Eine reservierungsfähige Version von IP über Ethernet für DROPS

Diplomarbeit von

Marco Rudolph
geboren am 1.11.1974 in Borna

Matr.–Nr.: 2296191

Betreuender Hochschullehrer:

Prof. Dr. Hermann Härtig
Fakultät Informatik
Institut für Betriebssysteme, Datenbanken und Rechnernetze

Technische Universität Dresden
April 2000

Erklärung

Hiermit erkläre ich, daß ich die vorliegende Diplomarbeit selbständig und ohne weitere Hilfe verfaßt und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Dresden, den 27. April 2000

(Marco Rudolph)

Die vorliegende Arbeit wurde im Zeitraum von November 1999 bis April 2000 unter der Betreuung von

Dipl.–Inf. Michael Hohmuth, Fakultät Informatik, Institut für Betriebssysteme, Datenbanken und Rechnernetze, Technische Universität Dresden

durchgeführt. Ich danke meinem Betreuer für die interessante Themenstellung und die Ermöglichung der fachbereichsübergreifenden Arbeit.

Ebenso gilt mein Dank allen, die wissentlich oder unwissentlich zum Gelingen der Arbeit beigetragen haben, insbesondere den weiteren Mitarbeitern des Instituts.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Index

Abkürzungsverzeichnis

ARP	Address Resolution Protocol
ATM	Asynchronous Transfer Mode
BSD	Berkeley Software Distribution
BPL	Berkeley Public License
COM	Common Object Model
CSMA/CD	Carrier Sense Multiple Access with Collision Detection
ARPANET	Advanced Research Project Agency Network
DROPS	Dresden Real-time Operating System
FIFO	First In First Out
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol
IPC	Inter Process Communication
ISI	Information Sciences Institute
ISO	International Standardisation Organization
OSI	Open Systems Interconnection
RFC	Request for Comments
RSVP	Resource Reservation Protocol
ST–II	Stream Protocol Version 2
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
XNS	Xerox Network System

Aufgabenstellung

*Für DROPS, das **D**resden **R**ealtime **OP**erating **S**ystem, soll eine echtzeitfähige Version des IP–Protokolls über Ethernet pragmatisch und prototypartig entworfen, implementiert und bewertet werden. Die entstehende Komponente soll in der Lage sein, verfügbare Netzwerkressourcen anteilig bestimmten Anwendungen exklusiv zur Verfügung zu stellen. Über ein Konzept der Überbuchung und Priorisierung soll ebenfalls nachgedacht werden.*

Als Grundlage der Implementation kann ein geeigneter bereits existierender, im Quellcode verfügbarer IP–Stack für Ethernet verwendet werden.

Kapitel 1

Einführung

1.1 Motivation

In heutiger Zeit gewinnt die Kopplung von Computern immer größere Bedeutung. Zum einen lassen sich dadurch die Ressourcen preiswerter Rechner gemeinsam für die Lösung komplexer Probleme einsetzen. Zum anderen wird durch eine mögliche zentrale Speicherung der Wartungs- und Pflegeaufwand der Daten entscheidend vereinfacht. Netzwerke erlauben es dann, Daten weltweit zwischen beliebigen Rechnern auszutauschen.[1]

Basierend auf dem Mikrokern *L4* [Lie95] wird an der Technischen Universität Dresden an einem Betriebssystem gearbeitet, welches in der Lage ist, neben "gewöhnlichen" Timesharing–Anwendungen auch Echtzeit–Applikationen auszuführen. Diese Arbeit läuft unter dem Titel *Dresden Realtime Operating System*, kurz *DROPS* [HBB+98]. Auf dem L4–μ–Kern läuft dabei — neben einer Reihe von Servern, die Echtzeitdienste zur Verfügung stellen — als Timesharing–System eine Portierung des freien Betriebssystems Linux. Diese Portierung wurde ebenfalls im Rahmen der Arbeit an Drops vorgenommen. Durch dieses als L4Linux bezeichnete System erschließt sich dem Drops–System das stetig wachsende Angebot an Linux–Software — und damit natürlich auch sämtliche Entwicklungswerkzeuge.

Unter Drops reservieren die Echtzeit–Anwendungen dabei wie üblich die für ihre Arbeit erforderlichen Ressourcen wie Arbeitsspeicher, Prozessorzyklen u.s.w. Darüberhinaus nicht genutzte Ressourcen stehen dann den Timesharing–Applikationen zur Verfügung.

Um Drops der oben geschilderten Entwicklung im Netzwerkbereich nicht zu verschließen, muß es über die entsprechenden Protokolle mit angekoppelten Rechnern kommunizieren können. Große Verbreitung hat dabei das Transportmedium Ethernet mit dem Kommunikationsprotokoll TCP/IP gefunden. Ein IP–Stack fungiert dabei als Bindeglied zwischen den Applikationen auf der einen und dem Netzwerkinterface auf der anderen Seite (siehe Abschnitt 2.1). Derzeit

[1] Natürlich nur, solange die Kommunikationspartner über ebensolche Netzwerke miteinander verbunden sind.

existiert noch keine Implementation der Internetprotokolle für DROPS. Ziel dieser Arbeit soll es daher sein, dieses Anwendungsgebiet durch die Entwicklung eines IP–Stacks für DROPS zu erschließen.

Da DROPS für den Einsatz in Echtzeitumgebungen entwickelt wird, sollte der IP–Stack darüberhinaus in der Lage sein, Zusagen über bestimmte Übertragungsparameter geben und diese natürlich auch einhalten zu können.

1.2 Gliederung der Arbeit

Das folgende Kapitel geht auf den derzeitigen Stand der Technik ein: Zunächst werden die Prinzipien der wichtigsten Internet–Protokolle *IP*, *TCP* und *UDP* kurz vorgestellt. Daran schließt sich eine Übersicht über die gebräuchlichsten Verfahren zur Reservierung bzw. Sicherstellung einer bestimmten Übertragungsgüte, *RSVP* und *ST–II*, an. Nach der Beschreibung einer Implementierung von ATM für L4 gehe ich auf die IP–Implementierungen von BSD und LINUX ein. Den Abschluß dieses Kapitels bildet eine Beschreibung der BSD–Socketschnittstelle.

Im Kapitel 3 wird nach einer geeigneten Implementationsmethode gesucht. Dazu werden verschiedene Möglichkeiten der Prozeßnachbildung, der Speicherverwaltung und der Ressourcenreservierung miteinander verglichen und hinsichtlich ihrer Eignung bewertet.

Das Kapitel 4 geht auf wesentliche Aspekte der Implementierung des IP–Stacks für das DROPS–System ein. Besondere Beachtung erfährt dabei das Prozeßkonzept, die Socketschnittstelle und der Paket–Scheduler.

Eine Bewertung der Leistung des realisierten IP–Stacks wird im Kapitel 5 vorgenommen.

Das letzte Kapitel enthält eine kurze Zusammenfassung der Ergebnisse.

Kapitel 2

Stand der Technik

2.1 Internet–Protokolle

Ende der sechziger Jahre unterstützte das US–amerikanische Verteidigungsministerium den Aufbau eines flächendeckenden Rechnerverbundes zwischen amerikanischen Universitäten. Im Rahmen dieses sogenannten ARPANET–Projektes wurden Standards und Verfahren entwickelt, die im wesentlichen noch heute gebräuchlich sind. Dabei sind die einzelnen Protokolle auf einen recht begrenzten Funktionsumfang beschränkt. Die Funktionalität des Gesamtsystems ergibt sich erst aus dem harmonischen Zusammenwirken dieser einzelnen Protokolle.

Diese Trennung hält zum einen die Komplexität überschaubar. Andererseits war es in dieser Frühphase der Entwicklung absehbar, daß keine endgültigen Standards geschaffen werden konnten. Durch eine geeignete Schnittstellenspezifikation läßt sich der Austausch einzelner Protokolle relativ einfach vornehmen, wenn die Entwicklung fortgeschrittenerer Protokolle dies erforderlich macht.

Zum Zwecke der Adressierung und Fragmentierung[1] wurde das sogenannte *Internet Protocol* [Pos81a] eingeführt. Eine darüberhinausgehende Funktionalität muß von übergeordneten Protokollen realisiert werden. Zu diesem Zweck wurden das *Transmission Control Protocol* [Pos81b] (Abschnitt 2.1.2, Seite 8) und das *User Datagram Protocol* [Pos80] (Abschnitt 2.1.3, Seite 9) entwickelt. Abbildung 2.1 soll die Struktur hinter dieser Aufgabenteilung verdeutlichen.

Im Laufe der Zeit wurde eine Vielzahl weiterer Spezifikationen erarbeitet. Diese entstanden einerseits, um Lücken bzw. Fehler in bestehenden Protokollen zu beheben. Ein Beispiel hierfür ist das im RFC[2] 1883 beschriebene Protokoll

[1] Daten werden über Netzwerkverbindungen üblicherweise in Paketen einer bestimmten Größe übertragen. Sollen nun Datenblöcke transportiert werden, die diese Größe übersteigen, so müssen sie beim Sender in kleinere Pakete zerlegt und beim Empfänger wieder zusammengefügt werden.

[2] *RFC (Request for Comments)* — Sammlung von Internet–Standard–Spezifikationen über Informationsschriften bis hin zu Vorschlägen für neue oder verbesserte Protokolle.
Wie der Name bereits andeutet, sind Anmerkungen und Hinweise zu den veröffentlichten Informationen durchaus erwünscht.

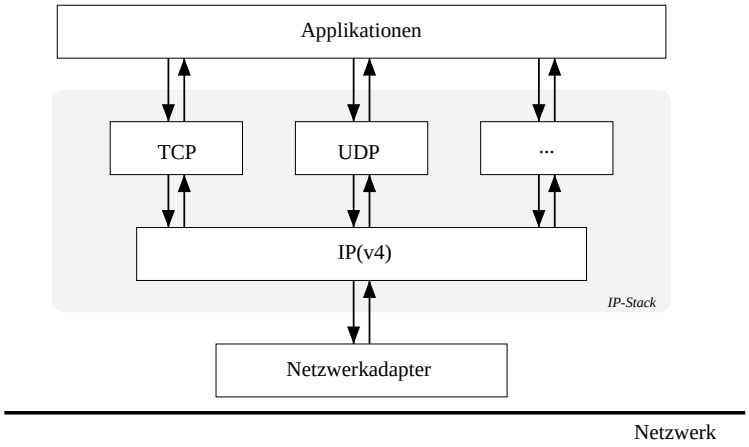


Abbildung 2.1: Beziehung zwischen den einzelnen Internet–Protokollen
Neben TCP und UDP sind natürlich noch weitere Protokolle denkbar, die weitergehende Anforderungen an den Datentransport, beispielsweise bei Echtzeitübertragungen, realisieren.

IPv6 [DH95], welches Limitationen, besonders bezüglich des Adreßumfangs, des zur Zeit gebräuchlichen Internet–Protokolls der Version 4 behebt (siehe Abschnitt 2.1.1, Seite 5). Ebenso werden Protokolle vorgeschlagen, die neue Funktionalitäten innerhalb der Internet–Protokollfamilie realisieren. Gerade in der letzten Zeit hat besonders die Verteilung von Multimedia–Daten über das Internet eine starke Verbreitung gefunden. Für deren Übermittlung ist es oftmals erforderlich, bestimmte Übertragungseigenschaften wie zum Beispiel eine gewisse Bandbreite jederzeit zur Verfügung zu haben, um eine gewünschte Wiedergabequalität der Daten sicherzustellen. Die im Abschnitt 2.2 beschriebenen Reservierungsverfahren fallen dabei in diese Kategorie.

2.1.1 Internet Protocol Version 4 (IPv4)

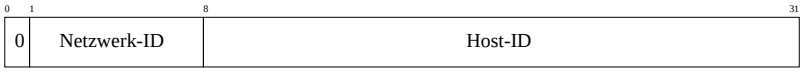
Das Internet–Protokoll der Version 4, kurz *IPv4*, hat innerhalb der IP–Protokollhierarchie (siehe Abb. 2.1) die Aufgabe, einzelne Datenpakete vom Sender zum Empfänger zu transportieren. Da es sich hier um ein verbindungsloses Protokoll handelt, wird jedes Paket separat behandelt. Die Reihenfolge, in der der Sender die Pakete abschickt, bestimmt nicht zwangsläufig die Empfangsreihenfolge. Der Weg jedes Pakets wird individuell von allen auf dem Transportweg befindlichen Rechnern bestimmt, d.h. die Entscheidung über die Wegewahl, *Routing* genannt, wird unabhängig vom Kontext des Paketinhalts getroffen. Alle Kommunikationspartner im Netzwerk sind dabei eindeutig durch ihre Adresse bestimmt.

IP–Adressen

Die von der IP–Version 4 verwendete Adressen bestehen aus einem 32–Bit–Wert, der in vier Gruppen zu jeweils 8 Bit aufgeteilt ist. Diese Adressen setzen

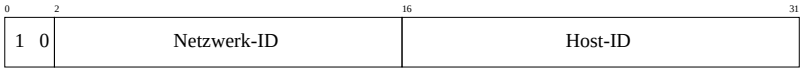
sich aus einem *Adreßtyp–Identifikator* sowie einer *Netzwerk–ID* und einer *Host–ID* zusammen. Logisch zusammengehörige Hosts werden häufig zu Netzwerken zusammengeschlossen. Nach außen erscheint ein solches Netzwerk als Einheit, welches sich intern in weitere Unter–Netzwerke sowie einzelne Hosts aufspalten läßt. Die Netzwerk–ID in der Adresse bestimmt nun ein solches Netzwerk, während die Host–ID die nächsttiefere Ebene adressiert.[3] Entsprechend ihres Adreßbereichs werden fünf Adreßklassen unterschieden:

- **Klasse A**



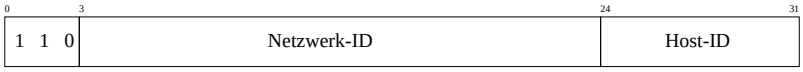
  Um den Adreßtyp identifizieren zu können, ist das erste Bit dieser Adressen reserviert und enthält stets den Wert *0*. Die nächsten sieben Bit stehen zur Kennzeichnung des Netzwerks, in dem sich der Host befindet, zur Verfügung. Damit lassen sich 126 Netzwerke adressieren, da die Netzwerk–Kennungen 0 und 127 für spezielle Zwecke genutzt werden. Die letzten 24 Bit dienen schließlich der Adressierung von bis zu 2^{24} Hosts im jeweiligen Netzwerk.

- **Klasse B**



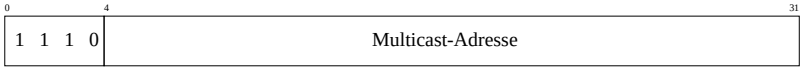
  In Adressen der Klasse B sind zur Kennzeichnung die ersten beiden Bits reserviert und enthalten stets den (binären) Wert *10*. Daneben stehen 14 Bits zur Adressierung von 2^{14} Netzwerken und 16 Bits für jeweils 2^{16} Hosts zur Verfügung.

- **Klasse C**



  Die verbreiteteste Adreßklasse nutzt zur Klassifizierung die ersten drei Bits und belegt diese mit der Sequenz *110*. Es lassen sich 2^{21} Netzwerke mit jeweils 2^8 Hosts adressieren.

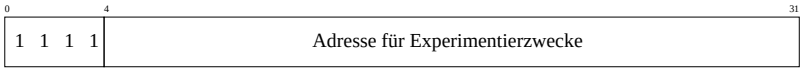
- **Klasse D**



  Adressen, deren erste vier Bits den binären Wert *1110* enthalten, enthalten keine Netzwerk– bzw. Host–ID. Stattdessen handelt es sich um Multicast–Adressen. Diese werden verwendet, um Gruppen von Hosts zu adressieren.

[3] Die *Host–ID* wird also nicht ausschließlich zur Adressierung einzelner Hosts eines Netzwerks genutzt, sondern adressiert ebenso kleinere Netzwerke, die an ein übergeordnetes angekoppelt sind.

- **Klasse E**



Diese Adreßklasse E ist für experimentelle Zwecke reserviert und steht für den normalen Gebrauch nicht zur Verfügung.

Klasse	Klassifizierung	adressierbare Netze	adressierbare Hosts
A	0	126	16646144
B	10	16256	65024
C	110	2080768	254
D	1110	–	–
E	1111	–	–

Tabelle 2.1: Übersicht über die Adreßklassen
(Da einige Adressen spezielle Bedeutung haben, stehen nicht alle theoretisch verwendbaren Adressen zur Verfügung.)

In den letzten Jahren ist ein enorm gestiegener Bedarf an IP–Adressen zu verzeichnen. Damit ist absehbar, daß mit den ca. 2^{32} verfügbaren Adressen auf lange Sicht die Anforderungen nicht mehr erfüllbar sind. Aus diesem Grund wurde die IP–Version 6 (auch *IPng* bzw. *IP next generation*) entwickelt, welche neben weiteren Modifikationen eine Erweiterung des Adreßumfanges auf 128 Bit vorsieht [DH95]. Die vorliegende Arbeit bezieht sich allerdings noch auf IP–Version 4.

Desweiteren übernimmt IP die Fragmentierung bzw. Reassemblierung von Paketen, sofern dies erforderlich ist. IPv4 ist lediglich darauf ausgelegt, Pakete von maximal 65535 Bytes zu übertragen. Darüberhinaus fordert die Spezifikation des Protokolls, daß jeder Host lediglich in der Lage sein muß, 576 Bytes große Datagramme[4] akzeptieren zu können[5] [Pos81a, S.12]. Sollen nun größere Pakete transportiert werden oder soll das Paket in einen Abschnitt des Netzwerks weitergeleitet werden, welches nur kleinere Paketgrößen verarbeiten kann, so werden die Daten auf mehrere Datagramme aufgeteilt und diese als zusammengehörend markiert. Der Transport der einzelnen Pakete erfolgt danach weiterhin unabhängig voneinander. Der Empfänger eines solchen Pakets erkennt nun an dieser speziellen Markierung, daß es sich bei diesem Paket um den Teil eines größeren Datagramms handelt. Er gibt deshalb das Paket nicht sofort an übergeordnete Protokolle wie TCP weiter, sondern wartet, bis alle zusammengehörenden Pakete eingetroffen sind. Erst nachdem er diese wieder zum ursprünglichen Datagramm zusammengefügt hat, reicht er das komplette Datagramm weiter. Dieses Verfahren läuft für die nachfolgenden Protokolle vollkommen transparent ab.

Eine darüberhinausgehende Funktionalität wird von IP nicht angeboten. So sind beispielsweise keine Vorkehrungen gegen Paketverluste oder -vervielfälti-

[4] *Paket* und *Datagramm* werden im Folgenden synonym verwendet.

[5] Diese 576 Bytes erlauben es, neben einem 64 Bytes großen Header noch 512 Bytes Daten in einem Paket aufzunehmen.

gungen, Verfälschungen des Paketinhaltes oder Transportverzögerungen getroffen worden. Die Behandlung solcher Probleme wird übergeordneten Protokollen überlassen.

Applikationen greifen meist nicht direkt auf IP zu, sondern benutzen übergeordnete Protokolle wie TCP und UDP zur Datenkommunikation, welche dann die von IP angebotenen Funktionen nutzen.

2.1.2 Transmission Control Protocol (TCP)

Im Gegensatz zu IP ist das *Transmission Control Protocol*, kurz *TCP* [Pos81b], verbindungsorientiert. Daraus ergeben sich eine Reihe von Eigenschaften der Datenkommunikation über TCP:

- Vor jedem Datentransfer ist explizit eine Verbindung zwischen den Kommunikationspartnern zu etablieren. Dies geschieht durch den Austausch von Synchronisationsnachrichten über IP–Pakete.

- Die Reihenfolge der Pakete bleibt erhalten. Dies bedeutet nicht zwangsläufig, daß alle Pakete den gleichen Weg im Netzwerk nehmen. Der Sender numeriert die Pakete fortlaufend, so daß sie vom Empfänger geordnet werden können. Nach dem Erhalt eines Paketes sendet der Empfänger eine Empfangsbestätigung an den Sender. Dieser erkennt dadurch, daß er nun ein weiteres Paket abschicken kann.
  Zur Verbesserung der Leistungsfähigkeit dieses Verfahrens kann der Empfänger mitteilen, welche Datenmenge er momentan zu empfangen in der Lage ist. Der Sender darf nun diese Menge an Daten, also gegebenenfalls auch mehrere Datagramme, abschicken. Vom Empfänger werden diese Pakete, sofern erforderlich, in die richtige Reihenfolge gebracht und nur der Erhalt des kompletten Datenblocks quittiert.

- TCP–Verbindungen sind zuverlässig. Sofern der Sender innerhalb einer bestimmten Zeitspanne keine Empfangsbestätigung für ein Paket erhalten hat, so sendet er es nochmals ab.
  In langsamen Netzen kann es vorkommen, daß auf diese Weise Duplikate entstehen. Der Empfänger ist deshalb in der Lage, aufgrund der fortlaufenden Numerierung der Pakete diese Duplikate zu erkennen und zu verwerfen.

- Fehlerhafte Pakete können durch ein Prüfsummenverfahren erkannt werden. Verfälschte Datagramme werden vom Empfänger verworfen und nicht quittiert, was zwangsläufig zu einer erneuten Übertragung führt.

Bei einer Datenübertragung mittels TCP bleibt also die Reihenfolge, in der Daten gesendet werden, erhalten. Es ist zudem sichergestellt, daß keine Pakete auf dem Übertragungsweg verlorengehen oder verfälscht werden.

Mit den Adressen des IP–Protokolls lassen sich lediglich einzelne Hosts adressieren. Eine Zuordnung zu einzelnen Applikationen läßt sich damit nicht

vornehmen. Aus diesem Grunde führt TCP (und auch UDP, siehe Abschnitt 2.1.3) das Konzept der *Ports* ein. Ein Port ist dabei ein numerischer Wert, der zusammen mit dem Protokoll–Identifikator einen lokalen Datenstrom eindeutig bestimmt.

Diese Funktionsvielfalt zieht natürlich einen erhöhten Protokollaufwand nach sich. Zum einen erhöht die Quittierung empfangener Pakete das Datenaufkommen im Netzwerk. Zum anderen sind Zeitgeber zu verwalten, welche das Ausbleiben von Paketen anzeigen und damit erneute Paketsendungen veranlassen. Nicht zuletzt müssen sämtliche gesendeten Daten bis zum Eintreffen der jeweiligen Empfangsbestätigungen gepuffert werden.

2.1.3 User Datagram Protocol (UDP)

Für viele Einsatzfälle ist der Funktionsumfang von TCP zu komplex und damit schwerfällig oder einfach nur unnötig. Deshalb wurde mit dem *User Datagram Protocol UDP* [Pos80] ein verbindungsloses Protokoll entwickelt, welches die von IP angebotene Funktionalität lediglich geringfügig um die Verwendung von Prüfsummen und Ports erweitert. Dadurch wird den Applikationen ermöglicht, mit einem Minimum an Protokolloverhead Daten übermitteln zu können. Solche Eigenschaften wie Zuverlässigkeit und Erhalt der Paketreihenfolge kann UDP dabei nicht realisieren. Diese müssen nötigenfalls von der Applikation selbst sichergestellt werden

2.2 Reservierungsverfahren

Die im Abschnitt 2.1 beschriebenen Internet–Protokolle beschränken sich darauf, Daten nach einem *Best–Effort–Prinzip* zu übertragen. In den letzten Jahren haben nun aber Anwendungen wie Videokonferenzsysteme oder Multimedia–Übertragungen an Bedeutung gewonnen, die oftmals andere Anforderungen an die Datenübertragung stellen. Bei *Video–On–Demand* ist es beispielsweise nicht ausreichend, Bilddaten lediglich zuverlässig zu übertragen. Wesentlich wichtiger ist es, daß diese Daten innerhalb bestimmter Zeitintervalle beim Konsumenten eintreffen. Zu diesem Zweck ist es erforderlich, bestimmte Dienstgüteparameter wie beispielsweise *Bandbreite* oder *maximale Latenzzeit* für entsprechende Verbindungen zusichern zu können. Mit den herkömmlichen Protokollen ist dieses Ziel kaum zu erreichen. Aus diesem Grund wurde eine Reihe weiterer Protokolle entworfen, die dieses Problem zu lösen versuchen. Die beiden bekanntesten, nämlich das *Resource Reservation Protocol (RSVP)* [BZBH97] und das *Stream Protocol 2 (ST–II)* [Top90], sollen im Folgenden vorgestellt werden.

2.2.1 Resource Reservation Protocol (RSVP)

Das in [BZBH97] beschriebene *Resource Reservation Protocol* orientiert sich sehr stark am Datagrammdienst UDP (siehe Abschnitt 2.1.3) der

IP–Protokollfamilie. Besonders macht es sich dessen Erweiterung zur Gruppenkommunikation [Dee89] zunutze und profitiert zugleich vom geringen Protokolloverhead. RSVP ist dabei als Ergänzung bzw. Erweiterung der IP–Protokollfamilie gedacht. Sämtliche Daten werden unverändert mittels IP übertragen.

In RSVP gehen Reservierungen vom Empfänger aus. Besonders bei den bereits erwähnten Multimedia–Verteildiensten wie Video–On–Demand ist dieses Vorgehen vorteilhaft. Sobald ein Interessent diese Dienste in Anspruch nehmen will, teilt er diesen Wunsch dem Anbieter mit und initiiert damit den Kommunikationsvorgang. Dabei werden über RSVP ausschließlich Signalisierungsinformationen übertragen, welche beschreiben, mit welcher Dienstgüte bestimmte UDP–IP–Datenflüsse erfolgen sollen. Die Datenübertragung selbst erfolgt über den normalen IP–Stack. Dies bedingt natürlich, daß dieser Kenntnis über die Reservierungen hat und sie bei der Behandlung der Datenpakete berücksichtigt. RSVP läßt sich dabei entsprechend Abbildung 2.2 in den "herkömmlichen" IP–Protokollstack einordnen.

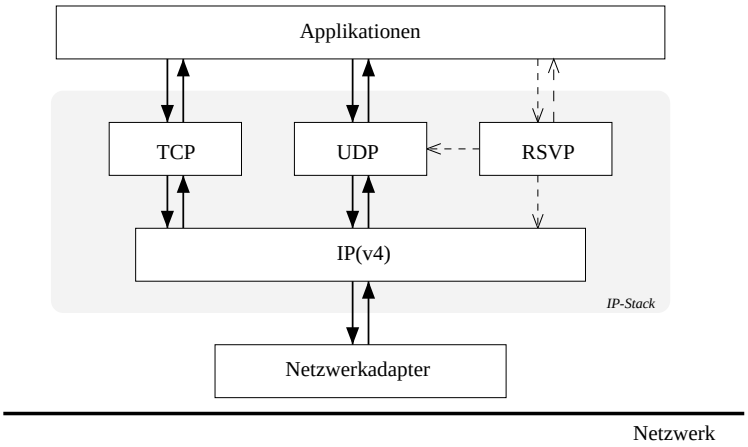


Abbildung 2.2: Einordnung des RSVP–Protokolls in den IP–Protokollstack

Die wichtigsten von RSVP genutzten Signalisierungsinformationen sind sogenannte Pfad- (*PATH*) und Reservierungsnachrichten (*RESV*). PATH–Nachrichten laufen vom Sender zum Empfänger des Datenstroms und nehmen dabei im Netzwerk genau denselben Weg wie die Daten, da sowohl Signalisierungs- als auch Nutzdaten in UDP–Datagrammen unter Nutzung der selben Routing–Informationen übertragen werden. RESV–Nachrichten laufen den entgegengesetzten Weg (siehe Abbildung 2.3). Jeder Empfänger kann hier individuell seine gewünschte Dienstgüte anfordern, d.h. nicht alle Empfänger eines Datenstroms müssen die gleiche Dienstgüte erhalten.[6]

[6]Im bereits erwähnten Video–On–Demand–System kann es für einen leistungsschwächeren Empfänger durchaus sinnvoll sein, eine geringere Dienstgüte anzufordern. Er wäre dann immerhin in der Lage, einen zwar qualitativ weniger hochwertigen, nichtsdestotrotz aber kontinuierlichen Film darzustellen. Alle anderen Empfänger bleiben von dieser Anforderung unbeeinflußt und können optimal auf die eigenen Bedürfnisse zugeschnittene Parameter vereinbaren.

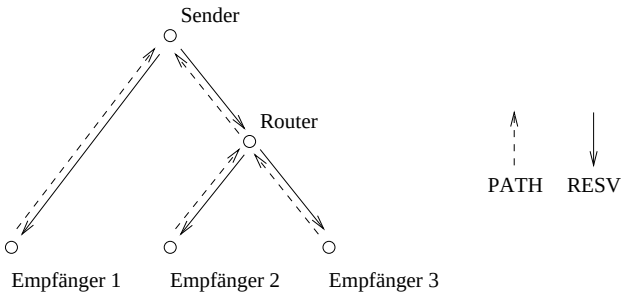


Abbildung 2.3: Nachrichtenfluß bei einer RSVP–Reservierung

RSVP etabliert keine festen Verbindungen zwischen Sender und Empfänger. Dies zeigt sich schon darin, daß das UDP–Protokoll zum Datentransport benutzt wird. Stattdessen werden periodisch PATH– und RESV–Nachrichten ausgetauscht, um bereits getätigte Reservierungen aufzufrischen bzw. nicht verfallen zu lassen. Falls über einen bestimmten Zeitraum keine RESV–Botschaften empfangen werden, können reservierte Ressourcen wieder freigegeben werden. Um eine Reservierung explizit wieder aufzuheben, kann aber auch eine spezielle Form der Signalisierungsbotschaft benutzt werden. Dieses Vorgehen wird als *Soft–State–Verfahren* bezeichnet. Eigenschaften werden nicht solange gespeichert, bis sie explizit wieder geändert werden, sondern sie behalten ihre Gültigkeit nur über einen bestimmten Zeitraum. Wenn bis zu deren Ablauf keine Bestätigung der Eigenschaften eintrifft, werden sie verworfen. Verlorengegangene Signalisierungsnachrichten können dabei — bei geeigneter Wahl der Auffrischungsintervalle — bis zu einem gewissen Maß toleriert werden.

Durch dieses Soft–State–Verfahren ist es möglich, vereinbarte Dienstgüteeigenschaften jederzeit zu ändern. Dazu müssen in der nächsten RESV–Nachricht lediglich Angaben über die jetzt gewünschte Dienstgüte enthalten sein, woraufhin der Empfänger dieser Botschaft seine Ressourcenreservierung anpaßt.

Reservierungsfreigaben können vom Sender und Empfänger durch spezielle Signalisierungsnachrichten oder durch das Ausbleiben von Auffrischungsnachrichten veranlaßt werden. Senderinitiierte Beendigungen löschen alle Reservierungen bis hin zu allen Empfängern, während vom Empfänger veranlaßte dagegen nur die dieses einen Empfängers auflösen.[7] Nach Beendigung einer Reservierung wird der entsprechende Anwendungsdatenstrom nach einem Best–Effort–Prinzip weitergeleitet.

Die Entwicklung gilt in diesem Bereich allerdings keinesfalls als abgeschlossen. Es haben sich beispielsweise einige Mängel gezeigt, die den Einsatz in den heute gebräuchlichen Weitverkehrsnetzen erschweren. Das wesentlichste Problem stellt sicherlich die schlechte Skalierbarkeit dar. Für jeden Anwen-

[7]Der Kunde eines Video–On–Demand–Systems kann aus einer Übertragung aussteigen, indem er als Empfänger die Reservierung beendet. Die anderen Kunden bleiben davon unbeeinflußt. Wenn aber beispielsweise der Sendebetrieb eingestellt wird, veranlaßt der Sender eine Aufhebung der Reservierungen, so daß alle Empfänger betroffen sind.

dungsdatenfluß müssen Reservierungen vorgenommen und verwaltet werden. Bei größeren Netzen dürfte dies die derzeit gebräuchlichen Router überfordern. Zudem erhöht der stetige Austausch von RESV– und PATH–Nachrichten die Netzbelastung. Deshalb empfiehlt die RSVP–Arbeitsgruppe den Einsatz derzeit nur in kleinen, begrenzten Netzen. [Bra99, S.123ff.]

2.2.2 Stream Protocol 2 (ST–II)

Das *Stream Protocol 2* unterscheidet sich hinsichtlich seiner Eigenschaften sehr stark von RSVP. Die Reservierungen gehen beispielsweise stets vom Sender aus. Im Gegensatz zu RSVP, welches UDP zum Datentransport nutzt, übermittelt ST–II sowohl Signalisierungsinformationen als auch Daten in einem eigenen Format. ST–II ist deshalb nicht auf die Kooperation mit einem bereits vorhandenen Standard–IP–Stack angewiesen, sondern kann diesen sogar ersetzen. Dann sind aber natürlich keine der "normalen" IP–Dienste mehr möglich.

Vor dem Nutzdatentransfer baut ST–II, genau wie beispielsweise TCP (siehe Abschnitt 2.1.2), eine Verbindung zwischen Sender und Empfänger auf, der die jeweils benötigten Ressourcen fest zugeordnet sind. In der Abbildung 2.4 ist der Ablauf eines solchen Verbindungsaufbaus skizziert: Der Sender alloziert die lokal erforderlichen Ressourcen und sendet bei Erfolg eine *CONNECT*–Nachricht,

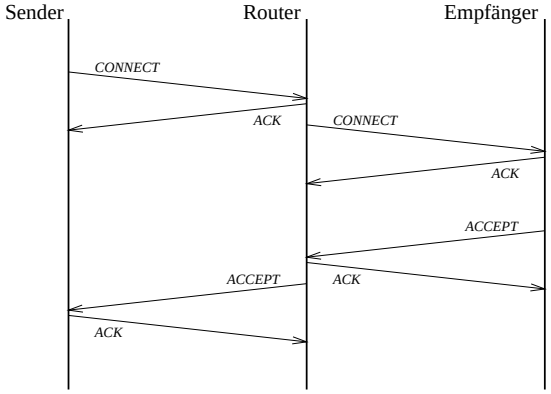


Abbildung 2.4: Verbindungsaufbau unter ST–II

welche die zu unterstützenden Dienstgüteparameter enthält, zum Empfänger. Dieser reserviert daraufhin ebenfalls die erforderlichen Ressourcen und quittiert Erfolg oder Mißerfolg. Routern ist es erlaubt, die geforderte Dienstgüte zu reduzieren und modifizierte CONNECT–Nachrichten weiterzusenden, sofern die Verfügbarkeit von Betriebsmitteln dies erfordert. Nach dem Eintreffen einer *ACCEPT*–Botschaft, die ihrerseits wieder eine Beschreibung der letztendlich bis zum Empfänger gelangten Dienstgüteanforderung beinhaltet, können vom Sender bzw. den Routern vorgenommene überreservierte Betriebsmittel wieder freigegeben werden. Die bestätigenden *ACK*-Nachrichten sollen lediglich die Zuverlässigkeit des Protokolls erhöhen.

ST–II schreibt vor, daß alle Empfänger eines Datenstroms mit der gleichen Dienstgüte bedient werden. Im Gegensatz zu RSVP ist es Empfängern nicht

möglich, individuelle Übertragungsparameter zu vereinbaren. Während der Lebensdauer einer ST–II-Verbindung können deren Übertragungseigenschaften aber, ähnlich wie unter RSVP, verändert werden. Davon sind dann aber stets sämtliche Empfänger betroffen.

ST–II paßt aufgrund seiner ähnlichen Eigenschaften sehr gut zum ATM–Reservierungskonzept. Trotzdem hat es eine weitaus geringere Verbreitung als beispielsweise RSVP erfahren. [Bra99, S.121ff.]

	RSVP	ST-II
senderinitiiert	–	×
empfängerinitiiert	×	–
verbindungsorientiert	– (Soft–States)	×
Standard–Transportprotokoll	× (UDP)	–
proprietäres Transportprotokoll	–	× (ST)
IP–Stack erforderlich	×	–
individuelle Dienstgüte je Empfänger	×	–

Tabelle 2.2: Vergleich der Eigenschaften von RSVP und ST–II

2.3 L4ATM

Im Rahmen seiner Promotion an der Technischen Universität Dresden hat sich Martin Borriss damit befaßt, eine Implementierung des ATM–Protokolls für L4 bzw. DROPS zu entwickeln [Bor99]. Als Grundlage der Arbeit diente die Implementation von ATM für LINUX [Alm97]. Um dem Anspruch von DROPS gerecht zu werden, bietet *L4ATM* natürlich eine erweiterte ATM–Schnittstelle an, die den Bedürfnissen eines parallelen Timesharing- und Echtzeitbetriebs gerecht wird. Sie erlaubt beispielsweise, Ressourcen des ATM–Systems bestimmten Verbindungen fest zuzuordnen. So läßt sich etwa eine Verbindung anlegen, der ein gewisser Anteil der verfügbaren Bandbreite exklusiv zur Verfügung steht. Ebenso ist die maximal zulässige Schwankung der Verzögerung bei der Verarbeitung der Datenpakete, *Jitter* genannt, sowie die *Latenzzeit*, also die Zeit, die zum Verarbeiten des Pakets im System benötigt werden darf, definierbar. Auch das Nichteinhalten der Verbindungsparameter durch Applikationen, die andere Verbindungen nutzen, führt nicht dazu, daß zugesagte Eigenschaften nicht gewährleistet werden können. Nicht durch Echtzeitübertragungen gebundene Transferkapazitäten stehen den Timesharing–Applikationen zur Verfügung.

Das L4ATM–System gliedert sich in jeweils eine Echtzeit– und eine Timesharing–Komponente. Die Echtzeitkomponente nutzt für ihre Arbeit mehrere Threads. Pro Verbindung wird ein *Workerthread* angelegt, der den Datenversand bzw. -empfang sowie das *Traffic Management* dieser Verbindung übernimmt. In der Timesharing–Komponente wird diese Aufgabe für alle Verbindungen von einem einzigen Thread erledigt. Daneben existiert ein weiterer *Servicethread* zur Behandlung sämtlicher zeitunkritischer Aufgaben wie dem

Verbindungsauf- und -abbau sowie dem Ändern bestimmter Verbindungseigenschaften. Ein sogenannter *(Pseudo–)Interruptthread*[8] wird vom Netzwerkadapter informiert, wenn Datenpakete eintreffen. Er leitet diese Pakete dann an den betreffenden Workerthread weiter und aktiviert diesen wieder, wenn er blockierend auf Daten wartet.

Die Implementation des entstandenen L4ATM–Systems gliedert sich in einen systemspezifischen und einen systemunabhängigen Teil. Zur Verbesserung der Portierbarkeit wurde vermieden, auf die systemspezifischen L4–Threads zurückzugreifen. Stattdessen entstand im Rahmen der Arbeit eine, allerdings nicht vollständige, POSIX–konforme Threadbibliothek [NBF96], welche bei der Implementation des L4ATM–Systems Verwendung fand.

Die Reservierung von Ressourcen wird von L4ATM in Zusammenarbeit mit dem *Ressourcenmanager* von L4 vorgenommen. Globale, systemweit verfügbare Ressourcen wie beispielsweise CPU–Zyklen werden dabei vom Ressourcenmanager vergeben. Ausschließlich von L4ATM genutzte Ressourcen, z.B. die verfügbare Netzwerk–Bandbreite, werden vom L4ATM–Subsystem selbst verwaltet. Beim Aufbau einer Verbindung werden zum Sicherstellen der gewünschten Dienstgüte erforderliche Puffer angelegt. Sie bleiben dieser Verbindung zugeordnet, solange sie existiert. Erst beim Abbau der Verbindung werden diese Puffer wieder freigegeben.

Falls eine ATM–Verbindung die ihr zugesicherte Dienstgüte überschreitet, so wird der betreffende Workerthread suspendiert. Erst am Beginn des nächsten ihm zugeordneten Intervalls kann er dann vom L4–Kern wieder aktiviert werden. Falls während der Suspendierung Daten auf dieser Verbindung eintreffen, kann der Interruptthread diese nur so lange weiterreichen, bis die dieser Verbindung zugeordneten Puffer erschöpft sind. Eine Verarbeitung durch den Worker findet ja in diesem Intervall nicht mehr statt. Diese die Kapazität übersteigende Pakete werden verworfen.

Um einen einfachen Zugriff auf die Dienste des L4ATM–Systems zu ermöglichen, wurde eine BSD–Socket–Schnittstelle (siehe Abschnitt 2.5) implementiert. Dies erlaubt es natürlich sämtlichen Sockets nutzenden Applikationen, über L4ATM zu kommunizieren. Diese Socket–Schnittstelle steht sowohl Echtzeit– als auch Timesharing–Applikationen zur Verfügung. Echtzeit–`read()`– und –`write()`–Operationen werden durch oben genannte Workerthreads abgewickelt. Um die Dienstgüte einer Echtzeitverbindung festzulegen, dient der Funktionsruf `setsockopt()`. Während diese Standard–Schnittstelle diverse Kopieroperationen nutzt, existieren einige zusätzliche Operationen (`read_ncp()`, `ncp_write()`), die sich stattdessen der L4–Operationen zum Einblenden von Speicherseiten in mehrere Adreßräume (*Mapping*) bedienen. Um eine Interaktion mit anderen DROPS–Komponenten zu ermöglichen, wurde deren Stream–Schnittstelle, welche *Flexpages* nutzt, für L4ATM nutzbar gemacht. Die bekannten Operationen `read()` und `write()` arbeiten dabei asyn-

[8] Üblicherweise löst der Netzwerkadapter beim Eintreffen von Daten pro Paket eine Unterbrechung aus. In der zugehörigen Unterbrechungsbehandlungsroutine wird dann das Paket weitergereicht.

chron. Eventuell erforderlicher Pufferspeicher wird von L4ATM entsprechend der vereinbarten Dienstgüte angefordert.

2.4 Der IP–Stack im Betriebssystem BSD

1983 wurde die Version 4.2 des an der University of California in Berkeley entwickelten UNIX–kompatiblen Betriebssystems BSD[9] veröffentlicht. Dieses System enthielt erstmals eine frei verfügbare Implementation der im Abschnitt 2.1 beschriebenen Internetprotokolle, die weite Verbreitung fand. Im Laufe der Jahre erfuhr dieser IP–Stack eine kontinuierliche Weiterentwicklung, in die viele Erkenntnisse wissenschaftlicher Forschungsarbeit einflossen. Nicht zuletzt dadurch entwickelte sich der IP–Stack dieses Betriebssystems quasi zur Referenzimplementierung für viele weitere Systeme. So basieren beispielsweise die IP–Implementierungen von IBMs *AIX* und Suns *Solaris* auf dem BSD–Programmcode. Daneben gibt es zahlreiche Literatur, die das BSD–System beschreibt (u.a. [MBKQ96]). Ausschließlich und sehr detailliert geht [WS95] auf den IP–Stack ein. Zudem ist der Quellcode unter den Bedingungen der *Berkeley Public License (BPL)* (siehe Anhang A) frei erhältlich und kann modifiziert werden. Die aktuelle Version *4.4BSD–Lite* kann beispielsweise unter [BSD] bezogen werden.

In den nächsten Abschnitten sollen zunächst die Systemruf–Schnittstelle und die Unterbrechungsbehandlung erläutert werden. Danach folgt eine Beschreibung der BSD–Kernel–Speicherverwaltung. Diese Konzepte werden vom IP–Stack verwendet, dessen Funktionsprinzip abschließend kurz dargelegt wird.

2.4.1 Systemrufe und Unterbrechungen

Wie in jedem modernen Betriebssystem gibt es auch in BSD zwei Betriebsarten: Sämtliche Anwendungsprogramme laufen grundsätzlich im *Benutzermodus* ab. Die Funktionalität in diesem Modus ist dabei stark eingeschränkt. Zugriffe auf besonders geschützte Bereiche, wie beispielsweise Systemdatenstrukturen oder Geräteregister, sind üblicherweise nur im besonders privilegierten *Kernmodus* erlaubt. Der Zugriff auf diesen Kernmodus ist nur über eine wohldefinierte Schnittstelle erlaubt, so daß lediglich bestimmte Operationen in dieser Betriebsart ausgeführt werden können.

Um nun diesen Kernmodus zu betreten, werden zu übergebende Argumente in definierte Übergaberegister geschrieben und eine Unterbrechung ausgelöst. Nach dem Ende der jeweiligen Unterbrechungsbehandlungsroutine stehen mögliche Ergebnisse wieder in bestimmten Registern zur Verfügung. Üblicherweise wird diese Systemruf–Schnittstelle in Bibliotheksfunktionen gekapselt, um deren Komplexität vor dem Nutzer zu verbergen.

Sobald ein Prozeß im Kernmodus abläuft, ergeben sich für ihn einige Konsequenzen. Wie bereits erwähnt, hat er nun Zugriff auf spezielle Ressourcen

[9] BSD steht dabei für *Berkeley Software Distribution*

des Systems. Zum anderen kann einem im Kernmodus befindlichen Prozeß der Prozessor nicht entzogen werden [MBKQ96, S.92]. Er bleibt demnach aktiv, bis er blockiert oder in den Benutzermodus zurückkehrt.

Unterbrechungsbehandlung

Sofern einer Unterbrechung eine Unterbrechungsbehandlungsroutine zugeordnet ist, wird sie unmittelbar nach dem Eintreten einer solchen Unterbrechung ausgeführt. Dazu wird der Kontext des momentan aktiven Prozesses[10] gesichert und die Unterbrechungsbehandlungsroutine ausgeführt. Nach deren Rückkehr wird der zum Unterbrechungszeitpunkt aktive Prozeß fortgesetzt.

In BSD gibt es verschiedene Prioritätsklassen für Unterbrechungen. Durch entsprechende Systemrufe lassen sich bestimmte Unterbrechungsklassen sperren, so daß Unterbrechungen geringerer Priorität unterdrückt werden. Tabelle 2.3 zeigt die einzelnen Prioritätsstufen in aufsteigender Ordnung.

Priorität	Sperrfunktion	Bedeutung
0	`spl0()`	alle Unterbrechungen sind erlaubt
1	`splsoftclock()`	Unterbrechungen des Zeitgebers werden gesperrt
2	`splnet()`	Unterbrechungsbehandlung im Netzwerk–Protokollstack ist gesperrt
3	`spltty()`	Unterbrechungen vom Terminal sind gesperrt
4	`splbio()`	Unterbrechungen von blockorientierten Geräten (Platte, Bandlaufwerk, ...) gesperrt
5	`splimp()`	Unterbrechungen von Netzwerkgeräten sind gesperrt
6	`splclock()`	Systemzeitgeber löst keine Unterbrechungen aus
7	`splhigh()`	alle Unterbrechungen sind blockiert

Tabelle 2.3: Unterbrechungs–Prioritätsklassen in BSD
(Es existieren zwei unterschiedliche Prioritätsstufen für Unterbrechungen vom Zeitgeber: Unterbrechungsbehandlungsroutinen der Prioritätsstufe 6 müssen vor dem Eintreffen der nächsten Zeitgeberunterbrechung abgearbeitet werden. Aufwendigere Routinen müssen in der Prioritätsstufe 1 ablaufen.)
(Quelle: [MBKQ96, S.58ff.])

Wenn der Anlaß für eine Unterbrechung gegeben ist, dessen Unterbrechungsniveau aber gerade gesperrt wurde, so wird das Auftreten der Unterbrechung so lange verzögert, bis diese wieder erlaubt ist.

[10] Auch wenn gerade kein Nutzerprozeß aktiv ist, so gibt es immer noch aktive Kernprozesse, zumindest den *Idle–Prozeß*, der dann aktiv ist, wenn kein weiterer Prozeß aktiv ist.

2.4.2 Kernel–Speicherverwaltung

BSD nutzt zum Datentransport innerhalb des Kernels eine Technik, die sich sogenannter *mbufs* als Datenbehälter bedient. Diese mbufs wurden dabei mit dem Ziel entworfen, die darin enthaltenen Daten mit geringem Aufwand weiterzugeben bzw. mit anderen mbufs zu verknüpfen. Diese Eigenschaft spielt

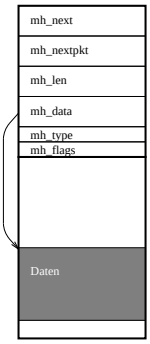


Abbildung 2.5: Grundlegender Aufbau eines *mbufs*

besonders bei der Protokollverarbeitung, beispielsweise im IP–Stack, eine große Rolle, da dort Daten mehrfach mit Headern versehen bzw. von diesen getrennt werden müssen. Statt nun diese Daten zwischen verschiedenen Speicherblöcken hin- und herzukopieren, bedienen sich mbufs einer anderen Technik.

Der grundlegende Aufbau eines mbuf ist in der Abbildung 2.5 dargestellt: Zusätzlich zu einem maximal 108 Byte großen Datenbereich und einigen Einträgen für Statusinformationen (Länge und Typ der enthaltenen Daten) existieren zwei Zeiger auf weitere mbufs. Einer dieser Zeiger läßt sich zum Aufbau von mbuf–Ketten nutzen (Abb. 2.6). Um beispielsweise einen Datenblock, der

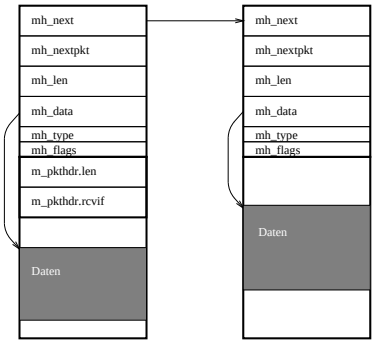


Abbildung 2.6: Verkettung zweier mbufs

sich bereits in einem mbuf befindet, mit einem Header zu versehen, so genügt es, den Header in einen weiteren mbuf zu kopieren und diesen lediglich vor dem Daten–mbuf einzuketten. Der zweite dieser Zeiger wird genutzt, um mbufs bzw. mbuf–Ketten in Listenform verwalten zu können.

Da ein mbuf nur maximal 108 Bytes Daten aufnehmen kann, ist diese Lösung für die Verwaltung größerer Datenmengen recht ineffizient. Deshalb wurde die

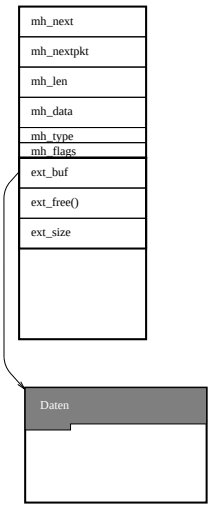


Abbildung 2.7: *mbuf* mit zusätzlichem Cluster

Möglichkeit vorgesehen, die Kapazität der mbufs durch sogenannte *Cluster* zu erweitern (Abb. 2.7). Dazu wird ein größerer zusammenhängender Speicherblock alloziert und dessen Startadresse im mbuf vermerkt. Sobald nun die Kapazität von zwei mbufs zur Aufnahme der entsprechenden Daten nicht ausreicht, wird die Aufnahmefähigkeit des ersten mbuf durch einen Cluster erweitert.[11] In der ursprünglichen Implementierung betrug die Größe eines Clusters 1024 Byte. Da nun beispielsweise ein Ethernet–Paket 1500 Bytes umfaßt, müßten zur Aufnahme eines solchen Pakets zwei Cluster benutzt werden. Um dies zu vermeiden, wählt man heute Clustergrößen von 2048 Bytes. Der Verwaltungsaufwand sinkt dadurch, und die Vergeudung von Hauptspeicher ist heutzutage wesentlich weniger problematisch.[12]

Da mbuf–Ketten gebildet werden können, sind mbufs auch nicht auf die Aufnahme von 2048 Byte Daten beschränkt. Die Verwendung von Clustern bringt aber noch einen weiteren Vorteil: Für jeden dieser Cluster wird ein Referenzzähler geführt. Dadurch ist es möglich, daß mehrere mbufs auf denselben Cluster verweisen. Der allozierte Speicherplatz wird erst dann freigegeben, wenn kein mbuf mehr diesen Cluster referenziert. Diese Technik erlaubt es, von verschiedenen Programmen genutzte Daten lediglich ein einziges Mal im Speicher zu halten.

Die Datenmengen, die ein mbuf bzw. Cluster aufnehmen kann, sind dabei nicht willkürlich gewählt. Vielmehr wurde bei der Entwicklung darauf geachtet,

[11] Sobald ein mbuf durch einen Cluster erweitert wurde, wird der in der mbuf–Datenstruktur selbst befindliche Platz aus Gründen der Einfachheit nicht mehr zur Datenspeicherung benutzt.

[12] Ein weiterer Grund ist Folgender: BSD sendet niemals mehr als einen Cluster pro TCP–Segment. Nun würde bei 1024 Bytes großen Clustern die maximal zur Verfügung stehende Netzwerkpaketgröße niemals ausgenutzt. Da die erreichbare Übertragungsrate steigt, je besser die Paketgröße ausgenutzt wird [Mog93], haben sich die Entwickler entschlossen, 2048 Bytes große Cluster zu verwenden. [WS95, S.33]

häufig vorkommende Datenmengen möglichst effektiv in mbufs verwalten zu können.

Zur Manipulation der mbufs stehen eine Vielzahl von Funktionen zur Verfügung, die es erlauben, mit geringem Programmieraufwand Daten in mbufs einzufügen bzw. aus diesen zu entnehmen.

2.4.3 IP–Stack

Der Netzwerk–Stack von 4.4BSD implementiert neben der IP–Protokollfamilie auch eine Reihe weiterer Kommunikationsprotokolle. Der Schwerpunkt der vorliegenden Arbeit liegt dabei aber lediglich auf der Implementierung der IP–Protokollfamilie.

- Das TCP/IP relativ ähnliche *Xerox Network System XNS* diente zur Verbindung von Geräten der Firma *Xerox*, vorwiegend Druckern und Dateiservern, über Ethernet. In heutiger Zeit wird es kaum noch genutzt.
- Die *OSI–Protokolle* [Ros90] wurden von der *International Standardization Organization (ISO)* mit dem Anspruch entwickelt, alle anderen Kommunikationsprotokolle durch diese universelle Technologie zu ersetzen. Diese Universalität brachte aber eine gewaltige Komplexität mit sich, so daß die OSI–Protokolle keine praktische Bedeutung erlangten.
- *UNIX–Domain–Protokolle* dienen dazu, eine Kommunikation zwischen Prozessen eines Systems (Interprozeßkommunikation, IPC) zu ermöglichen.

All diese unterschiedlichen Protokolle sind dabei über den gleichen Mechanismus der BSD–Sockets (siehe Abschnitt 2.5) zugänglich und nutzen dieselbe Schnittstelle zum Netzwerk. Die Aufteilung in drei Schichten ist in der Abbildung 2.8 zu sehen. Die Socketschicht (siehe Abschnitt 2.5) bietet den Klienten einen einheitlichen Zugang zu allen unterstützten Protokollen, während die Netzwerkschicht den Zugriff auf das zugrundeliegende Übertragungsmedium standardisiert. Diese Organisation hat zur Folge, daß beim Hinzufügen einer neuen Komponente innerhalb einer Schicht, also beispielsweise eines neuen Protokolls oder eines zusätzlichen Übertragungsmediums, die jeweils anderen Schichten ohne Änderung die zusätzliche Funktionalität sofort nutzen können.

Nachfolgend soll nun dargelegt werden, welche Operationen beim Senden bzw. Empfangen eines Paketes ausgeführt werden.

Senden: Der Klient führt einen `send()`–Systemruf (siehe Abschnitt 2.5) aus, mit dem er die zu sendenden Daten sowie die Adresse eines Empfängers an die Socketschicht übergibt. Nach einigen Plausibilitätsprüfungen kopiert diese Schicht Adresse und Daten in mbufs.[13] Anschließend wird die vom

[13] Gegebenenfalls wird der die Daten enthaltende mbuf mit Clustern erweitert.

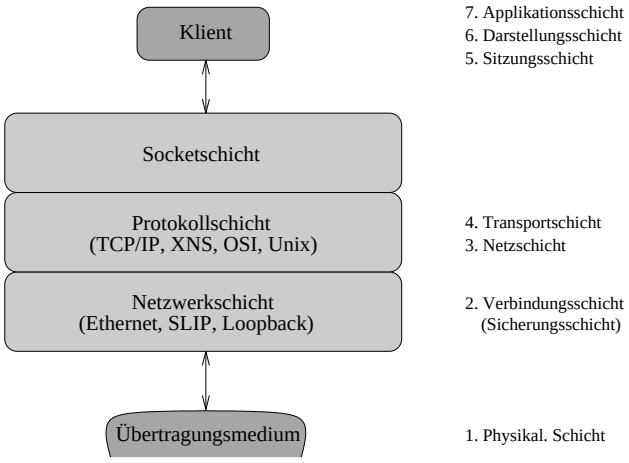


Abbildung 2.8: Funktionale Schichten des BSD–Netzwerkcodes
(Rechts sind zum Vergleich die sieben Schichten des OSI–Referenzmodells [PC93] gezeigt.)

Socket zu nutzende Protokollkomponente, also beispielsweise UDP, mit diesen mbufs als Parameter aktiviert.

Die Protokollkomponente muß nun den Daten–mbuf um den UDP– bzw. IP–Header ergänzen. Dazu alloziert sie einen weiteren mbuf, in den sie diese beiden Header hineinkopiert, und kettet diesen mbuf vor jenen, der die zu sendenden Daten enthält. Die Funktion `udp_output()` füllt nun den UDP–Header sowie die Elemente des IP–Headers, die zu diesem Zeitpunkt bereits bekannt sind. Die IP–Adresse des Empfängers kann beispielsweise schon hier eingetragen werden. Danach vervollständigt `ip_output()` den IP–Header und ermittelt das Netzwerkgerät, welches die Ausgabe auf das Übertragungsmedium übernehmen soll. Falls die Größe des IP–Pakets die vom Netzwerkgerät maximal verarbeitbare Paketgröße übersteigt, erfolgt hier zunächst noch die Fragmentierung. Danach wird die Daten–mbuf–Kette an die Ausgaberoutine des zuständigen Netzwerkgerätes gegeben. Im Falle eines Ethernet–Adapters ist dies die Funktion `ether_output()`.

`ether_output()` transformiert die vom Klienten angegebene IP–Adresse des Empfängers in eine Ethernet–Adresse, indem es gegebenenfalls ARP–Nachrichten[14] auf dem Netzwerk generiert. Danach fügt es einen Ethernet–Header, in welchem die Ethernet–Adressen des Senders bzw. Empfängers stehen, vor den IP–Header und fügt diese mbuf–Kette in die Liste der Pakete ein, die am entsprechenden Netzwerkadapter darauf warten, gesendet zu werden. Sofern der Netzwerkadapter nicht beschäftigt ist, wird dessen Senderoutine angestoßen, die die Daten aus den verketteten mbufs in einen zusammenhängenden Sendepuffer kopiert und auf das Übertragungsmedium gibt.

[14] *ARP (Address Resolution Protocol)* — Protokoll zur Umwandlung von IP–Adressen (siehe Abschnitt 2.1.1) in eine zugehörige Hardware–Adresse, also z.B. eine Ethernet–Adresse

Empfangen: Im Gegensatz zum Senden, das vom Klienten durch einen Systemruf veranlaßt wird, geschieht der Empfang asynchron beim Auslösen einer Unterbrechung durch den Netzwerkadapter. In der zugehörigen Unterbrechungsbehandlungsroutine wird das komplette empfangene Paket in einen mbuf bzw. eine mbuf–Kette kopiert und diese an eine universelle Empfangsroutine übergeben. In diesem das Ethernet nutzenden Beispiel ist dies die Funktion `ether_input()`. Diese ermittelt aus dem Inhalt des empfangenen Pakets, welchen Typs das Paket ist. Ein IP–Paket wird so in die Liste der empfangenen IP–Pakete eingefügt. Danach wird eine weitere Unterbrechung ausgelöst, in deren Behandlungsroutine die IP–Empfangsverarbeitung abläuft.

Die IP–Empfangsverarbeitungsroutine `ip_input()` entnimmt solange jeweils ein Paket aus der Empfangsliste und verarbeitet es, bis die Liste abgearbeitet ist. Sie vergleicht die Prüfsumme, bearbeitet die im IP–Paket enthaltenen Optionen und führt einige Plausibilitätsprüfungen durch. Wenn das Paket für einen anderen Host bestimmt ist, wird es an diesen weitergeleitet, sofern das System die Funktion eines Routers übernehmen kann.[15] Ist das Paket für dieses System bestimmt, ermittelt `ip_input()`, welche nachfolgende Behandlungsroutine aufzurufen ist. Im Falle eines UDP–Pakets wäre dies `udp_input()`. `udp_input()` prüft die Gültigkeit des UDP–Headers und ermittelt anhand der IP–Adresse des Absenders und der verwendeten Portnummer (siehe Abschnitt 2.5), welchem Prozeß diese Daten zugestellt werden müssen. Wurde dieser gefunden, so werden zunächst die IP– und UDP–Header des Pakets entfernt, danach die empfangenen Daten sowie die Adresse des Absenders in mbufs kopiert und der Socketschicht übergeben. Diese kettet die mbufs in eine Liste der empfangenen Pakete des jeweiligen Sockets ein und weckt den zugehörigen Prozeß auf.

Der empfangende Prozeß, der in einem `recv()`–Systemruf blockierte, wird aktiviert, `recv()` kopiert Adreßangaben sowie Daten in die vom Prozeß bereitgestellten Puffer und kehrt zum aufrufenden Programm zurück.

Die Spezifikationen der IP–Protokollfamilie in [Pos81a, Pos81b, Pos80] geben vor, welche Eigenschaften eine IP–Implementierung zu erfüllen hat. Im Rahmen dieser Vorgaben sind im IP–Stack von BSD viele verbesserte Verfahren verwirklicht, ohne den Spezifikationen zuwiderzulaufen. Einige von ihnen sollen im Folgenden kurz dargelegt werden.

- ***"silly window syndrome"*** [MBKQ96, S.469f.]: Ein System könnte mehrere kleinere Pakete versenden, statt diese Daten zu sammeln, bis ein größeres Datagramm effektiver verschickt werden kann. Diesem Problem kann in BSD sowohl vom Sender als auch vom Empfänger begegnet werden. Der Sender verzögert das Senden von Daten, bis mindestens ein komplettes Paket gefüllt werden kann oder eine bestimmte Zeitspanne auf neue Daten gewartet wurde. Der Empfänger signalisiert dem Sender, daß

[15] Falls die Router–Funktionalität nicht aktiviert wurde, wird das Paket verworfen.

er nicht in der Lage ist, neue Daten zu empfangen, sofern die von ihm empfangbare Datenmenge kleiner als ein Paket oder kleiner als $\frac{1}{4}$ seiner Empfangspuffergröße ist.

- ***"avoidance of small packets"*** [MBKQ96, S.470f.]: Beim Datentransfer in Netzwerken tendiert die Paketgröße zu zwei Extremen: Einerseits werden beim Massendatentransfer größtmögliche Pakete angestrebt, andererseits erfordern interaktive Dienste schnelle Antwortzeiten und damit kleine Pakete. Ein Vorschlag zur Lösung dieses Problems war, vor dem Senden etwa 50 – 100 ms zu warten und dann alle bis dahin angefallenen Daten zu versenden. In schnellen Netzen führte dies aber zu langen Latenzzeiten im interaktiven Betrieb. In BSD ist deshalb ein in [Nag84] vorgestellter Ansatz verwirklicht, der sich durch Eleganz und selbsttätiges Anpassen an die Transporteigenschaften auszeichnet: Das erste Paket wird unmittelbar nach dem Eintreffen der ersten Daten abgeschickt. Solange nun dieses Paket nicht bestätigt wurde, sendet der IP–Stack nur dann neue Daten, wenn damit ein komplettes Paket gefüllt werden kann. Wenn die Bestätigung eintrifft, werden alle bis dahin angefallenen Daten abgeschickt.

- ***"delayed acknowledgments"*** [MBKQ96, S.471f.]: Unter TCP müssen nicht nur Pakete zum Nutzdatentransfer übermittelt werden, sondern beispielsweise auch Bestätigungen empfangener Pakete. Statt nun jedes einzelne Paket separat zu bestätigen, kann gewartet werden, bis eine "geeignete Menge" an Daten eingetroffen ist, die dann gemeinsam durch ein einziges Datagramm bestätigt wird.

- ***"slow start"*** [MBKQ96, S.472f.]: Viele TCP–Verbindungen erstrecken sich über mehrere miteinander gekoppelte Netzwerke. Wenn diese Netze unterschiedliche Übertragungsraten haben, dann wird der Router, der in ein langsameres Netz hineinführt, oftmals mit mehr Paketen "überschwemmt", als er weiterzureichen imstande ist. Infolgedessen kann er gezwungen sein, bei Pufferüberlauf Pakete zu verwerfen. Diese Pakete wären dann erneut zu senden, was den Datendurchsatz herabsetzen würde. Anstatt nun sofort mit "voller Leistung" Daten zu senden, schickt der Sender zunächst nur ein Paket ab. Wenn dieses erste Paket bestätigt wurde, verdoppelt er jeweils die Anzahl der von ihm verschickten Pakete. Sofern nun eine Bestätigung ausbleibt, so ist mit einiger Sicherheit anzunehmen, daß die von der Verbindung verarbeitbare Datenrate erreicht ist.

2.5 BSD–Sockets

Um die Programmierung der Netzwerkschnittstelle zu vereinfachen, wurde mit 4.2BSD (siehe Abschnitt 2.4) das Konzept der *Sockets* eingeführt. Mittlerweile hat sich diese Schnittstelle als Standard etabliert.

Ein Socket ist dabei ein bidirektionaler Kommunikationsendpunkt, der durch durch folgendes Tupel eindeutig bestimmt ist:

$$S = \{Adressfamilie, Adresse, Port, Typ\}$$

- Sockets sind nicht nur darauf ausgelegt, die Internet–Protokolle (TCP, UDP, ...) zur Datenübertragung zu benutzen. Ebenso werden die ISO–Protokolle [Ros90] und eine Reihe weiterer unterstützt. *Adressfamilie* bestimmt deshalb das zu nutzende Übertragungsprotokoll und damit den Typ der Adresse.
- *Adresse* ist dabei die des Rechners, auf dem das Socket erzeugt wurde.[16] Bei Nutzung der Internet–Protollfamilie (siehe Abschnitt 2.1) kommen hier IP–Adressen (siehe Abschnitt 2.1.1) zum Einsatz.
- Ein *Port* ist lediglich ein ganzzahliger Wert, der der Identifizierung bzw. Unterscheidung mehrerer Verbindungen auf einem Rechner dient. (Wenn Nachrichten lediglich an eine Adresse geschickt werden würden, so müßten alle Sockets dieses Rechners diese Botschaft empfangen. Um diese Auflösung feiner gestalten zu können, werden Portnummern benutzt.)
- Der *Typ* spezifiziert die Art der Verbindung näher. Die Internet–Protolle (Abschnitt 2.1) erlauben beispielsweise sowohl die Verwendung verbindungsloser (UDP) als auch verbindungsorientierter (TCP) Datenübertragung. Diese Eigenschaft ist im Typ vermerkt.

Ein Datentransfer kann zwischen zwei Sockets also nur dann stattfinden, wenn folgende Bedingungen erfüllt sind:

1. *Adressfamilie* und *Typ* beider Sockets müssen übereinstimmen.
2. *Adresse* und *Port*, an die die Daten geschickt werden, müssen den jeweiligen Werten des empfangenden Sockets entsprechen.

Nahezu jedes Betriebssystem bietet mittlerweile diese Schnittstelle zu seiner Netzwerkfunktionalität an. Unter UNIX–kompatiblen Betriebssystemen ist das Socketsystem dabei in das normale Dateisystem eingebunden. Dies hat zur Folge, daß die üblichen Dateioperationen wie `open()`, `close()`, `read()` und `write()` auf Sockets angewendet werden können. Darüberhinaus sind folgende Operationen auf Sockets definiert:

- `socket()` legt ein Socket an und liefert einen Deskriptor zurück. Dabei wird allerdings noch keine Verbindung aufgebaut.
- `bind()` assoziiert das Socket mit einer Adresse sowie einem Port. Erst danach kann eine Verbindung zu diesem Socket aufgebaut werden.

[16] Es ist denkbar, daß ein Rechner über mehrere Netzwerkgeräte verfügt. Dann sind ihm in der Regel auch mehrere Adressen zugewiesen. Die Adresse des Sockets ist dabei die desjenigen Netzwerkinterfaces, über welches das Socket mit anderen Rechnern kommuniziert.

- `connect()` stellt eine Verbindung zum spezifizierten Socket her.
- `listen()` zeigt an, daß der rufende Prozeß ab sofort empfangsbereit ist und bestimmt, wieviele eintreffende Nachrichten zwischengespeichert werden können.
- `accept()` blockiert den rufenden Prozeß solange, bis eine Nachricht für dieses Socket eintrifft.
- `send()`, `sendto()` und `sendmsg()` dienen dazu, eine Nachricht an das spezifizierte Socket zu senden.
- `recv()`, `recvfrom()` und `recvmsg()` blockieren, bis eine Nachricht an diesem Socket vorliegt und liefern diese zurück. Es ist ebenso möglich, die Funktion nicht blockieren zu lassen und stattdessen einen Fehler anzuzeigen, wenn keine Nachrichten vorliegen.
- `shutdown()` beendet eine bestehende Verbindung wieder.

Daneben existieren einige weitere Funktionen, die es erlauben, Eigenschaften des Sockets abzufragen bzw. zu ändern.

2.6 Der IP–Stack im Betriebssystem Linux

Der IP–Stack des Betriebssystems LINUX ist in seiner grundsätzlichen Funktionsweise dem von BSD recht ähnlich [BBD+97, S.225ff.]. Obwohl LINUX komplett neu programmiert wurde, sind durch die Protokollspezifikationen sicher gewisse Grenzen vorgegeben. In einigen Punkten unterscheidet sich der in LINUX gewählte Ansatz aber doch von der Lösung in BSD. So hat man einen neuen Speicherverwaltungsdatentyp definiert, der im Gegensatz zu den im Abschnitt 2.4.2 beschriebenen mbufs nur für die Pufferverwaltung im Netzwerkcode verwendet wird. Abbildung 2.9 zeigt die wichtigsten Bestandteile dieses als *sk_buff* bezeichneten Datentyps und seine Einbindung in das System: Jeder sk_buff enthält einen Verweis auf ein Socket (siehe Abschnitt 2.5), so daß ein eindeutiger Bezug zwischen Socket und dem gespeicherten Datenblock hergestellt werden kann.[17] Zusätzlich sind Vorkehrungen getroffen worden, um Elemente dieses Datentyps zu Listen aneinanderzuketten. Um zu kennzeichnen, von welchem Netzwerkgerät das im sk_buff gespeicherte Paket stammt bzw. zu welchem es geschickt werden muß, ist ein Verweis auf dieses Gerät vorhanden.

Die Nutzdaten selbst werden entweder direkt in einem solchen sk_buff gespeichert, oder aber es existiert ein Verweis auf einen weiteren sk_buff, der dann seinerseits nur Daten enthält. Das letztere Verfahren ermöglicht es mittels eines Referenzzählers, beim Kopieren von Datenblöcken lediglich einen neuen Verweis auf den ursprünglichen sk_buff anzulegen und dessen Referenzzähler zu erhöhen

[17] Da in der von LINUX genutzten Socket–Datenstruktur ein Verweis auf einen sk_buff enthalten ist, kann eine Menge von Datenblöcken einem Socket direkt zugeordnet werden. Die Zuordnung ist deshalb auch von einem Socket ausgehend möglich.

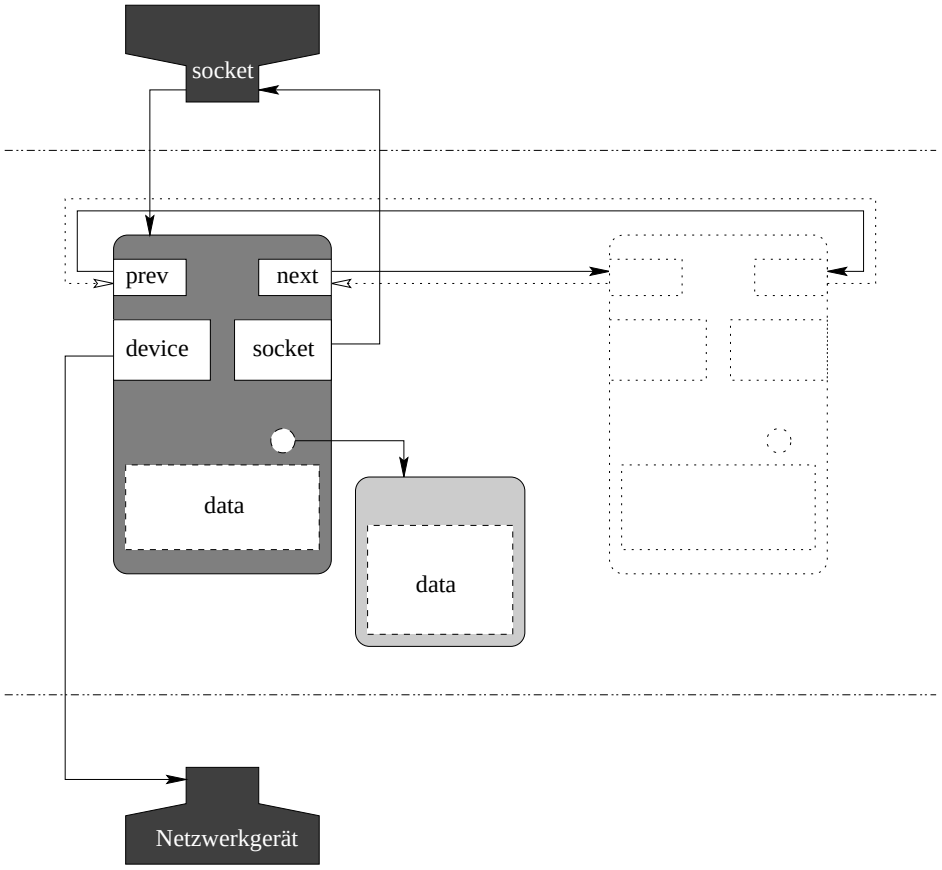


Abbildung 2.9: Wesentliche Bestandteile eines *sk_buff*

[BBD+97, S.234ff.]. Dieses Vorgehen ähnelt sehr stark der Nutzung von mbufs und Clustern unter BSD (siehe Abschnitt 2.4.2).

Während BSD beim Zusammenfügen der einzelnen Paketbestandteile mehrere mbufs verkettet, um Kopieroperationen zu vermeiden, geht LINUX einen anderen Weg. Da die Größen der einzelnen Paketbestandteile, also der Protokollheader und die maximale Nutzdatenkapazität, bereits im Vorfeld bekannt sind, wird ein einziger sk_buff alloziert, der in der Lage ist, all diese Daten aufzunehmen. Die jeweiligen Daten werden dabei an die entsprechende Stelle in diesem Speicherblock kopiert, so daß sich am Ende der Bearbeitung das komplette Paket zusammenhängend im Speicher befindet.

Kapitel 3

Entwurf

3.1 IP–Stack

Um die Funktionalität der im Abschnitt 2.1 beschriebenen Internetprotokolle für L4 bzw. DROPS zu erschließen, kann einerseits eine komplette Neuprogrammierung eines IP–Stacks erfolgen. Dabei ist es möglich, die speziellen Eigenschaften des Mikrokerns L4, vorrangig seinen extrem schnellen Mechanismus zur Interprozeßkommunikation, zu berücksichtigen. Das Ergebnis wäre ein optimal an die Fähigkeiten und Bedürfnisse von DROPS angepaßter IP–Stack. Nicht außer acht zu lassen ist aber die recht hohe Komplexität der Internetprotokolle, vor allem von TCP (siehe Abschnitt 2.1, bes. 2.1.2). Wie im Abschnitt 2.4.3 dargelegt, gibt es eine Reihe von Verfahren, die die Arbeit dieser Protokolle verbessern. Dabei laufen diese Verfahren den Protokolldefinitionen nicht zuwider, sind aber in diesen nicht explizit aufgeführt. Da die Arbeit eines IP–Stacks durch den Einsatz dieser Verfahren in der Praxis wesentlich verbessert wird, ergäbe sich zur Erzielung optimaler Leistung die Notwendigkeit, solche Algorithmen in der Implementation zu berücksichtigen. Dies würde zu einer wesentlichen Erhöhung der Komplexität der Implementierung führen.

Als Alternative bietet sich die Portierung einer bewährten Lösung an. Bei einer Neuimplementierung unvermeidliche Fehler können so vermieden werden. Gegebenenfalls müssen allerdings Kompromisse in Bezug auf die Nutzung des zugrundeliegenden L4–Systems getroffen werden. Die Übertragungsleistung eines Systems wird aber nicht ausschließlich von der Leistung der IP–Implementierung, sondern auch durch die des Übertragungsmediums bestimmt. Aus diesem Grund ist in meinen Augen ein gewisses Maß an Kompromißbereitschaft bei der Portierung durchaus zu vertreten.

Die im Abschnitt 2.4.3 beschriebene Implementierung des IP–Stacks des Betriebssystems 4.4BSD hat in der Praxis weite Verbreitung gefunden. Wie bereits erwähnt, stellt dieses System die Referenzimplementierung der Internetprotokollfamilie dar. Darüberhinaus zeichnet sich 4.4BSD durch eine außerordentlich umfangreiche und gute Dokumentation aus. Zahlreiche Bücher beschreiben jeden Aspekt der Implementierung. Der IP–Stack von LINUX schneidet dagegen

schlechter ab. Die Versorgung mit vergleichbarer Literatur ist derzeit noch wesentlich schlechter. Durch die rasante Weiterentwicklung und dem zuletzt sehr starken Popularitätsgewinn von LINUX sind hier zukünftig sicher noch Verbesserungen zu erwarten. Aufgrund der momentanen Situation habe ich mich allerdings für eine Portierung des IP–Implementierung von 4.4BSD entschieden.

3.1.1 Prozeßkonzept

Wie in vielen monolithischen Betriebssystemen läuft auch unter 4.4BSD der Netzwerkprotokoll–Programmcode im Kernmodus ab. Um diese Funktionalität zu nutzen, führen Nutzerprozesse einen Systemruf (siehe Abschnitt 2.4.1) aus, woraufhin die die Netzwerkfunktionalität erbringende Systemfunktion im besonders privilegierten Kernmodus abläuft. Sobald die Systemfunktion endet, wird der Kernmodus wieder verlassen. Dies hat zur Folge, daß sämtliche Daten des Nutzerprozesses für die Systemfunktion sicht- und modifizierbar sind.

Unter L4 bzw. DROPS läßt sich dieses Verhalten relativ einfach nachbilden. Der gesamte Netzwerkcode wird dazu in einer Bibliothek zusammengefaßt, welche zu der jeweiligen Anwendung hinzugelinkt wird. Dieser Einfachheit stehen einige wesentliche Nachteile entgegen. Zum einen würde diese Bibliothek und damit die jeweiligen Anwendungen einen recht großen Umfang annehmen. Da momentan von DROPS keine *shared libraries* unterstützt werden, wäre mit jeder entsprechenden Applikation eine neue Instanz dieses hinzugelinkten Netzwerkcodes im Arbeitsspeicher zu halten. Zudem teilen sich die verschiedenen Instanzen des Netzwerkcodes gemeinsame Daten. Diese müßten dann durch einen globalen Koordinator verwaltet und allen Nutzern durch geeignete Mechanismen wie beispielsweise *gemeinsamem Speicher* zur Verfügung gestellt werden. Dies wird spätestens dann zu einem Problem, wenn unterschiedliche Versionen des IP–Stacks Verwendung finden. Die gemeinsam genutzten Daten müßten immer demselben Format gehorchen bzw. der Zugriff auf diese Daten müßte immer gemäß demselben wohldefinierten Protokoll erfolgen. Sobald Änderungen an diesem Protokoll vorgenommen werden müssen, sind sämtliche den IP–Stack nutzende Anwendungen neu zu übersetzen.

Statt jede Applikation um diesen netzwerkspezifischen Programmcode zu ergänzen, bietet es sich an, ein unter DROPS sehr populäres Mittel anzuwenden: Die komplette Netzwerkfunktionalität wird in einen als *Server* bezeichneten separaten Task verlagert. Als *Klient* auftretende Applikationen wenden sich per IPC an diesen Server, lassen ihre Anforderungen bearbeiten und empfangen mögliche Ergebnisse von diesem wiederum per IPC. Diese Kommunikation mit dem Server läßt sich mit dem Mechanismus der Systemfunktionsrufe in monolithischen Systemen vergleichen. Der Nutzer übergibt zu diesem Zeitpunkt die Kontrolle an eine andere Systemkomponente — Server oder Systemkern — und erlangt sie erst dann wieder, sobald die geforderte Funktionalität erbracht ist. Der gesamte Programmcode wird so nur einmal im Speicher gehalten, und der Zugriff auf die gemeinsamen Daten läßt sich wesentlich einfacher realisieren. Problematischer ist die potentielle Nutzung von Daten aus dem Adreßraum des Klienten. Während dies sowohl bei Systemrufen als auch bei der Verwendung

als Bibliothek kein Problem darstellt, müssen beim Klient–Server–Ansatz diese Daten mittels eines geeigneten Mechanismus, beispielsweise durch IPC oder gemeinsamen Speicher, dem Server zugänglich gemacht werden. Darüberhinaus vermeidet die Implementierung als Server ein Sicherheitsproblem: Die Bibliothek könnte von potentiellen Angreifern modifiziert werden, um unzulässige Aktionen durchzuführen. Um dies zu verhindern, wären wiederum geeignete Zugriffskontrollmechanismen einzuführen. Stattdessen kann der Administrator eines DROPS-Systems einen IP–Server starten, von dessen Korrektheit er sich überzeugt hat. Manipulationen an dessen Code und daraus resultierende Sicherheitslücken sind dadurch ausgeschlossen.

Um am BSD–Netzwerkcode relativ wenig Änderungen vornehmen zu müssen, empfiehlt es sich, diesen in ein das BSD–Prozeßkonzept nachbildendes System einzubetten. Aufbauend auf den Erfahrungen der Arbeit von Martin Borriss (siehe Abschnitt 2.3) war die erste Idee, das von ihm entwickelte L4–Pthread–System zu nutzen. Sobald der IP–Server die Anfrage eines Klienten erhält, erzeugt er einen neuen Thread, welcher dann nur für die Bearbeitung dieser Anfrage zuständig ist. Sofern ein blockierender Zustand bei der Abarbeitung erreicht wurde, wird der entsprechende Thread einfach suspendiert. Der Scheduler aktiviert danach automatisch den nächsten arbeitsbereiten Thread. Um dem Arbeitsmodell des BSD–Kerns recht nahe zu kommen, müßten alle diese Threads nach dem Schedulingverfahren *FIFO*[1] behandelt werden, da einem sich im Kernmodus befindenden Programm niemals der Prozessor entzogen wird.[2] Erst wenn der aktive Thread blockiert, wird der nächste aktiviert.

Leider ist die L4–Pthread–Implementierung in der derzeitigen Version nicht in der Lage, ein solches FIFO–Scheduling durchzuführen. Eine weitere wesentliche Limitierung ergibt sich aus der Tatsache, daß gegenwärtig nur maximal 32 Pthreads unterstützt werden. Dies hätte zur Folge, daß nicht mehr als 32 Anfragen zur gleichen Zeit behandelt werden könnten.[3] Eine Weiterentwicklung des Pthread–Systems könnte diese Situation aber verbessern. Da das Scheduling ohne zusätzliche Maßnahmen nicht nach dem Verfahren FIFO abläuft, sind darüberhinaus Zugriffe auf globale Daten durch geeignete Synchronisationsmechanismen, die vom Pthread–Paket angeboten werden, zu serialisieren. Dies würde aber weitere Eingriffe in den BSD–Netzwerkcode erforderlich machen.

Statt jeweils einen Pthread pro Anfrage zu verwenden, könnte natürlich auch je ein L4–Thread genutzt werden. Leider ist die Anzahl verfügbarer Threads unter L4 derzeit auf 128 limitiert, was die Anzahl gleichzeitiger Anfragen wiederum begrenzt. Sobald diese Threadlimitierung unter L4 entfällt und ein FIFO–Scheduling für Threadmengen möglich ist, bietet sich damit aber eine interessante Lösungsmöglichkeit, welche insbesondere die Synchronisation der einzelnen Anfragen wesentlich erleichtert.

[1] Beim Scheduling nach FIFO bleibt ein Thread bzw. Prozeß so lange aktiv, bis er blockiert oder von sich aus den Prozessor abgibt. Ihm kann der Prozessor nicht entzogen werden.

[2] Dieses FIFO–Scheduling bezieht sich nur auf die Menge der den Netzwerkcode abarbeitenden Threads. Alle anderen Threads im System können natürlich davon unbeeinflußt bleiben.

[3] Unter der Annahme, daß einige Pthreads für Synchronisierungs- bzw. Steuerungsaufgaben benötigt werden, verringert sich diese Zahl sogar noch weiter.

Um all diese Limitierungen zu umgehen, bietet es sich an, die Bearbeitung der Anfragen in einem einzigen Thread auszuführen. Dazu ist ein Scheduler zu entwickeln, der in der Lage ist, mehrere Kontexte in diesem einen L4–Thread ausführen zu lassen. In jedem dieser Kontexte wird eine Anfrage bearbeitet, deren Anzahl dadurch nicht mehr begrenzt ist. Sofern dieser Thread bei der Bearbeitung des einer Anfrage entsprechenden BSD–Codes einen blockierenden Zustand erreicht, erfolgt ein Kontextwechsel. Aufgrund des ausschließlich verwendeten FIFO–Verfahrens kann der Scheduler relativ einfach aufgebaut sein.

Im verwendeten BSD-Code werden die Funktionen `tsleep()` und `wakeup()` genutzt, um den aufrufenden Prozeß zu suspendieren bzw. suspendierte Prozesse wieder der Menge der bereiten Prozesse hinzuzufügen. `tsleep()` stellt damit genau den Punkt dar, in dem ein Kontext den Prozessor abgibt und der nächste Kontext aktiviert wird. Die L4–Portierung muß demzufolge lediglich diese Funktion derart modifizieren, daß die folgenden Schritte ausgeführt werden:

1. Zunächst wird der aktuelle Registersatz auf den aktiven Stack gerettet.
2. Danach werden der aktuelle *Programm-* und *Stackzeiger* in einer Warteschlange gespeichert.
3. Nachdem der neu zu aktivierende Kontext bestimmt ist, werden die zugehörigen Programm- und Stackzeiger gesetzt.
4. Zum Schluß wird der Registersatz vom nun aktiven Stack wiederhergestellt.

`wakeup()` muß dann lediglich die betroffenen Kontexte in der erwähnten Warteschlange ausfindig machen und als nicht mehr blockiert markieren, so daß sie bei der nächsten Auswahl bereiter Kontexte berücksichtigt werden.

Ein Problem ergibt sich, wenn nach dem Senden des Ergebnisses einer Anfrage zuvor suspendierte, mittlerweile aber wieder bereite Kontexte existieren. Um zu vermeiden, daß der IP–Server in diesem Zustand möglicherweise längere Zeit auf neue Anfragen wartet, ohne diese wartenden Kontexte abzuarbeiten, muß vor jedem Warten geprüft werden, ob Kontexte bereit sind. Gegebenenfalls wird dann einer von diesen aktiviert, statt auf neue Anfragen zu warten.

3.1.2 Speicherverwaltung

Wie im Abschnitt 2.4.2, Seite 17, dargelegt, benutzt BSD sogenannte mbufs zum Transport von Daten innerhalb des Kernels. An dieser Stelle wurde auch dargelegt, daß diese mbufs vordergründig mit dem Ziel entwickelt wurden, effizient mit dem zur Verfügung stehenden Speicher umzugehen. Eventuelle Geschwindigkeitsnachteile haben die Entwickler dabei in Kauf genommen, da seinerzeit wesentlich geringere Hauptspeichermengen gebräuchlich waren.

Ein möglicher Ansatz wäre nun, auf die Verwendung dieser mbufs zu verzichten und für den Datentransport auf andere geeignete Techniken zurückzugreifen. Ein wesentlicher Vorteil von mbufs ist, daß Speicherblöcke in Listenform miteinander verknüpft werden können, so daß Kopieroperationen zum Zwecke des Zusammenfügens mehrerer Speicherblöcke durch einfaches Verketten zweier mbufs ersetzt werden können. Moderne Hardware und stetig erhöhte Hauptspeichermengen erlauben es mittlerweile aber, etwas großzügiger mit dem zur Verfügung stehenden Speicher umzugehen. Statt mehrere mbufs aneinanderzuketten (in die die jeweiligen Daten ja auch erst hineinkopiert werden müssen), ist denkbar, einen einzigen größeren Speicherblock zu allozieren. Da man bei der Bearbeitung der IP–Pakete über recht gute Kenntnisse verfügt, welche maximalen Datenmengen letztendlich zusammengefügt werden müssen, können beim Umkopieren *geeignete* Stellen ausgewählt werden, so daß für hinzuzufügende Daten genügend Raum vorhanden ist. Der komplette bei mbufs anfallende Verwaltungsaufwand kann also vermieden werden. Im Bild 3.1 ist gezeigt, wie sich dabei ein solcher zusammenhängender Speicherblock füllt.

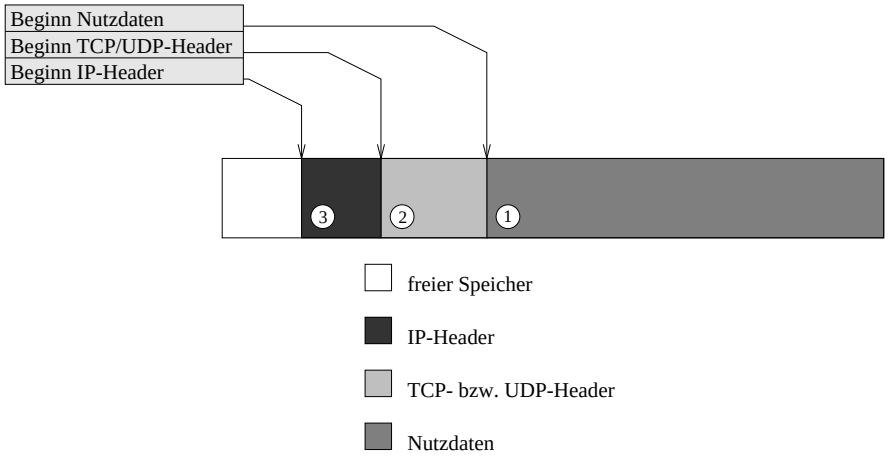


Abbildung 3.1: Aufbau eines Speicherblocks zum Ersatz eines mbufs
Zunächst wird ein Speicherblock alloziert, dessen Größe sich aus der Summe der größtmöglichen Nutzdatenmenge sowie der maximalen TCP–/UDP– und IP–Headerlänge ergibt. Die Nutzdaten werden so in den Speicherblock eingefügt, daß diese mit dem Pufferende abschließen. In den nächsten Protokollstufen wird dann der entsprechende Header unmittelbar vor die bereits im Puffer befindlichen Daten gesetzt. (Möglicherweise ungenutzter Speicher bleibt deshalb am Anfang des Speicherblocks frei.

Die eben skizzierte Lösung ist im Hinblick auf Geschwindigkeit sicher der Verwendung von mbufs vorzuziehen. Dann müßten aber im kompletten BSD–TCP/IP–Quelltext sämtliche Zugriffe auf mbufs durch entsprechende alternative Methoden ersetzt werden. Dies ist zwar denkbar, würde aber angesichts des Umfangs des Quellcodes einen relativ großen Aufwand nach sich ziehen.

Stattdessen soll in meiner Implementierung das System der mbufs nachgebildet werden. Der Implementierungsaufwand dieser Lösung ist gegenüber dem

Ersatz von mbufs wesentlich geringer. Zudem verringert sich dadurch die Fehlerträchtigkeit der Portierung, da sich die erforderlichen Änderungen auf einen relativ kleinen Teil des Quellcodes beschränken. Ein späterer Ersatz der mbufs durch geeignetere Lösungen ist aber auf jeden Fall denkbar.

3.1.3 Unterbrechungsbehandlung

In der BSD–Implementierung löst der Netzwerkadapter bei jedem Eintreffen eines Pakets eine Unterbrechung aus. Die zugehörige Unterbrechungsbehandlungsroutine unterbricht dann die Abarbeitung des aktiven Prozesses und verarbeitet dieses Paket (siehe Abschnitt 2.4.1). Neben diesen vom Netzwerkadapter veranlaßten Unterbrechungen finden im Netzwerkcode auch noch vom Zeitgeber ausgelöste Interrupts Verwendung: Jede Protokollschicht kann zwei Funktionen bestimmen, die in unterschiedlichen Intervallen aufgerufen werden. TCP benutzt diesen Mechanismus, um nicht bestätigte Pakete erneut zu übertragen, während IP auf diese Weise Paketfragmente verwirft, die zu lange nicht reassembliert werden konnten (siehe Abschnitte 2.1.1 und 2.1.2). Bei der Ausführung der jeweiligen Unterbrechungsbehandlungsroutine wird auf einen Mechanismus zurückgegriffen, der in ähnlicher Form bereits bei der im Abschnitt 3.1.1 beschriebenen Kontextumschaltung verwendet wird: Der Kontext des aktiven Prozesses wird zunächst auf dessen Stack gerettet. Danach wird sowohl der Programm- als auch der Stackzeiger dieses Prozesses so verändert, daß er die erforderliche Unterbrechungsbehandlungsroutine abarbeitet. Nach deren Ende wird der zuvor gesicherte Kontext wiederhergestellt, und der Prozeß setzt seine Arbeit an genau der Stelle fort, an der er zuvor unterbrochen wurde.

Es ist natürlich möglich, dieses Vorgehen genau so auf L4 zu übertragen. Dies ist aber mit einigen Problemen verbunden. So veranlaßt die Manipulation vom Programm- bzw. Stackzeiger, daß eine möglicherweise gerade ablaufende IPC–Operation des manipulierten sowie der am IPC beteiligten Threads abgebrochen wird. Es müßten also alle IPC–Operationen in Fehlerbehandlungsmaßnahmen zum Erkennen dieses Abbruchs und Wiederaufsetzen gekapselt werden.[4] Ein verbesserter Ansatz bietet sich, da L4 beim Scheduling sogenannte *harte Prioritäten* verwendet. Einem Thread niedriger Priorität wird keine Rechenzeit zugeteilt, solange ein höher priorisierter Thread rechenbereit ist. Es kann nun jeweils ein spezieller Thread pro Unterbrechungsniveau (siehe Tabelle 2.3 auf Seite 16) benutzt werden, der sämtlichen Unterbrechungsbehandlungsroutinen des entsprechenden Niveaus ausführt. Durch geeignete Wahl der Prioritäten entsprechend des zugehörigen Unterbrechungsniveaus ergibt sich das gewünschte Verhalten, daß Unterbrechungen höherer Priorität innerhalb Unterbrechungen niederer Priorität auftreten können, aber nicht umgekehrt. Beim Eintreffen einer Unterbrechung wird nun der zugehörige Bearbeitungsthread aktiviert, und er versucht, die Behandlungsroutine abzuarbeiten. Dies entspricht

[4] Beim im Abschnitt 3.1.1 beschriebenen Kontextwechsel während der Bearbeitung der Klientenanfragen ist dies nicht erforderlich, da der Umschaltpunkt — im Gegensatz zu dem jederzeit möglichen Eintreffen einer Unterbrechung — genau bekannt ist. IPC–Abbrüche können dort daher ausgeschlossen werden.

dann exakt dem Verhalten der Unterbrechungsbehandlung durch Kontextwechsel. Lediglich der Kontextwechsel wird implizit vom Scheduler vorgenommen, und beide Kontexte laufen in verschiedenen Threads ab.

Um *Wettbewerbsbedingungen* zu vermeiden, unterdrückt der BSD–Kernelcode das Auftreten von Interrupts, wenn es erforderlich ist. Während der Manipulation globaler Daten wie Warteschlangen und IP–Adreßzuordnungen wird beispielsweise beim Eintreffen eines Pakets keine Unterbrechung ausgelöst.[5] Um dieses Verhalten zu erreichen, hatte ich in einem ersten Ansatz einen zusätzlichen, diese Unterbrechungen koordinierenden, Thread vorgesehen. Um Unterbrechungsniveaus zu sperren bzw. wieder zu erlauben, mußte der entsprechende Thread dies dem Koordinator durch einen IPC mitteilen. Bevor einer der Unterbrechungsbehandlungsthreads mit der Bearbeitung einer Unterbrechungsbehandlungsroutine beginnen konnte, hatte er den Koordinator, wiederum per IPC, darüber zu informieren. Stellte nun dieser Koordinator fest, daß das geforderte Unterbrechungsniveau (siehe Tabelle 2.3 auf Seite 16) bereits gesperrt ist, fügte er den anfragenden Thread in eine Warteschlange ein. Wenn ein Interruptlevel wieder verfügbar wurde, wurden darauf wartende Threads der Reihe nach wieder aktiviert.

Diese Arbeit mit in Warteschlangen verwalteten blockierten Threads ist aber im L4–Kern–Scheduler bereits realisiert. Deshalb bietet es sich natürlich an, diese, ohnehin schon vorhandene, Funktionalität zu nutzen. Um ein Unterbrechungsniveau zu sperren, muß der aktive Thread seine Priorität lediglich über die des Threads setzen, der die zu blockierenden Unterbrechungen bearbeitet. Beim Freigeben erniedrigt er sie dann wieder entsprechend. Ein wesentlicher Vorteil dieser Lösung: Die, vorwiegend beim Sperren und Freigeben der Unterbrechungsniveaus relativ häufig erforderliche, Interaktion mit dem Unterbrechungskoordinator entfällt, ebenso wie sämtlicher Ressourcenverbrauch durch diesen Koordinierungsthread. Stattdessen ist lediglich ein wesentlich weniger aufwendiges Ändern der Threadpriorität durchzuführen.

Zeitgeber–Unterbrechungen

Im BSD–Code werden derzeit vier Zeitgeber benutzt, nach deren Ablauf jeweils eine Unterbrechung ausgelöst wird. In den Unterbrechungsbehandlungsroutinen werden beispielsweise Paketfragmente, die über einen gewissen Zeitraum nicht zusammengefügt werden konnten, verworfen oder versandte TCP–Pakete, deren Empfang noch nicht bestätigt wurde, erneut übertragen.

Die Implementierung dieser Zeitgeber kann durch einen L4–Thread realisiert werden, der eine gewisse Zeit auf ein niemals eintretendes Ereignis, beispielsweise eine Empfangsoperation von einem nichtexistenten Thread, wartet. Alle vier Intervalle könnten von einem einzigen Thread ermittelt werden. Dies bringt dann aber eine etwas erhöhte Komplexität in Bezug auf die Synchronisation

[5] Dies führt dann allerdings dazu, daß dieses Paket nicht bearbeitet wird, falls das Auftreten der Unterbrechung bis zum Empfang des nächsten Pakets nicht wieder erlaubt wird und der interne Puffer des Netzwerkadapters überläuft.

der einzelnen Intervalle mit sich. Deshalb habe ich mich vereinfachend dafür entschieden, einen Thread pro Zeitgeber zu verwenden. Desweiteren existiert ein als *softclockisr–Thread* bezeichneter Thread entsprechender Priorität, der die Unterbrechungsbehandlungsroutinen dieses Niveaus bearbeitet.

Beim Ablauf eines Zeitgeber–Intervalls wird der softclockisr–Thread vom betroffenen Zeitgeber-Thread darüber informiert und beginnt mit der Abarbeitung der zugehörigen Unterbrechungsbehandlungsroutine. Abhängig von seiner Priorität kann sich diese Abarbeitung verzögern, bis das Unterbrechungsniveau freigegeben ist.

3.1.4 Netzwerkschnittstelle

Während seiner Tätigkeit an der TU Dresden hat Michael Hohmuth den Ethernet–Treiber des an der University of Utah im Rahmen des Flux–Projektes entstandenden OSKits [FBB+99] aus diesem Paket herausgelöst und zu einem eigenständigen L4–Server entwickelt. Da dies der momentan einzige verfügbare Ethernet–Treiber für DROPS ist, soll er als Schnittstelle des IP–Servers zum Netzwerk eingesetzt werden.

Die Kommunikation mit dem Ethernet–Server kann derzeit auf zwei Arten geschehen: Zum einen gibt es eine innerhalb des OSKits standardmäßig verwendete, auf den Spezifikationen der Firma Microsoft basierende, COM–Schnittstelle [Tha99], und zum anderen können die üblichen L4–IPC–Mechanismen zum Austausch von *indirect strings* genutzt werden. Die zukunftssichere, weil portablere Lösung ist, die COM–Schnittstelle zu verwenden. Da sich mir deren Komplexität aber noch nicht vollständig erschlossen hat, verwende ich zunächst die IPC–Schnittstelle. Zu einem späteren Zeitpunkt ist eine Umstellung auf COM–Objekte durchaus vorstellbar.

Um ein Paket zu versenden, muß dieses lediglich in einem indirect string mittels IPC an den Ethernet–Server geschickt werden. Zum Datenempfang hat der empfangswillige Thread dem Ether–Server den Identifikator desjenigen Threads mitzuteilen, welcher auf eintreffende, empfangene Datagramme beinhaltende, indirect strings wartet. Während der Initialisierungsphase werden zwischen dem Ethernet–Server und dem ihn nutzenden Thread die jeweiligen Thread–IDs ausgetauscht.

In der BSD–Implementierung ist der Versand von Paketen folgendermaßen realisiert: Zu jedem im System befindlichen Netzwerkadapter existiert eine Datenstruktur, in der eine Warteschlange für zu sendende Datagramme verwaltet wird. Der die Anfrage des Klienten bearbeitende Thread kettet nun ein abzusendendes Paket in diese Warteschlange ein und veranlaßt durch Setzen der entsprechenden Hardwareregister, daß der Netzwerkadapter mit dem Abschicken dieses Pakets beginnt. Die Anpassung an den IP–Server kann nun analog erfolgen. Statt aber durch das Setzen der Hardwareregister das Senden zu veranlassen, wird das Datagramm mit einem IPC an den Netzwerkadapter geschickt, der es daraufhin in das Netzwerk gibt. Um den die Klientenanfragen abarbeitenden Thread während dieser Sendephase nicht zu blockieren, wird pro

Netzwerkadapter ein spezieller Sendethread erzeugt, welcher nach dem Einketten von Daten in die zugehörige Warteschlange sämtliche in der Warteschlange befindlichen Pakete der Reihe nach an den Ethernet–Server schickt.

Wie zuvor bereits erwähnt, sollte zum Empfang von Paketen ein spezieller Empfangsthread angelegt werden, um die Gefahr des Paketverlusts durch Pufferüberlauf möglichst gering zu halten. Denkbar ist, daß ein einziger Thread auf Pakete von allen im System befindlichen Netzwerkadaptern wartet. Dies erschwert aber die Zuordnung von Paketen zu den Netzwerkadaptern, von denen sie empfangen wurden, da der Ether–Server diese Information nicht bereitstellt. Deshalb soll für jeden Netzwerkadapter neben dem Sendethread auch ein spezieller Empfangsthread angelegt werden.

Sobald der Empfangsthread ein Paket erhalten hat, informiert er den als *impisr–Thread*[6] bezeichneten Thread entsprechender Priorität, der die Unterbrechungsbehandlungsroutinen dieses Niveaus bearbeitet. Der impisr–Thread führt einige Plausibilitätsprüfungen durch und kettet das Paket in eine globale Warteschlange aller empfangenen, aber noch nicht verarbeiteten Pakete ein. Anschließend löst er eine weitere Unterbrechung auf dem Niveau *splnet* (siehe Tabelle 2.3 auf Seite 16) aus und übergibt damit die Kontrolle an den dafür zuständigen niedriger priorisierten *netisr–Thread*. Dieser verarbeitet nun der Reihe nach alle in dieser Eingabewarteschlange wartenden Pakete.

Loopback–Gerät

An das Loopback–Gerät übergebene Pakete werden — im Gegensatz zu den von Netzwerkgeräten verarbeiteten Paketen — nicht über ein angeschlossenes Netzwerk versandt. Stattdessen kettet sie die Ausgaberoutine des Loopback–Geräts sofort in die globale Empfangswarteschlange des Systems ein und löst eine *net*–Unterbrechung aus, d.h. sie aktiviert den netisr–Thread. Anschließend werden die Pakete analog denjenigen, die über Netzwerkadapter empfangen wurden, verarbeitet.

3.1.5 Nutzerschnittstelle

Die Schnittstelle des L4–IP–Stacks nutzt im Gegensatz zur BSD–Implementierung keine Systemrufe, da es sich beim L4–IP–Stack um einen im Benutzermodus laufenden Thread handelt. Es muß also ein anderes Konzept gefunden werden, mit welchem sich die Schnittstelle zwischen IP–Server und Klienten realisieren läßt.

Für den Austausch von Daten zwischen Klienten und dem IP–Server sollen die IPC–Mechanismen des L4–Systems Verwendung finden. Eine Möglichkeit wäre, wie bereits bei der L4ATM–Implementierung (Abschnitt 2.3) einen gemeinsamen Speicherbereich zwischen Klient und IP–Server zu etablieren, über den der Datenaustausch in beide Richtungen erfolgen kann. Dieses Verfahren

[6] Die Bezeichnung *imp* rührt von der Bezeichnung ***I**nterface **M**essage **P**rocessor* her. Dies war der Name des ursprünglich im ARPANET verwendeten Routertyps.

bringt aber eine gewisse Komplexität mit sich. So müssen beim Einrichten dieses gemeinsamen Speicherbereichs beide Seiten wissen, in welchem Bereich ihres Adreßraums dafür noch Platz vorhanden ist.

Ich nutze zum Datenaustausch einen IPC, der indirect strings transportiert. Auf den ersten Blick ergibt sich dadurch ein großer Nachteil, da während des eigentlichen Kommunikationsvorgangs Daten kopiert werden müssen. Im Interesse einer einfachen und schnell realisierbaren Lösung habe ich mich aber trotzdem zu diesem Vorgehen entschlossen. Bei der Verwendung von gemeinsamem Speicher müßten die auszutauschenden Daten natürlich auch erst in diesen Speicherbereich kopiert werden, während die indirect strings direkt mit den übergebenen Speicherblöcken operieren können. Dies wiegt einen Teil des Geschwindigkeitsverlustes wieder auf.[7]

Um den IP–Server im System lokalisieren zu können, wird der L4–Nameserver *names* [Dan98] verwendet. Beim Start registriert sich der IP–Server beim Namensdienst, so daß sein Thread–Identifikator von Klienten einfach ermittelt werden kann. Eine Anfrage des Klienten beim IP–Server gestaltet sich demnach folgendermaßen:

1. Wenn der Klient den ersten Zugriff auf den IP–Server ausführen will, bemerkt er, daß ihm dessen Thread–ID, an welche die Anfrage gesendet werden muß, nicht bekannt ist. Er versucht deshalb, diese vom Namensdienst in Erfahrung zu bringen. Gelingt dies, merkt sich der Klient diese Thread–ID, um bei späteren Anfragen auf diese Kommunikation mit dem Namensdienst verzichten zu können.

2. Er baut den IPC–Nachrichtenpuffer unter Verwendung der zu übertragenden Daten und Empfangspuffer zusammen und sendet diesen an den IP–Server.

3. Der Klient wartet, bis ihm der IP–Server mögliche Ergebnisse per IPC liefert. Eventuell werden noch Ergebnisse in die vom Klienten bereitgestellten Empfangspuffer kopiert.[8]

3.1.6 Threadstruktur

In der Abbildung 3.2 ist noch einmal anschaulich dargelegt, welche Beziehungen zwischen den einzelnen Threads des IP–Servers bestehen und welche Kommunikationswege existieren. Abbildung 3.3 zeigt die Prioritäten, mit der die Threads im System operieren.

[7] Das Kopieren der Daten erfolgt bei diesem Verfahren aber natürlich *während* des vollständigen Sende- bzw. Empfangsvorgangs. Bei der Verwendung von gemeinsamem Speicher können die Daten bereits *zuvor* kopiert werden.

[8] Aufwendige Kopieroperationen sind hier nicht erforderlich. Größere Datenmengen, die in indirect strings übertragen wurden, befinden sich bereits im Empfangspuffer des Klienten. Lediglich einige *dwords* erfordern ein Umkopieren.

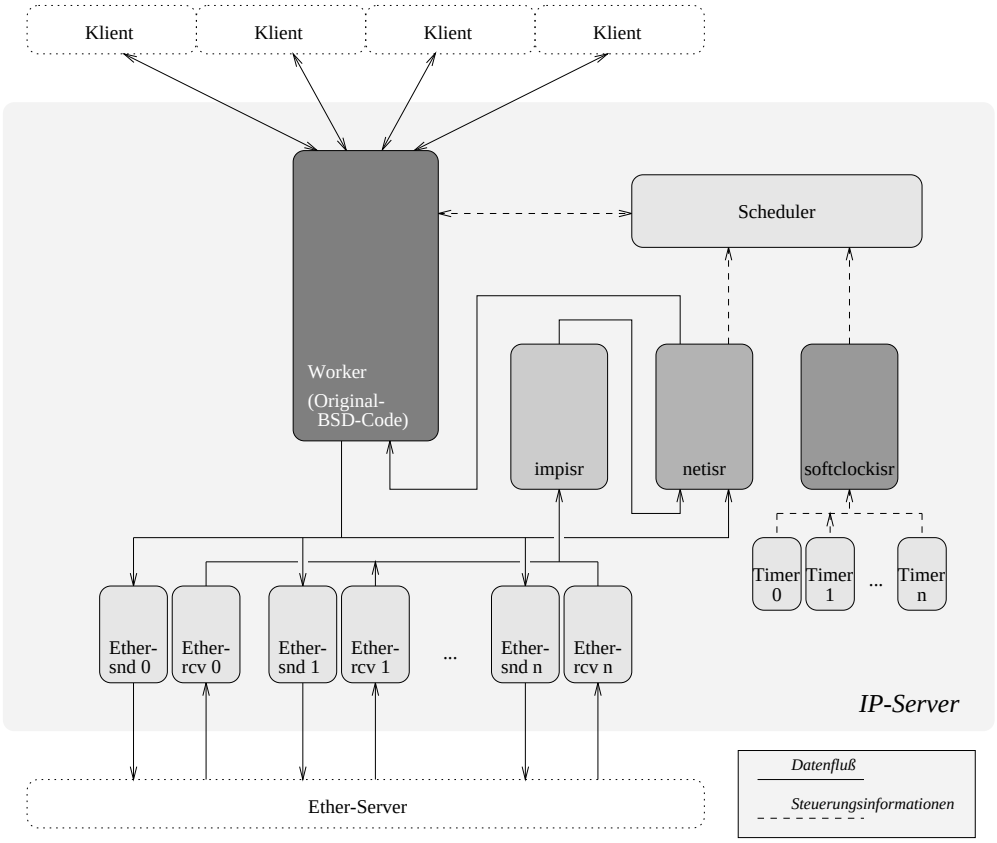


Abbildung 3.2: Struktur der Threads im IP–Server
Der *Worker* führt den ursprünglichen BSD–Code aus. Die zusätzlichen Threads dienen der Anpassung an das L4–System: In *impisr*, *netisr* und *softclockisr* laufen die Unterbrechungsbehandlungsroutinen der jeweiligen Unterbrechungsniveaus (siehe Abschnitt 2.4.1 und Tabelle 2.3).
Die Helligkeit der Threads korrespondiert mit deren Priorität im System: Ein hellerer Ton symbolisiert höhere Priorität gegenüber dunkleren Tönen.

3.2 Ressourcenreservierung

Die im IP–Stack implementierten Internet–Protokolle IP, TCP und UDP ermöglichen lediglich einen zuverlässigen Datentransport über das zugrundeliegende Übertragungsmedium. Zusagen hinsichtlich der nutzbaren Übertragungsgüte sind aber nicht machbar. Um den L4–IP–Server auf die Bedürfnisse der zugrundeliegenden Einsatzumgebung abzustimmen, ist eine solche Zusagefähigkeit allerdings erforderlich. Deshalb muß der IP–Stack um diese Eigenschaft erweitert werden. Eine strenge Echtzeitfähigkeit läßt sich dabei allerdings grundsätzlich nicht erreichen, da das verwendete Übertragungsmedium Ethernet nach dem Zugriffsverfahren *CSMA/CD* arbeitet. CSMA/CD *(Carrier Sense Multiple Access with Collision Detection)* beruht auf folgendem Prinzip: Wenn eine Station senden möchte, hört sie das Netzwerk ab und sendet ihre

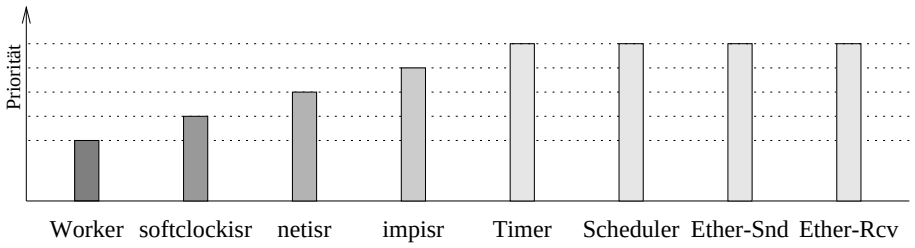


Abbildung 3.3: Prioritäten der vom IP–Server genutzten Threads

Daten, sofern momentan keine andere Station Daten ins Netzwerk gibt. Wenn sich die eigene mit Sendungen anderer Stationen überlagert, es also zu *Kollisionen* kommt, wird der Sendevorgang abgebrochen und nach einer willkürlich gewählten Zeitspanne erneut versucht. Dies hat allerdings zur Folge, daß, besonders unter hoher Last, relativ häufig Kollisionen auftreten und deshalb der Zeitpunkt des Netzzugriffs stark vom Zufall abhängt. Unter diesen Umständen kann zwar der sendende Host dafür sorgen, daß er die lokale Verarbeitung der Datenpakete gemäß einer existierenden Reservierung vornimmt. Das Verhalten der Datagramme im Netzwerk ist aber vom Sendeverhalten aller am Netz angeschlossenen Hosts abhängig, welches ohne extrem aufwendige (und damit praktisch unbrauchbare) Synchronisationsprotokolle nicht kontrollierbar ist.

In der Praxis haben einige Verfahren gewisse Verbreitung erfahren, die, teilweise basierend auf den Internet–Protokollen, eine Reservierung benötigter Ressourcen realisieren. Dies sind vorwiegend das im Abschnitt 2.2.1 beschriebene RSVP und ST–II (Abschnitt 2.2.2). ST–II nutzt allerdings ein proprietäres Übertragungsprotokoll, so daß ein zusätzlicher IP–Stack die ebenfalls gewünschten nicht–zusagefähigen IP–Dienste anbieten muß. Eine ideale Lösung wäre deshalb die Implementation des RSVP–Protokolls auf Basis des IP–Servers. Dadurch könnte neben den nicht–zusagefähigen IP–Diensten ein in der Praxis etabliertes Reservierungsverfahren genutzt werden können.

Leider ist die Implementierung des RSVP–Protokolls bzw. die Portierung einer vorhandenen Lösung, beispielsweise die des Information Sciences Institute (ISI) der University of Southern California, Los Angeles [ISI], im Rahmen dieser Arbeit nicht mehr möglich. Um die Funktionsfähigkeit meines IP–Server–Konzepts demonstrieren zu können, soll deshalb zunächst lediglich ein einfaches Signalisierungs– bzw. Reservierungsprotokoll realisiert werden, das es Systemen ermöglicht, Reservierungsnachrichten untereinander austauschen zu können. Ein in den IP–Server zu integrierender Paket–Scheduler entscheidet dann aufgrund dieser Reservierungsinformationen, in welcher Reihenfolge zu sendende bzw. empfangene Pakete vom IP–Stack zu bearbeiten sind und ob weitere Reservierungen vorgenommen werden können. Durch eine flexible Schnittstelle zwischen IP–Server, Paket–Scheduler und der Reservierungskomponente ist die Möglichkeit eines späteren Austauschs beispielsweise gegen einen vollständigen RSVP–Server gegeben.

3.2.1 Signalisierungsprotokoll

Beim Entwurf des Signalisierungsprotokolls habe ich mich an der Arbeitsweise von RSVP (siehe Abschnitt 2.2.1) orientiert. Die Signalisierungsinformationen sollen mit dem im Abschnitt 2.1.3 beschriebenen Transportprotokoll UDP übermittelt werden. Zur Übertragung der Nutzdaten wird ebenfalls der normale IP–Protokollstack genutzt. Durch Nutzung dieser bewährten Protokollfamilie kann die fehlerträchtige Entwicklung eines proprietären, und damit nicht universell einsetzbaren, Transportprotokolls vermieden werden. Im Interesse einer einfachen Realisierbarkeit habe ich jedoch auf die Verwendung von Soft–States sowie unterschiedlichen PATH– und RESV–Nachrichten verzichtet. Weiterhin gehe ich davon aus, daß auf dem Übertragungsmedium keine Paketverluste vorkommen. Dies hat die Konsequenz, daß das Protokoll keinerlei komplizierte Fehlerbehandlungen durchführen muß.

Reservierungen in Sende- und Empfangsrichtung sind zu unterscheiden. Daraus folgt, daß entweder nur der Sender oder der Empfänger eine Reservierung veranlassen kann. Andernfalls müßte das Protokoll eine Unterscheidung zwischen sender- und empfängerinitiierten Reservierungsnachrichten vornehmen. Um wieder dem Beispiel RSVP zu folgen, habe ich mich dafür entschieden, nur vom Empfänger ausgehende Reservierungen zu ermöglichen.[9]

Um eine Reservierung zu veranlassen, muß der spätere Empfänger des reservierungsbehafteten Datenstroms dem potentiellen Sender eine Beschreibung dieses Datenstroms sowie die zu garantierende Übertragungsgüte mitteilen. Zuvor prüft er aber, ob die erforderlichen Ressourcen auf seinem System überhaupt verfügbar sind. Wenn dem so ist, markiert er diese Ressourcen als belegt und schickt die Reservierungsnachricht an den Sender. Die Reservierung für den Datenstrom wird allerdings noch nicht wirksam. Der Sender versucht nun, diese Ressourcen auf seinem System zu reservieren. Im Erfolgsfall schickt er eine Bestätigung an den Empfänger, woraufhin dieser die zuvor noch nicht aktivierte Reservierung auf seinem System aktiviert. Auf der Seite des Senders wird die Reservierung unmittelbar nach dem Bestätigen wirksam. Falls der Sender die erforderlichen Ressourcen nicht allozieren kann oder bereits eine Reservierung für einen entsprechenden Datenstrom existiert, so informiert er den Empfänger darüber, woraufhin dieser die bereits belegten Ressourcen wieder freigibt.[10]

Um eine bestehende Reservierung zu beenden, gibt der Empfänger des Datenstroms die zugehörigen lokal allozierten Ressourcen wieder frei und schickt eine entsprechende Botschaft an den Sender. Dieser braucht dann nur noch seine zugehörigen belegten Ressourcen wieder freizugeben.

In der Abbildung 3.4 sind die Schritte schematisch dargestellt, die die jeweiligen Signalisierungsinformationen nach sich ziehen. Es ist ersichtlich, daß

[9] Eine Beschränkung auf nur vom Sender ausgehende Reservierungen wäre aber auch denkbar und würde das Protokoll nur unwesentlich ändern. Es wäre lediglich die Wirkungsrichtung der etablierten Reservierung umzukehren.

[10] Aufgrund der Annahme, daß keine Paketverluste auftreten, können Bestätigungspakete für Reservierungsnachrichten, die möglicherweise wiederum Bestätigungspakete nach sich ziehen, und das Hantieren mit Timeouts für die Paketlaufzeit vermieden werden.

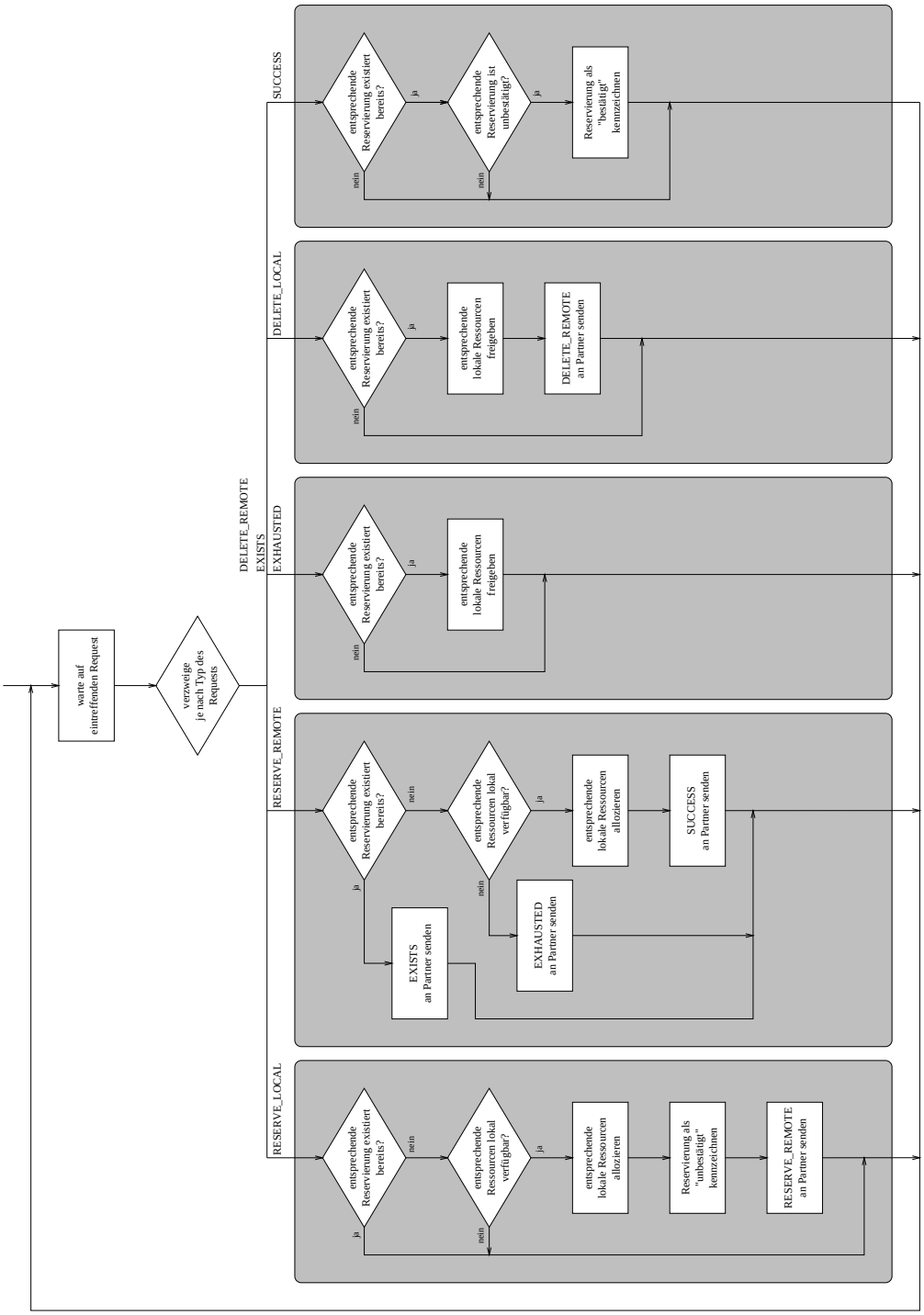


Abbildung 3.4: Reservierungsprotokoll

dieses Signalisierungsprotokoll nur die jeweiligen Endsysteme betrifft. Entlang des Kommunikationswegs befindliche Router sehen diese Signalisierungsdaten lediglich als reine UDP–Pakete. Zum einen ist diese Annahme getroffen worden, um die Komplexität des Protokolls zu reduzieren. Andererseits ist es in meinen

Augen eher fragwürdig, Reservierungen in Weitverkehrsnetzen aufrechterhalten zu wollen. Jeder betroffene Router müßte in der Lage sein, eine Vielzahl von Reservierungsinformationen zu verwalten. Heute gebräuchliche Router sind mit dieser Aufgabe sicher weitgehend überfordert.

Paketklassifizierung

Das Signalisierungsprotokoll ist lediglich für den Austausch von Signalisierungsinformationen zwischen den Endsystemen vorgesehen. Um eine Zuordnung von Paketen zu einem einer Reservierung unterliegenden Datenstrom zu ermöglichen, muß eine Beschreibungsmöglichkeit für einen solchen Datenstrom existieren. Theoretisch wäre jede Möglichkeit denkbar, die auf der Grundlage des Inhalts eines IP–Pakets eine Entscheidung fällt. Da der Paket–Scheduler (siehe Abschnitt 3.2.2) diese Entscheidung aber für jedes einzelne Paket treffen muß, verbieten sich relativ komplexe Vorgehensweisen. RSVP klassifiziert einen Datenstrom lediglich durch das Tupel $D = \{Senderadresse, Empfangsport\}$. Dabei können einzelne Elemente des Tupels durch Wildcards ersetzt werden oder einen bestimmten Wertebereich abdecken [Wro97]. So ist es beispielsweise möglich, einen Datenstrom zu definieren, dem Pakete, die von beliebigen Sendern stammen und an die Ports 2000 und 4000 gesendet werden, zuzuordnen sind.

Ich greife dieses Verfahren auf und verzichte lediglich auf die Unterstützung von Wertebereichen und Wildcards. Dadurch ist eine Übertragung dieser Klassifizierungsdaten in Puffern fester Größe möglich. Eine Spezifizierung des Transportprotokolls ist nicht vorgesehen. Da beispielsweise ein TCP–Paket Timeouts, Bestätigungsnachrichten und eventuelle erneute Übertragungen nach sich zieht, ist eine garantierte Übertragungsgüte schwer durchzusetzen. Deshalb wird als Transportprotokoll lediglich UDP unterstützt. Ein Datenstrom ist also durch das Tupel

$$D = \{Senderadresse, Empfangsport\}$$

eindeutig bestimmt.[11] Da diese Angaben im IP– bzw. UDP–Header jedes Pakets an genau definierten Stellen enthalten sind, ist eine Auswertung mit relativ geringem Aufwand möglich. Zu beachten ist lediglich, daß die UDP–Pakete nicht fragmentiert sein dürfen. Andernfalls wäre der UDP–Header nicht in jedem Datagramm vollständig enthalten, was die Ermittlung des in diesem Header enthaltenen Empfangsports unmöglich machen würde.

3.2.2 Paket–Scheduler

Der Paket–Scheduler hat die Aufgabe, basierend auf den vorliegenden Reservierungsinformationen zu entscheiden, ob bzw. wann ein Paket bearbeitet werden

[11] Ein BSD–Socket (siehe Abschnitt 2.5) wird durch ein ähnliches Tupel definiert. Hier fehlt lediglich die Angabe der Adreßfamilie, da der IP–Server nur die Internetprotokolle (siehe Abschnitt 2.1) unterstützt.

muß. Dazu hat er zunächst zu untersuchen, ob das zu klassifizierende Paket einer Reservierung unterliegt. Zu diesem Zweck wertet er die im IP– bzw. UDP–Header des Pakets befindlichen Angaben zur Senderadresse und des Empfangsports sowie des verwendeten Transportprotokolls aus. Stellt er fest, daß das vorliegende Paket nicht fragmentiert ist und eine Reservierung für diesen Datenstrom existiert, so ist zu prüfen, ob durch dieses Paket die Ausnutzung der reservierten Ressourcen nicht überschritten wird. Wenn das Paket reservierungskonform ist, wird es zur Bearbeitung freigegeben. Andernfalls kann es entweder verworfen oder, je nach Auslastung des Systems, sofort oder zu einem späteren Zeitpunkt bearbeitet werden. Ein Verwerfen nichtkonformer Pakete in einem nichtausgelasteten System halte ich nicht für sinnvoll. Sofern im System genügend freie Verarbeitungskapazität verfügbar ist, soll es einer Anwendung möglich sein, mehr Pakete zu senden bzw. empfangen als die von ihr reservierten Ressourcen erlauben. Die Reservierung ist lediglich eine Garantie dafür, daß die entsprechenden Ressourcen *in jedem Fall* dieser Applikation bzw. dem betroffenen Datenstrom zur Verfügung stehen und die gewünschte Übertragungsrate *mindestens* erreicht werden kann. Eine Überbuchung benötigter Ressourcen ist also für Applikationen nicht möglich. Stattdessen kann beim Einrichten einer Reservierung die Priorität angegeben werden, mit der nichtkonforme Pakete weitergereicht werden sollen.

Ressourcenverfügbarkeit

Darüberhinaus muß entschieden werden, ob neu angeforderte Ressourcen reserviert werden können oder ob der bisherige Auslastungsgrad dies verbietet. Mein IP–Stack soll es Applikationen ermöglichen, bestimmte Anteile der verfügbaren Übertragungsbandbreite für einen periodisch auftretenden Datenstrom zu reservieren.

Claude–Joachim Hamann hat in [Ham] exemplarisch dargelegt, welche Abhängigkeit der Prozessorauslastung von der Anzahl der abzuarbeitenden Tasks besteht, wenn an diese Tasks periodisch Anforderungen gestellt werden. Er zeigt, daß es für zwei Tasks einen Ablaufplan gibt, wenn die durch beide Tasks verursachte Prozessorauslastung der Gleichung 3.1 genügt. Die Verallgemeinerung für eine beliebige Anzahl von Tasks ergibt sich entsprechend der Gleichung 3.2. U_g entspricht dabei der maximal zulässigen Prozessorauslastung, während n die Anzahl der Tasks angibt.

$$U_g \leq 2(\sqrt{2} - 1) \tag{3.1}$$

$$U_g \leq n(\sqrt[n]{2} - 1) \tag{3.2}$$

Diese Erkenntnisse lassen sich auch auf die Verfügbarkeit von Ressourcen innerhalb meines IP–Servers übertragen. Unter der Annahme, daß im IP–Server die verfügbare Netzwerk–Bandbreite den Engpaß darstellt, genügt es, lediglich U_g aus Gleichung 3.2 durch die verfügbare Netzwerk–Bandbreite bzw. deren Auslastung U_B zu ersetzen (Gleichung 3.3). Die Anzahl der einer Reservierung unterliegenden Datenströme entspricht n.

$$U_B \leq n(\sqrt[n]{2} - 1) \tag{3.3}$$

Um entscheiden zu können, ob neu angeforderte Ressourcen verfügbar sind und deshalb reserviert werden können, ist demnach lediglich zu prüfen, ob die mit dieser zusätzlichen Reservierung erreichte Bandbreitenauslastung der Gleichung 3.3 genügt. Die Bandbreitenauslastung läßt sich mit der Gleichung 3.4 ermitteln. B_r entspricht dabei der Menge der bisher reservierten Bandbreite, während B_g deren insgesamt verfügbare Menge angibt. b bestimmt die neu zu allozierenden Ressourcen.

$$U_B = \frac{B_r + b}{B_g} \tag{3.4}$$

In Gleichung 3.3 eingesetzt und beachtet, daß die Anzahl der Datenströme durch den neu zu berücksichtigenden um eins zu erhöhen ist, ergibt sich Gleichung 3.5. Diese muß erfüllt sein, damit die neu angeforderten Ressourcen reserviert werden können.

$$\frac{B_r + b}{B_g} \leq (n+1)(\sqrt[n+1]{2} - 1) \tag{3.5}$$

Kapitel 4

Implementierung

4.1 mbufs und Cluster

In der aktuellen Version des IP–Servers werden die im Abschnitt 2.4.2 beschriebenen mbufs zur Speicherverwaltung verwendet. An den mbufs selbst wurden dabei keinerlei Änderungen vorgenommen. Lediglich die Verwaltung der Referenzzähler der Cluster mußte modifiziert werden.

Der BSD–Kern verwaltet alle Cluster in einem einzigen zusammenhängenden Speicherblock. Da die Cluster im Speicher entsprechend ihrer Größe ausgerichtet sind, ließ sich anhand einer beliebigen Adresse innerhalb des jeweiligen Clusters und der Anfangsadresse des die Cluster beinhaltenden Speicherblocks leicht ein Index des Clusters in diesem Speicherblock bestimmen. In einer zusätzlichen Tabelle wurde mit Hilfe dieses Indexes der zum Cluster gehörende Referenzzähler ermittelt. Das Verfahren hat allerdings den Nachteil, daß Kopieroperationen erforderlich sind, wenn neue Cluster alloziert werden sollen und der dafür vorgesehene Speicherblock nicht vergrößert werden kann.

Die nicht genutzten Cluster verwaltet der BSD–Kern in einer linearen einfachverketteten Liste. Da der gesamte Platz im Cluster für Daten genutzt wird, ist kein zusätzlicher Speicherplatz für den zur Verkettung erforderlichen Zeiger vorhanden. Stattdessen werden einige Bytes am Beginn des dann nicht verwendeten Cluster–Datenbereichs dafür genutzt.

Um auf diese Kopieroperationen verzichten zu können, alloziere ich einen Speicherblock pro Cluster, die wie in der ursprünglichen Implementierung in einer Liste gehalten werden. Dadurch ist es aber nicht mehr möglich, aus Clusteradressen einen Index zu ermitteln, ohne zusätzliche Tabellen zu verwenden. Deshalb sind in jedem Cluster zwei zusätzliche Elemente vorhanden, die einerseits als Referenzzähler und andererseits zur Verkettung in der Liste der ungenutzten Cluster dienen. Es findet allerdings nur jeweils einer dieser Werte Verwendung. Entweder wird der Cluster in der Liste gehalten, dann braucht kein Referenzzähler geführt zu werden. Oder der Cluster ist in Gebrauch, so daß der Verkettungszeiger nicht benötigt wird. Die einen Cluster beschreibende Datenstruktur könnte also folgenden Aufbau haben:

```
struct mcluster {
  union {
    char             mcl_buf[MCLBYTES]; /* Nutzdaten */
    struct mcluster* mcl_next;  /* Verkettungszeiger */
  } m;
  int mclrefcnt;    /* Referenzzähler */
};
```

Listing 4.1: Mögliche Cluster–Datenstruktur

Da im Programmcode aber die entsprechenden Variablennamen ohne die Indirektion über die *union* verwendet werden, habe ich mich entschieden, den geringen zusätzlichen Speicherverbrauch in Kauf zu nehmen und auf die Verwendung der *union* zu verzichten. Dadurch reduziert sich der Änderungsaufwand am Programmcode. In meiner Implementierung ist ein Cluster deshalb folgendermaßen definiert:

```
struct mcluster {
  char             mcl_buf[MCLBYTES];  /* Nutzdaten */
  struct mcluster* mcl_next;    /* Verkettungszeiger */
  int              mclrefcnt;       /* Referenzzähler */
};
```

Listing 4.2: Verwendete Cluster–Datenstruktur

Zu beachten ist, daß sich der Nutzdatenbereich `mcl_buf` unmittelbar am Beginn des Clusters befinden muß, da auf die Daten direkt über den Cluster–Verweis im mbuf zugegriffen wird. Das Element `mcl_buf` selbst wird im Programmtext nirgends über diesen Namen verwendet.

4.1.1 Hauptspeicherverwaltung

Der IP–Server benutzt zur Laufzeit die in der Standard–C–Bibliothek definierten Funktionen `malloc()` und `free()`, um dynamisch Hauptspeicherblöcke anzufordern bzw. wieder freizugeben. Unter DROPS/L4 wird diese Funktionalität über die im OSKit [FBB+99] enthaltene Standard–C–Bibliothek realisiert, welche im Rahmen anderer Arbeiten an DROPS angepaßt wurde. Da die mbuf–Verwaltung bzw. eine Vielzahl der Zugriffe auf mbufs im BSD–Kernel darauf beruht, daß mbufs im Speicher entsprechend ihrer Größe ausgerichtet sind, finden die vom OSKit zusätzlich angebotenen, nichtstandardisierten Funktionen `smemalign()` bzw. `sfree()` ebenfalls Verwendung, welche es erlauben, entsprechend ausgerichteten Speicher anzufordern. Um die Speicherfragmentierung zu vermindern, die entsteht, da `malloc()` zu Verwaltungszwecken einige zusätzliche Bytes alloziert, ist beim Freigeben des Speicherblocks zusätzlich dessen Größe anzugeben, so daß `smemalign()` auf die zusätzliche Allozierung dieser Verwaltungsdaten verzichten kann.

Von den beiden im OSKit enthaltenen, unterschiedlich umfangreichen bzw. vollständigen, C–Bibliotheken wird die einfachere verwendet, da diese beim Serverstart wesentlich weniger Initialisierungsaufwand erfordert und

während des Betriebes weniger Ressourcen verbraucht. Der gebotene Funktionsumfang deckt dabei die Erfordernisse ab.

4.2 Prozeßkonzept

Der innerhalb des Systemkerns ablaufende BSD–IP–Stack basiert auf dem Prozeßkonzept. Der Kern verwaltet für jeden Prozeß eine ihn beschreibende Datenstruktur, die sämtliche prozeßrelevanten Daten enthält. Unter anderem sind darin auch die Deskriptoren geöffneter Dateien enthalten, zu denen auch Socketdeskriptoren gehören. Da die unter L4 verwendeten, einem Prozeß entsprechenden, Tasks keine solche Datenstruktur kennen, muß ein vergleichbarer Mechanismus innerhalb des IP–Servers nachgebildet werden. In der dafür verwendeten Datenstruktur werden natürlich nur diejenigen Elemente aufgenommen, die zum Betrieb des IP–Servers erforderlich sind:

```
struct proc {
  TAILQ_ENTRY(proc) node;
  l4_taskid_t       p_pid;
  l4_threadid_t     t_sig;
  struct filedesc*  p_fd;
  struct pstats*    p_stats;
  struct myucred*   p_ucred;
  int               p_flag;
  char              p_stat;
  sockcontext_t     selectcontext;
  /* ----- */
  struct filedesc p_fd_data;
  struct pstats {
    struct rusage p_ru;
  } pstats_data;
  struct myucred {
    int cr_uid;
  } myucred_data;
  /* ----- */
};
```

Listing 4.3: Prozeß–Datenstruktur `proc`

Da diese Datenstrukturen im IP–Server in einer Liste gehalten werden, ist ein als `node` bezeichnetes Element enthalten, das der Verkettung der einzelnen Strukturen dient. Zur Listenverwaltung werden die im BSD–Systemheader `<sys/queue.h>` definierten Funktionen und Datentypen genutzt.

Um eine Zuordnung zwischen diesen Datenstrukturen und den Tasks im L4–System zu ermöglichen, wird der Identifikator des zugehörigen L4–Tasks in `p_pid` vermerkt. Daraus ergibt sich, daß für jeden den IP–Server nutzenden Task nur eine `proc`–Struktur angelegt wird. Dies ist auch dann der Fall, wenn mehrere Threads desselben Tasks Anfragen an den IP–Server stellen.

Sobald der IP–Server einen neuen Socket erzeugen soll, prüft er, ob bereits eine `proc`–Struktur für den Task existiert, zu dem der anfragende Thread

gehört. Ist dies nicht der Fall, dann wird eine neue Struktur alloziert. Diese Struktur wird dann so lange vom IP–Server verwaltet, wie Sockets im zugehörigen Task geöffnet sind. Sobald das letzte Socket geschlossen wurde, wird die entsprechende `proc`–Struktur freigegeben. Bei der nächsten Anfrage dieses Tasks muß dann natürlich eine neue Datenstruktur angelegt werden.

In bestimmten Situationen, beispielsweise bei Statusänderungen an nichtblockierenden Sockets, schickt der BSD–Kern ein Signal an den zum Socket gehörenden Prozeß. Unter L4 gibt es kein vergleichbares Signalkonzept. Deshalb ist in der `proc`–Struktur ein Element namens `t_sig` vorgesehen, welches einen vom Besitzer des Sockets[1] anzugebenden Thread–Identifikator aufnimmt. Der dadurch bestimmte Thread muß allerdings auch dem in `p_pid` vermerkten Task angehören. Sobald nun ein solches Signal geschickt werden muß, sendet der IP–Server eine IPC–Nachricht an den in `t_sig` vermerkten Thread. Die Sendeoperation verwendet einen Timeout von 0, um den IP–Server nicht zu blockieren, falls der das Signal empfangende Thread zum Sendezeitpunkt nicht in einer entsprechenden Empfangsfunktion wartet.

Die Informationen über die geöffneten Dateien bzw. Sockets werden im Element `p_fd`, hinter dem sich eine Datenstruktur des Typs `filedesc` (siehe Abschnitt 4.2.1) verbirgt, abgelegt.

Die zusätzlichen Elemente `p_stats`, `p_ucreds`, `p_flags` und `p_stat` dienen lediglich dem Sammeln statistischer Daten sowie Zugriffskontrollmechanismen, die derzeit vom IP–Server nicht benutzt werden. Statt alle Zugriffe auf diese Elemente aus dem Programmtext zu entfernen, habe ich einfach die benötigten Daten in die Struktur aufgenommen. Im Gegensatz zum ursprünglichen BSD–Code, der Speicherplatz für diese zusätzlichen Informationen separat alloziert, wird der erforderliche Speicher innerhalb der `proc`–Struktur bereitgehalten. Dies erspart Fehlerbehandlungsmaßnahmen bei fehlgeschlagenen Speicheranforderungen. Zum Initialisieren einer `proc`–Struktur sind deshalb die folgenden Schritte erforderlich:

```
struct proc* p = (struct proc*)malloc(sizeof(struct proc));

if (p != NULL) {
  memset(p,0,sizeof(struct proc));

  p->p_fd     = &(p->p_fd_data);
  p->p_stats  = &(p->pstats_data);
  p->p_ucred  = &(p->myucred_data);

  p->p_pid = get_taskid(client);
  p->t_sig = L4_NIL_ID;

  init_filedesc(p->p_fd);
}
```

Listing 4.4: Initialisierung einer `proc`–Struktur

[1] Als Besitzer eines Sockets gelten sämtliche zum in `p_pid` vermerkten Task gehörende Threads.

4.2.1 Dateideskriptoren

Die Informationen über die von einem Task geöffneten Dateien werden in der zuvor bereits erwähnten `filedesc`–Struktur abgelegt. Diese hat folgenden Aufbau:

```
struct filedesc {
  struct file*  fd_ofiles[MAXFILES];
  char          fd_ofileflags[MAXFILES];
  int           fd_nfiles;
  sockcontext_t fd_sockcontext[MAXFILES];
};
```

Listing 4.5: `filedesc`–Datenstruktur

Das erste Element, `fd_ofiles`, ist ein Feld von Zeigern auf die eigentlichen Dateiinformationen, während `fd_ofileflags` bestimmte Statusinformationen zu den einzelnen geöffneten Dateien enthält. In `fd_nfiles` wird die Anzahl momentan geöffneter Dateien vermerkt, und `fd_sockcontext` enthält sämtliche Informationen, die erforderlich sind, um die Anfragen an das entsprechende Socket in einem separaten Kontext auszuführen (siehe Abschnitt 4.2.2). In der `file`–Struktur sind folgende Elemente enthalten:

```
struct file {
  short f_flag;
#define DTYPE_VNODE     1
#define DTYPE_SOCKET    2
  short f_type;
  short f_count;
  short f_msgcount;
  struct filedesc* f_fd;
  struct  fileops {
    int (*fo_read)(struct file* fp, struct uio* uio,
                   struct ucred* cred);
    int (*fo_write)(struct file* fp, struct uio* uio,
                    struct ucred* cred);
    int (*fo_ioctl)(struct file* fp, int com,
                    caddr_t data, struct proc* p);
    int (*fo_select)(struct file* fp, int which,
                     struct proc* p);
    int (*fo_close)(struct file* fp, struct proc* p);
  }* f_ops;
  void* f_data; /* socketspezifische Daten */
};
```

Listing 4.6: `file`–Datenstruktur

In `f_type` ist vermerkt, auf welche Art von Daten `f_data` verweist. Da der IP–Server nur Sockets verwaltet, zeigt `f_data` stets auf eine Socket–Struktur, so daß `f_type` den Wert von `DTYPE_SOCKET` enthält. Da es möglich ist, daß eine Datei von mehreren Prozessen geöffnet ist, enthält `f_count` die Anzahl der Tasks, die diese Datei geöffnet haben. Um aufwendigere Suchoperationen zu vermeiden,

ist in `f_fd` ein Verweis auf die zugrundeliegende `filedesc`–Struktur vermerkt. Schließlich ist über die in `f_ops` vermerkten datentypspezifischen Funktionen ein Zugriff auf die entsprechenden Operationen auf Sockets bzw. Dateien möglich.

4.2.2 Threadkontexte

Der BSD–Kern führt jede Systemfunktion im Kernkontext des aufrufendenden Prozesses aus. Jeder dieser Kontexte verfügt über einen eigenen Stack sowie über Programm- und Stackzeiger. Da der IP–Server dieses Konzept in nur einem einzigen Arbeitsthread nachbilden will (siehe Abschnitt 3.1.1), muß er in der Lage sein, verschiedene Kontexte zu verwalten. Da jede Socketanfrage in einem separaten Kontext ablaufen soll, müssen die dazu benötigten Informationen auch für jedes Socket verwaltet werden. Dazu ist in der `file`–Datenstruktur (siehe Abschnitt 4.2.1) eine der folgenden `sockcontext`–Datenstrukturen pro Socket enthalten:

```
struct sockcontext {
  TAILQ_ENTRY(sockcontext) node;
  struct proc*             proc;
  unsigned                 working      : 1;
  unsigned                 waittimedout : 1;
  struct timeval           wakeuptime;
  int*                     stack;
  dword_t                  instrptr;
  dword_t                  stackptr;
  int                      priority;
  void*                    waitchannel;
};
```

Listing 4.7: `sockcontext`–Datenstruktur

Das erste Element `node` dient dem Verketten dieser Datentypen in Listen. Dies wird benötigt, um blockierte sowie wieder arbeitsbereite Kontexte in den entsprechenden Listen des Kontextschedulers (siehe Abschnitt 4.2.3) halten zu können. Um mit geringem Aufwand von einer `sockcontext`–Struktur auf die zugrundeliegende `proc`–Struktur schließen zu können, ist ein entsprechender Verweis im Element `proc` enthalten. In `stack`, `stackptr` und `instrptr` sind die zum Verwalten der Kontexte benötigten Informationen gespeichert. Da zum Sperren von Unterbrechungsniveaus die Threadpriorität geändert wird (siehe Abschnitt 3.1.3), ist ein Element `priority` erforderlich, um diese Prioritäten auch nach Kontextumschaltungen aufrechterhalten zu können.

Da Kontexte ihre Wartezeit an einem bestimmten `waitchannel` begrenzen können (siehe Abschnitt 4.2.3), ist in `wakeuptime` das Ende dieser Wartezeit vermerkt. Wenn die Wartezeit abgelaufen ist, ohne daß der entsprechende Kontext arbeitsbereit geworden ist, wird das Element `waittimedout` gesetzt. Dadurch ist es möglich, Klienten einen korrekten Fehlercode zu übermitteln.

4.2.3 Kontextscheduler

Die in separaten Kontexten ablaufenden Socketoperationen rufen gegebenenfalls Funktionen auf, die im BSD–System den aufrufenden Prozeß blockieren bzw. die Blockierung anderer Prozesse wieder aufheben:

```
int tsleep(void* chan, int pri, char* wmesg, int timo);
void wakeup(void* chan);
```

Listing 4.8: Vom BSD–Code verwendete Synchronisationsroutinen

Die Funktion `tsleep()` blockiert den aufrufenden Kontext, bis er durch ein nachfolgendes `wakeup()` eines anderen Kontextes diesen blockierenden Zustand wieder verläßt oder die im Parameter `timo` angegebene maximale Wartezeit überschritten ist. In `chan` ist dabei ein eindeutiger Wert anzugeben, der als Identifikator dieses Wartezustandes dient. Wird nun `wakeup()` mit demselben Wert für `chan` aufgerufen, verlassen alle mit diesem Wert wartenden Kontexte den blockierenden Zustand.

Sobald ein Kontext `tsleep()` ausführt, werden der Wert von `chan` in das Element `waitchannel` seiner `sockcontext`–Struktur eingetragen, die momentane Registerbelegung auf den Stack dieses Kontextes gerettet, die aktuellen Programm- bzw. Stackzeiger in `instrptr` und `stackptr` sowie die Kontextpriorität in `priority` gespeichert und die `sockcontext`–Struktur über das Element `node` in einer Liste aller blockierten Kontexte gespeichert. Danach werden die `sockcontext`–Strukturen sämtlicher blockierter Kontexte, deren Wartezeit abgelaufen ist, von der Liste der blockierten in die der arbeitsbereiten Kontexte übernommen. Anschließend wird der jetzt zu aktivierende Kontext bestimmt. Sofern die Liste der arbeitsbereiten Kontexte nicht leer ist, wird der erste darin wartende Kontext ausgewählt. Andernfalls muß ein neuer Kontext erzeugt werden, indem ein neuer Stack alloziert und der Programm- sowie Stackzeiger entsprechend initialisiert wird. Der so ermittelte, jetzt zu aktivierende, Kontext wird in den Arbeitsthread gesetzt, indem dessen Stack, Programm- und Stackzeiger geändert werden. Sofern es sich um einen zuvor suspendierten Kontext handelt, stellt er dann seinen Registersatz vom jetzt aktuellen Stack wieder her. Danach fährt er an der Stelle des neu gesetzten Programmzeigers mit der Arbeit fort. Zum Manipulieren der aktuellen Programm- und Stackzeiger wird die L4–Systemfunktion `l4_thread_ex_regs()` verwendet.

Die Funktion `wakeup()` veranlaßt den Kontextscheduler lediglich, alle auf diesen Wert von `chan` wartenden Kontexte von der Liste der blockierten in die der arbeitsbereiten Kontexte zu übernehmen. Hier erfolgt jedoch keine Kontextumschaltung, da, entsprechend dem Arbeitsmodell im BSD–Kernmodus, kein Prozessorentzug stattfindet.

4.3 Socket–Schnittstelle

Die Systemruf–Schnittstelle der ursprünglichen BSD–Implementierung wird, wie im Abschnitt 3.1.5 erläutert, durch eine Kommunikation über IPC–

Mechanismen des L4–Kerns ersetzt. Zur den IP–Server nutzenden Applikation wird eine Bibliothek hinzugelinkt, die die Argumente der einzelnen auf Sockets anwendbaren Funktionen (siehe Abschnitt 2.5) in einen IPC–Nachrichtenpuffer einfügt, diesen an den Workerthread des Servers schickt und anschließend mögliche Ergebnisse in den von der Applikation bereitgestellten Speicherbereichen zurückliefert. Entsprechend der größten zu übermittelnden Datenmenge ergibt sich der folgende IPC–Nachrichtenpuffer:

```
typedef struct {
  l4_fpage_t    fp;
  l4_msgdope_t  size_dope;    /* L4_IPC_DOPE(6,2) */
  l4_msgdope_t  send_dope;
  dword_t       d0,d1;              /* in Registern */
  dword_t       d2,d3,d4,d5;
  l4_strdope_t  str0,str1;
} reqbuf_t;
```

Listing 4.9: IPC–Puffer der Socketschnittstelle

Beim Start des IP–Servers wird ein neuer Thread erzeugt, der auf Anfragen von Klienten wartet und entsprechend der geforderten Funktion weiterarbeitet. Dazu führt er diese Codesequenz aus:

```
void worker_main(void)
{
  reqbuf_t reqbuf;
  init_reqbuf(&reqbuf);

  for(;;) {
    l4_i386_ipc_wait(&client,
                     &reqbuf, &opcode, &d1,
                     L4_IPC_NEVER, &result);

    errcode = (*worker_op[opcode])(client, &reqbuf, &retval);

    l4_i386_ipc_send(client,
                     &reqbuf, errcode, retval,
                     L4_IPC_NEVER, &result);
    reschedule();
  }
}
```

Listing 4.10: Worker–Hauptschleife

Sofern während der Abarbeitung der entsprechenden Socketoperation ein Kontextwechsel durch Aufruf von `tsleep()` erfolgt und kein arbeitsbereiter Kontext vorhanden ist, wird ein neuer Kontext innerhalb dieses Threads erzeugt, der wieder mit der Ausführung von `worker_main()` beginnt (siehe Abschnitt 4.2.3).

