Inhalt, vorige Seite, nächste Seite

Hintergrundinformationen/Begriffe

Netzwerkverbindungen

Netzwerkverbindungen können entweder verbindungsorientiert oder verbindungsfrei sein.

Verbindungsorientieres Netzwerk/Protokoll

Ein verbindungsorientiertes Netzwerk muß zuerst eine Verbindung mit einer anderen Anwendung aufbauen, bevor es Daten über das Netzwerk schicken kann. Als Vergleich eignet sich das Telefon: Um mit jemanden reden zu können, muß man zuerst die Nummer wählen und auf eine Antwort warten. Man kann solange nicht reden, bis jemand den Hörer auf der anderen Seite abgehoben hat. Auf die gleiche Weise können bei einem verbindungsorientierten Netzwerk keine Daten übertragen werden, bis eine Verbindung aufgebaut wurde. Das TCP-Protokoll ist ein verbindungsorientiertes Protokoll.

Verbindungsfreies Netzwerk/Protokoll

Bei einem verbindungsfreien Netzwerk wird keine direkte Verbindung zwischen Sender und Empfänger aufgebaut. Deshalb muß jede versendete Nachricht die für die Auslieferung wichtigen Daten enthalten. Vergleichbar ist dies mit dem Postdienst. Man muß die vollständige und richtige Adresse auf den Briefumschlag schreiben, damit die Post ihn ausliefern kann. Auch eine Nachricht im verbindungsfreien Netzwerk enthält die komplette und hoffentlich richtige Empfängeradresse. Das UDP-Protokoll ist ein verbindungsfreies Protokoll.

Virtuelle Kreise

Ein virtueller Kreis ist eine Verbindung, die sich wie eine dedizierte Punkt-zu-Punkt-verbindung darstellt. Wenn man z.B. zwischen Salzburg und München telefoniert, sieht es so aus, also würde eine direkt Verbindung zwischen Salzburg und München stehen, sozusagen ein eigenes Kabel. In Wirklichkeit wird das Gespräch von der Telefongesellschaft zwischen verschiedenen Schaltstellen geleitet. Es existiert also keine dedizierte Leitung zwischen Salzburg und München, auch wenn es so aussieht. Innerhalb des TCP/IP-Familie arbeitet das TCP-Protokoll mit einem virtuellen Kreis.

Sockettypen

Man unterscheidet im wesentlichen zwei Typen von Sockets. Mit der Wahl des Socket-Typs wird automatisch die Art des Datenaustauschs zwischen den beteiligten Prozessen festgelegt:

Stream-Sockets sind verbindungsorientiert und zuverlässig. Verbindungsorientiert heißt, daß zwischen den beiden beteiligten Prozessen eine feste Verbindung aufgebaut wird (ähnlich einer Standleitung), über die Daten in beide Richtungen fließen können. Sie bleibt solange bestehen, bis einer der Prozesse die

Verbindung abbaut. Daten werden darüber, wie der Name schon andeutet, in Form eines kontinuierlichen Byte-Stroms ohne äußere Struktur transportiert (analog einer bidirektionalen Pipe). Zuverlässig heißt, daß alle über einen Stream-Socket gesandten Daten garantiert am Ziel ankommen, und zwar in der Reihenfolge, in der sie abgeschickt wurden. Stream-Sockets werden oft auch *Virtual Circuits* oder *TCP-Sockets* genannt.

Datagram-Sockets sind verbindungslos und nicht zuverlässig. Verbindungslos heißt, daß keine feste Verbindung zwischen den beiden beteiligten Prozessen aufgebaut wird. Daten werden in Form von Paketen gesandt. Da keine Verbindung zum Ziel-Socket besteht, muß dessen Adresse explizit beim Versenden mit angegeben werden. Nicht zuverlässig heißt, daß nur garantiert wird, daß ein Datenpaket abgeschickt wird, nicht aber, daß es sein Ziel erreicht, daß es genau einmal am Ziel ankommt, oder daß mehrere Pakete ihr Ziel in der Reihenfolge des Abschickens erreichen. Datagram-Sockets werden oft auch *UDP-Sockets* genannt.

Es stellt sich die Frage, wozu es zwei unterschiedliche Socket-Typen gibt. Stream-Sockets besitzen zwar gegebenüber den Datagram-Sockets offensichtliche Vorteile, jedoch werden diese durch einen erhöhten Overhead beim Datentransport erkauft. Auch die Erhaltung von Paketgrenzen ist ein zu beachtender Aspekt. Die Verwendung von Datagram-Sockets bietet sich an bei unkomplizierten Kommunikationsstrukturen, wo etwa eine Anfrage an einen Server-Prozeß gesendet und diese einfach wiederholt wird, falls keine Antwort innerhalb eines bestimmten Zeitraumes eintrifft.

Telnet Protocol

Telnet ist ein remote access Protokoll, mit dem ein lokales Terminal in die Lage versetzt wird, wie ein Terminal an einem entfernten Rechner zu funktionieren. Mittels Telnet kann ein Personal Computer oder ein Terminalfenster in einer Benutzungsoberfläche in ein Terminal verwandelt werden. Das Telnet-Protokoll wird meisten durch zwei Prozesse implementiert. Ein Client Prozeß initiiert, vom Benutzer gestartet, eine Session mit dem entfernten System. Auf diesem arbeitet ein Server, welcher auf Verbindungen mit entfernten Benutzern wartet und diese dann bearbeitet.

File Transfer Protocol (FTP)

Das File Transfer Protocol wird benutzt, um Dateien über ein Netzwerk zu transportieren. Ein Rechner kann eine Verbindung mit einem entfernten Rechner initiieren und darüber Dateien senden und empfangen, Verzeichnisse anzeigen lassen und einfache Kommandos ausführen. Genauso wie Telnet, wird FTP üblicherweise als Client-Server Software implementiert. Der Benutzer arbeitet mit dem lokalen Client, der mit dem entfernten Server kommuniziert. Dieser Server bearbeitet dann die Befehle des entfernten Benutzers und schickt die Ergebnisse an den Client zurück.

Die TCP/IP - Protokollfamilie

Das **Transmission Control Protocol / Internet Protocol i**st ein weitverbreiteter Standard für die Verbindung von Rechnern vieler verschiedener Hersteller. Die verschiedenen Implementierungen der Protokolle der TCP/IP-Familie sind untereinander kompatibel, so daß Daten und Dienste von Rechnern verschiedener Hersteller gemeinsam genutzt werden können.

Die Entwicklung von TCP/IP wurde vom amerikanischen Verteidigungsministerium (Department of Defense/DoD) bzw. dessen Forschungsbereich Defense Advanced Research Projects Agency (DARPA) in Auftrag gegeben, um für ihr ARPANET ein verläßliches Kommunikationsprotokoll zu bekommen. Ziel war die Definition und Implementierung einer Protokollfamilie. Das Ergebnis ist heute unter dem Namen TCP/IP bekannt.

TCP/IP wurde danach das Standardprotokoll auf dem DARPA Internet, einer Reihe von Netzwerken, zu denen unter anderen ARPANET, Military Network/MILNET, National Science Foundation Network/NFSnet und diverse Universitäts-, Forschungs- und weitere Militärnetzwerke gehörten.

Dieses erste größere internetwork, wie im TCP/IP Sprachgebrauch ein mehrere Netzwerke verbindendes Netzwerk heißt, ist heute unter dem Namen Internet bekannt und umfaßt tausende Netzwerke mit zusammen einigen millionen Rechnern.

