Implementation eines Netzwerkprotokolls zur serverseitigen Clientkonfiguration

Bachelorarbeit im Studiengang Informatik

an der HTWK-Leipzig(FH)

Eingereicht von Falko Hofmann, 99 IN B(T)

19. September 2002

Erklärung

Hiermit versichere ich, daß ich die Arbeit selbstständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und daß alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben der Quellen kenntlich gemacht worden sind.

Falko Hofmann

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

In den letzten Jahren hat sich die Anzahl der Internetzugänge stark vergrößert. Gleichzeitig haben sich auch die verfügbaren Bandbreiten der Netzzugänge enorm erhöht. Diese Entwicklungen ermöglichen den verbreiteten Einsatz von multimedialen Anwendungen zur Kommunikation in Netzen mit einer akzeptablen Übertragungsqualität. Zu diesen Anwendungen sind zum Beispiel Internettelefonie- und Videokonferenzanwendungen zu zählen. Diese Einsatzgebiete stellen aber nicht nur an die Bandbreite der Netzwerke erhöhte Anforderungen, sondern auch an die verwendete Hard- und Software. Im Gegensatz zu konventionellen Anwendungen die Dienste wie zum Beispiel FTP und E-Mail bereitstellen, setzen diese neuen Anwendungen kontinuierliche Datenströme und geringe Antwortverzögerungen voraus. In den meisten der verwendeten Protokollstandards waren keine oder nur ungenügende Mechanismen zur Gewährleistung dieser Eigenschaften vorgesehen. Aus diesem Grund wurde es nötig, Erweiterungen für diese Standards zu schaffen, die die geforderten Qualitätsansprüche gewährleisten können. Als Beispiel sei hier der H.323-Standard der International Telecommunications Union (ITU) genannt. Dieser Standard ist eine Erweiterung für paketvermittelnde Netzwerke wie zum Beispiel Ethernet. Ebenso wichtig wie die Gewährleistung der Übertragungsqualität zwischen den Kommunikationspartnern ist die möglichst optimale Abstimmung der verwendeten Hard- und Software der am Kommunikationsprozeß beteiligten Rechner. Diese Abstimmung verlangt unter Umständen einen gesteigerten Konfigurationsaufwand.

Eine weitere Folge der eingangs genannten Entwicklungen ist, daß völlig neue Nutzerkreise erschlossen wurden. Diese Nutzergruppen interessiert zu allererst der praktische Nutzen dieser Anwendungen. Es kann nicht vorausgesetzt werden, daß die nötigen Kenntnisse und die Motivation vorhanden sind, sich mit aufwendigen Konfigurationsprozessen zu beschäftigen. Beispielhaft kann hierfür die Entwicklung bei den Betriebssystemen gesehen werden. Die Akzeptanz der WINDOWS-Betriebssysteme liegt darin

begründet, daß nur Grundlagenkenntnisse im Umgang mit Computern nötig sind, um mit diesen Rechnern zu arbeiten. Im Gegensatz dazu setzen z.B. LINUX-Systeme zur Zeit noch größere Kenntnisse über Betriebssysteminterna voraus. Dies ist ein Grund, weshalb solche Systeme schwerer Akzeptanz bei Nutzern mit geringen Vorkenntnissen finden.

Die Schlussfolgerung kann daher nur lauten, daß auch bei Netzwerkanwendungen die nötigen Konfigurationsprozesse so weit wie möglich vereinfacht oder vollständig automatisiert werden müssen. Eine Variante zur Vereinfachung ist es, die Software auf eine spezielle Hardware auszulegen. Dies führt aber eventuell zu Qualitätseinbußen, wenn nicht alle beteiligten Partner das gleiche Paket aus Hard- und Software einsetzen. Eine andere Möglichkeit besteht darin, die Ausstattung der beteiligten Rechner zentral abzufragen. Aus diesen Daten werden dann auf dem zentralen Rechner für alle anderen die Parameter ermittelt, mit denen diese in der bestmöglichen Qualität an der Kommunikation teilnehmen können.

Diesem Ansatz folgend ist der Gegenstand dieser Arbeit der Entwurf und die Implementierung eines Netzwerkprotokolls. Dieses Protokoll muß die Übermittlung der von einem zentralen Rechner angeforderten Ausstattungsdaten der beteiligten Rechner und die Übermittlung der daraus gewonnen Parameter zurück an die beteiligten Rechner unterstützen. Aufgrund der freien Verfügbarkeit wurde die Implementation des Protokolls auf Basis des Betriebssystems LINUX durchgeführt.

Zum allgemeinen Verständnis des Inhaltes dieser Arbeit erfolgt in Kapitel 2 zunächst eine Einführung in das Thema. In diesem Kapitel werden die Grundlagen von Netzwerken, das TCP/IP-Referenzmodell sowie die zur Netzwerkprogrammierung zur Verfügung stehende Socket-API erläutert. Das nachfolgende Kapitel wendet sich dem Entwurf des Protokolls zu. Dazu erfolgt eine ausführliche Analyse des Einsatzfeldes und die Formulierung der detaillierten Protokollanforderungen. Diese Anforderungen bilden die Grundlage für die Spezifikation des Protokolls und seiner Schnittstellen.

Nach diesen eher abstrakten Festlegungen befasst sich Kapitel 4 mit der praktischen Implementierung des Protokolls und seiner Schnittstellen. Es wird dabei auf wichtige Entwurfsentscheidungen und verwendete Konzepte eingegangen. Die Umsetzung wird anhand von Quelltextauszügen veranschaulicht. Eine Beispielanwendung demonstriert die Verwendung der Schnittstellen. Das daran anschließende Kapitel 5 ist dem Test der Protokollimplementation gewidmet. Dazu werden die in einer Testumgebung gewonnenen Ergebnisse in Bezug auf die Anzahl der gleichzeitig verarbeitbaren Clientanfragen dokumentiert. Ein weiterer Abschnitt dieses Kapitels wendet sich der Auswertung der Fehleranfälligkeit des Protokolls zu. Im abschließenden Kapitel werden mögliche Verbesserungen und Erweiterungen diskutiert.

Kapitel 2

Theoretische Grundlagen Netzwerkprotokolle

In diesem Kapitel werden zum besseren Verständnis die Grundlagen von Netzwerkprotokollen erläutert. Da dieses Gebiet sehr weitläufig ist, kann an dieser Stelle nur auf für die vorliegende Arbeit relevante Themen eingegangen werden. Als weiterführende Literatur sei an dieser Stelle auf [TAN96] verwiesen.

Die Grundlage für die Kommunikation von Rechnern in einem Netzwerk bilden Protokolle. In solchen Protokollen werden Regeln und Konventionen festgeschrieben, die von den Kommunikationspartnern einzuhalten sind. In Protokollen werden z.B. das Datenformat und die Bedeutung der Daten definiert.

Zu Beginn der Entwicklung von Rechnernetzwerken wurden Protokolle noch komplett auf einen Netzwerktyp ausgelegt, das heißt, diese Protokolle konnten nur innerhalb eines homogenen Netzwerkes genutzt werden. Schon kurze Zeit später kamen jedoch Bestrebungen auf, verschiedene Netzwerke miteinander zu verbinden. Eine Möglichkeit war die Erweiterung der bestehenden Protokolle, um sie auch für einen anderen Netzwerktyp nutzbar zu machen. Dieser Ansatz hat den Nachteil, daß die Erweiterung der bestehenden Protokolle mit sehr großem Aufwand verbunden ist und damit auch die Fehleranfälligkeit steigt.

Um dieses Problem effizient lösen zu können, wurden Kommunikationsmodelle entwickelt. Diese Modelle enthalten in Schichten angeordnete Protokollhierarchien. Jede Schicht stellt der jeweils darüberliegenden Schicht bestimmte Dienste zur Verfügung und nutzt Dienste der darunterliegenden Schicht. Die den Dienst nutzende Schicht hat dabei keine Kenntnis, wie die darunterliegende Schicht diesen Dienst erbringt. Die von einer Schicht angebotenen Dienste werden von der darüberliegenden Schicht über definierte Schnittstellen genutzt.

Bei der Verwendung eines solchen Kommunikationsmodelles erfolgt der Datenaus-

tausch zwischen zwei Anwendungen auf unterschiedlichen Rechnern in zwei Richtungen. Zwei Schichten der gleichen Hierarchiestufe auf verschiedenen Rechnern kommunizieren unter Nutzung des zu dieser Schicht gehörenden Protokolls miteinander. Diese Kommunikation findet aber nur virtuell statt. Der physikalische Datenfluß verläuft von der sendenden Schicht durch die unter ihr liegenden Schichten bis zum physikalischem Medium und auf dem Zielrechner in umgekehrter Reihenfolge hinauf bis zur Schicht der gleichen Stufe. Je nach Funktionalität der einzelnen Schichten werden an die eigentlichen Nutzdaten zusätzliche Kontrollinformationen angefügt. Jede Schicht behandelt die Daten, die von einer darüberliegenden Schicht geliefert werden komplett als Nutzdaten - sie hat also keine Kenntnis über den eigentlichen Inhalt der Daten.

Diese Schichtenmodelle haben den Vorteil, das z.B. das gesamte Protokoll einer Schicht gegen ein leistungsfähigeres ausgetauscht werden kann, ohne daß die darüber liegende Schicht geändert werden muß. Einzige Bedingung hierbei ist, daß die Schnittstellen für die darüberliegende Schicht beibehalten werden.

Die bekanntesten aus diesen Bestrebungen entstandenen Modelle sind das ISO/OSI-Modell und das TCP/IP-Modell. Während das siebenschichtige OSI(Open System Interconnection)-Modell in der Praxis kaum von Bedeutung ist, sondern hauptsächlich zur Bewertung anderer Modelle benutzt wird, ist heutzutage das TCP/IP-Modell das am weitesten verbreitete Modell. Benannt wurde das Modell nach seinen wichtigsten Protokollen, dem TCP-Protokoll (Transport Control Protocol) und dem IP-Protokoll (Internet Protocol). Deshalb wird an dieser Stelle hauptsächlich auf das TCP/IP-Protokoll eingegangen werden.

2.1 Das TCP/IP-Referenzmodell

Das TCP/IP-Referenzmodell definiert im Unterschied zum OSI-Modell nur 4 Schichten. Die im TCP/IP-Modell definierten Schichten haben folgende Aufgaben:

* **Verarbeitungsschicht (Application Layer):** Die Verarbeitungsschicht umfasst alle höherschichtigen Protokolle (z.B. TELNET, FTP). Diese Schicht ist für die Bereitstellung von Diensten, z.B. den E-Mail-Transfer, für darüberliegende Anwendungen zuständig .

* **Transportschicht (Transport Layer):** Die Transportschicht ist für die Ende-zu-Ende Verbindung zwischen zwei Prozessen auf verschiedenen Rechnern zuständig. Für diese Aufgabe sind zwei Protokolle vorgesehen: TCP (Transmission Control Protocol) und UDP (User Datagram Protocol).

* **Internetschicht (Internet Layer):** Die Internetschicht ist für die Übertragung von Datenpacketen (Datagrammen) und die Wegewahl (Routing) zuständig. Die in der TCP/IP-Protokollsuite zur Verfügung stehenden Protokolle sind: IP (Internet Protocol), ICMP (Internet Control Message Protocol) und IGMP (Internet Group Management Protocol).

* **Netzzugangsschicht (Network Access Layer):** Die Netzzugangsschicht ist die unterste Schicht des Modells. Diese Schicht gewährleistet den physikalischen Zugriff auf das Netzwerk. Sie ermöglicht weiterhin den Versand und den Empfang von Daten über das Netzwerkinterface. Die verwendeten Protokolle differieren je nach zugrundeliegenden Netztyp. Einige der bekanntesten in dieser Schicht verwendeten Protokolle sind zum Beispiel Ethernet, FDDI und ISDN.

In Abbildung 2.1 ist zur Veranschaulichung das Referenzmodell und eine Auswahl der Protokolle der TCP/IP-Protokollsuite inklusive ihrer Zuordnung zu den jeweiligen Schichten zu sehen.

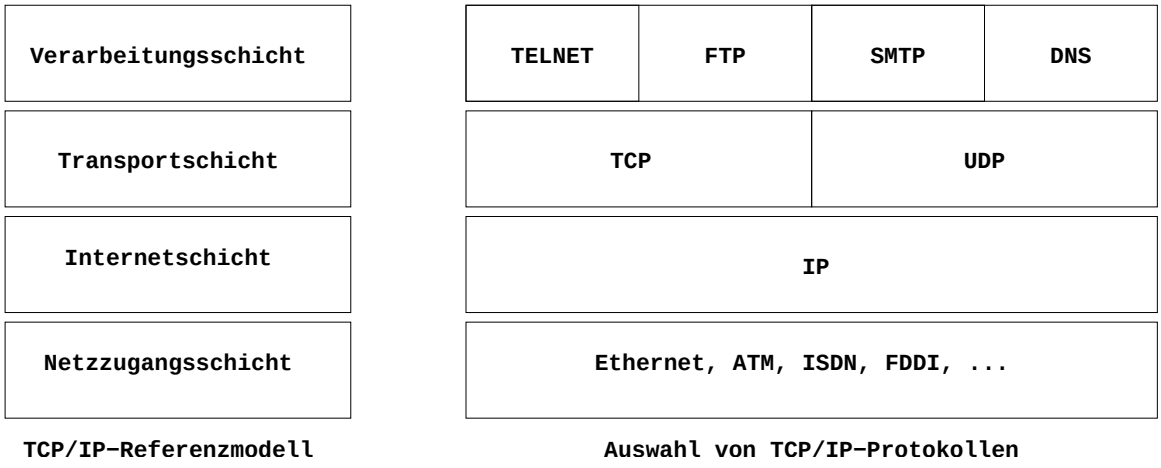

Abbildung 2.1: TCP/IP-Referenzmodell und Protokolle der TCP/IP Protokollsuite

An dieser Stelle werden zum besserem Verständnis die Eigenschaften der in der Transportschicht angesiedelten Protokolle TCP und UDP der TCP/IP-Protokollsuite näher erläutert.

Eigenschaften von TCP:

* Verbindungsorientiert, d.h es wird vor der Datenübertragung eine virtuelle Verbindung zwischen den beiden beteiligten Rechnern aufgebaut und nach dem Ende der Übertragung explizit wieder abgebaut.

* Flussorientiert, d.h. die übertragenen Daten kommen im Zielrechner genau in der selben Reihenfolge an, in der sie vom Quellrechner versendet wurden. Weiterhin wird dafür gesorgt, daß der Datenpuffer eines langsamen Empfängers nicht durch einen schnellen Sender überlastet werden kann.

