Netzwerkprogrammierung mit BSD-Sockets

Felix Opatz

Revision : 1.15

Vorwort

Liebe Leserinnen und Leser,

ich bin leider fürchterlich ungeübt was das Schreiben von Vorworten angeht. Das wird vielleicht nur noch durch das Verfassen von Reden übertroffen. Ich stelle mir unter einem Vorwort einen Text vor, der etwa $\frac{4}{5}$ einer Seite einnimmt, und dem Leser bzw. der Leserinn ein wenig darüber erzählt, wie es zu dem vorliegenden Buch kam, und was der Autor damit eigentlich will.

Ich habe im Sommer 2000 angefangen, eine Webseite zum Thema Socket-Programmierung zu pflegen, die „Socket-Tipps“. Dort findet man im wesentlichen eine Erklärung der Grundbefehle, Beispielprogramme und Hinweise auf die häufigsten Fallen, in die man als Anfänger so reintappen kann. Die Resonanz war gigantisch, zu manchen Zeiten habe ich pro Woche mehrere E-Mails mit Lob und Anregungen bekommen, aber auch viele Hinweise auf eine unklare Beschreibung und den Wunsch, doch etwas weiter auszuholen, den Leser mehr bei der Hand zu nehmen, und auch Grundlagen zu vermitteln. Genau diess ist das Ziel, das ich mit dem vorliegenden Buch anvisiert habe. Ich nenne es bewusst *Buch*, obwohl es mit rund 160 Seiten im Format DIN A4 sicher ein wenig seltsame Proportionen aufweist. Aber ein *Buch* ist für mich immer etwas, an dem man sich festhalten und orientieren kann. Man will etwas neues lernen, etwas von dem man vielleicht gar keine Ahnung hat. Man weiß nicht, was alles dazugehört, was wichtig ist, und bei „Computer-Dingen“: was in der Praxis funktioniert, *wie man es so macht*. Jetzt gibt es für so ein Vorhaben ein *Buch*. Ein in sich abgeschlossenes Dokument, das einen mit zartem Säuseln im Vorwort empfängt, im Inhaltsverzeichnis mit Erwartungen erfüllt, in den einzelnen Kapiteln (hoffentlich) mit Wissen belohnt, und am Ende freundlich zuwinkt und ein gutes Gelingen wünscht. Ist das nicht toll?

Meine Eltern und einige Bekannte halten mich für ein wenig gestört, daß ich etwa 9 Monate Zeit investiert habe um ein Buch zu schreiben, das ich *verschenke*. Zum einen hat es für mich selbst einen praktischen Wert, denn ich habe meine ersten Schritte mit dem Textsatzprogramm LaTeXgehen können, bevor es mit der Diplomarbeit so richtig losgeht. Zum anderen bin ich seit langer Zeit dem Open Source-Gedanken verfallen, und möchte eine weitere Lücke im Bereich anspruchsvoller, aber verständlicher Dokumentation schließen. Gute Bücher sollten für jedermann erschwinglich sein, doch leider muß man manchmal tief in die Tasche greifen, und gerade für Schüler und Studenten sitzt die Schmerzgrenze meist schon bei 50 €.

Ich gehe davon aus, daß dieses Buch kein statisches Objekt sein, sondern gewissermaßen leben wird. Deshalb habe ich das CVS-Tag `$Revision$` auf die Titelseite gesetzt, sodaß bei jeder Fassung sofort klar ist, wo sie einzuordnen ist. Die aktuelle Fassung wird voraussichtlich stets auf meiner Homepage zu finden sein. Selbstverständlich freue ich mich auch über Rückmeldungen; hat das Buch gefallen, sollte etwas verändert werden, ist etwas falsch? Schreibt mir E-Mails! Nur wenige Dinge im Leben sind so bequem. :-)

Nun wünsche ich ersteinmal viel Spaß beim Lesen und Ausprobieren!

Felix Opatz
`http://www.zotteljedi.de/`

Inhaltsverzeichnis

1. Einführung

Dieses Buch beschäftigt sich mit der Netzwerkprogrammierung unter Verwendung des BSD-Socket API. Es soll gleichermaßen als Lehrbuch, wie auch als Nachschlagewerk dienen. Andere Werke stürzen sich gleich in die Programmierung, was aus Sicht des ungeduldigen Lesers sicher angenehm ist, jedoch leicht zum Auslassen wichtiger Details führt.

Das vorliegende Buch vermittelt zuerst die unvermeidlichen Grundlagen, um danach eine Übersicht zu den einzelnen Socket-Befehlen zu verschaffen. Auf diesem Teil liegt der Schwerpunkt, gemeinsam mit den Socket-Optionen. Für den täglichen Gebrauch scheinen manche dieser „wissenswerten Details“ unnötig zu sein (was sie für einen Großteil der Anwendungen auch sind), jedoch gibt es Situationen, in denen ihre Kenntnis sich als wertvoller Vorteil entpuppen kann.

Die Kapitel zu Broadcast, Multicast, Raw Sockets und IPv6 weisen eine erhöhte Plattformabhängigkeit auf und sollten von Programmierern die besonders auf die Portabilität ihres Codes bedacht sind, mit Vorsicht genossen werden. Gerade im Bereich der Raw Sockets gibt es sehr viele betriebssystemspezifische Details, die sich nur durch die Lektüre der entsprechenden Fachdokumentation effizient nutzen lassen. Der besondere Fokus liegt wie immer im Bereich der freien UNIX-Derivate, sowie Windows, wobei Windows XP bezüglich der Raw Sockets einige Rückschläge einstecken musste.

Die im Text auftauchenden Fachbegriffe (zumeist an der *kursiven Schreibweise* bei ihrem ersten Auftauchen erkennbar) werden im Glossar ab Seite 151 erklärt, sofern ihre Bedeutung nicht aus dem sie umgebenden Text erschlossen werden kann.

Die empfohlene Lesereihenfolge ist von vorne bis hinten, ohne Auslassung einzelner Kapitel. Wer jedoch der Meinung ist, alles Wissenswerte über Netze und TCP/IP zu wissen, kann mit Kapitel 7 beginnen und gleich in den praktischen Teil der Programmierung eintauchen.

1.1. Referenzmodelle

Als ein Referenzmodell bezeichnet man im Allgemeinen ein Modellmuster, eine Abstraktion mit der man andere Modelle vergleichen oder davon ableiten kann. Im Folgenden sollen die zwei Referenzmodelle vorgestellt werden, mit denen die Schichtung einer „Netzwerkimplementierung“ beschrieben werden kann.

Aus Gründen der Austauschbarkeit und Interoperabilität wird in den meisten Bereichen der Softwareentwicklung auf die Mittel der Kapselung und Schichtenbildung zurückgegriffen. Bei einem Schichtenmodell ist es üblich, daß den einzelnen Schichten außer den Schnittstellen zu den anderen Schichten keine Details über deren Aufbau bekannt sein muß. Eine Schicht bietet Dienste an (das sind ihre exportierten Funktionen) und nimmt Dienste anderer Schichten in Anspruch.

In den beiden folgenden Kapiteln wird deutlich, daß dies mehr oder weniger streng eingehalten werden kann. Während das OSI-Modell feiner gegliedert ist, und damit auch eine höhere Abstraktion besitzt, bietet das TCP/IP-Modell durch die Bildung von *Abkürzungen* durch den Schichtenaufbau zumeist die größere Performanz und ermöglicht praxistaugliche Implementierungen.

1.2. Das OSI-Schichtenmodell

Das OSI-Modell – vollständige Bezeichnung: *Open Systems Interconnection Reference Model* – kann als Grundlage moderner Kommunikationstechniken gesehen werden. Die Entwicklung wurde Anfang der 1980er Jahre begonnen, 1984 wurde es von der ISO standardisiert.

Es besteht aus sieben Schichten, die jeweils nur mit den ihnen benachbarten Schichten kommunizieren. Beim Senden durchläuft die Information auf der Senderseite von oben nach unten sämtliche Schichten, während nach erfolgter Übertragung über die *Bitübertragungsschicht* auf der Empfängerseite der umgekehrte Weg durchlaufen wird, also von Schicht 1 bis Schicht 7. Der Hintergedanke der Schichtung ist die Zerlegung der Aufgabe „Übertragung von Daten“ in Teilaspekte, die gegen gleichartige Module austauschbare Implementierungen ermöglichen.

Anwendungsschicht	Schicht 7
Darstellungsschicht	Schicht 6
Sitzungsschicht	Schicht 5
Transportschicht	Schicht 4
Vermittlungsschicht	Schicht 3
Sicherungsschicht	Schicht 2
Bitübertragungsschicht	Schicht 1

Abbildung 1.1.: Das OSI-Schichtenmodell

Die *logische* Kommunikation erfolgt horizontal, also „spricht“ beispielsweise die *Vermittlungsschicht* der Senderseite mit der *Vermittlungsschicht* der Empfängerseite. Die dabei involvierten Schichten darunter dienen lediglich der Übertragung, sie wissen nichts vom Inhalt den sie übertragen.

Die Schichten 7 bis 4 stellen eine Ende-zu-Ende-Verbindung dar, das bedeutet, daß es fest definierte Endpunkte gibt und von diesen mehrere auf einem Host gelegen sein können. Ein Endpunkt ist immer ein Prozeß, wobei sich sowohl mehrere Prozesse einen Endpunkt teilen können, als auch ein Prozeß mehrere Endpunkte haben kann. Ohne Prozesse aber keine Endpunkte.

In den Schichten 3 bis 1 geht es um die Kommunikation zwischen Netzwerkteilnehmern, also zwischen Computern oder anderen Geräten. An dieser Stelle geht es nur darum, daß die Daten den richtigen Weg finden und an dem Endgerät herauskommen, auf dem der Endpunkt der Verbindung zu finden sein wird.

Eine andere Teilung wäre in einen abstrakten Teil (7 bis 5) und einen grundlegenden Teil (4 bis 1), wobei dieser für den eigentlichen Datentransport zuständig ist, während ersterer sich mit der Anwendung beschäftigt.

Die sieben Schichten im Einzelnen sind:

Anwendungsschicht, application layer

Die Anwendungsschicht ist die oberste der Schichten; sie bietet den Anwendungen eine Reihe von Funktionalitäten wie Datenübertragung oder E-Mail an. Hier ist Abstraktion am höchsten, denn um

den logischen Vorgang „Übertragung einer Datei“ auszulösen muß die Anwendung (beispielsweise eine graphische Shell mit Drag&Drop-Funktionalität) nichts von Netzwerken wissen; dies kann alles transparent vom Betriebssystem oder einer Bibliothek erledigt werden. Ein Beispiel mag die Qt-Bibliothek[20] sein, deren Widgets teilweise komplette Transparenz für Dateizugriffe bieten.

Darstellungsschicht, presentation layer

Die Darstellungsschicht ist für die Konversion systemabhängiger Daten (wie zum Beispiel Zeichensätze, oder *Byteorder*) in eine systemunabhängige Form zuständig. Kryptographische Aufgaben gehören auch in diese Schicht, ebenso wie beispielsweise eine für den Anwender transparente Kompression der Daten.

Sitzungsschicht, session layer

Aufgabe der Sitzungsschicht ist die Aufrechterhaltung der *Sitzung*. Zu diesem Zweck werden sogenannte Fixpunkte (engl. *Check Points*) eingeführt, mit deren Hilfe nach einem Ausfall einer Transportverbindung die Resynchronisation möglich ist, ohne daß hierzu die Übertragung erneut gestartet werden muß.

Transportschicht, transport layer

Als unterste Schicht der Host-zu-Host-Verbindung bietet die Transportschicht den anwendungsorientierten Schichten 5-7 eine Abstraktion des Kommunikationsnetzes. Hierzu zählt die Segmentierung von Datenpaketen und die Stauvermeidung (engl. *congestion control*).

Vermittlungsschicht, network layer

Bei verbindungorientierten Diensten wird in der Vermittlungsschicht die Verbindung geschaltet; bei paketorientierten Diensten sorgt sie für die Weitervermittlung der Pakete. In beiden Fällen geht die Datenübertragung über das gesamte Kommunikationsnetz hinweg und schließt dabei die Wegsuche (*Routing*) zwischen den Netzknoten mit ein, die hierzu nötigen Routingtabellen werden ebenfalls von der Vermittlungsschicht verwaltet. Die höheren Schichten sehen Pakete, die an andere Hosts weitervermittelt werden müssen, zu keiner Zeit. Neben der *Flußkontrolle* gehören auch die Netzwerkadressen zu dieser Schicht, sowie die Umsetzung verschiedener Netzwerktechnologien ineinander, um eine Kommunikation auch über Netzgrenzen hinweg zu ermöglichen.

Sicherungsschicht, data link layer

Die Bereitstellung einer sicheren (im Sinne von fehlerfreien) Übertragung ist die Aufgabe der Sicherungsschicht. Zu den Teilaufgaben zählt die Zerlegung des Datenstroms in Blöcke, sowie das Hinzufügen von Prüfsummen und weiteren Header-Informationen, die das jeweils zugrundeliegende Netzwerk vorsieht. Mit diesem Aspekt setzt sich Kapitel 2 intensiv auseinander.

Bitübertragungsschicht, physical layer

Die unterste Schicht dient der physikalischen Übertragung der Informationen. Dies kann beispielsweise mit elektrischen Signalen erfolgen, aber auch optische (Lichtleiter) oder elektromagnetische (drahtlose Netzwerke) Übertragungen sind üblich. Unüblichere Verfahren stellen die akkustische Übertragung oder der Einsatz von Brieftauben dar (siehe Kapitel 2.7).

So schön dieses Modell auch erscheint, so selten wurde es wirklich umgesetzt, eigentlich ausschließlich in ISO-Protokollen. Wie im nächsten Abschnitt zu sehen ist, kommt man in der Praxis mit weniger Abstraktion aus.

Weitere Informationen und Verweise finden sich in [6].

1.3. Das TCP/IP-Referenzmodell

Das TCP/IP-Referenzmodell ist ein Schichtenmodell, das auf die Bedürfnisse der *TCP/IP-Protokollfamilie* zugeschnitten ist. Gegenüber dem OSI-Modell sind hier nur vier Schichten vorgesehen, die noch dazu teilweise übersprungen werden können. Es wurde bereits vor dem OSI-Modell entwickelt (das RFC zu TCP stammt von 1981 und ist nicht das erste mal, daß jemand darüber nachgedacht hat), aber kommt auf ähnliche Ergebnisse.

Die Internet-Protokolle setzen keine Eigenschaften der Netzwerkschicht voraus und können deshalb unabhängig von der Übertragungstechnik eingesetzt werden. Die unterste Ebene kennt nur die Problematik der Punkt-zu-Punkt-Verbindung, *daß* also Daten von einem anderen Host ankommen, *wie* sie dorthin gelangen ist Sache der Netzwerkschicht.

Es heißt immer, daß das amerikanische Militär der Auftraggeber war, und das Ziel eine atombombensichere Infrastruktur zu erschaffen war. Fakt ist, daß es ein militärischer Forschungsauftrag an die DARPA (Defense Advanced Research Projects Agency) war, und als Ziel eine Netzwerkinfrastruktur, die zuverlässig (auch nach dem Ausfall einzelner Knoten) alles mögliche miteinander vernetzen kann, anvisiert wurde.

Um dies zu ermöglichen, mußte das gesamte System zum einen so universell sein, daß es mit jeder Art von Netzwerkkomponente verwendet werden kann (d.h. daß an die Netzwerkschicht keine detaillierten Anforderungen gestellt werden), und zum anderen muß ein Paket *irgendwie* seinen Weg zum Ziel finden können. Das Irgendwie wird von der Internetschicht besorgt, anhand der Zieladresse kann jeder Knotenpunkt entscheiden, welchen Weg das Paket einschlagen muß (oder sollte), und viel wichtiger: wenn ein Weg nicht passierbar ist, so kann jeder Knotenpunkt eine eventuell vorhandene Alternative versuchen. Das ganze klappt natürlich nur, wenn es redundante Verbindungen gibt, also mehrere Wege zum Ziel führen.

In Abbildung 1.2 ist der Aufbau des TCP/IP-Schichtenmodells veranschaulicht, zusammen mit den Entsprechungen im OSI-Schichtenmodell. Die Schichten im Einzelnen sind:

Anwendungsschicht

Die Anwendungsschicht stellt die Kommunikationsschnittstelle zum Benutzer bzw. der von ihm eingesetzten Anwendung dar (siehe auch Kapitel 5). Beispiele hierfür sind HTTP oder FTP, als Protokolle zur Übertragung von Dateien, oder das Simple Mail Transfer Protocol (SMTP) für das Versenden von E-Mail. Die Entsprechung im OSI-Modell sind die Schichten 5 bis 7. Diese Schicht wird üblicherweise durch Anwendungssoftware implementiert, während die unteren Schichten vom Betriebssystem bereitgestellt werden.

Schicht	im OSI-Modell	Beispiel
Anwendungsschicht	5-7	HTTP, FTP
Transportschicht	4	TCP, UDP
Internetschicht	3	IP
Netzwerkschicht	1-2	Ethernet, FDDI

Abbildung 1.2.: Das TCP/IP-Schichtenmodell

Transportschicht

In der Transportschicht wird die Ende-zu-Ende-Verbindung hergestellt und betrieben. Hier kommt TCP zum Einsatz, das in Kapitel 4.1.1 ausführlich behandelt wird. TCP bietet hierbei eine *sichere* Übertragung in dem Sinne an, daß verlorengegangene Daten erneut angefordert werden. Ein anderes Protokoll dieser Schicht ist UDP. Beide implementieren das Konzept des *Endpunktes*, eine Kombination aus IP-Adresse und Portnummer. Dies entspricht der 4. Schicht des OSI-Modells.

Internetschicht

Die Internetschicht ist für die Weitervermittlung von Paketen und die Wahl ihres Weges (Routing) zuständig. Ab dieser Schicht nach unten hin gibt es nur noch das Konzept der Punkt-zu-Punkt-Übertragung. Aufgabe der Internetschicht ist es, die Pakete zu ihrem nächsten Zielpunkt zu bringen und diesen nötigenfalls zu ermitteln. Router implementieren beispielsweise oftmals den Protokollstapel nur bis zu dieser Schicht. Das eingesetzte Protokoll ist IP (siehe Kapitel 3.5.1 zur ausführlichen Erklärung), das einen *unzuverlässigen* Paketauslieferungsdienst bereitstellt (verlorengegangene Pakete sind eben weg). Im OSI-Modell entspricht die Internetschicht der Vermittlungsschicht.

Netzwerkschicht

Obgleich die Netzwerkschicht im Modell spezifiziert ist, enthält sie keinerlei Protokolle der TCP/IP-Familie. Hier werden vielmehr all jene Übertragungstechniken einsetzbar, die im Kapitel 2 besprochen werden. Beispiele sind das weitverbreitete Ethernet (2.1), FDDI (ein System mit optischer Übertragung über Lichtwellenleiter) oder Wireless LAN (2.4). Das OSI-Modell sieht hierfür die Schichten 1 und 2 vor.

Eine wirklich saubere Trennung ist beim TCP/IP-Modell nicht möglich. Beispielsweise dient das ICMP (Internet Control Message Protocol, siehe Kapitel 4.3.1) zur Mitteilung von Netzwerproblemen, nutzt seinerseits jedoch das Internet Protocol IP, weshalb es nach diesem Modell in der Transportschicht anzusiedeln wäre, auch wenn es in dem Sinne gar keine (Nutzer-)Daten transportiert.

Ein Protokoll, das in Kapitel 3.4.1 vorgestellt wird, aber ebenfalls nicht eindeutig zugeordnet werden kann, ist das Address Resolution Protocol, ARP. Zum einen gehört es der Netzwerkschicht an, zum anderen wird es üblicherweise zur TCP/IP-Protokollfamilie hinzugezählt. Jedoch ist es nicht für jede Übertragungstechnik notwendig, und kann beispielsweise bei SLIP (siehe 2.5) vollends entfallen.

2. Die Netzwerkschicht

In diesem Kapitel soll eine kleine Übersicht zu den gebräuchlichsten und bekanntesten Netzwerktechnologien gegeben werden. Dabei wird das Ethernet detailliert beschrieben – die meisten Leser werden wohl mit diesem am ehesten in Kontakt kommen.

Die hier vorgestellten Netzwerktechnologien beschreiben jeweils zwei Bereiche. Das ist einmal die Art der Verkabelung, die Signalformen und die Topologie der Netze, im OSI-Modell als Bitübertragungsschicht einzuordnen, zum anderen die Organisation der Daten zu Übertragungseinheiten, genannt Frames. Dies ist der Sicherungsschicht zuzuordnen. Bei manchen Technologien wird die Bitübertragungsschicht nicht spezifiziert, weil sie bereits durch die Art der Übertragung vorausgesetzt wird. Damit sind beisipelsweise SLIP und PPP gemeint, die über eine serielle Leitung „gesprochen“ werden und sich selbst nur mit der Organisation der Daten zu Frames beschäftigen, während die Fähigkeit einzelne Bytes von einem zum anderen Ort zu schaffen schon durch die Spezifikation zur seriellen Verbindung (beispielsweise RS-232) abgedeckt wird.

Die etwas schwer einzuordnende Rolle des Address Resolution Protocols (ARP) ist im nächsten Kapitel über die Internetschicht nachzulesen.

2.1. Ethernet

Ethernet ist in den 1970er Jahren bei Xerox entwickelt worden, und fand in einer überarbeiteten Version in den 1980er Jahren als Ethernet II große Verbreitung. Später wurde es durch IEEE 802.3 standardisiert, musste jedoch danach durch ein als SNAP (RFC 1042[24]) bekanntes Verfahren erweitert werden, um wieder zu TCP/IP kompatibel zu werden.

Die meisten Informationen in diesem Kapitel habe ich aus [23], einer sehr schönen Erklärung, die auch geschichtliche Informationen bietet und auf viele weitere – teils eher historische – Ethernet-Varianten eingeht.

Im Folgenden wird Ethernet II beschrieben, weil es nach wie vor die größte Verbreitung hat und von jedem ernstzunehmenden Betriebssystem implementiert wird. In Abbildung 2.1 ist der Aufbau eines Ethernet II Frames (auch als *DIX-Frame* bezeichnet, die Anfangsbuchstaben von DEC, Intel und Xerox, die sich auf dieses Format geeinigt haben) veranschaulicht.

Preamble	Destination MAC	Source MAC	Type	Data	CRC

Abbildung 2.1.: Ethernet II Frame

Preamble (7 Oktetts)
Einleitung mit alternierender Bitfolge, dient der Synchronisation

Destination MAC (6 Oktetts)
Ethernet-Adresse des Ziels

Source MAC (6 Oktetts)
Ethernet-Adresse der Quelle

Type (2 Oktetts)
Typ des Frames, gültige Werte sind

0x0800	IPv4
0x0806	ARP
0x8035	RARP
0x809b	Appletalk
0x8137	Novell
0x8138	Novell
0x86dd	IPv6

Eine komplette Liste erhält man unter [14].

Data
Die Nutzdaten des Frames

CRC (4 Oktetts)
Checksumme zur Sicherstellung der Datenintegrität

Beim ursprünglichen Ethernet wurde im Type-Feld die Länge des Frames angegeben, die auf maximal 1500 Oktetts begrenzt ist. Bei Ethernet II nutzte man deshalb Werte über 1500, nämlich ab 0x0800 (2048 dezimal), zur Bezeichnung des Frameinhalts (*Ethertype*). Eine Unterscheidung, ob Ethernet I oder Ethernet II vorliegt ist daher sehr einfach.

Der Ethernet-Standard schreibt weiterhin die Art der Verkabelung vor, man kennt hier eine Reihe verschiedener Varianten. Die erste Zahl beschreibt jeweils die Geschwindigkeit in MBit/s (10, 100, usw.), die zweite den Kabeltyp (T steht hierbei für Twisted-Pair, es gibt auch F für Fibre, also Glasfaser, aber auch Buchstaben, die nicht als Abkürzung dienen). Die wichtigsten sind:

- 10base2 über Koaxialkabel („Thin Wire Ethernet“, auch „Cheapernet“) mit einem Wellenwiderstand von 50 Ω. Die Teilnehmer werden hierbei über T-Stücke mit dem Kabel verbunden, sodaß ein Bus entsteht, an dessen Ende Terminatoren sitzen. Ein Segment darf hierbei bis zu 185 m lang sein. Die Datenrate beträgt 10 Mbit/s.

- 10base5 über Koaxialkabel („Thick Wire Ethernet“, auch „Yellow Cable“), ebenfalls 50 Ω Wellenwiderstand, jedoch werden die Teilnehmer durch sog. *Vampirklemmen* angeschlossen, Stifte die in die Kabel gebohrt werden, um den Wellenleiter anzuzapfen. Die Datenrate beträgt ebenfalls 10 Mbit/s, jeodch können die Segmente bis zu 500 m lang sein.

- 10base-T über Twisted Pair-Kabel der Kategorie 3 oder 5. Es werden zwei Adernpaare aus je zwei verdrillten Adern eingesetzt, die Geräte untereinander sind über Hubs oder Switches verbunden. Der Abstand zum Hub bzw. Switch darf maximal 100 m betragen, Datenrate ist ebenfalls 10 Mbit/s.

- 100base-TX funktioniert wie 10base-T, jedoch ist zwingend Kabel der Kategorie 5 (Cat-5) zu verwenden. Die Datenrate beträgt 100 Mbit/s. Dies ist die heutzutage übliche Ethernet-Variante.

- 1000base-T, „Gigabit-Ethernet“, verwendet alle vier Adernpaare eines Cat-5 Kabels und besitzt eine Datenrate von 1000 Mbit/s. Hierbei ist sogar Cat-5e notwendig, welches eine genauere Spezifikation ist; ordentlich verlegte Cat-5-Verbindungen genügen oftmals auch Cat-5e.

Weiterhin ist die Medienzugriffskontrolle (Media Access Control, MAC) erwähnenswert. Das bei Ethernet eingesetzte Verfahren wird CSMA/CD genannt, Carrier Sense Multiple Access / Collision Detection. Das Schema funktioniert wie folgt:

1. Das Medium überwachen, ob es frei ist

2. Ist das Medium frei: Übertragung beginnen; andernfalls: Schritt 5

3. Daten übertragen, dabei gleichzeitig das Medium abhören. Tritt eine Kollision auf: definiertes Störsignal senden, weiter bei Schritt 5

4. Daten erfolgreich übertragen, Erfolgsmeldung an höhere Netzwerkschichten absetzen

5. Leitung ist belegt, warten bis die Leitung frei ist

6. Leitung ist gerade frei geworden, zufällige Zeit warten, dann Schritt 1

Wenn die maximale Anzahl Übertragungsversuche überschritten ist, so wird die Übertragung abgebrochen und den höheren Netzwerkschichten eine entsprechende Meldung übergeben.

Von besonderer Bedeutung ist die zufällige Zeit, die gewartet wird. Diese wird mit jedem fehlgeschlagenen Versuch verlängert, in der Hoffnung die Zugriffe weiter zu streuen und damit keine weiteren gleichzeitigen Übertragungen mit resultierender Kollision zu erzeugen. Das Verfahren ist in [25] sehr schön beschrieben.

Durch diese indeterministische Abfolge beim Senden ist Ethernet nicht realzeitfähig, d.h. es gibt keine garantierte (maximale) Reaktionszeit, es kann sofort klappen, oder auch niemals. Dies ist anders bei den Verfahren, die mit Tokens arbeiten. Dafür müssen die Stationen nur dann tätig werden, wenn sie selbst Daten versenden wollen. Ethernet wird auch als *passives* Verfahren bezeichnet.

2.2. Token Ring

Bei Token Ring, standardisiert in IEEE 802.5, handelt es sich um eine von IBM weiterentwickelte (und daher auch dort häufig anzutreffende) Netzwerktechnologie, die eine *kollisionsfreie* Übertragung ermöglicht.

Die Topologie des Aufbaus entspricht einem Ring, auf dem ein sogenanntes *Token* weitergereicht wird. Dies ist ein spezieller Frame, der von jeder Empfangsstelle gelesen und weitergegeben wird. Dabei sind alle Stationen ständig aktiv, auch wenn sie nicht selbst senden wollen, sondern nur Daten weitergeben (deshalb wird Token Ring auch als *aktives* Verfahren bezeichnet, vgl. Ethernet).

In neueren Formen ist auch eine Sterntopologie anzutreffen, wobei Hubs die Beförderung der Tokens an die Endstationen überwachen. Der Vorteil liegt darin, daß die nichtbeteiligten Stationen nicht mit der Weitergabe nicht an sie adressierter Frames belästigt werden. Außerdem kommen auch mehrere Tokens zum Einsatz.

Wenn eine Station nun Daten versenden möchte, so wartet sie, bis das Token sie erreicht hat. Dann werden die Nutzdaten dem Token angehängt, und das Token als *besetzt* markiert, aus dem Frei-Token wird ein Datenrahmen (*frame*). Dieser Datenrahmen wird wiederum weitergereicht, wobei jede Station die Adresse mit der eigenen vergleicht und den Frame weitergibt, wenn er nicht für sie bestimmt ist. Wenn der Frame empfangen wurde, quittiert die Station den Empfang durch das Setzen eines Bits und gibt den Frame weiter bis zum ursprünglichen Sender. Dieser nimmt die Quittung zur Kenntnis und erzeugt ein neues Frei-Token.

Durch diese kollisionsfreie Kommunikation erreicht ein Token Ring-Netzwerk trotz der niedrigeren Geschwindigkeit von 4 bzw. 16 Mbit/s ähnliche Übertragungsraten wie ein 10 oder 100 Mbit/s schnelles Ethernet.

2.3. ARCNet

Das ARCNet (**A**ttached **R**esources **C**omputer **Net**work) wurde 1976 von der Firma Datapoint erfunden.

Die Topologie ist eine Stern- bzw. Baumform. Ursprünglich wurde ARCNet mit Koaxial-Kabeln aufgebaut, jedoch sind im Laufe der Entwicklung auch UTP (Unshielded Twisted Pair) und Glasfaser spezifiziert worden.

Das Zugriffsverfahren ist Token Bus. Ähnlich wie bei Token Ring wird ein Token benutzt, das jedoch in einer festgelegten Reihenfolge weitergegeben wird. Dieses ist – anders als bei Token Ring – nicht durch die Verkabelung festgelegt, sondern durch eine Durchnummerierung (oder „fortlaufende Adressierung“) der Netzwerkkomponenten.

Obwohl mit 2,5 Mbit/s bei ARCNet und 20 Mbit/s bei ARCNet-Plus die Übertragungsrate geringer als beispielsweise bei Ethernet liegt, können dennoch gute Geschwindigkeiten erreicht werden, weil durch das Token-Verfahren keine Kollisionen auftreten.

2.4. Wireless LAN

Wireless LAN, auch als WLAN oder WiFi (Wireless Fidelity) bezeichnet, ist in IEEE 802.11 standardisiert.

Als Übertragungsmedium sind hierfür zunächst Infrarotlicht sowie Radiowellen im lizenzfreien ISM-Band bei 2,4 GHz spezifiziert, die Übertragungsrate beträgt hierbei 2 Mbit/s. Die Kommunikation zwischen zwei Teilnehmern kann entweder im *Ad-hoc-Modus* erfolgen, oder sich im *Infrastruktur-Modus* auf Basisstationen (sogenannte *Access-Points*) stützen. Später kam in IEEE 802.11a eine weitere Variante hinzu, die das 5 GHz-Band einsetzt und damit auf Datenraten bis zu 54 Mbit/s kommt, sowie IEEE 802.11b das im 2,4 GHz-Band bis zu 11 Mbit/s ermöglicht.

Der Medienzugriff erfolgt mittels CSMA/CA (Carrier Sense Multiple Access / Collision Avoidance). Probleme ergeben sich, wenn zwei Endgeräte mit einem Kommunikationspartner in der Mitte kommunizieren, jedoch selbst voneinander zu weit entfernt sind, um sich zu sehen. In diesem Fall stören sich die Übertragungen gegenseitig, ohne daß die Verursacher es bemerken können. Um dieses Problem zu beheben gibt es eine Erweiterung „Request to send/Clear to send“ (RTS/CTS), bei der die Stationen sich durch Datenpakete absprechen (ähnlich dem RTS/CTS Verfahren bei seriellen Schnittstellen).

2.5. SLIP

SLIP steht für **S**erial **L**ine **IP**, die Übertragung von IP-Paketen über eine serielle Leitung. Es ist in RFC 1055[22] verbindlich beschrieben.

Das Protokoll kennt keine eigenen Spezifikationen von Kabeln oder Signalen, sondern setzt direkt auf eine serielle Übertragung auf, die beispielsweise via RS-232 erfolgen kann. SLIP definiert hierzu zwei spezielle Zeichen, END (oktal 300) und ESC (oktal 333).

Wenn Daten gesendet werden sollen, werden sie einfach gesendet. Falls ein Zeichen den gleichen Code wie das END-Zeichen hat, so wird stattdessen ein ESC-Zeichen sowie ein Zeichen mit dem oktalen Wert 334 gesendet. Tritt im normalen Datenstrom ein ESC-Zeichen auf, so wird die Kombination ESC, gefolgt von oktal 335 übertragen.

Dadurch, daß SLIP von zwei miteinander verbundenen Endgeräten ausgeht, ist weder eine Adressierung noch eine Kontrolle des Medienzugriffs notwendig. SLIP findet überwiegend bei der Einwahl über Modems Anwendung, ist aber auf Grund der Einfachheit auch für viele andere Zwecke naheliegend.

2.6. PPP

PPP steht für **P**oint-to-**P**oint-**P**rotocol und wird zumeist zum Aufbau für Wählleitungen verwendet. Es ist in RFC 1661[21] spezifiziert.

Das PPP spezifiziert zwei weitere Protokolle, einmal das LCP (Link Control Protocol) mit dem Verbindungsparameter ausgehandelt werden, und zum anderen eien Reihe von NCPs (Network Control Protocols) die je nach verwendetem darüberliegenden Protokoll Dinge wie die Zuweisung von IP-Adressen kontrollieren.

PPP geht nur davon aus, daß die Verbindung Daten bidirektional versenden kann und diese in der Reihenfolge ankommen, wie sie gesendet wurden. Die Header und Felder des LCPs sind deutlich komplexer als bei SLIP, jedoch dadurch auch viel flexibler, weshalb PPP heutzutage für Wählverbindungen viel verbreiteter ist. In Form von PPPoE (PPP-over-Ethernet) kommt es beispielsweise bei ADSL-Verbindungen zum Einsatz.

2.7. Weitere Technologien

IP over Avian Carriers

Bei IP over Avian Carriers handelt es sich um ein „1. April“-RFC von 1990, RFC 1149 [26].

Es beschreibt die Übertragung von TCP/IP-Paketen über Brieftauben als Netzwerkschicht. Auf Grund des Schichtenaufbaus ist eine Übertragung ja mit „nahezu allem“ möglich, weshalb auch dies *funktioniert*: die Bergen Linux User Group hat diesen Standard erfolgreich umgesetzt.

Weitere Informationen gibt es unter `http://www.blug.linux.no/rfc1149/`.

IP over Bongo Drums

Eine ähnliche Demonstration der Leistungsfähigkeit des Schichtenaufbaus wurde an der Algoma University erbracht. Hierbei war Schall das Übertragungsmedium, und die Signale sollten ursprünglich mit Bongo-Trommeln erzeugt werden, welche aus Konstruktionsgründen jedoch durch Lautsprecher ersetzt wurden.

Informationen hierzu findet man unter `http://eagle.auc.ca/~dreid/`.

FDDI

Um noch eine ernstgemeinte Übertragungstechnik vorzustellen: FDDI (Fiber Distributed Data Interface) bezeichnet eine Ende der 1980er Jahre entwickelte Netzwerkarchitektur, die unter Verwendung optischer Signalübertragung auf 100 Mbit/s ausgelegt ist.

Die Topologie ist eine Ringstruktur, die Zugriffskontrolle erfolgt über einen Token. 1994 wurde der FDDI-Standard über Kupferleitungen standardisiert, bekannt als CDDI (C steht hierbei für *Copper*, Kupfer).

3. Die Internetschicht

Das Internet ist eigentlich nur *ein besonders großes* Internet. Unter einem Internet versteht man den Zusammenschluß verschiedener Netze. Die im Kapitel Netzwerktechnologien beschriebenen Arten von lokalen Netzen sind zueinander inkompatibel. Das fängt bei der Art der Verkabelung, den Spannungspegeln und Signalformen an, aber zieht sich durch bis zu der Form der Frames und der Zugriffskontrolle auf das Medium.

Das Designziel von TCP/IP war, wie bereits erwähnt, alles mögliche, das es so an Netzen gibt, zu verbinden. Auf Grund der Inkompatibilität der Netze untereinander musste also eine Abstraktion her, die diese Netze nur zum Transport nutzt. In der Tat ist es einem IP-Datagramm recht egal, auf welcher Art von Netzwerkschicht es sitzt. Wie in Abschnitt 1.3 angesprochen ist die IP-Schicht für die Übertragung von Host zu Host zuständig, wälzt den Vorgang der Übermittlung jedoch auf die Netzwerkschicht ab.

Um eine Übertragung über Netzgrenzen hinweg zu ermöglichen bedarf es zweier Features, die das Internet Procotol bietet:

- Ein logisches, netzübergreifendes Adressierungs-Schema
- Eine Möglichkeit für die Datagramme, ihren Weg zu finden

Jedem dieser beiden Punkte wird im Folgenden ein kurzer Abschnitt gewidmet. Anschließend wird das Address Resolution Protocol vorgestellt, das die Schnittstelle zwischen dem Internet Protocol und einer konkreten Netzwerktechnologie herstellt, in unserer Betrachtung soll dies Ethernet sein. Darauf folgend gibt es eine ausführliche Beschreibung des Internet Protocols.

3.1. Adressierung in Internets

Um die Wanderung von einem Punkt zu einem anderen zu ermöglichen muß es möglich sein, den Weg dorthin zu bestimmen. Um dies zu ermöglichen, sind Internet-Adressen nicht willkürlich vergeben, sondern ihr Aufbau folgt einer gewissen Logik. So besteht jede Adresse aus einem Netzanteil und einem Hostanteil.

Ursprünglich wurden Netze in drei Klassen, A, B und C unterteilt. In Klasse-A-Netzen sind 8 Bit für die Netzadresse reserviert, in Klasse-B-Netzen 16 und in Klasse-C-Netzen 24 Bit. Es hat sich jedoch gezeigt, daß diese Adressierung zu unflexibel war. Ein Klasse-A-Netz mit 16 Millionen möglichen Hostadressen ist einfach zu groß, um es sinnvoll komplett zu belegen.

Um eine IP-Adresse in den Netzteil und Hostteil zu zerlegen verwendet man die sogenannte *Netzmaske*. Doch zuerst einmal zur Darstellung der IP-Adressen. Im Internet Protocol Version 4, wie wir es zur Zeit nutzen, sind 32 Bit für eine Adresse vorgesehen. Daraus ergeben sich rund 4 Milliarden Adressen (um genau zu sein: $2^{32} = 4294967296$), das erschien aus damaliger Sicht für „alle Zeiten“ ausreichend. Wie in Kapitel 3.3 zu sehen sein wird, erwies sich dies als Trugschluß. Die nächste Generation des Internet Procotols, Version 6, räumt diesen Mißstand aus, und sieht 128 Bit (also 2^{128} = eine 3 mit 38 Nullen und etwas Zerquetschtes) Bit vor. Die Anzahl Adressen ist erstmal so unverstellbar groß,

.... Die Erde ist groß, oder? Jetzt stellen Sie sich mal einen Quadratmillimeter vor, so ein winziges Stückchen. Mit dem Adreßraum von IPv6 ließe sich jedem dieser winzigen Stückchen nicht eine, nicht zehn oder hundert, sondern mehr als 600 Billiarden Adressen verpassen. Aus jetziger Sicht also erstmal genug...

Doch nun zurück zum Aufbau der Adressen. In IPv4 gibt es also 32 Bit lange Adressen, die üblicherweise in der „dotted decimal“ Form geschrieben werden, jeweils zu 8 Bit gruppiert und als Dezimalzahlen ausgedrückt, die durch Punkte getrennt sind.

Die Adresse 192.168.1.1 beispielsweise sähe binär folgendermaßen aus:

192	168	1	1
11000000	10101000	00000001	00000001

Nach der alten Klasseneinteilung wäre dies eine Klasse-C-Adresse. Um die Klasse zu bestimmen werden die höchstwertigen Bits herangezogen und nach folgender Tabelle zugeordnet:

High Order Bits	Class	Range	Bits/Net	Bits/Host
0	A	0.0.0.0 - 127.255.255.255	7	24
10	B	128.0.0.0 - 191.255.255.255	14	16
110	C	192.0.0.0 - 223.255.255.255	21	8
1110	D	224.0.0.0 - 239.255.255.255		
11110	E	240.0.0.0 - 255.255.255.255		

Die Klassen D und E kamen später hinzu. Klasse D beinhaltet alle Multicast-Adressen, Klasse E ist für zukünftige Verwendungen reserviert.

Die übliche Netzmaske für ein Klasse-C-Netz ist 255.255.255.0. Das heißt: alle Bits, die den Hostanteil ausmachen, werden auf 0 gesetzt, alle Bits des Netzanteils auf 1. Durch einfache UND Verknüpfung aus IP-Adresse und Netzmaske erhält man den Netzanteil, durch das Verknüpfen mit der invertierten Netzmaske (auch *Hostmaske* genannt) erhält man den Hostanteil. Die folgende Tabelle veranschaulicht dies:

IP-Adresse	11000000	10101000	00000001	00000001
Netzmaske	11111111	11111111	11111111	00000000
Verknüpfung	11000000	10101000	00000001	00000000

IP-Adresse	11000000	10101000	00000001	00000001
Hostmaske	00000000	00000000	00000000	11111111
Verknüpfung	00000000	00000000	00000000	00000001

Die Netzadresse ist also 192.168.1.0, die Hostadresse 0.0.0.1.

Jetzt wird auch klar, warum ein Klasse-C-Netz nur 256 Hosts enthalten kann. Aber aus den genannten Gründen der Inflexibilität ist man zu den klassenlosen Adressen übergegangen. Hierbei wird der Adresse die „Breite“ der Netzmaske mitgegeben. So wäre z.B. eine Adresse wie 192.168.14.189/27 so zu verstehen, daß die IP-Adresse 192.168.14.189 (also nach „alter“ Regel eine Klasse-C-Adresse) lautet, und die Netzmaske aus 27 Einsen gefolgt von 32 - 27 = 5 Nullen besteht.

IP-Adresse	11000000	10101000	00001110	10111101
Netzmaske	11111111	11111111	11111111	11100000
Verknüpfung	11000000	10101000	00001110	10100000

IP-Adresse	11000000	10101000	00001110	10111101
Hostmaske	00000000	00000000	00000000	11100000
Verknüpfung	00000000	00000000	00000000	00011101

Der Netzanteil ist also 192.168.14.160, der Hostanteil 0.0.0.29. In ein solches Netz passen $2^5 = 32$ Hosts.