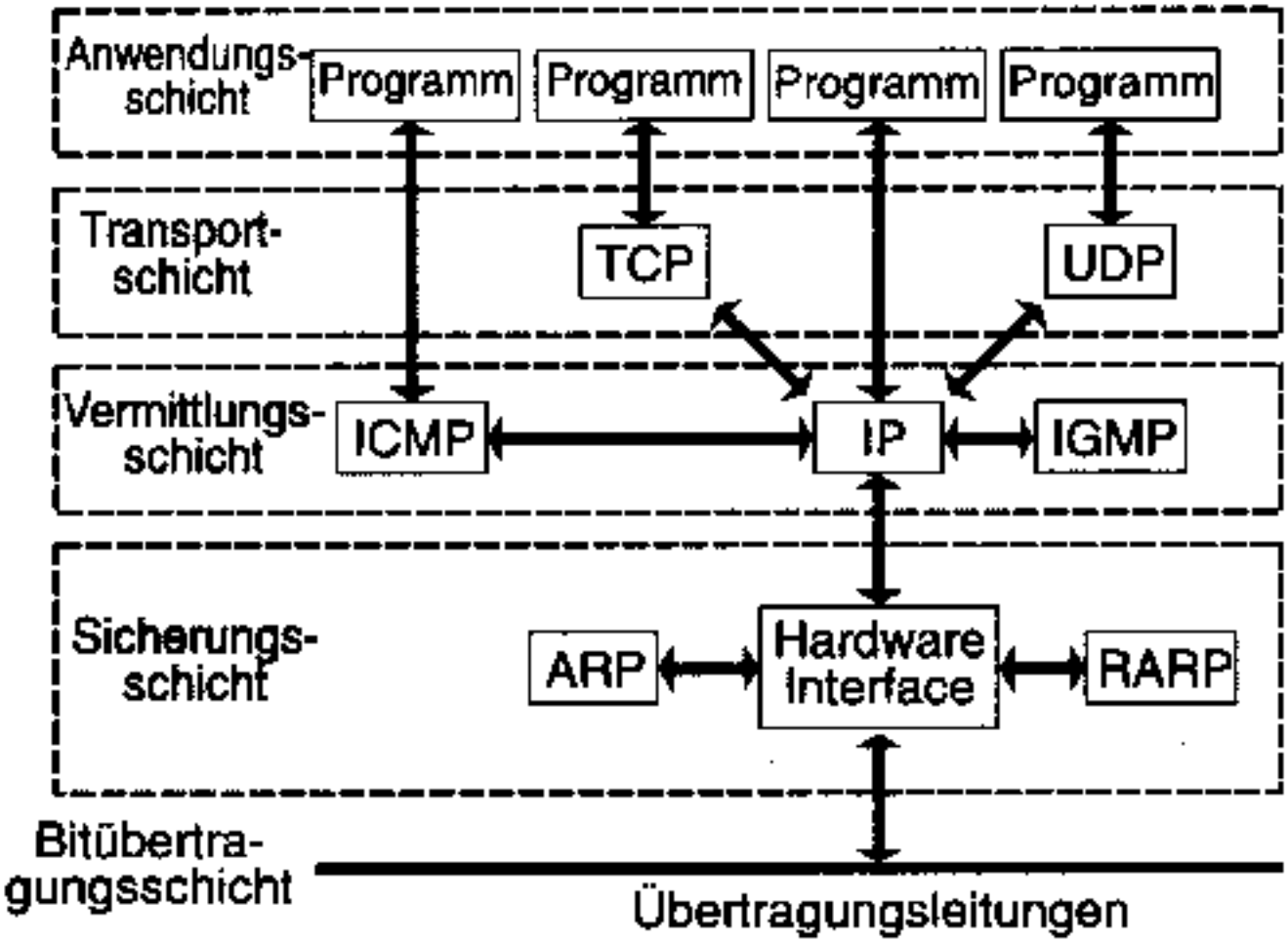
Mittlerweile ist TCP/IP bei fast jedem Rechner- oder Betriebssystemhersteller im Angebot und stellt dadurch einen Quasi-Standard bei den Netzwerkprotokollen dar.

Unter TCP/IP versteht man im Allgemeinen folgende Protokolle:

- Das Internetprotokoll (IP) ist ein Protokoll auf der Vermittlungsschicht des OSI-Modells und kümmert sich um die Datenübertragung zwischen den Hostrechnern.
- Das Transport Control Protokoll (TCP) liegt auf der Transportschicht und Überträgt Daten zwischen den Anwendungen.
- Das User Datagram Protocol (UDP) ist ein weiteres Protokoll auf der Transportschicht . Es überträgt ebenfalls Daten zwischen den Anwendungen, ist aber weniger komplex als das TCP-Protokoll.
- Das Internet Control Message Protocol (ICMP) überträgt Netzwerkfehlermeldungen und ist bei der Fehlerauswertung innerhalb der Netzwerksoftware hilfreich

Das IP-Protokoll

Die Vermittlungsschicht ist das "Herz" jedes auf TCP/IP basierenden Netzwerkes. Die Vermittlungsschicht entscheidet wie die Daten verpackt und über das TCP/IP-Netzwerk verschickt werden. Für diese Aufgabe benützt die Vermittlungsschicht das IP-Protokoll. Darum wird die Vermittlungsschicht in der Literatur häufig als IP-Schicht bezeichnet.Folgende Grafik zeigt die Zusammenhänge zwischen den einzelnen Schichten:



Das Internet Protocol ist für den Transport von Daten über mehrere Netzwerke hinweg zuständig. Das Internet Protocol nimmt Datensegmente vom TCP oder UDP entgegen und packt diese in Pakete ein, die Datagramme genannt werden. Für jedes dieser Datagramme wird dann ein Leitweg zum Ziel bestimmt. Datagramme werden dann durch ein internetwork, bestehend aus vielen Netzwerken, die durch Gateways verbunden sind, geschickt, bis sie ihr Ziel erreichen.

Das IP stellt einen Adressierungsmechanismus zur Verfügung, welcher die Wegewahl zwischen Netzwerken ermöglicht. Jedes Datagramm besteht aus einem Datenteil und einem Header, der sowohl Quell- als auch Zieladresse beinhaltet. Mit dieser Information kann jeder Rechner ein Datagramm entweder direkt oder über eine Reihe von Gateways zu seinem Ziel senden.

Um möglichst viele Netzwerke mit ihren unterschiedlichen Paketlängen zu unterstützen, kann das IP

Datagramme auf ihrem Weg zum Ziel in kleinere Datagramme unterteilen. Dieses Verfahren heißt Fragmentierung und wird erst am Zielrechner durch das Zusammenfügen eines fragmentierten Datagramms rückgängig gemacht.

Das IP ist ein unzuverlässiges Protokoll, da es lediglich für jedes Datagram den Weg zum Ziel bestimmt und es absendet. Es wird jedoch keine Überprüfung des Empfang vorgenommen, so daß Datagramme durchaus verloren gehen können. Diese Aufgabe wird in der Transportschicht vom TCP übernommen. Wird jedoch das UDP in der Transportschicht verwendet, ist die Empfangsbestätigung vom Anwendungsprotokoll zu leisten.

Das IP gehört zu den verbindungslosen Protokollen, da jedes einzelne Datagramm mit der vollständigen Adreßinformation versehen, separat auf den Weg geschickt wird.