Der Aufruf von `reschedule()` am Ende der Schleife bewirkt, daß der Worker–Thread in einem während der Bearbeitung der Socketoperation mögli-

cherweise arbeitsbereit gewordenen Kontext weiterläuft, statt auf neue Anfragen von Klienten zu warten.

4.4 Reservierungsschnittstelle

Um vom IP–Server Reservierungen durchführen zu lassen, kommunizieren die Applikationen ebenfalls unter Nutzung der L4–IPC–Mechanismen mit dem IP–Stack. Die Kommunikation der Reservierungskomponente mit anderen Rechnern läuft hingegen über die vom IP–Stack selbst realisierten Sockets. Um die Verarbeitung der Reservierungsnachrichten zu vereinheitlichen, empfängt ein separater Thread sämtliche von anderen Rechnern über das Netzwerk eintreffenden Reservierungsnachrichten und leitet sie an den Verarbeitungsthread weiter. Dazu wartet er an einem speziellen UDP–Port auf Reservierungsnachrichten. Da die Reservierungskomponente unmittelbar nach dem Start des Servers aktiviert wird, steht dieser Port auch in jedem Falle zur Verfügung.

Dem Verarbeitungsthread wird, zusätzlich zu einem die gewünschte Operation kennzeichnenden Funktionscode, die folgende, die Reservierungseigenschaften charakterisierende, Datenstruktur übergeben:

```
typedef struct {
  struct {
    struct sockaddr_in addr;
  } flow;
  struct {
    unsigned bandwidth;   /* [Bytes/s] */
    unsigned period;      /* [ms]      */
  } quality;
  int priority;
} resvparm_t;
```

Listing 4.11: Datenstruktur zur Aufnahme der Reservierungseigenschaften

Um die Zugehörigkeit eines Paketes zu einer Reservierung prüfen zu können, sind in `flow` die Angaben über Senderadresse und Zielport vermerkt. Die gewünschten Übertragungseigenschaften werden in den Elementen `bandwidth` und `period` gespeichert, während `priority` die Priorität enthält, mit der nichtkonforme Pakete weiterverarbeitet werden sollen.

Der Zugriff auf die lokale Reservierungskomponente wird Applikationen über die folgenden, ebenfalls in der im Abschnitt 4.3 beschriebene Socketbibliothek enthaltenen, Funktionen ermöglicht:

```
int tcpip_makereservation(struct sockaddr_in* flow,
                          unsigned bandwidth,
                          unsigned period, int priority,
                          l4_threadid_t sigthread);
int tcpip_cancelreservation(struct sockaddr_in* flow);
```

Listing 4.12: Funktionen zum Zugriff auf die Reservierungskomponente

Die Reservierungskomponente signalisiert dem in `sigthread` spezifizierten Thread sämtliche Veränderungen an dieser Reservierung. Dies ist dann der Fall, wenn der ein beteiligter entfernter Rechner die Reservierung beendet oder aufgrund von Ressourcenmangel gar nicht erst zustande kommen läßt. Der `sigthread` muß dabei zum selben Task wie der die Funktion ausführende Thread gehören.

4.5 Paket–Scheduler

Der Paket–Scheduler muß Reservierungen individuell für jedes Netzwerkgerät verwalten.[2] Aus diesem Grunde habe ich in die, bereits im BSD–Code definierten, Verwaltungsdatenstrukturen für Netzwerkgeräte zusätzliche Elemente aufgenommen:

```
struct ifnet {
  ...
  struct ifqueue if_snd;
  ...
  /* zusätzliche, reservierungsrelevante, Elemente */
  struct ifqueue if_snd_prio[RESERVATION_PRIOS];
  resvinfo_t      resvinfo;
  resvdatalist_t resvdata_sendlst, pending_resvdata_sendlst;
  resvdatalist_t resvdata_recvlst, pending_resvdata_recvlst;
};
```

Listing 4.13: Erweiterung der Netzwerkgeräte–Verwaltungsdatenstruktur

Neben der in der Datenstruktur bereits enthaltenen Sende–Warteschlange `if_snd` dienen die verschiedenen `if_snd_prio`–Warteschlangen zur Aufnahme der einer Reservierung unterliegenden Pakete entsprechend ihrer Konformität und Bearbeitungspriorität (siehe Abschnitt 4.4). `if_snd` wird jetzt für sämtliche Pakete verwendet, die keiner Reservierung unterliegen. Um die Ressourcenauslastung an einem Netzwerkgerät bewerten zu können, sind die zum Lösen der auf Seite 42 beschriebenen Gleichung 3.5 erforderlichen Werte im Element `resvinfo` gespeichert:

```
typedef struct {
  int total_bandwidth;
  int used_bandwidth;
  int resv_count;       /* Anzahl Reservierungen */
} resvinfo_t;
```

Listing 4.14: Datenstruktur zur Verwaltung der Ressourcenauslastung eines Netzwerkgeräts

Die Listen `resvdata_sendlst` und `resvdata_recvlst` enthalten Informationen zu sämtlichen bestätigten Reservierungen dieses Netzwerkgeräts, während

[2] Obwohl es sich dabei ebenfalls um ein "Netzwerkgerät" handelt, sind Reservierungen für das Loopback–Gerät nicht möglich, da sich eine Obergrenze für die verfügbare Bandbreite nur schwer ermitteln läßt und zeitkritische lokale Übertragungen besser mit anderen Mechanismen, beispielsweise L4–IPCs, durchgeführt werden sollten.

in `pending_resvdata_sendlst` und `pending_resvdata_recvlst` noch nicht bestätigte Reservierungen abgelegt sind. Sobald eine Reservierung vom entsprechenden entfernten Rechner bestätigt wurde, wird das zugehörige Listenelement aus dieser Liste in die der bestätigten Reservierungen übertragen. Die Listenelemente selbst haben folgenden Aufbau:

```
typedef struct resvdata {
  TAILQ_ENTRY(resvdata)  node;
  resvparm_t             resvparm;
  l4_threadid_t          client;
  int                    credits;
  volatile unsigned long lasttransfer;
} resvdata_t;
```

Listing 4.15: Datenstruktur zur Aufnahme der eine Reservierung beschreibenden Informationen

Zusätzlich zum Element `node`, das zum Verketten der Elemente innerhalb der Liste dient und dem die Reservierungseigenschaften beinhaltenden `resvparm` wird in `client` der Identifikator des Threads vermerkt, dem Statusänderungen per L4–IPC mitgeteilt werden sollen. `credits` und `lasttransfer` werden benutzt, um Pakete auf Konformität zu prüfen. Dazu bediene ich mich eines Verfahrens, das bereits von Martin Borriss in seiner L4ATM–Implementierung verwendet wird [Bor99, S.111f.]: Ein konformer Datenstrom darf, entsprechend der Menge reservierter Ressourcen, während eines Intervalls eine bestimmte Datenmenge senden bzw. empfangen. In `credits` ist deshalb vermerkt, welche Menge an Daten im aktuellen Intervall noch verarbeitet werden darf. `lasttransfer` beinhaltet den Zeitpunkt des letzten Datentransfers und wird benutzt, um entscheiden zu können, ob ein neues Intervall begonnen hat. In diesem Fall sind dann natürlich die `credits` wieder aufzufüllen:

```
if(nichtfragmentiertes UDP–Paket && Reservierung vorhanden) {
  if(neues Intervall) {
    credits = Rate;       /* Rate = Datenmenge / Intervall */
  }

  credits -= Paketgröße;

  if(credits >= 0) {
    /* konformes Paket */
  }
  else {
    /* nicht reservierungskonform */
  }
}
```

Listing 4.16: Algorithmus zur Bestimmung der Konformität eines Pakets

Kapitel 5

Leistungsbewertung

5.1 Ethernet–Geräte

Leider erwies sich die Zusammenarbeit zwischen Ether–Server und dem IP–Stack als problematisch. Dies ist aber nicht auf Probleme innerhalb des IP–Servers zurückzuführen. Vielmehr ist der Ether–Server derzeit noch relativ stark hardwareabhängig und war auf meinem Testrechner nicht zur Arbeit zu bewegen. Aus diesem Grund konnte die Leistungsfähigkeit des IP–Stacks bei Datenübertragungen im realen Netzwerk leider nicht bewertet werden.

5.2 Loopback–Gerät

Um einen ungefähren Anhaltspunkt bezüglich der Leistung des IP–Stacks treffen zu können, habe ich deshalb die Zeiten verglichen, die zum lokalen Transport von UDP–Datagrammen verschiedener Größe auf unterschiedlichen Systemen unter Verwendung des Loopback–Devices (siehe Abschnitt 3.1.4) erforderlich ist. In Abbildung 5.1 sind die Übertragungszeiten in graphischer Form dargestellt. Um Meßfehler und Schwankungen zu reduzieren, wurden jeweils 100000 Pakete übermittelt und die Ergebnisse gemittelt. Die verwendeten Meßprogramme sind im Anhang B aufgeführt.

Als Testumgebung kam ein Rechner des Typs *Siemens–Nixdorf PCD–5H* mit einem mit 100 MHz getakteten Pentium–Prozessor (256 KB Level–2–Cache) und 32 MB Hauptspeicher zum Einsatz. Als Betriebssysteme wurden LINUX (Kernelversion 2.2.10) und das zu L4 binärkompatible *Fiasco* [Hoh] verwendet.

Die vorliegende Version des IP–Servers verwendet indirect strings innerhalb der Socket–Schnittstelle zum Datentransport zwischen Klienten und Server. Darüberhinaus sind, bedingt durch das mbuf–System, einige zusätzliche Kopieroperationen erforderlich. Um den Einfluß dieser Kopieroperationen bewerten zu können, habe ich eine rudimentäre Serverversion gebaut, die gemeinsamen Speicher innerhalb der Socketschnittstelle nutzt. Die Übertragungszeiten dieser Version wurden ebenfalls gemessen.

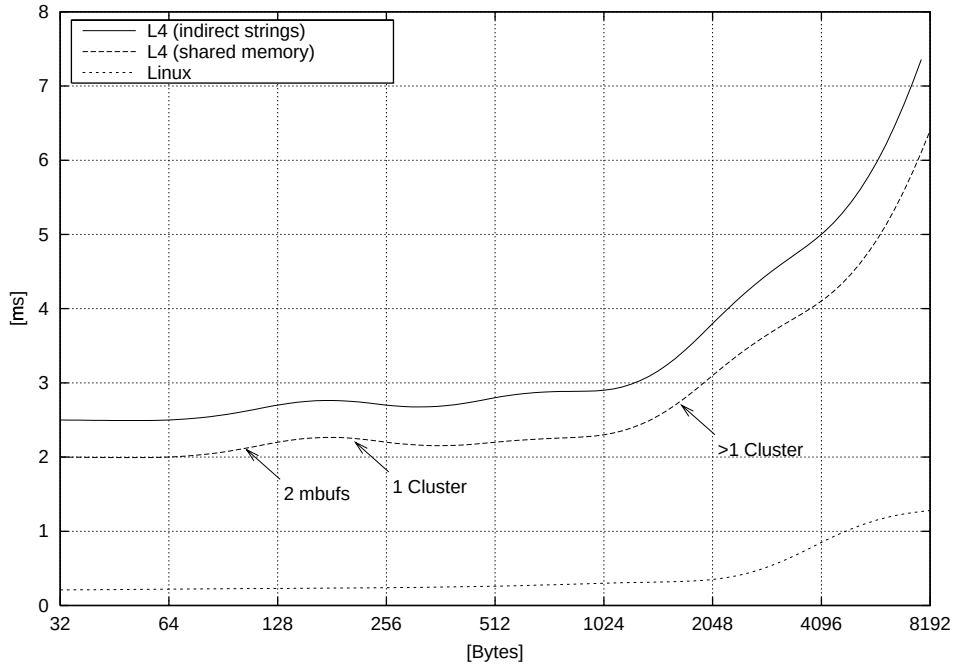


Abbildung 5.1: Leistungsvergleich zwischen verschiedenen IP–Implementierungen

Die L4–Implementierungen schneiden im Vergleich zu der von LINUX relativ schlecht ab. Dies ist einerseits natürlich durch die zusätzlich erforderliche IPC–Kommunikation zwischen Klient und Server begründet. Zum anderen bedingt aber auch das mbuf–System einen erheblichen Teil des Leistungsverlustes. Experimentelle IP–Implementierungen [Jac93] zeigten beispielsweise, daß Geschwindigkeitsverbesserungen um ein bis zwei Größenordnungen erzielbar sind, wenn der Entwurf der Implementierung auf das Erreichen maximaler Leistung mit moderner Hardware ausgerichtet ist.[1] [WS95, S.60]

Interessant ist der Leistungsgewinn der gemeinsamen Speicher nutzenden Implementierung im Vergleich zu derjenigen mit indirect strings. Eine Leistungsverbesserung von etwa 20 % läßt sich allein durch den Verzicht auf einen *Long IPC* erzielen. Eine Reduzierung der durch die mbufs bedingten Kopieroperationen sollte also eine weitere wesentliche Leistungsverbesserung ermöglichen.

[1] Neben dem Verzicht auf mbufs wurden in dieser Implementierung aber auch zahlreiche weitere Verbesserungen vorgenommen.

Kapitel 6

Zusammenfassung, Ausblick

Das Ziel der Arbeit, die Entwicklung eines zusagenfähigen IP–Stacks auf der Basis der Referenzimplementierung des Systems 4.4BSD–Lite, ist erreicht worden.

Der in einem separaten L4–Server gekapselte IP-Stack ist dabei in der Lage, sowohl einen "normalen" Datentransport über die Protokolle TCP und UDP zu realisieren als auch im Vorfeld der Übertragungen Zusagen hinsichtlich der erreichbaren Dienstgüte treffen zu können und diese beim Datentransfer durchzusetzen. Diese Funktionalität ist dabei über eine BSD–Socket–Schnittstelle zugänglich, die lediglich geringfügig um Funktionen zur Ressourcenreservierung erweitert wurde.

Leider konnte im Rahmen der Arbeit aus Zeitgründen keine auf standardisierten Protokollen, beispielsweise RSVP, aufbauende Reservierungskomponente realisiert werden. Die implementierte Version orientiert sich zwar an der Arbeitsweise von RSVP, soll in erster Linie aber lediglich die Funktionsfähigkeit der Reservierungsdurchsetzung demonstrieren. Aufbauend auf der vorhandenen Lösung kann diese aber mit geringem Aufwand durch eine standardisierte Komponente ersetzt werden.

Aufgrund des Fehlens eines funktionsfähigen Ethernet–Treibers konnte leider keine besonders aussagekräftige Bewertung der Leistungsfähigkeit der Implementierung vorgenommen werden. Lediglich Messungen am Loopback–Gerät ließen eine Bewertung zu. Aufgrund einiger Kopieroperationen innerhalb des IP–Servers schneidet meine Implementierung im Vergleich zu der von LINUX relativ schlecht ab. Ein Ersatz der zur Speicherverwaltung verwendeten mbufs durch eine moderne Lösung kann diesen Nachteil aber durchaus beseitigen.

Literaturverzeichnis

[Alm97] W. Almesberger. *ATM on Linux.* 1997. http://lrcwww.epfl.ch/linux-atm/.

[BBD+97] M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner. *Linux–Kernel–Programmierung.* Addison–Wesley, 1997.

[Bor99] M. Borriss. *Operating System Support for Predictable High-Speed Communication.* Dissertation. Technische Universität Dresden, 1999.

[Bra99] T. Braun. *IPnG: Neue Internet–Dienste und virtuelle Netze.* dpunkt.verlag, 1999.

[BSD] 4.4BSD–Lite–Quellcode. ftp://ftp.cdrom.com/pub/bsd_sources/4.4BSD-Lite.tar.gz.

[BZBH97] R. Braden, L. Zhang, S. Berson, and S. Herzog. *Resource Reservation Protocol (RSVP) — Functional Specification*, September 1997. RFC 2205.

[Dan98] U. Dannowski. *names – DROPS Name Service.* Technische Universität Dresden, Institut für Betriebssysteme, Datenbanken und Rechnernetze, Professur Betriebssysteme, 1998. http://os.inf.tu-dresden.de/~ud3/names.html.

[Dee89] S. E. Deering. *Host Extensions for IP Multicasting*, 1989. RFC 1112.

[DH95] S. Deering and R. Hinden. *Internet Protocol, Version 6 (IPv6)*, 1995. RFC 1883.

[FBB+99] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. *The OSKit: The Flux Operating System Toolkit.* University of Utah, Department of Computer Science, Juli 1999.

[Ham] Cl.–J. Hamann. Quantitative Methoden in Betriebssystemen. Vorlesungsskript der gleichnamigen Lehrveranstaltung. TU Dresden, Fakultät Informatik, Institut für Betriebssysteme, Datenbanken und Rechnernetze.

[HBB+98] H. Härtig, R. Baumgartl, M. Borriss, Cl. J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. DROPS — OS Support for Distributed Multimedia Applications. In *8th ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[Hoh] M. Hohmuth. *The Fiasco µ–Kernel.* Technische Universität Dresden, Institut für Betriebssysteme, Datenbanken und Rechnernetze, Professur Betriebssysteme.
http://os.inf.tu-dresden.de/fiasco.

[ISI] RSVP–Implementierung des Information Sciences Institute der University of Southern California, Los Angeles.
http://www.isi.edu/rsvp.

[Jac93] V. Jacobson. *Some Design Issues for High–Speed Networks.* Networkshop '93, Melbourne, 1993.
ftp://ftp.ee.lbl.gov/papers/vj-nws93-1.ps.Z.

[Lie95] J. Liedtke. On µ–Kernel Construction. In *ACM Symposium On Operating System Principles (SOSP)*, Copper Mountains CO, Dezember 1995.

[MBKQ96] K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System.* Addison–Wesley, 1996.

[Mog93] J. C. Mogul. *IP Network Performance.* Internet System Handbook, S. 575–675. Addison–Wesley, 1993.

[Nag84] J. Nagle. *Congestion Control in IP/TCP Internetworks*, 1984. RFC 896.

[NBF96] B. Nichols, D. Buttlar, and J. Proulx Farrell. *Pthreads Programming: "A POSIX Standard for Better Multiprocessing".* O'Reilly, 1996.

[PC93] D. M. Piscitello and A. L. Chapin. *Open Systems Networking: TCP/IP and OSI.* Addison–Wesley, 1993.

[Pos80] J. Postel. *User Datagram Protocol*, 1980. RFC 768.

[Pos81a] J. Postel. *Internet Protocol*, 1981. RFC 791.

[Pos81b] J. Postel. *Transmission Control Protocol*, 1981. RFC 793.

[Ros90] M. Rose. *The Open Book: A Practical Perspective on OSI.* Prentice–Hall, 1990.

[Tha99] T. L. Thai. *Learning DCOM.* O'Reilly, 1999.

[Top90] C. Topolcic. *Experimental Internet Stream Protocol: Version 2 (ST–II)*, Oktober 1990. RFC 1190.

[Wro97] J. Wroclawski. *The Use of RSVP with Integrated Services*, September 1997. RFC 2210.

[WS95] G. S. Wright and R. W. Stevens. *TCP/IP Illustrated, Volume 2: The Implementation.* Addison–Wesley, 1995.

Anhang A

Berkeley Public Licence

Anhang B

Testprogramme

```
/*
 * Sender
 */
#include <l4/libtcpip.h>
#include <oskit/c/stdio.h>
#include <oskit/c/arpa/inet.h>
#include <oskit/c/netinet/in.h>
#include <oskit/freebsd/sys/errno.h>

int main(void)
{
  int s;
  struct sockaddr_in sin;
  static char buf[DATASIZE];
  int i,err;

  for(i=0; i<100000; i++) {
    s = socket(AF_INET,SOCK_DGRAM,0);

    sin.sin_len=sizeof(struct sockaddr_in);
    sin.sin_family=AF_INET;
    sin.sin_port=htons(3000);
    inet_aton("127.0.0.1",&sin.sin_addr);

    err = sendto(s,buf,DATASIZE,0,
                 (struct sockaddr*)&sin,sizeof(struct sockaddr_in));
    if (err == -1) {
      printf("sendto failed (errno: %d)\n",errno);
    }
    close(s);
  }
}
```

```
/*
 * Receiver
 */
#include <l4/libtcpip.h>
#include <oskit/c/stdio.h>
#include <oskit/c/arpa/inet.h>
#include <oskit/c/netinet/in.h>
#include <oskit/freebsd/sys/errno.h>

int main(void)
{
  int s;
  struct sockaddr_in sin;
  struct sockaddr_in remote;
  int fromlen;
  static char buf[DATASIZE];
  int i,err;

  s = socket(AF_INET,SOCK_DGRAM,0);

  sin.sin_len=sizeof(struct sockaddr_in);
  sin.sin_family=AF_INET;
  sin.sin_port=htons(3000);
  sin.sin_addr.s_addr=htonl(INADDR_ANY);

  bind(s,(struct sockaddr*)&sin,sizeof(struct sockaddr_in));

  for(i=0; i<100000; i++) {
    fromlen = sizeof(struct sockaddr_in);
    err = recvfrom(s,buf,DATASIZE,0,(struct sockaddr*)&remote,&fromlen);
    if (err == -1) {
      printf("Server: recvfrom failed (errno: %d)\n",errno);
    }
  }

  close(s);
}
```