TKN Telecommunication Networks Group

Praktikum
Rechnernetze und Verteilte Systeme

Vertiefungsunterlagen Block 2

— Berkeley Socket API —

Termin: Vertiefung Block 2

1 Einleitung

In den folgenden Kapiteln findet Ihr einige Basisinformationen zur Berkeley Socket API. Sie dienen nur der Einführung und sollen Euch mit dem erforderlichen Hintergrundwissen versorgen. Je nach Eurem Wissenstand raten wir Euch, eigenständig nach zusätzlichen Informationen zu suchen. Wichtige Quellen sind dabei natürlich das WWW (z.B. http://beej.us/guide/bgnet/) und die man-Pages. Wir wünschen Euch viel Erfolg bei der Bearbeitung!

2 Client-Server Konzept

Das Client-Server Modell stellt eines der wesentlichen Konzepte des heutigen Internets dar. Bei dieser Architektur wird zwischen zwei Einheiten unterschieden, den Clients und Servern. Ein Server ist prinzipiell ein Prozess, welcher einen Dienst für Client-Prozesse bereitstellt. Um einen solchen Dienst nutzen zu können, schickt ein Client einen Service Request zu dem Server. Dabei kann es sich beispielsweise um den Zugriff auf eine Datenbank oder das Abrufen einer Web-Seite handeln. Nach erfolgter Bearbeitung antwortet der Server mit einem Service Reply, worin dem Client das Ergebnis der Bearbeitung mitgeteilt wird.

Das Client Server Konzept sieht keine Aufteilung der Prozesse auf unterschiedliche Computer vor. Beide Prozesse können auch auf einem einzelnen Computer agieren und miteinander oder mit entsprechenden Prozessen auf anderen Computern kommunizieren. Jedoch bietet die Aufteilung der Funktionalitäten eine ideale Möglichkeit, die Client- und Server-Prozesse auf unterschiedliche Computer zu verteilen. Für viele Anwendungen ergeben sich daraus wichtige Vorteile, wie z.B. Zugriffsmöglichkeiten auf spezialisierte Hardware oder das zentrale Management von Informationen.

3 Protokolle des Internets

Im Rahmen dieses praktischen Versuchs werdet Ihr Erfahrungen im Umgang mit einigen grundlegenden Protokollen des Internets sammeln. Dafür werdet Ihr Grundkenntnisse über das Internet Protocol (IP) sowie zwei der verbreitetsten Transportprotokolle, dem Transmission Control Protocol (TCP) und dem User Datagram Protocol (UDP), benötigen.

3.1 Internet Protocol (IP)

Das Internet Protocol [2] ist das Protokoll der Vermittlungsschicht des Internets. Es ist für die Weitergabe von sogenannten Datagrammen von einem Sender zu einem Empfänger innerhalb paketorientierter Datennetzwerke gedacht. Das Protokoll ist so konzipiert worden, dass die Datagramme über einen Verbund

aus unterschiedlichen Netzen hinweg weitergegeben werden können, um der Struktur des Internets mit seinen vielen lokalen Netzwerken und globalen Verbindungen gerecht zu werden. Die einzelnen Netzwerke können unterschiedliche Eigenschaften haben, eine wichtige davon ist die maximale Größe eines Datagramms. Unter Umständen kann es erforderlich sein, ein Datagramm auf dem Weg zu seinem Ziel in mehrere kleinere aufzuteilen. Beim Empfänger werden diese dann innerhalb der Vermittlungsschicht wieder zusammengesetzt.

Aus diesen Anforderungen ergeben sich die wesentlichen Eigenschaften des Internet Protocols. Es beherrscht die Adressierung, die Wegewahl sowie die Defragmentierung und Reassemblierung von Datagrammen. Andere Fähigkeiten wie beispielsweise zuverlässige Datenübertragung, Fehlerkontrolle für Nutzdaten oder Flusskontrolle gehören nicht in seinen Aufgabenbereich.

