Sicheres Broadcasting für DISCOUNT

Jürgen Lind

Projektarbeit
September 1992

durchgeführt am Fachbereich Informatik
der Universität Kaiserslautern
AG Avenhaus
Betreuer: Jörg Denziger
Peter Sturm

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 0

Einleitung

Auf dem Gebiet des automatischen Beweisens und der Vervollständigung sind sequentielle Algorithmen und Verfahren schon seit längerem bekannt, allerdings stellt eine effiziente Implementierung dieser Algorithmen nach wie vor hohe Anforderungen an die Entwickler entsprechender Systeme.

Im Zeitalter immer leistungsfähigerer, vernetzter Kleincomputer (workstations) erscheint es sinnvoll, die auf anderen Gebieten der Informatik bereits erprobte Methode der verteilten Berechnung auf die Probleme der Vervollständigung zu übertragen. Ein Ansatz, diese Verteilung zu realisieren, ist die von J. Denzinger (siehe [De91], [De93]) vorgestellte Methode des Teamworks. Mit diesem Ansatz ist es theoretisch möglich, eine unbegrenzte Zahl von Rechnern am gleichen Problem arbeiten zu lassen. In der Praxis wird man sich allerdings eher in den Dimensionen eines LAN's (Local Area Networks) bewegen.

Da bei der Teamwork-Methode des öfteren eine gebündelte Übertragung großer Datenmengen an viele Empfänger notwendig ist, sollten alle Möglichkeiten der Hardware und des Betriebssystems ausgenutzt werden, um diesen Transfer so effizient wie möglich zu gestalten. Die Benutzung von Broadcasts, d. h. der (physisch) gleichzeitigen Übermittlung einer Nachricht an mehrere Empfänger bietet sich hierbei an. Das zugrundeliegende Betriebssystem erlaubt zwar die Versendung von Broadcast-Nachrichten, allerdings ist diese Übertragung ungesichert. Aus diesem Grund muß ein sicheres Broadcasting innerhalb der Anwendungsprogramme realisiert werden. Das Ziel dieser Arbeit ist es, der Team-Vervollständigung eine effiziente und bequeme Möglichkeit zur Übertragung von Broadcast-Nachrichten zur Verfügung zu stellen.

In Kapitel 1 erfolgt zunächst eine kurze Einführung in die Kommunikationsprimitive des Betriebssystems UNIX und eine Darstellung der Schwierigkeiten, die sich bei der Versendung von Broadcasts ergeben. In Kapitel 2 wird dann die Arbeitsweise des Protokolls, in Kapitel 3 seine Implementierung vorgestellt. Kapitel 4 schließlich beschreibt die Erfahrungen mit dem Protokoll und seiner Verwendung im Rahmen der Team-Vervollständigung.

Ich möchte mich an dieser Stelle noch bei meinen Betreuern Jörg Denzinger und Peter Sturm, die mich bei dieser Arbeit in allen Belagen unterstützt haben, bedanken. Ebenso ergeht ein Dank an Werner Pitz, der mit seiner guten, verständlichen Implementierung des DISCOUNT-Systems diese Arbeit möglich gemacht hat.

Kapitel 1

Grundlagen

Im ersten Teil dieses Kapitels werden die Kommunikationsprimitive zwischen Prozessen im Betriebssystem UNIX[1] auf der Basis von Socket-Verbindungen und deren Verwendung in C-Programmen kurz erläutert. Insbesondere ist hier die Kommunikation zwischen Prozessen, die auf *verschiedenen* Rechnern innerhalb eines Netzwerks ablaufen, von Interesse. Im zweiten Teil werden die Schwierigkeiten, die sich bei Verwendung von Broadcast-Kommunikation in einem UNIX-Netzwerk ergeben, dargestellt.

1.1 Interprozesskommunikation unter UNIX

Die Team-Vervollständigung (siehe [De91] und [Pi92]), die in Rahmen dieser Arbeit um die Möglichkeit des Versendens von Daten mittels Broadcast-Nachrichten erweitert werden soll, läuft unter einer Sun-Adaption von UNIX 4.3BSD. Dieses Betriebssystem bietet als grundlegendes Konzept der Interprozeßkommunikation sog. Sockets an. Ein UNIX-Prozeß kann eine vom Betriebssystem bestimmte Anzahl von Sockets einrichten und konfigurieren, dabei ist die Anzahl der Sockets pro Prozeß und *pro Rechner* allerdings begrenzt. Konfigurieren bedeutet in diesem Zusammenhang, daß der Anwender bestimmte Eigenschaften (z. B. Broadcasting) für die Nachrichtenübertragung auswählen kann.

Damit zwei Prozesse über Sockets miteinander Nachrichten austauschen können, müssen die Sockets vom gleichen Typ (s. u.) sein und der Client muß die Adresse des Servers kennen. Das Format dieser Adresse ist von der verwendeten Protokollfamilie, der sog. *Domain*, abhängig. Man bezeichnet diese Protokollfamilien deshalb auch als Adreßfamilien.

[1]Unix ist eingetragenes Warenzeichen der Firma AT&T.

Die drei Domains, die momentan in UNIX 4.3BSD zur Verfügung stehen, sind:

- die UNIX-Domain (AF_UNIX), die nur Verbindungen zwischen Prozessen, die auf dem gleichen Rechner ablaufen, erlaubt. Bei dieser Protokollfamilie dient ein normaler Dateiname als Adresse eines Sockets.
- die Internet-Domain (AF_INET), welche die Internet-Protokollfamilie (ARPA-Netz) unterstützt. Wichtige Protokolle aus dieser Familie sind das *Transmission Control Protocol (TCP/IP)* und das *User Datagram Protocol (UDP/IP)*. Die Sockets werden hier durch eine 32-Bit Rechnernummer und eine 16-Bit Socketnummer adressiert.
- das von der Xerox Corporation entwickelte *Xerox Network System (AF_NS)*. Die in XNS vorhandenen Protokolle sind frei verfügbar und können bei Xerox eingesehen werden. Im Aufbau ähnelt XNS sehr dem TCP/IP-Protokollsatz, nähere Informationen finden sich z. B. in [Ste92].

Wie bereits oben erwähnt, unterscheidet man Sockets auch nach ihrem Typ. Dies können die folgenden sein:

- Datagramm-Sockets (SOCK_DGRAM) erlauben eine ungesicherte, bidirektionale Datenübertragung zwischen Prozessen. Ungesichert bedeutet, daß Datenpakete verlorengehen, in der Reihenfolge vertauscht oder auch dupliziert werden können. Da Datagramm-Sockets nicht verbindungsorientiert arbeiten, muß bei jeder Nachricht die Adresse des Empfängers angegeben werden; ein Vorteil dieser Art von Sockets ist jedoch die Möglichkeit des Broadcasts (siehe 1.2). In der Internet-Domain wird das UDP/IP Protokoll für diesen Sockettyp verwendet.
- Stream-Sockets (SOCK_STREAM) bieten eine gesicherte, bidirektionale Nachrichtenübermittlung, nachdem explizit eine Verbindung zwischen Client und Server aufgebaut worden ist. Da diese Art der Verbindung sicher ist, muß sich der Anwender nicht um verlorene, vertauschte oder verdoppelte Pakete kümmern. Es kann allerdings vorkommen, daß Datenpakete verschmolzen oder gesplittet werden, d. h. die Paketgrenzen bleiben nicht unbedingt erhalten. Aufgrund der verbindungsorientierten Arbeitsweise ist bei Streamverbindungen kein Broadcast möglich. In der Internet-Domain wird dieser Sockettyp durch das TCP/IP-Protokoll zur Verfügung gestellt.

- Sequenzed-Packet-Socket
  (SOCK_SEQPAKET) unterscheiden sich von Stream-Sockets nur durch die Tatsache, daß die Paketgrenzen garantiert erhalten bleiben. In UNIX 4.3BSD ist dieser Sockettyp in der XNS-Domain realisiert.

- Raw-Sockets (SOCK_RAW) sind für normale Benutzer nicht zugänglich, sie werden hauptsächlich beim Entwickeln und Testen von neuen Protokollen eingesetzt.

- Reliable-Delivered-Messages-Sockets (SOCK_RDM) entsprechen den Datagramm-Sockets, allerdings bieten sie eine gesicherte Übertragung der Daten. In UNIX4.3BSD gibt es keine Domain, die diesen Sockettyp unterstützt.

UNIX bietet eine Reihen von Systemaufrufen an, mit deren Hilfe Sockets erzeugt und konfiguriert, Verbindungen aufgebaut und Daten gesendet bzw. empfangen werden können (alle folgende Angaben beziehen sich auf die Internet-Domain, da die UNIX-Domain für die vorliegende Arbeit nicht geeignet ist).

Der `socket`-Systemaufruf erfolgt immer zuerst und hat die folgende Aufrufschnittstelle:

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

Mit diesem Aufruf wird ein Socket mit dem angegebenen Typ in der spezifizierten Domain erzeugt. Der Parameter *protocol* wählt ein bestimmtes Protokoll aus der Domain aus; hat er den Wert 0, so wird das erste Protokoll, das den Sockettyp realisiert, benutzt (i. A. gibt es für jeden Typ auch nur ein Protokoll). Der Rückgabewert von `socket` ist ein Deskriptor, der, genau wie ein Dateideskriptor eine Datei, den Socket eindeutig identifiziert.

Nachdem auf diese Art ein Socket erzeugt worden ist, kann der Benutzer diesem einen Namen bzw. eine Adresse geben.

```
int bind ( sock, addr, addrlen )
     int              sock;
     struct sockaddr* addr;
     int              addrlen;
```

Der erste Parameter ist der (von `socket` gelieferte) Deskriptor des Sockets, der eine Adresse erhalten soll. Der zweite ist ein Zeiger auf eine Datenstruktur, die die gewünschte Adresse enthält.

Da ein Rechner mehrere verschiedene Rechnernummern haben kann, verwendet man im Adreßfeld (`server.sin_addr`) normalerweise die Konstante INADDR_ANY, `bind` vergibt dann eine von den verschiedenen Nummer unabhängige, eindeutige Adresse. Die zu vergebende Portnummer (`server.sin_port`) kann festgeschrieben werden, dabei ist allerdings zu beachten, daß die Nummern 0 - 1023 privillegierten Ports zugeordnet sind und deshalb nicht benutzt werden können. Hat `server.sin_port` wie im nachfolgenden Beispiel den Wert 0, so wird der nächste freie Port an den Socket vergeben. Über den Bibliotheksaufruf `getsockname( ...)` kann dann die Portnummer ermittelt werden.

Beispiel für einen `bind`-Aufruf:

```
struct sockaddr_in server;

server.sin_family  = AF_INET;
server.sin_addr    = INADDR_ANY;
server.sin_port    = 0;
if( bind( sock, (struct sockaddr*)&server, sizeof server ) == -1 )
                   /* Fehler */
```

Der `bind`-Systemaufruf wird nur dann gebraucht, wenn der Socket von anderen Sockets adressiert werden soll, d. h. für reine Sendesockets ist er überflüssig.

Während bei Datagramm-Sockets nun sofort mit dem Versenden und Empfangen von Daten begonnen werden kann, müssen bei den verbindungsorientierten Stream-Sockets noch weitere Systemaufrufe erfolgen, insbesondere muß explizit eine Verbindung zwischen den beteiligten Prozessen hergestellt werden.

Zunächst erfolgt am Empfangssocket jedoch ein `listen`-Aufruf:

```
int listen ( sock, length )
    int sock;
    int length;
```

`listen` richtet eine Warteschlange der Länge *length* für ankommende Verbindungswünsche anderer Prozesse ein. Ist die Warteschlange voll, so werden weitere ankommende Verbindungswünsche ignoriert. Die Warteschlange ist nötig, weil in der Zeit, in der ein Server auf einen Verbindungswunsch reagiert, schon Verbindungswünsche anderer Clients eintreffen können. Der derzeitige, durch SunOS vorgegebene, Höchstwert für die Länge der Warteschlange ist 5.

Nachdem die Warteschlange eingerichtet ist, kann eine Verbindung aufgebaut werden:

```
int accept ( sock, fromname, fromlen )
      int              sock;
      struct sockaddr* fromname;
      int*             fromlen;
```

Falls in der Warteschlange ein Verbindungswunsch enthalten ist, erzeugt `accept` als Rückgabewert einen weiteren Socketdeskriptor. Dieser ist dann mit dem anfordernden Socket verbunden. Existiert kein solcher Eintrag, so wird der `accept` aufrufende Prozeß solange blockiert, bis ein Verbindungswunsch eintrifft. Neben dem Socketdeskriptor liefert `accept` in *fromname* und *fromlen* die Adresse des Clients und deren Länge zurück. Der ursprüngliche Socket, auf dem ein `accept` durchgeführt wurde, steht weiterhin zur Verfügung und kann zur Annahme weiterer Verbindungswünsche genutzt werden.