Nun gibt es noch pro Netz zwei besondere Adressen, die nicht für Hosts verwendet werden dürfen. Dies ist einmal die Netzadresse, hier sind alle Bits des Hostanteils Null, und die Broadcast-Adresse, bei der alle Bits des Hostanteils auf Eins stehen.

Die Broadcast-Adresse bezeichnet jeden Host im Netzwerk. Datagramme, die an diese spezielle Adresse gesendet werden, sollen also von jedem Host in diesem Netz „gesehen“ werden. Die Broadcast-Adresse für dieses Netz wäre also mit dem Hostteil 0.0.0.31 versehen, macht als ganze Adresse 192.168.14.191.

Während Broadcast-Adressen und Netzadressen bei der klassischen Klasseneinteilung sehr einfach zu erkennen waren (immer das letzte, die letzten beiden, oder die letzten drei Felder auf 0 bzw. 255), ist dies bei klassenlosen Adressen also nicht mehr gegeben, dies kann leicht zu Verwirrung führen, weil das binäre Zählen (noch) nicht zu des Menschen Stärken zu zählen ist.

Dies waren die speziellen Hostadressen, aber es gibt auch spezielle Netzadressen. Während die meisten IP-Adressen weltweit eindeutig sein müssen, und daher zentral von der ICANN (Internet Corporation for Assigned Names and Numbers, Nachfolger IANA) vergeben werden, gibt es drei Bereiche, die für eine private Nutzung freigegeben sind. Diese Netze wurden schon seit Anbeginn des Internet Protocols festgelegt und sind daher wenig überraschend in Klasse A, B und C Netze eingeteilt. Im Einzelnen sind dies:

- Ein Klasse-A-Netz, 10.0.0.0
- Klasse-B-Netze von 172.16.0.0 bis 172.31.0.0
- Klasse-C-Netze von 192.168.0.0 bis 192.168.255.0

Diese lokalen Netze können beispielsweise in Privathaushalten oder firmenintern genutzt werden, sodaß hier keine teuren „richtigen“ IP-Adressen benötigt werden. Eine weitere Anwendung wird im Kapitel 3.3 angesprochen.

3.2. Das Routing in Internets

Das zweite Feature des Internet Protocols, das es für die Entwicklung großer Netze geeignet macht, ist das Routing. Das bedeutet, daß ein IP-Datagramm seinen Weg zum Ziel findet.

Die IP-Adressen haben geographisch gesehen nichts mit den Zielen zu tun. Anhand der Netzadressen kann aber darauf geschlossen werden, über welche Wege man dort hin kommt. Das ist ein bißchen ein graphentheoretisches Problem, aus Sicht der Pakete gibt es nur Ecken (Router) und Kanten (Leitungen), die irgendwie zusammenhängen, aber über Längen und Position weiß niemand etwas.

Jeder Router weiß anhand seiner Routing-Tabelle, wie er mit Datagrammen, die eine bestimmte Zieladresse haben, verfahren muß. Ein Router, der ein LAN mit der großen Welt verbindet weiß

zum Beispiel, daß alle Datagramme mit Zieladresse innerhalb des LANs über dieses Interface müssen, alle anderen, die „raus“ sollen, über das andere wandern sollen.

Wenn Router nun netzartige Konstrukte erzeugen, also z.B. vier oder fünf Leitungen haben, dann könnten die Entscheidungen komplexer getroffen werden. Oft sind die gleichen Ziele auch über mehrere Verbindungen zu erreichen, nur zu mehr oder weniger guten Konditionen, oftmals als *Kosten* bezeichnet. Der Router versucht dann die Datagramme auf den besten Weg zu schicken, und wenn dieser nicht nutzbar sein sollte (weil z.B. eine Leitung vom Bagger durchtrennt wurde), kann er sich entscheiden, die Datagramme über eine andere Leitung zu schicken, auf daß sie vom dortigen Router weiter geleitet werden.

Einen Router bezeichnet man in diesem Zusammenhang auch als *Hop*. Ein Datagramm hüpft also zu seinem Ziel. Diese Hops kann man beispielsweise mit dem *traceroute*-Programm sichtbar machen, unter Windows heißt es `tracert` (ein Relikt aus dem Zeitalter auf 8 Zeichen begrenzter Dateinamen). Es arbeitet eigentlich furchtbar simpel: es sendet Datagramme mit geringer, inkrementell wachsender TTL (Time-to-live, siehe Kapitel 3.5.1) auf die Reise, und fängt die ICMP-Meldungen auf, wenn die TTL abgelaufen und das Datagramm deshalb verworfen wurde. Eine Ausgabe könnte wie diese aussehen:

```
C:\>tracert jpl.nasa.gov

Routenverfolgung zu jpl.nasa.gov [137.78.160.180]  über maximal 30 Abschnitte:

  1    <1 ms    <1 ms    <1 ms  192.168.0.1
  2    44 ms    45 ms    42 ms  217.0.116.28
  3    42 ms    42 ms    41 ms  217.0.66.22
  4   132 ms   131 ms   132 ms  was-e4.was.us.net.dtag.de [62.154.14.134]
  5   132 ms   132 ms   131 ms  p16-1-0-3.r20.asbnva01.us.bb.verio.net [129.250.9.141]
  6   165 ms   167 ms   166 ms  p16-0-1-1.r21.dllstx09.us.bb.verio.net [129.250.5.34]
  7   232 ms   233 ms   232 ms  p16-1-1-2.r20.lsanca01.us.bb.verio.net [129.250.4.196]
  8   213 ms   211 ms   212 ms  p64-0-0-0.r21.lsanca01.us.bb.verio.net [129.250.2.113]
  9   212 ms   211 ms   212 ms  129.250.16.6
 10   215 ms   218 ms   217 ms  ge-9-3.a01.lsanca02.us.ce.verio.net [198.172.117.162]
 11   212 ms   213 ms   212 ms  jpl-vlan2004.ln.net [130.152.181.51]
 12   219 ms   214 ms   213 ms  b230-edge-171.jpl.nasa.gov [137.78.10.230]
 13   213 ms   214 ms   214 ms  jpl.nasa.gov [137.78.160.180]

Ablaufverfolgung beendet.
```

3.3. Network Address Translation

Das Verfahren der Network Address Translation wird vorwiegend aus Gründen der Sparsamkeit mit öffentlichen IP-Adressen eingesetzt, aber bietet auch ein paar Vorteile was die Sicherheit der Netze gegen Angriffe von außen angeht.

Bei diesem Verfahren besitzt nur ein System eine „echte“ IP-Adresse, das zu verbindende Netz jedoch verwendet private Adressen, aus einer der drei unter 3.1 genannten Bereichen. Die Datagramme dieser Hosts sind natürlich nicht dafür geeignet, durch das Internet zu reisen. Zum einen werden sie sowieso vom nächsten Router verworfen, zum anderen wüsste aber auch bei den Antworten niemand, wohin sie gehen müssten. Die Adresse 192.168.0.1 ist mit Sicherheit eine der häufigsten Adressen der Welt.

Anstatt die Datagramme also ungerüstet auf den Weg zu schicken, nimmt der *Gateway* eine Address Translation vor. Unter einem Gateway versteht man ein System, das zwei Netze miteinander verbindet, in diesem Fall bindet er das LAN an das Internet an. Der Gateway besitzt eine „echte“ IP-Adresse, und bildet sämtlichen Verkehr, der aus dem LAN kommt, auf diese ab. Dabei merkt sich das System quasi über welchen Port die Daten rausgesendet werden, und welche lokale Adresse sich dainter verbirgt. Von Außen sieht es genau so aus, als ob der Gateway alle Anfragen selbst losschickt (was er ja gewissermaßen auch tut), das Verfahren ist für alle Teilnehmer völlig transparent.

Der einzige Haken an der Geschichte ist, daß die Rechner innerhalb des LANs nicht direkt von außen angesprochen werden können. Der Gateway muß stets informiert sein, welche Verbindung auf welchen Host abgebildet werden soll. Wenn beispielsweise einer dieser lokalen Rechner einen Webserver laufen haben soll, der vom Internet aus erreicht werden können soll, so muß der Gateway angewiesen werden, alle Anfragen, die an seinen Port 80 gehen, direkt an den Rechner im LAN weiterzuleiten. Dies wird als *port forwarding* bezeichnet.

Diese Einschränkung ist aber gleichzeitig auch ein Bonus für die Sicherheit, denn wenn ein nicht beabsichtigter Service angeboten wird (beispielsweise ein trojanisches Pferd, in diesem Fall ein Schadprogramm, das über das Internet Befehle entgegen nehmen kann), so kann dieser nicht von außen erreicht werden. Das schützt freilich nicht, wenn die Schadsoftware aktiv raussendet (oftmals als „nach Hause telefoniert“ bezeichnet, in Anlehnung an E.T. den Außerirdischen aus dem gleichnamigen Film), Schutz davor bietet nur ein vernünftig administrierter Firewall, doch dies geht über den Rahmen dieses Kapitels deutlich hinaus. Ein nettes Buch dazu ist im O'Reilly-Verlag erschienen [30].

3.4. ARP – das Address Resolution Protocol

3.4.1. Überblick über das Address Resolution Protocol

Das Address Resolution Protocol (RFC 826 [16]) dient als Bindeglied zwischen Netzwerkschicht und Internetschicht. Im folgenden soll davon ausgegangen werden, daß die darunterliegende Netzwerktechnologie Ethernet ist, wobei sie auch Token Ring oder ARCNet sein könnte.

Eine Ethernetadresse (auch MAC-Adresse genannt) besteht aus 48 Bit. Die übliche Schreibweise ist durch Doppelpunkte getrennte Gruppen aus jeweils zwei Hexadezimalziffern, z.B. 00:04:76:9C:C8:E9, mancherorts sind die Doppelpunkte auch durch Bindestriche ersetzt.

Diese Ethernetadresse ist nötig, um Daten in einem Ethernet zu versenden. Auf dieser Ebene weiß niemand etwas von IP-Adressen, für das Netzwerk ist völlig transparent, welche Art von Daten auf den Frames reist. Das Internet besteht nun jedoch aus Hosts, die sich über ihre IP-Adressen identifizieren, und um Datagramme von einem Host zum anderen zu schaffen, müssen sie über das dazwischen liegende Medium versandt werden.

Weil sich aus den IP-Adressen in keinster Weise die Ethernetadressen ableiten lassen, muß ein Mechanismus zum *rausfinden* der Ethernetadressen bereitgestellt werden. Genau dies ist die Aufgabe von ARP. Der sendende Host fragt dabei im lokalen Netzwerk (welches das richtige ist, wenn er an mehrere angeschlossen ist, verrät ihm seine Routing-Tabelle, siehe Kapitel Routing, 3.2) herum, wie etwa ein Besitzer eines Autos im Supermarkt ausgerufen wird, wenn er jemanden zuparkt: „Der Besitzer der IP-Adresse xyz soll sich bei mir mit seiner Hardwareadresse melden!“. In Abbildung 3.4.1 ist der Aufbau einer solchen ARP-Nachricht beschrieben.

0	15	16	31
Hardware Address Type		Protocol Address Type	
HAddr Length	PAddr Length	Operation	
Sender Hardware Address (first 4 octets)			
Sender HAddr (last 2 octets)		Sender PAddr (first 2 octets)	
Sender PAddr (last 2 octets)		Target HAddr (first 2 octets)	
Target HAddr (last 4 octets)			
Target PAddr (all 4 octets)			

Abbildung 3.1.: ARP Nachricht

Hardware Address Type
Code für die Art der Hardware-Adresse. Einige gebräuchliche Codes sind:

1	Ethernet (10Mb)
3	Amateur Radio AX.25
4	Proteon ProNET Token Ring
6	IEEE 802 Networks
7	ARCNET
24	IEEE 1394.1995

Eine vollständige Liste findet man bei der *IANA* (Internet Assigned Numbers Authority) unter [13].

Protocol Address Type
Code für die Art der Protokoll-Adresse. Für IPv4 verwendet man hier 0x800, die Komplette Liste findet man unter [14].

Hardware Address Length
Dieses Feld enthält die Länge der Hardware-Adresse, in Oktetts. Im Fall von Ethernet ist dies 6.

Protocol Address Length
Analog dazu enthält dieses Feld die Länge der Protokoll-Adresse, bei IP also 4.

Operation
Anhand diesem Codes werden die verschiedenen Operationen unterschieden, die beiden wichtigsten sind 1 für Requests und 2 für Reply. Die komplette Liste findet sich in [13].

Sender Hardware Address
Dieses 6 Oktetts breite Feld enthält die Hardware-Adresse des Senders. Dieses Feld trägt immer eine gültige Adresse.

Sender Protocol Address
Dieses 4 Oktetts breite Feld berherbergt die Protokoll-Adresse des Senders, hier also die IP-Adresse.

Target Hardware Address
Dieses ist die Hardware-Adresse des Ziels. Für ARP Requests ist der Inhalt undefiniert.

Target Protocol Address
Die Protokoll-Adresse des Ziels, im Request-Fall also die Adresse, die man auflösen möchte, im Reply-Fall die Adresse des Urhebers der Anfrage.

3.4.2. Auflösung von IP-Adressen

Als Beispiel-Netzwerk soll das in Anhang C angegebene Referenznetzwerk dienen. Die Workstation work1.local möchte nun an work3.local ein IP-Datagramm senden. Dazu liegt die Information vor, daß work3.local die IP-Adresse 192.168.1.43 hat.

Zuerst sendet work1.local ein ARP Request. Die Zieladresse des Ethernetframes ist hierbei die Broadcast-Adresse ff:ff:ff:ff:ff:ff, doch wir möchten diese unterste Schicht des Linklayers nicht betrachten. Die ARP-Nachricht enthält als Operation den Wert 1 für REQUEST, Hardware Address Type ist 1 für Ethernet, Protocol Address Type ist 0x800 für IPv4. Als Sender Hardware Address steht die 01:23:9a:4b:8a:2b drin, Sender Protocol Address ist 192.168.1.41. Die Target Protocol Address ist 192.168.1.43, die Target Hardware Address enthält keine Information.

Dieser Frame wird nun auf das Netzwerk gegeben und jeder Rechner schaut sie an (Ethernet-Broadcast). Wenn das ARP-Modul des TCP/IP-Stacks eine seiner IP-Adressen im Feld Target Protocol

Address erkennt, so wird die Nachricht verarbeitet, andernfalls ignoriert (jedoch ist es legitim die Sender Hardware Address und Sender Protocol Address in den ARP-Cache zu übernehmen).

work3.local hat diese Nachricht empfangen und sendet eine Antwort. Dazu geht der Frame zurück an 01:23:9a:4b:8a:2b (Linklayer), Sender Hardware Address enthält 01:26:da:52:fb:6e (Ethernet-Adresse von work3.local), Sender Protocol Address enthält 192.168.1.43 (IP-Adresse von work3.local). Die Target Hardware Address ist hierbei 01:23:9a:4b:8a:2b und die Target Protocol Address ist 192.168.1.41. Der Wert für Operation wird auf 2 für REPLY gesetzt.

work3.local merkt sich praktischerweise die IP-Adresse und Hardware-Adresse von work1.local in seinem ARP-Cache, damit für weitere Anfragen keine Requests nötig werden. Wenn work1.local das Reply sieht, so fügt auch er diese Informationen seinem Cache hinzu.

Die Einträge im ARP-Cache haben üblicherweise einen Zeitstempel, damit sie nach einer gewissen Zeit verworfen werden können. Andernfalls würde eine Änderung der Zuordnung nur dadurch auffallen, daß ein Rechner mit der Ziel-IP-Adresse zwar am Netz ist, aber keine Frames erhält, weil jemand noch an die alte Hardware-Adresse sendet (diese Frames werden dann ignoriert, sofern nicht jemand anderes die alte Hardware-Adresse zugeteilt bekommt).

Dieses Beispiel ist nun das Senden innerhalb eines Netzwerkabschnitts. ARP existiert auch nur auf dieser Ebene, ARP-Frames werden nie über Routergrenzen hinweg verbreitet; außerhalb „ihres“ Netzes sind Hardwareadressen auch nutzlos.

Das Schicksal eines IP-Datagramms, das über das Internet reist, ist aber etwas verwickelter, denn es passiert in der Regel viele lokale Netze. Hierbei löst jeder Router ein ARP-Request aus, falls er die Hardwareadresse des nächsten Hops nicht kennt.

3.5. IP – das Internet Protocol

3.5.1. Überblick über das Internet Protocol

Das Internet Protocol (IP) ist in RFC 791 [17] beschrieben. Es dient der Internetschicht (vgl. Abschnitt 1.3) zum Transport der Daten von einem Host zum anderen. Lange Pakete können hierbei in kürzere Teile zerlegt (*fragmentiert*) werden (immer dann nötig, wenn ein Datagramm länger als die Framelänge der darunterligenden Übertragungsschicht ist, bei Ethernet z.B. 1500 Oktetts). Dieser Mechanismus wird in Abschnitt 3.5.2 ausführlich behandelt. Jedes IP-Paket (oder *Datagramm*) beginnt mit einem Header, gefolgt von Nutzdaten, die weitere Header wie z.B. TCP (4.1.1) oder UDP (4.2), enthalten können. Der Aufbau eines solchen Headers ist in Abbildung 3.2 dargestellt.

<table>
<tr><td colspan="2">0</td><td colspan="2">15</td><td colspan="2">16</td><td>31</td></tr>
<tr><td>Version</td><td>IHL</td><td colspan="2">Type of Service</td><td colspan="2">Total Length</td></tr>
<tr><td colspan="3">Identification</td><td>Flags</td><td colspan="2">Fragment Offset</td></tr>
<tr><td colspan="2">Time to Live</td><td colspan="2">Protocol</td><td colspan="2">Header Checksum</td></tr>
<tr><td colspan="6">Source Address</td></tr>
<tr><td colspan="6">Destination Address</td></tr>
<tr><td colspan="5">Options</td><td>Padding</td></tr>
</table>

Abbildung 3.2.: Internet Protocol (Version 4) Header

Version
Die Versionsnummer, bei IPv4 ist dies 4.

IHL
IHL steht für *Internet Header Length* und bezeichnet die Länge des IP-Headers in 32-Bit-Worten. Anhand diesen Eintrags kann erkannt werden, ab welchem Offset die Nutzlast beginnt. Die Mindestgröße eines IP-Headers ist 20 Oktetts, also ist der kleinste hier zulässige Wert 5. Auf Grund der Begrenzung auf 4 Bit kann der größte Header also $2^4 * 4 = 64$ Oktetts lang sein.

Type of Service
Dieses Headerfeld bestimmt in abstrakter Form die Qualität des angebotenen Dienstes. Router können anhand dieser Parameter zum Beispiel die Route wählen, die das Datagramm zum Ziel einschlagen soll.

Bits	Funktion	Bedeutung
0-2	Dringlichkeit	siehe Tabelle unten
3	Verzögerung	0 → normal, 1 → gering
4	Durchsatz	0 → normal, 1 → hoch
5	Zuverlässigkeit	0 → normal, 1 → hoch
6-7	reserviert	sollte auf 0 gesetzt werden

111	Network Control
110	Internetwork Control
101	CRITIC/ECP
100	Flash Override
011	Flash
010	Immediate
001	Priority
000	Routine

Diese Einträge sind nur als „Wünsche“ zu verstehen, ob und wie ein Router oder Host darauf reagiert ist völlig ihm selbst überlassen.

Total Length
Dieses Feld enthält die Gesamtlänge des Pakets, in Oktetts. Durch die Breite von 16 Bit ist die maximale Größe eines IP-Datagramms auf $2^{16} = 64KByte$ begrenzt.

Identification
Anhand diesen Felds können die Fragmente eines Pakets erfolgreich zusammengesetzt werden. Jedes Paket sollte eine einzigartige ID haben (was bei 16 Bit natürlich mit der Zeit zu Dopplungen führt).

Flags
Das Flag-Feld enthält drei mögliche Flags:

Bit	Name	Bedeutung
0	reserviert	muß auf 0 stehen
1	Don’t Fragment	verhindert das Fragmentieren des Pakets
2	More Fragments	es folgen weitere Fragmente

Ist das ”Don’t Fragment”-Bit gesetzt, so wird ein Datagramm entweder unfragmentiert weitergesendet, oder – wenn es zu „dick“ für die Leitung ist – weggeworfen. Hierbei wird eine ICMP-Nachricht mit dem Code für ”Destination unreachable” an den Absender des Datagramms gesendet (vgl. Abschnitt 4.3.1).

Das ”More Fragments”-Bit ist in jedem fragmentierten Paket auf 1 gesetzt, das noch Folgefragmente hat. Das letzte Fragment hat dieses Bit also auf 0 stehen.

Fragment Offset
Dieser Wert ist der Offset, an dem aus Sicht des Gesamtdatagramms das vorliegende Fragment

eingefügt werden soll. Der Wert ist mit 8 zu multiplizieren, um auf den Offset in Oktetts zu kommen.

Time to Live
Dieses Feld enthält einen Zähler, der bei jedem Hop um eins dekrementiert wird. Erreicht er 0, so wird das Datagramm verworfen. Der Hintergedanke ist das endlose Kreisen von Datagrammen zu verhindern (z.B. auf Grund einer falschen Routing-Information).

Protocol
Die über der Internetschicht liegende Transportschicht muß wissen, welches Protokoll verwendet wird. Diese Information wird im Protocol-Feld gespeichert. Werte für die einzelnen Protokolle sind (auszugsweise):

1	ICMP
2	IGMP
6	TCP
17	UDP

Die komplette Liste findet sich in [15].

Header Checksum
Dieses Feld enthält eine Checksumme über das komplette Datagramm. Die Checksumme wird folgendermaßen gebildet:

- Das Checksummen-Feld wird auf 0 gesetzt
- Die Daten werden als 16-Bit-Worte aufgefasst
- Von jedem 16-Bit-Wort wird das (Einer-)Komplement gebildet
- Diese Komplemente werden aufaddiert
- Das Komplement der Summe ist die Checksumme

Der Hintergedanke ist die Einfachheit beim Überprüfen: es werden wieder alle 16-Bit-Worte aufaddiert. Wenn die Summe nur aus 1-Bits besteht (also das Feld 0xffff als Wert hat), dann sind die Daten in Ordnung.

Eine sehr ausführliche Erklärung mit Beispielen findet sich in [18].

Source Address
Die Quelladresse des Datagramms.

Destination Address
Die Zieladresse des Datagramms.

Options
Das Internet Protocol bietet eine Reihe von Optionen teils variabler Länge.

Eine Option kann entweder ein Oktett lang sein, oder aber aus einem Oktett für die Kennung, einem für die Längenangabe, und dann den eigentlichen Daten bestehen. Ist die Gesamtlänge des IP-Headers kein Vielfaches von 32 Bit (wie es das *IHL*-Feld fordert), so müssen entsprechend viele Nullen eingefügt werden (*Padding*).

Die einzelnen Optionen werden in Abschnitt 3.5.3 vorgestellt.

3.5.2. Fragmentierung und Reassemblierung

Verschiedene Netze können verschiedene MTUs haben. MTU steht für Maximum Transfer Unit, und bezeichnet die maximale Länge eines Datagramms, das über das entsprechende Netz reisen kann.

Die MTU hängt von der darunterliegenden Netzwerktechnologie ab, so ist bei Ethernet beispielsweise 1500 Oktetts der höchstmögliche Wert für die MTU. Sie kann in der Regel willkürlich verkleinert werden (bis zu einem festen Minimum). Dies ist sinnvoll, wenn z.B. bekanntermaßen ein Netz mit geringerer MTU folgt. Wenn ein IP-Datagramm nämlich nicht durch eine Verbindung „passt“, so kann es fragmentiert werden – sofern das Don't Fragment Bit im Header nicht gesetzt ist.

Bei der Fragmentierung werden die Datagramme in kleinere Datagramme zerlegt, die erst wieder am Endpunkt der Übertragung zusammengesetzt werden. Das heißt, wenn unterwegs kleinere MTUs auftreten dürfen die Fragmente erneut fragmentiert werden, nicht jedoch bei größeren MTUs zusammengesetzt werden. Es wäre möglich, daß ein Teil der Fragmente eine andere Route nimmt, in diesem Fall wäre eine Reassemblierung ohnehin nur beim Ziel möglich.

Die für eine Fragmentierung relevanten Headerinformationen sind die beiden Flags Don't Fragment (DF) und More Fragments (MF). Ist das DF-Bit gesetzt, und eine Fragmentierung wäre nötig, so wird das Datagramm verworfen, dem Absender wird eine ICMP Nachricht (Typ 3, Code 4) zugestellt, die ihn davon in Kenntnis setzt.

Das MF-Bit wird in jedem Fragment eines Datagramms auf 1 gesetzt, um anzuzeigen, daß weitere Datagramme folgen. Im letzten Datagramm wird es auf 0 gesetzt.

Die nächste Information steckt im Feld *Fragment Offset*, das 13 Bit breit ist. Während die maximale Größe eines Datagramms 65536 Oktetts beträgt, können in 13 Bit nur 8192 Informationen untergebracht werden. Aus diesem Grund werden Fragmentierungen nur an 8 Oktett Grenzen vorgenommen, sodaß die Offsets also nur Vielfache von 8 sein können. Damit reichen auch wieder 8192 mögliche Offsets aus (8192 * 8 = 65536).

Als Beispiel soll ein Datagramm mit dem folgenden Header dienen:

Total Length	1450
ID	20495
DF	0
MF	0
Fragment Offset	0

Die anderen Header-Felder sind für die Betrachtung irrelevant, und bleiben identisch. Die ID bleibt ebenfalls in jedem Fragment gleich, denn nur anhand dieser Information ist die Zusammensetzung möglich.

Angenommen die neue MTU beträgt 1250 Oktetts, dann ist die maximale Länge der Nutzdaten in einem solchen Datagramm 1230 Oktetts, wenn man davon ausgeht, daß dem IP-Header keine Options-Header folgen. Im Ursprungsdatagramm befinden sich unter dieser Prämisse 1430 Oktetts Nutzdaten. Das größte Vielfache von 8, das gerade kleiner oder gleich der 1230 Oktetts, die in dem Fragment untergebracht werden können, ist 1224 (nämlich 153 * 8). Damit sieht das erste Fragment so aus:

Total Length	1244
ID	20495
DF	0
MF	1
Fragment Offset	0

Die Gesamtlänge (*Total Length*) ist immer die des Datagramms in dessen Header sie steht, also nicht etwa die Gesamtlänge des komplettierten Datagramms! MF steht auf 1, um anzuzeigen, daß weitere Fragmente folgen. Ein weiteres um genau zu sein, nämlich:

Total Length	226
ID	20495
DF	0
MF	0
Fragment Offset	153

Die 226 berechnen sich aus 1430 Oktetts (Nutzdaten des Ursprungsdatagramms) minus 1224 Oktetts (die im ersten Fragment untergebrachten) plus 20 Oktetts für den Header. Das MF Bit ist auf 0 gesetzt, denn hierbei handelt es sich um das letzte Fragment.

Angenommen, als nächstes schließt sich ein Netz an, dessen MTU 560 Oktetts beträgt. Das bedeutet, das erste Fragment muß erneut fragmentiert werden, während das zweite ohne Probleme durchpasst.

Die Nutzdaten in einem solchen Fragment können maximal 540 Oktetts betragen, das größte Vielfache von 8, das gerade kleiner gleich 540 ist, ist 536, nämlich 67 * 8. Das erste Fragment besitzt 1224 Oktetts Nutzdaten, damit werden drei weitere Fragmente fällig, mit zweimal 536 und einmal 152 Oktetts Nutzdaten. Diese lauten:

Total Length	556
ID	20495
DF	0
MF	1
Fragment Offset	0

Total Length	556
ID	20495
DF	0
MF	1
Fragment Offset	67

Total Length	172
ID	20495
DF	0
MF	1
Fragment Offset	134

Wenn Datagramme, deren MF-Bit gesetzt ist, weiter zerlegt werden, so behalten alle Teilstücke das MF-Bit, denn das Ende der Fragmentliste ist in einem anderen Datagramm unterwegs. Wenn diese vier Fragmente mit den Fragment Offsets 0, 67, 134 und 153 beim Zielhost ankommen, so werden sie wieder reassembliert.

Das erste Fragment bildet den Anfang, 556 - 20 = 536 Oktetts Nutzdaten. Das zweite trägt ebenfalls 536 Oktetts Nutzdaten, am Offset 67 * 8 = 536 anzufügen. Das Dritte Fragment enthält 152 Oktetts Nutzdaten, am Offset 134 * 8 = 1072, und das letzte, aus dem ersten Durchgang erzeugte, besitzt 206 Oktetts Nutzdaten, mit dem Offset 153 * 8 = 1224. Macht zusammen 1430 Oktetts, auffallend genau die Länge der Nutzdaten, die das ursprüngliche Datagramm befördern sollte.

Man sollte auf jeden Fall immer im Hinterkopf behalten, daß die Fragmentierung von Datagrammen mit zusätzlichem Aufwand bei demjenigen, der die Fragmentierung vornimmt, und bei demjenigen, der die Reassemblierung vornimmt, verbunden ist. Häufige Fragmentierung kann durch die Bestimmung der *Path MTU* verhindert werden. Das ist sinnbildlich die engste Stelle der Gesamtverbindung. Das

Verfahren zur Ermittlung der Path MTU ist sehr einfach: es werden Datagramme mit zunehmender Größe und gesetztem DF-Bit an das Ziel geschickt, und sobald eine ICMP-Meldung wegen zu großer MTU zurückkommt, hat man die Path MTU überschritten. Das Allheilmittel ist dies allerdings nicht, weil sich die Routen der Datagramme auch während einer bestehenden Verbindung ändern können, werden sie doch dynamisch für jedes einzelne Datagramm von den beteiligten Routern bestimmt.

3.5.3. IP-Optionen

Das Internet Protocol ermöglicht dem Sender verschiedene Optionen zu setzen. Sie werden dem Header angehängt und können eine variable Länge besitzen. Dabei werden zwei Fälle berücksichtigt:

option-type

option-type	option-length	option-data

Optionen, die aus nur einem Oktett bestehen, benötigen also keine Längenangabe. Das Feld option-type ist weiter untergliedert:

1 bit	copied flag
2 bits	option class
5 bits	option number

Das *copied flag* gibt an, daß die Option im Falle einer Fragmentierung des Datagramms in jedes Fragment kopiert werden soll. Die *option class* hat einen der vier folgenden Codes, wobei nur zwei verwendet werden:

0	control
1	reserved for future use
2	debugging and measurement
3	reserved for future use

Hier die (vollständige) Liste der Optionen wie sie in [17] zu finden ist:

class	number	length	description
0	0	—	End of Option List
0	1	—	No Operation (1 octet)
0	2	11	Security
0	3	var.	Loose Source Routing
0	9	var.	Strict Soruce Routing
0	7	var.	Record Route
0	8	4	Stream ID
2	4	var.	Internet Timestamp

Im Folgenden sollen die Optionen nur kurz angesprochen werden; in der Praxis trifft man sie sowieso fast nie an. Im RFC findet sich eine ausführliche Erklärung.

End of Option List
Wird am Ende der Optionsliste verwendet, falls dies nicht mit dem Ende des Headers zusammenfällt.

No Operation
Tut nichts; kann zwischen Optionen gesetzt werden, um folgende Optionen auf 32-Bit-Grenzen auszurichten.

Security
Das Internet Protocol wurde ursprünglich für einen militärischen Auftraggeber entwickelt. Mit dieser

Option lassen sich Dinge beschreiben, die als *security* (Sicherheit), *compartmentation* (Separierung), *handling restrictions* (Einschränkungen der Handhabung) sowie *TCC (closed user group) parameters* bezeichnet werden.

Loose Source and Record Route
Mit dieser Option kann der Weg, den ein Datagramm nehmen wird, beeinflusst werden. Dabei kann jedoch eine beliebige Anzahl Hops „zwischengeschoben“ werden, wenn ein Router dies für sinnvoll hält. Erreicht das Datagramm einen Hop aus der Liste, so ersetzt dieser seine Adresse durch diejenige des Interfaces, über den das Datagramm weiter geschickt wird.

Strict Source and Record Route
Ähnlich der Loose Source Route bestimmt auch die Strict Source Route den Weg, jedoch muß jeder Gateway oder Host das Datargamm zwingend an die nächste vermerkte Adresse weitersenden.

Record Route
Im Datenfeld dieser Option wird die Route des Datagramms aufgezeichnet.

Stream Identifier
Ermöglicht die Übertragung eines SATNET (Atlantic Packet Satellite Network [28]) stream identifiers über Netzwerke, die das Stream-Konzept nicht unterstützen.

Internet Timestamp
Jeder Hop fügt einen Zeitstempel ein, einen 32 Bit breiten, rechtsbündig ausgerichteten Wert. Ist das oberste Bit nicht gesetzt, so handelt es sich um Millisekunden seit Mitternacht (UTC), ist es gesetzt, so handelt es sich um eine andere Einheit.

Es gilt anzumerken, daß bis auf *Record Route* keine der Optionen wirklich Anwendung findet. Gerade das *Source Routing* wird zumeist unterbunden (indem Router solche Datagramme einfach wegwerfen), weil sich damit allerlei Schapernack treiben lässt, und in düsteren Zeiten wie den heutigen als Angriffswaffe gewertet wird. Früher war es als Mittel zum Zweck ersonnen, nämlich um über schlechte oder unvollständige Routing-Tabellen hinweg zu kommen. Durch die automatische Erzeugung von Routing-Tabellen durch sog. Routing-Protokolle ist die Sachlage aber inzwischen deutlich angenehmer geworden.

4. Die Transportschicht

4.1. TCP – das Transmission Control Protocol

4.1.1. Überblick über das Transmission Control Protocol

Das Transmission Control Protocol (TCP) ist in RFC 793 [19] beschrieben. Es bietet eine *sichere* Übertragung zwischen zwei Hosts als Host-zu-Host-Verbindung auf der Ebene der Transportschicht (vgl. Kapitel 1.3). Mit sicher ist gemeint, daß verlorengegangene Daten „nachbestellt“ werden können. Mit TCP lässt sich ein Bytestrom erzeugen, das heißt, daß die Daten, die auf der einen Seite reingehen auf der anderen Seite in genau der Reihenfolge und genau diesem Inhalt wieder herauskommen.

TCP bietet, wie UDP auch, das Konzept der Port-Nummern. Dies ist eine Form von Multiplexing, eine weitere Verfeinerung der Adressierung. Mit der IP-Adresse kann ein Host genau beschrieben werden. Die Port-Nummern beschreiben nun einen Prozeß (genauer: einen Endpunkt, ein Prozeß kann mehrere haben) auf diesem Host.

Die Port-Nummern < 1024 sind sogenannte *well-known ports*, den Bereich von 1024 bis 49151 bezeichnet man als *known ports*. Sie sind (nicht alle) Diensten zugeordnet, die man dort zu finden erwartet. Verwaltet werden diese Port-Nummern durch die IANA, eine vollständige Liste der TCP und UDP Ports findet man unter [29].

Der Header eines TCP-Datagramms ist in Abbildung 4.1 zu sehen.

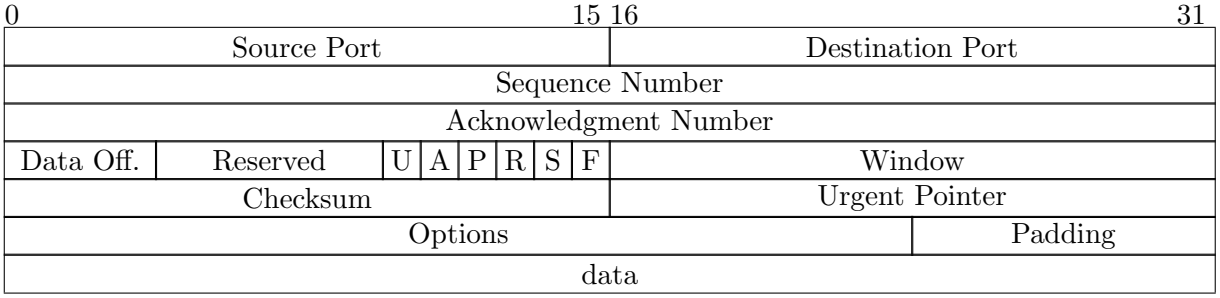

Abbildung 4.1.: Transmission Control Protocol Header

Source Port
Quellport der Verbindung. Das Feld ist 16 Bit breit, es stehen also 65536 Ports zur Verfügung.

Destination Port
Zielport der Verbindung.

Sequence Number
TCP bietet eine *Verbindung* zwischen den beteiligten Hosts (durch den Drei-Wege-Handshake aufgebaut, siehe Abschnitt 4.1.3); durch die Sequence Number können die Pakete diesen Verbindungen zugeordnet werden. Die TCP-Sequenznummern werden in Abschnitt 4.1.2 ausführlich erörtert.

Acknowledgment Number
Der Empfang eines jeden TCP-Datagramms muß quittiert werden; dieses Feld beschreibt dabei das

(die) Datagramm(e), deren Empfang bestätigt wird. Der Zusammenhang wird in Abschnitt 4.1.4 ausführlich veranschaulicht.

Data Offset
Entsprechend dem *IHL*-Feld des IP-Headers gibt dieses Feld den Offset zu den Nutzdaten an, in 32-Bit-Worten. Typischer Wert ist 5 (wenn keine Optionen gesetzt sind).

URG
Das Urgent-Bit ist gesetzt, wenn das Feld *Urgent Pointer* einen gültigen Wert enthält.

ACK
Das Acknowledgment-Bit ist gesetzt, wenn das Feld *Acknowledgment Number* einen gültigen Wert hat.

PSH
Das Push-Bit beschreibt ein Paket, das umgehend „durch die Leitung gedrückt“ werden muß. Noch ausstehende Daten werden dann sofort versandt, anstatt sie zum Beispiel für größere Segmente anzusammeln.

RST
Das Reset-Bit wird in Paketen gesetzt, die eine Verbindung zurücksetzen sollen.

SYN
Das Synchronize-Bit wird beim Aufbau der Verbindung verwendet.

FIN
Das FIN-Bit dient dem Abbau der Verbindung, siehe Abschnitt 4.1.3.

Window
Beschreibt die Anzahl von Oktetts, beginnend ab demjenigen, das im Acknowledment-Feld angezeigt wird, die der Absender des Datagramms annehmen wird. Dies ist der Mechanismus der Flußkontrolle, die anschließend besprochen wird.

Checksum
Dieses Feld enthält die Checksumme des Datagramms. Sie geht nicht nur über das Datagramm ab dem TCP-Header, sondern deckt auch einen *Pseudoheader*, der zwecks Berechnung der Checksumme dem TCP-Header vorangestellt wird, ab (siehe Abbildung 4.2).

0	15 16	31
Source Address		
Destination Address		
zero	PTCL	TCP Length

Abbildung 4.2.: TCP Pseudo Header

Dabei enthält PCTL die Protokoll-Nummer aus dem *Procotol*-Feld des IP-Headers, die beiden Adressen stammen entsprechend ebenfalls denen des IP-Headers; der Inhalt des Felds *TCP Length* entspricht der Länge des TCP-Datagramms, also der Gesamtlänge des Datagramms minus der Länge des IP-Headers inklusive Optionen.

Urgent Pointer
Mit dem Urgent-Bit und dem Urgent-Pointer können besonders wichtige Daten innerhalb eines Datenstroms gekennzeichnet werden. Der Urgent-Pointer versteht sich hierbei als Offset zur aktuellen Acknowledgement Nummer, der auf das erste Oktett hinter den dringenden Daten zeigt. In Abschnitt 4.1.4 wird dies genauer erklärt.

Options
TCP kennt drei Optionen, die dem Header folgen können. Dies sind das *NOP* und das *End of Option List*, die identisch zum Internet Protocol sind, sowie eine weitere Option, mit der die maximale Segmentgröße festgelegt werden kann. Diese Option darf nur beim Verbindungsaufbau gesendet werden.

data
Bei TCP ist es durchaus üblich, daß Datagramme ohne Nutzdaten versendet werden. Dies ist zum einen beim Verbindungauf- und -abbau üblich, zum anderen aber auch bei Quittungen, die nur das ACK-Bit und die entsprechenden Sequenznummern als Information übermitteln müssen.

4.1.2. TCP-Sequenznummern

Das Konzept der Sequenznummer ist der Schlüssel zur sicheren Übertragung mittels TCP.

Jedes einzelne Oktett einer Übertragung besitzt eine eigene Sequenznummer. Beim Verbindungsaufbau (siehe 4.1.3) werden die Sequenznummern ausgehandelt, sodaß das Gegenüber weiß, welche Daten als nächstes kommen werden (bzw. es sollten).

Wenn ein Host ein Datagramm empfangen hat, dann quittiert er dies durch ein ACK-Paket, dessen Acknowledgement-Nummer die als nächstes erwartete Sequenznummer ist. Das ist also die Sequenznummer des erhaltenen Datagramms plus die Menge an enthaltenen Nutzdaten.

Die genauen Umstände zur Wahl der initialen Sequenznummer sollten im RFC nachgelesen werden, da die Thematik sehr verwickelt ist, und dieses Kapitel keine erschöpfende Abhandlung des TCPs werden soll. Wichtig ist bei der Wahl der Sequenznummern, daß keine Verwechslung mit bestehenden Verbindungen auftreten dürfen. Dies ist insbesondere nach einem Crash zu beachten, wenn die TCP-Implementierung das „Gedächtnis“ verloren hat. Das RFC schreibt vor, daß für eine gewisse Zeit nichts gesendet werden darf, bis alle noch umherschwirrenden Datagramme sicher eingefangen, und etwaige ACKs oder RSTs empfangen wurden.

Ein Acknowledgement ist kumulativ, das heißt wenn mehrere Datagramme mit aufeinanderfolgenden Sequenznummern empfangen wurden, muß nur eine Quittung gesendet werden, die der letzten Sequenznummer plus die in diesem Datagramm enthaltenen Daten entspricht (= die Sequenznummer des als nächstes erwartete Oktetts).

4.1.3. Der Drei-Wege-Handshake

Um Daten über eine TCP-Verbindung senden zu können, müssen einige Dinge ausgehandelt werden, allen voran die Sequenznummern. Bei TCP wird eine Verbindung durch den sogenannten Drei-Wege-Handshake aufgebaut, bei dem drei Datagramme ausgetauscht werden. Man kann dabei zwei Szenarien unterscheiden: entweder die Verbindung soll aktiv aufgebaut werden (oft als *avtive open* bezeichnet), oder aber eine Anfrage trifft ein (als *passive open* bezeichnet). Der erste Fall ist also gegeben, wenn man den Client betrachtet, der zweite im Fall des Servers.