Adressierung

Das Internet Protocol verwendet 32 Bit lange Adressen. In jedem Datagramm wird die Adresse des Senders und Empfängers angegeben. In den Adressen ist jeweils eine Netzwerknummer und eine Host-Nummer kodiert. Die Netzwerknummer besteht aus den höherwertigen Bits einer Adresse. Internet Protocol Adressen sind in unterschiedliche Klassen, A bis E, eingeteilt. Die Anzahl der Bits für die Netzwerknummer richtet sich nach dieser Klasse. Die übrigen Bits kodieren den Host innerhalb des Netzwerks.

Header eines IP Pakets

Der Header eines Internet Protocol Datagramms besteht aus einem festen Teil mit 20 Byte Länge sowie einem optionalen Teil mit flexibler Größe. In ihm finden sich alle Informationen, die für die Internet Protocol Funktionalitäten erforderlich sind. Der Header beginnt mit einem Versionsfeld, das die Nummer 4 enthält. Um das endlose Zirkulieren von Datagrammen innerhalb des Internets zu verhindern, existiert ein Time To Live (TTL) Feld. Dieses wird in jedem Internet Router dekrementiert. Wenn der Wert Null ist, wird das Paket gelöscht. Um Übertragungsfehler erkennen zu können, wird eine Prüfsumme über den Header des Datagramms berechnet. Da sich der Header in jedem Router ändert, muss die Prüfsumme jedes Mal neu berechnet werden.

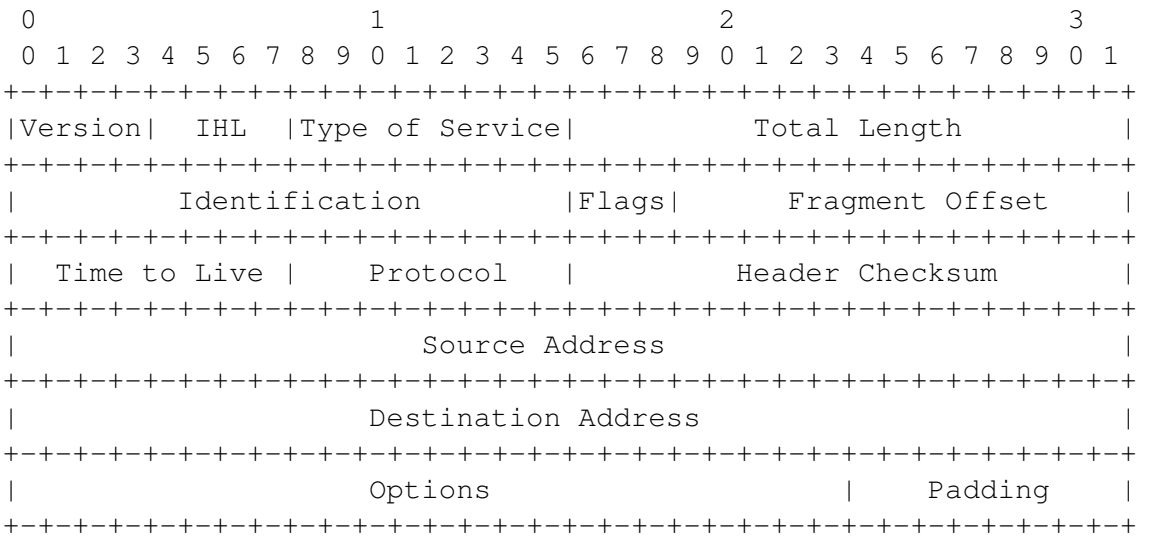

Abbildung 1: Internet Datagram Header (Quelle [2])