Die Internetadresse

Eine Internetadresse ist eine sogenannte IP-Adresse. Dabei wird die IP-Adresse nicht dem Hostrechner zugeordnet, sondern dem Netzwerkadapter (z.B. Netzwerkkarte). Jeder Computer kann mehrere Netzwerkadapter besitzen, d.h. er kann auch mehrere IP-Adressen besitzen. Im folgenden wird zur Vereinfachung davon ausgegangen, daß die IP-Adresse dem Computer zugeordnet wird.

Die Dotted Decimal-Notation

Eine IP-Adresse ist 32 Bit lang. In C kann man sie also als "Long Integer" Variable darstellen.

Üblicherweise wird die IP-Adresse aber in einer speziellen Notation geschrieben, nämlich der "Dotted Decimal-Notation". Diese Notation stellt jedes Byte der IP-Adresse als Dezimalzahl dar und trennt die einzelnen Bytes mit Punkten (XXX.XXX.XXX.XXX). Folgende Zahlen stellen dieselbe IP-Adresse in unterschiedlichen Zahlen dar:

- IP-Adresse als binäre Zahl: 11000000 10100100 01100100 00001010
- IP-Adresse als Dezimalzahl: 3231998986
- IP-Adresse als Hexadezimalzahl: 0xC0A4640A
- IP-Adresse als Dotted Decimal-Notation: 192.164.100.10

Die IP-Adresse dekodieren

Das Internet besteht aus vielen kleinen unabhängigen Netzwerken. In der 32-bit Netzwerkadresse ist die Netzwerknummer und die Hostnummer enthalten, d.h. man kann mit der IP-Adresse den Hostrechners zu seinem Netzwerk zuordnen. Früher war das höchstwertige Byte der IP-Adresse die Netzwerkadresse und alle nachfolgenden Bytes die Hostadresse. Hier war es aber nur möglich bis zu 255 Netzwerke anzusprechen. (Nullbit und Nullbyte sind reservierte Adressen)

Heute verwendet man die höchstwertigsten Bits im höchstwertigen Byte, um eine Adressklasse festzuzulegen. Man unterscheidet die Klassen A bis E.

1 Bit	7 Bits	24 Bits
0	Network-ID	Host-ID

Class A-Adresse

Bei der Klasse A wird das erste bit als Klassendefinition verwendet, damit bleiben nur noch 7 Bit für die Netzwerk-ID zur Verfügung. Es könnten also nur 127 Netzwerke miteinander verbunden werden, die dafür aber 24 Bit für die Hostadresse bereitstellen, könnten theoretisch 16.777.216 Rechner miteinander verbunden werden.

Bei der Klasse B werden maximal die ersten zwei Byte für die Klassendefinition und die Netzwerkadresse verwendet, d.h. es können bei Verwendung von zwei Bit als Klassendefinition 14 Bit als Netzwerkadresse verwendet werden. Mit den restlichen 16 Bit lassen sich insgesamt 65.536 Rechner verbinden.

2 Bits		14 Bits	16 Bits
1	0	Network-ID	Host-ID

Class B-Adresse

Analog dazu wird in der Klasse C verfahren. Hier werden die ersten drei Bit als Klassendefinition verwendet. Dementsprechend weniger Bit stehen für die Adressierung der einzelnen Hostrechner zur Verfügung.

3 Bits			21 Bits	8 Bits
1	1	0	Network-ID	Host-ID

Class C-Adresse

Auf die Klasse D für Mutlicastadressen und die Klasse E für zukünftige Verwendungen wird hier nicht eingegangen.

Die Subnetz Adressierung

Da das Adressierungschema für einige hundert Netzwerke konzipiert war, die Zahl der Netzwerke aber stetig stieg, taten sich bald zwei Probleme auf.

- Der zentrale administrative Aufwand für die Adressverwaltung stieg enorm.
- Die Routing-Tabellen nahmen unhaltbare Dimensionen an.

Im Angesicht dieser Probleme wurde das Adressierungsschema um das Subnetzkonzept erweitert. Pro Organisation wird nur noch eine Netzwerkadresse zugewiesen. Die Klasse dieser Adresse richtet sich dabei nach der Zahl der Netzwerke und Rechner der Organisation. Dadurch werden die beiden Probleme gelöst.

Innerhalb einer Organisation wird jetzt das Rechnerfeld der Adresse, welches man auch als lokales Feld bezeichnen kann, weiter unterteilt in ein Netzwerkfeld, das Subnetzfeld, und ein Rechnerfeld. Da diese lokale Aufteilung außerhalb der Organisation nicht sichtbar ist, brauchen externe Gateways lediglich einen routing-Eintrag für ein solches Netzwerk.

Trotz der auf den ersten Blick unglaublichen Anzahl von Adressen war schnell abzusehen, daß die Zahl der Adressen bei der explosionsartigen Verbreitung von LANs und der TCP/IP Protokollfamilie nicht mehr lange ausreichen würde. Als temporäre Maßnahme werden jetzt Organistionen auch mehrere Class C Adressen zugeordnet, um den Engpaß bei den Class B Adressen zu umgehen. Class A Adressen können gar nicht mehr vergeben werden. Parallel zu diesen Maßnahmen wird ein neues Adressierungsschema getestet.

Die Subnetzmaske

Wenn innerhalb einer Organisation Subnetze verwendet werden, muß eine sogenannte Subnetzmaske ausgewählt werden. Diese Maske unterteilt das lokale Adreßfeld in den Subnetz- und den Rechnerteil.

Eine Subnetzmaske besteht wie eine IP-Adresse aus 32-Bit. Die Netzwerk- und Subnetzfelder werden in der Maske durch gesetzte Bits repräsentiert. Der Rechnerteil weist Null-Bits auf.

Will ein Rechner mit einem anderen kommunizieren, muß er erst einmal herausfinden, ob dieser sich auf dem gleichen Netzwerk befindet. Dazu nimmt er die Zieladresse und blendet mit Hilfe der Subnetzmaske den Rechneranteil aus. Ist die so erhaltene Netzwerkadresse mit der eigenen identisch, liegt der Zielrechner im gleichen Netzwerk. Anderenfalls muß die Kommunikation über ein Gateway erfolgen.

Prinzipiell kann jedes LAN innerhalb einer Organisation seine eigene Einteilung zwischen Subnetz- und Rechnerteil haben. In der Praxis wird dies jedoch sehr unübersichtlich und von vielen Betriebssystemen auch nicht fehlerfrei unterstützt.

Die Transportprotokolle

Die Transportschicht

Wie bereits gezeigt wurde, ist die Vermittlungsschicht mit dem IP-Protokoll für die Übertragung zwischen den einzelnen Hostrechner verantwortlich. Die Transportschicht mit ihren Protokollen übernimmt nun die Aufgabe, den Datenaustausch zwischen dein einzelnen Anwendungen zu betreiben. Die wichtigsten Protokolle der Transportschicht sind UDP und TCP, welche im folgenden noch erklärt werden.

Die Transportprotokolle

Die Ports der Transportschicht

In der TCP/IP-Terminologie entspricht ein Port einer IP-Adresse, wobei TCP/IP den Port aber mit einem Protokoll assoziiert und nicht mit einem Computer. Die Protokolle der Transportschicht speichern Quell- und Zielport des Protokolls. Dies ist vergleichbar mit den Schnittstellen des Computers (z.B. Druckerport): Nachrichten werden einfach an die dazugehörigen Ports verschickt (z.B. Druckauftrag an LPT1).