Der Client (also der Initiator der Verbindung) sendet ein Datagramm mit gesetztem SYN-Flag an den Server. In diesem Datagramm gibt er seine aktuelle Sequenznummer an.

Der Server empfängt nun dieses Datagramm. Wenn auf dem angegebenen Port ein Service verfügbar ist, so wird er mit einem ACK antworten, in dem er in der Acknowledge-Number die Sequenznummer des Clients, um eins erhöht, angibt. Seinerseits sendet er ein SYN-Datagramm mit seiner Sequenznummer. Diese beiden Datagramme können und werden in der Regel als ein einziges Datagramm versandt.

Der Client empfängt nun dieses SYN/ACK-Datagramm, entnimmt die Sequenznummer des Servers, erhöht sie um eins, und sendet diese im Acknowledge-Feld eines ACKs seinerseits zurück. Damit ist die Verbindung vollständig aufgebaut.

Sollte der Zielport auf dem Server nicht „offen“ sein (d.h. daß dort kein entsperchender Prozeß auf eingehende Verbindunganfragen reagiert), so sendet dessen TCP-Modul anstatt eines ACKs ein Datagramm mit gesetztem RST-Flag zurück, und der passenden Acknowledge-Nummer, damit der Client es korrekt zuordnen kann.

Etwas ähnliches passiert beim (ordnungsgemäßen) Abbau einer Verbindung. Hier sendet der Host, der nichts mehr senden möchte, ein Datagramm mit gesetztem FIN-Bit. Der Partner nimmt dies an, und quittiert es entsprechend in einem ACK. Wenn auch er nichts mehr senden möchte, so sendet er ebenfalls ein FIN-Datagramm, das der Empfänger mit ACK quittiert. Dies muß nicht nötigerweise in drei Übertragungen passieren. Wenn ein Host die Verbindung beenden möchte, aber der andere Host noch Daten zu senden hat, dann sendet er diese natürlich bevor die Verbindung abgebaut wird.

Die Alternative dazu ist ein Abbruch einer Verbindung, die durch ein RST-Datagramm verursacht wird. Hierbei ist keinerlei Acknowledgement nötig.

Wie diese Vorgänge von der Socket-Schicht aus wirken, wird in den Abschnitten zu close (7.7) und shutdown (7.8) besprochen.

4.1.4. TCP Datenübertragungen

Wie im Abschnitt über Sequenznummern bereits erwähnt, ist jedem Oktett einer Übertragung eine Sequenznummer zugeordnet. Die beteiligten Hosts wissen nach dem Verbindungsaufbau genau, welche Sequenznummer das nächste Datagramm besitzen sollte. Dadurch ist es möglich zur Verbindung gehörende Datagramme zu identifizieren (genauer: man identifiziert die Bytesequenzen innerhalb dieser Datagramme).

Doch warum ist diese Methode *sicher*?

Angenommen, Host A steht mit seiner Sequenznummer bei 300, Host B bei 500. Die Verbindung wurde bereits wie beschrieben ausgehandelt, d.h. jeder weiß, bei welcher Sequenznummer der Kommunikationspartner steht.

Host A sendet nun ein Datagramm mit 50 Oktetts Nutzdaten, das jedoch bei Host B nie ankommt. Host B sendet folglich keinerlei Quittung, und nach Ablauf eines Timeouts sendet Host A die Daten erneut. Auf diese Weise können Daten schonmal nicht verloren gehen.

Nun hat Host A bereits zwei Datagramme gesendet, einmal 50 und einmal 30 Oktetts, jedoch nur das zweite Datagramm trifft bei Host B ein, also Daten sind zwischendrin verloren gegangen. Host B erwartet ein Datagramm mit der Sequenznummer 300, sieht aber eines mit 350. Er darf das Datagramm zwar annehmen, aber keinesfalls quittieren (denn Host A würde dann zu recht davon ausgehen, daß das erste Datagramm ebenfalls angekommen ist). Nach Ablauf des Timeouts sendet Host A das Datagramm mit der fehlenden Bytefolge erneut. Host B quittiert dieses, und wenn das zuvor aufgefangene Datagramm mit der Bytefolge 350+ nahtlos passt, dann darf er dieses ebenfalls quittieren, also auch beide in einem Rutsch, sodaß die erneute Übertragung des 30 Oktetts langen Datagramms nicht nötig ist.

Zu beachten ist hierbei, daß das Sender-TCP keinesfalls verpflichtet ist, den Bytestrom immer an den gleichen Oktettgrenzen aufzubrechen. Praktischerweise wird eine Implementierung jedoch die Datagramme, die sie gesendet hat, aufheben, und genau diese nochmal versenden, da sich der ganze Aufwand des Einpackens, Checksummen berechnen, ggf. Speicher allokieren etc. somit nicht wiederholt.

Fenstergröße und Flußkontrolle
Ein anderer Mechanismus, der bei Datenübertragungen eine Rolle spielt, ist die Fenstergröße. Das Feld *Window* im TCP-Header kann einen 16 Bit breiten Wert aufnehmen. Mittels diesen Wertes bestimmt der Sender, wie viele Oktetts das an ihn zu sendende Datagramm maximal enthalten darf. Das heißt, es dürfen nur Oktetts mit Sequenznummern < der Acknowledgement-Nummer plus der Fenstergröße enthalten sein.

Durch dieses Verfahren wird verhindert, daß das Ziel mit mehr Daten überflutet wird, als es verarbeiten kann. Das RFC verbietet es einem sendenden TCP während der Verbindung die Fenstergröße zu verkleinern, fordert aber ein solches Verhalten bei anderen TCPs zu tolerieren. Die Fenstergröße darf auf 0 sein, dann muß das Ziel-TCP jedoch in der Lage sein genau ein Byte aufnehmen zu können. (vgl. "Managing the Window", Seite 42 in [19])

Das Push-Flag
Der Sinn oder Unsinn des Push-Flags ist offenbar nicht ganz klar zu benennen. Das Push-Flag soll anzeigen, daß der Sender an dieser Stelle des Bytestroms einen Abschnitt sieht, und diese Daten bis dorthin als Einheit zu betrachten sind. Es ist ein wenig wie das Flushen eines Puffers zu sehen, jedoch hat der Benutzer nicht die Gewissheit, daß es auch wirklich passiert. TCP kann z.B. noch immer Daten ansammeln, um sie günstiger über das Netzwerk zu schicken (Stichwort *Nagle-Algorithmus*, siehe auch Socket-Option TCP_NODELAY, 8.1.11).

Urgent Pointer und Out-of-band-Daten
Das letzte Konzept, das noch etwas genauer beleuchtet werden soll, ist das des Urgent Pointers. Es kann bei einer Verbindung nötig sein, wichtige Daten als solche kenntlich zu machen. Dies könnte bei einer Telnet-Sitzung zum Beispiel das Drücken von Strg + C sein, das schnellstmöglich zum Server übertragen werden soll.

Solche dringenden Daten werden ganz normal im TCP Datenstrom verpackt, jedoch wird zum einen ein Push durchgeführt, das heißt das TCP wird aufgefordert sämtliche Puffer augenblicklich rauszuschreiben, zum anderen das URG-Bit gesetzt und der Urgent Pointer auf das erste Oktett nach den wichtigen Daten gesetzt. Der Urgent Pointer versteht sich als Offset innerhalb des aktuellen Datagramms, d.h. die Sequenznummer des entsprechenden Oktetts ergibt sich aus der Summe der Sequenznummer des Datagramms plus den Wert des Urgent-Feldes. Leider ist nicht klar, wie viele Oktetts zu diesen wichtigen Daten gehören sollen. Die Implementierung der BSD-Sockets geht soweit, daß nur das letzte Oktett als urgent angesehen wird. Die Daten müssen auch nicht zwingend im Datagramm enthalten sein, das RFC fordert, daß in jedem Datagramm, bis einschließlich das, in dem die Daten drin sind, das URG-Bit und den Urgent Pointer entsprechend gesetzt haben müssen, sodaß im Idealfall *sofort* bekannt wird, daß etwas außergewöhnliches passiert ist, auch wenn die Information, *was* es genau ist, erst später (oder ggf. niemals, wenn die Verbindung stirbt) kommt.

Das RFC schreibt vor, daß der Benutzer auf der Gegenseite von der TCP-Implementierung entsprechend in Kenntnis zu setzen ist. Im Kapitel zu den Socket-Funktionen send und recv wird der Mechanismus genauer erklärt (7.11 bzw. 7.9)).

4.2. UDP – das User Datagram Protocol

4.2.1. Überblick über das User Datagram Protocol

Das User Datagram Protocol bietet ebenfalls Multiplexing an, indem es das Adressierungskonzept des Internet Protocols um Port-Nummern erweitert. Während bei TCP die Übertragung als zuverlässig gilt, ist sie es bei UDP nicht. Geht auf der Reise durch das Netzwerk ein UDP-Datagramm verloren, so ist es einfach weg. UDP ist in RFC 768 [31] beschrieben.

Der Lohn für das einfachere Konzept, das völlig ohne Handshakes und Bestätigungen auskommt, ist eine höhere Performanz und deutlich weniger Protokoll-Overhead. Zwar ist der Header nur wenig kleiner (8 Oktetts anstatt 20), dafür jedoch entfällt sehr viel Aufwand für die Implementierung. Einen UDP Stack für einen vorhandenen IP-Stack zu entwickeln ist sehr einfach.

Durch die unzuverlässige Übertragung ist das Einsatzgebiet für das UDP ein anderes. UDP wird häufig für Multimedia-Anwendungen verwendet, bei denen die Übertragung zeitkritisch, jedoch nicht zwangsläufig verlustfrei sein muß. Bei einem Videostream mit ein paar Mbit/s ist es wenig schlimm, wenn ein paar Informationen wegfallen, mal ein Frame fehlt, oder der Ton stolpert. Sollen solche Übertragungen in Echtzeit erfolgen, so sind die Nachrichten sowieso uninteressant, sobald sie veraltet sind. Ein schönes Beispiel dafür ist das NTP (Network Time Protocol) – wenn hierbei ein Datagramm verloren geht, dann ist es sowieso nutzlos. NTP beachtet die Laufzeit der Datagramme, und durch eine Retransmission wäre diese völlig unnachvollziehbar anders.

Ebenfalls eingesetzt wird es im Domain Name Service, der in Kapitel 6 ausführlich behandelt wird.

Eine andere Anwendung ist zum Beispiel TFTP (Trivial File Transfer Protocol). Hier müssen die Daten korrekt übertragen werden, jedoch soll die Implementierung simpel gehalten werden. TFTP wird gerne für Datenübertragungen zu relativ „dummen“ Geräten genutzt, wie z.B. Rechner in der Bootphase. Damit die Datenübertragung fehlerfrei und zuverlässig abgewickelt werden kann, ist eine Bestätigungs-Mechanik wie bei TCP nötig, jedoch kann sie (auf Kosten des Datendurchsatzes) sehr einfach erfolgen. Das TFTP wird in Kapitel 5.4 genauer besprochen.

0 … 15	16 … 31
Source Port	Destination Port
Length	Checksum
data	

Abbildung 4.3.: User Datagram Protocol Header

Source Port
Der Quellport des Datagramms, kann Werte von 0 bis 65536 annehmen.

Destination Port
Der Zielport des Datagramms.

Length
Die Länge des Datagramms, inklusive Header. Auf grund der festen Größe eines UDP Headers ist keine Angabe eines Datenoffsets wie etwa bei TCP nötig. Die Länge der Nutzdaten errechnet sich immer aus *Length* - 8.

Checksum
Wie bei TCP auch wird dem UDP-Datagramm zur Berechnung der Checksumme ein Pseudo-Header vorangestellt, der genauso aufgebaut ist:

0 … 15	16 … 31	
Source Address		
Destination Address		
zero	PTCL	UDP Length

Abbildung 4.4.: UDP Pseudo Header

Das Feld *PTCL* enthält die Protocol Number von UDP, die gleiche wie im entsprechenden Feld es IP-Headers. Die *UDP Length* ist die Länge des UDP-Datagramms, also Headerlänge plus Länge der Nutzdaten. Die Quell- und Zieladresse werden ebenfalls aus dem IP-Header übernommen. Die Berechnung der Checksumme erfolgt gemäß [18].

4.3. ICMP – das Internet Control Message Protocol

4.3.1. Überblick über das Internet Control Message Protocol

Die Einordnung des ICMP in die Transportschicht erfolgt nur deshalb, weil ICMP-Datagramme ebenfalls IP als darunterliegende Schicht einsetzen. ICMP zählt genauso als Bestandteil der Internetschicht, und muß von jeder IP-Implementierung ebenfalls unterstützt werden.

Mittels ICMP können Fehlersituationen beschrieben und gemeldet werden. Beispielsweise wenn ein Datagramm nicht zustellbar ist, weil es das DF-Flag gesetzt hat, aber zu groß für die Übertragung durch ein bestimmtes Netz ist (vgl. 3.5.2). In diesem Fall wird das Datagramm verworfen, und der Absender erhält eine ICMP Nachricht, die dies mitteilt. Um Schleifenbildung vorzubeugen, werden beim Verlust von ICMP-Datagrammen keine weiteren Meldungen vorgenommen.

Das Format einer ICMP-Nachricht hängt vom Typ der Nachricht ab. Viele geben noch den IP-Header sowie 64 Bit der Nutzdaten mit an, damit höhere Protokollschichten den Fehler an die passende Anwendung melden können. Gemeinsam haben sie alle, daß sie mit diesen vier Oktetts beginnen:

0	15 16	31
Type	Code	Checksum

Abbildung 4.5.: Internet Control Message Protocol Header

Type und **Code**
Diese beiden Felder beschreiben die Art und den genauen Umstand des Fehlers. Eine Übersicht bietet die folgende Tabelle, eine genaue Beschreibung wird in den folgenden Abschnitten vorgenommen.

Type	Code	Description
0		Echo Reply
3		Destination Unreachable
	0	net unreachable
	1	host unreachable
	2	protocol unreachable
	3	port unreachable
	4	fragmentation needed and DF set
	5	source route failed
4		Source Quench
5		Redirect
	0	Redirect datagrams for the Network
	1	Redirect datagrams for the Host
	2	Redirect datagrams for the Type of Service and Network
	3	Redirect datagrams for the Type of Service and Host
8		Echo
11		Time Exceeded
	0	time to live exceeded in transit
	1	fragment reassembly time exceeded
12		Parameter Problem
13		Timestamp
14		Tiemstamp Reply
15		Information Request
16		Information Reply

Checksum
Die Checksumme wird über das komplette Datagramm inklusive Header gemäß [18] berechnet.

4.3.2. Destination Unreachable

0	15 16	31
Type	Code	Checksum
unused		
Internet Header + 64 bits of Original Data Datagram		

Abbildung 4.6.: ICMP Destination Unreachable

Eine Destination Unreachable-Nachricht wird erzeugt, wenn ein Datagramm sein eigentliches Ziel nicht erreichen konnte. Es werden hierbei sechs Fälle unterschieden:

0. *net unreachable*: das Zielnetz ist nicht erreichbar, zum Beispiel führ kein Router dorthin, oder eine Routing-Tabelle ist falsch
1. *host unreachable*: der Zielrechner existiert im Zielnetz nicht, beispielsweise antwortet keiner auf die entsprechenden ARP Requests
2. *protocol unreachable*: im Falle eines Gateways ist das Zielprotokoll nicht implementiert, wodurch eine Weiterleitung unmöglich wird
3. *port unreachable*: der Zielport ist keiner Anwendung zugeordnet, also „nicht offen“
4. *fragmentation needed and DF set*: das Datagramm ist für ein Netz zu groß, darf aber nicht fragmentiert werden, weil im IP-Header das DF (*don't fragment*) Bit gesetzt ist
5. *source route failed*: die durch IP-Optionen vorgegebene Route kann nicht eingehalten werden

Von der Destination Unreachable-Nachricht wird der Fall *fragmentation needed and DF set* genutzt, um die Path MTU einer Verbindung zu bestimmen. Dabei werden laufend Datagramme mit wechselnder Größe und gesetztem DF-Flag versendet, bis an die Grenze, an der sie gerade nicht mehr verworfen werden, „ertastet“ wurde.

Die anderen Nachrichten kommen oft vor, wenn Netze instabil oder falsch konfiguriert sind, oder man sich bei den Zieladressen bzw. Portnummern geirrt hat.

4.3.3. Time Exceeded

0	15 16	31
Type	Code	Checksum
unused		
Internet Header + 64 bits of Original Data Datagram		

Abbildung 4.7.: ICMP Time Exceeded

Wenn beim Verarbeiten eines Datagramms ein Zeitlimit überschritten wird, wird eine Time Exceeded-Nachricht erzeugt. Es werden zwei Fälle unterschieden:

0. *time to live exceeded in transit*: Das TTL-Feld des IP-Headers ist auf 0 dekrementiert worden, und der bearbeitende Router oder Gateway hat das Datagramm verworfen
1. *fragment reassembly time exceeded*: Wenn der Zielhost die Reassemblierung eines fragmentierten Datagramms innerhalb einer bestimmten Zeit nicht vornehmen konnte, weil Fragmente fehlen, so wird das Datagramm verworfen

Beim Traceroute-Program wird von dieser Nachricht Gebrauch gemacht, indem z.B. UDP Datagramme mit inkrementell wachsendem Wert für das TTL-Feld rausgesendet werden, und durch Aufsammeln der Time Exceeded-Meldungen die Route des Datagramms nachvollzogen werden kann.

4.3.4. Parameter Problem

0	15 16	31
Type	Code	Checksum
Pointer	unused	
Internet Header + 64 bits of Original Data Datagram		

Abbildung 4.8.: ICMP Parameter Problem

Die Parameter Problem-Nachricht wird erzeugt, wenn ein verarbeitender Router, Gateway oder Host auf ein Problem stösst, weil ein Header-Feld einen ungültigen Wert enthält. Falls in diesem Falle das Datagramm verworfen werden muß, so wird die Nachricht an den Absender versendet.

Pointer
Dieses Feld enthält den Offset des „kaputten“ Oktetts im betreffenden Datagramm.

4.3.5. Source Quench

0	15 16	31
Type	Code	Checksum
unused		
Internet Header + 64 bits of Original Data Datagram		

Abbildung 4.9.: ICMP Source Quench

Wenn ein Router oder Gateway keinen Speicher mehr zur Verfügung hat, um einkommende Datagramme aufzunehmen, und deshalb selbige verwerfen muß, so kann er dem Sender eine Source Quench-Nachricht senden. Dies ist eine Aufforderung, langsamer zu senden, und darf auch gesendet werden, wenn die Datagramme zu schnell eintreffen (also auch ohne dadurch resultierenden Verlust). Der Sender sollte dann die Geschwindigkeit reduzieren, und darf sie anschließend solange erhöhen, bis er erneut Source Quench-Nachrichten erhält.

Die Nachrichten dürfen gesendet werden, *bevor* es zu Problemen kommt, d.h. es kann durchaus vorkommen, daß ein Datagramm, das eine Source Quench-Nachricht ausgelöst hat, noch erfolgreich ausgeliefert wird.

4.3.6. Redirect

0	15 16	31
Type	Code	Checksum
Gateway Internet Address		
Internet Header + 64 bits of Original Data Datagram		

Abbildung 4.10.: ICMP Redirect

Redirect-Nachrichten werden versandt, wenn ein Router oder Gateway auf einer Route liegt, die abgekürzt werden kann, indem der sendende Host direkt an einen anderen Router oder Gateway

senden kann. In diesem Falle sendet der Router eine Redirect-Nachricht mit Hinweis auf den besseren Hop an den Sender zurück.

Es werden die folgenden Codes unterschieden:

0. *Redirect datagrams for the Network*: alle Datagramme an das Zielnetz können günstiger über den angegebenen Hop erreicht werden
1. *Redirect datagrams for the Host*: Datagramme an den Zielhost können günstiger über den angegebenen Hop erreicht werden
2. *Redirect datagrams for the Type of Service and Network*: unter Einhaltung des angegebenen Type of Service kann das Zielnetz über den angegebenen Hop günstiger erreicht werden
3. *Redirect datagrams for the Type of Service and host*: unter Einhaltung des angegebenen Type of Service kann der Zielhost über den angegebenen Hop günstiger erreicht werden

Wenn Datagramme eine Source Route enthalten, so werden diese Nachrichten nie versandt, auch wenn es eine bessere Route gibt. Der Aufbau einer Redirect-Nachricht ist in Abbildung 4.10 zu sehen.

Gateway Internet Address
Adresse des besseren Hops zum Erreichen des Ziels.

4.3.7. Echo und Echo Reply

0	15	16 31
Type	Code	Checksum
Identifier		Sequence Number
Data...		

Abbildung 4.11.: ICMP Echo / Echo Reply

Die Echo und Echo Reply-Nachrichten dienen nur zu Diagnosezwecken, sie werden nicht im Fehlerfall oder automatisch produziert. Eine Echo-Nachricht wird vom Zielhost durch eine Echo Reply-Nachricht beantwortet.

Beide Typen kennen keine Unterarten, und haben denselben, in Abbildung 4.11 gezeigten, Aufbau.

Identifier
Ein Wert, um die Datagramme zuzordnen, kann 0 sein.

Sequence Number
Eine Sequenznummer, um Datagramme zu unterscheiden, kann 0 sein.

Data Beliebige Daten.

Üblicherweise wird der *Identifier* fest gewählt, wie eine Portnummer bei TCP oder UDP, um eine Serie von Nachrichten zusammenzufassen. Die Sequenznummer wird dabei inkrementiert, um die einzelnen Nachrichten innerhalb der Gruppe zu unterscheiden.

Die Daten, die in einer Echo-Nachricht empfangen wurden, müssen – ebenso wie Identifier und Seuqenznummer – in der Antwort übernommen werden. Sie können dazu dienen, Fehler in der Übertragung festzustellen (der Sender weiß ja, was er zurückerwartet).

Anwendung finden Echo und Echo Reply-Nachrichten im ping-Programm, mit dessen Hilfe die Anwesenheit und Erreichbarkeit eines Hosts geprüft werden kann. Der Name „ping“ ist in Anlehnung an das Geräusch, das bei Sonar-Peilungen entsteht, gewählt worden [32].

4.3.8. Timestamp und Timestamp Reply

0	15 16	31
Type	Code	Checksum
Identifier		Sequence Number
Originate Timestamp		
Receive Timestamp		
Transmit Timestamp		

Abbildung 4.12.: ICMP Timestamp / Timestamp Reply

Mit Hilfe der Timestamp-Nachrichten lassen sich Informationen über die Laufzeit der Datagramme durch ein Netzwerk gewinnen. Diese Funktion hat gewisse Ähnlichkeiten mit der IP-Option *Internet Tiemstampt* (vgl. 3.5.3), jedoch fügt nur der Zielhost weitere Informationen ein.

Die Funktionen der in Abbildung 4.12 bezeichneten Felder sind:

Identifier
Der Identifier dient zum Zuordnen der einzelnen Timestamp und Timestamp Reply-Nachrichten, kann 0 sein.

Sequence Number
Diese Sequenznummer wird verwendet, um die einzelnen Nachrichten einer Gruppe zu unterscheiden, kann 0 sein.

Originate Timestamp
Zeitpunkt, zu dem der Sender das Datagramm vor dem Senden zuletzt angefasst hat.

Receive Timestamp
Zeitpunkt, zu dem der Empfänger das Datagramm erstmals angefasst hat.

Transmit Timestamp
Zeitpunkt, zu dem der Empfänger das Datagramm vor dem Rücksenden zuletzt angefasst hat.

Die Zeitpunkt sind hierbei als Millisekunden seit Mitternacht, UTC, zu verstehen. Wenn diese Maßeinheit nicht zur Verfügung steht, so kann eine beliebige andere Zeit eingesetzt, und dies durch das Setzen des höchstwertigen Bits angezeigt werden.

4.3.9. Information Request und Information Reply

Diese beiden Nachrichten sind veraltet und werden nicht mehr eingesetzt. Sie dienten früher zum Herausfinden der IP-Adressen für Hosts, die nach dem Bootvorgang ihre eigene Adresse nicht kannten. Neuere Methoden sind BOOTP [33] und DHCP [34].

Der Sender setzt in der Quell- und Zieladresse [des IP-Headers] den Netzanteil der Adresse auf 0 (z.B. 0.0.0.32 als Quelladresse); der Empfänger der die Anfrage beanwortet füllt diese Informationen aus.

5. Die Anwendungsschicht

In diesem Kapitel sollen ein paar Protokolle der Anwendungsschicht beispielhaft vorgestellt werden. Zum einen dienen sie hier als Fallstudie, wie man so ein Protokoll implementieren könnte, zum anderen kann man recht gut erste Gehversuche mit der Netzwerkprogrammierung unternehmen, wenn man einen Server bzw. einen Client zum Testen hat, der nachweislich korrekt funktioniert. Dies ist natürlich umso leichter, je verbreiteter und „offizieller“ ein Protokoll ist.

Die Auswahl ist nicht zufällig. Die drei auf TCP aufbauenden Protokolle zeigen verschiedene Aspekte: FTP hat zwei parallele Verbindungen, HTTP ist 8-bit-fest obwohl Daten inline übertragen werden, und für POP3 ist es als Anfänger sehr leicht einfache Tools zu schreiben.

Als Kontrast dazu gibt es ein Protokoll, das UDP einsetzt. Hierbei wird die Zuverlässigkeit, die TCP bietet, auf einfache Weise nachgebildet, indem eigene Vorkehrungen zur Erkennung und erneuten Anforderung verlorengegangener Pakete getroffen werden.

Die drei Protokolle auf TCP sind komplett textbasiert, und können deshalb sehr einfach mit `telnet` „gesprochen“ werden. So kann man sich beispielsweise durch einen einfachen Aufruf von `telnet www.zotteljedi.de 80` mit dem Webserver meiner Homepage verbinden, um ihn beispielsweise nach der letzten Änderung der Index-Seite zu befragen. Dies ist auch eine sehr gute Übung, um mit den Protokollen vertraut zu werden, bevor man versucht sie zu implementieren.

An dieser Stelle sei angemerkt, daß Textzeilen im Netzwerk üblicherweise durch die Zeichenfolge $\langle CR \rangle \langle LF \rangle$ terminiert werden, das ist Carriage Return (0x0d, 13 dezimal, '\r' als Ersatzdarstellung in C) und Line Feed (0x0a, 10 dezimal, '\n' in C). Ein Telnet-Client erzeugt diese beispielsweise, während `netcat` die Zeilen mit einfachem $\langle LF \rangle$ abschliesst.

Es ist in jedem Fall dringend empfohlen, das zu einem Protokoll gehörige RFC genau zu lesen, wenn das Protokoll implementiert werden soll. Um zum einen eine bessere Orientierung zu ermöglichen, aber vor allem um eine allzu holprige Übersetzung zu vermeiden, wurden einige der Begriffe nicht übersetzt. Begrifflichkeiten wie *reply code* oder *authentication* lassen sich oftmals auch nur unter Informationsverlust übersetzen, weil es keine deutsche Entsperchung gibt (man denke nur an *security* und *safety*, die beide auf *Sicherheit* gemappt werden müssten). Dies ist ebenfalls einer der Gründe, warum RFCs üblicherweise nicht in andere Sprachen übersetzt werden.

5.1. HTTP

Das Hypter-Text Transfer Protocol verwendet den TCP-Port 80. Die aktuelle Version ist 1.1 [12], Vorgänger dazu war 0.9, das jedoch nie offizieller Standard wurde. Viele Implementierungen setzen HTTP/1.0 ein, das ebenfalls durch keinen Standard definiert ist. Eingesetzt wird HTTP zur Übertragung von Daten zwischen Webserver und Webclients (z.B. Webbrowsern). Eine Webseite besteht üblicherweise aus einer Seite, auf der Bilder oder andere Dokumente eingebunden sind. Um diese ebenfalls zu übertragen, muß der Client weitere Anfragen senden. Bei HTTP/1.1 ist das Konzept der persistenten Verbindungen dazugekommen, das nicht länger den Aufbau einer neuen Verbindung für jede einzelne Ressource erfordert, wie es bei HTTP/0.9 nötig war.

Man unterscheidet primär zwischen zwei verschiedenen Arten von Übertragungen, Requests und Responses. Ein Reqeust ist eine Anfrage, beispielsweise die Anforderung einer Seite. Der Response ist dann die Antwort darauf, oder aber eine Fehlermeldung. Sowohl Requests als auch Responses können aus zwei Teilen bestehen, einem Header und einem Body. Der Header besteht stets aus Textzeilen, gefolgt von einer leeren Zeile. Diese Textzeilen enthalten ausschliesslich Zeichen des ASCII-Zeichensatzes, auch als *us-ascii* oder *7-bit-ASCII* bezeichnet. Die erste Zeile nimmt eine Sonderrolle ein, alle weiteren sind gleichberechtigt und können in beliebiger Reihenfolge auftreten.

In der ersten Zeile steht die HTTP-Methode, ggf. mit einem URI, gefolgt von dem Protokoll mit Versionsangabe. HTTP folgende Methoden:

OPTIONS	Auskunft über unterstützte Optionen
GET	Anforderung einer Ressource (Header und Body)
HEAD	wie GET, jedoch nur der Header wird versendet
POST	der Server soll den Body des Requests verarbeiten
PUT	der Server soll den Body speichern
DELETE	eine Ressource auf dem Server soll gelöscht werden
TRACE	das Request wird zurückgesendet (Loopback)
CONNECT	reserviert für Proxy-Server

Die dem Request folgenden Zeilen bestehen alle aus einem Schlüsselwort, gefolgt von einem Doppelpunkt und einer Zeichenkette (die auch aus Ziffern bestehen kann), abhängig vom Schlüsselwort.

General Header	Request Header	Response Header	Entity Header
Cache-Control	Accept	Accept-Ranges	Allow
Connection	Accept-Charset	Age	Content-Encoding
Date	Accept-Encoding	ETag	Content-Language
Pragma	Accept-Language	Location	Content-Length
Trailer	Authorization	Proxy-Authenticate	Content-Location
Transfer-Encoding	Expect	Server	Content-MD5
Upgrade	From	Vary	Content-Range
Via	Host	WWW-Authenticate	Content-Type
Warning	If-Match		Expires
	If-Modified-Since		Last-Modified
	If-None-Match		extension-header
	If-Range		
	If-Unmodified-Since		
	Max-Forwards		
	Proxy-Authorization		
	Range		
	Referer		
	TE		
	User-Agent		

Bei HTTP/1.1 **muß** der Host-Header im Reqeust enthalten sein. Dieser bezeichnet den Host, von dem die Ressource angefordert wird. Dies ist nötig, weil auf einem Server möglicherweise mehrere *virtuelle Hosts* gelegen sein können. Der Date-Header muß in jedem Response vorhanden sein.

Mit dem Connection-Header wird festgelegt, ob die Verbindung nach der Übertragung des Responses geschlossen ("Close"), oder offen gehalten werden soll ("Keep-Alive"). Ist dieser Header im Request nicht vorhanden, so wird bei HTTP/1.1 Keep-Alive angenommen, sofern der Client eben diese Version angemeldet hat. Der Client kann auch eine andere Version in der ersten Zeile des Requests anfordern.

Wenn HTTP/0.9 gesprochen wird, so enthalten die Responses keinen Header, sondern nur den Body, denn HTTP/0.9 kannte das Konzept der Response-Header nicht, und folglich würden HTTP/0.9-Clients die Header ebenfalls als Teil der angeforderten Ressource interpretieren.

Durch den Einsatz von Keep-Alive wird das Header-Feld Content-Length besonders interessant, denn dieses gibt die Länge des Bodys in Oktetts an, und ist für den Client der einzige Hinweis, wie weit er lesen darf, bevor er auf den nächsten Header stösst.

Das Feld Content-Type gibt die Art des Bodys an, bekannte Werte sind z.B.

text/plain	text/html	application/octet-stream
image/gif	image/jpeg	image/png
image/tiff	video/quicktime	video/mpeg

Und ein paar Dutzend andere. Im Folgenden können unmöglich alle Header erklärt werden, stattdessen soll anhand eines Beispiels gezeigt werden, wie HTTP-Meldungen aussehen, quasi am lebenden Server gezeigt.

```
C:\>telnet www.freebsd.org 80
HEAD /index.html HTTP/1.1
Host: www.freebsd.org
Connection: Close

HTTP/1.1 200 OK
Date: Sat, 02 Jul 2005 13:37:47 GMT
Server: Apache/1.3.x LaHonda (Unix)
Last-Modified: Wed, 29 Jun 2005 22:27:27 GMT
ETag: "26fb58-835b-42c3204f"
Accept-Ranges: bytes
Content-Length: 33627
Connection: close
Content-Type: text/html
X-Pad: avoid browser bug
```

Wäre die Methode GET gewesen, so wären noch 33627 Byte HTML-Code gefolgt, der Header ist jedoch bis auf das Fehlen des Bodys exakt identisch. Dies ist insbesondere bei *CGI-Programmen* interessant. Dies sind externe Programme, die vom Server gestartet werden. Die erzeugte Ausgabe wird dann an den Client zurückgeliefert. Oftmals ist es nicht die Ausgabe, die interessant ist, sondern die Seiteneffekte, die das Programm auslöst. Um die Anzahl der Oktetts, die die Ausgabe beinhaltet, festzustellen muß jedoch das Programm mit eventuellen Seiteneffekten ausgeführt werden – sofern die Anzahl der Oktetts nicht a priori bekannt ist.

Ein CGI-Programm kann alles erdenkliche tun. Nett sind z.B. immer Spielereien mit dem Traceroute-Programm, so könnte der Server die Route von sich aus zum Client ausgeben. Auch Web-Shops, Chat-Systeme und andere Webanwendungen machen heftigen Gebrauch dieser Technik.

Eine einfache Anwendung, die HTTP benutzt und sich mit dem Wissen aus diesem Buch leicht erstellen lässt, könnte beispielsweise die Header bestimmter Seiten in regelmäßigem Intervall abfragen, um den Benutzer bei einer Neuerung darauf aufmerksam zu machen. Der Last-Modified-Header gibt genau

diese Information an, wann die angeforderte Ressource zuletzt geändert wurde. An dieser Stelle sollte noch darauf aufmerksam gemacht werden, daß eine *Ressource* nicht notwendigerweise eine *Datei* sein muß. Der URI kann auch etwas bezeichnen, das der Server zur Laufzeit generiert, im Prinzip also etwas, das er sich „ausgedacht“ hat. Es ist auch üblich, daß Seiten komplett aus Datenbanken erzeugt werden. Dies ist bei großen Datenmengen durchaus eine gute Idee, jedoch tun sich manche Suchmaschinen damit schwer, was zu einer fehlerhaften Indexierung führt. Die besten Inhalte bringen nicht viel, wenn sie niemand finden kann.

Eine sehr willkommene Neuerung in HTTP/1.1 ist die Einführung der *Ranges*. Ein Client kann beispielsweise nur Teile einer großen Datei vom Server anfordern (wenn dieser es unterstützt). Auf diese Weise können abgerissene Downloads fortgesetzt werden, aber auch Download-Accelerator bedienen sich dieses Features. Sie öffnen mehrere Verbindungen gleichzeitig zum Server, wobei auf jeder ein Teil der Datei übertragen wird. Damit umgeht man die Limitierung der Bandbreite pro Verbindung (die ist außerhalb der HTTP Spezifikation eine Sache der Server-Implementierung). Server-Betreiber sind dem durch eine Limitierung der Verbindungszahl von einer IP-Adresse begegnet, was NAT-Betreiber gelegentlich sauer macht. Der Server in obigem Beispiel kündigt die Fähigkeit durch das Schlüsselwort `Accept-Ranges` an. Bytes ist die derzeit einzige Einheit, die definiert ist, aber der Server darf auch andere ankündigen, vielleicht Sekunden für einen Server, der Videos oder Audiodaten bereitstellt.

HTTP kennt noch einige Status-Codes, die durch dreistellige Ziffernfolgen, gefolgt von einer textuellen Meldung, repräsentiert werden. Die erste Ziffer gibt an, um welche Art von Statusmeldung es sich handelt. Die mit einer 1 beginnen Meldungen sind *Informational*; man bekommt sie in der Regel nie zu sehen. Die Meldungen mit einer 2 als erste Ziffer sind *Successful*, wenn das Request also erfolgreich erfüllt werden kann. Eine administrative Gruppe ist die mit 3 beginnende *Redirection* Klasse. Mit Hilfe dieser Meldungen können beispielsweise die Robots von Suchmaschinen geleitet werden, wenn eine Seite verzogen ist, und unter einer neuen Adresse zu finden ist. Die am häufigsten gesehenen Fehlermeldungen stammen aus der 4xx Gruppe: *Client Error*, das Gegenstück dazu bilden die *Server Error*, deren Fehlercode mit einer 5 beginnt.

100	Continue	400	Bad Request
101	Switching Protocols	401	Unauthorized
200	OK	402	Payment Required
201	Created	403	Forbidden
202	Accepted	404	Not Found
203	Non-Authoritative Information	405	Method Not Allowed
204	No Content	406	Not Acceptable
205	Reset Content	407	Proxy Authentication Required
206	Partial Content	408	Request Timeout
300	Multiple Choices	409	Conflict
301	Moved Permanently	410	Gone
302	Found	411	Length Required
303	See Other	412	Precondition Failed
304	Not Modified	413	Request Entity Too Large
305	Use Proxy	414	Request-URI Too Long
306	(Unused)	415	Unsupported Media Type
307	Temporary Redirect	416	Requested Range Not Satisfiable
		417	Expectation Failed
		500	Internal Server Error
		501	Not Implemented
		502	Bad Gateway
		503	Service Unavailable
		504	Gateway Timeout
		505	HTTP Version Not Supported

5.2. FTP

Das File Transfer Protocol [48] dient dem Datenaustausch zwischen Hosts über TCP/IP. Interessant hierbei ist, daß nicht nur eine Datenverbindung verwendet wird, sondern zeitweise zwei Verbindungen existieren: eine zur Steuerung und eine zur Datenübertragung.

Ein FTP-Server lauscht üblicherweise auf dem well-known Port 21. Das RFC ist ein tolles Beispiel für eine gewachsene Spezifikation, die heutzutage nur zu geringen Teilen umgesetzt wird. So wird beispielsweise festgelegt, daß Daten in verschiedenen Formaten, Strukturen und Datenformaten vorliegen bzw. übertragen werden können, sowie verschiedene Modi für die Übertragung selbst existieren können. Eine Zusammenstellung dieser Optionen bildet die folgende Tabelle:

Format Control	Data Types	Data Structures	Transmission Modes
non print	ASCII	file-structure	Stream Mode
telnet format controls	EBCDIC	record-structure	Block Mode
carriage control (ASA)	Image	page-structure	Compressed Mode
	Local		

In der Praxis unterstützen die meisten FTP-Server den Großteil dieser Optionen nicht. Das RFC schreibt vor, daß der Datentyp *ASCII* und die Datenstrukturen *file* und *record* unterstützt werden müssen, alle anderen sind freiwillig. Die meisten Server werden jedoch auch den Datentyp *image* unterstützen, da dieser zur Übertragung von Binärdateien verwendet wird. Dies ist gleichbedeutend mit dem Datentyp *local* und einer Bytegröße von 8 Bit. Als Übertragungsmodus wird in der Regel der *stream mode* eingesetzt.

FTP kennt weiterhin eine Reihe von Befehlen, die auf der zu Port 21 bestehenden Kontrollverbindung übertragen werden. Mit ihnen kann auch ein Datentransfer eingeleitet werden, wozu dann eine zweite Verbindung aufgebaut wird. Hierbei unterscheidet man zwischen aktivem und passivem FTP. Beim aktiven FTP baut der Server eine Verbindung zum Client auf, beim passiven FTP umgekehrt der Client zum Server. Letzteres ist vor allem im Zusammenhang mit NAT (Network Address Translation, siehe 3.3) recht nützlich. Die genauen Parameter für die Verbindung werden mittels eines PORT-Befehls ausgehandelt.

Interessant ist, daß der Host für die Datenverbindung nicht der gleiche Host sein muß, der die Kontrollverbindung bedient. Somit kann ein Host mit einer langsamen Verbindung beispielsweise einen Datentransfer zwischen zwei Hosts mit schneller Verbindung einleiten, und damit sogar Daten von einem Server zu einem anderen Server befördern, ohne den Umweg über sich selbst nehmen zu müssen.

Die FTP-Kommandos werden in drei Gruppen unterteilt, die *Access Control Commands*, die *FTP Service Commands* und die *Transfer Parameter Commands*. Im Folgenden sollen die Befehle dieser Gruppen vorgestellt werden, wobei die *FTP Service Commands* auf Grund ihrer Fülle nur kurz in tabellarischer Form wiedergegeben werden.

Access Control Commands

USER
Legt den Benutzernamen für die Sitzung fest.

PASS
Gibt das zu USER gehörige Passwort an.

ACCT
Zusätzliche Angaben über einen *account*. Dies kann notwendig sein, und wird in diesem Fall durch den Rückgabewert von PASS angezeigt.

CWD
Ändert das aktuelle Arbeitsverzeichnis.

CDUP
Wechselt in das oberste Verzeichnis des Baums. Dies ist nützlich wenn ganze Verzeichnisbäume übertragen werden sollen, und die Wurzel des Baums nicht auf allen Systemen gleich heißt.

SMNT
Mountet ein zusätzliches Dateisystem. Wird in der Regel nicht unterstützt.

REIN
Reinitialisiert die Sitzung. Danach kann erneut eine Anmeldung über USER erfolgen.

QUIT
Beendet die Sitzung (wenn eine Datenübertragung im Gange ist, wird zuerst auf deren Beendigung gewartet, bevor die Sitzung geschlossen wird).

FTP Service Commands

RETR	Der Server sendet eine Datei zum Client (Datenverbindung)
STOR	Der Server akzeptiert eine Datei vom Client (Datenverbindung)
STOU	Wie STOR, jedoch wird ein einmaliger Dateiname als Ziel erzeugt
APPE	Hängt die Daten an eine bestehende Datei an (Datenverbindung)
ALLO	Alloziiert im Voraus den nötigen Platz für eine Datei
REST	Setzt den Dateizeiger an eine bestimmte Stelle („resume“)
RNFR	Rename From, der Quellname beim Umbenennen
RNTO	Rename To, der neue Zielname beim Umbenennen
ABOR	Bricht die Ausführung des vorhergehenden Kommandos ab
DELE	Löscht die Datei die angegebene Datei
RMD	Löscht das angegebene Verzeichnis
MKD	Erstellt das angegebene Verzeichnis
PWD	Gibt das aktuelle Arbeitsverzeichnis aus
LIST	Sendet ein Directory-Listing an den Client (Datenverbindung)
NLST	Sendet eine Liste der Dateinamen an den Client (Datenverbindung)
SITE	Ausführung systemspezifischer Befehle
SYST	Gibt das Betriebssystem des Servers aus
STAT	Sendet einen Statusbericht (Kontrollverbindung)
HELP	Hilfe zu einem der Befehle (Kontrollverbindung)
NOOP	Keine Seiteneffekte, erzeugt immer ein OK

Transfer Parameter Commands

PORT
Legt den Zielpunkt für eine vom Server initiierte Datenverbindung fest. Das Format des Befehls ist `PORT h1,h2,h3,h4,p1,p2`, wobei h1 das höchstwertigste Byte der Hostadresse ist, analog dazu p1 das höherwertige Byte der Portnummer.

