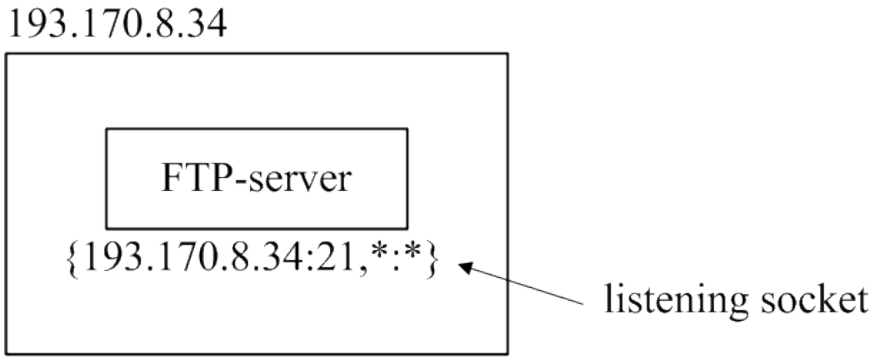
Alternativ kann der Client aber auch nur **eine** Richtung der Vollduplexverbindung (meist die Richtung vom Client zum Server) abbauen, um dem Server zu signalisieren, dass dieser keine weiteren Daten vom Client zu erwarten hat. Dies wird durch den Aufruf des **`shutdown()`** SysCall mit **`SHUT_WR`** als zweitem Parameter erreicht.



Nach einem (erfolgreichen) Aufruf von **`shutdown(..., SHUT_WR)`** kann der Client nur mehr von dem Socket lesen, und somit etwaige Restdaten eines Replies vom Server empfangen. – Sobald der Server seinerseits die Verbindung beendet (entweder durch den Aufruf von **`close()`** oder durch den Aufruf von **`shutdown(..., SHUT_RDWR)`**, erhält der Client nachdem er alle Daten vom Server konsumiert hat der Returnwert 0 vom **`read()`** SysCalls.

3.4. TCP-Client-Server Verbindungsaufbau

Für die folgenden Überlegungen betrachten wir einen Internetknoten (193.170.8.34) auf dem ein einfacher *iterativer* FTP-Server läuft, der zu einer Zeit nur einen Client bedienen kann (*iterative server, simple server, single threaded server*). Nach dem der Server ein erfolgreiches ***passive open*** (**`socket(), bind(), listen()`**) ausgeführt hat, ist ein sogenannter ***listening socket*** vorhanden. Der Server ist nach dem **`accept()`** blockiert und wartet auf eine Anforderung.

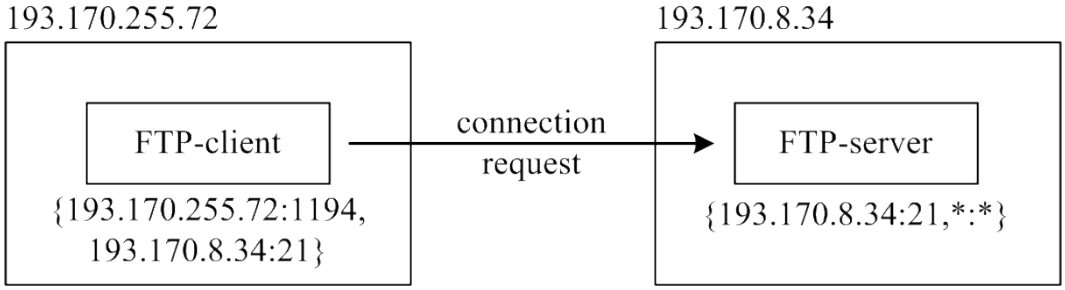


Anmerkung:

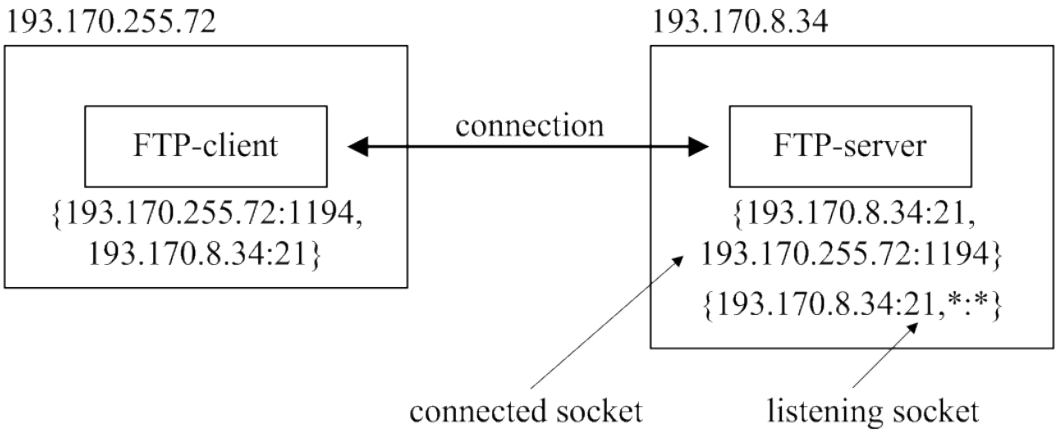
> Der *listening socket* ist eigentlich ein Socket-Paar, bei dem der eigene Socket (*host socket*) definiert ist und der Socket des Verbindungspartners (*peer socket*) noch undefiniert ist. Es ist durchaus üblich die Bezeichnung Socket sowohl für einen Socket als auch für ein Socket-Paar zu verwenden.

Fordert nun ein FTP-Client eine Verbindung von dem FTP-Server an, so läuft das beschriebene *3-way-handshake* (SYN, SYN / ACK, ACK) in den TCP-Modulen des Clients und des Servers ab. Dieser Vorgang ist für die Applikationen vollkommen transparent. Nach einem erfolgreichen Verbindungsaufbau kehrt der **`connect()`** SysCall erfolgreich zum Client und der **`accept()`** SysCall erfolgreich zum Server zurück. Client und Server sind im Zustand ESTABLISHED und können Daten austauschen.

Die folgende Abbildung zeigt die Situation unmittelbar nach der Verbindungsanforderung durch den Client.



Nachdem erfolgreichen Abschluss des Verbindungsaufbaus verfügt der Server über ein vollständiges Socket-Paar (*connected socket*) und zusätzlich über den *listening socket*.



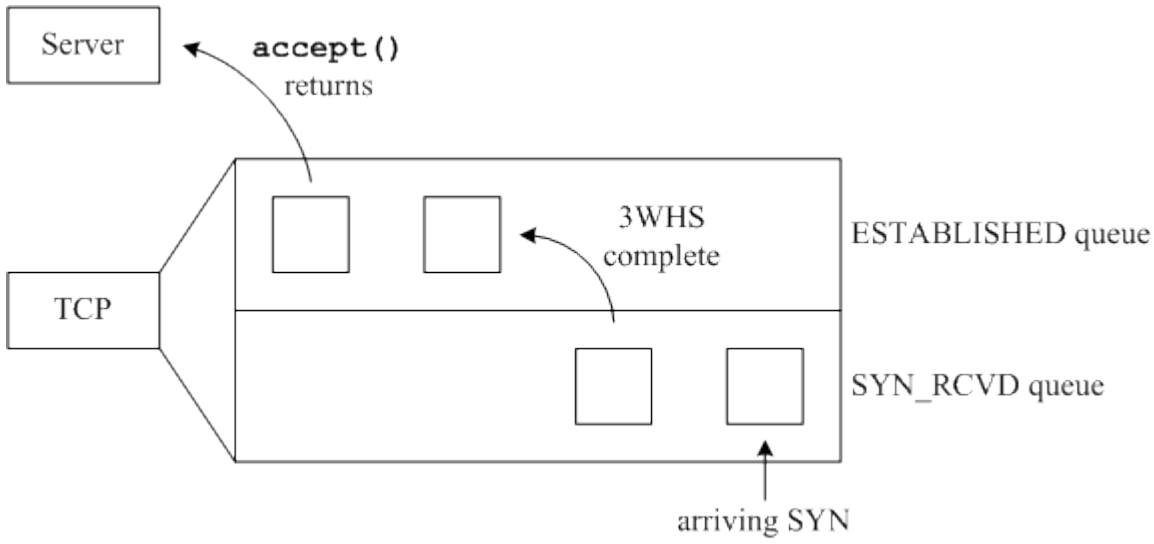
Weitere Verbindungswünsche von Clients könnten vom TCP-Modul durchaus erfolgreich behandelt werden, d.h. es kommt zu weiteren *connected sockets* beim Server. Da der (*simple*) Server jedoch nur eine Verbindung zu einer Zeit bearbeitet, kann kein Datenaustausch stattfinden, obwohl es weitere ESTABLISHED Client-Server Verbindungen gibt.

