Forschungszentrum Jülich
in der Helmholtz-Gemeinschaft

Zentralinstitut für Angewandte Mathematik

Untersuchungen zum Einsatz von Low-Level Protokollen bei der Rechnerkommunikation in heterogenen Netzwerken

Daniel Erpelding

Jül-3970

Untersuchungen zum Einsatz von Low-Level Protokollen bei der Rechnerkommunikation in heterogenen Netzwerken

Daniel Erpelding

Berichte des Forschungszentrums Jülich ; 3970
ISSN 0944-2952
Zentralinstitut für Angewandte Mathematik Jül-3970

Zu beziehen durch: Forschungszentrum Jülich GmbH · Zentralbibliothek
D-52425 Jülich · Bundesrepublik Deutschland
☎ 02461/61-5220 · Telefax: 02461/61-6103 · e-mail: zb-publikation@fz-juelich.de

Kurzfassung

Bei der Rechner-Rechner-Kopplung kommt vor allem TCP/IP als Protokoll zum Einsatz, da es als einziges durchgängig von allen Rechnern und auf allen Netz-Medien unterstützt wird.

TCP/IP hat dabei jedoch einen erheblichen Protokolloverhead welcher sich negativ auf Bandbreite und Latenz des Datendurchsatzes auswirkt.

Um statt dessen netzwerkspezifische Low-Level-Protokolle einsetzen zu können, wurde beispielhaft eine modular aufgebaut Softwarebibliothek entwickelt, welche jene Low-Level-Protokolle für Myrinet und ATM nutzt, und es wurde ein, diese Bibliothek nutzenden Gatewayprozess implementiert.

Abstract

When ist is up to determine the protocol for coupling two computers, it is quite always TCP/IP, since only TCP/IP is supported by nearly all kind of computers and on all net media.

Unfortunately, TCP/IP comes with a considerable protocol overhead, which has a negative impact on both bandwidth and latency of the data flow rate. In order to use network specific low-level protocols instead, a modular software library has been developed, using low-level protocols for ATM and Myrinet and a gateway process, using this library, has been implemented as well.

Afterwards, performance of this library was measured and compared with that of conventional TCP/IP socket communication. The result is presented and discussed in this paper.

Inhaltsverzeichnis

1 Motivation und Einführung **1**

1.1 Über die Kommunikation in Rechnernetzen 1

1.2 Zwei Beispiele 2

1.3 Erklärung für den geringen Durchsatz bei TCP/IP 3

1.4 Alternative: Protokollumsetzer (Gateway) 3

2 Über Netzwerkhardware und APIs **7**

2.1 ATM, die Technologie. 7

2.1.1 TCP/IP über ATM 9

2.1.2 XTI FOREatm direkt über AAL5 9

2.2 Myrinet 11

2.2.1 Über die Hardware 11

2.2.2 Andere APIs für Myrinet 12

2.2.3 GM 13

2.2.4 Senden und Empfangen mit GM 14

3 Die entwickelte Software **17**

3.1 Hybridnetbibliothek 17

3.1.1 Funktionsübersicht 17

3.1.2 SEAP 19

3.1.3 Die Parameter 19

3.1.4 Herstellen einer Verbindung mit Hybridnet 20

3.2 Gateway 23

3.2.1 Prinzipielle Funktionsweise 23

3.2.2 Realisierung des Fullduplex-Betriebes 24

3.3 Das Messprogramm hybridpp 25

4 Messungen und Auswertung **27**

4.1 Durchführung 27

4.1.1 Aufbau des Versuchsnetzes 27

4.1.2 Das Messverfahren 28

4.2 Overhead der Hybridnet-Bibliothek 29

4.3 ATM: Vergleich zwischen FOREatm und CLIP 31

4.4 Myrinet: Vergleich zwischen TCP/IP und GM direkt 32

4.5 Der Gateway im Einsatz 33

4.6 Ein Blick auf die Latenzzeiten 34

5 Schlussbetrachtung **37**

5.1 Zusammenfassung 37

5.2 Ausblick 38

6 Anhang **39**

Kapitel 1

Motivation und Einführung

1.1 Über die Kommunikation in Rechnernetzen

Rechner werden aus den unterschiedlichsten Gründen vernetzt und längst werden nicht nur solche gleicher Bauart und desselben Herstellers miteinander zu homogenen Rechnernetzen verbunden. Die einzelnen Knoten der heute üblichen heterogenen Netzwerke können sich stark unterscheiden und für die Verbindungen stehen mannigfaltige Technologien zur Verfügung welche ihre jeweiligen Stärken und Schwächen haben. (zu den hier verwendeten Begriffen siehe auch [BrSp99])

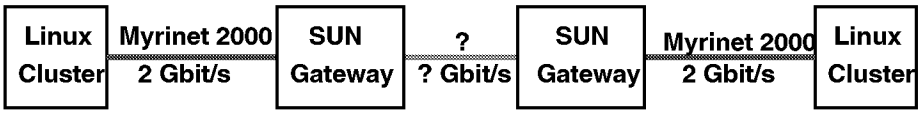
Damit zwei Rechner miteinander kommunizieren können, ist es notwendig, dass zwischen den beiden Einigkeit darüber besteht, wie diese zu erfolgen hat. Es muss für beide gleichermassen festgelegt sein, auf welche Aktion wie zu reagieren ist, oder anders ausgedrückt, beide müssen sich an ein und dasselbe (Gesprächsablauf-) Protokoll halten.

Nun ist *Protokoll* längst nicht mehr ein nur umgangssprachlicher Begriff. Wesentlich für erfolgreiche Herstellung und Betrieb von Netzwerken war die Standardisierung ihrer Komponenten. Die International Standard Organisation (ISO) nahm sich dieser Aufgabe bereits 1977 an und präsentierte unter anderem das Open Systems Interconnection (OSI) Referenzmodell, eine Abstraktion der Kommunikationsaufgabe, wobei sie in Teilschritte zerlegt und diese dann 7 unterschiedlichen aufeinander aufbauenden Schichten zugewiesen wird. Gängige Bezeichnung daher auch noch: Schichtenmodell. "Protokoll" wird nun zur Bezeichnung für das Regelwerk zur Kommunikation in einer dieser Schichten[Ke93]. Mit diesem Wissen wäre die obige Aussage präziser zu formulieren als: "Die beiden Rechner müssen sich zumindest in der obersten, der Anwendungsschicht des gleichen Protokolles bedienen."

Somit liegt es dann auch nahe, in einem heterogenen Netzwerk ein einheitliches Protokoll zu verwenden, das auf allen Medien in jedem Teilennetz implementiert ist. Die Protokollsuite TCP/IP, bestehend aus dem Transportschichtprotkoll *Transmission Control Protocol* (TCP) und dem Vermittlungsschichtprotokoll (die Vermittlungsschicht sei im folgenden zwecks besserem Verständnis in An-

erzielter Durchsatz 280 Mbit/s

Abbildung 1.1: Gigabit Testbed West:2 Parallelrechner über ATM-WAN verbunden



erzielbarer Durchsatz ?

Abbildung 1.2: RePhoNet : PC-Cluster über WAN verbunden

gleichung an die englische Bezeichnung; *Netzwerkschicht* genannt) *Internet Protocol* (IP), bietet sich an. Denn das für das Internet entwickelte Protokollpaar TCP/IP hat bereits eine lange Geschichte, weißt daher eine weite Verbreitung auf und darf als stabil und ausgereift gelten. Nun hinterließ bei der Einsatz von TCP/IP in verschiedenen Fällen aber eine gewisse Unzufriedenheit und warf verschiedene Fragen auf, welche dann Anlass zu dieser Arbeit gaben. Dies sei nun an zwei Beispielen erläutert.

1.2 Zwei Beispiele

Bei den Versuchen des ZAM (Zentralinstitut für angewandte Mathematik am Forschungszentrum Jülich) zum Gigabit Testbed West[Ei00] wurde ein Parallelrechner in Jülich mit einem weiteren sich in Sankt Augustin befindlichen Parallelrechner verbunden. Dazu wurden die Parallelrechner jeweils über ihr HiPPI (*High Performance Parallel Interface*)[1] Netzwerk mit einem Rechner des Herstellers SUN an ihrem Einsatzort verbunden, wobei diese SUN zu einem auf der Netzwerktechnologie asynchronous transfer mode (ATM) basierenden Wide Area Network (WAN) gehören. (siehe Abb. 1.1)

Wir haben hier also ein heterogenes Netzwerk vorliegen, bei dem zwei Netzwerke (HiPPI) über ein auf einer anderen Technologie beruhendes weiteres verbunden sind, der eingangs geschilderte Fall. Betrachtet man nun die auf den jeweiligen Teilnetzen erzielbaren Durchsätze von 800Mbits/s bei HiPPI und 622 Mbits/s bei ATM, so erstaunt der mit TCP/IP erzielte Gesamtdurchsatz von 280 Mbits/s durch seine Niedrigkeit, es wird ja offensichtlich nicht mal die Hälfte der Bandbreite des langsamsten Teilnetzes erreicht.

Das wirft die Frage auf, was wäre beim nächsten Beispiel, der Vernetzung zweier PC Cluster über ein WAN zu erwarten ? Eine solche Problemstellung ergäbe sich beim geplanten Regionalen Photonischen Netzwerk (RePhoNet) einem weiterem Projekt an dem das ZAM beteiligt ist.

Ein Cluster besteht aus mehreren "Kleinrechnern" (hier PCs mit dem Betriebssystem Linux) welche über ein Netzwerk (hier Myrinet 2000 mit 2 GBit/s erzielbarem Durchsatz[4]) verbunden sind. Sofern dies technisch zu realisieren ist, sollten die Cluster auch über das WAN mit Myrinet verbunden werden. Falls das nicht gelingt, muß auf andere Technologien, etwa Gigabit-Ethernet, ausgewichen werden. In diesem Fall ist wieder mit einer ähnlichen Problematik wie beim Gigabit Testbed West zu rechnen. Auch hier könnten SUN Rechner als Gateway dienen. (Siehe hierzu Abb. 1.2

1.3 Erklärung für den geringen Durchsatz bei TCP/IP

Bei Verbindungen die über TCP/IP aufgebaut und betrieben werden, muss auf jeden Fall auf beiden Seiten (Sender und Empfänger) fast der gesamte Protokollstapel durchlaufen werden. Allerdings setzen die verfügbaren Application Programm Interfaces (API) des Betriebssystemes, also die Funktionen mit denen ein Anwendungensprogramm Netzwerkverbindungen ansprechen kann, gleich auf dem Transportschichtprotokoll auf, Darstellungs- und Steuerungsebene werden in der Praxis selten unterstützt. Dieses Durchlaufen kostet Rechenzeit, denn die Daten müssen dann mehrfach im Speicher kopiert werden und dies verlangsamt die Übertragung.[Zie01]

Hinzu kommt noch der von TCP/IP verursachte Protokolloverhead, zu deutsch Verwaltungsmehrbedarf [BrSp99], was bedeutet, dass neben den Nutzdaten noch zusätzliche, von den Protokollen TCP und IP benötigte Steuerinformationen übertragen werden müssen.

Weiterhin muss der Protokollstapel in den Koppelungsknoten der unterschiedlichen Teilnetze, in den oben genannten Beispielen in den Gateway SUN Rechnern, bis zur Netwerkebene (IP) durchlaufen werden. (vgl. hierzu Abb. 1.3)

1.4 Alternative: Protokollumsetzer (Gateway)

Für einige Netzwerktechnologien sind APIs, welche direkt auf Low-Level Protokollen aufsetzen, spezifiziert und implementiert. Etwa für Myrinet die GM Bibliothek, welche im ISO-OSI Modell in der Sicherungsschicht zu situieren ist. Solange die Kommunikation innerhalb desselben Teilnetzes erfolgt, können statt der TCP/IP sockets genauso gut diese API programmiert und eingesetzt werden.

Doch sobald ein Rechner erreicht werden muss, welcher sich in einem Teilnetz befindet, was auf einer anderen Technologie basiert, muss zwischen diesen beiden Teilnetzen eine Protokollumsetzung erfolgen. Diese kann softwaremässig durchgeführt werden, sofern ein Rechner zur Verfügung steht, welche über Netzanschlüsse für beide Teilnetze verfügt. In der englischsprachigen Literatur heissen solche Protokollumsetzter Gateway, was zu Verwirrungen führen kann, weil die zur Kopplung von Teilnetzen eingesetzten Rechner selber die Bezeichnung Gateway tragen.

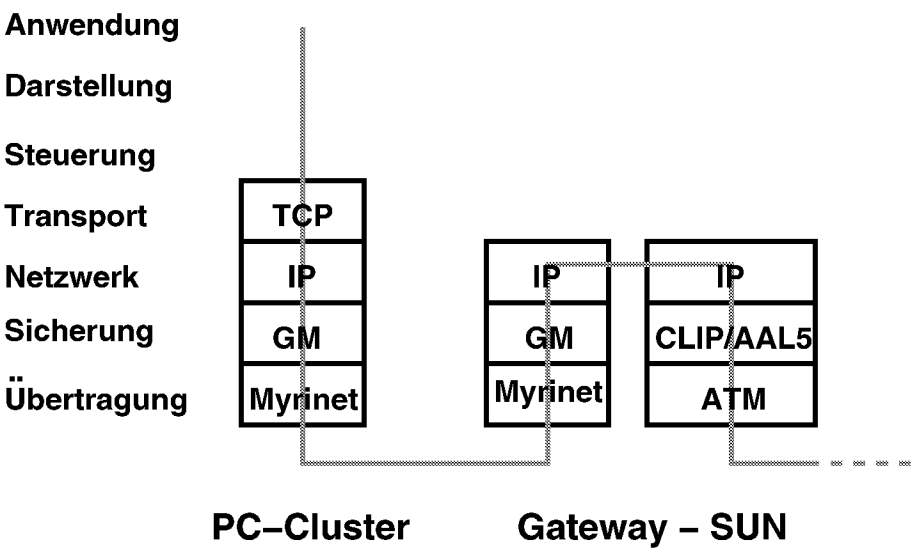


Abbildung 1.3: Durchlaufen des Protokollstapel bei Kommunikation mit TCP/IP

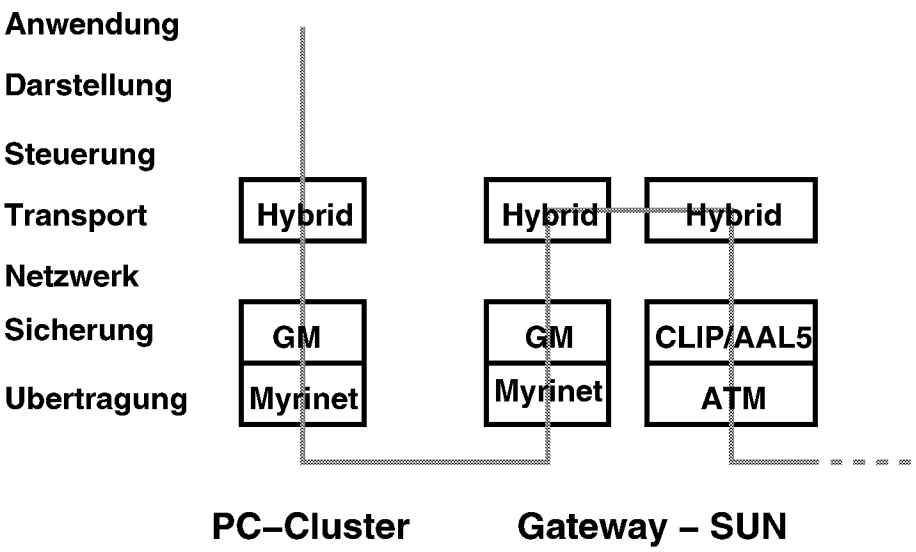


Abbildung 1.4: Durchlaufen des Protokollstapel bei Kommunikation beim Einsatz eines Protokollumsetzers

Ziel dieser Arbeit ist es, diese Alternative umzusetzen und auf ihre Leistungsfähigkeit hin zu untersuchen. Es wird dabei nur der wichtige Spezialfall einer gesicherten Vollduplexverbindung zwischen zwei Rechnern (denn dieser wird für die obigen Beispiele benötigt) implementiert. Soll heißen, die Hoffnung bzw. die Erwartung besteht, dass durch den Verzicht auf Funktionalität Performance gewonnen wird.