PASV
Der Server wird in den passiven Modus versetzt und lauscht auf einem Port, damit der Client eine Datenverbindung dahin aufbauen kann. In der Antwort ist auch hier Hostadresse und Portnummer enthalten.

TYPE
Legt den Datentyp der Übertragung fest. Die gültigen Typen sind A für ASCII, E für EBCDIC, I für Image und L für Local. Die beiden ersten Typen können von einem weiteren Parameter für die Formatangabe gefolgt werden: N für non-print, T für Telnet und C für Carriage Control (ASA). Der

Typ Local muß von einem Parameter gefolgt werden, der die Bytegröße in Bit angibt. Üblicherweise unterstützen Server hier nur die Typen A und I.

STRU
Mit diesem Befehl wird Struktur der Übertragung eingestellt. Gültige Parameter sind F für File, R für Record und P für Page. Üblicherweise wird nur F unterstützt.

MODE
Der Modus der Übertragung kann ebenfalls eingestellt werden, und zwar mit S für Stream, B für Block und C für Compressed. Der Standardwert ist S, und meist auch der einzige der unterstützt wird.

Man kann bei vielen FTP-Clients für die Kommandozeile direkt Befehle an den Server senden, indem man der Befehlszeile ein bestimmtes Kommando voranstellt. Bei dem FTP-Client von Windows ist dies beispielsweise `literal`, bei BSD `quote` usw., mit `help` bekommt man in der Regel die Hilfe zu den (lokalen) Befehlen.

Jeder Meldung vom Server ist ein numerischer Antwortcode vorangestellt. Bei mehrzeiligen Antworten wird in der ersten Zeile der Antwortcode von einem Minus (-) gefolgt, in der letzten Zeile von einem Leerzeichen. Die Zeilen dazwischen können beliebig aussehen, es sei denn sie beginnen mit einer dreistelligen Zahl gefolgt von einem Leerzeichen; damit diese nicht mit dem Ende der Ausgabe verwechselt werden können, müssen sie durch den Server kenntlich gemacht werden, beispielsweise durch das Voranstellen von Leerzeichen.

Die Antwortcodes bestehen alle aus drei Ziffern und können in mehrere Gruppen unterschieden werden, die ihrerseits eine feine Gliederung besitzen, bis hin zu den individuellen Antwortcodes, die ihrer Fülle wegen dem RFC entnommen werden müsse. Im Folgenden sollen nur der Mechanismus zur Unterteilung in Gruppen vorgestellt werden.

Die erste Ziffer beschreibt die Gruppe, der die Antwort angehört. Es wird unterschieden zwischen positiven Rückmeldungen, die entweder auf vollständige Vorgänge hinweisen, oder weitere Informationen anfordern. Negative Rückmeldungen können entweder auf permanente oder temporäre Ursachen hinweisen. Die folgende Tabelle führt die fünf Grupppen auf:

1yz	Positive Antwort, weitere Meldungen werden folgen. Es ist nicht zulässig sofort mit weiteren Befehlen fortzufahren.
2yz	Positive Antwort, Vorgang wurde erfolgreich beendet.
3yz	Positive Antwort, jedoch sind weitere Kommandos zur Vervollständigung notwendig.
4yz	Negative Antwort, jedoch temporär und der gleiche Befehl kann zu späterem Zeitpunkt möglicherweise erfolgreich abgearbeitet werden
5yz	Negative Antwort, der Anwender soll den Befehl in der Form nicht erneut abzusetzen (*The user-process is discouraged from repeating the exact request (in the same sequence)*)

Die nächste Gliederungsstufe umfasst die Art des Fehlers, ob es sich um ein Syntaxproblem handelt, oder beispielsweise um Meldungen die ihre Ursache im Dateisystem haben (z.B. eine volle Festplatte). Hier wurden die Begriffe aus dem RFC übernommen ohne eine Übersetzung zu wagen.

x0z	Syntax: Syntaxfehler, syntaktisch korrekte Befehle die jedoch in keine funktionale Gruppe passen, oder unimplementeirte bzw. überflüssige Befehle
x1z	Information: Antworten auf Informationsanforderungen (z.B. status, help)
x2z	Connections: Antworten die sich auf die Kontroll- und Datenverbindung beziehen
x3z	Authentication and accounting: Antworten zum Login-Vorgang etc.
x4z	bisher unspezifiziert
x5z	File system: diese Antworten zeigen den Status des Dateisystems an

5.3. POP3

Mit dem Post Office Protocol Version 3 [46] kann der Inhalt einer Mailbox untersucht und verändert werden. Die häufigste Anwendung ist die Übertragung von E-Mail von einem Server, der meistens beim Mail-Provider steht, zu einem Client, der die Darstellung und Verarbeitung ermöglicht (oft als MUA bezeichnet, Mail User Agent). Der well-known Port von POP3 ist 110.

Das POP ist ziemlich einfach aufgebaut, und eine Sitzung läuft im Dialog zwischen Client und Server ab. Aktionen wie das Löschen von E-Mails werden erst dann tatsächlich vorgenommen, wenn die Verbindung zwischen Client und Server durch eine Ordnungsgemäße „Verabschiedung“ getrennt wurde; eine abgerissene Verbindung führt zu keiner Veränderung des Datenbestands.

In der folgenden Tabelle sind die Aktionen, die POP3 kennt und durchführen kann, mit ihren Befehlen aufgeführt. Die mit einem Stern markierten Befehle sind optional und müssen nicht zwingend implementiert werden, um RFC-konform zu bleiben.

Befehl	Wirkung
QUIT	baut eine Verbindung ab, Mailbox wird synchronisiert
STAT	gibt die Anzahl der vorhandenen Nachrichten und die Gesamtgröße an
LIST	gibt ein Listing der Nachrichten und der jeweiligen Größe an
RETR	überträgt die angegebene Nachricht zum Client
DELE	markiert die angegebene Nachricht zum späteren Löschen
NOOP	keine Aktion, immer positive Antwort vom Server
RSET	alle Markierungen werden zurückgesetzt
*TOP	gibt den Header plus die angegebene Anzahl Zeilen der Nachricht aus
*UIDL	gibt zu jeder Nachricht eine eindeutige ID aus
*USER	legt den Benutzernamen zum Zugriff fest
*PASS	legt das Passwort zum Zugriff fest
*APOP	Authentifikation mittels MD5 Digest

Manche dieser Befehle besitzen (teils optionale) Parameter. Die genaue Beschreibung kann leicht im RFC ersehen werden. Einige Befehle erzeugen Antworten, die über mehrere Zeilen gehen, wobei die letzte Zeile dann aus nur einem ’.’ besteht (ähnlich wie bei SMTP). Um für den Client das Parsen zu erleichtern, beginnen alle Statusmeldungen mit ’+OK’ für eine Erfolgsmeldung und mit ’-ERR’ im Falle eines Fehlers.

Wie bereits erwähnt, werden Daten erst manipuliert, wenn der Client sich ordnungsgemäß abgemeldet hat. Um dies zu realisieren, werden Nachrichten *markiert*, wenn sie gelöscht werden sollen. Sie sind für darauffolgende Befehle wie LIST nicht mehr sichtbar, werden jedoch erst nach einem QUIT tatsächlich gelöscht. Durch ein RSET können diese Markierungen aufgehoben werden; dies zeigt unmittelbar Wirkung. Erwähnenswert, obwohl naheliegend, ist die Tatsache, daß jede Nachricht während einer Sitzung eine Platznummer besitzt (1, 2, 3 ...), die auch durch das Markieren von Nachrichten nicht beeinflusst wird. Die Nachricht, die nach der Begrüßung als Nummer 5 bekannt war, ist auch nach dem Markieren von Nachricht 1 und 4 noch immer Nummer 5, auch wenn sie bei einem LIST in der dritten Zeile auftaucht!

Darüber hinaus kann mittels des Befehls UIDL jeder Nachricht ein eindeutiges Identifikationsmerkmal in Form einer Zeichenkette mit bis zu 70 Zeichen aus dem Zeichenvorrat 0x21 bis 0x7e (z.B. `whqtswO00WBw418f9t5JxYwZ`) zugeordnet werden. Diese Zeichenkette verändert sich auch über mehrere Sitzungen hinweg nicht. Allerdings ist es dem Server erlaubt, diese Zeichenkette aus einem Hash zu erzeugen, und Clients sind daher angehalten, mehrere Nachrichten mit der gleichen ID fehlerfrei bearbeiten zu können. Das legt bereits den Grundstein für einen clientseitigen ach-das-wird-schon-nie-passieren-Fehler in der Implementierung...

Interessant ist, daß die Benutzer/Passwort-Zugangskontrolle optional ist. Üblicherweise wird sie zwar angeboten, jedoch ist dies nicht zwingend. Ein Problem bei der Authentifikation über USER/PASS ist, daß das Passwort im Klartext über das Netzwerk verschickt wird, und damit möglicherweise mitgelesen werden kann (wie bei FTP auch). Um dieses Problem zu beheben, wurde der Befehl APOP eingeführt. Hierbei gibt der Server bereits im Banner (die Begrüßung beim Login) eine Zeichenkette im Format `<process-ID.clock@hostname>` an, die als eindeutiger String angesehen werden kan. Der Client hängt nun das Passwort an diese Zeichenkette an, berechnet einen Message Digest (in diesem Fall MD5 [47]) darüber, und gibt diesen zusammen mit dem Benutzernamen mit dem APOP-Befehl an. Durch die eindeutige Zeichenkette, die Teil des Digests ist, wird eine *replay-Attacke* ausgeschlossen, d.h. für eine spätere Sitzung kann dieser Digest nicht mehr verwendet werden, weil die Zeichenkette des Servers und damit die gesamte Hashsumme anders aussehen würde.

5.4. TFTP

Das Trivial File Transfer Protocol [35] dient der Übertragung von Dateien mittels UDP. Die Vorteile gegenüber dem „echten“ FTP liegen in der Einfachheit, sowohl in der Implementierung, als auch an den Anforderungen an die darunterliegenden Netzwerkschichten, denn ein UDP-Stack lässt sich sehr viel einfacher implementieren als ein TCP-Stack. Der well-known Port von TFTP ist 69.

Das Anwendungsgebiet von TFTP ist das einfache Übertragen von Dateien, ohne Zugriffsrechte oder Angabe von Dateilängen oder Directory-Listings. Ein TFTP-Client muß genau wissen, welche Datei er downloaden möchte. Ein Szenario dafür wäre ein Bootloader in einem eingebetteten System mit einem Netzwerk-Anschluß.

Bei Dateiübertragungen dürfen sich natürlich keine Fehler einschleichen, und UDP ist per Definition ein unzuverlässiges Protokoll. Aus diesem Grund implementiert TFTP eine eigene Form der Quittierung für Datagramme. Alle TFTP-Datagramme haben in den ersten zwei Oktetts einen Opcode, anhand dessen die Bedeutung des Datagramms ermittelt werden kann. Die Werte sind:

01	Read Request
02	Write Request
03	Data
04	ACK
05	Error

Mit einem Read Request (RRQ) meldet der Client seinen Wunsch, eine Datei downzuloaden, an. In Abbildung 5.1 ist das Layout dieses Datagramms zu sehen. Der Dateiname ist hierbei eine ASCII-Zeichenkette, die durch eine binäre 0 ('\0') terminiert ist, also ein ganz normaler C-String. Der Modus beschreibt das Datenformat der Übertragung, und ist eine der Zeichenketten "netascii", "octet" oder "mail". Letzterer stammt aus der Zeit, zu der SMTP noch nicht als universelles Mittel zur E-Mail-Übertragung eingesetzt werden konnte. Zu dieser Zeit hat man Mail mit so ziemlich allem übertragen, und dies hier war eine Variante. "Now considered obsolete". Das Format "netascii' ist für Textdateien gedacht, wobei die beteiligten Hosts die Daten ggf. konvertieren müssen. Dies sollte auch als Altlast betrachtet werden, heutige Implementierungen verwenden fast ausschließlich den Typ "octet", der eine Übertragung in 8-Bit-Bytes vorsieht. Die Daten werden dabei nicht interpretiert oder sonstwie verändert.

2 bytes	string	1 byte	string	1 byte
01	Filename	0	Mode	0

Abbildung 5.1.: RRQ Datagramm

2 bytes	string	1 byte	string	1 byte
02	Filename	0	Mode	0

Abbildung 5.2.: WRQ Datagramm

Das Write Request, Abbildung 5.2, ist identisch, nur mit geändertem Opcode.

Um laufende Verbindungen auseinander halten zu können, sieht TFTP das Konzept der Transfer Identifiers (TIDs) vor. Dabei wählt zum Aufbau der Verbindung jeder Teilnehmer zufällig einen Quellport. Das erste Datagramm wird hierbei vom zufällig gewählten Quellport des Clients an den well-known Port des Servers gesendet. Dieser sendet das erste Datenpaket bzw. ACK von einem ebenfalls zufällig gewählten Quellport an den Quellport des Clients. Von nun an werden nur diese gewählten Ports verwendet.

2 bytes	2 bytes	n bytes
03	Block #	Data

Abbildung 5.3.: DATA Datagramm

Im Erfolgsfall sendet der Server gleich das erste Datenpaket, in Abbildung 5.3 dargestellt. Die Blocknummern beginnen mit 0, und werden für jedes Datagramm um eins erhöht. Da das Feld 16 Bit breit ist, ergibt sich nach 65536 Datargammen das Problem des Überlaufs, das beachtet werden muß. Das RFC schreibt hier keine Behandlung vor.

Der Datenteil ist hierbei entweder 512 Oktetts, oder weniger, falls das Ende der Datei erreicht worden ist. Durch diesen Mechanismus kann der Client auch erkennen, wann die Übertragung abgeschlossen ist: immer dann, wenn $\neq$ 512 Datenoktetts enthalten sind. Sollte die Datei glatt aufgehen, so ist ein Datagramm ohne Nutzdaten zu senden.

2 bytes	2 bytes
04	Block #

Abbildung 5.4.: ACK Datagramm

Durch den Mechanismus der Blocknummern können einzelne Datagramme genau bezeichnet werden, ähnlich der Sequenznummern in TCP bezüglich der Oktetts. Wenn der Client ein Datenpaket erhalten hat, so quittiert er es durch das Senden eines ACKs (Abbildung 5.4), mit der entsprechenden Blocknummer des erhaltenen Datagramms. Um die Implementierung einfach zu halten, wird immer nur ein Datenblock gleichzeitig versendet. Das bedeutet, daß keine Datagramme im Speicher gehalten werden müssen, um sie auf Anforderung nachsenden zu können, wie es bei TCP der Fall ist.

Wenn nun ein Datagramm verloren geht, so laufen Client und Server in einen Timeout, und wiederholen ihr zuletzt gesendetes Datagramm, das kann ein Datenpaket oder ein ACK sein. Durch die Maßgabe, daß nur ein Datenpaket zur Zeit unterwegs sein kann, quittiert ein ACK auch alle vorhergehenden Datagramme – andernfalls hätte auch gar kein weiteres Datagramm gesendet werden dürfen.

Im Falle, daß der Client eine Datei zum Server überträgt, sind die Rollen entsprechend zu vertauschen; der Client sendet die Datenpakete, der Server die ACKs.

2 bytes	2 bytes	string	1 byte
05	Errorcode	ErrMsg	0

Abbildung 5.5.: ERROR Datagramm

Das letzte Datagramm, das TFTP kennt, ist das Error-Datagram (Abbildung 5.5). Der Fehlercode ist für die automatische Bearbeitung gedacht, die Fehlermeldung in textueller Form dient einer genaueren Beschreibung des Fehlerfalls, und muß vom Benutzer interpretiert werden. Die Fehlercodes sind:

0	Not defined, see error message (if any)
1	File not found
2	Access violation
3	Disk full or allocation exceeded
4	Illegal TFTP operation
5	Unknown transfer ID
6	File already exists
7	No such user

Ein Fehler führt meistens zum Abbruch der Übertragung. Es gilt anzumerken, daß Fehlermeldungen nicht quittiert werden, sie könnten auch heimlich verlorengehen.

6. Hostnamen, IP-Adressen und Portnummern

Die wenigsten Menschen haben ein Talent dafür sich IP-Adressen in numerischer Form zu merken. Tippfehler werden beim bloßen Drüberlesen nicht erkannt, und beim Austausch kann der Empfänger nichtmal einen Plausibilitätstest vornehmen (daß es sich bei www.speigel.de um einen Buchstabendreher handeln könnte, ist gar nicht so weit hergeholt). Kurzum: die Zahlen müssen Namen bekommen.

Eine einfache Zuordnung in einer Textdatei ist zwar eine gängige Lösung (UNIX-Systeme kennen `/etc/hosts`, Windows-Systeme `%SystemRoot%\system32\drivers\etc\hosts`), jedoch für größere Bestände indiskutabel. Abgesehen davon macht es keinen Spaß, solche Listen zu pflegen, schon gar nicht auf mehreren Systemen.

Stattdessen hat man einen Service ersonnen, der servergestützt eine zentrale Anlaufstelle bietet, aber dennoch dezentral aufgebaut ist, sodaß kein System alle Adressen kennen muß (bei 4 Milliarden verbietet sich das auch irgendwo von selbst). Dieses Domainname-System wird im folgenden Kapitel vorgestellt, deren praktische Anwendung aus Sicht des Programmierers in den darauf folgenden.

Ein weiteres Vorkommen „magischer Zahlen“ sind die Portnummern. So sind die Ports bis 1024 zwar „well-known“, was jedoch noch lange nicht heißt (oder heißen muß), daß jeder sie auswendig kennt. Die häufig vorkommenden wie 80 für HTTP oder 21 für FTP sind sicher den meisten bekannt, jedoch wissen die wenigsten, was sich hinter 515 oder 540 verbirgt, von den „known“ Ports ganz zu schweigen.

Die Portnummern werden aus einer Textdatei gelesen, die auf UNIX-Systemen unter `/etc/services` und auf Windows-Systemen unter `%SystemRoot%\system32\drivers\etc\services` zu finden ist. Das Interface für den Programmierer ist der Auflösung von Hostnamen sehr ähnlich.

6.1. Funktionsweise des Domainname-Systems

Das Domainname-System ist hierarchisch aufgebaut. Die Bezeichner sind Pfaden in einem Dateisystem nicht unähnlich, jedoch steht die Wurzel am Ende, und die Blätter am Anfang. Ein Hostname kann hierbei entweder relativ sein (z.B. `pc50`), oder absolut (z.B. `www.fbmnd.fh-frankfurt.de`). Letztere werden auch als FQDN, Fully Qualified Domain Name, bezeichnet. Sie enden stets mit einer Top-Level-Domain (TLD).

Top-Level-Domains sind zumeist Ländern (.de, .fr, .uk) zugeordnet. Weiterhin gibt es spezielle Domains, die eine Gruppierung nach Inhalten bieten sollten (.org für Organisationen, .net für Provider und ähnliche Dienstanbieter, .edu für Ausbildungsstätten, .gov für Regierungsbehörden, ...). Man erkennt deutlich, daß die Entwicklung in den USA vonstatten ging, und die meisten Inhaber von edu- und gov-Domains sitzen ebenfalls in den USA. Die Toplevel-Domains werden regelmäßig erweitert, beispielsweise um .eu.

Wenn man sich den Namensraum als Baum vorstellt, sitzen in der Wurzel die (deshalb) sogenannten Rootserver. Sie sind geographisch verteilt, um eine Verteilung der Last und eine Ausfallsicherheit zu schaffen, wobei die meisten in den USA stehen. Zur Zeit gibt es 13 Stück (aber zumeist als Cluster organisiert, nicht wirklich einzelne Maschinen) von A bis M [38]. Diese Rootserver kennen die zuständigen Nameserver für die einzelnen Top-Level-Domains.

Die nächste Ebene besteht aus den Top-Level-Nameservern, die Nameserver für alle registrierten Second-Level-Domains kennen. Diese können beantragt und zugewiesen werden, wobei größere Firmen in der Regel selbst die weitere Infrastruktur bereitstellen (insbesondere die Nameserver für Subdomains), während kleinere Firmen und Privatleute das professionellen Hostern überlassen. Die Second-Level-Domains können relativ frei vergeben werden, die meisten Einschränkungen dürften im Marken- und Urheberrecht begründet sein.

Wenn nun also die Adresse von `www.fbmnd.fh-frankfurt.de` herauszufinden ist, wird zunächst ein Rootserver nach der Zuständigkeit für `.de` gefragt. Die Antwort könnte z.B. `f.nic.de` sein. Dieser weiß nun, welcher Nameserver für `fh-frankfurt.de` zuständig ist, nämlich `medusa.fh-frankfurt.de` (unter anderem). In diesem Fall ist die Fragerei zunächst beendet, denn dieser Nameserver kennt auch die Hosts in `fbmnd.fh-frankfurt.de`. Es wäre aber auch denkbar, daß diese Subdomain einen eigenen Nameserver betreibt, der ebenfalls befragt werden müsste.

Damit nicht jedesmal alle bis zum Rootserver hinauf befragt werden müssen, werden Antworten gecachet. Wie lange ein Eintrag im Cache leben darf, wird von dem Nameserver bestimmt, der die Antwort gegeben hat. Damit kann der Betreiber eines Nameservers bestimmen, wie oft sein Server befragt werden muß.

Wenn ein Programm nun einen Hostnamen auflösen will, ruft es die entsprechende Funktion des Resolvers auf. Dieser wird zunächst (je nach Konfiguration) eine lokale Datei prüfen, und dann entweder einen lokalen Nameserver oder einen eingestellten fremden Nameserver (beispielsweise den des Internet Service Providers wie T-Online) befragen. Kennt dieser die Antwort nicht, fragt er sich durch, und merkt sich die Antwort für künftige Anfragen. Der Anwender hat damit nicht viel zu schaffen, für ihn ist nur der Weg zum ersten Nameserver interessant.

Einträge im DNS werden als Resource Records (RR) bezeichnet. Sie enthalten üblicherweise mehrere Records, die unterschiedliche Informationen enthalten. A-Records enthalten IPv4-Adressen, AAAA-Records IPv6-Adressen, MX-Records die Namen der für diesen Host zuständigen Mailserver (MX steht für Mail Exchanger), und CNAME-Records enthalten „kanonische Namen“ (Aliasnamen). Jeder Host kann mehrere dieser Records haben, z.B. mehrere IP-Adressen und mehrere Hostnamen. Gerade letzteres ist recht nützlich, um häufige Umstellungen zu vermeiden: wenn der Host `greg.example.com` z.B. neben einem Webserver für die Domain auch deren FTP-Server und vielleicht noch einen Timeserver beherbergt, können ihm die kanonischen Namen `www`, `ftp` und `ntp` gegeben werden. Die Rechner, deren Zeit nun per NTP gesetzt werden sollen, müssen nur auf `ntp.example.com` eingestellt werden, und falls dieser Service auf `mike.example.com` umziehen sollte, muß lediglich eine Anpassung im Zonefile des Nameservers erfolgen, anstatt hunderte Rechner umkonfigurieren zu müssen.

6.2. Hostnamen auflösen

Zum Auflösen von Hostnamen in IP-Adressen stehen mehrere Funktionen zur Verfügung, jedoch wird im Folgenden nur `gethostbyname()` und das Gegenstück `gethostbyaddr()` besprochen, denn nur diese sind auch unter Windows verfügbar.

Die Deklaration der beiden Funktionen:

```
struct hostent *gethostbyname(const char *name);
struct hostent *gethostbyaddr(const char *addr, int len, int type);
```

Die von beiden Funktionen im Erfolgsfall zurückgegebene Struktur `hostent{}`:

```
struct hostent {
    char *h_name;               /* official name of host */
    char **h_aliases;           /* alias list */
```

```
    int  h_addrtype;            /* host address type */
    int  h_length;              /* length of address */
    char **h_addr_list;         /* list of addresses from name server */
};
#define h_addr h_addr_list[0]   /* address, for backward compatibility */
```

Im Fehlerfall geben die Funktionen NULL zurück.

Obwohl die Adressen jeweils als `char*` deklariert sind (`gethostbyaddr()` bzw. `h_addr_list`), handelt es sich hierbei keineswegs um die Adressen in gepunkteter Darstellung (etwa „192.168.0.1“), sondern um ihre binäre Repräsentation. Die Zeiger kann man also getrost auf `struct in_addr*` casten. Obige Deklaration habe ich aus der FreeBSD-Manpage entnommen, nach POSIX ist zwar der erste Parameter von `gethostbyaddr()` inzwischen `const void*`, jedoch hat sich die Struktur `hostent{}` nicht verändert.

Während viele UNIX-Implementierungen bei `gethostbyname()` die Angabe einer IP-Adresse in der *dotted notation* annehmen (es kommt dann ein hostent-Eintrag zurück, dessen `h_addr`-Eintrag gültig ist), funktioniert dies bei Windows nicht. Um portabel zu bleiben muß man also Benutzereingaben zuerst mit `inet_addr()` überprüfen, und erst im Falle einer ungültigen IP-Adresse einen Hostnamen vermuten.

Es ist weit verbreitet, nur `h_addr` auszuwerten, wenn man einen Hostnamen zu einer Adresse auflösen will. Jedoch sind auch weitere Einträge möglich, und nach einem erfolglosen Verbindungsaufbau zu der Adresse in `h_addr` ist es sicher eine gute Idee, auch die anderen zu versuchen.

Weiter gilt es unter UNIX zu beachten, daß im Unterschied zu den übrigen Funktionen die Gründe für den Fehlschlag einer Resolverfunktion durch Auswertung von `h_errno` bzw. dem Aufruf von `herror()` und `hstrerror()` zu beziehen sind.

In Anhang A.1 ist ein Beispielprogramm für die Auflösung von Hostnamen abgedruckt.

6.3. Servicenamen auflösen

Das Auflösen von Servicenamen bzw. Portnummern passiert in einer ähnlichen Weise, die Funktionen zu diesem Zwecke sind:

```
struct servent *getservbyname(const char *name, const char *proto);
struct servent *getservbyport(int port, const char *proto);

struct  servent {
    char *s_name;       /* official name of service */
    char **s_aliases;   /* alias list */
    int  s_port;        /* port service resides at */
    char *s_proto;      /* protocol to use */
};
```

Auch diese Funktionen geben im Fehlerfall NULL zurück.

Die Portnummer ist (wie bei allen Socketfunktionen) in Network Byte Order anzugeben, bzw. auch in dieser in der Ergebnisstruktur zu finden. Das Protokoll wird als Zeichenkette angegeben, z.B. “tcp“.

Die Anwendung der Funktionen erfolg ohne größere Überraschungen, ein Beispiel findet sich in Anhang A.2.

7. Socket-Funktionen

7.1. Allgemeines

Die Windows-Sockets wurden ursprünglich den BSD-Sockets nachempfunden, weshalb die meisten Funktionen gleichermaßen verwendet werden. Dies ist sehr erfreulich, denn somit sind Programme in der Regel sehr leicht übernehmbar. Allerdings hat sich das Socket-API seit Entwicklung der Windows-Sockets ein wenig verändert, und diese Änderungen spiegeln sich unter Windows nicht wider. Außerdem sind ein paar Datentypen, und damit verbunden auch Rückgabewerte, subtil anders, und das Ignorieren der Unterschiede kann sich möglicherweise böse rächen.

Unter UNIX fasst man Sockets wie Dateideskriptoren auf. Sie passen in den Datentyp `int`, sind jedoch nicht negativ. Windows hingegen setzt den Datentyp `SOCKET` ein, der vorzeichenlos ist. Während fehlgeschlagene Systemaufrufe unter UNIX nun -1 zurück liefern können, passt dieser Wert nicht in einen vorzeichenlosen Datentyp wie `SOCKET`. Deshalb ist der hier für Vergleiche zu verwendende Wert `INVALID_SOCKET`.

Zahlen wie Portnummern oder IP-Adressen in binärer Repräsentation (wie von `inet_addr()` geliefert) sind generell in Network Byte Order anzugeben. Die Umwandlung erfolgt durch die in 7.15 angegebenen Funktionen.

Wenn unter UNIX ein neuer Socket erzeugt wird (oder eine Datei geöffnet wird), so erhält der Deskriptor den nächsten freien Wert. Hat ein Prozeß beispielsweise bereits drei Dateien offen, so sind die Deskriptoren 0 - 5 belegt (0, 1 und 2 für *stdin*, *stdout* und *stderr*, 3, 4 und 5 für die drei Dateien), und der neu angelegte Socket erhält garantiert den Deskriptor 6. Dies ist unter Windows keinesfalls so, man sollte `SOCKET` tunlichst als Blackbox auffassen, und ähnlich wie bei `HANDLE` keinerlei Vermutungen über den Inhalt anstellen.

Eine weitere Windows-Spezialität sind einige Funktionen, die zum Betreiben der Sockets nötig sind. Sie werden unter 7.16 vorgestellt. Ohne sie funktioniert kein Socket-Befehl, auch nicht das Auflösen von Hostnamen.

Als echter Bestandteil des Betriebssystems fassen sich die Socket-Befehle unter UNIX genau wie andere Systemaufrufe an. Das ist einerseits die -1 als Rückgabewert im Fehlerfall, zum anderen die genauere Information über das Scheitern in der globalen Fehlervariablen `errno`. Zum Auswerten stehen die Funktionen `perror()` und `strerror()` der C-Standardbibliothek zur Verfügung. Eine Ausnahme bilden hierbei die Resolverfunktionen (vgl. 6.2). Unter Windows sind die Fehlergründe **nicht** wie bei normalen Befehlen über `GetLastError()` zu beziehen, sondern über `WSAGetLastError()`, siehe 7.16.

Generell gilt es als guter Stil, alle Systemaufrufe auf einen Fehlschlag hin zu untersuchen. Andererseits gibt es Befehle, die „normalerweise“ nie fehlschlagen, wie etwa `listen()`. Es liegt selbstverständlich ganz im Ermessen des Programmierers, ob er alle Fehlerquellen überprüfen möchte, oder an einigen strategisch günstigen Punkten testet, und dort ggf. etwas undeutlichere Meldungen in Kauf nimmt. Ein Beispiel: ich überprüfe immer `socket()` auf Fehlschlag (z.B. keine Ressourcen mehr frei), jedoch würde man einen Fehler in einer Serveranwendung auch bei `bind()` oder `accept()` bemerken, da diese ohne einen Socket, auf den sie operieren sollen, schwerlich funktionieren werden. Meine Strategie ist, Fehler immer dort zu erkennen, wo sie zuerst auftreten.

Die Deklarationen der Befehle im Folgenden sind alle aus UNIX Manpages entnommen, allen voran FreeBSD. Wenn es entscheidende Unterschiede zu Windows gibt, wird dies angemerkt. Unter Windows sind viele der aufgeführten Header nicht verfügbar, ihrer statt ist `winsock.h` zu verwenden. Nicht existierende Datentypen wie z.B. `socklen_t` sind entsprechend der Funktionsdeklaration durch z.B. `int` zu ersetzen.

7.2. socket

Zweck
Erzeugt einen neuen Socket.

Deklaration

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

Beschreibung
Mit `socket()` wird vom Betriebssystem ein neuer Socket angelegt, und der dazugehörige Deskriptor an den Aufrufer zurückgeliefert.

Der Parameter *domain* beschreibt den Einsatzzweck des Sockets. Unter UNIX gibt es hier in der Regel viele Möglichkeiten, die Windows Sockets stellen als Subset nur die Kommunikation über TCP/IP-Netzwerke bereit. Die Konstante für diesen Bereich ist `PF_INET` (PF steht hierbei für Protocol Family). Man sieht in vielen Progammen auch `AF_INET`, was vom Konzept her nicht richtig ist (AF steht für Address Family), diese Konstante wird bei anderen Befehlen eingesetzt, und zwar um die Art der Adressen anzugeben – die natürlich durchaus mit der Art des verwendeten Protokolls zusammenhängt. Aus historischen Gründen haben diese beiden Konstanten in allen mir bekannten Implementierungen den gleichen Wert, weshalb diese Vertauschung auch funktioniert.

Mit dem Parameter *type* wird der Typ des Sockets festgelegt, sein Verhalten. Im portablen Bereich von `PF_INET` sind nur `SOCK_STREAM` und `SOCK_DGRAM` von Belang. Mit `SOCK_STREAM` wird ein verbindungsorientierter Socket erzeugt, der einen Bytestrom zur Verfügung stellt. Wird kein anderes Protokoll vorgeschrieben, wird TCP verwendet. Bei `SOCK_DGRAM` weist der Socket Datagramm-Eigenschaften auf, d.h. Pakete werden so versendet wie vom Benutzer spezifiziert. Hierbei kommt üblicherweise UDP zum Einsatz.

Wenn ein von der Voreinstellung abweichendes Protokoll erwünscht ist, kann dies mit dem Parameter *procotol* erreicht werden. Die Konstanten haben Bezeichnungen wie `IPPROTO_TCP`, `IPPROTO_UDP`, etc. Um das vom System vorgesehene Protokoll einzusetzen, wird als Parameter 0 angegeben.

Rückgabewerte
Unter UNIX wird im Fehlerfall -1 zurückgegeben, im Erfolgsfall der Deskriptor des neu angelegten Sockets.

Unter Windows ist der Datentyp des Rückgabewerts `SOCKET` und vorzeichnlos. Die Konstante im Fehlerfall ist `INVALID_SOCKET`, im Erfolgsfall wird ebenfalls der Deskriptor des neuen Sockets zurückgegeben.

Fehler
Die Funktion `socket()` kann fehlschlagen, wenn dem System zu wenig Ressourcen zur Verfügung stehen, ungültige Werte für Protokollfamilie, Typ oder Protokoll übergeben wurden, oder die gewünschte Kombination nicht möglich ist.

Bemerkungen
Ein Socket muß ähnlich wie eine Datei am Ende des Verwendungszeitraums geschlossen werden (`close()` bzw. `closesocket()`, siehe 7.7). Dies passiert implizit beim Verlassen des Prozesses, kann jedoch auch bereits vorher nützlich oder erwünscht sein.

Es gilt als schlechte Idee, über den numerischen Wert eines Deskriptors Annahmen zu machen. Unter UNIX kann jedoch davon ausgegangen werden, daß der nächste nicht-benutzte Wert verwendet werden wird, während unter Windows auch „Löcher“ entstehen können.

7.3. bind

Zweck
Weist einem Socket eine lokale Protokoll-Adresse zu.

Deklaration

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int s, const struct sockaddr *addr, socklen_t addrlen);
```

Komplexe Datentypen

```
struct sockaddr_in {
    uint8_t          sin_len;
    sa_family_t      sin_family;
    in_port_t        sin_port;
    struct in_addr   sin_addr;
    char             sin_zero[8];
};
```

Beschreibung
Abstrakt gesprochen weist `bind` einem Socket eine lokale Protokoll-Adresse zu. Unter UNIX gibt es neben `AF_INET` noch einige weitere, die jeweils eine eigene Aderssierungsart kennen. Damit das Interface allgemein bleibt, wird der abstrakte Datentyp `struct sockaddr{}` eingesetzt. Für jede konkrete Adress-Familie, wie in unserem Falle `AF_INET`, existiert ein angepasster Datentyp, in diesem Fall die Struktur `sockaddr_in{}`. In diesem Kontext besteht eine Adress-Familie aus dem Verbund einer IP-Adresse und einer Port-Nummer, in dieser Form auch als *Endpunkt* einer Verbindung bezeichnet.

Die obenstehende Form der Struktur `sockaddr_in{}` stammt von FreeBSD, unter Linux, Windows und anderen Systemen, die ihre Socket-Schicht von BSD recht früh übernommen haben, existiert das Element `sin_len` nicht. Dies ist allerdings kein Problem, denn die meisten Funktionen erhalten als zusätzlichen Parameter eine Angabe zur Größe der Struktur (weshalb der Sinn von `sin_len` allgemein umstritten ist).

Das Element `sin_family` erhält im TCP/IP-Kontext die Konstante `AF_INET`, der Bereich `sin_zero` vergrößert die Struktur auf die Größe des generischen Typs `sockaddr{}`, den `bind()` auch als Parameter erwartet. Deshalb kann (und muß) bei der Verwendung der Konkretisierung `sockaddr_in{}` gecastet werden:

```
  struct sockaddr_in addr;
  ...
  bind(sock, (struct sockaddr*) &addr, sizeof(addr));
```

Es ist eine gute Idee, zu Beginn mit `memset()` die gesamte Struktur mit 0 zu überschreiben, und im Folgenden nur die bekannten Elemente zu setzen. Damit ist auch die Problematik von `sin_len` vermieden.

Sollen mehrere Endpunkte (systemweit betrachtet) gebunden werden, so müssen sie sich entweder in ihrer IP-Adresse oder in ihrer Port-Nummer unterscheiden (oder beides). Das bedeutet: eine Port-Nummer kann mehrfach gebunden werden, wenn für jeden Socket eine andere lokale IP-Adresse verwendet wird. Umgekehrt können mehrere Sockets der gleichen IP-Adresse zugeordnet sein, wenn sie sich in ihrer Port-Nummer unterscheiden.

Als IP-Adresse kann die Konstante `INADDR_ANY` angegeben werden, um den Socket jedem lokalen Interface zuzuordnen. Wenn ein solcher Socket zusätzlich zu einem spezielleren (mit einer einzelnen IP-Adresse) existiert, so wird der Speziellere bevorzugt, wenn ein entsprechendes Datagramm eintrifft.

Für die Port-Nummer kann 0 angegeben werden, um vom System einen kurzlebigen Port vorgeschlagen zu bekommen. Diesen kann man nach erfolgreichem Binden mit dem Befehl `getsockname()` herausfinden.

Das Binden von Sockets ist nicht nur für Server-Anwendungen interessant, sondern kann auch vor einem Aufruf von `connect()` erfolgen. In diesem Fall wird der zugewiesene Endpunkt als Quelle der Datagramme eingesetzt. Dies kann für die Implementierung mancher Anwendungsprotokolle erforderlich sein.

Rückgabewerte
Im Erfolgsfall liefert `bind()` 0 als Rückgabewert, im Fehlerfall -1.

Fehler
Der angegebene Socket kann entweder bereits gebunden oder ungültig sein. Der angeforderte Endpunkt kann bereits belegt sein.

Bemerkungen
Darauf achten, daß der Wert in `sin_port` in Network Byte Order vorliegt.

Wenn sowohl die Port-Nummer 0 als auch die IP-Adresse `INADDR_ANY` ist, kann `getsockname()` erst dann einen vernünftigen Wert liefern, wenn der Socket bereits verbunden ist.

7.4. listen

Zweck
Versetzt einen Socket in den Lauschmodus.

Deklaration

```
#include <sys/types.h>
#include <sys/socket.h>

int listen(int s, int backlog);
```

Beschreibung
Um von einem Server-Socket Verbindungen anzunehmen, muß dieser zuerst in den Lauschmodus versetzt werden. Der Parameter *backlog* gibt hierbei die Länge der Warteschlange für nicht vollständig verbundene Clients an.

Auf TCP-Ebene sind dies Client, die ihr SYN-Datagramm bereits versendet haben. Die Verbindung ist jedoch erst dann vollständig abgewickelt, wenn der Server dieses Verbindungsgesuch akzeptiert; dies geschieht mit dem Befehl `accept()`.

Wird für diesen Parameter der Wert 0 angegeben, wählt das System einen Wert aus. Wie hoch der Wert für *backlog* sein darf, ist plattformabhängig, und ggf. konfigurierbar.

Rückgabewerte
Im Erfolgsfall gibt `listen()` 0 zurück, im Fehlerfall -1.

Fehler
Der socket kann ungültig sein, oder von einer Art, die das Lauschen nicht kennt.

7.5. accept

Zweck
Nimmt eine Verbindung an.

Deklaration

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int s, struct sockaddr * restrict addr,
    socklen_t * restrict addrlen);
```

Komplexe Datentypen
`struct sockaddr_in{}`, siehe 7.3.

Beschreibung
Die Funktion `accept()` wird in einer Server-Applikation genutzt, um einen Client von einem lauschenden Socket „abzuholen“. Der Parameter *s* ist hierbei der lauschende Socket, der Rückgabewert von `accept()` ist der neue Client-Socket. Dementsprechend ist die Deklaration unter Windows auch abgewandelt und gibt `SOCKET` zurück.

Wenn Informationen über den Endpunkt der Verbindung gewünscht werden, muß als zweiter und dritter Parameter ein Zeiger auf eine Struktur `sockaddr_in{}` sowie ein Zeiger auf eine Variable, die die Größe der Struktur enthält, angegeben werden. Dies sind Wert-Ergebnis-Parameter, d.h. nach Aufruf der Funktion enthält *addrlen* die tatsächliche Länge der Struktur, und es müsste mit üblen Dingen zugehen, wäre dies nicht entsprechend der Größe von `struct sockaddr_in{}`. *addrlen* ist vorzubelegen!

Wenn keine Informationen über den Endpunkt gewünscht sind, kann für beide Parameter NULL bzw. 0 angegeben werden. Diese Informationen können auch später noch über den Befehl `getpeername()` eingeholt werden.

Rückgabewerte
Im Erfolgsfall ein neuer Socket, im Fehlerfall -1 bzw. `INVALID_SOCKET` unter Windows.

Fehler
Der lauschende Socket kann ungültig sein. Die Deskriptor-Tabelle des Prozesses kann erschöpft sein.

Bemerkungen
Bei `accept()` handelt es sich um einen Befehl, der für längere Zeit blockiert, wenn kein Client zur Verfügung steht. Hier bietet sich die Verwendung von `select()` an (der Lauschsocket wird *lesbar* wenn eine neue Verbindung verfügbar ist).

7.6. connect

Zweck
Baut eine Verbindung auf.