* zuverlässige Datenübertragung, d.h es wird sichergestellt, daß es beim Transport der Daten nicht zu Verlusten oder Verfälschungen dieser Daten kommt. Zu diesem Zweck werden Checksummen aus den Daten gebildet und im Header übermittelt.

Der Vorteil von TCP liegt in der hohen Sicherheit der Datenübertragung, jedoch wirkt sich dies nachteilig auf die Übertragungsgeschwindigkeit aus, da jedes Datenpacket einen relativ großen Overhead an Kontrolldaten umfasst. Als alternatives Protokoll dazu gibt es UDP.

Eigenschaften von UDP:

* Verbindungslos, d.h. es wird keine Verbindung zwischen den beteiligten Rechnern aufgebaut.

* Versendung einzelner Datagramme, d.h es wird nicht garantiert, daß die Pakete in der selben Reihenfolge beim Zielrechner eintreffen, in der sie vom Quellrechner versendet wurden.

* Unsichere Datenübertragung, d.h es werden keine zusätzliche Maßnahmen zum Schutz vor Datenverlust bzw. Datenverfälschungen getroffen.

Der Vorteil dieses Protokolls liegt in der hohen Geschwindigkeit bei der Datenübertragung, da auf den Overhead von TCP verzichtet wird. Natürlich ergibt sich daraus ein höherer Aufwand bei der Verwendung dieses Protokolls, da die Fehlersicherung und die sequentielle Ordnung der Datagramme in der Verarbeitungsschicht sichergestellt werden müssen.

Aus den oben genannten Charakteristika und den sich daraus ergebenden Vor- und Nachteilen der jeweiligen Protokolle kann man ableiten, daß i.A. TCP beim Transport von Daten mit sensiblen Inhalten zur Anwendung kommen sollte, allerdings unter der Bedingung, daß die Datenmengen relativ klein sind. Für die Übertragung großer Datenmengen, bei denen sich einzelne fehlende Datagramme nicht negativ auf das Gesamtergebnis auswirken, ist es besser das schnellere UDP zu verwenden.

2.2 Client/Server Prinzip

Das Client/Server-Modell ist eines der am häufigsten verwendeten Kommunikationsmodelle in Netzwerken. Eines der Ziele dieses Kommunikationsmodelles ist die effiziente Nutzung von Ressourcen und Applikationen, indem sie zur zentralen Nutzung bereitgestellt werden. Beispiele für die Anwendung dieses Modells sind z.B. Webserver, Printserver und Applikationsserver.

Bei diesem Modell wird von mindestens zwei Kommunikationspartnern ausgegangen. Einer der Partner, der Server, bietet einen Dienst an. Dieser Dienst wird von dem anderen Partner, dem Client, in Anspruch genommen. Der Server ist bei dieser Art der Kommunikation der passive Partner, der auf die Anfrage eines Clients wartet. Nach erfolgter Anfrage bauen Server und Client eine eigene Verbindung auf, über die sie miteinander kommunizieren können. Am Ende der Kommunikation wird die Verbindung auf beiden Seiten geschlossen und der Server wartet auf weitere Anfragen.

Der Server ist über ein eindeutiges Paar aus IP-Adresse und Portnummer, das vor Beginn der Kommunikation fest zugewiesen wird, für den Client identifizierbar. Der Client besitzt nur eine vorher festgelegte, eindeutige IP-Adresse und bekommt nur für die Zeit der Kommunikation eine Portnummer zugeteilt.

Ein Nachteil dieses Modells ist die Konzentration auf einen zentralen Kommunikationspunkt. Dies kann bei vielen gleichzeitigen Anfragen zur Überlastung des Servers führen, so daß er keine weiteren Clientanfragen entgegennehmen kann. Zur Umgehung dieses Problems wurden in den letzten Jahren verschiedene andere Modelle entwickelt, die meist davon ausgehen, daß nicht die gesamte Last auf Seiten des Servers liegt.

2.3 Die Berkeley Socket-API

Nach der Veröffentlichung des TCP/IP-Referenzmodells entwickelte die Universität Berkeley eine Socket-API, die dieses Modell umsetzte. API steht für Application Program Interface. Diese API wurde in BSD-UNIX integriert und hat sich aufgrund der Offenlegung der Quellen als Quasistandard etabliert. Sie wird deshalb von allen BSD-UNIX basierten Systemen wie z.B. LINUX und UNIX SVR4, aber auch zu großen Teilen von anderen Systemen, wie z.B. WINDOWS direkt oder über Bibliotheken, unterstützt.

Die Berkeley Socket-API wurde als abstrakter Vermittler zwischen verschiedenen Netzwerkprotokollen entwickelt. Dies führt zwar zu einer komplexeren Schnittstelle, hat aber den großen Vorteil, daß neue Protokolle ohne Änderung der Schnittstelle hinzugefügt werden können. Ein Socket ist hierbei ein Kommunikationsendpunkt, der

durch eine IP-Adresse und eine Portnummer eindeutig identifiziert werden kann.

Die Socket-API definiert eine Anzahl von Operationen und Datenstrukturen, die den einheitlichen Zugriff auf die Protokolle der Transportschicht ermöglichen. Aufgrund der Herkunft der Socket-API verwendet sie viele Konzepte, die aus UNIX bekannt sind. So erfolgt die Ein-/Ausgabe wie bei Dateien über Deskriptoren, welche das Objekt kennzeichnen, auf dem Operationen ausgeführt werden. Wie in UNIX üblich basieren alle Operationen auf dem Prinzip open-read/write-close, d. h. beim Öffnen eines Sockets wird ein Deskriptor zurückgegeben, der bei allen nachfolgenden read/write-Operationen nur noch als Argument angegeben werden muß.

Auf die Verwendung von Sockets zur Interprozesskommunikation auf lokalen Rechnern, auch als UNIX-Domain Sockets bezeichnet, wird an dieser Stelle nicht weiter eingegangen, sondern nur auf die Nutzung der Sockets zur Netzwerkkommunikation unter Verwendung der TCP/IP-Protokollsuite.

In Tabelle 2.1 sind die wichtigsten Funktionen für den Zugriff auf die Socket-API mit einer kurzen Erläuterung ihrer Bedeutung aufgelistet.

Funktion	Beschreibung
socket()	Anlegen des Sockets
bind()	Socket an die Adressinformationen binden
listen()	Einrichten der Verbindungswarteschlange
accept()	Warten auf Verbindungswünsche von Clients
connect()	Verbindungswunsch an den Server senden
read()/write() send()/recv()	Übertragung von Daten über einen STREAM- Socket
sendto()/recfrom()	Senden bzw. Empfangen von Datagrammen über einen DGRAM-Socket
close()	Schließen des Sockets, Verbindungsende

Tabelle 2.1: Funktionsübersicht Socket-API

Basierend auf den zwei wichtigsten Protokollen der Transportschicht, TCP und UDP, haben sich zwei unterschiedliche Socket-Typen für die Netzwerkkommunikation etabliert, der STREAM-Socket und der DGRAM-Socket. Dabei realisiert ein STREAM-Socket die Funktionalität von TCP, der DGRAM-Socket die Funktionalität von UDP. Je nach verwendetem Typ unterscheiden sich die Kommunikationsabläufe und verwendeten Operationen.

Bei der Verbindung mittels eines STREAM-Sockets müssen auf Seite des Servers vier Schritte durchlaufen werden, ehe auf eingehende Verbindungswünsche von Clients reagiert werden kann. Im ersten Schritt muß mit der Funktion *socket()* ein Socket angelegt werden. Im Anschluß muß dieser Socket mit der Funktion *bind()* an die

Adressinformationen gebunden werden. Als nächstes muß mit dem Aufruf von *listen()* eine Verbindungswarteschlange eingerichtet werden. Wurden diese Schritte erfolgreich durchlaufen, kann die *accept()*-Funktion aufgerufen und damit auf eingehende Verbindungswünsche von Clients reagiert werden. Ist der *accept()*-Aufruf erfolgreich, d.h. kam es zu einer Verbindungsanfrage, so ist das Ergebnis eine Punkt-zu-Punkt Verbindung zwischen dem Client und dem Server. Nun können zwischen Client und Server beliebige Daten mittels der Funktionen *read()/write()* ausgetauscht werden. Mit dem Aufruf der Funktion *close()* wird der Socket und damit die Verbindung geschlossen.

Der Verbindungsaufbau über einen STREAM-Socket erfordert auf der Clientseite weniger Schritte. Im ersten Schritt ist es wie auf Serverseite notwendig, mit dem Aufruf der Funktion *socket()* einen neuen Socket anzulegen. Danach wird durch den Aufruf der Funktion *connect()* eine Verbindungsanfrage gesendet. War dieser Aufruf erfolgreich, ist die Verbindung zwischen Client und Server etabliert und der Datenaustausch kann beginnen. Für das Schließen des Sockets und damit den Verbindungsabbau steht, wie auf der Serverseite, die Funktion *close()* zur Verfügung. In Abbildung 2.2 ist dieser Ablauf zur Veranschaulichung schematisch dargestellt.

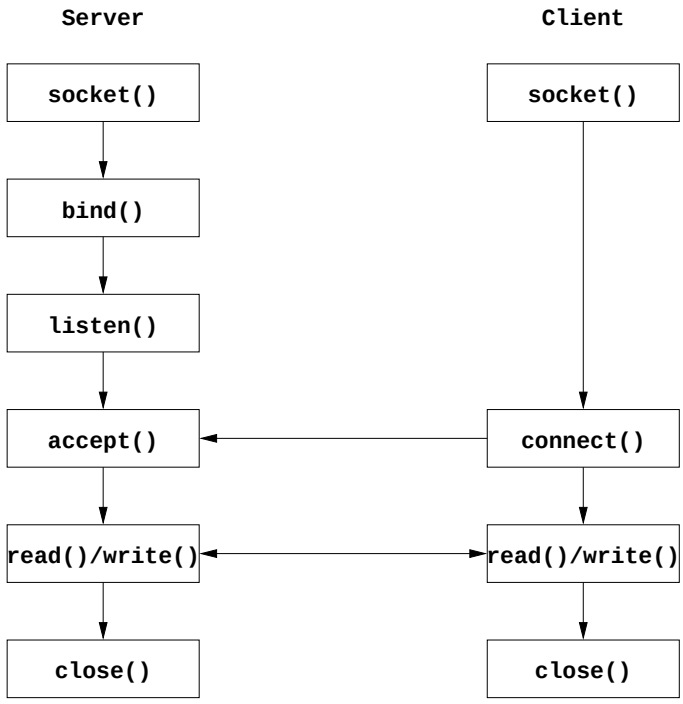

Abbildung 2.2: Kommunikationsablauf mittels STREAM-Socket

Für den Aufbau einer Kommunikation mittels DGRAM-Sockets sind aufgrund der verbindungslosen Kommunikation weniger Initialisierungsschritte auf der Client- und der Serverseite notwendig, als bei der Kommunikation mittels STREAM-Sockets. Die

nötigen Schritte sind in diesem Fall auf der Client- und der Serverseite gleich. Zuerst muß auf beiden Seiten durch den Aufruf der Funktion *socket()* jeweils ein neuer Socket angelegt werden. In nächsten Schritt müssen die auf beiden Seiten kreierten Sockets mit der Funktion *bind()* an die Adressinformationen gebunden werden. Nach diesem Schritt sind beide Kommunikationspartner in der Lage mit den Funktionen *sendto()/recvfrom()* Datagramme zu versenden bzw. zu empfangen. Das Ende der Kommunikation wird durch das Schließen des Sockets mittels der Funktion *close()* ausgelöst. In Abbildung 2.3 ist dieser Ablauf zur Veranschaulichung schematisch dargestellt.

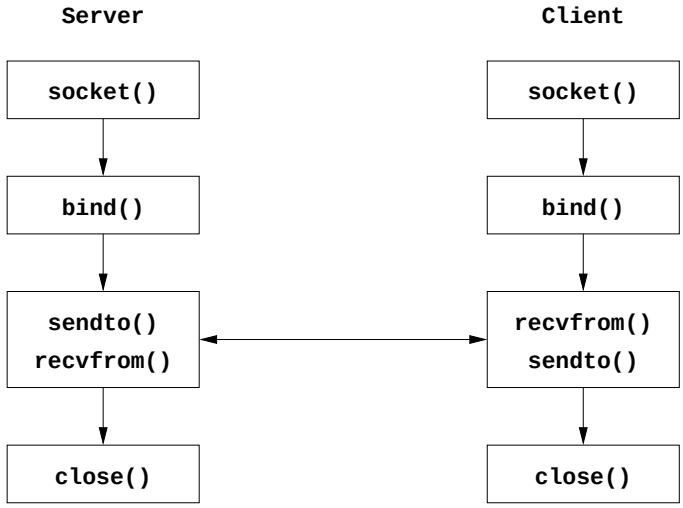

Abbildung 2.3: Kommunikationsablauf mittels DGRAM-Socket

Auf die genaue Anwendung der Funktionen und die von ihnen verwendeten Strukturen sei an dieser Stelle auf das Kapitel 4 verwiesen. Eine umfassende Einführung zum Thema Socketkommunikation mit verschiedenen Beispielen ist in [HER99] zu finden. Der Hauptschwerpunkt liegt dort allerdings auf der Anwendung von UNIX-Domain Sockets. Eine Einführung zum Thema Sockets in Netzwerkumgebungen mit Beispielen ihrer Anwendung ist z.B. in [BEE01] zu finden.

Kapitel 3

Entwurfsentscheidungen

Nachdem im vorangegangenen Kapitel ein kurzer Überblick über die theoretischen Grundlagen von Netzwerkprotokollen gegeben wurde, folgt zunächst eine Analyse des Einsatzfeldes. Im Anschluß werden die Protokollanforderungen formuliert. Aus diesen Anforderungen wird dann die genaue Spezifikation des Protokolls abgeleitet.

