1 Einleitung

Der weitverbreitete Zugang zu öffentlichen Netzen hat in der Vergangenheit eine starke Veränderung in der Nutzung von Computern hervorgerufen. Die reine Datenverarbeitung wurde zunehmend durch den Informationsaustausch verdrängt, dies geht soweit, dass komplexe Anwendungen nur noch im Verbund von Rechnern bewältigt werden können. Um diese Art von Anwendungen zu ermöglichen, benötigt man Mechanismen um die Nutzung der Netze dem Anwender/Programmierer zugänglich zu machen.

Einen solchen Mechanismus stellen die Berkley Sockets dar. Sie zählen zu einem grundlegenden Kommunikationsmechanismus heutiger verteilter Systeme. Sockets sind in vielen Application Programmers Interfaces (API's) von Betriebssystemen integriert und ermöglichen somit Prozessen über Plattformgrenzen hinweg miteinander zu kommunizieren.

Das Ziel der hier vorliegenden Arbeit ist es, im Rahmen des Hauptseminars das Kommunikationsmodell der Sockets vorzustellen und an einem Beispielprogram in der Programmiersprache C zu verdeutlichen.
Dabei soll eine Basis für weitere Arbeiten im Umfeld der Sockets im Rahmen des CYBOP-Projektes geschaffen werden.

Kapitel 2 stellt das CYBOP-Projekt vor. In 2.1 werden die Bestandteile von CYBOP, CYBOL und CYBOI erläutert. In 2.2 wird kurz verdeutlicht wie Sockets in CYBOI integriert werden können.
Kapitel 3 gibt einen Einblick in die wichtigsten Sockverbindungsarten: TCP- , UDP- , Dateisystem- und Raw Sockets. Anschliessend wird kurz verdeutlicht wie nichtblockierendes Senden realisiert wird.
Abschliessend folgen die Schlussbemerkungen mit Zusammenfassung und Ausblick.

2 Das CYBOP-Projekt

2.1 Vorstellung des Projektes

„CYBOP steht für Cybernetics Oriented Programming und ist eine Sammlung von neuartigen Konzepten für den Softwareentwicklungsprozess, die aus der Natur entlehnt sind“
[Heller 2004].
Heutzutage existieren eine Vielzahl von Paradigmen zur Softwareentwicklung so z.B. die strukturierte Programmierung oder die objektorientierte Programmierung. Doch alle bisherigen Paradigmen haben das Ziel von flexiblen und effektiven Softwarelösungen noch nicht erreicht. Eine Reihe von Problemen tretten auf:

- falsche Kombination und Anordnung von Informationen
- Vermischen von Wissens- und Systemsteuerungsinformationen
- Zusammenfassung statischer und dynamischer Aspekte

Aufgrund der Probleme die diese Paradigmen mit sich bringen, müssen in der Softwarearchitektur Kompromisse bezüglich der Abhängigkeiten zwischen den einzelnen Teilen der Software eingegeangen werden:

- Einfügen von redundantem Code um die Abhängigkeiten zwischen den verschiedenen Teilen der Software zu verringern
- statische Managerobjekte die allen anderen Objekten zugänglich sind werden eingefügt
- ...

Diese Maßnahmen können wiederrum Fehler nach sich ziehen: Endlosschleifen, schlechte Performanz...

Die genannten konzeptuellen Problemstellungen sind die Motivation für CYBOP. Das Ergebniss des Softwareentwicklungsprozesses sind Abstraktionen bezüglich der zu Grunde liegenden Problemstellung. Um diese in die Architektur der Software einzubringen modelliert das Projekt die Arbeitsweise des menschlichen Denkens und wendet dabei dessen Grundkonzepte auf die Softwareentwicklung an.