3.2 User Datagram Protocol (UDP)

Bei dem User Datagram Protocol [1] handelt es sich um ein sehr einfaches Protokoll zur Übertragung von Datagrammen über ein paketorientiertes Netzwerk, welches auf der Transportschicht arbeitet. Wie

bei dem darunterliegenden Internet Protocol stellt das User Datagram Protocol nur die Möglichkeit für eine verbindungslose Übertragung bereit, die nicht zuverlässig ist. Dies bedeutet, dass Datagramme unter Umständen verloren gehen oder in der falschen Reihenfolge beim Empfänger ankommen. Für die Erkennung von Übertragungsfehlern kann eine optionale Prüfsumme über Teile des IP Headers, den UDP Header und die gesendeten Daten berechnet werden.

Auf der Ebene der Transportschicht müssen Daten den zugehörigen Prozessen zugeordnet werden können, die auf einem Rechner laufen. Das Protokoll verwendet dafür sogenannte Port-Nummern. Jeder Prozess, sowohl auf dem Sender als auch Empfänger, besitzt eine eigene Port-Nummer.

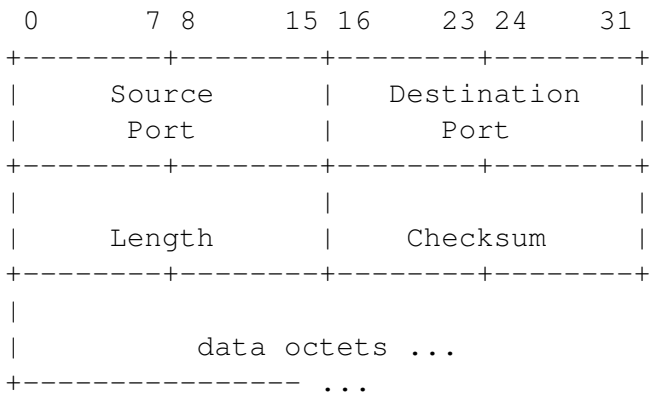

Abbildung 2: User Datagram Header Format (Quelle [1])

3.3 Transmission Control Protocol (TCP)

Das Transmission Control Protocol [3] ist ebenfalls ein Protokoll der Transportschicht. Im Gegensatz zu UDP arbeitet es jedoch verbindungsorientiert. Seine Aufgabe ist es, einen Datenstrom zuverlässig von einem Prozess auf einem Rechner zu einem anderen zu transportieren. Auf der Vermittlungsschicht wird dafür normalerweise das bereits vorgestellte, unzuverlässige Internet Protocol verwendet.

Um auf der Basis des unzuverlässigen Internet Protocols dennoch einen zuverlässigen Dienst realisieren zu können, verfügt das Transmission Control Protocol über verschiedene Mechanismen. Für die Erkennung von Datenverlusten werden Bestätigungen der bereits empfangengenen Datensequenzen verschickt. Auf Seite des Senders werden Timer verwendet, nach deren Ablauf eine neue Übertragung veranlasst wird. Durch die Verwendung von Sequenznummern für die Daten können Vertauschungen der Reihenfolge und Duplikate von Datensequenzen erkannt und entsprechende Fehler behoben werden. Wie bei UDP können Verfälschungen der Segmente anhand einer Prüfsumme über Teile des IP Headers, des TCP Headers und die Daten festgestellt werden.

Eine Besonderheit des Transmission Control Protocols sind die integrierten Mechanismus für die Flusskontrolle zwischen Sender und Empfänger und die Staukontrolle innerhalb der Netzwerke. Ausserdem handelt es sich, wie bereits erwähnt, um ein verbindungsorientiertes Protokoll. Für den Aufbau einer Verbindung zwischen Sender und Empfänger sowie für die Gegenrichtung wird ein spezielles Dreiwege-Handshake verwendet. Zuerst muss eine Seite passiv auf einen Verbindungsaufbau warten. Wenn die andere Seite einen Verbindung aufbauen möchte und eine entsprechende Anfrage schickt, hat die wartende Seite die Möglichkeit, die Anfrage anzunehmen oder abzulehnen. Wird sie angenommen, erfolgt eine Antwort mit einer Bestätigung und kombinierter Verbindungsanfrage in die Gegenrichtung, die im dritten Schritt bestätigt werden muss. Zur Unterscheidung der unterschiedlichen Prozesse verwendet TCP genau wie UDP das Konzept der Port-Nummern. Einer der bekanntesten Ports hat die Nummer 80. Hinter ihm arbeitet in der Regel ein Web Server. Es ist jedoch möglich, auch andere Ports für diesen zu verwenden.