Hierzu wird nun eine Hybridsoftwarebibliothek entwickelt werden, welche ihrerseits auf die unterschiedlichen APIs zu den Low-Level Protokollen zugreifen können wird. Diese Bibliothek wird nun ihrerseits eine API darstellen und somit das eingangserwähnte einheitliche Protokoll für Sender und Empfänger zur Verfügung stellen. An den Koppelstellen zwischen den Teilnetzen wird ein auf dieser Hybridbibliothek aufbauender Gatewayprozess eingesetzt werden, welcher die Protokollumsetzung durchführen und für die Weiterleitung der Nachrichten verantwortlich sein wird. Abb. 1.4zeigt dies schematisch am Beispiel einer GM-ATM Umsetzung.

Kapitel 2

Über Netzwerkhardware und APIs

An dieser Stelle werden nun die beiden für die Untersuchung ausgewählten Technologien besprochen und die auf ihnen arbeitenden Protokollen vorgestellt. Myrinet wurde ausgewählt, weil es eine wichtige Technologie für Cluster ist und ATM, weil sich über ATM räumlich weiter entfernt liegende Rechner gut einbinden lassen. Zudem lagen für ATM bereits im Rahmen einer ebenfalls am ZAM angefertigten Diplomarbeit gewonnene einschlägige Erfahrungen mit dem Einsatz von Low-level Protokolle vor, wenngleich in jener Arbeit nicht XTI sondern eine proprietäre API verwendet wurde.[Le98]

2.1 ATM, die Technologie.

ATM ist eine Übertragungstechnik, deren Grundidee die Zerlegung der Nachrichten in sehr kleine Zellen fester Größe ist. Diese Zerlegung ermöglicht statistisches Multiplexen der Daten mehrere Verbindungen. D.h. die Zellen mehrerer Verbindungen werden nach einem bestimmten Schema gemischt und zwar so dass die auf dem gegebenen physikalischem Medium verfügbare Bandbreite optimal ausgenutzt wird.[HeLa97] Eine Zelle besteht aus einen 5 Bytes großem Kopf (Header) mit Steuerinformationen und 48 Bytes Nutzlast, also insgesamt 53 Bytes, laut Empfehlung der International Telecommunication Union (ITU).[Ky95]

Neben der ITU erarbeitet auch das ATM-Forum, ein ursprünglich (gegründet 1991) im wesentlichen von Rechner- und Netzwerkherstellern bestimmtes, heute von Mitglieder unterschiedlicher Interesselagen durchsetztes Konsortium, Standardisierungsvorschläge aus. [3][Ky95]

Wegen der festen Grösse können die Zellen hardwaremässig schnell und effizient vermittelt werden und weil die Zellenlänge klein ist, werden höher priorisierte Zellen weniger stark verzögert und somit kann auch ein isochroner Datenverkehr, wie er für Sprach- und Videosignale erforderlich ist, realisiert werden. Denn von sich aus sind Sender und Empfänger bei ATM nicht synchronisiert, die Abkürzung steht schliesslich für *asynchroner Transfermodus.*

Der Nachteil dieses Verfahrens ist allerdings auch ohne weiteres einzusehen: angesichts 5 Bytes Steuerinformation auf nur 48 Bytes Nutzlast ist der Overhead einer ATM automatisch über 10 %.[Zie01] und zudem kann ATM zur Erreichung dieses Zieles weniger hoch priorisierte Zellen verwerfen, die dann verloren gehen und erneut angefordert werden müssen.

Weitere Eigenschaften von ATM sind:

- ATM ist eine Universalnetztechnologie,
- ATM ermöglicht den Aufbau virtueller Netze, sogenannter VLAN will sagen: Mit ATM ist es z.B. möglich dass eine Firma oder Organisation ihre geographisch weitverstreut liegenden Rechner, als ein logisch zusammengehörendes Netz betrachtet mit allen Vorteilen welche dies haben kann.
- ATM ist skalierbar. d.h. ATM kann mit verschiedenen Verarbeitungsgeschwindigkeiten betrieben werden.
- ATM erlaubt es den Nutzern die Dienstgüte (Quality of Service QoS) auszuhandeln.
- ATM lässt sich auf unterschiedlichen physikalischen Übertragungssystemen (elektrische Leitung, Lichtwellenleiter...) realisieren und kann dennoch immer gleich behandelt werden. Es kann sowohl im LAN als auch im WAN Bereich eingesetzt werden.

ATM orientiert sich bei der Klassifizierung seiner Verbindungsmöglichkeiten stark am Fernsprechnetz und unterscheidet grundsätzlich zwischen:

1. Permanent Virtual Connections (PVC)
   Das sind Verbindungen welche, in Analogie zu Standleitungen beim Telefonnetz, einmal eingerichtet und dem Benutzer dauerhaft zugewiesen werden.
2. Switched Virtual Connections (SVC)
   Analog zu den Wählverbindungen sind dies Verbindungen welche der Benutzer bei jeder Sitzung neu auf- und nach Beendigen wieder abbaut.

Die Spezifikation von ATM definiert drei Schichten: die Bitübertragungsschicht, die ATM Schicht und die ATM Anpassungsschicht (engl. Atm AdaptionLayer (AAL)) wobei die beiden letzteren beim ISO-OSI Referenzmodell der Sicherungsschicht zuzurechnen sind. Zur Beschreibung von ATM beschränkt man sich aber nicht auf das OSI Schichtenmodell, sondern arbeitet noch mit einem ATM eigenen, dreidimensionalen Model welches die Bezeichnung ATM-Referenz-Würfel führt, das neben den Schichten noch Ebenen kennt und zwar die Benutzer- , die Steuer- und die Managementebene.[HeLa97]

Für den AAL wiederum sind fünf verschieden Auslegungen spezifiziert, weil auch fünf verschiedene Dienstklassen definiert sind, wobei sich jede Klasse für unterschiedliche Anwendungen besonders eignet. Die folgende Tabelle mag Aufschluss hierüber geben:

AAL-Typ	1	2	3/4	5
Dienstklasse	isochron	isochron	asynchron	asynchron
Verkehr	konstant	variabel	variabel	variabel
verbindungsorientiert	ja	ja	ja/nein	ja
Beispiel	Sprache/Video	Video	Datentransfer	Datentransfer

Der in dieser Arbeit interessierende Fall ist variabler Datentransfer, was bedeutet dass AAL5 verwendet wurde.

Die AAL zerfällt ihrerseits nochmal in zwei Teilschichten:

einerseits dem Convergence Sublayer (CS), welchem die Behandlung von Zellverzögerungen, Verzögerungsschwankungen, Zellverlusten obliegt und welcher Flusskontrollmechanismen zur Verfügung stellt.

Dieser CS baut seinerseits auf dem Segmentation an Reassembly Sublayer (SAR) auf, welcher die Daten die ihm der CS liefert, zum Transport in die erwähnten Zellen aufteilt, bzw. aus den Zellen die Daten für den CS zusammenstellt.

2.1.1 TCP/IP über ATM

TCP/IP ist die Protokollreihe welche die Anwendungen aus dem Internet verwenden. Da das Internet in den 1990er Jahren einen regelrechten Boom erlebt hat wurde eine beträchtliche Anzahl solcher Anwendungen entwickelt, auf welche Kunden eines ATM Netzes nicht verzichten mögen. Daher drängte es sich auf, Möglichkeiten zu entwickeln mit denen Verkehr über TCP/IP unter Benutzung von ATM Netzen erfolgen kann.

Drei Methoden wurden bisher entwickelt, welche zur Erfüllung dieser Aufgabe in Frage kommen: Classical IP (CLIP) entwickelt von der Internet Engineering Task Force (IETF), die LAN-Emulation (LANE) des ATM-Forums und Multiple Protocol Over ATM (MPOA) ebenfalls vom ATM-Forum. Eine eingehendere Untersuchung dieser Methoden ist in [Le98] zu finden, auf eine Wiederholung kann hier verzichtet werden. Bei den Versuchen zu dieser Arbeit wurde auf die Möglichkeit des Classical IP zurückgegriffen, weshalb wir bei diesem Verfahren noch kurz verweilen:

CLIP ist eine Schicht welche auf die AAL aufsetzt und zusammen mit diesem beim ISO-OSI Modell der Sicherungsschicht zugeordnet wird. CLIP kann ausschliesslich dem Netzwerkschichtprotokoll IP unterlegt werden, es ist auf eine Bereitstellung von IP Diensten zugeschnitten. Wichtigste Aufgabe ist es dabei, die vier Byte langen IP Adressen auf ATM Adressen umzurechnen.

2.1.2 XTI FOREatm direkt über AAL5

Zur direkten Nutzung von AAL5 konnte auf die XTI API zurückgegriffen werden. Diese Abkürzung steht für X/Open Transport Layer Interface, wobei das X/Open uns verrät dass es sich hierbei um eine offene, standardisierte und nicht um eine proprietäre (firmeneigene) Software handelt. Das Standardisierungsgremium das sich hinter dem Label verbirgt ist die Open Group, ein internationales

Konsortium aus Herstellern, industriellen Kunden, Regierungskreisen und Universitäten welches sich mit der Standardisierung von UNIX befasst.

XTI ist eine Weiterentwicklung von TLI der API für das AT&T Betriebssystem UNIX System V, welches eine Parallelentwicklung zu dem Berkley UNIX war.[BrSp99]

Die Firma FOREatm hat sich hier also für eine API entschieden, welche auch für die Programmierung von TCP/IP verwandt wird, dadurch wird die Programmierung von XTI zur direkten Ansteuerung von AAL5 jener der Sockets für TCP/IP sehr ähnlich, gleichwohl hier die Namen der Funktionen etwas anders lauten und einige Strukturen anders sind.

Eine Besonderheit bei ATM ist die Adressbildung. Während bei TCP/IP jede Maschine durch ihre 4 Bytes lange IP Adresse identifiziert wird und die Unterscheidung der einzelnen Verbindungen (es können auf einem Rechner ja durchaus auch mehrere Prozesse Netzwerkaktivitäten haben) dadurch erzielt wird dass für jeden Dienst ein anderer Port (dagestellt als eine Integerzahl) zu wählen ist, kommt diese Rolle bei XTI dem *selector* zu.

Der *selector* ist ein Byte groß und wird bei der Adressbildung der ansonsten 20 Bytes langen Geräteadresse angefügt.

2.2 Myrinet

2.2.1 Über die Hardware

Anders als ATM ist Myrinet nicht als Universallösung für die unterschiedlichsten Dienstanforderungen gedacht, sondern für den Einsatz als LAN optimiert. So ist etwa die Länge der elektrischen Verbindungsleitungen auf 25 Meter beschränkt. Erst seit kurzem sind optische Verbindungen mit immerhin schon 200 Reichweite verfügbar. Eingesetzt wird Myrinet hauptsächlich um Cluster zu bilden, d.h. einzelne Rechner werden über Myrinet miteinander verbunden und bilden so einen Parallelrechner. Dies nimmt nicht weiter Wunder, denn die Entwicklungsgeschichte von Myrinet, nahm ihren Ausgang in der Parallelrechnerforschung, genauer in den zwei Projekten: MOSAIC des Caltech, und dem ATOMIC LAN Projekt des USC/Information Science Instiute (USC/ISI). Bei diese Projekten bauten auf Netzwerkstrukturen massiv paralleler Parallelrechner (MPPs)BoCoFeKuSeSeKi95[?]

Verbreitet ist Myrinet ganz besonders an Universitäten und an anderen Forschungseinrichtungen, was seinen Grund wohl in der Tatsache findet, dass, gleichwohl Myrinet eine proprietäre Technologie der Firma Myricom ist, die verfügbare Software und die Hardwarespezifikationen offen sind.

Ein Myrinet Netzwerk besteht aus drei Komponenten:

1. Der Schnittstelle zum Rechner, dem Knoten des Netzes. (PCI Netzwerkkarte). Besonderheit hier: sofern das Betriebssystem es zulässt, sind diese in der Lage direkt auf den Hauptspeicher der Rechner zuzugreifen (Direct Memory Access: (DMA))
2. Den Myrinet Vermittlungseinrichtungen (multiple-port crossbar switches). An diese können sowohl weitere Switches oder die Rechner angeschlossen werden.
3. Den Vollduplexverbindungen zwischen Rechnern und Vermittlungseinrichtungen.

Die Myrinet-Hardware weiss noch folgende Merkmale zu bieten:

- Flusskontrolle, Fehlerüberwachung und Überwachung der einzelnen Verbindungen
- Skalierbarkeit durch die Möglicheit, mehrere Switches hierarchisch miteinander zu verbinden
- Möglichkeit von alternativen Kommunikationspfaden zwischen den Rechnern
- die Möglichkeit das Betriebssystem bei Ein- Ausgabeoperationen zu umgehen
- cutthrough routing (vulgo *wormhole*): Aufgrund der geringen Übertragungsfehlerwahrscheinlichkeit im Myrinet wird darauf verzichtet, an jedem Vermittlungsknoten die Daten zu überprüfen, sondern dies erfolgt erst im Zielknoten, dies bedeutet einen geringeren Protokolloverhead.[BoCoFeKuSeSeSu95]

2.2.2 Andere APIs für Myrinet

Zur Ansteuerung des Myrinet wurden mehrere APIs entwickelt, wobei die am ZAM verwendete GM Library diejenige ist, welche der Hersteller Myricom zur Verfügung stellt, auf die wir gleich im untenstehenden Abschnitt noch etwas genauer eingehen wollen. Doch der Vollständigkeit halber, zunächst zu zwei weiteren APIs, sie seien hier nur kurz umrissen, da sie nicht benutzt werden. Für weitergehende Informationen sei auf eine brasilianische Studie des Instituts für Information der Bundesuniversität von Rio Grande do Sul verwiesen.[OlBaAvNa99]

- BIP

Die Abkürzung steht für Basic Interface for Parallelism. BIP wurde an der Universität von Lyon entwickelt. BIP bietet eine Vielzahl von blockierenden und nicht blockierenden Funktionen zum Nachrichtenaustausch an.