Bei eigener Wahl einer Port-Nummer besteht immer die Gefahr, daß diese bereits anderweitig belegt ist, sei es durch Betriebssystemdienste oder andere Prozesse. Aus diesem Grund ist der Wertebereich der Port-Nummern folgendermaßen unterteilt:

0 -- 255	Dieses sind die Port-Nummern der sogenannten *well known services* wie beispielsweise **ftp**, **telnet** und anderen.
256 -- 1023	In diesem Bereich finden sich die *UNIX-specific services*, z.B. **who** und **talk**
024 -- 65535	Dies ist der für Anwender freie Bereich, durchsetzt allerdings von den *registered ports*. Ein Gesamtverzeichnis vorbelegter Port- Nummern findet sich im RFC 1062 (*assigned numbers*).

Möchte man einen bestimmten, über Sockets angebotenen Dienst selbst in Anspruch nehmen, so ist dafür das Wissen um die Port-Nummer und das verwendete Protokoll vonnöten. Diese Angaben finden sich unter UNIX in der Datei `/etc/services`.

Einige Ports für die Transportprotokolle:

- FTP Protocol 17
- Telnet Protocol 23
- Simple Mail Transfer Protocol 25
- Finger Protocol 79

Die Transportprotokolle

Das UDP-Protokoll

Das UDP-Protokoll arbeitet verbindungsfrei. Man kann es mit einem Postdienst vergleichen. Wenn man die Analogie mit dem Postdienst ein wenig abändert, kann man die Beziehung zwischen UDP, Ports und Anwendungen einfach verstehen. Der Hostcomputer stellt das Postamt dar, die Briefkästen sind die Ports und die Besitzer der Briefkästen die Anwendungsprotokolle. Das IP-Protokoll ist das Auslieferungssystem des Netzwerks. Nun kann man sagen, das IP ist der Postwagen und die Transportprotokolle sind die Fahrer/Postboten. Die Postautos (IP) liefern ganze Wagenladungen an Post (Daten) zwischen den Postämtern (Hostcomputern) aus. Nachdem die Post sortiert ist, werfen die Postboten (UDP) die Briefe (Daten) in die Briefkästen (Ports). Die Besitzer der Briefkästen (Anwendungen) überprüfen die Briefkästen regelmäßig und entnehmen ihre Post. Es erfolgt keine Rückmeldung, ob die Daten angekommen sind oder nicht. Der Vorteil von UDP ist, daß es billig ist, d.h. nicht viel Aufwand zur Erstellung einer Anwendung nötig und auch die Wartung ziemlich leicht zu überblicken ist.

Die Transportprotokolle

Das TCP-Protokoll

Das TCP-Protokoll arbeitet verbindungsorientiert. Bei der Datenauslieferung steht die Verbindung im Mittelpunkt und nicht der Port. Eine leicht verständliche Analogie ist eine Telefonkommunikation. Der Hostcomputer ist das Büro, eine Telefonnummer ist ein Port und der Telefonanruf die Verbindung. Die Angestellten im Büro sind die Anwendungsprotokolle und die geführten Gespräche entsprechen dem Austausch von Daten. Die Angestellten (Anwendungsprotokolle), die in diesem Büro (Hostcomputer) arbeiten, nutzen das Telefonsystem (IP) voll aus. Im Büro (Hostcomputer) wird jedem Mitarbeiter (Anwendungsprotokoll) eine eigene Nummer zugewiesen (Port). Leitet die Telefongesellschaft (IP) ein Gespräch an das Büro (Hostcomputer) weiter, klingelt ein Telefon. Die Telefonnummer (Port) des eingehenden Anrufs bestimmt, wer den Anruf (Verbindung) entgegennimmt. Wollen de Anrufer und der Mitarbeiter miteinander reden, kommt es zur Konversation (Datenaustausch). Das TCP-Protokoll ist ein teueres Protokoll, d.h. es ist viel Aufwand nötig und es zu implementieren und zu warten.

Das Socket Interface

Was ist ein Socket?

Ein Socket ist ein standardisierter interprozess Kommunikationskanal für Anwendungen (ähnlich den Pipes). Im Gegensatz zu Pipes unterstützen Sockets auch die Kommunikation zwischen nichtzusammenhängenden Prozessen oder gar zwischen Prozessen, die auf anderen Rechnern laufen. Das Socket ist jeweils der Endpunkt der Kommunikation. Das Socket wurde nach dem I/O-Kommunikationkonzept von UNIX entwickelt und wird auch so angesprochen. Es wird unter C ähnlich wie ein Filedescriptor behandelt. Das Socket-Modell verwendet also grundsätzlich einen Öffnen-Schreiben/Lesen-Schließen-Prozeß.

Das Socket Interface ist nun eine Gruppe von Funktionen für TCP/IP Netzwerke. Das Socket Interface definiert eine Anzahl von Funktionen (Routinen), die Programmierer zum Entwickeln von Anwendungen für TCP/IP-Netzwerke verweden können.

Das Socket Interface

Einen Socket erzeugen

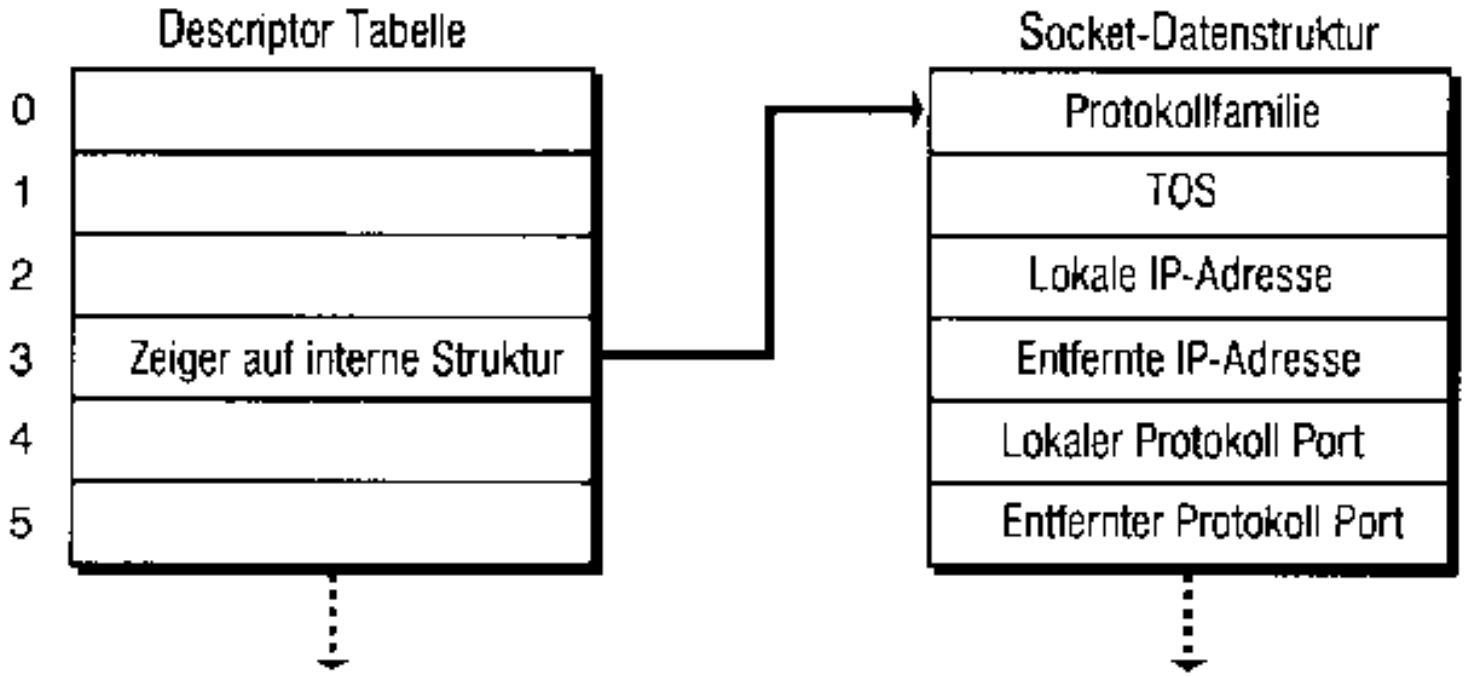
Mit einem Socket kann man sowohl verbindungsfreie als auch verbindungsorientierte Protokolle verwenden. Mit dem Socket Interface kann man beide Protkolltypen durch eine Socket-Verbindung nutzen. Allerdings verwendet man separate Schritte, um einen Socket zu erzeugen und ihn mit dem Zielhost zu verbinden.