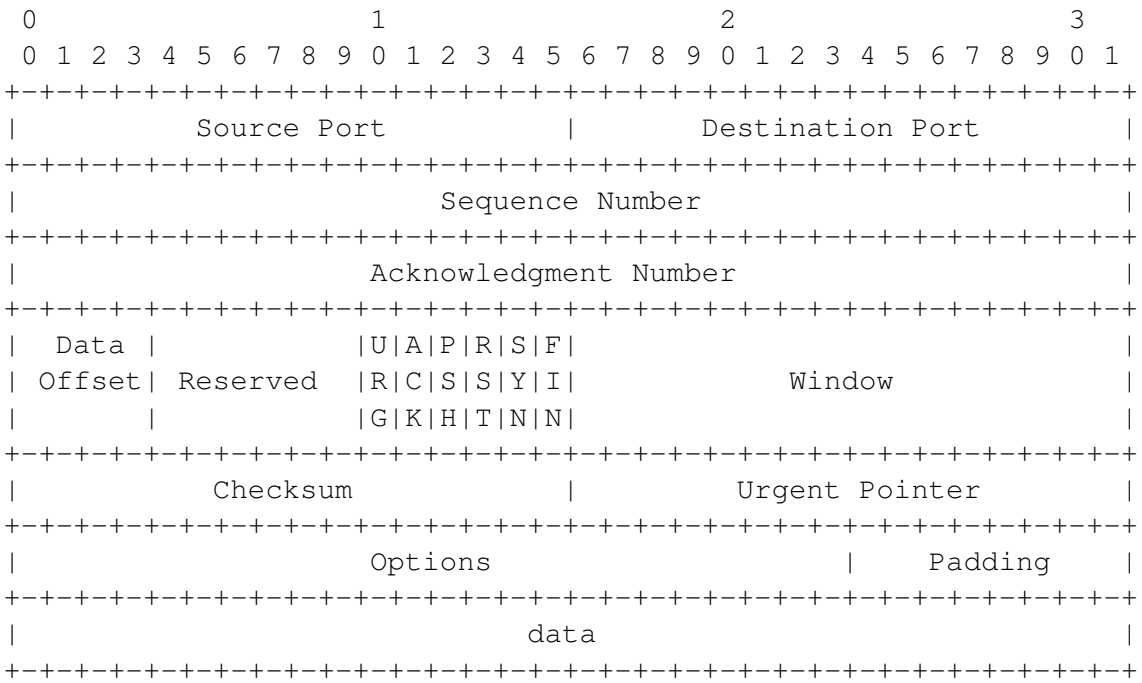

Abbildung 3: TCP Header Format (Quelle [3])

4 Berkeley Socket API

Das Berkeley Socket Application Programming Interface stellt eine flexible Schnittstelle für die Kommunikation von Prozessen über Internet Sockets zur Verfügung. Bei den Internet Sockets handelt es sich um Kommunikationsendpunkte eines Internet Protocol basierten Netzwerkes. Die drei am weitesten verbreiteten Internet Sockets sind Datagram Sockets mit UDP, Stream Sockets mit TCP und Raw Sockets. Die Raw Sockets ermöglichen einen direkten Zugriff auf die Header der eingehenden und ausgehenden Pakete, was bei diesem Praktikum nicht benötigt wird. Daher beschränken sich die folgenden Beschreibungen auf die ersten beiden Socket-Typen.