Insbesondere unterscheidet BIP zwischen *kurzen* und *langen* Nachrichten, wobei der Benutzer festlegen kann was er unter einer kurzen und was unter einer langen Nachricht verstanden wissen möchte, denn sie werden auf unterschiedliche Weise dem Empfänger übergeben:

Lange Nachrichten werden, in Abhängigkeit von ihrer Gesamtlänge in mehrere gleich große Pakete aufgespalten per DMA bei Sender ausgelesen, über eine "Pipeline" versandt und beim Empfänger gleich wieder in den Hauptspeicher geschrieben, derweil kurze Nachrichten einfach nur gepuffert werden, da sich für diese der für DMA zu treibende Aufwand nicht lohnt.

Was Übertragungsfehler anbelangt, so werden diese von BIP nur entdeckt aber nicht korrigiert, dies wird dem Benutzer überlassen, BIP bietet somit von sich aus keine gesicherte Kommunikation.

- FM

Diese Abkürzung steht Fast Messages. FM wurde an der Universität von Illinois entwickelt, und ihr liegt ein anderes Konzept zu Grunde als BIP (und auch GM). FM orientiert sich am, in Berkley vorgestelltem Active Messages (AM) Mechanismus:

FM bietet nur sehr wenige Funktionen an und versieht dafür seine Nachrichten mit einem sogenannten "handler" Kopf, welchem die empfangende Funktion dann Anweisungen entnehmen kann, wie mit den Daten zu verfahren ist.

Die Entwicklung von FM wurde an jener des Message Passing Interface (MPI), einem Prokotoll höherer Ordnung ausgerichtet, dessen Kommunikation es zu optimieren bestimmt ist. FM unterstützt daher auch sogenannte Gather- und Scatter- Operationen welche bei MPI eine gewisse Rolle spielen.[Mpi95]

Ebenso wie BIP, unterscheidet auch FM zwischen kurzen und langen Nachrichten und kann, wie jenes nicht mehr als einer Anwendung gleichzeitig den Zugriff auf die Netzwerkkarte erlauben. Das kann nur GM.

FM übernimmt die Kontrolle des Datenflusses, derweil dies bei GM und BIP höhere Protokolle übernehmen müssen.

Die Eigenschaften der einzelnen hier vorgestellten APIs faßt folgende Tabelle zusammen (aus [OlBaAvNa99]):

	GM	BIP	FM
korrekte Reihenfolge	X	X	X
Fehlerkorrektur	X		X
Unterstützung für Gather/Scatter			X
low-level Flußkontrolle			X
Unterscheidung kurz/lange Nachrichten		X	X
Zugriff für mehrere Anwendungen	X		

2.2.3 GM

Wenden wir uns nun GM zu, der API des Herstellers. Wofür diese Abkürzung steht, darüber schweigen sich die verwendeten Quellen aus. Wir sahen bereits dass GM mehreren Prozessen gleichzeitg die Benutzung der Netzwerkschnittstelle erlaubt, es anderseits auf der Benutzung von DMA beharrt.

Auf GM lassen sich natürlich auch höhere Protokolle aufsetzen, so MPI und selbstverständlich auch TCP/IP. Zu sagen ist zu ersterem, dass dort wegen des dann größeren Protokolloverheades bei weitem nicht mehr dieselbe Bandbreite zu erreichen ist wie mit GM alleine, und zu letzterem, dass die hier erreichten Übertragungsraten stark von der Implementierung des TCP/IP Protokollstacks bei dem jeweiligen Betriebssystemes abhängt.[Zie01]

GM ist unter die paketvermittelnden, nichtverbindungsorientierten Protokolle einzuordnen, das heisst vor dem Senden ist keine Verbindung aufzubauen. In einem Knoten empfangene Daten, sind allerdings mit Informationen über Sender und Bestimmungsort gekennzeichnet, so dass die Anwendung entscheiden kann, ob sie für sie bestimmt sind oder nicht!

GM hat seine Schnittstelle so gestaltet, dass der Benutzer für GM bestimmte Daten weitestgehend nur mit den von GM dafür vorgesehenen Funktionen manipulieren kann. Zur Kenntlichmachung dieses Umstandes und zur Schonung des Namensraumes ist allen diesen Funktionsnamen ein “gm_” vorangestellt und sie setzen sich aus mehreren durch “_” getrennten Wörtern zusammen. Damit wird auch die Wahrscheinlichkeit eines eventuellen irrtümlichen Ansprechen von Speicherplätzen, welche vom Betriebssystem nicht für DMA freigegeben wären minimiert, ein Unterfangen welches ohnehin nur mit einer Fehlermeldung quittiert würde.

Ein wichtiger Vorteil von GM: Die Kommunikation zwischen zwei als *Port* bezeichneten Endpunkten ist gesichert, das heisst die Ankunft garantiert, und die Pakete kommen auch in der richtigen Reihenfolge an! Allerdings gilt dies nur innerhalb desselben Prioritätslevel, GM betreibt zwei solcher Grade: HIGH und LOW. Für diese Arbeit wurde nur letzterer verwendet, entsprechend der Empfehlung von Myricom[5], Nachrichten mit hoher Priorität nur für Steuerinformationen und nicht für Daten zu verwenden.

Die Ports lassen sich im übrigen durchaus ähnlich auffassen wie ihre Homonyme unter TCP/IP. Anders als dort sind wegen der Verbindunglosigkeit von GM Sendungen zwischen Ports mit unterschiedlicher ID allerdings gestattet.

2.2.4 Senden und Empfangen mit GM

Einerseits werden alle zur Kommunikation erforderlichen Aktionen von GM gehandhabt (etwa die Verwaltung der Puffer), und sind nur über, von GM verfügbar gemachte Funktionen abzuwickeln was einleuchtend ist, immerhin arbeitet GM mit der nicht von allen unkritisch gesehenen Möglichkeit des Direct Memory Access.

Andererseits bietet GM dem Programmierer die Möglichkeit, Teilschritte des Kommunikationsablaufes selber zu bestimmen, was natürlich damit erkauft wird, dass er sich auch um Details kümmern muss, was wiederum eine gewisse Sorgfalt abverlangt:

GM verwaltet Empfangs- und Ausgabepuffer, der Benutzer muss sie aber anlegen.

Ein weiteres Beispiel: Anwendungen müssen sowohl zum Senden als auch zum Empfangen, sich von GM dazu die notwendige Erlaubnis einholen. Dies erfolgt durch den Aufruf entsprechender Funktionen, welche im Erfolgsfalle diese Erlaubnis in Form einen sogenannten Token, im Fehlerfalle eine entsprechende Meldung zurückliefern. Hat eine Anwendung sich einen solchen Token besorgt kann sie die gewünschte Sende/Empfangsfunktion aufrufen, womit sie damit die Berechtigung zu weiteren Aktionen wieder aufgibt.

Durch diese Maßgabe behält GM die Kontrolle über die Vorgänge und kann, so fern alle Anwender dafür Sorge tragen, dass sie keinerlei Sende- oder Empfangsfunktionen aufrufen, wenn sie nicht im Besitze eines solchen Tokens sind, gegenseitiges Blockieren (Deadlock) ausschliessen.[5]

Um dem Leser etwas mehr Klarheit über die Vorgehensweise im einzelnen zu verschaffen, soll das Beispiel des Datenempfangs über GM an dieser Stelle vorgestellt werden:

1. Zunächst ist GM mit der Funktion *gm_ init()* zu initialisieren.
2. Anschliessend ist ein Port zu öffnen. Dabei muss diesem Port eine Kennzeichnung, genannt *port_ id* gegeben werden. Diesem ein Byte großen Bezeichner *port_ id* kommt eine ähnliche Rolle zu, wie dem *selector* bei XTI oder dem Intergerwert *port* unter TCP/IP, er erlaubt etwa die Unterscheidung mehrerer Verbindungen pro Rechner. Mit diesem Öffnen erhält der Prozess eine Anzahl von recieve token.
3. es werden nun einige Parameter gesetzt wie: maximale Nachrichtenlänge, maximale Nutzlast
4. dann wird für diese Verbindung mindestens ein Empfangspuffer angelegt mit *gm_ dma_ malloc*() oder (*gm_ dma_ calloc*()) Funktion und stellt sie zusammen mit einem recieve token GM zur Verfügung.

5. zum eigentlichen Empfangen sind, wie bereits angeführt viele verschiedene Funktionen verfügbar, welche alle unterschiedliche Funktionalität bieten. (z.B. blockierend und nicht blockierend)

6. die Funktion liefert nun eine *event* genannte Datenstruktur zurück, welcher das Programm entnehmen kann, welches Ereignis eingetreten ist: Empfang von Daten, wohin sind diese abgelegt worden ? wielviele Bytes waren es ? von welchem Rechner (ablesbar an der node_id) stammen sie ? welche port_id benutzte der Sender ? welche Priorität haben die Daten ?

7. Nachdem er die Daten bearbeitet hat (z.b. sie sich aus dem DMA Speicherraum in seinen eigenen Speicherbereich kopiert hat, oder aber als nicht für sich bestimmt verworfen hat) gibt der Benutzerprozess Puffer und Token wieder frei.

Senden funktioniert analog, hier ist aber noch ein Callback, eine Bestätigung dass alle Daten erhalten wurden, erforderlich.

Schlussendlich darf der Hinweis nicht fehlen¸ dass zum sauberen Schliessen des Kommunikationsendpunktes ¸ analog zu *gm_ init()* beim Öffnen, eine Funktion *gm_finalize* aufgerufen werden soll.

Kapitel 3

Die entwickelte Software

Wie im Eingangskapitel bereits erwähnt, bedarf es zu der angestrebten Lösung für die Rechnerkommunikation in heterogenen Netzwerken zweier Komponenten:

Erstens einer einheitlichen API für Client und Server, der sich auf unterschiedlichen Rechnern befindlichen beteiligten Anwenderprozesse, welche auf die APIs zu den jeweiligen eingesetzten Low-Level Protokollen aufsetzt und

zweitens eines Prozesses welcher die erforderliche Protokollumsetzung an der Schnittstelle der Teilnetze durchführt.

Eine graphische Darstellung mag die Funktionsweise dieser Lösung verdeutlichen:

In dem in Abbildung 3.1 vorgestellten Fall ruft ein Applikationsclient über die Hybridnetschnittstelle Funktionen der XTI API auf und kommuniziert über das physikalische Medium ATM mit dem Gateway, welcher seinerseits in gleicher Weise die Verbindung zum Zielserver, diesmal über die GM Schnittstelle über das Übertragungsmedium Myrinet aufnimmt und betreibt.

3.1 Hybridnetbibliothek

3.1.1 Funktionsübersicht

Die Hybridbibliothek ist in der Programmiersprache C implemtentiert und der Quelltext besteht zur Zeit aus den Dateien (und ihren jeweiligen Headerdateien) *hybridnet.c* (*hybridnet.h*), *gm_hybrid.c* (*gm_hybrid.h*), *atm_svc.c* (*atm_svc.h*), *foreatmcommon.c* (*foreatmcommon.h*), *tcp_ip.c* (*tcp_ip.h*) und *constants.h.* Sie bietet ein API, welches es den Anwendungsprogrammen ermöglicht, Daten mit anderen sie benutzenden Programmen auf anderen Rechnern, über Netzwerk- und Protokollgrenzen hinweg auszutauschen. *Hybridnet* benutzt seinerseits die jeweilige API zu den von ihr bedienten Protokollsuiten zu den entsprechenden Netzwerkarten: TCP über Berkley Sockets für alle, XTI-FOREatm für das ATM Netz, die GM Bibliothek für Myrinet. Ihren vollen Funktionsumfang erreicht sie

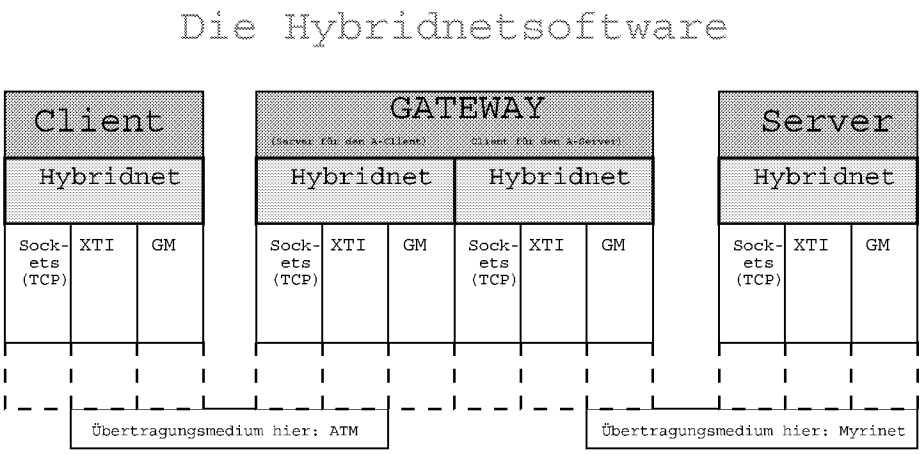


Abbildung 3.1: schematische Darstellung der Kommunikation über Hybridnet

nur, wenn auch ein Gatewayprozess des unten beschriebenen Programmes Gateway auf einem erreichbaren Rechner, am besten auf einer speziell zur Kopplung der verschiedenen Netze eingesetzten Maschine, läuft.

Das Hybridnet-API ist von seinem Aufbau her an verbindungsorientierten Protokollsuiten wie TCP/IP ausgerichtet, gleichwohl etwa das von ihr benutzte GM-Myrinet es nicht ist. Sie bietet dem Benutzer die folgenden Funktionen an:

```
struct conn h_connectserver(struct seapservice *service,int flag);
struct conn h_connectclient
(struct seapservice *zieldienst, struct seapservice *gateway ,int flag);
int h_send(struct conn fd,char *buffer, int len);
int h_recv(struct conn fd,char *buffer, int len);
int h_close(struct conn fd);
```

Das sind :

`h_connectserver`: Dienst (Service) anbieten und EINE Verbindung annehmen,

`h_connectclient`: sich mit einem Dienstanbieter (Server) verbinden lassen,

`h_send`: Daten versenden und

`h_recv`: Daten empfangen.

`h_close`: Verbindung wieder schließen.

Um auch Dienste als concurrent server (d.h. mehr als eine Verbindung annehmen) anbieten zu können, wurden zusätzlich noch die Funktionen:

```
struct conn h_startaserver(struct seapservice *service,int flag);
struct conn h_accept(struct conn urfd);
```

eingerichtet. Der Benutzer wird hierbei erst mit *h_startaserver* seinen Dienst starten, und muss dann sukzessive mit `h_accept` seine Verbindungen annehmen und sie dann verwalten.

3.1.2 SEAP

Ein Anbieter wird mit *h_connectclient* oder *h_startaserver* seinen Dienst unter einem frei gewählten Protokoll auf eingehende Verbindungswünsche warten lassen. Hybridnet wird diesen Dienst mit der Funktion *seap_publishf* beim Service Announcement Protocol (SEAP) anmelden und diesem alle Daten mitteilen, welche ein Client der sich mit diesem Dienst verbinden lassen möchte, wissen muss.