Um einen Socket zu erzeugen, ruft man die socket()-Funktion auf. Diese Funktion liefert ein Handle zurück, das mit dem File-Handle vergleichbar ist.

```
int socket (int protokoll_familie, int socket_typ, int protokoll);
```

Die Protokollfamilie bestimmt die "Familien" (Sammlung) der Protokolle, die verwendet werden können. In unserem Falle ist dies die TCP/IP-Familie. Dies wird durch die Konstante AF_INET definiert. Der zweite Übergabeparameter gibt den Kommunikationstyp an. In diesem Falle SOCK_STREAM (Streams) für TCP und SOCK_DGRAM (Datagramme) für UDP. Der dritte Parameter gibt das zu verwendende Protokoll an, IPPROTO_TCP für TCP und IPPROTO_UDP für UDP.

Der Aufruf der socket()-Funktion bestimmt nur dir Protokollfamilie, den Socket-Typ und ein bestimmtes Protokoll. Es wird noch keine Netzwerkadresse angegeben. Der Aufruf liefert einen Descriptor zurück, der auf eine intere Datenstruktur zeigt. Diese Struktur enthält den Platz für vier Adressen: lokales IP, entferntes IP, lokaler Port, entfernter Port.



Das Socket Interface

Den Socket konfigurieren

Nachdem der Socket erzeugt wurde, werden noch weitere Funktionen aufgerufen, um den Socket zu konfigurieren. Wenn Daten durch den Socket übertragen werden sollen, kann man z.B. Streams oder Datagramme verwenden. Ebenso kann man das Socket für Client- oder Serverfunktionen konfigurieren. Folgende Tabelle zeigt einen Überblick über die Funktionen, die dafür verwendet werden.

Nutzung des Sockets	Lokale Information	Entfernte Information
Verbindungsorientierer Client	Ein Aufruf von connect() speichert lokale und entfernte Informationen in der Socket-Datenstruktur	
Verbindungsorientierer Server	bind()	listen() und accept()
Verbindungsfreier Client	bind()	sendto()
Verbindungsfreier Server	bind()	recvfrom()

Die bind()-Funktion weist nun dem Socket eine Adresse zu:

```
int bind (int socket_handle, struct sockaddr *lokale_socket_adresse, int adress_laenge);
```

bind() bindet den Socket-Deskriptor an einen Namen. Auf Seiten eines Server-Prozesses entspricht dies einer Bekanntgabe des Namens nach außen, da Client-Prozesse erst jetzt gezielt Verbindung mit ihm aufnehmen können. Der `bind()`-Aufruf ist auf der Server-Seite also obligatorisch.

Das Socket Interface

Verbindung zu einem Socket herstellen

Wie bereits beschrieben, baut ein verbindungsorientiertes Protokoll einen virtuellen Kreis zwischen den beiden Endpunkten auf. Dieser Kreis wird von TCP verwaltet, d.h. es hält die Verbindung offen, indem es Bestätigungsmeldungen zwischen den Endpunkten austauscht. Ein verbindungsorientierter Client nutzt die Funktion connect(), um einen Socket für die Netzwerkkommunikation zu konfigurieren. Diese Funktion speichert die Daten des lokalen und entfernten Endpunktes in der Socket-Datenstruktur. Als Parameter müssen der Funktion das Socket-Handle (Rückgabewert des connect()-Funktion), eine Struktur mit Informationen zum entfernten Host und die Länge dieser Andresstruktur übergeben werden:

```
int connect (int socket_handle, struct sockaddr *remote_socket_adresse, int adress_laenge);
```

Der zweite Parameter ist ein Zeiger auf eine bestimmte Adress-Struktur. Diese Struktur enthält eine Adressfamilie, einen Protokollport und eine Netzwerkhostadresse. Die connect()-Funktion speichert diese Informationen in der Socket-Struktur, die über den Socket-Handle referenziert wird.

Programme, die ein verbindungsorientiertes Protokoll verwenden, müssen genau wie Programme, die ein verbindungsfreies Protokoll verwenden, am Protokollport "horchen" ob Daten ankommen. Das ein verbindungsfreier Client keine Verbindung zu einem entfernten Host herstellt, muß er immer am Protokollport "horchen", um eine Antwort zu erhalten.

Das Socket Interface

Daten durch einen Socket übertragen

Folgende Liste zeigt eine Aufstellung der C-Funktionen zum Übertragen von Daten:

Funktion	Beschreibung
send()	Überträgt Daten durch einen verbindungsorientierten Socket. Spezielle Flags können verwendet werden, um das verhalten des Socket zu steuern.
write()	Überträgt Daten durch einen verbindungsorientierten Socket. Verwendet einfachen Datenpuffer
writev()	Überträgt Daten durch einen verbindungsorientierten Socket. Verwendet dabei nicht zusammenhängende Speicherbereiche als Datenpuffer
sendto()	Überträgt Daten durch einen verbindungsfreien Socket. Verwendet einen einfachen Datenpuffer
sendmdg()	Überträgt Daten durch einen verbindungsfreien Socket. Verwendet eine flexible Nachrichtenstruktur als Puffer.

Eine nähere Beschreibung der Funktionen findet man im GNU C Library - Reference Manual!

Das Socket Interface

Daten durch einen Socket übertragen

Daten durch einen verbindungsorientierten Socket senden

Die Parameternamen sprechen für sich:

```
int send (int socket_handle, void *nachrichen_puffer, int puffer_laenge, unsigned int flags);
```

send() schreibt Daten auf einen Socket-Deskriptor. Kann nur bei TCP-Sockets verwendet werden, wo der Empfänger durch die bestehende Verbindung bereits feststeht. Mit `flags` läßt sich ein spezielles Verhalten beim Senden der Daten einstellen, worauf hier nicht näher eingegangen wird.

```
int write (int socket_handle, char *nachrichten_puffer, int puffer_laenge);
```

Funktionier gleich wie send(). Es können jedoch keine Flags angegeben werden.