4.1 Internet Sockets

Neben den Internet Protokollen unterstützt die Socket-Schnittstelle auch noch eine Reihe anderer Protokollfamilien, wie z.B. Appletalk oder ITU-T X.25. Diese Protokollfamilien verwenden unterscheidliche Adressformen und Längen. Damit die Socket-Schnittstelle mit diesen arbeiten kann, muss ihr die Protokollfamilie bekannt sein.

socket();

Der Befehl socket() erstellt einen Socket und gibt einen Deskriptor zurück. Der Aufruf erhält drei Argumente.

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

a) *domain*, welches die Protokollfamilie von dem zu erstellenden Socket spezifiziert. Zum Beispiel:
   - PF_INET for network protocol IPv4 oder
   - PF_INET6 for IPv6.

b) *type*, einen von folgenden:

- SOCK_STREAM
- SOCK_DGRAM
- SOCK_SEQPACKET oder
- SOCK_RAW

c) *protocol*, normalerweise mit IPPROTO_IP, welches als 0 definiert ist, um das default-Protokoll für die entsprechende *domain* und typ darzustellen.

- TCP: PF_INET oder PF_INET6 und SOCK_STREAM
- UDP: PF_ Werte und SOCK_DGRAM

Aber es ist auch möglich ein anderes Protokoll (<netinet/in.h>) anzugeben.
Bei einem Fehler wird -1 zurückgegeben, ansonsten der Deskriptor.

close();

Der Befehl close() terminiert einen Deskriptor. Der Aufruf erfolgt mit int close(int *fd*). Mit dem Aufruf wird die Verbindung des Sockets beendet. Der Aufruf gibt bei Erfolg 0 zurück, andernfalls -1.

4.2 TCP Client und Server Befehle

Im folgenden lernt Ihr einige Socket Befehle kennen, die für die Realisierung eines TCP Clients und Servers erforderlich sind.

bind();

Der Befehl bind() weisst dem Socket *sockfd* die lokale Adresse *my_addr* mit
int bind(int *sockfd*, const struct sockaddr **my_addr*, socklen_t *addrlen*)
zu. Diese ist *addrlen* Bytes lang. Wenn ein Socket mit dem Befehl socket() erstellt wird, existiert er im Namensraum (address family), hat jedoch keine Adresse zugeordnet bekommen.

Die *sockaddr* Struktur kann wie folgt definiert sein:

```
struct sockaddr {
sa_family_t sa_family;
char sa_data[14];
}
```

Bei Erfolg wird eine Null zurückgegeben, ansonsten -1.

listen();

Der Aufruf listen() dient dazu, um auf einem Socket zu ”horchen”. Der Befehl wird mit int listen(int *sockfd*, int *backlog*) aufgerufen.

Um Verbindungen akzeptieren zu können, muss zuerst ein Socket mit Hilfe von socket() erstellt werden. Danach wird mit listen() auf eingehende Verbindungen gehorcht und ein Warteschlangenlimit für die eingehenden Verbindungen festgelegt. Anschließend werden die Verbindungen mit dem Aufruf accept() angenommen.

Der Befehl listen() kann nur mit Sockets vom Typ SOCK_STREAM oder SOCK_SEQPACKET verwendet werden. Der Parameter *backlog* definiert die maximale Länge der Warteschlange. Nach einem erfolgreichen Funktionsaufruf wird eine Null zurückgegeben, andernfalls eine -1.

accept();

Der Aufruf accept() wird dazu verwendet, eine Verbindung auf einem Socket anzunehmen. Der Befehl wird mit int accept(int *sockfd*, struct sockaddr **addr*, socklen_t **addrlen*) aufgerufen.

Der System-Aufruf accept() wird im Zusammenhang mit verbindungsorientierten Socket-Typen benutzt (SOCK_STREAM, SOCK_SEQPACKET).

Der Befehl nimmt die erste Verbindungsanfrage von den wartenden Verbindungen, erstellt einen neuen verbundenen Socket und gibt einen neuen Deskriptor zurück, der zu dem neuen Socket in Bezug steht. Der neue Socket befindet sich nicht im "listening"-Zustand. Der ursprüngliche Socket *sockfd* bleibt unverändert.

Das Argument *sockfd* ist ein Socket, der mit socket() erstellt wurde, mit bind() an eine lokale Adresse gebunden wurde und nun nach dem Befehl listen() auf Verbindungen wartet.

Das Argument *addr* ist ein Zeiger auf eine sockaddr Struktur. Der Parameter *addrlen* sollte zu Beginn die Größe der Struktur beinhalten auf die *addr* zeigt. Am Ende wird dort die aktuelle Länge der Adresse zurückgegeben. Wenn *addr* NULL ist, wird nicht geschrieben.

connect();

Der Befehl connect() initialisiert eine Verbindung auf einem Socket. Der Aufruf erfolgt mit int connect(int *sockfd*, const struct sockaddr **serv_addr*, socklen_t *addrlen*).

Der Befehl connect() verbindet den Socket, dessen Deskriptor mit *sockfd* übergeben wird, mit einer Adresse *serv_addr*. Die Variable *addrlen* gibt dabei die Länge der Adresse an.

Wenn der Deskriptor *sockfd* vom Typ SOCK_DGRAM ist, dann ist *serv_addr* die Adresse, zu der alle Datagramme per default gesendet werden und die einzige Adresse die Datagramme empfängt. Wenn der Socket vom Typ SOCK_STREAM oder SOCK_SEQPACKET ist, versucht dieser Aufruf eine Verbindung zu dem Socket herzustellen, welcher mit der Adresse *serv_addr* verbunden ist.

Allgemein kann gesagt werden, dass verbindungsorientierte Socket-Protokolle nur einmal connect() verwenden können, während verbindungslose Socket-Protokolle connect() mehrfach verwenden dürfen, um ihre Assoziierung zu ändern. Bei einer erfolgreichen Verbindung wird eine 0 zurück gegeben, während bei einem Fehler eine -1 zurückgegeben wird.

send();

Der Aufruf send() sendet eine Nachricht an einen anderen Socket. Der Befehl wird mit ssize_t send(int *s*, const void **buf*, size_t *len*, int *flags*) aufgerufen. Der send()-Befehl kann nur verwendet werden, wenn der Socket verbunden ist. Der einzige Unterschied zwischen send() und write() ist die Existenz von *flags*.

Wenn *flags* den Wert 0 hat, sind send() und write() völlig gleich. Der Parameter *s* ist der Deskriptor vom sendenden Socket.

Die Nachricht hat die Länge *len* und ist in *buf* zu finden. Lokale Fehler werden durch einen Rückgabewert von -1 gekennzeichnet. Fehler, die nicht lokal auftreten, werden nicht angezeigt.

recv();

Mit dem Aufruf recv() wird eine Nachricht von einem anderen Socket empfangen. Der Befehl wird mit ssize_t recv(int *s*, void **buf*, size_t *len*, int *flags*) aufgerufen.

Der recv()-Aufruf wird normalerweise nur mit einem verbundenen Socket verwendet. Bei erfolgreicher Ausführung wird die Länge der Nachricht zurückgegeben.

Wenn keine Nachricht zum Empfangen verfügbar ist, wird auf die Ankunft einer Nachricht gewartet (Ausnahme: der Socket ist nicht blockierend).

write();

Der Befehl write() wird mit ssize_t write(int *fd*, const void **buf*, size_t *count*); aufgerufen.

Der Aufruf schreibt *count* Bytes an den Deskriptor, der mit *fd* angegeben ist, beginnend bei *buf*.