Deklaration

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int s, const struct sockaddr *name, socklen_t namelen);
```

Komplexe Datentypen
`struct sockaddr_in{}`, siehe 7.3.

Beschreibung
Diese Funktion betrifft Client-Programme, die aktiv eine Verbindung zu einem Ziel aufbauen. Der Zielendpunkt muß vollständig angegeben werden, *name*. Der Parameter *namelen* gibt die Größe der Struktur an, die übergeben wurde.

Der Befehl kann für längere Zeit blockieren, insbesondere unter Windows wurde beobachtet, daß sich das Programm damit für einige Minuten wie abgestürzt verhält. Abhilfe schaffen hier nur nicht-blockierende Sockets.

Auch wenn der häufigste Grund für einen Fehlschlag ein fehlender Server auf der Gegenseite ist, sollte der genaue Fehlergrund untersucht werden, einige Implementierungen geben hier wertvolle Hinweise (z.B. network unreachable).

Rückgabewerte
0 im Erfolgsfall, -1 wenn der Aufruf fehlschlägt.

Fehler
Das Ziel ist nicht erreichbar, genauere Gründe sind über errno/`WSAGetLastError()` erreichbar.

7.7. close

Zweck
Schließt einen Socket.

Deklaration

```
#include <unistd.h>

int close(int d);
```

Unter Windows:

```
#include <winsock.h>

int closesocket (SOCKET s);
```

Beschreibung
Ein Socket muß nach Beendigung der Kommunikation freigegeben werden. Dies geschieht implizit beim Beenden des Prozesses, kann jedoch auch schon davor sinnvoll sein.

Das Verhalten beim Schließen eines Sockets hängt vom Zustand der Socket-Option `SO_LINGER` (vgl. 8.1.5) ab.

Ist die Option `SO_LINGER` nicht gesetzt, so kehrt die Funktion `close()` sofort zurück, jedoch werden ausstehende Daten noch versendet. Ob dies klappt bekommt der Aufrufer jedoch nicht mit, denn nach erfolgtem `close()` ist der Socket ungültig.

Ist die Option gesetzt, aber der Timeout beträgt 0, so wird der Socket sofort geschlossen, evtl. ausstehende Daten werden verworfen. Dies ist im allgemeinen unhöflich, und wird auch als „hard close“ bezeichnet.

Ist die Option aktiv und ein Timeout gesetzt, so blockiert `close()` solange, bis entweder alle noch ausstehenden Daten versendet wurden, oder aber der Timeout abgelaufen ist.

Rückgabewerte
Im Erfolgsfall 0, im Fehlerfall -1.

Fehler
Der Socket ist ungültig, bei blockierendem `close()` auch ein unterbrochener Systemaufruf möglich.

Bemerkungen
Bei Server-Anwendungen mit `fork()` daran denken, daß in jedem Prozeß der Socket geschlossen werden muß, damit die Verbindung tatsächlich beendet wird!

7.8. shutdown

Zweck
Schließt einen Teil einer full-duplex-Verbindung.

Deklaration

```
#include <sys/types.h>
#include <sys/socket.h>

int shutdown(int s, int how);
```

Beschreibung
Eine TCP-Verbindung ist ein Bytestrom, der in beide Richtungen funktioniert. Mit dem Befehl `shutdown()` kann man diese Verbindung teilweise schließen.

Der Parameter *how* kann die Werte `SHUT_RD`, `SHUT_WR` oder `SHUT_RDWR` annehmen, und damit entweder das Lesende oder das Schreibende Ende schließen. In der verbleibenden Richtung kann nach wie vor kommuniziert werden.

Wozu es gut sein mag, weder lesen noch schreiben zu wollen, jedoch den Socket weiterhin offen zu halten (indem kein `close()` aufgerufen wird), entzieht sich meiner Kenntnis.

Auf UDP-Sockets hat es auch einen Effekt (TODO!)

Rückgabewerte
0 im Erfolgsfall, -1 im Fehlerfall.

Fehler
Der Socket kann ungültig oder nicht verbunden sein.

Bemerkungen
Unter Windows heißen die Konstanten `SD_RECEIVE`, `SD_SEND` und `SD_BOTH`.

7.9. recv

Zweck
Empfängt Daten von einem Socket.

Deklaration

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t recv(int s, void *buf, size_t len, int flags);
```

Beschreibung
Die Funktion `recv()` wird eingesetzt, um Daten von einem verbundenen Socket zu lesen.

Der Parameter *buf* zeigt hierbei auf einen Puffer, der die zu empfangenden Daten aufnehmen soll. Seine Größe wird in *len* übergeben. Wenn mehr Daten vorhanden sind, als in den Puffer passen, ist das Verhalten abhängig vom Socket-Typ. Bei Byteströmen (`SOCK_STREAM`) wird bei einem weiteren Aufruf dort fortgesetzt, wo der Pufferplatz erschöpft war. Bei Datagrammen (`SOCK_DGRAM`) wird der „überstehende“ Rest des Datagramms verworfen, er geht also verloren.

Windows kennt nur die beiden folgenden Flags, jedoch sind unter UNIX üblicherweise weitere bekannt:

- `MSG_PEEK` – die Daten werden in den Puffer kopiert, aber nicht aus der Warteschlange gelöscht
- `MSG_OOB` – Out Of Band-Daten werden gelesen

Out Of Band-Daten werden bei der Funktion `send()` (7.11) genauer besprochen. Grundsätzlich kann hier nur ein Byte gelesen werden, auch wenn mehr Daten als Out Of Band-Daten versendet wurden. Die OOB-Daten müssen gelesen werden sobald sie anstehen, nach einem weiteren erfolgreichen Aufruf von `recv()` werden sie verworfen (genauer: der Zustand „Es liegen OOB-Daten vor“ wird verlassen). Durch „normale“ `recv()`-Aufrufe werden sie *nicht* gelesen. Ein ausführliches Beispiel ist in Anhang A.3 zu finden.

Das Lesen von einem blockierenden Socket legt den Prozeß solange schlafen, bis Daten eingetroffen sind. Um dies zu verhindern kann der Befehl `select()` benutzt werden, der in Kapitel 9.2.1 vorgestellt wird.

Rückgabewerte
Im Erfolgsfall wird die Zahl der gelesenen Byte zurückgeliefert, im Fehlerfall -1. Wenn 0 Byte gelesen werden, wurde der Socket geschlossen, ohne daß weitere Daten gesendet wurden.

Fehler
Der Socket kann ungültig oder nicht verbunden sein.
Es wurde versucht OOB-Daten zu lesen, obwohl keine vorhanden sind.

Bemerkungen
Unter UNIX kann für einen verbundenen Socket auch `read()` eingesetzt werden.

7.10. recvfrom

Zweck
Empfängt Daten von einem Socket.

Deklaration

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t recvfrom(int s, void * restrict buf, size_t len, int flags,
    struct sockaddr * restrict from, socklen_t * restrict fromlen);
```

Komplexe Datentypen
`struct sockaddr_in{}`, siehe 7.3.

Beschreibung
Die Funktion `recvfrom()` arbeitet im Wesentlichen genauso wie `recv()`, jedoch mit dem Unterschied, daß der Socket nicht verbunden sein muß.

Die ersten vier Parameter sind die gleichen wie bei `recv()`. Der Parameter *from* nimmt die Quelladresse des Datagramms auf, während *fromlen* als Wert-Ergebnis-Parameter beim Aufruf die Größe der Struktur enthalten muß (vgl. auch `accept()`, 7.5). Wird für diese Parameter NULL eingesetzt, so werden die Daten entsprechend verworfen.

`recvfrom()` wird meistens in Verbindung mit UDP-Sockets eingesetzt.

Rückgabewerte
Im Erfolgsfall wird die Zahl der gelesenen Byte zurückgeliefert, im Fehlerfall -1. Wenn 0 Byte gelesen werden, wurde der Socket geschlossen, ohne daß weitere Daten gesendet wurden.

Fehler
Der Socket kann ungültig sein.

7.11. send

Zweck
Sendet Daten über einen Socket.

Deklaration

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t send(int s, const void *msg, size_t len, int flags);
```

Beschreibung
Mit der Funktion `send()` können Daten über einen verbundenen Socket versendet werden.

Der Paramter *msg* zeigt auf einen Puffer, der die zu versendenden Daten enthält. Die Länge dieser Daten wird durch *len* angegeben. Der Parameter *flags* kann Flags enthalten, die das Verhalten weiter beeinflussen. Unter UNIX stehen hier in der Regel mehrere zur verfügung, Windows kennt jedoch nur `MSG_OOB`, das zum Versenden von Out-of-band-Daten verwendet wird, sowie `MSG_DONTROUTE` (Datagramme werden nicht geroutet).

Versenden von Out-of-band-Daten
TCP beschreibt mit dem URG-Flag und dem Urgent-Pointer (vgl. 4.1.4) eine Möglichekeit besonders „dringende“ Daten auszuzeichnen. Allerdings ist man bei der Interpretation etwas uneinig, was die Länge der Daten angeht, denn TCP lässt nur das *Zeigen* auf den Anfang dieses Blocks zu, wie viel nun dazugehört ist ungewiss.

Konsequenterweise kann mit dem Socket-API auch nur *ein Byte* als OOB-Datensatz versendet und empfangen werden. Die Daten werden hierbei innerhalb des normalen Datenstroms versendet, jedoch durch einen Aufruf von `recv()` oder `recvfrom()` nicht gelesen, solange das Flag `MSG_OOB` dort nicht gesetzt wird. Erfolgt nach dem Auftreten der OOB-Daten ein weiterer Aufruf von `recv()` bzw. `recvfrom()`, so werden die OOB-Daten verworfen. Wichtig in diesem Zusammenhang ist noch, daß im Kernel der Zustand „es liegen OOB-Daten vor“ erst dann verlassen wird, wenn ein weiteres `recv()` erfolgt ist. Das heißt insbesondere, daß `select()` nocheinmal zuschlägt, obwohl keine OOB-Daten vorhanden sind! In Anhang A.3 gibt es ein ausführliches Beispiel hierzu.

Es gibt nun drei Möglichkeiten, OOB-Daten zu erkennen. Zum einen kann mit einem Aufruf von `recv()` bzw. `recvfrom()` mit gesetztem `MSG_OOB`-Flag getestet werden, ob solche Daten anliegen (andernfalls kehrt der Aufruf mit -1 zurück, `errno` steht auf EINVAL). Eine andere Lösung besteht im Einsatz von `select()` (siehe Kapitel 9.2.1). Ein Socket, auf dem OOB-Daten gelesen werden können, wird hier als „Ausnahme“ bezeichnet, siehe dort. Die dritte Lösung ist das Setzen eines „Eigentümers“ des Sockets (mit `fcntl()`, F_SETOWN); dem angegebenen Prozeß wird dann beim Eintreffen von OOB-Daten das Signal SIGURG zugestellt. Das Behandeln von Signalen wird in Kapitel 11.5 vorgestellt.

Die Verwendung des OOB-Mechanismus kann wirklich nicht empfohlen werden. Er bietet außerdem keine wirklichen Vorteile, denn Pakete mit URG-Flag werden nicht schneller zugestellt als normale Pakete auch. Der zweite logische Datenkanal, der bei der Verwendung von OOB-Daten vorgegaukelt wird, sollte viel lieber durch einen echten zweiten Kanal implementiert werden, wie bei FTP (vgl. Kapitel 5.2) vorgemacht wird.

Rückgabewerte
Die Zahl der erfolgreich gesendeten Bytes, oder -1 im Fehlerfall.

Fehler
Der Socket ist nicht verbunden, oder ungültig.

Bemerkungen
Bei Stream-Sockets ist nicht garantiert, daß die Daten in den „Portionen“ ankommen, in denen sie gesendet wurden. Dies bleibt den Datagramm-Sockets vorbehalten (vgl. 7.2).
Unter UNIX kann für einen verbundenen Socket auch `write()` verwendet werden.

7.12. sendto

Zweck
Sendet Daten über einen Socket.

Deklaration

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t sendto(int s, const void *msg, size_t len, int flags,
    const struct sockaddr *to, socklen_t tolen);
```

Komplexe Datentypen
`struct sockaddr_in{}`, siehe 7.3.

Beschreibung
Die Funktion `sendto()` arbeitet im wesentlichen wie `send()` mit dem Unterschied, daß sie auch auf nicht-verbundene Sockets angewendet werden kann. Mit dem Parameter *to* wird das Ziel des Datagramms angegeben, während *tolen* die Größe der Struktur angibt, die als *to* übergeben wird.

Wenn die Daten nicht *atomar* (also in einem Aufruf ohne Unterbrechung) an die darunterliegende Schicht überreicht werden können, schlägt der Aufruf fehlt. Der Fehlercode ist hierbei `EMSGSIZE` (UNIX) bzw. `WSAEMSGSIZE` (Win32).

`sendto()` wird üblicherweise für UDP-Sockets eingesetzt.

Rückgabewerte
Die Zahl der geschriebenen Bytes im Erfolgsfall, -1 sonst.

Fehler
Der Socket kann ungültig sein, die Nachricht kann zu lang sein.

7.13. getpeername

Zweck
Holt Informationen zum fernen Endpunkt ein.

Deklaration

```
#include <sys/types.h>
#include <sys/socket.h>

int getpeername(int s, struct sockaddr * restrict name,
    socklen_t * restrict namelen);
```

Komplexe Datentypen
`struct sockaddr_in{}`, siehe 7.3.

Beschreibung
Mit `getpeername()` wird der „Name“ des Verbindungspartners bei einem verbundenen Socket eingeholt. Dies ist insbesondere dann nützlich, wenn nur noch ein Socket-Deskriptor vorliegt, ohne weitere Information über Zieladresse und Zielport, wie etwa beim Einsatz von inetd (vgl. 9.1.2). Auch wenn bei `accept()` der zweite und dritte Parameter NULL bzw. 0 war, kann die Information hiermit nachträglich eingeholt werden.

Rückgabewerte
Im Erfolgsfall ist der zurückgegebene Wert 0, im Fehlerfall -1.

Fehler
Der Socket kann ungültig oder nicht verbunden sein.

Bemerkungen
namelen ist mit der Größe der Struktur vorzubelegen.

7.14. getsockname

Zweck
Holt Informationen zum lokalen Endpunkt sein.

Deklaration

```
#include <sys/types.h>
#include <sys/socket.h>

int getsockname(int s, struct sockaddr * restrict name,
    socklen_t * restrict namelen);
```

Komplexe Datentypen
`struct sockaddr_in{}`, siehe 7.3.

Beschreibung
Um den lokalen Endpunkt einer Verbindung herauszufinden, setzt man `getsockname()` ein. Der Socket muß dazu an ein lokales Interface gebunden, nicht jedoch notwendigerweise verbunden sein. Wenn als IP-Adresse für den lokalen Endpunkt `INADDR_ANY` angegeben wurde, enthält dieses Feld der Struktur `sockaddr_in{}` erst nach dem Verbinden einen zuverlässigen Wert.

Rückgabewerte
Die Funktion gibt im Erfolgsfall 0 zurück, im Fehlerfall -1.

Fehler
Der Socket kann ungültig sein.

Bemerkungen
Das Element *namelen* muß mit der Größe der Struktur vorbelegt werden.

7.15. htonl, htons, ntohl, ntohs

Zweck
Umwandlung von Werten zwischen Host Byte Order und Network Byte Order.

Deklaration

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

Beschreibung
Die Funktionen `htonl()` und `htons()` wandeln ganzzahlige Werte mit 32 bzw. 16 Bit Breite von der Host Byte Order (Little oder Big Endian) in die Network Byte Order (Big Endian) um.

Die Funktionen `ntohl()` und `ntohs()` wandeln ganzzahlige Werte mit 32 bzw. 16 Bit Breite von der Network Byte Order (Big Endian) in die Host Byte Order (Little oder Big Endian) um.

Auf Plattformen, die Big Endian zur Darstellung von Ganzzahlen verwenden, sind diese Funktionen in der Regel als leere Makros definiert.

Rückgabewerte
Die jeweils umgewandelten Werte.

7.16. Spezielle Winsock-Funktionen

Unter Windows gibt es eine Reihe von Funktionen, die zusätzlich aufgerufen werden müssen, oder aber Funktionen von UNIX ersetzen. Eine solche Ersetzung ist die Funktion `closesocket()` an Stelle von `close()`. Hierbei ist nichts weiter zu beachten.

Vor der ersten Verwendung einer Winsock-Funktion (dazu gehört auch das Auflösen von Hostnamen) muß `WSAStartup()` aufgerufen werden, nach dem letzten Aufruf `WSACleanup()`. Der Effekt gilt für alle Threads des aufrufenden Prozesses. Die Funktionen sind folgendermaßen deklariert:

```
int WSAStartup(WORD wVersionRequested, LPWSADATA lpWSAData);

int WSACleanup(void);
```

Als *wVersionRequested* wird ein mittels des Makros `MAKEWORD` erzeugter Wert angegeben, der die Major (low byte) und Minor (high byte) Version Number der gewünschten Winsock-Version beschreibt. Angegeben wird die höchste Version, die man verwenden möchte, und im Ergebnis nach dem Aufruf steht die höchste Version, die verwendet werden kann. Ein Beispiel:

```
WSADATA wsa;
...
if (WSAStartup(MAKEWORD(1, 1), &wsa) != 0)
{
    fprintf(stderr, "WSAStartup() failed: %lu\n", WSAGetLastError());
    return 1;
}
```

Winsock 1.1 unterstützt alles, das in diesem Buch beschrieben wird. Die nächste erwähnenswerte Version ist 2.0, jedoch sind darin keine für die Grundfunktionen relevanten Verbesserungen oder Veränderungen enthalten.

Eine weitere Besonderheit ist, daß Fehlercodes unter Windows nicht in *errno* zu finden sind, und auch nicht mit `perror()` oder `strerror()` ausgewertet werden können. Hierzu dient die Funktion `WSAGetLastError()`:

```
int WSAGetLastError(void);
```

Die ermittelten Fehlercodes (siehe auch Anhang B) können mittels der Funktion `FormatMessage()` in textuelle Repräsentationen umgewandelt werden:

```
char buffer[1024];
...
FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM, 0, ErrorNumber, 0, buffer,
    sizeof(buffer), 0);
```

Wobei *ErrorNumber* den Wert von `WSAGetLastError()` enthält. Alternativ zur Bereitstellung eines eigenen Puffers kann auch das Flag `FORMAT_MESSAGE_ALLOCATE_BUFFER` angegeben werden, wobei als 5. Parameter die Adresse eines Zeigers bereitgestellt werden muß. In diesem Fall muß der Puffer nach Gebrauch durch einen Aufruf von `LocalFree()` freigegeben werden. Der 6. Parameter gibt in diesem Fall das Minimum an Speicherplatz an, der von `FormatMessage()` bereitgestellt werden soll.

Hinweis: die Funktion `FormatMessage()` verfügt über weitere Eigenschaften, wie beispielsweise das Erzeugen von Meldungen aus Formatstrings ähnlich `sprintf()`. In [2] findet sich eine ausführliche Beschreibung.

Achtung: Die Ausführung von Winsock-Funktionen schlägt auf mysteriöse Weise fehl, wenn die Umgebungsvariable `%SystemRoot%` nicht oder nicht korrekt gesetzt ist.

8. Weitere Operationen auf Sockets

8.1. get- und setsockopt

Sockets haben eine Reihe von Parametern, die vom Benutzer manipuliert werden können. Dazu muß die Bezeichnung der Option sowie die Ebene, der sie zugeordnet ist, bekannt sein. Zur Manipulation stehen die beiden Funktionen `getsockopt()` und `setsockopt()` zur Verfügung:

```
#include <sys/types.h>
#include <sys/socket.h>

int getsockopt(int s, int level, int optname, void * restrict optval,
    socklen_t * restrict optlen);

int setsockopt(int s, int level, int optname, const void *optval,
    socklen_t optlen);
```

Die meisten der folgenden Optionen beziehen sich auf die Socket-Ebene, der Wert für *level* ist hierbei SOL_SOCKET. Sie sind größtenteils sowohl unter UNIX als auch unter Windows bekannt, im Einzelfall wird eine entsprechende Plattformabhängigkeit angegeben. Der Parameter *optname* bezeichnet die Option selbst; die Konstanten entsprechen den Überschriften der folgenden Kapitel.

optval und *optlen* haben verschiedene Bedeutungen. Bei einem Aufruf von `setsockopt()` enthält *optval* den zu setzenden Wert (dennoch wird eine Adresse übergeben!), während *optlen* die Größe der Variable beschreibt (beispielsweise `sizeof(int)`). Wenn `getsockopt()` zum Einsatz kommt, wird der abgefragte Wert an der von *optval* beschriebenen Adresse abgespeichert. *optlen* gibt hierbei beim Aufruf die zur Verfügung stehende Speichergröße an, und beinhaltet nach Rückkehr des Befehls die tatsächlich in Anspruch genommene Größe. Es handelt es sich also um einen Wert-Ergebnis-Parameter.

Der Rückgabewert ist bei beiden Funktionen im Erfolgsfall 0; im Fehlerfall beträgt der Rückgabewert -1, die globale Fehlervariable `errno` wird unter UNIX entsprechend gesetzt. Unter Windows kann mittels der Funktion `WSAGetLastError()` der Fehlergrund in Erfahrung gebracht werden (vgl. Kapitel 7.16).

Es sei noch angemerkt, daß die meisten Benutzer nie in die Verlegenheit kommen werden, Socket-Optionen setzen zu müssen. Es handelt sich meistens um Feintuning oder um das Umgehen seltsamer Eigenheiten mancher Implementierungen. Es kann jedoch nicht schaden eine Übersicht über die verbreiteten Optionen zu haben – auch sollte die Dokumentation des verwendeten Betriebssystems genauer betrachtet werden, oftmals existieren noch etliche weitere Optionen.

8.1.1. SO_BROADCAST

Datentyp: `int`

Mit dieser Option kann das Versenden von Broadcasts aktiviert bzw. deaktiviert werden. Broadcasting steht nur für Datagramm-Sockets zur Verfügung, und auch nur, wenn das darunterliegende Netzwerk

das Konzept der Broadcasts unterstützt (wie beispielsweise Ethernet). Das Thema Broadcasts wird in Kapitel 12.1 genauer besprochen.

Diese Option ist deshalb sinnvoll, weil eine Anwendung möglicherweise eine IP-Adresse vom Benutzer entgegennimmt, und ohne weitere Überprüfungen als Zieladresse eines Datagramms einsetzt. Wenn nun eine Broadcast-Adresse angegeben wurde, aber die Option SO_BROADCAST nicht gesetzt wurde, schlägt der Aufruf zum Senden fehl, und der Fehlergrund wird mit EACCES beschrieben.

8.1.2. SO_DEBUG

Datentyp: `int`

Diese Option verhält sich je nach Plattform unterschiedlich. Winsock implementiert sie beispielsweise überhaupt nicht, während unter BSD-UNIX davon die Rede ist, daß das Debugging in den darunterliegenden Schichten aktiviert wird.

Konzeptionell ist es möglich, daß unter Windows ein IP-Stack eines Drittanbieters eingesetzt wird, der möglicherweise SO_DEBUG unterstützt.

8.1.3. SO_DONTROUTE

Datentyp: `int`

Mit dieser Option kann der normale Routing-Mechanismus des verwendeten Protokolls umgangen werden. Um nur auf ein Datagramm Einfluß zu nehmen, kann auch bei einem Aufruf von `send()` oder `sendto()` das MSG_DONTROUTE-Flag gesetzt werden.

8.1.4. SO_KEEPALIVE

Datentyp: `int`

Ist die Option SO_KEEPALIVE bei einem TCP-Socket gesetzt, wird nach 2 Stunden der Inaktivität (also wenn weder Daten empfangen noch gesendet wurden) automatisch ein TCP-Segment an den Partner gesendet, auf das eine Reaktion erfolgen muß.

Wird das Segment wie erwartet beantwortet, passiert nichts zweiter. Nach weiteren zwei Stunden wird erneut ein Segment gesendet.

Antwortet der Partner mit einem RST-Datagramm, wird davon ausgegangen, daß die Gegenstelle zwischenzeitig zusammengebrochen ist, und die Verbindung deshalb als Unterbrochen angesehen werden muß. Der Fehlercode wird hierbei auf ECONNRESET gesetzt und der Socket geschlossen.

Falls die Gegenstelle nicht reagiert, und auch auf nun folgende Versuche ebenfalls nicht antwortet, wird der Fehlercode des Sockets auf ETIMEDOUT gesetzt und der Socket ebenfalls geschlossen.

8.1.5. SO_LINGER

Datentyp: `struct linger{}`

```
#include <sys/socket.h>

struct linger {
    int l_onoff;
    int l_linger;
}
```

Diese Option beeinflusst die Wirkung des Befehls `close()` für ein verbindugnsorientiertes Protokoll. Üblicherweise kehrt `close()` sofort zurück, stehen noch zu versendende Daten aus versucht das System diese im Hintergrund zuzustellen. Der Aufrufer bekommt hiervon nichts mit, insbesondere nicht wenn es zu einem Fehler kommt.

Durch das Setzen der Option SO_LINGER können folgende Szenarien entstehen:

1. `l_onoff` ist 0, der Wert von `l_linger` wird ignoriert, das Standardverhalten von `close()` kommt zum tragen, es kehrt unmittelbar zurück.

2. `l_onoff` ist ungleich 0 und `l_linger` ist gleich 0. Die Verbindung wird sofort geschlossen, und etwaige ausstehende Daten im Sendepuffer werden verworfen. Außerdem wird ein RST-Datagramm an den Partner gesendet, anstatt des üblichen Drei-Wege-Abbaus der Verbindung (siehe auch 4.1.3).

3. `l_onoff` ist ungleich 0 und `l_linger` ist ungleich 0. In diesem Fall versucht der Kernel die ausstehenden Daten zu versenden. Der Prozeß wird hierbei schlafen gelegt, bis entweder alle Daten versandt wurden, oder der durch `l_linger` angegebene Timeout abgelaufen ist. Der Rückgabewert von `close()` gibt an, ob der Timeout abgelaufen ist (Rückgabewert -1, Fehlercode EWOULDBLOCK) und Daten verworfen wurden, oder ob alle Daten versendet werden konnten (Rückgabewert 0).

   Ein Problem in diesem Zusammenhang ist, daß die Einheit des Werts `l_linger` nicht standardisiert ist. Manche Systeme rechnen hier mit 1/100 Sekunden, während POSIX.1g Sekunden vorschreibt. Außerdem ist ungewiss, wie viele signifikante Bit ausgewertet werden, von Berkeley abgeleitete Systeme verwenden beispielsweise intern eine vorzeichenbehaftete 16-Bit-Ganzzahl, womit die Zeit auf 32767 Sekunden beschränkt ist. [8]

8.1.6. SO_OOBINLINE

Datentyp: `int`

Ist diese Option gesetzt, werden Out-of-Band-Daten in der gleichen Warteschlange wie normale Daten gespeichert. Das Flag MSG_OOB kann dann nicht mehr benutzt werden.

8.1.7. SO_RCVBUF und SO_SNDBUF

Datentyp: `int`

Mit diesen beiden Optionen kann die Größe des Empfangs- und Sendepuffer eingestellt werden. Bei TCP wird die Größe des Empfangspuffers durch die Fenstergröße dem Verbindungspartner bekannt gegeben und sorgt für die Flußkontrolle („überstehende“ Daten werden gelöscht, und müssen später

erneut übertragen werden, siehe 4.1.4). Bei UDP gehen Daten, die im Empfangspuffer nicht mehr untergebracht werden können, einfach verloren.

Bei TCP ist es wichtig, daß die Fenstergröße bei einem Client vor dem Aufruf von `connect()` geändert werden muß, bei einem Server vor dem Aufruf von `listen()`.

Hinweis: Wenn man nicht weiß, was man tut, ist es eine gute Idee diese Werte nicht zu verändern.

8.1.8. SO_RCVLOWAT und SO_SNDLOWAT

Datentyp: `int`

Diese Optionen beeinflussen die sog. Niedrigwassermarken eines Sockets. Wird die Empfangs-Niedrigwassermarke erreicht, gibt `select()` „lesbar“ zurück. Der übliche Wert ist hier 1 Byte. Ähnliches gilt beim Senden: ist im Sendepuffer der Sockets so viel oder mehr Platz verfügbar, wie die Sende-Niedrigwassermarke angibt, erkennt `select()` den Socket als „beschreibbar“. Üblich sind hier 2048 Bytes für TCP-Sockets. Bei UDP-Sockets ändert sich der Platz im Sendepuffer niemals, da keine Kopien behalten werden. Ein UDP-Socket gilt also immer als beschreibbar, wenn der Platz im Sendepuffer größer als die angegebene Marke ist.

8.1.9. SO_RCVTIMEO und SO_SNDTIMEO

Datentyp: `struct timeval{}`

Mit diesen beiden Optionen kann der Timeout für Empfangs- und Versendefunktionen beeinflusst werden. Standardmäßig sind beide Timeouts deaktiviert, was dem Setzen beider Elemente auf 0 gleichkommt.

8.1.10. SO_REUSEADDR

Datentyp: `int`

Eine TCP-Verbindung wird durch einen lokalen und einen fernen Endpunkt eindeutig beschrieben. Beim Aufruf von `bind()` kann es jedoch passieren, daß eine lokale Adresse bereits von einem Socket benutzt wird, und der Aufruf deshalb fehlschlägt. Durch das Setzen dieser Option wird das Binden an eine bereits belegte Adresse dennoch erlaubt.

8.1.11. TCP_NODELAY

Datentyp: `int`

Mit dieser Option wird der *Nagle-Algorithmus* aktiviert bzw. deaktiviert. Er ist standardmäßig aktiviert, und reduziert die Anzahl der kleinen Pakete auf einem WAN. Hierbei werden *kleine* Pakete nur dann verschickt, wenn keine Quittungen mehr für solche kleinen Pakete ausstehen. Der Hintergedanke ist der, daß die Daten angesammelt werden, bis sie größere Pakete bilden, die dann mit weniger Overhead übertragen werden können (und somit auf weniger Quittungen gewartet werden muß).

Aufgrund der höheren Geschwindigkeit von LANs tritt der Nagle-Algorithmus hier selten in Aktion. Typische Anwendungen für kleine Pakete sind Telnet- oder rlogin-Clients, die nach Möglichkeit für jeden Tastendruck ein eigenes Paket versenden.

8.1.12. IP_HDRINCL

Datentyp: `int`

Diese Option tritt im Zusammenspiel mit Raw Sockets (siehe Kapitel 13) auf. Solange sie nicht gesetzt ist, stellt der Kernel den IP-Header selbst zusammen, und hängt die vom Benutzer angegebenen Daten als Nutzlast an. Wenn sie gesetzt ist, kann auch der IP-Header vom Benutzer selbst erzeugt werden, mit den folgenden Ausnahmen:

- Die Checksumme wird immer vom Kernel berechnet
- Steht das ID-Feld auf 0, wird es vom Kernel gesetzt
- Ist die Quell-IP-Adresse INADDR_ANY, wird sie auf die primäre IP-Adresse der ausgehenden Schnittstelle gesetzt

Außerdem beeinflusst die Option das Verhalten gegenüber IP-Optionen, die mit IP_OPTIONS gesetzt wurden. Manche Systeme hängen diese dennoch an den IP-Header an, andere wiederum erwarten, daß der eigene IP-Header sämtliche gewünschten Optionen bereits enthält.

8.1.13. IP_TTL

Datentyp: `int`

Diese Option legt den Wert für das TTL-Feld ausgehender Datagramme fest.

8.1.14. IP_OPTIONS

Datentyp: unterschiedlich

Mit dieser Option können IP-Optionen (vgl. Kapitel 3.5.1) gesetzt werden. Welche Optionen unterstützt werden hängt von der Implementierung ab.

8.1.15. IP_ADD_MEMBERSHIP

Datentyp: `struct ip_mreq{}`

```
#include <netinet/in.h>

struct ip_mreq
{
    struct in_addr imr_multiaddr;
    struct in_addr imr_interface;
};
```

Diese Option fügt eine Interface-Adresse und eine Mutlicast-Gruppenadresse in die Liste der Mitgliedschaften hinzu. Wird für `imr_interface` INADDR_ANY angegeben, wählt der Kernel automatisch eine passende Schnittstelle aus.

Für einen Socket kann diese Option mehrfach aufgerufen werden, jedoch muß sich dabei entweder `imr_multiaddr` oder `imr_interface` unterscheiden. Häufig begrenzt das System die Anzahl der möglichen Joins pro Socket.

8.1.16. IP_DROP_MEMBERSHIP

Datentyp: `struct ip_mreq{}`

Diese Option entfernt einen Eintrag aus der Liste der Multicast-Mitgliedschaften. Ist der Wert der Schnittstelle INADDR_ANY, wird das erste Vorkommen gelöscht. Wird ein Socket geschlossen, erlöschen automatisch alle Mitgliedschaften. Die Mitgliedschaft des Hosts selbst erlischt erst, wenn der letzte Prozeß die Gruppe verlassen hat.

8.1.17. IP_MULTICAST_IF

Datentyp: `struct in_addr{}`

Mit dieser Option wird die Standard-Schnittstelle für ausgehende Multicast-Datagramme festgelegt. Wird als Parameter INADDR_ANY verwendet, werden alle bisher festgelegten Schnittstellen gelöscht, und für jedes Datagramm wird erneut eine passende Schnittstelle ermittelt.

8.1.18. IP_MULTICAST_TTL

Datentyp: `u_char`

Setzt den Wert des TTL-Felds für ausgehende Multicast-Datagramme. Der Default-Wert ist 1, womit die Datagramme auf das lokale Subnetz beschränkt sind. Datentyp beachten!

8.1.19. IP_MULTICAST_LOOP

Datentyp: `u_char`

Aktiviert bzw. deaktiviert den Loopback für ausgehende Multicasts. Diese Option ist standardmäßig aktiviert, d.h. jedes versendete Datagramm wird auch lokal zugestellt, falls der Host zu der entsprechenden Multicast-Gruppe gehört. Datentyp beachten!

8.2. fcntl

Neben den Socket-Optionen, die über `getsockopt()` und `setsockopt()` manipuliert werden können, gibt es unter UNIX eine Reihe von Einstellungen, die auch für andere Dateideskriptoren möglich sind. Diese werden über die Funktion `fcntl()` erreicht:

```
#include <fcntl.h>

int fcntl(int fd, int cmd, ...);
```

Mit *fd* wird der Deskriptor bezeichnet, dessen Einstellungen manipuliert werden sollen. Die Art der Manipulation wird durch den zweiten Parameter, *cmd* bestimmt.

Der Rückgabewert der Funktion ist abhängig vom verwendeten Wert für *cmd*, jedoch stets -1 im Fehlerfall.

8.2.1. F_DUPFD

Mit F_DUPFP wird der übergebene Deskriptor dupliziert. Der Rückgabewert ist im Erfolgsfall der neue Deskriptor. Wird ein drittes Argument übergeben, so ist der neue Deskriptor der kleinste Wert größer als der übergebene.

Der neue Deskriptor referenziert das gleiche Objekt, und damit den gleichen Offset innerhalb einer Datei. Änderungen am Lesezeiger sind an allen Deskriptoren dieses Objekts sichtbar. Der Zugriffsmodus ist ebenfalls identisch mit dem des Deskriptors, der zur Erzeugung des Duplikats verwendet wurde. Weiterhin werden das close-on-exec-Flag sowie sämtliche andere Flags vererbt.

8.2.2. F_GETFD und F_SETFD

Das close-on-exec-Flag gibt an, ob ein Deskriptor nach einem Aufruf einer der `exec`-Funktionen geschlossen werden soll. Einen Anwendungsfall dafür kann man Kapitel 11.8 entnehmen.

Die Konstante FD_CLOEXEC ist eine Bitmaske, die genau ein Bit gesetzt hat. Wird sie als dritter Parameter bei einem Aufruf von `fcntl()` mit F_SETFD übergeben, wird das close-on-exec-Flag für den entsprechenden Deskriptor gesetzt. Um es zu löschen wird als Parameter 0 angegeben.

Analog dazu gibt ein Aufruf mit F_GETFD einen Wert zurück, der mit dieser Maske über ein logisches UND verknüpft werden kann. Ist das Ergebnis 0, wird der Deskriptor bei einem `exec` nicht geschlossen.

8.2.3. F_GETFL und F_SETFL

Ein Deskriptor verfügt über eine Reihe von Flags. Diese können über F_GETFL erfragt werden (Rückgabewert von `fcntl()`) und mittels F_SETFL veränder werden. Dabei ist es notwendig, einzelne Bits über Bitoperationen zu testen/löschen, und nicht mit direkten Vergleichen zu arbeiten.

Die bekannten Flags sind:

O_NONBLOCK	Systemaufrufe blockieren nicht, stattdessen wird -1 geliefert und errno auf EAGAIN gesetzt
O_APPEND	Schreibzugriffe erfolgen am Ende der Datei
O_DIRECT	Cache- und Puffereffekte werden reduziert bzw. nach Möglichkeit komplett eliminiert
O_ASYNC	ermöglicht die Zustellung des Signals SIGIO wenn Daten bereit sind

Beim Einsatz von O_NONBLOCK ist zu beachten, daß ständiges Testen der Deskriptoren auf Lesbarkeit eine höhere Belastung des Systems mit sich bringt. Wenn ungewiss ist, ob Daten gelesen werden können, oder etwa mehrere Deskriptoren überwacht werden müssen, ist es fast immer eine bessere Lösung `select()` einzusetzen (siehe Kapitel 9.2.1). Dennoch hier ein Beispiel zum Setzen und Löschen dieses Flags, da dies immer noch eine häufig gestellte Frage ist:

```
int flags;
...
/* setzen */
flags = fcntl(sock, F_GETFL);
flags |= O_NONBLOCK;
fcntl(sock, F_SETFL, flags);
...
/* loeschen */
flags = fcntl(sock, F_GETFL);
flags &= ~O_NONBLOCK;
fcntl(sock, F_SEFL, flags);
```

Der Einsatz von O_ASYNC erfordert, daß dem Deskriptor ein Besitzer zugeordnet ist, der das Signal SIGIO erhalten soll, wenn Daten gelesen werden können. Hierzu ist ein Aufruf von `fcntl()` mit F_SETOWN notwendig. Ein ähnliches Verfahren wird auch beim Lesen von Out-of-Band-Daten mittels SIGURG eingesetzt, das Beispielprogramm in Anhang A.3 mag als Vorlage dienen.

8.2.4. F_GETOWN und F_SETOWN

Ein Aufruf von `fcntl()` mit F_GETOWN liefert als Rückgabewert die Prozeß-ID (PID) desjenigen Prozesses, der die Signale SIGURG bzw. SIGIO erhält.

Mit F_SETOWN kann dieser Eigentümer gesetzt werden, die PID ist als dritter Parameter zu übergeben.

8.3. ioctl

Die dritte Gruppe interessanter Operationen auf Sockets stellen die durch die Funktion `ioctl()` zugängigen Informationen dar. Hierzu gehören insbesondere Schnittstelleninformationen wie beispielsweise IP-Adressen, Subnetzmasken etc., aber auch Informationen über den ARP-Cache. Sämtliche in diesem Kapitel aufgeführten Operationen sind nur unter UNIX durchführbar, eine Entsprechung für die Winsock-Implementierung von Windows mag existieren, ist dem Autor jedoch gänzlich unbekannt.

Die Funktion `ioctl()` hat die folgende Deklaration:

```
#include <sys/ioctl.h>

int ioctl(int d, unsigned long request, ...);
```

Über die Funktion `ioctl()` „spricht“ man mit dem Gerätetreiber. Üblicherweise ist daher der erste Parameter ein Dateideskriptor, den man nach dem Öffnen eines Geräts aus `/dev/` erhält. Um `ioctl()` auf Sockets anzuwenden setzt man einfach einen gültigen Socket ein. Manche der Befehle wirken jedoch auf alle Sockets bzw. systemweit.

Der Typ und die Art des dritten Arguments hängt von dem *request* ab. Im Folgenden werden die für Netzwerkbelange interessanten Requests vorgestellt, mit Name, Datentyp und Funktion. Eine genaue Beschreibung jeder einzelnen ist verlockend, aber würde zu einem sehr langen Kapitel führen. Stattdessen soll dies eher als eine Art Wegweiser fungieren, denn kennt man ersteinmal den Namen des passenden IOCTLs, ist es ziemlich einfach sich ein Beispielprogramm zu ergoogeln.

Name	Datentyp	Funktion
SIOCATMARK	int*	Liefert 0 wenn keine OOB-Daten vorliegen, andernfalls einen Wert ungleich 0
SIOCGPGRP	int*	Liefert die Prozeß-ID bzw. Prozeßgruppen-ID zurück, die zum Empfang der Signale SIGIO und SIGURG genutzt wird (wirkt wie fcntl() mit F_GETOWN)
SIOCSPGRP	int*	Setzt die Prozeß-ID bzw. Prozeßgruppen-ID für den Empfang von SIGIO und SIGURG (wirkt wie fcntl() mit F_SETOWN)
FIONBIO	int*	Setzt oder löscht das nicht-blockierende Flag
FIOASYNC	int*	Setzt oder löscht das Flag für den Empfang von SIGIO
FIONREAD	int*	Liefert die Anzahl Bytes im Empfangpuffer zurück
FIOSETOWN	int*	Äquivalent zu SIOCSPGRP für einen Socket
FIOGETOWN	int*	Äquivalent zu SIOCGPGRP für einen Socket
SIOCGIFCONF	struct ifconf*	Holt Schnittstellenkonfiguration ein
SIOCGIFADDR	struct ifreq*	liefert Unicast-Adresse in ifr_addr
SIOCSIFADDR	struct ifreq*	Setzt die Schnittstellenadresse aus ifr_addr
SIOCGIFFLAGS	struct ifreq*	Liefert Schnittstellen-Flags in ifr_flags
SIOCSIFFLAGS	struct ifreq*	Setzt Schnittstellen-Flags aus ifr_flags
SIOCGIFDSTADDR	struct ifreq*	Liefert Point-to-Point-Adresse in ifr_dstaddr
SIOCSIFDSTADDR	struct ifreq*	Setzt die Point-to-Point-Adresse aus ifr_dstaddr
SIOCGIFBRDADDR	struct ifreq*	Liefert Broadcast-Adresse in ifr_broadaddr
SIOCSIFBRDADDR	struct ifreq*	Setzt Broadcast-Adresse aus ifr_broadaddr
SIOCGIFNETMASK	struct ifreq*	Liefert die Subnetz-Maske von ifr_addr
SIOCSIFNETMASK	struct ifreq*	Setzt die Subnetz-Maske in ifr_addr
SIOCGIFMETRIC	struct ifreq*	Liefert die Schnittstellenmetrik in ifr_metric
SIOCSIFMETRIC	struct ifreq*	Setzt Schnittstellenmetrik aus ifr_metric
SIOCSARP	struct arpreq*	Fügt einen Eintrag in den ARP-Cache ein
SIOCDARP	struct arpreq*	Löscht einen Eintrag aus dem ARP-Cache
SIOCGARP	struct arpreq*	Liest einen Eintrag aus dem ARP-Cache
SIOCADDRT	struct rtentry*	Fügt einen Eintrag in die Routing-Tabelle ein
SIOCDELRT	struct rtentry*	Löscht einen Eintrag aus der Routing-Tabelle

Zu SIOCGIFCONF sollte noch erwähnt werden, daß das Element `ifc_buf` auf einen Datenbereich der Größe `ifc_buflen` zeigen muß, der die Daten aufnehmen soll. Diese sind dann als ein Array aus Elementen des Typs `struct ifreq{}` zu verstehen. Für die Aufrufe SIOCGIF... und SIOCSIF... sind ebenfalls solche Strukturen `ifreq{}` zu benutzen, wobei die Schnittstelle über ihren Namen identifiziert wird, d.h. das Element `ifr_name` muß entsprechend gesetzt werden.

Bei den Aufrufen zur Manipulation des ARP-Caches wird das Element `arp_pa` der Struktur `arpreq{}` zur Idenfitikation des entsprechenden Eintrags verwendet. Hierbei handelt es sich um eine Internet Socket-Adreßstruktur, in der die IP-Adresse des Eintrags enthalten ist.

Über den `ioctl()`-Mechanismus ist es nicht möglich, eine vollständige Liste aller ARP- oder Routing-Einträge zu erhalten. Hierfür gibt es ein Interface über `sysctl()`. Die genaue Funktionsweise kann der entsprechenden Manpage entnommen werden.

9. Server-Anwendungen

Die allgemeine Auffassung der Server/Client-Architektur besagt, daß der Server derjenige ist, der einen Dienst anbietet, während der Client diesen in Anspruch nehmen kann. Diese Beschreibung der Rollen besagt noch nichts über die technische Umsetzung. In der Socket-Programmierung bezeichnet man hingegen denjenigen Teil gerne als Server, der auf einem Socket auf eingehende Verbindungen wartet („lauscht“).

Dies sieht auf den ersten Blick nach Haarspalterei aus (auf den zweiten auch noch), aber ein Dienstanbieter kann durchaus aus Sicht der Sockets als Client auftreten. Besonders schön wird dies am Beispiel SMTP (Simple Mail Transfer Protocol) deutlich. Ein SMTP-Server, der eine E-Mail von einem Client erhält (hier also in der Server-Rolle) verbindet sich zu einem anderen SMTP-Server (nun als Client), um diesem die E-Mail zur weiteren Zustellung zu überlassen.

Um Verwirrungen vorzubeugen soll im Folgenden immer derjenige Teilnehmer als Server verstanden werden, der einen Socket mit `listen()` in den lauschenden Zustand versetzt, während Aufrufer von `connect()` als Client verstanden werden sollen.

9.1. Struktur und Aufbau

Als erstes Kriterium zur Unterscheidung der verschiedenen Serverarten kann man die Art des Aufrufs heranziehen. Gemeint ist damit wie der Prozeß zum Client kommt. In der Form des alleinstehenden Servers (*standalone*) versetzt der Server-Prozeß selbst einen (oder mehrere) Socket(s) in den lauschenden Zustand, und ergreift im Fall eingehender Verbindungen die notwendigen Maßnahmen. Im Gegensatz dazu steht die Variante, in der ein Superserver wie beispielsweise `inetd` oder `xinetd` eingesetzt wird. Dieser übernimmt die Aufgabe des Lauschens, und wenn eine Verbindung eingeht wird der passender Server-Prozeß erzeugt und mit dem Client „verkuppelt“. Die Vorteile dieses Verfahrens werden in 9.1.2 näher beleuchtet.

9.1.1. Alleinstehend

Der grobe Aufbau eines alleinstehenden Server ist folgender:

socket()	Anlegen eines Sockets
bind()	Zuweisen einer lokalen Protokolladresse
listen()	Versetzen in den Lauschmodus
accept()	Annehmen eines Clients
...	Funktionalität des Servers

Der Aufruf von `accept()` ist hierbei blockierend, sodaß der Prozeß solange schlafen gelegt wird, bis tatsächlich ein Client eingetroffen ist. Weitere Clients verbleiben solange in der Warteschlange, bis das nächste mal `accept()` aufgerufen wird. Um diesen Vorgang zu parallelisieren bedient man sich einer der Techniken, die in Abschnitt 9.2 vorgestellt werden.

Bei einem alleinstehenden Server gibt es (zunächst) nur einen Prozeß, der mit der Ressource *lauschender Socket* verknüpft ist. Dies hat insbesondere zur Folge, daß der Prozeß während der gesamten Laufzeit des Servers (also auch der Zeit, in der keine Clients behandelt werden) lebt, und weitere Ressourcen verbraucht (Speicher, Einträge in diversen Kernel-Strukturen usw. usf.). Dies ist bei der Verwendung eines Superservers (siehe 9.1.2) anders.

Weiterhin gilt zu beachten, daß Sockets mit Portnummern < 1024 in der Regel nur vom Superuser (`root`) angelegt werden können (genauer: angelegt werden die Sockets natürlich vorher, der Aufruf von `bind()` ist es, der dem normalen Nutzer versagt bleibt). Eine Möglichkeit, die weite Verbreitung gefunden hat, ist das SUID-Bit für die Anwendung zu setzen. Unter UNIX wird der entsprechende Prozeß dann mit den Rechten des *Eigentümers der Datei* ausgestattet, also beispielsweise `root`. Wenn der Prozeß dann alle Vorgänge abgeschlossen hat, für die diese Rechte benötigt werden, kann er seine effektive UID wieder auf die reale UID setzen, und somit die Rechte abgeben.

So schön dieses Verfahren auch aussieht, so undurchschaubar ist es für den Administrator des Systems. Kennt er die Anwendung nicht sehr genau, so weiß er nur, daß es eine Anwendung gibt, die (ihm) unbekannte Anweisungen enthält, die mit Rechten von `root` ausgeführt werden. Nach Gesichtspunkten der Sicherheit eine nicht sehr schöne Sache. Auch dieses Problem wird durch den Einsatz eines Superservers umgangen.

Soviel zu den Nachteilen, doch warum nutzt man dann überhaupt alleinstehende Server? Aus Geschwindigkeitsgründen. Ein Prozeß, der bereits existiert, kann wesentlich schneller auf einen Client reagieren, als wenn er erst von einem anderen Prozeß ins Leben gerufen werden muß.

Damit alleinstehende Server nicht ständig eine Konsole blockieren, werden sie oftmals als sog. *Daemons* gestartet. Das bedeutet, daß der Prozeß im Hintergrund, getrennt vom Kontrollterminal, vor sich hin werkelt. Hierzu sind nur wenige Schritte notwendig, die in Kapitel 11.6 vorgestellt werden.

Ein ausführlich kommentierter Quellcode eines alleinstehenden Servers ist in A.4 abgedruckt.

9.1.2. Mit einem Superserver

Wie bereits erwähnt stellt der Einsatz eines Superservers eine in mancherlei Hinsicht sehr wünschenswerte Alternative zum alleinstehenden Server dar. Ein Superserver ist ein Prozeß, der eine Reihe von Sockets zum Lauschen erzeugt und auf Client-Anfragen wartet. Treffen solche Anfragen ein, wird ein spezieller Prozeß erzeugt, der die eigentliche Funktionalität des Servers implementiert.

Dies hat drei Vorteile:

1. Es werden weniger Ressourcen unnötig gebunden
2. Die Vergabe von Rechten kann anders aussehen
3. Die Implementierung des Servers kann unspezifischer ausfallen

Während der Betrieb mehrerer untätiger Server-Prozesse unnötig Arbeitsspeicher (und auch andere Ressourcen) in Anspruch nimmt, ist es kein nennenswerter Aufwand, wenn ein Superserver auf einen weiteren Socket aufpassen muß. Es ist nicht unüblich, daß ein Superserver 20 oder mehr Sockets anlegt, möglicherweise tagelang keinen Client zu sehen bekommt, aber dennoch ständig und jederzeit bereit sein muß. Diese Strategie hat gegenüber 20 Prozessen, die faulenzend das System belasten (aber im Regelfall sowieso ausgeswappt werden), einen deutlichen Vorteil.

Der zweite Punkt, die Rechtevergabe, geht einher mit einer möglichen Trennung des Socket-Teils vom Server-Teil. Wird eine Portnummer < 1024 benötigt (wie für die meisten „gängigen“ Dienste wie FTP,

HTTP, SMTP etc. notwendig ist), so muß nur der Superserver über die Rechte dazu verfügen. Der eigentliche Server-Prozeß wird erst aufgerufen, wenn ein interessierter Client vorhanden ist, und dann mit der bestehenden Verbindung verknüpft. Dem Administrator eines Systems verschafft dies insoweit ebenfalls ein ruhigeres Gewissen, als daß der potenziell unbekannte Code des Servers zu keiner Zeit mit anderen Rechten laufen kann, als die vom Superserver zur Verfügung gestellten. Man legt für diese Prozesse häufig eigene Benutzer bzw. Gruppen an, die besondere Zugriffsrechte auf die Ressourcen besitzen, die sie manipulieren müssen. Beispiele häufig vorhandener Benutzer-Accounts für solche Zwecke sind `ftp`, `cvs` oder `news` (mit speziellen Rechten) sowie `nobody` und `daemon` (die in der Regel nichts dürfen).

Der dritte Punkt ist ebenfalls interessant für Anwender von Programmiersprachen, die als solche nicht für die Verwaltung von Sockets vorgesehen sind (Shell-Scripte beispielsweise). Die gängige Methode, eine bestehende Verbindung zu einem Client weiterzugeben, besteht in der Verknüpfung von *stdin*, *stdout* und ggf. *stderr* mit dem Socket. Das bedeutet: die eigentliche Server-Implementierung muß nur noch von der Standardeingabe lesen und auf die Standardausgabe schreiben können. Damit wird für Programme in C auch möglich, `fgets()` und `printf()` zu verwenden, was im Umgang mit textbasierten und in Zeilen unterteilten Protokollen eine nicht zu unterschätzende Erleichterung darstellen kann.

Tipp: um an die IP-Adresse des Clients zu kommen kann man `getpeername()` auf STDIN_FILENO aufrufen.

Ein Beispiel zum Einsatz von `inetd` befindet sich in Anhang A.5.

9.2. Parallele Server

In manchen Situationen ist es nicht akzeptabel Clients warten zu lassen. Das gilt insbesondere für zeitlich ausgedehnte Vorgänge wie das Ausliefern einer Datei beispielsweise. Es gibt gleich mehrere Ansätze zur Parallelisierung solcher Vorgänge, die sich im wesentlichen durch den Grad der Interaktionsmöglichkeiten der Clients untereinander unterscheiden.

Es gibt Fälle, in denen die Clients nichts gemeinsam haben oder haben müssen. Das Ausliefern einer Datei ist so ein Vorgang, selbst wenn alle Clients die gleiche Datei haben wollen, so müssen sie nicht untereinander darüber kommunizieren. Das andere Extrem stellt ein Chat-Server dar, hier ist das erklärte Ziel, daß jeder Client jedem Client eine Nachricht zukommen lassen kann, und zwar direkt ohne auf dem Server zwischenzuspeichern (wie etwa bei Foren oder dem Usenet).

Im Folgenden sollen drei Ansätze zur Parallelisierung vorgestellt werden.

9.2.1. Mit select

Die Anwendung von `select()` stellt eine der elegantesten Lösungen dar. Mit ihr kann eine starke Kopplung zwischen den Clients hergestellt werden. Alle Sockets verbleiben hierbei in einem Prozeß, inklusive dem lauschenden Socket. Sie werden sequentiell bedient, und zwar in einer vom Programmierer festzulegenden Art und Weise (dies erlaubt eine gute Kontrolle der Ressourcen und Prioritäten für verschiedene Clients).

Die zentrale Funktion `select()` ist folgenderweise deklariert:

```
#include <sys/select.h>

int select(int nfds, fd_set *readfds, fd_set *writefds,
    fd_set *exceptfds, struct timeval *timeout);
```

Außerdem existieren vier Makros zum Zugriff und zur Manipulation der Deskriptor-Sets:

```
FD_SET(fd, _fdset);
FD_CLR(fd, _fdset);
FD_ISSET(fd, _fdset);
FD_ZERO(_fdset);
```

Ein solches Deskriptor-Set (Datentyp `fd_set`) kann sich Sockets (und unter UNIX auch beliebige andere Dateideskriptoren) „merken“. Es ist eine Blackbox der jeweiligen Implementierung und sollte aus Portabilitätsgründen nicht manipuliert werden, außer über die Makros `FD_SET` (trägt einen Deskriptor ein), `FD_CLR` (löscht einen Deskriptor), `FD_ISSET` (testst ob ein Deskriptor gesetzt ist) und `FD_ZERO` (löscht alle Deskriptoren).

Die Funktion select kennt drei verschiedene Sets, oben bezeichnet als *readfds*, *writefds* und *exceptfds*. Die Deskriptoren des ersten Sets werden auf Lesbarkeit überprüft, d.h. ob Daten vorhanden sind und eine Funktion wie `recv()` aufgerufen werden kann, ohne daß sie blockieren würde. Analog dazu das zweite Set, hier wird auf Schreibbarkeit geprüft. Eine Sonderrolle spielt das dritte Set, als eine „Ausnahme“ werden OOB-Daten (Out-of-band, siehe 4.1.4) bezeichnet.

`select()` kann also dazu eingesetzt werden, das Blockieren in einem Aufruf von `recv()` und verwandten Funktionen zu verhindern. Durch die Möglichkeit, mehrere Deskriptoren in einem Set anzugeben, hat man hiermit die Möglichkeit mehrere Socket-Verbindungen auf Daten abzuprüfen.

Der Aufruf von `select()` selbst blockiert, abhängig vom letzten Argument *timeout*. Ist dieser Parameter NULL, wird *ewig* gewartet. Ist es ein Zeiger auf eine gültige Struktur `timeval{}` wird solange gewartet, wie dort angegeben. Stehen beide Elemente, `tv_sec` und `tv_usec`, auf 0, kehrt `select()` sofort zurück.

Es ist möglich durch Übergabe von NULL als Deskriptor-Sets und unter Verwendung des Elements `tv_usec` eine Sleep-Funktion mit besserer Auflösung als `sleep()` zu implementieren. Jedoch sollte man nicht von Mikrosekunden-Auflösung ausgehen, die tatsächliche Genauigkeit ist plattformabhängig und oftmals von weiteren Betriebsumständen (Multitasking) abhängig.

Besondere Bedeutung verdient der erste Parameter, *nfds*. **Dies ist der am häufigsten gemachte Fehler im Umgang mit select()**, denn hier muß der wertemäßig größte Deskriptor *plus eins* übergeben werden. Dies ist auch die einzige Situation, in der es unabdingbar ist den zahlenmäßigen Wert eines Sockets auszuwerten.

Die gängige Praxis ist es, die Clients (oder generell Socket-Verbindungen) in einer Art Liste (oder Baum etc.) zu verwalten, und entweder für jeden Aufruf von `select()` durch Vergleichen den größten Wert zu ermitteln, oder diesen Wert gesondert mitzuführen und bei der Annahme oder Verabschiedung von Clients zu aktualisieren. Hat man diesen Wert, kann er beim Aufruf von `select()` bequem übergeben werden – plus eins! (wirklich oft falsch gemacht).

Anhand des Rückgabewerts von `select()` kann entschieden werden, was zu tun ist. Der Wert ist die Anzahl der Deskriptoren, auf die das entsprechende Kriterium zutrifft. Das bedeutet implizit, daß ein Rückgabewert von 0 auf das Ablaufen des Timeouts zurückzuführen ist, und -1 einen Fehlerfall anzeigt (`errno` auswerten!).

Achtung: der Inhalt der Struktur `timeval{}` ist nach Rückkehr der Funktion implementierungsabhängig. Bei manchen Systemen steht hier die verbliebene Restzeit drin, bei anderen hingegen nur Quatsch. Um portable Programme zu erhalten, sollte vor *jedem* Aufruf von `select()` diese Struktur erneut mit sinnvollen Werten gefüllt werden, und nach dem Aufruf als uninitialisierte lokale Variable betrachtet werden.

Wenn ein Client die Verbindung beendet, ist der Socket lesbar, ein Aufruf von `recv()` jedoch liefert 0. Wird die Verbindung direkt hinter einer Portion Daten geschlossen, so kann der Verbindungsabbau

nicht ohne weiteres erkannt werden, sodaß tote Clients möglicherweise in der Liste verbleiben. Dieses Problem kann durch ein passendes Anwenderprotokoll gelöst werden, in dem der Client seinen Wunsch zum Verbindungsabbau ankündigt (wie bei FTP beispielsweise durch "QUIT").

Die sich nun quasi aufdrängende Grundstruktur eines Servers mit `select()` besteht aus einer Schleife, in der zunächst die Deskriptor-Sets vorbereitet werden, dann `select()` aufgerufen wird, und danach die Deskriptor-Sets ausgewertet werden. Der Quelltext eines solchen Programms ist in A.6 angegeben.

9.2.2. Mit Prozessen

Die Verwendung mehrerer unabhängiger Server bietet sich immer dann an, wenn die Clients untereinander nichts voneinander wissen müssen. Fileserver, Webserver, etc. sind die üblichen Anwendungsfälle. Wenn ein Client eintrifft wird ein neuer Prozeß erzeugt, der sich dann exklusiv um diesen einen Client kümmert. Durch die getrennten Prozesse ist eine Kommunikation untereinander nicht möglich, jedoch wirkt sich auch das Blockieren oder gar ein Absturz des Prozesses nicht auf die anderen aus.

Unter UNIX wird mit dem Befehl `fork()` ein neuer Prozeß erzeugt, unter Windows gibt es keine Entsprechung in diesem Sinne. Allerdings gibt es dort ein `CreateProcess()`, mit dem ein Programm als Prozeß gestartet werden kann, und über andere Methoden der Inter-Prozeß-Kommunikation (IPC) kann diesem der Socket mitgeteilt werden, es sei auf [10] verwiesen.

Der Elternprozeß sollte nach Annahme des Clients und nach erfolgtem `fork()` den Client-Socket schliessen, denn durch das Forken wird der Deskriptor dupliziert. Das bedeutet, daß die Verbindung erst dann tatsächlich geschlossen wird, wenn alle darauf verweisenden Deskriptoren geschlossen wurden. Bleibt dies beim Elternprozeß aus, kann die Verbindung zum Client nicht beendet werden, der Client „hängt" fest. Abgesehen davon kann ein Prozeß nicht beliebig viele Deskriptoren besitzen, sodaß in einem längeren Betrieb, in dem solche Handles weggeworfen werden, Ressourcenprobleme auftreten können (man spricht auch von einem *descriptor leak*). Das gleiche gilt natürlich auch für den lauschenden Socket im Kindprozeß, und generell für alle unbenötigten Deskriptoren.

Wenn ein Kindprozeß beendet wird, muß der Elternprozeß einige Aufräumarbeiten unternehmen. Unterbleibt dies, so fallen sog. *Zombies* an. Einen ausführlichen Einstieg in das Prozeß-Modell unter UNIX bietet Kapitel 11, ein Beispiel für einen parallelen Server mit `fork()` gibt es in Anhang A.7.

9.2.3. Mit Threads

Threads werden oftmals als „leichte Prozesse" bezeichnet. In der Tat weisen sie manche Parallelen zu Prozessen auf. Threads sind, wie der Name andeutet, voneinander unabhängige Ausführungsfäden innerhalb eines Prozesses. Sie sind jedoch nicht so stark getrennt wie einzelne Prozesse, denn der Adressraum wird gemeinsam genutzt, und damit auch globale und statische Variablen.

Achtung: Funktionen, die ein „Gedächtnis" besitzen (beispielsweise statische oder globale Variablen) können im Zusammenhang mit Threads zu unerwünschten Ergebnissen oder Seiteneffekten führen. Die passenden Stichworte sind *reentrant* und *thread-safe*. Oftmals werden auch entsprechende Versionen von Bibliotheken angeboten, die diese Eigenschaften aufweisen.

Unter UNIX werden Threads in Form von POSIX-Threads (*pthreads*) angeboten, unter Windows sind Threads sehr viel besser integriert und auch das Mittel der Wahl um parallele Vorgänge zu betreiben. Die Programmierung von Multithread-Anwendungen ist keineswegs trivial und bedarf einer sorgfältigen Planung, damit Ressourcen vernünftig genutzt werden, und bei gegenseitiger Abhängigkeit keine *Deadlocks* entstehen, sich die Threads also untereinander blockieren und jeder auf einen anderen

wartet. Belohnt wird der Aufwand durch kompakte Anwendungen, die sowohl eine hohe Kopplung zwischen den Fäden als auch echte Parallelität (soweit dies von der zugrundeliegenden Hardware unterstützt wird) ermöglichen.

Ein exzellentes Buch zur systemnahen Programmierung unter Windows ist [10], in dem sehr genau auf Probleme des Multi-Threadings eingegangen wird. Für einen Einstieg in die Programmierung mit POSIX-Threads könnte sich [37] als geeignet erweisen.

10. Client-Anwendungen

Analog zur Definition eines Servers ist ein Client aus Sicht der Socket-Ebene ein Prozeß, der sich mit einem anderen aktiv verbinden möchte, erkennbar durch einen Aufruf von `connect()`.

Bei Clients ist die Artenvielfalt nicht so groß, denn in der Regel wird nur ein Client-Prozeß gestartet, eine Parallelisierung ist nicht notwendig. Allerdings gibt es auch einige Anwendungen, die zur Steigerung der Geschwindigkeit mehrere Anfragen an Server gleichzeitig versenden. Das sind zum einen Webbrowser, die beispielsweise für jedes Bild, das in einer Seite eingebunden ist, einen weiteren Thread starten, damit alle Ressourcen, die zum Rendern der Seite benötigt werden, gleichzeitig vorhanden sind. Ein anderes Beispiel sind Downloadmanager, die parallel unterschiedliche Teile einer Datei downloaden (der entsprechende Zielserver muß dies unterstützen), um die Bandbreite der Verbindung vollständig auszunutzen, und etwaige Begrenzungen pro Verbindung seitens des Servers zu umgehen.

10.1. Struktur und Aufbau

Ein Client folgt meistens dem folgenden Aufbau:

socket()	Anlegen eines Sockets
...	Auflösen der Zieladresse
connect()	Aufbauen der Verbindung
...	Funktionalität des Clients

Das Auflösen der Zieladresse betrifft `inet_aton()` bzw. `inet_addr()` wenn es sich um IP-Adressen in der *dotted notation* handelt. Um Hostnamen aufzulösen steht `gethostbyname()` zur Verfügung. Diese Thematik wird in Kapitel 6.2 vorgestellt.

Nachdem die Zieladresse und Portnummer bekannt sind, kann ein Aufruf von `connect()` erfolgen. Dieser Aufruf ist blockierend und kehrt entweder erfolgreich zurück, wenn die Gegenstelle die Verbindung angenommen hat, oder kehrt mit -1 als Rückgabewert zurück, wenn es ein Problem gibt. Dies kann entweder bedeuten, daß das Zielsystem nicht erreichbar war, oder daß auf dem Zielsystem dieser Port nicht besetzt ist. Die genaue Fehlerangabe in `errno` gibt Aufschluß darüber.

Nach erfolgtem Verbindungsaufbau kann der Client seinem eigentlichen Geschäft nachgehen. Die Gestaltung dessen geht über den Rahmen dieses Dokumentes hinaus, jedoch kann der Leser aus dem Beispiel in A.8 einige nützliche Informationen entnehmen.

11. Prozesse und Dateideskriptoren unter UNIX

Als Prozeß bezeichnet man ein in der Ausführung befindliches Programm. Wenn ein Benutzer ein Programm startet, erzeugt das System hierfür einen neuen Prozeß, lädt in dessen Speicherbereich den auszuführenden Programmcode, und führt ihn aus. Dabei kann ein Prozeß nur ein Programm ausführen, ein Programm kann jedoch mehrere Prozesse umfassen. Außerdem kann ein Programm mehrfach gestartet werden, wobei es immer noch dasselbe Programm bleibt, jedoch mehrere Prozesse dazu existieren.

Es ist ungemein schwer, das Prozeß-Modell anschaulich zu erklären, jedoch wird es im Allgemeinen intuitiv klar, wenn man einige Zeit damit gearbeitet hat.

11.1. Start eines Prozesses

Die Ausführung eines C-Programms beginnt aus Sicht des Programmierers am Anfang der Funktion `main()`. Dem voran geht jedoch der sog. *Startup-Code*, der aus der Laufzeitumgebung der C-Bibliothek stammt. Dieser Startup-Code besorgt ein paar Dinge, wie beispielsweise das Anlegen der Variablen `argc` und `argv[]`, die `main()` als Parameter erhält.

Der Parameter `argc` enthält die Anzahl der Argumente, also die Anzahl der gültigen Elemente in `argv[]` (arg**c** für *count*, arg**v** für *value*). Dabei ist das letzte Argument, also `argv[argc]` (denn die Indizierung beginnt bei `argv[0]`), immer NULL.

In `argv[]` steht an erster Stelle (`argv[0]`) die Bezeichnung des Prozesses. Dies ist meistens der Name, unter dem das Programm aufgerufen wurde (z.B. `/usr/bin/perl` oder `./a.out`, aber auch `grep` wenn der Pfad in der Umgebungsvariable `$PATH` bekannt ist und das Programm ohne Pfadangabe aufgerufen wurde). Ab `argv[1]` folgen die Parameter, die dem Programm auf der Kommandozeile mitgegeben wurden. Man sieht deshalb häufig folgendes Konstrukt, um alle Parameter abzuarbeiten:

```
for (i = 1; i < argc; i++)
    do_something(argv[i]);
```

11.2. Beendigung eines Prozesses

Ein Prozeß wird beendet, wenn entweder das Ende der Ausführung erreicht ist (z.B. das Ende von `main()`), der Prozeß sich explizit selbst beendet (`exit()`, `abort()`), oder aber von außen beendet wird (z.B. vom Kernel, weil versucht wurde eine Speicheradresse zu beschreiben, die dies nicht zulässt).

Die Funktion `main()` ist immer mit dem Rückgabewert `int` versehen. Dieser Wert wird auch dem Prozeß mitgeteilt, von dem aus das Programm gestartet wurde. Er wird als *Exit-Status* bezeichnet, und ist in der Regel 0 für eine erfolgreiche Beendigung, während ein Wert ungleich 0 einen Fehlercode darstellt. Es zeugt von gutem Programmierstil, sich an diese Konvention zu halten.

In C99 wird zugesichert, daß `main()` ohne explizites `return 0` dennoch 0 zurückgibt. Um kompatibel zu Compilern zu bleiben, die dies nicht unterstützen (oder zu ANSI-C bzw. C90), sollte dennoch kein Gebrauch davon gemacht werden.

Ein Prozeß kann ebenfalls durch Aufruf von `exit()` beendet werden. Dies wirkt wie `return` auf der Ebene von `main()`. Jedoch sollte man bedenken, daß Code mit `exit()` anstatt von `return` nicht 1:1 als Unterfunktion übernommen werden kann, weshalb es nur verwendet werden sollte, wenn der Prozeß wirklich beendet werden soll. Dies ist zum Beispiel nach dem Abarbeiten einer Funktion eines Kindprozesses sinnvoll (siehe A.7).

Die Funktion `abort()` beendet einen Prozeß abnormal. Je nach Konfiguration des Systems wird hierbei auch ein Eintrag im Logfile und/oder ein Coredump erzeugt. Sies sollte nur einem fatalen Fehlerfall passieren, in dem der Entwickler etwas mit dem Coredump anfangen kann.

Es ist möglich Exit-Handler einzurichten, die beim (normalen) Beenden eines Prozesses via `return` von main oder `exit()` ausgeführt werden. Dies geschieht mit der Funktion `atexit()`, die als Parameter einen Zeiger auf eine void-void-Funktion (also `void` als Rückgabewert und ohne Parameter) erhält. Diese Mechanik wird allerdings in den seltensten Fällen verwendet.

11.3. Eigenschaften eines Prozesses

Ein Prozeß hat verschiedene Attribute und Eigenschaften, die ihn ausmachen. Als erstes sei die *Process ID*, PID, genannt. Jeder Prozeß hat eine einzigartige PID, die ihn genau identifiziert. Das Betriebssystem vergibt sie und garantiert, daß unter dieser ID der entsprechende Prozeß erkannt wird. Das ist insofern wichtig, als daß sämtliche Inter-Prozeß-Kommunikation (IPC, siehe 11.5) unter Nennung der PID geschieht.

Neben der PID besitzt jeder Prozeß eine PPID (Parent PID), eine UID (User ID, die Bezeichnung des Benutzers, der den Prozeß erzeugt hat) und eine GID (Group ID, die Gruppe der dieser Benutzer angehört). Diese können alle erfragt werden (die Funktionen lauten `getuid()`, `getpid()`, etc.), und die UID sowie GID können sogar geändert werden (die nötigen Berechtigungen vorausgesetzt; in der Regel ist dies nur `root` erlaubt).

Weiterhin besitzt jeder Prozeß ein *Environment*, Umgebungsvariablen. Diese werden auch vererbt, sodaß ein Prozeß über diese Weise von der Shell aus weitere Informationen überliefert bekommen kann. Zur Manipulation der Umgebungsvariablen stehen die Funktionen `getenv()` und `putenv()` zur Verfügung.

Als *Arbeitsverzeichnis* (auch *working directory* genannt) wird dasjenige Verzeichnis bezeichnet, aus dem der Prozeß erzeugt wurde. Bei Daemons (siehe 11.6) spricht man auch häufig davon, daß ein Prozeß in diesem Verzeichnis *lebt*. Das trifft es insofern recht gut, als daß neue Dateien ohne Pfadangabe in diesem Verzeichnis erzeugt werden, und Laufwerke, die auf ein Verzeichnis gemountet wurden, nicht wieder ungemountet werden können, solange nicht alle darin lebenden Prozesse beendet wurden (oder sie sich hinausbewegt haben). Das Arbeitsverzeichnis wird mit dem Befehl `chdir()` geändert und mit `getcwd()` (für *get current working directory*) in Erfahrung gebracht.

In weiteren Sinne kann man geöffnete Dateien und Sockets ebenfalls als Eigenschaften eines Prozesses verstehen, denn sie werden beibehalten, wenn ein neues Programmabbild mit dem existierenden Prozeß überlagert wird. So können die Dateihandles *stdin*, *stdout* und *stderr* beispielsweise mit einem Socket verbunden werden, wie es der Superserver `inted` macht (siehe auch 9.1.2).

11.4. Erzeugung eines Prozesses

Ein Prozeß wird durch Abspalten eines vorhandenen Prozesses erzeugt. Dieser Vorgang wird auch als Forken (engl. *to fork*: gabeln, verzweigen) bezeichnet, der dazugehörige Systemaufruf trägt den Namen `fork()`:

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

Beim Forken wird ein exaktes Duplikat des Prozesses erzeugt, das sich nur in der PID, PPID und einigen weiteren Details (wie z.B. die bisher verbrauchten Ressourcen) unterscheidet. Um genau zu sein, werden die folgenden Attribute vererbt:

- alle offenen Dateideskriptoren
- IDs (reale und effektive UID und GID, Session-ID, SUID, SGID)
- aktuelles Arbeitsverzeichnis
- Rootverzeichnis
- Umgebungsvariablen
- Ressourcen-Limits
- Signalhandler und Signalmaske
- Dateikreierungsmaske (umask)
- Kontrollterminal
- Shared Memory-Segmente

Der Aufruf von `fork()` erfolgt genau einmal, vom Elternprozeß aus, die Funktion kehrt jedoch *zweimal* zurück, nämlich im Elternprozeß und im Kindprozeß. Damit die Prozesse unterschieden werden können, erhält der Elternprozeß als Rückgabewert die PID des Kindprozesses, wohingegen der Kindprozeß den Rückgabewert 0 geliefert bekommt (seine PID und die PID seines Parents kann er selbst in Erfahrung bringen, der Elternprozeß hingegen hat keine Chance die PID seines Kinds zu erfahren, wenn er diesen Rückgabewert verwirft).

Erwähnenswert ist noch das Verhalten duplizierter Dateideskriptoren. Die Dateien werden erst dann tatsächlich geschlossen, wenn der letzte referenzierende Deskriptor geschlossen wird, d.h. sowohl im Eltern- als auch im Kindprozeß. Dies ist gerade bei Sockets wichtig, denn wenn der Elternprozeß den duplizierten Client-Socket „vergisst“, kann die Verbindung nicht geschlossen werden, und der Client „hängt“ am Elternprozeß fest. Außerdem ist zu bedenken, daß jeder Deskriptor einen Zeiger auf die Lese- bzw. Schreibposition in einer Datei beinhaltet. Wird von beiden über einen duplizierten Deskriptor geschrieben, so treten gegenseitige Überschreibungen innerhalb der Datei auf. Es ist deshalb eine gute Idee, ungenutzte Deskriptoren nach dem `fork()` zu schliessen.

Eltern haften für ihre Kinder! Wenn ein Kindprozeß beendet wird, wird der in Abschnitt 11.2 erwähnte Exit-Status vom System aufgehoben, damit der Elternprozeß diesen in Erfahrung bringen kann. Neben diesem Status werden auch weitere Informationen über den Ressourcenverbrauch aufbewahrt. Solange diese Informationen nicht eingeholt wurden, existiert noch ein Eintrag in der Prozeßtabelle (bei einem Aufruf von `ps` als Z für *Zombie* markiert). Der Elternprozeß muß nun durch einen Aufruf von `wait()` (es existieren noch weitere Varianten `waitpid()`, `wait3()` und `wait4()` mit weiteren Funktionalitäten) diese Informationen abfragen, damit der Zombie seine Ruhe findet.

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
```

Der Rückgabewert ist die PID des verstorbenen Kinds (der Aufruf blockiert solange, bis ein Kind gestorben ist!), in dem Parameter *status* werden Exit-Status sowie weitere Informationen hinterlegt. Zur Auswertung stehen einige Makros zur Verfügung, von denen nur WIFEXITED und WEXITSTATUS erklärt werden sollen, für die weiteren sollte ein Blick in die Manpage erfolgen (z.B. über [1]).

Wenn das Makro WIFEXITED *wahr* liefert, hat sich der Prozeß normal beendet. Dann, und nur dann, ist der von WEXITSTATUS ermittelte Wert der Rückgabewert, der von `main()` zurückgegeben wurde, bzw. bei `exit()` übergeben wurde. Ein Beispiel hierzu findet sich in Anhang A.7.

Wie bereits erwähnt handelt es sich bei `wait()` um einen blockierenden Aufruf. Damit die Anwendung nicht unnötig zum Stehen kommt, wird ein Elternprozeß über den Tods eines Kindes durch ein *Signal* informiert: SIGCHLD. Signal werden im nächsten Abschnitt vorgestellt. Wenn kein Elternprozeß existiert, wird der Prozeß `init` zum Vormund für den Kindprozeß, und behandelt damit auch das Abholen der Informationen zur Beendigung. Wird ein Prozeß erzeugt, der daraufhin gleich wieder einen Prozeß erzeugt und sich daraufhin beendet, sodaß die Behandlung von `init` übernommen wird, kann der eigentliche Elternprozeß (nun eigentlich Urgroßvater) von der Sorgfaltspflicht entbunden werden. Diese Technik nennt man auch *double fork*.

11.5. Kommunikation zwischen Prozessen

Kommunikation zwischen Prozessen wird als *Inter Process Communication*, IPC, bezeichnet. Man unterscheidet hierbei zwischen miteinander verwandten Prozessen und fremden Prozessen. Ein Mittel zur Kommunikation wurde bereits ausführlich vorgestellt: Sockets. Signale wurden bereits kurz angerissen, und werden aus zweckdienlichen Gründen genau eingeführt. Weitere Maßnahmen sind Shared Memory, Pipes, FIFOs sowie Semaphoren. Diese sollen hier jedoch nicht weiter erwähnt werden; Literatur hierzu ist [9].

Signale sind eine sehr einfache Art der Kommunikation. POSIX beschreibt rund 25 Signale, von denen die bekanntesten SIGINT, SIGTERM, SIGKILL, SIGSEGV und SIGCHLD sind. SIGINT erhält ein Prozeß, wenn beispielsweise auf der Konsole die Tastenkombination Strg+C gedrückt wurde, SIGTERM hingegen wenn es explizit versand wird um dem Prozeß den Hinweis zu geben, sich bitte zu beenden. SIGTERM geht beim Herunterfahren des Systems dem SIGKILL voraus, damit ein Prozeß noch etwaige Aufräumarbeiten unternehmen kann. SIGSEGV erfolgt bei einem illegalen Speicherzugriff (*segmentation violation*), und SIGCHLD wurde bereits im Zusammenhang mit `wait()` vorgestellt.

Eine komplette Liste der Signale findet sich meistens in der Manpage zu `kill`, dem Kommandozeilen-Programm mit dem ebenfalls jedes Signal versendet werden kann. Üblicherweise verwendet man `kill <pid>` um einen Prozeß zu beenden, wobei hier SIGTERM verschickt wird, und mit `-KILL` oder `-9` SIGKILL.

Um in einem Programm auf Signale reagieren zu können, muß ein sog. *Signal-Handler* installiert werden. Dies erfolgt mit dem Befehl `sigaction()` (ein altes Interface hierzu ist `signal()`, das jedoch nicht mehr benutzt werden sollte). Die Funktion sieht folgendermaßen aus:

```
#include <signal.h>

int sigaction(int sig, const struct sigaction *restrict act,
    struct sigaction *restrict oact);
```

Die Struktur `sigaction{}` besteht aus den vier Elementen `sa_handler`, `sa_flags`, `sa_mask` und `sa_sigaction`. Im einfachsten Fall werden nur die beiden ersten eingesetzt, wobei die anderen auf 0 bzw. NULL gesetzt werden sollten.

`sa_handler` erhält einen Zeiger auf den Signal-Handler, eine Funktion die `int` als Parameter erhält (nämlich das Signal, durch das sie ausgelöst wurde) und `void` als Rückgabewert besitzt. In `sa_flags` können verschiedene Flags gesetzt werden, die beispielsweise daß unterbrochene Systemaufrufe automatisch fortgesetzt werden sollen (SA_RESTART), eine berühmte Inkonsistenz im Verhalten von `signal()` auf unterschiedlichen Systemen.

Wird als dritter Parameter die Adresse einer weiteren Struktur `sigaction{}` übergeben, wird der alte Handler sowie die dazugehörigen Informationen dort gesichert. Hier darf NULL übergeben werden, wenn diese Daten nicht interessieren.

Ein Beispiel für den Einsatz von `sigaction()` befindet sich ebenfalls in Anhang A.7 und A.3.

11.6. Daemon-Prozesse

Eine besondere Art Prozesse sind die sog. *Daemons*. Dies sind Programme, die ihre Aufgaben im Hintergrund verrichten und nicht mit dem Benutzer in Interaktion treten sollen. Dies ist häufig für Server-Anwendungen der Fall. Ein Daemon wird einmalig vom Benutzer gestartet (meistens beim Hochfahren des Systems) und kommuniziert danach ausschließlich über IPC (Inter Process Communication, siehe vorangegangenes Kapitel), sowie beispielsweise mit dem Dateisystem, externer Hardware etc.

Um einen Daemon zu erzeugen spaltet die Anwendung zunächst einen Prozeß ab, mit `fork()`. Daraufhin wird ein Kindprozeß erzeugt, der später die Arbeit des Daemons verrichtet. Der Elternprozeß schreibt häufig die PID des Kindprozesses in eine Datei, um später noch mit diesem kommunizieren zu können, und beendet sich danach wieder, womit der Prozeß `init` die Verantwortung für den Daemonprozeß übernimmt.

Um sich von der Konsole zu lösen, ruft der Daemon `setsid()` auf, womit eine neue SessionID erzeugt wird. Dieser Vrogang wird auch als *detaching* bezeichnet. Die nicht länger benötigten Dateideskriptoren für *stdin*, *stdout* und *stderr* können daraufhin auch geschlossen werden. Kommunikation kann entweder über ein selbst geführtes Logfile erfolgen, oder aber über den Systemlogdienst `syslogd`, der im folgenden Kapitel kurz vorgestellt wird.

Ein Daemonprozeß sollte sein aktuelles Arbeitsverzeichnis mit einem Aufruf von `chdir()` dahingehend verändern, daß keine Funktionen des Dateisystems gestört werden. Häufig leben Daemons im Rootverzeichnis, oder in einem eigens für sie hergerichteten Verzeichnis. Außerdem sollte durch einen Aufruf von `umask()` die *Dateikreierungsmaske* zurückgesetzt werden. Diese Maske betrifft die Zugriffsrechte neu erzeugter Dateien und wird vom aufrufenden Prozeß geerbt.

Ein kurzes Beispiel für die notwendigen Schritte:

```
int daemonize(void)
{
    pid_t pid;

    pid = fork();

    /* fork fehlgeschlagen */
    if (pid < 0)
        return -1;

    /* Elternprozess */
    if (pid > 0)
        exit(0);
```

```
    /* Kindprozes */

    setsid();       /* Session uebernehmen */
    chdir("/");     /* Arbeitsverzeichnis wechseln */
    umask(0);       /* Maske zuruecksetzen */

    return 0;
}
```

Diese Funktion kann beispielsweise von `main()` aus aufgerufen werden. Sämtlicher Code danach wird nur noch vom Daemon ausgeführt (es sei denn es wird -1 zurückgegeben, weil `fork()` fehlgeschlagen ist).

Auf manchen Systemen wird eine Funktion `daemon()` angeboten, die genau dies alles erledigt. Sie ist jedoch implementierungsspezifisch und durch ihren Einsatz verliert das Programm die POSIX-Kompatibilität.

11.7. Der Systemlogdienst syslogd

Auf jedem UNIX-System gibt es in der Regel einen Daemon `syslogd`, der für das Schreiben des Systemlogfiles zuständig ist. Über die Konfigurationsdatei `/etc/syslogd.conf` wird das Verhalten gesteuert, insbesondere wie mit welchen Nachrichten zu verfahren ist. Üblicherweise werden dringende Meldungen auf die Konsole geschrieben, sowie für spezielle Dienste wie E-Mail oder Authentifikationen separate Logfiles geführt. Die Unterscheidung der Quelle bzw. eine Unterteilung in Gruppen erfolgt über die sog. *facility*. Auf einem NetBSD-System sind die folgenden Konstanten bekannt:

LOG_AUTH	LOG_AUTHPRIV	LOG_CRON	LOG_DAEMON
LOG_FTP	LOG_KERN	LOG_LPR	LOG_MAIL
LOG_NEWS	LOG_SYSLOG	LOG_USER	LOG_UUCP
LOG_LOCAL0	LOG_LOCAL1	LOG_LOCAL2	LOG_LOCAL3
LOG_LOCAL4	LOG_LOCAL5	LOG_LOCAL6	LOG_LOCAL7

Eine Beschreibung der Dringlichkeit erfolgt über den sog. *level*. Die Levels sind geordnet, sodaß das Festlegen einer Grenzgröße, oberhalb derer alle Meldungen beispielsweise auf der Konsole ausgegeben werden sollen, möglich ist. Die dazugehörigen Konstanten sind, in ihrer Priorität absteigend (wie man liest, von links nach rechts, dann die nächste Zeile):

LOG_EMERG	LOG_ALERT	LOG_CRIT	LOG_ERR
LOG_WARNING	LOG_NOTICE	LOG_INFO	LOG_DEBUG

Ein Eintrag in `/etc/syslogd.conf` könnte nun folgendermaßen aussehen:

```
*.err;kern.*;auth.notice;authpriv.none;mail.crit        /dev/console
```

Damit werden alle Meldungen mit LOG_ERR, alle von LOG_KERN, sowie die beiden Kombinationen bestehend aus `LOG_AUTH|LOG_NOTICE` und `LOG_MAIL|LOG_CRIT` auf die Konsole geschrieben. Durch die Angabe `authpriv.none` wird erreicht, daß keine Meldung von LOG_AUTHPRIV ausgegeben wird, auch nicht die Kombination `LOG_AUTHPRIV|LOG_ERR`, die nach der ersten Regel erfasst worden wäre.

Die Übermittlung einer Nachricht an das Systemlogfile erfolgt über einen Aufruf von `syslog()` bzw. `vsyslog()`, der über einen Formatstring ähnlich dem von `printf()` beliebige Informationen formatiert ausgeben kann. Den beiden Funktionen wird eine *priority* als erster Parameter angegeben, ein Wert der aus einer *facility* und einem *level* durch bitweises ODER erzeugt werden kann. Zusätzlich dazu kann ein einmaliger Aufruf von `openlog()` erfolgen, der einen Defaultwert für die *facility* festlegen kann (sodaß bei Aufrufen von `syslog()` bzw. `vsyslog()` nur noch ein *level* angegeben werden muß), sowie eine Zeichenkette festlegt, die jedem Logeintrag vorangestellt wird. Symmetrisch zu `openlog()` muß ein Aufruf von `closelog()` erfolgen.

Achtung: der Einsatz `chroot()` erfordert möglicherweise eine Anpassung der `syslogd`-Konfiguration.

11.8. Sockets und Dateideskriptoren

An vielen Stellen wurden schon Tricks vorgestellt, in denen Sockets wie Dateideskriptoren arbeiten, und auch über die Vergabe der numerischen Werte wurden schon Mutmaßungen angestellt. In diesem Kapitel sollen sie zusammengetragen, und noch ein paar weitere Hinweise gegeben werden.

Unter UNIX ist ein Dateideskriptor eine kleine, nicht-negative Zahl. Bei einem normalen Prozeß sind die Deskriptoren 0, 1 und 2 bereits vergeben, nämlich für die Standardeingabe (*stdin*), Standardausgabe (*stdout*) und den Fehlerkanal (*stderr*). Diese Deskriptoren haben deshalb auch symbolische Namen: STDIN_FILENO, STDOUT_FILENO und STDERR_FILENO.

Dateideskriptoren die von `open()` geliefert wurden, haben Zugriffsrechte. Sie könnnen z.B. nur lesbar, nur beschreibbar, oder les- und beschreibbar sein. Dies wird beim Öffnen bereits festgelegt (O_RDONLY, O_WRONLY und O_RDWR). Weiterhin gibt es Pipes, die zur Kommunikation zwischen Prozessen dienen. Diese haben stets ein lesbares und ein beschreibbares Ende.

Ein verbundener Socket vom Typ SOCK_STREAM verhält sich (sofern nicht mit `shutdown()` beeinflusst) wie ein Dateideskriptor, von dem gelesen und auf den geschrieben werden kann. Deshalb lassen sich neben `send()` und `recv()` auch die Funktionen `read()` und `write()` verwenden, die sich nur in ihrem fehlenden vierten Argument, den Flags, unterscheiden.

Durch diese Verwandschaft können auf Sockets auch diverse Funktionen aufgerufen werden, die sonst für Dateien gedacht sind. Ein bereits ausführlich behandeltes Beispiel ist die Funktion `fcntl()` aus Kapitel 8.2. Eine weitere, manchmal sehr praktische Funktion ist `fdopen()`:

```
FILE *fdopen(int fildes, const char *mode);
```

Durch ein „Öffnen“ des Sockets mit dieser Funktion wird er für die gesamten Standard I/O-Befehle wie `fprintf()` nutzbar. Dies ist natürlich sehr komfortabel, und gerade für textbasierte Protokolle, die nochdazu in Zeilen unterteilt sind, bietet sich `fgets()` geradezu an. Vorsicht ist jedoch beim Abwechseln von Standard I/O und elementarer I/O wie `read()` und `write()` geboten, da eine Pufferung dazwischen liegt. Hier verwendet man besser zunächst `fflush()` und dann `fwrite()` bzw. `fread()`.

Trotz der Verwandschaft gibt es natürlich entscheidende Unterschiede zwischen Sockets und Dateien. So ist zum Beispiel der Einsatz von `fseek()` oder `ftell()` nicht sinnvoll, und liefert im günstigsten Fall sinnlose Ergebnisse.

Ein weiteres nettes Detail, das bisher noch nicht erwähnt wurde, ist das Verknüpfen von Sockets mit der Standard Ein- und Ausgabe anderer Programme. Mit den Systembefehlen `dup()` sowie `dup2()` lassen sich Deskriptoren duplizieren. Dabei wird dem neuen Deskriptor der niedrigste freie Wert zugewiesen. Wenn die Deskriptoren 0, 1 und 2 nun explizit geschlossen werden, werden nach dreimaligem `dup()` die Kanäle für *stdin*, *stdout* und *stderr* mit dem duplizierten Deskriptor verbunden. Wenn danach

eine Funktion der `exec`-Familie aufgerufen wird, startet das neue Programm ohne jegliche Kenntnis darüber, und schreibt auf den TCP-Socket, bzw. liest von ihm. Dieses Verfahren verwendet auch der Superserver `inetd`, der in Kapitel 9.1.2 vorgestellt wurde. Hinweis: eine Datei, eine Pipe und ein Socket werden erst dann tatsächlich geschlossen, wenn *alle* Deskriptoren geschlossen wurden, also auch alle Duplikate und das Original. Bei Dateien kommt hinzu, daß das Löschen erst dann stattfindet, wenn der letzte Prozeß diese Datei geschlossen hat.

Auf ähnliche Weise kann auch ein Webserver das Ausführen von CGI-Programmen ermöglichen. Hierzu ist es allerdings notwendig, daß die Kommunikation zu dem Kindprozeß über Pipes erfolgt, damit der Server die auszugebenden Daten in das HTTP-Protokoll einbetten kann.

Zur Illustration folgt ein kleines Beispiel einer Funktion, die den laufenden Prozeß mit dem Programm `uptime` überlagert, sodaß die Ausgabe der Uptime des Hostsystems über das Netzwerk verschickt wird. Dabei sei *sock* ein bereits mit einem Client verbundener Socket.

```
void execute(int sock)
{
    int i;

    for (i = 0; i < 3; i++)
    {
        close(i);
        dup(sock);
    }

    close(sock);

    execl("/usr/bin/uptime", "uptime", NULL);
}
```

Für den Aufrufer ergibt sich dann folgendes Bild:

```
$ nc localhost 1234
 9:51PM  up 19 days, 11:47, 3 users, load averages: 0.00, 0.01, 0.00
```

12. Broadcast und Multicast

Bisher wurde nur die *Unicast*-Kommunikation betrachtet. Das bedeutet, daß die Datagramme genau eine Zieladresse hatten, nämlich die bei `connect()` bzw. `sendto()` angegebene. Je nach verwendetem Physical Layer (vgl. Kapitel 1.2) können weiterhin *Broadcasts* und *Multicasts* ermöglicht werden. Unterstützt werden beide beispielsweise von Ethernet, Token Ring oder FDDI.

Broadcast und Multicast benötigen zwingend UDP als Protokoll. TCP unterstützt ausschließlich Unicast.

12.1. Broadcast

Broadcast-Meldungen werden von *jedem* Teilnehmer empfangen. Wie weit sie reichen hängt von der Konfiguration der Router, die lokale Netze verbinden, ab. Üblicherweise verbleibt ein Broadcast auf dem physikalischen Netz, in dem er ausgesandt wurde. Der Vorteil einer Broadcast-Meldung gegenüber mehreren Unicast-Meldungen besteht darin, daß der Verkehr auf dem Netzwerk kleiner gehalten wird. Es muß schließlich nur eine einzige Meldung versendet werden, anstatt viele, bis auf die Adresse identische Meldungen. Ein gravierender Nachteil bei Broadcasts ist, daß *jeder* Host die Meldungen aufnehmen und bearbeiten muß, auch wenn er nicht an ihnen interessiert ist. Dazu müssen sie so weit bearbeitet werden, daß diese Entscheidung getroffen werden kann. Dabei ergeben sich die folgenden Fälle:

- Der Host verfügt über eine UDP-Implementierung, jedoch lauscht auf dem entsprechenden Port keine Anwendung. Die Meldungen müssen die Netzwerkschicht, die Internetschicht und die Transportschicht durchlaufen, ehe sie verworfen werden können.

- Der Host verfügt über keine UDP-Implementierung, jedoch über IP. Die Meldungen müssen durch die Netzwerkschicht in die Internetschicht gereicht werden, damit diese sie verwerfen kann.

- Der Host verfügt über keine IP-Implementierung. In diesem Fall müssen die Meldungen noch immer vom Netzwerk über den Treiber zur Netzwerkschicht gelangen, damit diese die Frames verwerfen kann.

Dies führt zu einer unnötigen Belastung, die je nach Datendurchsatz durchaus erheblich sein kann.

Wenn eine Anwendung eine Broadcast-Meldung versenden möchte, muß als Zieladresse die Broadcast-Adresse des lokalen Netzes verwendet werden. Diese zeichnet sich dadurch aus, daß im Hostanteil der Adresse alle Bits auf 1 stehen (vgl. Kapitel 3.1). Außerdem muß für den Socket die Socket-Option SO_BROADCAST (siehe 8.1.1 gesetzt worden sein.

Jeder Host im Netz wird dieses Datagramm empfangen, das schließt den sendenen Host mit ein. In Anhang A.9 finden sich zwei Beispielprogramme, die einen UDP-Broadcast senden bzw. empfangen können.

12.2. Multicast

Broadcasts haben den entscheidenden Nachteil, daß auch unbeteiligte Hosts in die Verarbeitung der Datagramme involviert werden. Dieses Problem soll durch Multicasts gelöst werden. Bei Multicast-Sendungen wird auf Hardwareebene (Ethernet, Token Ring, FDDI, ...) kenntlich gemacht, ob ein Host ein Datagramm annehmen und bearbeiten muß, oder nicht. Dadurch werden unbeteiligte Hosts nicht „belästigt“, und solche Datagramme können auch guten Gewissens geroutet werden, und damit auch Systeme außerhalb des LANs erreichen.

Die grundlegende Idee ist die IP-Adressen der Klasse D (von 224.0.0.0 bis 239.255.255.255, siehe auch 3.1) als sog. *Gruppenadressen* zu verwenden, wobei die untersten 28 Bit als *Gruppen ID* bezeichnet werden. Sie beschreiben das Ziel eines Multicast-Datagramms, und jeder Host, auf dem diese empfangen werden sollen, trägt sich in die entsprechende Gruppe ein. Die tieferliegenden Schichten werden informiert, daß passende Datagramme anzunehmen sind, und Router werden durch IGMP (Internet Group Message Protocol [45]) angewiesen, Datagramme die zu dieser Gruppe gehören in das entsprechende Netz „zu holen“.

Diese Lösung erfordert, daß Gruppenadressen auf Hardwareadressen abgebildet werden können. Für IPv4 existieren RFCs für Ethernet ([39]), Token Ring ([40]) und FDDI ([41]) als Netzwerkschicht, für IPv6 gibt es welche für Ethernet ([42]) und FDDI ([43]). Als Beispiel soll IPv4 auf Ethernet dienen. Im Ethernet haben die Adressen eine Länge von 48 Bit, also sechs 8-Bit-Bytes. Die ersten drei Bytes haben für Multicast-Frames die Werte 0x01, 0x00 und 0x5e. Vom vierten Byte ist das oberste Bit immer 0, die restlichen sieben Bits bilden mit den 16 Bits der beiden letzten Bytes ein 23 Bit langes Feld, in welches die 23 niedrigwertigsten Bits der IPv4-Multicast ID eingetragen werden.

Bei den Multicast-Adressen sind zunächst drei spezielle Adressen bzw. Adressbereiche bekannt:

- 224.0.0.1 bezeichnet die *all-hosts*-Gruppe, auf diese Datagramme reagieren alle Multicast-fähigen Hosts
- 224.0.0.2 bezeichnet die *all-routers*-Gruppe, der alle Multicast-fähigen Router angehören
- 224.0.0.0 bis 244.0.0.255 gelten als *link local*, d.h. die Datagramme werden nicht geroutet

IPv4 kennt jedoch weitere Gruppen, die aus einer Notlage heraus gebildet wurden, nämlich der Beschreibung des *Gültigkeitsbereichs* der Datagramme. Während in IPv6 ein 4 Bit der Multicast-Adresse das sog. *scope*-Feld bilden, muß in IPv4 auf eine andere Art unterschieden werden. Ursprünglich hat man dazu das TTL-Feld verwendet, was jedoch zu Mißverständnissen führte, und deshalb zugunsten einer besseren Lösung aufgegeben wurde. Diese Lösung sieht die Unterteilung der Gruppen IDs in vier verschiedene Gültigkeitsbereiche vor. Die folgende Tabelle aus [8] gibt einen Überblick über die verschiedenen Gültigkeitsbereiche:

Gültigkeitsbereich	IPv6 scope-Feld	IPv4 TTL-Feld	IPv4 Adressbereich
Knoten-lokal	1	0	
Link-lokal	2	1	224.0.0.0 - 224.0.0.255
Site-lokal	5	< 32	239.255.0.0 - 239.255.255.255
organisationslokal	8		239.192.0.0 - 239.195.255.255
global	14	< 255	224.0.1.0 bis 238.255.255.255

Im ursprünglichen TTL-Modell gab es noch „Region-lokal“ und „Kontinental-loka“, die bei diesem Modell in „organisationslokal“ aufgegangen sind.

Die Realisierung von Multicast-Sendungen für BSD-Sockets erfolgt über spezielle Socket-Optionen. Sie wurden ab Seite 87 in Abschnitt 8.1 vorgestellt. Beim Senden ist zu beachten, daß die Absenderadresse

keine Multicast-Adresse sein darf (vgl. [44]). Deshalb wird man meistens zwei Sockets anlegen, einen zum Empfangen (der an die Multicast-Adresse gebunden wird) und einen zum Versenden (der nicht an eine Multicast-Adresse gebunden werden darf).

13. Raw Sockets

Raw Sockets ermöglich einerseits das Empfangen von Datagrammen, die man mit regulären Sockets nicht erhalten kann, und andererseits das Versenden von selbst aufgebauten Datagrammen. Die Möglichkeiten im Einzelnen sind:

- ICMP und IGMP Datagramme empfangen und senden
- Eigene Protokolle auf IP aufbauend entwickeln
- Komplette Datagramme inklusive IP-Header selbst erzeugen

Ein Raw Sockest wird wie jeder andere Socket durch einen Aufruf von `socket()` erzeugt. Als zweiten Parameter gibt man SOCK_RAW an. Mit dem dritten Parameter kann ein Protokoll ausgewählt, oder 0 für ein beliebiges eingesetzt werden. Die mit IPPROTO_ beginnenden Konstanten sind in der Header-Datei `<netinet/in.h>` enthalten. Sie entsprechen den Werten, die im IP-Header in das Protokollfeld eingetragen werden, also beispielsweise 6 für TCP, 17 für UDP usw.

Die Befehle `bind()` und `connect()` haben für Raw Sockets auch eine Bedeutung, werden jedoch selten verwendet. Mit `bind()` wird die lokale Adresse festgelegt. Diese wird als Quelladresse für ausgehende Datagramme eingesetzt. Die Funktion `connect()` hingegen setzt die ferne Adresse. Bei ausgehenden Datagrammen wird diese Adresse als Zieladresse eingesetzt. Damit ist der Einsatz von `send()` bzw. `write()` anstelle von `sendto()` möglich. In beiden Fällen wird die Komponente `sin_port` ignoriert; Raw Sockets kennen das Konzept der Ports nicht.

Die dem Socket übergebenen Daten werden vom Kernel durch einen IP-Header ergänzt, und die beim Anlegen des Sockets angegebene Protokoll-Nummer wird in das entsprechende Feld eingesetzt. Dies kann durch die auf Seite 87 vorgestellte Socket-Option IP_HDRINCL verhindert werden. Wenn sie gesetzt wurde, nimmt der Kernel das Datagramm in der Form entgegen, wie es `sendto()` übergeben wurde. Jedoch werden einige Header-Felder durch den Kernel geändert:

- Die Checksumme wird berechnet und eingesetzt
- Das ID-Feld wird vom Kernel gesetzt, sofern es auf 0 steht
- Falls die Quell-IP-Adresse auf INADDR_ANY steht, wird sie gesetzt

Aufgrund mangelnder Spezifikationen ist nicht klar geregelt, welche Byte Order für mehrbytige Felder verwendet werden soll. Bei Linux liegen alle Felder in Network Byte Order vor, während bei den von Berkeley abgeleiteten Implementierungen die Felder `ip_len` und `ip_off` in Host Byte Order vorliegen (für eingehende Pakete kommt `ip_id` noch hinzu).

Wenn die Option IP_HDRINCL nicht gesetzt ist, wird vom Kernel der IP-Header automatisch erzeugt und den übergebenen Daten vorangestellt. Im Protokollfeld wird der Wert eingetragen, der beim Anlegen des Sockets als drittes Argument übergeben wurde.

Das Empfangen von Datagrammen geschieht über den Befehl `recvfrom()`. Wurde der Socket mit `bind()` an eine lokale Adreses gebunden, werden Datagramme mit einer davon abweichenden Zieladresse verworfen. Die Quelladresse des Datagramms muß der durch `connect()` gesetzten fernen Adresse

entsprechen. Außerdem filtert ein Raw Socket all jene Datagramme aus, deren Protokoll-Feld im IP-Header nicht dem dritten Argument von `socket()` entspricht. Das ganze heißt auch: wurde für einen Raw Socket weder `bind()` noch `connect()` aufgerufen, und 0 als Protokoll angegeben, empfängt er alle Datagramme, die er bekommen kann. Dies sind nicht alle Datagramme, die das System empfängt, wie folgende Ausnahmen zeigen:

- TCP und UDP-Datagramme werden *niemals* empfangen
- ICMP-Datagramme werden vom Kernel abgearbeitet und dann übergeben, nicht jedoch Echo-, Timestamp- und Information-Request.
- IGMP-Datagramme werden vom Kernel abgearbeitet und dann übergeben
- Datagramme mit dem Kernel unbekanntem Protokollfeld werden übergeben

In all diesen Fällen werden die Prüfsummen beachtet, und nur intakte Datagramme finden ihren Weg zum Socket. Sollten sie fragmentiert eintreffen, kehrt der Leseaufruf erst dann zurück, wenn das Datagramm vollständig zusammengesetzt worden ist. Der IP-Header wird stets mit übergeben. Achtung: die Byte Order mancher Felder ist nicht notwendigerweise Network Byte Order, siehe obenstehende Bemerkung hierzu.

Wenn in einem System mehrere Raw Sockets existieren, erhält *jeder* eine Kopie des entsprechenden Datagramms.

Beim Erzeugen der Datagramme sollte bedacht werden, daß ein Compiler je nach Einstellung die Elemente einer Struktur an Speichergrenzen ausrichten darf, um einen schnelleren Zugriff zu ermöglichen. Wenn nun eine Struktur zum Zusammenbau eines Datagramms dienen soll, ist dies im Allgemeinen unerwünscht und kann zu Problemen führen. Hier will man die Elemente an Bytegrenzen ausgerichtet haben. Die meisten Compiler unterstützen dies durch `#pragma`-Anweisungen an den Präprozessor, oder mit Parametern auf der Kommandozeile. Die Dokumentation des Compilers sollte Aufschluß darüber geben.

Für die dem System bekannten Dataramme existieren Strukturen in den Header-Dateien. Unter NetBSD beispielsweise existieren `<netinet/ip.h>`, `<netinet/ip_icmp.h>`, `<netinet/tcp.h>` usw. Unter Linux (bzw. mit der GNU C Library glibc) ist zu beachten, daß es zwei Sätze von Strukturen gibt. Eine mit den Linux-typischen Benennungen (`struct iphdr{}`, `struct tcphdr{}`, usw.) und eine mit BSD-typischen (`struct ip{}`, `struct tcp{}`, usw.). Durch das Definieren von __FAVOR_BSD bzw. __USE_BSD (der genaue Name hat sich in der Vergangenheit geändert, und es ist eine gute Idee im konkret vorliegenden Header nachzusehen, wie es tatsächlich heißt) wird unter Linux das Verwenden der BSD-typischen Strukturen möglich. Umgekehrt gibt es keine Möglichkeit, unter BSD die Linux-typischen Strukturen zu nutzen.

Um die größten Fallstricke vorwegzunehmen, werden in den beiden folgenden Abschnitten die Header-Strukturen für IP und TCP sowohl in der Linux- als auch BSD-Variante vorgestellt. Insbesondere das Setzen von Flags sowie die Formate der Adressen sind abweichend und können für Verwirrung sorgen. Etwaige `#ifdef`-Anweisungen wurden ausgewertet, und zwar unter der Annahme, daß es sich um ein Little Endian System handelt, wie beispielsweise i386.

13.1. Der IP-Header

Der größte inhaltliche Unterschied ist hier, daß bei Linux die Source- und Destination Address als `u_int32_t` aufgefasst werden, während bei BSD eine `struct in_addr{}` vorliegt. Diese beiden Darstellungen sind selbstverständlich binärkompatibel, jedoch bei der Zuweisung muß der entsprechende Datentyp beachtet werden.

Bei BSD werden zusätzlich die Bitmasken und Flags als Konstanten definiert, bei Linux ist dies nicht der Fall.

IP-Optionen werden bei beiden Systemen in separaten Strukturen verwaltet. Sie unterscheiden sich ebenfalls, und die GNU C Library bietet auch hier beide Varianten an, während unter BSD üblicherweise nur die BSD-Variante vorhanden ist.

```
struct iphdr
  {
    unsigned int ihl:4;
    unsigned int version:4;
    u_int8_t tos;
    u_int16_t tot_len;
    u_int16_t id;
    u_int16_t frag_off;
    u_int8_t ttl;
    u_int8_t protocol;
    u_int16_t check;
    u_int32_t saddr;
    u_int32_t daddr;
    /*The options start here. */
  };
```

```
struct ip {
        u_int8_t  ip_hl:4,                /* header length */
                  ip_v:4;                 /* version */
        u_int8_t  ip_tos;                 /* type of service */
        u_int16_t ip_len;                 /* total length */
        u_int16_t ip_id;                  /* identification */
        u_int16_t ip_off;                 /* fragment offset field */
#define IP_RF 0x8000                      /* reserved fragment flag */
#define IP_EF 0x8000                      /* evil flag, per RFC 3514 */
#define IP_DF 0x4000                      /* dont fragment flag */
#define IP_MF 0x2000                      /* more fragments flag */
#define IP_OFFMASK 0x1fff                 /* mask for fragmenting bits */
        u_int8_t  ip_ttl;                 /* time to live */
        u_int8_t  ip_p;                   /* protocol */
        u_int16_t ip_sum;                 /* checksum */
        struct    in_addr ip_src, ip_dst; /* source and dest address */
} __attribute__((__packed__));
```

13.2. Der TCP-Header

Beim TCP-Header in der Linux-typischen Schreibweise hat jedes Flag (FIN, SYN, RST, ...) ein eigenes Element in Form eines Bits, das auf 0 oder 1 stehen kann. In der BSD-Variante hingegen übernimmt diese Funktion das 8 Bit lange Element `th_flags` und die Flags sind als Konstanten definiert, die durch Bitoperationen gesetzt und gelöscht werden können.

```
struct tcphdr
  {
    u_int16_t source;
    u_int16_t dest;
    u_int32_t seq;
    u_int32_t ack_seq;
    u_int16_t res1:4;
    u_int16_t doff:4;
    u_int16_t fin:1;
```

```
    u_int16_t syn:1;
    u_int16_t rst:1;
    u_int16_t psh:1;
    u_int16_t ack:1;
    u_int16_t urg:1;
    u_int16_t res2:2;
    u_int16_t window;
    u_int16_t check;
    u_int16_t urg_ptr;
};
```

```
typedef u_int32_t tcp_seq;

struct tcphdr {
        u_int16_t th_sport;             /* source port */
        u_int16_t th_dport;             /* destination port */
        tcp_seq   th_seq;               /* sequence number */
        tcp_seq   th_ack;               /* acknowledgement number */
        /*LINTED non-portable bitfields*/
        u_int8_t  th_x2:4,              /* (unused) */
                  th_off:4;             /* data offset */
        u_int8_t  th_flags;
#define TH_FIN    0x01
#define TH_SYN    0x02
#define TH_RST    0x04
#define TH_PUSH   0x08
#define TH_ACK    0x10
#define TH_URG    0x20
        u_int16_t th_win;               /* window */
        u_int16_t th_sum;               /* checksum */
        u_int16_t th_urp;               /* urgent pointer */
} __attribute__((__packed__));
```

A. Quellcode

A.1. gethostbyname, gethostbyaddr

```
/* gethostbyname.c -- Beispiel zum Aufloesen von Hostnamen */

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

int resolve(const char *hostname)
{
    struct in_addr addr;
    struct hostent *host;
    int i;

    /* zuerst testen IP-Adresse in dotted notation */
    addr.s_addr = inet_addr(hostname);
    if (addr.s_addr == -1)
    {
        /* ist also ein Hostname */
        host = gethostbyname(hostname);
        if (!host)
        {
            herror("gethostbyname() failed");
            return 1;
        }

        /* die komplette Liste abarbeiten */
        for (i = 0; host->h_addr_list[i] != NULL; i++)
        {
            addr = *(struct in_addr*) host->h_addr_list[i];
            printf("%s -> %s\n", hostname, inet_ntoa(addr));
        }
    }
    else
    {
        /* ist also eine IP-Adresse */
        host = gethostbyaddr((char*)&addr, sizeof(addr), AF_INET);
        if (!host)
        {
            herror("gethostbyaddr() failed");
            return 1;
        }

        /* zuerst den "echten" Namen ausgeben */
        printf("%s -> %s\n", inet_ntoa(addr), host->h_name);

        /* dann alle Aliase ausgeben */
        for (i = 0; host->h_aliases[i] != NULL; i++)
        {
```

```
            printf("%s -> %s\n", inet_ntoa(addr), host->h_aliases[i]);
        }
    }

    return 0;
}

int main(int argc, char *argv[])
{
    int n;

    for (n = 1; n < argc; n++)
        resolve(argv[n]);

    return 0;
}
```

Dieses Programm nimmt Argumente von der Kommandozeile an, und löst sie auf. Handelt es sich um Hostnamen wird `gethostbyname()` bemüht, handelt es sich um Adressen in der *dotted notation* wird `gethostbyaddr()` verwendet. Es wird jeweils die komplette Liste der Adressen bzw. Alias-Namen ausgegeben.

Hier ein Beispielhafter Aufruf:

```
$ ./gethostbyname www.yahoo.com
www.yahoo.com -> 216.109.118.67
www.yahoo.com -> 216.109.118.68
www.yahoo.com -> 216.109.118.74
www.yahoo.com -> 216.109.118.75
www.yahoo.com -> 216.109.117.108
www.yahoo.com -> 216.109.117.110
www.yahoo.com -> 216.109.117.206
www.yahoo.com -> 216.109.118.64
$ ./gethostbyname 216.109.118.67 216.109.118.68 216.109.118.74 216.109.118.75
216.109.118.67 -> p4.www.dcn.yahoo.com
216.109.118.68 -> p5.www.dcn.yahoo.com
216.109.118.74 -> p11.www.dcn.yahoo.com
216.109.118.75 -> p12.www.dcn.yahoo.com
```

A.2. getservbyname, getservbyport

```
/* getservbyname.c -- Beispiel zum Aufloesen von Dienstenamen */

#include <stdio.h>
#include <netdb.h>

int resolve(const char *servname)
{
    struct servent *serv;
    int i;
    unsigned short port;

    /* zuerst testen ob es sich um eine Portnummer handelt */
    port = strtoul(servname, NULL, 0);

    if (port)
        serv = getservbyport(htons(port), "tcp");
    else
        serv = getservbyname(servname, "tcp");

    if (!serv)
    {
        fprintf(stderr, "no such service found.\n");
        return 1;
    }

    port = ntohs(serv->s_port);
    printf("%u -> %s\n", port, serv->s_name);

    /* Liste mit Aliasen abgrasen */
    for (i = 0; serv->s_aliases[i] != NULL; i++)
    {
        printf("%u -> %s\n", port, serv->s_aliases[i]);
    }

    return 0;
}

int main(int argc, char *argv[])
{
    int n;

    for (n = 1; n < argc; n++)
        resolve(argv[n]);

    return 0;
}
```

Dieses Programm löst Servicenamen bzw. Portnummern auf. Es arbeitet in der Form nur für TCP, kann aber ziemlich einfach erweitert werden. Ein Beispiel:

```
$ ./getservbyname 21 22 23
21 -> ftp
22 -> ssh
23 -> telnet
$ ./getservbyname http
80 -> www
80 -> http
```

A.3. OOB-Daten senden/empfangen

```
/* oob_client.c -- versendet OOB Daten */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

#define BUF_SIZ 1024

int main(int argc, char *argv[])
{
    int s, bytes;
    char line[BUF_SIZ], *p;
    struct sockaddr_in remote;

    if (argc < 3)
    {
        fprintf(stderr, "usage: %s <ip-address> <port>\n", argv[0]);
        return 1;
    }

    /* Ziel festlegen */
    remote.sin_addr.s_addr = inet_addr(argv[1]);
    remote.sin_port = htons(atol(argv[2]));
    remote.sin_family = AF_INET;

    s = socket(PF_INET, SOCK_STREAM, 0);
    if (s == -1)
    {
        perror("socket() failed");
        return 2;
    }

    /* Meldung an den Benutzer, connect() kann u.U. laengere
     * Zeit blockieren.
     */
    printf("connecting to %s:%u...", inet_ntoa(remote.sin_addr),
        ntohs(remote.sin_port));
    fflush(stdout);

    if (connect(s, (struct sockaddr*) &remote, sizeof(remote)) == -1)
    {
        perror("connect() failed");
        return 3;
    }

    puts("OK, start typing.");

    /* Von Tastatur einlesen, CRLF abschneiden, auf Socket
     * senden. Wenn die Zeile mit einem Ausrufezeichen beginnt,
     * wird sie als OOB-Daten versendet (ohne das Ausrufezeichen).
     */
    while (fgets(line, sizeof(line), stdin))
    {
        p = line;
```

```
        while (*p && (*p != '\r') && (*p != '\n'))
            p++;
        *p = '\0';

        if (line[0] == '!')
            bytes = send(s, line + 1, strlen(line + 1), MSG_OOB);
        else
            bytes = send(s, line, strlen(line), 0);

        if (bytes < 0)
        {
            perror("send() failed");
            return 4;
        }

        printf("sent %i bytes\n", bytes);
    }

    close(s);
    return 0;
}
```

Dieser Client verbindet sich mit einem der beiden folgenden Server, und sendet wahlweise „normale“ oder OOB-Daten. Dabei wird einfach in einer Schleife von *stdin* gelesen (`fgets()`) und auf den Socket geschrieben (`send()`). Wenn die Zeile mit einem Ausrufezeichen beginnt, so wird dieses nicht mitgesendet, der Rest der Zeile jedoch als OOB-Daten (MSG_OOB) markiert. Wie erwartet und in Kapitel 4.1.4 begründet wird nur ein Byte der Daten tatsächlich als OOB-Datensatz erkannt und gelesen.

```
/* oob_server1.c -- OOB-Daten mit select() empfangen */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

#define BUF_SIZ 4096

int handle_client(int sock)
{
    int bytes, oob_flag;
    char buffer[BUF_SIZ];
    fd_set rfds, efds;

    do
    {
        FD_ZERO(&rfds);
        FD_ZERO(&efds);

        FD_SET(sock, &rfds);
        FD_SET(sock, &efds);

        if (select(sock + 1, &rfds, NULL, &efds, NULL) == -1)
        {
            perror("select() failed");
            return 1;
        }
```

```
        bytes = 0;

        if (FD_ISSET(sock, &efds))
        {
            oob_flag = 1;

            bytes = recv(sock, buffer, sizeof(buffer) - 1, MSG_OOB);
            if (bytes == -1 && errno == EINVAL)
            {
                oob_flag = 0;
                bytes = recv(sock, buffer, sizeof(buffer) - 1, 0);
            }

            if (bytes == -1)
            {
                perror("recv() failed");
                return 2;
            }
            buffer[bytes] = '\0';

            printf("got \"%s\" as %s data\n", buffer,
                oob_flag ? "OOB" : "normal");
        }

        if (FD_ISSET(sock, &rfds))
        {
            bytes = recv(sock, buffer, sizeof(buffer) - 1, 0);
            if (bytes == -1)
            {
                perror("recv() failed");
                return 2;
            }
            buffer[bytes] = '\0';

            printf("got \"%s\" as normal data\n", buffer);
        }

    }
    while (bytes > 0);

    return 0;
}

int main(int argc, char *argv[])
{
    unsigned short port;
    struct in_addr addr;
    struct sockaddr_in local, remote;
    int s, c, remote_len;

    if (argc < 3)
    {
        fprintf(stderr, "usage: %s <local address> <port>\n", argv[0]);
        return 1;
    }

    /* die "Kontaktdaten" einsammeln */
    addr.s_addr = inet_addr(argv[1]);
    port = strtoul(argv[2], NULL, 10);
```

```
    printf("going to bind server to %s:%u\n", inet_ntoa(addr), port);

    /* Socket erzeugen */
    s = socket(PF_INET, SOCK_STREAM, 0);
    if (s == -1)
    {
        perror("socket() failed");
        return 2;
    }

    /* an die lokale Adresse binden */
    local.sin_addr = addr;
    local.sin_port = htons(port);
    local.sin_family = AF_INET;

    if (bind(s, (struct sockaddr*) &local, sizeof(local)) == -1)
    {
        perror("bind() failed");
        return 3;
    }

    /* in den Lauschmodus versetzen */
    if (listen(s, 3) == -1)
    {
        perror("listen() failed");
        return 4;
    }

    /* in einer Schleife Client abarbeiten, nacheinander */
    for (;;)
    {
        /* dran denken: remote_len vorbelegen! */
        remote_len = sizeof(remote);
        c = accept(s, (struct sockaddr*) &remote, &remote_len);
        if (c == -1)
            continue;

        /* Meldung fuer den Aufrufer: Client angekomen */
        printf("client from %s:%u arrived\n",
            inet_ntoa(remote.sin_addr), ntohs(remote.sin_port));

        /* Client bedienen */
        handle_client(c);

        printf("client left.\n");

        /* Client-Socket schliessen */
        close(c);
    }

    close(s);

    return 0;
}
```

Bei diesem Server werden OOB-Daten mit `select()` behandelt. Wie man erkennen kann werden zwei Deskriptor-Sets verwendet, eins für *readable* und eins für *exception*. Die Auswertung ist allerdings etwas komplizierter, als man für nötig halten würde. Wenn ein Datagramm mit OOB-Daten eingetroffen ist, kehrt `select()` zurück, und setzt das Flag in *efds*. Wenn zusätzlich normale Daten vorliegen, wird auch das in *rfds* gesetzt. Es wird `recv()` mit MSG_OOB aufgerufen, um die OOB-Daten zu lesen.

Jetzt gibt es zwei Möglichkeiten:

- Es liegen tatsächlich OOB-Daten vor, und sie werden gelesen
- Es liegen keine OOB-Daten vor, -1 wird zurückgegeben, *errno* auf EINVAL gesetzt

Die zweite Möglichkeit existiert deshalb, weil nur das gesetzet Flag in *efds* noch keine Garantie für das Vorhandensein von OOB-Daten ist. Es sagt viel mehr aus, daß im Kernel das Flag „OOB-Daten können gelesen werden“ gesetzt ist, und *irgendwelche* Daten vorliegen. Dieses Flag durch das Lesen mit MSG_OOB **nicht** gelöscht, sondern erst, wenn danach ein erfolgreicher Aufruf von `recv()` ohne MSG_OOB stattgefunden hat.

Aus diesem Grund wird im Falle „`recv()` mit MSG_OOB fehlschlagen“ UND „*errno* steht auf EINVAL“ noch ein normales `recv()` hinterhergeschoben, um normale Daten zu lesen, die ja offensichtlich vorliegen (denn `select()` kam ja zurück).

Diese Problematik zeigt nocheinmal deutlich, daß die OOB-Mechanik ziemlich verkorkst ist, und in der Praxis handelt man sich damit nur Probleme ein. Im folgenden Beispiel ist die Behandlung zwar etwas weniger verwirrend, aber das grundsätzliche Problem besteht noch immer: es kann nur ein Byte gelesen werden.

```
/* oob_server2.c -- OOB-Daten mit SIGURG empfangen */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <fcntl.h>
#include <signal.h>

#define BUF_SIZ 4096

/* damit auch der Handler davon lesen kann */
int glbl_sock;

/* Handler fuer OOB-Daten */
void oob_handler(int sig)
{
    int bytes;
    char buffer[BUF_SIZ];

    printf("oob recv... ");
    fflush(stdout);
    bytes = recv(glbl_sock, buffer, sizeof(buffer) - 1, MSG_OOB);
    if (bytes == -1)
    {
        perror("recv() failed");
        return;
    }
    buffer[bytes] = '\0';

    printf("got \"%s\"\n", buffer);
}

int handle_client(int sock)
{
    int bytes;
```

```
    char buffer[BUF_SIZ];

    glbl_sock = sock;

    for (;;)
    {
        printf("normal recv... ");
        fflush(stdout);
        bytes = recv(sock, buffer, sizeof(buffer) - 1, 0);
        if (bytes == -1)
        {
            perror("recv() failed");
            return 2;
        }
        buffer[bytes] = '\0';

        printf("got \"%s\"\n", buffer);

        if (bytes == 0)
            break;
    }

    return 0;
}

int main(int argc, char *argv[])
{
    unsigned short port;
    struct in_addr addr;
    struct sockaddr_in local, remote;
    int s, c, remote_len;
    struct sigaction sa;

    if (argc < 3)
    {
        fprintf(stderr, "usage: %s <local address> <port>\n", argv[0]);
        return 1;
    }

    /* die "Kontaktdaten" einsammeln */
    addr.s_addr = inet_addr(argv[1]);
    port = strtoul(argv[2], NULL, 10);

    printf("going to bind server to %s:%u\n", inet_ntoa(addr), port);

    /* Socket erzeugen */
    s = socket(PF_INET, SOCK_STREAM, 0);
    if (s == -1)
    {
        perror("socket() failed");
        return 2;
    }

    /* an die lokale Adresse binden */
    local.sin_addr = addr;
    local.sin_port = htons(port);
    local.sin_family = AF_INET;

    if (bind(s, (struct sockaddr*) &local, sizeof(local)) == -1)
    {
        perror("bind() failed");
```

```
        return 3;
    }

    /* in den Lauschmodus versetzen */
    if (listen(s, 3) == -1)
    {
        perror("listen() failed");
        return 4;
    }

    /* Handler fuer SIGURG installieren */
    sa.sa_handler = oob_handler;
    sa.sa_flags = SA_RESTART;
    sigaction(SIGURG, &sa, NULL);

    /* in einer Schleife Client abarbeiten, nacheinander */
    for (;;)
    {
        /* dran denken: remote_len vorbelegen! */
        remote_len = sizeof(remote);
        c = accept(s, (struct sockaddr*) &remote, &remote_len);
        if (c == -1)
            continue;

        /* Meldung fuer den Aufrufer: Client angekomen */
        printf("client from %s:%u arrived\n",
            inet_ntoa(remote.sin_addr), ntohs(remote.sin_port));

        /* WICHTIG: Besitzer des Sockets setzen, sonst ist nicht
         * klar, wer SIGURG erhalten soll!
         */
        if (fcntl(c, F_SETOWN, getpid()) == -1)
        {
            perror("fcntl() failed");
            return 5;
        }

        /* Client bedienen */
        handle_client(c);

        printf("client left.\n");

        /* Client-Socket schliessen */
        close(c);
    }

    close(s);

    return 0;
}
```

Hier wird das Signal SIGURG mit einem Signalhandler ausgewertet. Dazu ist es notwendig, daß die PID desjenigen Prozesses, der SIGURG empfangen soll, mit einem Aufruf von `fcntl()` gesetzt wird (dies geschieht hier in `main()`). Es ist zu beachten, daß wenn OOB-Daten eintreffen der Aufruf von `recv()` ohne MSG_OOB unterbrochen wird, das `recv()` mit MSG_OOB erfolgt, und danach das unterbrochene `recv()` fortgesetzt wird (weil SA_RESTART beim Aufruf von `sigaction()` gesetzt wurde).

A.4. Alleinstehender Server

```
/* simple_server.c -- ein einfacher standalone Server */
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

#define BUF_SIZ 4096

/* Client bedienen */
int handle_client(int sock)
{
    char buffer[BUF_SIZ];
    int bytes;

    /* Zuerst mal hoeren was er zu sagen hat */
    bytes = recv(sock, buffer, sizeof(buffer) - 1, 0);
    if (bytes <= 0)
    {
        perror("recv() failed");
        return 0;
    }
    buffer[bytes] = '\0';

    /* auf der Konsole ausgeben */
    printf("client said '%s'\n", buffer);

    /* enthaelt der String 'quit': beenden */
    if (strstr(buffer, "quit"))
        return 0;

    /* ansonsten weiter laufen lassen */
    return 1;
}

int main(int argc, char *argv[])
{
    unsigned short port;
    struct in_addr addr;
    struct sockaddr_in local, remote;
    int s, c, remote_len, running;

    if (argc < 3)
    {
        fprintf(stderr, "usage: %s <local address> <port>\n", argv[0]);
        return 1;
    }

    /* die "Kontaktdaten" einsammeln */
    addr.s_addr = inet_addr(argv[1]);
    port = strtoul(argv[2], NULL, 10);

    printf("going to bind server to %s:%u\n", inet_ntoa(addr), port);

    /* Socket erzeugen */
    s = socket(PF_INET, SOCK_STREAM, 0);
```

```
    if (s == -1)
    {
        perror("socket() failed");
        return 2;
    }

    /* an die lokale Adresse binden */
    local.sin_addr = addr;
    local.sin_port = htons(port);
    local.sin_family = AF_INET;

    if (bind(s, (struct sockaddr*) &local, sizeof(local)) == -1)
    {
        perror("bind() failed");
        return 3;
    }

    /* in den Lauschmodus versetzen */
    if (listen(s, 3) == -1)
    {
        perror("listen() failed");
        return 4;
    }

    /* in einer Schleife Client abarbeiten, nacheinander */
    running = 1;
    while (running)
    {
        /* dran denken: remote_len vorbelegen! */
        remote_len = sizeof(remote);
        c = accept(s, (struct sockaddr*) &remote, &remote_len);
        if (c == -1)
            continue;

        /* Meldung fuer den Aufrufer: Client angekomen */
        printf("client from %s:%u arrived\n",
            inet_ntoa(remote.sin_addr), ntohs(remote.sin_port));

        /* Client bedienen */
        running = handle_client(c);

        printf("client left\n");

        /* Client-Socket schliessen */
        close(c);
    }

    close(s);

    return 0;
}
```

Dieser einfache Server lauscht auf der gegebenen Adresse/Port auf eingehende Verbindungen. Um auf allen lokalen Adressen zu lauschen, kann 0.0.0.0 angegeben werden (das entspricht INADDR_ANY). Wird ein Client angenommen, so wird ein Paket von ihm angenommen und ausgegeben. Dazu sollte man einen Client wie netcat [36] verwenden, der eine ganze Zeile annimmt und dann erst versendet (das `telnet` unter UNIX macht das ebenfalls so, der Windows-Client jedoch sendet die Zeichen einzeln).