3.1 Analyse des Einsatzfeldes

Wie schon in Kapitel 1 beschrieben, ist ein Haupteinsatzfeld für das zu entwickelnde Protokoll im Bereich der Anwendung von Videokonferenzsoftware oder ähnlich gelagerter netzwerkgestützter Anwendungen zu sehen, bei denen die Abstimmung der Hard- und Softwarekomponenten wichtig ist, um eine optimale Übertragungsqualität zu gewährleisten. Viele der bisher verfügbaren Anwendungen erfordern die manuelle Abstimmung der Verbindungsparameter zwischen den beteiligten Kommunikationspartnern durch die jeweiligen Nutzer. Dies erfordert jedoch eine erweiterte Kenntnis der verwendeten Hard- und Software. Da jedoch das potentielle Nutzerfeld solcher Anwendungen durch die große Verbreitung von Rechnern mit leistungsfähigen Netzzugängen nicht nur im Bereich der Informatik anzusiedeln ist, können diese Kenntnisse nicht vorausgesetzt werden. Deshalb ist es von großem Vorteil, diese verbindungsabhängigen Konfigurationsvorgänge zu automatisieren. Um dies jedoch umzusetzen sind mehrere Voraussetzungen zu erfüllen. So wird neben der reinen Kommunikations-Software auf jedem an der Kommunikation teilnehmenden Rechner eine Softwarekomponente benötigt, die es ermöglicht die aktuelle Hard- und Softwarekonfiguration abzufragen. Weiterhin ist auf dem Server eine Softwarekomponente nötig, die in der Lage, ist die Konfigurationen aller teilnehmenden Rechner auszuwerten und daraus die optimalen Parameter für alle abzuleiten. Als letzte Komponente ist ein Hilfsprotokoll von nöten, das fähig ist, diese Konfigurationsdaten von den Clientrechnern zum Server und die

Parameter vom Server zu den beteiligten Clients zu übertragen. Die Anforderungen an solch ein Hilfsprotokoll werden im Abschnitt 3.2 formuliert.

3.2 Protokollanforderungen

Die Formulierung der Protokollanforderungen erfolgt aus Sicht darüberliegenden Anwendungen. Es sind zwei Szenarien für den Kommunikationsbeginn möglich. Diese sind nachstehend näher erläutert.

* Szenario 1: Im Szenario 1 bestehen schon ein oder mehrere Client/Server - Verbindungen auf Ebene der Anwendung, und der Austausch der Konfigurationsdaten wurde zuvor erfolgreich abgeschlossen. Während der laufenden Verbindung kommen von anderen Clients Anfragen mit dem Ziel, an der Kommunikation teilnehmen zu dürfen. Bevor dies geschehen kann, müssen zwischen Client und Server erst die Konfigurationsdaten ausgetauscht werden. Erst nach dem erfolgreichen Austausch der Daten und der sich daran anschließenden Konfiguration der Anwendung auf dem Client, kann der neue Client an der bestehenden Kommunikation teilnehmen.

* Szenario 2: Im Szenario 2 besteht noch keinerlei Verbindung auf Anwendungsebene zwischen dem Server und ein oder mehreren Clients. Um die Clients davon zu unterrichten, daß der Server kommunikationsbereit ist, sendet er an ihm bekannte Clients eine Nachricht. Verbindungswillige Clients können nun mit dem Server in Interaktion treten und die Konfigurationsparameter mit ihm austauschen, um bei Erfolg die Kommunikation über die Anwendung zu beginnen.

Die Abbildung 3.1 veranschaulicht die beiden Kommunikationsszenarien. In der Darstellung des Szenarios 1 ist zu erkennen, daß Client 1 und 2 schon auf Ebene der Anwendung kommunizieren. Client 3 möchte nun an dieser Kommunikation teilnehmen - dazu sendet er eine Verbindungsanfrage an den Server. Die Darstellung des Szenarios zeigt, daß der Server an drei ihm bekannte Clients eine Start-Nachricht sendet. Der ebenfalls kommunikationswillige Client 3 sendet daraufhin eine Verbindungsanfrage an den Server. Die beiden anderen Clients reagieren nicht, da sie z.B. nicht aktiv sind oder kein Bedarf an der Teilnahme an dieser Verbindung besteht.

Da es sich bei den Anwendungen, die das Protokoll nutzen werden, um Netzwerkanwendungen und damit um Client/Server-Anwendungen handelt, unterscheiden sich auch die Anforderungen an das Protokoll in einigen Punkten. Prinzipiell muß das Protokoll über folgende Schnittstellen verfügen:

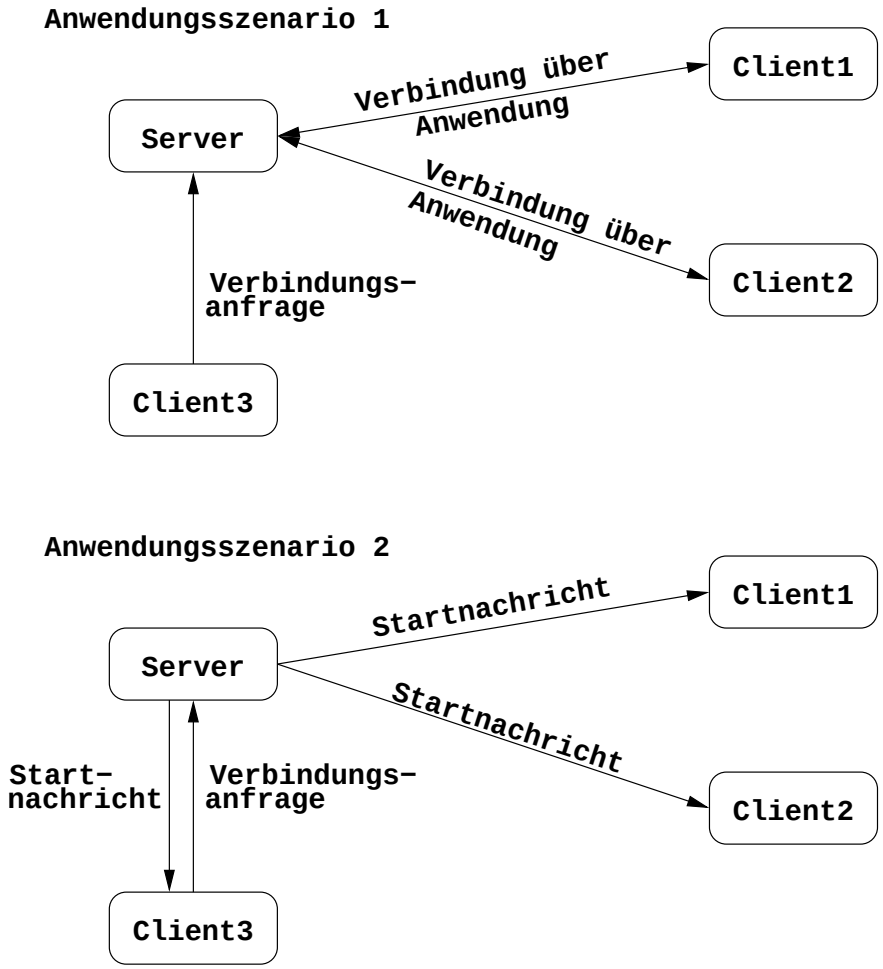

Abbildung 3.1: Szenarien für den Kommunikationsaufbau

* Senden einer Startbenachrichtigung
* Empfangen einer Startbenachrichtigung
* Aufbau einer Verbindung
* Schließen einer bestehenden Verbindung
* Senden von Dateien
* Empfangen von Dateien

Daten, die an die Anwendung weitergegeben werden, müssen im ASCII-Dateiformat auf dem jeweiligen Rechner abgelegt werden. Da es sich bei den Daten um relativ sensible Daten handelt, hat das Protokoll dafür zu sorgen, daß es zu keinen Datenverlusten bzw. -verfälschungen während des Datentransports kommt.

3.3 Spezifikation des Protokolls

Das zu entwickelnde Protokoll ist ein Teil der Verarbeitungsschicht des TCP/IP - Referenzmodells. Diese Einordnung hat mehrere Gründe. Der wichtigste Grund besteht darin, daß das Protokoll zur Umsetzung spezieller Anforderungen der Anwendung dient. Das heißt, es muß gewisse Kenntnisse über die Abläufe der darüberliegenden Anwendung haben. Diese Anforderung widerspricht aber der Spezifikation der darunterliegenden Schichten des Modells. Weiterhin ist es nicht nötig, daß das Protokoll Aufgaben übernimmt, wie z.B. die Sicherstellung einer Ende-zu-Ende Verbindung, die in der Verantwortlichkeit der Protokolle der Transportschicht liegen. Die Ansiedlung oberhalb der Transportschicht hat den Vorteil, daß auf die vorhandenen, ausgereiften Protokollimplementationen der TCP/IP-Protokollsuite zurückgegriffen werden kann. Dies vermindert die Möglichkeit zur Verursachung ungewollter Fehler und erhöht die Portabilität.

3.3.1 Kommunikationsablauf

Aus den in Abschnitt 3.2 formulierten Anforderungen lässt sich der Kommunikationsablauf ableiten, der aus sechs Hauptschritten besteht: Benachrichtigung bekannter Clients, Warten auf Verbindungswünsche von Clients, Aufbau der Client/Server-Verbindung und Übertragung der Anforderungsdatei, Empfang der auf der Clientseite ermittelten Daten, Senden der vom Server ermittelten Konfigurationsdaten an den Client und als letzten Schritt der Abbau der Verbindung.

Der genaue Kommunikationsablauf gliedert sich in die nachfolgend aufgeführten Teilschritte:

Schritt(0): Der Server sendet an ihm bekannte Clients die Nachricht, daß er jetzt erreichbar ist. Dieser Schritt ist nicht zwingend erforderlich.

Schritt(1): Der Client stellt eine Verbindungsanfrage an den Server, um die Kommunikation zu eröffnen.

Schritt(2): Dieser Schritt unterteilt sich in vier Teilschritte, die bis zum Ende dieses Schrittes immer wiederholt werden.

- Schritt(2a): Auslesen der vom Client benötigten Informationen aus einer Datei.
- Schritt(2b): Übertragen von jeweils einer Zeile an den Client.

Schritt(2c): Der Client speichert die empfangenen Daten in einer Datei zeilenweise zwischen.

Schritt(2d): Der Client sendet eine positive oder negative Quittung an den Server.

Schritt(3): Nachdem der Client die benötigten Informationen in einer Datei abgelegt hat, erfolgt die Übertragung dieser Datei an den Server. Diese Übertragung erfolgt wie im Schritt (2) in vier Teilschritten.

Schritt(3a): Auslesen einer Zeile aus der Datei mit den benötigten Informationen.

Schritt(3b): Übertragung einer Zeile aus der Datei an den Server.

Schritt(3c): Der Server schreibt die empfangenen Daten in eine Datei.

Schritt(3d): Der Server sendet eine positive oder negative Quittung an den Client.

Schritt(4): Nachdem der Server anhand der empfangenen Konfigurationsdaten die günstigsten Verbindungsparameter errechnet hat, werden diese in einer Datei zwischengespeichert. Diese Datei wird in jeweils vier Teilschritten an den Client übertragen. Das Prinzip folgt dem im Schritt (2) verwendeten.

Schritt(5): Nach der Übertragung aller Daten wird die Verbindung in zwei Schritten definiert beendet.

Schritt(5a): Der Client sendet die Benachrichtigung zur Beendigung der Kommunikation.

Schritt(5b): Der Server sendet eine Quittung über den Erhalt der Clientnachricht.

Nach der Beendigung von Schritt (5) ist die Verbindung zwischen Client und Server geschlossen. Wurden alle Schritte erfolgreich durchlaufen, kann die Anwendung auf der Clientseite mit den übertragenen Parametern gestartet werden.

Die Abbildung 3.2 dient der Veranschaulichung des Kommunikationsablaufs zwischen Client und Server.

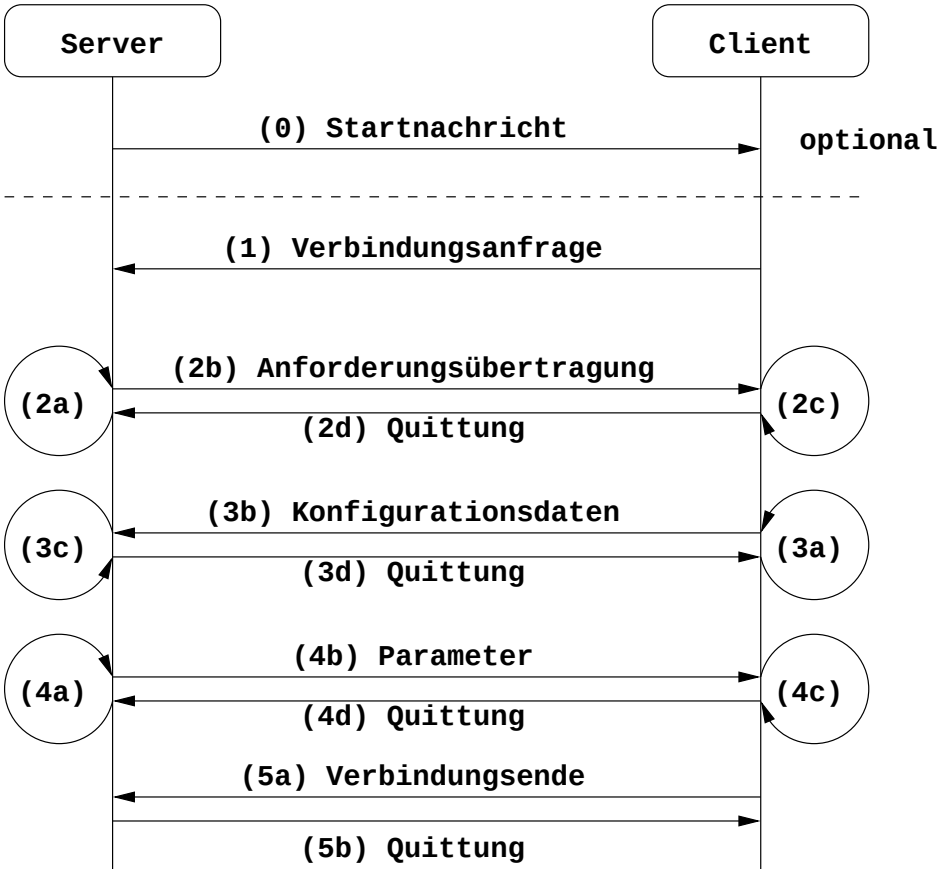

Abbildung 3.2: Protokollablauf zwischen Client und Server

3.3.2 Schnittstellen

In Abschnitt 3.2 wurden die Dienste und allgemeinen Schnittstellen formuliert, die den Anwendungen zur Verfügung zu stellen sind. An dieser Stelle werden die Schnittstellen mit den von ihnen geforderten Parametern festgelegt. Zusätzlich werden zu jeder Schnittstelle die im Hintergrund ablaufenden Vorgänge erläutert.

* **initServerConn(Port, Adresse)**: Dient zur Grundinitialisierung des Servers. Nach Beendigung der Funktion ist der Socket an die lokalen Adressinformationen gebunden. Der Parameter Port wird beim Binden des Sockets an die Adressinformationen benötigt. Der Parameter Adresse ist eine Datenstruktur der Socket-API. Sie wird der Schnittstelle uninitialisiert übergeben. Nach dem Aufruf der Schnittstelle ist die Adress-Struktur initialisiert. Diese Struktur wird zur Nutzung der Schnittstelle *getServerConn()* benötigt. Ist der Aufruf der Schnittstelle erfolgreich gewesen, so liefert sie einen gültigen Socketdeskriptor zurück. Dieser Socket wird benötigt um eingehende Verbindungswünsche entgegen zu nehmen.

* **initClientConn(Zielhost, Zielport):** Dient zur Initialisierung des Sockets auf Clientseite und zur Verbindung mit dem mittels Zielhost/Zielport spezifizierten Server. Nach Aufruf der Funktion existiert im Erfolgsfall eine aufgebaute Verbindung zwischen Client und Server. Der Parameter *Zielhost* dient zur Angabe des Rechners, auf dem der Server befindlich ist. *Zielport* dient zur Angabe der Portnummer, die dem Server auf dem Zielrechner zugeordnet ist. Ist der Aufruf der Schnittstelle erfolgreich, so wird ein gültiger Socketdeskriptor zurückgegeben.

* **getServerConn(Socket, Adresse):** Serverseitige Funktion, die auf Verbindungswünsche von Clients wartet. Bei erfolgreicher Rückkehr aus der Funktion ist eine Punkt-zu-Punkt Verbindung zwischen einem Client und dem Server aufgebaut und es wird ein gültiger Socketdeskriptor für die Clientverbindung zurückgegeben. Als ersten Parameter ist der Schnittstelle der von *initServerConn()* gelieferte Socketdeskriptor zu übergeben. Der zweite Parameter ist die von *initServerConn()* initialisierte Adress-Struktur.

* **transmitFile(Socket, Datei):** Dient zur zeilenweise Versendung der angegebenen Datei über den ebenfalls anzugebenden Socket. Der Parameter Socket ist im Client der von *initClientConn()* und im Server der von *getServerConn()* gelieferte Socketdeskriptor.

* **receiveFile(Socket, Datei):** Speichert die über den angegebenen Socket empfangenen Daten in der Datei. Der Parameter Socket ist wie in der Schnittstelle *transmitFile()* entweder der Deskriptor der im Client von *initClientConn()* oder im Server von *getServerConn()* gelieferte Deskriptor.

* **waitClientEnd(Socket):** Serverschnittstelle, die zum warten auf das Eintreffen der Ende-Nachricht dient. Als Parameter erfordert diese Schnittstelle den Socketdeskriptor, der zur Clientkommunikation genutzt wird.

* **closeClientConn(Socket):** Sendet eine Nachricht an den Server um die bestehende Kommunikation zu beenden. Als Parameter ist ein gültiger Socketdeskriptor zu übergeben.

* **serverShutdown():** Serverschnittstelle, die den Server veranlasst, keine weiteren Verbindungswünsche von Clients anzunehmen. Schon bestehende Clientverbindungen werden nicht unterbrochen. Die Schnittstelle benötigt keine Parameter.

* **informClients(Datei):** Sendet eine Start-Nachricht via UDP an die in der als Parameter angegebenen Datei aufgeführten Clients.

* **receiveStartMsg(IP-Adresse):** Wartet auf der Clientseite auf das Eintreffen einer Start-Nachricht von einem Server, um bei Bedarf mit einer Anfrage starten zu können. Der Parameter IP-Adresse dient zur Rückgabe der IP-Adresse des Servers.

Die Anwendung dieser Schnittstellen unterscheidet sich je nach Aufgabenbereich der Kommunikationspartner. Die Serverseite hat bei der Initiierung der Verbindung einen etwas höheren Aufwand zu betreiben. Der Ablauf der Kommunikation zwischen einem Server und einem Client ist in Abbildung 3.3 schematisch dargestellt und folgt den an das Protokoll gestellten Anforderungen. Aus dieser Abbildung ist auch ersichtlich, welche Schnittstellen zur Verwendung in einem Server bzw. Client vorgesehen sind und in welcher Reihenfolge die Aufrufe der Schnittstellen zu erfolgen haben. Die Schnittstelle *serverShutdown()* wurde nicht mit in die Abbildung aufgenommen, da sie in diesem Kontext nicht sinnvoll darstellbar ist.

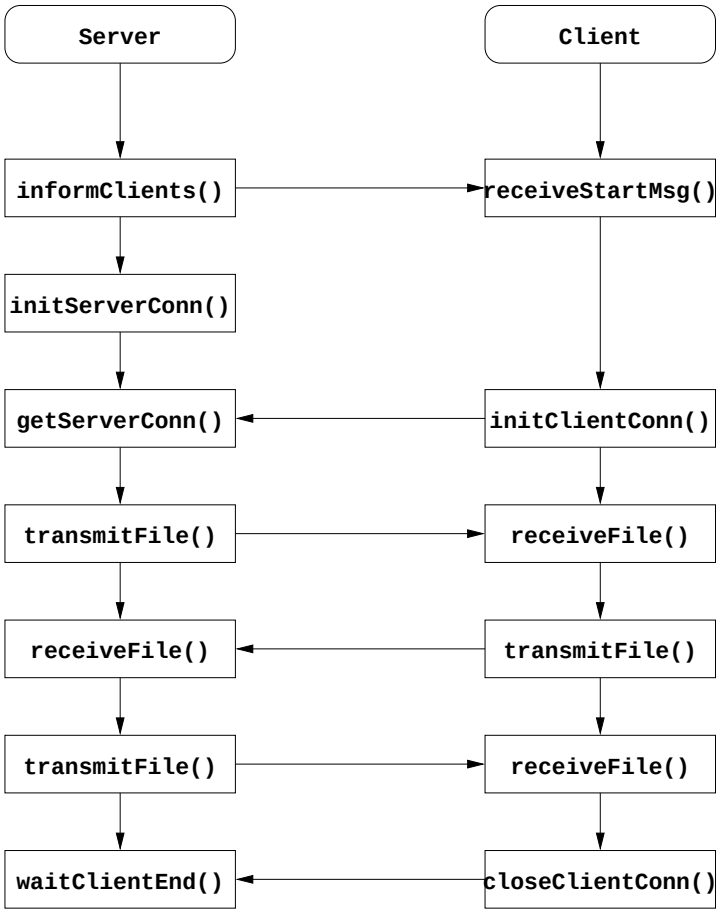

Abbildung 3.3: Verwendung der Schnittstellen durch Client bzw. Server

3.3.3 Auswahl der Protokolle der Transportschicht

Wie im Abschnitt 2.1 dargelegt, stellt die TCP/IP-Protokollsuite auf Ebene der Transportschicht zwei verschiedene Protokolle zur Datenübertragung bereit. Diese Protokolle unterscheiden sich in den von ihnen zur Verfügung gestellten Diensten. Bei der Auswahl eines Protokolls müssen daher die Vor- und Nachteile des jeweiligen Protokolls im Zusammenhang mit den Anforderungen des zu erstellenden Protokolls betrachtet werden. Aufgrund der zu übertragenden Daten lässt sich das Protokoll in zwei Abschnitte unterteilen. Der erste Teil ist die Übertragung der Start-Nachricht durch den Server an die Clients. Der zweite Teil ist verantwortlich für die Übertragung der Konfigurationsdaten zwischen Client und Server und trägt damit die Hauptlast der Kommunikation. Beide Teile stellen völlig unterschiedliche Anforderungen an das zu verwendende Protokoll. Im folgenden werden die Gründe zur Wahl des Protokolls der Transportschicht für die Realisierung des jeweiligen Protokollabschnittes erläutert.

Senden der Start-Nachricht

Das Senden sowie das Empfangen der Start-Nachricht ist ein Teil des Protokolls, der keine hohen Ansprüche an die Fehlerkorrektur oder Flusskontrolle stellt, da es sich lediglich um eine einzelne Nachricht handelt. Für die Verwendung des verbindungslosen Modus sprechen zwei Argumente. Erstens kann nicht vorausgesetzt werden, daß alle zu benachrichtigenden Clients wirklich aktiv sind. Somit ist nicht gesichert, daß eine Verbindung zu ihnen aufgebaut und die Nachricht übermittelt werden kann. Außerdem bedeutet der explizite Verbindungsaufbau zu jedem Client einen erhöhten Programmier- und Kommunikationsaufwand, der durch die Größe der zu übertragenden Nachricht nicht gerechtfertigt ist.

Aus den oben genannten Gründen und den in Abschnitt 2.1 aufgeführten Merkmalen der in der TCP/IP-Protokollsuite zur Verfügung stehenden Protokolle der Transportschicht wird dieser Teil des Protokolls unter Verwendung von UDP realisiert.

Austausch der Konfigurationsdaten

Das Senden und Empfangen der Konfigurationsdaten stellt wesentlich höhere Anforderungen an das zu verwendende Protokoll. Das zu wählende Protokoll muss absichern, daß die Daten fehlerfrei und in der richtigen Reihenfolge bei dem Empfänger ankommen. Außerdem muss es sicherstellen, das es nicht zu einer Überlastung und damit zu Datenverlusten beim Empfänger kommen kann. Der durch das Protokoll verursachte Overhead an Kontrolldaten bei der Übertragung zur Sicherung der geforderten Eigenschaften kann dabei in Kauf genommen werden, da die zu übertragenden Daten-

mengen relativ klein sein werden und damit keine nennenswerten Performanzeinbrüche zu erwarten sind.

Aus diesen Gründen und den in Abschnitt 2.1 aufgeführten Merkmalen von TCP und UDP fällt die Wahl für die Realisierung dieses Teils des Protokolls auf TCP.

3.3.4 Fehlersicherung

Für die Übertragung und den Empfang der Start-Nachricht ist keine explizite Fehlersicherung vorgesehen, da wie schon im Abschnitt 3.3.3 dargelegt, dieser Teil des Protokolls lediglich aus einer Nachricht besteht.

Der zweite Teil des Protokolls, die Versendung der Konfigurationsdaten, wird durch zwei Fehlersicherungsmechanismen abgesichert. Der erste Teil der Fehlersicherung obliegt TCP, das wie im Abschnitt 3.3.3 ausgeführt, für diesen Teil genutzt wird. TCP nimmt eine Fehlersicherung auf Bit- bzw. Byteebene vor. Zusätzlich zu der vorhandenen Absicherung wird auf der darüberliegenden Ebene der Versand der Daten abgesichert. Dazu wird vom Empfänger die Länge der Daten mit der im Sender ermittelten Länge der Daten verglichen. Um das zu ermöglichen wird vom Sender neben Daten auch die Länge übermittelt. Bei Übereinstimmung beider Längen sendet der Empfänger eine positive Antwort an den Absender, anderenfalls wird eine negative Antwort gesendet. Erhält der Sender letztere, so sendet er die gleichen Daten erneut.

Der Absicherung der Systemfunktionen, die zur Kommunikation über die Sockets genutzt werden, kommt eine besondere Bedeutung zu. Wird die Fehlersicherung an dieser Stelle vernachlässigt, kann es passieren, daß die gesamte Anwendung oder ein Teil davon "hängenbleibt", d.h. die aufgerufene Funktion kehrt nicht zurück und blockiert damit die weitere Programmabarbeitung. Zur Umgehung dieses Problems existieren mehrere Methoden. Die einfachste Methode ist, die eventuell blockierenden Systemrufe durch ein Timeout abzusichern. Dazu wird vor der Ausführung des Systemrufes ein Timer gesetzt, der im Fall des Blockierens der Systemfunktion ausgelöst wird und die Funktion unterbricht. Diese Methode ist unter Linux aber nicht praktikabel, da z.B. das vom Server ausgeführte *accept()* sofort wieder vom System neu aufgerufen wird. Eine andere Möglichkeit besteht darin, mittels der Funktion *fcntl()* den Deskriptor in den nichtblockierenden Modus zu versetzen. Nachteil dieser Methode ist es, daß alle Abfragen, z.B. die des *read()*-Aufrufes, im Polling-Modus erfolgen müssen. Dies kann im ungünstigsten Fall zu einer nahezu 100%-igen Systemauslastung führen und würde zur Vermeidung dessen einen erhöhten Programmieraufwand nach sich ziehen. Der beste Kompromiss ist die Verwendung des zum E/A-Multiplexing vorgesehenen Mechanismus unter Nutzung der Funktion *select()*, obwohl hier nur ein Deskriptor

verwendet wird. Die Funktion wartet auf das Eintreffen eines E/A-Ereignisses in einer der drei Deskriptormengen, die der Funktion übergeben werden können, oder bis zum Ablauf eines Timeouts. Jede der drei Mengen steht für ein bestimmtes Ereignis. Dies kann ein Lese-, Schreib- oder Fehlerereignis sein. Diese Methode bietet einen guten Kompromiss zwischen dem blockierenden Modus und dem Polling-Modus unter der Voraussetzung, daß der Timeout nicht zu kurz gewählt wird.

3.3.5 Datenformate, Strukturen

Der Zweck des Protokolls ist der Austausch von Konfigurationsdaten. Daher wird vorausgesetzt, daß die zur Übermittlung bestimmten Daten im ASCII-Format vorliegen. Aus diesem Grund werden auch alle Daten die zwischengespeichert werden müssen in ASCII-Dateien abgelegt. Dies ermöglicht die komfortable Weiterverarbeitung der Daten durch andere Programme.

Für die Übertragung der gewünschten Daten ist eine Struktur vorgesehen. Dies bietet die Möglichkeit, zusammen mit den eigentlichen Daten, Kontrollinformationen zu übermitteln. Zu diesem Zweck besteht die verwendete Struktur aus drei Komponenten. Die größte der drei Komponenten ist für die eigentlichen Daten vorgesehen. Die zweite Komponente ist für die zu übertragende Längeninformation vorgesehen. Die dritte Komponente wird zur Übermittlung der Statusinformationen genutzt, wie z.B. “Ende der Datei” oder “Fehler bei Übertragung”.

Die für die Benachrichtigung der Clients mittels der Start-Nachricht vorgesehene Datei muss ebenfalls im ASCII-Format vorliegen. Das genaue Format der Einträge ist dem Abschnitt 4.2 zu entnehmen. Die Start-Nachricht besteht aus einer Zeichenkette, die zwei Informationen für den Client enthält. Der erste Teil der Zeichenkette enthält die eigentliche Start-Nachricht. Im zweiten Teil ist die IP-Adresse des Servers enthalten, der diese Nachricht gesendet hat.

Kapitel 4

Protokollimplementation

Nach dem im Kapitel 3 die strukturellen Anforderungen an das zu entwickelnde Protokoll definiert wurden, beschäftigt sich dieses Kapitel mit der konkreten Implementierung des Protokolls.

In einer Beispielanwendung wird die Verwendung der Schnittstellen des Protokolls demonstriert. Dabei kommt das Ressourcenanalyseprogramm *resinfo* zum Einsatz. Der Server überträgt die Konfigurationsdatei für *resinfo* an den Client. Auf dem Clientrechner wird dann *resinfo* unter Verwendung der übertragenen Konfigurationsdatei ausgeführt. Die dadurch ermittelten Daten werden in einer Datei zwischengespeichert. Diese Datei wird anscließend an den Server übermittelt.

4.1 Implementierungsaspekte

Aus den im Abschnitt 2.3 genannten Gründen wird zur Implementierung des Protokolls die in LINUX in der Standard-C Bibliothek *glibc* integrierte Socket-API verwendet. Dies ermöglicht den Verzicht auf Zusatzbibliotheken und erhöht damit die Portabilität. Die Auswahl der Socket-API war auch einer der Gründe für den Einsatz der Programmiersprache C zur Implementierung des Protokolls. Die Entscheidung zum Einsatz der Programmiersprache C wurde außerdem durch die relativ geringe Komplexität des Protokolls bekräftigt.

Die im Abschnitt 3.3.3 getroffene Auswahl von TCP zur Übertragung der Konfigurationsdaten hat zur Folge, daß der Server eine komplette Ende-zu-Ende Verbindung zu jedem anfragenden Client aufbaut und für die Dauer der Kommunikation mit diesem keine weiteren Verbindungswünsche anderer Clients entgegen nehmen kann. Das heißt, daß mehrere gleichzeitige Anfragen von Clients nur sequentiell nacheinander abgearbeitet werden können. Zur Erhöhung der Leistungsfähigkeit des Servers wird deshalb nach jeder Clientanfrage mittels *fork()* ein eigener Prozeß abgespaltet. In diesem erfolgt die

weitere Kommunikation mit dem Client. Der Hauptprozeß wartet zur gleichen Zeit auf weitere Verbindungsanfragen.

4.1.1 Konzept

Die Protokollimplementation wird in Form einer Header-Datei mit statischer Bibliothek zur Verfügung gestellt. Somit ist die Nutzung der angebotenen Dienste dieser Implementation mit geringem Aufwand möglich. Es ist lediglich notwendig die Header-Datei *protokoll.h* in die Quelltexte mit *include* einzubinden, in denen die Schnittstellen verwendet werden. Zusätzlich ist es erforderlich zum Übersetzungszeitpunkt die Bibliothek *libprotokoll.a* anzugeben, andernfalls kommt es zu Fehlern während des Übersetzungsvorganges, sogenannten Linker-Fehlern.

4.1.2 Fehlerformat und -ausgaben

Zur Signalisierung von auftretenden Fehlern bei der Verwendung der Schnittstellen wurde ein einheitlicher Mechanismus festgelegt. Jeder Fehler wird durch einen negativen Rückgabewert angezeigt. Zur genaueren Lokalisierung der Fehlerursache wird zusätzlich ein Eintrag in eine Logdatei geschrieben. Die direkte Ausgabe von Fehlermeldungen auf eine Konsole wurde vermieden, da je nach Verwendung des Protokolls nicht vorausgesetzt werden kann, daß eine Konsole zu Verfügung steht. Dies ist zum Beispiel bei der Verwendung des automatischen Starts des Clients durch *inetd* der Fall. Der Internetsuperserver *inetd* startet den Client und übergibt diesem den Socket, indem er den Socket auf *stdin* mapped. Deshalb ist es dem Client in diesem Fall nicht möglich, Meldungen auf eine Konsole auszugeben.

Zur zentralen Protokollierung von Status- und Fehlermeldungen des Kernels und von Anwendungen steht deshalb in allen UNIX-Derivaten der *syslogd*-Dämon zur Verfügung.

Nutzung des syslog-Mechanismus

Zur Verwendung des *syslog*-Mechanismus stehen in der *glibc*-Bibliothek drei Funktionen zur Verfügung, die durch Einbindung der Header-Datei *syslog.h* nutzbar sind:

* `void openlog(char *kennung, int option, int facility);`
* `void syslog(int priorität, char *format, ...);`
* `void closelog(void);`

Die Funktion *openlog()* öffnet den Log-Mechanismus und bietet die Möglichkeit als ersten Parameter eine Kennung anzugeben, die bei jedem Logeintrag an den Anfang der Meldung gesetzt wird. Der zweite Parameter bietet die Möglichkeit verschiedene Optionen anzugeben. Eine mögliche Option ist zum Beispiel *LOG_PID*. Diese Option veranlasst, daß zu jeder Meldung die Prozess-ID mitprotokolliert wird. Der Parameter *facility* erlaubt die Auswahl verschiedener vordefinierter Quellen. Dies dient zur genaueren Unterscheidung der Quelle der Meldungen in den Logdateien und der von *syslogd* auszuführenden Aktionen. Standardmäßig wird die Quelle *LOG_USER* verwendet. Die Funktion *syslog()* bietet die Möglichkeit Einträge an den *syslog*d-Dämon zu senden. Der erste Parameter gibt dabei die Priorität der Meldung an. Für die niedrigste Priorität, die für Debug-Meldungen vorgesehen ist, wird z.B. die Konstante *LOG_DEBUG* angegeben. Insgesamt stehen acht Prioritätsstufen zur Verfügung. Der zweite und alle weiteren Parameter beziehen sich auf die eigentliche Meldung, die in die Logdatei geschrieben wird. Die Funktion *closelog()* schließt den Deskriptor, der zur Kommunikation mit dem *syslogd*-Dämon verwendet wird. Der Aufruf von *openlog()* und *closelog()* ist optional. Für weitergehende Informationen zur Verwendung der Programmierschnittstellen und ihrer Parameter sei auf die Man-Pages von *syslog* und auf [HER99] verwiesen.

Die Protokollimplementation verwendet als Kennung den Bezeichner "NETConfig". Zusätzlich wird durch Angabe der Option *LOG_PID* zu jeder Meldung die Prozess-ID mitprotokolliert. Als Quelle wird *LOG_USER* verwendet. Alle Nachrichten der Protokollimplementation werden mit der Priorität *LOG_ERR*, d.h. der Priorität von Fehlermeldungen protokolliert.

Konfiguration des syslog-Dämons

Die von *syslogd* erzeugten Logdateien befinden sich überlicherweise im Verzeichnis */var/log* oder dem Verzeichnis */var/adm*. Diese Angabe ist jedoch nicht verbindlich und kann sich je nach verwendeter LINUX-Distribution unterscheiden. Der genaue Name der Logdatei, in der auftretende Fehler protokolliert werden, hängt von der Konfiguration des *syslogd*-Dämons ab. Die Konfiguration des Dämons ist in der Datei */etc/syslog.conf* zu finden. Einträge in der Datei */etc/syslog.conf* haben folgendes vereinfachtes Format:

```
<Quelle>.<Priorität>   <Logdatei>
```

Ein Eintrag in die Konfigurationsdatei, der zum Schreiben der Fehlermeldungen des Protokolls in die Datei */var/adm/usererr.log* dient, hat demnach folgende Form:

```
user.error   /var/adm/usererr.log
```

Für weitere Informationen zu *syslogd* und seiner Konfigurationsmöglichkeiten sei auf die Man-Pages von *syslogd* und *syslog.conf* verwiesen.

4.2 Schnittstellen

In diesem Abschnitt werden die von der Protokollimplementation zur Verfügung gestellten Schnittstellen und die von ihnen geforderten Parameter näher erläutert. Die Prototypen der Funktionen sind der Datei *protokoll.h* entnommen. Die vollständige Datei befindet sich im Anhang A. Die vereinbarten Prototypen orientieren sich an den in Abschnitt 3.3.2 geforderten Schnittstellen.

```
int informClients(char *filename);
```

Die Funktion *informClients()* ist zur Nutzung durch den Server vorgesehen und dient der Versendung der Start-Nachrichten. Die Funktion erwartet im Parameter *filename* den Namen der Datei inklusive Pfad, in der die Liste der bekannten Clients abgelegt ist.

Gültige Einträge in der Datei sind IP-Adressen oder Hostnamen. Pro Zeile darf nur ein Host aufgeführt sein. Als Kommentarzeichen ist die Raute '#' zulässig. Eine Datei mit gültigen Einträgen hat also die folgende Form:

```
# Bekannte Clients
192.168.1.2
goliath.imn.htwk-leipzig.de
homer
# Ende bekannte Clients
```

Bei Erfolg gibt *informClients()* den Wert 0 zurück, ansonsten den Wert -1.

```
int receiveStartMsg(char *ip);
```

receiveStartMsg() wird vom Client genutzt. Die Funktion wartet auf die Start-Nachricht vom Server. Sie erwartet einen initialisierten char-Zeiger als Parameter. In diesem Parameter wird bei erfolgreich empfangener Start-Nachricht die IP-Adresse des Servers zurückgegeben. Bei Erfolg ist der Rückgabewert der Funktion 0, andernfalls -1.

```
int initServerConn(int port, struct sockaddr_in *address);
```

Die Funktion *initServerConn()* wird vom Server genutzt um den Socket zu initialisieren. Sie erwartet als ersten Parameter den Port auf dem lokalen Rechner. Im Parameter *address* werden die Adressinformation zurückgegeben, die später von *getServerConn()* zur Weiterbearbeitung benötigt werden. Im Erfolgsfall gibt die Funktion den Socketdeskriptor zurück, ansonsten -1.

```
int initClientConn( char *dest, int port);
```

In der Funktion *initClientConn()* wird der Socket initialisiert und die Verbindung zum Server erstellt. Im Parameter *dest* erwartet die Funktion die Angabe des Zielrechners entweder als IP-Adresse oder Hostnamen. Der zweite Parameter ist die Portnummer unter der der Server erreichbar ist. Im Erfolgsfall gibt die Funktion den Socketdeskriptor zurück, ansonsten -1.

```
int getServerConn(int *sockfd, struct sockaddr_in *address);
```

Die Funktion *getServerConn()* wird auf der Serverseite eingesetzt. Sie ist dafür verantwortlich, bei eingehenden Clientanfragen mittels *fork()* einen neuen Prozess abzuspalten. Der neue Prozess übernimmt danach die weitere Kommunikation mit dem Client, während der Hauptprozess auf weitere Verbindungsanfragen von Clients wartet. Als ersten Parameter erwartet die Funktion den initialisierten Socketdeskriptor, der von *initServerConn()* geliefert wird. Der zweite Parameter ist die initialisierte Adress-Struktur, die ebenfalls von *initServerConn()* geliefert wird. Nach erfolgreicher Ausführung liefert die Funktion den Socketdeskriptor auf die eingerichtete Client/Server-Verbindung zurück, andernfalls -1.

```
int transmitFile(int connfd, char *filename);
```

Die Funktion *transmitFile()* kann von beiden Kommunikationspartnern genutzt werden und ist für die Übertragung einer Datei verantwortlich. Als ersten Parameter erwartet sie einen gültigen Socketdeskriptor. Als zweiten Parameter erwartet *transmitFile()* den Namen inklusive kompletten Pfad der Datei, die an den Kommunikationpartner zu übermitteln ist. Im Erfolgsfall liefert die Funktion 0, ansonsten -1.

```
int receiveFile(int *connfd, char *filename);
```

Die von beiden Seiten nutzbare Funktion *receiveFile()* ist für den Empfang einer Datei zuständig. Sie erwartet als ersten Parameter den Socketdeskriptor und als zweiten Parameter den Namen inklusive vollständigen Pfad der Datei, in der die Empfangsergebnisse abgespeichert werden. Bei einem erfolgreichen Durchlaufen der Funktion wird als Rückgabewert 0, sonst -1 zurückgegeben.

```
int waitClientEnd(int *sockfd);
```

Die für die Serverseite vorgesehene Funktion *waitClientEnd()* dient dem Empfang der Nachricht zum beenden der Kommunikation, dem Versand der Quittung und dem geordneten Schließen der Verbindung. Sie erwartet als einzigen Parameter einen gültigen Socketdeskriptor. Der Rückgabewert ist im Erfolgsfall 0, ansonsten -1.

```
int closeClientConn(int *sockfd);
```

Die Funktion *closeClientConn()* ist für die Verwendung durch die Clientseite vorgesehen. Sie dient der geordneten Beendigung der Kommunikation mit dem Server. Dazu versendet die Funktion eine Ende-Nachricht an den Server und wartet anschließend auf die Empfangsbestätigung. Wird diese Bestätigung empfangen, so wird die Verbindung auf der Clientseite geschlossen. Als Parameter erwartet die Funktion einen gültigen Socketdeskriptor. Im Erfolgsfall ist der Rückgabewert der Funktion 0, sonst -1.

```
int serverShutdown(void)
```

Die Funktion *serverShutdown()* bietet der Serverseite die Möglichkeit, weitere Verbindungsanfragen von Clients zu unterbinden. Diese Funktion benötigt keine Parameter und gibt bei erfolgreichen Durchlaufen 0, andernfalls -1 zurück.

```
int confirmShutdown(void)
```

Die Funktion *confirmShutdown()* ist zur Verwendung durch den Server vorgesehen. Es ist eine Servicefunktion, die zur Vermeidung von Zombies eingesetzt wird. Zombies sind beendete Prozesse, die aber immer noch in der Prozesstabelle stehen. Sie entstehen, wenn ein Elternprozeß nicht das Ende seiner Kindprozesse explizit abfragt oder selbst vor dem Ende aller Kindprozesse beendet wird. Die Aufgabe von *confirmShutdown()* ist es, den Vaterprozess solange warten zu lassen, bis alle Kindprozesse beendet wurden. Wenn keine Kindprozesse mehr aktiv sind, gibt die Funktion den Wert 0 zurück, ansonsten -1.

4.3 Schnittstellenimplementierung

Im Abschnitt 4.2 wurden die Protokoll-Schnittstellen in der Form von C-Funktionsprototypen sowie deren Parameter festgelegt. In diesem Abschnitt liegt das Hauptaugenmerk auf der Implementation dieser Schnittstellen. Zum besseren Verständnis der Vorgehensweise werden Auszüge aus den Quelltexten der Implementation näher beleuchtet. Außerdem werden wesentliche Funktionsaufrufe erläutert.

4.3.1 Verwendete Variablen und Strukturen

Im folgendem werden wichtige Variablen und Strukturen der Implementation erläutert.