Der Verbindungsaufbau vom Client zum Server wird durch den `connect`- Systemaufruf möglich:

```
int connect ( sock, toname, tolen )
      int              sock;
      struct sockaddr* fromname;
      int              fromlen;
```

Um die Struktur *toname* mit den benötigten Werten füllen zu können, muß dem Client sowohl der Name des Hosts, auf dem der Serverprozeß abläuft, als auch die dem Empfangssocket zugeordnete Portnummer bekannt sein. Kann keine Verbindung aufgebaut werden, weil der Server mit der Abwicklung eines anderen Verbindungswunsches beschäftigt ist oder noch gar kein `accept` durchgeführt hat, wird der Client solange blockiert, bis der Server die Verbindung annimmt. Das Zusammenspiel zwischen Client und Server ist in Abbildung 1.1 noch einmal dargestellt.

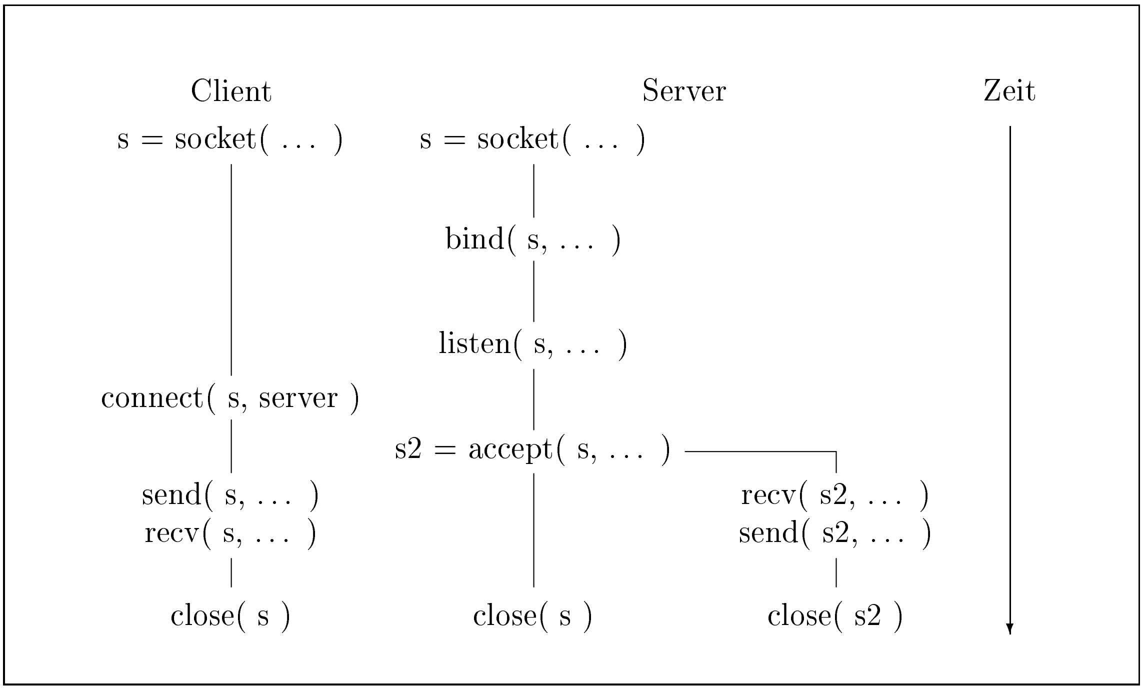

Abbildung 1.1: Verbindungsaufnahme bei Stream-Sockets

Wegen der verbindungslosen Arbeitsweise von Datagramm-Sockets können die normalen `read/write` Funktionen zum Lesen und Schreiben von Daten nicht mehr verwendet werden. Zwei neue Funktionen sind für den Datenaustausch notwendig.

```
int sendto ( sock, buff, nbytes, flags, toaddr, tolen )
     int                sock;
     char*              buff;
     int                nbytes;
     int                flags;
     struct sockaddr*   toname;
     int                tolen;
int recvfrom ( sock, buff, nbytes, flags, fromaddr, fromlen )
     int                sock;
     char*              buff;
     int                nbytes;
     int                flags;
     struct sockaddr*   fromname;
     int*               fromlen;
```

Die ersten Parameter entsprechen denen eines `read`- bzw. `write`-Aufrufs (siehe [SUN87a], [SUN87b]), *toname* und *tolen* spezifizieren bei `sendto` den Empfänger, `recvfrom` liefert in den Variablenparametern den Absender bzw. die Länge der Absenderadresse zurück. Bei Broadcasts wird anstatt der Empfängeradresse die spezielle Broadcast-Adresse des lokalen Netzwerks angegeben.

Für den Datentransport gibt es verschiedene Optionen, die über *flags* eingestellt werden, dies kann z. B. das Lesen einer Nachricht aus dem Buffer ohne sie zu löschen (MSG_PEEK) oder eine extra schnelle Übertragung (MSG_OOB) sein.

Der Rückgabewert der Funktionen ist jeweils die Länge der empfangenen bzw. gesendeten Daten. Es ist möglich, daß mit einem Aufruf von `sendto` bzw. `recvfrom` nicht alle Daten gesendet bzw. empfangen werden. Aus diesem Grund kleidet man die beiden Aufruf üblicherweise in Schleifen, die die Aufrufe (mit angepaßten Parametern) solange wiederholen, bis alle Daten übertragen sind.

Bei den verbindungsorientierten Stream-Sockets stehen neben den üblichen `read/write` Operationen noch folgende Möglichkeiten zum Datentransfer zur Verfügung:

```
int send ( sock, buff, nbytes, flags )
    int   sock;
    char* buff;
    int   nbytes;
    int   flags;
int recv ( sock, buff, nbytes, flags )
    int   sock;
    char* buff;
    int   nbytes;
    int   flags;
```

Die Parameter und Rückgabewerte entsprechen denen von `sendto` bzw. `recvfrom`, die Adreßangaben können natürlich entfallen.

Oftmals ist es notwendig, an mehreren Sockets gleichzeitig auf bestimmte Ereignisse zu warten oder das Warten auf eine Nachricht nach einer bestimmten Zeit abzubrechen. Bei einem normalen `receive`-Aufruf wartet ein Empfänger solange, bis eine Nachricht eintrifft, im ungünstigsten Fall (es existiert, aus welchen Gründen auch immer, kein Sender mehr) unendlich lange. Diese Tatsache ist in vielen Fällen nicht zu tolerieren, also muß man Maßnahmen ergreifen, um ein solches „Hängen“ des empfangenden Prozesses zu vermeiden. Eine Möglichkeit wäre zum Beispiel, vor dem `receive`-Aufruf einen neuen Prozeß zu starten, der nach einer bestimmten Zeit eine Nachricht an den Vater-Prozeß schickt und damit das Ende der Wartezeit andeutet. Ebenso wäre es möglich, den `receive`-Aufruf nach einer bestimmten Zeit durch ein Alarmsignal zu unterbrechen und dann vom Signalhandler aus eine Nachricht an den Socket, an dem gewartet wird, zu schicken. Diese Möglichkeiten sind zwar durchaus praktikabel, mit dem

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

System-Aufruf geht es allerdings wesentlich einfacher.

Die Parameter 2 bis 4 des `select`-Aufrufs sind Mengen vom Typ fd_set und enthalten diejenigen Socketdeskriptoren, die zu überprüfen sind. Die Deskriptoren der Mengen *readfds* und *writefds* werden daraufhin geprüft, ob sie bereit zum Lesen bzw. Schreiben sind, die Deskriptoren in *exceptfds* werden auf besondere Bedingungen hin abgefragt (z. B. Out-Of-Band-Daten für einen Socket). *maxfdpl* enthält jeweils die höchste Socketnummer, die geprüft werden soll.

Mit dem letzten Parameter kann das Verhalten des `select`-Aufrufs beeinflußt werden:

- *timeout* ist ein Zeiger auf eine `timeval` Struktur, deren Zeitzähler gleich 0 ist. `select` kehrt in diesem Fall sofort nach der Überprüfung der Sockets zurück.

- *timeout* ist ein Zeiger auf eine `timeval` Struktur, deren Zeitzähler ungleich 0 ist. `select` wartet dann solange bis mindestens ein Socket bereit zur Ein/Ausgabe ist, jedoch nicht länger als die angegebene Zeit. Der Rückgabewert im Fall eines Timeouts ist 0.

- *timeout* hat den Wert `NULL`. `select` wartet auf jeden Fall, bis ein Deskriptor bereit ist, der Aufruf kann allerdings durch ein Signal unterbrochen werden.

Die Parameter *readfds, writefds* und *exceptfds* sind Variablenparameter, nach dem Ende des Aufrufs enthalten sie die Deskriptoren, bei denen eine Überprüfung erfolgreich war. Der Rückgabewert von `select` ist die Anzahl dieser Deskriptoren.

Wird ein Socket nicht mehr benötigt, so wird er mit `close` geschlossen:

```
int close ( sock )
    int sock;
```

Mit diesem Aufruf wird auch eine Adresse bzw. Portnummer, die an den Port gebunden wurde, wieder freigegeben.

Eine andere Funktion, die zur Konfigurierung von Sockets zur Verfügung steht, sei an dieser Stelle ebenfalls noch erwähnt:

```
int setsockopt ( sock, level, optname, optval, optlen )
    int   sock;
    int   level;
    int   optname;
    char* optval;
    int   optlen;
```

`setsockopt` ist eine sehr mächtige Funktion, deren genaue Arbeitsweise hier nicht näher erläutert werden soll. Es genügt zu sagen, daß mit dieser Funktion ein Socket u. a. auf das Versenden von Broadcast-Nachrichten umgeschaltet werden kann:

```
int on = 1;
setsockopt( sock, SOL_SOCKET, SO_BROADCAST, on, sizeof on );
```

1.2 Broadcast in UNIX-Netzwerken

Unter einem Broadcast versteht man ganz allgemein, daß ein Sender an mehrere Empfänger Nachrichten verschickt. So ist z. B. das Radio eine Art der Broadcast-Kommunikation. Während beim Radio dem Sender die Empfänger i. A. nicht bekannt sind und der Dienst unbestätigt ist (d. h. der Sender weiß nicht, ob die Nachricht angekommen ist), wird im folgenden davon ausgegangen, daß dem Sender eine Liste mit Empfängern vorliegt, welche die Datenpakete quittieren (bestätigter Dienst). Im Zusammenhang mit Rechnernetzen unterscheidet man zwischen physischem und virtuellem Broadcast. Beim virtuellen Broadcast hat der Anwender lediglich den Eindruck, daß die Nachrichten an alle Empfänger gleichzeitig geschickt werden. Tatsächlich ist es aber so, daß das Betriebssystem die Nachrichten *nacheinander* an die einzelnen Empfänger sendet. Beim physischen Broadcast laufen die Nachrichten wirklich nur einmal über das Netz (so auch beim hier verwendeten Ethernet).

Beim Broadcast in einem UNIX-Netzwerk sind verschiedene Einschränkungen zu beachten, von denen zwei wesentliche im nächsten Abschnitt kurz dargestellt werden.

1.2.1 Broadcast über das lokale Netz hinaus

Ein LAN (Local Area Network) besteht normalerweise aus mehreren Subnetzen die über sogenannte *Gateways* (korrekt: Schicht-4-IWU (Internetworking Unit)) miteinander verbunden sind (siehe Abb. 1.2). Wird eine Nachricht von einem Rechner A in Netz 1 an einen Rechner Y in Netz 2 verschickt, übernimmt das Gateway die Weiterleitung der Nachricht (sofern die Datenübertragung zwischen den beiden Netzen erlaubt ist). Falls die Netze nicht direkt miteinander verbunden sind, muß die Nachricht über mehrere Gateways laufen. Die Wege, welche die Nachricht dabei nimmt, sind dem Benutzer i. A. nicht bekannt (Routing).

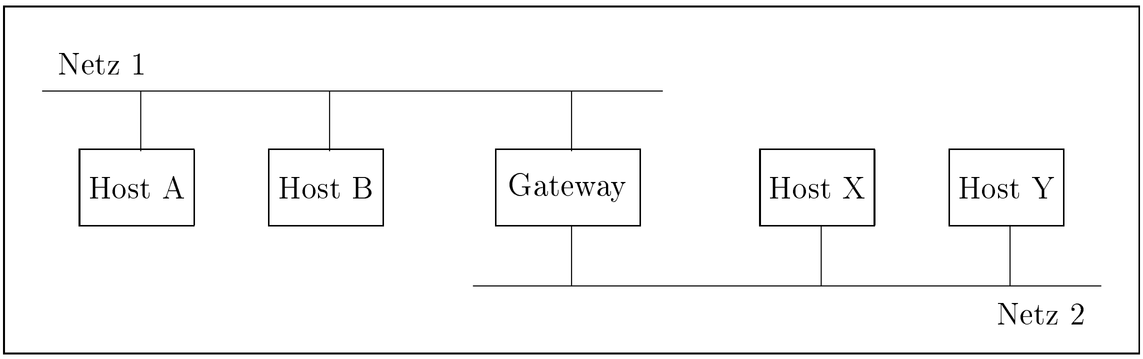

Abbildung 1.2: Broadcast über das lokale Netz hinaus

Gateways können neben der einfachen Weiterleitung von Daten zwischen Rechnernetzen auch noch andere Aufgaben wahrnehmen. Wie bereits im vorangegangenen Absatz erwähnt, werden sie z. B. dazu genutzt, bestimmte Nachrichten nicht in das lokale Netz hinein-, bzw. aus ihm herauszulassen. Dies kann aus Sicherheitsgründen oder einfach zur Reduzierung des Nachrichtenaufkommens geschehen.

Dieses Nicht-Weiterleiten von Datenpaketen ist genau das Problem, das beim Broadcasting auftaucht: Wird nämlich ein Socket mittels `setsockopt( ...)` auf das Versenden von Broadcast-Nachrichten umgeschaltet, so bezieht sich dies nur auf das lokale Netz, Gateways leiten Broadcast-Nachrichten i. A. nicht an benachbarte Netze weiter. Möchte man das Broadcasting auf mehrere Subnetze ausdehnen, so muß man auf jedem Gateway einen Dämonprozeß starten, der die Weiterleitung der Nachrichten auf das andere Subnetz übernimmt. Dies ist allerdings auch nur dann möglich, wenn es sich bei dem Gateway um einen General Purpose-Rechner (z. B. eine Workstation) handelt.

Diese Erweiterung des Broadcasting über das lokale Netz hinaus hat zwei entscheidende Nachteile:

(1) Alle Nachrichten und Quittungen müssen doppelt verschickt werden, die Geschwindigkeit der Kommunikation nimmt drastisch ab

(2) Ist das Netz, in das der Dämon die Nachrichten weiterleiten soll, wesentlich größer als das lokale Netz und sollen in diesem Netz nur wenige Rechner durch den Broadcast erreicht werden, so ist die Belastung dieses Netzes unverhältnismäßig hoch (Beispiel: Netz 1 ist ein AG-Netz, Netz 2 das Campus-Netz)

Da für die Team-Vervollständigung nur wenige Rechner (< 10) eingesetzt werden, ist es nicht sinnvoll, diese Erweiterung zu implementieren, denn in einem lokalen Netz sind i. A. eine ausreichende Anzahl von Rechnern vorhanden.

1.2.2 Mehrere Empfänger auf einem Rechner

Wie bereits in 1.1 erwähnt, werden in der Internet-Domain Sockets durch eine 32-Bit Rechnernummer und eine 16-Bit Portnummer adressiert. Diese Portnummer wird jedoch nicht innerhalb eines jeden Prozesses sondern vielmehr prozeßübergreifend für den ganzen Rechner vergeben. Durch dieses Verfahren ist es unmöglich, daß mehrere Prozesse eines Rechners an ein und demselben Port auf Nachrichten warten.

Auch hier kann man durch den Einsatz eines Kommunikationsdämons Abhilfe schaffen, doch ist dies, ebenso wie in 1.2.1, für die Team-Vervollständigung nicht unbedingt sinnvoll. Der Dämonprozeß läuft auf allen Rechnern und wickelt die Broadcastkommunikation

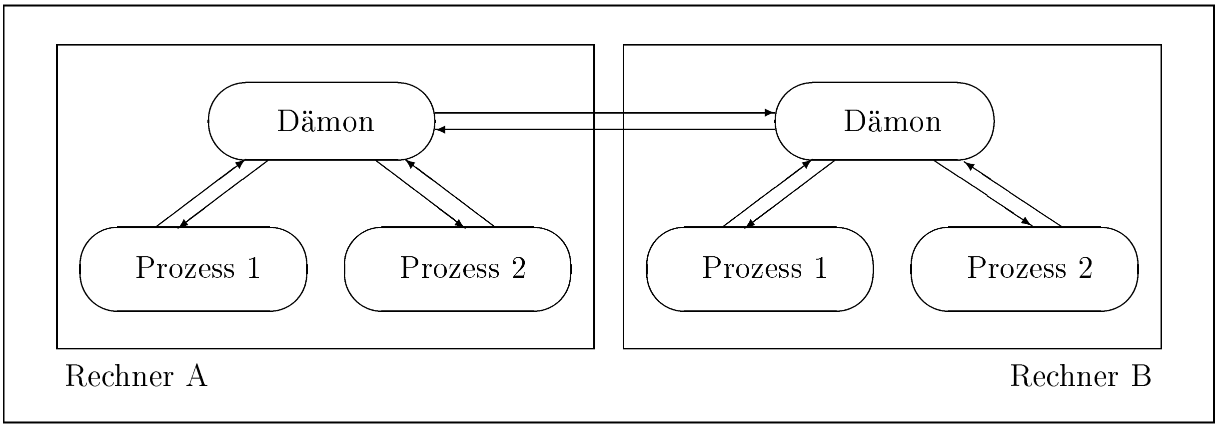

Abbildung 1.3: Mehrere Empfänger auf einem Knoten

zwischen diesen ab, d. h. er empfängt Datagramme, quittiert sie und leitet sie über Stream-Verbindungen an die Klienten auf seinem Knoten weiter (siehe 1.3). Das Versenden von Broadcasts kann nach wie vor von jedem Prozeß selbst durchgeführt werden.

Bei dieser Lösung wird das Nachrichtenaufkommen mehr als verdoppelt, die Leistungssteigerung, die durch den Einsatz von Broadcasting erzielt werden soll, wird höchstwahrscheinlich aufgehoben. Genaugenommen sinkt der Durchsatz wahrscheinlich noch unter den alten Wert, weil die Daten zusätzlich zum Broadcast noch über Streams an die eigentlichen Empfänger weitergeleitet werden müssen. Dies ist auch dann der Fall, wenn auf einem Rechnerknoten nur ein Empfängerprozeß läuft. Neben diesem schwerwiegenden Nachteil ist zudem die Implementierung dieser Lösung sehr aufwendig, so daß sie im Rahmen dieser Arbeit nicht näher betrachtet wird.

Alles in allen ist zu sagen, daß eine Verteilung der Team-Vervollständigung über ein lokales Netz hinaus wegen der aufwendigen Implementierung nicht sinnvoll ist. Falls eine Verteilung auf eine große Zahl von Rechner jedoch irgendwann einmal nötig werden sollte, ist eine Umstellung der gesamten Kommunikation auf existierende Verteilungssysteme wie z. B. LIPS (siehe [Bu91]) in Betracht zu ziehen.

Kapitel 2

Das Protokoll

Um Broadcast-Nachrichten gegen Verluste etc. zu schützen, muß ein Protokoll entwickelt werden, das in der Lage ist, verlorene Pakete zu erkennen und erneut anzufordern. Außerdem sollten duplizierte Pakete erkannt und vernichtet werden. Bei manchen Anwendungen, so auch bei der Team-Vervollständigung, ist es zudem wichtig, die Erhaltung der Nachrichtenreihenfolge zu garantieren.

Das Protokoll, welches hier vorgestellt wird, ist in der ISO-Terminologie zwischen der IP-Schicht und der Anwendung angesiedelt (vgl. Abb. 2.1).

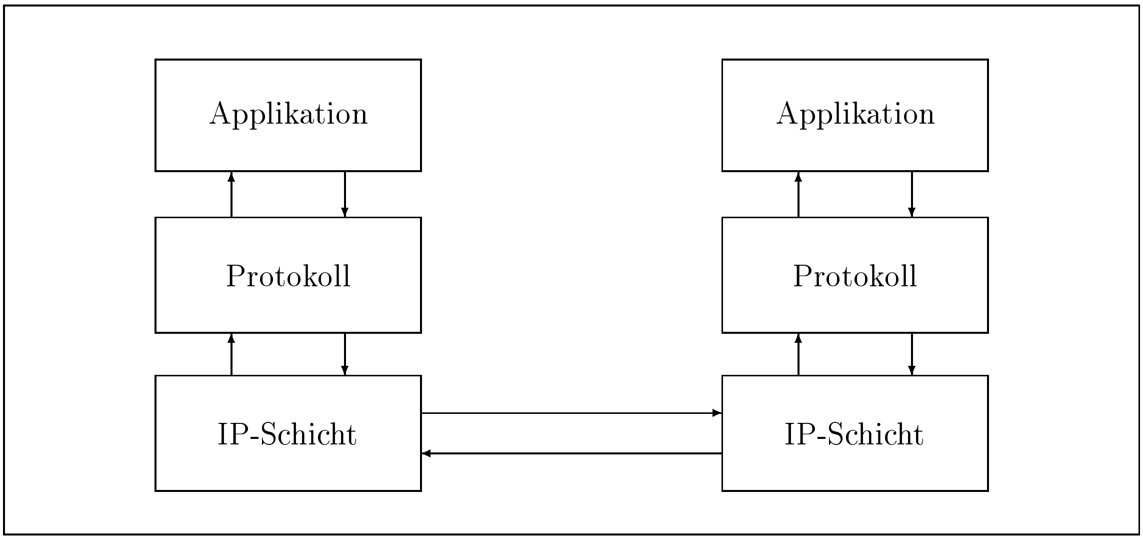

Abbildung 2.1: Lage des Protokolls im ISO-Schichtenmodell

Als zugrundeliegende Schicht wird die IP-Schicht angegeben, da sowohl UDP als auch TCP darauf aufsetzen. Für das Verschicken von Broadcasts stehen nur Datagramme (UDP) zur Verfügung, Quittungen und fehlende Pakete werden dagegen durch sichere Stream-Verbindungen (TCP) übertragen. Dies geschieht deshalb, weil Datenpakete i. A. nicht netzweit, sondern vielmehr rechnerspezifisch verloren gehen. Aus diesem Grund bringt eine erneute Übertragung der verlorenen Pakete über einen weiteren Broadcast normalerweise keine Vorteile, bedingt aber weitere Quittungen. Jedesmal, wenn ein Rechner sich zum Versenden von Broadcast-Nachrichten vorbereitet, erweitert sich das Netz der Stream-Verbindungen, und zwar solange, bis ein vollständig vermaschtes Netz entstanden ist.

Um das Broadcast-Modul von der Applikation möglichst unabhängig zu halten, wird die Hauptaufgabe folgendermaßen (informell) spezifiziert:

> „Versende einen Buffer der Länge n Bytes sicher und reihenfolgeerhaltend an eine angegebene Menge von Empfängern“

Wie dieser Datenbuffer gelesen bzw. beschrieben wird, ist an dieser Stelle völlig unerheblich und wird im nächsten Kapitel kurz erläutert. Das Broadcast-Modul ist also im wesentlichen von der Applikation, die es verwendet, unabhängig, einige Implementierungsgesichtspunke orientieren sich jedoch an der Team-Vervollständigung.

Eine Broadcast-Session zerfällt grundsätzlich in zwei Phasen:

(1) Vorbereitung

(2) Datentransfer

Diese beiden Phasen werden im Folgenden erläutert.

2.1 Vorbereitung zum Broadcast

Bei der Vorbereitung zum Broadcast muß man unterscheiden zwischen dem allerersten und allen folgenden Aufrufen der Vorbereitungsprozedur. Beim ersten Mal werden nämlich noch verschiedene Verwaltungsinformationen vom (initialen) Sender an die Empfänger übertragen, damit diese später ebenfalls in der Lage sind, als Sender tätig zu werden (siehe Kapitel 3.1). Abgesehen davon läuft die Vorbereitungsprozedur aber gleich ab:

Der Sender überprüft in seiner Liste mit den Broadcast-Teilnehmern, ob er bereits mit allen Empfängern verbunden ist. Ist dies nicht der Fall, so wartet er in mehreren `accept`-Aufrufen darauf, daß die Fehlenden sich an seinen Empfangsport anschließen. Die

Empfänger ihrerseits müssen irgendwie erfahren, welcher Host der aktuelle Broadcast-Versender ist und führen dann, sofern sie nicht bereits mit ihm verbunden sind, einen `connect`-Aufruf aus und identifizeren sich durch eine Nachricht. Die Information, welcher Host der aktuelle Sender ist, muß von der Applikation ermittelt werden. Bei der Team-Vervollständigung geschieht dies, wenn der Teamleiter wechselt.

Für einen Rechner gibt es zwei Möglichkeiten, mit allen anderen verbunden zu sein:

(1) er war bereits einmal Sender

(2) alle anderen waren schon Sender

Ist eine dieser Bedingungen erfüllt, werden keine weiteren Verbindungen mehr aufgebaut. Dies hat zur Konsequenz, daß die aufwendigen `connect`/`accept`-Aufrufe nur dann erfolgen, wenn sie unbedingt notwendig sind (siehe Abbildung 2.2). Da von Stream-Sockets sowohl gelesen als auch auf sie geschrieben werden kann, genügt eine Verbindung von einem Host A zu einem Host B, unabhängig davon, welcher von beiden `connect` bzw. `accept` ausgeführt hat (d. h. die Grenzen zwischen den Begriffen Client und Server verschwimmen). Bei der Team-Vervollständigung werden i. A. nicht soviele Rechner eingesetzt, daß die Systembeschränkungen bei der Anzahl der offenen Sockets überschritten werden.

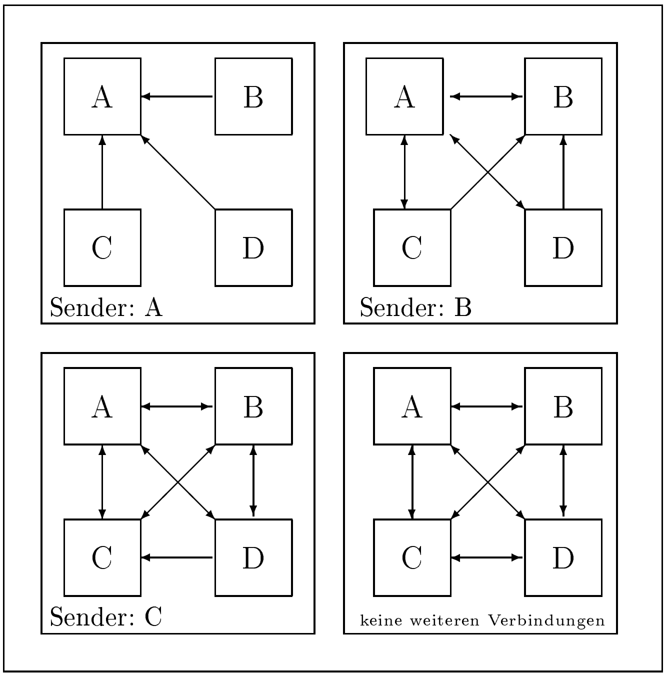

Abbildung 2.2: Vollständiger Verbindungsaufbau

Nachdem die Vorbereitungen abgeschloßen sind, können nun Daten übertragen werden, Sender und Empfänger müßen sich lediglich noch geeignet synchronisieren (Näheres siehe Kapitel 3.1).

Eine mögliche Verbesserung der momentanen Implementierung der Team-Vervollständigung wäre eine Umstellung der restlichen Kommunikation auf die permanenten Verbindungen. Bei der derzeitigen Lösung werden noch bei jedem Datenaustausch Sockets geöffnet bzw. geschlossen, was sehr zeitaufwendige Operationen erfordert.

2.2 Datenaustausch über Broadcasts

Wie bereits erwähnt, besteht das grundsätzliche Problem bei Broadcasts darin, die Datagramme gegen eventuelle Verluste, Duplizierungen etc. zu schützen. Abbildung 2.3 zeigt in einem Diagramm, wie die Übertragung eines beliebig großen Buffers vor sich geht. *PtP* steht für Point-to-Point, also Stream-Verbindungen, *br* bedeutet Broadcast-Datagramme.

Der Sender wartet zu Beginn des Datenaustauschs darauf, daß alle Empfänger ihre Aufnahmebereitschaft signalisiert haben. Fehlt auch nur ein einziger, so wird das Protokoll nach einer bestimmten Zeit (Timeout) abgebrochen. Bei stark abweichendem Zeitverhalten von Sender und Empfänger kann es vorkommen, daß die Wartezeit beim Sender abläuft und dieser daraufhin die Verbindung mit einer Fehlermeldung unterbricht. Ist von vorneherein klar, daß sich das Zeitverhalten der beteiligten Prozesse stark voneinander unterscheidet, sollte die Timeoutzeit nicht zu knapp gewählt werden. Man kann die Wartezeit auch auf unendlich setzen, d. h. der Sender wartet auf jeden Fall, bis ein `ReadyToReceive` von jedem Empfänger vorliegt. In diesem Fall besteht allerdings die Gefahr, daß ein Deadlock im Programm nicht erkannt wird und das System stehenbleibt.

Wenn alle Empfänger bereit sind, startet der Sender mit der Übertragung der ersten n Datenpakete. Das letzte Paket enthält außer den normalen Daten die Aufforderung an die Empfänger, diese n Pakete zu quittieren bzw. fehlende Pakete anzuzeigen. Ein Quittungsverfahren, das so arbeitet (d. h. die *fehlenden* Pakete anzeigt), wird als *negative-Acknowledgement* Verfahren bezeichnet. Der Sender empfängt nacheinander alle Quittungen und überträgt die fehlenden Pakete über die während der Vorbereitung aufgebauten Steam-Verbindungen erneut. Diese fehlenden Pakete werden von den Empfängern an der richtigen Stelle einsortiert, d. h. die Daten werden reihenfolgeerhaltend verschickt. Das Datenpaket, welches die Quittungsanforderung enthält, ist durch einen Timeout gegen einen Verlust geschützt, d. h. falls gerade dieses Paket verlorengehen sollte, wird dies an der fehlenden Quittung eines Hosts erkannt. Das Paket wird dann so oft erneut übertragen, wie die Konstante `ACKNREQUEST` es vorschreibt. Bleibt nach der letzten Wiederholung die Quittung immer noch aus, bricht das Protokoll ab. Durch eine eindeutige Broadcast-Nummer wird das duplizierte Paket von den Hosts, die es korrekt empfangen haben, erkannt und dann vernichtet.

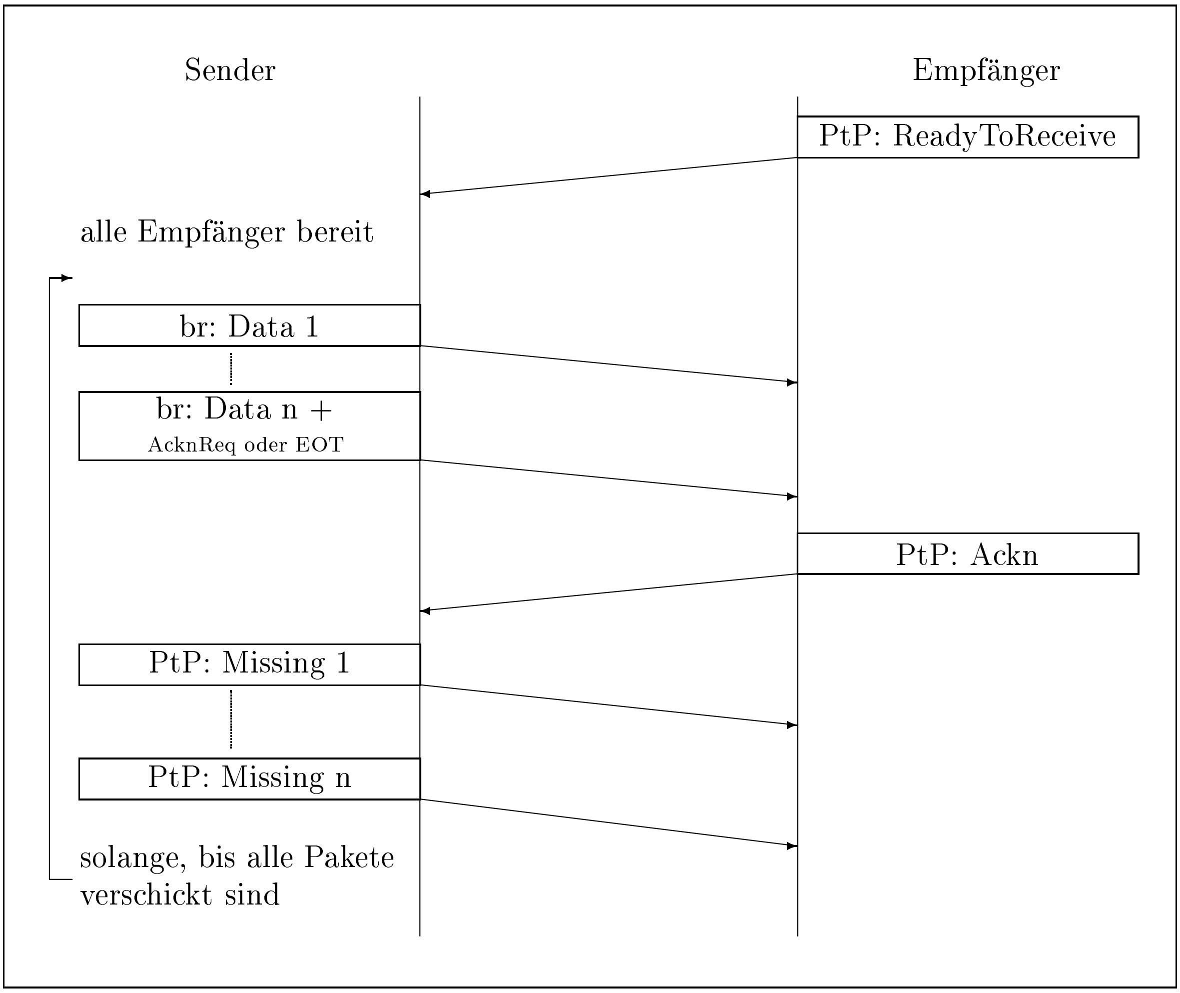

Abbildung 2.3: Das Protokoll

Beim letzten Datagramm einer Broadcast-Session wird statt der normalen Quittungsanforderung `SendAckn` der Befehl `EndOfTransmission` mitgeschickt. Dieser Opcode impliziert eine Quittungsanforderung und zeigt gleichzeitig das Ende der Datenübertragung an.

Der Datenaustausch auf Applikationsebene erfolgt nach gängigen Definitionen asynchron. Dies bedeutet, daß das Kommunikationssubsystem Nachrichten puffert und damit die Abstimmung zwischen Sender und Empfänger übernimmt. Durch die Synchronisation über ein Rendezvous werden die Lese- bzw. Schreiboperationen auf dem Datenbuffer zu blockierenden Aufrufen. Dies ist bei der Entwicklung von zeitabhängigen Applikationen zu beachten.

Wie bereits angesprochen, bricht das Protokoll bei einer fehlenden Bereitmeldung oder Quittung mit einer Fehlermeldung ab. Dies kann durch eine geringfügige Änderung im Programm verhindert werden; der Sender wartet dann solange, bis sich alle Broadcastteilnehmer gemeldet haben (siehe 1.1, der `select`-Aufruf). Ob zu dieser Möglichkeit gegriffen werden kann, hängt von der Problemstellung ab. Wird nämlich die Wartezeit auf „unendlich lange" gesetzt, so kann der Sender nicht erkennen, ob alle anderen Prozesse noch ablaufen oder aber abgestürzt sind. Es ist also nicht mehr möglich, einen Deadlock zu erkennen.

Bei der Team-Vervollständigung spielt dieses Problem eine untergeordnete Rolle, das System kann ohnehin nicht weiterarbeiten, wenn ein Teilnehmer ausgefallen ist. Bei Systemen, die mehr Fehlertoleranz erforden und die nach dem Ausfall einer Komponente noch in der Lage sind weiterzuarbeiten, sollte auf die Möglichkeit des „unendlichen" Timeouts verzichtet werden.

Kapitel 3

Die Implementierung

In diesem Kapitel werden einige Implementierungsdetails der vorliegenden Arbeit dargestellt und kommentiert. Neben dem Modulzusammenhang (Aufrufreferenzen) werden vor allem die zentralen Prozeduren zur Realisierung des Protokolls genauer beschrieben. Die nachfolgenden Ausführungen sollen in erster Linie dazu dienen, das Broadcast-System für Dritte besser verständlich und änder- bzw. wartbar zu machen. Falls es Punkte gibt, an denen eventuell noch Verbesserungen vorgenommen werden könnten, wird dies an der entsprechenden Stelle erläutert. Grundsätzlich ist jedoch zu sagen, daß ich mich um einen möglichst guten Kompromiß zwischen Effizienz und Überschaubarkeit bemüht habe. Trotz des starken Zuschnitts auf die Team-Vervollständigung ist das System noch flexibel genug, um problemlos an andere Aufgaben angepaßt werden zu können.

3.1 Der Modulzusammenhang

In Abbildung 3.1 ist dargestellt, wie die einzelnen Module der Broadcast-Einheit zusammenhängen. Die Komponenten *br_time* und *br_stat* werden ausgelassen, sie werden nur zu Studienzwecken genutzt und würden die Abbildung unnötig komplizieren.

Man erkennt drei zentrale Einheiten: *br_send, br_recv* und *br_util*. *br_send* stellt alle Funktionen zur Verfügung, die ein Host benötigt um als Sender tätig zu werden. Außerdem werden in *br_send* alle globalen Variablen (in den anderen Modulen als `extern` gekennzeichnet) vereinbart. Analog zu *br_send* stellt *br_recv* alle Prozeduren bereit, die zum Empfangen von Broadcasts notwendig sind. Sowohl das Sende- als auch das Empfangsmodul enthalten die für den Datenbufferzugriff benötigten Lese (`br_ReadXX`) und Schreiboperationen (`br_WriteXX`).

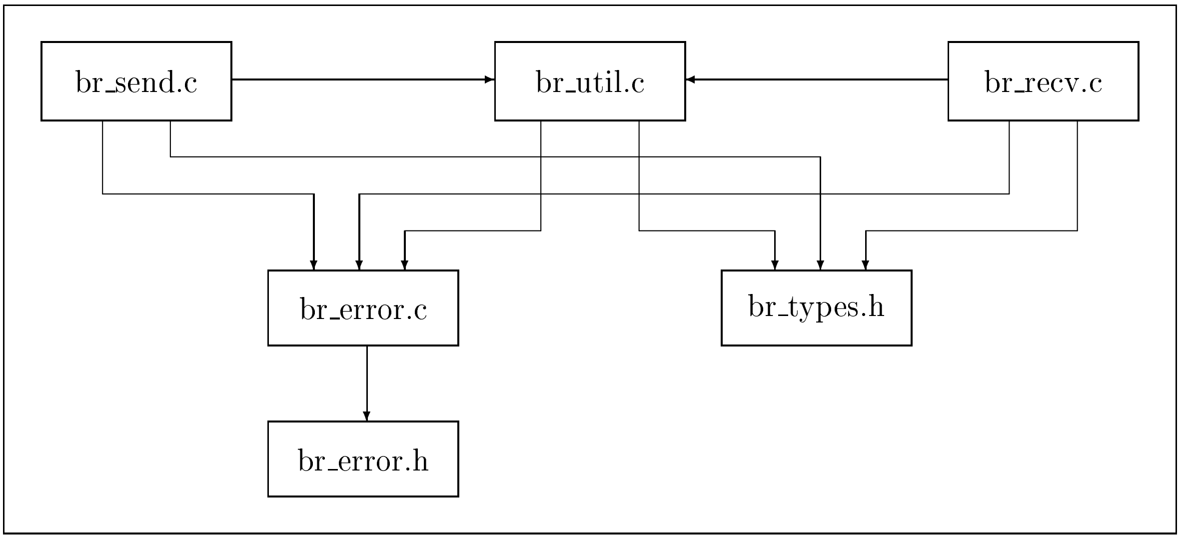

Abbildung 3.1: Der Modulzusammenhang

br_util schließlich enthält Funktionen, die von Sender und Empfänger gleichermaßen benutzt werden, so z. B. das Lesen und Schreiben auf Streams, das Ermitteln der Broadcast-Adresse des lokalen Netzes, etc.

Eine Hierarchiestufe unter diesen Moduln liegen *br_error* und *br_types*. *br_types* enthält alle Datentypen und Konstanten, die von mehr als einem Modul genutzt werden, z. B. den zentralen Datentyp `tMessage`. *br_error* dient zur Behandlung von Fehlern, unter anderem auch zur Ausgabe der entsprechenden Fehlermeldungen, die in Form eines String-Arrays in *br_error.h* bereitgestellt werden.

3.2 Das Modul *br_types*

In diesem Modul werden alle globalen Konstanten und Datentypen vereinbart. Die wichtigsten von ihnen werden in diesem Abschnitt vorgestellt.

Die interessantesten Konstanten sind diejenigen, mit denen das Broadcast-Protokoll konfiguriert werden kann. `ACKNnPACKETS` legt fest, wieviele Pakete jeweils zusammen quittiert werden sollen, dies geschieht `TIMES` mal, also werden pro Protokolldurchlauf `ACKNnPACKETS * TIMES` (= `MAXPACKETS`) Datenpakete übertragen.

Da die maximale Größe eines Datagramms auf 1472 Bytes beschränkt ist, kann der reine Datenbereich einer Nachricht nur 1400 Bytes (`PACKETSIZE`) groß sein, der Rest der

Nachricht wird zur Aufnahme von Verwaltungsinformationen benötigt (siehe `tMessage`).

Aus diesen Informationen ergibt sich nun auch die Größe des Datenbuffers, der vom Programm beschrieben bzw. gelesen werden kann: `DATABUFSIZE = PACKETSIZE * MAXPACKETS`.

Die Konstante `ACKNREQUESTS` legt fest, wie oft eine verlorengegangene Quittungsanforderung wiederholt wird, bevor das Protokoll abbricht. `TIMEOUTLENGTH` ist die Länge des Timeouts, mit dem auf das Eintreffen einer Quittung gewartet wird, bevor sie wiederholt wird. Die Länge des Timeouts wird in Sekunden angegeben.

Die zentrale Datenstruktur des gesamten Protokolls ist `tMessage`. Alle Nachrichten, egal ob über Streams oder durch Datagramme verschickt, haben dieses einheitliche Format:

```
typedef enum{
    DataPacket,
    MissingPacket,
    AcknPacket,
    CommunicationMessage
} tPacketType;

typedef enum{
    OK,
    NotOK,
    Nop,
    SendAckn,
    ReadyToReceive,
    EndOfTransmission
} tOpcode;

typedef struct msg{
    tPacketType     Type;
    tOpcode         Opcode;
    union cnt{
        struct dtapkt{
                    int       BroadcastNumber;
                    int       PacketNumber;
```

```
            int      DataEnd;
            char     Data[PACKETSIZE];
        } DataPacket;

        struct msgpkt{
            int      BroadcastNumber;
            int      PacketNumber;
            char     Data[PACKETSIZE];
        } MissingPacket;

        struct acknpkt{
            int      BroadcastNumber;
            int      PacketsMissing[ACKNnPACKETS+1];
        } AcknPacket;

        struct commmsg{
            int      nReceivers;
            int      MissingConnections;
            int      Pid;
            tString  Hosts[MAXRECEIVERS];
            int      Ports[MAXRECEIVERS];
        } CommunicationMessage;
    } Contents;
} tMessage;
```

Die Strukturkomponente `Opcode`, die mit Werten aus `tOpcode` belegt werden kann, dürfte selbsterklärend sein, wichtig sind an dieser Stelle vor allem die verschiedenen Nachrichtentypen (Komponente `Type`). Zur Zeit gibt es deren vier:

- normale Datenpakete: `DataPacket`. Neben der aktuellen Broadcast- bzw. Paketnummer enthält dieser Typ noch eine Komponente `DataEnd`. Sie wird genau dann benötigt, wenn ein nicht ganz gefüllter Datenbuffer übertragen wurde (ausgelöst durch `br_FlushBuffer`). Der Wert dieser Komponente ist dann die Position des letzten Datenbytes. Diese wird beim Empfänger benötigt, um zu erkennen, ob der Buffer vollständig gelesen wurde.

- fehlende Pakete: `MissingPacket`. Dieser Typ entspricht (bis auf `DataEnd`) genau dem eines Datenpakets. Die Unterscheidung zwischen normalen Datenpaketen und fehlenden Paketen ist im Grunde bedeutungslos, im Programmlisting wird dadurch allerdings deutlicher, welche Phase des Protokolls gerade abgearbeitet wird (siehe Abbildung 2.3).

- Quittungspakete: `AcknPacket`. Diese Pakete enthalten Quittungen für jeweils `ACKNnPACKETS` Datenpakete. Im ersten Element des Arrays steht die Anzahl der fehlenden Pakete, daraus folgt auch, daß das Array die Länge `ACKNnPACKETS + 1` haben muß.

- Verwaltungsnachrichten: `CommunicationMessage`. Immer wenn Informationen übertragen werden, die das Protokoll selbst betreffen, wird auf diesen Nachrichtentyp zurückgegriffen. Dies ist z. B. beim Aufbau des vollständig vermaschten Streamverbindungs-Graphen (siehe Abbildung 2.2) der Fall.

Die letzte interessante Datenstruktur in diesem Modul ist `tReceiver`:

```
typedef struct recver{
    tString  Host;
    int      Sock;
    tbool    Ready;
    tbool    Ackned;
} tReceiver;
```

Jeder Host hält eine Liste mit den Teilnehmern am Broadcasting. In dieser Liste sind zunächst die Namen und Sockets, über die die jeweiligen Host zu erreichen sind, gespeichert. Die Komponenten `Ready` und `Ackned` enthalten Informationen darüber, ob einzelne Hosts bereit zum Datenempfang sind, bzw. ob sie die empfangenen Daten bereits quittiert haben (siehe `br_FlushBuffer` und `br_DoAcknoledge`).

3.3 Das Modul *br_error*

Das Modul *br_error* stellt die Fehlerbehandlungsprozeduren der Broadcast-Einheit zur Verfügung :

```
int br_Error ( text, code )
     tString     text;
     tErrorcode  code;
```

setzt lediglich die beiden globalen Variablen *br_Errorcode* und *br_Errortext*, diese beiden Werte werden durch den Aufruf

```
int br_PrintError ()
```

ausgegeben. Die Funktion hat keine Parameter und erzeugt eine Ausgabe der Form

br_Errortext : *br_ErrorMsg[br_ErrorCode]*

`Systemmeldung` : *perror()*

Anschließend ruft sie eine Funktion auf, die die Applikation, die das Broadcast-Modul benutzt, ordnungsgemäß beendet. Bei der Team-Vervollständigung ist dies die Funktion `TerminateTeam ()`.

3.4 Das Modul *br_util*

In *br_util* sind diejenigen Prozeduren zusammengefaßt, die von Sender und Empfänger gleichermaßen benutzt werden. Die Prozeduren

```
int br_OpenAcceptsock ( ... )
int br_CloseAcceptsock ( ... )
int br_ConnectPtP ( ... )
int br_AcceptPtp ( ... )
int br_SendPtP ( ... )
int br_RecvPtP ( ... )
```

arbeiten im wesentlichen genau wie die im ersten Kapitel beschriebenen Prozeduren `Open`, `Close`, `Connect` etc. und brauchen daher nicht näher erläutert zu werden. Sie bieten lediglich etwas komfortablere Schnittstellen zu den entsprechenden Systemfunktionen. Mit `br_OpenAcceptsock ( ... )` öffnen alle Prozesse einen Empfangssocket, an den sich die anderen Broadcast-Teilnehmer gegebenenfalls anschließen.

Eine der wichtigsten Funktionen des gesamten Broadcast-Moduls ist die Funktion, mit der die Broadcast-Adresse des Subnetzes ermittelt wird (`br_GetBroadcastAddress ()`). Um diese Adresse zu erhalten, muß die Netzadresse des Host, auf dem die Applikation läuft, festgestellt und auf bestimmte Art und Weise mit der Netzmaske des Subnetzes verknüpft werden (siehe vorletzte Zeile der Funktion).

Das größte Problem hierbei ist es, diese Netzmaske zu bestimmen: Zuerst wird mit einem `ioctl( ..., SIGGIFCONFIG, ... )` Aufruf eine Liste der Netzwerk-Adapter des Hosts angefordert. Im Allgemeinen hat ein Host nur einen solchen Adapter und ein sog. *Loopback*-Interface. Handelt es sich jedoch um ein Gateway, so können es durchaus auch mehrere Adapter sein. Man kann die Netzmaske des ersten Interfaces in der Liste verwenden, sofern es nicht um das Loopback-Interface handelt. Daß dies nicht der Fall ist, stellt der Vergleich (`strcmp( ifp->ifr.name, INTERFACENAME ) != 0`) sicher. `INTERFACENAME` wird in *br_types.h* auf den Wert des Loopbacks gesetzt (i. A. `lo0`), bei einer Portierung des Systems sollte *unbedingt* überprüft werden, ob dies auch der Loopback-Name des neuen Netzes ist. Der Rückgabewert der Funktion ist die Broadcast-Adresse des Subnetzes.

Die letzten drei Funktionen von *br_util* dienen der Verwaltung der internen Datenstrukturen des Broadcast-Moduls.

```
int br_GetHostId ( host )
     char*  host;
```

gibt den Index des Broadcastteilnehmers mit Namen `host` in der `Receivers`-Liste des aufrufenden Hosts zurück.

```
int br_GetConnection ( host )
     char*  host;
```

liefert die Socketnummer, über die `host` erreicht werden kann.

```
int br_SetMaster ( newmaster )
     char*   newmaster;
```

überprüft zunächst, ob ein Broadcastteilnehmer `newmaster` überhaupt existiert. Ist dies der Fall, so wird die globale Variable `br_Master` (= der aktuelle Sender) auf diesen Wert gesetzt.

3.5 Die Module *br_send* und *br_recv*

Diese beiden Module realisieren das Protokoll wie es in Kapitel 2 beschrieben ist. Da sie unmittelbar miteinander kommunizieren, ist es sinnvoll, sie in einem einzigen Unterkapitel zu behandeln.

Bevor der Datenaustausch über Broadcasts überhaupt beginnen kann, sind jedoch einige Vorbereitungen zu treffen. Zunächst einmal muß die Applikation, welche das Broadcast-System nutzen will, die Prozedur

```
int br_ConfigureBroadcast ( hosts, num )
     char    hosts[MAXEXPERT][MAXHOSTNAMELEN];
     int     num;
```

aufrufen. Der erste Parameter wurde so gewählt, daß das Datenformat, das bei der Team-Vervollständigung für die benötigten Informationen verwendet wird, unmittelbar übernommen werden kann.

Durch den Aufruf dieser Prozedur werden dem Broadcast-System die Namen und die Anzahl der teilnehmenden Hosts bekannt gemacht. Außerdem wählt der initiale Sender (nur dieser ruft die Prozedur auf) hier einen (zufälligen) Broadcast-Port. Dieser Port wird den anderen Teilnahmern per Kommandozeilenparameter bekannt gemacht.

Die beiden Prozeduren

```
int br_OpenSendBroadcast ()
```

und

```
int br_OpenRecvBroadcast ()
```

werden bei jedem Wechsel des Senders aufgerufen und führen dann bestimmte Initialisierungen durch. Außerdem bauen sie den vollständig vermaschten Verbindungsgraphen auf (vgl. 2.1).

Abhängig vom Zustand des Broadcast-Systems werden unterschiedliche Anweisungsblöcke abgearbeitet:

- *br_FirstTime* == TRUE
  Dieser Anweisungsteil wird nur beim ersten Aufruf der Prozeduren bearbeitet, unabhängig davon, ob die Vorbereitungsprozedur als Sender oder als Empfänger aufgerufen wurde. Das bedeutet, daß, wenn ein Prozeß z. B. vom Sender zu Empfänger wird, dieser Teil nicht nocheinmal abgearbeitet wird, auch wenn der Prozeß nun zum ersten mal die Rolle eines Empfängers übernommen hat.

  Der initiale Sender wartet in einer Schleife darauf, daß sich alle Empfänger an seinen Accept-Port anschließen und sich durch eine Nachricht identifizieren. Die Portnummer dieses Accept-Ports wird den Empfängern beim Starten des Teams als Kommandozeilenparameter mitgegeben. Nachdem der Sender sich so einen Überblick verschafft hat, ob alle Empfängerprozesse ordnungsgemäß gestartet wurden, füllt er eine Nachricht mit Verwaltungsinformationen und schickt sie der Reihe nach an alle Empfänger. Diese Nachricht enthält zum Beispiel die Namen aller beteiligten Rechner, deren Anzahl etc.

  Ein Problem beim Broadcasting in einem Netzwerk ist, daß auf allen Rechnern, die die Nachrichten erhalten sollen, die gleiche Portnummer geöffnet werden muß. Aus diesem Grund wird an dieser Stelle noch ein kleines Protokoll abgewickelt, das sicherstellt, daß ein gemeinsamer Port auf allen Rechner erfolgreich geöffnet werden konnte. Der Sender erhält zunächst von allen Empfänger eine Nachricht mit der sie bekanntgeben, ob der Port, den sie über die Kommandozeile bekommen haben, geöffnet werden konnte. War diese Portnummer bei einem oder mehreren Rechnern schon belegt, so wählt der Sender eine neue Portnummer und versucht zunächst selbst diesen Port zu öffnen. Wenn dies fehlerlos geschehen ist, schickt er die Nummer an alle Empfänger und wartet wieder auf die Erfolgsmeldungen. Dies wiederholt sich solange, bis auf allen Rechner ein gemeinsamer Port geöffnet werden konnte.

- *br_FullConnected* == FALSE
  Dieser Fall tritt immer dann ein, wenn der Sender wechselt und der neue Sender

noch nicht mit allen anderen Hosts verbunden ist. Jeder Prozeß hat eine Variable *br_MissingConnections*, welche die Anzahl der noch fehlenden Verbindungen enthält. Anhand dieser Information weiß der neue Sender, wieviele `accept`-Aufrufe er noch ausführen muß, um mit allen Empfängern verbunden zu sein.

  Die Empfänger prüfen zunächst, ob schon eine Verbindung zum neuen Sender besteht. Ist dies nicht der Fall, so erzeugen sie einen neuen Socket und schließen sich mittels `connect` an den Sender an.

- immer
  Bestimmte Initialisierungen müssen bei jedem Senderwechsel vorgenommen werden, z. B. wird an dieser Stelle der Datenbufferzeiger auf den Anfang des Buffers gesetzt.

Das in Kapitel 2 beschriebene Protokoll wird in vier Prozeduren realisiert. Hierbei sind

```
int br_FlushBuffer ()
```

und

```
int GetBuffer ()
```

die Hauptprozeduren, das Quittieren ist nach

```
int HandleAckn ()
```

bzw.

```
int DoAcknowledge ( ackn )
     tMessage ackn
```

ausgelagert.

Bevor der Sender jedoch mit dem Verschicken der Broadcast-Nachrichten beginnen kann, muß er sich vergewissern, daß alle Empfänger zur Datenaufnahme bereit sind. Dies geschieht in der Prozedur

```
tbool ReceiversReady ()
```

Der Sender wartet in einem `select`-Aufruf (siehe Kapitel 1) an allen Empfängersockets auf die Bereitmeldungen. Der Timeout wird nach der Abarbeitung einer Bereitmeldung immer neu gesetzt, d.h. die Prozedur kehrt mit einer Fehlermeldung zurück, wenn noch Bereitmeldungen ausstehen und sich *timeout* Sekunden nach der letzten empfangenen Meldung kein Empfänger mehr meldet. Konnte `ReceiversReady ()` ordnungsgemäß beendet werden, so beginnt der Sender nun mit dem Aussenden der Datenpakete. Nachdem er die Daten aus dem Sendebuffer in die Nachricht kopiert hat, prüft er, ob eine besondere Situation innerhalb des Protokolls vorliegt. Besondere Situationen können sein:

- Der Buffer ist vollständig übertragen ($pos \geq DatabufferPos$). In diesem Fall wird als Opcode `EndOfTransmission` (mit impliziter Quittungsanforderung) gesetzt und die Hauptschleife kann, nach Abarbeitung der Quittierungsprozedur, verlassen werden (*exit* `= TRUE`). Mit *reqackn* `= TRUE` wird angezeigt, daß nach dem Abschicken des aktuellen Pakets eine Quittierungsphase folgt.
- es wurden `ACKNnPACKETS` Pakete übertragen (`((`*packno* `+ 1) % ACKNnPACKETS) == 0`) Als Opcode wird in diesem Fall `SendAckn` mitgeschickt, der Ablauf ist derselbe wie oben, nach der Quittierungsphase wird allerdings die Hauptschleife nicht verlassen.

Der Empfang der Datenpakete ist nicht ganz so einfach. Den Empfängern obliegt es nämlich, die Sequenznummer der erhaltenen Pakete zu kontrollieren und dieser Nummer entsprechende Maßnahmen zu ergreifen. Nachdem ein Empfänger ein Paket erhalten hat, gibt es drei Möglichkeiten, was zu tun ist:

- Die Sequenznummer ist korrekt (d. h. die erwartete): Die Daten werden an der entsprechenden Stelle in den Empfangsbuffer kopiert
- Die Sequenznummer des Pakets ist größer als die erwartete. In diesem Fall werden alle Pakete, die zwischen der erhaltenen und der erwarteten Sequenznummer liegen als fehlend gekennzeichnet und die Anzahl der fehlenden Pakete entsprechend erhöht (*ackn.Contents.AcknPacket.MissingPackets[0]*). Anschließend werden die erhaltenen Daten an der richtigen Stelle (die sich anhand der Paketnummer ermitteln läßt) in den Buffer einsortiert und die erwartete Sequenznummer auf den neuen Wert (= erhaltener Wert + 1) gesetzt. Falls zu einem späteren Zeitpunkt Pakete eintreffen, die bereits als fehlend gekennzeichnet sind, so werden sie ignoriert (s. u.). Daß dies zulässig ist, ohne einen Leistungsverlust durch doppelt übertragene Pakete zu

riskieren, ergibt sich aus der Tatsache, daß während der intensiven Testphase des Protokolls kein Fall beobachtet wurde, in dem ein Paket tatsächlich zu spät eintraf.

- Die Sequenznummer ist kleiner als die erwartete: Das Paket wird ignoriert (s. o.). Dieser Fall tritt insbesondere immer dann ein, wenn ein oder mehrere Empfänger das letzte Paket einer Sequenz nicht erhalten haben. Da dieses Paket die Quittungsanforderung enthält, wissen diese Empfänger nicht, daß sie jetzt quittieren sollen, folglich wartet der Sender vergeblich auf die Quittungen. Nach einer bestimmten Zeit sendet er das letzte Paket erneut über das Netz, allerdings dürfen die Empfänger, die es korrekt erhalten haben natürlich keine Quittung mehr abschicken. Dies wird dadurch gewährleistet, daß sie doppelt erhaltene Pakete ignorieren.

Nach jeweils `ACKNnPACKETS` Datenpaketen (bzw. falls der Bufferinhalt mit weniger Paketen vollständig geleert werden kann) folgt im Rahmen des Austauschprotokolls eine Quittierungsphase. Die Empfänger bereiten während der Empfangsphase die Quittungen bereits so vor, daß sie direkt abgeschickt werden können. *ackn.Contents.AcknPacket.MissingPackets[0]* enthält die Anzahl der fehlenden Pakete, deren Nummer sind in den nachfolgenden Feldelementen abgelegt.

Der Sender wartet in einem `select`-Aufruf jeweils auf die Quittung eines Empfängers und überträgt, nachdem er diese erhalten hat, die fehlenden Datenpakete erneut. Da jede Paketnummer für den gesamten Datenbuffer eindeutig ist, kann der Sender aus ihr errechnen, an welcher Stelle im Buffer die fehlenden Daten zu finden sind (Paketnummer * `PACKETSIZE`).

Falls nach `TIMEOUT` (siehe *br_types.h*) Sekunden keine Quittung von einem Empfänger vorliegt, kehrt `HandleAckn ()` mit dem Wert `FALSE` zurück. In `br_FlushBuffer ()` wird in diesem Fall geprüft, ob das letzte Paket bereits `ACKNREQUESTS`-mal übertragen wurde. Ist dies der Fall, so bricht das Protokoll mit einer Fehlermeldung ab, ansonsten wird dieses letzte Paket erneut abgeschickt. Die Quittierungsprozedur fährt beim erneuten Aufruf mit dem Empfänger fort, der nicht rechtzeitig quittiert hat.

Die Aufgabe der Quittierungsprozedur auf Empfängerseite, `DoAcknowledge(` *ackn* `)`, ist wesentlich einfacher. Hier wird lediglich die vorbereitete Quittung abgeschickt und die empfangenen Daten werden an der richtigen Stelle (s. o.) in den Buffer einsortiert.

Zum Schluß dieses Kapitels soll noch schematisch dargestellt werden, wie die Prozeduren zum Schreiben bzw. Lesen des Datenbuffer aussehen. Zur Zeit werden folgende Prozeduren zur Verfügung gestellt:

```
int br_WriteInt ( val )
     int     val
int br_WriteData ( buf, len )
     char*   buf
     int     len
int br_ReadInt ( val )
     int*    val
int br_ReadData ( buf, len )
     char*   buf
     int     len
```

Anhand der weiter unten angegebenen Prozedurskelette dürfte es aber einfach sein, eigene Prozeduren für andere Datentypen zu entwickeln.

In der Schreibprozedur ist zunächst zu prüfen, ob der verbleibende Bufferplatz noch ausreicht, um die Daten aufzunehmen. Ist dies nicht der Fall, so muß zuerst mit `br_FlushBuffer ()` der vorhandene Bufferinhalt an die Empfänger verschickt werden. Analog arbeitet die Leseprozedur: Auch hier wird zunächst geprüft, ob noch genügend Daten im Buffer stehen, um die Anforderung zu erfüllen. Falls nicht, wird der Buffer durch `GetBuffer ()` neu gefüllt. Bei dieser Arbeitsweise wird vorausgesetzt, daß Sender und Empfänger Schreib- bzw. Leseoperationen symmetrisch ausführen, dies ist bei einer datenorientierten Kommunikation normalerweise auch immer der Fall.

Hier nun noch die Prozedurrümpfe:

```
void br_WriteXX ( data )
        datatype*data;
{
        int     len;
        len = sizeof( data );
        if( (DatabufferPos + len) < DATABUFSIZE ){
                bcopy( data, &(Databuffer[DatabufferPos]), len );
                DatabufferPos += len;
        }
        else{
                if( br_FlushBuffer () == -1 )
```

```
                        br_PrintError ();
                bcopy( data, &(Databuffer[DatabufferPos]), len );
                DatabufferPos += len;
        };
};

void br_ReadXX ( data )
        datatype* data;
{
        int     len;
        len = sizeof( datatype );
        if( (DatabufferPos + len) > Bufferend ){
                if( Getbuffer () == -1 )
                        br_PrintError ();
                bcopy( &(Databuffer[DatabufferPos]), data, len );
                DatabufferPos += len;
        }
        else{
                bcopy( &(Databuffer[DatabufferPos]), data, len );
                DatabufferPos += len;
        };
};
```

Bei der Verwendung der Schreib/Lese-Operationen in einer Applikation ist zu beachten, daß am Ende einer Datenübertragungsphase die Prozedur `br_FlushBuffer( ... )` explizit aufgerufen werden muß. Anderenfalls wird ein nur teilweise gefüllter Buffer nicht mehr ordnungsgemäß an die Empfänger verschickt und diese warten erfolglos (und unendlich lange) auf die fehlenden Daten.

Kapitel 4

Das Protokoll im Einsatz

4.1 Die Testapplikation

Im Rahmen dieser Arbeit wurde eine Applikation entwickelt, die es gestattet, Leistungsbetrachtungen für das Broadcast-System anzustellen. Diese Testapplikation ist sehr einfach gehalten: Sie überträgt zunächst eine größere Datenmenge (ca. 1,6 MByte) via Broadcast und erlaubt dem Anwender dann die Auswahl eines neuen Senders. Anschließend beginnt der Zyklus von vorne. Das Modul *br_statistics*, welches hier nicht näher beschrieben ist, liefert dann Daten über das Broadcasting. Zum Beispiel wird festgehalten, wieviel Zeit das Programm im Broadcast-Subsystem verbracht hat, wieviele Pakete verlorengingen, etc. Das Mitführen einer Statistic kann jederzeit durch das Einfügen der Anweisung

```
#define STATISTICS
```

in *br_types.h* aktiviert werden. Durch den Befehl `stat_PrintStatistic(` *filename* `)` am Ende eines Programms wird dann die Broadcast-Statistik für den jeweiligen Rechner nach *filename* geschrieben. Hat *filename* den Wert `NULL`, so geht die Ausgabe nach stdout.

Das Netzwerk, welches zum Testen des Broadcast-Systems verwendet wurde, besteht aus Rechnern der Typen SUN Sparc SLC, SUN Sparc ELC, SUN Server und SUN 3/60. Im Laufe der verschiedenen Testreihen hat sich eine Tatsache besonders deutlich herausgestellt: Es ist nicht sinnvoll, Broadcasting in einer heterogenen Menge von Rechnern durchzuführen. Heterogen ist in diesem Fall dahingehend zu verstehen, daß die Rechenleistungen (Geschwindigkeiten) der einzelnen Rechner stark differieren. In einer Konfiguration mit SUN ELC's und SUN 3/60'ern ist ein Paketverlust von 70 % eher die Regel als die Ausnahme (bei SUN ELC als Sender). D. h. mehr als 2/3 der gesamten Datenmenge müssen mehr als einmal über das Netz geschickt werden. Der Vorteil, der durch das Broadcasting erzielt werden sollte, wird praktisch vollständig wieder aufgehoben, durch den zusätzlichen Verwaltungsaufwand durch das Protokoll sinkt die Leistung insgesamt wahrscheinlich noch unter die reiner Stream-Verbindungen.

Bei einer homogenen Konfiguration liegen die Dinge völlig anders. Hier zeigt sich, daß auch Datagramme (zumindest innerhalb eines LAN's) eine relativ hohe Ausfallsicherheit haben. Der Datenverlust liegt i. A. unter 10%, ein Wert, der durchaus toleriert werden kann. Daß durch den Einsatz von Broadcast-Nachrichten wirklich Leistungssteigerungen erzielt werden, läßt sich mit entsprechenden Messungen nachweisen (siehe 4.2.3). Die Zeit, die benötigt wird, um eine Nachricht mittels Broadcasting an *alle* Empfänger zu schicken, liegt nur knapp über der, die Stream-Verbindungen für die Übertragung an *einen* Empfänger brauchen.

4.2 Die Team-Vervollständigung mit Broadcasting

4.2.1 Was ist die Teamwork-Methode?

Die Teamwork-Methode wurde von J. Denzinger (vgl. [De91], [De93]) an der Universität Kaiserslautern zur verteilten Problemlösung, insbesondere im Bereich der Reduktionssysteme, entwickelt. Ausgangspunkt ist die Vorstellung von einer Gruppe von Menschen, die gemeinsam ein gegebenes Problem lösen wollen.

Ein Team im Sinne der Teamwork-Methode ist eine Menge von *Experten*, die an der Lösung des Problems arbeiten; eine Menge von *Gutachtern*, welche die Arbeit der Experten nach jedem Arbeitszyklus beurteilen und ein *Teamleiter*, der die Arbeit der Experten koordiniert (z. B. durch Auswahl geeigneter Zwischenergebnisse). Des weiteren gibt es noch sog. *Spezialisten*, die nicht direkt an der eigentlichen Problemlösung arbeiten, sondern lediglich den Suchraum verkleinern.

Bei der Übertragung des Modells auf einen Rechner werden die einzelnen Teammitglieder durch Prozesse dargestellt, die je nach Zustand des Systems verschiedene Rollen übernehmen (Experte, Gutachter, Teamleiter). Bemerkenswert an der Team-Vervollständigung ist, daß ein Problem nicht in Teilprobleme aufgesplittet wird, sondern daß die einzelnen Experten mit verschiedenen Strategien am Gesamtproblem arbeiten. Am Ende jeder

Arbeitsphase werden dann alle Zwischenergebnisse vom Teamleiter zu einem neuen, gemeinsamen Ausgangspunkt zusammengefaßt. Anschließend wird dieses gemeinsame Zwischenergebnis an alle Teammitglieder verschickt.

Neben dem Zusammenstellen eines möglichst guten Zwischenergebnisses hat der Teamleiter noch andere Aufgaben zu erledigen. Die Teamwork-Methode bezieht einen großen Teil ihrer Flexibilität aus der Tatsache, daß ein bestehendes Team am Ende jeder Arbeitsphase umbesetzt werden kann, d. h. ein Experte kann gegen einen anderen ausgetauscht werden. Ausgehend von den Bewertungen der Gutachter wählt der Teamleiter diejenigen Experten, die über mehrere Arbeitszyklen nur geringe Fortschritte gemacht haben, aus und ersetzt sie durch andere. In Zusammenhang mit der Rolle des Teamleiters ist zu beachten, daß diese nicht statisch ist. Am Ende einer Arbeitsphase wird derjenige Experte zum neuen Teamleiter, der das beste Zwischenergebnis errechnet hat. Dadurch müssen die Daten nicht vom Experten zum Teamleiter und dann an alle anderen Experten verschickt werden, man spart die Übertragung also einmal ein.

Ein Engpass bei der Team-Vervollständigung läßt sich relativ leicht identifizieren: Zu Beginn einer jeden Arbeitsphase müssen (abhängig von der Problemgröße) relativ große Datenmengen an *alle* Experten geschickt werden. Geschieht dies nacheinander (über Stream-Verbindungen), so kehrt sich der Vorteil der Verteilung rasch in einen Nachteil. Schon bei nur drei Prozessen sind (bei kleineren bis mittleren Problemen) die Zeiten, die zur eigentlichen Berechnung benötigt werden, kleiner als die Zeiten der Datenübertragung. Ein Faktor, nämlich die Kommunikation, die mit der eigentlichen Methode nichts zu tun hat, bestimmt also die Effektivität entscheidend mit. Dies sollte aber nach Möglichkeit vermieden, bzw. so stark wie möglich reduziert werden.

4.2.2 Einbindung des Broadcasting

Bei der ersten Version des DISCOUNT-Systems (siehe [Pi92]) stand die grundsätzliche Realisierung der Team-Vervollständigung, d. h. insbesondere die Verteilung auf verschiedene Rechner im Vordergrund. Der Schwerpunkt dieser Arbeit liegt nun darin, die vorhandene Kommunikationsstruktur zu verbessern und durch die Verwendung von Broadcasting Leistungssteigerungen, speziell bei großen Problemen, zu erzielen. An der Implementierung von W. Pitz hat sich nur wenig verändert, aus diesem Grund möchte ich darauf auch nicht mehr näher eingehen.

Da das Broadcast-System von der Team-Vervollständigung weitestgehend unabhängig gehalten ist, sind keine großen Änderungen notwendig, um die Erweiterung in das bestehende System einzubinden. Durch die Gliederung des Gesamtsystems in verschiedene Abstraktionsstufen müssen die Änderungen nur auf der höchsten und auf der niedrigsten Stufe stattfinden, die dazwischenliegenden bleiben unangetastet.

Nach dem Programmstart ruft der initiale Sender die Vorbereitungsprozedur `br_ConfigureBroadcast( ... )` auf und macht dem Broadcast-System die teilnehmenden Rechner bekannt. Danach öffnet jeder Host mit `br_OpenAccceptSock( ... )` seinen Empfangssocket und übermittelt die Portnummer des Sockets an den Sender. Dieses geschieht durch den Aufruf von `br_OpenRecvBroadcast( ... )` (Empfänger) und `br_OpenSendBroadcast( ... )` (Sender). Damit sind bereits alle Vorbereitungen zum Versenden bzw. Empfangen von Broadcasts abgeschloßen. Es ist allerdings zu beachten, daß die Aufrufe der `br_Open`... Prozeduren bei jedem Wechsel des Teamleiters erneut durchgeführt werden müssen. Dies geschieht in `StartCompletion`.

Um nun Daten mittels Broadcast senden bzw. empfangen zu können, ruft die Applikation wie gewohnt die Prozeduren `ReadInt( ... )`, `WriteInt( ... )` etc. auf (siehe *socket.c*). Allerdings wird nun der Socketdeskriptor `BROADCAST`, der in *socket.h* vereinbart ist, verwendet. Die Grundprozeduren erkennen daran, ob eine normale Datenübertragung oder ein Broadcast gewünscht ist und reagieren entsprechend. Durch dieses Vorgehen ist es möglich, alle Prozeduren, die auf den Zwischenebenen liegen (wie z. B. `SendCriticalPairs( ... )`), unangetastet zu lassen. Die Anweisung, wie letztendlich mit den Daten zu verfahren ist, wird von ganz oben nach ganz unten durchgereicht.

Ein Problem bei der Einbindung des Broadcastings lag darin, zu erkennen, an welchen Stellen im Programm Daten an alle Empfänger verschickt werden und dann für jede Stelle zu entscheiden, ob es überhaupt sinnvoll ist, zu broadcasten. Es lohnt sich nicht, geringe Datenmengen mittels Broadcast zu versenden, da der Overhead des Protokolls sonst zu groß ist. Nach einer eingehenden Analyse des Programms und verschiedener Ablaufprotokolle hat sich gezeigt, daß die Verwendung von Broadcasts an drei Stellen sinnvoll ist:

- bei der Versendung der kritischen Ziele
- bei der Versendung der kritischen Paare
- bei der Versendung des Regel- und Gleichungssystems nach der Interreduktion

Nach der Ersetzung der `for`-Schleifen, die die Daten sequentiell an alle Team-Mitglieder verschickten, durch die entsprechenden Read/Write-Aufrufe waren alle Vorebereitungen abgeschloßen und die Team-Vervollständigung mit Broadcast einsatzbereit.

4.2.3 Leistungssteigerungen durch Broadcasting

Um überhaupt beurteilen zu können, ob eine Erweiterung eines Software-Systems auch einen Erfolg darstellt, müssen die Ziele, die mit dieser Erweiterung erreicht werden sollen, genau umrissen werden. Die Möglichkeiten, die Ziele zu beschreiben, reichen von einer groben, umgangsprachlichen Vorgabe bis hin zu einer sehr formalen Spezifikation. Beim DISCOUNT-System, für das im Rahmen dieser Arbeit ein sicheres Broadcasting entwickelt wurde, waren die beiden wesentlichen Zielsetzungen informell folgendermaßen gegeben:

(1) Da durch die Team-Vervollständigung mit 2 Experten nachweißlich Leistungssteigerungen gegenüber einer sequentiellen Vervollständigung erzielt werden können (vgl. [De93]), sollen diese Verbesserungen auch beim Einsatz des Broadcasting erhalten bleiben. Da es in diesem Fall nur *einen* Empfänger gibt, spricht man normalerweise nicht von einem Broadcast, der Datenaustausch soll aber dennoch über das Protokoll abgewickelt werden. Durch diese Erweiterung des Begriffs „Broadcast“ wird das Programm einfacher, da keine Unterscheidung zwischen einem und mehreren Empfängern notwendig ist.