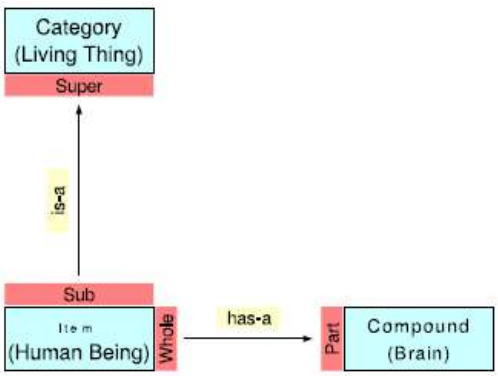

Abb. 2.1/1 Arbeitsweise des menschlichen Denkens

Abbildung 2.1/1 zeigt die Abstraktionen des menschlichen Denkens. Ein Objekt oder Gegenstand (Item) wird durch seine physische Abgrenzung zur Umwelt als solcher erkannt.

Dabei besitzt ein Objekt eine Zugehörigkeit zu einer Kategorie, um nicht jedes Objekt einzeln benennen zu müssen, so z.B. ist Tisch eine Kategorie von Gegenständen und ein Esszimmertisch ein mögliches konkretes Objekt. Weiterhin bestehen Objekte wiederrum aus Objekten, z.B. ein Körper besteht aus Haut, Gehirn und so weiter. Diese Eigenschaft ist in der Abbildung Compound also Bestandteil modelliert.

Die besprochenen Eigenschaften eines Objektes werden in einem Modell als realer existierender Gegenstand zusammengefasst. Abbildung 2.1/2 zeigt die abstrahierten Eigenschaften des Modells:

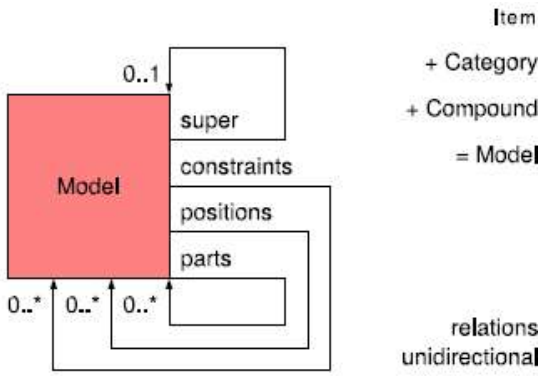

Abb. 2.1/2 Das Modell bestehend aus Katogorie, Bestandteilen und Objekt

Das Modell besitzt eine Menge von unidirektionalen Beziehungen, die sich auf die Eigenschaften des Modells als Summe von Kategorie und Bestandteilen ergeben. Das Modell besitzt ein Vaterobjekt (super-Beziehung), sowie es weitere Bestandteile hat (parts-Beziehung). Zu diesen Bestandteilen besitzt es noch Beziehungen bezüglich deren Position (positions-Beziehung) und Zusicherungen bezüglich deren Bestandteile (constraints-Beziehung) z.B. besteht ein Esszimmertisch aus 4-6 Tischbeinen.

Die zuvor beschriebenen Konzepte sind im CYBOP-Projekt in CYBOL, Cybernetics Oriented Language umgesetzt. CYBOL ist in XML verfasst. In CYBOL ist also die Wissenbasis verfasst.
Eine weitere Komponente ist der CYBOI – Cybernetics Oriented Interpreter. Seine Aufgabe ist es die in CYBOL Dateien zu lesen, diese zu interpretieren und auszuführen.
CYBOI übernimmt also das gesamte Hardwaremanagement und ist somit eine Schnittstelle zwischen der Hardware und der Wissensbasis verfasst in CYBOL.
Abbildung 2.1/3 zeigt einen Überblick über den Zusammenhang CYBOL-CYBOI-Hardware.