3.5. TCP-Verbindungsaufbau im TCP-Modul

Wie schon im letzten Punkt angesprochen, verwaltet des TCP-Modul (des Betriebssystems) den eigentlichen TCP-Verbindungsaufbau. Die entsprechenden SysCalls **`accept()`** und **`connect()`** kehren erst dann zu den aufrufenden Prozessen zurück, wenn die Verbindung steht oder die Verbindung definitiv nicht hergestellt werden konnte (Fehlerfall).

Zu diesem Zweck verwaltet das TCP-Modul zwei Warteschlagen:

- *Completed connection queue* für Verbindungen im ESTABLISHED Zustand
- *Incomplete connection queue* für Verbindungen im SYN_RCVD Zustand



Die Anzahl der Elemente aus beiden Warteschlangen dürfen den sogenannten ***backlog*** Wert nicht übersteigen, welcher beim **`listen()`** SysCall angegeben werden muss.

Anmerkung:

> Die Bedeutung des *backlog* Wertes kann von UNIX zu UNIX und von TCP-Modul zu TCP-Modul verschieden sein. Z.B. seit Linux 2.2 definiert *backlog* nur die Länge der *completed connection queue*. Die Länge der *incompleted connection queue* ist seit Linux 2.2 über die TCP-Steuervariable **`tcp_max_syn_backlog`** festgelegt.

3.6. Servertypen

Im abschließenden Kapitel dieser konzeptionellen Betrachtungen zum Thema TCP/IP Client Server Programmierung wird noch kurz auf die unterschiedlichen Servertypen eingegangen. Der einfachste Typ ist ein sogenannter ***iterative server***, während die anderen in die Kategorie der ***concurrent server*** fallen.

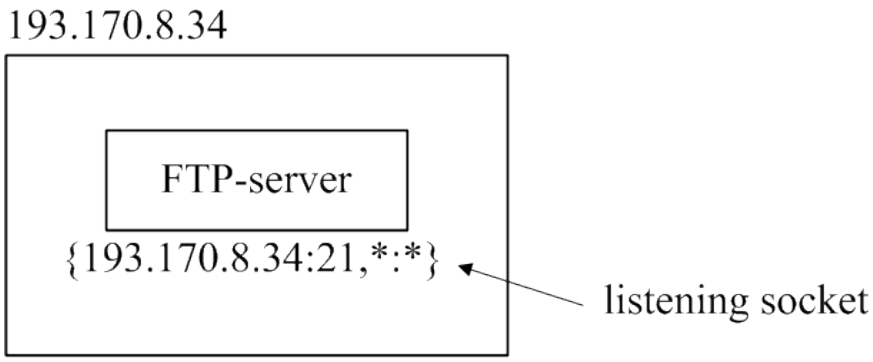
3.6.1. Simple-Server

Den einfachsten Fall eines TCP-Servers haben wir schon im Kapitel 3.5 im Rahmen der Diskussion über den Verbindungsaufbau zwischen Client und Server besprochen. Der Aufbau des Server-Codes entspricht exakt der Abbildung in Kapitel 3.2.

