Netzmonitoring auf ATM-Ebene

Diplomarbeit im Fach Informatik

vorgelegt von

Falko Dreßler

geboren am 2.12.1971 in Dresden

Angefertigt am

Institut für Mathematische Maschinen und Datenverarbeitung IV

Friedrich-Alexander Universität Erlangen-Nürnberg

Betreuer: **Dr. Peter Holleczek (RRZE)**

Prof. Dr. Fridolin Hofmann

Beginn der Arbeit: 3. Februar 1998

Ende der Arbeit: 23. Juli 1998

Erklärung

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 23. Juli 1998

Falko Dreßler

Danksagung

Hiermit möchte ich mich bei allen Mitarbeitern des Regionalen Rechenzentrums (RRZE) sowie des Lehrstuhls für Betriebssysteme (IMMD4) der Friedrich-Alexander Universität Erlangen-Nürnberg für die tatkräftige und bereitwillige Unterstützung bedanken.

Vor allem aber geht mein Dank an Herrn Dr. Peter Holleczek. Ohne seine Unterstützung hätte ich die Arbeit nicht in diesem Umfang und der relativ knappen Zeit bewerkstelligen können.

Inhaltsverzeichnis

Vorwort

Seit der Einführung von Computernetzwerken und insbesondere durch das unaufhaltsame und vor allem extrem schnelle Wachstum des Internet ist man auf der Suche nach geeigneten Werkzeugen, um sowohl eine Abrechnung genutzter Ressourcen durchführen zu können, als auch eine Erkennung und Rückverfolgung von unerlaubten Zugriffen über das Netz.

Dieses Streben prägte zwei zentrale Begriffe: Monitoring und Accounting.

Die Aufgabe des Monitorings ist das Sammeln von Verbindungsinformationen der Datenübertragungen über Netzwerke. Diese Daten dienen als Basis für eine weitere Auswertung und damit als Kontrollmechanismus für die Netzsicherheit, aber auch als Grundlage einer strafrechtlichen Verfolgung von böswilligem Verhalten.

Ausgehend von den durch das Monitoring gesammelten Daten, kommt dem Accounting die Aufgabe des Sammelns von Informationen über die von bestimmten Rechnern (Hosts) bzw. Teilnetzen (Subnetworks) verbrauchten Netzwerkressourcen, d.h. die gezielte Auswertung der aufgezeichneten Daten, zu. Allgemein versteht man darunter im einfachsten Fall die Anzahl übertragener Byte pro Zeiteinheit, aber zunehmend auch die zur Verfügung gestellte Qualität einer Netzwerkverbindung. Zusammenfassend kann man also sagen, das Accounting wird für die Abrechnung und Kontrolle der zur Verfügung gestellten Leistungen (in Form von Netzwerkressourcen) benötigt.

Immer mehr im Mittelpunkt des Accounting steht auch die Erkennung unerlaubter Aktivitäten im Netzwerk. Für eine wirksame Unterbindung dieser Aktionen ist es wichtig, daß das Accounting zeitlich gekoppelt mit dem Aufzeichnen der Daten erfolgt, um rechtzeitig Gegenmaßnahmen einleiten oder zumindest Warnungen ausgeben zu können.

An der Universität Erlangen-Nürnberg wurde in der Vergangenheit ein Programmpaket entwikkelt, mit dem sich eine Datensammlung, aber auch eine komplette Auswertung der gefundenen Informationen abwickeln läßt. Dieses, “Moni-Box” genannte Softwarepaket, wurde für den Einsatz in Ethernet und FDDI Netzwerken konzipiert. Analysierbare Netzwerkprotokolle sind z.B. IP, TCP, UDP oder IPX.

Vor allem in Weitverkehrsnetzen, aber auch in größeren lokalen Netzwerken, wird zunehmend auf ATM[1]-Technologie gesetzt. Bedingt durch die gute Skalierbarkeit und die Möglichkeit der Vorgabe von QoS-Merkmalen[2] ist das ATM-Protokoll für diese Netzwerke ideal.

Um Monitoring von ATM-Netzen zu betreiben, muß man derzeit auf sehr teure Spezialgeräte, sogenannte ATM-Monitore, zurückgreifen. Diese sind zudem in ihrem Einsatzgebiet stark eingeschränkt und bieten nur geringe bis gar keine Accounting-Mechanismen.

Eine neue, auf preiswerten Arbeitsplatzcomputern einsetzbare Lösung ist zu suchen.

Ziel der vorliegenden Arbeit ist es, die Machbarkeit eines Werkzeugs für das Monitoring von ATM-Netzen zu untersuchen. Die wesentliche Vorraussetzung dafür war ein ausgewogenes Ver-

1. Asynchronous Transfer Mode
2. Quality of Service - Dienstgüte einer Netzwerkverbindung (z.B. Übertragungsrate oder Verzögerung)

hältnis zwischen guter Portabilität (es werden unterschiedenen Arbeitsplattformen eingesetzt) und hoher Performance.

Zur Verfügung stehen eine Sun ULTRA 1/170 unter Solaris 2.6 mit zwei SunATM-Karten und ein i586-PC mit zwei Fore-PCI-ATM-Karten. Der i586-PC wurde unter dem frei erhältlichen UNIX-Betriebssystem FreeBSD 2.2.5 betrieben. Die Anschlußrate der verwendeten ATM-Karten beträgt max. 155MBit/s.

Die Arbeit ist in folgende Kapitel unterteilt:

In Kapitel 1 “ATM-Netzwerke” soll die Technik von ATM-Netzwerken erläutert sowie einige für ATM typische Anwendungen besprochen werden. Besondere Bedeutung kommt dem Abschnitt über die Analyse von ATM-Verkehr zu, in welchem der Aufbau von ATM-Zellen und AAL[3]-PDUs beschrieben wird.

In Kapitel 2 “Einsatzumgebung für einen ATM-Logger” werden die technischen Möglichkeiten des Zugriffs auf den ATM-Datenstrom und die zur Verfügung stehende Hardware incl. Betriebssystem- und Treibersoftware diskutiert. Ziel ist es, erste Informationen über die Leistungsfähigkeit der eingesetzten Rechner zu gewinnen.

In Kapitel 3 “Entwurfsüberlegungen für den Aufbau eines ATM-Monitors” sollen allgemeine Fragen zur Entwicklung und Implementierung eines ATM-Monitors geklärt werden. So wird z.B. ein Mehrprozeßsystem für den strukturellen Aufbau des Gesamtsystems entworfen. Die gefundenen Aussagen sind Hardware- und Betriebssystemunabhänig.

Im zentralen Kapitel 4 “Vergleich struktureller Ansätze der Kommunikation zwischen Kernel- und Benutzerprozessen” werden Techniken des Sammelns, der Übergabe und Speicherung von Verbindungsdaten beschrieben. Es werden verschiedene Ansätze untersucht und getestet. Ziel ist es, Aussagen über die Performance eines ATM-Monitors auf den eingesetzten Arbeitsplatzcomputern zu treffen.

In Kapitel 5 “Ankopplung von Benutzerprozessen zur Protokollanalyse”, werden Möglichkeiten der Ankopplung von weiteren Benutzerprozessen zur Protokollanalyse an die eigentliche Monitoringeinheit entworfen und Lösungsansätze der schon erwähnten Moni-Box beschrieben. Um weitere Aussagen über die Implementierbarkeit auf Arbeitsplatzrechnern zu finden, wird die bereits entwickelte Funktionalität des ATM-Monitors in das Programmpaket Moni-Box integriert bzw. die Integration rudimentär durchgeführt und die weitere Vorgehensweise beschrieben.

3. ATM Adaption Layer

1 ATM-Netzwerke

Um ein Werkzeug für das Monitoring von ATM-Verkehr zu erstellen, ist es zuerst nötig, sich einen Überblick über die technischen Grundlagen von ATM zu erstellen. Wichtige Gesichtspunkte sollen hier sein:

- allgemeiner Aufbau des ATM-Protokolls
- Funktionsweise der ATM-Schicht und der ATM-Anpassungsschicht
- Dienstgütemerkmale und Serviceklassen
- Adressierungsschemata und Signallingstrategien

Für weitere Informationen zu ATM verweise ich hiermit auf weiterführende Literatur, wie z.B. [2], [6], [7] und auch auf [20].

Weiterhin sollen einige wichtige Anwendungen von ATM erwähnt und als Vorbereitung für die Analyse von ATM-Verkehr die konkreten Protokolldetails für ATM-Zellen und AAL-PDUs eingeführt werden. Anschließend wird der Aufbau des Protokolls “Classical IP over ATM” erläutert.

1.1 Technik

ATM wurde speziell als Transportprotokoll für Anwendungen, die hohe Datenraten über große Entfernungen bei extremen Qualitätsanforderungen erwarten, konzipiert. Verfügbar sind Übertragungsgeschwindigkeiten von 25Mbps bis über 2,4Gbps. Für die Übermittlung von z.B. Audio- oder Videosequenzen sind Dienstgüteparameter (QoS[1]) wie die für die Anwendung verfügbare Bandbreite oder das maximale Delay vorgebbar. ATM ist ein integrierter Service, da “beliebig viele” VCs mit unterschiedlicher QoS für eine Gesamtverbindung definierbar sind.

Von der ITU-T[2] ist ATM als Schichtenprotokoll eingeführt wurden, welches sich am OSI[3]-Modell der ISO[4] orientiert. Die folgende Abbildung zeigt einen Überblick über die Protokollhierarchie von ATM und die Aufgaben der einzelnen Schichten.

1. Quality of Service
2. International Telecommunications Unision’s Telecommunications Standardization Sector (früher CCITT - Comité Consultatif Internationale Télégraphique et Téléfonique)
3. Open Systems Interconnection
4. International Standards Organisation

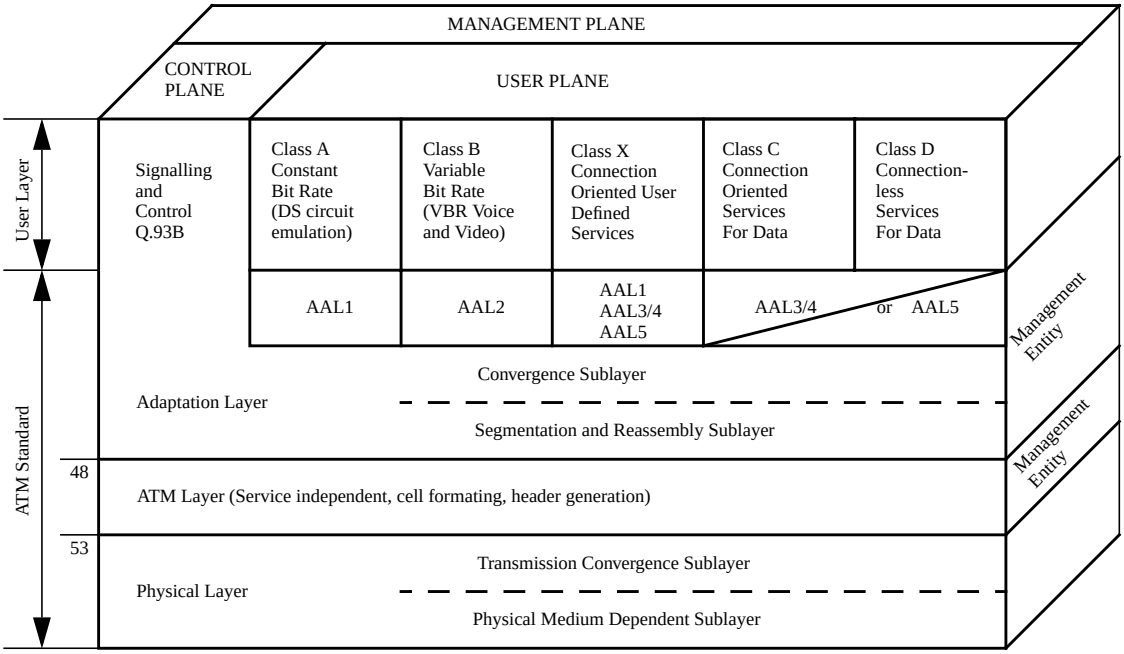


Abbildung 1.1 - ATM-Modell [21]

1.1.1 ATM Layer

Die ATM-Schicht ist vom physikalischen Medium, das zum Transport der Zellen verwendet wird, und damit von der physikalischen Schicht, völlig unabhängig [7].

Die wichtigsten Aufgaben dieser Schicht sind: Multiplexen und Demultiplexen von Zellen verschiedener Verbindungen; Bereitstellung von Dienstqualitäts-Klassen; Verwaltungsfunktionen und Flußsteuerung.

1.1.2 ATM Adaption Layer (AAL)

Die ATM-Anpassungsschicht verbessert den von der ATM-Schicht bereitgestellten Dienst auf ein von der nächsthöheren Schicht benötigtes Maß. Sie erfüllte Funktionen für die Anwender-, Steuerungs- und Verwaltungsschichten und unterstützt die Anpassung zwischen der ATM-Schicht und der nächsthöheren Schicht [7].

Wie in Abbildung 1.1 zu sehen ist, ist diese Schicht zweigeteilt. Die Unterschicht für Teilung und Wiederherstellung (SAR) übernimmt die Aufgabe der Segmentierung der von der höheren Schicht kommenden Daten für die Versendung über das ATM-Netz bzw. den Zusammenbau von Teildatenelementen. Die Konvergenzunterschicht (CS) übernimmt Funktionen wie die Indentifikation von Nachrichten und die Zeit/Taktauffrischung.

An der Konvergenzunterschicht können die zu übertragenden Datenblöcke noch bis zu 64 kByte umfassen. Erst beim Übergang zur SAR-Teilschicht werden diese Blöcke zu kleinen Abschnitten aufbereitet, die als Payload für ATM-Zellen (53Byte Gesamtgröße, 48Byte Nutzdaten) geeignet sind.

Aus der Abbildung 1.1 ist weiterhin zu entnehmen, daß fünf verschiedene Typen der ATM-Anpassungsschicht definiert wurden. Sie unterscheiden sich vor allem in Übertragungssicherheit (Fehlererkennung / -erholung) und in den Möglichkeiten der Garantierung von Dienstgüteeigenschaften (siehe auch Abschnitt 1.1.3 oder 1.2).

1.1.3 Service classification of the AAL

Die folgende Tabelle zeigt die für ATM definierten Dienstklassen und deren Eigenschaften:

Transmission characteristic	**Class A**	**Class B**	**Class C**	**Class D**
AAL type	AAL1	AAL2	AAL3/4 AAL5	AAL3/4 AAL5
Timing relation between source and destination	Required	Required	Not required	Not required
Bit rate	Constant	Variable	Variable	Variable
Connection mode	Connectionoriented	Connectionoriented	Connectionoriented	Connectionless
Example communication services within this class	Circuitswitched or leaseline-like connections (e.g. telephone, E1, T1, n x 64 kbit/s etc.)	Packet audio or video signals	Frame relay, X.25 etc.	IP (Internet), SMDS etc.

Tabelle 1.1 - AAL Service Klassen [6]

In Kapitel 1.2 sollen die wichtigsten Anwendungen eines ATM-Netzwerkes / einer ATM-Verbindung beschrieben werden. Obige Tabelle liefert dafür die technologischen Grundlagen. Z.B. ist AAL1 ein verbindungsorientierter Dienst mit konstanter Bitrate, wodurch dieses Protokoll für Applikationen, die extreme Anforderungen an die Verbindung stellen (kleines maximales Delay, kleiner Jitter, konstante Bitrate) prädestiniert ist.

1.1.4 VCs und Routing

Eine ATM-Verbindung, d.h. ein ATM-VC[5] wird im wesentlichen durch zwei Parameter charakterisiert: dem VPI[6], der Pfadidentifikation, und dem VCI[7], der Kanalidentifikation. Beide zusammen identifizieren genau die Verbindung zwischen zwei Netzknotenpunkten.

Es gibt zwei Möglichkeiten, einen VC aufzubauen:

(1) PVC[8]: man konfiguriert per Hand einen statischen PVC, über den dann Daten übertragen werden können.

(2) SVC[9]: man baut Verbindungen dynamisch auf. Der Weg durch das ATM-Netzwerk wird durch das “Signalling” festgelegt.

Das Signalling, oder auch “qos based routing” (aus [8]) genannt, ist das Verfahren, Wege (Routen) durch ein u.U. sehr komplexes Netz zu finden. Das Signalling-Protokoll (vergleiche [43]) beschreibt die möglichen Verfahren, diese Routen effizient und Protokollkonform zu finden und dadurch ATM-VCs aufzubauen.

Man unterscheidet zwei verschiedene Arten von Routern (im ATM-Jargon Switches genannt): VP-Switches und VC-Switches. Die VP-Switches “routen” (besser “switchen”) VCs nur anhand der VPI-Information, d.h. sie bilden eine Art Weg (engl. path) durch das ATM-Netzwerk. Alle VCs, die denselben Pfad benutzen werden gebündelt auf denselben Weg gerouted. Die VC-Switches hingegen nutzen für die Wegewahl zusätzlich die VCI-Information, d.h. für jeden einzelnen VC ist ein eigener Weg durch das ATM-Netzwerk vorgebbar (siehe Abbildung 1.2).

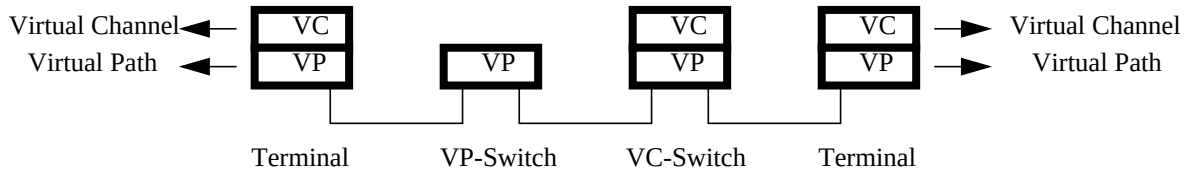


Abbildung 1.2 - VC- und VP-Verbindungen [7]

1.1.5 Adressierung

Im Vergleich zu Adressierungsschemata anderer Netzwerkprotokolle ist eine ATM-Adresse mit 20Byte relativ lang. IPv4 benutzt z.B. nur 4Byte und die neue Version IPv6 nur 6Byte.

Es gibt drei verschiedene Möglichkeiten eine ATM-Adresse zu kodieren:

5. Virtual Channel
6. Virtual Path Identifier
7. Virtual Channel Identifier
8. Permanent Virtual Circuit
9. Switched Virtual Circuit

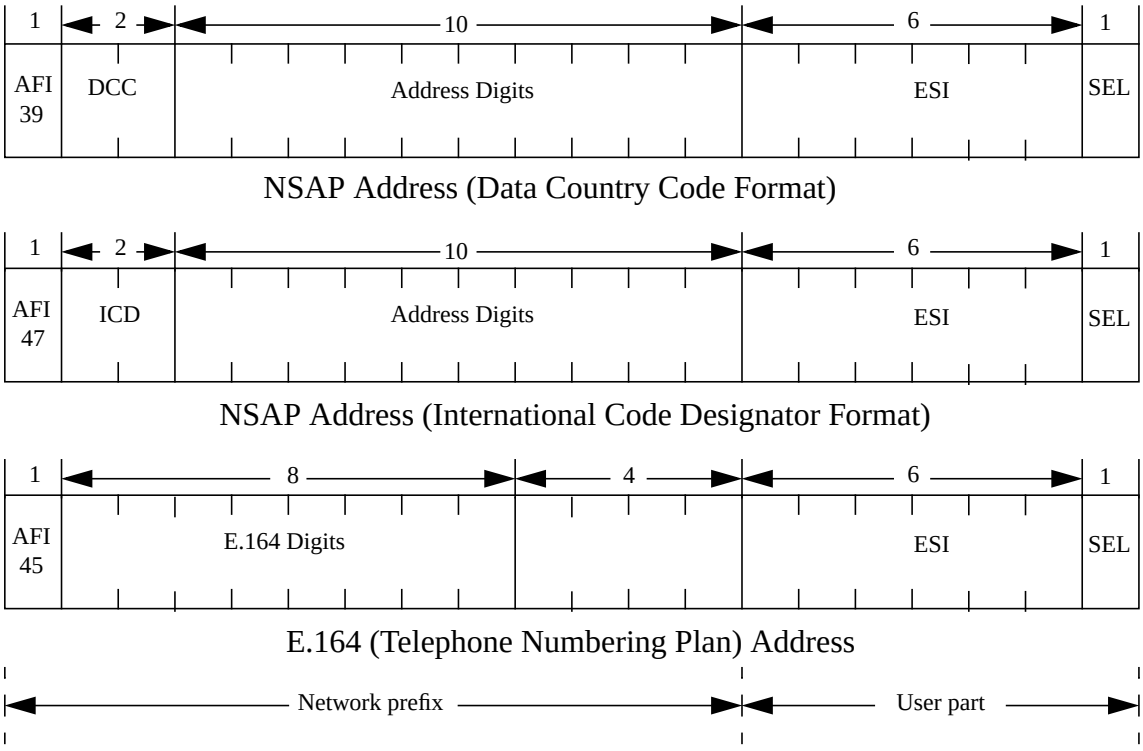


Abbildung 1.3 - ATM-Adressierungsschemata [44]

Die ersten 13Byte der Adresse beziehen sich auf das Netzwerk, wobei der AFI[10] im ersten Byte angibt, welches Adressierungsschema benutzt wird. Der ESI[11] kennzeichnet eindeutig (z.B. durch eine der Ethernet-MAC-Adressen ähnlichen Struktur) das Endsystem.

Die NSAP[12]-Adressen sind hierarchisch organisiert, um das Routing zu vereinfachen. Dies ähnelt dem Subnetzkonzept in IP-Netzwerken.

Über eine spezielle Konfigurationsprozedur werden dem lokalen ATM-Switch alle nötigen Informationen über die angeschlossenen Stationen mitgeteilt. Diese Prozedur wird immer bei beim Neustart des Netzes, bei neuen oder entfernten Stationen durchgeführt.

Weiterführende Informationen zur Adressierung und dem Signalling sind in [43] und [44] zu finden.

10. Authority and Format Identifier
11. End System Identifier
12. Network Service Access Point

1.2 Anwendungen

In diesem Abschnitt sollen kurz die wichtigsten Anwendungsgebiete der ATM-Technologie beschrieben werden (siehe auch Tabelle 1.1 auf Seite 5). D.h. es werden die gebräuchlichsten Protokolle erwähnt, die ATM-Dienste in Anspruch nehmen mit Hinblick auf die verschiedenen Anpassungsschichten, die im letzten Kapitel beschrieben wurden.

1.2.1 Raw-ATM

Raw-ATM oder auch “cell relay”, d.h. eine Übertragung unter Umgehung der Anpassungsschicht (auch AAL0 genannt) ist wenig gebräuchlich. Erwähnung soll diese Möglichkeit der Nutzung von ATM-Netzwerken vor allem deshalb finden, da einige Werkzeuge zur Messung der von einem Netzprovider garantierten Dienstgütemerkmale auf AAL0 aufsetzen. Ein Beispiel ist “`tmt4atm`” (siehe auch [20]).

1.2.2 MPOA

Das vom ATM Forum spezifizierte MPOA[13] ist die erste standardisierte Lösung, die es erlaubt, gerouteten Netzwerken die Vorteile von ATM zu sichern. MPOA setzt auf Protokollen wie LANE oder Classical IP over ATM auf, um eine standardisierte Notation eines virtuellen Routers zu schaffen.

Der Einsatz von MPOA hilft, die Latenzzeit in einem Computer-Netzwerk zu reduzieren, da sich das gesamte ATM-Netz für höhere Protokolle wie ein einzelner Hop darstellt. Dadurch wird die Gesamtzahl der Hops, bei der eine Verarbeitung transportierter Pakete durch Router vorgenommen wird, minimiert.

Die wichtigsten Komponenten des “switched-routing”-Ansatzes sind [35]:

- “Route Servers” - diese integrieren Routingintelligenz (Ebene 3) in eine geswitchte (Ebene 2) Transportinfrastruktur.
- “Edge Devices” - über diese werden traditionelle LANs an ein MPOA fähiges ATM-Netzwerk angeschlossen (z.B. Ethernet, Token Ring, etc.)

Für viele Betreiber größerer Netzwerke (LANs / MANs) stell die derzeitige Lösung eines aus vielen Subnetzen, die durch Router verbunden werden ein Performanceproblem dar. “Interne” Verbindungen gehen zumeist über mindestens einen Router (Hop). Mit MPOA und geswitchten VLANs[14] versucht man diese Probleme in den Griff zu bekommen [31], [32].

13. Multi-Protocol over ATM
14. Virtual LAN

Per Voreinstellung benutzt MPOA die LLC[15] Encapsulation nach RFC 1483 [26]. Mindestens diese Datenkapselung muß jede MPOA-Implementation unterstützen. Im LLC-Header wird durch ein reserviertes Feld das über MPOA transportierte Protokoll identifiziert.

Die genaue Protokollspezifikation von MPOA findet man in [36].

1.2.3 LANE / Classical IP over ATM

Für die Übertragung von IP über ATM-Netze wurden zwei verschiedene Möglichkeiten entwikkelt: “LANE” und “Classical IP over ATM”.

LANE[16] ist vor allem für den stufenlosen Übergang von traditionellen lokalen Netzen zu ATM-LANs gedacht, indem verschiedene LANs durch ein LANE-Segment gekoppelt werden. Sowohl Ethernet, als auch Token-Bus können einfach über ein LANE-Segment weitergeleitet (bridged) werden. Leider ist LANE nur für eine kleine Anzahl von Ebene-1-Protokollen ausgelegt, so daß der direkte Anschluß an andere Netze wie z.B. FDDI nicht möglich ist.

Classical IP over ATM soll als kompletter Ersatz von LAN-Segmenten dienen. Da heute immer mehr Endgeräte mit ATM-Hardware bestückt sind, ist es eine logische Folge, diese Ressourcen auch für die klassischen Dienste wie IP zu nutzen.

Ziel von Classical IP over ATM ist es, eine kompatible und interoperable Basisimplementation für die Übermittlung von IP-Datagrammen und ATM-ARP-Anfragen und -Antworten über AAL5 zu schaffen [27].

Nach RFC1755 [28] ist auch eine Unterstützung des ATM Signallings für Classical IP over ATM definiert. Die einzelnen Endgeräte können so die lokalen ATM-Signalling-Einheiten nutzen, um IP Verbindungen aufzubauen bzw. zu beenden.

1.2.4 Frame-Relay over ATM

Frame-Relay (siehe auch [42]) wurde entwickelt, um Informationen mit minimalem Delay sehr effizient zu übertragen. Normalerweise wird dieses Protokoll benutzt, um Datenkommunikationen zu unterstützen, die große Datenpakete mit enormen Spitzenleistungen übertragen. Frame-Relay benötigt den Übertragungskanal nur, wenn wirklich Nutzdaten vorliegen. Leider bietet das Protokoll keine garantierte Performance und nur schlechte Fehlererkennungs- bzw. -erholgungsmechanismen. Daher ist es z.B. für den Transport von Videosequenzen ungeeignet. Trotzdem wird das Protokoll sehr häufig auf Weitverkehrsstrecken mit Übertragungsraten von 56kbps bis 2Mbps eingesetzt.

Die große Bandbreite des ATM-Netzes ausnutzend wird Frame-Relay over ATM in WANs genutzt.

---

15. Logical Link Control
16. LAN-Emulation

1.2.5 Audio, Video (MPEG 1/2/3, M-JPEG)

Für die Übertragung von Audio und Video wird ein kontinuierlicher Datenstrom über die Netzverbindungen vorausgesetzt. Wichtig sind Delay-Management und das Timing zwischen Quelle und Ziel. AAL1 stellt eine konstante Bitrate (CBR) zur Verfügung und sichert zudem die korrekte Sequenzfolge übertragener AAL1-PDUs. Das ist der Grund, warum sowohl die ITU-T als auch das ATM Forum AAL1 für Audio empfehlen. Für Videoübertragungen wird wurde von der ITU-T zwar die ATM Anpassungsschicht 2 vorgesehen, welche das Segmentieren großer Datenblocke zu kleineren Einheiten (Zellen) und das Wiederzusammenführen dieser erlaubt, aber diese Empfehlung hat sich nicht in der Praxis durchsetzen können. Immer mehr wird auch AAL5 als Trägerprotokoll benutzt.

Für Audioübertragungen guter Qualität reichen noch 64kbps, aber für eine sinnvolle Videoübertragung sind bis zu 384kbps nötig. ATM qualifiziert sich durch die zur Verfügung stehenden Dienstgütevorgaben wie auch durch die Übertragungskapazität für diese Aufgaben [25].

1.3 Analyse von ATM-Verkehr

Für die Analyse von aufgezeichnetem Netzwerkverkehr, hier ATM, ist es nötig, die genaue Protokolldefinitionen zu kennen. Im folgenden wird der Aufbau von ATM-Zellen, AAL-PDUs und “Classical IP over ATM”-Paketen beschrieben.

1.3.1 Raw-ATM

Viele Anwendungen setzen nicht auf höhere Protokolle wie z.B. IP auf, sondern versuchen diesen Overhead zu umgehen, indem sie direkt untere Protokollschichten benutzen. Aber es ist auch sonst von Interesse, eine Analyse der ATM-Protokollebenen durchzuführen. Z.B. ist nur dadurch eine genaue Abrechnung des verursachten Netzwerkverkehrs möglich.

1.3.1.1 Aufbau von ATM-Zellen

Abbildung 1.4 zeigt den Aufbau einer ATM-Zelle am UNI[17]. Eine ATM-Zelle hat eine Länge von 53 Byte. Der Nutzdatenanteil (Payload) davon sind 48 Byte. Die Abschnitte einer ATM-Zelle haben folgende Bedeutung:

17. User-Network-Interface

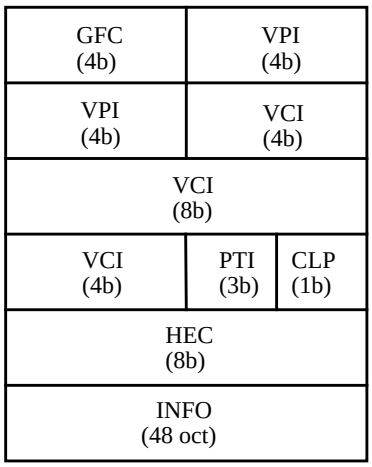


Abbildung 1.4 - Aufbau einer ATM-Zelle [7]

1.3.1.2 Aufbau von AAL-PDUs

Der Aufbau von AAL-PDUs, d.h. Datenpaketen aus der ATM-Anpassungsschicht, die erst später in ATM-Zellen zerlegt werden soll hier nicht näher beschrieben werden. Eine Aufstellung der genauen Formate ist in Anhang A zu finden.

Da die Aufgaben der AAL von den meisten ATM-Karten durch eine spezielle Hardware übernommen werden und ein effizientes Monitoring nur durch den Einsatz dieser Hardware überhaupt möglich ist, ist es am wahrscheinlichsten, daß man vom Interface nur die reinen Nutzdaten, also den Payload-Anteil der PDUs erhalten kann.

Es ist zu beachten, daß PDUs unterschiedlichen Aufbaus an den beiden Teilschichten der ATM Anpassungsschicht, “Convergence Sublayer” (CS) und “Segmentation and Reassembly Sublayer” (SAR) existieren.

1.3.2 Classical IP over ATM

Unabhängig von den genauen Daten / Protokollebenen, die man bei ATM-Übertragungen aufzeichnen kann, wird mindestens der Payload-Anteil, d.h. der eigentliche Nutzdatenanteil, der ATM-Schichtenprotokolle zur Analyse gewonnen werden können.

Der logisch nächste Schritt ist also eine genauere Betrachtung der höheren Protokollebenen. Ein wichtiger Vertreter ist IP. Da eine Betrachtung oder sogar feinere Untersuchung den Rahmen dieser Arbeit sprengen würden und auch (vorerst) nur Aufzeichnungs-, aber keine Analysewerkzeuge entstehen sollen, wird hier nur ein Beispiel gewählt: Classical IP over ATM.

Es gibt (theoretisch) mehrere Möglichkeiten, IP-Pakete für den Transport über einen ATM-VC vorzubereiten (einzupacken). Die einzige, in der Praxis angewandte ist die sogenannte LLC[18] Encapsulation.

18. Logical Link Control

Mit derselben Methode der Einkapselung ist es auch möglich, verschiedene Protokolle über ein und denselben ATM-VC zu transportieren (MPOA, siehe auch [28]).

Durch einen speziellen Header (IEEE 802.2 LLC header) ist es möglich, eingepackte höhere Protokolle zu unterscheiden. Für die aktuelle Betrachtung sind folgende Payload-Formate von Interesse (ebenso sind Formate für andere Protokolle definiert):

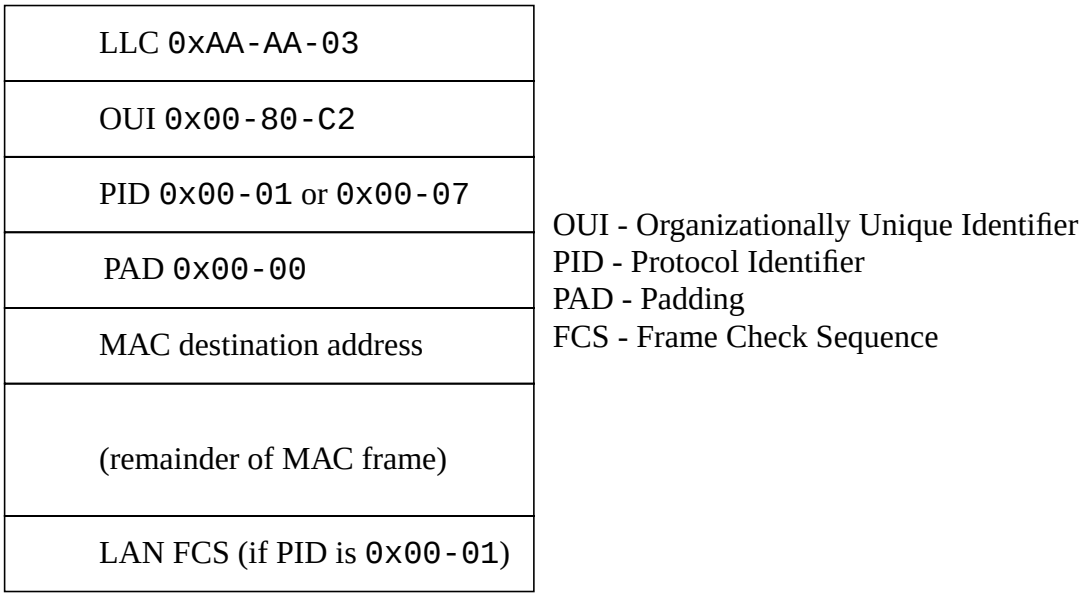


Abbildung 1.5 - Payload Format for Bridged Ethernet/802.3 PDUs [28]

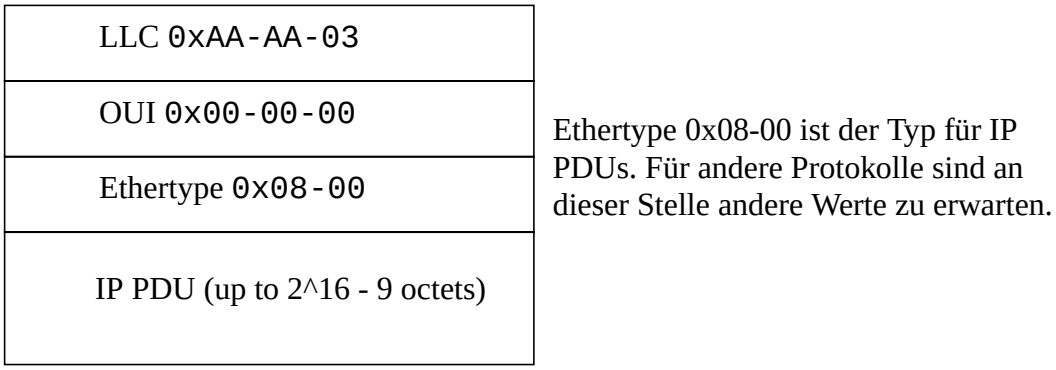


Abbildung 1.6 - Payload Format for Routed IP PDUs [28]

Für die weitere Analyse bzw. den Aufbau von Ethernet/802.3 PDUs bzw. IP-Paketen, wird das Studium weiterführender Literatur empfohlen. Die konkreten Definitionen für Classical IP over ATM sind in folgenden RFCs nachzuschlagen: RFC1483 [26], RFC1577 [27], und RFC1755 [28].

2 Einsatzumgebung für einen ATM-Logger

In Vorüberlegungen zu dieser Arbeit wurden mögliche Hard- und Softwareumgebungen für den zu erstellenden ATM-Logger diskutiert.

Die endgültige Entscheidung für den Einsatz von einer Sun ULTRA1 Workstation mit Sun-ATM-Karten und einem Pentium PC unter FreeBSD mit Fore-PCI-ATM-Karten und den HARP[19]-Treibern wurde aufgrund der Verfügbarkeit der Quellen von sowohl Betriebssystem als auch der Treiber gefällt.

2.1 Zugriff auf den Datenstrom

Bevor man über die Hardware der verwendeten ATM-Karten oder der Computer und ihrer Betriebssysteme nachdenkt, sollte man zuerst Informationen über die technischen Möglichkeiten des Zugriffs auf den Datenstrom ein einem zu überwachenden Netzwerk sammeln.

Für die physikalische Schicht von ATM-Netzwerken werden geht die Tendenz zu Glasfaserleitungen. Deshalb soll sich die vorliegende Arbeit auch auf dieses Medium beschränken. Normale Kupferleitungen[20] werden nur noch in lokalen Netzwerken - die vorhandene Netz-Infrastruktur nutzend - eingesetzt.

Die Lichtleiterkabel setzen sich aus vielen kleinen Einzelsträngen zusammen. Um Informationen aus einer Leitung abgreifen zu können, setzt man sogenannte optischen Leitungssplitter ein. Das Prinzip dieser Vorrichtungen ist, die Leitung in zwei Teile zu trennen.

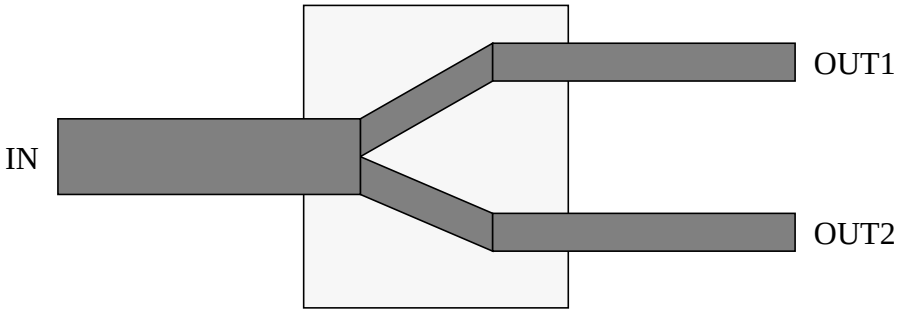


Abbildung 2.1 - Arbeitsweise eines optischen Leitungssplitters

Da an den beiden Ausgängen nur noch die Hälfte des als Übertragungsmedium benutzten Lichtes ankommt, d.h. die Lichtintensität halbiert wird, ist die Reichweite der Verbindungsleitungen durch den Einsatz der Leitungssplitter stark eingeschränkt.

19. Host ATM Research Platform
20. Twisted-Pair-Leitungen

Netzmonitore sollen hauptsächlich vor den Knoten für den Weitverkehrsanschluß angeschlossen werden, weshalb dieser Nachteil zu ignorieren ist.

Da es je eine Leitung für den Hin- und Rückkanal gibt, sind zwei Leitungssplitter einzusetzten und folglich auch zwei ATM-Karten (je eine für das Abhören einer Richtung). Das bringt ein neues Problem mit sich, das an dieser Stelle aber nur Erwähnung finden soll: Empfangene Daten sind zeitlich korrekt einzuordnen, d.h. es ist für eine zeitliche Konsistenz aufgezeichneter Daten zu sorgen.

2.2 Hardware der Analysestationen

In diesem Kapitel soll ein grober Vergleich der eingesetzten Hardware durchgeführt werden. Da es aber hauptsächlich auch auf die Effizienz des eingesetzten Betriebssystems (Kopieren großer Datenmengen quer durch den Speicher) und der Treiber für die zur Verfügung stehenden ATM-Karten ankommt, sind die genauen Leistungsdaten nicht von so großer Bedeutung.

2.2.1 Sun ULTRA1/170 und Sun SPARC5 mit SunATM-SBus-Karten

Geplant war eigentlich nur der Einsatz der ULTRA1, aber für umfangreichere Tests und zu Zwecken der Eingrenzung gesammelter Daten auf Betriebssystem bzw. Rechenleistung stand noch eine SPARC5 zur Verfügung.

Ausstattung der ULTRA1:

- Prozessor: UltraSparc, 167MHz
- Arbeitsspeicher: 128MB
- Festplattenspeicher: 2GB SCSI
- Bussystem: SBus
- ATM-Interfaces: 2 x SunATM, 155MBit/s, Fiber

Ausstattung der SPARC5:

- Prozessor: MicroSparc
- Arbeitsspeicher: 48MB
- Festplattenspeicher: 2GB SCSI
- Bussystem: SBus
- o.g. ATM-Interfaces

Der in den Sun-Rechnern eingesetzte Datenbus (SBus) ist mit 25 MHz getaktet. Bei einer Busbreite von 32 Bit (es gibt den SBus auch in einer 64 Bit-Version) ergibt das eine maximale Datentransferrate von rechnerisch 100 MByte/s. Ein realistischer Wert für den Betrieb ist etwa die Hälfte der maximalen Leistung, also 50 MByte/s.

Wie bei Netzwerkkarten dieser Leitungsklasse üblich, wird auch vom SunATM-Interface die meiste Arbeit, das Handling der ATM-Anpassungsschicht, vollständig per Hardware (durch einen speziellen ASIC[21]) durchgeführt. Es gibt für den Anwender keine Möglichkeit, zu sagen: “Ich will alles, was auf der Leitung vorbeikommt!”, d.h. eine Art Promiscous Mode. Das ist für Anwendungen, wie das Monitoring von ATM-Netzen ein Nachteil.

Für Geschwindigkeitsklassen von 155 MBit/s und höher (die SunATM-Karte wird auch als 622 MBit/s-Interface angeboten) ist es aber sicher sinnvoll, die CPU maximal zu entlasten.

In der Vergangenheit war z.B. bei X.25-Interfaces (ISO-Ebene 3) für Cisco-Router ein umgekehrtes Verhalten zu beobachten. Seit ca. 1985 gab es Spezial-Chips, die die untere Protokollschicht (LAPB[22]) per Hardware konnten. Bei Cisco wurden diese aber erst seit 1992 eingebaut (vor allem in Low-End-Routern), aber niemals voll eingesetzt. Alle Funktionen wurden weiterhin durch die CPU durchgeführt, obwohl eine Geschwindigkeitssteigerung um ca. Faktor 2-4 hätte erreicht werden können.

Für Geräte, die ein Protokoll für 2MBit/s-Übertragungen per CPU handeln sollen, ist dies gerade noch machbar, aber für Geschwindigkeiten jenseits der 100MBit/s unmöglich.

2.2.2 Pentium i586-PC und Fore-PCI-ATM-Karten

Ausstattung des i586-PC:

- Prozessor: Pentium, 133MHz
- Arbeitsspeicher: 128MB
- Festplattenspeicher: 2GB SCSI
- Bussystem: PCI
- ATM-Interfaces: 2 x Forerunner PCA-200E, 155MBit/s, Fiber

Da für die geplante Anwendung weniger die Rechengeschwindigkeit des Rechners, als mehr die Leistungsfähigkeit des Bussystems und die Geschwindigkeit von Kopieraktionen im Speicher und auf die Festplatte von Bedeutung ist, soll hier nur kurz die Leistung des PCI-Bussystems erwähnt werden.

21. Application Specific Integrated Circuit
22. LAPB wurde als Übertragungsprotokoll der Datensicherungsschicht auf seriellen X.25-Leitungen benutzt

Bustyp / Busbreite	Übertragungsleistung	Übertragungsleistung im Burst-Mode
ISA (8/16 Bit)	5 MByte/s	-
VLB (32 Bit)	50 MByte/s	80 MByte/s (nur lesen)
EISA (32 Bit)	33 MByte/s	-
PCI (32 Bit)	66 MByte/s	133 MByte/s
PCI (64 Bit)	133 MByte/s	264 MByte/s

Tabelle 2.1 - PCI-Bus im Vergleich zu älteren PC-Bussystemen [9]

Wie man der Tabelle entnehmen kann, kommt erst der PCI-Bus für Anwendungen wie ATM-Netzwerke in Frage, da zwar auch andere Systeme eine Leistung von ca. 20MByte/s erbringen können, dann aber der Bus kaum für andere Aktivitäten frei ist.

Die Fore-ATM-Karte ist durch das Coprozessor-Design hervorragend für die Verarbeitung von Daten bei Geschwindigkeiten von 155MBit/s geeignet. Der Intel i960-Coprozessor kann einen Großteil der Kommunikationsverarbeitung übernehmen und so den Hauptprozessor des Rechners entlasten. Zusätzlich sind noch spezielle ASICs als Hilfsprozessoren integriert. Neue Softwarereleases sind durch die downladbare Firmware kein Problem.

Für die Fore-ATM-Karte gelten die gleichen Einschränkungen, wie sie bei der SunATM-Karte besprochen wurden, d.h. daß alle Funktionen der AAL schon über die Hardware gelöst werden und keine Möglichkeit besteht, auf vollständige AAL-PDUs oder gar ATM-Zellen zuzugreifen.

Zusätzlich ist hier noch der Bereich anwählbarer VCs stark eingeschränkt. Für den VPI kann nur der Wert 0 und für den VCI nur Werte von 1 bis 1023 gewählt werden.

2.3 Betriebssystem / Treiber

Wie schon erläutert, sollten als Betriebssysteme für die zu untersuchenden Hardwareplattformen Solaris 2.6 für die Sun ULTRA1 und FreeBSD 2.2.5 für den PC zum Einsatz kommen. Beide Systeme sind zwar “UNIX”-Varianten, aber in Aufbau und Verhaltenen sehr unterschiedlich.

2.3.1 FreeBSD 2.2.5 / HARP 2.1

FreeBSD ist ein frei erhältliches UNIX-Betriebssystem für Intel-Plattformen. Ursprünglich war es 4.3BSD basiert, aber aufgrund von Lizenzproblemen wurde seit der Version 2.0 4.4BSDlite als Basis eingesetzt, welches vollkommen frei von Lizenzbindungen kommerzieller Unternehmen ist. Aktuell zum Zeitpunkt der Darstellung ist zwar FreeBSD schon in der Version 2.2.6 erhältlich, die aber keine wesentlichen Unterschiede zur eingesetzten Version 2.2.5 aufweist.

Wie für BSD-Systeme typisch, kommen auch bei FreeBSD Socket-basierte Treiber für Netzwerkschnittstellen zum Einsatz.

Sockets (siehe z.B. [3] oder [23]), die 1981 im 4.1BSD eingeführt wurden, bieten ein Programmierinterface sowohl für IPC[23] als auch für Netzwerkkommunikationen. Ein Socket ist ein Kommunikationsendpunkt und stellt ein abstraktes Objekt dar, welches Nachrichten empfangen und senden kann.

Durch das allgemeingültige Programmierinterface können Anwendungen auf sehr verschiedene Dienste ohne Änderung am Programmcode angepaßt werden. Z.B. können über einen Socket Prozesse auf die gleiche Art miteinander kommunizieren, ob sie nun auf einer Rechenanlage nebenläufig arbeiten, oder über ein Kommunikationsnetzwerk auf verschiedenen Netzknoten. Im Gegensatz zu den SystemV-Streams (siehe Kapitel 2.3.2) sind Änderungen am Verhalten eines Sockets nur durch die Änderung der kompletten Software im Kernel möglich.

Die Kommunikationssteuerung und den Datentransfer von und zur Fore-ATM-Karte übernimmt ein von der MSCI-ANG[24] entwickelter Treiber, der für Forschung und Lehre ohne Einschränkungen zur Verfügung steht.

23. Inter Process Communication
24. Minnesota Supercomputer Center, Inc. Advanced Networking Group

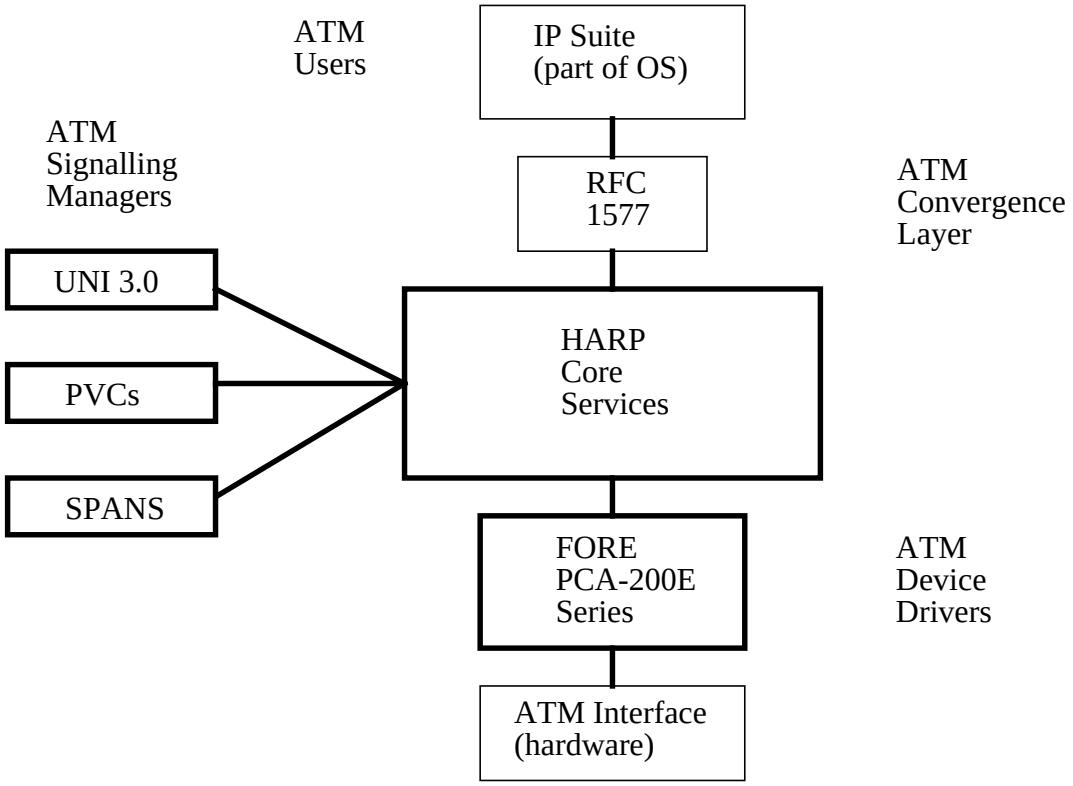


Abbildung 2.2 - Struktur des HARP-Systems [40]

In der Abbildung ist die Struktur der HARP-Komponenten zu sehen. Die Aufgabe des ATM Device Drivers übernimmt jeweils ein an die Adapter-Hardware angepaßtes Modul.

In der vorliegenden Version 2.1 unterstützt der Treiber “Classical IP over ATM” unter Verwendung der Protokolldefinition UNI 3.0. Auf den ersten Blick ist der Treiber sehr strukturiert und übersichtlicht aufgebaut. Eine normale IP-Verbindung über die Fore-ATM-Karte ließ sich problemlos nach wenigen Minuten Konfigurationsarbeit aufbauen.

HARP gibt es außer für FreeBSD / Fore-PCI-Adapter auch noch für FreeBSD / ENI[25]-PCI-Adapter und SunOS 4.x / Fore-SBus-Adapter.

2.3.2 Solaris 2.6 / SunATM 2.1

Solaris ist der, jetzt SystemV R4.2 basierte, Nachfolger des populären SunOS der Firma Sun Microsystems, Inc.. Das Solaris-Betriebssystem ist ein sehr stabiles und performantes (zumindest auf den neuen Sun ULTRA Architekturen, für welche es speziell angepaßt wurde) System. Besonderer Wert wurde bei der Entwicklung auf die Qualität und Effizienz der Streams-Implementierung gelegt.

25.Effizient Networks, Inc.

Diese Streams-Umgebung ist aufgrund der u.U. sehr vielen Kopieraktionen zwischen den einzelnen Modulen bei vergleichbaren Systemen sehr schlecht für große Datenübertragungen geeignet. In den letzten Versionen von Solaris wurden aber die meisten Kopieraktionen eingespart, indem nicht mehr Speicherblöcke weitergereicht werden (durch Anlegen einer Kopie), sondern nur Zeiger auf den ursprünglichen Block. Dies erhöhte natürliche die Komplexität der Implementierung (wo wird der Speicher belegt, wo freigegeben, ab welcher Größe von Datensegmenten lohnt sich dieser Aufwand, etc.), brachte aber einen großen Vorteil in Punkto Performance. Natürlich wurden und werden auch noch andere Konzepte zur Verminderung von Kopieraktionen eingesetzt, wie z.B. das (sehr systemnahe) Copy-On-Write, aber das soll hier nicht näher erläutert werden (siehe z.B. [24]).

Es geht die ganze Zeit um Streams und Streams-Modulen. Was ist das eigentlich und wie funktioniert das?

Ein Stream ist eine Vollduplexverbindung[26] zwischen einem Treiber im Betriebssystemkern und einem Benutzerprozeß. Ein Stream kennzeichnet sich durch seine Unterteilung in Stream-Head, der Verbindung zum Benutzerprozeß, den Stream-Modulen, diese bearbeiten die Daten auf bestimmte Art und Weise und dem Stream-Tail oder auch Stream-End, welcher die Interfacebehandlung übernimmt (siehe folgende Abbildung).

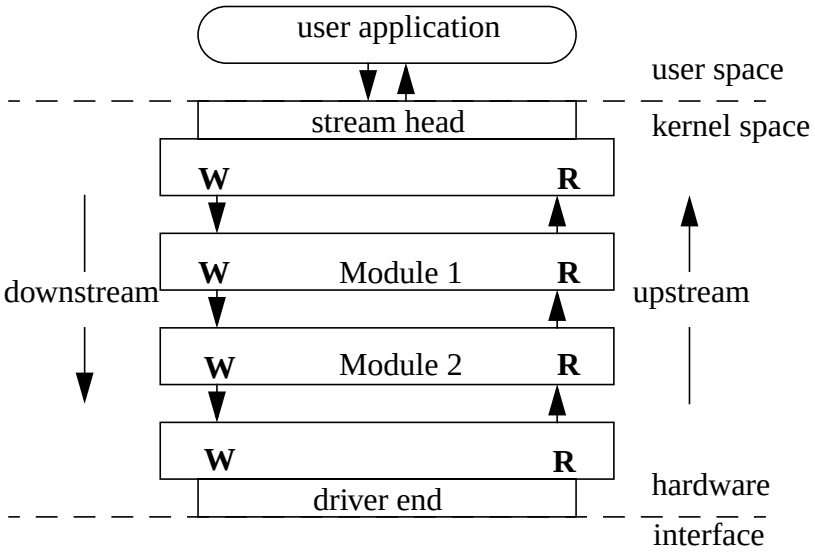


Abbildung 2.3 - Ein typischer Stream [3]

26. Vollduplex = bidirektionaler Datenverkehr

Streammodule können verschiedene Aufgaben übernehmen. Z.B. Verschlüsselungen, protokollkonformes Paketieren / Depaketieren, etc.. Durch spezielle Systemcalls können Module zu einem Stream hinzugefügt bzw. aus diesem entfernt werden. Dieser Aufbau erleichtert die Wartung der Funktionalität erheblich, da für Änderungen am Verhalten eines Streams nur ein Modul hinzugefügt bzw. ausgetauscht werden muß.

Wie in Abbildung 2.3 zu sehen ist, müssen natürlich die empfangenen Daten (upstream) bzw. die zu sendenden Daten (downstream) alle Module passieren. Der einfachste Weg, einen Datenblock zum Weiterreichen immer zu kopieren, ist nicht sehr effizient. Dies gilt auch, wenn mit Methoden wie Copy-On-Write gearbeitet wird, da die meisten Module die Daten tatsächlich verändern. Weitere Informationen zum Thema Streams für verschiedene UNIX-Versionen und vor allem Details über die Implementation und die Benutzung sind in [3] zu finden.

Bei Solaris 2.6 (zumindest in der Netzwerkumgebung) wird folgender Weg begangen: Kleine Datenpakete werden immer kopiert, da hierbei der Verwaltungsaufwand größer wäre, als der Performanceverlust durch das Kopieren. Große Datenblöcke hingegen werden durch Weiterreichen eines Zeigers auf den Block von Modul zu Modul übergeben. Dabei wird auch ein Zeiger auf eine Funktion, die den Speicher am Ende der Aktion freigeben soll, mitgegeben.

Für die Steuerung der SunATM-Karte kam der SunATM-Treiber in Version 2.1 zum Einsatz, welche z.B. auch für die Implementierung eines Delay-Meßprogramms[27] benutzt wurde.

Sun stellt ein sehr umfangreiches API[28] zur Verfügung, welches aber für die Erstellung eines Monitoring-Werkzeugs wenig bis gar nicht zum Tragen kommt, da dieses API hauptsächlich für die Implementierung von höheren Protokollen (ISO-Ebene 6/7) und eigentlichen Anwendungen gedacht ist und einen dementsprechenden Funktionsumfang hat.

Die wichtigsten Steuerungsaufgaben können über Systemaufrufe durchgeführt werden und der Datentransport von und zur Karte über die normalen Lese- und Schreiboperationen auf Streams.

27. tmt4atm - Transmit delay Measurement Tool for ATM networks [20]
28. Application Programming Interface

Die folgende Abbildung zeigt die interne Struktur des Treibers von Sun:

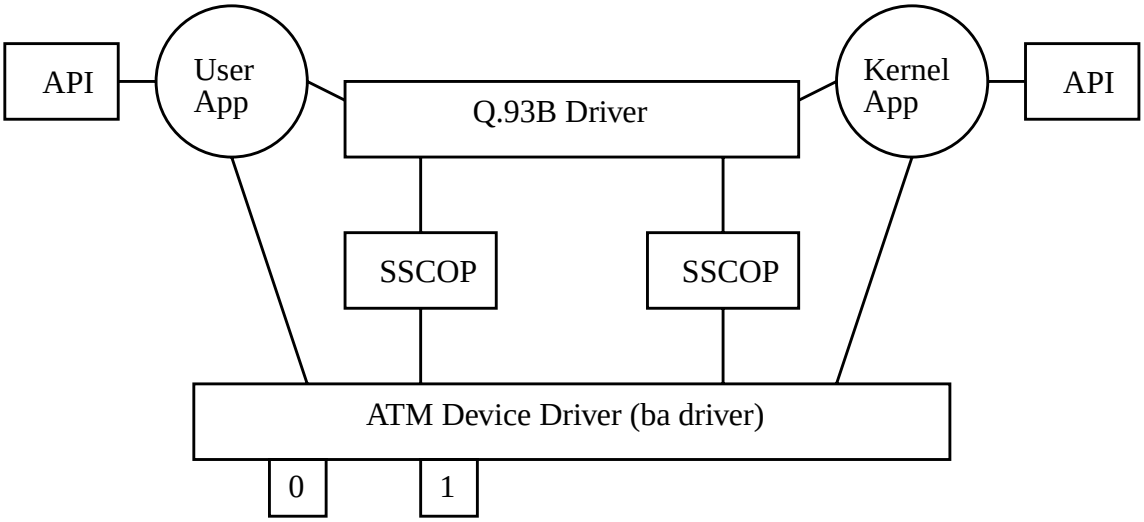


Abbildung 2.4 - Treiberkonfiguration des API [39]

2.4 Performancetests mit Standardumgebungen

Nach dem Einbau der ATM-Karten und deren Konfiguration sind einige erste (sehr einfache) Performancetests durchgeführt worden. Die ersten wurden mit vorhandener Software durchgeführt und setzten darum auf höhere Protokollebenen auf (TCP/IP). Für Solaris und die Sun-ATM-Karte wurde ein weiterer Test mit einem kleinen selbstgeschriebenen Tool durchgeführt.

Die Tests wurden auf allen drei zur Verfügung stehenden Maschinen durchgeführt, um präzisere Erkenntnisse über die Unterschiede der eingesetzten Betriebssysteme und die notwendige CPU-Leistung zu erhalten.

Die Ergebnisse sollen im folgenden dargestellt und im Anschluß kurz interpretiert werden. Zu beachten ist, daß für einen VC technisch bedingt (eingeschränkt durch die eingesetzte ATM-Hardware) nur max. 134 Mbit/s von den 155 Mbit/s reserviert werden können, für einen IP-VC sogar nur 133 Mbit/s.

2.4.1 Classical IP over ATM (“ftp”)

Der erste Test wurde mit dem Standard-”ftp” von UNIX durchgeführt. Es wurden immer 200MB große Datenblöcke übertragen, um alle Cache-Mechanismen der eingesetzten Systeme außer Kraft zu setzen. Gelesen wurden die Daten von der lokalen Festplatte und nach der Übertragung direkt verworfen (auf `/dev/null` geschrieben).

Quelle	Ziel	Übertragungsrate	
		MByte/s	MBit/s
SPARC5	ULTRA1	4	32
ULTRA1	SPARC5	8	64
SPARC5	i586-PC	3,8	30,4
i586-PC	SPARC5	6	48
ULTRA1	i586-PC	8,6	68,8
i586-PC	ULTRA1	8	64

Tabelle 2.2 - Übertragungsraten mittels “ftp”

Während der Übertragung war die CPU der SPARC5 maximal ausgelastet. Die ULTRA1 und der i586-PC jedoch nur zu einem geringen Prozentsatz. Statt dessen war die Übertragungskapazität der Festplatte hier das Nadelöhr. Dies ist aufgrund der Tatsache, daß für die geplante Anwendung für das Monitoring von ATM-Netzen die gewonnenen Daten und Informationen auch auf einem Hintergrundspeicher - der Festplatte - aufgezeichnet werden sollen, von großem Interesse.

2.4.2 Classical IP over ATM (“push”)

Ein weiterer Test wurde mit “push”[29] durchgeführt. Die zu übertragenden Daten werden dabei nicht, wie im vorhergegangenen Versuch, von Festplatte gelesen, sondern speziell für die Transmission erzeugt und danach verworfen.

Das komplette Meßprotokoll und aus den Meßdaten erzeugte Diagramme wurden aufgrund der Größe in Anhang C aufgenommen.

Die für die wichtigsten Grafiken sind hier kurz zusammengestellt.

29. vgl. Anhang B.1

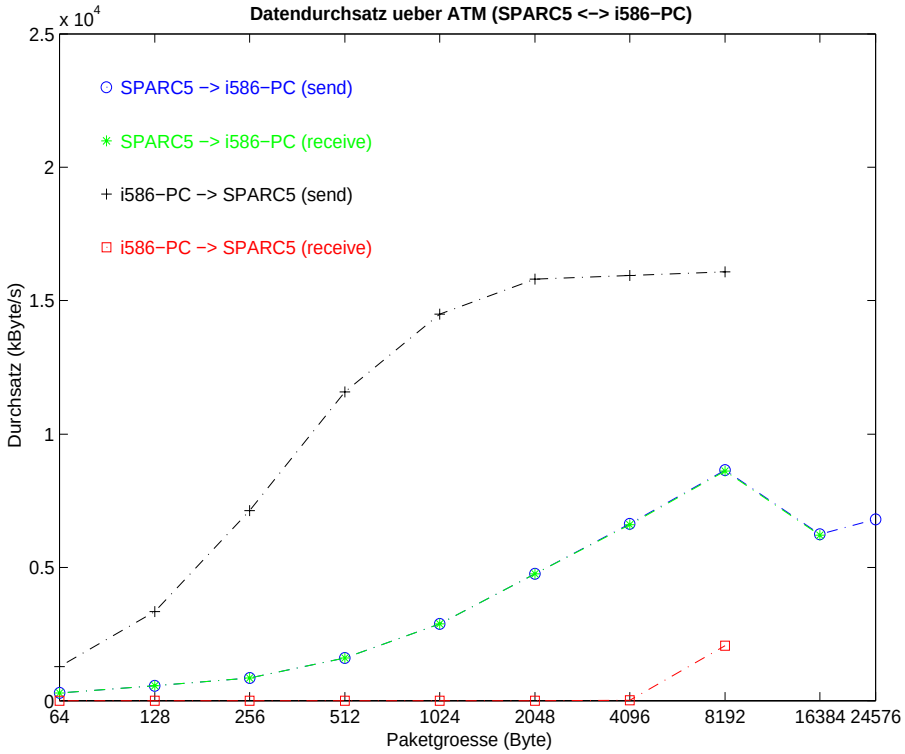


Abbildung 2.5 - Datendurchsatz über ATM zwischen SPARC5 und i586-PC

Offensichtlich schafft es die SPARC5 nicht einmal, einen UDP-Datenstrom mit maximalem Durchsatz zu erzeugen (siehe Abbildung 2.5). Im Gegenteil, soll diese Maschine große Datenmengen entgegennehmen, wird sie von den Daten “überschwemmt”, so daß sie keinerlei Reaktion mehr zeigt.

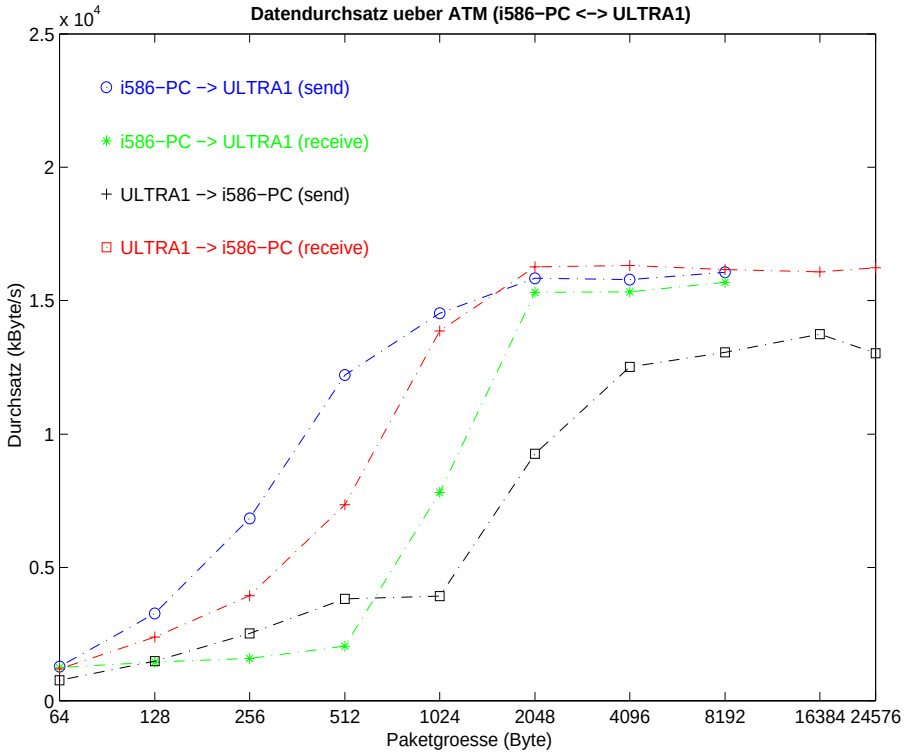


Abbildung 2.6 - Datendurchsatz über ATM zwischen i586-PC und ULTRA1

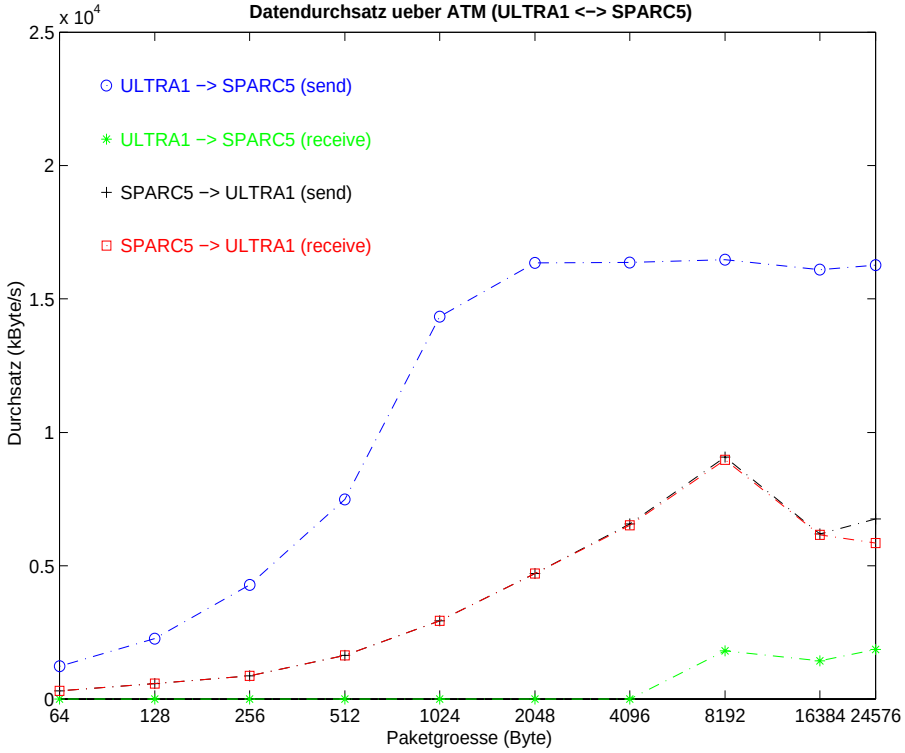


Abbildung 2.7 - Datendurchsatz über ATM zwischen ULTRA1 und SPARC5

Wie den Abbildungen zu entnehmen ist, nutzen die ULTRA1 und der i586-PC sendeseitig fast die maximal mögliche Bandbreite von 134MBit/s. Auch beim Empfangen der UDP-Pakete schneiden sie (zumindest bei größeren Paketen) sehr gut ab.

2.4.3 ATM-AAL5 (“atmpush”)

Für diesen Test wurde das Tool “atmpush”[30] benutzt, welches AAL5-PDUs variabler Länge direkt über den Streams-Treiber der ATM-Karte versendet.

Bei sehr kleinen Datenpaketen (bis 40 Byte passen diese kleinen AAL5-PDUs in eine einzelne ATM-Zelle) ist dies eine für das Betriebssystem sehr aufwendige Aufgabe, da sehr viele, sehr kleine Datenblöcke in kürzester Zeit quer durch den Speicher transportiert werden müssen. Aber für eine Anwendung, die möglichst alle ankommenden ATM-Zellen verarbeiten soll, ist das eine Art Maximaltest oder ein Test des “worst case”. In einer realistischen Umgebung kommen so kurze Pakete, welche in einzelne ATM-Zellen[31] passen, nur sehr selten vor.

Große Datenpakete hingegen lassen sich sehr effizient durch das Betriebssystem verwalten, da für die gleiche Datenmenge wesentlich weniger Systemaufrufe für die Verarbeitung nötig sind.

Das vollständige Meßprotokoll ist in Anhang D zu finden.

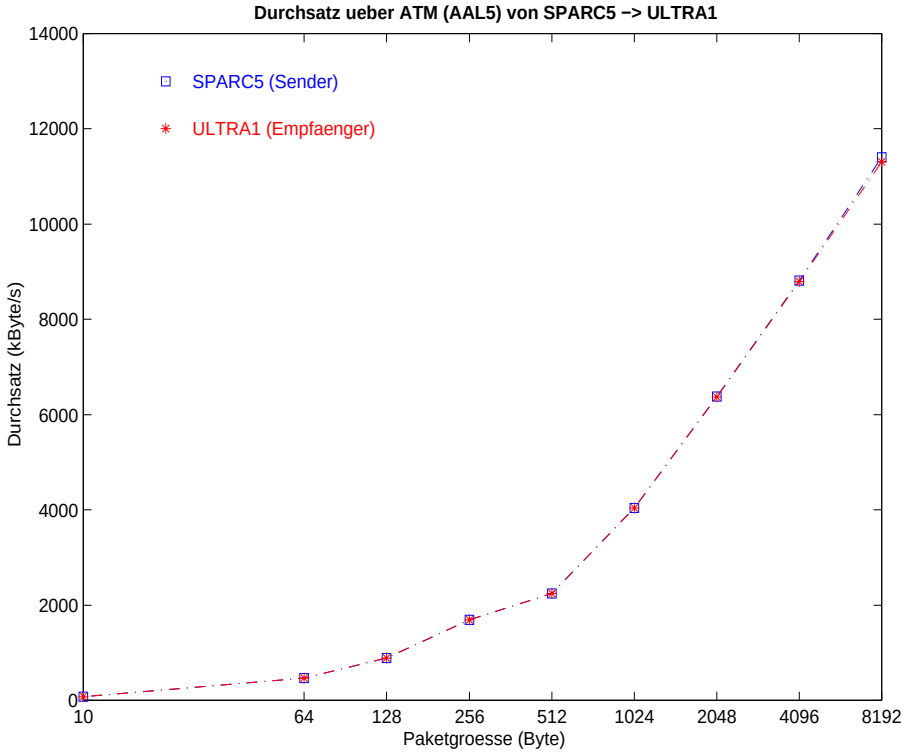


Abbildung 2.8 - Datendurchsatz über ATM (AAL5) von SPARC5 zu ULTRA1

30. vgl. Anhang B.2
31. eine ATM-Zelle kann maximal 48 Byte Nutzdaten transportieren

Wie in Abbildung 2.8 zu sehen ist, kann die SPARC5 nur bei sehr großen Paketen (>8000 Byte) die zur Verfügung stehende Netzkapazität von (netto) 134 MBit/s annähernd ausnutzen. Die ULTRA1 hat keine Probleme, die verschickten AAL5-PDUs wieder zu empfangen.

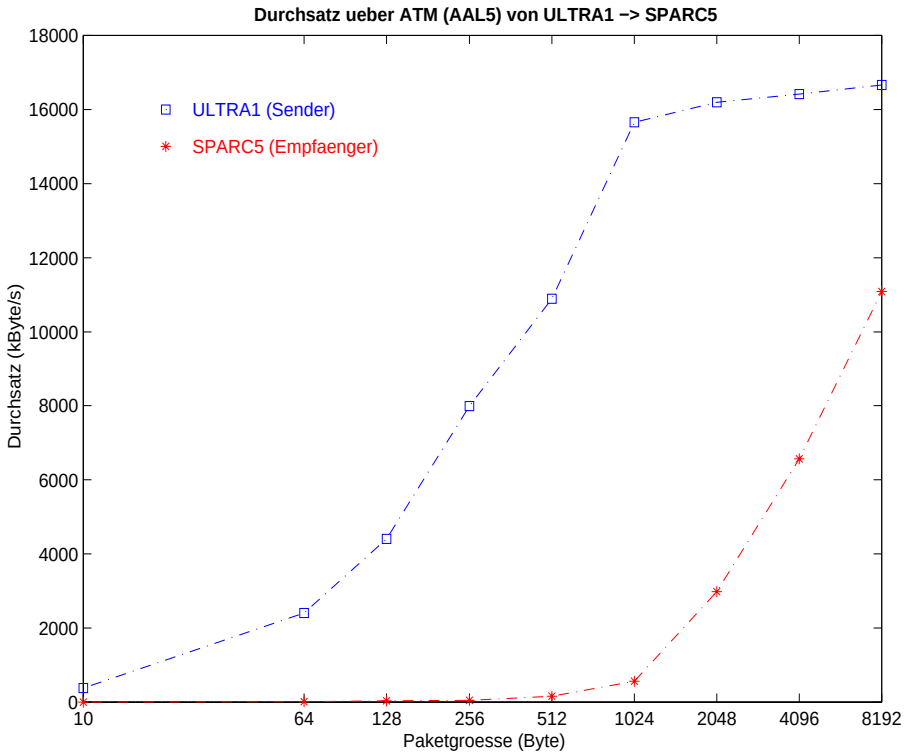


Abbildung 2.9 - Datendurchsatz über ATM (AAL5) von ULTRA1 zu SPARC5

Die ULTRA1 kann im Gegensatz zur SPARC5 die Kapazität des ATM-Netzes wesentlich besser ausnutzen. Leider schafft es die SPARC5 nicht annähernd, den Datenstrom wieder zu empfangen.

2.4.4 Erkenntnisse

Die CPU-Leistung einer Sun SPARC Station 5 ist offensichtlich für keinerlei effiziente Nutzung eines ATM-Netzes ausreichend. Schon bei einfachen Tests ist die Workstation überfordert.

Die Sun ULTRA1 und ein Pentium-i586-PC unter FreeBSD sind in Punkto Leistung der ATM-Schnittstelle und Effizienz der Treiberprogrammierung vergleichbar. Die ULTRA1 kann durch die Streams-Schnittstelle des Solaris-Betriebssystems sehr kleine und besonders große Datenpakete schneller und mit weniger CPU-Aufwand transportieren, als es die Socket-Schnittstelle des BSD-Derivates erlaubt. Diese ist dafür bei mittleren Paketgrößen ein klein wenig besser.

Zusammenfassend kann man sagen, daß generell keine Möglichkeit besteht, sehr kleine Datenblöcke (wie z.B. ATM-Zellen) schnell genug durch das Betriebssystem zu schleusen, um eine Aufzeichnung dieser Daten zu ermöglichen.

Da aber die ATM-Karten die Aufgaben der ATM-Anpassungsschicht per Hardware übernehmen, ist die Bewertung nicht mehr so einfach.

Beide Maschinen (die ULTRA1 und der i586-PC) scheinen sich für die Aufgabenstellung, eine Aufzeichnung und Auswertung des Netzverkehrs über ATM-Netze, zu eignen. In einer beispielhaften Implementierung ist diese Aussage aber noch zu erhärten.

Für diese Implementierung zu beachtende Punkte bzw. Ideen sind:

- Anzahl der Kopieraktionen im Hauptspeicher (natürlich minimal, da dies maximale CPU-Last verursacht)
- Menge der auf einen Hintergrundspeicher zu kopierenden Daten (evtl. schon bei der Aufzeichnung aussondern)

3 Entwurfsüberlegungen für den Aufbau eines ATM-Monitors

Die Aufgabe einer ATM-Moni[32], ist in drei wesentliche Teilbereiche zu untergliedern: das “Aufzeichnen” der Daten vom Netz, das “Transportieren” dieser Informationen (evtl. incl. einer Verarbeitung) und dem “Analysieren” der Daten, um so z.B. Statistiken über den Netzverkehr erstellen zu können.

Der wichtigste Punkt für eine solche Anwendung ist die Datenerfassung. Das Aufzeichnen (Kollektieren) von Netzwerkdaten ist eine stark hardwareabhänige Aktion. Es sind auch Fragen der Interfacebehandlung zu berücksichtigen. Das Transportieren und Analysieren hingegen sind relativ Hardware- und Betriebssystemunabhängig lösbar.

In diesem Kapitel sollen die grundsätzlichen Problemstellungen und Lösungsansätze für den Aufbau einer ATM-Moni erarbeitet werden, ohne allerdings Details bestimmter Betriebssysteme oder Hardwarelösungen zu beachten.

Wichtigste Zielstellung soll trotz des Notwendigkeit eines sehr hardwarenahen Aufzeichnungsteils eine maximale Portabilität der erstellten Software sein. Diese kann z.B. durch eine Mehrstufenlösung (bzw. Mehrschichten-) erreicht werden.

3.1 Grundaufbau einer ATM-Moni

Wie schon erwähnt, läßt sich die Funktionalität einer ATM-Moni in drei wesentliche Bereiche untergliedern:

Abbildung 3.1 - Grundaufbau einer ATM-Moni

Diese Teile werden immer in einer Implementierung zu unterscheiden sein, unabhängig wie sie im einzelnen realisiert werden.

32. Dieser Terminus soll ab sofort als Bezeichnung der Gesamtheit einer Anwendung gelten, die Daten vom ATM-Netz aufzeichnet, vorverarbeitet und analysiert und die Ergebnisse in weiterverarbeitbarer Form abspeichert, d.h. eines ATM-Monitors.

3.1.1 Aufzeichnungsteil

Die komplette Behandlung des Netzwerkinterfaces ist in diesen sehr hardwarenahen Teil zu integrieren. Hier sind alle Aufgaben der Steuerung der Hardware und der Kollektion der Daten vom Netz zu finden. Wahrscheinlich werden diese Aufgaben nur im UNIX-Kern und nicht von einem Benutzerprozeß lösbar sein.

3.1.2 Transport- und Verarbeitungsteil

Sich in beide Richtungen mit den angeschlossenen Modulen koordinierend nimmt dieser Teil eine zentrale Stellung ein. Er sollte schon relativ hardwareunabhängig implementiert werden, steuert aber trotzdem auch die Hardwareschnittstelle. Zu beachten ist, daß Daten mit extremen Raten transportiert werden müssen (155MBit/s und mehr).

3.1.3 Analyseteil

Der aus Anwendersicht wichtigste Teil ist die Analyse der gesammelten Daten. Dies ist gleichzeitig der hardwareunabhängigste Teil einer Implementierung einer ATM-Moni. Erweiterungen, die neue (oder durch die ATM-Moni noch nicht realisierte) Protokolle analysieren sollen, sind komplett in diesen Teil zu integrieren.

3.2 Dekomposition in ein Prozeßsystem

Um eine möglichste flexible und portable Lösung zu erhalten, empfiehlt es sich, die in Kapitel 3.1 “Grundaufbau einer ATM-Moni” erarbeitete Grundstruktur als Mehrprozeßsystem zu entwerfen.

Im folgenden sollen die wichtigsten Gesichtspunkte herausgestellt werden, die bei der Aufteilung der Prozesse eine Rolle spielen.

3.2.1 Kernel- vs. Userlevelimplementation

Das Aufzeichnen von Netzverkehr bei Transfergeschwindigkeiten von 155MBit/s und mehr ist eine sehr zeitkritische und hardwarenahe Problematik. Der erste Gedanke im Vorfeld einer Implementierung ist also: “Das muß im Kernel gemacht werden.”.

Dem widerspricht aber die Zielstellung einer maximalen Portabilität. Außerdem soll in dieser Arbeit die Möglichkeit untersucht werden, Monitoring-Aufgaben durch Standard-Arbeitsplatzrechner vornehmen zu lassen. Wird der UNIX-Kernel so extrem manipuliert, daß die volle Programmfunktionalität an dieser Stelle implementiert wird, so ist ein weiteres Arbeiten mit dem Computer auf Benutzerebene nicht mehr möglich (wahrscheinlich wird sogar der komplette Betrieb des Rechners unmöglich).

Diese Gedanken führen unweigerlich zu einer Mehrstufenlösung, bei der nur die unterste, besonders hardwareorientierte Schicht in den Kernel eingegliedert wird.

Eine erste Unterteilung des Prozeßsystems wäre also:

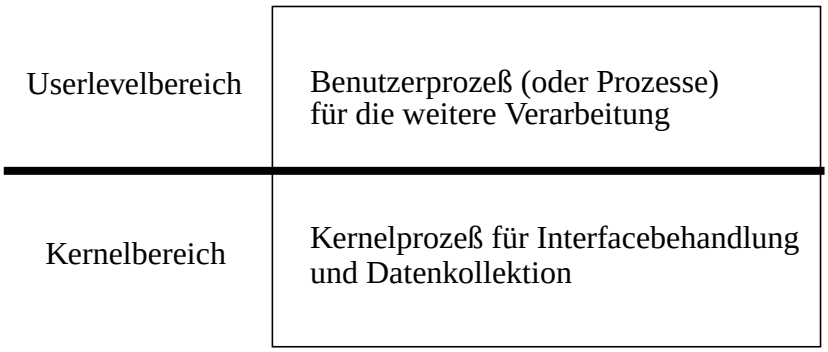


Abbildung 3.2 - Aufteilung in Userlevel- und Kernelprozeß

Die logische Folge ist ein Mehrprozeßsystem (Kernel- und Userlevelprozeß) mit allen Vor- und Nachteilen, wie z.B. gute Portabilität, aber auch Koordinierungs- und Datentransferprobleme. Wie diese Probleme gelöst werden, soll erst an späterer Stelle erläutert werden.

3.2.2 Ein- bzw. Mehrprozeßlösung auf Benutzerebene

Nach der Frage “Wie bekomme ich die Daten vom Interface?” kommt man an die nächste Entscheidungsschranke: “Ein Prozeß für alles, oder eine Aufgabenaufteilung auf zwei oder mehr Prozesse?”. Konkret ist zu überlegen, ob sich der Transport- und Verarbeitungsteil mit dem Analyseteil in einem Benutzerprozeß vereinigen lassen (sollen).

Sowohl die Ein- als auch die Mehrprozeßlösung haben Vor- und Nachteile:

3.2.2.1 Einprozeßlösung für Transport, Verarbeitung und Analyse

Der wichtigste Vorteil einer Einprozeßlösung ist die einfache Implementierbarkeit. Es sind keine Koordinierungsprobleme zu erwarten, die zwischen mehreren Prozessen eine Rolle spielen, aber auch nicht die Flexibilität der anderen Lösung (s.u).

3.2.2.2 Mehrprozeßlösung für Transport, Verarbeitung und Analyse

Gerade Netzwerkverkehr ist kein kontinuierlicher Datenstrom, sondern einer mit (meist) relativ niedriger Grundlast und sehr hohen Spitzen. Diese “Bursts” verlangen von der aufzeichnenden (analysierenden) Maschine eine hohe CPU-Verfügbarkeit während dieser Spitzen. In einem einzelnen Prozeß lassen sich zwar Puffer einbauen, diese verlangen aber ab einer gewissen Anzahl ein hohes Maß an Verwaltungsaufwand. Mehrprozeßlösungen lassen sich flexibel koppeln und

so mit Hilfe von Eigenschaften des Betriebsystems soweit entzerren, daß das entstehende System auch lange und besonders hohe Spitzen in der Datenübertragung ohne Schaden (d.h. ohne Verluste) überstehen kann.

3.2.3 Spezifikation des Prozeßsystems der ATM-Moni

Den Betrachtungen der letzten Abschnitte folgend, läßt sich ein erweitertes Prozeßsystem für eine ATM-Moni erstellen:

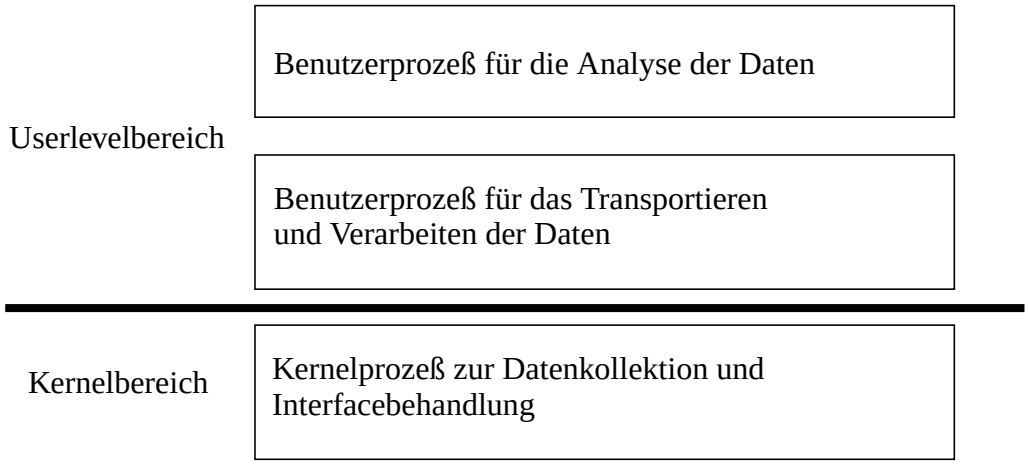


Abbildung 3.3 - Prozeßsystem der ATM-Moni

Das bedeutet aber nicht, daß es die ATM-Moni wirklich nur mit drei Prozessen implementiert werden muß! Ausgehend von der Tatsache, daß eine Implementierung in der Regel nur “schwach prozeßtreu” sein muß [45], d.h. daß man spezifizierte Prozesse bei der Implementierung in mehrere (Teil-)Prozesse zerlegen darf.

3.3 Problemstellungen

In den folgenden Abschnitten sollen - auf theoretischer Basis - Problemstellungen erläutert und analysiert werden, die sich aus den bisherigen Überlegungen ergeben. Dabei soll vor allem das erarbeitete Prozeßsystem betrachtet werden.

3.3.1 Koordinierung und Kommunikation im Mehrprozeßsystem

Ein Mehrprogramm- oder Mehrprozeßsystem muß sorgfältig koordiniert werden. Im vorliegenden Fall sind die wichtigsten Gesichtspunkte:

- Übergabe von Statusinformationen zwischen den Prozessen
- Steuerung des Kernelmoduls, welches für die Datenaufzeichnung verantwortlich ist
- Datentransfer (mit extremen Raten) zwischen den Prozessen

Diese Punkte lassen sich in zwei Kategorien aufteilen: “Koordinierung” und “Kommunikation” zwischen den Prozessen.

3.3.1.1 Warten oder Pollen

Eine wichtige Frage bei der Datenübergabe zwischen den Prozessen ist: “Warten oder Pollen?”. Dabei ist es egal, auf welchem Weg bzw. auf welche Art und Weise die Daten wirklich übergeben werden. Immer müssen erst Daten gesammelt und dann an die nächste Schicht des Programmsystems weitergereicht werden.

Um nicht sinnlos Rechenzeit zu verschwenden, ist die Alternative “Warten” sicher zu empfehlen. Werden Mehrprozessorrechner benutzt, ist diese Empfehlung nur noch bedingt korrekt, da dann weitere Prozessoren für die zusätzlichen Aufgaben zur Verfügung stehen.

Bedingt durch die hohen Datenraten und die Tatsache, daß dadurch wohl stets Daten anliegen werden, ist die Alternative “Pollen” trotz des eben genannten Nachteils nicht uninteressant.

Der beste Weg liegt sicherlich - wie fast immer - in der Mitte. Dies soll durch Testimplementierungen herausgefunden werden.

3.3.1.2 Kopieren aufgezeichneter Daten

Ein einschränkender Punkt für eine durch mehrere Instanzen strukturierte Lösung ist die sehr hohe Datenübertragungsgeschwindigkeit von mindestens 155MBit/s (die ATM-Definition läßt wesentlich höhere Geschwindigkeiten zu) zu beachten.

Das Resultat einer Mehrschichtenlösung sind meist sehr aufwendige Kopieraktionen im Speicher (oder sogar auf der Festplatte), die auf allen zu untersuchenden Systemen besonders Ressourcenintensiv sind. Hier sind Lösungsansätze zu suchen, die diesen Aufwand bei gleichzeitiger Strukturerhaltung minimieren.

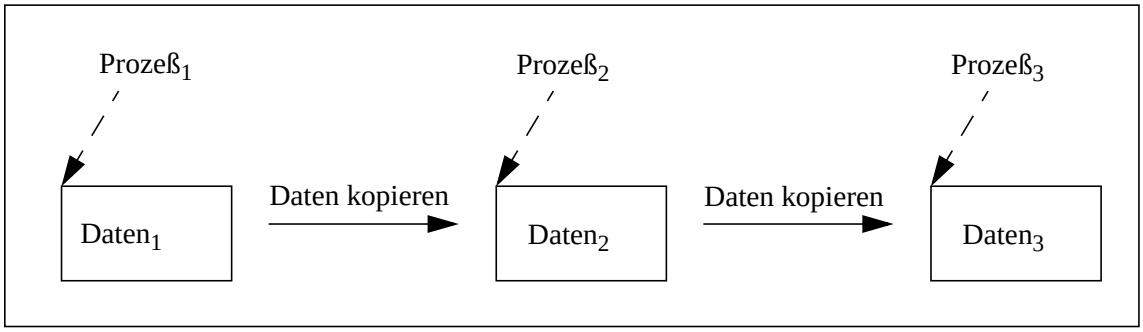


Abbildung 3.4 - Datenaustausch durch Kopieren

Dem Gedanken folgend, kommt man zu einer in den modernen UNIX-Betriebssystemen sehr häufig angewandten Methode des Datentransports über mehrere Module, der Datenübergabe durch Zeiger. Dabei werden die Daten nicht kopiert, sondern nur Zeiger auf die Position der Daten im Hauptspeicher übergeben. Das führt zwar zu einem weiteren Problem, wo und wie der Speicher für eine erneute Benutzung freigegeben wird. Dafür ist diese Methode aber sehr performant.

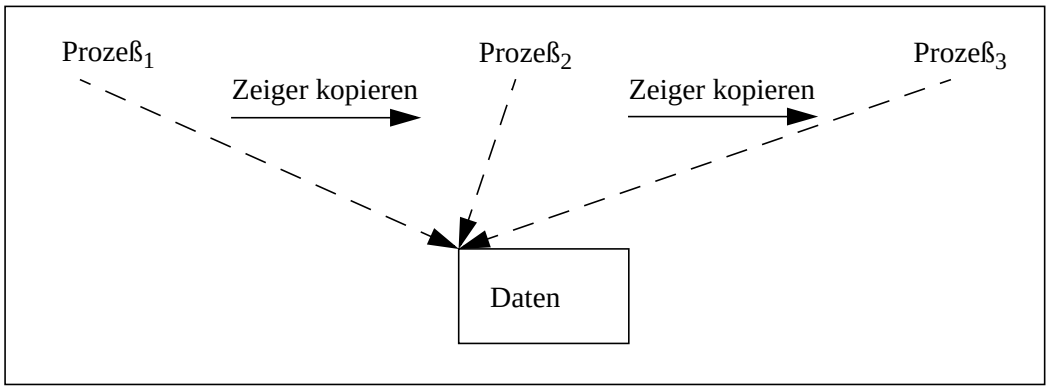


Abbildung 3.5 - Datenaustausch durch Zeigerübergabe

Da nun Teile des Programmsystems auf Kernelebene implementiert werden, ist noch ein weiterer Punkt zu beachten: Die Übergabe der Daten vom Kernel- in den Benutzeradreßraum.

Für die zwischen den Prozessen der ATM-Moni nötigen Kommunikationsvorgänge sind Lösungen zu suchen, die sich an den verschiedenen Verbindungspunkten stark unterscheiden werden. Die Schnittstelle “Kernelprozeß zum Aufzeichnen der Daten” <-> “Benutzerprozeß zum Transport und Datenverarbeitung” wird in Kapitel 4 “Vergleich struktureller Ansätze der Kommunikation zwischen Kernel- und Benutzerprozessen” und die zwischen den beiden Benutzerprozessen für “Transport und Verarbeitung” und “Analyse” der Daten in Kapitel 5 “Ankopplung von Benutzerprozessen zur Protokollanalyse” diskutiert werden.

3.3.2 Zeitliche Konsistenz der aufgezeichneten Daten

Um den gesamten ATM-Verkehr aufzuzeichnen, ist es nötig, zwei ATM-Interfacekarten (je eine für jede Richtung) einzusetzen (siehe Kapitel 2.1 auf Seite 13). Das führt zu einem weiteren Problem: Die zeitliche Konsistenz der aufgezeichneten Daten.

Eine einfache Vorgehensweise, diese Konsistenz zu erzeugen, ist, Sequenznummern für alle empfangenen Datenblöcke einzuführen. Die einzige Bedingung für diese ist, daß eine echt monoton steigende Folge von Zahlen generiert wird, die für jeden (Teil-)Prozeß auf dem Rechner gültig ist.

Insofern die eingesetzte Betriebssystemsoftware 64 Bit große Zahlen erlaubt, kann man das Problem von Integerüberläufen ignorieren.

Ausgehend von dem kleinsten möglichen Datenpaket mit einer Länge von einem Byte und einer maximalen Datenrate von 155 MBit/s, würden 64 Bit-Integer folglich weit über 28.000 Jahre reichen, ohne einen Überlauf befürchten zu müssen.

Sollten nur 32 Bit-Integer zur Verfügung stehen, so sind Mechanismen einzubauen, die einen Überlauf problemlos überstehen lassen. Z.B. könnte man 64 Bit-Integer künstlich per Software erzeugen.

4 Vergleich struktureller Ansätze der Kommunikation zwischen Kernel- und Benutzerprozessen

Dieses Kapitel soll die Möglichkeiten der Prozeßkoordinierung und -kommunikation zwischen Kernelprozessen und Benutzerprozessen unter verschiedenen UNIX-Systemen behandeln.

Diese Möglichkeiten sollen durch Testimplementationen des “Kernelprozesses zur Aufzeichnung von Netzwerkdaten” und des “Benutzerprozesses zum Datentransport und deren Weiterverarbeitung” der ATM-Moni konkretisiert und verifiziert werden. Diese Betrachtungen sind sehr stark Betriebssystemabhängig. Aus diesem Grund sollen sie unabhängig voneinander unter Solaris 2.6 und FreeBSD 2.2.5 durchgeführt werden.

4.1 Vorbetrachtungen für eine Implementation unter Solaris

Um die interessanteren Punkte der Prozeßkommunikation bearbeiten zu können, werden in einem einleitenden Abschnitt zuerst die Fragen der Datenkollektion besprochen. Erst danach werden mögliche Implementationen der in Kapitel 3 “Entwurfsüberlegungen für den Aufbau eines ATM-Monitors” beschriebenen Spezifikation des Prozeßsystems erarbeitet.

4.1.1 Datenkollektion

Der erste Schritt, ein fertiges Programm - in diesem Fall das ATM-Treiber-Modul “`ba`” von Sun - zu modifizieren, ist es, den Quellcode sorgfältig zu analysieren.

Im wesentlichen ist der Code in zwei C-Dateien aufgeteilt: `atm_common.c` und `ba_drv.c`.

In `atm_common.c` sind alle für die allgemeingültige Initialisierung notwendigen sowie die für die Verarbeitung von Systemaufrufen benötigten Prozeduren zu finden. Zusätzliche Systemcalls, eigene Initialisierungen und globale Variablen sind an dieser Stelle einzufügen.

`ba_drv.c` stellt die Funktionalität des eigentlichen Treibers zur Verfügung: die Interruptverarbeitung, das Streamsmanagement und die Verarbeitung von unerwarteten Ereignissen (z.B. Fehler). Funktionen, die für die reine Datenkollektion zuständig sind, aber u.U. auch Prozeduren, die die Kommunikation (hier den Datentransfer) mit einem Benutzerprogramm regeln sollen, sind hier zu integrieren.

Leider stellt die SunATM-Karte keine Möglichkeiten zur Verfügung, auf allen möglichen ATM-VCs Daten zu sammeln. Von der Karte werden nur Daten an den Treiber weitergegeben, die auf “angemeldeten” VCs ankommen. D.h. eine Art allgemeiner Promiscous-Mode ist aufgrund der zur Verfügung stehenden Hardware nicht implementierbar. Da aber (in LANs und MANs) in der Praxis nur relativ wenige VCs gleichzeitig benutzt werden, stellt das nur ein geringes Hindernis dar. Man kann per Systemaufruf “beliebig viele” VCs öffnen, auf denen dann auch Daten erfaßt werden können.

Konkret kann man sobald ein VC geöffnet wird in das Kernelmodul eingreifen, um die Daten vollständig (leider nur den Datenteil von AALx-PDUs, die Funktionalität der ATM-Anpassungsschicht wird komplett in der Hardware durchgeführt) zu erfassen. Dazu gibt es zwei interessante Stellen im Code.

Die Funktion ba_intr() wird für jeden Interrupt, der durch ein ankommendes Paket ausgelöst wird, aufgerufen. Der Nachteil dieser Stelle ist, daß man die komplette Interface- und Pufferbehandlung selbst durchführen muß.

Diese Arbeit kann man sich ersparen, indem man erst in der Funktion ba_read() eingreift. Hier kann man die Daten abgreifen, ohne die restliche Treiberfunktionalität zu beeinflussen. In welcher Art und Weise die nun vorliegenden Daten abgespeichert und an den Benutzerprozeß weitegegeben werden, soll in den folgenden Abschnitten erläutert werden.

4.1.2 Prozeß- / Programmiersystem

In Kapitel 3.2 “Dekomposition in ein Prozeßsystem” wurde ein Mehrprozeßsystem für die Implementierung einer ATM-Moni entworfen. Dabei werden drei Prozesse unterschieden:

- Kernelprozeß für die Datenkollektion (“Logger”)
- Userlevelprozeß für die Datenverarbeitung und den -transport (“Logger”)
- Userlevelprozeß für die Analyse der Daten (“Analyzer”)

Am Rechenzentrum der Universität Erlangen-Nürnberg wurde ein Produkt für die Analyse von IP-basiertem Netzverkehr entwickelt: die “Moni-Box”, welche ein Monitoring von Ethernet- und FDDI-Netzwerken erlaubt. Die Terme “Logger” und “Analyzer” wurden durch diese Applikation geprägt.

An dieser Stelle soll aber der Analyseprozeß nicht weiter betrachtet werden. Überlegungen, die die Kommunikation zwischen den Benutzerprozeß des Loggers und weiteren angekoppelten Prozessen betrachten, sind in Kapitel 5 “Ankopplung von Benutzerprozessen zur Protokollanalyse” zu finden.

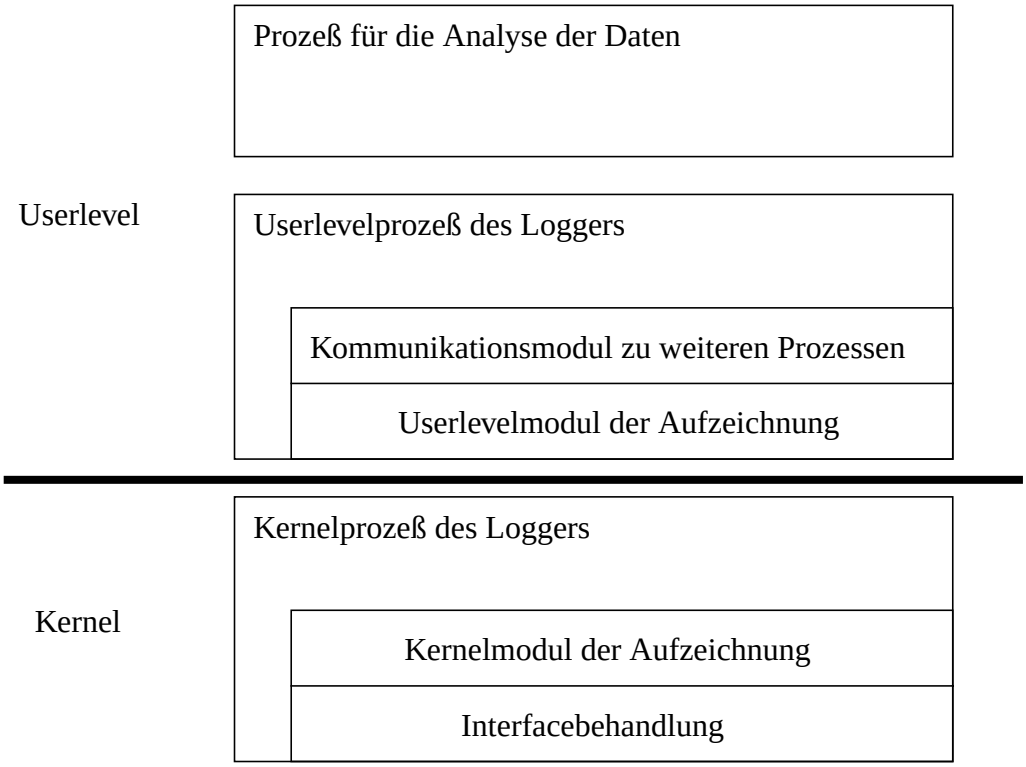


Abbildung 4.1 - Prozeß- / Programmiersystem

Zusätzlich zu der Unterteilung der ATM-Moni in ein Mehrprozeßsystem, werden die beiden Prozesse des Loggers noch in je zwei (Programmier-)Schichten unterteilt.

Benutzerprozeß des Loggers

- Aufzeichnungsmodul (Kommunikationsschnittstelle und Steuerung des Kernelmoduls)
- Kommunikationsschnittstelle zu weiteren Benutzerprozessen

Kernelprozeß des Loggers

- Aufzeichnungsmodul (Datenkollektion und -übergabe an das Userlevelmodul)
- Interfacebehandlung (SunATM, mit Patches)

Im folgenden werden aber die beiden Prozesse des Loggers nicht getrennt behandelt, da mehr Wert auf die Prozeßkommunikation und den Datenaustausch gelegt werden soll. Es werden aber immer beide Teile des Prozeßsystems gleichzeitig untersucht und beschrieben.

4.2 Vergleich verschiedener Lösungsansätze für die Kommunikation innerhalb des Loggers unter Solaris

Nach wichtigen Vorbetrachtungen soll jetzt der eigentlich interessante und wichtige Teil dieses Kapitels folgen: Der Vergleich struktureller Ansätze der Kommunikation zwischen Kernel- und Benutzerprozessen anhand einer Testimplementierung der Funktionalität des Loggers einer ATM-Moni.

4.2.1 Methode 1: SYSCALL

4.2.1.1 Prinzipien / Vor- und Nachteile

Der erste Gedanke, schnell Daten vom Kernel- zum Benutzerprozeß zu übertragen war, den bestehenden Kanal (den Stream) für diese Übertragung mitzubenutzen. Um sich nicht zu stark einzuschränken, sollte aber der normale Verkehr über das ATM-Interface unbeeinflußt bleiben. Auch die Übergaben von Statusinformationen und Koordinierungsaufgaben sollen über den gleichen Kanal gelöst werden.

Über einen Stream kann man mittels I_STR ioctl(3c) Aufrufen “out-of-band”-Informationen in beiden Richtungen verschicken.

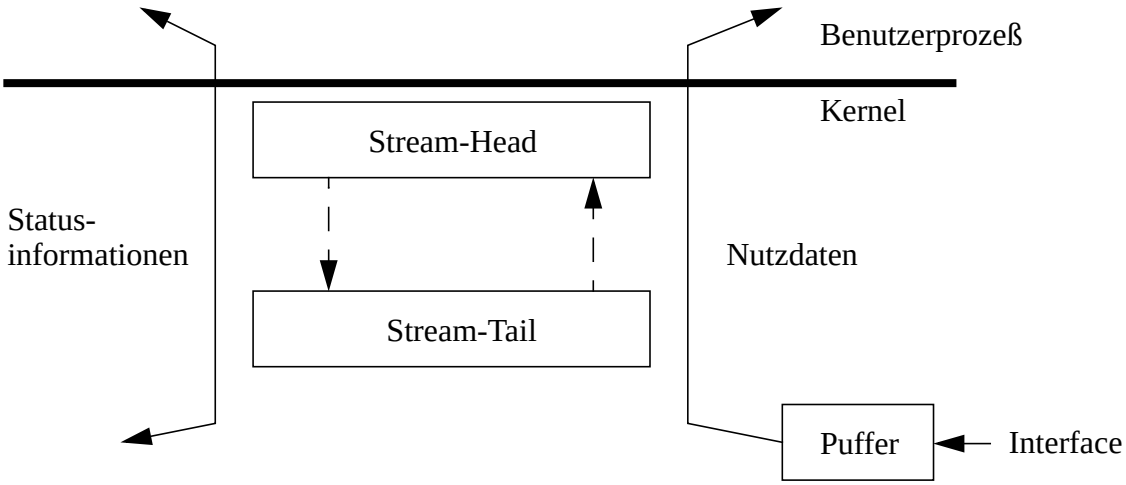


Abbildung 4.2 - Nutzung des Streams in Methode 1

Da vom ATM-Netz gelesene Daten nicht sofort und direkt an den Benutzerprozeß weitergegeben werden können muß im Kernel ein Zwischenspeicher implementiert werden. Dieser Puffer beeinflußt natürlich direkt die Koordinierungsaufgaben zwischen den beiden Prozessen.

Als Vorteil ist der (fast) unbeeinflußte normale ATM-Verkehr zu sehen. Nachteilig wirkt sich der Zusatzaufwand für die Pufferverwaltung aus. Es bleibt zu klären, ob die zu erwartende große Anzahl von Systemaufrufen pro Zeiteinheit vom Betriebssystem bewältigt werden kann.

4.2.1.2 Koordinierung / Übergabe von Statusinformationen

Wie schon beschrieben sollen sich Kernel- und Userlevelprozeß über einen SystemV-Stream koordinieren. Diese Koordinierung erfolgt hauptsächlich durch die Übergabe von Statusinformationen vom Kernel- an den Benutzerprozeß. So wird z.B. mitgeteilt, daß ein gefüllter Datenpuffer zur Übergabe bereitsteht, oder daß durch fehlenden Zwischenspeicher aufgezeichnete Daten verworfen werden mußten.

In der anderen Richtung muß eine Möglichkeit geschaffen werden, das Kernelmodul zu initialisieren bzw. neue Pufferspeicher zu allokieren.

Folgende Systemaufrufe wurden zum originalen ATM-Kernelmodul hinzugefügt (C-Code siehe Anhang E.1):

- `A_ATMMONI_INIT`
- `A_ATMMONI_SETDEBUG`
- `A_ATMMONI_ADDMEM`
- `A_ATMMONI_GETSTATS`
- `A_ATMMONI_GETBLOCK`

Für Koordinierungsaufgaben sind aber nur die ersten vier zuständig. Der fünfte Aufruf wird in Abschnitt 4.2.1.3 beschrieben.

Über den `A_ATMMONI_INIT`-Aufruf werden die für die Monitoringaufgaben zusätzlich zum Kernelmodul von Sun hinzugefügten Prozeduren und Variablen initialisiert. Wichtig ist das vor allem, damit man - so man auf ATM-Monitoring verzichten möchte - mit dem gleichen Kernelmodul “normale” ATM-Verbindungen ungestört nutzen kann.

Bei zwei der nachfolgend beschriebenen Methoden des Datenaustausches mit einem Benutzerprogramm kann man die Monitoring-Funktionalität zwar theoretisch parallel zu anderen Aufgaben der ATM-Karten nutzen, aber nach einigen Versuchen ist davon abzuraten. Erstens ist die Last auf den Interfaces ohnehin schon sehr hoch und zweitens sind dazu noch einige Abgrenzungen für den Normalbertrieb nötig.

Nur zur Erleichterung der Programmierung ist der `A_ATMMONI_SETDEBUG`-Aufruf gedacht. Dieser Systemaufruf kontrolliert eine interne Integervariable, mit der Debugausgaben abhängig vom Wert der Variablen hinzu- bzw. abgeschaltet werden können.

Dem Treibermodul wurde die Fähigkeit verliehen, einen internen Puffer zur Zwischenspeicherung von Daten anzulegen. Dieser Puffer kann dynamisch mit dem `A_ATMMONI_ADDMEM`-Aufruf vergrößert werden.

Das Monitoring-Modul im Kern sichert verschiedenste Daten für die Verwaltung freier Pufferspeicherplätze, für statistische Zwecke und zur Koordinierung mir dem Benutzerprozeß. Mit dem für die Koordinierung der Prozessen wichtigsten Aufruf `A_ATMMONI_GETSTATS` können die Daten übertragen werden (eine zweite Möglichkeit des Transfers von Statusinformationen wird in Kapitel 4.2.3 “Methode 3: DIRECT” vorgestellt).

Die wichtigsten Statusinformationen sind (die vollständige C-Struktur ist in Anhang E.1 abgebildet):

- `init` - Kernelmodul initialisiert?
- `cells` - Anzahl empfangener Zellen seit der Initialisierung
- `dropped_cells` - Anzahl der Zellen, die nicht an einen Benutzerprozeß weitergegeben werden konnten (d.h. der “verlorenen gegangenen” Zellen)
- `pdus` - Anzahl empfangener PDUs
- `dropped_pdus` - Anzahl der “verloren gegangenen” PDUs
- `blocks` - Anzahl Zwischenpuffer
- `filled_blocks` - gefüllte Zwischenpuffer
- `free_blocks` - freie Zwischenpuffer

4.2.1.3 Kommunikation / Nutzdatentransfer

Die wesentliche Problemstellung der vorliegenden Arbeit ist nicht “Wie kann ich Informationen aufzeichnen?”, sondern “Wie schaffe ich es, die gesammelten Informationen schnell genug zu weiterverarbeitenden Prozessen zu transportieren?”!

4.2.1.3.1 Datenstrukturen

Für die später folgende Analyse der Daten ist es erforderlich, daß nicht nur reine Nutzdaten abgespeichert werden, sondern auch zusätzliche Kanalinformationen. Da es leider nur möglich ist, die Informationsteile von AAL-PDUs zu monitoren (siehe Abschnitt 4.1.1), werden diese Zusatzinformationen sogar besonders wichtig. Ohne diese wäre es nicht möglich, z.B. eine Verbindungen höherer Protokolle (z.B. IP) ATM-VCs zuzuordnen.

Folgende Informationen werden gespeichert (die zugehörige C-Struktur ist in Anhang E.1 abgebildet):

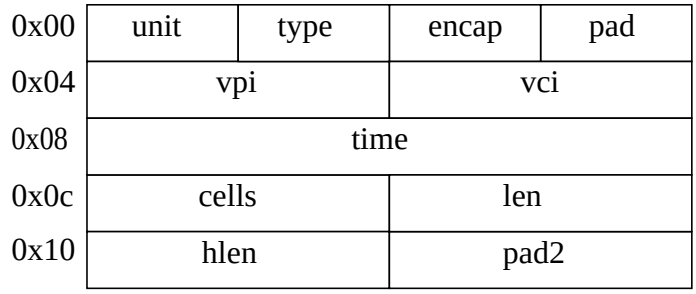


Abbildung 4.3 - Informationsdatenblock zu jeder PDU

- `unit` - über welche ATM-Karte kamen die Daten (Richtung!)
- `type` - AAL Level
- `encap` - Encapsulation
- `vpi` - VPI
- `vci` - VCI
- `time` - Ergebnis eine gethrtime(3c)-Aufrufes. Dieser Aufruf liefert unter Solaris einen 64 Bit-Wert (high resolution time; in Nanosekunden), welcher streng monoton steigend ist und dadurch die Reihenfolge der empfangenen PDUs sicherstellen kann.
- `cells` - Anzahl empfangender Zellen zu dieser PDU
- `len` - Länge der PDU in Byte
- `hlen` - Länge des Headers
- `pad` und `pad2` - Padding Byte zum Auffüllen der Struktur auf Wort- bzw. Langwortgrenze

4.2.1.3.2 Datentransfer

Im Kernel wurde ein dynamisch erweiterbarer Ringpuffer als Zwischenspeicher integriert. Sobald Daten vom ATM-Interface empfangen werden, werden diese in einen freien Puffer kopiert.

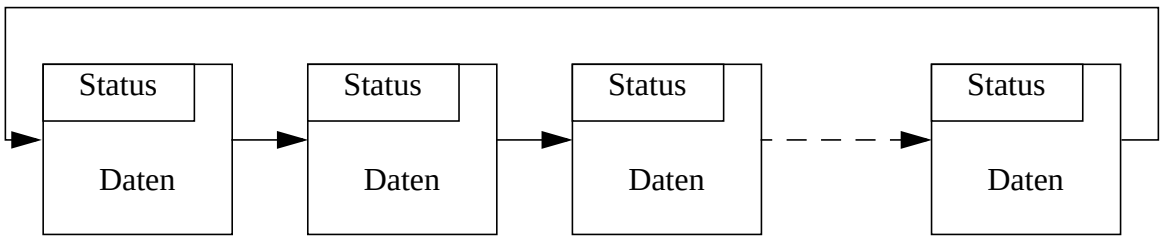


Abbildung 4.4 - benutzter Ringpuffer

Der Benutzerprozeß erfragt von Zeit zu Zeit den aktuellen Status (siehe Abschnitt 4.2.1.2). Sind gefüllte Puffer für die Übertragung bereit, werden diese mit den gleichen Mitteln übertragen, wie auch die Statusinformationen: mit `I_STR` ioctl(3c) Aufrufen.

4.2.2 Methode 2: READ

4.2.2.1 Prinzipien / Vor- und Nachteile

Ein wesentlicher Nachteil der ersten Methode ist, daß für jeden (Nutz-)Datentransfer eigentlich zwei Systemaufrufe nötig sind. Mit dem ersten wird erfragt, ob ein Puffer für die Übergabe bereit ist und mit dem zweiten wird dieser Puffer übertragen.

Da wahrscheinlich die normale Nutzung der ATM-Karte während des Monitoring-Vorganges doch nicht möglich ist, ist die logisch konsequente Frage: “Warum nicht die Standard-Übertragungsmöglichkeiten des Streams nutzen?”.

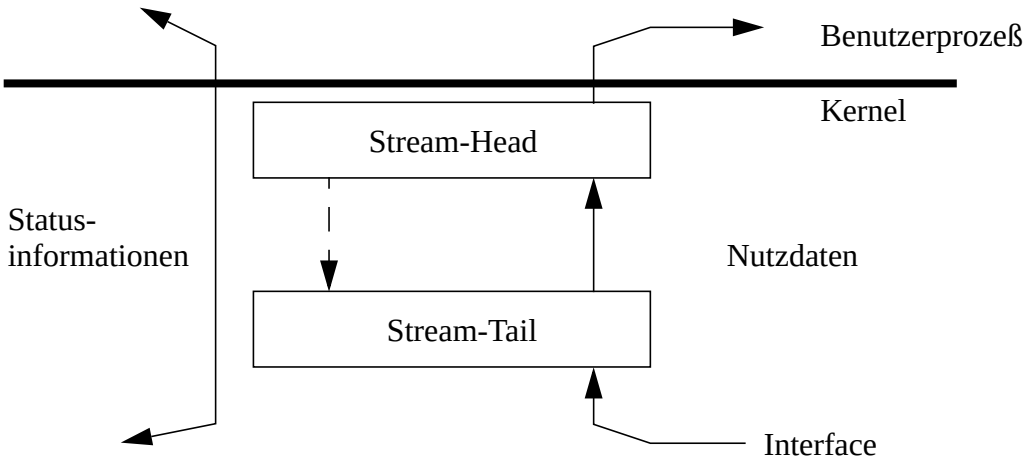


Abbildung 4.5 - Nutzung des Streams in Methode 2

Dies hat mehrere Vorteile: Erstens kann man sich die Pufferverwaltung im Kern sparen, da man die Daten für die Übertragung an den Benutzerprozeß “einfach” am Stream-End in diesen hineinsteckt und den Rest der Verwaltungsaufgaben dem Betriebssystem überläßt. Da gerade bei Solaris 2.6 diese Mechanismen besonders effizient implementiert sind, ist das sicher kein großer Nachteil gegenüber der Benutzung von “out-of-band”-Daten. Und zweitens spart man sich die koordinierende Übergabe der Statusinformationen vor jedem Datenblock. Die Statusinformationen sind jetzt nur noch für die Feststellung von Interesse, ob gesammelte Daten durch überfüllte Puffer (das Betriebssystem stellt nur eine begrenzte Menge Speicher für die Datenübergabe über Streams zur Verfügung) verworfen werden mußten.

4.2.2.2 Koordinierung / Übergabe von Statusinformationen

Die Übergabe der Statusinformationen erfolgt wie bei Methode SYSCALL über `I_STR` ioctl(3c) Aufrufe. Die Datenstruktur ist aber etwas einfacher geworden, da keine Informationen über Kernel-interne Puffer übergeben werden müssen.

Die Koordinierung ist wesentlich einfacher (und damit effizienter) als bei Methode SYSCALL. Es wird nicht mehr nach den Nutzdaten “gepollt”, sondern durch die Benutzung der Standardmethoden zum Lesen von Streams - read(3c) - wird der Benutzerprozeß bei Nichtanliegen von Daten blockiert.

4.2.2.3 Kommunikation / Nutzdatentransfer

Der Datentransfer wird durch ein direktes Modifizieren der Stream-Tail-Routine eingeleitet, so daß alle Daten über den Stream übertragen werden. Dadurch kann man die (mittlerweile) sehr guten (in bezug auf Geschwindigkeit) Fähigkeiten von Solaris 2.6 in der Messageverwaltung / im Messagetransport auf Streams ohne Einschränkungen ausnutzen. Das Hauptproblem ist, daß das ATM-Interface nicht mehr für andere Aktionen als das ATM-Monitoring nutzbar ist.

Der Datentransfer erfolgt mit den Standard-Mechanismen des Betriebssystems, konkret über read(3c) Aufrufe.

4.2.3 Methode 3: DIRECT

4.2.3.1 Prinzipien / Vor- und Nachteile

Das Problem der beiden ersten Methoden ist offensichtlich die hohe Anzahl von Systemaufrufen pro Zeiteinheit (für SYSCALL sind pro Datenblock zwei und für READ ein Systemaufruf nötig). Für diese braucht das Betriebssystem sehr viel Zeit. Der Datentransfer muß weiter optimiert werden!

Der zuerst verworfene Gedanke über die Nutzung eines gemeinsamen Speichers zur Datenübertragung vom Kern- in den Benutzeradreßraum soll an dieser Stelle neu aufgegriffen werden. Verworfen wurde er deshalb, da bei dieser Art ein maximaler Verwaltungsaufwand für die Pufferverwaltung entsteht. Dies kostet natürlich sehr viel CPU-Zeit, aber auf der anderen Seite vermeidet man viele Systemaufrufe.

Für den gemeinsamen Speicher gibt es zwei Möglichkeiten:

- Der Kernel hat einen Speicher und ein Benutzerprozeß darf diesen lesen bzw. schreiben (via `/dev/kmem`). Der Vorteil ist, daß die keine Koordinierung bzgl. der Lauffähigkeit des Benutzerprozesses nötig ist. Weiterhin besteht so die einfache Möglichkeit des Neustartens des Benutzerprozesses, ohne Manipulation am Kerneltreiber.

- Der Benutzerprozeß hat einen Speicher und der Kernelprozeß schreibt dort hinein. Das größte Problem ist, daß der Speicher gelockt, d.h. vor Auslagerung auf einen Hintergrundspeicher (swap) und vor Verlegung in andere physikalische Speichersegmente geschützt werden muß. Außerdem muß der Kern vor jedem Schreibversuch klären, ob der Benutzerprozeß, zu welchem der Speicher gehört, noch lauffähig ist.

Offensichtlich ist die erste Lösung wesentlich unaufwendiger und einfacher implementierbar.

Wesentliche Vorteile dieser Methode sind die Einsparung von (u.U. sehr vielen) Systemaufrufen und der (Wieder-)Nutzbarmachung des Interfaces für normalen ATM-Verkehr. Auf jeden Fall Nachteilig ist die hohe Komplexität der Koordinierung. Die Pufferverwaltung kann fast unverändert aus Methode 1 übernommen werden.

4.2.3.2 Gemeinsamer Speicher zwischen Kernel- und Benutzerprozeß unter Solaris 2.6

Die Nutzung von gemeinsamen Speichers zwischen einem Kernel- und einem Benutzerprozeß unter UNIX ist relativ einfach zu realisieren. Über das Pseudo-Device `/dev/kmem`[33] ist es möglich, direkt auf den Kernelspeicher zuzugreifen.

Um das nicht über umständliche Fileoperationen machen zu müssen, kann man dieses File (wie jedes andere auch) durch den mmap(3c) Aufruf in den Speicher abbilden. Danach kann man den Speicher ganz normal benutzen.

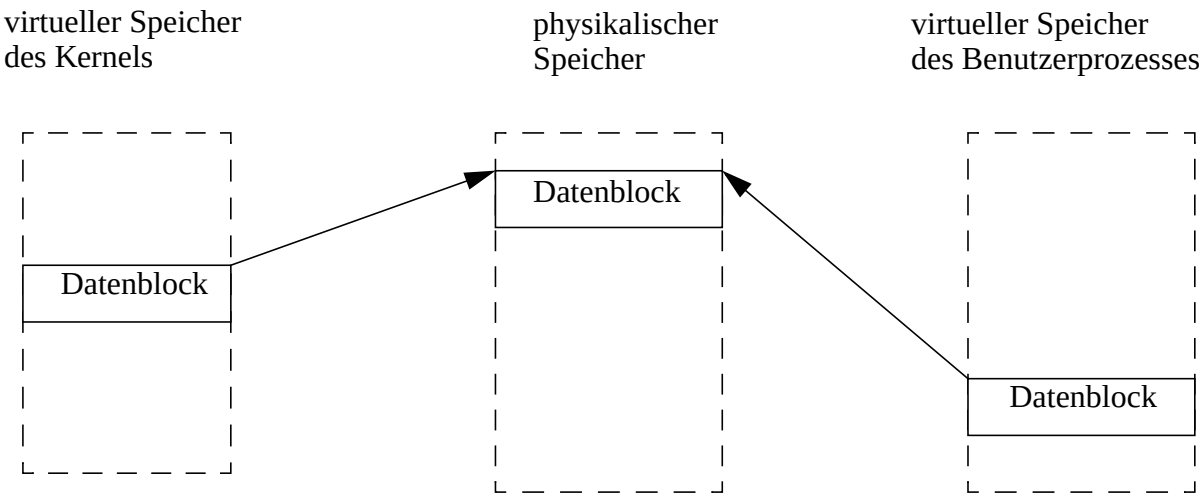


Abbildung 4.6 - Abbildung des Kernelspeichers in den Benutzerprozeß

Das einzige was man wissen muß, ist die Anfangsadresse des Speicherblocks, den man lesen/modifizieren will. Diese Adresse wird von dem `A_ATMMONI_ADDMEM`-Systemaufruf als Ergebnis zurückgeliefert.

33. Der Name kann je nach UNIX-Derivat variieren.

4.2.3.3 Koordinierung / Übergabe von Statusinformationen

Nur noch für die Initialisierung und für die Erweiterung der Pufferspeicher sind `I_STR` ioctl(3c) Aufrufe zu implementieren. Diese sind mit denen aus der Methode SYSCALL fast identisch.

Die Koordinierung erfolgt über einen gemeinsamen Speicher. Dieser enthält folgende Struktur (C-Code siehe Anhang E.1):

init
cells
dropped_cells
pdus
dropped_pdus
blocks

Abbildung 4.7 - Datenstruktur für die Statusinformationen

Diese Statusinformationen braucht der Benutzerprozeß nur von Zeit zu Zeit zu lesen, um festzustellen, ob Daten wegen überfüllter Puffer verloren gegangen sind.

Die Koordinierung bzgl. der Zwischenspeicher erfolgt über Flags in den Datenblöcken. Diese kennzeichnen einen Datenblock als “filled”, wenn er für die Datenübergabe an den Benutzerprozeß bereit ist und “read”, wenn der Benutzerprozeß die Daten gelesen hat, d.h. daß er wieder für eine erneute Nutzung bereitsteht.

4.2.3.4 Kommunikation / Nutzdatentransfer

Der Nutzdatentransfer kann durch einfache Leseoperationen im Speicher erfolgen, da der Benutzerprozeß auf den selben physikalischen Speicher zugreifen kann, den auch der Kern benutzt. Auf diese Art und Weise ist es möglich, die notwendige Anzahl von Kopieraktionen im Speicher zu minimieren. Auch ist es denkbar, weiteren angekoppelten Benutzerprozessen zur Datenanalyse Zuriff auf diese Speicher zu geben, so daß weiter Kopieraktionen eingespart werden können. Dem steht aber entgegen, daß Kernelspeicher eine Ressource ist, mit der möglichst sparsam umzugehen ist.

4.2.4 Vergleich und Auswertung der entwickelten Methoden

Die entwickelten Methoden unterscheiden sich vor allem in der Anzahl nötiger Systemaufrufe und der Komplexität der Implementierung.

Die Anzahl der Systemaufrufe (ohne Initialisierung) ist bei dem Methoden SYSCALL, READ, DIRECT im Verhältnis 2 : 1 : 0 zu sehen. Bei der Methode SYSCALL sind pro Datenblock zwei Systemcalls nötig, bei READ nur noch einer. Bei DIRECT werden die Daten nicht mehr über Systemaufrufe, sondern über einen gemeinsamen Speicher kopiert, d.h. das System wird hier nicht mehr durch die vielen Systemaufrufe ausgebremst.

Hinzugefügt werden muß aber, daß bei der Aufzeichnung von vielen, sehr kleinen Paketen die Anzahl der benötigten Systemaufrufe bei der Methode READ um ein Vielfaches höher ist, als bei den beiden anderen, da hier nicht zwischengepuffert wird.

Aber auch die Anzahl von Kopieraktionen im Speicher auf dem Weg der Daten zum Benutzerprozeß variiert:

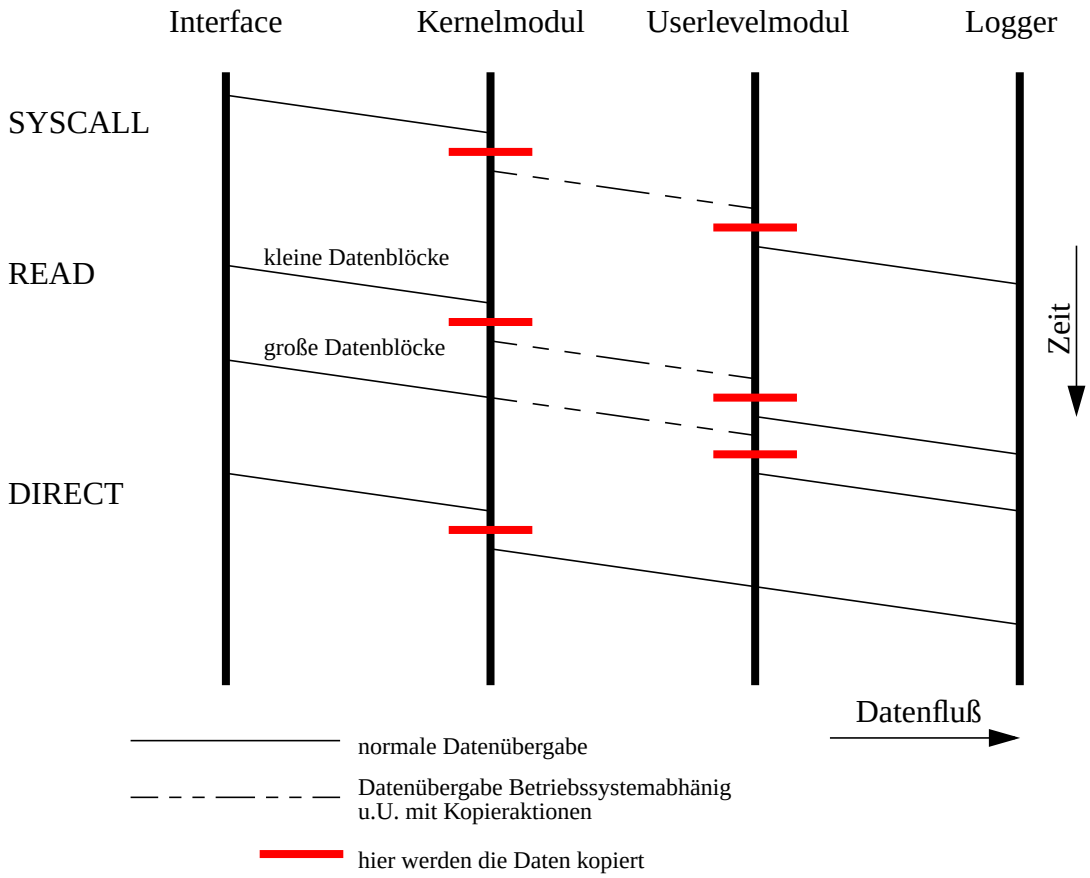


Abbildung 4.8 - Vergleich der nötigen Kopieroperationen

Bei der SYSCALL-Methode werden die Daten immer zweimal kopiert. Für große Datenblöcke wird bei der READ-Methode eine Kopieraktion eingespart (Solaris Streams-Management). Am besten schneidet in diesem Vergleich die DIRECT-Methode ab, bei der nur eine einzige Kopieraktion nötig ist.

Zusammenfassend kann man sagen, daß nach den theoretischen Überlegungen und “normalem” Netzverkehr (d.h. mittlere Rate, mittlere Paketgrößen) die Methode DIRECT als einzige bestehen wird, gefolgt von SYSCALL und READ.

Ob sich diese Behauptungen belegen lassen, wird sich durch konkrete Messungen der Leistungsfähigkeit der verschiedenen Methoden in Kapitel 4.4 “Ergebnisse und Auswertung” zeigen.

4.3 Vorbetrachtungen für eine Implementation unter FreeBSD

Wie auch unter Solaris, ist das erste (und für die Funktionalität wichtigste) zu lösende Problem die Datenkollektion. Um dieses anzugehen, sind Schnittpunkte in der Treiberimplementierung der ATM-Interfaces unter FreeBSD zu finden, an denen das Abgreifen des Netzwerkverkehrs besonders effizient und einfach möglich ist.

Die Entwicklung des HARP-Treibers wurde im Rahmen einer Master-Thesis am Minnesota Supercomputer Center begonnen und aufgrund regen Interesses aus der FreeBSD-Gemeinde fortgesetzt [40]. Besonderer Wert wurde auf eine solide Strukturierung gelegt, bei der vor allem die einzelnen Protokollstufen von ATM gut ersichtlich sein sollten.

Leider gibt es bis heute keine Dokumentation des Treibers. Nur die Konfiguration des Treibers für Classical IP over ATM wird kurz beschrieben und ist auch kein Problem.

Da zu keiner Zeit Wert auf eine vollständige Implementierung der ATM- und AAL-Funktionalität gelegt wurde, sondern als Beispielanwendung nur IP funktionieren sollte (und das ist derzeit die einzige Anwendung in der Praxis), sind viele Teile des Treibers noch unvollständig bzw. überhaupt nicht integriert. So ist es z.B. noch nicht einmal möglich, über ein Benutzerprogramm einen PVC zu öffnen, ohne auf Signalling-Funktionen zu verzichten.

Für eine ATM-Moni muß aber jegliches Signalling ausgeschaltet sein, da erstens sonst der normale Verkehr beeinflußt wird (zur Erinnerung: es werden optische Leitungssplitter zum Abhören eines Netzes eingesetzt) und zweitens ist Signalling nur mit einem Rückkanal möglich, der aber gänzlich fehlt (wieder der Leitungssplitter).

Versuche, ohne Dokumentation und nur mit dem Verständnis der Funktionalität des Treibers, welches aus dem Lesen der Quellen erlangt wurde, die fehlenden Teile des Treibers in möglichst kurzer Zeit zu implementieren, schlugen leider fehl.

FreeBSD mit Fore-ATM-Karten und dem HARP-Treiber mußte als Basis für eine ATM-Moni vorerst aufgegeben werden. Trotz allem ist aber ein Pentium-PC unter FreeBSD eine interessante Alternative zu einer Sun ULTRA1. Erstens ist der Preis eines PCs unschlagbar günstig und zweitens sind die ersten Tests über IP auf dem PC ähnlich gut ausgefallen wie auf der Sun. Sobald eine bessere Unterstützung seitens der HARP-Entwickler vorliegt oder die Entwicklung eines alternativen Treibers erfolgt, sollten die Tests unter FreeBSD fortgesetzt werden.

4.4 Ergebnisse und Auswertung

Um die Implementierung des Loggers zu testen, wurde ein einfaches Programm geschrieben, welches im wesentlichen die Funktionalität einer ATM-Moni ohne den analysierenden Teil hat. Der “test_logger” ist eine stark vereinfachte Form des zukünftigen Loggers der ATM-Moni. Es werden zwar die Daten bis in den Benutzeradreßraum kopiert, dann aber sofort verworfen, d.h. man muß beachten, daß eine weitere Verfügbarkeit der CPU für die Analyse gewährleistet sein muß.

Zu Testzwecken (z.B. der Datenintegrität oder stichprobenhafte Analysen) ist es bei diesem Programm aber möglich, Stichproben der aufgezeichneten Daten zu entnehmen (z.B. durch Ausgabe in eine Datei).

Trotz der Tatsache, daß unter FreeBSD derzeit keine Aufzeichnung möglich ist, wurde das “test_logger”-Programm so geschrieben, daß die grundlegenden Programmteile und Funktionen Architektur- und Compilerunabhängig sind. Der Quellcode wurde außerdem so ausgelegt, daß er ohne weitere Veränderungen als Basis für die weitere Entwicklung der ATM-Moni dienen kann.

Für die Ermittlung konkreter Zahlen, welche die Qualität der entwickelten und implementierten Methoden ausweisen sollen, wurde mit dem i586-PC ATM-Verkehr erzeugt. Dies vor allem deshalb, um exakte Aussagen über das Verhalten der ATM-Moni bei verschiedener Last und bei verschiedenen Paketgrößen zu bestimmen. Die Last wurde mit dem schon für die allgemeinen Tests benutztem “push” erzeugt.

An dieser Stelle sollen anhand von einigen Grafiken die wichtigsten Ergebnisse gezeigt werden. Die vollständigen Ergebnisse sind in Anhang F zu finden. Zu beachten ist, daß bei den Vergleichen der Paketverlustrate nur die Methoden verglichen werden sollen. Es handelt sich relative Raten, da auch die erzeugte Last auf dem ATM-Netz Paketgrößenabhängig ist (vgl. Kapitel 2.4 “Performancetests mit Standardumgebungen”).

Wie man aus nachfolgender Abbildung sofort sieht, eignet sich eine SPARC5 auf keinen Fall für den Einsatz als ATM-Moni. Die CPU ist fast immer voll ausgelastet und die Paketverlustraten sind nicht akzeptabel. Nur mit der Methode DIRECT und bei sehr großen Paketen kommt die SPARC5 hält sich die Paketverlustrate in Grenzen.

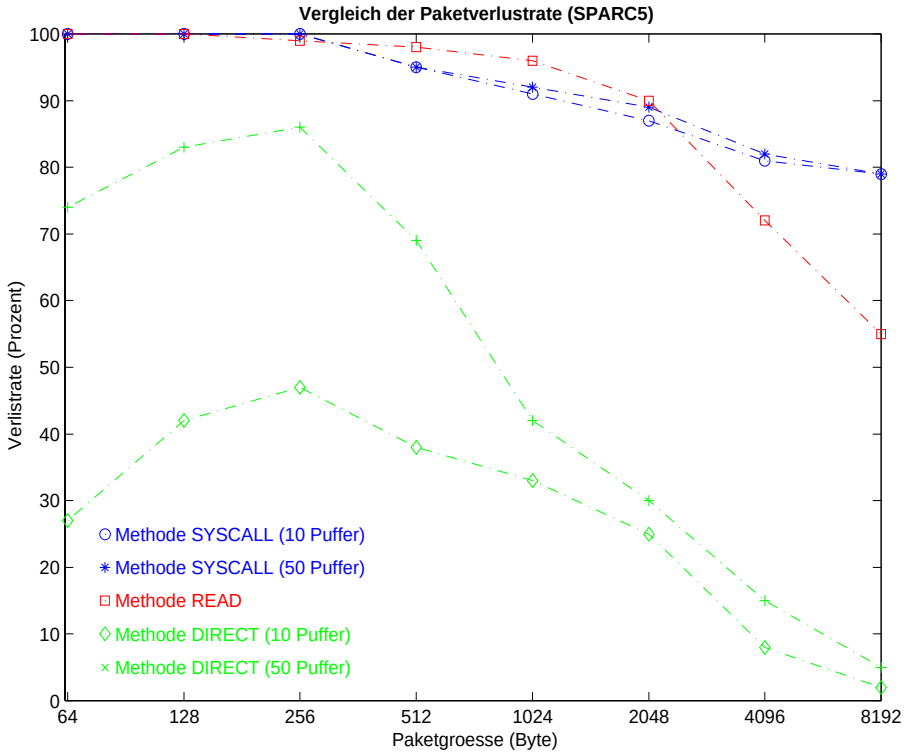


Abbildung 4.9 - Vergleich der Paketverlustrate auf der SPARC5

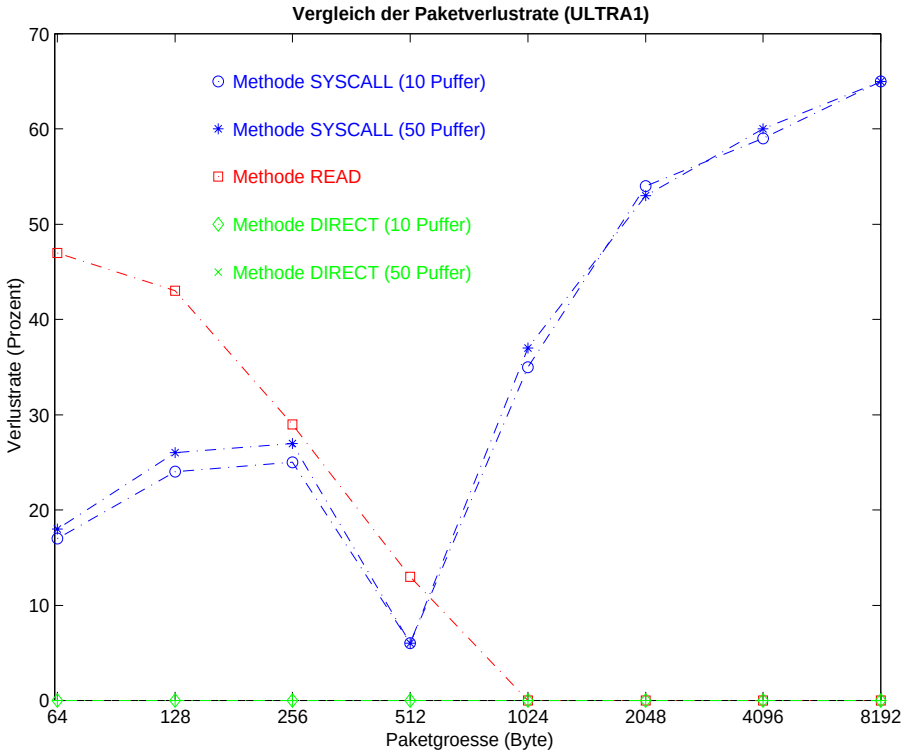


Abbildung 4.10 - Vergleich der Paketverlustrate auf der ULTRA1

Im Gegensatz zur SPARC5 sind bei der ULTRA1 die Unterschiede der einzelnen getesteten Methoden besser sichtbar. Aber trotz der hohen Rechengeschwindigkeit der ULTRA1 kann nur die Methode DIRECT im Vergleich überzeugen. Schlechter als erwartet zeigt sich im Test die Methode SYSCALL. Bei kleinen Paketgrößen ist diese zwar erwartungsgemäß besser als READ, aber trotzdem ist die Paketverlustrate nicht akzeptabel.

Zusammenfassend kann man feststellen, daß nur die Datenübergabe über einen gemeinsamen Speicher (Methode DIRECT) schnell genug ist, um die hohe Datenrate von bis zu 155MBit/s zwischen einem Kernel- und einem Benutzerprozeß zu bewältigen. Auch hat hierbei die CPU noch Reserven für weitere Aufgaben (siehe Abbildung 4.11).

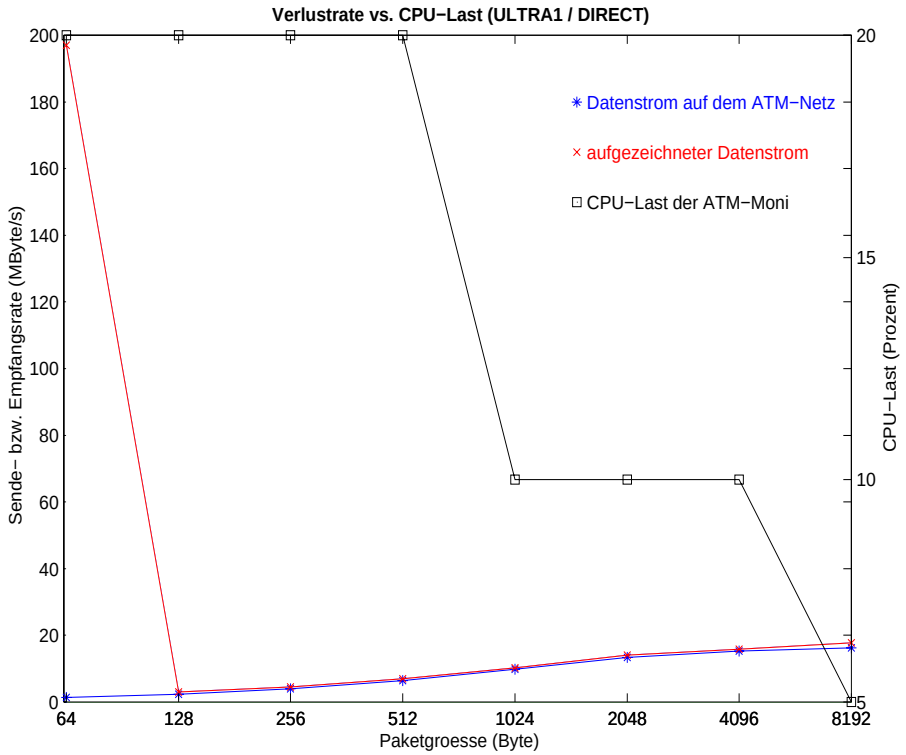


Abbildung 4.11 - Verlustrate vs. CPU-Last (ULTRA1 / DIRECT)

An dieser Stelle noch nicht betrachtet ist allerdings die Notwendigkeit der Ankopplung weiterer Prozesse für die Datenanalyse. Dies soll im folgenden Kapitel betrachtet werden.

Eine allgemeine Aussage über die Möglichkeit der Implementierung und Nutzung einer ATM-Monitoring-Einheit auf normalen Arbeitsplatzrechnern, ist also noch nicht möglich, da möglicherweise andere Anforderungen als bisher getestet zum Tragen kommen. Nur die testweise eingesetzte SPARC5 hat sich als zu wenig performant erwiesen. Eine zusammenfassende Auswertung ist in Kapitel 6 “Zusammenfassung und Ausblick” zu finden.

5 Ankopplung von Benutzerprozessen zur Protokollanalyse

In diesem Kapitel soll die Kommunikation zwischen dem Benutzerprozeß des Loggers (im weiteren einfach Logger genannt) und weiteren angekoppelten Benutzerprozessen (speziell des Analyzers) erörtert werden. Grundlage der Diskussion ist wieder die an der Universität Erlangen-Nürnberg entwickelte Moni-Box[34], welche eine ähnliche Funktionalität wie eine ATM-Moni für Ethernet- bzw. FDDI-basierte Netze bietet.

Im Vordergrund der Untersuchungen soll nicht der angekoppelte Prozeß (Analyzer), sondern die Kommunikation und Koordinierung stehen.

Ziel dieses Kapitels ist aber auch die Integration der entwickelten Funktionalität eines ATM-Monitors in die bestehende Moni-Box.

5.1 Problemstellung

Im Prinzip gelten bei der Kommunikation zwischen dem Logger und weiteren Benutzerprozessen dieselben Gesetzmäßigkeiten, die schon zwischen Kernel- und Userlevelprozeß des Loggers beobachtet wurden (Kapitel 4 “Vergleich struktureller Ansätze der Kommunikation zwischen Kernel- und Benutzerprozessen”).

Es gilt auch an dieser Stelle ein Mehrprozeßsystem zu koordinieren und einen Datentransfer mit extrem hoher Datenrate durchzuführen. Trotzdem bestehen Unterschiede. Z.B. ist das zeitliche Verhalten nicht so sehr von Bedeutung, da kein Datenverlust entsteht, wenn ein Prozeß nicht schnell genug neue Daten aufnehmen kann. Die Kopplung der Prozesse kann also “lose” erfolgen.

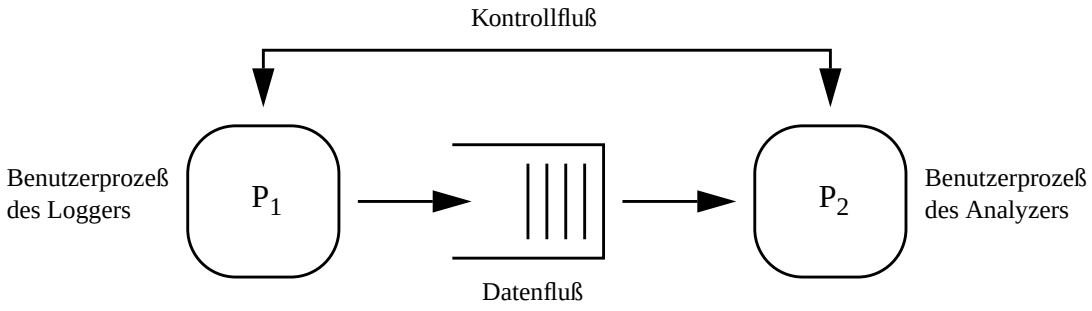


Abbildung 5.1 - Kopplung zwischen Logger und Analyzer

34. Die erste Version der Moni-Box wurde von Bernd Flemming an der Universität Erlangen-Nürnberg im Rahmen einer Diplomarbeit entwickelt [46].

Durch eine Entzerrung der Verbindung entstehen natürlich Probleme bei der Koordinierung der Prozesse. Diese Probleme gilt es zu lösen. In einer Testimplementierung soll herausgefunden werden, ob die Funktionalität einer ATM-Moni auch durch die weitere Ankopplung von Benutzerprozessen noch auf normalen Arbeitsplatzrechnern (speziell einer Sun ULTRA1) möglich ist.

5.2 Kommunikationsmöglichkeiten unter UNIX

Zu unterscheiden sind hier die reine Koordinierung (Kontrollfluß) zwischen Logger und Analyzer und der Datentransfer (Datenfluß). Unter UNIX stehen für die IPC[35] Möglichkeiten wie z.B. Signals, Pipes, Sockets oder Dateien zur Verfügung[36]. Es sollen im folgenden nur die wichtigsten und für aktuelle Betrachtung als am geeignetsten anzusehenden Mechanismen für die Interprozeß-Kommunikation betrachtet werden.

Signals

Durch die Übermittlung von Signalen können 1-Bit-Informationen zwischen Prozessen übermittelt werden. Leider ist die Anzahl möglicher Signale sehr begrenzt (zwei frei verwendbare Signals). Außerdem ist eine Neusynchronisation nach dem Neustart eines der kommunizierenden Prozesse schwierig, da die dynamische Feststellung dessen PID[37] schwierig ist.

Shared Memory

Über einen gemeinsamen Speicher können mehrere Prozesse sowohl Informationen als auch Daten austauschen. Dies ist eine sehr effiziente Art der Datenübergabe. Da aber sehr große Datenmengen übertragen werden müssen, ist voraussichtlich die Größe des zur Verfügung stehenden Speichers eine wichtige Grenze.

Pipes

Ein sehr einfacher Weg, Daten zwischen zwei Prozessen zu übergeben sind Pipes. Aber auch hier wirkt sich die Größe des Hauptspeichers und zusätzlich die von betriebssysteminternen Puffern begenzend aus. Ein Neustart eines teilnehmenden Prozesses ist nicht möglich.

---

35. Interprocess Communication
36. Es soll an dieser Stelle keine vollständige Auflistung der Möglichkeiten der IPC unter UNIX (und seiner Derivate) folgen. Vielmehr sollen, sondern ausgehend von den speziellen Aufgaben für die Koordinierung und Kommunikation zwischen Logger und Analyzer einer ATM-Moni, die interessantesten und für die Anwendung am sinnvollsten erscheinenden Mittel vorgestellt und analysiert werden.
37. Prozeß-ID

Named Pipes

Named Pipes sind eine Erweiterung der “normalen” UNIX-Pipes. Über eine Referenz im Filesystem können diese nach dem Neustart eines Prozesses wieder geöffnet werden.

Sockets

Sockets sind ein im 4BSD eingeführter Mechanismus zur vereinheitlichten Kommunikationssteuerung auf UNIX-Rechnern. Dabei ist es unwichtig, ob die Kommunikation lokal auf dem Rechner stattfindet, oder über angeschlossene Datennetze. Ein Neustart eines Kommunikationspartners ist kein Problem (ein Neustart der Kommunikation ist möglich), aber auch hier sind die Zwischenpuffer auf dem Kommunikationsweg sehr begrenzt.

Dateien

Über Dateien im Filesystem können (fast) beliebig große Datenmengen übergeben werden. Die Geschwindigkeit ist vom physikalischen Medium abhängig. Wird z.B. eine RAM-Disk benutzt, sind die Operationen auf der Datei sehr schnell.

Eine Zusammenstellung und Bewertung der IPC-Mechanismen ist in Tabelle 5.1 zu finden. Dabei stehen “xx” für sehr gut und “oo” für unbrauchbar. Wichtig ist, daß keineswegs alle von UNIX zur Verfügung gestellten Möglichkeiten aufgelistet wurden, sondern nur solche, die auf den ersten Blick für die Kommunikation zwischen dem Benutzerprozeß des Loggers und dem Analyzer in Frage kommen. Auch ist die Wertung ist rein auf das Einsatzgebiet zugeschnitten und nicht als allgemeingültig anzusehen.

IPC-Mechanismus	Eignung für die Koordinierung	Eignung für den Datentransfer	Wiederaufnahme (nach Neustart eines Prozesses)
Signals	xx	oo	o
Shared Memory	x	xx	x
Pipes	o	x	oo
Named Pipes	x	x	x
Sockets	x	x	x
Files	x	x	xx

Tabelle 5.1 - Eignung der IPC-Mechanismen für den Datenaustausch

Eignung steht hier für maximale Geschwindigkeit bei der Koordinierung bzw. maximalen Durchsatz beim Datentransfer. So ist also die Prozeßkoordinierung durch Signals sehr effizient und für den Datentransport eignet sich Shared Memory am besten.

Da aber zusätzlich zur Frage nach der Performance auch die nach der Funktionalität und der Sicherheit von Interesse ist, ist die Möglichkeit des Wiederanlaufs eines teilnehmenden Prozesses sehr wichtig. Am unproblematischsten läßt sich dies durch die IPC durch Files ermöglichen. Es ist also ein Kompromis zwischen Funktionalität und Performance zu finden.

5.3 Lösungsansätze der Moni-Box

Die bereits desöfteren erwähnte Moni-Box soll als Grundlage für die Implementierung der Analyseeinheit der ATM-Moni dienen, da auf eine komplette Protokollanalyse für IP und höhere Protokolle (z.B. TCP) zurückgegriffen werden kann. In den folgenden Abschnitten soll die Struktur der Moni-Box untersucht und auf Optimierungsmöglichkeiten in Bezug auf die Kommunikation zwischen Logger und Analyzer überprüft werden.

5.3.1 Strukturanalyse des Programmpaketes “Moni-Box”

Die Moni-Box in ihrer bestehenden Form wurde von verschiedenen Mitarbeitern entwickelt und programmiert. Die Entwicklungsumgebung ist sehr stark architekturabhängig. Alle Arbeiten fanden unter FreeBSD 2.x statt. Jeder Programmierer kopierte sich den Code in ein privates Verzeichnis und arbeitete dort daran weiter, es gab keine allgemeingültige Programmierumgebung. Erst in der letzten Version (2.3.1) wurde die Entwicklung unter CVS[38]-Kontrolle gestellt, um ein paralleles Arbeiten mehrerer Programmierer am Code zu ermöglichen.

Die Aufzeichnung der Daten vom Netz erfolgte mit dem unter BSD-Systemen sehr beliebten und auch sehr mächtigen BPF[39]. Mit der libpcap[40] existiert ein API welches die Komplexität des BPF vor dem Programmierer verbirgt. Außerdem ist die Analyse der Netzwerkdaten mit der Bibliothek sehr einfach. Sowohl die Trägerprotokolle (z.B. Ethernet oder FDDI) als auch die höheren Transportprotokolle (z.B. IP, TCP, UDP oder IPX) sind mit den zur Verfügung stehenden Funktionen analysierbar. Leider ist die Verknüpfung der Moni-Box-Programme mit der libpcap so eng, daß eine Integration der im Rahmen von Kapitel 4 “Vergleich struktureller Ansätze der Kommunikation zwischen Kernel- und Benutzerprozessen” entstandenen Funktionen für einen ATM-Monitor schwierig erscheint.

Es gibt aber erste Strukturierungsansätze. Z.B. sind allgemeine Funktionen, die sowohl im Logger als auch im Analyzer Verwendung finden zu einer Library zusammengefaßt (`libcommon.a`). Diese Bibliothek aber noch nicht vollständig Architektur- und Compilerunabhängig.

38. Concurrent Versions System - Versionskontrolle für parallele Entwicklung
39. Berkeley Packet Filter
40. Universelle Programmierbibliothek für Netzwerkschnittstellen

Folgende Abbildung soll die bestehende Struktur der Programmierumgebung der Moni-Box besser verdeutlichen:

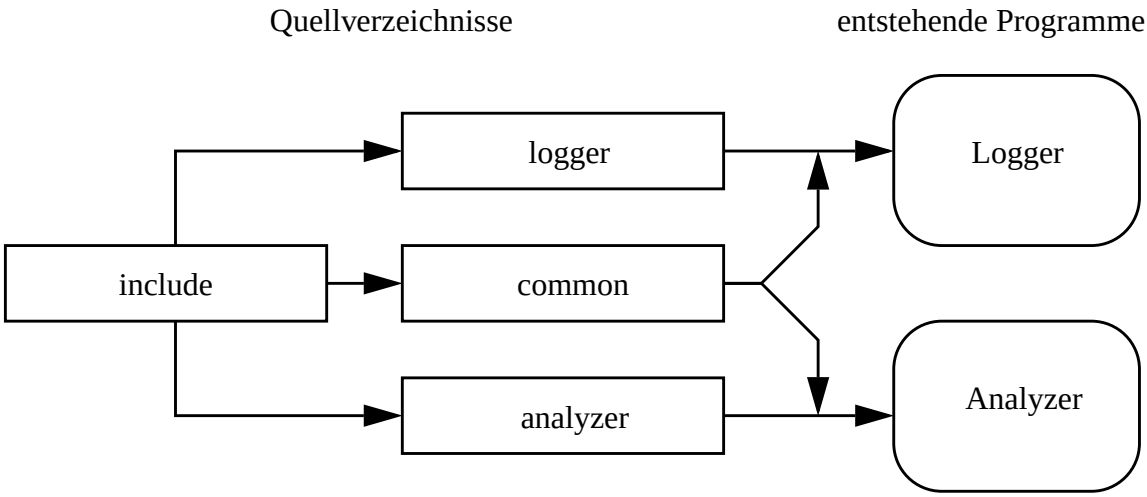


Abbildung 5.2 - Struktur des Moni-Box-Programmpaketes

5.3.2 Kommunikationssteuerung zwischen Logger und Analyzer

Nachdem im letzten Abschnitt rein programmiertechnische Aspekte der Moni-Box betrachtet wurden, soll jetzt die Kommunikation zwischen Logger und Analyzer der Moni-Box analysiert und kritisch betrachtet werden. Vor allem sollen mögliche Schwachpunkte entdeckt werden.

Sowohl Datenübertragung als auch Koordinierung (im Fall der Moni-Box besteht die Koordinierung lediglich aus der Kontrolle der Datenübergabe) funktionieren über das Filesystem.

Für den reinen Nutzdatentransfer werden Verzeichnisse zu einem Warteschlangensystem verknüpft. Alle Transferverzeichnisse bekommen anhand ihrer Geschwindigkeit Prioritäten zugeteilt. So erhält z.B. eine RAM-Disk eine sehr hohe Priorität. Ein weiterer Parameter ist die Größe der Warteschlange, d.h. die Kapazität des Filesystems.

Der Logger zeichnet den Netzverkehr auf und füllt mit den erhaltenen Daten die Verzeichnisse. Über ein weiteres Verzeichnis, welches nur koordinierende Aufgaben erfüllt, werden die Dateinamen der gespeicherten Nutzdaten an den Analyzer weitergegeben. Dieser verarbeitet die Daten und gibt danach den Speicher durch Löschen der Datei wieder frei.

Die beiden folgenden Abbildungen zeigen sowohl eine logische als auch eine physikalische Sicht auf dieses Modell.

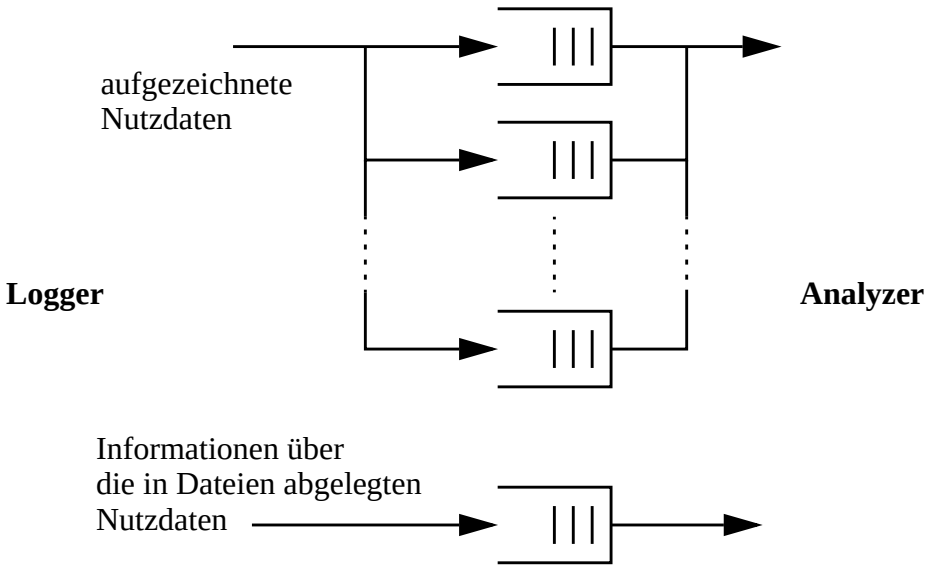


Abbildung 5.3 - Logische Sicht der Datenübertragung vom Logger zum Analyzer

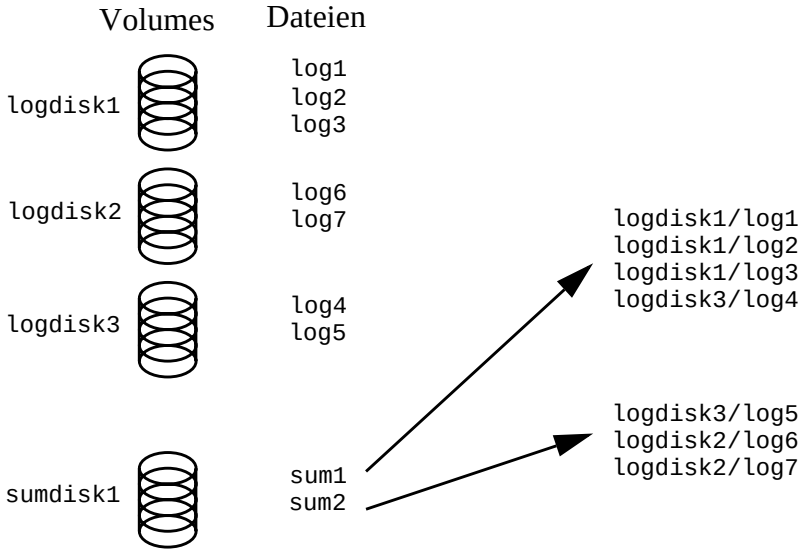


Abbildung 5.4 - Physikalische Sicht der Datenübertragung vom Logger zum Analyzer

Das wichtigste Problem dieser Konfiguration ist, daß der Datentransfer immer über die Dateioperationen abgewickelt wird, welche im Gegensatz z.B. zu einem gemeinsamen Speicher relativ langsam sind.

Es gibt aber auch einen entscheidenden Vorteil. Beide Prozesse (Logger und Analyzer) können jederzeit beendet und neu gestartet werden, ohne daß sich eine Beeinträchtigung der Funktionalität des Gesamtsystems ergibt. Dies ist auch der Grund, warum die in älteren Versionen der Moni-Box eingesetzte Kommunikation über Sockets durch diese Variante ersetzt wurde.

Ob diese Lösung in Hinblick auf die Performance der Analyse auch für ATM-Monitoring sinnvoll ist, muß eine Implementierung mit anschließenden Tests zeigen.

5.4 Vorgehensweise für die Integration des ATM-Monitors in die Moni-Box

Ziel dieses Kapitels ist es, die prinzipielle Vorgehensweise bei der Integration des entwickelten ATM-Monitors in die bestehende Moni-Box zu zeigen. Wichtig ist, daß es sich tatsächlich um eine Integration, nicht um eine Neuimplementierung handeln soll. D.h. die volle Funktionalität der Moni-Box soll erhalten bleiben und um eine Aufzeichnungs- und Analyseeinheit für ATM erweitert werden.

5.4.1 Allgemeine Schritte

In einem ersten Schritt gilt es eine Umgebung einzurichten, die sowohl unter FreeBSD (original Moni-Box), als auch unter Solaris (entwickelter ATM-Monitor) funktioniert. Die wichtigsten Schritte dabei sind das Einrichten von Makefiles, die für verschiedenste Compiler funktionieren (konkret für den GNU C-Compiler und den der Firma Sun). Ebenso gehört dazu eine Analyse der verwendeten Tools auf Verfügbarkeit und Funktionalität unter den unterschiedlichen Betriebssystemen.

Die zweite Aufgabe ist es, die Funktionsbibliothek `libcommon.a` vollständig architektur- und compilerunabhängig zu machen. Im wesentlichen beschränkt sich diese Aufgabe auf die Ersetzung von architektur- bzw. compilerabhängigen Funktionsaufrufen durch allgemeingültige bzw. den Einsatz bedingter Compilierung.

Der dritte und letzte vorbereitende Schritt ist es, die Abhängigkeiten von der `libpcap.a` auf FreeBSD und die Aufzeichnung bzw. Analyse von Ethernet- und FDDI-Paketen zu beschränken. Dies ist wichtig, damit die vorhandene Funktionalität der Moni-Box nicht verloren geht. Eine einfache Methode dieses Ziel unkompliziert zu erreichen, ist der Einsatz bedingter Compilierung.

5.4.2 Logger

Die Integration ATM-Loggers in den Loggerprozeß der Moni-Box ist leider nicht ohne Probleme möglich, da die Struktur der Aufzeichnungsroutinen bedingt durch die Aufgabe sehr unterschiedlich ist. Nur die Programmteile, die die Kommunikation mit dem Analyzer steuern, sind weiterverwendbar.

Im Endeffekt kann das aus Kapitel 4 bekannte “test_logger”-Programm so umgeschrieben und um die Kommunikationsroutinen des Moni-Box-Loggers erweitert werden, daß die gewünschte Funktionalität vollständig erreicht wird.

5.4.3 Analyzer

Eine vollständige Implementierung eines ATM-Analyzers ist durch Verwenden des Analyzers aus der Moni-Box leider nicht möglich. Die Abhängigkeiten des Moni-Box-Analyzers von der `libpcap.a` sind zwar “einfach entfernbar”, aber damit wird auch die komplette Protokollanalyse abgeschaltet.

Eine minimale Version des Analyzers kann durch den Ersatz der Analyse durch eine Routine, welche die aufgezeichneten Daten nicht weiterverarbeitet, sondern sofort verwirft, erstellt werden. Das entstehende Programmsystem kann aber für die Bewertung der Leistungsfähigkeit der Kommunikation zwischen Logger und Analyzer genutzt werden, da die Analyse der Daten nur noch reine “straight-forward”-Programmierung ist. Das ist der Grund, weshalb die CPU-Last der Analyse linear von der Anzahl zu erkennender Protokolle bzw. Protokollschichten abhängt. Die Leistungsfähigkeit des Gesamtsystems ATM-Moni wird so innerhalb sehr enger Grenzen abschätzbar.

5.5 Ergebnisse und Auswertung

Mit dem in Abschnitt 5.4.3 beschriebenen minimalen Analyzer und einem vollständig implementierten Logger wurden weitere Tests auf der Sun ULTRA1 durchgeführt. Diese dienen im wesentlichen nur der Abschätzung der Leistungsfähigkeit der ULTRA1 unter verschiedenen implementierten Methoden der Datenübertragung vom Kernel- zum Benutzeradreßraum (auf Tests mit der SYSCALL-Methode wurde bewußt verzichtet, ebenso wie auf solche mit der SPARC5, da die Ergebnisse der ersten Untersuchungen diese sich schon als unbrauchbar für die Aufgaben eines ATM-Monitors erwiesen haben).

Der Verkehr auf dem ATM-Netz wurde wieder künstlich durch den Einsatz von “push” erzeugt. Die vollständigen Ergebnisse sind in Anhang G aufgelistet.

Wie schon bei vorangegangenen Messungen, sollen die wichtigsten Daten anhand einiger anschaulicher Grafiken gezeigt werden.

In Abbildung 5.5 ist die Anzahl der Datenverluste im Vergleich zur CPU-Last unter Einsatz der Methode READ aufgezeigt. Offensichtlich ist diese Methode nicht für den Einsatz in einem ATM-Monitor geeignet.

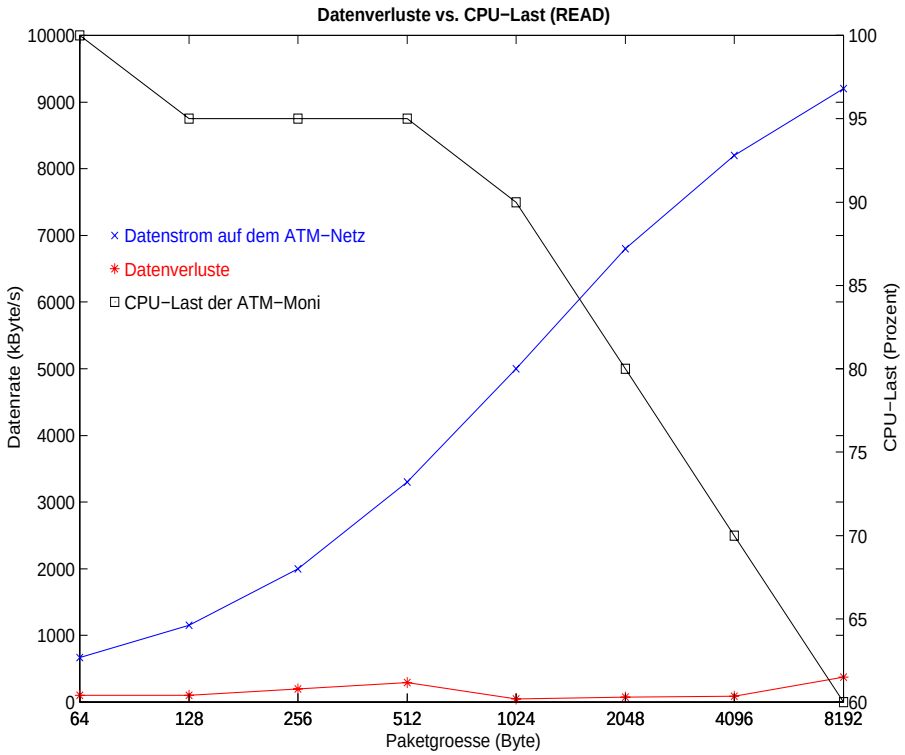


Abbildung 5.5 - Datenverluste vs. CPU-Last (Methode READ)

Die folgende Grafik zeigt dieselben Daten für die DIRECT-Methode. Es sind keine Datenverluste mehr zu verzeichnen und die CPU-Last während der Messung ist so niedrig, daß weitere Aktivitäten zur Protokollanalyse keine Schwierigkeiten bereiten sollten.

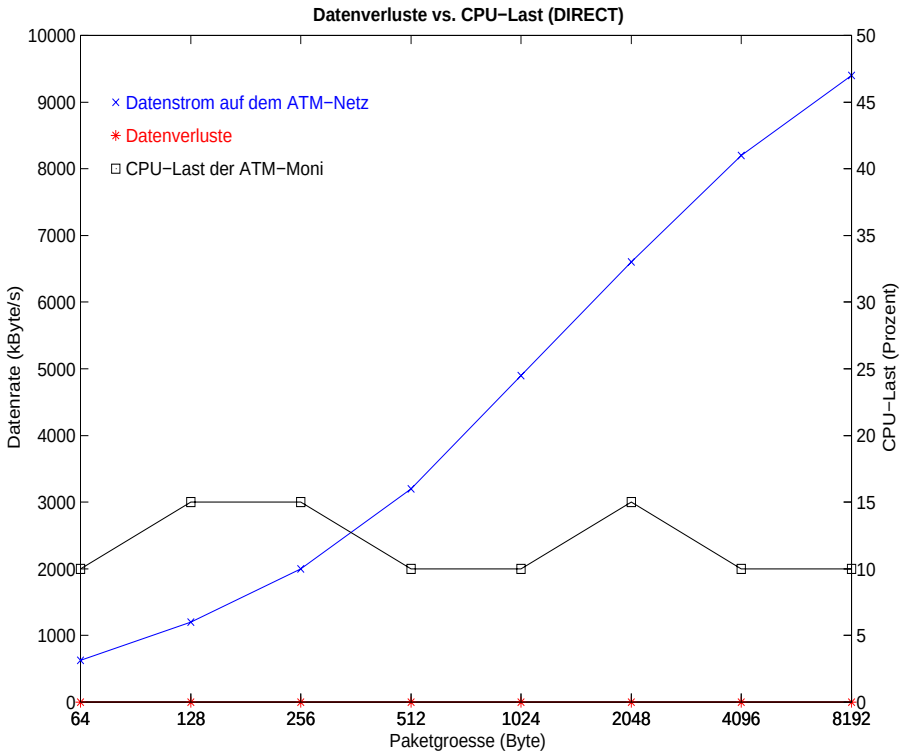


Abbildung 5.6 - Datenverluste vs. CPU-Last (Methode DIRECT)

Zusammenfassend kann man also feststellen, daß die ULTRA1 für den Einsatz als ATM-Monitor in Frage kommt, vorausgesetzt man setzt geeignete Mittel für den Datentransport und die Prozeßkommunikation auf dem Rechner ein.

6 Zusammenfassung und Ausblick

Wichtigstes Ergebnis der vorliegenden Arbeit ist, daß die Frage nach der Möglichkeit des Monitorings von ATM-Netzen und der Analyse der gesammelten Daten mit Hilfe moderner Arbeitsplatzrechner positiv beantwortet werden kann.

In den verschiedenen Teststufen wurde die Leistungsfähigkeit der eingesetzten Computer (SPARC5, ULTRA1 und i586-PC) und der benutzten Betriebssysteme ermittelt und anhand der gewonnenen Ergebnisse beurteilt. Resultat ist, daß die testweise eingesetzte SPARC5 nicht fähig ist, Daten mit einer Rate von 155 MBit/s zu verarbeiten.

Für die Gewinnung von aussagekräftigen Daten über die Performance eines ATM-Monitors auf o.g. Rechnern, wurde eine Implementierung der Funktionalität des Monitorings vorgenommen. Das entstandene Programm wurde in ein Mehrprozeßsystem aufgegliedert, wobei drei wesentliche Prozesse entstanden:

- Kernelprozeß des Loggers. Dieser im Kernel des UNIX-Betriebssystems laufende Prozeß dient der Aufzeichnung der Daten vom Netzwerkinterface. Er ist sehr speziell auf die zu Grunde liegende Hardware zugeschnitten.
- Benutzerprozeß des Loggers. Um eine Schnittstelle zwischen der hardwarabhängigen Kollektion der Verbindungsdaten und der nur protokollabhängigen Analyse zu schaffen und außerdem eine lose Kopplung zwischen diesen beiden Teilen zu bilden, wurde dieser Prozeß eingefügt. Er übernimmt die Daten vom Kernelteil des Loggers und stellt sich weiteren Prozessen als eigentlicher Logger entgegen.
- Benutzerprozeß des Analyzers. Dieser an den Logger angekoppelte Benutzerprozeß nimmt die gesammelten Daten zum Zwecke der Analyse und Auswertung entgegen. Die Resultate könnten z.B. durch weiter Skripten verarbeitet werden und so einen Überblick über die Verkehrsstruktur im Netz geben.

Der Datenaustausch zwischen den Prozessen, speziell zwischen Kernel- und Benutzeradreßraum erwies sich wie erwartet als kritischer Faktor. Für die Kommunikation wurden drei verschiedene Methoden des Datenaustausches zwischen den beiden Prozessen des Loggers entwickelt und durch in der Implementierung erprobt.

- SYSCALL. Ziel dieser Methode ist es, die Daten über die bestehende Verbindung zwischen den Prozessen, dem Stream, zu transportieren. Dazu wird im Kernel ein Zwischenpuffer für die aufgezeichneten Daten eingerichtet. Der Benutzerprozeß stößt von sich aus die Übertragung durch einen Systemaufruf an. Angekoppelt an den Rückgabewert des Aufrufes wird ein kompletter Puffer als sogenannte “Out-of-band”-Information übergeben. Das Problem sind die vielen Systemcalls. Pro Datenblock sind zwei Aufrufe nötig, einer zur Sicherstellung, daß ein Datenblock bereitsteht und einer für die Übergabe.

- READ. Um die Anzahl der Systemaufrufe zu vermindern, wird in dieser Methode der Stream in seiner eigentlichen Funktionalität manipuliert und dient danach nur noch der Kommunikation zwischen den beiden Prozessen des Loggers. Dies spart im Mittel einen Systemaufruf pro Datenblock ein.
- DIRECT. Da die READ-Methode zu langsam war, wurde der Datenaustausch über einen gemeinsamen Speicher vorgenommen. Alle Kommunikations- und Koordinierungsaufgaben erfolgen über diesen Speicher. Das Resultat ist, daß kein einziger Systemaufruf mehr benötigt wird.

Der Datenaustausch zwischen dem Logger und dem Analyzer wurde mit Mitteln der in der Arbeit beschriebenen Moni-Box realisiert. Dabei erfolgt sowohl die Koordinierung wie auch der Datentransfer über das UNIX-Filesystem, wobei aber primär die RAM-Disk, d.h. ein Teil des Hauptspeichers, welcher sich dem Anwender wie eine Festplatte präsentiert, eingesetzt wurde.

Die eingesetzte ULTRA1 erwies sich in den durchgeführten Performancetests als geeignet, die Aufgaben des ATM-Monitorings und der Protokollanalyse bei einer Datenrate von 155 MBit/s durchzuführen. Der i586-PC konnte zwar bei den ersten Performancetests über “IP über ATM”-Verbindungen überzeugen, jedoch konnte aufgrund des unausgereiften Treibers keine Testimplementierung eines ATM-Monitors erfolgen, so daß keine endgültige Aussage über die Einsatzmöglichkeiten kann.

Für zukünftige Weiterentwicklungen der ATM-Moni kann es empfohlen werden, die Untersuchungen unter FreeBSD mit anderen bzw. weiterentwickelten Treibern fortzusetzen, da der i586-PC unter FreeBSD in den Voruntersuchungen ein ähnliches Leistungsprofil wie die ULTRA1 zeigte.

Für den praktischen Einsatz der ATM-Moni fehlt noch die Implementierung der Protokollmaschine für die Datenanalyse. Dies kann durch nachfolgende Arbeiten fortgesetzt werden.

A Aufbau von AAL-PDUs

Zuerst der Aufbau der SAR[1]-PDUs, also der Datenpakete in der unteren ATM Anpassungsschicht (AAL):

SN = Sequence Number, SNP = Sequence Number Protection

Abbildung A.1 - AAL1 SAR PDU [2]

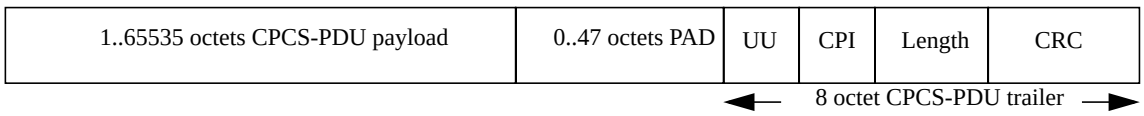
CT = Celltype, LEN = Length, CRC = Cyclic Redundancy Check

Abbildung A.2 - AAL2 SAR PDU [2]

CT = Celltype, LEN = Length, CRC = Cyclic Redundancy Check

Abbildung A.3 - AAL3/4 SAR PDU [2]

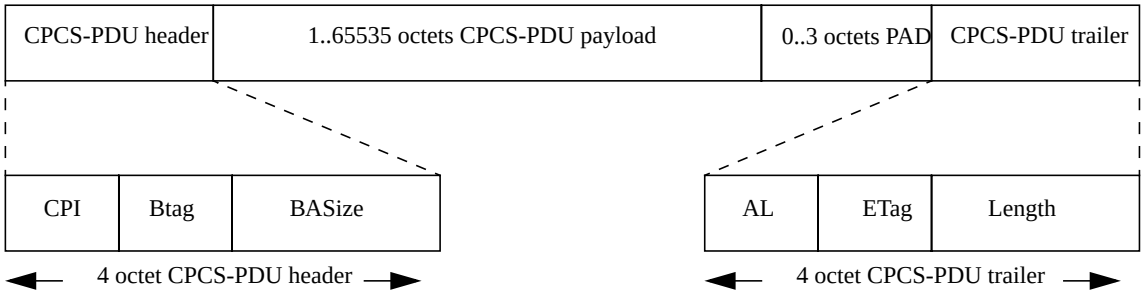
Folgend, der Aufbau der CPCS[2]-PDUs, der Datenpakete im oberen Teil der AAL:



(CPCS-)UU - CPCS user-to-user
CPI - Common part indicator
CRC - Cyclic redundancy check
PAD - Padding
Length - Length of payload

Abbildung A.4 - AAL3/4 CPCS PDU [4], [7]

1. Segemtation and Reassembly
2. Common Part Convergence Sublayer



AL - Alignment
BASize - Buffer allocation size
Btag - Beginning tag
CPI - Common part indicator
Etag - End tag
Length - Length of payload
PAD - padding

Abbildung A.5 - AAL5 CPCS PDU [4], [7]

B Eingesetzte Nicht-Standard-Werkzeuge

Für die Performancetests und die Vergleiche der implementierten Programme kamen folgende Nicht-Standard-Werkzeuge zum Einsatz:

B.1 “push”

“push” ist ein von Toerless Eckert am Lehrstuhl für Betriebssysteme (IMMD4) der Universität Erlangen-Nürnberg entwickeltes Programm. Es dient dazu, konfigurierbare Datenraten zu generieren und über UDP/IP zu versenden bzw. wieder zu empfangen. Dabei kann es sowohl für Lasttests angeschlossener Netzkomponenten, wie auch zur Messung des maximalen Durchsatzes einer IP-Verbindung benutzt werden.

B.2 “atmpush”

“atmpush” wurde von Falko Dreßler am Regionalen Rechenzentrum der Universität Erlangen-Nürnberg für Durchsatztests über ATM-Netzwerke entwickelt. Wie auch “push” kann es sowohl für die Generierung von (konfigurierbarer) Netzlast, als auch zur Bestimmung des maximalen Durchsatzes eingesetzt werden.

C Protokoll der “push”-Messungen

Bei den Durchsatzmessungen wurden immer sowohl auf Seite des Senders, als auch auf der des Empfängers der Durchsatz ermittelt (in den folgenden Tabellen und Grafiken als gemittelte Werte). Dieses Vorgehen wurde gewählt, da UDP (das angewandte Transportprotokoll) keine Sicherung der Übertragung vornimmt, d.h. Pakete verloren gehen können. Für die Bestimmung der Fähigkeiten der einzelnen Rechner und ihrer Netzwerkverbindungen interessieren sowohl Sendedurchsatz (was schafft man auf das Netz zu schreiben), als auch Empfangsdurchsatz (wieviel kann die Maschine vom Netz empfangen). Dabei wurden alle Tests mit verschiedenen Paketgrößen durchgeführt, da dieser Wert von bedeutendem Einfluß ist.

C.1 Messungen über das Loopback-Interface

Um einen groben Überblick über die Leistungsfähigkeit der IP-Implementierungen der verschiedenen eingesetzten Arbeitsplatzrechner und ihrer Betriebssysteme zu bekommen, wurden die “push”-Messungen zuerst auf dem Loopback-Interface durchgeführt (ACHTUNG: Im Gegensatz zu den “richtigen” Messungen über die ATM-Schnittstellen, sind hier Sender und Empfänger auf einer Maschine vereint!):

	Senden		**Empfangen**	
Paketgröße	**Pakete/s**	**kByte/s**	**Pakete/s**	**kByte/s**
64	3881.5	242.2	1587.6	98.8
128	3307.0	413.0	1635.2	204.0
256	3488.8	871.8	1368.6	341.6
512	2727.3	1363.5	1545.6	772.6
1024	2899.0	2899.0	1201.0	1201.0
2048	2316.3	4633.0	1044.6	2089.4
4096	1636.0	6545.2	815.4	3262.6
8192	1063.5	8511.8	578.8	4633.4
16384	401.0	6420.0	266.2	4265.2

Tabelle C.1 - “push”-Messung über das Loopback-Interface der SPARC5

Bemerkung: Für größere Pakete keine Zunahme des Durchsatzes (CPU am Anschlag).

	Senden		Empfangen	
Paketgröße	**Pakete/s**	**kByte/s**	**Pakete/s**	**kByte/s**
64	10418.2	650.7	10408.4	650.2
128	9833.2	1228.8	9816.2	1226.5
256	8798.6	2199.4	8840.0	2209.8
512	8323.8	4161.6	8330.2	4164.8
1024	7627.0	7627.0	7630.8	7630.8
2048	5790.4	11581.0	5837.8	11676.0
4096	3759.4	15040.0	3760.5	15042.0
8192	1989.2	15917.8	2004.0	16034.2
16384	-	-	-	-

Tabelle C.2 - “push”-Messung über das Loopback-Interface des i586-PC

Bemerkung: Größere Pakete als 8192 Byte können nicht über die UDP/IP-Implementierung von FreeBSD gesendet (wg. CERT/CC[3] security advisory), aber wohl empfangen werden.

	Senden		Empfangen	
Paketgröße	**Pakete/s**	**kByte/s**	**Pakete/s**	**kByte/s**
64	14763.7	922.3	6954.3	434.3
128	13248.0	1655.5	6562.8	820.0
256	13718.0	3429.2	5748.0	1436.6
512	11289.8	5644.7	4749.7	2374.7
1024	12847.0	12847.0	4827.7	4827.7
2048	11487.2	22975.0	4447.5	8895.2
4096	8775.7	35104.3	3680.8	14724.3
8192	6550.5	52406.2	3248.7	25993.2
16384	2593.2	41494.2	1824.8	29205.3

Tabelle C.3 - “push”-Messung über das Loopback-Interface der ULTRA1

Bemerkung: Die ULTRA1 schafft es nicht, alle selbst geschickten Pakete wieder zu empfangen, dafür schickt sie aber wesentlich mehr Pakete über ihr Interface, als der i586-PC.

3. CERT Coordination Center / Computer Emergency Response Team

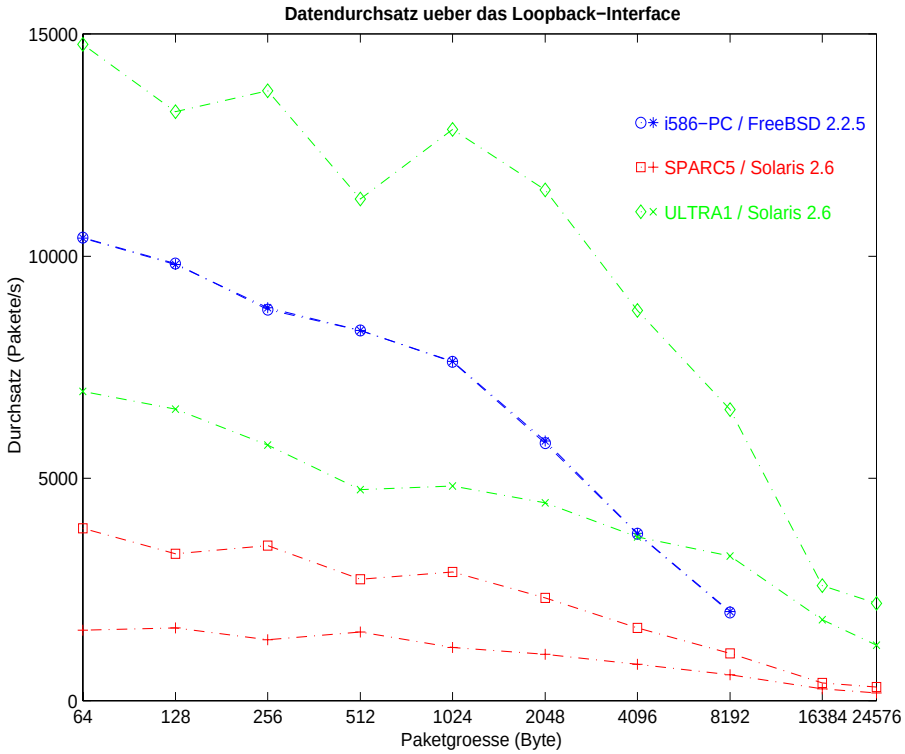


Abbildung C.1 - Datendurchsatz über das Loopback-Interface (Pakete/s)

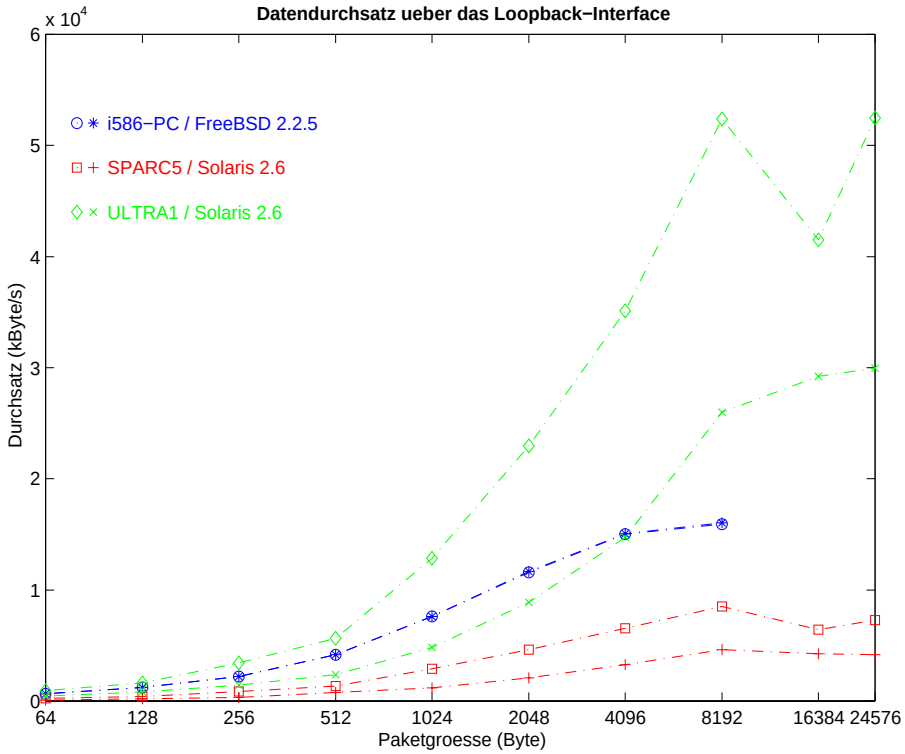


Abbildung C.2 - Datendurchsatz über das Loopback-Interface (kByte/s)

C.2 Messungen über ATM (SPARC5 <-> i586-PC)

Es folgen die eigentlich interessanten Messungen über die ATM-Karten der Workstations. Zuerst zwischen der SPARC5 und dem i586-PC.

Paketgröße	Sender (SPARC5)		Empfänger (i586-PC)	
	Pakete/s	kByte/s	Pakete/s	kByte/s
64	4745.5	296.3	4733.8	295.4
128	4510.0	563.3	4498.8	561.8
256	3413.0	852.8	3409.8	852.2
512	3213.0	1606.0	3212.6	1606.0
1024	2889.8	2889.8	2882.2	2882.2
2048	2378.8	4758.2	2381.0	4763.0
4096	1658.2	6634.3	1648.4	6595.8
8192	1079.5	8637.7	1076.2	8613.0
16384	389.2	6233.7	388.0	6215.8

Tabelle C.4 - “push”-Messung über ATM von SPARC5 zu i586-PC

Bemerkung: SPARC5 sendet mit maximaler Leistung, der i586-PC hat keine Probleme die ankommenden Daten korrekt zu empfangen.

Paketgröße	Sender (i586-PC)		Empfänger (SPARC5)	
	Pakete/s	kByte/s	Pakete/s	kByte/s
64	20563.2	1284.7	23.3	1.1
128	26732.4	3341.4	18.2	1.8
256	28537.4	7134.0	4.0	0.8
512	23147.0	11573.2	3.0	1.2
1024	14494.8	14494.8	2.8	2.8
2048	7900.0	15801.0	2.0	4.0
4096	3984.4	15938.6	2.7	11.5
8192	2010.4	16086.0	258.0	2066.0
16384	-	-	-	-

Tabelle C.5 - “push”-Messung über ATM von i586-PC zu SPARC5

Bemerkung: Der i586-PC überflutet die SPARC5 mit UDP-Paketen (>20.000 64-Byte-Pakete pro Sekunde bis immerhin noch knapp 8.000 2048-Byte-Pakete). Daraufhin zeigt die SPARC5 keinerlei Reaktion mehr am Netzinterface. Erst ab einer Paketgröße von 8192 Byte können aufgrund des geringer werdenden Paketstromes wieder Daten empfangen werden.

Zu den folgenden Abbildungen ist anzumerken, daß in der Legende der erste Teil (z.B. “SPARC5 -> i586-PC”) die Verbindung beschreibt (also hier von der SPARC5 zum i586-PC) und der zweite Teil (z.B. “send”) den Ort, an dem die Messung des Durchsatzes erfolgte (im Beispiel sendet also die SPARC5 Daten mit der in der Grafik ersichtlichen Rate).

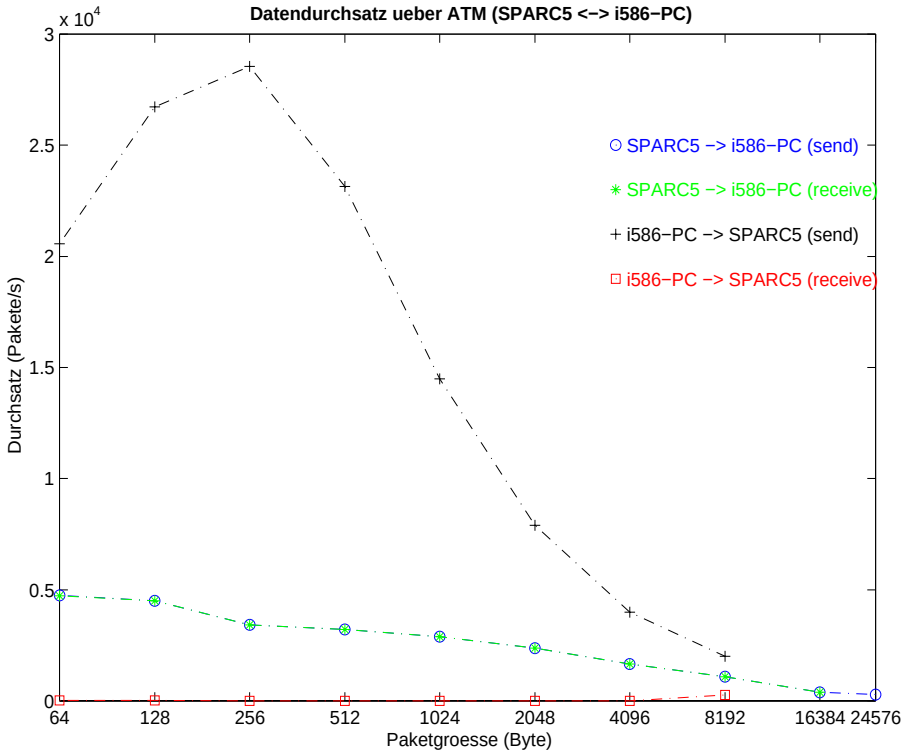


Abbildung C.3 - Datendurchsatz über ATM zwischen SPARC5 und i586-PC (Pakete/s)

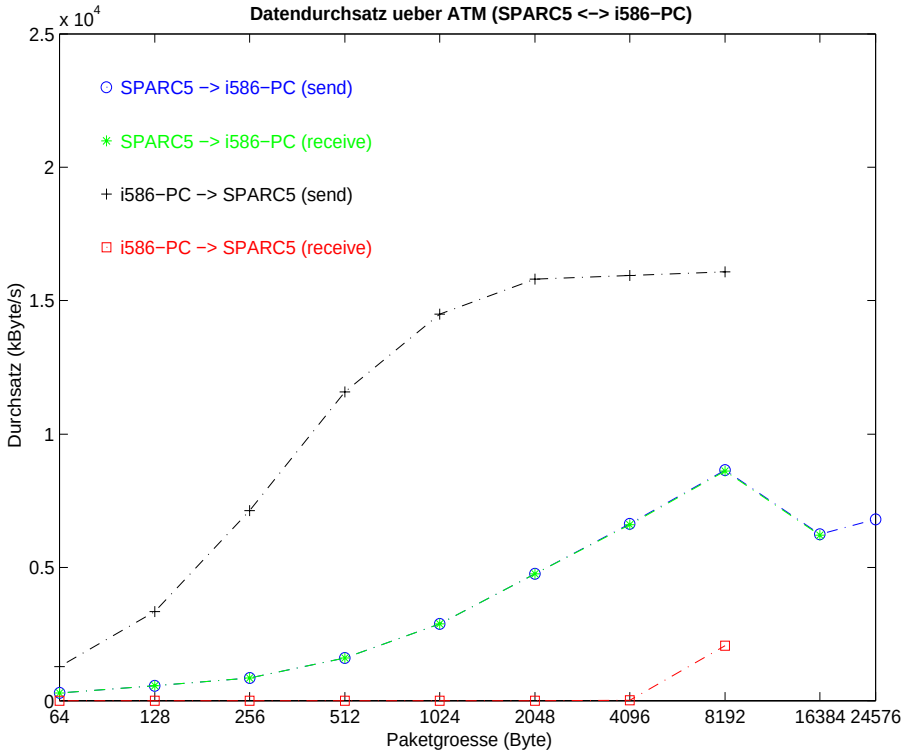


Abbildung C.4 - Datendurchsatz über ATM zwischen SPARC5 und i586-PC (kByte/s)

C.3 Messungen über ATM (i586-PC <-> ULTRA1)

Die nächsten Messungen wurden zwischen dem i586-PC und der ULTRA1, also den beiden “potentesten” Maschinen, durchgeführt.

Paketgröße	Sender (i586-PC)		Empfänger (ULTRA1)	
	Pakete/s	kByte/s	Pakete/s	kByte/s
64	20480.2	1279.7	19921.0	1244.6
128	26222.0	3277.4	11650.2	1455.8
256	27383.2	6845.4	6377.2	1593.8
512	24417.2	12208.4	4105.8	2052.8
1024	14525.0	14525.0	7809.8	7809.8
2048	7923.6	15847.6	7658.2	15316.8
4096	3946.4	15786.0	3832.5	15330.0

Tabelle C.6 - “push”-Messung über ATM von i586-PC zu ULTRA1

Paketgröße	Sender (i586-PC)		Empfänger (ULTRA1)	
	Pakete/s	kByte/s	Pakete/s	kByte/s
8192	2008.6	16071.0	1961.0	15690.5
16384	-	-	-	-

Tabelle C.6 - “push”-Messung über ATM von i586-PC zu ULTRA1

Bemerkung: Bei “wenigen” kleinen Paketen und auch bei großen Paketen mit hohem Durchsatz kommt die ULTRA1 beim Empfangen des vom i586-PC erzeugten Datenstromes hinterher. Im Mittelfeld kommt es zu Einbusen.

Paketgröße	Sender (ULTRA1)		Empfänger (i586-PC)	
	Pakete/s	kByte/s	Pakete/s	kByte/s
64	19194.8	1199.5	12193.8	761.8
128	19113.2	2388.7	11923.0	1490.0
256	15791.7	3947.5	10104.6	2525.8
512	14686.0	7342.7	7644.0	3821.8
1024	13859.7	13859.7	3923.0	3923.0
2048	8133.8	16268.3	4626.8	9253.8
4096	4080.4	16324.0	3130.2	12523.2
8192	2020.5	16166.7	1633.0	13066.4
16384	1005.2	16089.2	859.2	13752.6

Tabelle C.7 - “push”-Messung über ATM von ULTRA1 zu i586-PC

Bemerkung: Der i586-PC verhält sich ähnlich wie die ULTRA1 beim Empfangen von großen Datenraten. Die Schmerzgrenze liegt aber hier schon bei ca. 13 MByte/s (ca. 104 Mbit/s) im Gegensatz zur ULTRA1, die bis zu ca. 15 MByte/s (ca. 120 Mbit/s) empfangen kann.

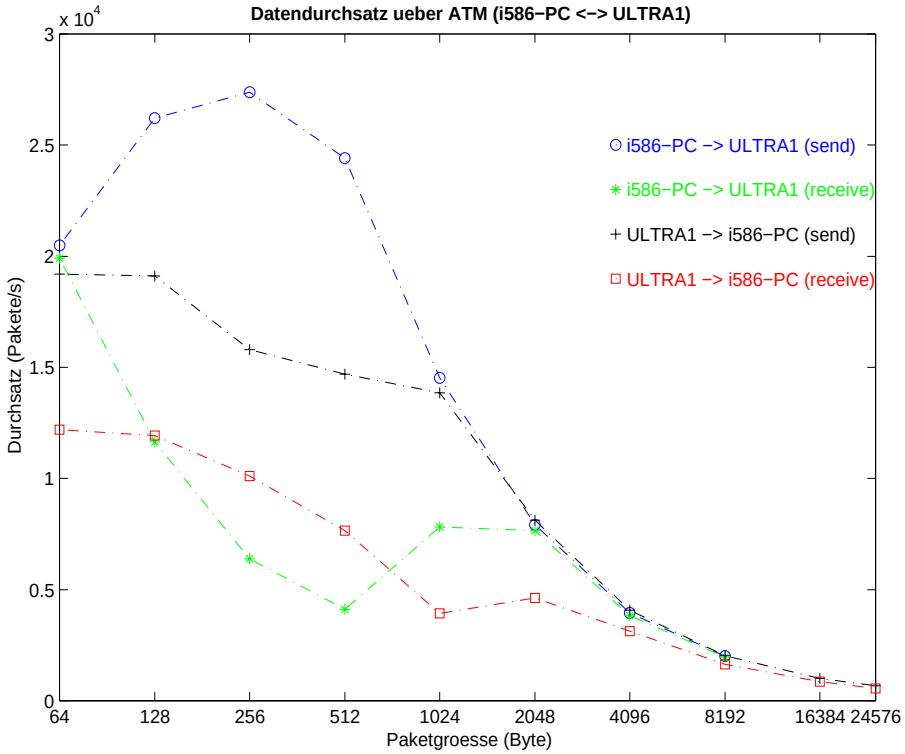


Abbildung C.5 - Datendurchsatz über ATM zwischen i586-PC und ULTRA1 (Pakete/s)

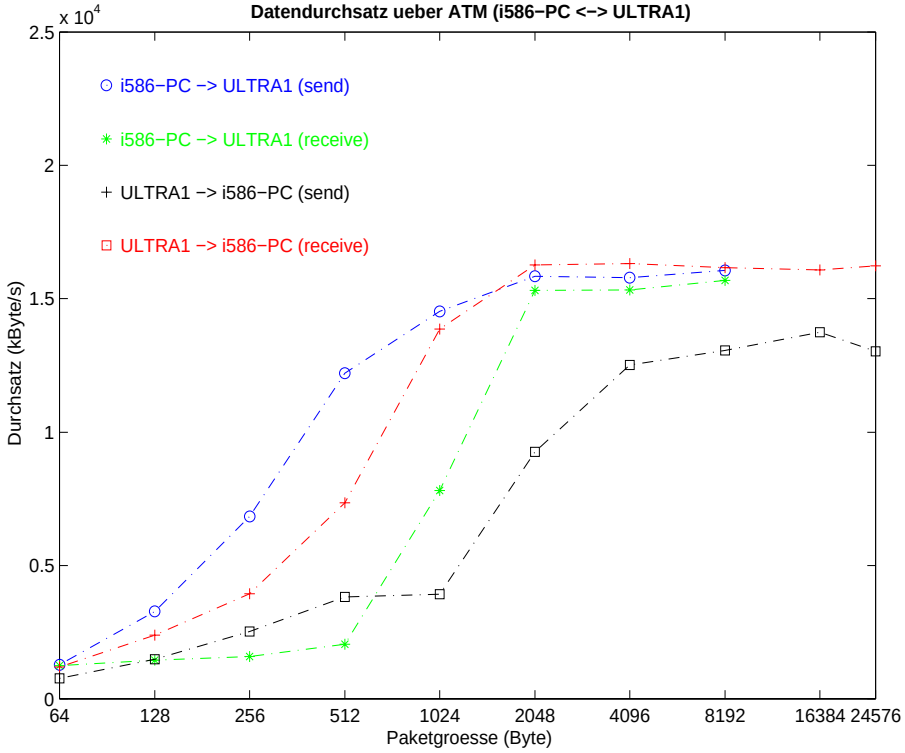


Abbildung C.6 - Datendurchsatz über ATM zwischen i586-PC und ULTRA1 (kByte/s)

C.4 Messungen über ATM (ULTRA1 <-> SPARC5)

Abschießend wurden noch Messung zwischen den beiden Suns, der ULTRA1 und der SPARC5 vorgenommen.

Paketgröße	Sender (ULTRA1)		Empfänger (SPARC5)	
	Pakete/s	kByte/s	Pakete/s	kByte/s
64	19525.8	1220.0	50.5	2.8
128	18212.7	2276.0	21.8	2.4
256	17140.5	4285.0	2.7	0.3
512	14971.8	7485.7	7.7	3.5
1024	14345.0	14345.0	1.7	1.7
2048	8177.2	16354.8	0.0	1.0
4096	4092.5	16371.3	0.8	4.0
8192	2058.2	16468.8	226.0	1811.7
16384	1005.4	16096.0	88.8	1428.3

Tabelle C.8 - “push”-Messung über ATM von ULTRA1 zu SPARC5

Bemerkung: Auch die ULTRA1 überflutet die SPARC5 mit UDP-Paketen, so daß diese keinerlei Reaktion mehr am Netzinterface zeigt. Erst ab einer Paketgröße von 8192 Byte können aufgrund des geringer werdenden Paketstromes wieder (ein paar) Daten empfangen werden.

Paketgröße	Sender (SPARC5)		Empfänger (ULTRA1)	
	Pakete/s	kByte/s	Pakete/s	kByte/s
64	4788.3	299.2	4785.8	298.6
128	4576.0	571.7	4578.4	571.8
256	3456.3	863.7	3454.8	863.4
512	3272.5	1636.0	3268.4	1634.0
1024	2938.8	2938.8	2934.0	2934.0
2048	2357.2	4714.8	2349.2	4698.8
4096	1640.3	6562.2	1628.0	6513.2
8192	1134.2	9076.7	1121.4	8974.2

Paketgröße	Sender (SPARC5)		Empfänger (ULTRA1)	
	Pakete/s	kByte/s	Pakete/s	kByte/s
16384	386.0	6185.0	384.2	6150.8

Tabelle C.9 - push Messung über ATM von SPARC5 zu ULTRA1

Bemerkung: SPARC5 sendet mit maximaler Leistung, die ULTRA1 hat - wie auch schon der i586-PC - keine Probleme die ankommenden Daten korrekt zu empfangen.

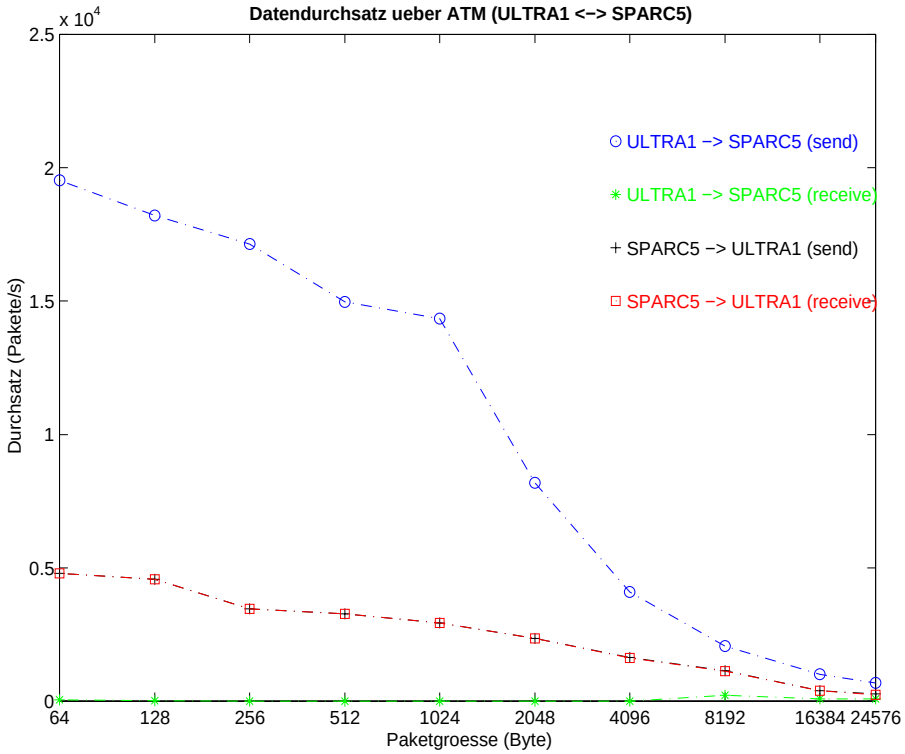


Abbildung C.7 - Datendurchsatz über ATM zwischen ULTRA1 und SPARC5 (Pakete/s)

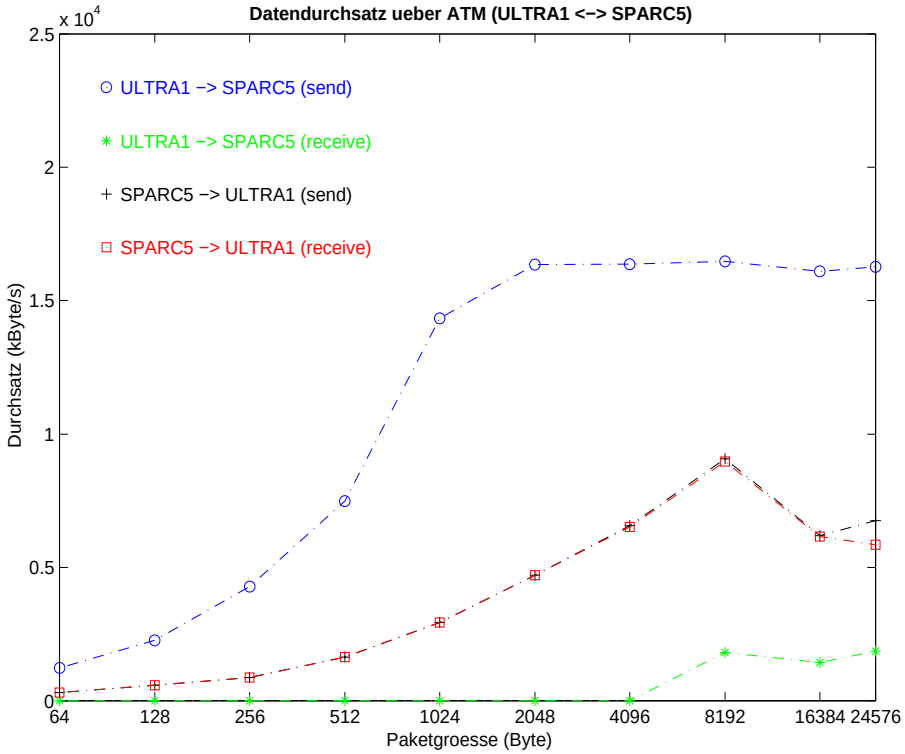


Abbildung C.8 - Datendurchsatz über ATM zwischen ULTRA1 und SPARC5 (kByte/s)

D Protokoll der “atmpush”-Messungen

Die folgenden Ergebnisse stammen von Messungen mittels “atmpush”, welches AAL5-PDUs über das API der Sun-ATM-Karte direkt über die Streams-Umgebung verschickt. Es wurden sowohl Messungen von der SPARC5 zur ULTRA1 vorgenommen, als auch in der Gegenrichtung.

Paketgröße	Sender (SPARC5)		Empfänger (ULTRA1)	
	Pakete/s	kByte/s	Pakete/s	kByte/s
10	7285	72.85	7264	72.64
64	7249	463.93	7248	463.91
128	6948	889.40	6947	889.32
256	6603	1690.49	6603	1690.36
512	4386	2245.77	4385	2245.48
1024	3941	4036.50	3940	4035.42
2048	3114	6378.51	3110	6370.00
4096	2153	8818.08	2148	8797.80
8192	1392	11403.10	1380	11307.54

Tabelle D.1 - “atmpush”-Messung von SPARC5 zu ULTRA1

Bemerkung: Die ULTRA1 hat keine Probleme die von der SPARC5 mit maximale möglicher Last gesendeten AAL5-PDUs vollständig zu empfangen. Offensichtlich ist es aber für die SPARC5 nicht möglich, die volle Netzkapazität von (netto) 134 MBit/s auszunutzen.

Paketgröße	Sender (ULTRA1)		Empfänger (SPARC5)	
	Pakete/s	kByte/s	Pakete/s	kByte/s
10	38117	381.17	108	1.08
64	37586	2405.54	126	8.08
128	34425	4406.48	245	31.37
256	31237	7996.75	213	54.19
512	21272	10891.42	323	165.34
1024	15284	15651.42	552	566.04
2048	7912	16198.96	1457	2985.72

Tabelle D.2 - “atmpush”-Messung von ULTRA1 zu SPARC5

Paketgröße	Sender (ULTRA1)		Empfänger (SPARC5)	
	Pakete/s	kByte/s	Pakete/s	kByte/s
4096	4010	16411.96	1602	6563.86
8192	2036	16666.82	1353	11089.73

Tabelle D.2 - “atmpush”-Messung von ULTRA1 zu SPARC5

Bemerkung: Die ULTRA1 schafft es ab einer Paketgröße von ca. 1024 Byte, die volle Kapazität des ATM-Netzes zu nutzen. Die CPU-Last ist ab dieser Paketgröße vernachlässigbar. Leider kann die SPARC5 nur Bruchteile der verschickten AAL5-PDUs empfangen.

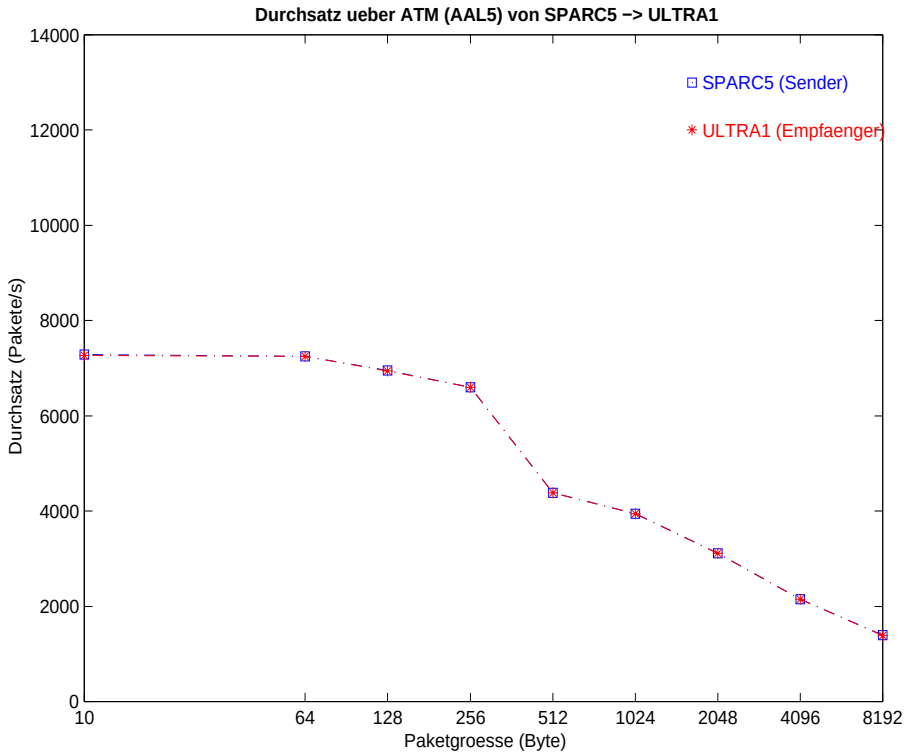


Abbildung D.1 - Datendurchsatz über ATM (AAL5) von SPARC5 zu ULTRA1 (Pakete/s)

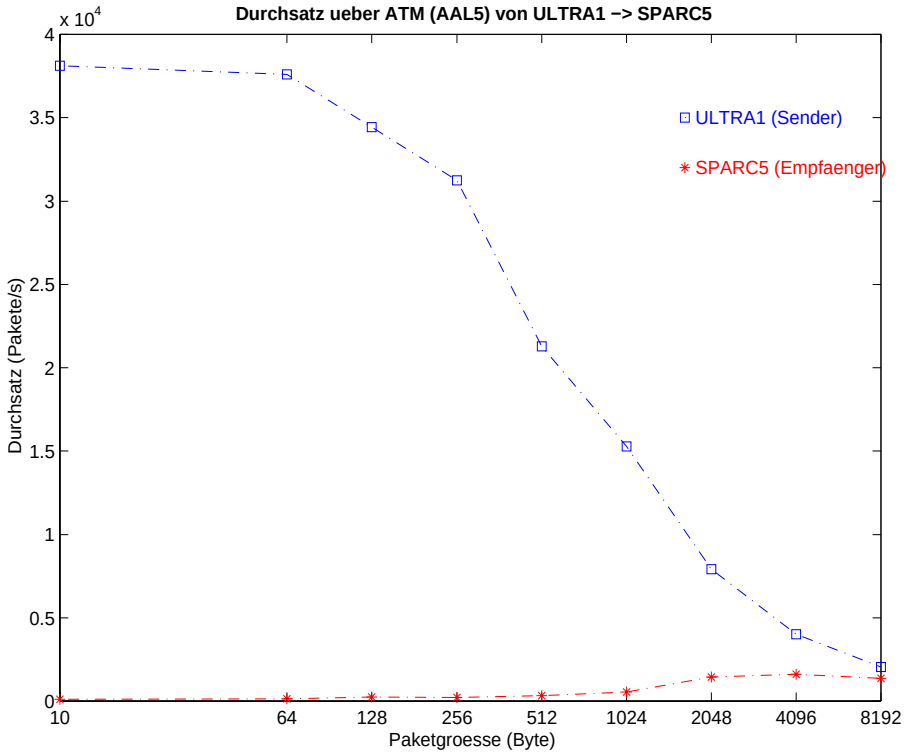


Abbildung D.2 - Datendurchsatz über ATM (AAL5) von ULTRA1 zu SPARC5 (Pakete/s)

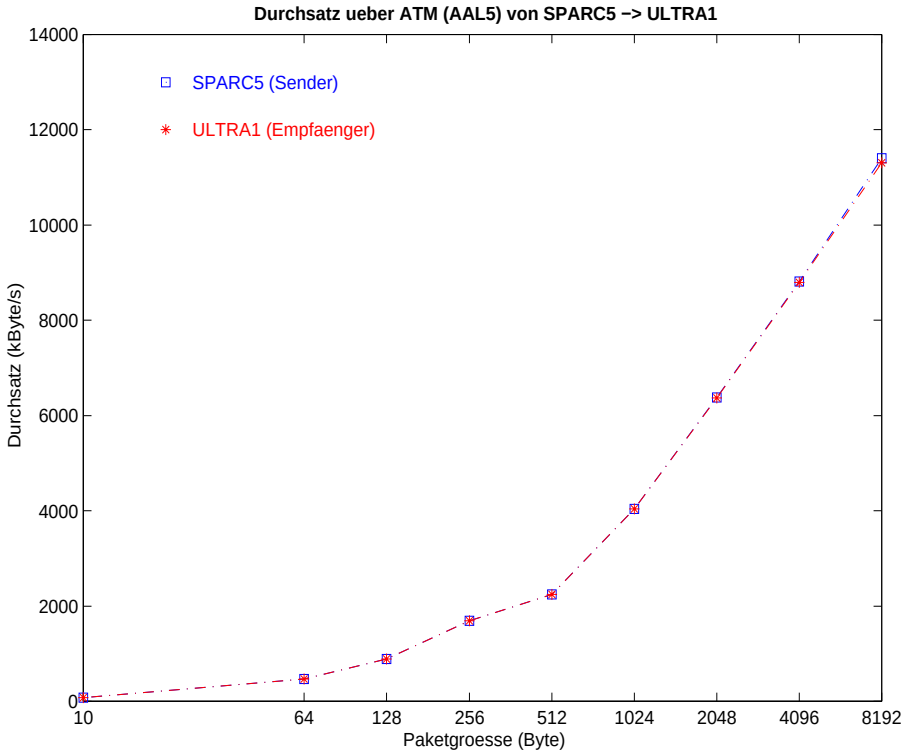


Abbildung D.3 - Datendurchsatz über ATM (AAL5) von SPARC5 zu ULTRA1 (kByte/s)

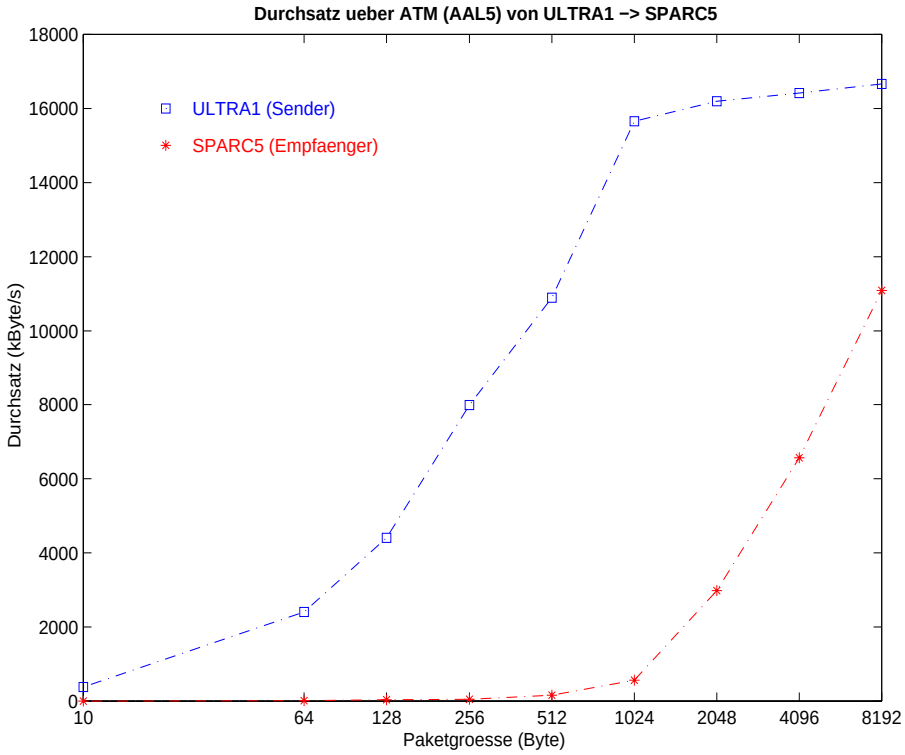


Abbildung D.4 - Datendurchsatz über ATM (AAL5) von ULTRA1 zu SPARC5 (kByte/s)

E Ausgewählte Programmteile

E.1 Kernelmodul ba (Solaris ATM-Treiber)

Datei: include/atm/types.h (Typdefinitionen)

```
...
#if ATMMONI_USE_SYSCALL == 1 || ATMMONI_USE_DIRECT == 1
typedef struct atmmoni_buffer {
#if ATMMONI_USE_SYSCALL == 1
     int filled;
#endif
#if ATMMONI_USE_DIRECT == 1
     short filled;
     short read;
#endif
     int count;
     u_char buffer[_ATMMONI_BUF_BUFFERSIZE];
     struct atmmoni_buffer *next;
} atmmoni_buffer_t;
#endif

typedef struct atmmoni_pdu {
     u_char unit;       /* Minor unit number */
     u_char type;       /* AAL type */
     u_char encap;      /* Encapsulation */
     u_char pad;        /* PAD */
     u_short vpi;       /* VPI */
     u_short vci;       /* VCI */
     hrtime_t time;     /* hrtime (64bit) */
     u_short cells;     /* # cells */
     u_short len;       /* PDU length */
     u_short hlen;      /* header length */
     u_short pad2;      /* PAD */
} atmmoni_pdu_t;

typedef struct atmmoni_stats {
     u_int init;
     u_int cells;
     u_int dropped_cells;
     u_int pdus;
     u_int dropped_pdus;
```

```
#if ATMMONI_USE_DIRECT == 1
     u_int blocks;
#endif
#if ATMMONI_USE_SYSCALL == 1
     u_int blocks;
     u_int free_blocks;
     u_int filled_blocks;
#endif
} atmmoni_stats_t;
...
```

Datei: drv/atm_common.c (globale Variablen und Systemaufrufe)

```
...
atmmoni_stats_t _atmmoni_stats;
u_int _atmmoni_initialized=0;
u_int _atmmoni_debug=0;

#if ATMMONI_USE_SYSCALL == 1 || ATMMONI_USE_DIRECT == 1
atmmoni_buffer_t *_atmmoni_first=0L;
atmmoni_buffer_t *_atmmoni_last=0L;
atmmoni_buffer_t *_atmmoni_curr=0L;
#if ATMMONI_USE_SYSCALL == 1
atmmoni_buffer_t *_atmmoni_nexttoget=0L;
#endif
#endif

#if ATMMONI_USE_MUTEX == 1
kmutex_t _atmmoni_mutex;
#endif

static void
atmmoni_init (queue_t *wq, mblk_t *mp)
{
#if ATMMONI_USE_SYSCALL == 1
     atmmoni_buffer_t *tmp;
#endif

#if ATMMONI_USE_DIRECT == 1
     if(mp->b_cont == NULL || MBLKL(mp->b_cont) != sizeof(caddr_t)) {
          miocnak(wq, mp, 0, EINVAL);
          return;
     }
#endif
```

```
     if(_atmmoni_initialized==1) {
          printf("ATMMONI: warning: atmmoni_init(): kernel driver is already
initialized.\n");
#if ATMMONI_USE_SYSCALL == 1 || ATMMONI_USE_READ == 1
          miocnak(wq, mp, 0, EINVAL);
#endif
#if ATMMONI_USE_DIRECT == 1
          *(caddr_t *)mp->b_cont->b_rptr=(caddr_t)&_atmmoni_stats;
          miocack(wq, mp, msgsize(mp->b_cont), 0);
#endif
          return;
     }

#if ATMMONI_USE_SYSCALL == 1
     tmp=(atmmoni_buffer_t *)kmem_alloc(sizeof(atmmoni_buffer_t),
          KM_NOSLEEP);
     if(!tmp) {
          printf("ATMMONI: panic: atmmoni_init(): cannot allocate first block
of buffer memory.\n");
          miocnak(wq, mp, 0, ENOMEM);
          return;
     }
#endif

#if ATMMONI_USE_MUTEX == 1
     mutex_init(&_atmmoni_mutex,"ATMMONI",MUTEX_DRIVER,NULL);
#endif

#if ATMMONI_USE_SYSCALL == 1
     tmp->filled=0;
     tmp->count=0;
     tmp->next=0L;
     _atmmoni_first=_atmmoni_last=_atmmoni_curr=tmp;
#endif

     _atmmoni_initialized=1;
     _atmmoni_stats.init=1;
     _atmmoni_stats.cells=0;
     _atmmoni_stats.dropped_cells=0;
     _atmmoni_stats.pdus=0;
     _atmmoni_stats.dropped_pdus=0;
#if ATMMONI_USE_SYSCALL == 1
     _atmmoni_stats.blocks=1;
```

```
     _atmmoni_stats.free_blocks=0;
     _atmmoni_stats.filled_blocks=0;
#endif
#if ATMMONI_USE_DIRECT == 1
     _atmmoni_stats.blocks=0;
#endif

     printf("ATMMONI: info: atmmoni_init(): kernel driver has been initiali-
zed successfully.\n");
#if ATMMONI_USE_SYSCALL == 1 || ATMMONI_USE_READ == 1
     miocack(wq, mp, 0, 0);
#endif
#if ATMMONI_USE_DIRECT == 1
     *(caddr_t *)mp->b_cont->b_rptr=(caddr_t)&_atmmoni_stats;
     miocack(wq, mp, msgsize(mp->b_cont), 0);
#endif
}

static void
atmmoni_setdebug (queue_t *wq, mblk_t *mp)
{
     if(mp->b_cont == NULL || MBLKL(mp->b_cont) != sizeof(int)) {
          miocnak(wq, mp, 0, EINVAL);
          return;
     }

     if(_atmmoni_initialized==0) {
          printf("ATMMONI: panic: kernel atmmoni_setdebug(): driver has not
yet been initialized.\n");
          miocnak(wq, mp, 0, EINVAL);
          return;
     }

#if ATMMONI_USE_MUTEX > 0
     mutex_enter(&_atmmoni_mutex);
#endif
     _atmmoni_debug=*(int *)mp->b_cont->b_rptr;
#if ATMMONI_USE_MUTEX > 0
     mutex_exit(&_atmmoni_mutex);
#endif

     printf("ATMMONI: info: atmmoni_setdebug(): set _atmoni_debug to %d.\n",
          _atmmoni_debug);
     miocack(wq, mp, 0, 0);
```

```
}

#if ATMMONI_USE_SYSCALL == 1 || ATMMONI_USE_DIRECT
static void
atmmoni_addmem (queue_t *wq, mblk_t *mp)
{
     atmmoni_buffer_t *tmp;
     int blocks;
     int i=0;

#if ATMMONI_USE_SYSCALL == 1
     if(mp->b_cont == NULL || MBLKL(mp->b_cont) != sizeof(int)) {
          miocnak(wq, mp, 0, EINVAL);
          return;
     }
#endif
#if ATMMONI_USE_DIRECT == 1
     if(mp->b_cont == NULL || MBLKL(mp->b_cont) != sizeof(caddr_t)) {
          miocnak(wq, mp, 0, EINVAL);
          return;
     }
#endif

     if(_atmmoni_initialized==0) {
          printf("ATMMONI: panic: atmmoni_addmem(): kernel driver has not yet
been initialized.\n");
          miocnak(wq, mp, 0, EINVAL);
          return;
     }



#if ATMMONI_USE_SYSCALL == 1
     blocks=*(int *)mp->b_cont->b_rptr;
#endif
#if ATMMONI_USE_DIRECT == 1
     blocks=*(int *)mp->b_cont->b_rptr;
#endif

     for(i=0;i<blocks;i++) {
          tmp=(atmmoni_buffer_t *)kmem_alloc(si-
zeof(atmmoni_buffer_t),KM_NOSLEEP);
          if(!tmp) {
               i--;
               break;
```

```
          }
          if(_atmmoni_first)
               printf("tmp: %x first: %x second: %x last: %x\n",
                    tmp,_atmmoni_first,_atmmoni_first->next,_atmmoni_last);
#if ATMMONI_USE_MUTEX > 0
          mutex_enter(&_atmmoni_mutex);
#endif
          if(_atmmoni_first) {
               _atmmoni_last->next=tmp;
          }
          else {
               _atmmoni_first=_atmmoni_curr=tmp;
          }
          _atmmoni_last=tmp;
          _atmmoni_last->filled=0;
#if ATMMONI_USE_DIRECT == 1
          _atmmoni_last->read=0;
#endif
          _atmmoni_last->count=0;
          _atmmoni_last->next=0L;
          _atmmoni_stats.blocks++;
#if ATMMONI_USE_SYSCALL == 1
          _atmmoni_stats.free_blocks++;
#endif
#if ATMMONI_USE_MUTEX > 0
          mutex_exit(&_atmmoni_mutex);
#endif
     }

#if ATMMONI_USE_SYSCALL == 1
     *(int *)mp->b_cont->b_rptr=i;
#endif
#if ATMMONI_USE_DIRECT == 1
     *(caddr_t *)mp->b_cont->b_rptr=(caddr_t)_atmmoni_first;
#endif

     printf("ATMMONI: info: atmmoni_addmem(): %sadded %s%d of %d block(s)
buffer memory.\n",
          i==blocks?"":"warning: ",i==blocks?"":"only ",i,blocks);
     miocack(wq, mp, msgsize(mp->b_cont), 0);
}
#endif

#if ATMMONI_USE_SYSCALL == 1 || ATMMONI_USE_READ == 1
```

```
static void
atmmoni_getstats (queue_t *wq, mblk_t *mp)
{
     if(mp->b_cont == NULL || MBLKL(mp->b_cont) != sizeof(atmmoni_stats_t)) {
          miocnak(wq, mp, 0, EINVAL);
          return;
     }

#if ATMMONI_USE_MUTEX > 0
     mutex_enter(&_atmmoni_mutex);
#endif
     _atmmoni_stats.init=_atmmoni_initialized;
     bcopy(&_atmmoni_stats,mp->b_cont->b_rptr,sizeof(atmmoni_stats_t));
#if ATMMONI_USE_MUTEX > 0
     mutex_exit(&_atmmoni_mutex);
#endif

     if(_atmmoni_debug>2)
          printf("ATMMONI: debug: atmmoni_getstats(): statistics: init: %d
cells: %d dropped_cells: %d pdus: %d dropped_pdus: %d\n",
               _atmmoni_stats.init,_atmmoni_stats.cells,
               _atmmoni_stats.dropped_cells,_atmmoni_stats.pdus,
               _atmmoni_stats.dropped_pdus);
     miocack(wq, mp, msgsize(mp->b_cont), 0);
}
#endif

#if ATMMONI_USE_SYSCALL == 1
static void
atmmoni_getblock (queue_t *wq, mblk_t *mp)
{
     atmmoni_buffer_t *buffer;
     u_char *pos;
     mblk_t *act;
     int size,rest;

     size=0;
     act=mp->b_cont;
     while(act) {
          size+=MBLKL(act);
          act=act->b_cont;
     }

     if(_atmmoni_initialized==0) {
```

```
          printf(“ATMMONI: panic: atmmoni_getblock(): kernel driver has not
yet been initialized.\n”);
          miocnak(wq, mp, 0, EINVAL);
          return;
     }

     if(size!=_ATMMONI_BUFFERSIZE) {
          printf(“ATMMONI: panic: atmmoni_getblock(): bad data size: %d (ex-
pected: %d).\n”,
               size,_ATMMONI_BUFFERSIZE);
          miocnak(wq, mp, 0, EINVAL);
          return;
     }

#if ATMMONI_USE_MUTEX > 0
     mutex_enter(&_atmmoni_mutex);
#endif
     if(_atmmoni_stats.filled_blocks) {
          pos=_atmmoni_nexttoget->buffer-sizeof(int);
          act=mp->b_cont;
          for(rest=_ATMMONI_BUFFERSIZE;rest>0;) {
               size=MBLKL(act)<rest?MBLKL(act):rest;
               bcopy(pos,act->b_rptr,size);
               pos+=size;
               rest-=size;
               act=act->b_cont;
          }

          _atmmoni_nexttoget->filled=0;
          _atmmoni_nexttoget->count=0;

          if(--_atmmoni_stats.filled_blocks>0) {
               if(_atmmoni_nexttoget->next) {
                    _atmmoni_nexttoget=_atmmoni_nexttoget->next;
               }
               else {
                    _atmmoni_nexttoget=_atmmoni_first;
               }
          }
          else {
               _atmmoni_nexttoget=0L;
          }

          _atmmoni_stats.free_blocks++;
```

```
          if(_atmmoni_debug>5)
               printf("ATMMONI: debug: atmmoni_getblock(): transmitted 1
block of buffer memory.\n");

#if ATMMONI_USE_MUTEX > 0
          mutex_exit(&_atmmoni_mutex);
#endif
          miocack(wq, mp, msgsize(mp->b_cont), 0);
     }
     else {
          if(_atmmoni_debug>3)
               printf("ATMMONI: warning: atmmoni_getblock(): there is no fil-
led buffer to transmit.\n");

#if ATMMONI_USE_MUTEX > 0
          mutex_exit(&_atmmoni_mutex);
#endif
          miocack(wq, mp, 0, 0);
     }
}
#endif
...
```

Datei: drv/ba_drv.c (Hauptteil des Treibers mit Interruptverarbeitung und Streammangement)

```
...
static int
ba_read(register atm_t *sahip, register rxc_t *rxc_cp, unsigned int headp)
{
...
     if(_atmmoni_initialized) {
          atmmoni_pdu_t atmmoni_pdu;
          hrtime_t atmmoni_hrtime;
#if ATMMONI_USE_READ == 1
          mblk_t *atmmoni_mblk;
#endif

#if ATMMONI_USE_MUTEX == 1
          mutex_enter(&_atmmoni_mutex);
#endif

#if ATMMONI_USE_READ == 1
          slp = vp->slp;
```

```
          if (!(canput(slp->sl_rq->q_next))) {
               goto _atmmoni_nobuffer;
          }
#endif

#if ATMMONI_USE_SYSCALL == 1 || ATMMONI_USE_DIRECT == 1
          if(_atmmoni_curr->filled ||
               (_ATMMONI_BUF_BUFFERSIZE-_atmmoni_curr->count)<
                    (sizeof(atmmoni_pdu_t)+len)) {
               if(!_atmmoni_curr->filled) {
                    _atmmoni_curr->filled=1;
#if ATMMONI_USE_SYSCALL == 1
                    _atmmoni_stats.filled_blocks++;
#endif
               }
#if ATMMONI_USE_SYSCALL == 1
               if(!_atmmoni_nexttoget) _atmmoni_nexttoget=_atmmoni_curr;

               if(_atmmoni_curr->next)
                    if(!_atmmoni_curr->next->filled)
                         _atmmoni_curr=_atmmoni_curr->next;
                    else
                         if(!_atmmoni_first->filled)
                              _atmmoni_curr=_atmmoni_first;

               if(_atmmoni_curr->filled) goto _atmmoni_no_more_free_blocks;
#endif

#if ATMMONI_USE_DIRECT == 1
               if(_atmmoni_curr->next) {
                    if(!_atmmoni_curr->next->filled ||
                         _atmmoni_curr->next->read)
                         _atmmoni_curr=_atmmoni_curr->next;
               }
               else {
                    if(!_atmmoni_first->filled || _atmmoni_first->read)
                         _atmmoni_curr=_atmmoni_first;
               }

               if(_atmmoni_curr->read) {
                    _atmmoni_curr->filled=0;
                    _atmmoni_curr->read=0;
                    _atmmoni_curr->count=0;
               }
```

```
              if(_atmmoni_curr->filled) goto _atmmoni_no_more_free_blocks;
              goto _atmmoni_direct_continue;
#endif

#if ATMMONI_USE_SYSCALL == 1
              if(_atmmoni_stats.free_blocks>0) {
                   _atmmoni_stats.free_blocks--;
              }
              else {
#endif
_atmmoni_no_more_free_blocks:
                   if(_atmmoni_debug>2)
                        printf("ATMMONI: debug: ba_read(): no more free me-
mory blocks.\n");

                   goto _atmmoni_nobuffer;
#if ATMMONI_USE_SYSCALL == 1
              }
#endif
         }

#if ATMMONI_USE_DIRECT == 1
_atmmoni_direct_continue:
#endif
#endif

#if ATMMONI_USE_MUTEX > 0
         mutex_exit(&_atmmoni_mutex);
#endif

         atmmoni_pdu.unit=sahip->unit;
         atmmoni_pdu.type=(vpflags&RXV_AAL5)?_ATMMONI_AAL5:(vpflags &
              RXV_AAL34)?_ATMMONI_AAL34:_ATMMONI_AAL0;
         atmmoni_pdu.encap=(vpflags&RXV_NULLENC)?_ATMMONI_ENCNULL:
              (vpflags&RXV_LLCENC)?_ATMMONI_ENCLLC:_ATMMONI_ENCNLPID;
         atmmoni_pdu.pad=0;
         atmmoni_pdu.vpi=(vpci>>16)&0xff;
         atmmoni_pdu.vci=vpci&0xffff;
         atmmoni_hrtime=gethrtime();
         /* atmmoni_hrtime=0x0102030405060708; */
         atmmoni_pdu.time=atmmoni_hrtime;
         atmmoni_pdu.cells=cells;
         atmmoni_pdu.len=len;
```

```
          atmmoni_pdu.hlen=hlen;
          atmmoni_pdu.pad2=0;

#if ATMMONI_USE_SYSCALL == 1 || ATMMONI_USE_DIRECT == 1
          bcopy((caddr_t) &atmmoni_pdu,
               (caddr_t) &_atmmoni_curr->buffer[_atmmoni_curr->count],
               sizeof(atmmoni_pdu_t));
          _atmmoni_curr->count+=sizeof(atmmoni_pdu_t);

...
```

Im Anschluß werden jetzt die Daten je nach Methode entweder in einen freien Puffer geschrieben oder zum Stream-Head weitergereicht.

E.2 Userlevelprozeß der Aufzeichnung - “test_logger”

Datei: sunatm.c (Interface- und Treibersteuerung)

```
...
/* Statistiken holen */
int
atmmoni_getstats(stats,out)
atmmoni_stats_t *stats;
int out;
{
#if ATMMONI_USE_SYSCALL == 1 || ATMMONI_USE_READ == 1
     struct strioctl strioctl;

     strioctl.ic_cmd=A_ATMMONI_GETSTATS;
     strioctl.ic_timout=-1;
     strioctl.ic_len=sizeof(atmmoni_stats_t);
     strioctl.ic_dp=(caddr_t)stats;
     if(ioctl(atmfd,I_STR,&strioctl)<0) {
          perror("ioctl(2) A_MONIBOX_GETSTATS failed");
          return -1;
     }
     if(strioctl.ic_len<sizeof(atmmoni_stats_t)) {
          fprintf(stderr,"atmmoni_getstats(): I got currupted statistic in-
formations (len: %d) ... exiting!\n",
               strioctl.ic_len);
          return -2;
     }
#endif
```

```
     if(out || DEBUG>=DEBUG_MAX) {
          fprintf(stderr,"atmmoni_getstats(): ");
          fprintf(stderr,"init: %d cells: %d dropped_cells: %d pdus: %d
dropped_pdus: %d\n",
               stats->init,stats->cells,stats->dropped_cells,stats->pdus,
               stats->dropped_pdus);
     }

     return 0;
}

/* Block holen */
int
atmmoni_getblock(buffer,size)
u_char *buffer;
int size;
{
#if ATMMONI_USE_READ == 1
     struct strbuf ctl;
     struct strbuf data;
     int flags;
     char ctlbuf[256];

     fd_set fdset;

     ctl.buf = ctlbuf;
     ctl.maxlen = 256;
     ctl.len = 0;
     data.buf = (char *)buffer;
     data.maxlen = size;
     data.len = 0;

     flags = 0;

     FD_ZERO(&fdset);
     FD_SET(atmfd,&fdset);
     FD_SET(atmfd2,&fdset);
     if(select(atmfd+atmfd2,&fdset,NULL,NULL,NULL)<0) {
          perror("select(2) failed");
          return -2;
     }
     if(FD_ISSET(atmfd,&fdset))
          if(getmsg(atmfd,&ctl,&data,&flags)<0) {
               perror("getmsg(2) failed");
```

```
                return -1;
           }
     if(FD_ISSET(atmfd2,&fdset))
           if(getmsg(atmfd2,&ctl,&data,&flags)<0) {
                perror("getmsg(2) failed");
                return -1;
           }

     if(DEBUG>=DEBUG_MAX) fprintf(stderr,"Well, I got %d data bytes.\n",da-
ta.len);

     return data.len;
#endif

#if ATMMONI_USE_SYSCALL == 1
     struct strioctl strioctl;

     strioctl.ic_cmd=A_ATMMONI_GETBLOCK;
     strioctl.ic_timout=-1;
     strioctl.ic_len=size;
     strioctl.ic_dp=(caddr_t)buffer;
     if(ioctl(atmfd,I_STR,&strioctl)<0) {
          perror("ioctl(2) A_MONIBOX_GETBLOCK failed");
          return -1;
     }

     if(DEBUG>=DEBUG_MAX) {
          if(strioctl.ic_len==0)
               fprintf(stderr,"Well, there was no memory block.\n");
          else
               fprintf(stderr,"Well, I got a memory block (len: %d):\n",
                    *(int *)buffer);
     }

     return *(int *)buffer;
#endif

#if ATMMONI_USE_DIRECT == 1
     if(buffers[act_buffer]->filled == 0 || buffers[act_buffer]->read == 1)
          return 0;
     return buffers[act_buffer]->count;
#endif
}
```

```
#if ATMMONI_USE_DIRECT == 1
/* Kernel memory mappen */
caddr_t
kvm_map(kp)
caddr_t kp;
{
     int i;
     int pid;
     int file;
     caddr_t kvm_p;
     off_t page;
     off_t offset;
     caddr_t base;

     page =  ((u_int)kp/0x2000)*0x2000;
     offset = (u_int)kp - (u_int) page;
     if ((file = open("/dev/kmem", O_RDWR)) == -1)
          return NULL;

     fprintf(stderr,"mapping: kp: %x page: %x offset: %x\n",kp,page,offset);

     if((base = (caddr_t) mmap(NULL, 0x10000+offset, PROT_READ|PROT_WRITE,
          MAP_SHARED, file, page))==MAP_FAILED) {
          close(file);
          perror("mmap() failed");
          fprintf(stderr,"kp: %x page: %x offset: %x\n",kp,page,offset);
          return NULL;
     }
     close(file);
     kvm_p = (caddr_t) ((u_int) base + offset);
     return kvm_p;
}
#endif
...
```

F Protokoll der “test_logger”-Messungen

Die folgenden Messungen wurden unter Solaris mit dem für ATM-Monitoring modifizierten Kernelmodul “ba” und einem für Tests geschriebenen Userlevelprogramm “test_logger” durchgeführt. Zu beachten ist, daß die Menge der gesendeten Daten von der der empfangenen abweicht, da zu den Nutzdaten auch noch Kanalinformation mitgeloggt wird. Alle Messungen wurden mehrfach durchgeführt und die Ergebnisse als Mittelwerte in die Tabellen aufgenommen.

Anzumerken ist zu den Ergebnissen, daß der Logger immer etwas Anlaufzeit braucht, bevor ein stabiler, gleichmäßiger Zustand erreicht wird. D.h. am Anfang gehen **immer** einige Pakete verloren!

.

Methode (siehe Kapitel 4.2)	**Pufferplätze**	**Paketgröße**	**CPU-Last (ca., %)**	**Datenstrom (MByte/s)**	**Empfangsleistung (MByte/s)**	**Verlustrate (%)**
SYSCALL	10 à 64k	64	100	1.24	0	100
		128	100	2.14	0	100
		256	100	3.47	0	100
		512	100	5.68	0.48	95
		1024	100	8.73	1.16	91
		2048	100	11.93	1.79	87
		4096	100	13.50	2.85	81
		8192	100	14.58	3.23	79
SYSCALL	50 à 64k	64	100	1.15	0	100
		128	100	2.19	0	100
		256	100	3.57	0	100
		512	100	5.72	0.39	95
		1024	100	8.63	1.18	92
		2048	100	12.08	1.48	89
		4096	100	13.38	2.64	82
		8192	100	14.51	3.11	79

Tabelle F.1 - “test_logger”-Messung mit einer SPARC5

Methode (siehe Kapitel 4.2)	Pufferplätze	Paketgröße	CPU-Last (ca., %)	Datenstrom (MByte/s)	Empfangsleistung (MByte/s)	Verlustrate (%)
READ	-	64	100	1.15	0	100
		128	100	2.18	0	100
		256	100	3.66	0.02	99
		512	100	5.59	0.14	98
		1024	100	9.32	0.37	96
		2048	100	11.83	1.23	90
		4096	100	13.55	4.21	72
		8192	100	14.56	6.99	55
DIRECT	10 à 64k	64	100	1.15	0.31	74
		128	100	2.08	0.32	83
		256	100	3.55	0.50	86
		512	100	5.62	1.82	69
		1024	100	8.53	5.18	42
		2048	100	11.81	8.75	30
		4096	100	13.43	12.35	15
		8192	95	14.78	14.61	5
DIRECT	50 à 64k	64	100	1.17	0.84	27
		128	100	2.08	1.06	42
		256	100	3.46	1.69	47
		512	100	5.79	3.58	38
		1024	100	8.70	5.73	33
		2048	100	11.89	9.61	25
		4096	95	13.09	12.45	8
		8192	90	14.50	14.76	2

Tabelle F.1 - “test_logger”-Messung mit einer SPARC5

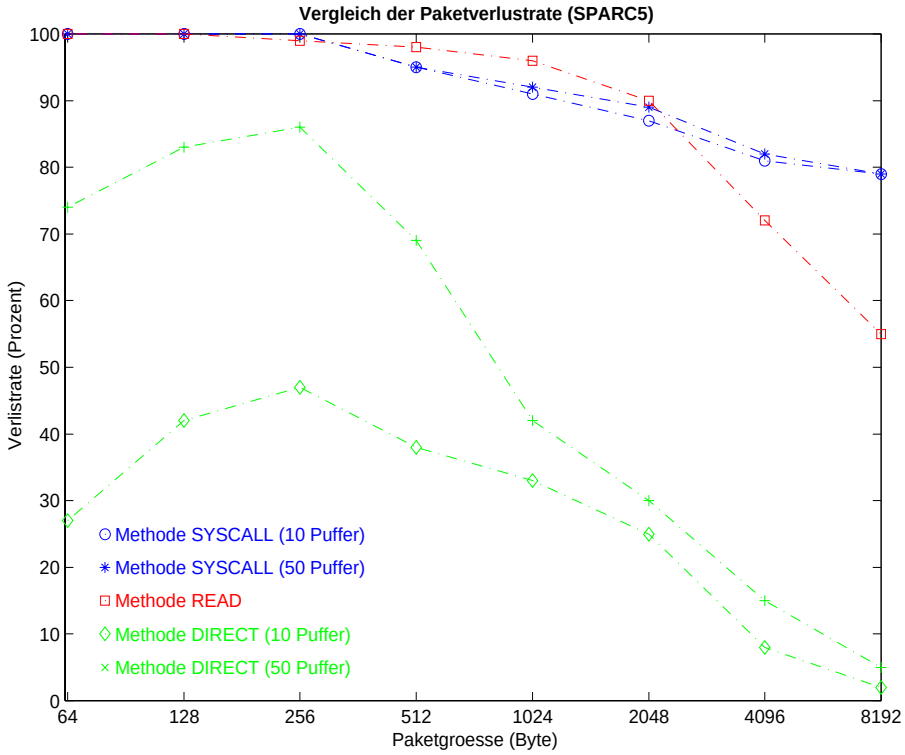


Abbildung F.1 - Vergleich der Paketverlustrate auf der SPARC5

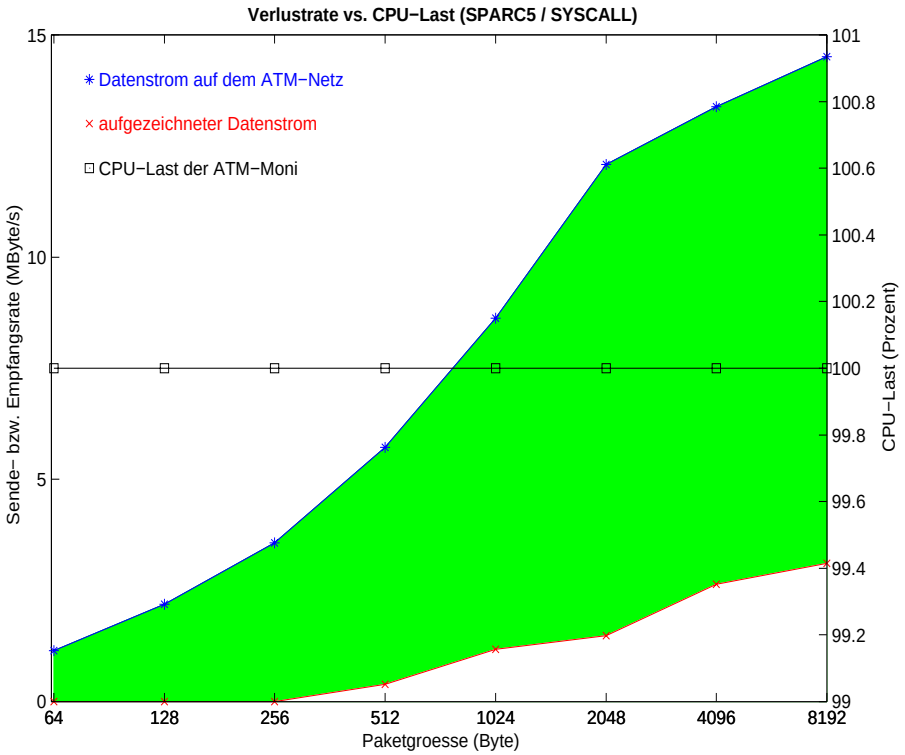


Abbildung F.2 - Verlustrate vs. CPU-Last (SPARC5 / SYSCALL)

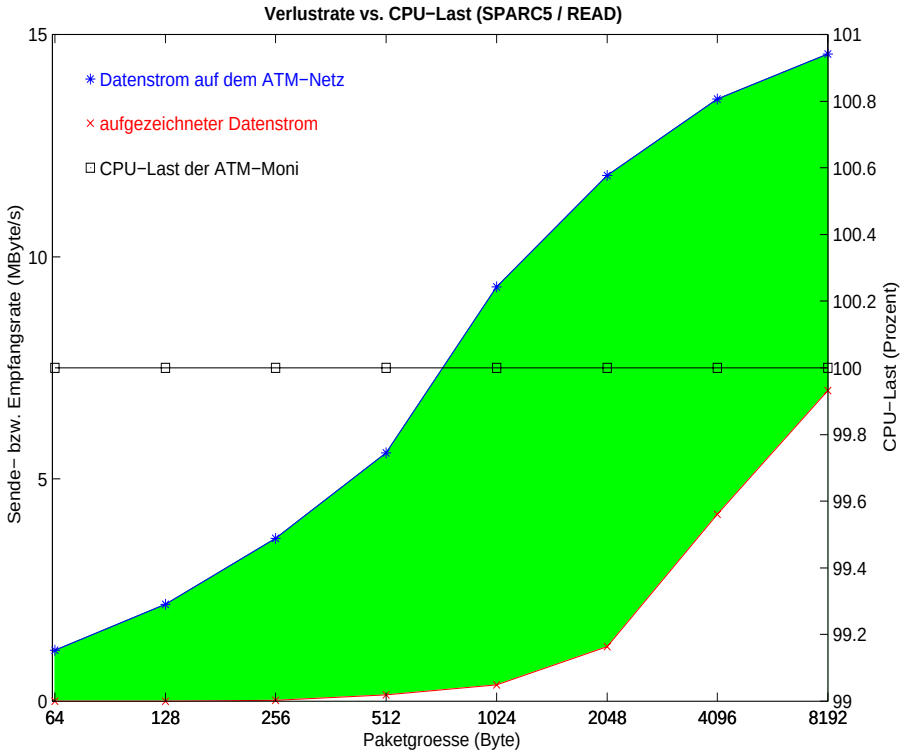


Abbildung F.3 - Verlustrate vs. CPU-Last (SPARC5 / READ)

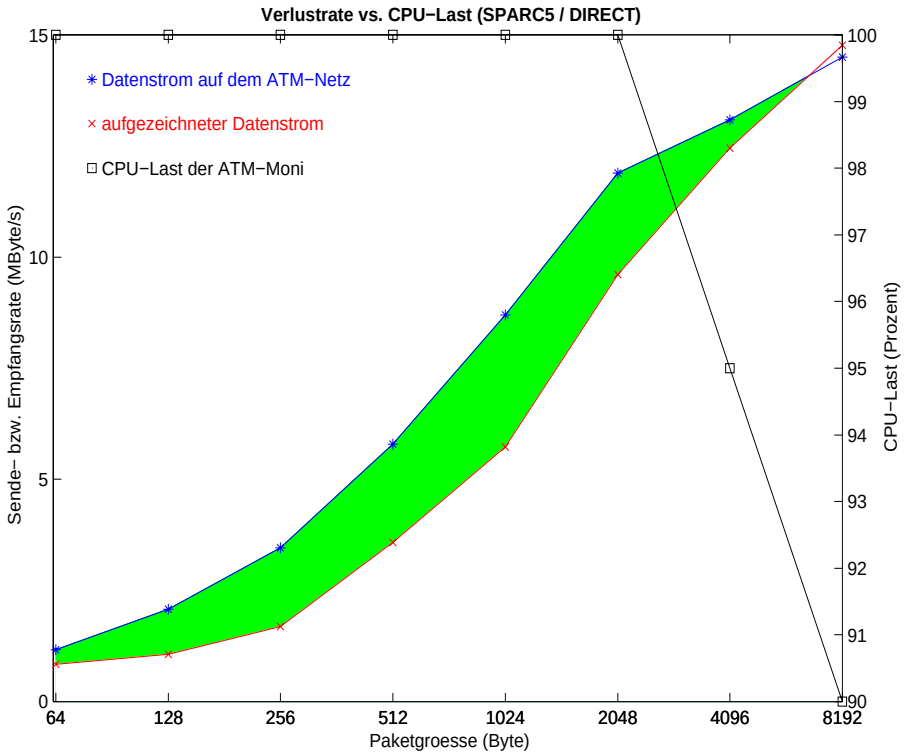


Abbildung F.4 - Verlustrate vs. CPU-Last (SPARC5 / DIRECT)

.

Methode (siehe Kapitel 4.2)	Pufferplätze	Paketgröße	CPU-Last (ca., %)	Datenstrom (MByte/s)	Empfangsleistung (MByte/s)	Verlustrate (%)
SYSCALL	10 à 64k	64	15	1.27	1.64	17
		128	10	2.23	2.18	24
		256	10	3.81	3.27	25
		512	10	6.27	6.61	6
		1024	5	9.53	6.54	35
		2048	5	13.46	6.45	54
		4096	5	14.76	6.19	59
		8192	1	16.24	5.76	65
SYSCALL	50 à 64k	64	15	1.31	1.64	18
		128	10	2.31	2.18	26
		256	10	3.91	3.20	27
		512	10	6.30	6.43	6
		1024	5	9.79	6.54	37
		2048	5	13.42	6.45	53
		4096	5	15.24	6.19	60
		8192	1	16.24	5.76	65
READ	-	64	100	1.29	1.18	47
		128	100	2.34	1.87	43
		256	100	3.84	3.39	29
		512	100	6.30	6.29	13
		1024	90	9.40	10.58	0
		2048	80	13.41	14.63	0
		4096	40	14.73	15.64	0
		8192	30	16.14	16.80	0

Tabelle F.2 - “test_logger”-Messung mit einer ULTRA1

Methode (siehe Kapitel 4.2)	Pufferplätze	Paketgröße	CPU-Last (ca., %)	Datenstrom (MByte/s)	Empfangsleistung (MByte/s)	Verlustrate (%)
DIRECT	10 à 64k	64	30	1.32	2.05	0
		128	30	2.31	2.95	0
		256	30	3.94	4.55	0
		512	20	6.25	6.77	0
		1024	20	9.52	10.01	0
		2048	20	13.38	13.96	0
		4096	10	15.34	15.88	0
		8192	10	16.24	16.74	0
DIRECT	50 à 64k	64	20	1.31	197	0
		128	20	2.33	2.98	0
		256	20	3.98	4.57	0
		512	20	6.35	6.92	0
		1024	10	9.79	10.30	0
		2048	10	13.41	14.02	0
		4096	10	15.24	15.80	0
		8192	5	16.24	17.74	0

Tabelle F.2 - “test_logger”-Messung mit einer ULTRA1

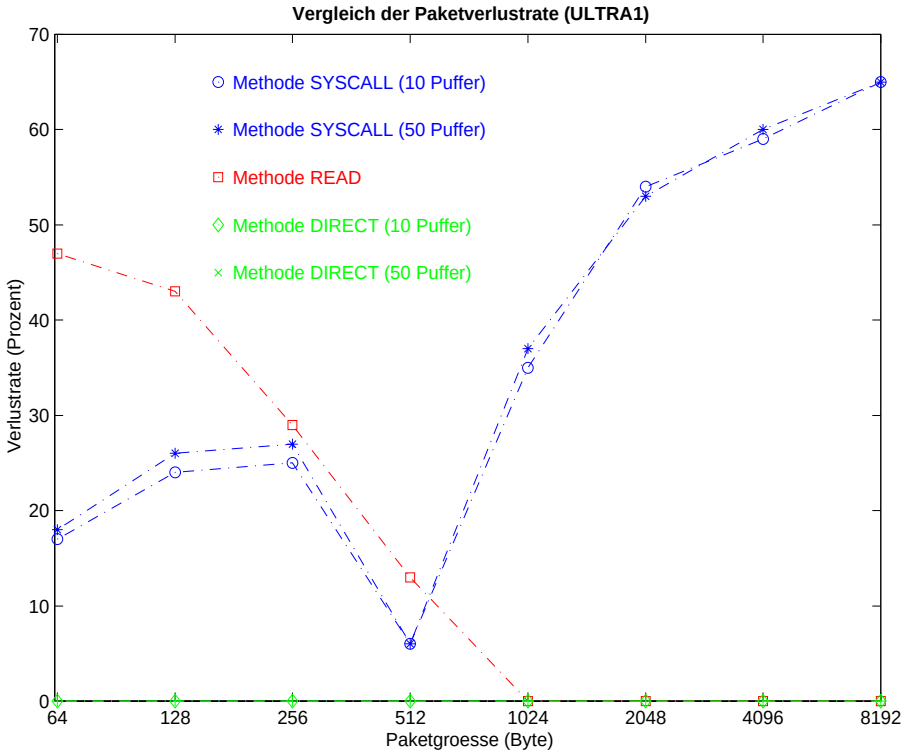


Abbildung F.5 - Vergleich der Paketverlustrate auf der ULTRA1

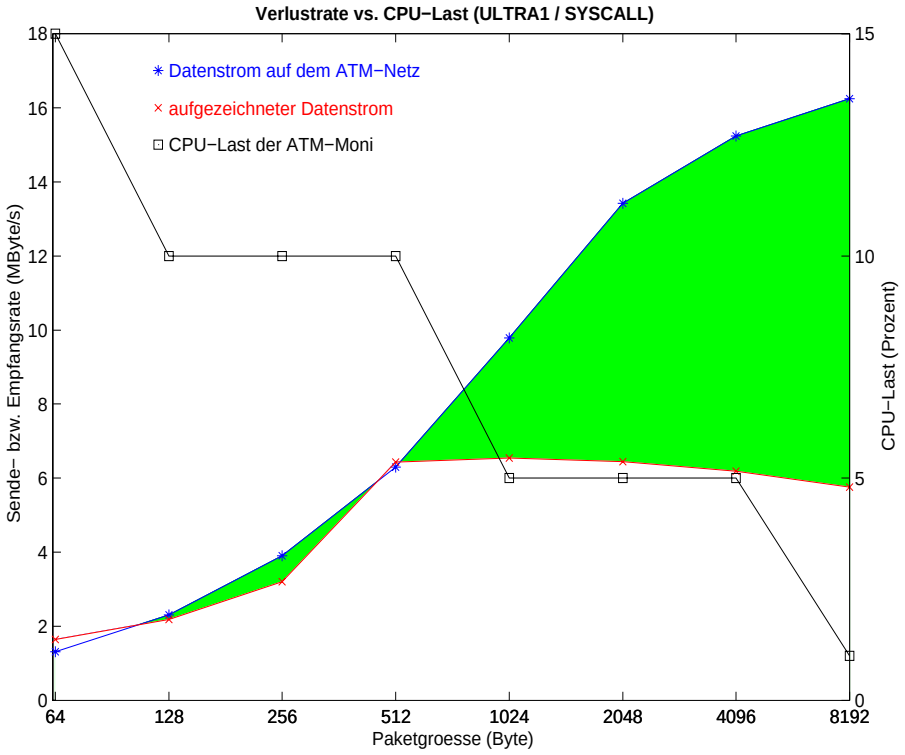


Abbildung F.6 - Verlustrate vs. CPU-Last (ULTRA1 / SYSCALL)

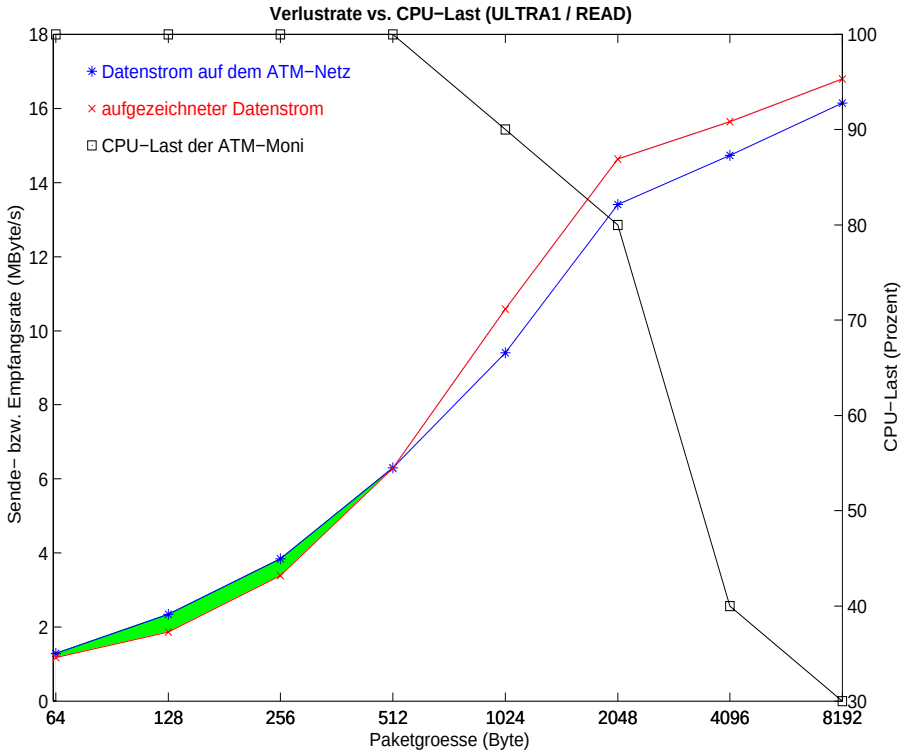


Abbildung F.7 - Verlustrate vs. CPU-Last (ULTRA1 / READ)

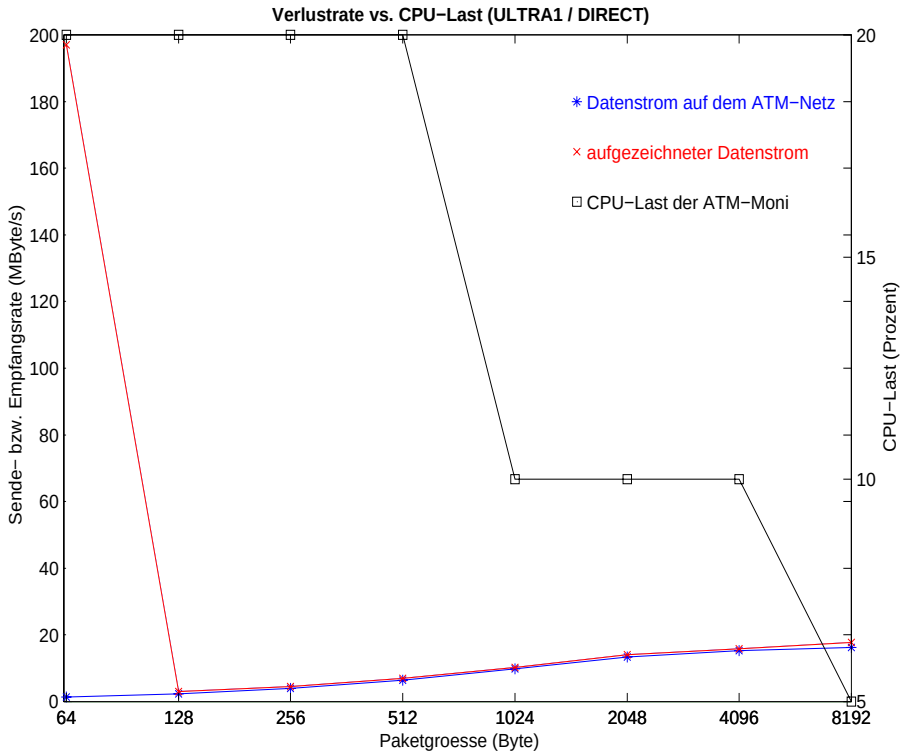


Abbildung F.8 - Verlustrate vs. CPU-Last (ULTRA1 / DIRECT)

G Protokoll der “ATM-Moni”-Tests

Um die Leistungsfähigkeit der Kommunikation zwischen dem Benutzerprozeß des Loggers und dem des Analyzers zu testen, und dadurch Aussagen über das Verhalten des Gesamtsystems “Moni-Box” zu erhalten, wurden mit den implementierten Funktionen zusätzlich zu den “test_logger”-Messungen aus Anhang F weiter Tests vorgenommen.

Konkret wurden die in Kapitel 5.4 entwickelten Programme einem künstlich erzeugten[4] ATM-Datenstrom ausgesetzt. Wesentliche Aussagen sind die Anzahl verlorener Pakete (Byte) und die CPU-Belastung während der Aufzeichnung.

Paketgröße (Byte)	**Datenstrom (kByte/s)**	**Datenverluste (kByte/s)**	**CPU-Last (ca., %)**
64	670	104.7	100
128	1150	105.5	95
256	2000	196.6	95
512	3300	294.3	95
1024	5000	45.1	90
2048	6800	75.2	80
4096	8200	87.7	70
8192	9200	377.2	60

Tabelle G.1 - Tests mit der READ-Methode

Bemerkung: Wie schon die “test_logger”-Messungen zeigten, ist die Methode READ aufgrund der hohen Datenverluste nicht für die Konstruktion einer ATM-Moni geeignet.

4. Wieder wurde der Datenstrom mit dem “push”-Programm erzeugt.

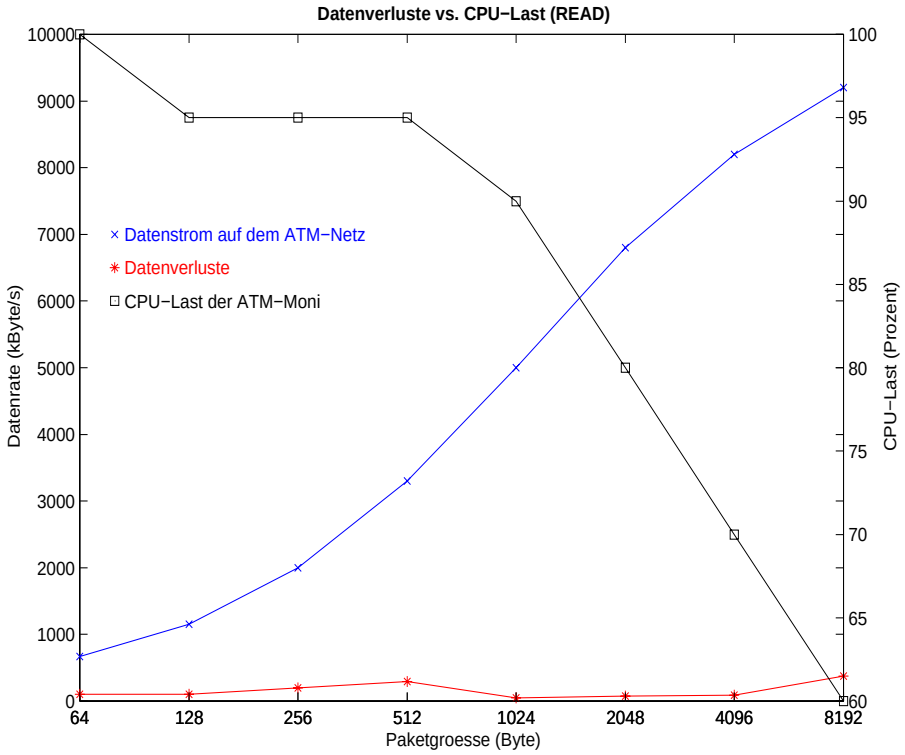


Abbildung G.1 - Datenverluste vs. CPU-Last (Methode READ)

Paketgröße (Byte)	**Datenstrom (kByte/s)**	**Datenverluste (Byte/s)**	**CPU-Last (ca., %)**
64	630	0	10
128	1200	0	15
256	2000	0	15
512	3200	0	10
1024	4900	0	10
2048	6600	0	15
4096	8200	0	10
8192	9400	0	10

Tabelle G.2 - Tests mit der DIRECT-Methode (mit 100 Puffern à 64kByte)

Bemerkung: Nach sehr kurzem Einschwingvorgang von weniger als einer Sekunde ist die Verlustrate gleich Null. Die CPU-Last von ca. 10-15% zeigt, daß noch genügend Ressourcen für eine Protokollanalyse vorhanden sind.

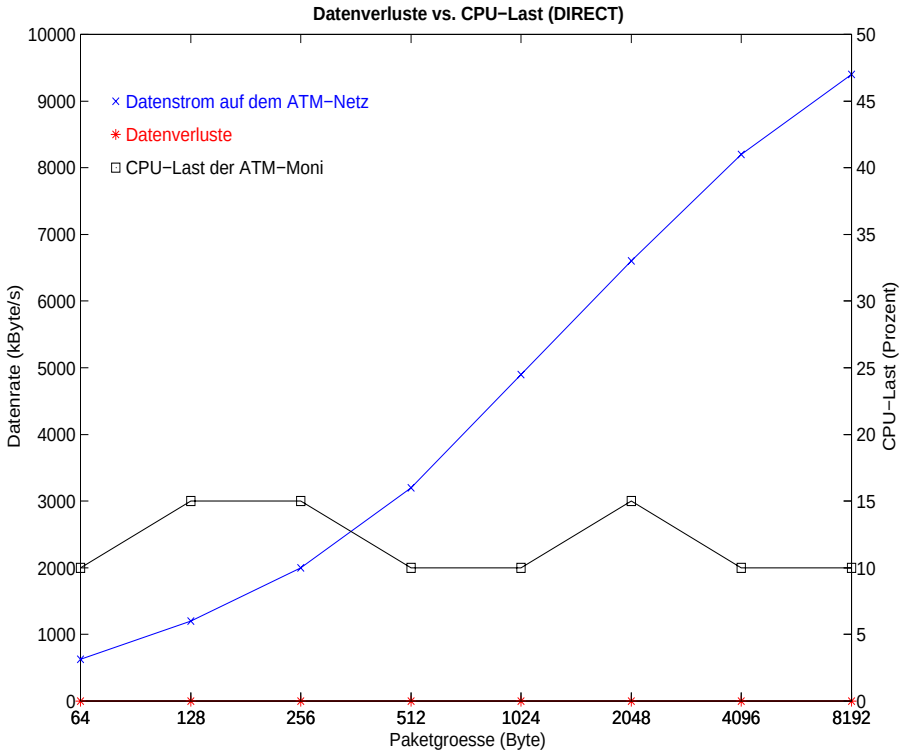


Abbildung G.2 - Datenverluste vs. CPU-Last (Methode READ)

H HARP Distribution Terms[5]

The Minnesota Supercomputer Center, Inc. is making the HARP software available "as is" for non-commercial uses. Please read and agree to the following distribution terms before downloading the HARP software.

Citing HARP

When citing HARP in published works, please use the following citation:

HARP Version 2, Minnesota Supercomputer Center, Inc. This software was developed with the support of the Defense Advanced Research Projects Agency (DARPA).

Support and Disclaimer

HARP Version 2 (the "Software") is made available by the Minnesota Supercomputer Center, Inc. (MSCI) "AS IS". MSCI does not provide maintenance improvements or support of any kind, however, comments, suggestions and problem reports are welcome and may be addressed to MSCI by email to the following address: harp-bugs@magic.net.

MSCI MAKES NO WARRANTIES OR REPRESENTATIONS, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT, AS TO ANY ELEMENT OF THE SOFTWARE OR ANY SUPPORT WHICH MAY BE PROVIDED IN CONNECTION WITH THE USE OF THE SOFTWARE. In no event shall MSCI be responsible for any damages, including, but not limited to, direct, indirect, consequential, special or incidental damages, arising from or relating to any use of the Software or provided support.

Copyright and Permitted Use

5. Kopie von [41]

Persons obtaining this Software through distribution by MSCI or redistribution as permitted in this paragraph ("Users") are permitted to use the Software for their non-commercial research and internal use only. For these purposes, non-commercial use means that no portion of the Software is incorporated in any program or other tangible or intangible product or service which is licensed or sold in any manner, or for use of which charges of any kind are otherwise imposed. Copies of the Software may be made for use permitted in this paragraph; Users must reproduce and include this copyright notice in all copies of the Software made by them. Modifications of the Software for use in accordance with this paragraph are permitted. The Software may be redistributed to others so long as the User making that redistribution has provided an accurate list of the email addresses of each recipient of that redistribution to MSCI by email addressed to: harp-bugs@magic.net. No use, copying, distribution, or display of the Software is permitted except as set forth in this paragraph.

Portions of this Software include materials copyrighted by the Regents of the University of California and by Sun Microsystems, Inc. The applicable copyright notices are reproduced in the files where the material appears.

No use, copying, distribution, or display of the Software is permitted except as set forth in this paragraph.

Literaturverzeichnis

[1] HANS KRÖNER: Verkehrssteuerung in ATM-Netzen. 62. Bericht über verkehrstheoretische Arbeiten, Institut für Nachrichtenvermittlung und Datenverarbeitung, Prof. Dr.-Ing. habil. P. J. Kühn, Universität Stuttgart, 1995

[2] GERD SIEGMUND: ATM - Die Technik des Breitband-ISDN. 2., überarb. und erw. Aufl., Heidelberg, 1994

[3] URESH VAHALIA: UNIX internals: the new frontiers. Prentice Hall, New Jersey, 1996

[4] JOHANNES WISSING: Dynamisches Bandbreitenmanagement am ATM-Netzzugang für burstartige, verzögerungstolerante Datenströme. Dissertation, FernUniversität-Gesamthochschule in Hagen, 1995

[5] MICHAEL HOCHMUTH: ATM-Netze: Architektur und Funktionsweise. 1. Aufl., Bonn, 1995

[6] MARTIN P. CLARK: ATM networks: principles and use. Wiley-Teubner, Chichester, 1996

[7] MARTIN DEPRYKER: Asyounchronous Transfer Mode. Prentice Hall, München, 1996

[8] ALEXANDER SCHILL: ATM-Netze in der Praxis: Einsatzszenarien, Produktkategorien, aktuelle Standards. Addision-Wesley-Longman, Bonn, 1997

[9] AXEL KLOTH: PCI und VESA Local Bus. Franzis, München, 1994

[10] ANATOL BADACH: ATM-Anwendungen: Multimedia-Kommunikation über Datenautobahnen. VDE-Verlag, Berlin, 1995

[11] RAINER HÄNDEL: ATM networks. Addison-Wesley, Wokingham, 1995

[12] MATHIAS HEIN: ATM. Thomson Publ., Bonn, 1997

[13] GEORGE KESIDIS: ATM network performance. Kluwer, Dordrecht, 1996

[14] OTHMAR KYAS: ATM-Netzwerke. DATACOM, Bergheim, 1996

[15] ACHIM SCHARF: ATM ohne Geheimnis. Heise, Hannover, 1996

[16] ANTONI BRONISLAW PRZYGIENDA: Link state routing with QoS in ATM LANs. Dissertation, ETH, Zürich, 1995

[17] TSONG-HO WU: ATM transport and network integrity. Academic Press, San Diego, 1997

[18] ADRIAN COCKROFT: Sun performance and tuning. Sun Microsystems, Mountain View, 1995

[19] Writing Device Drivers. SunSoft, Mountain View, 1997

[20] FALKO DREßLER: Entwurf und Implementierung von Dienstgütemessungen im ATM-Netz. Studienarbeit, Universität Erlangen-Nürnberg, 1996

[21] ATM Pocket Guide. Tekelec, Publication 908-0119-01, Revision B, 1994

[22] W. RICHARD STEVENS: Advanced Programming in the UNIX Environment. Addison-Wesley, Reading, Massachusetts, 1992

[23] W. RICHARD STEVENS: UNIX Network Programming, Volume 1: Networking APIs: Socket and XTI. 2. Aufl., Prentice Hall, Upper Saddle River, NJ, 1998

[24] HARVEY M. DEITEL: An Introduction to Operating Systems. 2. Aufl., Addison-Wesley, Reading, Massachusetts, 1990

[25] WULF BAUERFELD: ATM und Realzeit. PEARL 95, Workshop über Realzeitsysteme, Fachtagung der GI-Fachgruppe 4.4.2 Echtzeitprogrammierung, Dr. Peter Holleczek, Boppard, 1995

[26] JUHA HEINANEN: Multiprotocol Encapsulation over ATM Adaption Layer 5. RFC 1483, Telecom Finland, 1993

[27] MARK LAUBACH: Classical IP and ARP over ATM. RFC 1577, Hewlett-Packard Laboratories, 1994

[28] M. PEREZ U.A.: ATM Signalling Support for IP over ATM. RFC 1755, ISI, FORE Systems, Inc., Motorola Codex, Ascom Timeplex, Inc., 1995

[29] Asynchronous Transfer Mode - Networking the Future. Technical Whitepaper, Sun Microsystems, Inc., Montain View, 1995

[30] JOHN DAVID CAVANAUGH, TIMOTHY J. SALO: Internetworking with ATM WANs. Minnesota Supercomputer Center, Inc., 1992

[31] ELLIS NOLLEY: MPOA Goes Under the Microscope at User PANEL DISCUSSION. 53 Bytes, The ATM Forum Newsletter: Volume Four. Number Three (http://www.atmforum.com/atmforum/library/53bytes/backissues/v4-3/article-05.html, 27.5.1998)

[32] GEROGE SWALLOW: MPOA, VLANS and Distributed ROUTERS. 53 Bytes, The ATM Forum Newsletter: Volume Four. Number Three (http://www.atmforum.com/atmforum/library/53bytes/backissues/v4-3/article-04.html, 27.5.1998)

[33] LIVIO LAMBARELLI: ATM Service Categories: The Benefits to the User. White Paper, The ATM Forum, 1996 (http://www.atmforum.com/atmforum/library/service_categories.html, 27.5.1998)

[34] Voice Networking in the WAN. White Paper, The ATM Forum, 1997 (http://www.atmforum.com/atmforum/library/vtoa.html, 27.5.1998)

[35] Multi-Protocol Over ATM (MPOA) - A Brief Description. White Paper, The ATM Forum, 1997 (http://www.atmforum.com/atmforum/library/mpoa.html, 27.5.1998)

[36] Multi-Protocol Over ATM Version 1.0. AF-MPOA-0087.000, The ATM Forum, 1997

[37] EGIDO PERRETTI, FRÉDÉRIC THEPOT: ATM in Europe: The User Handbook. The ATM Forum, 1997

[38] GREG WETZEL: Reaping the Benefits: Frame Relay to ATM Interworking. 53 Bytes, The ATM Forum Newsletter: Volume Six. Number One (http://www.atmforum.com/atmforum/library/53bytes/53_4_98/53_4_98_03.html, 27.5.1998)

[39] SunATM-155 SBus Cards Manual. Sun Microsystems, Inc., Revision A, 1995

[40] HARP2, MSCI Advanced Networking Group (http://www.msci.magic.net/harp/harp2.html, 27.5.1998)

[41] HARP Distribution Terms, MSCI Advanced Networking Group (http://www.msci.magic.net/harp/dist-terms2.html, 27.5.1998)

[42] T. BRADLEY, C. BROWN, A. MALIS: Multiprotocol Interconnect over Frame Relay. RFC 1294, Wellfleet Communications, Inc., BBN Communications, 1992

[43] ATM User-Network Interface (UNI) Signalling Specification, Version 4.0. AF-SIG-0061.000, The ATM Forum, 1996

[44] Implementing ATM Signalling: Avoiding teh Interoperablility Pitfalls. BSTS Solutions Note 5963-7514E, Hewlett Packard Company, 1995

[45] FRIDOLIN HOFMANN: Betriebssysteme: Grundkonzepte und Modellvorstellungen. 2. überarb. Aufl., Teubner, Stuttgart, 1991

[46] BERND FLEMMING: Überwachung und Analyse von IP-bastiertem Datenverkehr. Diplomarbeit, Universität Erlangen-Nürnberg, 1994

Tabellenverzeichnis

Abbildungsverzeichnis