View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Digitale Bibliothek Thüringen

IPv6-Message-Passing mit Open MPI

DIPLOMARBEIT

zur Erlangung des akademischen Grades
Diplom-Informatiker

FRIEDRICH-SCHILLER-UNIVERSITÄT JENA

Fakultät für Mathematik und Informatik

eingereicht von Adrian Knoth
geb. am 18.08.1981 in Weimar

Betreuer: Dipl-Inf. Christian Kauhaus
Prof. Dr. Dietmar Fey

Jena, 27.03.2007

Kurzfassung

Zur Lösung komplexer wissenschaftlicher Simulationsprobleme kommen heutzutage ob ihres Rechenbedarfs verstärkt Cluster zum Einsatz, die mit Hilfe von Message-Passing-Frameworks programmiert werden.

Stehen mehrere Cluster zur Verfügung, kann durch Cluster-Cluster-Kopplung oftmals eine höhere Gesamtrechenleistung erzielt werden. IPv6 bietet für den Aufbau und den Betrieb dieser Clusterverbünde konzeptionelle Vorteile, die sich für den Anwendungsprogrammierer jedoch nur mit IPv6-fähigen Message-Passing-Frameworks erschließen.

Gegenstand dieser Diplomarbeit ist die Erweiterung des Message-Passing-Frameworks *Open MPI* um IPv6-Unterstützung. Es wird gezeigt, daß durch eine geeignete Implementierung sowohl administrationsarme als auch leistungsfähige Cluster-Cluster-Kopplung realisiert werden kann.

Der IPv6-fähige Open-MPI-Quellcode befindet sich auf der beigefügten CD-ROM sowie unter `https://cluster.inf-ra.uni-jena.de/svn/mcm/openmpi/` und wird nach gegenwärtiger Planung als Bestandteil von Open MPI 1.3 veröffentlicht.

Inhaltsverzeichnis

Abkürzungsverzeichnis **5**

1 Einleitung **6**
1.1 Motivation . . . 6
1.2 Bestehende Lösungen für den Multicluster-Betrieb . . . 7
1.3 Anforderungen . . . 12

2 Grundlagen **16**
2.1 Netzwerkgrundlagen . . . 16
2.1.1 IPv4-Grundlagen . . . 16
2.1.2 IPv6-Grundlagen . . . 16
2.1.3 Site-Local-Adressen . . . 17
2.1.4 Spezielle IPv6-Präfixe . . . 17
2.2 Programmiertechnische Grundlagen . . . 18

3 Open MPI **21**
3.1 OPAL - Open Portable Access Layer . . . 21
3.2 ORTE - Open Runtime Environment . . . 22
3.3 OMPI - Open Message Passing Interface . . . 24
3.3.1 Grundlegende Funktionen des OMPI-Layers . . . 24
3.3.2 BTL-Komponenten und -Instanzen . . . 25

4 Implementierung **28**
4.1 Portierung von OPAL . . . 29
4.1.1 Erweiterung grundlegender Datenstrukturen und Hilfsfunktionen . 29
4.1.2 Interface-Discovery . . . 32
4.2 Portierung des OOBs . . . 36
4.2.1 Technische Aspekte der IPv6-Erweiterung . . . 36
4.2.2 Adressauswahl in Multi-Transport-Single-Cell-Clustern . . . 42
4.3 Portierung des Byte-Transfer-Layers . . . 45
4.3.1 Grundlegende Maßnahmen der IPv6-Erweiterung . . . 45
4.3.2 Oversubscription . . . 47
4.4 Hostfiles . . . 50

5 Leistungsbewertung **51**
5.1 Meßvorhaben und -aufbau . . . 51
5.2 Leistungsunterschiede zwischen IPv4 und IPv6 im LAN . . . 52
5.3 Leistungsbewertung im realen Multicluster-Betrieb . . . 52

6 Zusammenfassung und Ausblick **58**

Literaturverzeichnis **59**

Anlagen 62

A Meßwerte 62

A.1 Intra-Cluster-Kommunikation mit IPv4 62
A.2 Intra-Cluster-Kommunikation mit IPv6 64
A.3 Headnode-Headnode-Kommunikation via IPv4 66
A.4 Headnode-Headnode-Kommunikation via IPv6 68
A.5 Ende-zu-Ende-Kommunikation mit IPv6 70

B Connection-Strings 72

B.1 Process-Launch-Protokoll für IPv4-orteds 72
B.2 Process-Launch-Protokoll für IPv4/6-fähige orteds 73
B.3 Illustration des Adress-Auswahl-Algorithmus 74

Abkürzungsverzeichnis

Open-MPI-spezifische Abkürzungen

Kürzel	Beschreibung	Position im Quelltext
BTL	Byte-Transfer-Layer	`ompi/mca/btl`
DSS	Data-Switch-System	`orte/dss`
GPR	General-Purpose-Registry	`orte/mca/gpr`
OOB	Out-of-Band-Communication	`orte/mca/oob`
PLS	Process-Launching-System	`orte/mca/pls`
PML	Peer-to-Peer-Management-Layer	`ompi/mca/pml`
RMGR	Ressource Manager	`orte/mca/rmgr`
RML	Runtime-Messaging-Layer	`orte/mca/rml`

Allgemeine Abkürzungen

Kürzel	Beschreibung
ACK	Acknowledge
HPC	High Performance Computing
I/O	Input/Output
IP	Internet Protocol
LAN	Local Area Network
MPI	Message Passing Interface
MSS	Maximum Segment Size
NAT	Network Address Translation
NIC	Network Interface Card, Netzwerkkarte
SUSv2	Single UNIX Specification, Version 2
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VPN	Virtual Private Network

1 Einleitung

1.1 Motivation

Die computergestützte Simulation gilt heutzutage neben Theorie und Experiment als dritte große Säule der Naturwissenschaft. In nahezu allen Disziplinen ist sie das Bindeglied zwischen Modellannahme und realweltlicher Verifizierung. Simulationen ersetzen dabei häufig empirische Versuche, wenn diese zu kostspielig oder aufgrund der Problemstellung sogar generell unmöglich sind.

Der Wunsch nach immer präziseren Simulationsmodellen führt seit Jahren zu einem stetig steigenden Bedarf an Rechenleistung, der in vielen Fällen nicht mehr durch einzelne Computer gedeckt werden kann und daher den Einsatz von Clustern erfordert.

Die Spitze der bundesweiten Bereitstellungspyramide für wissenschaftliches Rechnen bilden die Bundeshöchstleistungsrechenzentren in Jülich, Stuttgart und München, ihnen folgen die Landeshochleistungsrechenzentren der einzelnen Bundesländer. Am unteren Ende rangieren kleinere Einzelcluster mit üblicherweise 8-32 Knoten, wie sie an vielen Lehrstühlen oder in industriellen Entwicklungsabteilungen anzutreffen sind.

Da dem Wachstum bestehender Clusterinstallationen häufig sowohl finanzielle als auch bauliche Grenzen gesetzt sind, erscheint es verlockend, mehrere Cluster miteinander zu verbinden und somit bei CPU-intensiven Problemen mit geringen Kommunikationsanforderungen eine höhere Gesamtrechenleistung zu erzielen. Diese Clusterverbünde werden *Multi-Domain-Cluster* (kurz: *Multicluster*) genannt [1]. Sie bestehen aus zwei oder mehreren bereits existierenden Einzelclustern, die nachträglich über ein Netzwerk miteinander verbunden werden. Abbildung 1 zeigt schematisch den typischen Aufbau eines solchen Multiclusters.

Innerhalb der beteiligten Einzelcluster kommen dabei im Low-Cost-Segment üblicherweise private IP-Netze zum Einsatz, die bisweilen sogar für die Definition des Clusterbegriffes herangezogen werden:

> **Cluster:** Set of interconnected computing nodes with nonvalid IP-addresses hidden behind an IP addressable front-end computing node [2].

Auch wenn diese Definition keine Allgemeingültigkeit besitzt, so steht sie doch stellvertretend für eine Vielzahl kleiner Clusterinstallationen. Die privaten IP-Adressen ermöglichen zwar die Kommunikation innerhalb der Einzelcluster, erschweren jedoch eine durchgängige Ende-zu-Ende-Kommunikation aller beteiligten Rechner im Multicluster-Betrieb, da sie üblicherweise nur innerhalb des jeweils lokalen Netzes gültig sind und somit nicht ohne Weiteres miteinander gekoppelt werden können.

Jeder einzelne Cluster bildet dabei eine *Zelle*, die all seine Rechner umfaßt. Innerhalb dieser Zelle können zwei beliebige Maschinen direkt miteinander kommunizieren, die Kommunikation mit der Außenwelt erfordert hingegen in vielen Fällen den Einsatz von *Network-Address-Translation* (NAT) [3] und ist folglich stark eingeschränkt, insbesondere sind die Compute-Nodes nicht auf herkömmlichem Wege aus dem Internet erreichbar.

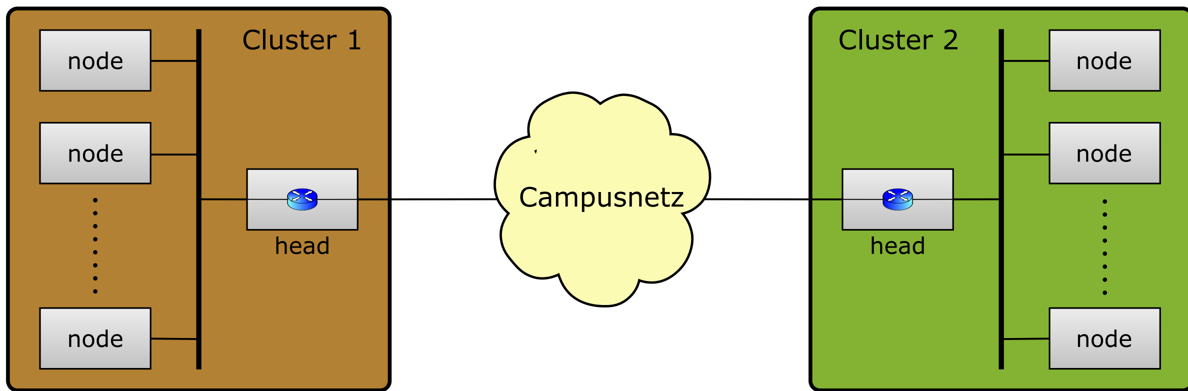

Abbildung 1: Schematischer Aufbau eines Multiclusters: Mehrere bestehende Cluster werden miteinander verbunden, um eine höhere Gesamtleistung zu erzielen.

Eine Kopplung mehrerer solcher Zellen zu einem Multicluster wird daher auch *Multi-Cell-Aufbau* genannt und beschreibt das Treffen geeigneter Maßnahmen, eigenständige Zellen zu einem gemeinsamen *Universum* zusammenzufassen.

Mit der Verfügbarkeit IPv6-fähiger Betriebssysteme eröffnet sich die Möglichkeit, alle beteiligten Rechner direkt zu adressieren und somit bereits auf Netzwerkebene eine Kopplung zu erzielen. Die zur Programmierung paralleler Anwendungen eingesetzten Message-Passing-Frameworks sind gegenwärtig bis auf prototypische Ausnahmen nur für IPv4 verfügbar und können deshalb nicht für IPv6-Multicluster herangezogen werden. Es ist daher Gegenstand dieser Diplomarbeit, am Beispiel von Open MPI [4] eine solche Bibliothek für den Nachrichtenaustausch um IPv6-Unterstützung zu erweitern, zur Praxisreife zu führen sowie auf ihre Eignung zur Cluster-Cluster-Kopplung zu untersuchen.

Ziel der Implementierung ist es, einen IPv6-basierenden, selbstkonfigurierenden und leistungsfähigen Nachrichtentransport für den Multicluster-Betrieb zu schaffen, der darüber hinaus auch mit bestehenden, ausschließlich IPv4 verwendenden Clusterinstallationen zusammenarbeitet. Ein derartig angepaßtes Message-Passing-Framework erleichtert die schrittweise Migration auf IPv6 und bietet gleichzeitig eine kostengünstige Möglichkeit, bereits vorhandene, aber über mehrere Cluster verteilte Ressourcen für das Lösen CPU-sensitiver Probleme heranzuziehen.

1.2 Bestehende Lösungen für den Multicluster-Betrieb

Das Verbinden mehrerer Einzelcluster zu einem Multicluster ist in erster Linie eine Frage der Kommunikation: Wie sollen Daten die Cluster-Zellgrenzen überwinden, um Teilnehmer in nicht-lokalen Netzen zu erreichen?

Die dafür existierenden Lösungen lassen sich konzeptionell in zwei Klassen einteilen:

1. Systeme, die Cluster auf der *Netzwerkebene* miteinander verbinden sowie
2. *anwendungsbasierte* Ansätze.

Die Kopplung auf Netzwerkebene erfolgt dabei heutzutage üblicherweise mit Hilfe von *Virtual Private Networks* (VPNs). VPNs transportieren private Daten über ein öffentliches Netz, indem die angeschlossenen internen Netze durch dedizierte Tunnel miteinander verbunden werden. Sollen lokale Daten einen nicht-lokalen VPN-Teilnehmer erreichen, werden die internen IP-Pakete des Datenstroms auf dem *VPN-Gateway* in neue IP-Pakete mit externen Adressen verpackt und darin über das öffentliche Netz versandt. Am anderen Tunnelende entfernt das dortige Gateway die Transport-Header wieder und leitet die Nutzpakete in das eigene interne Netz.

Beschränkt sich die Funktionalität des VPN-Tunnels wie beschrieben auf das Voranstellen eines IP-Transport-Headers, spricht man von *IP-in-IP-Kapselung* oder kurz von *IPIP-Tunneln*. Soll ein VPN-Tunnel die von ihm transportierten Daten hingegen auch verschlüsseln können, kommt statt dessen häufig IPsec [5] zum Einsatz.

Unabhängig von der Art des Tunnels können innerhalb eines VPNs private IP-Adressen geroutet werden, so daß damit die Ende-zu-Ende-Kommunikation für zwei beliebige Rechner auch über Clustergrenzen hinweg ermöglicht wird.

Wesentlicher Nachteil VPN-basierender Lösungen ist ihre fehlende Skalierbarkeit. Alle beteiligten Cluster müssen ihre internen Adressen so vergeben, daß sie nicht mit Netzen anderer Teilnehmer kollidieren, da anderenfalls die benötigte Eindeutigkeit nicht mehr gewährleistet wäre.

Dieses nachträgliche IP-Renumbering kann erheblichen administrativen Aufwand bedeuten, wenn Nameserver und Netzwerk-Dienste (z.B. Fileserver, Firewalls) ebenfalls auf den anderen, neuen Adressbereich umkonfiguriert werden müssen. Solch ein Vorhaben stößt aufgrund seiner Dimension häufig auf Ressentiments, so daß die generelle Bereitschaft zur Teilnahme am Multicluster-Betrieb in vielen Fällen sinkt.

Ferner mag zwar mit IP-Renumbering eine kollisionsfreie Adressallokation erzielt werden, möchte man jedoch den Einzelcluster später einem anderen Multicluster hinzufügen, kann erneut eine Rekonfiguration erforderlich sein, wenn der eigene Adressbereich innerhalb des neuen Multiclusters bereits vergeben ist.

Der Aufbau der VPN-Verbindung(en) gestaltet sich je nach verwendeter Technologie ebenfalls nicht immer völlig trivial: während sich die Konfiguration von IPIP-Tunneln im Wesentlichen auf die Angabe der Zieladresse reduziert, erfordern verschlüsselte VPNs vorab den Austausch von Schlüsselinformation. Das Erzeugen, Signieren und Hinterlegen der Zertifikate muß mit allen beteiligten Einzelclustern abgestimmt und dazu passende Krypto-Protokolle, z.B. die einzelnen Betriebsmodi für IPsec, vereinbart werden.

Übernimmt der Headnode die Rolle des lokalen VPN-Gateways, wird sein Prozessor durch das Verschlüsseln des Datenstroms nenneswert belastet und somit die zur Verfügung stehende Leistung in durchaus kritischem Umfang reduziert. Eine hohe Systemlast wirkt sich auf die Verschlüsselungsleistung aus und beeinträchtigt damit Latenz und Durchsatz für Verbindungen zu anderen VPN-Teilnehmern. Insbesondere skaliert dieser Ansatz nicht mit steigender externer Bandbreite, so daß die erzielbare Performance selbst bei Einsatz spezialisierter VPN-Hardware üblicherweise hinter der Leitungsgeschwindigkeit zurückbleibt.

Im Unterschied zu netzwerkbasierender Kopplung setzt die anwendungsbasierte Kopplung auf Proxy-ähnliche Lösungen im Userspace. Um die Programmierung von paral-

lelen Anwendungen zu erleichtern, kommen im Clusterumfeld häufig Bibliotheken für den Nachrichtenaustausch zum Einsatz, beispielsweise eine MPI-Implementierung oder PVM. Bei Kenntnis der diesen Softwarepaketen zugrundeliegenden Kommunikationsprotokolle ist es möglich, mehrere Zellen miteinander zu koppeln, indem Verbindungen an einen entfernten Teilnehmer auf dem lokalen Headnode entgegen genommen, zum entfernten Headnode weitergeleitet und von dort aus dem gewünschten Zielrechner zugestellt werden.

Die Arbeitsweise dieser Protokollkonverter entspricht dem klassischen Proxy-Konzept, bei dem ein exponierter Gateway-Prozess die nicht-lokalen Prozesse repräsentiert. Insbesondere kommt dabei das MPI- bzw. PVM- Adressierungsschema zum Tragen, das vom darunterliegenden Netzwerktransport abstrahiert und somit selbst das Koppeln gleichnamiger privater IP-Netze ermöglicht, ein etwaiges IP-Renumbering also entfallen kann.

Die Konfiguration der Proxy-Server gestaltet sich üblicherweise eher kompliziert, da aufgrund des Multi-Zellen-Aufbaus eine geeignete Zuordnung zwischen lokalen und entfernten Prozessen gefunden werden muß.

Frühe Ansätze wie Beolin [2] waren daher auf lediglich einen Prozess pro Compute-Node beschränkt, diese Limitierung wurde jedoch mit ePVM [2] überwunden. Die implementierte 3-schichtige Kommunikationserweiterung verursacht deutliche Leistungseinbußen von bis zu 75% und kann daher für den Einsatz im HPC-Umfeld nicht überzeugen.

MD-PVM [1] versucht dieses Defizit mit Hilfe eines eigenen Transportprotokolls zu kompensieren, das die Geschwindigkeitsvorteile von UDP mit der Zuverlässigkeit von TCP vereint. Neben zwei weiteren Management-Daemonen wurden dazu in einer separaten Bibliothek bekannte Konzepte reimplementiert: Fragmentierung, Sequenznummern, (kumulierte) ACK-Pakete, Congestion-Control sowie weitere TCP-Merkmale [6, 7, 8, 9]. Im Vergleich zu ePVM konnte damit die Geschwindigkeit um 15%-50% gesteigert werden, sie bleibt aber gemessen an der theoretisch verfügbaren Bandbreite nach wie vor deutlich hinter den Möglichkeiten zurück.

Für MPICH-basierende Umgebungen steht ein H2O-MPI-Proxy-Pluglet [10] zur Verfügung, das im Gegensatz zu bisher angesprochenen Lösungen kaum administrativen Mehraufwand erfordert: die beteiligten MPICH-Programme instanziieren selbstständig das Proxy-Pluglet, das Nachrichten an entfernte Prozesse über das H2O-Framework umlenkt. Auf der Gegenseite werden sie vom H2O-Kernel empfangen, an das dortige Proxy-Pluglet übergeben und schlußendlich der lokalen MPICH-Bibliothek zugeführt.

Die durch das H2O-Pluglet erzielte Geschwindigkeit liegt 26,8% unterhalb des Durchsatzes einer direkten MPICH-TCP-Verbindung, allerdings wurden nur Nachrichtenlängen oberhalb von 1024 Byte betrachtet, so daß keine Aussage über die durchaus relevante Latenz kurzer Nachrichten getroffen werden kann.

Ein weiterer Vertreter der Cluster-Cluster-Kopplung ist PACX [11], das als eigenständiger Gateway-Prozess MPI-Installationen miteinander verbindet, indem es Nachrichten der lokalen Vendor-MPI-Implementierung bei Bedarf über TCP an ein entferntes PACX-Gateway sendet und der dortigen Vendor-MPI-Bibliothek zuführt. Der erzielbare Durchsatz bewegt sich je nach Konfiguration zwischen 24 und 50% der physikalischen Leitungsbandbreite, die Latenz wird aufgrund zusätzlicher Steuer-Nachrichten und ggf. zeitintensiver Paketkompression ebenfalls um mindestens 150% verschlechtert. Zur Si-

cherung der Kommunikationsleistung kann PACX dedizierte Gateway-Nodes nutzen, die nicht in den Berechnungsprozess involviert sind und somit exklusiv für den Nachrichtentransport zur Verfügung stehen. Der für PACX erforderliche administrative Aufwand ist nicht zuletzt durch die Existenz und Konfiguration dieser Gateway-Maschinen als vergleichsweise hoch einzustufen.

Neben den bereits gezeigten VPN- und proxybasierten Lösungen stellt der Einsatz von IPv6 eine dritte Möglichkeit der Cluster-Cluster-Kopplung dar. Als Vertreter der netzwerkbasierten Kopplung erfordert er neben IPv6-fähigen Betriebssystemen auch IPv6-fähige Message-Passing-Bibliotheken, bietet dann aber mit seinem 128bit großen Adressraum genügend global eindeutige Adressen, die im Gegensatz zu IPv4 auch noch nahezu kostenfrei bezogen werden können.

Für PVM wurde schon 2005 eine experimentelle IPv6-Implementierung [12] geschaffen, die gänzlich auf IPv4 verzichtet. Diese vollständige Abkehr erscheint dabei wenig sinnvoll: in Ermangelung von IPv4-Unterstützung kann die Bibliothek nur auf Systemen mit IPv6-Konnektivität eingesetzt werden, auf absehbare Zeit wird jedoch das Gros der Installationen nicht mit IPv6 ausgestattet sein.

Als Ausweg wird das Verwenden von *6to4-Adressen* [13] vorgeschlagen. Diese speziellen IPv6-Adressen werden aus dem Präfix *2002::/16* und der öffentlichen IPv4-Adresse (32 Bit) gebildet, so daß von den 128 Bits einer IPv6-Adresse noch ein 80 Bit breiter IPv6-Adressraum für den clusterinternen Gebrauch zur Verfügung steht. Offenkundig können lediglich Cluster mit mindestens einer öffentlichen IPv4-Adresse den 6to4-Mechanismus nutzen, isolierte (z.B. firmeninterne) Installationen ohne externe Anbindung erhalten auf diesem Wege keine gültigen IPv6-Adressen.

Die Kommunikation zu anderen IPv6-Teilnehmern erfolgt bei 6to4-Adressen über *Anycast-Gateways* [14]. Alle Anycast-Gateways tragen die stets gleiche Anycast-IPv4-Adresse 192.88.99.1. Anycast-Adressen können weltweit an mehrere Rechner gleichzeitig vergeben werden, die an sie adressierten Datenströme werden dann durch die Router den jeweils netztopologisch nächstgelegenen Maschinen zugeleitet.

Sollen IPv6-Pakete den eigenen 6to4-Adressraum verlassen, werden sie in IPv4-Pakete gekapselt und an 192.88.99.1 versandt. Das nächstgelegene 6to4-Anycast-Gateway extrahiert aus den eintreffenden IPv4-Paketen die IPv6-Nutzlast und leitet sie über die eigene IPv6-Infrastruktur an den IPv6-Empfänger weiter, bei Bedarf auch durch erneutes Aussenden an die öffentliche IPv4-Adresse eines anderen 6to4-Teilnehmers. 6to4-Anycast-Gateways verbinden also isolierte IPv6-Inseln im IPv4-Raum mit dem übrigen IPv6-Adressbereich.

Da 6to4-Anycast-Gateways üblicherweise nur von großen Backbone-Providern mit eigener IPv6-Infrastruktur betrieben werden, führt die Cluster-Cluster-Kommunikation im Falle von 6to4-Adressen zu erheblichem externen Datenvolumen, wenn zwei interne Cluster miteinander in Kontakt treten wollen. Pakete werden dazu vom Headnode zum Anycast-Gateway geschickt, dort ausgewertet und danach via IPv4 dem anderen 6to4-Adressbereich zugeleitet. Die netztopologische Distanz zwischen Headnode und Anycast-Gateway wird aber für Message-Passing stets zu groß sein, da der Datenverkehr mehrere Router überwinden muß und somit die Nachrichtenlatenz empfindlich erhöht wird. Darüber hinaus sind 6to4-Anycast-Gateways nicht für Multicluster-typische Bandbreiten

ausgelegt, eine Gigabit-Cluster-Cluster-Kopplung ist damit zumindest heutzutage nicht möglich.

Mit Hilfe von zusätzlichen internen 6to4-Routen können im Falle mehrerer interner Cluster die 6to4-Anycast-Gateways vermieden werden, indem die beteiligten Headnodes ihre Daten direkt an die öffentliche IPv4-Adresse der anderen Cluster schicken. Offenkundig erfordert dies wieder administrativen Aufwand, wenn für jeden neu hinzukommenden Cluster auf allen Systemen eine weitere 6to4-Strecke konfiguriert werden muß.

Neben den gezeigten Einschränkungen des 6to4-Mechanismus spricht vor allem die möglicherweise fehlende IPv6-Unterstützung des Betriebssystems gegen eine nicht-IPv4-fähige Message-Passing-Bibliothek. Zwar dürften im High-Performance-Umfeld vorrangig moderne, IPv6-fähige Betriebssysteme anzutreffen sein, postulieren kann man es jedoch nicht. Ferner gibt es noch immer eine Vielzahl bestehender Installationen älterer Systeme, die mit einer derartigen Implementierung ausgeschlossen würden und damit die universelle Verwendbarkeit der Bibliothek beschneiden.

Einen ganz ähnlichen Ansatz zu der bereits gezeigten Lösung verfolgt die ebenfalls experimentelle MPICH-IPv6-Erweiterung [15] der Universität Potsdam. Die Autoren weisen darauf hin, daß eine spätere Implementierung sowohl IPv4 als auch IPv6 unterstützen sollte, um clusterintern via IPv4 kommunizieren zu können und IPv6 für den Multicluster-Fall zu nutzen. Die Forderung nach interner IPv4-Kommunikation wird nicht näher begründet, insbesondere kann sie nicht durch die gezeigten Leistungsdaten motiviert werden, im Gegenteil, sie belegen statt dessen die generelle Eignung von IPv6 für Message-Passing.

Die Autoren schlagen ferner vor, Cluster mit privaten IPv4-Netzen über IPv6-IPsec zu einem VPN zu verbinden. Neben den bereits bekannten Performance- und Renumbering-Problemen bezeugt diese Idee vor allem die anhaltende Dominanz privater IPv4-Adressen im Clusterumfeld.

Die konzeptionellen Unterschiede zwischen anwendungsbasierter und netzwerkbasierter Clusterkopplung motivieren dennoch eine Abkehr von nicht-öffentlichen Adressen: alle anwendungsbasierten Kopplungsverfahren begegnen dem Problem des Nachrichtentransports auf Anwendungsebene und damit weit oberhalb der Netzwerkhardware. Eintreffende Pakete werden von der Netzwerkkarte an das Betriebssystem geleitet, durchlaufen den IP- und ggf. sogar TCP-Stack, verlassen den Kernel in Richtung Standardbibliothek und werden erst dann der möglicherweise ebenfalls mehrschichtigen Proxy-Anwendung übergeben. Diese überprüft Quell- und Zieladresse, extrahiert die Nutzdaten und verpackt sie mit geänderten Adressen in ein neues Paket, das daraufhin die Kommunikations- und Standardbibliothek durchläuft, vom Netzwerk-Stack des Betriebssystem entgegegen genommen und schlußendlich der Netzwerkhardware zugeführt wird.

Alle diese Schritte kosten Zeit, das Problem gestaltet sich jedoch noch weiteraus schwieriger: empfängt ein Rechner neue Pakete, legt die Netzwerkhardware diese in ihrem Zwischenspeicher ab und löst einen Interrupt aus, der das Betriebssystem veranlaßt, die Daten zu verarbeiten.

Im Falle von IP-Paketrouting auf Kernelebene wird das Paket nach dem Dekodieren der L2-Header (z.B. Ethernet) dem IP-Stack übergeben, dort ausgewertet und danach wieder mit neuer Zieladresse versandt.

Während kernelbasierende Lösungen also nicht nur deutlich weniger Abstraktionsschichten zu durchlaufen haben, werden sie darüber hinaus sogar mittels Hardware-Interrupt über das Vorhandensein neuer Aufgaben informiert. Userspace-Programme hingegen erhalten ihre Ausführungszeit vom Scheduler des Betriebssystems. Liegen Daten für sie vor, werden diese in einem kernelinternen Puffer gesammelt und erst nach Zuteilung eines Zeitslots von der Anwendung ausgelesen.

Die Zeitspanne zwischen dem Eintreffen neuer Pakete und ihrer Verarbeitung wird maßgeblich durch die Systemlast bestimmt. Ohne spezielle Maßnahmen (z.B. Realtime-Scheduler) gilt, daß in einem vielbeschäftigten System die Proxy-Anwendung vergleichsweise lange warten muß, bis sie vom Betriebssystem Ausführungszeit zugeteilt bekommt. Liegt diese Wartezeit in der nicht untypischen Größenordnung von 10ms, so ist bereits die Kernel-Userspace-Übergangslatenz 100mal höher als die Nachrichtenlatenz von Gigabit-Ethernet.

Userspace-basierende Lösungen haben neben ihren oftmals hohen administrativen Anforderungen somit einen systeminhärenten Nachteil, der zu den bereits gezeigten Leistungseinbußen führt.

Für den Multicluster-Betrieb erscheint es daher wünschenswert, die Kopplung der Einzelcluster auf Netzwerkebene durchzuführen und damit von In-Kernel-Routing oder gar spezialisierter Routing-Hardware zu profitieren. Soll zwischen allen beteiligten Compute-Nodes echtes Routing stattfinden, müssen diese auf IP-Ebene eindeutig adressierbar sein. Private IP-Adressen bieten diese (globale) Eindeutigkeit nicht, da sie naturgemäß beliebig vergeben werden können.

Um sowohl das bereits gezeigte Renumbering-Problem zu umgehen als auch direktes Routing zu unterstützen, müssen alle Compute-Nodes mit öffentlichen IP-Adressen ausgestattet werden. Der so entstehende Multicluster besitzt dadurch einen gemeinsamen, eindeutigen Adressraum, einen durchgängigen Transport.

Wesentlich für den Begriff der Zelle ist, daß alle Rechner innerhalb einer Zelle direkt, d.h. ohne Adress-Übersetzungsmechanismen (z.B. NAT, Gateways), miteinander kommunizieren können. Ein derartig aufgebauter Multicluster besteht zwar nachwievor aus mehreren Einzelclustern, diese befinden sich aber in einem gemeinsamen, durchgängig adressierbaren Netzwerk und folglich in einer einzigen Zelle, weshalb dieser Ansatz naheliegenderweise *Single-Cell-Konzept* genannt wird.

Die Vergabe von öffentlichen IPv4-Adressen für Clusternodes ist jedoch oftmals nicht möglich, da öffentliche IPv4-Adressen aufgrund der bestehenden Adressallokation in vielen kleineren Einrichtungen nicht in ausreichendem Maße zur Verfügung stehen.

Mit dem anhaltenden Wachstum des Internets wird sich diese Situation noch weiter verschärfen, so daß IPv4 bereits heute nur in seltenen Fällen ein geeignetes Transportprotokoll für Single-Cell-Multicluster darstellt.

1.3 Anforderungen

Die Diskussion bestehender Lösungen für den Multicluster-Betrieb hat gezeigt, daß IPv6 ein vielversprechender alternativer Ansatz netzwerkbasierter Cluster-Cluster-Kopplung darstellt: der 128 Bit große Adressraum ermöglicht die Vergabe weltweit eindeutiger

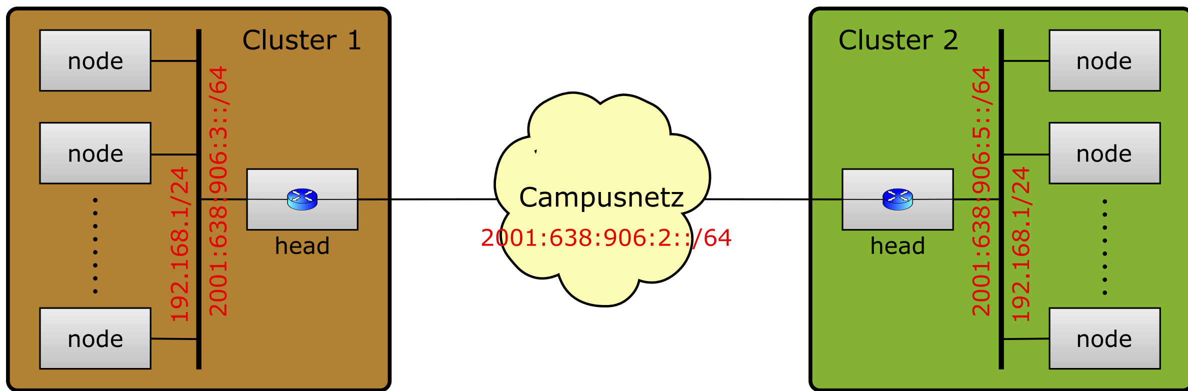

Abbildung 2: Ziel der Entwicklung: Eine MPI-Bibliothek, die sowohl IPv4 als auch IPv6 unterstützt und damit gleichermaßen für Einzel- und Multicluster geeignet ist.

IPv6-Adressen und erlaubt dadurch effizientes kernelbasiertes Routing, ohne dabei erheblichen administrativen Aufwand für die Konfiguration der angeschlossenen Systeme und Netze zu verursachen.

Die existierenden experimentellen Implementierungen stellen jedoch lediglich Machbarkeitsstudien dar, die sich insbesondere aufgrund ihrer fehlenden IPv4-Kompatibilität nicht für den universellen Alltagseinsatz eignen und sich folglich auch nicht etablieren konnten.

Gegenstand dieser Diplomarbeit ist es, am Beispiel der freien MPI-Implementierung *Open MPI* [4] IPv6-Message-Passing zur Anwendungsreife zu führen, dabei möglicherweise auftretende konzeptionelle Probleme zu untersuchen und die Leistungsfähigkeit kernelbasierter Cluster-Cluster-Kopplung in einem IPv6-Multicluster durch Messung zu belegen.

Die Open-MPI-IPv6-Portierung soll dabei folgende Anforderungen erfüllen:

1. Uneingeschränkter Erhalt der IPv4-Fähigkeit.

2. Unterstützung für SUSv2-basierende Betriebssysteme, die nicht über die neue Socket-API nach RFC 2460 verfügen

3. Plattformunabhängigkeit innerhalb unixoider Betriebssysteme, mindestens jedoch Linux, Solaris und *BSD/MacOSX auf HPC-typischen Plattformen wie AMD64, ix86, ppc oder ppc64

4. Verzicht auf IPv4-mapped-IPv6-Adressen nach RFC 3493

Der Erhalt der bestehenden IPv4-Fähigkeit bedarf nunmehr kaum einer Begründung: Die Unterstützung beider Adressfamilien innerhalb einer Bibliothek ist notwendig, da auf absehbare Zeit eine Vielzahl von reinen IPv4-Installationen existieren wird. Die Alternative, separate Bibliotheken für IPv4 und IPv6 aus der gleichen Codebasis zu generieren, kann nicht überzeugen: zum einen wird durch das Pflegen zweier Installationen der administrative Aufwand erhöht, zum anderen erfordert das richtige Auswählen der

passenden Bibliothek zur Laufzeit Wissen beim Open-MPI-Endanwender, über das er nicht notwendigerweise verfügt.

Ferner ist es bis auf wenige Ausnahmen wie ping-vs-ping6 üblich, daß eine IPv6-fähige Anwendung sowohl IPv4 als auch IPv6 unterstützt. Analog dazu sollte eine Open-MPI-Standardinstallation gleichermaßen beide Adressfamilien bedienen können, um a) die bestehende Konvention im Interesse konsistenter Nutzerschnittstellen zu erfüllen und b) den Anwender von der zwingenden Notwendigkeit der Protokollauswahl zu befreien, wie sie im Falle separater Open-MPI-Bibliotheken unvermeidlich wäre.

Die Forderungen nach SUSv2-Kompatibilität sowie Plattformunabhängigkeit sind ebenfalls einsichtig: Open MPI wird auf vielen unterschiedlichen Systemen eingesetzt; eine Erweiterung, die bestehende Plattformen gänzlich ausschließt, hätte keine Chance auf Übernahme in das Projekt.

SUSv2-Kompatibilität bedeutet daher, Open MPI auch auf alten Betriebssystemen ohne IPv6-Socket-API kompilieren zu können, Plattformunabhängigkeit hingegen verlangt, daß der IPv6-Support nicht nur unter Linux sondern nach Möglichkeit auf allen unixoiden Betriebssystemen zur Verfügung stehen sollte.

Der Verzicht auf IPv4-mapped-IPv6-Adressen mag überraschen: sie wurden eingeführt, um IPv4-Verbindungen in IPv6-fähigen Anwendungen nutzen zu können, ohne im Quellcode stets beide Protokollfamilien gesondert behandeln zu müssen.

Ein Programm verwendet dazu intern ausschließlich IPv6-Konstrukte wie `struct sockaddr_in6` oder `AF_INET6`, kann aber trotzdem IPv4-Verbindungen bedienen, indem diese vom Betriebssystem in den Adressbereich ::ffff:a.b.c.d/96 abgebildet werden. Insbesondere lauscht die Anwendung lediglich auf einem AF_INET6-Socket, über den sowohl IPv4- als auch IPv6-Daten ausgetauscht werden. So wird beispielsweise eine IPv4-Verbindungsanforderung von 141.35.14.189 als ::ffff:141.35.14.189 repräsentiert und programmintern als gewöhnliche IPv6-Verbindung betrachtet, sofern sie nicht explizit mit Hilfe des Makros `IN6_IS_ADDR_V4MAPPED` als IPv4-Verbindung identifiert wurde.

Es ist offensichtlich, daß der Einsatz dieser Adressen eine wesentliche Erleichterung für das Portierung von Anwendungen auf IPv6 darstellt. Dennoch existieren zwei Gründe, auf sie zu verzichten: IPv4-mapped-IPv6-Adressen sind in Sicherheitsfragen umstritten, da sie IPv4-Adressen in einer Repräsentation verwenden, die so möglicherweise nicht vom Nutzer überblickt wird. Es besteht daher die Gefahr, daß (String-) Vergleiche in sicherheitsrelevantem Netzwerkcode, insbesondere in Firewalls, falsche Ergebnisse liefern, wenn sie beispielsweise *::ffff:10.0.0.1/104* gegen *10.0.0.1/8* prüfen müssen. Beide Strings repräsentieren das gleiche Netz, der erste in IPv4-mapped-IPv6-Notation, der zweite in herkömmlicher IPv4-Schreibweise, dennoch würde eine naive Implementierung Ungleichheit feststellen. Handelt es sich hierbei um einen zu filternden Adressbereich, würde diese vom Nutzer intendierte Regel aufgrund der internen Repräsentierungsunterschiede keine Beachtung finden.

Um solche Gefahren generell zu vermeiden, ermöglichen viele Betriebssysteme das Deaktivieren von IPv4-mapped-IPv6-Adressen, wie es insbesondere auf einigen BSD-Varianten die Standardeinstellung ist. Programme, die auf IPv4-mapped-IPv6-Adressen basieren, würden in diesem Falle ihre IPv4-Fähigkeit einbüßen.

Man kann argumentieren, daß es sich hierbei um eine administrative Einstellung han-

delt, die im Clusterumfeld immer auf den richtigen Wert zu setzen ist, wodurch die Zuständigkeit für funktionierende IPv4-Verbindungen vom Programmierer zum Clusterbetreiber verlagert wird. Es existiert jedoch noch ein zweiter wesentlicher Grund, der schlußendlich für Open MPI ausschlaggebend war: Microsoft Windows unterstützt keine IPv4-mapped-IPv6-Adressen, da dort IPv4- und IPv6-Stack strikt getrennt sind, es also eben keine administrative Entscheidung sondern per se unmöglich ist.

Obwohl Microsoft Windows weder in dieser Diplomarbeit noch im High-Performance-Umfeld großartig Beachtung findet, würde doch eine auf IPv4-mapped-IPv6-Adressen basierende Open-MPI-IPv6-Erweiterung dauerhaft eine Portierung auf Windows verhindern oder zumindest erheblich erschweren. Eine Windows-Anwendung, die sowohl IPv4 als auch IPv6 bedienen soll, muß daher mit unterschiedlichen Sockets arbeiten: der erste Socket verwaltet alle IPv4-Anfragen (AF_INET), ein zweiter dient für IPv6-Verbindungen (AF_INET6). Da solch eine Konstellation auch unter UNIX erforderlich ist, wenn IPv4-mapped-IPv6-Adressen nicht zum Einsatz kommen, führt der Verzicht auf IPv4-mapped-IPv6-Adressen zu Programmcode, der später vergleichsweise einfach auf Windows portierbar ist.

Die Diplomarbeit gliedert sich wie folgt: Kapitel 2 vermittelt die Grundlagen für IP-Netze. Neben der seit Jahren etablierten Terminologie für IPv4 wird insbesondere die neue IPv6-Sprachkonvention eingeführt sowie auf Besonderheiten und Unterschiede hingewiesen.

Kapitel 2.2 betrachtet grundlegende Aspekte der C-Netzwerkprogrammierung, um darauf aufbauend Unterschiede und notwendige Änderungen für IPv6-fähige Software zu erarbeiten. Schwerpunkt bildet dabei die neue Socket-API nach RFC 3493.

Kapitel 3 beschreibt den generellen Aufbau der MPI-Implementierung Open MPI, die die Grundlage dieser Diplomarbeit bildet. Darüber hinaus wird die für das Verständnis der Arbeit notwendige projektspezifische Terminologie vermittelt.

Kapitel 4 illustriert im Detail die schrittweise Erweiterung der Open-MPI-Bibliothek um IPv6-Unterstützung und bespricht die sich daraus ergebenden konzeptionellen Probleme.

In Kapitel 5 wird die resultierende MPI-Implementierung einer Leistungsbewertung unterzogen und in Kapitel 6 zusammenfassend beurteilt.

2 Grundlagen

2.1 Netzwerkgrundlagen

Das Internet-Protokoll ist der de-facto-Standard für Rechnernetze mit allgemeinem Anwendungscharakter. Als Grundlage für das Internet ist es in jedem modernen Computer-Betriebssystem anzutreffen, und auch mobile Endgeräte wie Handys oder Smartphones nutzen es in wachsendem Maße. Mehrere Millionen Installationen weltweit zeugen von der universellen Verwendbarkeit des Protokolls, wenngleich für Spezialaufgaben mitunter besser geeignete Lösungen existieren[1].

Die derzeit vorherrschende Version 4 des Internetprotokolls, kurz IPv4, wurde 1981 im RFC 791 [16] definiert und bildet bis heute die Grundlage des Internets, obwohl seit 1998 [17] IPv6 als Nachfolger designiert ist. Der Übergang zwischen diesen Adressfamilien vollzieht sich seit Jahren eher langsam, wurde doch mit RFC 2893 [18] das grundsätzliche Vorgehen bereits im August 2000 publiziert. Aktuelle Betriebssysteme unterstützen deshalb üblicherweise beide Protokolle, um eine transparente Migration zu ermöglichen. So ausgestattete Betriebssysteme werden auch *Dual-Stack*-Konfiguration genannt, da sie sowohl über einen IPv4- als auch über einen IPv6-Protokollstapel verfügen.

2.1.1 IPv4-Grundlagen

IPv4 bietet einen 32bit-Adressraum, der jedoch nicht durchgängig Verwendung findet. RFC 3330 [19] spezifiziert Adressblöcke, die nicht oder nur eingeschränkt genutzt werden können. Von zentraler Bedeutung sind hierbei RFC 1918-Netzwerke [20], die das Internet semantisch in einen öffentlichen (public) und viele private Teile segmentieren. Rechner mit öffentlichen Adressen bilden dabei eine große Zusammenhangskomponente, in der Daten zwischen zwei Repräsentaten dieser Äquivalenzklasse uneingeschränkt geroutet werden können. Die Kommunikation in privaten Netzen ist ebenfalls unbeschränkt, der Zugriff auf öffentliche Adressen oder andere private Netze unterliegt aber erheblichen Einschränkungen, insbesondere ist kein direkter Datenverkehr zwischen diesen Klassen möglich. Als Ausweg hat sich das in RFC 2663 [3] spezifizierte Network-Address-Translation-Verfahren (NAT) etabliert, das analog zu Application-Level-Proxies heutzutage eine Vielzahl von privat adressierten Rechnern hinter einer öffentlichen Adresse zu bedienen vermag und häufig bei Clusterinstallationen zum Einsatz kommt, deren RFC 1918-adressierte Knoten üblicherweise nur über den öffentlich adressierten Headnode Zugang zum Internet erhalten.

2.1.2 IPv6-Grundlagen

Die wesentliche Neuerung von IPv6 in Bezug auf IPv4 ist der mit 128 Bit erheblich größere Adressraum, der es nicht nur ermöglicht, weltweit jedes elektrische Gerät[2] mit dem

[1] Insbesondere Message-Passing mittels IP ist vergleichsweise langsam. Infiniband sei als überlegene Alternative exemplarisch genannt.

[2] Im Gespräch sind neben Mobiltelefonen, Computern und Autos auch Kühlschränke, Fensterläden, TV-Settop-Boxen, MP3-Player...

Internet zu verbinden, sondern auch mit speziellen Adresspräfixen zusätzliche Funktionalität auf höherer Ebene bereitzustellen. Diese Präfixe implizieren gleichzeitig den **Scope**, der den Gültigkeitsbereich einer Adresse spezifiziert. Man unterscheidet im Wesentlichen die nachfolgenden Scopes:

global scope: Die angegebene Adresse ist weltweit eindeutig und kann daher für die Kommunikation mit anderen Global-Scope-Teilnehmern verwendet werden. Global-Scope-Adressen sind das Pendant zu öffentlichen IPv4-Adressen.

site local: Die angegebene Adresse ist nur innerhalb der Organisation gültig und kann nicht weltweit geroutet werden.

link local: Die angegebene Adresse ist nur für den physikalisch anliegenden Netzwerkteil gültig. Link-Local-Adressen dienen üblicherweise dazu, den lokalen Next-Hop-Router zu finden oder andere Teilnehmer im gleichen Netzsegment zu identifizieren[3].

Insbesondere für Multicast existieren weitere Scopes, die hier jedoch von untergeordneter Bedeutung sind und daher nicht näher betrachtet werden.

2.1.3 Site-Local-Adressen

IPv4 bietet mit RFC 1918-Adressen die Möglichkeit, lokal eigene Netze zu definieren, wenn keine öffentlichen Adressen verfügbar sind. Für IPv6 ist dies nicht notwendig, da auf absehbare Zeit genügend globale Unicast-Adressen vergeben werden können, sodaß das Konzept privater Adressbereiche im Allgemeinen nicht erwünscht ist, unterbricht es doch die direkte Ende-zu-Ende-Kommunikation. Da jedoch der Gedanke, alle Geräte in einem potenziell weltweit erreichbaren Netzwerk zu haben, nicht unisono begrüßt wird, bietet auch IPv6 private Adressen. Der bisherige Netzbereich für Site-Local-Adressen, fec0::/10, wurde mit RFC 3879 [21] als veraltet (deprecated) eingestuft und durch die in RFC 4193 [22] definierten Unique-Local-Adresses (ULA) im Bereich fc00::/7 ersetzt. ULAs sind im Gegensatz zu Site-Local-Adressen nicht frei wählbar sondern werden auf Antrag vergeben, um eine spätere Kommunikation zwischen diesen privaten Netzwerken zu ermöglichen, ohne der Gefahr der Adresskollision ausgeliefert zu sein.

Keine Form der RFC 1918-Pendants spielt bei IPv6 eine nennenswerte Rolle, führen sie doch den Sinn eines global nutzbaren Adressraums ad absurdum und kommen daher im Allgemeinen nicht zum Einsatz.

2.1.4 Spezielle IPv6-Präfixe

Wie auch bei IPv4 bezeichnen bei IPv6 spezielle führende Adressbits unterschiedliche Adressklassen und damit ihnen zugeordnete Anwendungszwecke:

[3] Vergleichbar mit Apples IPv4-basierter Rendez-Vous-Technik, um beispielsweise MP3-Daten mit anderen Nutzern im LAN zu teilen. Microsoft Vista verwendet Link-Local-Adressen, um mit mehreren Mitarbeitern gleichzeitig an einer Präsentationen zu arbeiten.

```
struct sockaddr_in6 {
    uint8_t         sin6_len;       /* length of this struct */
    sa_family_t     sin6_family;    /* AF_INET6 */
    in_port_t       sin6_port;      /* transport layer port # */
    uint32_t        sin6_flowinfo;  /* IPv6 flow information */
    struct in6_addr sin6_addr;      /* IPv6 address */
    uint32_t        sin6_scope_id;  /* set of interfaces for a scope */
};
```

Abbildung 3: Die Adress-Struktur `struct sockaddr_in6` nach RFC 3493.

::1/128 IPv6-Loopback-Adresse. Im Gegensatz zu IPv4 ist dies die einzige für Loopback vorgesehene Adresse, ein Äquivalent für 127.0.0.0/8 existiert nicht.

fe80::/10 Link-Local-Adressen.

::ffff:a.b.c.d/96 IPv4-mapped-IPv6-Adressen nach RFC 3493 [23], um IPv4-Adressen in IPv6-Anwendungen zu repräsentieren.

2000::/3 Von der IANA als Global Unicast vergeben.

2001:: An Provider zugeteilter Global-Unicast-Adressbereich

2002:: 6to4-Adressen nach RFC 3056 [13], die via IPv4-Anycast-Gateway 192.88.99.1 (RFC 3068 [14]) erreichbar sind.

2.2 Programmiertechnische Grundlagen

Für die Programmierung von netzwerkfähiger Software in C hat sich im unixoiden Umfeld die BSD-Socket-API etabliert, die in leichter Abwandlung ebenso bei Win32 Verwendung findet. Diese Schnittstelle ist seit jeher protokollunspezifisch, so daß sie in RFC 3493[23] um IPv6-Fähigkeiten erweitert wurde. Grundsätzlich[4] können daher bestehende Anwendungen nach dem in [24] skizzierten Verfahren mit wenigen Modifikationen auf IPv6 umgestellt werden.

Wesentliche Neuerungen haben die Datenstrukturen zur Adressrepräsentation erfahren, um die deutlich längeren IPv6-Adressen aufnehmen zu können. Mit `struct sockaddr_storage` steht nun ein 128 Byte großer Datentyp zur Verfügung, der auch für zukünftige Adressfamilien genügend Speicherplatz bietet und damit erneute Änderungen der Anwendungssoftware vermeiden bzw. mindern soll. Er ersetzt `struct sockaddr`, dessen 16 Byte zwar für eine IPv6-Adresse ausreichend wären, nicht jedoch weitere Felder wie `af_family` oder `uint16_t port` aufnehmen könnten.

Das Pendant zu `struct sockaddr_in` stellt der in Abbildung 3 gezeigte `struct sockaddr_in6` dar, der in Abhängigkeit seines `sin6_family`-Felds sowohl für IPv4 als

[4] Im juristischen Wortsinn: *Solange nichts anderes dagegen spricht.*

auch IPv6 geeignet ist. Die generische Vorgehensweise ist jedoch, generell `struct sockaddr_storage` zu verwenden und diesen entsprechend des `ss_family`-Eintrags per Cast-Befehl als `struct sockaddr_in*` oder `struct sockaddr_in6*` zu interpretieren:

```
void example(void) {
    struct sockaddr_storage adresse;
    /* some kind of assignment, code, whatever */

    switch (adresse.ss_family) {
    case AF_INET:
        {
            struct sockaddr_in *addr = (struct sockaddr_in*) &adresse;
            /* code for the IPv4 case */
        }
        break;
    case AF_INET6:
        {
            struct sockaddr_in6 *addr = (struct sockaddr_in6*) &adresse;
            /* code for the IPv6 case */
        }
        break;
    default:
        /* error handling for unknown address families */
    }
}
```

Im Kontext dieser Erweiterung wurden `connect()`, `bind()` und `accept()` an die neue Adressfamilie angepaßt, entgegen der Manpage zeigt der ihnen übergebene Pointer aber nicht auf einen `struct sockaddr`-Speicherbereich sondern wie vorgesehen auf `struct sockaddr_storage`. Solaris erfordert zusätzlich, daß die `addrlen`-Parameter die tatsächliche Größe des der jeweiligen Adressfamilie zugeordeneten `struct sockaddr_AF`s angeben, z.B. `sizeof(struct sockaddr_in)` oder `sizeof(struct sockaddr_in6)`. Glibc-basierende Systeme wie GNU/Linux akzeptieren die universelle Angabe `sizeof(struct sockaddr_storage)`, im Sinne portabler Software sollte davon jedoch abgesehen und statt dessen zuvor in Abhängigkeit der Adressfamilie der korrekte Wert ermittelt werden:

```
void example(void) {
    struct sockaddr_storage adresse;
    socklen_t addrlen;
    int fd, rc;
    /* some kind of assignment, code, whatever */

    switch (adresse.ss_family) {
```

```
    case AF_INET:
        addrlen = sizeof (struct sockaddr_in);
        break;
    case AF_INET6:
        addrlen = sizeof (struct sockaddr_in6);
        break;
    default:
        /* error handling for unknown address families */
    }

    rc = connect (fd, (struct sockaddr*)&adresse, addrlen);
    /* code for rc and fd not shown for the sake of simplicity */
}
```

Adresskonvertierungsfunktionen

Die programminterne Darstellung der Adressinformation in einem Struct der sockaddr-Familie erfolgt stets in Form von Bitfeldern in Network-Byte-Order und unterscheidet sich somit grundlegend von der stringbasierten Repräsentation für Rechnernamen, manuell angegebenen IP-Adressen oder Statusausgaben gegenüber dem Benutzer. Für IPv4 existiert eine Vielzahl von Funktionen, die diese unterschiedlichen Formate ineinander überführen können, allerdings sind sie fast ausnahmslos nicht für IPv6 geeignet. Konkret:

inet_addr(): Veraltet (obsolete), sollte nicht einmal mehr bei IPv4 verwendet werden.

inet_aton() und inet_ntoa(): Beide Funktionen sind nicht IPv6-fähig und wurden daher durch `inet_pton()` sowie `inet_ntop()` ersetzt.

gethostbyname(): Die Funktionssignatur würde IPv6-Fähigkeit ermöglichen, jedoch ist sie üblicherweise nicht implementiert.

Eine umfassende Aufstellung weiterer Adresskonvertierungsfunktionen findet sich in [25], die Detailkenntnis ist aber für IPv6 nicht erforderlich, da mit `getnameinfo()` und `getaddrinfo()` zwei POSIX-Funktionen vorliegen, die alle anderen vollständig ersetzen. Insbesondere können diese neuen Funktionen nach RFC 3493 sowohl für IPv4 als auch IPv6 genutzt werden und sollten daher heute grundsätzlich den Vorzug erhalten, um eine mögliche spätere Portierung der Anwendung auf IPv6 zu begünstigen.

Zu beachten ist, daß DNS-Anfragen selbst bei Beschränkung der Adressfamilie mehrere Ergebnisse liefern können und diese daher alle von der Anwendung zu überprüfen sind, bevor ein Verbindungsaufbau als unmöglich gemeldet werden darf. Üblicherweise iteriert man dazu in einer Schleife über einer einfach-verketteten Liste, die von der Standardbibliothek zur Verfügung gestellt und im Anschluß mit `freeaddrinfo()` freigegeben wird.

Detaillierte Programmbeispiele und weiterführende Erläuterungen zu `getnameinfo()` und `getaddrinfo()` finden sich in den IPv6-Porting-Guides [26, 27, 28] sowie in [25].

3 Open MPI

Open MPI [4] ist eine freie MPI-2.0-Implementierung, die aus den MPI-Bibliotheken FT-MPI, LA-MPI, LAM/MPI und PACX-MPI hervorgegangen ist, um gemeinsam die „beste verfügbare MPI-Library[5]“ zu schaffen. Neben den Universitäten Tennessee und Indiana gehören u.a. Cisco, IBM, Sun, das Los Alamos National Laboratory sowie das HLRS zu den Projektteilnehmern, die ihre langjährige Erfahrung auf dem Gebiet des Clustercomputings einbringen. Eine detaillierte Liste aller Beteiligten findet sich in [29].

Den namensprägenden Zusatz *Open* trägt das Projekt nicht nur aufgrund seiner BSD-ähnlichen Lizenz: Interessenten werden offen als neue Partner begrüßt und können so an der Bibliothek mitwirken. Die Frage, welche MPI-Implementierung um IPv6-Support erweitert werden sollte, wurde schlußendlich durch diese generelle Bereitschaft zugunsten von Open MPI entschieden.

Das dominierende interne Konzept hinter Open MPI heißt *modular component architecture*, kurz MCA. Es beschreibt ein plugin-ähnliches Design, das die Basis für das Gros der bereitgestellten Funktionalität bietet, die üblicherweise[6] durch Nachladen von dynamischen Bibliotheken automatisch an die Gegebenheiten der Umgebung angepaßt wird. Insbesondere ist es möglich, verschiedene Kommunikationswege gleichzeitig und unabhängig voneinander zu verwenden, indem die passende Komponente instanziiert wird. In der Open-MPI-Terminologie heißen diese Instanzen *modules*. So gibt es beispielsweise lediglich eine TCP-MPI-Transfer-Komponente, für jede verfügbare Netzwerkkarte wird aber zur Laufzeit ein separates TCP-Modul erzeugt[7].

Den Komponenten übergeordnet rangiert der Begriff des *Frameworks*, er steht für eine einzelne, wohldefinierte Aufgabe und abstrahiert von den möglichen Implementierungen, um dieses Ziel zu erreichen. Ein wesentlicher Vertreter ist der *Byte-Transfer-Layer*, kurz BTL, der unabhängig vom verwendeten Netzwerk die MPI-Datenkommunikation bereitstellt, indem er Komponenten für Infiniband, Myrinet, TCP und andere bietet.

Auf oberster Ebene gliedert sich Open MPI in die drei Teilbereiche *OMPI*, *ORTE* und *OPAL*, die in dieser Reihenfolge zwar aufeinander aufbauen, nicht jedoch im Sinne eines Callstacks nacheinander durchlaufen werden.

3.1 OPAL - Open Portable Access Layer

OPAL, der Open Portable Access Layer, liefert unterstützende Funktionalität für OMPI und ORTE, insbesondere alle internen projektweiten Datenstrukturen, z.B. `opal_list_t` für verkettete Listen oder `opal_object_t` für die Open-MPI-eigene Objektorientierung. Darüber hinaus beinhaltet OPAL eine eigenständige Speicherverwaltung, die `malloc()`-Aufrufe gegenüber der Standardlibrary reduziert, indem vorab größere Speicherblöcke angefordert werden. Speicheranfragen der Anwendung sowie der übrigen Bibliothek werden dann aus dem internen Pool beantwortet, wodurch auch auf Systemen mit subopti-

[5] Englischsprachiges Originalzitat der Website: *Open MPI is a project combining technologies and resources from several other projects [..] in order to build the best MPI library available.*

[6] Bei Verzicht auf shared libraries wird die Funktionalität statisch einkompiliert.

[7] Die C++-Analogie zwischen Klasse (component) und Instanz (module) ist zutreffend.

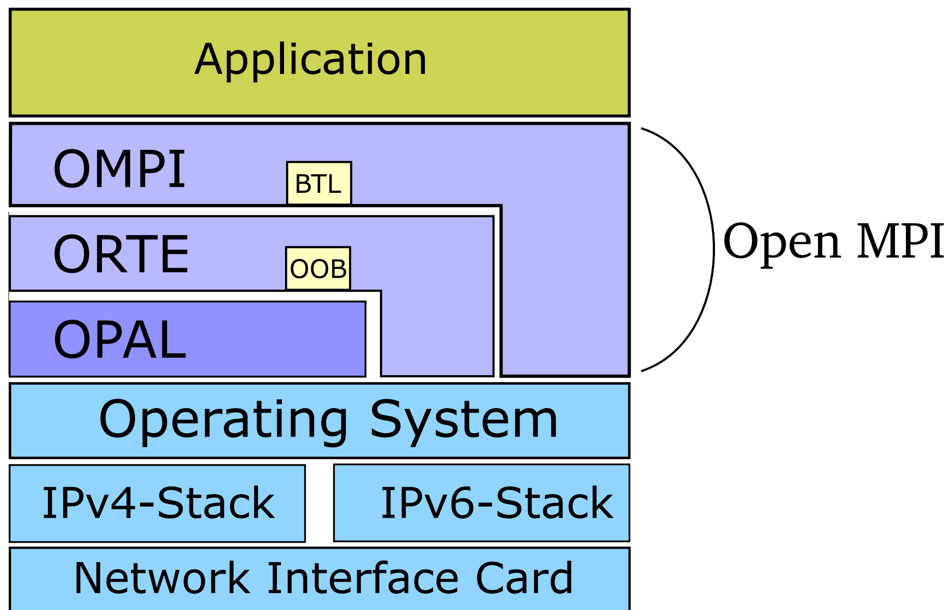

Abbildung 4: Schematischer Aufbau der Open-MPI-Biblbiothek.

maler `malloc()`-Implementierung eine leistungsstarke Speicherverwaltung zur Verfügung gestellt wird.

OPAL bietet ferner ein eigenes Thread- und Eventhandling, das konkurrierende Programmierung innerhalb der Open-MPI-Bibliothek ermöglicht. Diese wird beispielsweise für Timeouts und Retry-Zyklen beim Verbindungsaufbau genutzt, indem Timer-Events nach Ablauf einer Frist entsprechende Callback-Funktionen[8] auslösen. Darüber hinaus können Ereignisse auch an Sockets gebunden werden, so daß bei Aktivität einer Verbindung[9] die passende verarbeitende Prozedur von der Event-Library aufgerufen wird.

Darüber hinaus stellt OPAL eine Sammlung von Hilfsfunktionen zur Verfügung, die im gesamten Projekt Anwendung finden. Neben Konvertierungs- und Ausgabeprozeduren gehören beispielsweise auch Implementierungen für `qsort()`, `strncpy()` oder `basename()` dazu, die immer dann zum Einsatz kommen, wenn ein System diese nicht oder nur unzulänglich bietet. Schlußendlich zeichnet sich OPAL für die Ethernet-Interface-Discovery verantwortlich, die sowohl von OMPI als auch ORTE benötigt wird.

3.2 ORTE - Open Runtime Environment

Das Open Runtime Environment war ursprünglich ein kleiner Teil von Open MPI, ist aber inzwischen zu einem umfangreichen Projekt mit eigener Organisationsstruktur

[8]Übliches Szenario: Wenn nach fünf Sekunden keine Verbindung hergestellt werden konnte, brich ab und versuche es nach zehn Sekunden erneut.

[9]Typische Aktivitäten sind eintreffende Daten, Verbindungsauf- und -abbau sowie Störungen, z.B. vorzeitiger Verbindungsabbruch.

herangewachsen, dessen vorrangige Aufgabe das Bereitstellen einer geeigneten Umgebung für Message-Passing ist. Das Runtime-Environment wird auf jedem Host durch den ORTE-Daemon, *orted*, repräsentiert. Herzstück bildet dabei die General-Purpose-Registry (GPR), die Informationen zu allen beteiligten Prozessen enthält. Sie ist als MCA-Framework ausgeführt und dient anderen Komponenten als zentraler Datenspeicher, muß dafür aber auf das Data-Switch-System (DSS) zurückgreifen, um plattformspezifische Unterschiede wie Byte-Ordering[10] oder Packing[11] auszugleichen.

ORTEs zentraler Abstraktionsansatz ist die Zelle (*cell*), sie umfaßt alle Computer, die einen gemeinsamen Prozessstart-Mechanismus[12] aufweisen und wird von einem Ressource Manager (RMGR) verwaltet, der neben Ressourcenfindung und Ressourcenallokation vor allem für das Prozess-Mapping sowie die Prozess-Ausführung zuständig ist. Dazu nutzt er das Process Launching System (PLS), das je nach Zelle variieren kann[13], um auf allen beteiligten Knoten den orted zu starten, die mit Hilfe des Runtime Messaging Layers (RML) untereinander kommunizieren und somit einen zentralen Anlaufpunkt für den Austausch performanceunkritischer Nachrichten darstellen.

In einem typischen Start-Szenario erzeugt der Nutzer einen Headnode-Prozess (HNP), der das passende bzw. konfigurierte Prozessstart-Verfahren auswählt und daraufhin weitere Nodes kontaktiert. Die zu startenden orteds erhalten über Kommandozeilenparater alle notwendigen Informationen für die Zelle, insbesondere, wie sie den Headnode-Prozess erreichen können. Dafür existiert ein weiteres MCA-Framework, die Out of Band Communication (OOB), die derzeit aber lediglich eine Komponente (oob/tcp) enthält. Sie stellt den gesamten Datentransport für den Runtime-Messaging-Layer bereit, mit dessen Hilfe die neuen Prozesse in einem Callback-Verfahren den HNP bezüglich der eigenen Erreichbarkeit informieren. ORTE übernimmt daraufhin auf allen beteiligten Nodes das Überwachen und Verwalten der eigentlichen Nutzanwendung. Dazu gehört neben dem Starten und Beenden von (MPI-)Programmen auch das IO-Forwarding, das Ein- und Ausgaben der entfernten Knoten zum Headnode-Prozess umleitet. Ein Error-Manager regelt ferner das Verhalten in Ausnahmesituationen, z.B. wenn Anwendungen vorzeitig abgebrochen werden müssen oder gar nicht erst gestartet werden können. Da ORTE selbst keine Message-Passing-Funktionalität im Sinne des MPI-Standards bietet, kann es auch für das Ausführen beliebiger Programme[14] Verwendung finden, indem diese als Argument an *orterun* übergeben werden. Tatsächlich sind die beiden MPI-Äquivalente *mpirun* sowie *mpiexec* lediglich symbolische Links auf orterun und funktional gleichwertig, wie das nachfolgende Beispiel zeigt:

[10]Wie in heterogenen Umgebungen üblich müssen sich alle Beteiligten wahlweise aber einheitlich auf big endian oder little endian festlegen.

[11]Struct-Alignment ist immer wieder eine Fehlerquelle bei heterogener Kommunikation, die auch in der kommenden Open-MPI-Version 1.2 nur mittels Workaround (performancesenkendes Padding kann mittels configure-Switch zur Compile-Zeit aktiviert werden) suboptimal beseitigt sein wird.

[12]Englischsprachiges Originalzitat der Website: *one or more computers sharing the same launching system/point.*

[13]Bekannte Vertreter sind rsh (ssh), gridengine (für SGE-Umgebungen), bproc oder slurm.

[14]Es ergeben sich Einschränkungen für interaktive Programme, insbesondere I/O kann Probleme bereiten. X-Forwards sind möglich.

```
adi@ipc654:~$ orterun -np 2 -host inge,dana hostname
inge
dana
adi@ipc654:~$ mpirun -np 2 -host inge,dana hostname
inge
dana
adi@ipc654:~$ mpiexec -np 2 -host inge,dana hostname
inge
dana
```

Die Vereinigung aller Zellen wird *Universe* (Universum) genannt, das auch dann existiert, wenn wie im Falle klassischer Single-Cluster lediglich eine Zelle beteiligt ist. ORTEs Unterstützung für Multi-Domain-Cluster befindet sich derzeit noch in einem sehr frühen Stadium, transparente Gridintegration sowie Multi-Cell-Setups sind jedoch für OpenRTE-2.0 geplant, wenngleich die Entwickler den aktuellen Stand mit "For now, it's an idea." umschreiben. Ihr eigener Anspruch ist es dabei, daß ORTE ohne manuelle Konfiguration nach Möglichkeit stets „das Richtige tut"[30]. Dieses Ziel ist jedoch nicht trivial erfüllbar, wie Kapitel 4.2.2 zeigen wird.

3.3 OMPI - Open Message Passing Interface

3.3.1 Grundlegende Funktionen des OMPI-Layers

Dritter und gleichzeitig größter Bestandteil des Open-MPI-Projekts ist OMPI; schon die Namensgebung deutet an, daß es sich hierbei um die eigentliche Implementierung des MPI-(2)-Standards handelt. Sie beinhaltet derzeit Unterstützung für Anwendungsprogramme in C, C++, Fortran77 sowie Fortran90, intern sind aber alle Sprach-Bindings als Wrapper für C-Funktionen realisiert, da nur Letztere im Interesse der Performance vollständig ausprogrammiert vorliegen.

Das Gros der Funktionalität wird auch bei OMPI in Form von MCA-Komponenten zur Verfügung gestellt: Es existieren unter anderem Frameworks für Topologiesupport, kollektive Operationen oder MPI-IO, für diese Arbeit von besonderer Bedeutung ist jedoch der Peer-to-Peer-Management-Layer (PML) bzw. der ihm untergeordnete Byte-Transfer-Layer, kurz BTL.

Vorrangige Aufgabe des PMLs ist das Fragmentieren und Disponieren von MPI-P2P-Nachrichten über verschiedene Transportwege, der eigentliche Nachrichtentransfer wird hingegen von BTL-Instanzen übernommen. Während der Initalisierungsphase prüfen dazu alle BTL-Instanzen die Erreichbarkeit der anderen Teilnehmer, indem sie deren Kontaktinformation aus ORTEs General Purpose Registry mit der lokalen Konfiguration vergleichen. Diese sowie die eigenen Daten wurden zuvor mit Hilfe der PML-Funktionen `mca_pml_base_modex_recv()` und `mca_pml_base_modex_send()` zwischen den Beteiligten über ORTEs (möglicherweise langsamen) OOB-Kommunikationskanal ausgetauscht.

Aus der Menge der möglichen Transportwege wählt der PML diejenigen BTL-Instanzen aus, die a) den jeweiligen Prozess erreichen können und b) dafür am geeignesten sind.

In einem typischen Szenario ist der lokale Prozess via self-Komponente am schnellsten verfügbar, Prozesse auf dem gleichen Host nutzen das Shared-Memory-BTL (SM) und entfernte Kommunikationspartner bedürfen eines Netzwerks, z.B. in Form von Infiniband (OpenIB), Myrinet oder TCP. Die auf Erreichbarkeit und Leistung basierende Auswahl wird mit Hilfe einer internen Gewichtung[15] durchgeführt, so daß auch mehrere unterschiedliche Netzwerktechnologien gleichzeitig zum Einsatz kommen können[16]. Transportwege (BTLs), die nach der PML-Entscheidung nicht mehr für die Kommunikation vorgesehen sind, werden vom PML deaktiviert, ihr Speicher freigegeben und im Falle von dynamischen Bibliotheken sogar vollständig entladen.

Aufgabe des PMLs ist es im Anschluß, für die verbleibenden BTLs einen geeigneten Scheduling-Algorithmus[17] zu implementieren und anstehende Nachrichten auf die unterschiedlichen Kommunikationskanäle zu verteilen, wenn ein Ziel über mehrere Routen erreichbar ist [31]. Dieser Vorgang heißt *Message-Striping*.

3.3.2 BTL-Komponenten und -Instanzen

Aufgrund der unterschiedlichen BTL-Ausprägungen ist die Zuständigkeit für Message-Striping, also die Lastverteilung über verschiedene Leitungswege, nicht zwangsläufig beim PML angesiedelt sondern kann auch vom BTL selbst durchgeführt werden. Dieser Teilaspekt erfordert einen tiefergehenden Exkurs in den BTL, der jedoch gleichzeitig erheblich für das Verständnis späterer Implementierungsentscheidungen dienlich ist:

Der Byte-Transfer-Layer ist ein MCA-Framework für den MPI-P2P-Nachrichtentransport und unterliegt damit der üblichen MCA-Unterteilung in *Komponente* und *Modul*: Verzeichnisse[18] unterhalb von `ompi/mca/btl` sind Komponenten, ihre Instanzen heißen Module. Darüber hinaus bezeichnet der Begriff der Komponente aber auch die Zusammenfassung der Codeteile, die vom übergeordneten PML nur einmal aufgerufen werden und für alle Module (gleichzeitig) Wirkung erzielen. Das Beispiel soll es verdeutlichen:

Jede BTL-Komponente (lies: tcp, openib, mx…) implementiert die Funktion `mca_btl_base_component_init_fn_t()`, deren Rückgabewert eine der drei folgenden Ausprägungen annehmen kann:

1. eine NULL-Liste, wenn die Netzwerktechnologie nicht verfügbar ist[19]
2. eine Liste mit nur einem BTL-Modul, das selbst eine Abstraktionsschicht für den Umgang mit mehreren physischen Geräten (z.B. Netzwerkkarten) bietet.

[15]Der OMPI-spezifische Begriff für diese Gewichtung lautet *exclusivity*, er ist hier aber nicht von Bedeutung und findet daher nur in einer Fußnote Erwähnung.

[16]Denkbar ist ein Cluster, der nur teilweise über Infiniband verfügt und dessen verbleibende Nodes mittels TCP über Ethernet erreichbar sind.

[17]Der PML-Quelltext nennt exemplarisch Round-Robin sowie eine gewichtete Verteilung anhand des exclusivity levels.

[18]Das Verzeichnis selbst ist natürlich keine Komponente, der Begriff wird aber projektintern für die durch das Verzeichnis repräsentierte Codebasis verwendet. Im Sinne der C++-Terminologie handelt es sich hierbei um eine Klasse ohne Instanz, der Begriff ist also etwas metaphysisch.

[19]Die Infiniband-Komponente wird beispielsweise auf Rechnern ohne IB-Adapter NULL zurückliefern.

3. eine Liste mit mehreren BTL-Modulen, wobei jedes Modul ein einzelnes physisches Gerät repräsentiert

Während dieser Initialisierungsphase sollten die BTL-Komponenten mittels `mca_pml_base_modex_send()` ihre Kontaktinformation via ORTEs General Purpose Registry an die Peers übermitteln. Im Falle der BTL-TCP-Komponente öffnet dazu vorher der gemeinsame Komponentencode den Listen-Socket für eingehende Verbindungen und erzeugt danach für jede einzelne Netzwerkkarte neue Instanzen, die im Gegensatz zur „Komponente" jedoch ausschließlich abgehende Verbindungen bedienen und somit keinen eigenen Listen-Socket pro Interface erzeugen. Es besteht also eine Arbeitsteilung zwischen Komponente (lediglich eine für eingehende Verbindungen) und Modulen (eine pro Netzwerkkarte für abgehende Verbindungen). Diese Differenzierung spiegelt sich auch in der unterschiedlichen Anwendung der pml_base_modex-Funktionen wider: Die Komponente übermittelt eigene Adressen sowie den zugehörigen Listen-Port mit Hilfe von `mca_pml_base_modex_send()`, die Module hingegen empfangen die Peer-Information durch `mca_pml_base_modex_recv()`.

Es ist ferner zu beachten, daß diese Peer-Information erst im Anschluß der Komponenten-Initialisierungsphase und damit nach Verlassen der `mca_btl_base_component_init_fn_t()`-Funktion zur Verfügung steht, sie deshalb also nicht während der Modul-Initialisierung verwendet werden kann. Da ein Modul aber nur mit Hilfe der Peer-Information die Erreichbarkeit entfernter Prozesse zu entscheiden vermag, wird diese Frage später in einer separaten Phase beantwortet, indem der PML für jedes BTL-Modul dessen `mca_btl_base_add_proc_fn_t()`-Funktion ruft.

Daraus ergeben sich einige Implikationen:

1. Es ist zu Beginn nicht zu entscheiden, welche Peers auf welchem Weg erreichbar sind, da die notwendige Information noch nicht vorliegt.

2. Es ist aus gleichem Grunde nicht möglich, die eigene Erreichbarkeit in Abhängigkeit der anderen Teilnehmer zu melden, um dadurch Deadlocks zu vermeiden. Eine Entscheidung bezüglich eines Weges kann immer erst bei ausgehenden Verbindungen erfolgen.

3. Das gleichzeitige Verwenden mehrerer Interfaces kann entweder im Falle multipler BTL-Instanzen im PML erfolgen oder anderenfalls vom einzigen BTL-Modul direkt bewerkstelligt werden[20].

4. Wenn mehrere BTL-Instanzen existieren, können diese in Abhängigkeit ihrer eigenen Adresse zu unterschiedlichen Bewertungen der Erreichbarkeit eines Peers gelangen. Diese Entscheidung ist den anderen Modulen jedoch nicht bekannt, da Module private Informationen (private state) besitzen.

[20] Im Falle mehrerer verschiedenartiger Transportwege, z.B. Infiniband und TCP, können Mischformen existieren.

Eine IPv6-fähige Open MPI-Implementierung wird diese Punkte berücksichtigen müssen, wenn Peers sowohl über IPv4- als auch über IPv6-Adressen verfügen, um ein reibungsloses Zusammenspiel beider Adressfamilien zu ermöglichen.

4 Implementierung

Das Erweitern bestehender Anwendungen um IPv6-Unterstützung ist ein geradliniger Prozess [24], der sich üblicherweise in drei Abschnitte gliedert:

1. Auffinden aller Stellen im Quellcode, die netzwerkspezifische Funktionalität liefern, z.B. `struct sockaddr*`, `connect()`, `bind()`,...
2. Ersetzen dieser Strukturen und Prozeduraufrufe durch ihr IPv6-Pendant
3. Sofern Netzwerkadressen auf höherer Ebene benutzt werden müssen, z.B. als Kommandozeilenparameter oder als Ausgabe eines User-Interfaces, ist diese nachgelagerte Funktionalität ebenfalls zu erweitern.

Das Identifizieren relevanter Codeabschnitte erfolgt in erster Näherung durch Suchen nach typischen Zeichenketten im Quelltext. Als Schlüsselworte dienen dabei die von IPv4 bekannten Funktionen, Strukturen und Konstanten: `struct sockaddr`, `struct sockaddr_in`, `struct in_addr`, `listen()`, `accept()`, `bind()`, `socket()`, `INADDR_NONE`, `INADDR_ANY`, `INADDR_LOOPBACK`, `AF_INET` und weitere mehr.

Um diesen Vorgang zu erleichtern, bietet Sun mit *Socket Scrubber* ein Programm, das automatisiert alle einschlägigen Zeichenketten aufspürt und die Ergebnisse dieser Suche in einem Report zusammenfaßt. Ob ein Treffer tatsächlich relevant ist, muß manuell durch den Programmierer geprüft werden, da Socket Scrubber durchaus eine Vielzahl unnötiger Stellen markiert, wenn diese eines der o.g. Schlüsselworte enthalten. So ist

```
sd = socket (AF_INET, SOCK_STREAM, 0);
```

ein gefundenes Fragment, das im Zuge der IPv6-Erweiterung sehr wahrscheinlich berücksichtigt werden muß, wohingegen

```
float socket_voltage = 230.0;
```

zwar ebenfalls ausgewiesen wird, jedoch offenkundig keine Netzwerkfunktionalität bietet und somit auch nicht weiter betrachtet werden muß.

Nach Analyse von SLOCCount umfaßt der 28MB große Open-MPI-Quelltext ca. 300.000 Programmzeilen in 2700 Dateien, von denen aber lediglich ca. 13.000 Zeilen in 36 Dateien für TCP-Unterstützung verantwortlich sind. Dank des modularen Aufbaus des Projekts beschränkten sich somit notwendige Änderungen im Zuge der IPv6-Erweiterung im Wesentlichen auf die drei Unterverzeichnisse `orte/mca/oob/tcp` (OOB), `ompi/mca/btl/tcp` (BTL) sowie `opal/util` (OPAL). Diese kleine Anzahl relevanter Stellen war nach manueller Erstorientierung im Quellcode auch ohne den Einsatz des Socket Scrubbers überschaubar.

4.1 Portierung von OPAL

4.1.1 Erweiterung grundlegender Datenstrukturen und Hilfsfunktionen

Der Open Portable Access Layer (OPAL) bietet unterstützende Funktionen für IPv4-Kommunikation, die sowohl von OOB/TCP als auch von BTL/TCP genutzt werden. Der Code in `opal/util/if.[ch]` war daher Ausgangspunkt für die IPv6-Erweiterung. Ein Top-Down-Ansatz, der OPAL-Modifikationen an das Ende der Entwicklung gestellt hätte, wäre wenig erfolgreich verlaufen:

1. OOB und BTL benutzen exzessiv OPAL-Funktionen. Ein IPv6-fähiges OOB hätte für den Aufruf der OPAL-Prozeduren immer auf Parameter zurückgreifen müssen, die lediglich IPv4 ermöglicht hätten und daher

2. Tests während der Entwicklungsphase niemals die neue Funktionalität abgedeckt hätten, Fehler damit erst ganz zum Schluß gefunden werden könnten und das Konzept von Testdriven Development generell ad absurdum geführt worden wäre.

Grundlegende Datenstruktur in if.c ist eine verkettete Liste des Typs `opal_if_t`, die alle gefundenen Netzwerkkarten beinhaltet:

```
struct opal_if_t {
    opal_list_item_t    super;
    char                if_name[IF_NAMESIZE];
    int                 if_index;
    int                 if_flags;
    int                 if_speed;
    struct sockaddr_in  if_addr;
    struct sockaddr_in  if_mask;
    uint32_t            if_bandwidth;
};
typedef struct opal_if_t opal_if_t;
```

Offenkundig ist `struct sockaddr_in` nicht geeignet, IPv6-Adressen zu repräsentieren und muß daher durch `struct sockaddr_in6` oder besser noch `struct sockaddr_storage` ersetzt werden. Beide Varianten unterstützen sowohl IPv4 als auch IPv6, `struct sockaddr_storage` ist aber als generische Datenstruktur für beliebige Adressfamilien gedacht und stellt somit die allgemeinere, flexibel erweiterbare Lösung dar.

Die Entscheidung fiel dennoch auf `struct sockaddr_in6`, da die höheren Schichten OOB und BTL in ihren OPAL-Aufrufen bisher `struct sockaddr_in` nutzen. Eine Änderung zugunsten von `struct sockaddr_storage` hätte eine Vielzahl an Pointer-Konvertierungen nach sich gezogen, ohne auf absehbare Zeit einen funktionellen Mehrwert zu liefern. Ferner steht `struct sockaddr_storage` auf Systemen nach SUSv2 nicht zur Verfügung.

Auf SUSv2-Installationen existiert allerdings auch `struct sockaddr_in6` nicht, so daß im Interesse der Rückwärtskompatibilität in diesen Fällen weiterhin ein `struct`

`sockaddr_in` verwendet werden muß. Um dies zu ermöglichen, kann Open MPI mit dem configure-Flag `--disable-ipv6` übersetzt werden. Die Entscheidung, IPv6 nur fallweise abzuschalten (optional out) und ansonsten immer zu aktivieren, sofern das Betriebssystem die neue Socket-API nach RFC 2460 unterstützt, bezeugt das ernsthafte Interesse des Open-MPI-Projekts an dieser Erweiterung.

Das configure-Skript setzt je nach Nutzerwunsch das neu eingeführte Präprozessor-Makro `OPAL_WANT_IPV6` auf 1 oder 0, so daß die geforderte SUSv2-Unterstützung mit Hilfe bedingter Kompilierung ermöglicht werden kann: steht keine IPv6-fähige Socket-API zur Verfügung oder soll Open MPI ohne IPv6-Fähigkeiten übersetzt werden, kommt der bereits bestehende Code zum Einsatz, anderenfalls die neue Fassung:

```
#if OPAL_WANT_IPV6
    struct sockaddr_in6 if_addr;
#else
    struct sockaddr_in  if_addr;
#endif
```

Diese Fallunterscheidung zwischen altem und neuem Code entwickelte sich im Rahmen der IPv6-Erweiterung zu einem Standardwerkzeug, wenngleich die betroffenen Programmfragmente selten so kurz wie im Beispiel ausfielen: OPAL verwendet insgesamt 25x das Makro, BTL/TCP derzeit 48x und OOB/TCP sogar 128x. Die Zahlen korrespondieren mit der Komplexität der getroffenen Änderungen, sinken aber nennenswert, wenn SUSv2-Kompatibilität eines Tages als Anforderung aufgegeben wird. Es ist davon auszugehen, daß mit dem Trend zu neuer Hardware auch die alten Betriebssysteme aus dem High-Performance-Computing-Umfeld verschwinden werden und damit der Bedarf für SUSv2-Unterstützung entfällt.

Die oben dargestellte ursprüngliche Fassung von `opal_if_t` enthält gleich zwei `struct sockaddr_in`-Einträge: `if_addr` für eine IPv4-Adresse sowie `if_mask` für die zugehörige Netzmaske.

Die Umstellung von `if_addr` auf `struct sockaddr_in6` bietet kaum Anlaß zur Diskussion, ein analoges Vorgehen für `if_mask` ist jedoch weniger sinnvoll: IPv4-Netzmasken sind genauso wie IPv4-Adressen 32 Bit lang, IPv6-Netzmasken hingegen 128 Bit.

Das Studium des Quelltextes zeigte, daß Netzmasken lediglich von OOB und BTL benutzt werden, um für eine Remote-Adresse zu überprüfen, ob sie sich im gleichen Netzsegment mit einem lokalen Interface befindet. Dies geschieht ganz klassisch durch Ausmaskieren des Hostanteils mittels einer UND-Verknüpfung, indem IPv4-Adresse und zugeordenete Netzmaske als `uint32_t` interpretiert werden und somit die herkömmliche 32bit-Integer-Arithmetik zum Einsatz kommt. Für IPv6 würde dieses Konzept sowohl 128bit-Integer-Datentypen als auch 128bit-Integer-Arithmetik erfordern, beides wird aber durch C-Compiler nicht zur Verfügung gestellt, da sie im Standard nicht vorgesehen sind.

Ein möglicher Lösungsansatz ist, die fehlenden Datentypen samt zugehörigen Operationen manuell zu implementieren. Dieses Verfahren ist jedoch unnötig kompliziert, steht doch mit Classless-Inter-Domain-Routing (CIDR [32]) seit langem eine deutlich

kompaktere Repräsentation für Netzmasken zur Verfügung. Statt 32 bzw. 128bit langen (1)*-(0)*-Strings ist in der CIDR-Notation lediglich die Anzahl führender Einsen relevant. So wird aus

IPv4-Netzmaske dezimal	255	255	0	0
IPv4-Netzmaske binär	11111111	11111111	00000000	00000000

die knappe Angabe */16*, wobei der Schrägstrich lediglich zur optischen Trennung der Netzmaske von ihrer Adresse benötigt wird. Es genügt also, einen Integer zwischen 0 und 128 zu speichern, um eine IPv6-Netzmaske zu repräsentieren. Da dieses Konzept auch für IPv4 Verwendung findet (mit Zahlen zwischen 0 und 32), wurde im Zuge der IPv6-Erweiterung von OPAL ausgehend auf CIDR-Notation umgestellt. `struct sockaddr_in if_mask` konnte somit einheitlich durch `uint32_t if_mask` ersetzt werden, allerdings war es erforderlich, zwei neue Hilfsfunktionen für das Konvertieren zwischen CIDR-Notation und Bitmasken zu ergänzen:

```
/* convert a netmask (in network byte order) to CIDR notation */
static int prefix (uint32_t netmask);

/* convert a CIDR prefixlen to netmask (in network byte order) */
uint32_t opal_prefix2netmask (uint32_t prefixlen);
```

`prefix()` wird benötigt, um die vom Betriebssystem bereitgestellte Netzmaske in die interne CIDR-Notation zu überführen, `opal_prefix2netmask()` hingegen dient der Konvertierung einer CIDR-Angabe in einen 32bit-Integer, um ihn dann beispielsweise für oben genannte UND-Verknüpfung von IPv4-Adresse und -Netzmaske zu verwenden.