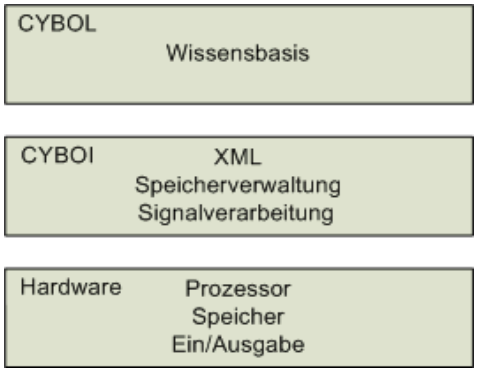

Abb. 2.1/3 CYBOI: Schnittstelle zwischen CYBOL und Hardware

2.2 Socketkommunikation im CYBOP-Projekt

Die Bestandteile des CYBOI-Interpreters werden in Abbildung 2.2/1 gezeigt.

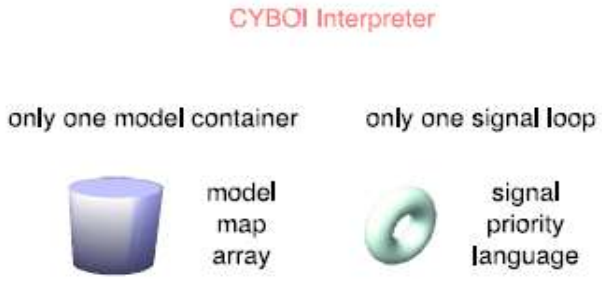

Abb. 2.1/1 Bestandteile von CYBOI

Der Interpreter besitzt einen Container um die Wissensbasis von CYBOL dynamisch zu speichern, weiterhin wird eine Signalschleife benutzt um sämtliche Signale abzufangen und zu bearbeiten. Beispiele für diese Signale sind unteranderem Prozessor- oder Ein-/Ausgaberufe. Um nun Sockets in dieser Architektur aufzunehmen, ist es nötig sie in die Signalschleife zu integrieren, um Verbindungswünsche entgegennehmen zu können oder aber auch Verbindungen aufzubauen und Daten zu senden bzw. zu empfangen.

3 Sockets

3.1 Kurze Historie und Einführung in Sockets

In den 60'ziger Jahren begann die DARPA(Defense Advanced Research Projects Agency) mit der Entwicklung des ARPAnet, einem Vorläufer des heutigen Internets. Fast zeitgleich begann die Entwicklung des UNIX-Betriebssystems, wobei die University of California, Berkley ihren eigenen Ableger, begannt als BSD, entwickelte. 1979 bekam die University den Zuschlag von der DARPA das ARPAnet weiterzuentwickeln. Die folgenden BSD-Versionen ab 4.1 enthielten die ersten in Berkley entworfenen Socket-Mechanismen für das ARPAnet, wie sie noch heute bekannt sind. Sockets werden in vielen Betriebsystemen, z.B. Linux und Windows, verwendet.
Sockets sind, wie schon in der Einleitung erwähnt, ein Mittel zur Interprozesskommunikation. Sie dienen somit zur Kommunikation zwischen unterschiedlichen Adressräumen und es wird kein gemeinsamer Speicher genutzt. Sie stellen einen Kommunikationsendpunkt dar, von dem gesendet oder empfangen werden kann.

In Abbildung 3.1/1 wird der grundlegende Aufbau der Kommunikation verdeutlicht. Zwei Prozesse auf verschiedenen Rechnern kommunizieren miteinander über ein Netzwerk. Die Kommunikation findet stets über ein Socketpaar statt. Beide Sockets, sowohl bei dem Empfänger als auch bei dem Sender, haben einen Port reserviert, über den sie Nachrichten empfangen und senden. Dafür stehen 65536 Ports zu Verfügung, wobei die ersten 1024, die

so genannten „well known“ Ports, für reservierte Dienste, wie z.B. FTP oder TELNET, vergeben sind, und nur von Prozessen mit root-Rechten bedient werden können.