(2) Der Zuwachs an Overhead, der durch verlorene Pakete (siehe 4.1) entsteht, soll so gering sein, daß es möglich ist, sehr viele, eventuell auch hochspezialisierte Experten, einzusetzen. Hinter diesen Überlegungen steckt der folgende Gedanke: Werden viele Experten mit unterschiedlichen Strategien eingesetzt, so steigt die Wahrscheinlichkeit, daß einer dieser Experten für das gegebene Problem genau die richtige Taktik anwendet. Bei der bisherigen Implementierung (mit Stream-Verbindungen) war dieses Vorgehen nicht praktikabel, da der Kommunikationsoverhead schon bei drei Experten jede Verbesserung zunichte machte. Wie später noch nachgewiesen werden wird, liegen die Dinge beim Broadcasting völlig anders. Da die Zeit, die benötigt wird, um die Daten an einen Empfänger zu schicken, ungefähr gleich derjenigen ist, die benötigt wird um die Daten an viele zu schicken, können sehr viele verschiedene Experten eingesetzt werden. Liefert einer dieser Experten ein gutes Zwischener-

gebnis, so wird damit weitergerechnet, liefert er nichts Brauchbares so bringt dies, zumindest vom Laufzeitverhalten des Systems her, keine Nachteile.

Noch einmal zusammengefaßt ergibt dies die folgenden Anforderungen:

- möglichst geringer (gar kein) Leistungsverlust bei nur einem Empfänger
- geringer Aufwandszuwachs bei jedem weiteren Empfänger

Um nun nachzuweisen, daß die Zielvorgaben tatsächlich erreicht wurden, muß eine Versuchsanordnung gefunden werden, die Laufzeitverbesserungen durch Synergieeffekte weitestgehend ausschließt. Aus diesem Grund werden die in [De93] vorgestellten *Quasiteams* zur Laufzeitmessung verwendet. Bei diesen Quasiteams laufen auf allen beteiligten Rechnern die gleichen Experten. Zugewinne sind also nur dann möglich, wenn ein Rechner schneller ist als der andere (z. B. bei ungleicher Lastverteilung). Im Lauf der Vervollständigung kann zwar der Teamleiter wechseln, da aber alle Teammitglieder dieselbe Strategie verfolgen, hat man es im Grunde mit einer sequentiellen Vervollständigung zu tun.

Bevor nun die verschieden Laufzeiten vorgestellt und kommentiert werden, soll noch ein kurzer Hinweis erfolgen:

> Das zur Messung der Broadcasting-Testläufe verwendete Programm enthält verschiedene Erweiterungen (die nichts mit der Kommunikation zu tun haben) gegenüber der Version mit Stream-Verbindungen. Der Grund hierfür liegt in der Tatsache, daß das Broadcasting-Protokoll sich als so praxistauglich erwiesen hat, daß nur noch Programmversionen mit Broadcasting bei Neuerungen unterstützt werden. Aus diesem Grund sollten auch nicht die *absoluten* Laufzeiten im Vordergrund stehen, viel interessanter sind die relativen Zuwächse.

Im folgenden wird nun untersucht, wie die Quasiteams mit einer unterschiedlichen Zahl von Experten ein gegebenes Problem lösen. Da die Problemgröße ausschlaggebend dafür ist, ob über die Messungen überhaupt fundierte Aussagen gemacht werden können, sind hier nur zwei Problemstellungen aufgegriffen worden. Alle anderen Aufgaben, die im Moment zur Verfügung stehen, werden in so kurzer Zeit gelöst, daß sich eine nähere Betrachtung nicht lohnt. Sogar das erste angegebene Problem (*luka3*) liegt an der Untergrenze, bei der sich noch sinnvolle Aussagen machen lassen. Auf eine genauere Beschreibung der einzelnen Aufgabenstellungen wird hier verzichtet, Erläuterungen finden sich in [De93].

Beispiel/ Zykluszeit in Sek.		2 Experten	3 Experten	5 Experten
luka3/30 s	Broadcast	323.83 s	327.95 s	348.40 s
	Streams	401.93 s	540.24 s	734.05 s
luka3/50 s	Broadcast	313.25 s	384.52 s	401.16 s
	Streams	356.51 s	418.87 s	583.36 s
lusk6/50 s	Broadcast	5462.54 s	5800.90 s	5376.12 s
	Streams	6556.38 s	7545.49 s	10035.01 s
lusk6/500 s	Broadcast	4571.37 s	4589.26 s	4613.31 s
	Streams	5412.51 s	5699.99 s	6080.65 s
lusk6/1000 s	Broadcast	4589.09 s	4578.28 s	4697.50 s
	Streams	5257.67 s	5339.32 s	5577.23 s
lusk6/2000 s	Broadcast	4840.59 s	4808.50 s	4809.75 s
	Streams	5401.79 s	5217.59 s	5318.93 s

Abbildung 4.1: Leistungsvergleiche

Bei der Betrachtung der ersten Messung (*luka3*/30 s Zykluszeit) sieht man deutlich, daß beide, im vorrangegangenen Abschnitt angesprochenen, Zielvorgaben erfüllt wurden. Sehr deutlich tritt zu Tage, daß der Laufzeitzuwachs sich beim Broadcasting in sehr engen Grenzen hält. Auch bei *luka3*/50 s ist der Zuwachs im Vergleich zu Streams geringer, wenn auch nicht so ausgeprägt wie im ersten Lauf.

Ein sehr eindrucksvolles Argument für die Verwendung von Broadcasts liefert die Messung *lusk6*/50 s. Während beim Broadcasting bei 5 Experten sogar eine geringere Laufzeit erzielt wurde als bei 2 und 3 Experten, hat die Version mit Stream-Verbindungen keine Chance mehr. Eine Laufzeitverschlechterung von 6556.38 s bei 2 Experten auf 10035.01 s bei 5 Experten entspricht einem Zuwachs von ca. 53 %. Dies ist natürlich nicht tragbar.

Die Ursache für die besseren Ergebnisse bei 5 Experten als bei zweien liegt darin, daß geringe Unterschiede in der Lastverteilung schon große Auswirkungen auf die Gesamtlaufzeit haben. Die Messungen müßten, um völlig exakt zu sein, auf Rechnern durchgeführt werden, auf denen keine anderen Prozesse ablaufen. Dies ist aber zum einen nicht praktikabel und zum anderen eine absolut unrealistische Situation.

Bertrachtet man nun die weiteren Messungen, so sieht man, daß die Programmversion mit Stream-Verbindungen bei längeren Zykluszeiten gegenüber der Broadcast-Version aufholen kann und letztlich auch ein etwa konstantes Zeitverhalten zeigt. Der Grund hierfür liegt aber nur in der Verringerung des Kommunikationsvolumens und der Kommunikationshäufigkeit.

Insgesamt sieht man beim Beispiel *lusk6* sehr deutlich, daß eben dieses Kommunikationsvolumen der entscheidende Faktor ist. Bei einer Zykluszeit von 50 s werden in ca. 60 Teamtreffen ca. 1 Million kritischer Paare versendet (siehe [De93]). Bei einer durchschnittlichen Größe von 20 - 30 Byte pro kritischem Paar hat man hier ein Gesamtvolumen von 20 - 30 MByte, die über das Netz geschickt werden müssen. Bei einer Zykluszeit von 2000 s gibt es nur noch 2 - 3 Teamtreffen, während denen nur etwa 40000 kritische Paare versendet werden müssen. Dies bestätigt also die Aussage, daß bei kleinen Problemen mit wenig Kommunikation kaum ein Unterschied zwischen Streams und Broadcasts besteht, bei hohem Aufwand das Broadcasting aber deutliche Vorteile bietet.

Mit den vorgestellten Messungen läßt sich also zeigen, daß die Zielvorgaben bei der Entwicklung des Protkolls erfüllt wurden und daß das Brodcasting-Modul eine zumindest gleichwertige, wenn nicht gar bessere Alternative zu Streams bietet.

Kapitel 5

Abschlußbemerkungen

Die Team-Vervollständigung mit Broadcasting befindet sich schon seit einiger Zeit im Einsatz und arbeitet so gut, daß nur noch diese Programmversionen bei Erweiterungen unterstüzt werden. Trotzdem gibt es noch Verbesserungsmöglichkeiten und Erweiterungen, die an dieser Stelle kurz skizziert werden sollen. Dabei fließen selbstverständlich die Erfahrungen, die in den letzten Monaten mit dem Programm gemacht wurden, mit ein.

Zunächst einmal erscheint es ratsam, die unterschiedlichen Rechengeschwindigkeiten und Auslastungen der am Teamwork teilnehmenden Rechner in den gesamten Protokollablauf zu integrieren. Dies könnte dadurch geschehen, indem jedem Rechner ein individueller Geschwindigkeitsindex zugeordnet wird, der mit der momentanen Last, d. h. dynamisch (z. B. während der Quittierungsphase), verrechnet wird. Auf diese Weise kann das Überfluten (flooding) von langsamen Rechner (vgl. 4.1) zumindest eingeschränkt, wenn nicht gar verhindert werden. Obwohl das Versenden der Datenpakete sich durch dieses Vorgehen an den langsamsten Rechnern orientiert, kann es dennoch zu Verbesserungen im Gesamtverhalten kommen, da auf diese Weise weniger Pakete verlorengehen und doppelt übertragen werden müssen.

Eine weitere Verbesserung wäre die dynamische Gestaltung des Timeouts. In der n-ten Quittierungs- (oder Bereitmeldungs-)phase können die Informationen der (n-1) vorhergehenden Phasen genutzt werden um zu erkennen, ob die Timeouts z. B. verlängert werden sollten. Dies kann der Fall sein, wenn die Prozesse stark „auseinanderlaufen“, d. h. wenn die Teamtreffen extrem asynchron begonnen werden.

Ein letzter Punkt, der hier noch angesprochen werden soll, betrifft nicht das Broadcasting-Protokoll an sich, sondern vielmehr die gesamte Implementierung des DISCOUNT-Systems. Betrachtet man diese Implementierung, so fällt auf, daß sehr viele Details, die von der Abstraktionsstufe her nach „unten“ gehören, in höheren Ebenen zu finden sind. Dies betrifft insbesondere die Socketkommunikation und die Verarbeitung der Signale.

Die Teamwork-Methode bietet eine Möglichkeit des verteilten Problemlösens, die sich für sehr viele Problemstellungen eignet. Falls ein neues Projekt auf der Basis dieser Methode durchgeführt werden soll, sollte darauf geachtet werden, alle Kommunikationskomponenten (Sockets und Signale) in einem, in sich abgeschlossenen Subsystem zusammenzufassen. Dieses Subsystem könnte auch alle Anweisungen zum Hochfahren und Terminieren eines Teams enthalten. Durch diese Maßnahme wird das Gesamtsystem einfacher und übersichtlicher; Schwierigkeiten, die mit der eigentlichen Verteilung nichts zu tun haben, werden auf diese Weise von der konkreten Problemstellung entkoppelt.

Dies waren noch einige Ideen, die mir nach der Fertigstellung und während der Arbeit mit dem Protokoll gekommen sind. Es gibt sicher noch weitere gute Ideen, die in (vielleicht) kommende Versionen einfließen werden.

Literaturverzeichnis

[Co87] Coffield, D., Sheperd, D. *Tutorial Guide to UNIX sockets for network Communications*, Computer Communications Vol. 10 No. 1 Feb. 87, S. 21 ff.

[De91] Denzinger, J. *Distributed Knowledge-based Deduction using the Teamwork Method*, SEKI Report SR-91-12 (SFB)

[De93] Denziger, J. *Teamwork: Eine Methode zum Entwurf verteilter, wissensbasierter Theorembeweiser*, Dissertation, Universität Kaiserslautern 1993

[Gau87] Gauweiler, T. *UNIX als Basis für verteilte Systeme* Informationstechnik it 29. Jahrgang, Heft 6/1987, S. 386 ff.

[Pi92] Pitz, W. *Realisierung eines Systems zum verteilten, wissensbasierten Gleichheitsbeweisen mit Hilfe der Teamwork Methode*, Diplomarbeit, Universität Kaiserslautern 1992

[Bu91] Buchman, J., Diehl, M., Roth, R. *LIPS - A System For Ditributed Application* Preprint, FB-14 Informatik, Universität Saarbrücken 1991

[Ste92] Stevens, W. R. *Programmieren von UNIX-Netzen* München; Wien: Carl Hanser Verlag; London: Prentice-Hall, 1992

[SUN87a] SUN Microsystems *Network Programming*, Handbücher zu SUN 3/60 Anlagen, 1987/88

[SUN87b] SUN Microsystems *SunOS Reference Manual*, Handbücher zu SUN 3/60 Anlagen, 1987/88