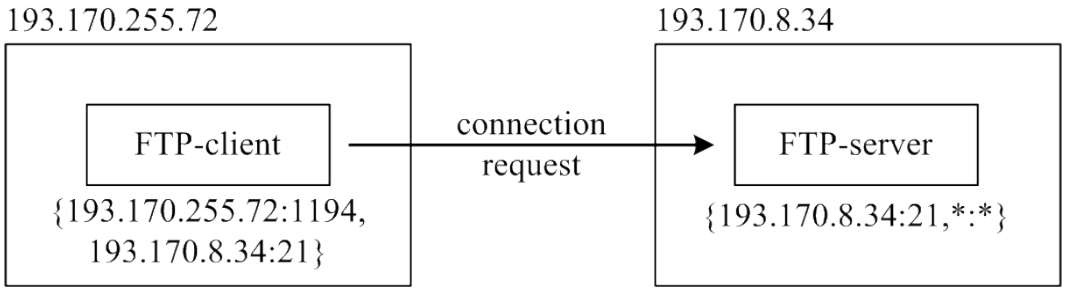
3.6.2. Forking-Server

Die meisten TCP/IP Server sind sogenannte *forking server*. Nach jedem erfolgreichen Verbindungsaufbau, somit nach jeder fehlerfreien Rückkehr des **`accept()`** SysCalls, führt der Server einen **`fork()`** SysCall aus, um die eigentliche Serverfunktion in einem *child-process* zu starten. Häufig ist die eigentliche Serverfunktion in ein externes Programm ausgelagert, welches vom *child-process* mit einem **`exec()`** SysCall gestartet wird. Die Kombination von **`fork()`** gefolgt von einem **`exec()`** wird in der Fachliteratur oft als ***spawn*** (*spawning a process* – gebären eines Prozesses) bezeichnet, weshalb dieser Servertyp auch als *spawning server* bekannt ist.

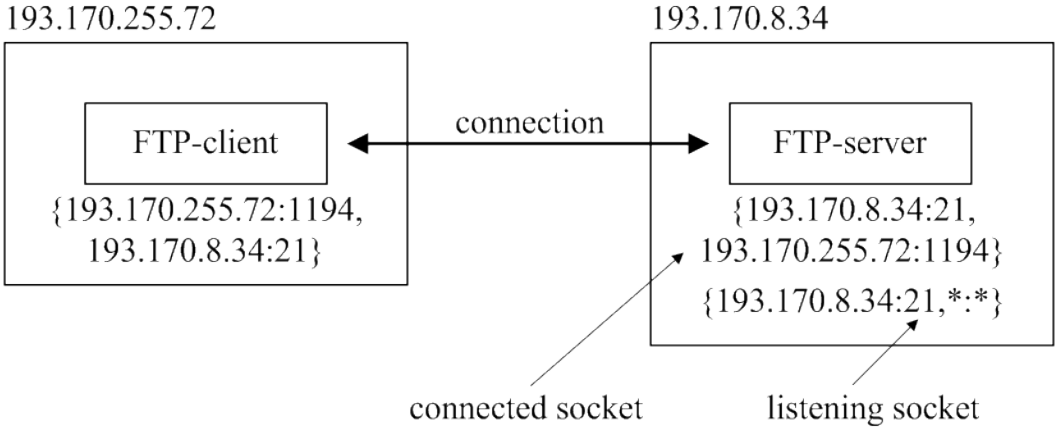
Da, wie schon erwähnt, dieser Servertyp sehr häufig in Implementierungen zu finden ist, wird an dieser Stelle das Verbindungsszenario visualisiert. Die erste Abbildung zeigt den Server mit dem *listening socket*.



Nach einem *connection request* findet im TCP-Modul der 3WHS Vorgang statt.

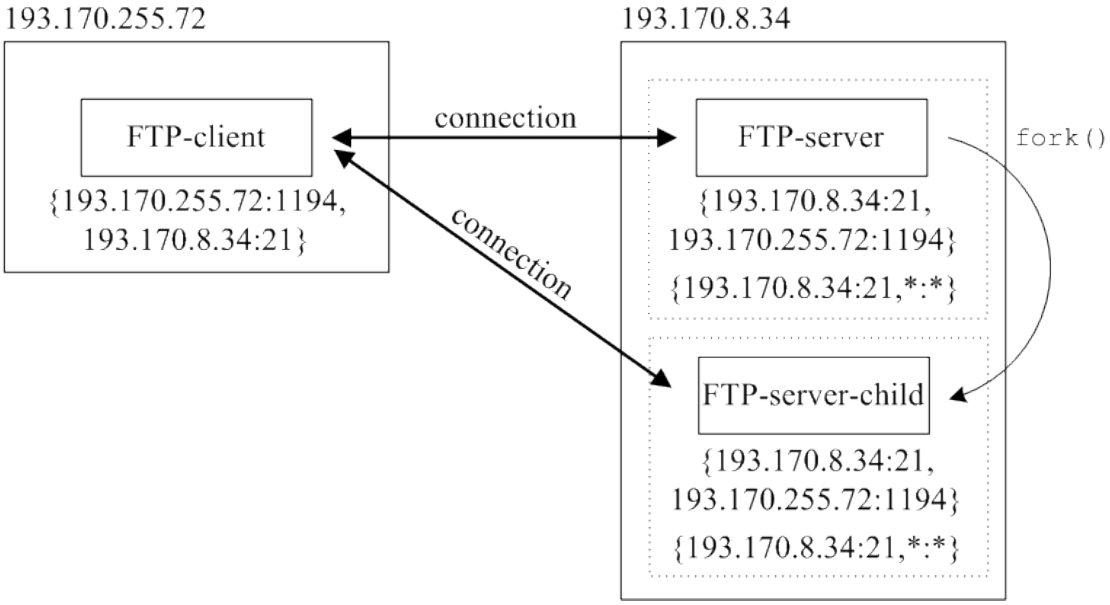


Nach erfolgreichem Abschluss des 3WHS kehrt der **`accept()`** SysCall mit einem connected socket zum Server-Prozess zurück.

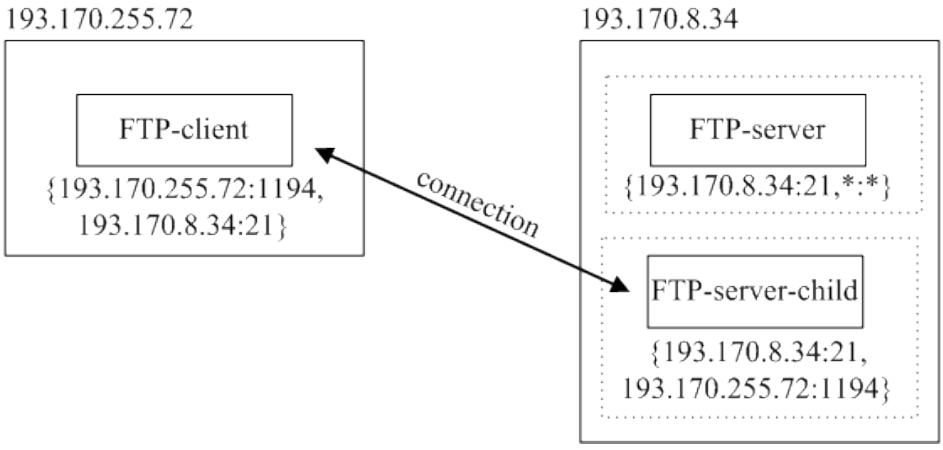


Bis zu diesem Zeitpunkt verhalten sich der Simple-Server und der Forking-Server gleich.

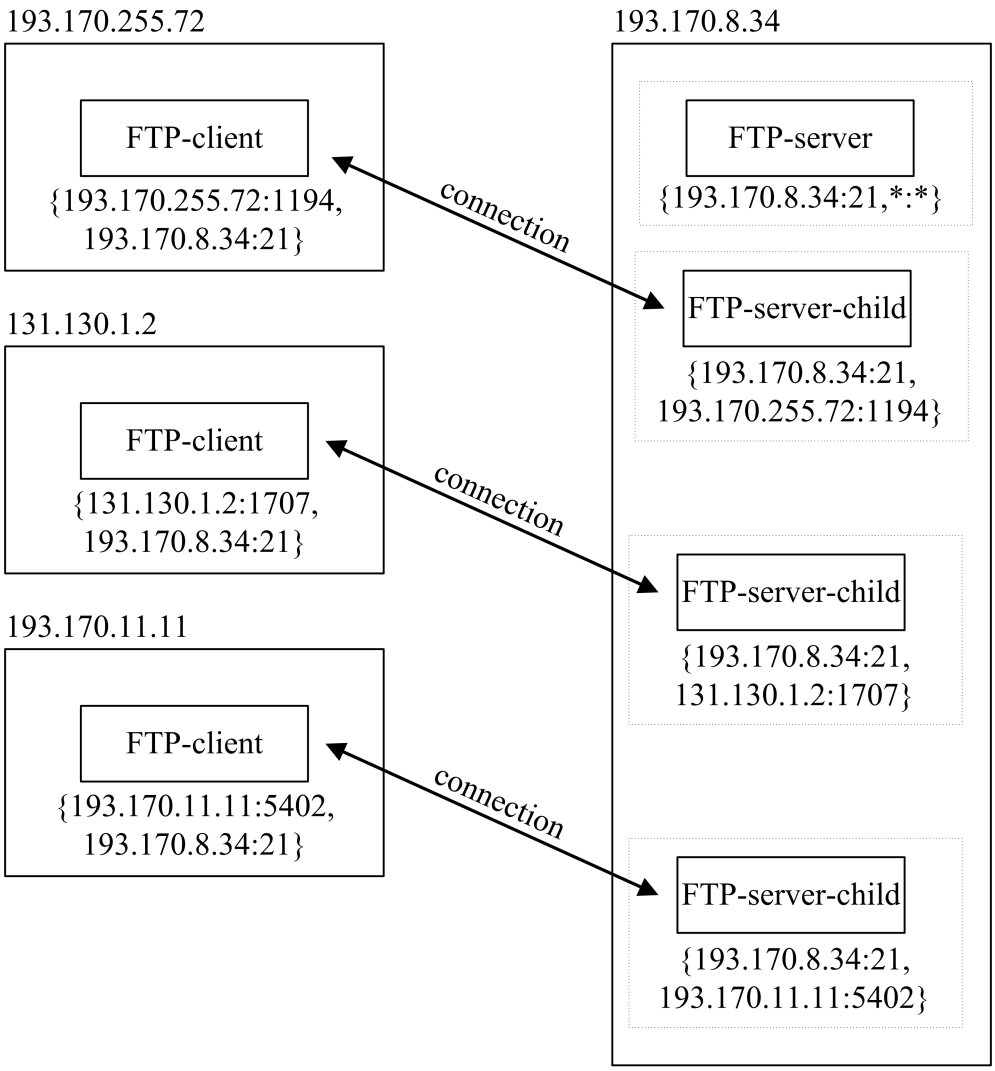
Der Simple-Server beginnt sofort mit der Serverfunktion während der Forking-Server an dieser Stelle einen *child process* startet. Der **`fork()`** SysCall dupliziert unter anderem alle *file descriptors*, und damit auch alle *sockets* (bzw. *socket descriptors*) für den Kindprozess.