SEAP ist ein am ZAM entwickelter und eingesetzter Dienst. Er ermöglicht es Client und Server anderer Anwendungen, ihren Dienstzugangspunkt auszuhandeln. TCP-IP Anwendungen z.B. handeln aus welcher Port benutzt werden soll. Eine detailliertere Beschreibung von SEAP ist in [EiFri00], wobei die dort behandelte Version 1.0 ihr Angebot noch auf TCP-IP Anwendungen beschränkte, SEAP wurde für diese Diplomarbeit um zusätzliche Funktionalität erweitert und kann nun beliebige Informationen verwalten. Den in [EiFri00] beschriebenen C-Funktionen, deren Namen nun um den Buchstaben "f" verlängert sind, kann analog zur C-Funktion *printf* ein Formatstring mitgegeben und somit festgelegt werden welche weiteren Informationen in welchem Format über SEAP ausgetauscht werden sollen.

Kurz: der Client braucht nur Namen, und ein freigewähltes Passwort von dem anzusteuernden Dienst zu kennen und kann die benötigten Informationen zum Dienstzugangspunkt mit *seap_queryf* abfragen. Voraussetzung ist natürlich, dass ein für den Clienten erreichbarer SEAP Server im Hintergrund läuft. Durch den Einsatz von SEAP muss der Client beim Aufruf von *h_connectclient* weder wissen, auf welchem Rechner noch in welchem Netz, ja noch nicht mal über welches Protokoll der angestrebte Dienst zu erreichen ist.

3.1.3 Die Parameter

Um die von den Funktionen benötigten Parameter in eine möglichst übersichtliche Form zu bringen, wurden neue Strukturen einführt:

die Struktur *conn* enthält alle Daten um den Endpunkt einer eingerichteten Verbindung eindeutig zu beschreiben und sie so beim Senden und Empfangen auf dieser Verbindung ansprechen zu können und um sie zu schliessen:

```
typedef struct conn{
    int protocol;
    int blockflag;
    union u {
        int sockfd;
        int fd;
        struct gmconn portd;
```

```
    } u;
} conn;
```

die Struktur enthält Information darüber, welches Protokoll verwendet wird, ob die Verbindung blockierend ist sowie eine protokollspezifische Beschreibung der jeweiligen Verbindung. Bei den Berkley Sockets für TCP ist das ein Integerwert der *socketfiledescriptor* genannt wird und analog dazu bei XTI schlicht *filedescriptor* heisst. Dieser verbreiteten Benennung zugrunde liegt die Auffassung oder das Bestreben, unter Unix Verbindungen ähnlich wie eine Datei behandeln zu können. Für GM reicht ein einfacher Integerwert nicht: Zwar lässt sich eine Verbindung durch Angabe einer *port_id.* eindeutig beschreiben, einige Funktionen benötigen aber weitere Informationen, welche über eine Struktur *gm_port* ansprechbar sind. Dazu wird zudem noch ein Pointer auf void übergeben. Diese beiden Werte wurden hier in der Struktur *gmconn* zusammengefasst:

```
typedef struct gmconn{
    int port_id;
    int remote_node_id;
    int remote_port_id;
    int local_node_id;
    void *port;
    int blockflag; }
gmconn;
```

Umfangreicher gestaltet sich bereits die Struktur seapservice. Wie der Name bereits andeutet, enthält sie vor allem Informationen, die an SEAP übergeben oder von SEAP ausgelesen werden sollen.

```
typedef struct seapservice {
    char name[MAXNAME];
    char password[MAXNAME];
    union {

        char          host[MAXNAME];  /* fuer TCP/IP */
        unsigned char selector;       /* fuer ATM-SVC */
        struct gmconn portd;          /* fuer Myrinet */
    } u;
    int32_h protocol;
} seapservice ;
```

3.1.4 Herstellen einer Verbindung mit Hybridnet

Zunächst einmal braucht *h_connectclient*, wie oben beschrieben, Name und Password des anzusteuernden Zieldienstes um sich bei SEAP erkundigen zu können. Dazu muss *h_connectclient* ein Pointer auf eine entsprechende *seapservice*

Struktur übergeben werden. Gleiches gilt für den zu benutzenden Gatewaydienst. Wird hier nun ein Nullpointer übergeben so erfolgt ein Verbindungsaufbau direkt mit dem Zieldienst unter Umgehung des Gateway! Vorausgesetzt natürlich dass dies technisch möglich ist, also der Zieldienst in einem erreichbaren Netz läuft und zwischen beiden Endpunkten dasselbe Protokoll verwendet werden kann. Der Client braucht auch in diesem Fall beim Aufruf von *h_ connectclient* noch nicht zu wissen, unter welchem Protokoll der Dienst angeboten wird.

Wird der Dienst eines Gateways in Anspruch genommen, so wird *h_ connectclient* zunächst eine untergeordnete aber ebenfalls protokollunabhängige Funktion *h_ connect2* aufrufen, welche sich bei `SEAP` nach einem Dienst mit dem im Parameter *struct seapservice *gateway* festgelegten Namen erkundigen wird. Von SEAP wird sie nun erfahren, dass eine TCP-IP Verbindung zu diesem Gatewayserver (s.u.) aufzubauen ist und das wird sie dann tun. Anschliessend wird *h_ connectclient* dem Gatewayserver

1) die Daten des anzusteuernden Zieldienstes (Servicename und Servicepassword bei `SEAP`) übermitteln.

2) das eigentlich für den Datentransfer zwischen Clienten und Gateway erwünschte Protokoll mitteilen.

Zur Bezeichnung der implementierten Protokolle wurden ganze Zahlen ausgewählt. Welche Zahl für welches Protokoll steht, ist ebenfalls in der Datei *constants.h* festgehalten. Weicht das mitgeteilte Protokoll von TCP/IP ab, so wird gatewayseitig ein weiterer Server unter dem gewünschten Protokoll gestartet und *h_ connectclient* wird zu diesem eine Verbindung aufbauen und anschließend die TCP/IP Verbindung schliessen. Diese Verbindungsaufnahme wird *h_ connectclient* ebenfalls über die interne Funktion *h_ connect2* durchführen. Erneut wird diese sich bei `SEAP` erkundigen, diesmal nach einem Dienst welcher einen Namen trägt, der sich aus der Bezeichung des gewünschen Protokolls (z.Z.: *tcp-ip, atm* oder *myrinet*) und dem des anzusteuerendem Zieldienst zusammensetzt, und eine Verbindung unter dem erwünschten Protokoll herstellen. Anschließend wird er die TCP/IP Verbindung schließen.

Aus Sicht des Clienten ist der Verbindungsaufbau nun abgeschlossen und *h_ connectclient* wird zurückkehren. Diese Vorgehensweise zum Verbindungsaufbau ist nicht die einzig denkbare, sie ergab sich zunächst in Laufe der Entwicklung des Programmes als die praktikabelste Lösung weil TCP-IP auf jedem Rechner zur Verfügung steht und es somit das leichteste war, den Dienstzugangspunkt für den Gatewayserver auszuhandeln. Der Einsatz von SEAP hat weitere Möglichkeiten denkbar gemacht, worauf noch zurückzukommen sein wird.

Zum besseren Verständnis wurde dieser Vorgang in Abb. 3.2 graphisch dargestellt. Wieder wurde dazu das Beispiel eines Clienten, der mittels XTI überATM mit dem Gateway kommuniziert und sich an einen unter GM laufenden Server via Myrinet verbinden lässt. Angefügt an den Aufbau der Verbindung ist noch ein Datenaustausch vom Clienten zum Server, welcher diese dann zurückgibt.

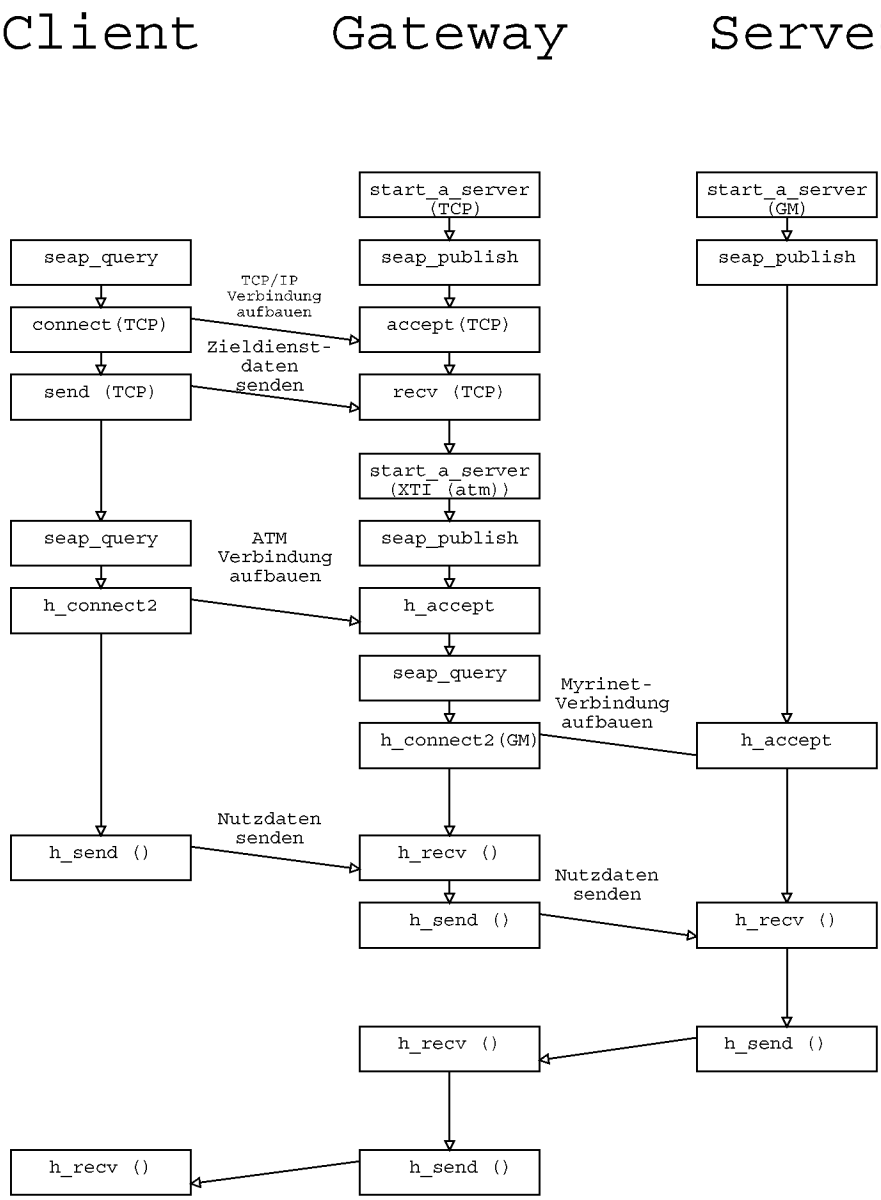


Abbildung 3.2: Verbindungsaufbau und ein Datenaustausch mit Hybridnet

3.2 Gateway

Damit die Hybridnetbibliothek ihren vollen Funktionsumfang erreicht, muss an geeigneter Stelle ein Gatewayprozess laufen, welcher die Kopplung zwischen verschiedenartigen Netzen bewerkstelligt, die Verbindungswünsche in die anderen Netze weiterreicht und den Datentransfer mit diesen Diensten handhabt. Idealerweise sollte dieser Prozess auf einem speziell zu diesem Zwecke eingerichteten Rechner laufen, welcher direkten Zugang zu allen beteiligten Teilnetzen hat.

Der hier implementierte Gatewayprozess ist nun seinerseits ein Benutzer der Hybridnetbibliothek und der von ihr bereitgestellten Funktionen.

3.2.1 Prinzipielle Funktionsweise

Statt dass eine Applikation als Client direkt mit der Applikation welche den Part des Servers übernimmt Kontakt aufnimmt und mit ihr die Daten austauscht, wird der Applikationsclient zunächst mit dem Gateway in Verbindung treten und dieser wird nun zum Server für den Applikationsclienten. Seinerseits baut der Gateway dann eine Verbindung zum Applikationsserver auf, für den er dann der Client ist. Sind beide Verbindungen aufgebaut wird der Gateway von nun an Nachrichten, die er vom Applikationsclient erhält, an den Applikationsserver weitersenden und umgedreht. Beendet einer der Kommunikationspartner, Applikationsclient oder Applikationsserver seine Verbindung, muss der Gateway auch die Verbindung zu dem anderen Partner abbauen. Soweit das Funktionsprinzip eines Gateway.

Im praktischen Einsatz würde ein Gatewaydienst sicherlich nicht nur einem einzigen Client-Server Paar zur Verfügung stehen müssen. Um mehrere Verbindungen annehmen zu können, wurde sich dafür entschieden nach dem Konzept des *concurrent server* vorzugehen:

Bei diesem Modell, wartet ein Server auf eingehende Verbindungswünsche und wird wenn ein solcher vorliegt, zur Behandlung derselben einen weiteren Serverprozess starten, welcher dann exklusiv dieser Verbindung zu diesem Clienten zur Verfügung steht. Der Vaterprozess hingegen wartet auf weitere eingehende Verbindungswünsche.[Ste98],S. 45.

Die einfachste Möglichkeit, einen concurrent server zu realisieren war durch Aufrufen der Unix Systemfunktion *fork*. Leider erlaubt die für Myrinet Aufrufe verwendete Bibliothek GM Library gegen sie gelinkten Programmen keinerlei *fork* Aufrufe und ist ausserdem nicht threadsicher![5] Daher musste das Programm aufgeteilt werden in einen nicht gegen GM gelinkten Gatewaymaster, welcher deshalb *fork*en darf, und einen Gatewayslave. Der Master wird auf Verbindungswünsche warten und dann den Slave, als eigenständigen Prozess mit der funktion *exec* starten und letzterer übernimmt nun nun folgende Aufgaben:

1) Wenn die Verbindung zwischen Gateway und Applikationsclient nicht über TCP/IP betrieben werden soll: Starten eines weiteren Servers unter dem gewünschten Protokoll, Verbindungsaufbau zum Applikationsclient.

2) Verbindungsaufbau zum Applikationsserver.

3) Datenaustausch in beide Richtungen realisieren.

Der Aufbau der Verbindung zwischen Applikationsclient und Gateway erfolgt in der oben beschriebenen Weise. Dies hat den Vorteil, dass nur ein zentraler Gatewayprozess auf Verbindungswünsche wartet und jeder Client weiss, dieser ist über TCP/IP zu erreichen. Denkbar wäre es aber auch gewesen für jedes implementierte Protokoll einen eigenen Server warten zu lassen, sei es als eigenständiger Prozess, oder aber mit Hilfe der Methoden *polling* und *threading*, welche wir im nächsten Abschnitt behandeln wollen. Wie wir bereits gesehen haben, kommt die letztere Methode in unserem Fall nicht in Frage.