* Die Struktur *buffer*:

```
struct buffer
{
  int len;
  int flag;
  char data[MAX_CHAR];
};
```

Die für die Übertragung der Daten und Steuerinformationen verwendete Struktur besteht aus drei Teilen. Die Komponente *len* dient zur Speicherung der Länge des Datenfeldes *data*. Die Komponente *flag* dient zur Übertragung verschiedener Steuerzeichen, die in der Aufzählungskonstante *message* definiert sind. In der Implementation werden zwei Strukturen dieses Types verwendet. Die Variable *sndbuf* dient dem Versand und die Variable *rcvbuf* dem Empfang von Nachrichten.

* Die Aufzählungskonstante *message*:

```
enum message { OK=1, FAIL, END_OF_FILE, END_OF_COMM };
```

Mit *OK* wird der erfolgreiche und mit *FAIL* der fehlerhafte Empfang von Daten signalisiert. *END_OF_FILE* kennzeichnet den letzten Datensatz. *END_OF_COMM* signalisiert die Bereitschaft zur Beendigung der Kommunikation.

* Die Variable *address*:

```
struct sockaddr_in address;
```

Diese Struktur dient der Speicherung der Adressfamilie, der Portnummer und der IP-Adresse und wird als Parameter für die Funktionen *accept()*, *connect()* und *bind()* benötigt.

* Die Variablen *sockfd* und *connfd*:

Diese beiden Variablen sind vom Typ *int*. Sie repräsentieren Socketdeskriptoren.

4.3.2 informClients()

Die Implementierung der Funktion informClients() besteht aus folgenden wesentlichen Abschnitten:

* Anlegen des Sockets und Initialisierung der Adress-Struktur:

```
if((sockfd = socket(PF_INET,SOCK_DGRAM,0)) < 0)
{   ... Fehlerbehandlung }
if(initAddressStructure(0, NULL, &client) < 0 )
{   ... Fehlerbehandlung }
```

* Binden der Adress-Struktur an den Socket:

```
if (bind(sockfd, (struct sockaddr *) &client, sizeof(client)) < 0)
{   ... Fehlerbehandlung }
```

* Erweiterung der Start-Nachricht um die IP-Adresse des Servers:

```
if( gethostname( name, MAX_CHAR) < 0)
if( (host = gethostbyname(name)) == NULL)

strcat(msg,inet_ntoa(( *(struct in_addr*)host->h_addr_list[0])));
```

* Zeilenweises Auslesen der Clientliste aus der angegebenen Datei und Versand der Start-Nachricht an diese Clients:

```
while(fgets(buffer, MAX_CHAR-1, fp) != NULL)
{
  ... zeilenweises Auslesen der Datei mit der Clientliste
  if(initAddressStructure(port, bp, &server) < 0 )
       continue;
  if (sendto(sockfd, msg, strlen(msg), 0,(struct sockaddr *) &server,
      sizeof(server)) < 0)
  { ... Fehler bei Versenden der Start-Nachricht an einen Client }
}
```

4.3.3 receiveStartMsg()

Die Funktion *receiveStartMsg()* besteht nur aus wenigen Aufrufen, da das Anlegen des Sockets und die komplette Socketinitialisierung von *inetd* übernommen wird. Durch *inetd* wird sichergestellt, daß die erwartete Nachricht über *stdin* gelesen werden kann.

* Deskriptor für die Standardeingabe zuweisen und in den nichtblockierenden Modus schalten:

```
sockfd = STDIN_FILENO;
if( setNonBlock(&sockfd) < 0)
   return(-1);
```

* Lesen der Nachricht über stdin und mit der Start-Nachricht vergleichen:

```
recvd = read(sockfd,&buf,MAX_CHAR-1);
if (recvd < 0)
{ ... Fehlerbehandlung  }
if(!strcmp(buf,"Bereit"))
{ ... Fehlerbehandlung  }
```

* Extraktion der IP-Adresse des Servers aus der Start-Nachricht und zur Rückgabe in die Zeichenkette *ip* kopieren:

```
strcpy(ip, bp);
```

4.3.4 initServerConn()

Die Funktion *intiServerConn()* gliedert sich in folgende Teilschritte:

* Anmeldung der Signalbehandlungsfunktion, um auf die von der Serverseite verwendeten Signale reagieren zu können:

```
if(signal(SIGCHLD,sig_handler)== SIG_ERR)
  syslog(LOG_ERR,"Signalanmeldung %s",strerror(errno));
if(signal(SIGUSR1,sig_handler)== SIG_ERR)
  syslog(LOG_ERR,"Signalanmeldung %s",strerror(errno));
```

Der Server verwendet zwei Signale. Das Signal *SIGCHLD* teilt dem Vaterprozess mit, daß sich ein Kindprozess beendet hat. Das Signal *SIGUSR1* wird verwendet um den Vaterprozess zu veranlassen, keine weiteren Verbindungswünsche entgegenzunehmen. Die Signalbehandlungsfunktion *sig_handler()* wird im Abschnitt 4.3.13 genauer erläutert.

* Anlegen des STREAM-Sockets:

```
if((sockfd = socket(PF_INET,SOCK_STREAM,0)) < 0)
{
   syslog(LOG_ERR,"Anlegen des Socket %s",strerror(errno));
   return(sockfd);
}
```

Der erste Parameter der Funktion *socket()* gibt an, daß der anzulegende Socket zur Kommunikation mittels IPv4 bestimmt ist. In Verbindung mit dem zweiten Parameter *SOCK_STREAM* ergibt sich damit die Verwendung von TCP.

* Binden des Sockets an die Adress-Struktur:

```
if(bind(sockfd,(struct sockaddr *)address,sizeof(struct
   sockaddr_in)))
{
   syslog(LOG_ERR,"bind-Fehler %s",strerror(errno));
   return(-1);
}
```

Die Initialisierung der Adress-Struktur geschieht vor dem Binden in der Hilfsfunktion *initAddressStructure()* und wird im Abschnitt 4.3.13erläutert .

* Einrichten der Verbindungswarteschlange:

```
if(listen(sockfd, 5))
{
   syslog(LOG_ERR,"listen-Fehler %s",strerror(errno));
   return(-1);
}
```

Der zweite Parameter von *listen()* gibt an, wie viele wartende Verbindungen akzeptiert werden. Für diesen Wert gibt es verschiedene Richtwerte in der Literatur. Aus Gründen der Portabilität wird jedoch in den meisten Fällen davon abgeraten einen größeren Wert als 5 zu wählen [HER99].

4.3.5 getServerConn()

Die Funktion *getServerConn()* ist, wie schon im Abschnitt 4.2 erwähnt, für die Verarbeitung von Clientanfragen verantwortlich. Der Aufbau der Funktion ist folgender Auflistung zu entnehmen:

* Einrichten der Strukturen zur Verwendung mit *select()* und Aufruf der *select()*-Funktion:

```
FD_ZERO(&readset);
FD_SET(*sockfd, &readset);
timeout.tv_sec = TIME_OUT;
timeout.tv_usec = 0;
rc = select(*sockfd + 1, &readset, NULL, NULL, &timeout);
```

Der *select()*-Aufruf bewirkt, daß bis zum Ablauf des Timers auf ein Ereignis auf dem Socket gewartet wird. Ist dies der Fall, ist rc größer 0 und es kann eine weitere Verarbeitung des Ereignisses stattfinden. Andernfalls muß eine Fehlerbehandlung erfolgen.

* Aufruf von *accept()*, um eine anstehende Verbindung entgegenzunehmen:

```
if( (connfd = accept(*sockfd, (struct sockaddr*)&address,
        &adrlength)) < 0 )
{ ... Fehlerbehandlung   }
```

* Parallelisierung:

```
pid = fork();
switch (pid)
{
    case -1: ... Fehlerbehandlung
             break;
    case 0: // Das ist der neu abgespaltene Prozess
             close(*sockfd);
             *sockfd = -1;
             break;
    default: // Das ist der Hauptprozess
             close(connfd);
             connfd = -1;
             break;
}
```

Das Warten auf die eingehenden Clientverbindungen und die Abspaltung der Kindprozesse erfolgt in einer Schleife. Die Bedingung der Schleife ist so formuliert, daß sie nur von den abgespalteten Kindprozessen und bei der durch serverShudown() gesetzten Ende-Bedingung auch vom Hauptprozess verlassen werden kann.

4.3.6 waitClientEnd()

Die Hauptbestandteile dieser Funktion sind im folgenden aufgelistet:

* Der erste Teil der Funktion ist für das Warten auf die Ende-Nachricht vom Client zuständig. Zur Absicherung der *read()*-Funktion wird wieder auf *select()* zurückgegriffen:

```
while(!ok)
{
  rc = select(*sockfd +1, &readset, NULL, NULL, &timeout);
  if(rc > 0)
  {
     rc = read(*sockfd, &rcvbuf, sizeof(struct buffer));
     if( rc > 0 )
     {
        if(rcvbuf.flag == END_OF_COMM)
           ok++;
     }
     else  {    ... Fehlerbehandlung und Prozessende   }
  }
  else {  ...Fehlerbehandlung und Prozessende    }
}
```

* Nach Erhalt der Ende-Nachricht vom Client wird ihm eine Quittung gesendet:

```
sndbuf.flag = OK;
sndbuf.data[0] = '\0';
sndbuf.len = 0;
rc = write(*sockfd, &sndbuf, sizeof(struct buffer));
```

4.3.7 serverShutdown()

Die Funktion *serverShutdown()* besteht nur aus einem Funktionsaufruf:

```
kill(getppid(),SIGUSR1);
```

Dieser Aufruf wird genutzt, um ein Signal an den Hauptprozess zu senden. Dieses wird in der Signalbehandlungsfunktion ausgewertet. Bei Erhalt des Signals wird die Abbruchbedingung für *getClientConn()* gesetzt.

4.3.8 confirmShutdown()

Zur Umsetzung der geforderten Funktionalität wertet die Funktion die globale Variable *child* aus. Solange der Wert von *child* größer 0 ist, wird -1 zurückgegeben. Ist *child* gleich 0 so liefert die Funktion ebenfalls 0.

4.3.9 initClientConn()

Wie im Abschnitt 4.2 festgelegt, ist *initClientConn()* für den Aufbau der Verbindung zu einem Server verantwortlich. Dafür sind folgende Schritte notwendig:

* Initialisierung der Adress-Struktur, anlegen des Sockets und umschalten in den nichtblockierenden Modus:

```
if(initAddressStructure(port, dest, &address) < 0 )
  { ... Fehlerbehandlung  }
if( (sockfd = socket(PF_INET, SOCK_STREAM, 0)) < 0 )
  { ... Fehlerbehandlung  }
if(setNonBlock(&sockfd) < 0)
  { ... Fehlerbehandlung  }
```

* Aufnahme der Verbindung zum Server:

```
while(connfailure < MAX_FAIL)
{
  if( connect(sockfd,(struct sockaddr *) &address, sizeof( address ) ))
  { ...Fehlerbehandlung und Socket schließen  }
}
```

Die Rückgaben der Funktion *connect()* werden unterschieden in kritische Fehler wie z.B. Netzwerk nicht erreichbar und Fehler, die eine Wiederholung des Aufrufes nach einer gewissen Wartezeit zulassen, wie z.B. momentane Überlastung des Servers. Im Fall eines kritischen Fehlers wird der Socket geschlossen und die Funktion mit negativem Rückgabewert verlassen. Im Fall eines nicht kritischen Fehlers wird bis zum Erreichen der Konstante *MAX_FAIL* ein erneuter Verbindungsaufbau zum Server versucht.

4.3.10 closeClientConn()

Für den Verbindungsabbau durchläuft der Client die nachstehend aufgeführten Schritte:

* Senden der Nachricht zum beiderseitigen Verbindungsabbau:

```
sndbuf.flag = END_OF_COMM;
rc = write(*sockfd, &sndbuf, sizeof(struct buffer));
if(rc < 0) { ...Fehlerbehandlung   }
```

* Warten auf den Empfang der Quittung vom Server:

```
while(!ok)
{
   rc = select(*sockfd + 1, &readset, NULL, NULL, &timeout);
```

```
   if(rc > 0)
   {
      if((rc = read(*sockfd, &rcvbuf, sizeof(struct buffer))) > 0)
      {
         if(rcvbuf.flag == OK)
         {
            close(*sockfd);
            *sockfd = -1;
            ok++;
         }
     }
     else { ... Fehlerbehandlung für den read-Aufruf}
   }
   else { ... Fehlerbehandlung für den select-Aufruf  }
}
```

4.3.11 transmitFile()

Der Versand einer Datei erfolgt in den nachfolgend aufgeführten Schritten.

* Öffnen der zu übertragenden Datei:

```
fp = fopen( filename,"r");
if(fp == NULL) {... Fehlerbehandlung }
```

* Auslesen einer Zeile der Datei, Versand und Warten auf die Quittung:

```
if(connerr == 0)
  if(!fgets(sndbuf.data,MAX_CHAR,fp))
  { ... Ende der Datei erreicht, Ende-Nachricht initialisieren  }
if(connerr == MAX_FAIL)  { ... Fehlerbehandlung   }
sndbuf.len=strlen(sndbuf.data);
if(write(connfd,&sndbuf,sizeof(sndbuf))<0) { ... Fehlerbehandlung  }
while(!ok)
{
  timeout.tv_sec = TIME_OUT_SEC;
  timeout.tv_usec = TIME_OUT_MSEC;
  FD_SET(connfd , &readset);
  if((rc = select(connfd +1, &readset, NULL, NULL, &timeout)) < 0)
  { ... Fehlerbehandlung für die select-Funktion  }
  else if(rc == 0)    { ... Timeoutbehandlung  }
  else if(rc > 0)
  {
     if(FD_ISSET(connfd, &readset))
     {
       rc = read(connfd,&rcvbuf,sizeof(struct buffer));
       if(rc <= 0) { ... Fehlerbehandlung für die read-Funktion  }
       else
```

```
        {
           ... Testen, ob positive oder negative Antwort vom
               Kommunikationspartner gesendet wurde.
        }
    }
}
if(connerr == MAX_FAIL)
{  ... Maximale Anzahl Verbindungsfehler erreicht.
      Fehlerbehandlung
}
```

Die aufgeführten Schritte werden in einer Schleife ausgeführt bis das Ende der Datei oder die maximale Anzahl erlaubter Verbindungsfehler erreicht wurde. Bei schwerwiegenden Verbindungsfehlern erfolgt ein sofortiger Abbruch der Schleifenabarbeitung. Die abgebildete Schleife ist für das Empfangen der Quittung zuständig.

4.3.12 receiveFile()

Für den Empfang einer Datei durchläuft der Client folgende Schritte:

* Öffnen der zum Zwischenspeichern der Daten angegebenen Datei:

```
fp = fopen( filename, "w");
if(fp == NULL) { ...Fehlerbehandlung }
```

* Lesen der Nachrichten und Zwischenspeicherung in der Datei mit anschließendem Versenden der Quittung:

```
l = select(*connfd +1, &readset, NULL, NULL, &timeout);
if( l < 0)  { ... Fehlerbehandlung für die select-Funktion  }
else if(l == 0) { ... Zeitgeber abgelaufen  }
else if(l > 0)
{
   if( (l = read(*connfd, &rcvbuf, sizeof(struct buffer)) ) > 0)
   {
      if(strlen(rcvbuf.data) == rcvbuf.len)
      {
         sndbuf.flag = OK;
         strcpy(sndbuf.data,rcvbuf.data);
         if(rcvbuf.flag != END_OF_FILE)
           if(fputs(rcvbuf.data,fp) <=0)
           { .. Zwischenspeicherung fehlgeschlagen }
      }
      else { ... Datenverluste in der Nachricht }
      if(write(*connfd, &sndbuf, sizeof(sndbuf))<=0)
      { ... Senden einer positiven oder negativen Antwort }
```

```
            if(rcvbuf.flag == END_OF_FILE)
            { ... Ende der Dateiübertragung erreicht }
        }
        else { ... Fehlerbehandlung für read-Funktion }
    }
```

Die aufgeführten Schritte werden in einer Schleife bis zum Empfang der Dateiende-Nachricht oder einem schwerwiegenden Fehler durchgeführt. Wurden bei dem Längenvergleich der Daten Verluste festgestellt, so wird eine negative Antwort zurück gesendet und die Nachricht nicht in die Datei gespeichert. Andernfalls wird eine positive Antwort gesendet, die empfangenen Daten werden in die Datei geschrieben.

4.3.13 Hilfsfunktionen

Es werden zwei Hilfsfunktionen (*setNonBlock()*, *initAddressStructure()*) eingesetzt, um ständig wieder auftretenden Aufgaben zu übernehmen sowie die Funktion *sig_ handler()* zur Behandlung der im Server genutzten Signale. Diese Funktionen werden an dieser Stelle etwas näher beleuchtet.

Die Funktion setNonBlock()

Die Funktion *setNonBlock()* dient dazu, einen ihr übergebenen Deskriptor in den nichtblockierenden Modus zu versetzen. Der Parameter ist ein Zeiger auf eine Integer-Variable. Um den Modus für den Deskriptor zu setzen sind folgende Schritte notwendig:

* Auslesen der aktuellen Deskriptorflags:

```
int flags;
if( (flags = fcntl(*fd,F_GETFL)) < 0)   { ...Fehlerbehandlung }
```

Zum Auslesen wird die Systemfunktion *fcntl()* genutzt. Diese Funktion benötigt als ersten Parameter den Deskriptor. Der zweite Parameter ist die Konstante *F_ GETFL,* welche angibt, daß die gesetzten Flags des Deskriptors ausgelesen werden sollen. Die Flags werden in die Variable *flags* geschrieben.

* Setzen des Flags für den nichtblockierenden Modus:

```
if(fcntl(*fd,F_SETFL, flags | O_NONBLOCK) < 0)   { ... Fehlerbehandlung }
```

Das Setzen von Flags für einen Deskriptor erfolgt ebenfalls mit der Funktion *fcntl()*, der als erster Parameter der Deskriptor übergeben wird. Als zweiter Parameter wird die Konstante *F_SETFL* angegeben. Für das Setzen ist die Angabe eines dritten Parameters nötig. Dieser beinhaltet in diesem Fall die vorher in die Variable *flags* ausgelesenen Flags, die mit der Konstante O_NONBLOCK bitweise ODER-Verknüpft werden.

Die Funktion initAddressStructure()

Die Funktion *initAddressStructure()* dient dazu, eine übergebene Adress-Struktur mit den ebenfalls übergebenen Parametern Portnummer und Zielrechner zu initialisieren. Der Zielrechner wird in einer Zeichenkette angegeben. Der Inhalt der Zeichenkette kann in zwei Formaten vorliegen: IP-Adresse im Standardformat oder ein Rechnername im Klartext. Für die Initialisierung sind folgende Schritte nötig:

* Setzen der Adressfamilie und des Ports:

```
struct in_addr inaddr;
struct hostent *host;
address->sin_family = AF_INET;
address->sin_port = htons(port);
```

* Setzen der IP-Adresse in Abhängigkeit vom Parameter *dest:*

```
if( dest == NULL)
  address->sin_addr.s_addr = htonl(INADDR_ANY);
else
{
  if(inet_aton(dest, &inaddr))
    host = gethostbyaddr((char *) &inaddr, sizeof(inaddr), AF_INET);
  else
    host = gethostbyname(dest);
  if( host == NULL )      { ...Fehlerbehandlung und Rückkehr  }
  memcpy( &address->sin_addr, host->h_addr_list[0],
  sizeof(address->sin_addr));
}
```

Ist der für die Angabe des Zielrechners genutzte Parameter *dest* gleich NULL, so wird als IP-Adresse die Konstante *INADDR_ANY* gesetzt. Diese Konstante steht für eine beliebige IP-Adresse. Diese Initialisierung wird benötigt, wenn auf Verbindungsanforderungen beliebiger Rechner reagiert werden muß, so wie es z.B. für einen Server typisch ist.

Ist der Parameter *dest* ungleich NULL, so wird zuerst mit der Funktion *inet_ aton()* versucht, die Adresse des Zielrechners von der IP-Adressform in ein binäres Format umzuwandeln und in der Struktur *inaddr* zu speichern. Ist dieser Schritt erfolgreich, so wird im nächsten Schritt die Struktur host mit dem Aufruf der Funktion *gethostbyaddr()* initialisiert.

Schlug im ersten Schritt der Aufruf der Funktion *inet_ aton()* fehl, so wird versucht die *host*-Struktur unter Nutzung der Funktion *gethostbyname()* zu initialisieren.

Die Funktion sig_handler()

Die Funktion *sig_ handler()* wird für die Behandlung der Signale *SIGCHLD* und *SIGUSR1* genutzt. Beim Eintreffen des Signals *SIGCHLD* wird der Endestatus des Kindprozesses abgefragt und die von *confirmShutdown()* ausgewertete Variable *child* dekrementiert. Trifft das, von der Funktion *serverShutdown()* gesendete, Signal *SIGUSR1* ein, so wird die von *getServerConn()* ausgewertete Variable *final* gesetzt und somit die Verbindungsaufnahme zu neuen Clients verhindert. Die Funktion hat folgenden Aufbau:

```
if(signr == SIGCHLD)
{
  while((pid=wait(&status)) > 0)
   child--;
}
if(signr == SIGUSR1)
   final = 1;
```

4.4 Beispielimplementierung

An dieser Stelle wird auf die Beispielimplementation näher eingegangen. Wie bereits in der Einleitung des Kapitels 4 erwähnt, kommt zur Demonstration zusätzlich das Programm *resinfo* zum Einsatz [BH02]. Es dient zur Gewinnung von Informationen zur Hard- und Softwareausstattung eines Rechners. Der Informationsumfang kann mittels einer Konfigurationsdatei, die *resinfo* als Argument übergeben wird, beeinflusst werden.

Zur Demonstration des Einsatzes der Protokollimplementation ist es nötig eine Client/Server-Anwendung zu entwickeln. Von ihr wird folgende Funktionalität erwartet:

* Optionale Versendung einer Start-Nachricht vom Server an bekannte Clients.
* Versandt der Konfigurationsdatei für *resinfo* vom Server an anfragende Clients.

* Ausführung des Programmes *resinfo* auf der Clientseite mit der vom Server empfangenen Konfigurationsdatei als Argument.

* Übertragung der auf der Clientseite gewonnenen Ergebnisdatei an den Server und Ende der Kommunikation

Die weitergehende Auswertung der empfangenen Daten auf der Serverseite ist momentan noch nicht möglich, deshalb wird auf den letzten Protokollschritt verzichtet. Dieser sieht die Sendung einer Parameterdatei an den Client vor.

4.4.1 Server

An dieser Stelle wird auf die Implementation der Serverseite der Beispielanwendung näher eingegangen. Zur Verdeutlichung der Verwendung der Schnittstellen der Protokollimplementation werden Auszüge des Quelltextes eingefügt. Bevor auf die Implementation eingegangen wird, erfolgt die Klärung der verfügbaren Kommandozeilenparameter.

Parameter

Der Server verfügt über drei optionale Kommandozeilenparameter. Die Parameter haben folgende Bedeutung:

* *-s* : Mit dem Parameter *-s* wird der Server veranlasst, die Start-Nachricht an Clients zu versenden. Dazu liest der Server die Datei *clients.lst* im aktuellen Arbeitsverzeichnis aus.

* *-c <Dateiname>* : Durch die Angabe des Parameter *-c* mit nachgestelltem Dateinamen werden die Clients aus dieser statt aus der standardmäßig verwendeten Datei *clients.lst* ausgelesen.

* -h : Der Parameter -h gibt die Hilfe zur Verwendung der Parameter aus.

Implementation

Die nachfolgenden Ausschnitte aus dem Quelltext des Servers dienen zur Illustration der Verwendung der Schnittstellen des Protokolls:

* Verwendete Header-Dateien und Definitionen:

```
#include <ctype.h>
#include "protokoll.h"
#define PORT_NUMBER 55300
#define CLIENTLIST "client.lst"
```

* Variablenvereinbarungen:

```
int sockfd, connfd, startmsg=0, option;
struct sockaddr_in address;
char *filename;
```

* Senden der Start-Nachricht:

```
if(startmsg)
    if(informClients(filename, PORT_NUMBER) < 0)
      { ...Fehlerbehandlung }
```

* Initialisierung der Verbindung:

```
sockfd = initServerConn(PORT_NUMBER, &address);
  if(sockfd < 0)
    exit(1);
```

* Warten auf eingehende Clientanfragen:

```
connfd = getServerConn( &sockfd, &address );
```

* Bei eingegangener Clientanfrage Versenden der Konfigurationsdatei, Empfang der Ergebnisdatei mit anschließendem Empfang der Ende-Nachricht des Clients:

```
if(connfd >= 0)
{
  if(transmitFile(connfd, "resinfo.conf") < 0)
    { ...Fehlerbehandlung }
  if(receiveFile(&connfd, "resinfo.erg") < 0)
    { ... Fehlerbehandlung }
  waitClientEnd(&connfd);
}
```

4.4.2 Client

Parameter

Der Client verfügt über vier Kommandozeilenparameter. Die Parameter haben folgende Bedeutung:

* *-s <Rechnername>* : Der Parameter -s mit nachgestelltem Rechnernamen oder IP-Adresse steht für die direkte Verbindungsaufnahme zu einem Server zur Verfügung.

* *-d* : Der Parameter *-d* muss angegeben werden, wenn der Start des Clients über *inetd* vorgesehen ist. Die nötigen Konfigurationen zur Arbeit in diesem Modus sind im Abschnitt 4.4.3 aufgeführt.

* *-v <Verzeichnis>* : Mit dem Parameter *-v* kann das Arbeitsverzeichnis gesetzt werden. Auf diesem Verzeichnis müssen Lese- und Schreibrechte vorhanden sein, da in diesem Verzeichnis alle Dateien abgelegt werden.

* *-h* : Der Parameter -h dient zur Anzeige einer Hilfe zu den verfügbaren Parametern.

Die Parameter *-s* und *-d* dürfen nicht gemeinsam angegeben werden.

Implementation

Nachfolgend sind zur Veranschaulichung der Schnittstellenverwendung Auszüge aus dem Quelltext des Clients aufgeführt:

* Verwendete Header-Dateien und Definitionen:

```
#include <ctype.h>
#include "protokoll.h"
#define PORT_NUMBER 55300
```

* Variablenvereinbarungen:

```
int sockfd, daemon=0, host=0;
char *ip, *dir;
```

* Empfang der Start-Nachricht, falls von *inetd* gestartet:

```
if(daemon)
{
   if(receiveStartMsg(ip) < 0)
   { ... Fehlerbehandlung  }
}
```

* Initialisierung der Verbindung:

```
if((sockfd = initClientConn(ip, PORT_NUMBER)) < 0 )
{ ... Fehlerbehandlung }
```

* Empfang der Konfigurationsdatei für *resinfo:*

```
if(receiveFile(&sockfd, "resinfo-client.conf") < 0)
{ ...Fehlerbehandlung }
```

* Ausführung von *resinfo:*

```
if( system("resinfo -c resinfo-client.conf -f resinfo-client.erg") != 0)
{ ...Fehlerbehandlung }
```

* Versenden der Ergebnisse des Programmlaufs von *resinfo:*

```
if(transmitFile(sockfd, "resinfo-client.erg") < 0 )
{ ...Fehlerbehandlung }
```

* Beenden der Kommunikation mit dem Server:

```
if(closeClientConn(&sockfd) <0)
{ ...Fehlerbehandlung }
```

4.4.3 Verwendung der Start-Nachricht

Um die beim Serverstart generierte Start-Nachricht zur Benachrichtigung bekannter Clients zu nutzen, ist es notwendig, den Client in die Konfigurationsdatei */etc/inetd.conf* des Internet-Superservers *inetd* aufzunehmen. Außerdem muß der Client in der Datei */etc/services* eingetragen werden. Dadurch wird gewährleistet, daß der Client automatisch beim Eintreffen einer Start-Nachricht gestartet wird. Einträge in der Datei */etc/inetd.conf* haben folgendes Schema:

```
<Servicename>  <Sockettyp>  <Protokoll>  <Flags>  <Nutzer> \
<Serverpfad>  <Programmargumente>
```

Der Eintrag für den automatischen Start des Clients unter der Voraussetzung, daß der Client im Verzeichnis */usr/bin* befindlich ist, hat die Form:

```
client  dgram  udp  wait  root  /usr/bin/client  client -d \
-v <Arbeitsverzeichnis>
```

Das Schema für Einträge in die Datei */etc/services* ist der nachfolgenden Zeile zu entnehmen:

```
<Programmname>    Portnummer/udp   <Beschreibung>
```

Der Eintrag für den Client hat demnach folgende Form:

```
client            55300/udp        Dienst zur aut. Konfiguration
```

Kapitel 5

Protokollbewertung

In diesem Kapitel liegt das Hauptaugenmerk auf der Bewertung der Protokollimplementation unter Einbeziehung verschiedener Aspekte. Eine herausragende Stellung nimmt dabei der Aspekt der Leistungsfähigkeit in Bezug auf die Verarbeitung gleichzeitiger Clientanfragen durch den Server ein. Die durchgeführten Tests und die daraus gewonnenen Ergebnisse werden in Abschnitt 5.1 genauer beleuchtet.

Die Leistungsfähigkeit in Bezug auf die Übertragungsgeschwindigkeit wird keiner näheren Betrachtung unterzogen. Der Grund sind die geringen Datenmengen, die zu erwarten sind. So bewegten sich die zu übertragenden Dateigrößen während der Tests mit der Beispielanwendung im Bereich von 512 Byte für die Konfigurationsdatei von *resinfo* und 2042 Byte für die von *resinfo* generierte Ergebnisdatei.

Der Abschnitt 5.2 beschäftigt sich mit einem weiteren wichtigen Aspekt der Bewertung der Protokollimplementation, der Fehleranfälligkeit.

5.1 Leistungsfähigkeit

Der Server wurde so gestaltet, daß jede Clientanfrage in einem eigenen Prozess abgearbeitet wird. Damit ist die Verarbeitung gleichzeitiger Anfragen von der erlaubten Zahl an Prozessen auf einem System abhängig. Zu dieser Zahl existieren teilweise sehr widersprüchliche Angaben. Den Ausgaben des Befehls *ulimit* zu Folge stehen auf dem Testsystem maximal 2047 Prozesse pro realer Nutzer-ID zur Verfügung. Die Abfrage der Konstante *_SC_CHILD_MAX* mit der Funktion *sysconf()* liefert den Wert 999 für die maximale Prozessanzahl pro realer Nutzer-ID. Keiner der beiden Werte kann zur Laufzeit als verfügbar angesehen werden, da sie beide Maximalwerte darstellen und nicht der aktuell verfügbaren Prozessanzahl entsprechen müssen. Weitere Faktoren, die die Anfrageverarbeitung stark beeinflussen sind die Systemauslastung und die Speicherbelegung.

Konkrete Aussagen zur Leistungsfähigkeit können nur mit praktischen Tests getroffen werden. Zum Testzeitpunkt stand kein großes Netzwerk zur Verfügung, dessen Rechner gleichzeitig Anfragen an den Server schicken können. Um trotzdem eine Abschätzung der Anzahl der gleichzeitig verarbeitbaren Anfragen treffen zu können, wurde auf ein aus zwei Rechnern bestehendes Testsystem zurückgegriffen. Ein Rechner fungierte dabei als Server. Der andere Rechner diente zur Generierung der Clientanfragen. Beide Rechner waren über ein 10 MBit Netzwerk miteinander verbunden. Bei dem zur Anfragegenerierung genutzten Rechner handelt es sich um einen AMD Athlon mit einer Taktfrequenz von 1200 MHz und 256 MB Arbeitsspeicher. Der als Server fungierende Rechner ist ein Intel Pentium mit 233 MHz Taktfrequenz und 98 MB Arbeitsspeicher.

Zur Emulation mehrerer gleichzeitiger Clientanfragen erfolgte der Aufruf des Clientprogramms aus einem Hilfsprogramm heraus. In diesem Hilfsprogramm wird für jeden Clientaufruf ein eigener Prozess abgespaltet, so daß diese ihre Anfragen relativ zeitgleich an den Server senden können. Zur Erfüllung dieser Aufgabe erfolgt in einer for-Schleife die Abspaltung der Kindprozesse mittels *fork()*. In jedem dieser Kindprozesse wird ein Client durch den Aufruf der Funktion *system()* gestartet. Dazu wird der Funktion als Parameter die Zeichenkette "client -s bart" übergeben.

Da die Betonung auf der Ermittlung gleichzeitig durch den Server beantwortbarer Anfragen lag, wurde im Client auf die Ausführung des Programms *resinfo* verzichtet. Somit wurde die Systemlast auf dem für die Clientanfragen zuständigen Rechner vermindert und eine höhere Anfragegeschwindigkeit erreicht. Der logische und physikalische Aufbau sind zur Veranschaulichung in Abbildung 5.1 dargestellt.

Bei jedem Durchlauf des Hilfsprogramms wurden 500 Clients erzeugt, die ihre Anfragen an den Server schicken. Nach dem Durchlauf wurde anhand der Logdateien bewertet, wieviele der Clients ihre Daten fehlerfrei senden und empfangen konnten.

Da alle kritischen Funktionsaufrufe im nichtblockierenden Modus, meist in der Verbindung mit dem *select()*-Aufruf erfolgen, sind die erzielten Ergebnisse stark von der Größe des in der Konstante $TIME\_OUT\_SEC$ festgelegten Zeitintervalls abhängig. Das Zeitintervall wird in der Einheit Sekunde angegeben. Wenn ein Zeitgeber abläuft ohne das ein Ereignis eintritt, so wird eine Fehlervariable erhöht. Erreicht diese Variable den Wert MAX_FAIL, so wird dies als Fehler gewertet und die Schnittstelle, in der dieser Fehler auftrat mit einem negativem Rückgabewert und einer Fehlermeldung verlassen.

Für die Messungen wurde die Konstante MAX_FAIL auf den Wert 5 gesetzt. Die Tests wurden mit den Werten 1, 2, 4, 6, 8, 10, 12 für die Konstante $TIME\_OUT\_SEC$ durchgeführt. Die erzielten Ergebnisse sind in Tabelle 5.1 dargestellt. Der Abbruch der nichterfolgreichen Clients erfolgte in der Mehrzahl bei dem Versuch der Verbin-

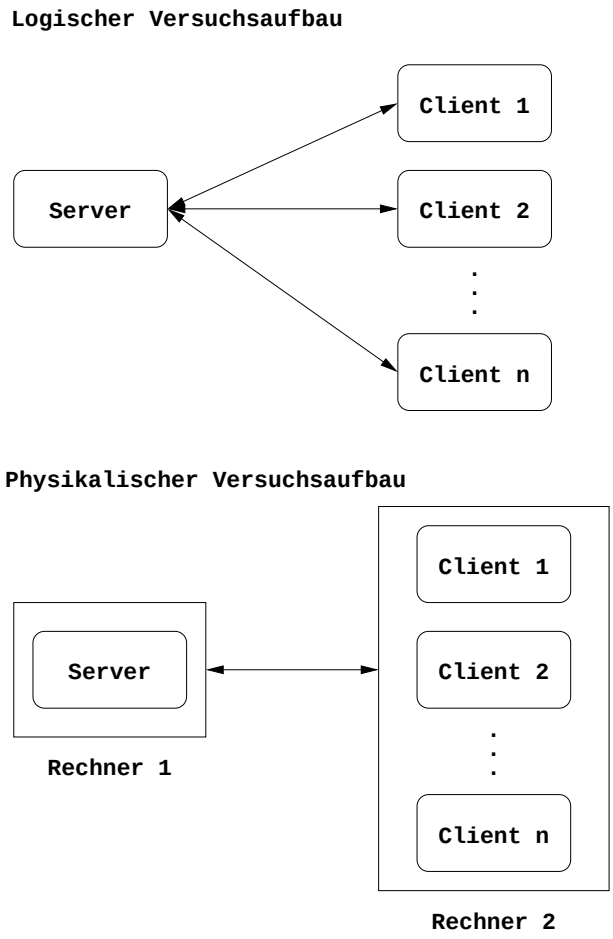

Abbildung 5.1: Logischer und physischer Versuchsaufbau

dungsaufnahme zum Server. In seltenen Fällen erfolgte der Abbruch während der Dateiübertragung. Die Ursache der Abbrüche ist in allen Fällen in der hohen Auslastung des Servers zu sehen. Aus diesem Grund war der Server nicht in der Lage in der geforderten Zeitspanne zu antworten.

5.2 Fehleranfälligkeit

Fehler können auf verschiedenen Ebenen während der Kommunikation auftreten. Es kann zu Verlusten bzw. Verfälschungen auf Bit- bzw. Byteebene kommen. Diese Fehler werden von Fehlererkennungsmechanismen von TCP behandelt. Da es sich dabei um ausgereifte Implementationen handelt, kann davon ausgegangen werden, daß diese Fehler von diesem Protokoll zuverlässig behandelt werden. Zur zusätzlichen Absicherung werden Längeninformationen mit den eigentlichen Daten übertragen und im Empfänger verglichen, um im Fehlerfall die Daten erneut anzufordern.

Zeitintervall [s]	Erfolgreiche Clients	Nichterfolgreiche Clients
1	59	441
2	149	351
4	299	201
6	401	99
8	461	39
10	496	4
12	500	0

Tabelle 5.1: Anzahl erfolgreicher und nichterfolgreicher Clients in Abhängigkeit von der Intervall-Länge (*TIME_OUT_SEC*)

Eine weitere Möglichkeit für Fehler sind Ausfälle auf Client- oder Serverseite während der Übertragung. Diese Fehler dürfen nicht dazu führen, daß der Kommunikationspartner in einem unbestimmten Zustand stehen bleibt. Diese Fehlerart wurde in verschiedenen Szenarien getestet. So wurden Ausfälle nach Schnittstellenaufrufen und innerhalb der Schnittstellenaufrufe simuliert. Dazu wurde der Ausfall entweder im Client oder im Server mit der Systemfunktion *exit()* erzwungen. Diese Funktion bewirkt das sofortige Programmende und entspricht damit in der Wirkung dem unvorhergesehenen Ausfall eines Kommunikationspartners. In allen getesteten Fällen wurde der Programmablauf wie vorgesehen mit einer Fehlermeldung beendet.

Fehler können auch während des Kommunikationsaufbaus auftreten. Auch diese Fehlerart wurde abgetestet. So wurde zum Beispiel das Verhalten eines Clients getestet, wenn der angegebene Server nicht aktiv ist. Auch in diesem Fall wurde der Programmablauf, nach dem in der Konstante *MAX_FAIL* angegebene Anzahl Fehlversuche erreicht wurde, mit einer Fehlermeldung unterbrochen.

Kapitel 6

Schlusswort

Rückblickend auf die vorliegende Arbeit kann festgestellt werden, daß eine an den in Abschnitt 3.2 formulierten Protokollanforderungen orientierte Implementation erstellt wurde, die in ersten Tests den Erwartungen insbesondere in Bezug auf die Leistungsfähigkeit standhält. Trotz allem können gesicherte Aussagen erst nach ausführlichen Praxistests gemacht werden.

Die übertragenen Daten sind im gewissen Maße als sensibel und sicherheitskritisch einzustufen, da sie detaillierte Informationen über die Ausstattung der beteiligten Rechner beinhalten. Unbefugte, die an diese Informationen gelangen, können diese nutzen um Schwachstellen in den beteiligten Systemen auszukundschaften und danach gezielt Angriffe auf diese Rechner zu starten. Deshalb bieten sich für den Einsatz in unsicheren Netzwerken wie dem Internet einige Erweiterungen an, um die Daten besser vor dem Zugriff Unbefugter zu schützen. Eine Maßnahme ist die Einführung von Identifizierungsschlüsseln, die nur den beteiligten Rechnern bekannt sind. Diese Maßnahme würde es verhindern oder zumindest erschweren, daß ein Unbefugter Informationen von anderen Rechnern abfragt, indem er sich als Server ausgibt. Eine weitere Sicherung der Datenübertragung kann durch eine zusätzliche Verschlüsselung der Daten erreicht werden. Dies verhindert das Mithören bestehender Verbindungen durch die Verwendung von Packetsniffern.

Die jetzt zur Verfügung stehende Funktionalität, wie sie in der Beispielanwendung unter Verwendung von *resinfo* demonstriert wurde, kann zur zentralen Abfrage gewünschter Informationen von Rechnern genutzt werden. Mit der Erweiterung um eine Datenbank auf dem Server können so innerhalb eines Netzwerkes alle wichtigen Informationen zur Hard- und Software zentral gehalten werden und zu Administrationszwecken genutzt werden. Für die in der Einleitung erwähnte Funktionalität ist die Erstellung einer datenbankgestützten Anwendung nötig.

Literaturverzeichnis

[BH02] H. Bürger, F. Hofmann: System-Ressourcenanalyse unter Linux, Leipzig, 2002

[BEE01] Hall, Brian: Beej's Guide to Network Programming, http://www.ecst.csuchico.edu/~beej/guide/net/html/ , 2001

[HER99] Herold,Helmut: Linux-Unix-Sytemprogrammierung, Addison-Wesley, Bonn, 1999

[TAN96] Tanenbaum, Andrew S.: Computernetzwerke - Dritte Auflage, Prentice Hall,München, 1997

Anhang A

protokoll.h

```
#ifndef __PROTOKOLL_H__
#define __PROTOKOLL_H__
#define MAX_CHAR 300
#define TIME_OUT_SEC 10
#define TIME_OUT_MSEC 0
#define CHILD_WAIT 20
#define MAX_FAIL 5
#define IDENT "NETConfig"
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <signal.h>
#include <errno.h>
#include <wait.h>
#include <syslog.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/select.h>
#include <sys/times.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netdb.h>
enum message { BEGIN=1, OK, FAIL, END_OF_FILE, END_OF_COMM};
struct buffer {   int len;   int flag;   char data[MAX_CHAR]; };
int child, final;
/**** Serverschnittstellen ****/
/* Signalbehandlungsfunktion für den Server*/
void sig_handler(int signr);
/* Servernachricht an Clientliste*/
int informClients(char *filename, int port);
/* Initialisierung des Sockets auf Serverseite*/
```

```
int initServerConn(int port, struct sockaddr_in *address);
/* Warten auf Clientverbindungen und Parallelisierung */
int getServerConn(int *sockfd, struct sockaddr_in *address);
/* Warten auf Ende des Clients*/
int waitClientEnd(int *sockfd);
/* Beendigung des Servers */
int serverShutdown(void);
int confirmShutdown(void);
/******** Clientschnittstellen ********/
/* Warten auf Benachrichtigung von Server */
int receiveStartMsg(char *ip);
/* Initialisierung des Sockets auf Clientseite */
int initClientConn( char *dest, int port);
/* Ende der Kommunikation */
int closeClientConn(int *sockfd);
/******** Schnittstellen für Dateitransfer *******/
/* Zeilenweises Versenden einer Datei */
int transmitFile(int connfd, char *filename);
/* Empfang von Daten via Socket und Speicherung in Datei*/
int receiveFile(int *connfd, char *filename);
/******Hilfsfunktionen******/
int setNonBlock(int *fd);
int initAddressStructure(int port, char *dest,struct sockaddr_in *address);
#endif /* __PROTOKOLL_H__ */
```