Wenn der Server in der Zeichenkette den Bestandteil 'quit' findet, beendet er seine Hauptschleife, andernfalls wird danach der nächste Client angenommen. Hier ein Beispiel:

```
$ ./simple_server 127.0.0.1 2233
going to bind server to 127.0.0.1:2233
client from 127.0.0.1:65518 arrived
client said 'Dies ist ein lustiger Text
'
client left
client from 127.0.0.1:65517 arrived
client said 'Will noch jemand eine Spendenquittung?
'
client left
$
```

Man erkennt recht schön, daß den Meldungen vom Client ein Zeilenumbruch folgt, weshalb das schließende ' der Ausgabe in die nächste Zeile rutscht.

Charakteristisch für den einfachen Server ist, daß die Clients nur nacheinander bedient werden können. Weitere Verbindungsaufbauversuche bleiben in der Warteschlange bis der Server erneut bei `accept()` vorbeikommt. Dies macht sich beim Client als „hängen“ bemerkbar.

Für eine richtige Server-Anwendung wird man den Prozeß von der Konsole lösen, irgendwo ein PID-File hinterlegen und eine passende start/stop-Mechanik einbauen. Wie das geht, wird in Kapitel 11.6 erklärt. Die übliche Position für PID-Files hängt vom eingesetzten Betriebssystem ab, unter NetBSD ist es z.B. üblich unter `/var/run/` eine Datei mit Namen *programm*`.pid` anzulegen.

A.5. Server mit inetd

```
#include <stdio.h>

#define BUF_SIZ 1024

void reverse(char *s)
{
    char tmp, *p;

    p = strchr(s, '\0') - 1;
    for (; p > s; p--, s++)
    {
        tmp = *p;
        *p = *s;
        *s = tmp;
    }
}

int main(int argc, char *argv[])
{
    char buffer[BUF_SIZ];

    fgets(buffer, sizeof(buffer), stdin);
    reverse(buffer);
    fputs(buffer, stdout);

    return 0;
}
```

Das Programm selbst ist nichts besonderes. Es arbeitet ganz normal auf stdin/stdout, die Magie steckt im inetd-Server (oder einem vergleichbaren Superserver). Dieser wird üblicherweise über die Datei `/etc/inetd.conf` konfiguriert, und benötigt eine zusätzliche Zeile wie diese:

```
2233    stream  tcp     nowait  felix /tmp/inetd_server inetd_server
```

Die erste Position ist die Portnummer (oder der Name, falls ein Eintrag in `/etc/services` existiert), das zweite die Art des Sockets (hier ein Stream-Socket, also wie bei SOCK_STREAM), mit TCP als Protokoll. Das `nowait` gibt an, daß dieser Server mehrfach aufgerufen werden darf (stünde hier `wait` wäre der Socket erst wieder bereit, wenn der Prozeß zurückgekehrt ist). `felix` ist der Benutzername, unter dessen Identität (und mit dessen Rechten!) der Prozeß laufen soll. Der nächste Eintrag ist der Pfad zum ausführbaren Programm, gefolgt von sämtlichen Parametern. Achtung: der erste Parameter ist, was im Programm später als `argv[0]` verfügbar ist, für gewöhnlich der Name des Prozesses! Das ist völlig analog zum Systemaufruf `execl()`.

Ein Vorteil von Servern für den inetd ist die Möglichkeit, auch auf der Konsole mit so einem Programm zu arbeiten, da es ja nur auf stdin/stdout arbeitet. Somit kann man auch automatisierte Tests ausführen, die mit Umleitungen auf Dateien operieren, oder mit Pipes funktionieren.

A.6. Server mit select

```
/* select_server.c -- ein Server mit select() */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

#define BUF_SIZ 4096
#define MAX_CLIENTS 10

/* Deskriptor-Set fuellen */
int set_fds(int s, int *cli, fd_set *fds)
{
    int i, max;

    FD_ZERO(fds);
    FD_SET(s, fds);
    max = s;

    for (i = 0; i < MAX_CLIENTS; i++)
    {
        if (cli[i] != -1)
        {
            if (cli[i] > max)
                max = cli[i];
            FD_SET(cli[i], fds);
        }
    }

    /* select() braucht das */
    return max;
}

/* Neuen Platz fuer Client suchen */
int next_free_client(int *clients)
{
    int i;

    for (i = 0; i < MAX_CLIENTS; i++)
        if (clients[i] < 0)
            return i;
    return -1;
}

/* Client-Meldung verarbeiten */
int handle_cli_request(int *clients, fd_set *fds)
{
    int i, j, bytes = 0, sent;
    char buffer[BUF_SIZ];

    /* wer hat was gesagt? */
    for (i = 0; i < MAX_CLIENTS; i++)
    {
        /* ungueltige Clients */
        if (clients[i] < 0)
```

```
            continue;

        /* jeden Client testen */
        if (FD_ISSET(clients[i], fds))
        {
            /* Meldung lesen */
            bytes = recv(clients[i], buffer, sizeof(buffer) - 1, 0);
            if (bytes == 0)
            {
                close(clients[i]);
                clients[i] = -1;
                printf("client %i left (socket closed)\n", i);
                return 0;
            }
            if (bytes == -1)
            {
                clients[i] = -1;
                printf("client %i left (recv() failed)\n", i);
                return 0;
            }
            break;
        }
    }

    /* duerfte eigentlich nie passieren */
    if (i == MAX_CLIENTS)
    {
        printf("huh, no one sent data!\n");
        return 0;
    }

    printf("client %i sent data (%i bytes)\n", i, bytes);

    /* Meldung an alle uebrigen senden */
    for (j = 0; j < MAX_CLIENTS; j++)
    {
        /* ungueltige Clients */
        if (clients[j] < 0)
            continue;

        /* Sender erhaelt keine Meldung */
        if (j == i)
            continue;

        /* jedem anderen Client senden */
        sent = send(clients[j], buffer, bytes, 0);
        if (sent == -1)
        {
            clients[j] = -1;
            printf("client %i left (send() failed)\n", j);
        }
    }

    return 0;
}

int main(int argc, char *argv[])
{
    unsigned short port;
    struct in_addr addr;
    struct sockaddr_in local;
```

```
int s, c, clients[MAX_CLIENTS], max, i;
fd_set fds;

if (argc < 3)
{
    fprintf(stderr, "usage: %s <local address> <port>\n", argv[0]);
    return 1;
}

/* die "Kontaktdaten" einsammeln */
addr.s_addr = inet_addr(argv[1]);
port = strtoul(argv[2], NULL, 10);

printf("going to bind server to %s:%u\n", inet_ntoa(addr), port);

/* Socket erzeugen */
s = socket(PF_INET, SOCK_STREAM, 0);
if (s == -1)
{
    perror("socket() failed");
    return 2;
}

/* an die lokale Adresse binden */
local.sin_addr = addr;
local.sin_port = htons(port);
local.sin_family = AF_INET;

if (bind(s, (struct sockaddr*) &local, sizeof(local)) == -1)
{
    perror("bind() failed");
    return 3;
}

/* in den Lauschmodus versetzen */
if (listen(s, 3) == -1)
{
    perror("listen() failed");
    return 4;
}

/* alle Clients auf 'ungueltig' setzen */
for (i = 0; i < MAX_CLIENTS; i++)
    clients[i] = -1;

for (;;)
{
    max = set_fds(s, clients, &fds);
    select(max + 1, &fds, NULL, NULL, NULL);

    /* neuer Client angekommen */
    if (FD_ISSET(s, &fds))
    {
        c = accept(s, NULL, 0);
        i = next_free_client(clients);
        if (i >= 0)
        {
            printf("new client arrived, -> %i\n", i);
            clients[i] = c;
        }
        else
```

```
            {
                printf("server full\n");
                send(c, "full\r\n", 6, 0);
                close(c);
            }
            continue;
        }

        /* vorhandener Client hat was gesagt */
        handle_cli_request(clients, &fds);
    }

    close(s);

    return 0;
}
```

In diesem Beispiel wird ein Server mit `select()` aufgebaut. Es handelt sich um ein Programm, das (in dieser Form) bis zu 10 Clients annimmt, und jede Meldung eines Clients an alle übrigen sendet. Dies ist ein typischer Anwendungsfall für `select()`: viele Benutzer, die innerhalb desselben Prozesses verwaltet werden sollen, weil sie miteinander agieren sollen.

Die Client-Verwaltung ist hierbei sehr simpel über ein Array realisiert, das nur die Sockets der Clients aufnimmt. Freie Einträge werden mit -1 markiert, einem Wert der von keinem Socket angenommen werden kann. Die Reihenfolge der Einträge ist zunächst die gleiche wie die, in der die Clients angenommen wurden. Sobald jedoch ein Client den Server verlässt (freigewordenen Platz wieder auf -1 setzen!), entsteht ein „Loch“ im Array, das mit dem als nächstes angenommen Client gefüllt wird. Die Reihenfolge ist dann nicht mehr erkennbar.

Es ist naheliegend, eine andere Datenstruktur für die Speicherung der Verbindungsdaten (evtl. Zeitpunkt des Verbindungsaufbaus, Benutzername, evtl. irgendwelche Zugriffsrechte etc.) zu nehmen. Im einfachsten Fall wäre dies ein Array aus structs, aber je nach Anwendungsfall vielleicht besser eine Liste (keine statischen Grenzen für die Benutzeranzahl mehr notwendig), oder eine baumartige Struktur die nach irgendeinem sinnvollen System sortiert ist.

An dieser Stelle noch ein Tipp: wenn Benutzername und Passwort abgefragt werden soll, ist es eine schlechte Idee dies im Dialog zu machen, sodaß ein `recv()` auf die Antwort wartet. Bleibt diese nämlich aus, werden alle übrigens Clients ebenfalls blockiert. Besser ist es, diesen Vorgang *stateful* zu machen, also pro Benutzer ein Flag vergeben, das sich merken kann an welcher Stelle des Login-Prozederes der Client gerade steht. Dann können folgende Eingaben entsprechend behandelt werden, unabhängig vom Zeitpunkt, in dem sie eintreffen, und in der Zwischenzeit kann der Server regulär weiterarbeiten.

A.7. Server mit fork

```
/* fork_server.c -- ein einfacher paralleler Server mit fork */
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <sys/wait.h>
#include <signal.h>

#define BUF_SIZ 4096

/* Signal-Handler fuer tote Kinder ;-) */
void sigchld_handler(int sig)
{
    int status;
    pid_t pid;

    /* PID und Status erfragen ... */
    pid = wait(&status);

    /* ... und was nuetzliches damit anfangen */
    if (WIFEXITED(status))
        printf("Process %u exited with %i\n", pid, WEXITSTATUS(status));
}

/* Client bedienen */
int handle_client(int sock)
{
    char buffer[BUF_SIZ], *p;
    int bytes;
    FILE *file;

    /* den angeforderten Dateinamen einholen */
    bytes = recv(sock, buffer, sizeof(buffer) - 1, 0);
    if (bytes <= 0)
    {
        perror("recv() failed");
        return 1;
    }
    buffer[bytes] = '\0';

    /* das CRLF abschneiden */
    p = buffer;
    while (*p && (*p != '\r') && (*p != '\n'))
        p++;
    *p = '\0';

    /* die Datei oeffnen... */
    file = fopen(buffer, "r");
    if (!file)
    {
        sprintf(buffer, "couldn't open file: %s\n", strerror(errno));
        send(sock, buffer, strlen(buffer), 0);
        return 2;
    }
```

```
    /* ...und "vorlesen" */
    while ((bytes = fread(buffer, 1, sizeof(buffer), file)) > 0)
    {
        if (send(sock, buffer, bytes, 0) != bytes)
        {
            perror("send() failed");
            return 3;
        }
    }

    /* nach Gebrauch schliessen (eigentlich nicht notwendig,
     * denn der Prozess wird eh gleich beendet).
     */
    fclose(file);
    return 0;
}

int main(int argc, char *argv[])
{
    unsigned short port;
    struct in_addr addr;
    struct sockaddr_in local, remote;
    int s, c, remote_len, ret;
    pid_t pid;
    struct sigaction sa;

    if (argc < 3)
    {
        fprintf(stderr, "usage: %s <local address> <port>\n", argv[0]);
        return 1;
    }

    /* die "Kontaktdaten" einsammeln */
    addr.s_addr = inet_addr(argv[1]);
    port = strtoul(argv[2], NULL, 10);

    printf("going to bind server to %s:%u\n", inet_ntoa(addr), port);

    /* Socket erzeugen */
    s = socket(PF_INET, SOCK_STREAM, 0);
    if (s == -1)
    {
        perror("socket() failed");
        return 2;
    }

    /* an die lokale Adresse binden */
    local.sin_addr = addr;
    local.sin_port = htons(port);
    local.sin_family = AF_INET;

    if (bind(s, (struct sockaddr*) &local, sizeof(local)) == -1)
    {
        perror("bind() failed");
        return 3;
    }

    /* in den Lauschmodus versetzen */
    if (listen(s, 3) == -1)
    {
        perror("listen() failed");
```

```
        return 4;
    }

    /* Signal-Handler installieren */
    sa.sa_handler = sigchld_handler;
    sa.sa_flags = SA_RESTART;
    sigaction(SIGCHLD, &sa, NULL);

    /* in einer Schleife Client abarbeiten, parallel */
    for (;;)
    {
        /* dran denken: remote_len vorbelegen! */
        remote_len = sizeof(remote);
        c = accept(s, (struct sockaddr*) &remote, &remote_len);
        if (c == -1)
            continue;

        /* Meldung fuer den Aufrufer: Client angekomen */
        printf("client from %s:%u arrived\n",
            inet_ntoa(remote.sin_addr), ntohs(remote.sin_port));

        /* einen neuen Prozess erzeugen */
        pid = fork();
        switch (pid)
        {
            case -1:
                perror("fork() failed");
                return 6;

            case 0:
                /* Kind-Prozess */

                /* Server-Socket schliessen */
                close(s);

                /* Client bedienen -- Achtung: die
                 * Veraenderung von 'running' wirkt
                 * sich NICHT aus, siehe Text
                 */
                ret = handle_client(c);
                printf("client left\n");

                close(c);

                /* Wichtig: Prozess *beenden*! */
                exit(ret);

            default:
                /* Eltern-Prozess */
                printf("new child with %u created.\n", pid);
                break;
        }

        /* Client-Socket schliessen */
        close(c);
    }

    close(s);

    return 0;
}
```

Dieses Beispielprogramm verwirklicht einen Server, der für jeden neuen Client einen eigenen Prozeß kreiert. Dies bietet sich immer dann an, wenn die Prozesse untereinander nicht kommunizieren müssen. Hier ist der Zweck des Servers das Liefern einer Datei, die der Client angeben kann. Achtung: alles ist erlaubt, also auch Angaben wie "/etc/passwd".

Wenn der Client bedient wurde, wird der Server-Prozeß mittels `exit()` beendet. Der Signal-Handler für das Signal SIGCHLD des Parents bekommt dies mit, und gibt den Exit-Code auf der Konsole an. Diese Ausgabe könnte in etwa so aussehen:

```
$ ./fork_server 0 1234
going to bind server to 0.0.0.0:1234
client from 127.0.0.1:65533 arrived
new child with 573 created.
client from 192.168.0.2:1104 arrived
new child with 478 created.
client left
Process 478 exited with 2
client left
Process 573 exited with 0
^C
```

Diesmal wird das Programm durch Strg+C abgebrochen. Das Setzen einer globalen Variable `running` würde nicht funktionieren, denn jeder Prozeß hat einen eigenen Adressraum, und deshalb eigene globale Variablen. Eine Möglichkeit doch Daten von einem Kindprozeß zum Elternprozeß zu „schmuggeln" wäre der Rückgabewert beim Beenden des Prozesses (den der Elternprozeß mit `wait()` rausbekommt).

Die Ausgaben der Kindprozesse (in diesem Beispiel gibt es nur im Fehlerfall welche) kommen ebenfalls auf die Konsole, von der das Programm gestartet wurde, denn das Kontrollterminal wurde vererbt. Dies wird in Kapitel 11.4 bzw. 11.6 genauer betrachtet.

A.8. Einfacher Client

```
/* http_last.c -- ermittelt letzte Aenderung Einer Datei via HTTP
 *
 * Bitte beachten: der Code ist *nicht* schoen oder gar narrensicher,
 * es handelt sich nur um ein Beispiel, das moeglichst viel
 * Funktionalitaet zu implementieren versucht.
 */

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>

#define BUF_SIZ 4096

int check(const char *uri)
{
    char hostname[BUF_SIZ], *p, *q;
    char buffer[BUF_SIZ];
    struct sockaddr_in remote;
    struct in_addr addr;
    struct hostent *host;
    int s, bytes;

    /* sicher ist sicher... */
    assert(strlen(uri) < BUF_SIZ);

    /* bis zum Hostnamen vorruecken */
    while (*uri && *uri != '/')
        uri++;
    while (*uri && *uri == '/')
        uri++;

    /* diesen kopieren */
    p = hostname;
    while (*uri && *uri != '/')
        *p++ = *uri++;
    *p = '\0';

    /* Hostnamen aufloesen */
    addr.s_addr = inet_addr(hostname);
    if (addr.s_addr == -1)
    {
        host = gethostbyname(hostname);
        if (!host)
        {
            herror("gethostbyname() failed");
            return 1;
        }
        addr = *(struct in_addr*) host->h_addr;
    }

    /* Socket anlegen, TCP */
```

```
s = socket(PF_INET, SOCK_STREAM, 0);
if (s == -1)
{
    perror("socket() failed");
    return 2;
}

/* Adressstruktur vorbereiten */
remote.sin_addr = addr;
remote.sin_family = AF_INET;
remote.sin_port = htons(80);    /* HTTP-Port */

/* Gute Idee: vor blockierenden Meldungen eine Statusmeldung
 * ausgeben, damit der Benutzer weiss warum sich evtl. nichts
 * tut. Nach Zeilen ohne \n das fflush() nicht vergessen!
 */
printf("connecting to %s...", inet_ntoa(addr));
fflush(stdout);

/* Verbindung zum Server aufbauen */
if (connect(s, (struct sockaddr*) &remote, sizeof(remote)) == -1)
{
    perror("connect() failed");
    return 3;
}

/* Meldung vor dem Absenden (blockiert *eigentlich* nie) */
printf(" OK\nsending request...");
fflush(stdout);

/* Request basteln */
sprintf(buffer, "HEAD %s HTTP/1.1\r\nHost: %s\r\n"
    "Connection: Close\r\n\r\n", uri, hostname);

/* Request senden */
if (send(s, buffer, strlen(buffer), 0) < 0)
{
    perror("send() failed");
    return 4;
}

/* Meldung machen -- recv() blockiert meistens */
printf(" OK\nwaiting for response...");
fflush(stdout);

/* Daten empfangen, sizeof(buffer) - 1 weil noch ein '\0'
 * dahinter gesetzt werden soll.
 */
bytes = recv(s, buffer, sizeof(buffer) - 1, 0);
if (bytes < 0)
{
    perror("recv() failed");
    return 5;
}
buffer[bytes] = '\0';

puts(" OK");

/* El-cheapo Auswertung des Responses */
p = strstr(buffer, "Last-Modified");
if (!p)
```

```
    {
        fprintf(stderr, "marker 'Last-Modified' not found\n");
        return 6;
    }

    /* Bis zum Datum vorspulen. Zeile sieht so aus:
     * Last-Modified: Sat, 22 Oct 2005 16:06:25 GMT
     */
    while (*p && isspace(*p))
        p++;
    while (*p && !isspace(*p))
        p++;
    q = p;
    while (*q && *q != '\r')
        q++;
    *q = '\0';

    /* Ausgabe machen */
    printf("Date of last modification: %s\n", p);

    /* Socket schliessen */
    close(s);

    return 0;
}

int main(int argc, char *argv[])
{
    int n;

    for (n = 1; n < argc; n++)
        check(argv[n]);

    return 0;
}
```

Dieser Client fragt einen HTTP-Server nach einer Datei und gibt das Datum der letzten Änderung aus. Dies soll nur als einfaches Beispiel dienen, normalerweise gehört man für eine derartig fehleranfällige Auswertung eines Responses gezüchtigt. Insbesondere deshalb, weil Fehlermeldungen des Servers nicht verstanden werden, die Header-Felder nicht case-insensitiv durchsucht werden usw. Außerdem muß der übergebene URI dem einfachen Schema `http://<hostname><pfad>` folgen, welches nur ein Sonderfall für gültige URIs ist (siehe [12]).

Worum es geht: ein Client zerfällt (wie in Kapitel 10.1 besprochen) in die Aufgaben a) eine IP-Adresse + Portnummer für den Verbindungsaufbau zu ermitteln, b) die Verbindung herzustellen, c) das Protokoll abzuspulen und d) die Verbindung wieder zu beenden.

Ein Beispielaufruf:

```
$ ./http_last http://www.zotteljedi.de/index.xhtml
connecting to 212.227.119.70... OK
sending request... OK
waiting for response... OK
Date of last modification:  Sat, 22 Oct 2005 16:06:25 GM
```

A.9. Broadcasts

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

#define BUFFER_SIZE 1024

int main(int argc, char *argv[])
{
    int sock, bytes, one;
    char message[BUFFER_SIZE], *p;
    struct sockaddr_in remote;

    if (argc < 3)
    {
        fprintf(stderr, "usage: %s <address> <port>\n", argv[0]);
        return 1;
    }

    sock = socket(PF_INET, SOCK_DGRAM, 0);
    if (sock == -1)
    {
        perror("socket() failed");
        return 2;
    }

    one = 1;
    if (setsockopt(sock, SOL_SOCKET, SO_BROADCAST,
        &one, sizeof(one)) == -1)
    {
        perror("setsockopt() failed");
        return 3;
    }

    remote.sin_family = AF_INET;
    remote.sin_port = htons(atol(argv[2]));
    remote.sin_addr.s_addr = inet_addr(argv[1]);

    while (fgets(message, sizeof(message), stdin))
    {
        p = message;
        while (*p && (*p != '\r') && (*p != '\n'))
            p++;
        *p = '\0';

        bytes = sendto(sock, message, strlen(message), 0,
            (struct sockaddr*) &remote, sizeof(remote));
        if (bytes == -1)
        {
            perror("sendto() failed");
            return 4;
        }

        printf("sent %i bytes as broadcast.\n", bytes);
```

```
    }

    close(sock);
    return 0;
}
```

Dieses Programm erzeugt zunächst wie gewohnt einen UDP-Socket (SOCK_DGRAM), und setzt dann die Socket-Option SO_BROADCAST. Damit können Datagramme auch an Broadcast-Adressen (wie beispielsweise 192.168.0.255) gesendet werden – ohne diese Option würden sie verworfen werden. Das ist die einzige Änderung, die notwendig ist.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#ifdef WIN32
#include <winsock.h>
#else
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <unistd.h>
#define INVALID_SOCKET -1
#define SOCKET_ERROR -1
#endif

#define BUFFER_SIZE 1024

int main(int argc, char *argv[])
{
#ifdef WIN32
    SOCKET sock;
    int remote_len;
#else
    int sock;
    socklen_t remote_len;
#endif
    int bytes;
    char message[BUFFER_SIZE];
    struct sockaddr_in local, remote;


    if (argc < 2)
    {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        return 1;
    }

#ifdef WIN32
    {
        WSADATA wsa;
        if (WSAStartup(MAKEWORD(1, 1), &wsa) != 0)
        {
            fprintf(stderr, "WSAStartup() failed: %i\n", WSAGetLastError());
            return 1;
        }
    }
#endif

    sock = socket(PF_INET, SOCK_DGRAM, 0);
    if (sock == INVALID_SOCKET)
```

```
    {
        perror("socket() failed");
        return 2;
    }

    local.sin_family = AF_INET;
    local.sin_addr.s_addr = INADDR_ANY;
    local.sin_port = htons(atol(argv[1]));

    if (bind(sock, (struct sockaddr*) &local,
        sizeof(local)) == SOCKET_ERROR)
    {
        perror("bind() failed");
        return 3;
    }

    do
    {
        remote_len = sizeof(remote);
        bytes = recvfrom(sock, message, sizeof(message) - 1, 0,
            (struct sockaddr*) &remote, &remote_len);
        if (bytes == SOCKET_ERROR)
        {
            perror("recvfrom() failed");
            return 4;
        }
        message[bytes] = '\0';

        printf("got \"%s\" from %s\n", message, inet_ntoa(remote.sin_addr));
    }
    while (bytes > 0);

#ifdef WIN32
    closesocket(sock);
#else
    close(sock);
#endif

    return 0;
}
```

Mit diesem Programm werden die Broadcast-Datagramme empfangen. Wie man sieht ist es ein ganz gewöhnliches Programm zum Empfangen von UDP-Datagrammen. Für das Empfangen von Broadcast-Datagrammen ist das Setzen der Option SO_BROADCAST nicht notwendig. Daß es sich um Broadcast-Dataramme handelt sieht der Prozeß sogar überhaupt nicht, diese Information wird bereits vor der IP-Schicht verworfen.

Weiterhin zeigt dieses Programm, wie man Quellcode gestalten kann, damit er sowohl unter UNIX als auch unter Windows übersetzt werden kann. Die Übersichtlichkeit leidet ganz klar ein wenig. Die Definitionen SOCKET_ERROR und INVALID_SOCKET wurden für den UNIX-Zweig getroffen, damit hier weitere `#ifdef` entfallen können. Hier wird keine Annahme über die konkrete Implementierung unter Windows gemacht!

B. Winsock-Fehlercodes

Name	Code	Beschreibung
WSAEINTR	10004	Interrupted function call
WSAEACCES	10013	Permission denied
WSAEFAULT	10014	Bad address
WSAEINVAL	10022	Invalid argument
WSAEMFILE	10024	Too many open files
WSAEWOULDBLOCK	10035	Resource temporarily unavailable
WSAEINPROGRESS	10036	Operation now in progress
WSAEALREADY	10037	Operation already in progress
WSAENOTSOCK	10038	Socket operation on nonsocket
WSAEDESTADDRREQ	10039	Destination address required
WSAEMSGSIZE	10040	Message too long
WSAEPROTOTYPE	10041	Protocol wrong type for socket
WSAENOPROTOOPT	10042	Bad protocol option
WSAEPROTONOSUPPORT	10043	Protocol not supported
WSAESOCKTNOSUPPORT	10044	Socket type not supported
WSAEOPNOTSUPP	10045	Operation not supported
WSAEPFNOSUPPORT	10046	Protocol family not supported
WSAEAFNOSUPPORT	10047	Address family not supported by protocol family
WSAEADDRINUSE	10048	Address already in use
WSAEADDRNOTAVAIL	10049	Cannot assign requested address
WSAENETDOWN	10050	Network is down
WSAENETUNREACH	10051	Network is unreachable
WSAENETRESET	10052	Network dropped connection on reset
WSAECONNABORTED	10053	Software caused connection abort
WSAECONNRESET	10054	Connection reset by peer
WSAENOBUFS	10055	No buffer space available
WSAEISCONN	10056	Socket is already connected
WSAENOTCONN	10057	Socket is not connected
WSAESHUTDOWN	10058	Cannot send after socket shutdown
WSAETIMEDOUT	10060	Connection timed out
WSAECONNREFUSED	10061	Connection refused
WSAEHOSTDOWN	10064	Host is down
WSAEHOSTUNREACH	10065	No route to host
WSAEPROCLIM	10067	Too many processes
WSASYSNOTREADY	10091	Network subsystem is unavailable
WSAVERNOTSUPPORTED	10092	Winsock.dll version out of range
WSANOTINITIALISED	10093	Successful WSAStartup not yet performed
WSAEDISCON	10101	Graceful shutdown in progress
WSATYPE_NOT_FOUND	10109	Class type not found
WSAHOST_NOT_FOUND	11001	Host not found
WSATRY_AGAIN	11002	Nonauthoritative host not found
WSANO_RECOVERY	11003	This is a nonrecoverable error
WSANO_DATA	11004	Valid name, no data record of requested type

C. Referenznetzwerk

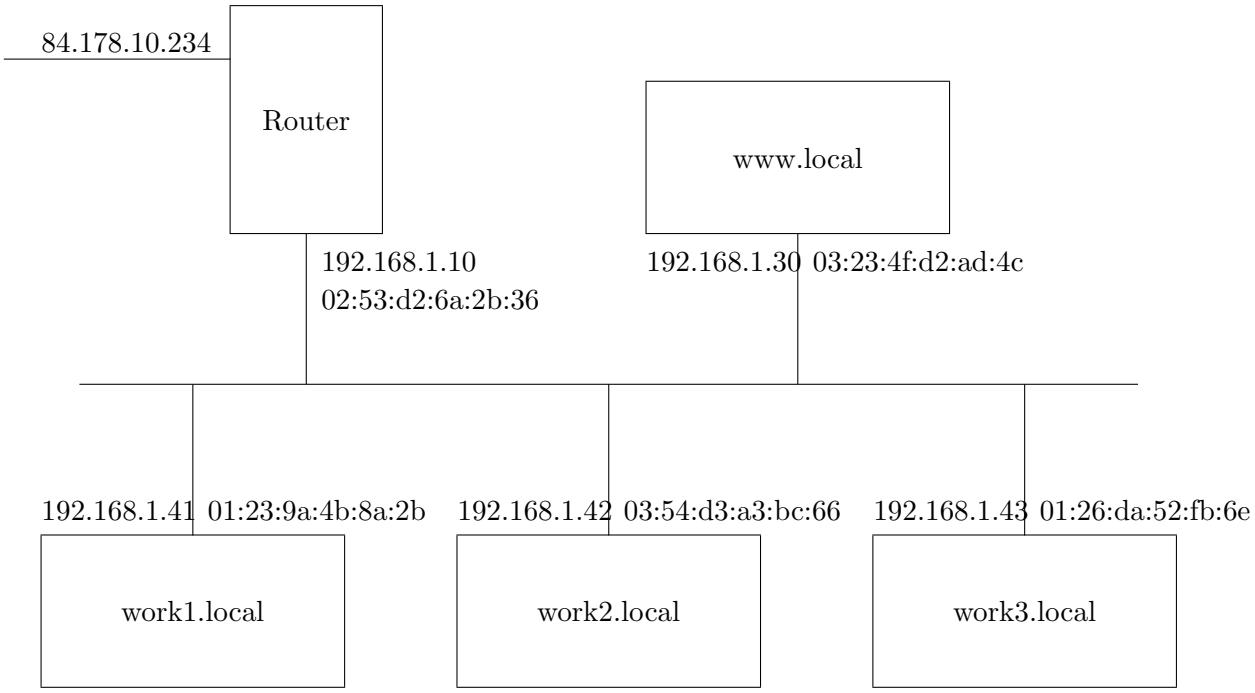

Abbildung C.1.: Referenznetzwerk

D. Glossar

Byte Order
Die Byte Order bezeichnet die Anordnung der Bytes bei ganzzahligen Werten, die aus mehr als 8 Bits bestehen. Es gibt zwei konkurrierende Anordnung, Little Endian (wie sie bei i386-Architekturen verwendet wird) und Big Endian (wie beim PowerPC). Bei Little Endian ist das Byte, das das Least Sginificant Bit (LSB) enthält, an der höchsten Speicheradresse untergebracht. Betrachtet man also den Wert 0xaabbccdd, indem man die Bytes einzeln interpretiert, so erhält man [addr] = 0xdd, [addr+1] = 0xcc, [addr+2] = 0xbb und [addr+3] = 0xaa. Bei Big Endian hingegen ist das MSB (Most Significant Bit) an der höchsten Speicheradresse untergebracht.

Die Byteorder betrifft im heutigen Internet nur 16-Bit-Werte und 32-Bit-Werte, aber spielt natürlich auch bei beliebigen anderen Vielfachen eine Rolle. In den Headern der Internet-Protokolle (und generell in den meisten Formaten) wird immer Big Endian benutzt, das in diesem Zusammenhang auch als Network Byte Order bezeichnet wird. Als Host Byte Order bezeichnet man die Darstellung, die das betrachtete System verwendet, bei i386 also Little Endian. Bei PowerPC Prozessoren wäre Host Byte Order gleich Network Byte Order.

Datagramm
Unter einem Datagramm versteht man eine Protokoll-Einheit, bestehend aus Header und Nutzdaten. Der Begriff wird meistens synonym zu Paket verwendet.

Dotted Notation
Die Darstellung einer IPv4-Adresse in der Form 127.0.0.1 wird dotted Notation genannt. Sie wird üblicherweise zur Darstellung als Zeichenkette eingesetzt, während IPv4-Adressen sonst als 32-bit Binärzahl vorliegen.

Endpunkt
Ein Endpunkt ist eine eindeutige Beschreibung eines Endes einer Netzwerkverbindung, bestehend aus IP-Adresse und Portnummer. Endpunkte gibt es bei UDP und TCP. Da ein Endpunkt ein Feature der Anwendungsschicht ist, steht hinter einem solchen immer eine Anwendung, und damit ein Prozeß. Ein Prozeß kann jedoch mehrere Endpunkte verwalten, ebenso kann ein Endpunkt in mehreren Prozessen existieren. Siehe Kapitel 9.2.2.

Flußkontrolle
Flußkontrolle bezeichnet die Beeinflussung des Datendurchsatzes. Sie kann einmal auf der Sicherungsschicht stattfinden, aber auch in der Transportschicht (vgl. 1.2).

Frame
Als einen Frame bezeichnet man eine Übertragungseinheit auf der Netzwerkschicht. Ein Frame besteht meistens aus einem Header (mit beispielsweise Adressdaten), einem Nutzdatenbereich (vielleicht ein IP-Datagramm) und oftmals einem Trailer (könnte eine Checksumme beinhalten).

IANA
Internet Assigned Numbers Authority

ICANN
Internet Corporation for Assigned Names and Numbers

IEEE
IEEE (als „I triple E“ gesprochen) steht für Institute of Electrical and Electronics Engineers. Das IEEE ist für die Standardisierung von Technologien wie beispielsweise Netzwerke (IEEE 802, 1394) zuständig, aber auch Software Schnittstellen (IEEE 1003, POSIX) oder die Darstellung von Gleitkommazahlen (IEEE 754).

LAN
Local Area Network. Eine Netzwerkverbindung, die sich über eine überschaubar große Fläche erstreckt. Die Latenz (Verzögerung der einzelnen Pakete) ist hier in der Regel gering.

Oktett
Unter einem Oktett versteht man eine Dateneinheit, die aus 8 Bit gebildet wird. Im Netzwerkbereich nennt man dies nicht Byte, um Verwechslungen auszuschließen, denn exotische Hardware versteht unter Bytes gelegentlich etwas anderes.

Prozeß
Ein Prozeß wird im allgemeinen als „in der Ausführung befindliches Programm“ umschrieben, das soll für unsere Zwecke ausreichen.

Punkt-zu-Punkt-Verbindung
Unter einer Punkt-zu-Punkt-Verbindung versteht man eine Verbindung, die ohne Zwischenstationen gebildet wird, den OSI-Schichten 1-3 entsprechend. „Virtuelle“ Verbindungen zwischen den oberen Schichten werden als Ende-zu-Ende-Verbindung bezeichnet, und haben in Wirklichkeit beliebig viele Zwischenstationen.

Redundanz
Redundanz bezeichnet das mehrfache Vorhandensein von irgendwas. Ein redundantes Netzwerksystem verfügt beispielsweise über mehrere Verbindungen, die jede für sich die Funktion der anderen übernehmen könnte, sodaß man eine Ausfallsicherheit erzeugt.

RFC
RFC steht für Requests for Comments. RFCs sind technische Spezifikationen, die ursprünglich zur Diskussion gestellt wurden, eben mit der Bitte um Kommentare. Die RFCs beschreiben unter anderem alle wesentlichen Protokolle des Internets sowie einige andere Richtlinien, wie beispielsweise eine Netiquette (RFC 1855).

Sitzung
Als eine Sitzung bezeichnet man eine bestehende Verbindung zwischen zwei oder mehreren Teilnehmern, meistens einem Server und einem Client. Sitzungen sind logische Konstrukte der Sitzungsschicht (Schicht 5 im OSI-Modell).

WAN
Wide Area Network. Eine Netzwerkverbindung, die sich über eine größere Fläche erstreckt, und deshalb meistens auch eine größere Verzögerung (Latenz) der einzelnen Pakete aufweist.

Winsock
Die von Microsoft angebotene und seit Windows 95 im Lieferumfang enthaltene Socket-Implementierung wird Winsock genannt. Sie ist theoretisch durch eine beliebige andere Implementierung austauschbar.

Literaturverzeichnis

[1] **Manpages online**
`http://www.freebsd.org/cgi/man.cgi`

[2] **MSDN Library**
`http://msdn.microsoft.com/library/`

[3] **How to Obtain POSIX Standards and Drafts**
`http://www.posix.com/posix.html`

[4] **WinSock Development Information**
`http://www.sockets.com/`

[5] **Winsock Programmer's FAQ**
`http://tangentsoft.net/wskfaq/`

[6] **Wikipedia – OSI-Modell**
`http://de.wikipedia.org/wiki/OSI-Modell`

[7] **Kernighan/Ritchie**
Programmieren in C, 2. Auflage
Carl Hanser Verlag München Wien; 1990

[8] **Stevens, W. Richard**
Programmieren von UNIX-Netzwerken, 2. Auflage
Carl Hanser Verlag München Wien; 2000

[9] **Herold, Helmut**
Linux-Unix-Systemprogrammierung, 2. Auflage
Addison Wesley Verlag; 1999

[10] **Richter, Jeffrey M.**
Microsoft Windows Programmierung für Experten
Microsoft Press Deutschland

[11] **Petzold, Charles**
Windows-Programmierung
Microsoft Press Deutschland

[12] **RFC 2616: Hypertext Transfer Protocol – HTTP/1.1**
`ftp://ftp.ietf.org/rfc/rfc2616.txt`

[13] **Address Resolution Protocol Parameters**
`http://www.iana.org/assignments/arp-parameters`

[14] **Ether Types**
`http://www.iana.org/assignments/ethernet-numbers`

[15] **Protocol Numbers**
`http://www.iana.org/assignments/protocol-numbers`

[16] **RFC 826: An Ethernet Address Resolution Protocol**
ftp://ftp.ietf.org/rfc/rfc826.txt

[17] **RFC 791: Internet Protocol**
ftp://ftp.ietf.org/rfc/rfc791.txt

[18] **RFC 1071: Computing the Internet Checksum**
ftp://ftp.ietf.org/rfc/rfc1071.txt

[19] **RFC 793: Transmission Control Protocol**
ftp://ftp.ietf.org/rfc/rfc793.txt

[20] **Qt Overview**
http://www.trolltech.com/products/qt/index.html

[21] **The Point-to-Point Protocol (PPP)**
ftp://ftp.ietf.org/rfc/rfc1661.txt

[22] **A Nonstandard for Transmission of IP Datagrams over Serial Lines: SLIP**
ftp://ftp.ietf.org/rfc/rfc1055.txt

[23] **Wikipedia – Ethernet**
http://de.wikipedia.org/wiki/Ethernet

[24] **A Standard for the Transmission of IP Datagrams over IEEE 802 Networks**
ftp://ftp.ietf.org/rfc/rfc1042.txt

[25] **Wikipedia – Carrier Sense Multiple Access / Collision Detection**
http://de.wikipedia.org/wiki/CSMA/CD

[26] **A Standard for the Transmission of IP Datagrams on Avian Carriers**
ftp://ftp.ietf.org/rfc/rfc1149.txt

[27] **How Network Address Translation Works**
http://computer.howstuffworks.com/nat.htm/printable

[28] **Packet Satellite Technology Reference Sources**
ftp://ftp.ietf.org/rfc/rfc829.txt

[29] **Port Numbers**
http://www.iana.org/assignments/port-numbers

[30] **Elizabeth D. Zwicky, Simon Cooper, D. B. Chapman**
Einrichten von Internet Firewalls
O'Reilly Verlag

[31] **User Datagram Protocol**
ftp://ftp.ietf.org/rfc/rfc768.txt

[32] **The Story of the PING Program**
http://ftp.arl.mil/~mike/ping.html

[33] **Bootstrap Protocol (BOOTP)**
ftp://ftp.ietf.org/rfc/rfc951.txt

[34] **Dynamic Host Configuration Protocol**
ftp://ftp.ietf.org/rfc/rfc.txt2131

[35] **The TFTP Protocol (Revision 2)**
ftp://ftp.ietf.org/rfc/rfc.txt783

[36] **The GNU Netcat project**
http://netcat.sourceforge.net/

[37] **Multi-Threaded Programming With POSIX Threads**
http://users.actcom.co.il/~choo/lupg/tutorials/multi-thread/multi-thread.html

[38] **Root Server Technical Operations Assn**
http://www.root-servers.org/

[39] **Host Extensions for IP Multicasting**
ftp://ftp.ietf.org/rfc/rfc1112.txt

[40] **IP Multicast over Token-Ring Local Area Networks**
ftp://ftp.ietf.org/rfc/rfc1469.txt

[41] **Transmission of IP and ARP over FDDI Networks**
ftp://ftp.ietf.org/rfc/rfc1390.txt

[42] **A Method for the Transmission of IPv6 Packets over Ethernet Networks**
ftp://ftp.ietf.org/rfc/rfc1972.txt

[43] **A Method for the Transmission of IPv6 Packets over FDDI Networks**
ftp://ftp.ietf.org/rfc/rfc2019.txt

[44] **Requirements for Internet Hosts – Communication Layers**
ftp://ftp.ietf.org/rfc/rfc1122.txt

[45] **Internet Group Management Protocol, Version 2**
ftp://ftp.ietf.org/rfc/rfc2236.txt

[46] **Post Office Protocol – Version 3**
ftp://ftp.ietf.org/rfc/rfc1939.txt

[47] **The MD5 Message-Digest Algorithm**
ftp://ftp.ietf.org/rfc/rfc1321.txt

[48] **File Transfer Protocol(FTP)**
ftp://ftp.ietf.org/rfc/rfc959.txt

Index