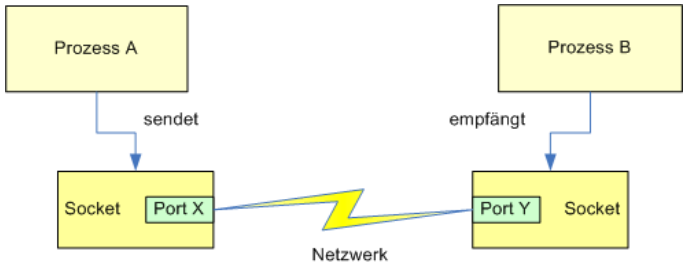

Abbildung 3.1/1 Funktion von Sockets

Die Kommunikationsbeziehung zwischen zwei Sockets ist durch ein Fünfertupel definiert:

(Protokoll, lokale Adresse, lokaler Prozess, entfernte Adresse, entfernter Prozess)

Beide Sockets benutzen das gleiche Protokoll, der Gebrauch der zwei wichtigsten Socketarten (TCP und UDP) wird in den folgenden Kapiteln erläutert. Die Adressen der beiden Rechner werden nach der Konvention des IP-Protokolls mit 4 Byte angegeben z.B. 192.168.0.0 . Die Prozesse werden durch die eindeutige Zuordnung der Portnummern adressiert, das Betriebssystem kann so die eintreffenden Daten an den jeweiligen Prozess weiterleiten.

Abbildung 3.1/2 verschafft einen Überblick über die Einordnung der Socket-Programmierschnittstelle in die zwei Referenzmodelle ISO/OSI und den Internet Protocol Stack.

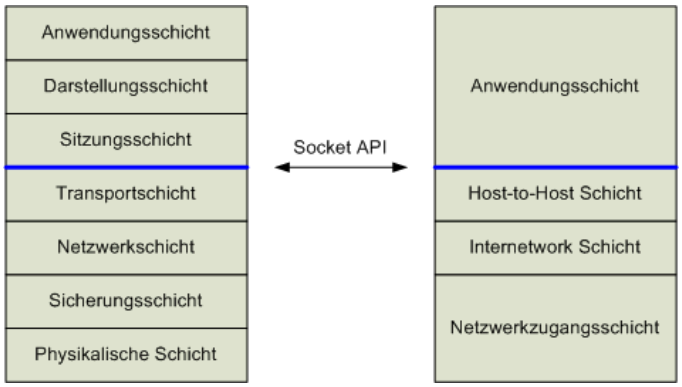

Abb. 3.1/2 Referenzmodelle, links ISO/OSI, rechts Internet Protocol Stack

Die Socket API befindet sich oberhalb der Transportschicht. Es können folglich Protokolle der unterliegenden Schicht benutzt werden, Protokolle oberhalb dieser Schicht können aber nicht benutzt werden, so z.B. SMTP, FTP, POP3...

3.2 Verlässliche Verbindung mit TCP

Eines der wichtigsten und am häufigsten eingesetzte Protokoll ist TCP. Es ist eine weitgehend verlässliche Übertragungsart, da das Protokoll die Übertragung von Botschaften durch Quittierungen und eventuell wiederholter Übertragung sicherstellt. Ausserdem ist das Protokoll Reihenfolge erhaltend und elemeniert Duplikate.

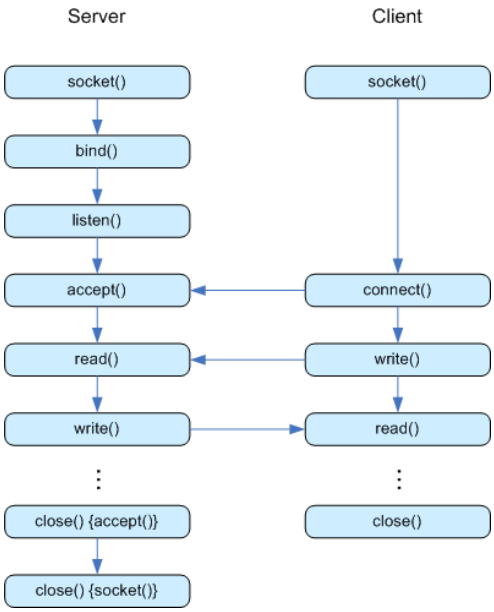

Abb. 3.2/1 TCP-Verbindungsablauf

Die wichtigsten Schritte eines Verbindungsablaufs sind in Abbildung 3.2/1 gezeigt. Im folgenden werden alle Schritte erläutert.