3.2.2 Realisierung des Fullduplex-Betriebes

Ist die Verbindung zwischen Applikationsclient und -Server eingerichtet, hat der Gateway folgende Aufgabe zu erfüllen: er muss aus beiden Richtungen (Client und Server) Daten empfangen und diese dann in die komplementäre Richtung weitersenden. Vollduplexbetrieb heisst hier: es gilt also bereit zu sein, dass von mehr als nur einer Seite Daten eintreffen können und zwar auch zur gleichen Zeit! In der Theorie gibt es, laut Stevens [Ste98] fünf Ein-/Ausgabemodelle (I/O Modells) für welche man sich entscheiden könnte, unter Unix bieten sich im Prinzip drei Wege, dieses Problem zu lösen:

- polling (nicht blockierende Funktionsaufrufe)

Wird eine Empfangsfunktion aufgerufen und es sind noch keine zu empfangenden Daten eingetroffen, so kann sie entweder warten bis sie das sind, in dem Fall ist die Funktion blockierend, oder sie kann so einstellt werden, dass sie sofort mit entsprechender Fehlermeldung wieder zurückkehrt. In letzterem Fall wird die Funktion als nichtblockierend bezeichnet. Ähnliches gilt für Sendefunkionen. Für TCP und für XTI gibt es hierfür die Funktion *fcntl*, GM-Myrinet bietet unterschiedliche Sende- und Empfangsfunktionen [6] für blockierenden oder nichtblockierenden Transfer an.

Bei nichtblockierenden Funktionen obliegt es dann dem Benutzer dafür Sorge zu tragen, dass er die Daten vollständig empfangen/gesendet hat. Kehrt eine solche Funktion mit dem Vermerk "Daten liegen nicht vor" zurück hat das sie aufrufende Hauptprogramm nun die Möglichkeit, z.B. weitere Funktionen aufzurufen und auf andere Ereignisse abzufragen (engl. *to poll*, daher *polling*) und entsprechend zu handeln. Anschliessend wird das Programm diese Prozedur wiederholen, bis das eingetreten ist worauf es wartet.

Vorteile dieser Möglichkeit sind ihre einfache Handhabung und die Überschaubarkeit des Ablaufes, da die Aufrufe einfach nur sequentiell abgearbeitet werden. Entsprechend schnell ist auch die Reaktion des Systems auf Ereignisse.

Nachteil ist, dass wir es hier mit aktivem Warten zu tun haben, was die CPU belastet, denn diese Abfrage obliegt dem Hauptprogramm. Dieser kann aber in Kauf genommen werden kann, insbesondere dann wenn ein eigener Rechner speziell zu diesem Zwecke als Gateway zwischen den Netzen eingesetzt wird.

- selecting (blockierende Funktionen)

Sollen blockierende Funktionen verwendet werden, kann man das Warten auf das Eintreten der Ereignisse auch dem Betriebssystem überlassen, wodurch die zum Abfragen benötigte CPUzeit dann nicht vom Benutzerprogramm benötigt wird. Beim Selecting mittels der Betriebssystemfunktion *select* (oder poll je nach UNIX Derivat), wird dieser vorher mitgeteilt, auf welche Ereignisse sie achten soll und bei ihrem Aufruf wählt sie dann jenes, das als erstes ansteht zur Abarbeitung aus, woher auch ihr Name rührt.

Nachteil ist hier, dass selecting nicht funktioniert, wenn die Empfangssoftware wie die GM Library für Myrinet das Betriebssystem umgeht und diesem gar nicht die Kontrolle über seine Ein-/Ausgabefunktionen belässt.

- Posix-Threads (blockierende Funktionen)

Eine gute Alternative zum selecting wäre der Einsatz der, dem vom Institute for Electrical and Electronics Engineers, Inc. (IEEE) Standard Portable Operating System Interface (POSIX)[BrSp99] entsprechenden sogenannten POSIX-Threads welche ebenfalls mit blockierenden Empfangs- bzw. Sendefunktionen arbeiten könnten. Threads sind sogenannte *leichtgewichtige Prozesse*, welche zwar neben dem Hauptthread, gleichzeitig mit diesem laufen können, welche aber nicht von diesem unabhängig sind und z.B. einen gemeinsamen Adressraum haben und mit Beenden des sie ins Leben rufenden Hauptthreades beendet werden.

Beim Gateway wurde das dann so gehandhabt, dass der Hauptthread mit blockierendem Funktionsaufruf auf Nachrichten vom Clienten wartet, derweil ein Thread die Gegenrichtung abhört. Vorteil dieser relativ eleganten und ausbaufähigen Lösung ist seine ebenfalls nur geringe CPU Belastung (bei den durchgeführten Messungen weniger als 2%). Nachteil ist, dass GM nicht threadsicher ist und eine Programmierung mit Einsatz von Threads für den Sonderfall GM einen völlig anderen, weniger überschaubaren Aufbau des Gatewayprozesses erfordert hätte.

3.3 Das Messprogramm hybridpp

Das Programm Hybridpp kann als Beispiel für eine, die hybridnet API benutzende Anwendung gesehen werden. Es ist ein Messprogramm, welche ein einfaches Pingpong zwischen zwei Stationen ausführt und dabei Minimal-, Durchschnitt- und Maximalwerte, sowie die Standardabweichung für die Übertragungszeit und für die Übertragungsgeschwindigkeit (Bandbreite) ermittelt. Hybridpp wurde aus dem Programm *tcppp*[Ei01] entwickelt und ebenfalls aus diesem die Programme *atmpp* und *gmpp*. Diese Programme führen ein Pingpong jeweils für die entsprechenden Protokolle durch: atmpp mit dem Protokoll FOREatm über AAL5, tcppp unter Benutzung der TCP/IP Sockets, gmpp über GM. Konsequenterweise wurden diese Programme und die mit ihnen bestehende Analogie

herangezogen um Vergleichsmessungen durchführen zu können und das Messsystem auf seine Glaubwürdigkeit hin zu überprüfen, die ersten Messungen mit welchen wir uns im nachstehenden Kapitel beschäftigen wollen.

Kapitel 4

Messungen und Auswertung

4.1 Durchführung

4.1.1 Aufbau des Versuchsnetzes

Nachdem die Hybridsoftware und das Messprogramm hybridpp entwickelt und ausgetestet waren, konnten die Messungen durchgeführt werden. Zur Wiederholung, Ziel der Arbeit war es, zu untersuchen, welche Änderungen bezüglich Durchsatz und Latenz beim Einsatz von Low-Level Protokollen in ein heterogenen Netzen, gegenüber dem Einsatz von TCP/IP ergeben. Ein zu untersuchendes Netz muss also aus mindestens zwei auf unterschiedlicher Hardware basierenden Teilnetzen bestehen, wobei für mindestens ein Low-Level Protokolle verfügbar ist.

Ausgewählt wurde die folgende Anordnung (siehe hierzu auch Abbildung4.1):

Im Zentrum der Anordnung befindet sich ein SUN Rechner (Ultra60, 2 CPU s, Rechnername: ZAM065). Sie verfügt über eine 622 Mbits/s ATM-Netzwerkkarte welche mit einem ATM Switch verbunden ist, sowie über eine Myrinetkarte, welche ihrerseits an einem 16 -fach Myrinet Switch hängt. Die SUN fungiert also als Gateway zwischen einem ATM Netz und einem Myrinet Netz und wird daher im folgenden als Gateway SUN bezeichnet. Auf diesem Rechner läuft der von der Hybridsoftware benötigte Gatewayprozess.

Zum ATM Teil des Netzes gehört dann ein SGI Rechner,(O200, Rechnername: ZAM101), welcher direkt über den entsprechenden Switch (auf dem Bild unten links) mit der Gateway SUN verbunden ist. Auf diesem Rechner lief dann ein Prozess des Pingpong Messprogrammes, wahlweise der Client oder der Server.

Auf der Myrinetseite des Netzes befindet sich der PC-Cluster ZAMPANO (Rechnername ZAM008)[7], in der Abbildung dargestellt auf der rechten Seite. ZAMPANO ist eine Abkürzung welche für *Zentralinstitut für Angewandte Mathematik, Parallel Nodes* steht. Genaugenommen, belegt die Gateway SUN nur einen freien Port des 16- fach Switches dieses Parallelrechners. Letzterer besteht aus dem Myrinet-Switch, 8 PCs (4x Intel Pentium III Xeon 550 MHz, 512 KB

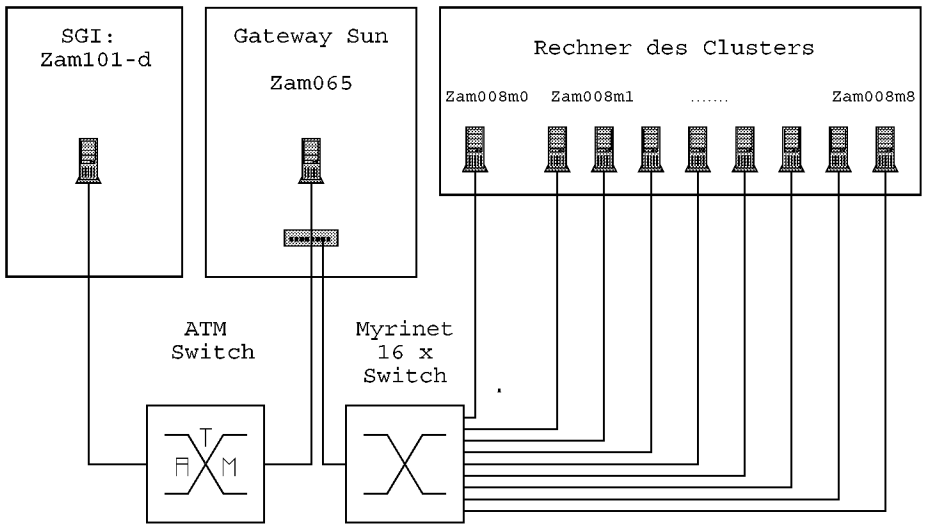


Abbildung 4.1: Versuchsnetz

on-chip cache, Intel 450 NX chipset) welche als Rechenknoten dienen und einem koordinierendem PC. Zur Versuchsdurchführung wurde jeweils ein Knoten reserviert welcher dann exklusiv dem Pingpongmessprogramm zur Verfügung stand, wobei dann hier das entsprechende Gegenstück zu dem Prozess auf der SGI (Server oder Client) lief.

Alle Rechner standen zur Versuchsdurchführung exklusiv den Hybridprozessen zur Verfügung. Der ATM Switch war während der Messungen nur gering belastet, so dass eine Beeinflussung der Messungen durch andere Prozesse oder Datentransfers weitgehend ausgeschlossen war.

Zu sagen bleibt noch, dass die genannten Rechner, neben den Myrinet- bzw. ATM Karten allerdings noch über weitere Technologien (z.B. Ethernet Karten) an weitere Netze verbunden sind. Daher musste während der Durchführung, beim Aufbau von TCP/IP-Verbindungen über Sockets dafür Sorge getragen werden dass dabei auch über die "richtigen" Wege kommuniziert wurde. Dies wurde erreicht durch entsprechende Eintragungen in die Routertabellen der betroffenen Rechner.

4.1.2 Das Messverfahren

Bei einer Pingpong Messung wird von einem Clienten eine Nachricht an einen Server gesendet, welcher die empfangene Nachricht einfach zurücksendet. Das Programm misst dabei die Zeit, welche zwischen dem Losschicken der Nachricht und dem Wiederempfangen verstrichen ist. Aus dieser gemessenen Zeit können nun die Latenzzeit (die Hälfte der gemessenen Zeit) und die Übertragungsgeschwindigkeit (Bandbreite) berechnet werden.

Begonnen wurde mit der Sendung eines Bytes, dann wird die Nachrichtenlänge erhöht und ein erneutes Pingpong durchgeführt. Jede Sendung wurde mehrfach

wiederholt, so dass Mindest-, Höchst- und Durchschnittswerte bestimmt werden konnten.

Durchgeführt wurden nun die folgenden Versuchsreihen:

- Versuche zur Ermittlung des Overheades.
- Messungen auf der ATM Strecke (SGI-SUN)
- Messungen auf der Myrinet Strecke (SUN-ZAMPANO)
- Messungen über die gesamte Strecke (SGI-SUN-ZAMPANO)

Für die Messungen wurden alle Verbindungsparameter auf optimalen Durchsatz eingestellt (Im Falle TCP/IP: ein Tuning aller Sockets) gemäß den Empfehlungen des ZAMinternen Berichtes IB-9634.[LuScWe96]

4.2 Overhead der Hybridnet-Bibliothek

Zunächst galt es zu ermitteln, wie sich der Verwaltungsmehraufwand, welcher für die Hybridnet API betrieben werden musste, auswirken könnte. Hierzu wurden dann die Zeitmessungen, welche unter Ausnutzung der Hybridnet Bibliothek durch das Programm hybridpp erfolgten, verglichen mit Messungen, welche unter identischen Bedingungen (soweit dieses möglich ist) mit den Pingpongprogrammen (atmpp, gmpp, tcppp) durchgeführt wurden. gmpp und atmpp greifen direkt auf die entsprechenden APIs (GM, XTI) zu. Da hybridpp, genauso wie atmpp und gmpp aus dem bereits vor dieser Arbeit bestehendem Programm tcppp entwickelt worden waren, sind die Programme vom Prinzip her identisch, von der letztendlichen Implementierung her sehr ähnlich und es stand daher zu erwarten, dass wenn sich Abweichungen ergeben sollten, diese so gut wie ausschliesslich auf den Overhead für die Hybridnetbibliothek zurückzuführen sein werden.

Wie aus der in Abbildung 4.2 gezeigten Gegenüberstellung der gemessenen Latenzzeiten im Falle einer Übertragung mit TCP/IP ersichtlich ist, dauert ein Pingpong mit hybridpp zwar etwas länger als mit tcppp, doch verbleibt dieser Unterschied doch erheblich unterhalb der sich einstellenden Fehlertoleranz (Abweichung von Minimal- und Maximalübertragungszeit).

Somit muss festgehalten werden , dass ein Overhead der Bibliothek nicht feststellbar ist. Um bestehenden Unterschiede leichter ablesbar zu machen als in Graphik 4.2 mit ihren zwei fast verschmolzenen Kurven, wurde noch die Abbildung 4.3 angefertigt. Aufgetragen gegen die Nachrichtenlänge ist hier die Differenz der Zeitmessungen der beiden Pingpongprogramme hybridpp und tcppp.

Bleibt noch nachzutragen, dass sich über die Messungen zu GM und XTI dasselbe sagen läßt!