Bei Erfolg wird die Anzahl der geschriebenen Bytes zurückgegeben. Wenn 0 zurückgegeben wird, bedeutet dies, dass nicht geschrieben wurde. Bei einem Fehler wird -1 zurückgegeben.

read();

Der Befehl read() wird mit ssize_t read(int *fd*, void **buf*, size_t *count*) aufgerufen.

Der Aufruf versucht *count* Bytes vom Deskriptor *fd* zu lesen, beginnend bei *buf*. Wenn *count* 0 ist, wird 0 zurückgegeben. Wenn *count* größer als SSIZE_MAX ist, ist das Ergebnis nicht spezifiziert.

Bei Erfolg wird die Anzahl der gelesenen Bytes zurückgegeben, und die Position innerhalb der "Datei" wird um diese Anzahl erhöht. Bei einem Fehler wird -1 zurückgegeben.

4.3 UDP Client und Server Befehle

sendto();

Der Aufruf sendto() sendet eine Nachricht zu einem Socket. Der Befehl wird mit ssize_t sendto(int *s*, const void **buf*, size_t *len* , int *flags*, const struct sockaddr **to*, socklen_t *tolen*) aufgerufen.

Der Parameter *s* ist der Socket-Deskriptor vom sendenden Socket. Die Variable *to* enthält die übergebene Zieladresse und *tolen* gibt dessen Größe an.

Die Nachricht befindet sich in *buf* und hat die Länge *len*. Bei Erfolg wird die Anzahl der übertragenen Zeichen zurückgegeben, andernfalls -1.

recvfrom();

Der Aufruf recvfrom() dient zum Empfangen einer Nachricht von einem Socket.

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

Wenn *from* nicht NULL ist, wird die Quell-Adresse eingefügt. Das Argument *fromlen* wird mit der Größe des Buffers initialisiert, welcher mit der Adresse in *from* assoziiert ist.

Wenn keine Nachricht vorhanden ist, wartet der Aufruf auf eine Nachricht. Nach einem erfolgreichem Empfang wird die Länge der Nachricht zurückgegeben.

4.4 File Descriptor Polling

In viele Situationen ist es erforderlich, mehrere Socket-Deskriptoren in einem Programm zu verwenden und Veränderungen an ihnen zu überwachen, um entsprechend reagieren zu können. Damit die Deskriptoren nicht ständig einzeln abgefragt werden müssen, existiert die Funktionen select() (bzw. die POSIX-Variante pselect()). An diese werden beim Aufruf Listen der relevanten Deskriptoren übergeben.

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

int pselect(int *n*, fd_set **readfds*, fd_set **writefds*, fd_set **exceptfds*, const struct timespec **timeout*, const sigset_t **sigmask*);

Es wird zwischen drei Listen unterschieden

- Die Liste *readfds* beinhaltet alle Deskriptoren, an denen eine Eingabe erwartet wird.
- In *writefds* stehen die Deskriptoren, in die geschrieben werden soll. Es wird also überwacht, ob ein Schreibversuch zu einer Blockierung führen würde.
- In *exceptfds* stehen die relevanten Deskriptoren für die Behandlung von Ausnahmen.

In der Variablen *n* muss die Nummer des größten Deskriptors plus 1 stehen.

Der Aufruf der select() und pselect() Funktion ist blockierend. Sobald sich einer der Deskriptoren verändert hat, wird der Aufruf beendet. Die übergebenen Listen werden dabei modifiziert. Um die Listen zu verändern und um deren Status abfragen zu können, existieren 4 Makros.

- Mit FD_SET wird ein Deskriptor zu einer Liste hinzugefügt.
- FD_CLR löscht den Deskriptor aus der Liste.
- Zum Löschen aller Deskriptoren aus einer Liste wird FD_ZERO verwendet.
- Mit FD_ISSET wird überprüft, ob ein bestimmter Deskriptor zu einer Liste gehört.