Der erste Schritt beinhaltet das Anlegen eines Sockets und wird durch folgende Methode realisiert:

```
int socket(int family, int type, int protocol)
```

Der socket-Ruf spezifiziert das Element Protokoll aus dem Fünfertupel der Kommunikationsbeziehung. Dazu nehmen die Parameter für TCP-Protokoll folgende Werte an:

```
int socket(AF_INET, SOCK_STREAM, 0)
```

family spezifiziert die Domäne, eine Menge von Protokollen, in diesem Beispiel die Domäne *AF_INET*. *type* gibt die Art der Übertragung an, hier also *SOCK_STREAM*, ein Datenstrom der in der Domäne *AF_INET* durch das TCP-Protokoll realisiert wird. Der Rückgabewert ist eine Nummer, mit der der Socket identifiziert wird.
Der *socket*-Ruf initialisiert die Warteschlangen für ein- und ausgehende Nachrichten, noch ist der Socket nicht mit einem Netzwerk verbunden.

Die nächsten Schritte unterscheiden sich für Dienstanbieter(Server), der eingehende Verbindungen annimmt und Dienstnutzer(Client), der Verbindungen aufbaut bzw Dienste benutzt.
Der Server muss zunächst den noch unbekannten Sockets Binden, d.h. Anmelden am System, unter Angabe der Adresse des lokkalen Prozesses:

```
int bind(int socket, struct sockaddr *myaddr, int addrlen)
```

Die Parameter enthalten die Nummer des Sockets der zu binden ist, sowie einen Datentyp *sockaddr* der im wesentlichen die Domäne, den Port und den IP-Adressbereich angibt von dem Verbindungen angenommen werden. Die genauen Felder werden im Anhang angegeben. Der letzte Parameter gibt die Längen des Datentyps *sockaddr* an. Das Binden gibt die lokale Adresse und den lokalen Prozess im Fünfertupel der Kommunikationsbeziehung an. Als

Ergebnis ist der Socket dem System bekannt und eintreffende Verbindungswünsche werden an den Prozess, der den Socket eingerichtet hat, weitergeleitet.

Nun wird mit der *listen*-Anweisung eine Warteschlange für eingehende Verbindungen eingerichtet:

```
int listen(int socket, backlog)
```

Durch *socket* wird wieder die Nummer des Sockets angegeben und mit backlog die Anzahl der Anforderungen die in der Warteschlange verwahrt werden. Übliche Größen für die Warteschlange sind 5-20, bei Multiprozessservern 60, Plätze. Die Funktion wandelt den Socket in einen listen-socket um, d.h. er kann nur noch Verbindungen annehmen und nicht mehr Senden.
Nach diesen Schritten kann der Client sich zum Server verbinden:

```
int connect (int socket, struct sockaddr *serveraddr, int addrlen)
```

Diese Funktion richtet eine Verbindung vom lokalen zum entfernten Rechner ein, wobei der entfernte Prozess und die entfernte Adresse in **serveraddr*, dem schon bekannten Datentyp, enthalten ist. Damit ist das Fünfertupel der Kommunikationsbeziehung auf der Seite des Client's vollständig. Nach *connect* besteht ein Kommunikationskanal zum entfernten Rechner und die beiden Kommunikationspartner sind bekannt.

Der Server geht durch den Aufruf von *accept* die Verbindung ein:

```
int accept(int socket, sockaddr *peer, int *addrlength)
```

Nach dem Aufruf sind **peer* und **addrlength* mit den Verbindungsdaten des Verbindungspartners gefüllt. Dem Server sind jetzt auch die zwei noch fehlenden Elemente des Fünfertupels bekannt.
Der Funktionsruf entnimmt einen Verbindungswunsch aus der Warteschlange, ist sie leer blockiert die Funktion die Abarbeitung bis eine Verbindung eintrifft.
Accept legt nun einen neuen Socket mit den selben Parametern (also auch dem selben Port) wie der Orginalsocket hat an, d.h. jeder accept-Aufruf etabliert einen neuen Kommunikationskanal. Der Orginalsocket bleibt dann für neue Verbindungswünsche erhalten. Der neue Socket ist jetzt ein bidirektionaler Socket, d.h. er kann Senden und Empfangen, während der Orginalsocket nur zum Verbinungsaufbau dient.

Jetzt können Client und Server den Socket dazu benutzen Nachrichten in den Socket zu schreiben. Der Socket übernimmt transparent das Verpacken in eine oder mehrere Packete. Anschliessend kann aus dem Socket gelesen werden. Dabei verhält sich der Socket wie ein File-Deskriptor und kann sogar in *FILE** umgewandelt werden, um höhere I/O-Funktionen, wie *printf* oder *scanf*, zu benutzen:

```
int read(int socket,char[] *buffer, int maxbuf)
int write (int socket, char[] *buffer, int length)
```

Beide Funktionen spezifizieren den Socket, übergeben/übernehmen einen String der die Daten enthält und geben die Länge des Strings oder die maximale Länge des Strings an.

Die *close*-Anweisung dient zum Verbindungsabbau. Sockets sollten auf jeden Fall geschlossen werden, auch bei erfolgloser Verbindung oder Programmabruch, da sie sonst vom Betriebssystem geschlossen werden müssen, dadurch länger existieren können und die

Verbindung aufrecht erhalten, wobei andere Client's sich nicht zum Server verbinden können. Nach dem *close* steht beim Server wieder der Socket zur Verfügung mit dem er Verbindugen entgegennimmt, soll nun der Server terminieren, also keine Verbindungen mehr annehmen, muss auch dieser Socket geschlossen werden.

3.3 Unverlässliche Verbindung mit UDP

Im Gegensatz zu TCP ist UDP eine unverlässliche Übertragungsart, d.h. es werden keine Vorkehrungen getroffen um verlorene Botschaften neu zu senden oder die Reihenfolge beim Sender wiederherzustellen. Abbildung 3.3/1 zeigt den Verbindungsablauf:

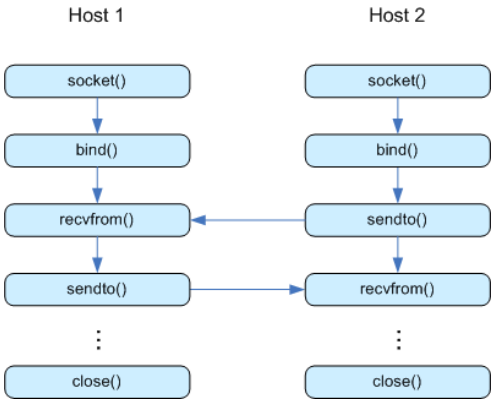

Abbildung 3.3/1 UDP-Verbindungsablauf

Die Abbildung zeigt unter anderem schon bekannte Routinen. Ein wesentlicher Unterschied besteht darin das, im Unterschied zum TCP-Protokoll, gleichberechtigte Kommunikationspartner (im Sinne der Rollenverteilung) vorhanden sind. Jeder kann Senden und Empfangen zu/von unterschiedlichen Verbindungspartnern, dazu ist es nötig den eigenen Socket zu binden. *Recvfrom* und *sendto* sind den *read-* und *write*-Funktionen sehr ähnlich. Ihnen wird zusätzlich die Datenstruktur sockaddr für den Verbindungspartner als Parameter übergeben. Nach erfolgter Kommunikation wird der Socket mit *close* wieder geschlossen.

