Technische Universität Braunschweig
Institut für Betriebssysteme und Rechnerverbund
Kommunikation und Multimedia
Prof. Dr. L. Wolf

Praktikum Kommunikationssysteme im SS06

– Begleitdokumentation –

Betreuer: Zefir Kurtisi, NN

http://www.ibr.cs.tu-bs.de/courses/ss06/pks/

Kurzbeschreibung

In diesem Praktikum sollen die in der Vorlesung ,,Kommunikationssysteme" vermittelten Inhalte praktisch angewandt werden. Thematisiert werden u.a. verbindungsorientierte und -lose Kommunikation, zeitkritische Kommunikation, Flusskontrolle, privilegierte Nachrichten, Client-Server-Kommunikation, Synchronisation unterschiedlicher Datenströme.

Für die Bearbeitung der Aufgaben werden Abgabefristen vorgegeben, so dass eine kontinuierliche Mitarbeit erforderlich und eine begleitende Leistungskontrolle möglich ist.

Programmiert wird in C.

Ablauf

In einer Einführungsveranstaltung zum Semesterbeginn werden Formalitäten geklärt und die Teilnehmer in Zweiergruppen aufgeteilt. Jede Gruppe bearbeitet alle Aufgaben, wobei eine Arbeitsteilung vorgesehen ist. Diese erfolgt gruppenintern – beispielsweise nach Funktionsblöcken oder Teilkomponenten – und muss deutlich dokumentiert werden.

Es sind drei bis vier Aufgaben vorgesehen, für die, der Komplexität entsprechend, Bearbeitungszeiten von zwei bis fünf Wochen zur Verfügung stehen. Dokumentation, Fristen und Programmgerüste werden schrittweise der Praktikums-Webseite hinzugefügt, die neben allgemeiner Information auch Verweise auf relevante Seiten für den Einstig enthält.

Vor dem Programmieren wird von jeder Gruppe ein schriftliches Konzept verlangt, anhand dessen die Betreuer die Umsetzbarkeit der Vorhaben prüfen und ggf. frühzeitig auf Probleme hinweisen können.

Die Quelltexte der lauffähigen Programme müssen den Betreuern vor der Abnahme zur Prüfung per Email zugesandt werden. Die Abnahme selbst erfolgt während der betreuten Zeit und spätestens zur genannten Frist. Zur Abnahme müssen beide bzw. alle Gruppenmitglieder anwesend sein. Jeder Teilnehmer wird dabei im Gespräch mit den Betreuern seinen Anteil an der Entwicklung dokumentieren und Fragen beantworten. Eine Aufgabe gilt als abgenommen, wenn die geforderte Funktionalität fehlerfrei nachgewiesen wurde und das Gespräch zufriedenstellend verläuft.

Die gesetzten Fristen sind dabei unbedingt einzuhalten, da andernfalls erfahrungsgemäß ein erfolgreicher Abschluss des Praktikums nur schwer möglich ist. Der zeitliche Aufwand für die Bearbeitung wird oft unterschätzt, besonders wenn keine oder wenig praktische Programmiererfahrung vorhanden ist.

Das Praktikum gilt als bestanden, wenn alle Aufgaben erfolgreich bearbeitet wurden.

Arbeitsgerät

Für die Programmierung stehen im CIP-Pool G40 etwa 20 iMacs von Apple zur Verfügung. Diese werden für das Praktikum an einem betreuten und zwei unbetreuten Terminen in der Woche reserviert. In der übrigen Zeit können die Rechner benutzt werden, solange sie nicht von Teilnehmern anderer Praktika benötigt werden.

Für den Zugang zu den Rechnern werden vor Praktikumsbeginn Accounts eingerichtet, die auch einen Email-Adresse enthalten. Für die auf das Praktikum bezogene Kommunikation (also Mailing-Liste und Kontakt zu den Betreuern) ist ausschließlich diese zu verwenden.

Das Betriebssystem OS/X basiert auf einem UNIX-Kern, so dass evtl. vorhandene Erfahrung mit Linux anwendbar sind. Für die Entwicklung werden Grundgerüste vorgegeben, die um die geforderten Funktionen erweitert und über `Makefiles` übersetzt werden. Alternativ steht es jedem Teilnehmer frei, eine der integrierten Entwicklungsumgebungen `XCode` oder `Eclipse` zu verwenden.