Sollte sich keiner der Deskriptoren innerhalb des durch *timeout* angegebenen Wertes geändert haben, wird der Funktionsaufruf beendet.

5 Repräsentation der Daten

Unterschiedliche Reihenfolgen der Bytes

Wenn Daten aus mehreren Bytes zusammengesetzt sind, existieren unterschiedliche Reihenfolgen, in denen die zugehörigen Bytes angegeben werden können. Von Bedeutung sind heutzutage zwei unterschiedliche Varianten.

- Big Endian: Das Byte, welches den Teil der Daten repräsentiert, die den höchsten Wert haben, wird an der ersten Stelle gespeichert.
- Little Endian: Das Byte, welches den Teil der Daten repräsentiert, die den höchsten Wert haben, wird an der letzten Stelle gespeichert.

Auf der Ebene der Prozessoren eines Computers gibt es keine vorgeschriebene Reihenfolge. Beispielsweise verwenden Motorola Prozessoren in der Regel das Big Endian Format, bei Intel ist es Little Endian. Somit ist es nicht möglich, Daten ohne das Wissen über das verwendete Format zwischen mehreren Hosts auszutauschen und entsprechend zu verarbeiten. Um dieses Problem zu lösen, wird bei der Spezifikation eines Protokolls das zugehörige Format mit angegeben. Man unterscheidet folglich zwischen der Byte-Reihenfolge der Daten auf dem Host (Host Byte Order) und der Reihenfolge während des Transports über das Netzwerk (Network Byte Order).

Die im Internet verwendeten Protokolle verwenden nahezu ausschliesslich Big Endian als Network Byte Order. Für die Konvertierung der Daten zwischen Host und Network Byte Order existieren einige wichtige Befehle.

- Die Funktion uint32_t htonl(uint32_t *hostlong*) wandelt einen 32 Bit Wert von der Host in die Network Byte Order um.
- Die Funktion uint16_t htons(uint16_t *hostshort*) leistet das gleiche für 16 Bit Werte.
- Mit uint32_t ntohl(uint32_t *netlong*) wird ein 32 Bit Wert von der Network zurück in die Host Byte Order gewandelt.
- Und uint16_t ntohs(uint16_t *netshort*) erreicht die Umwandlung für 16 Bit Werte.

Prinzipiell wäre es nicht erforderlich, auf Big Endian Systemen Konvertierungen vorzunehmen. In der Tat nehmen die vier Funktionen dort auch keine Veränderungen an den Daten vor. Aus Gründen der Software-Portabilität sollten sie jedoch immer integriert werden.

Darstellungen von IP Adressen

Zur besseren Lesbarkeit werden die einzelnen 4 Byte einer IP Adressen in der Regel in Dezimalschreibweise notiert, wobei die einzelnen Bytes durch einen Dezimalpunkt voneinander abgetrennt sind (z.B. "192.168.34.1"). Bei der Übertragung über das Netzwerk wird die kompaktere Darstellung in 4 Bytes gewählt. Für die Umwandlung der Darstellungsformen stehen zwei Befehle zur Verfügung.

- Die Funktion in_addr_t inet_addr(const char **cp*) konvertiert eine Zeichenkette, welche die IP Adresse in Dezimalschreibweise enthält, in die Darstellung mit 4 Bytes. Wenn die Umwandlung erfolgreich war, liegt das Ergebnis in Network Byte Order vor. Sollte die Umwandlung nicht funktioniert haben, ist das Ergebnis die IP Broadcast Adresse ("255.255.255.255").
- Der Befehl char *inet_ntoa(struct in_addr *in*) macht die Umwandlung rückgängig.

Literatur

[1] J. Postel. RFC 768: User datagram protocol, August 1980.

[2] J. Postel. RFC 791: Internet Protocol, September 1981.

[3] J. Postel. RFC 793: Transmission control protocol, September 1981.