3.4 Dateisystemsockets

In Unix-Systemen besteht die Möglichkeit, das Prozesse über das Dateisystem, also gemeinsamen Speicher, kommunizieren. Dabei sind die Schritte für einen Verbindungsablauf dieselben wie bei der Kommunikation mit TCP. Dazu wird im Dateisystem ein Socket angelegt, der wie eine Datei im aktuellen Verzeichnis vorliegt. Soll der Socket gebunden werden, muss statt der bisher bekannten Adressstruktur eine Struktur mit dem Pfad des Sockets angegeben werden (siehe Anhang). Diese Art der Verwendung findet man z.B. bei Druckerservern wieder. Die weitere Benutzung unterscheidet sich nicht von der bisher bekannten Vorgehensweise.

3.5 Raw Sockets

Raw Sockets sollen an dieser Stelle nur wegen der Vollständigkeit erwähnt werden. Sie unterscheiden sich grundsätzlich von dem bisher kennengelernten Socketkonzept. Sie benutzen nicht die Protokolle der Host-to-Host-Schicht, sondern Protokolle der Internetwork-Schicht, wie z.B. ICMP. Weil Raw Sockets eine Schicht tiefer ansetzen, vermeiden sie das zusätzliche Einpacken in TCP/UDP-Packete und gewinnen dadurch auch an Performanz. Diese Vorteile werden aber auch durch eine Reihe an Nachteilen, wovon einige nachfolgend aufgelistet sind:

- Das Portkonzept ist nicht vorhanden in der Internetwork-Schicht, d.h. eintreffende Nachrichten werden an alle Sockets gesandt die das entsprechende Protokoll implementieren. Die Pakete müssen also im Prozess gefiltert werden und die nicht benötigten verworfen werden.
- Der ausführende Prozess muss root-Rechte besitzen.
- Die Verlässlichkeit geht verloren, wie auch viele andere Eigenschaften von Protokollen der Host-to-Host-Schicht.

3.6 Nicht blockierendes Empfangen

Da in CYBOI nur eine einzige Signalschleife zum Einsatz kommt, ist es nötig die Möglichkeit von nicht blockierendem Empfangen in Betracht zu ziehen, da sonst die gesamte Signalschleife auch für anderweitige Signale nicht weiterarbeiten könnte.
Dazu müssen nach dem Anlegen der Sockets die Flags m.H. der Funktion *fcntl* gesetzt werden:

```
int flags = fcntl( socket, F_GETFL, 0)
fcntl(socket, F_SETFL, O_NONBLOCK|flags)
```

Der *accept* – Aufruf arbeitet jetzt nicht blockierend. Im Fall von UDP-Sockets ist die Anwendung von *fcntl* äquivalent, nur das jetzt der *recvfrom*-Aufruf nicht blockiert.

4 Schlussbemerkungen

4.1 Zusammenfassung

In den vorigen Kapiteln wurde das CYBOP-Projekt und dessen Architektur vorgestellt, im Kapitel 3 das grundlegende Socketkonzept und die wichtigsten Protokolle und Sockettypen erläutert, weiterhin wurde noch kurz aufgezeigt wie sich Sockets im Hinblick auf das nicht blockierende Empfangen implementieren lassen.

Sockets ermöglichen verteilte Anwendungen und sind die Grundlage für höhere Kommunikationsparadigmen wie z.B. Fernaufrufe oder strombasierte Kommunikationsmodelle. Das in vielen Betriebssystemen verfügbare API ermöglicht es portable Anwendungen zu erzeugen. Ausserdem sind Sockets in vielen Programmiersprachen, z.B. Java, C++, integriert. Somit sind Sockets auch 20 Jahre nach ihrer Einführung ein wichtiges und grundlegendes Kommunikationsparadigma.