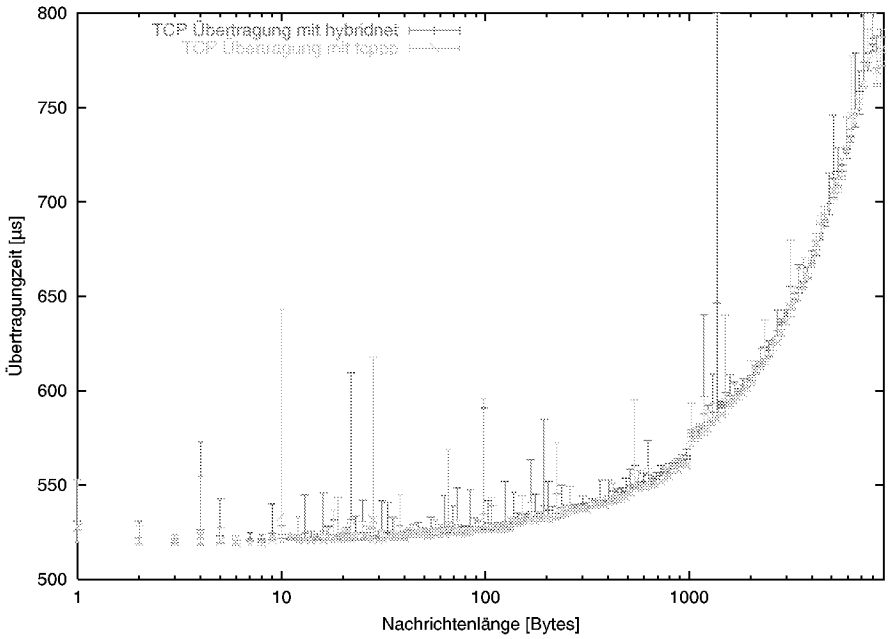


Abbildung 4.2: Vergleich der Pingpongprogramme hybridpp mit tcppp

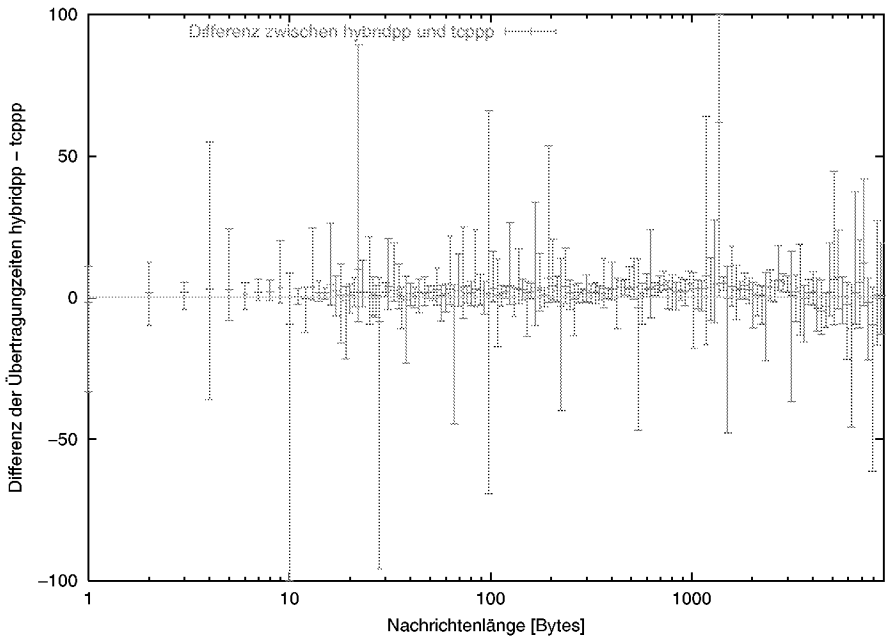


Abbildung 4.3: Zeitdifferenzen zwischen hybridpp und tcppp

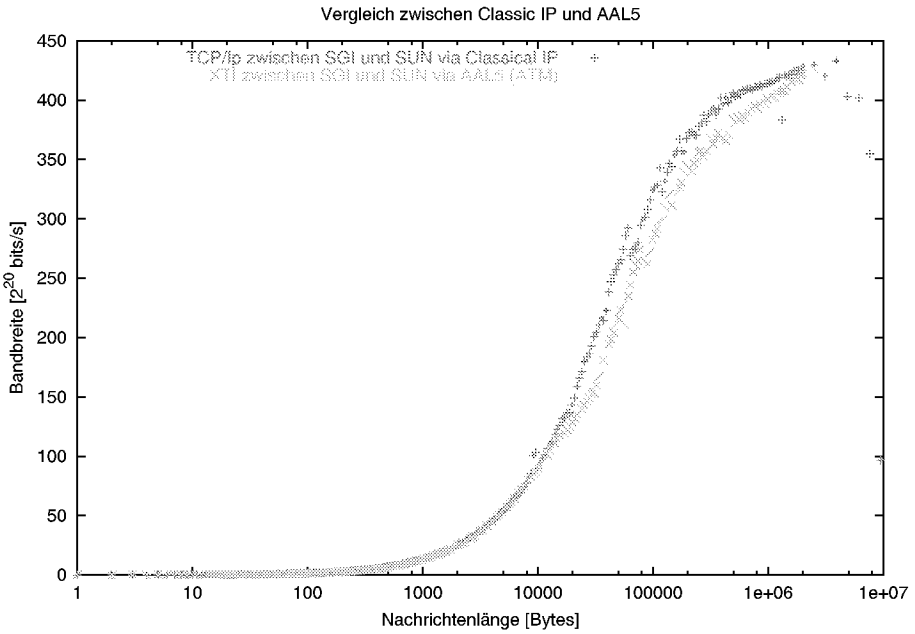


Abbildung 4.4: Bandbreite, Vergleich zwischen Classical IP und XTI auf AAL5 über ATM

4.3 ATM: Vergleich zwischen FOREatm und CLIP

Auf der Strecke zwischen der SGI und der Gateway Sun wurden mit dem Programm hybridpp die Daten ermittelt,

- die Übertragung unter Benutzung der API FOREAtm XTI, welche direkt auf den AtmAdapterLayer5 aufsetzt und zum Vergleich
- dieselbe Strecke über die klassischen Sockets (TCP/IP über Classical IP)

Das Ergebnis für die mittlere Bandbreite hält die Graphik in Abbildung 4.4 fest.

Dieses Ergebnis ist nun überraschend. Es zeigte sich, dass der Durchsatz beim Einsatz des Low-Level Protokoll mittels XTI sogar noch unter demjenigen liegt, der erzielt wird beim Einsatz von TCP/IP über Classical IP.

Auch kann man nicht sagen, dass sich bezüglich der Latenz mit XTI (526 Mikrosekunden) eine erhebliche Verbesserung gegenüber Classical IP (554 Mikrosekunden) erzielen läßt.

Erschwerend kommt noch hinzu:

Ähnlich wie bei dem bekannten Transportschichtprotokoll UDP ist bei dieser Implementierung von XTI nicht sichergestellt, dass alle Pakete ankommen und dass dabei die Reihenfolge erhalten bleibt. Wegen der geringen Belastung der Systeme traten jedoch erst dann Paketverluste auf, wenn die Nachrichtenlänge die Größe des Empfangspuffers (hier auf 1 MByte eingestellt) überschreitet.

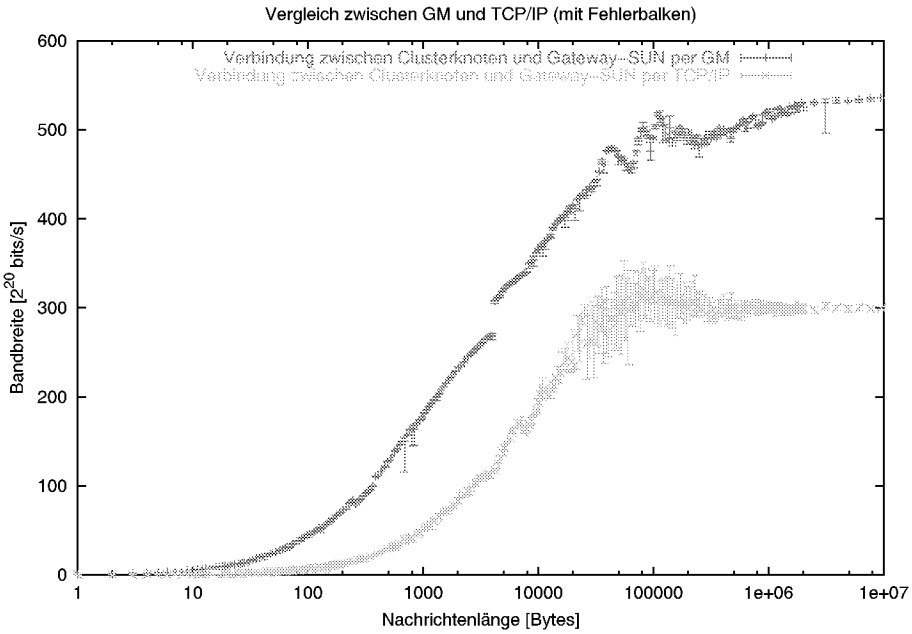


Abbildung 4.5: Ergebnisse Myrinet mit Fehlerbalken

Wohl sieht die Spezifikation für XTI ein sogenanntes *special secure connection oriented protocol* (SSCOP)[Ky95] vor, diese ist aber bei der vorliegenden Version (FOREatm 5.1) nicht implementiert. Auf eine eigene Implementierung zur Sicherstellung der korrekten Ankunft wurde aber verzichtet, weil die mit dem Einsatz des Low-Level-Protokoll erzielte Bandbreite ohnehin unter derjenigen von TCP/IP zurückbleibt.

4.4 Myrinet: Vergleich zwischen TCP/IP und GM direkt

Die grössten Unterschiede zwischen dem Einsatz eines Low-Level Protokolls gegenüber der TCP/IP zeigen sich im Falle Myrinet. Ausgemessen wurde hier die Strecke zwischen der Gateway SUN und einem Knoten des ZAMPANO. (siehe Abb. 4.5) Die Bandbreite aufgetragen gegen die Länge der verschickten Nachrichten, diese im logarithmischen Maßstabe, zeigen auch hier in beiden Fällen (GM und TCP/IP) ein zunächst nur langsames Angwachsen der Bandbreite, in einem mittleren Bereich starkes Wachstum welches anschliessend in die Sättigung übergeht. Jedoch lassen sich mit GM erheblich höhere Bandbreiten erzielen (etwa im Sättigungsbereich um 520 Mbits/s bei GM gegenüber 300 Mbits/s bei TCP/IP).

Auffällig ist die weit geringere Schwankung der Messwerte bei GM.

Die Kurve bei GM weist zwei Stellen auf, an denen es zum sprunghaften Anstieg der Bandbreite kommt:

1) bei 32000 Bytes. Dies ist die Grösse des Empfangspuffers der Hybridnetansteuerung für GM. Ist dieser Puffer voll, die Nachricht aber länger muss hybridnet sie fragmentieren. Üblicherweise werden die Daten aus dem GM-Puffer in den Puffer des aufrufenden Programm kopiert , schliesslich hatte dieses sie ja angefordert.

Die für eine Datenübertragung benötigte Zeit hat zwei Anteile:

1. die Zeit in welcher eine Nachricht auf dem Netz unterwegs ist und
2. der benötigten Rechenzeit.

Rechenzeit fällt hauptsächlich dazu an die empfangene Nachricht aus dem Puffer in den Hauptspeicher zu kopieren.

Wird nun eine Nachricht fragmentiert, so kann während der Zeit in der die vorangegangene Nachricht noch auf dem Netz unterwegs ist, der Puffer kopiert werden. Somit ist die für den gesamten Vorgang benötigte Zeit kürzer, als wenn beide Vorgänge erst nacheinander abgearbeitet werden müssten, wie es ja bei der unfragmentierten Nachricht der Fall ist! Das schlägt sich in einen Anstieg der Bandbreite nieder.

Diese Erklärung wird dadurch gestützt dass, wenn man durch eine Änderung des Programmes dieses Kopieren unterdrückt, die Kurve wieder glatt verläuft!

2) bei 4096 Bytes, soviel beträgt die gm_mtu, die Maximum Transfer Unit, die maximale Paketlänge für GM. Möglicherweise liegt dem Anstieg der Bandbreite an dieser Stelle ein ähnlicher Mechanismus zugrunde. Ohne eine genaue Untersuchung der GM Implementierung, die den Rahmen dieser Arbeit sprengen würde, läßt sich das jedoch nicht verifizieren.

Bezüglich der Latenzzeiten lohnt ein Einsatz von GM auf jeden Fall, denn mit 13.4 Mikrosekunden bei Direktansteuerung von GM gegenüber 107 Mikrosekunden, wenn GM noch TCP/IP aufgesetzt wird.

4.5 Der Gateway im Einsatz

Kommen wir schliesslich zu den von der Aufgabenstellung suggerierten Messungen, bei welchem wir das implementierte Gateway einsetzen, um die Teilnetze zu koppeln:

Wären die Ergebnisse der Messungen zu ATM anders ausgefallen, wäre die interessanteste der zu untersuchenden Kombinationen möglicher Netzwerkverbindungen sicherlich jene gewesen, von der SGI mittels XTI auf AAL5 über ATM zur Gateway SUN, vom Gatewayprozess Weiterleitung mittels GM über Myrinet zu einem Clusterknoten. Doch da die vorangegangenen Messungen besagten, dass XTI grosse Datenpakete nicht gesichert zustellen kann, wird die Untersuchung dieser Kombination unterlassen.

Verbleibt uns der andere Fall, in dem wir noch Low-Level Protokolle einsetzen können:

Eine Verbindung von der SGI mittels Socketaufrufe über TCP/IP auf Classical IP, dieses auf AAL5 über ATM zur Gateway SUN, Weiterleitung durch den Gatewayprozess mittels GM über Myrinet zu einem Clusterknoten.

Aufgetragen sind wieder die erzielten Bandbreiten gegen die logarithmisch skalierte Nachrichtenlänge. (siehe Abb. 4.6)

Bei diesen beiden Kurven zeigen sich erhebliche Unterschiede:

Die Kurve für TCP/IP verläuft qualitativ nicht anders als die entsprechenden Kurven auf den einzelnen Teilstrecken, aber von den Werten her betrachtet, bleibt sie mit 250 Mbits/s stark unter den Werten für die Teilstrecken (450 MBits/s für ATM und 300 Mbits/s für Myrinet) zurück. Dies war zu erwarten, da für das Routing in der Gateway SUN ebenfalls Zeit benötigt wird.

Anders verhält es sich mit der Kurve, welche unter Einsatz des Gateway aufgezeichnet werden konnte:

Während sich bei kurzen Nachrichtenlängen kaum Unterschiede zu TCP/IP bemerkbar machen, bleiben die mit der Gatewaylösung erzielten Werte im Bereich von 10.000 Bytes bis 80.000 etwas unter denen mit TCP/IP Routing zurück. Doch während letztere Kurve bereits in die Sättigung übergeht, steigen die Werte für die Gatewaylösung immer noch an und ab 100.000 Bytes Nachrichtenlänge ist die Variante mit Einsatz des Low-Level Protokolles der reinen TCP/IP Lösung klar überlegen!

Wenn nur die erste Hälfte der Strecke mit TCP/IP betrieben wird (weil wie wir gesehen haben, kein lohnendes Low-Level Protokoll zur Verfügung steht), unter Vermittlung des Gateway dann die zweite auf Myrinet mit GM zurückgelegt werden kann, ist als mittlere Bandbreite für große Nachrichtenlängen 350 Mbits/s zu erreichen, derweil Übertragung mit reinem TCP/IP Routing auf derselben Strecke im Mittel, für große Nachrichtenlängen nur 250 Mbits/s zu erzielen sind.