Das Socket Interface

Daten durch einen Socket übertragen

Daten durch einen verbindungsfreien Socket senden

Die Parameternamen sprechen für sich:

```
int sendto (int socket_hanlde, void *nachirchten_puffer, int puffer_laenge,
unsigned int flags, struct sockaddr *socket_adress_struktur, int
adress_strukutr_laenge);
```

sendto() schreibt Daten auf einen Socket-Deskriptor. Wird im allgemeinen nur bei UDP-Sockets verwendet. Der Empfänger ist hier explizit anzugeben. In der socket_adress_Struktur stehen die Empfängerdaten. Mit `flags` läßt sich ein spezielles Verhalten beim Senden der Daten einstellen, worauf hier nicht näher eingegangen wird.

Das Socket Interface

Daten durch einen Socket empfangen

Folgende Liste zeigt eine Aufstellung der Funktionen zum Empfangen von Daten. Die Funktionen korrespondieren mit den Funktionen zum Senden, d.h. die Parameter sind gleich:

Sendefunktion	**Korrespondierende Empfangsfunktion**
send()	recv()
write()	read()
writev()	readv()
sendto()	recvfrom()
sendmdg()	recvmsg()

Das Socket Interface

Die Funktion listen()

Die Listenfunktion reiht alle eingehenden Serviceanrufe in eine Queue und gibt sie anschließend zur Bearbeitung weiter. Dies ist nötig, damit keine Anrufe verlorengehen, wenn mehrere Anfornderungen zur gleichen Zeit eintreffen:

```
int listen (int socket_handle, int queue_laenge);
```

Man kann eine Maximallaenge von fünf Nachrichten angeben.

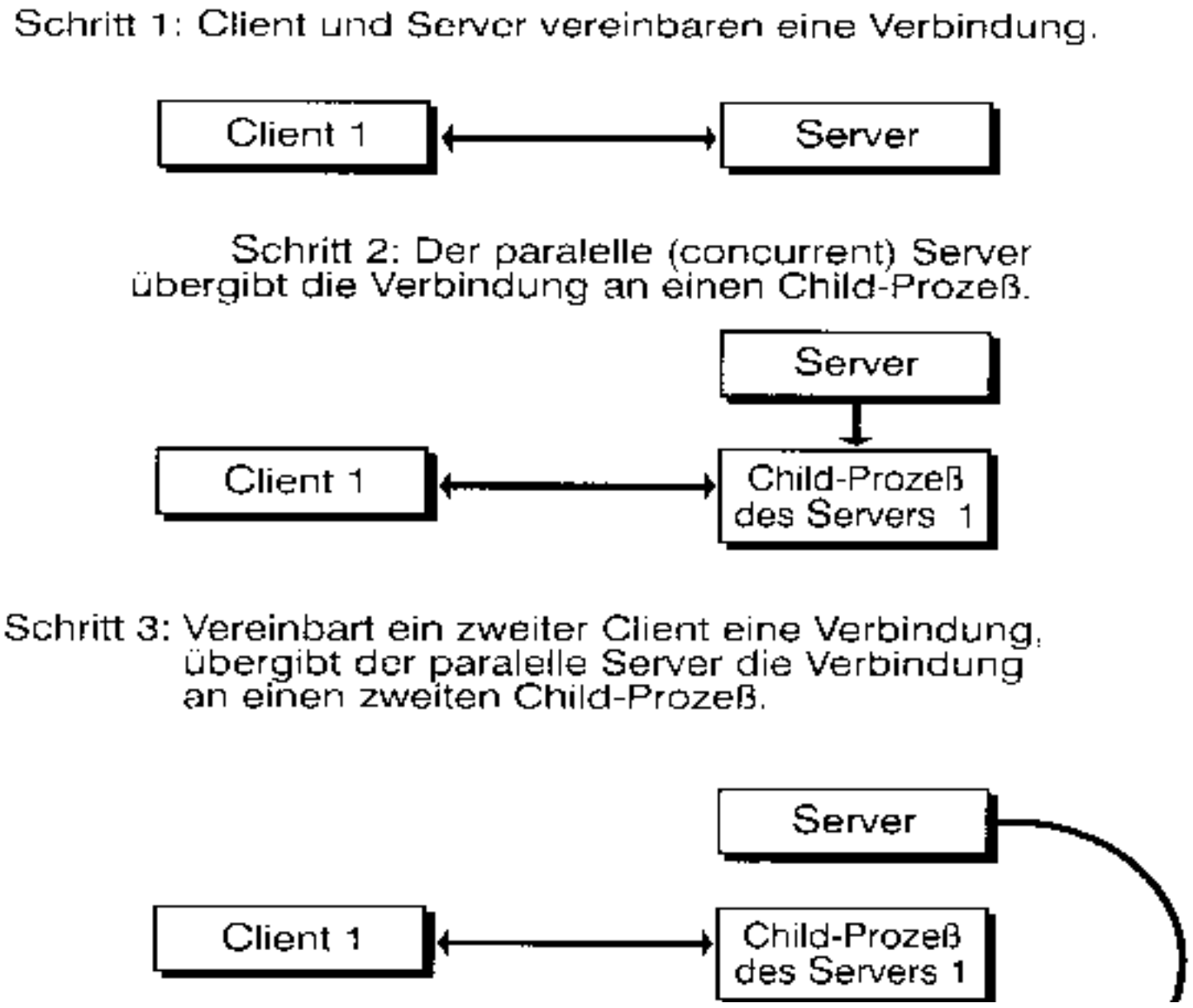
Das Socket Interface

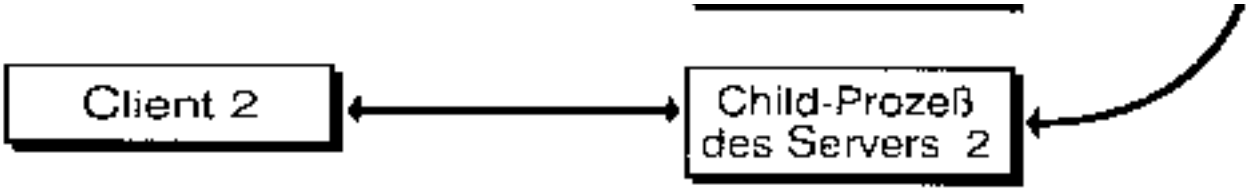
Die Funktion accept()

Wie der Name andeutet, akzeptiert die accept()-Funktion auf der Serverseite Verbindungen von einem Client. Nachdem eine Queue aufgebaut wurde, ruft ein Server die accept()-Funktion auf und wartet dann auf eingehende Verbindungen vom Client. Die Funktion benötigt drei Parameter: den Socket-Handle, die Socket-Adresse und die Adress-Länge:

```
int accept (int socket_Handle, struct sockaddr *socket_adresse, int
*adress_laenge);
```

Beim Eintreffen einer neuen Verbindungsanforderung an einem von accept() überwachten Socket, erzeugt die Socket-Implementierung automatisch einen neuen Socket und verbindet diesen dann sofort mit dem Client. Der ursprüngliche Socket bleibt dadurch weiterhin offen für weitere Verbindungen.