Der Server schließt dann den *connected socket* und der *server child process* den *listening socket*.



Für jede weitere Client-Anforderung wiederholt sich dieser Vorgang. Die letzte Abbildung zeigt einen Forking-Server, der gleichzeitig mit 3 Clients kommuniziert.



Wird eine Verbindung beendet, dann terminiert üblicherweise der Kindprozess. Besondere Vorkehrungen im Elternprozess (listening server) sind notwendig, damit der gerade terminierte Kindprozess nicht in den *Zombie*-Zustand fällt.

3.6.3. Pre-Forking-Server

Der Pre-Forking-Server erzeugt zu Beginn, wie ja schon der Name sagt, ein gewisse Anzahl von Server-Kindprozessen, die alle über denselben *listening socket* verfügen. Jeder Kindprozess blockiert im **`accept()`** SysCall; der *parent process* ruft kein **`accept()`** auf. Der *parent process* hat eigentlich keine operative Aufgabe mehr, außer vielleicht das terminieren aller Kindprozesse, wenn das Service niedergefahren werden muss. Nähere Details, sowie eine ausführliche Diskussion über Probleme und deren Lösung findet sich im Kapitel 27 von [3].

3.6.4. Pre-Threading-Server

Threads sind sogenannte leichtgewichtige Prozesse (*light weight processes*), welche nicht soviele Systemressourcen wie ein Prozess benötigen und um einen Faktor 10-100 schneller erzeugt werden können. Alle Threads eines Prozesses teilen sich denselben Speicher. Aus diesem Grund kann auch eine einfache Thread-Kommunikation über globale Variablen implementiert werden. Genauere Details z.B. im Kapitel 23 von [3].

Der Pre-Threading-Server arbeitet im Prinzip genauso wie der Pre-Forking-Server, nur das für jeden Client ein Thread und nicht ein Prozess verwendet wird. Näheres dazu wiederum im Kapitel 27 von [3].

4. Glossar

3WHS ***3 Way Hand-Shake***

bezeichnet den Prozess des Aufbaus einer TCP-Verbindung.

API ***Application Programming Interface***

manchmal auch als *Application Programmers Interface* bezeichnet stellt i.A. ein (Software-) Schnittstelle zwischen Anwendungs- und Systemprogrammen dar.

ARP ***Address Resolution Protocol***

die Zuordnung zwischen physikalischer Netzwerk(mediums)adresse (z.B. Ethernet MAC Adresse) und der logischen Internetadresse erfolgt mittels ARP.

ENET ***Ethernet-Modul des Betriebssystems***

auch als Ethernet (oder ganz allegmein Netzwerk-) Treiber des Betriebssystems.

FTP ***File Transfer Protocol***

eines der ältesteten (TCP basierte) Anwendungsprotokoll im Internet um Dateien zu übertragen.

IP ***Internet Protocol***

das zentrale Protokoll in der TCP/IP- bzw. Internet-Protokollfamilie auf der Ebene 2 (entspricht ungefähr dem OSI-Layer 3)

MAC ***Medium Access Control***

regelt die Art und Weise wie in einem Netzwerk auf das Medium zugegriffen wird.

MSL ***Maximum Segment Lifetime***

Zwei Mal MSL ist die Wartezeit (*timeout*) vom TIMED_WAIT in den CLOSED Zustand der TCP-*State-Machine*. Dieser *timeout*-Wert beträgt ca. 2 Minuten.

MSS ***Maximum Segment Size***

ein TCP-Parameter der die maximale Segmentlänge eines TCP-Segments angibt. Der Wert wird i.A. so gewählt, dass ein TCP-Segment in ein IP-Datagramm passt, welches seinerseits wieder in einen Netzwerk-Rahmen passt, sodass auf der IP-Ebene nicht fragmentiert werden muss.

NFS ***Network File System***

der „klassische“ verteilte Filesystem-Standard in der UNIX-Welt. Von SUN-Microsystems entwickelt und 1985 als Teil von Sun-OS veröffentlicht.

NTP ***Network Time Protocol***

das (UDP basierte) Protokoll für die Uhrensynchronisation im Internet.

STD ***State Transition Diagram***

Das STD (deutsch: Zustandsübergangsgraph) dient der graphischen Beschreibung von sequentiellen Prozessen.

TCP ***Transmission Control Protocol***

das „verbindungsorientiertes“ Transportprotokoll (Ebene 3, entspricht ungefähr dem OSI-Layer 4) der Internet-Protokollfamilie.

UDP ***User Datagram Protocol***

das „verbindungslose“ Transportprotokoll (Ebene 3, entspricht ungefähr dem OSI-Layer 4) der Internet-Protokollfamilie.

5. Lehrzielorientierte Fragen

1. Wie öffnet ein TCP-Server eine Verbindung?
2. Wie öffnet ein TCP-Client eine Verbindung?
3. Was versteht man unter einem Port?
4. Was ist ein Socket?
5. Wodurch ist eine TCP/IP Verbindung zwischen Client und Server eindeutig identifiziert?
6. Wie funktioniert der *3 way handshake* bei einem TCP-Verbindungsaufbau?
7. Was ist der Unterschied zwischen UDP und TCP?
8. * Wie hängt das OSI-7-Schichtenmodell mit dem Internet-Modell zusammen?
9. Was versteht man unter einem *iterative server*?
10. Wieso können zu einem iterativen Server mehrere TCP/IP-Verbindungen im Zustand ESTABLISHED sein?
11. Wer verwaltet die lokalen TCP bzw. UDP Portnummern?
12. Nach welchen Regeln werden die Portnummern vergeben?
13. Was versteht man unter einem *raw socket*?
14. Wie nennt man einen UDP-Socket noch?
15. Was ist ein *stream socket*?
16. Was ist ein connected socket?
17. Was ist ein listening socket?
18. Warum verfügt der *child process* eines Forking-Servers sowohl über den *listening socket* als auch über den *connected socket*?
19. Welche Aussage hat der *backlog* Wert?
20. Manche Betriebssysteme haben anstelle des **`fork()`** einen **`spawn()`** SysCall. Worin liegt (meist) der Unterschied zwischen **`fork()`** und **`spawn()`**?
21. Welche Auswirkung kann die *sliding window* Fluss-Steuerung auf den Datenaustausch zwischen Client und Server haben? (Tipp: Denken Sie an die Eigenschaften der SysCalls **`read()`** und **`write()`**)
22. * Was ist im Sinne von UNIX bzw. Linux ein *socket descriptor*?
23. * Unter welchen Bedingungen kehrt der **`read()`** SysCall zum Aufrufer zurück?

24. * Was bedeutet die Aussage „die Funktion **`accept()`** blockiert“?
25. * Wie kann eine bestehende TCP-Verbindung „abreißen“?
26. * Welche Vorkehrungen sind im *parent process* notwendig, damit ein terminierter *child process* nicht in den *Zombie*-Zustand fällt?