Der Beitrag dieser Arbeit zum CYBOP-Projekt ist es die Integration des Socketkonzepts in CYBOI zu ermöglichen. Mit Hilfe dieses Konzepts ist es erst möglich mit CYBOP Anwendungen zu erstellen die über ein Netzwerk kommunizieren können. CYBOP spezifische Problematiken wurden ebenso herausgestellt, wie auch der allgemeine Kommunikationsablauf in Hinblick auf die Verwendung eingehend erläutert. Im Anhang werden weiter nützliche Implementierungsdetails wie z.B. der Aufbau der Adressstrukturen gezeigt. Das implementierte Programmbeispiel verdeutlicht ausserdem praktisch die Arbeitsweise des Kommunikationskonzepts. Damit wurde ein grundlegender Schritt zur Erweiterung bzw. Weiterentwicklung des CYBOI vorbereitet.

4.2 Ausblick

Das gesamte Spektrum des Themengebietes der Sockets konnte hier nur oberflächlich beleuchtet werden. So gibt es noch viele mögliche Anwendungsfelder die mit in den CYBOI einfliessen könnten, so z.B. Multicast- bzw. Broadcast-Nachrichten, sichere Verbindungen mit OpenSSL, Maßnahmen zur Sicherung von Servern oder aber das Generieren von Fernaufrufen aus Socketverbindungen und viele weitere.
Der Einsatz von Sockets wirft aber auch eine Reihe von Problemfeldern auf, die in dieser Art in „nicht vernetzten“ Anwendungen nicht enthalten sind. So kann zum Beispiel die synchrone Übertragung und parallele Abarbeitung unter Umständen zu Verklemmungen führen.
Schon allein die Tatsache, das unterschiedliche Prozessortypen verschiedene Datenrepräsentationen enthalten (Stichwort little endian und big endian) und somit eine Konvertierung nötig ist, zeigt die unter keinen Umständen vernachlässigbare Komplexität verteilter Anwendungen auf.

Abbildungsverzeichnis

Literaturanmerkungen

Eine gute Einführung in das Thema der Sockets mit Beispielen wird gegeben in [Mat 01].
Eine umfassende Einführung in das Thema mit Beleuchtung vieler Randgebiete, wie IP-Adressierung, und vertiefenden Informationen über das Socketkonzept, sowie einem sehr umfangreichen Anhang gibt [Wal 01].
Eine kurze Zusammenfassung der Portierung von Sockets auf das Ipv6-Protokoll kann gefunden werden bei [Hyn 02].
Eine knappe Einführung in Form eines Übungsbuches gibt [Krü 02].

Literaturquellen

[Heller 2004]	Heller, Christian. Cybernetics Oriented Language (CYBOL). IIIS Proceedings: 8th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2004). July, 2004.
[Mat 01]	Neill Mathew und Richard Stones, Linux-Programmierung 1. Auflage, Verlag mitp, 2001
[Wal 01]	Sean Walton, Linux Socket Programming 1.Auflage, Verlag SAMS, 2001
[Hyn 02]	Hynek, Portieren von Software nach IPv6, 2002 http://me.in-berlin.de/~hys/proting2ipv6.html
[Krü 02]	Gerhard Krüger und Dietrich Reschke, Lehr- und Übungsbuch Telematik 2.Auflage, Carl Hanser Verlag, 2002

Anhang

Domänen

AF_UNIX	Dateisystemsockets
AF_INET	IP-Protokolle wie TCP oder UDP
AF_INET6	Protokolle die IPv6 statt IPv4, da IP ein Protokoll der Internetwork-Schicht ist, werden hier weiterhin die bekannten Protokolle aus der AF_INET-Domäne verwendet
AF_ISO	ISO-Protokolle
AF_IPX	IPX-Protokoll von Novell

Adressstrukturen