Client 2
Child-Prozeß
des Servers 2

Das Socket Interface

Schließen eines Sockets

Das Socket wird kann mit zwei verschiedenen Befehlen geschlossen werden:

```
int close (int socket_handle);

int shutdown (int socket_handle, int wie);
```

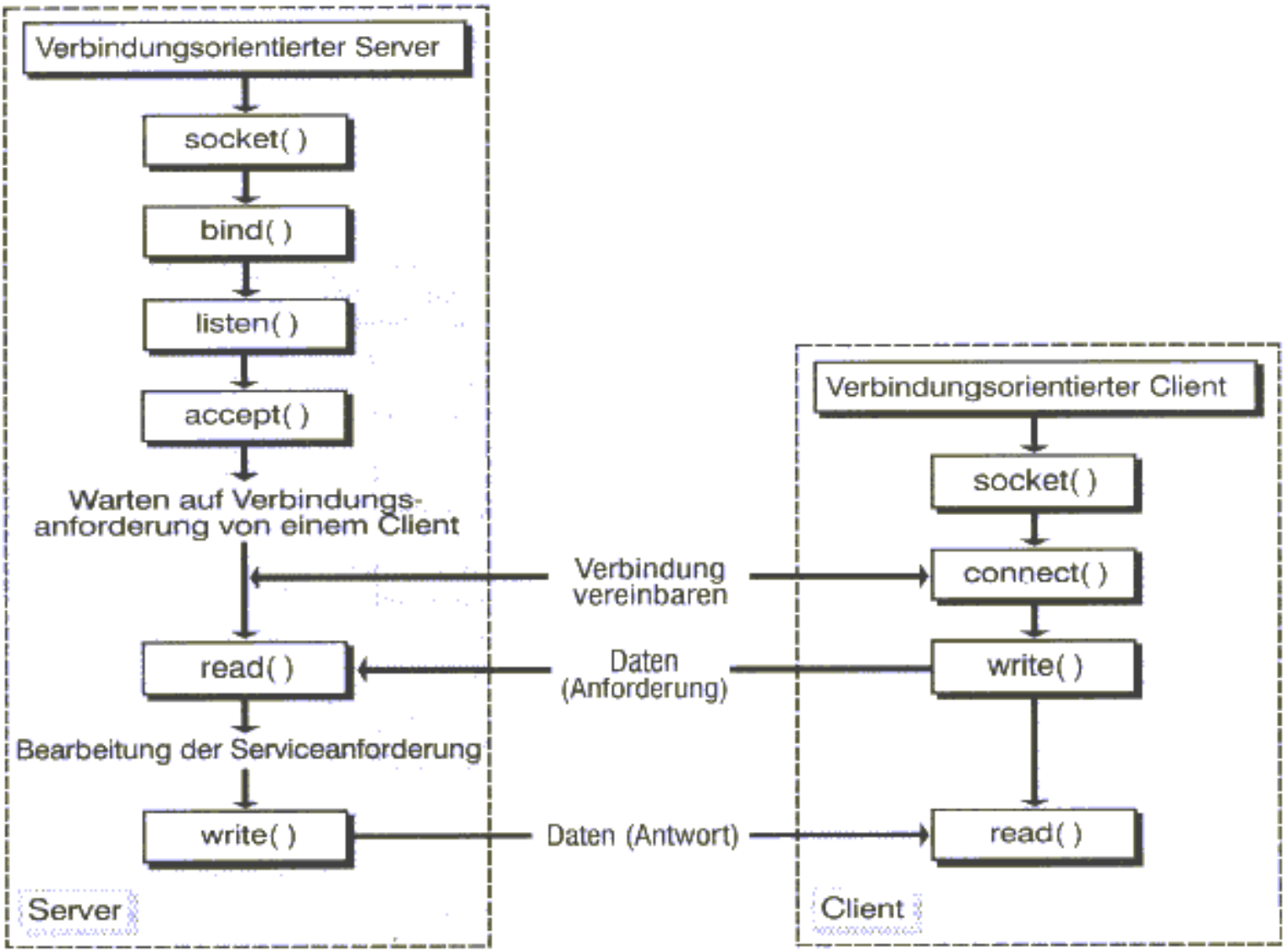
Beide Funktionen lößen den Socket-Deskriptor auf. Der Unterschied zwischen den beiden Funktionen liegt in der Behandlung von Daten, die evtl. noch der Übertragung harren. Während `close()` versucht, noch nicht übertragene Daten zuzustellen, bestehen bei `shutdown()` über den Parameter `'wie'` differenziertere Möglichkeiten, die Übertragung zu beenden.

0	Stoppt den Empfang von Daten. Weiter eingehende Daten werden ingoriert
1	Stoppt das Versenden von Daten. Wartende Nachrichten werden ignoriert. Überprüft, ob Daten abgekommen sind, wiederholt aber die Sendung nicht.
2	Stoppt Senden und Empfangen

Das Socket Interface

Verwendung der Funktionen auf einem verbindungsorientierten Protokoll

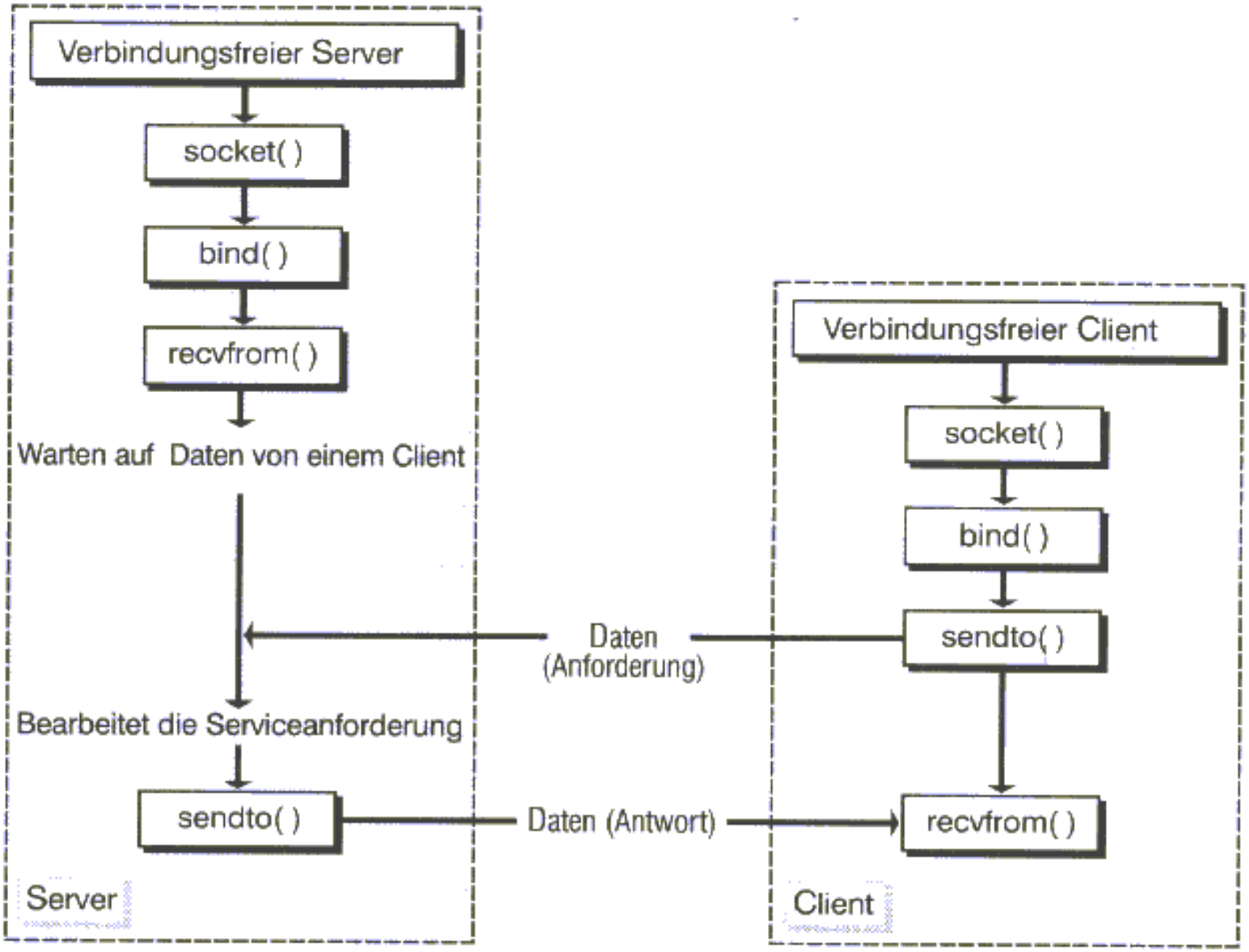
Folgende Grafik erläutert die Verwendung der Funktionen für ein verbindungsorientiertes Protokoll:



Das Socket Interface

Verwendung der Funktionen auf einem verbindungsfreien Protokoll

Folgende Grafik erläutert die Verwendung der Funktionen für ein verbindungsfreies Protokoll:



Programmbeispiele UDP

Der Server

Folgender Source Code in C ist ein Beispiel für einen UDP-Server in C. Der Server empfängt von einem Client eine Nachricht und sendet sie wieder zurück.

```
#include <stdio.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#define MAXMESSAGES 2048

main()
{
    struct sockaddr_in serv_addr, cli_addr; /*Adressstruktur fuer
Server und Client*/
    int sockfd; /*Socket Handle*/
    int length; /*Hilfsvariable fuer die Speicherung der
Laengen*/
    int udp_port; /*Variable fuer Portnummer*/
    char server_host_adresse[17]; /*Vektor zum Speichern der
IP-Adresse*/

    printf("\nBitte geben Sie den Port des Servers an:\n");
    scanf("%d",&udp_port);

    printf("\nDer Server wird gestartet!\n");


    /*UDP Socket definieren*/
    sockfd=socket(AF_INET, SOCK_DGRAM, 0); /*Definieren des
Socket*/
    if (!sockfd)
    { /*Definition schlug fehl*/
        printf ("Fehler beim definieren des UDP Sockets!\n");
        exit(1);
```

```
    }
    else
        printf ("UDP Socket erfolgreich definiert!\n");


    /*Konfigurieren des Socket*/
    serv_addr.sin_family = AF_INET; /*Serveradresstruktur TCP/IP
zuteilen*/
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY); /*Jede Adresse
akzeptieren*/
    serv_addr.sin_port = htons(udp_port); /*Serverport angeben*/
    bind(sockfd,(struct sockaddr*) &serv_addr,
sizeof(serv_addr));
    if(!sockfd)
    {
        printf("Fehler beim Binden des Ports!\n");
        exit(1);
    }
    else
        printf("Binden des Ports erfoldreich!\n");


    /*Den Zugewiesenen Port finden und anzeigen*/
    length=sizeof(serv_addr);
    if (getsockname(sockfd,(struct sockaddr*) &serv_addr,
&length) < 0)
    {
        perror("server; getting socket name\n");
        exit(1);
    }
    printf("Socket port: #%d\n", ntohs(serv_addr.sin_port));
    server_echo(sockfd, (struct sockaddr*) &cli_addr,
sizeof(cli_addr));

}



server_echo(int sockfd, struct sockaddr *pcli_addr, int maxclilen)
```

```
{
    int n, clilen;
    char mesg[MAXMESSAGES];
    for(;;)
    {
        clilen=maxclilen;
        n=recvfrom(sockfd, mesg, MAXMESSAGES, 0, pcli_addr,
    &clilen);
        if (!n) /*Keine Zeichen empfangen*/
        {
            perror("dg_echo: recvfrom error\n");
            exit(1);
        }
        mesg[n]=0;
        fputs(mesg,stdout); /*Ausgabe der Daten*/
        if (sendto(sockfd, mesg, n, 0, pcli_addr, clilen) != n)
        { /*Keine Zeichen versendet*/
            perror ("dg_echo: sento error\n");
            exit(1);
        }
    }
}
```

Programmbeispiele UDP

Der Client

Folgender Source Code in C ist ein Beispiel für einen UDP-Client in C.
Der Client sendet an einen Server eine eingegebene Message und wartet anschließend auf die Rückantwort vom Server.

```
#include <stdio.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>


#define MAXLINE 512
main()
{
    struct sockaddr_in serv_addr, cli_addr; /*Adresstruktur fuer
Server und Client*/
    int sockfd; /*Sockethandle*/
    int udp_port; /*Portnummer*/
    char server_host_adresse[17];

    printf("Bitte geben Sie die IP-Adresse des Servers in
Punktnotation an:\n");
    gets(server_host_adresse);

    printf("\nBitte geben Sie den Port des Servers an:\n");
    scanf("%d",&udp_port);

    printf("\nDer Client wird gestartet!\n");

    /*UDP Socket definieren*/
    sockfd=socket(AF_INET, SOCK_DGRAM, 0); /*0 bedeutet
beliebiges Protokoll*/
```

```
        if (!sockfd)
        {
            printf ("Fehler beim definieren des UDP Sockets!\n");
            exit(1);
        }
        else
            printf ("UDP Socket erfolgreich definiert!\n");


        /*Server-Informationen speichern*/
        serv_addr.sin_family = AF_INET;
        serv_addr.sin_addr.s_addr = inet_addr(server_host_adresse);
        serv_addr.sin_port = htons(udp_port);

        /*Lokale Clientinformationen speichern*/
        cli_addr.sin_family = AF_INET;
        cli_addr.sin_addr.s_addr = htonl(INADDR_ANY);
        cli_addr.sin_port = htons(0);

        bind(sockfd,(struct sockaddr*) &cli_addr, sizeof(cli_addr));
        if(!sockfd)
        {
            printf("Fehler beim Binden des Ports!\n");
            exit(1);
        }
        else
            printf("Binden des Ports erfoldreich!\n");

        echo_cli(stdin, sockfd, (struct sockaddr*) &serv_addr,
    sizeof(serv_addr));
        close (sockfd);
        exit(0);
    }
    echo_cli(FILE *fp,int sockfd,struct sockaddr *pserv_addr, int
servlen)
    {
        int n;
        char sendline[MAXLINE], recvline[MAXLINE+1];
```

```
    while(fgets(sendline, MAXLINE, fp) != NULL)
    {
        n=strlen(sendline);
        if(!(sendto(sockfd, sendline, n, 0, pserv_addr,
    servlen)))
        {
            perror("sendto: Fehler!\n");
            exit(1);
        }
        n=recvfrom ( sockfd, recvline, MAXLINE, 0, (struct
    sockaddr*) 0, (int*) 0);
        if (!n)
        {
            perror("recvfrom: Fehler!\n");
            exit(1);
        }
        recvline[n]='\0';
        fputs (recvline, stdout); /*Ausgabe der Daten*/

    }
}
```

Kommunikation mit TCP/IP unter UNIX

eine Einführung Ralf Mitteregger, © 8.5.97

Version 1.2, letzte Änderung am 18. Juni 1997

Inhalt

Zugriffe seit 13.05.97: