Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo

Resource Controlled Interprocessor Communication On NP-based Active Network Nodes

Michel Dänzer
michel@daenzer.net

14. Juli 2004

Diplomarbeit DA-2004.17
Sommersemester 2004

Betreuer
Lukas Ruf

Supervisor
Prof. Dr. Bernhard Plattner

Diese Arbeit ist Bernhard Scheuner gewidmet, der sich während ihrer Entstehungszeit viel zu jung seiner schweren Krankheit geschlagen geben musste. In ihm ging ein grosser Mensch und Physiker verloren, aber in den Herzen seiner Familie und Freunde wird er weiterleben.

Abstract

In dieser Arbeit wird eine generische Kommunikationsinfrastruktur zwischen von je einem Linux Kernel kontrollierten Ausführungseinheiten (Host- und Netzwerkprozessoren) entwickelt, welche auf paketbasierte Kanäle setzt und einen Middle Layer unter den PromethOS Plugin und Node Managers und über einem Hardware Abstraction Layer bildet. Die Implementation wird im Rahmen einer Evaluation verifiziert und ihre Performance gemessen.

In this thesis, a generic communication infrastructure between execution environments controlled by one Linux kernel each (host and network processors) is developed, which builds upon packet based channels and forms a middle layer below the PromethOS plugin and node managers and above a hardware abstraction layer. The implementation is verified and benchmarked for evaluation purposes.

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

Dieses Kapitel soll den geneigten Leser an diese Diplomarbeit heranführen.

1.1. Motivation

Zukünftige Router werden dynamisch programmierbar sein. Code wird zur Laufzeit ersetzt werden, um bei Updates oder Funtionalitätserweiterungen nicht den Router insgesamt abschalten zu müssen. Active Networking [10] entwickelte das Konzept von Active Network Nodes (ANNs), welche die dynamische Funktionalitätserweiterung um Extended (Network) Services vorsehen (Netzwerkdienste, die zusätzlich zum Base IP-Forwarding ausgeführt werden). Die Ausführung von Extended Services benötigt zusätzliche Rechenleistung, die in modernen Routern pro Netzwerk-Port durch Netzwerkprozessoren (NPs) zur Verfügung gestellt wird.

Am Institut für Technische Informatik und Kommunikationssysteme (TIK) der ETH Zürich (ETHZ) wurde im Rahmen der Semesterarbeit von Guindehi COBRA [9] alias PromethOS entwickelt, welches eine dynamisch erweiterbare Plattform im Linux-Kernel 2.4 zur Verfügung stellt. PromethOS wurde im Rahmen der Arbeit zu PromethOS NP [1, 11] erweitert, damit die Umgebung von NPs profitieren kann. Als Proof-of-Concept Implementation wurde PromethOS NP auf einem einfachen ANN bestehend aus einem PowerNP 4GS3 Application Reference Board (ARB)[12] und einem Host General Purpose Processor (GPP) entwickelt. Moderne Router verfügen jedoch über mehrere Netzwerk-Ports mit NPs. Man möchte sämtliche Rechenknoten zu einem einzigen, hierarchisch organisierten ANN bündeln können.

1.2. Problembeschreibung

Die offizielle Aufgabenstellung ist im Anhang A auf den Seiten 55ff nachzulesen.

1.2.1. Aufgabe

PromethOS unterstützt bereits sowohl General Purpose Prozessoren (GPPs) als auch Netzwerkprozessoren (NPs)[1]. Was noch fehlt, ist eine generische Möglichkeit, auf einfache und effiziente Art zwischen den Bestandteilen (insbesondere GPPs, sei es im Host oder in NPs eingebettet; im Weiteren Knoten genannt) eines Active Network Node (ANN) zu kommunizieren.

[1] Pascal Erni hat in [1] die Grundlagen dafür gelegt, siehe Abschnitt 2.1.1 auf Seiten 13f.

Um diese Lücke zu füllen, soll eine Kommunikationsinfrastruktur entwickelt werden. Sie soll nicht nur Netzwerktraffic, sondern auch Kontrollinformationen, Code usw. zwischen den PromethOS Komponenten auf den einzelnen Knoten transportieren können. Ausserdem sollen die von der Infrastruktur verwalteten Ressourcen pro Extended Service kontrolliert werden können.

Konkret soll die Infrastruktur für das PowerNP 4GS3 Application Reference Board implementiert werden. Die Architektur soll jedoch den Einsatz nicht auf diesen bestimmten Hardwaretyp beschränken.

Herausforderung

Die wichtigsten Herausforderungen dieser Aufgabe sind:

Generizität. Es muss ein generischer Kommunikationskanal zwischen einer beliebigen Anzahl Knoten bereitgestellt werden. Die Daten müssen transparent optimal geroutet werden. Die Herausforderung liegt dabei insbesondere darin, eine Schnittstelle zu definieren, welche eine flexible und effiziente Kommunikation ermöglicht.

Hardwareunabhängigkeit. Die Umsetzung muss möglichst allgemein gehalten sein, damit sie mit allen gegenwärtigen sowie hoffentlich auch zukünftigen NPs funktioniert. Da aber letztendlich bei einer konkreten Implementation Hardwareeigenschaften natürlich nicht ignoriert werden können, besteht die Herausforderung darin, die hardwareabhängigen und -unabhängigen Teile der Infrastruktur zu trennen und durch eine geeignete Schnittstelle zu verbinden. Idealerweise sollten künftige Anbdingungen an bestimmte Hardware lediglich eine entsprechende Implementation des hardwareabhängigen Teils erfordern, aber keine Änderungen am hardwareunabhängigen Teil.

Ressourcenkontrolle. Die Infrastruktur muss die Ressourcen (insbesondere die Bandbreite) der Kanäle überwachen und kontrollieren können.

1.3. Gliederung dieses Berichts

Das Kapitel 2 auf den Seiten 13ff diskutiert einige frühere Arbeiten, auf welchen diese Arbeit aufbaut, und erläutert die Entwicklungsumgebung, welche für die Implementation zum Einsatz kam.

Das Kapitel 3 auf den Seiten 19ff erläutert die Architektur der im Rahmen dieser Arbeit entwickelten Implementation. Es wird ausgehend von der Aufgabenstellung zunächst ein grobes Konzept erarbeitet, welches dann schrittweise verfeinert wird, um schliesslich die wichtigsten Aspekte der Implementation detailliert zu beschreiben, insbesondere die verschiedenen Schnittstellen.

Das Kapitel 4 auf den Seiten 29ff befasst sich eingehend mit der Implementation. Die zentralen Lösungsansätze, Datenstrukturen und Funktionen werden ausführlich erläutert.

Das Kapitel 5 auf den Seiten 41ff evaluiert die Implementation bezüglich ihrer Funktion und Performance. Die zur Evaluation eingesetzte Methodik wird beschrieben und die ermittelten Messwerte werden analysiert.

Das Kapitel 6 auf den Seiten 49ff schliesst die Arbeit ab, indem die erreichten Ziele rekapituliert werden und ein Ausblick auf weiterführende Arbeiten gemacht wird.

Der Anhang ab Seite 55 enthält diverse begleitende Materialien wie die ursprüngliche Aufgabenstellung, den zu Beginn der Arbeit aufgestellten Zeitplan, Referenzen auf bearbeiteten Source Code sowie einige eingesetzte Scripts.

1.4. Danksagung

Ich danke Lukas Ruf für seine exzellente Betreuung. Er hat mich kompetent und geduldig ans Ziel geführt.

Ich danke Herrn Professor Dr. Bernhard Plattner, dem Vorsteher des TIK an der ETH Zürich, dass er diese Arbeit möglich gemacht hat.

Ich danke Adrian von Bidder, der seine Diplomarbeit[13] parallel absolvierte, dass er zu meiner Enticklungsmaschine am TIK schaute, während ich von zu Hause aus arbeitete, und dass ich durch Gedankenaustausch mit ihm oft zu neuen Lösungsansätzen inspiriert wurde. Ich hoffe, dass dies auf Gegenseitigkeit beruhte.

Last, but not least danke ich meiner Freundin Fabienne von Gunten, die während der Zeit der Diplomarbeit viel Geduld und Verständnis für mich aufbringen musste.

2. Vorarbeit

Um mir einen Überblick über die Materie zu verschaffen, habe ich einige frühere Arbeiten studiert. Es folgen Zusammenfassungen der für diese Diplomarbeit wichtigsten Arbeiten.

2.1. Analyse verwandter und verwendeter Arbeiten und Dokumente

Gleich zu Beginn habe ich die Diplomarbeit von Pascal Erni studiert, um ein grundsätzliches Verständnis dafür zu erhalten, wie ein Netzwerkprozessor (NP) in einem Active Network Node (ANN) eingesetzt werden könnte. Danach verschaffte ich mir anhand eines Papers von meinem Betreuer Lukas Ruf, R. Keller und meinem Supervisor Professor Dr. Bernhard Plattner einen Überblick darüber, wie ein auf PromethOS basierender ANN aufgebaut ist.

2.1.1. Diplomarbeit DA-2003.13

Pascal Erni hat in seiner Diplomarbeit [1] die Grundlagen für den Einsatz von Netzwerkprozessoren in einem Active Network Node erarbeitet. Kapitel 2 beschreibt zunächst das auch für diese Arbeit verwendete PowerNP 4GS3 Application Reference Board sowie das zugehörige Advanced Software Offering (ASO).

In Kapitel 3 wird auf Basis einer Analyse aller im Active Network Node vorhandenen Komponenten eine Architektur entwickelt, um Datenpakete möglichst nahe bei den Ports verarbeiten, aber wenn nötig auch an eine weiter entfernte, jedoch flexiblere und/oder für die jeweilige Aufgabe leistungsfähigere Komponente weiterreichen zu können. Die Flaschenhälse dieser Architektur werden identifiziert und daraus die maximal erzielbare Performance abgeschätzt. Schliesslich wird als mögliche Erweiterung das Konzept von Pico Plugins erörtert, mit welchem die Pico Engines des Netzwerkprozessors für PromethOS Plugins genutzt werden könnten. Dieses Konzept könnte sich für meine Diplomarbeit als nützlich erweisen.

Kapitel 4 beschreibt die Implementation der Architektur. Es wird erläutert, wie die Kommunikation zwischen dem Netzwerkprozessor und dem Host realisiert wurde, und wie Linux auf der im Netzwerkprozessor enthaltenen PowerPC CPU zum Laufen gebracht wurde.

Kapitel 5 beschreibt, wie mit Hilfe eines Plugin Manager Simulators Durchsatz und

Latenz des Systems mit verschiedenen Datenpaketlängen gemessen wurden. Zuerst wurde der Control Point auf der Host CPU ausgeführt, in Systemen mit einem 32 Bit/33 Mhz bzw. 64 Bit/66 Mhz PCI Bus. Im ersteren war der PCI Bus der Flaschenhals, in letzterem das GBit Ethernet. Schliesslich wurde der Control Point noch auf der embedded PowerPC CPU ausgeführt, wobei dessen geringer Takt von nur 133 Mhz sowie die aufwändige Kommunikation mit dem Netzwerkprozessor den Durchsatz stark beeinträchtigte. Es wird jedoch angedeutet, dass sich die Kommunikation eventuell optimieren liesse, womit annähernd ein Durchsatz von einem GBit pro Sekunde erreicht werden könnte.

In Kapitel 6 schliesslich wird der mögliche Durchsatz mit Pico Plugins abgeschätzt. Es zeigte sich, dass Pico Plugins an sich keinen Flaschenhals darstellen sollten, was sich aber ändert, sobald sie die Nutzdaten eines Pakets vollständig bearbeiten müssen.

2.1.2. A Scalable High-Performance Router Platform Supporting Dynamic Service Extensibility On Network and Host Processors

Das Paper [2] bietet einen Überblick darüber, wie das PromethOS NP Framework den Aufbau eines ANN mit einer hierarchisch aufgebauten Mehrprozessorarchitektur ermöglicht.

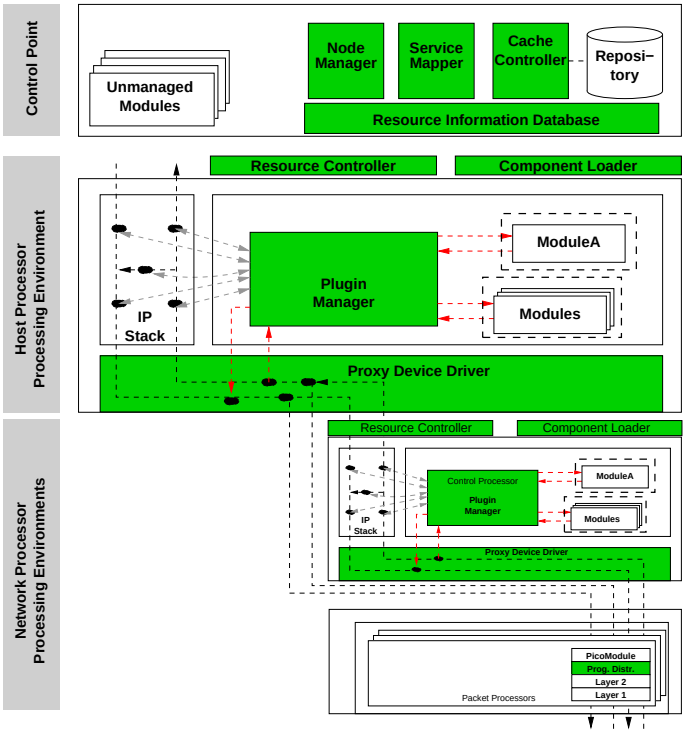


Abbildung 2.1.: Schematischer Aufbau eines PromethOS ANN

Abbildung 2.1 zeigt schematisch den Aufbau eines auf PromethOS basierenden ANN. Es wird grundsätzlich zwischen Laufzeit- und Managementumgebung unterschieden.

Laufzeitumgebung

- Die traditionellen Modelle für ANNs sind als *port extended* (statische Zuordnung von Recheneinheiten zu jedem Eingags- und Ausgangsport) und *pool extended* (Pool von Recheneinheiten, welchen dynamisch Flows zugewiesen werden) bekannt. Beide können einen hierarchischen ANN nicht abbilden, weshalb ein *hierarchy extended* Modell erarbeitet wurde, welches die beiden anderen Modelle als Spezialfälle enthält.

- Eine *Service Komponente* ist eine Funktionseinheit im Sinne eines Plugins. Sie verfügt über Eingangs- und Ausgangsports für Daten und Servicekontrolle. Der Dateneingang ist mit einem Filter versehen, das bestimmt, welche Daten von der Komponente zu bearbeiten sind. Optional sind auch komponentenspezifische Kontrollports möglich.

- Ein Service wird als Graph mit Service Komponenten als Kanten und deren Verbindungen als Knoten modelliert. Die Komponenten können zusätzliche Kontrollverbindungen aufweisen. Das Ganze wird mit *Service Graph* bezeichnet, eine Komponente und ihre Knoten als *Service Chain*. Service Chains werden in sogenannten Processing Environments (PEs) implementiert, z.B. auf einer General Purpose CPU oder einem NP.

- Ein sogenannter *programmierbarer Verteiler* puffert Pakete eingangs- und ausgangsseitig und verteilt sie unter Beachtung des Service Graphen und des Ressourcenverbrauchs an die Service Chains. Er fungiert somit als Knoten im Service Graphen.

Managementumgebung

- Die Resource Information Database (RID) enthält Spezifikationen der Services, ihrer Komponenten und des gesamten ANN.

- Der Cache Controller verwaltet die Binärkomponenten.

- Der Service Mapper teilt gemäss den Spezifikationen der RID Funktionseinheiten zu.

- Der Node Manager schliesslich kontrolliert den ANN als Ganzes. Er bedient sich dazu aller bisher genannter Komponenten.

2.2. Entwicklungsumgebung

Dieser Abschnitt führt aus, welche Hard- und Software für diese Arbeit wie eingesetzt wurde.

2.2.1. Linux

Da PromethOS auf das Netfilter Framework des Linux Kernels aufsetzt, ist es wenig überraschend, dass als Betriebssystem Linux zum Zuge kam.

Die Ausgangslage sah so aus, dass PromethOS für die aktuellen 2.4er und 2.6er Kernels verfügbar war, aber auf dem PowerNP embedded PowerPC (ePPC) nur der MontaVista[5] Kernel 2.4.17_mvl21 lief. Da ich aber nicht ständig Änderungen zwischen verschiedenen Kernelversionen hin- und herkopieren wollte, versuchte ich, diese auf den ersten Blick eher widersprüchlichen Voraussetzungen unter einen Hut zu bringen. Es gelang mir mit einigem Aufwand, den PromethOS Code sowie die für einen x86 PC benötigten Teile[1] des Linux Kernels in den MontaVista Kernel einzupflegen. Der resultierende Kernel verabschiedete sich jedoch auf der x86 Entwicklungsmaschine sehr früh im Bootprozess mit einer Kernel Panic. Nach einigen Experimenten stellte sich heraus, dass ältere 2.4er Kernel nicht korrekt funktionieren, wenn sie mit einem GNU Compiler Collection (gcc[6]) 3.x Compiler übersetzt wurden. Mit gcc 2.95 klappte es dann. Forthin konnte ich für den x86 Host und den ePPC den selben Kernel Source Code verwenden.

Abklärung Linux 2.6

Um abzuschätzen, wie viel Aufwand eine Portierung des für unser PowerNP Board spezifischen Codes im MontaVista Kernel 2.4.17_mvl21 auf einen aktuellen Linux Kernel 2.6.x erfordern würde, habe ich mit dem Standardtool `diff` die Unterschiede von 2.4.17_mvl21 zum offiziellen Kernel 2.4.17 ermittelt. Die resultierende `.diff` Datei war Dutzende von MB gross. Das Tool `diffstat` zeigte, dass die Änderungen über weite Teile des Kernel Sourcecode verteilt waren.

Zwar waren bestimmt nicht alle diese Änderungen wichtig für unsere Umgebung, aber Lukas Ruf ging mit mir einig, dass allein die Identifizierung, geschweige denn die Portierung der notwendigen Teile im Rahmen dieser Arbeit zu viel Zeit beanspruchen würde. Er wollte sich die Sache selbst ansehen, wenn er die Zeit dazu finden würde.

2.2.2. Entwicklungswerkzeuge

Um Binärdateien für den ePPC zu erzeugen, wurden die Cross Compile Tools der MontaVista HardHat Distribution verwendet. Sie basieren auf gcc 2.95. Das entspricht im Vergleich zur auf der x86 Host-Entwicklungsmaschine eingesetzten Debian Distribution

[1] Diese Teile fehlen im MontaVista Kernel Source Code, weil er nur für den Betrieb auf embedded PowerPC CPUs vorgesehen ist.

nicht dem aktuellen Stand, was aber wegen der im Abschnitt 2.2.1 beschriebenen Probleme mit aktuellen gcc Versionen nicht weiter ins Gewicht fiel.

Zur Kontrolle der Übersetzung der diversen Bestandteile dieser Arbeit kam GNU make zum Einsatz.

2.2.3. PowerNP Application Reference Board

Das für die konkrete Implementation der Infrastruktur verwendete Application Reference Board für den IBM PowerNP NP4GS3 Netzwerkprozessor wurde durch die Firma Silicon & Software Systems (S3) hergestellt[12]. Die Hauptkomponenten sind ein NP4GS3B PowerNP Netzwerkprozessor, drei 1000BaseT Ethernet Netzwerkanschlüsse und eine PCI-GMII Bridge. Eine Besonderheit des ARB ist, dass für die Kommunikation zwischen dem Host und dem NP4GS3 ein BroadCom BCM5700 Gigabit-Ethernet Netzwerkcontroller zuständig ist. Dadurch erscheint die Karte dem Host als normale BroadCom Netzwerkkarte, und der PowerNP muss so angesteuert werden, wie wenn er extern via Ethernet angeschlossen wäre. Es ist also keine direkte Kommunikation zwischen Host und PowerNP via den PCI Bus möglich.

2.2.4. Sourcecodeverwaltung

Zur Verwaltung des Source Codes und insbesondere der Änderungen daran kam das frei verfügbare Subversion (svn[7]) zum Einsatz, welches kurz zuvor das erste stabile Release 1.0 feierte. Subversion bietet einen sehr einfachen Umstieg vom altbewährten Concurrent Versions System (CVS[8]), aber darüber hinaus auch einige nützliche Neuerungen, zum Beispiel die Möglichkeit, das Kopieren und Verschieben (und damit auch Umbenennen) von Dateien und ganzen Verzeichnisbäumen zu verwalten. Damit kann man einfach Kopien bestehender Teilbäume neu anordnen und in verschiedene Richtungen entwickeln. Subversion bietet auch bessere Möglichkeiten als CVS, um verschiedene Entwicklungsstränge wieder zusammenzuführen.

2.2.5. Technische Probleme

Im Verlaufe der Arbeit traten teilweise erhebliche technischen Probleme auf. So funktionierte das Booten eines Linux Kernels auf der zu Beginn in meiner Entwicklungsmaschine eingebauten PowerNP Karte mittels des Tools `d6load` partout nicht.

Nachdem auch nach eingehender Inspektion des Source Code der beteiligten Softwarekomponenten dort keine Ursache gefunden werden konnte, gab es einen Hardwaredefekt zu befürchten. Und in der Tat verrichtete eine andere Karte anstandslos ihren Dienst. Mein Betreuer Lukas Ruf stellte fest, dass die Jumpereinstellungen evtl. nicht korrekt waren für die zusätzliche externe Stromversorgung via Harddisk-Stromkabel. Er konnte in einer seiner Maschinen mit überprüften Jumpereinstellungen und ohne externe Stromversorgung mit einem Diagnosetool von IBM keine Probleme feststellen.

Als zweite Karte in meiner Entwicklungsmaschine legte sie dann jedoch wieder ein äusserst merkwürdiges Verhalten an den Tag: Unmittelbar nach dem ersten Booten funktionierte sie bis zu einem gewissen Grad (einmal blieb sie erst im Verlauf von `d6load` hängen). Die Situation verschlechterte sich dann aber im Verlauf von Minuten, bis sie ihre Mitarbeit schliesslich ganz verweigerte. Nach folgenden Reboots wurde die Karte sogar vom Linux Kernel PCI Code deaktiviert, weil sie eine riesige Grösse ihres PCI Speicherbereichs meldete, welcher sich so nicht konfliktfrei in die gesamte PCI Speicherverteilung des Systems einbinden liess. Die Symptome waren in allen PCI Slots die selben, auch als einzige Karte und im gleichen Slot, in welchem die andere Karte einwandfrei funktionierte.

Unsere plausibelste Vermutung war, dass diese Probleme durch eine ungenügende Stromversorgung in meiner Entwicklungsmaschine hervorgerufen wurden. Dies wurde allerdings nicht direkt bestätigt, als die problematische Karte in Adrian von Bidders Entwicklungsmaschine die gleichen Symptome zeigte, welche allerdings meiner Entwicklungsmaschine ähnlich war. Es könnte sich also um eine Unverträglichekeit mit diesem Maschinentyp handeln.

3. Architektur

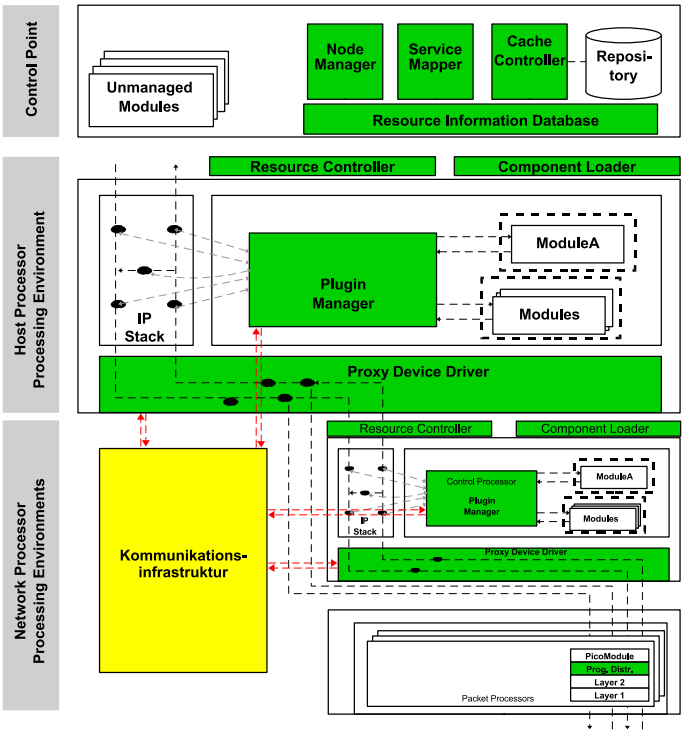


Abbildung 3.1.: Grobschema Kommunikationsinfrastruktur

3.1. Erarbeitung

Auf Grund der Aufgabenstellung (siehe Anhang A auf Seite 55ff) und der im Kapitel 2 besprochenen Arbeiten habe ich folgende Anforderungen abgeleitet:

- Es soll eine Kommunikationsinfrastruktur zwischen den Plugin Managers auf den verschiedenen General Purpose Prozessoren (GPPs) bereitgestellt werden.
- Die Plugin Managers sollen darüber sowohl Daten als auch Kontrollinformationen austauschen können.

- Falls genug Zeit vorhanden ist, soll die Infrastruktur dahingehend erweitert werden, dass sie Profiling Informationen über die verfügbaren Verbindungen und Kontrolle der Ressourcen wie z.B. Bandbreite anbietet.

Anders ausgedrückt sollen die Grundlagen für die Knoten des Service Graphen sowie die Kontrollverbindungen zwischen den Servicekomponenten bereitgestellt werden.

3.2. Grobkonzept

Die Kommunikationsinfrastruktur soll wie in Abbildung 3.1 illustriert Kommunikationskanäle zwischen den Plugin Managers bereitstellen, welche auf den diversen General Purpose CPUs laufen.

3.2.1. Kontrollinformationen

Grundsätzlich standen zwei Möglichkeiten zur Auswahl:

- Daten und Kontrollinformationen werden auf separaten Kanälen übertragen.
- Daten und Kontrollinformationen werden auf einem gemeinsamen Kanal übertragen.

Mein Betreuer Lukas Ruf und ich waren uns auf Anhieb einig, dass die gemeinsame Übertragung auf einem einzigen Kanal aus Gründen der Einfachheit und Flexibilität von Schnittstellen und Implementation vorzuziehen ist.

Es soll im Wesentlichen ein generischer Kommunikationskanal bereitgestellt werden, der Pakete in einem definierten Format überträgt.

Die Infrastruktur soll sich nicht um den Inhalt der eigentlichen Nutzdaten kümmern (Layering). Auf diese Weise soll die Infrastruktur für den Plugin Manager beliebig nutz- und erweiterbar sein.

3.2.2. Interface für Plugin Manager

Den wesentlichen Bestandteil der Schnittstelle zum Plugin Manager sollen eine Funktion zum Versenden eines Pakets sowie eine Callbackfunktion zum Empfang eines Pakets bilden.

Der Plugin Manager soll für die Kommunikation die Topologie des ANN nicht kennen müssen, sondern lediglich die Kennungen der Gegenstellen.

Die Infrastruktur soll auch Funktionalität zur Ressourcenkontrolle anbieten. Zum Beispiel soll die Bandbreite beschränkt werden können, die von einem Flow oder einer Service Chain benutzt werden kann.

3.3. Verfeinerung

In diesem Abschnitt wird das im vorherigen Abschnitt erarbeitete Grobkonzept verfeinert. Insbesondere wird der Aufbau des Paketheaders der Infrastruktur erläutert, wie die Infrastruktur die Verbindungen zwischen den Knoten aufbaut und wie die Schnittstellen zwischen der Infrastruktur und der Hardware, dem Node Manager und dem Plugin Manager aussehen.

3.3.1. Paketaufbau

Die Infrastruktur benutzt paketbasierte Kanäle zwischen den Knoten. Die Infrastrukturpakete enthalten am Anfang einen Header, welcher die von der Infrastruktur zum Transport benötigten Informationen enthält, und danach die Nutzdaten.

Abbildung 3.2.: Schematischer Aufbau eines Infrastrukturpakets

Header

Der Header setzt sich konkret aus folgende Bestandteilen zusammen:

Absenderkennung: Damit kann die Infrastruktur Bestätigungen, Rückweisungen oder Antworten an den Absender zurück senden. Ausserdem wird die Absenderkennung auch an die registrierte Empfangs-Upcallfunktion weitergereicht, so dass der Plugin Manager weiss, woher ein Paket stammt.

Empfängerkennung: Hieraus kann ein Knoten erkennen, ob das Paket an ihn gerichtet ist, oder wohin er es weiterleiten muss.

Pakettyp: Anhand des Pakettyps kann die Infrastruktur auf jedem Knoten entlang des Übertragungswegs die interne Information aus den Headerdaten lesen oder ihr unbekannte Pakete zurückweisen. Für die Kodierung des Pakettyps bieten sich zwei grundsätzliche Möglichkeiten an:

Enumeration der Pakettypen: Jeder einzelne Pakettyp erhält einen eigenen Wert.

Kodierung der enthaltenen Informationen als Bitfeld: Jedes Bit steht für eine bestimmte Information.

Mit der Enumeration können je nach Länge des Typfelds praktisch beliebig viele verschiedene Pakettypen unterschieden werden, andererseits ist das Bitfeld flexibler für die Zusammenstellung einer begrenzten Anzahl Informationseinheiten. Je nach

dem zu erwartenden Gebrauchsmuster muss bei der Implementation eine der Varianten, oder möglicherweise auch eine Mischform, gewählt werden.

Länge des Headers: Mit dieser Information kann die Infrastruktur falls nötig auch bei unbekannten Pakettypen zwischen Header- und Nutzdaten unterscheiden.

Länge der Nutzdaten: Aus der Länge des Headers und der Nutzdaten folgt die Gesamtlänge des Pakets.

Paketspezifische Headerdaten: Hier bringt die Infrastruktur gemäss dem Pakettyp interne Informationen unter, um sie von einem Knoten zu einem oder mehreren anderen zu verbreiten.

Nutzdaten

Der Nutzdatenbereich eines Infrastrukturpakets wird in erster Linie bei Datenpaketen für den Versand von Daten vom Plugin Manager eines Knoten zum Plugin Manager eines anderen Knoten verwendet[1]. Ein Pakettyp ist allein für diesen Zweck reserviert. Bei anderen Pakettypen kann der Nutzdatenbereich jedoch auch von der Infrastruktur für internen Informationsaustausch verwendet werden.

3.3.2. Initialisierung

Der Node Manager (NM) des Active Network Node (ANN) kennt den Aufbau des gesamten ANN und soll den von der Infrastruktur abgedeckten Komponenten für den gesamten ANN eindeutige Kennungen zuweisen, welche von der Infrastruktur zur Adressierung genutzt werden können. Die Host CPUs bilden dabei gemeinsam eine Einheit, während NPs usw. je eine Einheit darstellen. Im folgenden wird eine solche Einheit Knoten genannt.

Es folgt nun eine Beschreibung des Prozesses, wie neue Knoten erkannt und in die Infrastruktur eingebunden werden, indem ihnen eine eindeutige Kennung zugewiesen wird. Dabei wird folgendes Szenario zugrunde gelegt (siehe auch Abbildung 3.3):

Master Knoten Dies ist der zentrale Knoten, der a priori die Kennung 1 trägt. Er ist der massgebliche Verwalter der Infrastruktur, insbesondere kann nur er eindeutige Kennungen zuweisen.

Knoten A Ein Knoten, welchem bereits vom Master Knoten eine Kennung zugewiesen wurde.

Knoten B Ein neuer Knoten, welchem noch keine Kennung zugewiesen wurde. Er baut eine direkte Verbindung zu Knoten A auf.

Knoten C Ein weiterer Knoten, welchem bereits vom Master Knoten eine Kennung zugewiesen wurde.

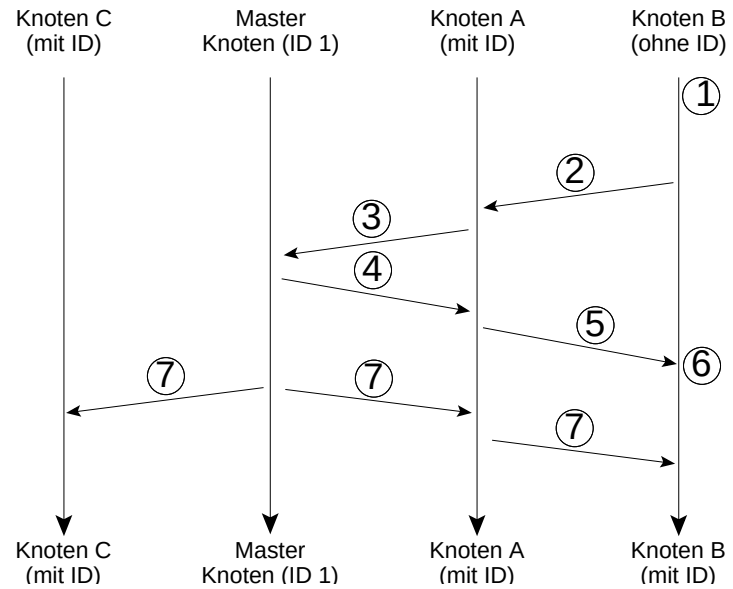


Abbildung 3.3.: State Flow Diagramm des Initialisierungsprozesses

1. Knoten B führt eine Hardwareerkennung durch, z.B. PCI Bus Scan, etc. Das Ergebnis davon ist eine Liste potenzieller direkter Kommunikationskanäle zu anderen Knoten.

2. Der NM auf Knoten B veranlasst die Infrastruktur dazu zu versuchen, über jeden in Schritt 1 ermittelten potenziellen Kanal ein wie in Abschnitt 3.3.1 auf Seite 21 beschrieben definiertes Discovery Paket zu senden.

3. Die Infrastruktur auf Knoten A erhält ein solches Discovery Paket und weiss dadurch, dass ein neuer Kommunikationskanal zu Knoten B besteht, einer Gegenstelle (im folgenden Peer genannt), und schickt ein Paket an die Infrastruktur auf dem Master Knoten, welches diese darum bittet, dem Peer B eine eindeutige Kennung zuzuweisen.

4. Die Infrastruktur auf dem Master Knoten erhält dieses Paket, weist dem neuen Knoten B eine eindeutige Kennung zu und schickt diese in einem speziellen Paket zurück an die Infrastruktur auf Knoten A, welche die Kennung für den Peer B anforderte[2].

5. Die Infrastruktur auf Knoten A leitet das Paket mit der zugewiesenen Kennung an die Infrastruktur auf seinem neuen Peer B weiter.

[1] Siehe Abschnitt 3.3.3 auf Seite 27 für ausführliche Informationen dazu.

[2] Die Infrastruktur auf dem Master kann dieses Paket nicht direkt an die Infrastruktur auf Knoten B senden, da es diesem ja erst seine Kennung zuweist, welche aber von der Infrastruktur für das Routing von Paketen benötigt wird.

6. Die Infrastruktur auf Knoten B erhält das Paket mit dessen Kennung und merkt sich diese. Sie nimmt von nun an Pakete entgegen, welche an diese Kennung adressiert sind, und leitet sonstige Pakete weiter, welche an ihr bekannte Kennungen adressiert sind.

7. Die Infrastruktur auf dem Master Knoten sendet die Liste aller ihr bekannten Verbindungen zwischen Knoten an alle Knoten (Broadcast). Daraus kann die Infrastruktur auf Knoten B ihre Routingtabelle aufbauen, und die Infrastruktur auf Knoten C erhält Kenntnis von Knoten B und kann damit ihre Routingtabelle aktualisieren.

Es kann geschehen, dass Schritt 2 auf Knoten B ausgeführt wird, bevor die Infrastruktur auf Knoten A zur Entgegennahme des Discovery Pakets bereit ist. Führt dann Knoten A zu einem späteren Zeitpunkt seinerseits Schritt 2 aus, so erhält die Infrastruktur auf Knoten B ein Discovery Paket von der Infrastruktur auf Knoten A. Sie schickt dann wiederum ein Discovery Paket an die Infrastruktur auf Knoten A, worauf der Prozess ab Schritt 3 weiterläuft.

Es ist leicht ersichtlich, dass dieser Prozess iterativ angewendet werden kann, um eine beliebige Anzahl von Knoten in die Infrastruktur einzubinden. Jeder neue Knoten erhält eine eindeutige Kennung, und die Infrastruktur auf jedem Knoten erlangt Kenntnis über alle anderen Knoten und die Verbindungen zwischen ihnen.

Damit die Ressourcen im laufenden Betrieb kontrolliert werden können, müssen diese mittels eines Profilings ausgemessen werden. Dazu können zum Beispiel in Schritt 4 spezielle Pakete zwischen dem neuen Knoten und seinem Peer ausgetasucht werden, um die Bandbreite und Latenz ihrer Verbindung zu messen. Diese Informationen können dann zusammen mit der Bitte um Zuweisung einer Kennung dem Master bekanntgegeben werden, welcher sie später in Schritt 8 an alle Knoten verteilt.

Sowohl Kennungszuweisung wie auch Profiling sollten idealerweise jederzeit individuell wiederholt werden können, z.B. im Hinblick auf Hotplugging (Hinzufügen bzw. Entfernen von Hardwarekomponenten zur Laufzeit).

3.3.3. Interfaces

Im folgenden werden die Schnittstellen der Kommunikationsinfrastruktur zu den anderen Komponenten näher erörtert.

Hardware

Um die Infrastruktur von Hardwareeigenschaften wie dem Typ eines NP, seiner Anbindung usw. abzuschirmen, braucht sie eine Schnittstelle im Sinne eines Hardware Abstraction Layer (HAL). Der hardwarespezifische Treiber muss hierfür einen Pointer auf

eine HAL Datenstruktur (`struct promethos_comm_hal_t`) anbieten, welche ihrerseits Pointer auf folgende Funktionen enthält[3]:

```
struct sk_buff* ( *alloc_skb )( int size, promethos_comm_hal_t *hal )
```

Die Infrastruktur beschafft sich hiermit einen Pointer auf einen Socket Buffer (`struct sk_buff`), um ein Paket der Grösse `size` (Gesamtgrösse Infrastrukturheader plus Nutzdaten) über diesen HAL zu verschicken. Kann kein entsprechender Socket Buffer alloziert werden, z.B. weil der Speicher knapp ist, gibt diese Funktion `NULL` zurück.

```
int ( *send )( struct sk_buff *skb, promethos_comm_hal_t *hal )
```

Die Infrastruktur benutzt diese Funktion, um ein Paket zu senden. `skb` zeigt auf einen mit `alloc_skb()` reservierten Socket Buffer, der den Infrastrukturheader und die Nutzdaten enthält. Der HAL muss prinzipiell Pakete beliebiger Grösse annehmen und diese nötigenfalls auf der Senderseite fragmentieren und auf der Empfängerseite reassemblieren.

```
int ( *register_receive )( promethos_comm_hal_t *hal,
promethos_comm_halrecv_f *receive, void *commpriv )

int ( *unregister_receive )( promethos_comm_hal_t *hal,
promethos_comm_halrecv_f *receive, void *commpriv )
```

Mit diesen Funktionen (de)registriert die Infrastruktur ihre Empfangsfunktion. Falls der HAL von einem Kernelmodul implementiert wird, muss dieses seine Entladung verhindern, während die Empfangsfunktion registriert ist, z.B. indem es in `register_receive()` das `MOD_INC_USE_COUNT` Makro und analog in `unregister_receive()` das `MOD_DEC_USE_COUNT` Makro verwendet. Hauptsächlich müssen diese Funktionen aber natürlich den folgenden Pointer verwalten:

```
int ( *receive )( struct sk_buff *skb, promethos_comm_hal_t *hal )
```

Wenn dieser Pointer nicht `NULL` ist, zeigt er auf die Empfangsfunktion der Infrastruktur, welche der HAL aufruft, um ein von ihm empfangenes Paket an die Infrastruktur zu übergeben. `skb` zeigt auf den Socket Buffer, der das empfangene Paket inklusive Infrastrukturheader und Nutzdaten beschreibt.

Ausserdem enthält die HAL Datenstruktur folgende Felder:

[3] Alle HAL Funktionen benötigen einen Pointer auf die HAL Datenstruktur, um diese auslesen und manipulieren zu können. Man kann dies als simples Objektmodell im Sinne objektorientierter Programmierung betrachten.

`void *halpriv`

Hier kann sich der HAL einen Pointer merken. Der HAL für das PowerNP ARB nutzt dies z.B., um das `struct net_device` zu finden, welches er benutzen kann, um Frames zu senden.

`promethos_comm_cpuid_t peer`

Hier merkt sich die Infrastruktur, welche Kennung der Peer hat, der auf der anderen Seite der vom HAL repräsentierten Verbindung steht.

`void *commpriv`

Hier kann sich die Infrastruktur ihrerseits einen Pointer merken. Bei einer Verbindung, die auf ein Netzwerkinterface aufbaut, merkt sie sich z.B. den Pointer auf das entsprechende `struct net_device`.

Node Manager

Node Manager (NM) steht hier als allgemeiner Begriff für die Userspace Komponenten, welche den ANN im allgemeinen und die Kommunikationsinfrastruktur im besonderen steuern, beispielsweise Knoten in die Infrastruktur einbinden oder aus ihr entfernen oder die Topologie der Infrastruktur ermitteln. Diese Funktionalität wird in einem Filesystembaum angeboten, beispielsweise im `/proc` Filesystem (2.4er Kernel) oder aber im `/sys` Filesystem (2.6er Kernel). Der Baum ist wie folgt aufgebaut:

`/proc/net/promethos_comm/`: Wurzelverzeichnis.

- `my_id`: Dieses File enthält die Infrastrukturkennung des Knoten. Enthält es `0`, so wurde dem Knoten noch keine Kennung zugewiesen, d.h. er hat noch keinen Kontakt zum Masterknoten.
- `links/`: Verzeichnis zur Auflistung der bekannten Verbindungen zwischen Knoten.
  - `register`: Dieses File dient der Einbindung einer neuen Verbindung zwischen zwei Knoten in die Infrastruktur. Der NM schreibt den Typ (z.B. `netif` für ein Netzwerkinterface) und die Kennung (z.B. `npctl0`) der Verbindung zum unter Umständen (es könnte sich auch um eine neue Verbindung zwischen zwei bereits bekannten Knoten handeln) neuen Knoten in das File, worauf die Infrastruktur versucht, Kontakt aufzunehmen, und falls das gelingt, den Knoten in die Infrastruktur aufnimmt[4], falls er nicht bereits bekannt ist. Ausserdem wird das Entladen des Kommunikationsinfrastruktur Kernelmoduls verhindert, solange mindestens eine Verbindung zu einem anderen Knoten besteht.

[4] D.h., der Masterknoten teilt dem neuen Knoten eine eindeutige Kennung zu, wodurch er für andere Knoten erreichbar wird.

`unregister`: Dieses File dient der Entfernung einer Verbindung zwischen zwei Knoten aus der Infrastruktur. Der NM schreibt wiederum Typ und Kennung (z.B. `netif npctl0`) der Verbindung in dieses File. Unter Umständen fällt damit auch einer der Knoten aus der Infrastruktur. Ausserdem kann das Kommunikationsinfrastruktur Kernelmodul wieder entladen werden, sobald keine Verbindung zu einem anderen Knoten mehr besteht.

`<Kennung>-<Kennung>`: Für jede Verbindung zwischen zwei Knoten: Dieses File repräsentiert die Verbindung zwischen den Knoten mit den im Verzeichnisnamen enthaltenen Kennungen. Hieraus kann der NM die Kennungen aller bekannten Knoten sowie die Verbindungen zwischen ihnen, d.h. die Topologie des ANN bestimmen.
Wenn die Kontrolle der Ressourcen implementiert ist, kann der NM durch Auslesen dieses Files ausserdem die entsprechenden Informationen wie Bandbreite und Latenz zwischen den beiden Knoten, bestehende Beschränkungen usw. erhalten sowie durch Schreiben in das File neue Beschränkungen aufstellen oder bestehende ändern bzw. löschen.

Mit dieser Kontrollstruktur ist der Node Manager in der Lage, Verbingungen zwischen Knoten in die Infrastruktur aufzunehmen und aus ihr zu entfernen, die Topologie der Infrastruktur zu ermitteln sowie den Ressourcenverbrauch zu kontrollieren.

Plugin Manager

Diese Schnittstelle wurde bereits im Abschnitt 3.2.2 auf Seite 20 skizziert. Hier nun eine detailliertere Beschreibung der angebotenen Funktionen.

```
struct sk_buff* promethos_comm_alloc_skb( promethos_comm_cpuid_t target,
int size )
```

Der Plugin Manager ruft diese Funktion auf, wenn er Nutzdaten der maximalen Grösse `size` an den Knoten mit der Kennung `target` senden will. Es wird die HAL-Funktion `alloc_skb()` (siehe Abschnitt 3.3.3 auf Seite 25) aufgerufen, um einen Socket Buffer zu reservieren, der zusätzlich Platz für den Infrastrukturheader (siehe Abschnitt 3.3.1 auf Seite 21) bietet, der Header wird mit sämtlichen zur Auslieferung benötigten Informationen initialisiert, und der Pointer auf den Socket Buffer wird zurückgegeben. Im Fehlerfall wird `NULL` zurückgegeben. Andernfalls kann der Plugin Manager die Standard Socket Buffer Funktionen (`skb_*()`) verwenden, um den allozierten Socket Buffer mit Daten zu füllen.

```
int promethos_comm_send( struct sk_buff *skb )
```

Der Plugin Manager benutzt diese Funktion, um ein Paket zu senden. `skb` zeigt auf einen Socket Buffer, der mit der Funktion `promethos_comm_alloc_skb()` reserviert worden sein muss, sonst wird der Fehlercode `-EINVAL` zurückgegeben, nichts gesendet und der

Socket Buffer nicht freigegeben. Sind die Voraussetzungen erfüllt, wird die HAL-Funktion `send()` (siehe Abschnitt 3.3.3 auf Seite 25) verwendet, um das Paket zu senden, und der Socket Buffer wird danach freigegeben.

```
int promethos_comm_register( promethos_commrecv_f *receive )
```

Mit dieser Funktion registriert der Plugin Manager die Upcall-Funktion zum Empfang von Paketen. Normalerweise wird `0` (Null) zurückgegeben, im Fehlerfall (wenn z.B. bereits eine Empfangsfunktion registriert ist) ein negativer Standardfehlerwert[5]. Die Funktion muss den Prototypen

```
int promethos_commrecv_f( promethos_comm_cpuid_t source,
struct sk_buff *skb );
```

aufweisen. `source` ist die Kennung des sendenden Knoten, `skb` der Pointer auf den Socket Buffer, der die Nutzdaten enthält. Der Plugin Manager soll bei erfolgreichem Empfang Null, im Fehlerfall einen negativen Standardfehlerwert[5] zurückgeben. Er ist in jedem Fall für die Freigabe des Socket Buffers mittels der Kernel-Funktion `dev_kfree_skb()` verantwortlich.

```
int promethos_comm_unregister( promethos_commrecv_f *receive )
```

Mit dieser Funktion deregistriert der Plugin Manager die Upcall-Funktion zum Empfang von Paketen wieder. Solange eine Empfangsfunktion registriert ist, kann das Kommunikationsinfrastruktur Kernelmodul nicht entladen werden. Stimmt der übergebene Pointer nicht mit dem registrierten Pointer überein, wird die Empfangsfunktion nicht deregistriert und der negative Standardfehlerwert `-EINVAL` zurückgegeben. Sonst wird die Funktion deregistriert und Null zurückgegeben.

Zwischenzeitlich war auch geplant, dass die Infrastruktur die Möglichkeit bieten sollte, Pakete zu empfangen, welche mit der von Pascal Erni in [1] entwickelten Methode in den Core Language Processors (CLPs) des PowerNP klassifiziert und direkt an den Host Knoten weitergeleitet werden. Diese Methode lässt sich jedoch nicht elegant in die bisher beschrieben Architektur integrieren; insbesondere würde das Layering durchbrochen, weil diese Methode hardwarespezifisch ist: entweder müsste der hardwareabhängige Teil der Infrastruktur Wissen darüber besitzen, wie der Plugin Manager den Paketinhalt interpretiert, um mit dieser Methode empfangene Pakete entsprechend aufzubereiten, oder der Plugin Manager müsste über hardwarespezifisches Wissen verfügen, um diese Pakete zu erkennen. Letztere Variante bietet keinen signifikanten Vorteil gegenüber der parallelen Nutzung dieser Methode, welche weiterhin möglich ist. Aus allen diesen Gründen wurde dieser Weg nicht weiterverfolgt.

[5] Siehe `linux/include/asm/errno.h` (2.4er Kernel) oder `linux/include/asm-generic/errno.h` (2.6er Kernel).

4. Implementation

Dieses Kapitel legt dar, wie die in Kapitel 3 ausgeführte Architektur implementiert wurde.

4.1. Abstraktion der hardwarespezifischen Interfaces

Dieser Abschnitt erläutert, wie der im Abschnitt 3.3.3 auf Seite 24 beschriebene Hardware Abstraction Layer (HAL) für die Kommunikationsinfrastruktur konkret für ein PowerNP Application Reference Board (ARB) implementiert wurde.

4.1.1. Kommunikation zwischen Host und PowerNP embedded PowerPC

Der NP4GS3 Netzwerkprozessor (NP) bietet zwei Arten von Schnittstellen zur Kommunikation an:

- Ethernet Ports, via Data Mover Units (DMUs)
- Lokaler Speicher und/oder PCI-Bus

Beim für diese Arbeit verwendeten Application Reference Board (ARB) fällt letztere Möglichkeit für die Kommunikation zwischen Host CPU und NP embedded PowerPC (ePPC) weg, weil der NP nur über einen Broadcom BCM5700 PCI-X 1000BASE-T Media Access Controller mit dem Host PCI Bus verbunden ist. Als einzige Option bleibt der Datenaustausch mittels Guided Traffic: Die Daten gelangen von der Host CPU via PCI Bus, BCM und DMU zum General PowerPC Handler (GPH), einem Picocode Thread, der für diesen Zweck reserviert ist. Der GPH schreibt die Daten in das D6 DRAM und signalisiert den ePPC, welcher im D6 DRAM auf die Daten zugreifen kann. In die andere Richtung geht es analog vom ePPC via D6 DRAM, GPH, DMU, BCM und PCI Bus zur Host CPU.

Damit ist leider auch eine direkte Kommunikation zwischen mehreren PowerNP ARBs über den PCI Bus ausgeschlossen. Der im Abschnitt 3.3.3 auf Seite 24 detaillierte Hardware Abstraction Layer (HAL) ist aber nicht auf diesen konkreten, eingeschränkten Kommunikationsweg massgeschneidert, sondern generisch für verschiedenste Kommunikationsarten einsetzbar, z.B. direkt über den PCI Bus oder Shared Memory.

4.1.2. Erweiterung des Proxy Device Driver

Für den Guided Traffic mit einem PowerNP ARB benutzt der HAL für die Kommunikationsinfrastruktur den Proxy Device Driver des IBM Advanced Software Offering

(ASO). Der Proxy Device Driver bietet Zugriff auf PowerNP ARBs in Form von Netzwerkinterfaces namens `npctl0`, `npctl1` usw., die sogenannten Kontrollinterfaces.

Zu Beginn dieser Arbeit unterstützte der Proxy Device Driver jedoch nur ein PowerNP ARB gleichzeitig. Folgende Änderungen mussten vorgenommen werden, um diese Beschränkung zu entfernen:

- Der Kernelmodul Parameter `eth=x`, mit dem ein einziges Parent Interface mit einem Namen der Form `ethx` gesetzt werden konnte, wurde durch einen neuen Parameter `ext_ifs='eth1 eth2 ...'` ersetzt, mit welchem eine beliebige Anzahl Parent Interfaces mit beliebigen Namen gesetzt werden kann.
- Die Funktion `rethCtrlDev_create()` wird neu für jedes mit dem Parameter `ext_ifs` gesetzte Parent Interface einmal aufgerufen. Sie nimmt keinen Namen für das Kontrollinterface mehr entgegen, sondern erzeugt `npctl0` für das erste gesetzte Parent Interface, `npctl1` für das zweite, usw. Neu setzt sie auch einen Pointer auf das `struct net_device` des dem Kontrollinterface zugeordneten Parent Interface in den privaten Bereich des `struct net_device` des Kontrollinterface und merkt sich Pointer auf alle Kontrollinterfaces in der neuen Liste `reth_ctrldev_list`.
- `rethCtrlDev` ist kein globaler Pointer auf das `struct net_device` des einzigen Kontrollinterface mehr, sondern wird den Funktionen, welche auf ein Kontrollinterface zugreifen müssen, als Parameter übergeben. Die aufrufenden Funktionen kennen aber meistens nur das zum Kontrollinterface gehörende Parent Interface; um das Kontrollinterface zu finden, benutzen sie die Funktion `reth_find_node()` auf der Liste `reth_ctrldev_list`.

Um PowerNP ARBs für die Kommunikationsinfrastruktur nutzbar zu machen, wurde der Proxy Device Driver weiterhin um Funktionen erweitert, welche das im Abschnitt 3.3.3 auf Seite 24 beschriebene HAL Interface implementieren, d.h. der HAL für die Infrastruktur benutzt den Proxy Device Driver nicht nur, sondern wurde auch direkt in diesen integriert. Es folgen nun ausführliche Beschreibungen der HAL Funktionen.

Funktionen zum Senden eines Pakets

```
int rethdd_promethos_comm_register_receive( promethos_comm_hal_t *hal,
promethos_comm_halrecv_f *receive, void *commpriv )
```

Diese Funktion wird von der Kommunikationsinfrastruktur aufgerufen, um ihre Empfangsfunktion zu registrieren, welche dann der HAL aufruft, um ein über ein Kontrollinterface erhaltenes Infrastrukturpaket an die Infrastruktur zu übergeben. `receive` zeigt auf die Empfangsfunktion, `hal` zeigt auf die im Abschnitt 3.3.3 auf Seite 24 beschriebene HAL Datenstruktur und `commpriv` ist ein Pointer, dessen Bedeutung nur die Infrastruktur kennt.

Es werden zunächst die Parameter überprüft und ein negativer Standardfehlerwert zurückgegeben, falls mit ihnen etwas nicht in Ordnung ist. Ansonsten werden `receive` und `commpriv` in der im Abschnitt 3.3.3 auf Seite 24 beschriebenen HAL Datenstruktur eingetragen und Null zurückgegeben.

`int rethdd_promethos_comm_unregister_receive( promethos_comm_hal_t *hal, promethos_comm_halrecv_f *receive, void *commpriv )`

Diese Funktion wird von der Kommunikationsinfrastruktur aufgerufen, um ihre Empfangsfunktion wieder zu deregistrieren. `receive` zeigt auf die Empfangsfunktion, `hal` zeigt auf die HAL Datenstruktur und `commpriv` ist ein Pointer, dessen Bedeutung nur die Infrastruktur kennt.

Es werden wiederum zunächst die Parameter überprüft und ein negativer Standardfehlerwert zurückgegeben, falls mit ihnen etwas nicht in Ordnung ist. Ansonsten wird ein eventuell noch für das Reassemblieren fragmentierter Pakete allozierter Socket Buffer (`struct sk_buff`) freigegeben, dann werden `receive` und `commpriv` aus der HAL Datenstruktur ausgetragen und Null zurückgegeben.

`struct sk_buff *rethdd_promethos_comm_alloc_skb( int size, promethos_comm_hal_t *hal )`

Diese Funktion wird von der Kommunikationsinfrastruktur aufgerufen, um einen Socket Buffer zu allozieren, dessen Inhalt später über die von diesem HAL repräsentierte Verbindung gesendet werden soll. `size` ist die Grösse der zu sendenden Daten, `hal` zeigt auf die HAL Datenstruktur.

Ist `size > PROMETHOS_COMM_MAX_SIZE`, so kann die angeforderte Datengrösse nicht in einem einzigen Frame gesendet werden und muss fragmentiert werden. In diesem Falle wird die Variable `hdrsize` (welche die Gesamtgrösse der HAL spezifischen Headers enthält) um die Grösse der Datenstruktur `promethos_comm_frag_t` inkrementiert, welche die Gesamtlänge eines fragmentierten Pakets enthält:

```
typedef struct promethos_comm_frag {
        u32     len;
} promethos_comm_frag_t;
```

Danach wird versucht, mit `dev_alloc_skb( size + hdrsize )` den Socket Buffer zu allozieren. Klappt das nicht, wird eine Fehlermeldung aus- und `NULL` zurückgegeben. Ansonsten wird mit `skb_reserve( skb, hdrsize )` der Anfang des Socket Buffers für die HAL Headers reserviert und schliesslich der Pointer auf den Socket Buffer zurückgegeben.

`int rethdd_promethos_comm_send_fragment( struct sk_buff *skb, struct net_device *dev, int independent )`

Diese Funktion wird vom HAL verwendet, um ein Fragment zu senden. `skb` zeigt auf den Socket Buffer, der das Fragment enthält, `net_device` zeigt auf das Kontrollinterface, über welches das Fragment gesendet werden soll, und `independent` besagt, ob es sich um ein einzelnes Fragment (also ein Paket, das in einem einzigen Fragment Platz findet) handelt oder um einen Teil eines fragmentierten Pakets.

Die hardwarespezifischen Teile dieser Funktion wurden der bestehenden Proxy Device Driver Funktion `rethdd_hard_start_xmit()` entnommen, hier werden hauptsächlich die für den HAL eingefügten Teile beschrieben.

Zunächst wird den Fragmentdaten ein Proxy Device Driver Guided Frame Header vorangestellt. Dieser wird geeignet initialisiert, damit das Frame auf die Gegenseite übertragen wird und dort wieder als Kommunikationsinfrastrukturframe erkannt werden kann. Ausserdem wird `independent` so hineinkodiert, dass beim Empfang fragmentierte Pakete erkannt und wieder reassembliert werden können.

Schliesslich wird dem Guided Frame ein Ethernet Frame Header vorangestellt und das Frame wird mittels der Funktion `hard_start_xmit()` des Parent Interface (auf dem Host) bzw. direkt mittels `reth_pciXmit()` (auf dem ePPC) gesendet. Es wird der Rückgabewert der eigentlichen Sendefunktion zurückgegeben.

`int rethdd_promethos_comm_send( struct sk_buff *skb, promethos_comm_hal_t *hal )`

Diese Funktion wird von der Infrastruktur aufgerufen, um ein Paket zu senden. `skb` zeigt auf den Socket Buffer, der das Paket enthält, `hal` zeigt auf die HAL Datenstruktur.

Zunächst werden wiederum die Parameter überprüft, und es wird ein negativer Standardfehlerwert zurückgegeben, falls mit ihnen etwas nicht stimmt.

Ist die Paketlänge `skb->len <= PROMETHOS_COMM_MAX_SIZE`, so kann das Paket in einem einzigen Frame gesendet werden, und es wird direkt der Rückgabewert von `rethdd_promethos_comm_send_fragment()` weitergereicht.

Andernfalls muss das Paket in mehrere Frames fragmentiert werden. Dazu wird in einer Schleife mit `rethdd_promethos_comm_alloc_skb()` für jedes Fragment ein Socket Buffer alloziert, der entsprechende Teil des Pakets hineinkopiert und das Fragment mit `rethdd_promethos_comm_send_fragment()` gesendet. Tritt dabei zu irgendeinem einem Zeitpunkt ein Problem auf, so wird die Schleife abgebrochen und ein entsprechender negativer Standardfehlerwert zurückgegeben. Hat alles geklappt, so wird schliesslich Null zurückgegeben.

Funktionen zum Empfangen eines Pakets

Hierfür wurde keine neue Funktion hinzugefügt, sondern lediglich eine bestehende Funktion erweitert:

```
int reth_ctrl_rcv( struct sk_buff *skb, netDevice_t *dev, struct
packet_type *pt)
```

Diese Funktion wird aufgerufen, wenn über ein Kontrollinterface ein Frame empfangen wurde. Neu werden von `rethdd_promethos_comm_send_fragment()` gesendete Frames erkannt. Handelt es sich dabei um ein unfragmentiertes Paket, so wird dieses direkt mittels der Infrastrukturhilfsfunktion `promethos_comm_hal_receive()` an die Infrastrukturempfangsfunktion übergeben und deren Rückgabewert weitergereicht.

Handelt es sich hingegen um das erste Fragment eines fragmentierten Pakets, so wird der `promethos_comm_frag_t` Struktur am Anfang dieses Fragments die Paketlänge entnommen und ein Socket Buffer dieser Grösse für das Reassemblieren alloziert. In den folgenden Aufrufen dieser Funktion wird dann der Inhalt jedes Fragments an die entsprechende Stelle dieses Socket Buffers kopiert. Nach dem letzten Fragment wird schliesslich das reassemblierte Paket an die Infrastrukturempfangsfunktion übergeben und deren Rückgabewert weitergegeben.

4.2. Hardwareunabhängiger Infrastrukturkern

Der hardwareunabhängige Infrastrukturkern wurde als Kernelmodul `promethos_comm` realisiert. Es werden zunächst einige allgemeine Funktionen beschrieben, dann die Funktionen, welche die Schnittstellen zum Node Manager und zum Plugin Manager implementieren.

4.2.1. Allgemeine Funktionen

```
int __init promethos_comm_module_init( void )
```

Diese Funktion wird aufgerufen, wenn das Kernelmodul geladen wird.

Es wird zunächst der Wert des Modulparameters `my_id` überprüft, welcher die Kennung dieses Knotens repräsentiert. Diese ist normalerweise zu Beginn `0` (`PROMETHOS_COMM_CPUID_UNASSIGNED`), was bedeutet, dass diesem Knoten noch keine Kennung zugewiesen wurde. Soll der Knoten andererseits die Rolle des Masterknotens der Infrastruktur übernehmen, so muss das Kernelmodul mit `my_id=1` (`PROMETHOS_COMM_CPUID_MASTER`) geladen werden. Andere Werte für `my_id` sind zu Beginn nicht erlaubt und werden mit einem Abbruch des Modulladevorgangs mit `-EINVAL` quittiert.

Ist die initiale Kennung in Ordnung, so wird mit der Funktion

`promethos_comm_init_routing()` (S. 34) die Routingtabelle initialisiert. Danach werden die `/proc` Filesystem Files angelegt, welche die Schnittstelle zum Node Manager ausmachen, siehe Seite 36. Tritt beim Anlegen eines dieser Files ein Fehler auf, so werden die zuvor erfolgreich angelegten Files wieder entfernt, und der negative Standardfehlerwert wird weitergeleitet, womit der Ladevorgang des Moduls abgebrochen wird. Ansonsten wird Null zurückgegeben und damit der Ladevorgand erfolgreich abgeschlossen.

`void __exit promethos_comm_module_exit( void )`

Diese Funktion wird beim Entladen des Kernelmoduls aufgerufen. Es werden zunächst mit der Funktion `promethos_comm_remove_all_links()` sämtliche allozierte Datenstrukturen mit Informationen über Verbindungen zwischen Knoten freigegeben und danach die in `promethos_comm_module_init()` angelegten `/proc` Filesystem Files entfernt.

`int promethos_comm_init_routing( void )`

Diese Funktion initialisiert die Routingtabelle. Um zu verstehen, wie dies geschieht, muss hier ein wenig ausgeholt werden, um den Aufbau der Routingtablle zu erläutern. Der Datentyp für die Repräsentation einer Kennung ist wie folgt definiert:

```
typedef u16 promethos_comm_cpuid_t;
```

Es wurde entschieden, Kennungen durch eine 16 Bit breite Zahl zu repräsentieren, weil das einerseits gross genug sein sollte, um auch für sehr grosse hierarchische ANNs genügend Spielraum zu bieten, aber andererseits auch klein genug ist, um Informationen zu den maximal 65536 verschiedenen Knoten direkt in Tabellen in Form von Arrays zu verwalten. Die Routingtabelle ist zum Beispiel folgendermassen definiert. Zunächst der Datentyp, der eine Metrik repräsentiert:

```
typedef enum {
        PROMETHOS_COMM_METRIC_HOPS      = 0,
        PROMETHOS_COMM_NUM_METRICS
} promethos_comm_metric_t;
```

Darauf aufbauend der Typ eines Routingtabelleneintrags:

```
typedef struct promethos_comm_route {
        struct {
                promethos_comm_hal_t    *hal;
                u32                     best;
        } metrics[PROMETHOS_COMM_NUM_METRICS];
} promethos_comm_route_t;
```

Und schliesslich die Routingtabelle als Array von Routingtabelleneinträgen:

```
#define PROMETHOS_COMM_MAX_TARGET       \
        ( ( 1 << ( 8 * sizeof( promethos_comm_cpuid_t ) ) ) - 1 )

static promethos_comm_route_t routing_table[PROMETHOS_COMM_MAX_TARGET+1];
```

Die Routingtabelle vermerkt also für jede Kennung und jede Metrik die HAL Datenstruktur, über welche Pakete optimal bezüglich dieser Metrik zum Knoten mit dieser Kennung verschickt werden können, und den entsprechenden optimalen Wert dieser Metrik. Bisher ist allerdings nur eine Metrik implementiert, die Anzahl Hops (d.h. Verbindungen zwischen Knoten), die von einem Paket zurückgelegt werden müssen, um einen Knoten zu erreichen. Diese Anzahl sollte immer minimal sein. Um die Routingtabelle zu initialisieren, wird deshalb die optimale Anzahl Hops für alle Kennungen auf `PROMETHOS_COMM_MAX_TARGET - 1` gesetzt, was mehr ist, als ein Paket zu irgend einem konkreten Knoten zurücklegen muss. Ausserdem wird der Pointer auf die HAL Datenstruktur für jede Kennung auf NULL initialisiert, was bedeutet, dass kein Knoten mit dieser Kennung erreichbar ist.

Sind Verbdingungen zu Peers bekannt, so werden diese nun in die Routingtablle eingearbeitet. Jeder Peer ist mit einem Hop erreichbar, und Pointers auf die HAL Datenstrukturen für alle Peers sind in der Liste `peer_hals` vermerkt.

Schliesslich wird mit der Funktion `promethos_comm_set_self_routing()` ein Routingtabelleneintrag angelegt, damit auch Pakete an die eigene Kennung gesendet werden können. Dies bietet den Vorteil, dass der Master Knoten auch gleichzeitig die Rolle eines "normalen" Knotens übernehmen kann, ohne dass der Code mit Master/nicht Master Tests übersät werden muss. Der Masterknoten befolgt einfach das Protokoll zwischen Master und "normalem" Knoten mit sich selbst in beiden Rollen.

```
static promethos_comm_hal_t* promethos_comm_get_hal_for_target(
promethos_comm_cpuid_t target, promethos_comm_metric_t metric )
```

Diese Funktion liefert den Pointer auf die HAL Datenstruktur des HALs, über welchen Pakete zum Knoten mit der Kennung `target` optimal bezüglich der Metrik `metric` gesendet werden können. Ist einer der Parameter ungültig, wird `NULL` zurückgegeben, ansonsten der in der Routingtabelle für die angegebene Kombination aus Zielkennung und Metrik vermerkte Pointer.

```
void promethos_comm_init_header( promethos_comm_header_t *header,
promethos_comm_type_t type, promethos_comm_cpuid_t target, u32 datasize )
```

Mit dieser Funktion kann ein im Abschnitt 3.3.1 auf Seite 21 beschriebener Paketheader initialisiert werden. `header` ist ein Pointer auf den Anfang des Headers, `type` ist der Pakettyp, `target` ist die Kennung des Knotens, für den das Paket bestimmt ist, und `datasize` ist die Länge der Nutzdaten.

Der Header enthält ausser den Parametern noch die Absenderkennung, welche auf den Wert der Variable `my_id` gesetzt wird, und die Headergrösse, welche einer Tabelle entnommen wird:

```
static u32
hdrsizes[PROMETHOS_COMM_NUM_TYPES] =
{
        [PROMETHOS_COMM_DATA]         = sizeof( promethos_comm_data_t ),
        [PROMETHOS_COMM_DISCOVER]     = sizeof( promethos_comm_discover_t ),
        [PROMETHOS_COMM_REQUEST_ID]   = sizeof( promethos_comm_request_id_t ),
        [PROMETHOS_COMM_ASSIGN_ID]    = sizeof( promethos_comm_assign_id_t ),
};
```

4.2.2. Interface zum Node Manager

Der Node Manager (NM) kann über das `/proc` Filesystem mit der Kommunikationsinfrastruktur interagieren:

`/proc/net/promethos_comm/my_id`: Aus diesem File kann der NM die aktuelle Kennung dieses Knotens auslesen, siehe `promethos_comm_proc_my_id()`.

`/proc/net/promethos_comm/links/register`: Mit diesem File kann der NM eine Verbindung zu einem Peer registrieren, siehe `promethos_comm_proc_register()`.

`/proc/net/promethos_comm/links/unregister`: Mit diesem File kann der NM die Verbindung zu einem Peer wieder deregistrieren, siehe `promethos_comm_proc_unregister()`.

Es folgen nun Beschreibungen der Funktionen, welche diese Schnittstelle implementieren.

```
int promethos_comm_proc_my_id( char *buffer, char **start, off_t offset,
int length, int *eof, void *data )
```

Diese Funktion wird aufgerufen, wenn vom Userspace aus dem File `/proc/net/promethos_comm/my_id` gelesen wird. Es wird dafür gesorgt, dass der Inhalt der Variable `my_id` ausgelesen wird, d.h. die aktuelle Kennung dieses Knotens.

```
int promethos_comm_proc_register( struct file *file, const char *buffer,
unsigned long count, void *data )
```

Diese Funktion wird aufgerufen, wenn vom Userspace in das File `/proc/net/promethos_comm/links/register` geschrieben wird. Es wird versucht, aus den geschriebenen Daten die Kennung eines Geräts zu parsen, welches eine im Abschnitt 3.3.3 auf Seite 24 beschrieben HAL Datenstruktur anbietet. Bisher werden nur Netzwerkinterfaces unterstützt, welche mit `netif <Interfacename>` identifiziert werden können, z.B. `netif npctl0`. Ist kein Netzwerkinterface mit dem angegebenen Namen bekannt, bietet es keine HAL Datenstruktur an oder enthält diese nicht Pointers zu allen benötigten Funktionen, so wird der Fehlerwert `-ENODEV` zurückgegeben.

Ist die HAL Datenstruktur bereits als Verbindung zu einem Peer vermerkt, dann wird

der Fehlerwert `-EBUSY` zurückgegeben. Sonst wird mit `promethos_comm_add_peer_hal()` ein entsprechender Vermerk in der Liste `peer_hals` angelegt.

Es wird mit den Funktionen `register_receive()`, `alloc_skb()` und `send()` aus der HAL Datenstruktur `promethos_comm_receive()` (S. 39) als Empfangsfunktion registriert und ein Discovery Paket an den Peer gesendet. Schliesslich wird mit `MOD_INC_USE_COUNT` der Reference Count des Kernelmoduls inkrementiert und damit verhindert, dass das Modul entladen werden kann, während Verbindungen zu Peers vermerkt sind. Dies ist eine Vorsichtsmassnahme, um zu verhindern, dass nach dem Entladen ungültige Pointer auf `promethos_comm_receive()` in Gebrauch bleiben.

```
int promethos_comm_proc_unregister( struct file *file, const char
*buffer, unsigned long count, void *data )
```

Diese Funktion wird aufgerufen, wenn vom Userspace in das File `/proc/net/promethos_comm/links/unregister` geschrieben wird. Wie in `promethos_comm_proc_unregister()` wird versucht, den Namen eines Netzwerkinterface zu parsen und die von diesem angebotene HAL Datenstruktur zu finden. Dabei auftretende Probleme werden durch Rückgabe entsprechender negativer Standardfehlerwerte signalisiert.

Ist die HAL Datenstruktur in der `peer_hals` Liste vermerkt, so wird mit ihrer `unregister_receive()` Funktion `promethos_comm_receive()` als Empfangsfunktion deregistriert und der Vermerk mit `promethos_comm_remove_peer_hal()` entfernt. Schliesslich wird mit `MOD_DEC_USE_COUNT` der Reference Count des Kernelmoduls dekrementiert und damit das Entladen des Modul wieder ermöglicht, sobald keine Verbindungen zu Peers mehr vermerkt sind.

4.2.3. Interface zum Plugin Manager

```
int promethos_comm_register( promethos_commrecv_f *receive )
```

Mit dieser Funktion kann der Plugin Manager seine Empfangsfunktion registrieren, welche dann von der Kommunikationsinfrastruktur aufgerufen wird, um ihm empfangene Pakete zu übergeben, die vom Plugin Manager auf einem anderen Knoten an diesen Knoten gesendet wurden. `receive` ist der Pointer auf die Empfangsfunktion vom Typ:

```
typedef int ( promethos_commrecv_f )( promethos_comm_cpuid_t source,
                                      struct sk_buff *skb );
```

Ist `receive == NULL`, so wird `-EINVAL` zurückgegeben. Ist bereits eine Empfangsfunktion registriert, wird `-EBUSY` zurückgegeben. Ist jedoch alles in Ordnung, wird die Empfangsfunktion registriert und Null zurückgegeben.

`int promethos_comm_unregister( promethos_commrecv_f *receive )`

Mit dieser Funktion kann der Plugin Manager seine Empfangsfunktion wieder deregistrieren. `receive` ist der Pointer auf die Empfangsfunktion. Ist `receive == NULL`, so wird `-EINVAL` zurückgegeben. Ist keine Empfangsfunktion registriert, so wird `-ENOENT` zurückgegeben. Zeigt `receive` nicht auf die registrierte Empfangsfunktion, so wird `-EINVAL` zurückgegeben. Ansonsten wird die Empfangsfunktion deregistriert und Null zurückgegeben.

`struct sk_buff* promethos_comm_alloc_skb( promethos_comm_cpuid_t target, int size )`

Mit dieser Funktion kann der Plugin Manager einen Socket Buffer (`struct sk_buff`) allozieren, um ein Paket der Grösse `size` an den Plugin Manager auf dem Knoten mit der Kennung `target` zu senden.

Es wird zunächst mit `promethos_comm_get_hal_for_target()` (S. 35) der Pointer auf die HAL Datenstruktur des Peers ermittelt, über welche das Paket mit der kleinsten Anzahl Hops zum Knoten mit der Kennung `target` gesendet werden kann. Ist der ermittelte Pointer `NULL`, so ist offenbar kein Knoten mit der Kennung `target` erreichbar, und es wird `NULL` zurückgegeben. Dasselbe geschieht, falls die HAL Datenstruktur keinen Pointer auf eine `alloc_skb()` Funktion anbieten sollte. Ansonsten wird mit dieser der Socket Buffer alloziert, wobei `size` um die Grösse des im Abschnitt 3.3.1 auf Seite 21 beschriebenen Paketheaders der Infrastruktur erweitert wird:

```
typedef struct promethos_comm_header {
        promethos_comm_cpuid_t  from, to;
        promethos_comm_type_t   type;
        u32                     hdrsize, datasize;
} promethos_comm_header_t;
```

Schlägt das Allozieren des Socket Buffers fehl, wird `NULL` zurückgegeben. Andernfalls wird mit `promethos_comm_init_header()` (S. 35) der Infrastrukturheader initialisiert und mit `skb_reserve()` der Headerbereich reserviert, so dass die Nutzdaten nach dem Header zu liegen kommen. Schliesslich wird der Pointer auf den Socket Buffer zurückgegeben.

`int promethos_comm_send( struct sk_buff *skb )`

Mit dieser Funktion kann der Plugin Manager ein Paket senden.

`skb` ist der Zeiger auf den Socket Buffer. Ist `skb == NULL`, so wird `-EINVAL` zurückgegeben. Andernfalls wird mit `skb_push` ein Pointer auf den Infrastrukturheader ermittelt und überprüft, ob der Plugin Manager den Socket Buffer wie erforderlich mit `promethos_comm_alloc_skb()` alloziert hat. Ist das nicht der Fall, so wird `-EINVAL` zurückgegeben.

Ansonsten wird mit `promethos_comm_get_hal_for_target()` (S. 35) der Pointer auf die HAL Datenstruktur des Peers ermittelt, über welche das Paket mit der kleinsten Anzahl Hops zum Knoten mit der im Header als Ziel vermerkten Kennung gesendet werden kann. Ist der ermittelte Pointer `NULL`, so ist offenbar kein Knoten mit der vermerkten Kennung erreichbar, und es wird `-ENOENT` zurückgegeben. Dasselbe geschieht, falls die HAL Datenstruktur keinen Pointer auf eine `send()` Funktion anbietet. Andernfalls wird mit dieser versucht, das Paket zu senden, und ihr Rückgabewert wird weitergegeben.

```
int promethos_comm_receive( struct sk_buff *skb, promethos_comm_hal_t *hal
)
```

Diese Funktion wird von der Kommunikationsinfrastruktur in HAL Datenstrukturen als Empfangsfunktion registriert und dann aufgerufen, wenn ein HAL ein Infrastrukturpaket empfangen hat. `skb` ist ein Pointer auf den Socket Buffer, der das Paket enthält, und `hal` ist ein Pointer auf die HAL Datenstruktur des HAL, welcher das Paket empfangen hat. Ist einer der Parameter `NULL`, so wird `-EINVAL` zurückgegeben.

Andernfalls wird der Infrastrukturheader am Anfang des Pakets ausgewertet. Da die Infrastruktur innerhalb eines ANN auf verschiedenen Prozessorarchitekturen laufen kann, müssen die Headerwerte, welche mehrere Bytes lang sind, im sogenannten Network Byte Order[1] im Header stehen und mit den `ntohl()` bzw. `ntohs()` Funktionen ausgelesen werden. Zuerst wird der Pakettyp überprüft:

```
typedef enum {
        PROMETHOS_COMM_DATA             = 0,
        PROMETHOS_COMM_DISCOVER         = 1,
        PROMETHOS_COMM_REQUEST_ID       = 2,
        PROMETHOS_COMM_ASSIGN_ID        = 3,
        PROMETHOS_COMM_LINKS            = 4,
        PROMETHOS_COMM_NUM_TYPES
} promethos_comm_type_t;
```

Ist der im Header kodierte Typ `>= PROMETHOS_COMM_NUM_TYPES`, so ist der Typ unbekannt und es wird `-EINVAL` zurückgegeben. Auch falls die Absenderkennung ungültig ist und es sich nicht um ein Discover Paket handelt wird `-EINVAL` zurückgegeben. Dasselbe gilt schliesslich auch, falls die im Header kodierte Headergrösse nicht mit der erwarteten Grösse übereinstimmt.

Ist das Paket nicht an diesen Knoten gerichtet, so wird versucht, es in Richtung seines Ziels weiterzuleiten. Dazu wird mit `promethos_comm_get_hal_for_target()` ein Pointer auf eine HAL Datenstruktur ermittelt und zuerst versucht, den Socket Buffer direkt mit deren `send()` Funktion weiterzuleiten. Klappt das nicht, so wird mit `alloc_skb()` aus der HAL Datenstruktur ein neuer Socket Buffer alloziert, das Paket in diesen kopiert und dann versucht, das Paket mit diesem Socket Buffer weiterzuleiten. Das Ergebnis der Bemühungen um die Weiterleitung spiegelt sich im Rückgabewert wider.

[1] Auch als Big Endian bekannt.

Ist das Paket an diesen Knoten gerichtet, so wird es zur weiteren Verarbeitung an eine Dispatchfunktion übergegeben, deren Rückgabewert weitergereicht wird:

```
promethos_comm_receive_dispatch_f*
receive_dispatch[PROMETHOS_COMM_NUM_TYPES] =
{
        [PROMETHOS_COMM_DATA]          = promethos_comm_receive_data,
        [PROMETHOS_COMM_DISCOVER]      = promethos_comm_receive_discover,
        [PROMETHOS_COMM_REQUEST_ID]    = promethos_comm_receive_request_id,
        [PROMETHOS_COMM_ASSIGN_ID]     = promethos_comm_receive_assign_id,
        [PROMETHOS_COMM_LINKS]         = promethos_comm_receive_links,
};
```

In `promethos_comm_receive_data()` werden empfangene Datenpakete an die registrierte Plugin Manager Empfangsfunktion übergeben, in den anderen Funktionen der Hauptteil des in der Abbildung 3.3 auf Seite 23 dargestellten Protokolls zur Einbindung neuer Knoten in die Infrastruktur und Zuweisung eindeutiger Kennungen implementiert:

- In `promethos_comm_receive_discover()` wird ein Antrag um Zuweisung einer eindeutigen Kennung an den Masterknoten gesendet, wenn das erhaltene Discover Paket von einem bisher unbekannten Peer stammt (Schritt 3).
- In `promethos_comm_receive_request_id()` wird auf dem Masterknoten der Antrag um Zuweisung einer eindeutigen Kennung entgegengenommen. Es werden eine neue eindeutige Kennung zurückgesendet (Schritt 4) und in einem Broadcast Informationen über alle bekannten Verbindungen zwischen Knoten an alle erreichbaren Knoten gesendet (Schritt 7).
- In `promethos_comm_receive_assign_id()` wird die Zuweisung der eindeutigen Kennung an den neuen Peer weitergeleitet (Schritt 5) bzw. die zugewiesene Kennung angenommen (Schritt 6).
- In `promethos_comm_receive_links()` schliesslich werden auf jedem Knoten die vom Masterknoten stammenden Informationen über Verbindungen zwischen Knoten ausgewertet, um eine aktuelle Routingtabelle aufzubauen (Schritt 7).

Bevor diese Funktion mit einem Fehler abbricht, gibt sie den Socket Buffer mit `dev_kfree_skb()` frei. Ansonsten ist die Funktion dafür verantwortlich, an welche das Paket weitergeleitet wird.

5. Evaluation

Dieses Kapitel widmet sich der Evaluation der in Kapitel 4 ausführlich beschriebenen Implementation. Es wird erläutert, wie ihr korrektes Funktionieren überprüft und ihre Performance gemessen wurde.

5.1. Methodik

Die Hauptfunktionalität der Kommunikationsinfrastruktur, nämlich das Senden und Empfangen von Paketen, ist nur im Kernel Space zugänglich. Somit muss auch die Überprüfung des korrekten Funktionierens und das Messen der Performance im Kernel Space erfolgen.

Zu diesem Zweck wurde ein Kernelmodul namens `promethos_comm_test` entwickelt. Seine Grundidee besteht darin, ein Testpaket wählbarer Grösse eine wählbare Anzahl Male zwischen zwei Knoten hin und her zu schicken und damit die Plugin Manager Schnittstelle der Kommunikationsinfrastruktur zu benutzen, die korrekte Übertragung zu verifizieren und deren Performance zu messen.

Es folgt eine Beschreibung seiner Funktionen, welche die wichtigsten Aspekte der Evaluation abdecken.

```
int __init promethos_comm_test_init( promethos_comm_cpuid_t target )
```

Diese Funktion wird beim Laden des Moduls aufgerufen. Es werden zuerst die Modulparameter überprüft:

`promethos_comm_cpuid_t target:` Die Kennung des Knotens, an den Testpakete gesendet werden sollen. Es muss eine gültige Kennung angegeben werden, sonst wird der Fehlerwert `-EINVAL` zurückgegeben und damit das Laden des Moduls abgebrochen.

`int size:` Die Grösse der zu sendenden Testpakete. Wird nicht explizit eine Grösse angegeben, so wird die Minimalgrösse verwendet, nämlich die Grösse des `struct promethos_comm_test`, in welchem die Parameter während des Hin- und Hersendens zwischen den beiden Knoten verwaltet werden. Wird eine kleinere Grösse übergeben, so wird wiederum das Laden des Moduls mit `-EINVAL` abgebrochen.

`int num:` Die Anzahl der durchzuführenden Sendevorgänge, Defaultwert 2. Ist die Anzahl gerade, so ist das Testpaket am Ende wieder beim Ausgangsknoten, und dieser

kann aussagekräftige Werte für den mittleren Durchsatz und die mittlere Latenz berechnen.

Danach wird versucht, mit `promethos_comm_register_receive()` die später beschriebene Funktion `promethos_comm_test_recv()` als Empfangsfunktion zu registrieren. Gelingt dies nicht, so wird der begründende Fehlerwert weitergeleitet und damit das Laden des Moduls abgebrochen.

Als nächster Schritt wird versucht, mit `promethos_comm_alloc_skb()` einen Socket Buffer für den Versand eines Pakets der Grösse `size` an den Knoten mit der Kennung `target` zu allozieren. Ist der erhaltene Pointer `NULL`, so wird das Laden des Moduls mit `-ENOMEM` abgebrochen.

Am Anfang des Socket Buffers wird das `struct promethos_comm_test` initialisiert. Die Felder `num` und `counter` werden auf den Wert des Modulparameters `num` gesetzt. Die Felder `sec` und `usec` werden auf die mit `do_gettimeofday()` ermittelten Werte der aktuellen Systemzeit gesetzt. Der Rest des Socket Buffers wird mit einem sich wiederholenden Bytemuster aufgefüllt.

Zu guter Letzt wird versucht, mit `promethos_comm_send()` das Paket zu senden. Geht das schief, wird der begründende Fehlerwert weitergeleitet und damit das Laden des Moduls abgebrochen. Ansonsten wird Null zurückgegeben und damit das Laden des Moduls erfolgreich abgeschlossen.

```
int promethos_comm_test_recv( promethos_comm_cpuid_t source, struct
sk_buff *skb )
```

Die Kommunikationsinfrastruktur ruft diese Funktion auf, um ein empfangenes Paket zu übergeben. `source` ist die Kennung des Knotens, von dem das Paket stammt. `struct sk_buff *skb` ist ein Pointer auf den Socket Buffer, der das Paket enthält.

Es wird zunächst das `counter` Feld des `struct promethos_comm_test` am Anfang des Pakets ausgelesen und dekrementiert.

Ist das Ergebnis grösser Null, so wird der dekrementierte Wert ins `counter` Feld zurückgeschrieben, das Paket in einen neuen Socket Buffer kopiert und an den Absender zurückgeschickt.

Ist das Ergebnis gleich Null, so ist das Paket am Ende seiner Reise angelangt. Es wird zunächst das Bytemuster am Ende des Pakets überprüft. Stimmt es nicht mit dem ursprünglichen Muster überein, so ist offenbar ein Übertragungsfehler aufgetreten, und es wird ausgegeben, an welcher Stelle das Muster wie abweicht. Sonst wird ausgegeben, dass die Paketdaten intakt sind.

Danach wird die Differenz Δt zwischen den im `struct promethos_comm_test` festgehaltenen und den erneut mit `do_gettimeofday()` ermittelten Zeitwerten berechnet. Aus der Anzahl Sendevorgänge `num` und der Länge des Pakets `skb->len` kann damit der mittlere Durchsatz berechnet werden:

$$Durchsatz = \frac{\texttt{num} \cdot \texttt{skb->len}}{\Delta t}$$

Ausserdem kann aus Δt und `num` die mittlere Latenz berechnet werden:

$$Latenz = \frac{\Delta t}{\texttt{num}}$$

Zum Schluss wird der empfangene Socket Buffer `skb` mit `dev_kfree_skb()` freigegeben.

`void __exit promethos_comm_test_exit()`

Diese Funktion wird beim Entladen des Moduls aufgerufen. Es wird versucht, mit `promethos_comm_unregister_receive()` die Empfangsfunktion wieder zu deregistrieren. Es wird eine Fehlermeldung ausgegeben, falls dies fehlschlägt, aber das Entladen des Moduls kann nicht verhindert werden.

Nachfolgend ein Beispiel der Ausgabe des Moduls, nachdem es mit den Parametern `target=1 size=55476 num=2000` geladen wurde:

```
promethos_comm_test_recv: Received last packet out of 2000 of size 55476 from node 1.
promethos_comm_test_recv: Data is intact.
promethos_comm_test_recv: Interchanged 110952000 bytes in 13.301126 seconds; average throughput: 8318300 B/s.
promethos_comm_test_recv: Average latency: 13301126 / 2000 = 6650 microseconds
```

Die erste Zeile bestätigt, dass die Modulparameter korrekt umgesetzt wurden. Die zweite Zeile zeigt an, dass das Datenmuster am Ende des Testpakets korrekt übertragen wurde. Die dritte Zeile zeigt die Berechnung des mittleren Durchsatzes. Die vierte und letzte Zeile zeigt schliesslich die Berechnung der mittleren Latenz.

5.1.1. Korrektheit

Aus Abschnitt 5.1 ist ersichtlich, dass das Modul `promethos_comm_test` alle Funktionen der Schnittstelle benutzt, welche die Kommunikationsinfrastruktur dem Plugin Manager anbietet. Folglich ist es dazu geeignet, das erwartungsgemässe Funktionieren dieser Schnittstelle zu überprüfen. Dies wurde nach Abschluss der Implementation erfolgreich gemacht.

5.1.2. Performance

Aus Abschnitt 5.1 ist ersichtlich, dass das Modul `promethos_comm_test` mit dem mittleren Durchsatz und der mittleren Latenz die massgeblichen Performancewerte eines paketbasierten Kanals ermittelt. Folglich ist es dazu geeignet, die konkrete Performance der Kommunikationsinfrastruktur auf einer bestimmten Hardwarekonfiguration zu messen. Der folgende Abschnitt enthält ausführliche solche Messungen.

5.2. Messungen

Es wurde eine Reihe von Messungen mit dem im Abschnitt 5.1 beschriebenen Kernelmodul `promethos_comm_test` durchgeführt. Dazu wurde das Modul zuerst auf einem Knoten geladen, dann auf einem anderen Knoten wiederholt mit verschiedenen Parametern geladen, auf die Ausgabe der Messungen gewartet, das Modul wieder entladen, usw. Ein Knoten lief dabei auf der embedded PowerPC CPU auf einem PowerNP Application Reference Board (ARB):

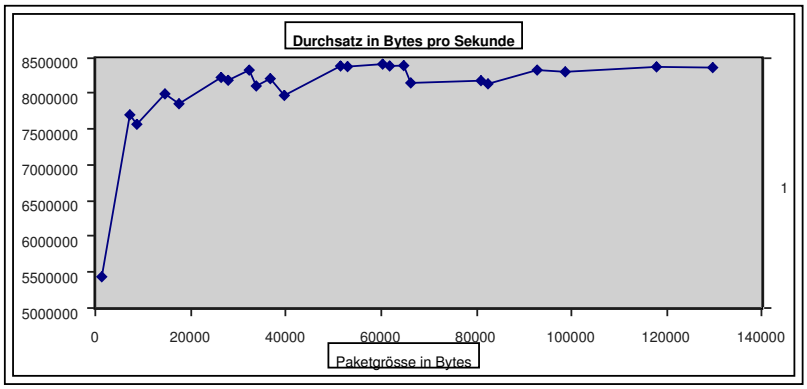
```
cpu             : unknown (40b10952)
clock           : 133MHz
revision        : 9.82 (pvr 40b1 0952)
bogomips        : 132.71
machine         : IBM Rainier
plb bus clock   : 100MHz
```

Der andere Knoten lief auf der Host CPU der x86 Entwicklungsmaschine:

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 8
model name      : Pentium III (Coppermine)
stepping        : 6
cpu MHz         : 803.432
cache size      : 256 KB
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 2
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 mmx fxsr sse
bogomips        : 1602.35
```

Im Rahmen dieser Messungen wurde nur der `size` Modulparameter variiert, um die Auswirkungen der Paketgrösse auf die Performance zu ermitteln. Der `num` Parameter wurde immer auf `2000` gesetzt, das heisst, das Testpaket wurde in jedem Messlauf von jedem der beiden Knoten je 1000 Mal an den anderen gesendet.

Um die gewählten Paketgrössen zu verstehen, muss man berücksichtigen, wie beim verwendeten PowerNP ARB die Kommunikation implementiert ist. Da der PowerNP und die Host CPU nur über einen Ethernet MAC Chip kommunizieren können, werden die Infrastrukturpakete in spezielle Ethernet Frames eingebettet. Deren Grösse ist durch die für die Kommunikation zwischen den PowerNP Picocode Einheiten und der embedded PowerPC CPU verwendeten D6 RAM Buffers begrenzt. Grössere Pakete werden wie in Abschnitt 4.1.2 auf Seite 31 beschrieben vor dem Senden fragmentiert und nach dem Empfang wieder reassembliert. Für die Messungen wurden nun die Paketgrössen so gewählt, dass sie eine bestimmte Anzahl Frames der Maximalgrösse ausfüllten.



Frames/Paket	Bytes/Paket	Messung 1	Messung 2	Messung 3	Durchschnitt
1	1460	5421472	5420707	5424554	5422244
5	7360	7681676	7687548	7688603	7685942
6	8836	7552023	7551333	7558700	7554019
10	14740	7976716	7976384	7983352	7978817
12	17692	7835043	7843346	7843934	7840774
18	26548	8208769	8209249	8208619	8208879
19	28024	8169454	8169463	8169154	8169357
22	32452	8311624	8311684	8311493	8311600
23	33928	8089347	8089498	8089240	8089362
25	36880	8192144	8191956	8191882	8191994
27	39832	7957362	7957491	7957221	7957358
35	51640	8368403	8368370	8368532	8368435
36	53116	8360286	8360229	8360352	8360289
41	60496	8395033	8395070	8395144	8395082
42	61972	8369013	8368843	8368905	8368920
44	64924	8374531	8374560	8374672	8374588
45	66400	8132015	8132059	8132044	8132039
55	81160	8163481	8163411	8163370	8163421
56	82636	8119433	8119391	8119401	8119408
63	92968	8312128	8312027	8312113	8312089
67	98872	8287419	8287373	8287442	8287411
80	118060	8356362	8356446	8356417	8356408
88	129868	8346594	8346593	8346572	8346586

Abbildung 5.1.: Messdaten des Durchsatzes

5.2.1. Durchsatz

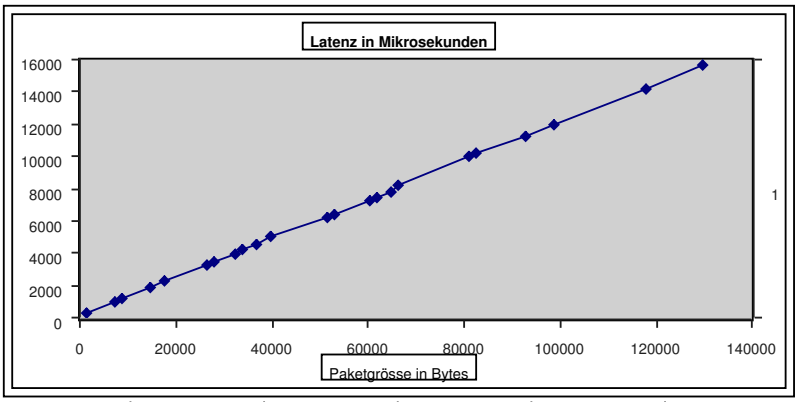
Abbildung 5.1 zeigt die Messwerte des mittleren Durchsatzes in Zahl und Bild. Es ist wenig überraschend, dass der Durchsatz grundsätzlich mit der Anzahl der Frames ansteigt, welche nacheinander in die gleiche Richtung gesendet werden. Bereits bei nur zwei Frames überwiegt dieser Vorteil gegenüber einem einzigen Frame den Overhead des Fragmentierens und Reassemblierens deutlich.

Es sticht jedoch sofort ins Auge, dass der Durchsatz immer wieder zurück fällt[1]. Vermut-

[1] Die in den Abbildungen gezeigten Messpunkte wurden ausgewählt, um die lokalen Extremalwerte hervorzuheben.

lich hängt das teilweise mit der begrenzten Zahl der D6 RAM Buffers zusammen. Der Verlauf ist jedoch so unregelmässig, dass wahrscheinlich auch andere Faktoren eine Rolle spielen. Messfluktuationen können jedoch als Ursache dieses Effekts ausgeschlossen werden, da die Messwerte innerhalb einer Paketgrösse sehr stabil sind, wie der Abbildung entnommen werden kann.

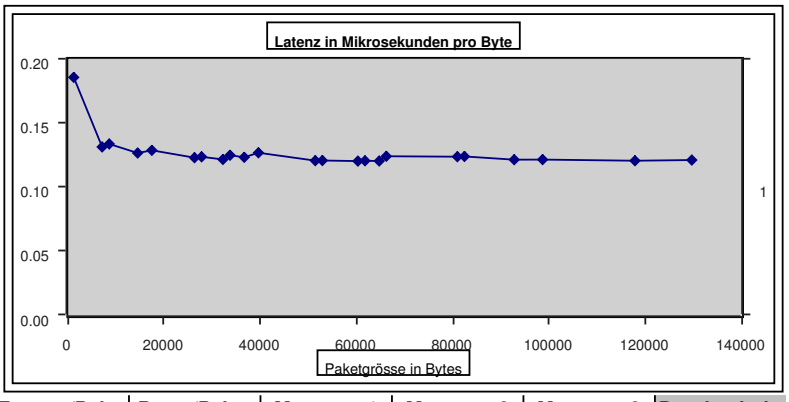
5.2.2. Latenz



Frames/Paket	Bytes/Paket	Messung 1	Messung 2	Messung 3	Durchschnitt
1	1460	269	269	269	269
5	7360	957	957	957	957
6	8836	1170	1170	1168	1169
10	14740	1847	1847	1846	1847
12	17692	2255	2253	2253	2254
18	26548	3230	3229	3230	3230
19	28024	3427	3429	3428	3428
22	32452	3903	3904	3903	3903
23	33928	4192	4193	4192	4192
25	36880	4498	4498	4498	4498
27	39832	5000	5000	4995	4998
35	51640	6163	6163	6163	6163
36	53116	6346	6345	6343	6345
41	60496	7201	7198	7198	7199
42	61972	7391	7389	7389	7390
44	64924	7730	7731	7730	7730
45	66400	8150	8152	8152	8151
55	81160	9931	9937	9938	9935
56	82636	10131	10134	10132	10132
63	92968	11165	11166	11164	11165
67	98872	11879	11881	11883	11881
80	118060	14082	14083	14081	14082
88	129868	15561	15560	15559	15560

Abbildung 5.2.: Messdaten der absoluten Latenz

Abbildung 5.2 zeigt die gemessenen mittleren Latenzwerte. Sie sind alles andere als spektakulär, die Latenz steigt wie erwartet beinahe perfekt linear mit der Paketgrösse an.



Frames/Paket	Bytes/Paket	Messung 1	Messung 2	Messung 3	Durchschnitt
1	1460	0.18425	0.18425	0.18425	0.18425
5	7360	0.13003	0.13003	0.13003	0.13003
6	8836	0.13241	0.13241	0.13219	0.13234
10	14740	0.12531	0.12531	0.12524	0.12528
12	17692	0.12746	0.12735	0.12735	0.12738
18	26548	0.12167	0.12163	0.12167	0.12165
19	28024	0.12229	0.12236	0.12232	0.12232
22	32452	0.12027	0.12030	0.12027	0.12028
23	33928	0.12356	0.12359	0.12356	0.12357
25	36880	0.12196	0.12196	0.12196	0.12196
27	39832	0.12553	0.12553	0.12540	0.12549
35	51640	0.11935	0.11935	0.11935	0.11935
36	53116	0.11947	0.11946	0.11942	0.11945
41	60496	0.11903	0.11898	0.11898	0.11900
42	61972	0.11926	0.11923	0.11923	0.11924
44	64924	0.11906	0.11908	0.11906	0.11907
45	66400	0.12274	0.12277	0.12277	0.12276
55	81160	0.12236	0.12244	0.12245	0.12242
56	82636	0.12260	0.12263	0.12261	0.12261
63	92968	0.12010	0.12011	0.12008	0.12010
67	98872	0.12015	0.12017	0.12019	0.12017
80	118060	0.11928	0.11929	0.11927	0.11928
88	129868	0.11982	0.11981	0.11981	0.11981

Abbildung 5.3.: Messdaten der relativen Latenz

Abbildung 5.3 ist da schon deutlich interessanter. Sie zeigt die gemessenen mittleren Latenzwerte im Verhältnis zur Paketgrösse, d.h. die durchschnittliche Latenz pro gesendetem Byte. Es sticht hier sofort ins Auge, dass der Overhead bei einem einzigen Frame deutlich am höchsten ist und mit steigender Anzahl Frames schnell gegen den Minimalwert zustrebt.

Aufwand der Fragmentation

Kurz vor Abschluss dieser Arbeit wurde noch der Aufwand des Fragmentierens und Reassemblierens im Proxy Device Driver untersucht. Die Zeit reichte leider nicht mehr für eine vollständige Messung und Analyse, aber es hat sich gezeigt, dass bei zwei Fragmenten

pro Paket ca. die Hälfte der mittleren Latenzzeit mit Fragmentieren und Reassemblieren verbracht wird, bei 88 Fragmenten pro Paket sogar ca. zwei Drittel. Es scheint naheliegend, dass dies ähnlich wie beim Verhältnis der Latenz zur Paketgrösse einen Grenzwert darstellt, der relativ schnell angenähert wird. Das Fragmentieren und Reassemblieren wurde relativ trivial durch Kopieren der Speicherbereiche zwischen Socket Buffers implementiert, es liesse sich mit geeigneten Optimierungen also eventuell eine signifikante Besserung erzielen. Es ist allerdings fraglich, ob die Latenz (und damit möglicherweise) damit oder bei gänzlichem Wegfall der Fragmentation tatsächlich so dramatisch verbessert würde. Möglicherweise würden dann lediglich andere Faktoren dominieren, wie zum Beispiel das Warten auf Pakete oder Ressourcen wie Buffers.

5.2.3. Fazit

Insbesondere die gemessenen Durchsatzwerte fallen auf den ersten Blick sehr enttäuschend aus, wenn man bedenkt, dass der MAC Chip Gigabit Ethernet beherrscht, die Durchsatzwerte aber nur knapp ausreichen, um einen Fast Ethernet Link auszulasten. Die gemessenen Werte liegen jedoch durchaus in dem Rahmen, der auch schon von Pascal Erni gemessen wurde[2]. Er hat in [1, S. 69f] dieses Phänomen ausführlich untersucht, hier nur eine kurze Rekapitulation: Die Ursache ist bei der Kommunikation zwischen den PowerNP Picocode Einheiten und der embedded PowerPC CPU[3] zu suchen, wahrscheinlich in erster Linie bei der Interruptlatenz. Durch den gezielten Einsatz von Polling liesse sich hier eventuell eine deutliche Steigerung erzielen, Pascal Ernis Messungen haben jedoch gezeigt, dass die mit lediglich 133 Mhz getaktete embedded PowerPC CPU für die Auslastung eines Gigabit Ethernet Links schlicht und einfach zu langsam zu sein scheint.

Es deutet jedoch nichts darauf hin, dass die Kommunikationsinfrastruktur an sich die Performance irgendwie beeinträchtigt oder gar begrenzt. Es wird interessant sein zu sehen, ob sich dieser Eindruck mit künftigen Hardwareanbindungen für die Infrastruktur bestätigen wird.

Es ist erwähnenswert, dass die Kommunikationsinfrastruktur während der Messungen sehr zuverlässig und stabil ihren Dienst verrichtete. Es traten keine Übertragungsfehler auf, und die gemessenen Werte sind wie bereits erwähnt ausserordentlich stabil. Die zukünftigen Hardwareanbindungen werden hier selbstverständlich auch eine Rolle spielen, aber der hardwareunabhängige Infrastrukturteil scheint jedenfalls grundsätzlich solide zu laufen.

[2] Es gilt auch zu bedenken, dass bei diesen Messungen immer nur in eine Richtung gesendet wurde. Die gemessenen Werte sollten jedoch prinzipiell in beide Richtungen gleichzeitig erzielbar sein.

[3] Siehe auch Abschnitt 4.1.1 auf Seite 29 für eine etwas ausführlichere Beschreibung der Funktionsweise der Kommunikation zwischen Host CPU und embedded PowerPC CPU.

6. Abschluss

Dieses Kapitel bildet den Abschluss dieser Dokumentation. Es folgt zuerst eine Zusammenfassung der bisherigen Kapitel, gefolgt von einer Bestandesaufnahme der erreichten Ziele und einem Ausblick auf potentielle weiterführende Arbeiten.

6.1. Zusammenfassung

Im Kapitel 1 ab Seite 9 wurde zunächst die Motivation für diese Arbeit dargelegt, welche darin besteht, dass man verschiedenste Hardwarekomponenten, insbesondere Netzwerkprozessoren (NPs), in einem einzigen PromethOS basierten hierarchischen Active Network Node (ANN) zusammenfassen möchte. Daraus ergab sich die Aufgabenstellung, welche die Entwicklung eines fehlenden Bindeglieds umfasste, nämlich einer Infrastruktur für die Kommunikation zwischen PromethOS basierten Hardwarekomponenten. Als Herausforderungen dieser Aufgabe wurden die geforderte Generizität und Hardwareunabhängigkeit sowie die gewünschte Möglichkeit der Ressourcenkontrolle herausgestrichen.

Das Kapitel 2 ab Seite 13 wurde eröffnet von Zusammenfassungen der Diplomarbeit von Pascal Erni [1], in welcher die Grundlagen für den Einsatz von PromethOS auf NPs gelegt wurden, und eines Papers von Lukas Ruf, R. Keller und Professor Dr. Bernhard Plattner [2], das die Gesamtarchitektur eines PromethOS basierten hierarchischen ANN beschreibt. Das Kapitel wurde mit der Beschreibung der für diese Arbeit eingesetzten Entwicklungsumgebung abgeschlossen, welche neben dem IBM Advanced Software Offering (ASO) vor allem aus diversen Tools aus dem reichen Fundus freier Software bestand.

Im Kapitel 3 ab Seite 19 wurde dokumentiert, wie die Architektur der Kommunikationsinfrastruktur erarbeitet wurde. Es wurde zunächst aus den identifizierten Hauptanforderungen ein Grobkonzept entwickelt, welches einen generischen, paketbasierten Kanal als zentrales Element enthielt. Dieses Grobkonzept wurde dann verfeinert. Der Aufbau des Paketheaders wurde detailliert beschrieben. Der Initialisierungsprozess für die Einbindung von Knoten in die Infrastruktur und die Zuweisung eindeutiger Kennungen wurde Schritt für Schritt erklärt. Zuletzt wurden die Schnittstellen der Infrastruktur zu den Hardwarekomponenten, zum Plugin Manager und zum Node Manager bis auf die Ebene von Parametern und vorgesehenem Verhalten der involvierten Funktionen erläutert.

Im Kapitel 4 ab Seite 29 wurde die Implementation der erarbeiteten Architektur beleuchtet. Zunächst wurde beschrieben, wie die Hardwareeigenschaften des eingesetzten

IBM PowerNP Application Reference Board (ARB) gemäss Architektur in einen Hardware Abstraction Layer (HAL) abgebildet wurden. Die Infrastruktur sollte mit beliebigen Hardwarekomponenten Pakete senden und empfangen können, welche diesen HAL anbieten. Dann wurde die Funktionsweise des hardwareunabhängigen Infrastrukturkerns, dem Kernelmodul `promethos_comm`, bis ins letzte Detail erläutert, indem seine wichtigsten Funktionen inklusive ihrer Parameter und ihres Verhaltens in den wichtigsten Fällen beschrieben wurden. Zuerst einige allgemeine, dann diejenigen Funktionen, welche die von der Architektur beschriebenen Schnittstellen umsetzen.

Im Kapitel 5 ab Seite 41 wurde dargelegt, wie mit Hilfe des eigens zu diesem Zweck entwickelten Kernelmoduls `promethos_comm_test` die Funktion der Kommunikationsinfrastruktur verifiziert und ihre Performance gemessen wurden. Es wurden eine Reihe von Messwerten des mittleren Durchsatzes und der mittleren Latenz zwischen einer Host CPU und einem PowerNP ARB dargestellt und diskutiert. Die Messwerte fielen alles andere als berauschend aus, was jedoch durch die besonderen Eigenschaften des PowerNP ARB erklärt werden konnte und somit kein schlechtes Licht auf die Infrastruktur an und für sich werfen sollte.

6.2. Erreichte Ziele

Die wichtigsten zu Beginn gesteckten Ziele wurden im Rahmen dieser Arbeit erreicht:

- Es wurde die *Architektur einer Infrastruktur* entwickelt, welche die Kommunikation zwischen beliebigen PromethOS basierten Ausführungseinheiten in einem Active Network Node (ANN) ermöglicht. Die Architektur erfüllt die geforderten Aspekte:
  - *Unterteilung in einen hardwareunabhängigen Infrastrukturkern und einen Hardware Abstraction Layer (HAL)*, um Hardwareeigenschaften vom Kern abzuschirmen.
  - Möglichkeit der Übertragung von *Nutz- wie auch Kontrolldaten*, indem *generische, paketbasierte Kanäle* zwischen den Infrastrukturknoten verwendet werden.
- *Implementation des Infrastrukturkerns* als Kernelmodul `promethos_comm`, welches den *Knoten eindeutige Kennungen* zuweist und *transparentes Routing* von Paketen zwischen diesen bietet.
- *Implementation des HAL für das IBM PowerNP Application Reference Board (ARB)* durch Erweiterung des IBM Advanced Software Offering (ASO) Proxy Device Driver.

Leider hat die Zeit nicht gereicht, um ein weiteres Wunschziel zu erreichen:

- *Resource Control* der Kommunikationskanäle ist noch nicht implementiert.

6.3. Ausblick

Dieser Abschnitt befasst sich mit Ansatzpunkten, welche diese Arbeit für zukünftige Arbeiten bietet. Zuerst wird als wichtigster Punkt die fehlende Resource Control behandelt, danach einige weitere Punkte.

6.3.1. Ressourcenkontrolle

Der Source Code der Kommunikationsinfrastruktur wurde mit Kommentaren versehen, welche Änderungen für die Implementation von Resource Control vorgenommen werden müssen. Die entsprechenden Kommentare haben die Form "Resource Control: ...", zum Beispiel

```
/*
 * List entry representing a link
 *
 * Resource Control: add link properties like bandwidth, latency, ...
 */
typedef struct promethos_comm_link {
        struct promethos_comm_link      *next;
        promethos_comm_cpuid_t          peers[2];
} promethos_comm_link_t;
```

Für die Kontrolle des Ressourcenverbrauchs wird die Kommunikationsinfrastruktur voraussichtlich unter anderem folgende Funktionen zur Verfügung stellen müssen:

```
int promethos_comm_setquota( promethos_comm_cpuid_t node0,
promethos_comm_cpuid_t node1, int maxbw )
```

Mit dieser Funktion kann der Plugin Manager den Ressourcenverbrauch (Bandbreite) eines Flows auf einer Verbindung zwischen den Knoten mit den Kennungen `node0` und `node1` auf den Maximalwert `maxbw` beschränken. Die Infrastruktur misst den Ressourcenverbrauch eines Flows fortlaufend und lässt bei Überschreiten der mit dieser Funktion festgelegten Grenze Pakete fallen oder hält sie zurück. Diese Funktion gibt im Erfolgsfall Null zurück, andernfalls einen negativen Standardfehlerwert.

```
int promethos_comm_getstats( promethos_comm_cpuid_t node0,
promethos_comm_cpuid_t node1, promethos_comm_stats_t *stats )
```

Der Plugin Manager kann mit dieser Funktion Statusinformationen über eine Verbindung zwischen zwei Knoten mit den Kennungen `node0` und `node1` erhalten, z.B. die aktuell gültigen Bandbreitenbeschränkungen sowie die Auslastung der Verbindung. `stats` ist ein Pointer auf den vom Plugin Manager reservierten Speicherbereich, der mit der Datenstruktur `promethos_comm_stats_t` gefüllt werden soll. Diese Funktion gibt im Erfolgsfall Null zurück, andernfalls einen negativen Standardfehlerwert.

Ausserdem werden folgende bestehende Funktionen erweitert werden:

`promethos_comm_alloc_skb()`

Diese Funktion wird einen neuen Parameter `promethos_comm_metric_t metric` entgegennehmen, welcher die Metrik (Anzahl Hops, Bandbreite, Latenz, usw.) vorgibt, welche für den Versand zum Zielknoten optimiert werden soll. Die Infrastruktur kann für den Versand je nach gewählter Metrik unterschiedliche Routing-Entscheidungen treffen.

Ein weiterer Parameter würde den Flow identifizieren, zu dem die zu sendenden Daten gehören.

`promethos_comm_receive()`

Diese Funktion muss für Datenpakete, welche an einen anderen Knoten gerichtet sind und dementsprechend weitergeleitet werden müssen, die bestehenden Ressourcenbeschränkungen für den Link zum nächsten Hop überprüfen und bei einer Überschreitung geeignete Massnahmen ergreifen, zum Beispiel das Paket droppen, in eine Warteschlange einreihen und zu einem späteren Zeitpunkt weiterverarbeiten, usw.

`promethos_comm_send()`

Diese Funktion könnte beim Überschreiten einer Beschränkung des Ressourcenverbrauchs einen negativen Standardfehlerwert zurückgeben, z.B. `-EAGAIN`, womit sie dem Plugin Manager signalisieren würde, dass er es später nochmals versuchen solle. Der Plugin Manager könnte dann entweder gleich warten oder das Paket in eine Warteschlange reihen und es mit einem anderen anstehenden Paket versuchen. Ein Problem stellt dabei dar, dass die Überschreitung der Beschränkung nicht schon auf dem Weg vom sendenden Knoten zu seinem Peer auftreten muss oder erkannt werden kann. Es müsste genauer untersucht werden, ob dieser Ansatz sinnvoll ist, oder ob z.B. die Infrastruktur Überschreitungen intern durch Droppen oder Queuen begegnen und die Sicherstellung der gewünschten Zuverlässigkeit dem Plugin Manager überlassen soll.

6.3.2. Andere Aspekte

Hier werden noch kurz einige weitere Aspekte angesprochen, wie die Kommunikationsinfrastruktur verbessert werden könnte.

Unterbruch von Verbindungen zwischen Knoten

Es könnte nützlich sein, für den Abbau von Verbindungen zwischen zwei Knoten genau wie für deren Aufbau ein Protokoll zu entwickeln, um das Wissen über den Unterbruch an alle Knoten zu verteilen. Einige Grenzfälle machen dies allerdings zu einem ziemlich komplexen Problem, und da sowieso nicht verhindert werden kann, dass Verbindungen hinter dem Rücken der Infrastruktur unterbrochen werden (zum Beispiel, indem ein Knoten rebootet wird), ist der Nutzen nicht über alle Zweifel erhaben, insbesondere nicht im Vergleich zum Aufwand.

Für den Rebbot eines Knotens oder allgemein den Fall, dass ein Knoten die Verbingung zum anderen verliert und dann wieder aufzubauen versucht, könnte eine andere Änderung nützlich sein. Die Funktion `promethos_comm_receive_discover()` zum Empfang eines Discover Pakets könnte erkennen, wenn die Kennung des "neuen" Peers bereits bekannt ist, und ihm ein entsprechendes Zuweisungspaket mit der bekannten Kennung und den Informationen über alle bekannten Verbindungen der Infrastruktur senden.

Hilfsfunktionen zum Senden der verschiedenen Pakettypen

Es könnte für jeden Pakettyp eine Hilfsfunktion hinzugefügt werden, welche die HAL Datenstruktur und die Felder des Paketheaders als Parameter entgegennimmt und das Allozieren des Socket Buffers, das Initialisieren des Headers und das Senden des Pakets übernimmt. Das sollte den Komfort beim Entwickeln am Infrastrukturcode sowie dessen allgemeine Lesbarkeit erhöhen und die Codeduplikation reduzieren.

A. Aufgabenstellung

Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Sommersemester 2004

Diplomarbeit

für

Michel Dänzer

Betreuer: Lukas Ruf

Ausgabe: 15.03.2004
Abgabe: 14.07.2004

Resource Controlled Interprocessor Communication On NP-based Active Network Nodes

1 Einführung

Zukünftige Router werden dynamisch programmierbar sein. Code wird zur Laufzeit ersetzt werden, um bei Updates oder Funktionalitätserweiterungen nicht den Router insgesamt abschalten zu müssen. Active Networking [10] entwickelte das Konzept von Active Network Nodes, welche die dynamische Funktionalitätserweiterung um Extended (Network) Services vorsehen (Netzwerkdienste, die zusätzlich zum Basis IP-Forwarding ausgeführt werden).
Die Ausführung von Extended Services benötigt zusätzliche Rechenleistung (Processing Capacity), die in modernen Routern pro Netzwerk-Port durch Netzwerkprozessoren (NP) zur Verfügung gestellt wird. Gewöhnlicherweise bestehen solche NPs aus einem Control Prozessor, spezialisierten Paket- und Co-Prozessoren, die zusammen auf einem einzigen Chip implementiert sind. Einer ihrer Vertreter ist der IBM PowerNP 4GS3 [5], welcher mit einem PowerPC e405 Control Prozessor und 16 der zuvor erwähnten Paketprozessoren aufgebaut ist (acht Dyadic Protocol Processor Units (DPPUs), welche je aus zwei Core Language Processors (CLPs) bestehen).
Am Institut für Technische Informatik und Kommunikationsnetze (TIK) der ETH Zürich (ETHZ) wurde im Rahmen der Semesterarbeit von Guindehi [6] CoBRA alias PromethOS [6] entwickelt, welches eine dynamisch erweiterbare Plattform im Linux-Kernel 2.4 zur Verfügung stellt. PromethOS wurde im Rahmen der Arbeit zu PromethOS NP [2, 8] erweitert, damit die Umgebung von Netzwerkprozessoren profitieren kann. Als Proof-of-Concept Implementation wurde PromethOS NP auf einem einfachen Node bestehend aus einem PowerNP4-Application Reference Board (ARB) [9] und einem Host Prozessor entwickelt, welche durch einen PCI-Bus [7] verbunden sind. Die Kommunikation zwischen den beiden General Purpose Prozessoren (GPPs, PowerPC e405 und Intel ia32) wurde nur rudimentär implementiert. Moderne Router bestehen jedoch aus mehreren Netzwerk-Ports mit NP-Support. Um den anvisierten Knotendurchsatz zu erreichen, ist zudem eine optimierte Inter-Prozessor-Kommunikationsinfrastruktur nötig, für welche Resource Limiten spezifiziert werden sollen.

2 Aufgaben: Resource Controlled Interprocessor Communication On NP-based Active Network Nodes

Im Rahmen dieser Diplomarbeit soll eine Inter-Prozessor Kommunikationsinfrastruktur für Daten- und Control-Verkehr zwischen den verschiedenen GPPs entwickelt werden, welche die dem Problem zugrundeliegende Komplexität genügend abstrahiert und dennoch die benötigte Effizienz erreicht. Die Control-Kommunikation soll für das Setup von Komponenten und für deren Kontrolle ausgelegt werden. Ressourcen auf dynamisch erweiterbaren Knoten sind rar. Das in Ihrer Semesterarbeit [1] entwickelte Konzept des Profiling und Kalibrierens von Komponenten soll erweitert werden, um ganze Ketten von Komponenten zu unterstützen. Zudem soll die Implementation in der Lage sein, die von Ihnen zu entwickelnde Inter-Prozessor Kommunikation zu kontrollieren.
Das TIK verfügt über eine NP-Entwicklungsumgebung für Linux, die auf einem ziemlich alten Linux-Kernel basiert. Der neue 2.6-Kernel stellt Mechanismen und Eigenschaften zur Verfügung, welche vorteilhaft ausgenützt werden können. Wenn der Aufwand für eine Portierung vertretbar ist, soll die Arbeit mit einem neuen 2.6-Kernel erfolgen.

3 Vorgehen

- Machen Sie sich vertraut mit den Unterlagen (Dokumentation und Source Code) zu PromethOS NP und dem Netzwerkprozessor IBM PowerNP 4GS3.
- Klären Sie innerhalb der ersten Woche ab, welcher Aufwand nötig wäre, die vorhandene Infrastruktur auf den neuesten Linux-Kernel zu portieren.
- Erstellen Sie einen Zeitplan, in welchem Sie die von Ihnen zu erreichenden Meilensteine Ihrer Arbeit identifizieren.
- Richten Sie sich eine Entwicklungsumgebung (GNU Tools) unter Linux ein.
- Falls der Aufwand für eine Portierung des Advanced Software Offering Toolkits von IBM [4], resp. der MontaVista Umgebung vertretbar ist, führen sie die Portierung durch. Diese Entscheidung wird zusammen mit dem Betreuer gefällt.
- Entwickeln Sie eine Architektur für die zu erstellende Kommunikationsinfrastruktur und die Resource Control Mechanismen.
- Implementieren Sie Ihre Architektur.
- Verifizieren, evaluieren und demonstrieren Sie das Erreichte.
- Dokumentieren Sie die Resultate ausführlich.

Auf eine klare und ausführliche Dokumentation wird besonders Wert gelegt. Es wird empfohlen, diese laufend nachzuführen und insbesondere die entwickelten Konzepte und untersuchten Varianten vor dem definitiven Variantenentscheid ausführlich schriftlich festzuhalten.

4 Organisatorische Hinweise

- Am Ende der zweiten Woche ist ein Zeitplan für den Ablauf der Arbeit vorzulegen und mit dem Betreuer abzustimmen.
- Mit dem Betreuer sind regelmässige, zumindest wöchentliche Sitzungen zu vereinbaren. In diesen Sitzungen sollen die Studenten mündlich über den Fortgang der Arbeit und die Einhaltung des Zeitplanes berichten und anstehende Probleme diskutieren.
- Am Ende des ersten Monates muss eine Vorabversion des Inhaltsverzeichnis zur Dokumentation dem Betreuer abgegeben und mit diesem besprochen werden.
- Nach der Hälfte der Arbeitsdauer soll ein kurzer mündlicher Zwischenbericht abgegeben werden, der über den Stand der Arbeit Auskunft gibt. Dieser Zwischenbericht besteht aus einer viertelstündigen, mündlichen Darlegung der bisherigen Schritte und des weiteren Vorgehens gegenüber Professor Plattner.

- Am Schluss der Arbeit muss eine Präsentation von **20 Minuten** im Fachgruppen- oder Institutsrahmen gegeben werden. Anschliessend an die Schlusspräsentation soll die Arbeit Interessierten praktisch vorgeführt werden.
- Die Arbeit muss regelmässig auf dem CVS-Server cvs.promethos.org gesichert werden. Es ist darauf zu achten, dass die **richtige CVS-Branch** verwendet wird.
- Ein einheitlicher Code Style muss eingehalten werden.
- Bereits vorhandene Software kann übernommen und gegebenenfalls angepasst werden.
- Die Dokumentation ist mit dem Satzsystem LaTeX zu erstellen.
- Es ist ein mit Bindespiralen gebundener Schlussbericht (am TIK vorhanden) über die geleisteten Arbeit abzuliefern (4 Exemplare). Dieser Bericht besteht aus einer Zusammenfassung , einer Einleitung, einer Analyse von verwandten und verwendeten Arbeiten, sowie einer vollständigen Beschreibung der Konfiguration von den eingesetzten Programmen. Der Bericht ist in Deutsch oder Englisch zu halten. Die Zusammenfassung muss in Deutsch und Englisch verfasst werden.
- Die Arbeit muss auf CDROM archiviert abgegeben werden. Stellen Sie sicher, dass alle Programme sowie die Dokumentation sowohl in der lauffähigen, resp. druckbaren Version als auch im Quellformat vorhanden, lesbar und verwendbar sind.
  Mit Hilfe der abgegebenen Dokumentation muss der entwickelte Code zu einem ausführbaren Programm erneut übersetzt und eingesetzt werden können.
- Diese Arbeit steht unter der GNU General Public License (GNU GPL) [3].
- **Für die Arbeit werden Informationen eingesetzt, welche nur durch das Unterzeichnen eines NDA (Non-Disclosure Agreement) mit IBM Corp. erhalten wurden. Die Arbeit als ganzes (Programmcode und Dokumentation) sowie alle Informationen, die unter das NDA fallen, dürfen nur an Dritte weitergegeben werden, wenn eine schriftliche Einwilligung von IBM Corp. vorliegt.**
- Diese Arbeit wird als Diplomarbeit an der ETH Zürich durchgeführt. Es gelten die Bestimmungen hinsichtlich Kopier- und Verwertungsrechte der ETH Zürich.

Literatur

[1] M. Dänzer and J. Gyger. iptables Rule Scheduling. Technical Report SA-2002-43, TIK ETH Zürich, Jul. 2002.

[2] P. Erni. *Einsatz und Programmierung des IBM NP4GS3 Netzwerkprozessors für Aktive Netzwerkknoten unter Linux*. TIK, ETH Zurich, 2003.

[3] GNU General Public License v2. http://www.gnu.org/copyleft/gpl.html, June 1991.

[4] IBM Corp. IBM PowerNP, Advanced Software Offering, User's Guide. http://www.ibm.com, 2002.

[5] IBM Corp. IBM PowerNP NP4GS3 Databook. http://www.ibm.com, 2002.

[6] R. Keller, L. Ruf, A. Guindehi, and B. Plattner. PromethOS: A Dynamically Extensible Router Architecture Supporting Explicit Routing. In *Proc. of the 4th Annual Int. Working Conf. on Active Networks IWAN*, number 2546 in Lect. Notes in Comp. Science. Springer Verlag, Dec. 2002.

[7] Oregon PCI Special Interest Group. *PCI Local Bus Spec., Rev. 2.2*, Dec. 1998.

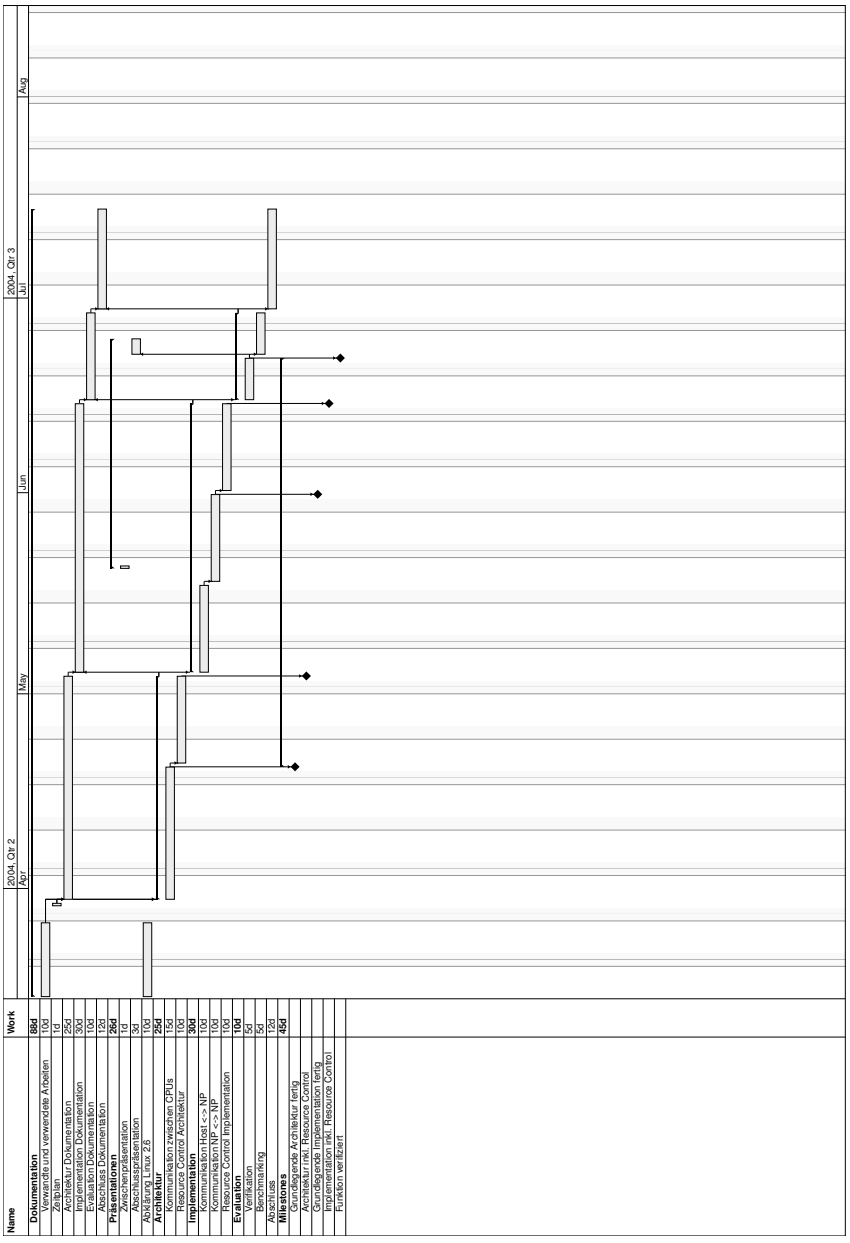
[8] L. Ruf, R. Pletka, P. Erni, P. Droz, and B. Plattner. Towards High-performance Active Networking. In *Proc. of the 5th Annual Int. Working Conf. on Active Networks IWAN*, number 2982 in Lecture Notes in Computer Science, Kyoto, Japan, December 2003. Springer Verlag, Heidelberg.

[9] Silicon Software System. Application Reference Board for the IBM PowerNP NP4GS3 Network Processor User Manual. http://www.s3group.com/network_processing/copernicus, 2002.

[10] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden. A Survey of Active Network Research. *IEEE Communications*, January, 1997.

Zürich, den 15.03.2004

B. Zeitplan


2004. Qtr 2
Apr
May
Jun
2004. Qtr 3
Jul
Aug
Name
Work
Dokumentation 88d
Verwandte und verwendete Arbeiten 10d
Zeitplan 1d
Architektur Dokumentation 25d
Implementation Dokumentation 30d
Evaluation Dokumentation 10d
Abschluss Dokumentation 12d
Präsentationen 26d
Architekturpräsentation 1d
Abschlusspräsentation 3d
Ablösung Linux 2.6 10d
Architektur 25d
Kommunikation zwischen CPUs 15d
Resource Control Architektur 10d
Implementation 30d
Kommunikation Host <-> NP 10d
Kommunikation NP <-> NP 10d
Resource Control Implementation 10d
Evaluation 15d
Verifikation 5d
Benchmarking 5d
Abschluss 12d
Milestones 45d
Grundlegende Architektur fertig
Kommunikation implementiert
Grundlegende Implementation fertig
Implementation inkl. Resource Control
Funktion verifiziert

C. Source Code Referenzen

Nachfolgend der Inhalt der Datei `README.structure`:

```
This file gives an overview of the source code structure and describes added
files / changes to existing files.

 Makefile

Toplevel Makefile, see README.make

 aso-134/src/Linux-debian.mk

New ASO build target for Debian systems.

 aso-134/src/linux/em405/sonic/utils/d6load.c

Added optional target IP address parameter.

 aso-134/src/pts/linux/ethernet/rethdd_module.c
 aso-134/src/pts/linux/ethernet/rethdd_module.h
 aso-134/src/pts/linux/ethernet/rethdd_proc.c
 aso-134/src/pts/linux/ethernet/rethdd_proc.h
 aso-134/src/pts/linux/ethernet/rethdd_structs.h

Implemented Hardware Abstraction Layer for communication infrastructure.

 aso-134/src/

Miscellaneous build fixes.

 bin/depmod.pl

Took utility for module interdependencies from busybox.

 config/busybox-1.00-pre10.config
 config/ksymoops-2.4.9

Build configurations for external tools.
```

```
 copernicus_drv/

Copernicus driver for PowerNP ARB.

 hardhat/

HardHat PPC cross build environment.

 linux-2.4.17_mvl21/arch/i386/

Added arch/i386 to MontaVista kernel tree and made other minor fixes to make it
build and work on i386.

 linux-2.4.17_mvl21/config-2.4.17-mvl21-promethos
 linux-2.4.17_mvl21/config-2_4_17_mvl21-promethos

Configuration files for host and ePPC.

 linux-2.4.17_mvl21/include/linux/netdevice.h

Added promethos_comm_hal field to struct net_device.

 linux-2.4.17_mvl21/include/linux/netfilter_ipv4/promethos.h

Moved PromethOS output macros to following file:

 linux-2.4.17_mvl21/include/net/promethos.h
 linux-2.4.17_mvl21/net/Config.in
 linux-2.4.17_mvl21/net/Makefile
 linux-2.4.17_mvl21/net/core/Makefile
 linux-2.4.17_mvl21/net/core/promethos_comm.c

Implemented communication infrastructure core.

 linux-2.4.17_mvl21/net/ipv4/netfilter/Config.in
 linux-2.4.17_mvl21/net/ipv4/netfilter/Makefile
 linux-2.4.17_mvl21/net/ipv4/netfilter/promethos_comm_test.c

Implemented test module.

 report/

Written report.
```

```
 schedules/

Schedules developed in the course of the thesis.
```

D. Scripts

Zuerst der Inhalt der Datei README.make:

```
The Makefile offers the following targets, among others:

boot-eppc: Do everything necessary to boot Linux on the embedded PowerPC
CPUs. The NUM_CARDS variable can be overridden to change the
number of cards to boot, default 1.

This requires gcc-2.95.

report: Build all the documentation in the report/ directory to PDF and
PS formats.

This requires a LaTeX installation and some related tools like
ps2eps.

all: All of the above.

clean: Clean up.
```

Und das eigentliche Toplevel Makefile:

```
NUM_CARDS=1

COPERNICUS_DRV:=copernicus_drv/copernicus.o
COPERNICUS_INTF=eth1 eth2
COPERNICUS_IP=4.4.4.4 4.4.4.5

ASO_HOST_ROOT=aso-134-host
ASO_HOST=$(ASO_HOST_ROOT)/exe/i386-debian-linux

ASO_TARGET_ROOT=aso-134
ASO_TARGET=$(ASO_TARGET_ROOT)/exe/ppc405-hardhat-linux

INITRD_MNTPOINT=/mnt

TARGET_LINUX=linux-2.4.17_mvl21
```

```
TARGET_LINUX_VERSION=2.4.17_mvl21-np4gs3-promethos
TARGET_PATH:=$(PWD)/hardhat/host/bin:$(PWD)/hardhat/devkit/
   ppc/405/bin

KERNEL_MODULES:=$(shell find $(INITRD_MNTPOINT)/lib/modules/
   $(TARGET_LINUX_VERSION)/kernel -name \*.o 2>/dev/null)
KERNEL_MODULES_SRC:=$(shell cond=''; for module in $(
   KERNEL_MODULES); do cond="$$cond -name `basename $$module
   .o`.c"; done; find $(TARGET_LINUX) $$cond -false)

ASO_MODULES:=$(shell find $(INITRD_MNTPOINT)/lib/modules/$(
   TARGET_LINUX_VERSION)/$(ASO_TARGET_ROOT) -name \*.o 2>/
   dev/null)
RETHDD:=$(ASO_HOST)/rethdd.o
RETHDD_INTF=npctl0 npctl1
RETHDD_IP=3.3.3.3 3.3.4.3

D6LOAD_IP=3.3.3.4 3.3.4.4

SYMLINKS:=hardhat/devkit/ppc/405/target/usr/lib/libpthread.
   so hardhat/devkit/ppc/405/lib/gcc-lib/powerpc-hardhat-
   linux/2.95.3/libstdc++.a


default:
        cat README.make

all:    boot-eppc report


boot-eppc:      $(COPERNICUS_DRV) $(RETHDD) $(TARGET_LINUX)/
   arch/ppc/boot/images/zvmlinux.initrd.vxboot
        sudo insmod $(COPERNICUS_DRV) || true
        ifs=($(COPERNICUS_INTF)) ips=($(COPERNICUS_IP)); for
            i in `seq $(NUM_CARDS)`; do sudo ifconfig $${ifs
           [$$i-1]} $${ips[$$i-1]} promisc || true; done
        #
        copernicus_ifs=($(COPERNICUS_INTF)); for i in `seq $
           (NUM_CARDS)`; do
                                          \
             if [ "$$i" = "1" ]; then ext_ifs=$${
                copernicus_ifs[$$i-1]}; else ext_ifs="
                $$ext_ifs $${copernicus_ifs[$$i-1]}"; fi;
                  \
```

```
        done; sudo insmod $(RETHDD) ext_ifs="$$ext_ifs" ||
           true
        ifs=($(RETHDD_INTF)) ips=($(RETHDD_IP)); for i in `
           seq $(NUM_CARDS)`; do sudo ifconfig $${ifs[$$i
           -1]} $${ips[$$i-1]} netmask 255.255.255.0 || true
           ; done
        #
        ips=($(D6LOAD_IP)); for i in `seq $(NUM_CARDS)`; do
           $(ASO_HOST)/linux/em405/sonic/utils/d6load $(
           TARGET_LINUX)/arch/ppc/boot/images/zvmlinux.
           initrd.vxboot $${ips[$$i-1]}; done

$(COPERNICUS_DRV):
        $(MAKE) -C copernicus_drv

$(RETHDD):      $(shell find $(ASO_HOST_ROOT)/src -name \*.c
    -o -name \*.h 2>/dev/null)
        if [ ! -d $(ASO_HOST_ROOT) ]; then cp -a $(
           ASO_TARGET_ROOT) $(ASO_HOST_ROOT); cd $(
           ASO_HOST_ROOT)/src; ./configure debian; fi
        $(MAKE) -C $(ASO_HOST_ROOT)/src

$(TARGET_LINUX)/.config:        $(TARGET_LINUX)/config-2
   _4_17_mvl21-promethos
        cp $^ $@
        PATH=$(TARGET_PATH):$$PATH $(MAKE) -C $(TARGET_LINUX
           ) oldconfig

$(TARGET_LINUX)/vmlinux:
        PATH=$(TARGET_PATH):$$PATH $(MAKE) -C $(TARGET_LINUX
           ) vmlinux

$(TARGET_LINUX)/arch/ppc/boot/images/zvmlinux.initrd.vxboot:
       $(TARGET_LINUX)/.config initrd-stamp
        PATH=$(TARGET_PATH):$$PATH $(MAKE) -C $(TARGET_LINUX
           ) zImage.initrd

initrd-stamp:   modules-stamp $(shell find $(INITRD_MNTPOINT
   ) -type f)
        sudo umount $(INITRD_MNTPOINT)
        gzip -cv $(TARGET_LINUX)/arch/ppc/boot/images/
           ramdisk.image >$(TARGET_LINUX)/arch/ppc/boot/
           images/ramdisk.image.gz
```

```
        $(MAKE) $(INITRD_MNTPOINT)/lib/modules/$(
           TARGET_LINUX_VERSION)
        touch $@

$(TARGET_LINUX)/arch/ppc/boot/images/ramdisk.image:     $(
   TARGET_LINUX)/arch/ppc/boot/images/ramdisk.image.gz
        if [ ! -f $@ ]; then gunzip -c $^ >$@; fi

$(INITRD_MNTPOINT)/lib/modules/$(TARGET_LINUX_VERSION): $(
   TARGET_LINUX)/arch/ppc/boot/images/ramdisk.image
        e2fsck -p $(TARGET_LINUX)/arch/ppc/boot/images/
           ramdisk.image; test $$? -lt 4
        sudo mount -o loop $(TARGET_LINUX)/arch/ppc/boot/
           images/ramdisk.image $(INITRD_MNTPOINT)

$(INITRD_MNTPOINT)/lib/modules/$(TARGET_LINUX_VERSION)/
   modules.dep: modules-stamp $(TARGET_LINUX)/vmlinux $(
   ASO_MODULES) $(INITRD_MNTPOINT)/lib/modules/$(
   TARGET_LINUX_VERSION) bin/depmod.pl
        sudo bin/depmod.pl -b $(INITRD_MNTPOINT)/lib/modules
           /$(TARGET_LINUX_VERSION) -k $(TARGET_LINUX)/
           vmlinux

modules-stamp: $(TARGET_LINUX)/.config $(shell find $(
   TARGET_LINUX) -name \*.c -o -name \*.h \! -name compile.h
   ) $(INITRD_MNTPOINT)/lib/modules/$(TARGET_LINUX_VERSION)
        PATH=$(TARGET_PATH):$$PATH $(MAKE) -C $(TARGET_LINUX
           ) modules
        sudo $(MAKE) -C $(TARGET_LINUX) modules_install
           INSTALL_MOD_PATH=$(INITRD_MNTPOINT)
        touch $@

$(ASO_MODULES): aso-stamp $(INITRD_MNTPOINT)/lib/modules/$(
   TARGET_LINUX_VERSION)
        sudo cp -u $(ASO_TARGET)/*.o $(INITRD_MNTPOINT)/lib/
           modules/$(TARGET_LINUX_VERSION)/$(ASO_TARGET_ROOT
           )/

$(ASO_TARGET_ROOT)/src/Linux-dflt-platform.mk:  $(
   ASO_TARGET_ROOT)/src/Linux-sonicehh.mk
        cd $(ASO_TARGET_ROOT)/src; PATH=$(TARGET_PATH):
           $$PATH ./configure sonicehh
```

```
aso-stamp:      $(ASO_TARGET_ROOT)/src/Linux-dflt-platform.
   mk $(SYMLINKS) $(shell find $(ASO_TARGET_ROOT)/src -name
   \*.c -o -name \*.h)
        PATH=$(TARGET_PATH):$$PATH $(MAKE) -C $(
           ASO_TARGET_ROOT)/src
        touch $@

$(SYMLINKS):
        ln -fs ../../lib/libpthread.so.0 hardhat/devkit/ppc
           /405/target/usr/lib/libpthread.so
        ln -fs ../../../../target/usr/lib/libstdc++-libc6
           .1-2.a.3 hardhat/devkit/ppc/405/lib/gcc-lib/
           powerpc-hardhat-linux/2.95.3/libstdc++.a


.PHONY: clean report

clean:
        $(MAKE) -C $(ASO_TARGET_ROOT)/src clean
        if [ -d $(ASO_HOST_ROOT) ]; then $(MAKE) -C $(
           ASO_HOST_ROOT)/src clean; fi
        $(MAKE) -C $(TARGET_LINUX) clean
        $(MAKE) -C report clean


report:
        $(MAKE) -C report report.{pdf,ps} executive-summary
           .{pdf,ps} kurzfassung.{pdf,ps}
```

Literaturverzeichnis

[1] P. Erni *Einsatz und Programmierung des IBM NP4GS3 Netzwerkprozessors für Aktive Netzwerkknoten unter Linux.*
Diplomarbeit DA-2003.13, TIK, ETH Zürich, 2003.

[2] L. Ruf, R. Keller and B. Plattner *A Scalable High-Performance Router Platform Supporting Dynamic Service Extensibility On Network and Host Processors.*
Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology (ETH) Zürich, 2004.

[3] L. Ruf *The PromethOS Homepage*
http://www.promethos.org/, 2004.

[4] Paul "Rusty" Russell et. al. *Netfilter Project Homepage*
http://www.netfilter.org/, 2004.

[5] MontaVista Software
http://www.mvista.com/, 2004.

[6] GNU Compiler Collection
http://gcc.gnu.org/, 2004.

[7] Subversion, a tool for version control
http://subversion.tigris.org/, 2004.

[8] Concurrent Versions System
http://www.cvshome.org/, 2004.

[9] A. Guindehi *COBRA - Component Based Routing Architecture*
Semester Thesis SA-2001.30, ETH Zürich, 2001.

[10] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden *A Survey of Active Network Research.*
IEEE Communications, January, 1997.

[11] L. Ruf, R. Pletka, P. Erni, P. Droz, and B. Plattner *Towards High-performance Active Networking.*
In Proc. of the 5th Annual Int. Working Conf. on Active Networks IWAN, number 2982 in Lect. Notes in Comp. Science, Springer Verlag, Dec. 2003.

[12] Silicon Software System *Application Reference Board for the IBM PowerNP NP4GS3 Network Processor User Manual.*
http://www.s3group.com/network_processing/copernicus, 2004.

[13] A. von Bidder *Programmable Distributors on Packet Processors*
Diploma Thesis DA-2004-01, TIK, ETH Zürich, 2004.