Die vorgegebenen Programmgerüste sind so ausgelegt, dass eine Entwicklung teilweise auch am heimischen Rechner erfolgen kann. Zu berücksichtigen ist hierbei allerdings, dass es sich bei den - auf PowerPC basierenden - iMacs um Big-Endian Rechner handelt, während die meisten PCs daheim Little-Endian x86-Maschinen sind.

Unabhängig davon, wo entwickelt wurde, müssen für die Abnahme die Programme zu allen Aufgaben auf den iMacs lauffähig sein.

Ergänzend zu den auf der Veranstaltungsseite genannten Dokumentation folgt nun ein erster Einblick in die Thematik von

- Internet-Protokollen (*Inhalt sollte aus der KS-Vorlesung bekannt sein*)
- Socket-Programmierung

Internet–Protokolle

Kommunikationsprotokolle für Netzwerke müssen eine Vielzahl unterschiedlicher Funktionen erfüllen (z.B. Adressierung des Zielrechners, Flußkontrolle, ggf. Übertragungswiederholungen etc.). Weil ein einziges Protokoll mit allen Eigenschaften zu komplex und unübersichtlich wäre, existieren für die verschiedenen Aufgaben jeweils eigene Protokolle. Diese werden in den folgenden Abschnitten kurz vorgestellt.

Die Internet Protokoll-Familie (TCP/IP)

Das Internet Protokoll (IP) hat sich aus dem ARPANET entwickelt. Die zum IP-Protokoll gehörende Protokollfamilie schaffte den weltweiten Durchbruch, als sie Bestandteil des Berkeley UNIX wurde. Sämtliche Protokolldefinitionen sind als Request for Comments (RFC) direkt über das weltweite Internet verfügbar.

Das Internet-Protokoll stellt ein zentrales Protokoll dar. Die grundlegende Aufgabe besteht in der Vermittlung der Daten zwischen Sender und Empfänger, die in verschiedenen Teilnetzen angesiedelt sein können.

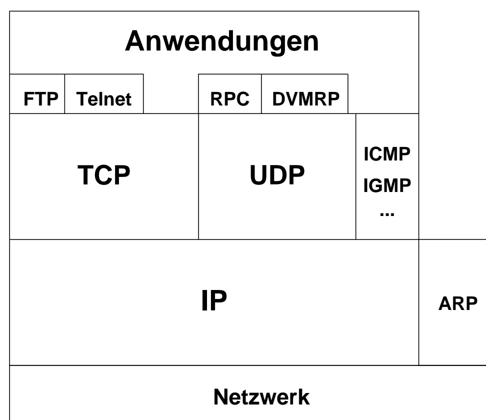

Fig. 1: Internet-Protokolle

Abbildung 1 zeigt die geschichtete Anordnung der Protokolle im Internet. Die Grundidee des IP-Protokolls ist die Schaffung eines gemeinsamen, netzübergreifenden Datagramm-Dienstes, der sich auf ganz verschiedenen Übertragungsmedien implementieren lässt. Dadurch wurde erreicht, dass sehr unterschiedliche existierende Netzwerke (vom einfachen seriellen Kabel bis hin zu Satelliten-Strecken) zur Übertragung benutzt werden können, was überhaupt erst die Schaffung eines weltweiten Netzwerks ermöglichte.

Die einzelnen Schichten haben folgende Aufgaben:

Anwendungsschicht: Auf dieser Schicht findet man die Anwendungen, die Dienste der unterliegenden Schichten nutzen. Die Daten dieser Dienste werden unabhängig von der Form an die Transportschicht weitergereicht. Beispielprotokolle sind Remote Procedure Call (RPC), E-Mail (SMTP), File Transfer (FTP) oder remote Terminal (Telnet bzw. rlogin).

Transportschicht: Die primäre Aufgabe dieser Schicht ist es, die Verbindung von einem Anwendungsprogramm zu einem anderen herzustellen, also die Ende-zu-Ende Kommunikation. In

dieser Schicht existieren zwei Transport-Protokolle: TCP (Transmission Control Protocol) stellt einen verbindungsorientierten und UDP (User Datagram Protocol) einen verbindungslosen Dienst zur Verfügung.

Normalerweise werden mehrere Anwendungen gleichzeitig mit der Transportschicht kommunizieren. Die Transportschicht muss die verschiedenen Daten der Anwendungen auseinanderhalten und entsprechend verteilen können. Mit Hilfe von Portnummern werden die einzelnen Anwendungen auf einem Rechner adressiert.

Vermittlungsschicht: Das Internet-Protokoll (IP) erlaubt die Kommunikation zwischen zwei oder mehreren Rechnern. Die Vermittlungsschicht erhält Datenpakete aus der Transportschicht, wandelt sie in IP-Datagramme um und stellt die Daten entweder direkt oder über einen oder mehrere Router dem Empfänger zu. Zur Adressierung eines Rechners werden 32-bit (oder 4 Byte) lange IP-Adressen benutzt, die gewöhnlich dezimal durch Punkte getrennt dargestellt werden (z.B. 134.169.34.12). In der Version 6 des IP-Protokolls wurden diese Adressen auf 128-bit (oder 16 Byte) erweitert (z.B. 3ffe:400:90:10::1). Anhand der Adresse kann IP – unter der Benutzung von Routingtabellen – entscheiden, wie der Zielrechner zu erreichen ist bzw. welcher Rechner der nächste Knoten auf dem Weg zum Zielrechner ist.

Netzwerkschicht: Hier werden die IP-Datagramme von den oberen Schichten in Empfang genommen und über das entsprechende Netzwerk geschickt. Erst auf dieser Ebene spielt der Typ des verwendeten Netzes (z.B. Ethernet) eine Rolle. Diese Schicht besteht entweder aus einem simplen Devicetreiber (z.B. falls sich die Empfängermaschine im gleichen LAN befindet) oder einem etwas komplexeren Subsystem mit einem eigenen Data Link Protokoll.

Das Transmission Control Protocol (TCP)

Das Transmission Control Protocol (TCP) stellt einen zuverlässigen Bytestrom zur Verfügung. Die Adressierung von Anwendungsprogrammen geschieht durch die IP-Adresse und eine Portnummer. Für Standard-Dienste sind feste Portnummern (*well known ports*) reserviert. Für eine Telnet-Verbindung ist dies beispielsweise Port 23, für FTP Port 21 und für X-Window Port 6000. Portnummern über 1023 sind i.a. frei verfügbar, während die übrigen Portnummern für privilegierte Prozesse reserviert sind.

Der Datentransport findet unter Verwendung einer fensterbasierten Flußkontrolle statt. Man stelle sich eine Art Fenster vor, das über die zu sendenden Daten hinweg gleitet. Die Daten innerhalb des Fensters werden byteweise nummeriert gesendet. Sobald eine Bestätigung vom Empfänger vorliegt, dass das erste Byte angekommen ist, rückt das Fenster eine Stelle weiter. Es unterscheidet bestätigte von unbestätigten Bytes. Die Breite des Fensters, also die Anzahl der Bytes, die jeweils gesendet werden und auf deren Bestätigung dann gewartet wird, hängt von der Pufferkapazität des Empfängers ab. Der Empfänger gibt dabei an, wie viele Daten er annehmen und verarbeiten kann. Die Datenrate muss also während einer Übertragung nicht unbedingt konstant bleiben. Die Breite des Fensters lässt sich dynamisch vergrößern oder verkleinern, wenn Ressourcen auf der Übertragungsstrecke frei werden oder abnehmen.

TCP verwendet eine Voll-Duplex-Verbindung zwischen Sender und Empfänger, d.h. es können über diese eine Verbindung gleichzeitig in beiden Richtungen Nutzdaten gesendet und Quittungen für die gesendeten Daten empfangen werden. Anhand dieser Bestätigungen kann der Sender feststellen, welche Daten der Empfänger erhalten hat und welche unterwegs verloren gegangen sind bzw. verfälscht wurden und somit nochmal gesendet werden müssen. TCP stellt (im Gegensatz zu UDP) einen zuverlässigen Dienst zur Verfügung.

In UNIX ist das TCP-Protokoll im Betriebssystemkern implementiert. Benutzt werden TCP-Dienste über entsprechende Systemaufrufe. Diese Aufrufe sind als Socket-Schnittstelle bekannt (näheres dazu in Kapitel über Socket-Programmierung).

Das User Datagram Protocol (UDP)

Das Transportprotokoll UDP ist ebenfalls auf das IP-Protokoll aufgesetzt. Im Gegensatz zu TCP arbeitet es verbindungslos, d.h. es werden keine Zustandsinformationen gehalten und damit auch keine Quittungen für die Datagramme versendet. Aufgrund dessen ist innerhalb von UDP eine gesicherte Datenübertragung nicht möglich, da UDP nicht feststellen kann, ob und ggf. welche Datagramme bei der Übertragung verloren gegangen sind bzw. verfälscht wurden. Als Folge muss sich jede UDP-basierte Anwendung selbst um die korrekte Behandlung von verlorenen Paketen kümmern. Es ist außerdem durchaus möglich, dass UDP-Pakete nicht in der richtigen Reihenfolge beim Empfänger eintreffen, da aufeinander folgende Pakete über unterschiedliche Wege zum Empfänger gelangen können.