Ein Vergleich der Latenzeiten ergibt folgendes Bild:

direktdurchschaltung mit TCP/IP: 566 Mikrosekunden

mit Einsatz des Gateway:557 Mikrosekunden

4.6 Ein Blick auf die Latenzzeiten

In den vorangegangenen Betrachtungen lag der Schwerpunkt auf den erzielten Durchsätzen, schliesslich wurde diese Arbeit unternommen, um nach Möglichkeiten der Performancesteigerung Ausschau zu halten. Dabei dürfen aber Latenzzeiten nicht ausser Acht gelassen werden und so soll zum Ausklang dieser Diskussion die Aufmerksamkeit noch mal auf die Zusammenstellung (in Abbildung 4.7) der in den einzelnen oben beschriebenen Versuchen erzielten Latenzzeiten gelenkt werden.

Auffällig ist vor allem der große Unterschied zwischen GM und TCP/IP über Myrinet, der auf eine nicht optimale Implementierung des IP Protokolls durch Myricom hindeutet.

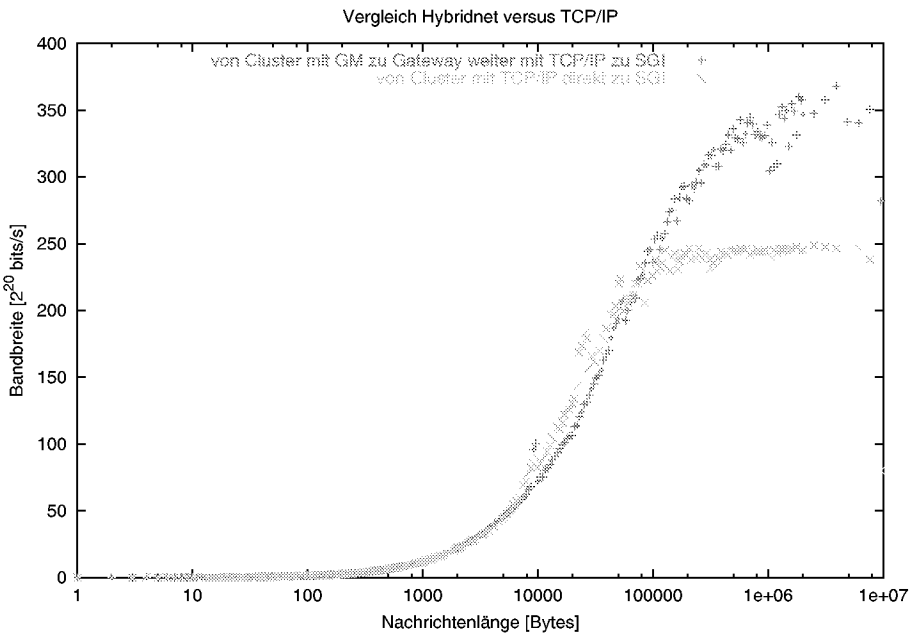


Abbildung 4.6: Ergebnisse Einsatz von Gateway und Low-level Protokoll GM

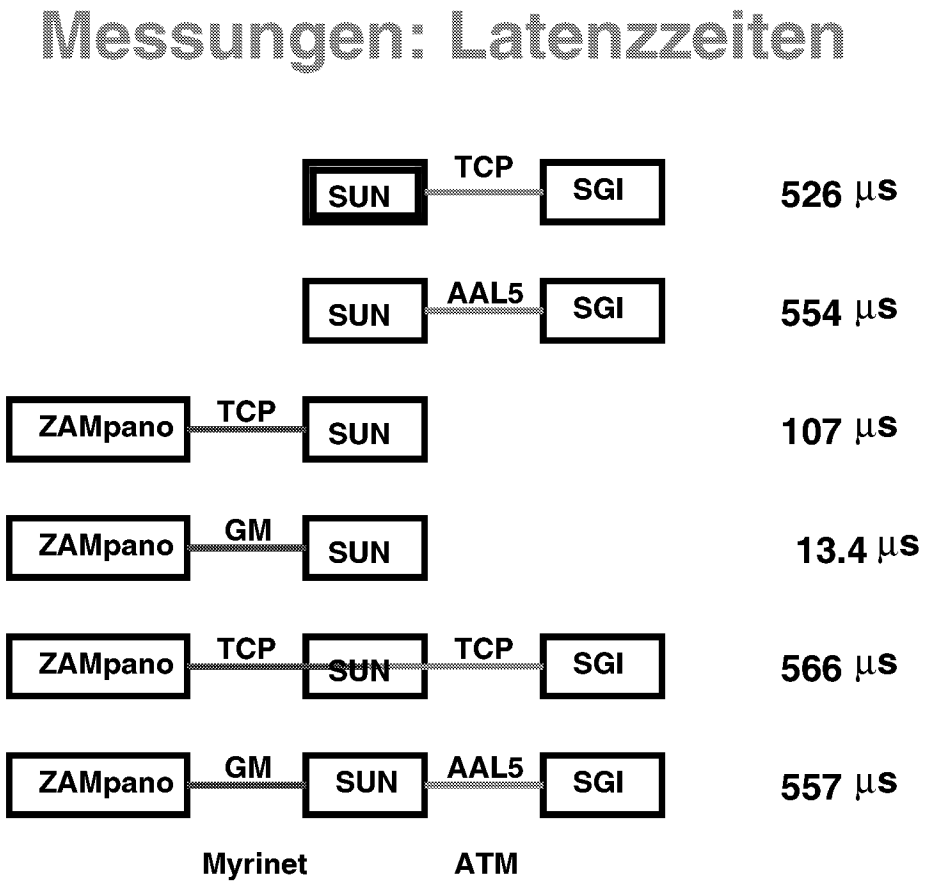


Abbildung 4.7: Zusammenstellung: Latenzzeiten

Kapitel 5

Schlussbetrachtung

5.1 Zusammenfassung

Im Rahmen dieser Arbeit wurde eine Softwarebibliothek entwickelt, die eine einheitliche API zur Verfügung stellt, um Rechnern in heterogenen Netzwerken eine Kommunikation unter Einsatz von Low-Level Protokollen oder TCP/IP zu ermöglichen. Diese Bibliothek ist modular aufgebaut und es wurden die Module für die Kommuniktion mittels der Protokolle FOREatm XTI über ATM und mittels GM über Myrinet implementiert. Dank des modularen Aufbaus ist die Bibliothek ohne großen Aufwand um weitere Protokolle erweiterbar. Zu denken wäre hier sicherlich an das *High Performance Parallel Interface* (HiPPI), einer Netzwerktechnologie, die entwickelt wurde um Supercomputer mit externen Speichersubsystemen zu verbinden und die im Cray-Komplex des ZAM im Einsatz ist.

Ebenfalls entwickelt wurde ein Gatewayprozess, welcher an der Schnittstelle zweier Teilnetze unterschiedlicher Technologie die Protokollumsetzung über die entsprechenden API durchführt und den Datenaustausch über Technologiegrenzen hinweg im heterogenen Netzwerk unter Einsatz von Low-Level Protokollen ermöglicht.

Als Ergebnis der Messungen kann festgehalten werden, dass bei Benutzung von GM statt TCP/IP deutlich höhere Bandbreiten und geringer Latenzen erzielt werden können. Der Durchsatz war so deutlich besser, dass es sich unter diesem Gesichtspunkt lohnt, GM auch dann einzusetzen wenn dabei nur die Kommunikation auf der Teilstrecke vom Rechnerknoten bis zum Gatewayrechner mit einem Low-Level Protokoll betrieben und die restliche konventionell über das TCP/IP abgewickelt wird. Spätestens ab Nachrichtenlängen grösser als 100.000 Bytes kann in dieser Konstellation eine deutlich bessere Bandbreite erzielt werden.

Demhingegen lohnt der Einsatz von FOREatm XTI auf AAL5 nicht:

- es lassen sich keine höheren Bandbreiten als mit TCP/IP auf CLIP auf AAL5 erzielen und

- die Kommunikation ist nicht gesichert, weiterer Aufwand wäre erforderlich um dem abzuhelfen.

5.2 Ausblick

- Einsatz finden könnte die entwickelte Software bei der Kopplung von Clustern bei dem geplanten RePhoNet mittels MetaMPI.[EiGrHe99]

- Bei der am ZAM entwickelten Software Visualisation Interface Toolkit (VISIT)[EiFri00], soll die Kommunikation durch die geplante Verwendung der Hybridnet Software beschleunigt werden. Bei VISIT handelt es sich um eine Software, welche die Ergebnisse von Simulationen in Echtzeit graphisch darstellen kann, während diese Simulation noch läuft und dann steuernd eingreifen. Damit die Visualisierung in Echtzeit erfolgen kann sind, natürlich die grösstmöglichen Bandbreiten zu wählen, die sich erzielen lassen.

Kapitel 6

Anhang

Implementierungsbeispiel:

Das gesamte Softwarepaket Hybridnet ist mit 4848 Zeilen Code zu umfangreich um es hier in Gänze abzubilden, deshalb wurde nur ein einzelnes Modul ausgewählt, welches hier vorgestellt wird.

Es handelt sich um das Modul zur Realisierung der Ansteuerung der besonders interessierenden GM API für die Kommunikation über Myrinet durch Hybridnet:

HeaderFile:

```
/*
* this is the "gm_hybrid.h" Module. implementations for
* connections to be made by using
* the GM Library (Myrinet)
*
* Headerfile for gm_hybrid.c
*/

struct gmconn h_gm_connectclient (struct seapservice *service,int flag);
struct gmconn h_gm_connectserver (struct seapservice *service,int flag);

int h_gm_close(struct gmconn portd);
int h_gm_send(gmconn portd, char *buffer, int len);
int h_gm_recv(gmconn portd, char *buffer, int len);

struct gmconn h_gm_startaserver(struct seapservice *service,int flag);
struct gmconn h_gm_accept (gmconn portd);
```

Hauptmodul:

```
/* Module for hybridnet, implementing the connections to be made on Myrinet
 * using the GM-Library
 */
#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>

#include <string.h>
#include <strings.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <fcntl.h>
#include <netdb.h>
#include <pwd.h>
#include <errno.h>
#include <assert.h>
#include <gm.h>

/* eigene includes */
#include "constants.h"
#include "gm_hybrid.h"

#define MAX_SIZE    15
#define HIGH_SIZE    4
#define MAXPORT_ID   8


#ifndef GM_N
#define GM_N(x) (x)
#endif



#define MY_RECV_FUNCTION gm_receive        /* 14 * 10^-6 sec latency */



/* Noch Definitionen (nicht in "gm_hybrid.h"  weil gm benoetigt */
void _initialize_gm(char* servicename, struct gmconn *returnvalue);
void sent_cb  (struct gm_port *port, void *context, gm_status_t status) ;
int  h_gm_s_send   (gmconn portd, void *buf, int size, unsigned int priority);
int  h_gm_s_recv_nb(gmconn portd, void *buf, int size);
void h_gm_s_recv   (gmconn portd, void *buf, int size);
```

```
struct sent_cb_arg {
  int done;
  unsigned int priority;
};

/* Globale Variablen */
static int max_chunk=0;           /* chunck dt. Brocken */
static void *send_buf;


/* ab hier: implementierungen */
/* Client */
struct gmconn h_gm_connectclient (struct seapservice *service,int flag) {
  struct gmconn returnvalue;
  unsigned char protocol[MAXWORD];
  int32_h  no_local_node_id;
  int32_h  no_port_id;

  returnvalue.port=NULL;
  /* erfrage Port_id und Remote_node_id des anzusteuernden Servers */
  if( ! seap_queryf(service->name,service->password,1,-1,"%s:%d:%d",
                    protocol,&(returnvalue.remote_node_id),&(returnvalue.remote_port_id))
    printf("seap-query failed\n");
    returnvalue.remote_port_id=-1;
  }
  returnvalue.blockflag = flag;
  returnvalue.port_id = returnvalue.remote_port_id;

  if (verbose>=IONIC) printf("Read from SEAP:\n");
  if (verbose>=IONIC) printf(" remote_node_id %i:\n", returnvalue.remote_node_id);
  if (verbose>=IONIC) printf(" remote_port_id %i:\n", returnvalue.remote_port_id);
  if (verbose>=IONIC) printf(" port_id        %i:\n", returnvalue.port_id);
  if(returnvalue.port_id!=-1)
    _initialize_gm(service->name,&returnvalue);
  /* Dem Server die eigene node_id Senden */

  no_local_node_id = gm_htonl(returnvalue.local_node_id);
  if (verbose>=ATTIC) printf("Sending my Node_id %i\n",returnvalue.local_node_id);
  h_gm_s_send(returnvalue, &no_local_node_id, sizeof(int32_h ), GM_LOW_PRIORITY);

  no_port_id       = gm_htonl(returnvalue.port_id);
  if (verbose>=ATTIC) printf("Sending my Port_id %i\n",returnvalue.port_id);
  h_gm_s_send(returnvalue, &no_port_id, sizeof(int32_h ), GM_LOW_PRIORITY);

  return(returnvalue);
}
/* Server */
struct gmconn h_gm_connectserver (struct seapservice *service,int flag) {
  struct gmconn returnvalue;
```

```
  returnvalue= h_gm_accept(h_gm_startaserver(service,flag));
  return(returnvalue);
}

struct gmconn h_gm_startaserver(struct seapservice *service,int flag) {
  struct gmconn returnvalue;
  struct gm_port *port = NULL;

  returnvalue.blockflag = flag;
  returnvalue.port=(struct gm_port *)service->u.portd.port;
  returnvalue.port_id = service->u.portd.port_id;

    _initialize_gm(service->name,&returnvalue);
  if (verbose)
    printf("returnvalue.port_id         %i\n",returnvalue.port_id);
  if (verbose)
    printf("returnvalue.remote_node_id  %i\n",returnvalue.remote_node_id);
  if (verbose)
    printf("returnvalue.local_node_id   %i\n",returnvalue.local_node_id);
  if(seap_publishf(service->name, service->password,
                   "%s:%d:%d",H_protocolname_GM,
                   returnvalue.local_node_id,
                   returnvalue.port_id)
     !=1)
    {
      fprintf(stderr,"seap_publishf (myri) failed\n");
      returnvalue.port_id=-1;

    }
  else
   if (verbose) printf("Started a GM service: '%s' on port %x at node %i \n",
                       service->name,
                       returnvalue.port_id,
                       returnvalue.local_node_id);
  return(returnvalue);
}

struct gmconn h_gm_accept (gmconn portd) {
  /* since GM is connectionless
   * a connection will be considered
   * as established, upon reception of
   * the peers node_id */

  struct gmconn returnvalue;
  int32_h no_remote_node_id;
  int32_h no_remote_port_id;

  returnvalue.blockflag     = portd.blockflag;
  returnvalue.port_id       = portd.port_id ;
  returnvalue.local_node_id = portd.local_node_id ;
```

```
  returnvalue.port          = portd.port ;
  h_gm_s_recv(portd,&no_remote_node_id,sizeof(int32_h));
   if (verbose>=ATTIC)
     printf(" gelesen : remote_node_id %i\n", gm_ntohl(no_remote_node_id));
  h_gm_s_recv(portd,&no_remote_port_id,sizeof(int32_h));
   if (verbose>=ATTIC)
     printf(" gelesen : remote_port_id %i\n", gm_ntohl(no_remote_port_id));
  returnvalue.remote_node_id= gm_ntohl(no_remote_node_id);
  returnvalue.remote_port_id= gm_ntohl(no_remote_port_id);

  if (verbose) printf("connection from Node %i Port_id %i accepted\n",
                      returnvalue.remote_node_id,returnvalue.remote_port_id );

  return(returnvalue);
}

/* Close */
int h_gm_close(struct gmconn portd) {
  struct gm_port *port = portd.port;

  gm_close(port);
  gm_finalize();
  return(0);
}
/* Senden */
int h_gm_send(gmconn portd, char *buffer, int len){
  return(h_gm_s_send(portd,buffer,len, GM_LOW_PRIORITY));
}
/* Empfangen */
int h_gm_recv(gmconn portd, char *buffer, int len){
  int bytes;

  if (portd.blockflag & H_NONBLOCKING)
    return(h_gm_s_recv_nb(portd,buffer,len));
  else
    while ((bytes=h_gm_s_recv_nb(portd,buffer,len))==0);
  return(bytes);
}


void _initialize_gm(char* servicename,struct gmconn *returnvalue) {
  int i;
  gm_status_t g_errno;
  struct gm_port *port;
  char *buf;

  /* initialization stuff
   *
   * GM provides stateless peer-to-peer communication,
   * there is no distinction between client and server.
```

```
 *
 * we let one process wait for an initial message from anywhere
 * and call that the server, and
 * let one process send an initial message to the server (and call
 * that the client)
 */
/* initialize GM */

port = returnvalue->port;
if( gm_init() != GM_SUCCESS ) {
  fprintf(stderr,"Error: gm_init failed\n");
  exit(1);
}

/* open a port == 'communication end-point' */
while( (g_errno = gm_open(&port, 0, returnvalue->port_id,
       servicename, GM_API_VERSION_1_1))
       != GM_SUCCESS)  {

  if ((g_errno != GM_BUSY) || (returnvalue->port_id > MAXPORT_ID)) {
    fprintf(stderr,"Error: could not open port \n");
    gm_perror("",g_errno);
    gm_exit(1);
  }
  else
    returnvalue->port_id++;

}

/* tell GM, which message-sizes are allowed */
  gm_set_acceptable_sizes(port, GM_LOW_PRIORITY, (1<<MAX_SIZE));

  /* tell GM that it may use all available send tokens */
  gm_free_send_tokens(port, GM_LOW_PRIORITY, gm_num_send_tokens(port)-1);

  /* max payload of our buffers */
  max_chunk = gm_max_length_for_size(MAX_SIZE);

  /* create a pool of receive buffers, and let GM manage them */
  for(i=0;i<10;i++) {
    buf = gm_dma_calloc(port, 1, max_chunk);
    assert(buf);
    gm_provide_receive_buffer_with_tag(port, buf, MAX_SIZE, GM_LOW_PRIORITY, i+1
    if(verbose >= IONIC )
      printf("h_initialize: providind obuf=%x size=%d tag=%d\n",buf,MAX_SIZE,i+1
  }

  /* create a send buffer */
  send_buf = gm_dma_calloc(port, 1, max_chunk);
  assert(send_buf);
```

```
  if (verbose>=IONIC) {
    printf("buffer of size %d bytes allocated\n",
           (int)gm_max_length_for_size (MAX_SIZE));
    printf("send_buf (send) = %x\n",(int)send_buf);
  }

  /* get out own node-id (the server does not really need this) */
  if( gm_get_node_id(port, &(returnvalue->local_node_id)) != GM_SUCCESS) {
    fprintf(stderr,"Error: could not determine local node-id\n");
    gm_exit(1);
  }
  if(verbose) printf("local node id is %d\n",returnvalue->local_node_id);

  returnvalue->port=port;  /*     parameteruebergabe                */
  if (verbose>=IONIC)
    printf("gm_init returnvalue.port= %x port %x\n",
           (int) returnvalue->port,port);
  /* hier kein Return, weil Funktion ist void */
}


/* has call back been send ? */
void sent_cb (struct gm_port *port, void *context, gm_status_t status) {
  struct sent_cb_arg *arg = context;

  if (status != GM_SUCCESS) {
    gm_perror ("send failed", status);
    gm_exit(1);
  }
  gm_free_send_token(port, arg->priority);
  if(verbose>=IONIC)
    printf("sent_cb: finished with priority %d\n",arg->priority);
  arg->done=1;
}


/*
 * a blocking send
 */
int h_gm_s_send(struct gmconn portd,
                void *buf, int size,
                unsigned int priority) {
  struct gm_port *port = NULL;
  int nsend;
  int bytes_sent =0;
  struct sent_cb_arg arg;

  port =(struct gm_port *) portd.port;
  if(verbose >= IONIC) {
```

```
    int i,n;

    printf("Sending %d bytes on port %x to remote_node_id %i, remote_port_id %i\
           , size, port,portd.remote_node_id,
           portd.remote_port_id);
    n = size;
    if(n>20) n = 20;
    for(i=0;i<n;i++) {
      printf("%d ",(int)*(((char *)buf)+i));
    }
    printf("\n");
  }
  do {
    arg.done = 0;
    arg.priority = priority;

    nsend = size > max_chunk ? max_chunk : size;

    memcpy(send_buf, buf, nsend);
    if( ! gm_alloc_send_token(port, priority) ) {
      fprintf(stderr,"Error: could not get send token for priority %d\n",
              priority);
      gm_exit(1);
    }
    gm_send_with_callback(port, send_buf, MAX_SIZE, nsend, priority,
                          portd.remote_node_id, portd.remote_port_id,sent_cb, &a


    /* loop until sent_cb sets done to 1 */
    while( !arg.done ) {
      gm_recv_t *e;
      e = (gm_recv_t *)MY_RECV_FUNCTION(port);

      gm_unknown(port,(union gm_recv_event *)e);
    }
    size -= nsend;
    buf  += nsend;
    bytes_sent += nsend;
  } while(size > 0);
  return( bytes_sent);
}

/*
 * a non-blocking receive:
 *
 * receive at most size bytes into buf, returns
 * the number of bytes received or -1 on error
 */

int h_gm_s_recv_nb(gmconn portd, void *buf, int size) {
```

```
int nrecv;
gm_recv_event_t *e;
struct gm_port *port = NULL;
static char *obuf=NULL;
static int obuf_head=0, obuf_tail=0;
static unsigned int obuf_size,obuf_tag;

/* initialize obuf at first call */

/* return data from obuf if available */
if( obuf_tail > obuf_head ) {
  nrecv = obuf_tail - obuf_head;
  if( nrecv > size ) {
    nrecv = size;
    memcpy(buf, obuf+obuf_head, nrecv);
  } else {
    memcpy(buf, obuf+obuf_head, nrecv);
    /* give buffer back to GM */
    if(verbose >=IONIC)
      printf("h_gm_s_recv: giving back obuf=%x size=%d tag=%d\n"
             ,obuf,obuf_size,obuf_tag);
    gm_provide_receive_buffer_with_tag(portd.port, obuf,
                                       obuf_size,
                                       GM_LOW_PRIORITY,
                                       obuf_tag);
  }
  obuf_head += nrecv;
  if(verbose >=IONIC)
    printf("h_gm_s_recv: returning %d bytes from buffer (%d requested)\n"
           ,nrecv,size);
  return nrecv;
}

/* obuf is empty */
obuf_head = obuf_tail = 0;

port = portd.port;
e = MY_RECV_FUNCTION(port);
if((verbose>=IONIC) && (GM_N(e->recv.type)!=0))
  printf("h_gm_s_recv: event type %d\n", GM_N(e->recv.type));

switch(GM_N(e->recv.type)) {

case GM_RECV_EVENT:
  nrecv = gm_ntohl(e->recv.length);

  if( nrecv > size ) {
    if(verbose >=IONIC)
      printf("h_gm_s_recv: received %d byte (%d requested)\n",nrecv,size);
```

```
        memcpy(buf, gm_ntohp(e->recv.buffer), size);

        /* memcpy(obuf, gm_ntohp(e->recv.buffer) + size, nrecv-size); */
        obuf = gm_ntohp(e->recv.buffer);
        obuf_size =  GM_N(e->recv.size);
        obuf_tag = GM_N(e->recv.tag);

        if(verbose >=IONIC)
          printf("h_gm_s_recv: obuf=%x size=%d tag=%d\n"
                 ,obuf,obuf_size,obuf_tag);
        obuf_head = size;
        obuf_tail = nrecv;

        nrecv = size;
      } else {
        memcpy(buf, gm_ntohp(e->recv.buffer), nrecv);
        /* give buffer back to GM */
        if(verbose >=IONIC)
          printf("h_gm_s_recv: immediately giving back obuf=%x size=%d tag=%d\n",
                 gm_ntohp(e->recv.buffer),
                 GM_N(e->recv.size),
                 GM_N(e->recv.tag));
        gm_provide_receive_buffer_with_tag(port, gm_ntohp(e->recv.buffer),
                                           GM_N(e->recv.size),
                                           GM_LOW_PRIORITY,
                                           GM_N(e->recv.tag));

      }
      break;
    case GM_NO_RECV_EVENT:
      nrecv = 0;
      break;
    default:
      if(verbose >=IONIC)
        printf("h_gm_s_recv: unknown event type %d\n",GM_N(e->recv.type));
      gm_unknown(port,(union gm_recv_event *)e);
      nrecv = 0;
    }
    /* ...... */
    if(nrecv && (verbose >= IONIC)) printf("h_gm: bytes_recieved %i\n",nrecv);
    return nrecv;
  }

  /*
   * a blocking receive that reads exactly size bytes
   */
  void h_gm_s_recv(gmconn portd, void *buf, int size) {
    int nrecv;

    while(size > 0 ) {
```

```
    nrecv = h_gm_s_recv_nb(portd,buf, size);

    if((verbose>=IONIC) && (nrecv>0))
      printf("h_gm_s_recv: nrecv = %d (size=%d)\n",nrecv,size);
    if( nrecv < 0 ) { exit(1); }

    size -= nrecv;
    buf  += nrecv;
  }
}
```

Abbildungsverzeichnis

1.1 Gigabit Testbed West:2 Parallelrechner über ATM-WAN verbunden 2

1.2 RePhoNet : PC-Cluster über WAN verbunden 2

1.3 Durchlaufen des Protokollstapel bei Kommunikation mit TCP/IP 4

1.4 Durchlaufen des Protokollstapel bei Kommunikation beim Einsatz eines Protokollumsetzers 4

3.1 schematische Darstellung der Kommunikation über Hybridnet . . 18

3.2 Verbindungsaufbau und ein Datenaustausch mit Hybridnet . . . 22

4.1 Versuchsnetz 28

4.2 Vergleich der Pingpongprogramme hybridpp mit tcppp 30

4.3 Zeitdifferenzen zwischen hybridpp und tcppp 30

4.4 Bandbreite, Vergleich zwischen Classical IP und XTI auf AAL5 über ATM 31

4.5 Ergebnisse Myrinet mit Fehlerbalken 32

4.6 Ergebnisse Einsatz von Gateway und Low-level Protokoll GM . . 35

4.7 Zusammenstellung: Latenzzeiten 36

Literaturverzeichnis

[BrSp99] MANFRED BROY, OTTO SPANIOL: *VDI Lexikon Informatik und Kommunikationstechnik*, 2. Auflage Springer Verlag 1999

[Ke93] HELMUT KERNER: Rechnernetze nach OSI, 2. Auflage Addison-Wesley 1993

[Ste98] W. RICHARD STEVENS: *Unix Network Programming* Volume 1, 2. Edition Prentice Hall, 1998

[Ei00] THOMAS EICKERMANN (Hrsg.): *Gigabit Testbed West - Abschlußbericht des DFN-Projektes*, Interner Bericht [FZJ-ZAM-IB-2000-17] des ZAM am Forschungszentrum Jülich, 2000

[Zie01] WOLFGANG ZIEGLER: *Netzwerktechnologien*, Seminar der Fernuniversität Hagen, 1.7.2001

[HeLa97] MATHIAS HEIN, NIKOLAUS VON DER LANCKEN: ATM - Konzept Trends Migration, International Thomson Publishing 1997

[Le98] VOLKER LEMPERT: Methoden und Untersuchungen zur Optimierung des Datendurchsatzes in ATM-Netzen, Diplomarbeit am ZAM am Forschungszentrum Jülich, 1998

[Ky95] OTHMAR KYAS: *ATM Netzwerke - Aufbau - Funktion-Performance*, 2. Auflage, Datacom Verlag 1995

[BoCoFeKuSeSeSu95] NANETTE BODEN, DANNY COHEN, ROBERT FELDERMAN, ALAN KULAWIK, CHARLES SEITZ, JAKOV SEIZOVIC, WEN-KING SU: *Myrinet - A Gigabit per Second Local Area Network*, IEEE Micro, 1995

[OlBaAvNa99] FABIO DE OLIVEIRA, MARCOS BARRETO,RAFAEL AVILA,PHILIPPE NAVAUX: *A comparative Study on Low-level APIs for Myrinet and SCI-based Clusters*, Institute of Informatics - Federal University of Rio Grande do Sul Porto Alegre 1999

[Mpi95] MESSAGE PASSING INTERFACE FORUM: *MPI:A Message Passing Interface Standar*d, University of Tennessee 1995

[EiFri00] THOMAS EICKERMANN , WOLFGANG FRINGS : *VISIT - a Visualization Interface Toolkit Version 1.0*, Interner Bericht [FZJ-ZAM-IB-2000-16] des ZAM am Forschungszentrum Jülich, 2000. Seiten 49-52

[WaTo95] TOM WAGNER, DON TOWSLEY: *Getting started with POSIX Threads*, Department of Computer Science - University of Massachusetts at Amherst 1995

[Ei01] THOMAS EICKERMANN: *tcppp, ein Benchmark Programm für Punkt-zu-Punkt Kommunikation*, Private Mitteilung Jülich 2001

[LuScWe96] ANSGAR LUKOSEK, MARTIN SCZIMAROWSKY, SABINE WERNER: *Das ZAM-Testbett Erfahrungen beim Betrieb Ergebnisse von Durchsatzmessungen*, Interner Bericht [FZJ-ZAM-IB-9634] des ZAM am Forschungszentrum Jülich, 1996

[EiGrHe99] THOMAS EICKERMANN , HELMUT GRUND, JÖRG HENRICHS: *Performance Issues of Distributed MPI Applications in a German Gigabit Testbed*, Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proc. of the 6th European PVM/MPI Users' Group Meeting, Barcelona, 1999

[1] Homepage der HiPPI Organisation: http://www.hippi.org

[2] Allgemeine Informationen zum RePhoNet: http://www.rephonet.de/

[3] Homepage des ATM-Forums: http://www.atmforum.com/

[4] Myricom Inc.: http://www.myri.com

[5] Myricom Inc.: http://www.myri.com/scs/GM_FAQ.html, Stand vom 22.8.2001

[6] Myricom Inc.: *The GM Message Passing System*, http://www.myri.com/scs/GM/doc/gm.txt, 15. November 2000

[7] Informationen zum ZAMPANO: http://zampano.zam.kfa-juelich.de/

Forschungszentrum Jülich
in der Helmholtz-Gemeinschaft

Jül-3970
Februar 2002
ISSN 0944-2952