IP-Netze

Jedes Netzwerk im Internet wird durch eine eindeutige Netzadresse und jeder Rechner (Host) in seinem Netz durch eine eindeutige Hostadresse adressiert. Die klassenbasierte Betrachtung von IP-Netzen (Class A, Class B, ...) ist mittlerweile veraltet. IP-Adressen gehören üblicherweise zu einem Netz, das durch eine Netzadresse und eine Netzmaske definiert wird [10]. Die Broadcast-Adresse eines Netzes ist die höchste und damit letzte Adresse eines Netzblockes. Sie kann wie folgt berechnet werden:

$$\text{IP-Netz} \lor (\neg\ \text{Netzmaske}) = \text{Broadcast-Adresse}$$

Beispiel für das Netz 10.0.0.0 mit der Netzmaske 255.255.255.128 (oder 10.0.0.0/25):

$$10.0.0.0 \lor (\neg\ 255.255.255.128) = 10.0.0.0 \lor 0.0.0.127 = 10.0.0.127$$

Tabelle 2 zeigt die Kurzschreibweise von Netzmasken. Grundidee ist dabei, dass man bei so genannten 'continuous netmasks', also Netmasken mit ausschließlich Einsen am Anfang und Nullen am Ende, einfach die Anzahl der Einsen zählen kann.

/0	0.0.0.0
/1	128.0.0.0
/2	192.0.0.0
/3	224.0.0.0
/4	240.0.0.0
/5	248.0.0.0
/6	252.0.0.0
/7	254.0.0.0
/8	255.0.0.0
/9	255.128.0.0
/10	255.192.0.0
/11	255.224.0.0
/12	255.240.0.0
/13	255.248.0.0
/14	255.252.0.0
/15	255.254.0.0
/16	255.255.0.0

/16	255.255.0.0
/17	255.255.128.0
/18	255.255.192.0
/19	255.255.224.0
/20	255.255.240.0
/21	255.255.248.0
/22	255.255.252.0
/23	255.255.254.0
/24	255.255.255.0
/25	255.255.255.128
/26	255.255.255.192
/27	255.255.255.224
/28	255.255.255.240
/29	255.255.255.248
/30	255.255.255.252
/31	255.255.255.254
/32	255.255.255.255

Fig. 2: Netzmasken

Weiterhin besteht die Möglichkeit ein gegebenes Netz in verschiedene *Subnetze* zu unterteilen, um z.B. das Routing innerhalb großer Netze zu vereinfachen. Dabei ist zu beachten, dass sich die einzelnen Subnetze nicht überlappen. Die erste Adresse (Netzwerkadresse) und die letzte Adresse (Broadcastadresse) von Netzen sollten dabei für Rechner verwendet werden.

Literaturhinweise

Die TCP/IP-Protokollfamilie wird sehr ausführlich in den Büchern von D.E. Comer beschrieben. Die Grundlagen findet man in [2]. Eine gute Einführung in die Materie bieten [1] und [12]. Ein Buch, das sich sehr ausführlich mit der Programmierung von Netzwerken unter UNIX befasst, ist [13].

Wer es ganz genau wissen will, dem seien die RFC-Dokumente empfohlen. Das IP-Protkoll ist in [9], TCP in [7] und UDP in [8] beschrieben. In [4] wird IP-Multicasting definiert, [14] beschreibt das DVMRP. [11] listet alle well-known Ports und reservierten Multicast–Adressen auf. Ansonsten geben die RFCs [6] und [5] einen Überblick über das Internet und weitere einführende Literatur.

Sockets

Unter dem Betriebssystem UNIX gibt es verschiedene Möglichkeiten zur Interprozesskommunikation (IPC). Eines dieser Konzepte stellen die an der University of California in Berkeley für das BSD–UNIX entwickelten Sockets ("Steckdose", abstrakter Kommunikationsendpunkt) dar. Die Benutzung zeichnet sich durch einfache Funktionsaufrufe aus. Als Preis für dieses primitive Konzept wird dem Programmierer etwas Mehrarbeit abverlangt.

Wollen mehrere Prozesse miteinander über Sockets kommunizieren, so schreibt ein Prozess, der Informationen verschicken will, diese in den erzeugten und für diesen Zweck vorgesehenen Socket. Prozesse, die von dieser Information Gebrauch machen wollen, lesen diese aus dem Socket, der mit dem sendenden Socket verknüpft ist. Diese Verknüpfung kann auf verschiedene Arten erfolgen. Diese Arten unterscheiden sich in ihren Fähigkeiten, in ihrer Zuverlässigkeit und in ihrer Verarbeitungsgeschwindigkeit.

Typen von Sockets

Stream Socket (`SOCK_STREAM`): bidirektional und zuverlässig. Daten, die in den Socket hineingeschrieben werden, können an dem anderen Socket mit Sicherheit auch abgeholt werden. Die Reihenfolge der Daten bleibt unverändert, es gehen allerdings die Datenblockgrenzen verloren (Schreibt ein Prozeß P_1 die Datenblöcke D_1 und D_2 in dieser Reihenfolge in den einen Socket, so kann Prozeß P_2 die Daten auch nur in dieser Reihenfolge abholen, wobei vom Empfängerprozeß nicht mehr differenziert werden kann, in welchen Portionen die Daten abgesendet wurden, d. h. sie präsentieren sich dem Empfänger als unstrukturierter Bytestrom.

Datagram Socket (`SOCK_DGRAM`): bidirektional und unzuverlässig. Gesendete Datagramme müssen nicht beim Empfängersocket ankommen, können aber auch mehrfach ankommen und die Reihenfolge der Datagramme ist nicht garantiert vorhersagbar. Datenblockgrenzen bleiben erhalten.

Raw Socket (`SOCK_RAW`): wird nur benötigt, um auf die Netzwerkebene direkt zuzugreifen.

Existierende Domänen

Um Sockets benutzen zu können, muss man wissen, in welcher Umgebung (Domäne) man sie benötigt. Sockets können nicht domänenübergreifend kommunizieren. Es gibt die folgenden Domänen:

unix domain (AF_UNIX): besteht aus dem Rechner, auf dem der die Sockets verwaltende Prozess existiert, d.h. es ist nicht möglich, in dieser Domäne mit einem anderen Rechner zu kommunizieren.

internet domain (AF_INET): besteht aus dem gesamten Internet (weltweit).

Für das Praktikum sind die Domäne AF_INET und die Socket Typen SOCK_STREAM oder SOCK_DGRAM relevant.

Benutzung von Sockets

Im folgenden wird erläutert, wie Sockets erzeugt und benutzt werden können. Es ist notwendig folgende header–files mit #include in das Programm einzubinden:

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

Für die UNIX–Domäne wird zusätzlich

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

benötigt, und für die Internet–Domäne:

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

Transportadressen werden allgemein in Strukturen vom Typ struct sockaddr gespeichert, der in socket.h definiert ist. Für die Internet–Domäne, die uns im Praktikum interessiert, werden die Adressen in Strukturen vom Typ struct sockaddr_in gespeichert, die in /usr/include/netinet/in.h definiert ist. Bei der Übergabe an Socket–Funktionen müssen die Zeiger auf diesen Typ zu Zeigern auf die sockaddr Struktur gecastet werden.

Erzeugen von Sockets

Um einen Socket zu erzeugen, wird der folgende Systemaufruf benutzt:

```
s = socket(domain, type, protocol);
```

domain: AF_UNIX oder AF_INET.
type: SOCK_STREAM oder SOCK_DGRAM.
protocol: hier kann explizit ein Protokoll angegeben werden oder 0, um das System zu veranlassen, sich selbst ein passendes Protokoll auszusuchen.
s: s enthält eine kleine positive ganze Zahl als Filedeskriptor für den erzeugten Socket. Falls es nicht möglich war, den Socket zu erzeugen, ist s < 0.

Beispiele:

```
int s;
s = socket(AF_INET, SOCK_STREAM, 0);
s = socket(AF_INET, SOCK_DGRAM, 0);
```

Namenszuweisung an Sockets

Damit andere Prozesse mit dem soeben erzeugten Socket kommunizieren können, muss unser Socket einen lokalen Namen (eine Adresse) bekommen. Die Art des Namens ist abhängig von der für den Socket gewählten Domäne. Dies geschieht mittels folgendem Funktionsaufruf:

```
result = bind(s, name, namelen);
```

`s`: Socket handle.
`name`: Zeiger auf eine `sockaddr_in` Struktur.
`namelen`: Größe des Namens in Bytes.
`result`: Im Fehlerfall -1, ansonsten 0.

Zur genauen Funktionsweise konsultiere man die Manualpage `bind(3n)`. Den Namen eines Sockets kann man über folgende Funktion erfragen:

```
result = getsockname(s, name, namelen);
```

`s`: Socket handle.
`name`: Zeiger auf eine `sockaddr_in` Struktur, die von der Funktion ausgefüllt wird.
`namelen`: Zeiger auf einen Integer-Wert, welcher die Größe des Namens angibt Die Funktion setzt dort die aktuelle Länge des Namens ein.
`result`: Im Fehlerfall -1, ansonsten 0.

In der Internet–Domäne besteht der Name aus einer 4 Byte langen Rechnernummer und einer 2 Byte langen Portnummer. In der UNIX–Domäne ist ein Socketname ein Dateiname.

Kommunikation zwischen Sockets

Es wird zwischen zuverlässiger und unzuverlässiger Kommunikation unterschieden. Die zuverlässige Kommunikation (TCP) arbeitet verbindungsorientiert, die unzuverlässige Kommunikation (UDP) dagegen verbindungslos.

Verbindungsaufbau zwischen Sockets

Das weit verbreitete Client-Server–Konzept kommt auch bei der Socket-Benutzung zum Tragen. Der Server erzeugt und benennt Sockets, und die Clients verbinden ihrerseits ihre eigenen Sockets mit den Server-Sockets (wobei eine Verbindung zwischen genau zwei Sockets besteht).

Vom Client wird also der nachfolgende Aufruf benutzt, um sich mit einem Server Socket zu verbinden:

```
result = connect(s, name, namelen);
```

`s`: Filedeskriptor eines neu erzeugten Socket. Wurde noch kein `bind()` ausgeführt, so weist `connect()` dem Socket automatisch eine lokale Adresse zu.
`name`: Zeiger auf eine Struktur `sockaddr_in`, in der die Adresse des Server-Sockets enthalten ist.
`namelen`: Größe der Server-Adresse in Bytes.
`result`: Ist < 0, falls beim Verbindungsaufbau ein Fehler aufgetreten ist; die genauen möglichen Fehlermeldungen sind in der Manualpage `connect(3n)` beschrieben.

Bevor sich jedoch Clients an einen Server-Socket anmelden können, muss der Server–Prozess einen Socket dafür vorbereiten und anschließend auf Verbindungswünsche warten:

```
result = listen(s, n);
```

`s`: Filedescriptor des mit `bind()` vorbereiteten Server-Sockets.
`n`: Maximale Länge der Warteschlange für einkommende Verbindungswünsche.
`result`: Im Fehlerfall -1, ansonsten 0.

Anschließend kann der Server den ersten Client in der Warteschlange bedienen. Dies geschieht folgendermaßen:

```
newsocket = accept(s, name, namelen);
```

`s`: Filedeskriptor des mit `listen()` vorbereiteten Sockets.
`name`: Zeiger auf eine Struktur `sockaddr_in`, in der die Funktion die Adresse des Clienten einträgt.
`namelen`: Zeiger auf einen Integer, welcher die Größe von `name` angibt. Die Funktion trägt die Länge der Client-Adresse ein.
`newsocket`: Neuer Socket über den mit dem Client kommuniziert werden kann.

Jetzt können sowohl Client als auch Server in den ihnen gehörenden Socket schreiben und auch aus ihm lesen. Die Aufrufe dazu lauten:

```
result = write(s, data, length);
```

`s`: Socket handle
`data`: Zu sendender Datenblock, ein Characterarray (Folge von Bytes).
`length`: Länge des zu sendenden Datenblocks.
`result`: Ist < 0, wenn bei dem Funktionsaufruf ein Fehler aufgetreten ist.

```
result = read(s, buffer, length);
```

`s`: Socket handle
`buffer`: Platz für Datenblock
`length`: Größe des bereitgestellten Puffers.
`result`: Anzahl der gelesenen Bytes. Im Fehlerfall < 0. Der Wert 0 bedeutet "End of File".

Verbindungslose Sockets

Soll ein Server mehrere Clients gleichzeitig bedienen, ohne mehrere Sockets zu erzeugen, so kann man einen verbindungslosen Socket des Typs `SOCK_DGRAM` benutzen.

Der Client verschickt nun seine Datenblöcke an den Server mit folgender Funktion:

```
result = sendto(s, msg, len, flags, to, tolen);
```

`s`: Filedescriptor des Client-Socket. Wurde dem Socket noch keine Adresse zugewiesen (`bind()`), so macht `sendto()` das automatisch.
`msg`: Datenblock (Characterarray)
`len`: Größe des Datenblocks in Bytes.
`flags`: siehe Manualpage `sendto(3n)`, meist einfach 0.
`to`: Zeiger auf eine `sockaddr_in` Struktur mit der Adresse des Server-Sockets
`tolen`: Größe der Serveradresse in Bytes
`result`: Die Anzahl der gesendeten Bytes. Im Fehlerfall -1.

Der Server muss seinem Socket mit `bind()` einen Namen zuweisen, unter dem die Clients ihn ansprechen können. Dann kann er die ankommenden Daten einfach aus dem Socket lesen:

```
result = read(s, buffer, length);
```

`s`: Filedescriptor des Server-Socket
`buffer`: Platz für Datenblock
`length`: Größe des bereitgestellten Puffers.
`result`: Anzahl der gelesenen Bytes. Im Fehlerfall < 0. Der Wert 0 bedeutet "End of File".

Alternativ kann jedoch auch folgender Aufruf verwendet werden, welcher zusätzlich noch den Absender des Datenblocks ermittelt:

```
result = recvfrom(s, buf, len, flags, from, fromlen);
```

`s`: Filedescriptor des Server-Socket
`buf`: Platz für Datenblock
`len`: Größe des bereitgestellten Puffers.
`flags`: siehe Manualpage `recvfrom(3n)`, meist einfach 0.
`from`: Zeiger auf eine (leere) `sockaddr_in` Struktur; wird von `recvfrom()` ausgefüllt
`fromlen`: Zeiger auf einen Integer-Wert, welcher die Größe von `from` angibt; wird von `recvfrom()` ausgefüllt.
`result`: Anzahl der gelesenen Bytes. Im Fehlerfall < 0. Der Wert 0 bedeutet "End of File".

Input/Output-Multiplexing

Mit den bisher beschriebenen Systemaufrufen ist es schwierig, Programme zu schreiben, die über mehrere Sockets "gleichzeitig" mit anderen Prozessen kommunizieren. Unter BSD–UNIX ist daher der Systemaufruf `select()` definiert. Mit diesem kann man eine Menge von File-Deskriptoren (nicht nur Sockets) gleichzeitig daraufhin prüfen, ob Verbindungs-, Lese- oder Schreibwünsche für diese vorliegen oder ein sonstiges Ereignis aufgetreten ist. Darüberhinaus kann man optional einen Timeout–Wert angeben: Spätestens nach Ablauf dieser Zeit kehrt der Systemaufruf zurück, auch wenn bis dahin keine I/O–Operationen bemerkt worden sind.

Die genaue Beschreibung von `select(3c)` findet sich in der Manualpage. An dieser Stelle ist sicherlich ein kleines Programmfragment als Beispiel nützlicher. Angenommen, ein Programm benutzt sowohl verbindungsorienterte als auch verbindungslose Kommunikation. Dazu hat es die beiden Sockets `tcpsock` und `udpsock` eingerichtet und möchte nun darauf warten, daß Verbindungswünsche bzw. Daten eingehen. Ist 10 Sekunden lang nichts eingetroffen, geht das Programm davon aus, daß irgendetwas nicht stimmt und möchte entsprechende Aktionen auslösen:

```
fd_set          readfds;        /* Menge von Filedeskriptoren */
int             dtbsz = getdtablesize(); /* Groesse der Menge */
struct timeval  to;            /* Struktur fuer Timeout-Zeit  */

...

FD_ZERO (&readfds);                 /* readfds := leere Menge */
FD_SET (tcpsock, &readfds);         /* tcpsock hinzufuegen    */
FD_SET (udpsock, &readfds);         /* udpsock hinzufuegen    */

to.tv_sec = 10;                     /* Timeout nach 10 sec    */
to.tv_usec = 0;

switch (select (dtbsz, &readfds, (fd_set *) 0,
                (fd_set *) 0, &to))
{
```

```
case -1:
  if (errno == EINTR)
    {
      /* select() wurde unterbrochen; im Normalfall einfach   */
      /* noch einmal versuchen...                             */
      ...
    }
  /* sonstiger Fehler! Jetzt wird's ernst. */
  perror ("select");
  ...
  break;
case 0:
  /* Timeout! Die 10 sec sind abgelaufen! */
  ...
  break;
default:
  if (FD_ISSET (tcpsock, &readfds))
  {
      /* Es hat sich jemand am TCP-Socket gemeldet! */
      newsock = accept (tcpsock, ... );
      ...
  }
  if (FD_ISSET (udpsock, &readfds))
  {
      /* Am UDP-Socket sind Daten angekommen! */
      len = recvfrom (udpsock, ... );
      ...
  }
}
```

In den meisten Fällen wird so ein `select()` in eine Schleife eingebaut sein.

Löschen von Sockets

Wenn man ein Socket nicht mehr benötigt, sollte man ihn aus dem System entfernen. Dazu dient der Systemaufruf `close()`:

```
result = close(s);
```

`s`: Socket Handle.
`result`: Im Fehlerfall -1, ansonsten 0.

Eine zusätzliche Möglichkeit, eine Verbindung in einer Richtung oder beiden Richtungen zu beenden, ist der Funktionsaufruf:

```
result = shutdown(s, how);
```

`s`: Socket handle.
`how`: 0: will nicht mehr lesen.
1: will nicht mehr schreiben.
2: will gar nichts mehr.
`result`: Im Fehlerfall -1, ansonsten 0.

Sonstiges

Es ist möglich, dass ein Prozess gleich zwei zusammen gehörige Sockets gleichzeitig erzeugt. Dies ist nützlich, wenn man mit einem eigenen Kind-Prozess kommunizieren will. Diese Methode ist nicht dazu geeignet, mehrere nicht verwandte Prozesse miteinander kommunizieren zu lassen. Der entsprechende Aufruf heißt `socketpair()` und ist in der Manualpage `socketpair(3n)` beschrieben.

Literaturhinweise

Das Thema Netzwerkprogrammierung wird sehr ausführlich in [13] behandelt. Eine gute Einführung mit Programmbeispielen findet man in [3], Kapitel 7 und 8. Die Manpages zum Thema Sockets sind `accept(3n)`, `bind(3n)`, `close(2)`, `connect(3n)`, `getsockname(3n)`, `getpeername(3n)`,
`listen(3n)`, `read(2)`, `recvfrom(3n)`, `select(3c)`, `sendto(3n)`, `setsockopt(3n)`, `shutdown(3n)`, `socket(3n)`, `socketpair(3n)`, `write(2)`.

Literatur

[1] T. Braun and M. Zitterbart. *Hochleistungskommunikation; Band 2: Transportdienste und -protokolle*. Oldenbourg, München, 1996.

[2] D.E. Comer. *Internetworking with TCP/IP Volume 1: Principles, Protocols, and Architecture.* Prentice Hall, 2 edition, 1991.

[3] CSRG, University of California. *Unix Programmer's Manual - Supplementary Documents 1*, April 1986.

[4] S. Deering. Host Extensions for IP Multicasting. RFC 1112, Stanford University, August 1989.

[5] E. Hoffman and L. Jackson. FYI on Introducing the Internet – A Short Bibliography of Introductory Internetworking Reading for the Network Novice. RFC 1463, Merit Network Inc., NASA, May 1993.

[6] E. Krol and E. Hoffman. FYI on “What is the Internet?”. RFC 1462, University of Illinois, Merit Network Inc., May 1993.

[7] J. Postel. TRANSMISSION CONTROL PROTOCOL. RFC 793, ISI, September 1981.

[8] Jon Postel. User Datagram Protocol. RFC 768, ISI, August 1980.

[9] Jon Postel. INTERNET PROTOCOL. RFC 791, ISI, September 1981.

[10] Y. Rekhter and T. Li. An Architecture for IP Address Allocation with CIDR. RFC 1518, September 1993.

[11] J. Reynolds and J. Postel. ASSIGNED NUMBERS. RFC 1700, ISI, October 1994.

[12] M. Santifaller. *TCP/IP und ONC/NFS in Theorie und Praxis.* Addison Wesley, 1995.

[13] W.R. Stevens. *UNIX Network Programming.* Prentice Hall, 1992.

[14] D. Waitzman, C. Partridge, and S. Deering. Distance Vector Multicast Routing Protocol. RFC 1075, BBN STC, Stanford University, November 1988.