Der Test, ob sich zwei Adressen im gleichen Netzsegment befinden, wurde im Zuge der IPv6-Erweiterung in eine separate OPAL-Funktion ausgelagert, die somit nicht nur gleichermaßen für IPv4 als auch IPv6 geeignet ist und dadurch eine Fallunterscheidung auf höheren Ebenen entbehrlich macht, sondern auch die bestehende Code-Duplizierung zwischen OOB und BTL beseitigt:

```
bool opal_samenetwork(struct sockaddr_storage *addr1,
                      struct sockaddr_storage *addr2,
                      uint32_t prefixlen);
```

Es fällt auf, daß `opal_samenetwork() struct sockaddr_storage` verwendet, obwohl dieser z.B. auf SUSv2-Systemen nicht zur Verfügung steht und folglich Open MPI nicht kompiliert werden kann. Um eine wiederholte Fallunterscheidung zwischen alter und neuer Socket-API zu vermeiden, bildet `opal/include/opal/ipv6compat.h` fehlende Datentypen auf ihr SUSv2-Äquivalent ab:

```
#ifndef OPAL_IPV6_COMPAT_H
#define OPAL_IPV6_COMPAT_H

#if (!OPAL_WANT_IPV6)
```

```
#ifndef sockaddr_storage
# define sockaddr_storage sockaddr
# define ss_family sa_family
#endif
/* and so on */
```

Damit ist es möglich, stets `struct sockaddr_storage` zu verwenden, indem dieser im Falle fehlender IPv6-Unterstützung des Betriebssystems mit Hilfe des Präprozessors im gesamten Quelltext durch `struct sockaddr` ersetzt wird.

Es bleibt ferner zu erwähnen, daß das Gros aller OPAL-IP-Hilfsfunktionen auf `struct sockaddr_storage` umgestellt wurde, nachdem diese zuvor noch via `OPAL_WANT_IPV6` zwischen `struct sockaddr_in` und `struct sockaddr_in6` unterschieden haben. Diese Differenzierung kann jedoch auf höherer Ebene zu weiteren Fallunterscheidungen führen, wenn bei Aufruf der OPAL-Funktionen wahlweise zu `struct sockaddr_in*` oder `struct sockaddr_in6*` gecastet werden muß. Da alle drei Adressstrukturen in den ersten Bytes das gleiche Layout haben, können sowohl `struct sockaddr_in` als auch `struct sockaddr_in6` einheitlich ohne Anwenden des `OPAL_WANT_IPV6`-Makros zu `struct sockaddr_storage*` konvertiert werden. Es ist daher sinnvoll, für Funktionsparameter ebenfalls `struct sockaddr_storage*` anzugeben und erst innerhalb der OPAL-Prozedur die Adressfamilien zu unterscheiden, wie es bereits in Kapitel 2.2 gezeigt wurde.

4.1.2 Interface-Discovery

Wesentliche Aufgabe von `opal/util/if.c` ist das Finden von IP-fähigen Netzwerkkarten, im Folgenden Interface-Discovery genannt. Sie werden in Form von `opal_if_t` in einer verketteten Liste repräsentiert, die gleichermaßen von OOB und BTL verwendet wird.

Die ursprüngliche Implementierung ermittelt zuerst in einem iterativen heuristischen Verfahren die Anzahl physikalischer IPv4-Interfaces, um den dafür notwendigen Speicherplatz vorhalten zu können. Leider existiert kein einheitlicher ioctl-Request, um diese Aufgabe plattformübergreifend sauber zu lösen. `SIOCGIFNUM` (lies: interface numbers) bzw. `SIOCGIFCOUNT` sind nicht auf allen Legacy-Unixsystemen vorhanden, ferner ist ihre IPv6-Tauglichkeit anzuzweifeln, da sie im Kontext der alten Socket-API einzuordnen sind, deren Datenstrukturen nicht für IPv6 ausgelegt sind.

Der gesamte IP-Code im Open MPI basiert auf der Annahme, daß lediglich eine IP-Adresse pro Netzwerkkarte konfiguriert ist, dies ist aber weder für Alias-Adressen noch für Dual-Stack-Systeme korrekt. Letztere haben z.B. durchaus pro Interface eine IPv4-Adresse, eine IPv6-link-local-Adresse sowie eine IPv6-global-scope-Adresse, so daß intern drei Adressen für dieses Interface repräsentiert werden müßten. Der bereits gezeigte `opal_if_t` enthält aber lediglich einmal `struct sockaddr_in6` und kann somit auch nur eine Adresse aufnehmen.

Die offensichtliche Lösung ist, in der verketteten Liste ein Interface mehrfach aufzuführen und für jede seiner Adressen einen neuen `opal_if_t` anzulegen. Besondere

Beachtung bedurfte dabei das `if_index`-Feld, das mit der Kernel-Interface-Nummer belegt wurde, sofern das Betriebssystem den ioctl `SIOCGIFINDEX` unterstützte. Jede Netzwerkkarte erhält dabei vom Betriebssystem eine eindeutige positive Zahl, anhand der sie später identifiziert werden kann. Für die frühere Annahme *„eine Adresse pro Interface“* war diese Kernel-Interface-Nummer ein tauglicher Index innerhalb der verketteten `opal_if_t`-Liste, bei sich wiederholenden Interfacenummern im Falle mehrerer Adressen pro Karte ist diese Bijektivität jedoch nicht mehr gegeben. Da einige OPAL-Funktionen wie z.B. `opal_ifindextomask()` (liefert die einer Adresse zugeordnete Netzmaske) auf die Eindeutigkeit des Indexes angewiesen sind, wurde `opal_if_t` so erweitert, daß die Kernel-Interface-Nummer in einem neuen Feld `uint16_t if_kernel_index` gespeichert wird und `if_index` lediglich eine Open-MPI-interne fortlaufende Nummer enthält, die mit jeder neuen gefundenen Adresse inkrementiert wird.

Die Interface-Discovery ist hochgradig betriebssystemspezifisch, so daß bis heute kein einheitlicher Standard für das Abfragen der Netzwerkkonfiguration besteht. Die klassische Variante öffnet einen Socket und bemüht im Anschluß mehrfach `ioctl()`, um IP-Adresse, Netzmaske, Flags und ähnliches zu erfragen. Die ioctl-Requests beginnen dabei alle mit `SIOCGIF`, wobei *GIF* für *Get InterFace* steht und nachfolgend genauer spezifiziert wird, welche Konfigurationseinstellung abgefragt werden soll. Typische ioctls sind in diesem Kontext `SIOCGIFADDR` (holt die IP-Adresse), `SIOCGIFNETMASK` (holt die Netzmaske), `SIOCGIFFLAGS` (holt Interface-Flags wie UP, Point-to-Point und andere) oder `SIOCGIFINDEX` (Kernel-Interface-Number).

Den ioctl-Requests zugeordnete Datenstrukturen sind `struct ifconf` und `struct ifreq`, sie sind aber genauso wie alle SIOCGIF-Aufrufe nicht IPv6-fähig. Dieser Umstand findet in der einschlägigen IPv6-Literatur wenig Beachtung, da das Abfragen von Netzwerkkonfigurationen für das Nutzen von IP-Verbindungen selten notwendig ist. Dem HPUX-IPv6-Porting-Guide [27] ist zu entnehmen, daß für IPv6-Interface-Discovery neue, größere Datenstrukturen bereitgestellt werden: `struct lifnum` für die Anzahl der insgesamt konfigurierten Adressen, `struct lifconf` für eine Übersicht der Konfiguration und `struct lifreq` für Detailanfragen. Die zugehörigen ioctl-Requests wurden ebenfalls um den Buchstaben *L* (wie long) erweitert: `SIOCGLIFCONF`, `SIOCGLIFADDR`, `SIOCGLIFINDEX` und andere mehr.

Leider verwenden lediglich Solaris und HP-UX die neuen ioctls, insbesondere werden sie nicht von Linux oder *BSD unterstützt. Unter Linux hätten die Funktionen um `if_nameindex()` die Chance, zusätzliche Adressinformation zu liefern, dies ist aber nicht implementiert und war wahrscheinlich auch niemals vorgesehen. BSD-basierende Systeme kennen zwar teilweise SIOCGLIFCONF & Co (OpenBSD z.B. SIOCGLIFADDR, FreeBSD hingegen nicht), verwenden jedoch `getifaddrs()`. Diese API bietet flexiblen Zugriff auf IPv6- und IPv4-Adressen, repräsentiert die gesamte Konfiguration in einer verketteten Liste und allokiert selbstständig den dafür notwendigen Speicherplatz. Unter Linux steht `getifaddrs()` ab GLIBC-2.4 zur Verfügung, so daß sich hier eine gewisse Vereinheitlichung abzeichnet. Für bestehende Linux-Systeme mit GLIBC-2.3 und früher kann entweder auf die RTNETLINK-Schnittstelle des Kernels zurückgegriffen oder die Textdatei `/proc/net/if_inet6` gelesen werden. Letzteres ist mit wenigen Programmzeilen zu bewerkstelligen, RTNETLINK hingegen erfordert eine zusätzliche umfangreiche

Bibliothek[21] und ist damit für den gegebenen Anwendungsfall als unnötig komplex zu betrachten.

Es bleibt festzustellen, daß IPv4-Adressen mit einigen Kniffen unter unixoiden Betriebssystemen weitestgehend plattformübergreifend in Erfahrung gebracht werden können, für IPv6 eine solche einheitliche Schnittstelle aber nicht existiert. Die neue Interface-Discovery verwendet deshalb nachwievor den ursprünglichen Quellcode für das Auffinden der IPv4-Adressen und benutzt erst im Anschluß je nach Betriebssystem eine der neuen APIs für die IPv6-Konfiguration.

Unterschiede im Funktionsumfang der einzelnen Programmierschnittstellen erfordern dennoch einige Aufmerksamkeit: die zugleich beste und einfachste Implementierung ist das Lesen von `/proc/net/if_inet6` unter Linux. Die Textdatei enthält pro Zeile eine IPv6-Adresse, den zugehörigen Kernel-Index der Netzwerkkarte, die Präfixlänge in CIDR-Notation, den Scope sowie den Interface-Namen. Besonderer Bedeutung wir dabei dem Scope zuteil, er ist null, wenn es sich um Global-Scope-Unicast-Adressen handelt, in allen anderen Fällen enthält das Feld einen von null verschiedenen Wert. Damit können alle nicht-global routbaren Adressen auf einfache Weise aussortiert werden, da weder Link-local- noch Site-local-Adressen für flexible Multicluster-Anwendungen geeignet sind.

Sowohl `getifaddrs()` als auch `SIOCGLIFADDR` liefern zwar ebenfalls einen Scope-Eintrag, dieser reflektiert aber nicht den tatsächlichen Gültigkeitsbereich der Adresse. Diese Diskrepanz ist durchaus zulässig, da die erweiterte Socket-API nach RFC 2460 das korrekte Belegen des Scope-Feldes nicht zwingend vorschreibt, sondern dem Betriebssystem Freiheiten in der Gestaltung einräumt (implementation defined). Es ist daher erforderlich, manuell mit Hilfe einiger Makros ungewünschte Adressen zu verwerfen: `IN6_IS_ADDR_LOOPBACK` erkennt das Loopback-Interface und `IN6_IS_ADDR_LINKLOCAL` findet Link-Local-Adressen (fe80::/10), die folglich einfach übersprungen werden. Dem Kommentar im Quelltext ist zu entnehmen, daß diese Tests nicht ausreichend sind, um Site-local- oder Multicast-Adressen zu filtern. Als möglicher Ausweg bliebe, lediglich alle Adressen innerhalb von 2000::/3 zu akzeptieren. Dies entspricht dem derzeit zugeteilten Global-Unicast-Adressraum, der jedoch in Zukunft erweitert werden könnte und somit eine Änderung im Open-MPI-Quelltext bedingen würde.

Auch wenn eine solche Erweiterungen im Augenblick äußerst unwahrscheinlich ist, so überwiegt die Hoffnung, daß Betriebssystemhersteller dazu übergehen, `sin6_scope_id` korrekt zu belegen und damit einen expliziten Test auf Global-Unicast zu ermöglichen. Der implementierte Workaround mit Hilfe der o.g. Makros ist daher als notdürfigte Übergangslösung zu betrachten, die aber im Rahmen der reell auftretenden Anforderungen als hinreichend eingestuft werden kann (good enough).

Weiteres Manko der `getifaddrs()`- sowie `SIOCGLIF*`-Lösung ist die fehlende Angabe der Präfixlänge, also die Größe des anliegenden Netzwerksegments. Für die Open-MPI-Praxis ist diese Information weitestgehend uninteressant, dient doch die Netzmaske nur für die Frage, ob sich zwei Hosts im gleichen LAN befinden und damit gegenseitig er-

[21] Aus netlink(7): *It is often better to use netlink via libnetlink or libnl than via the low level kernel interface.*

reichbar sind. Globale Erreichbarkeit wird aber ohnehin durch Global-Unicast-Adressen gewährleistet, wenngleich damit ein Lokalitätstest nicht generell ausgeschlossen werden soll. Zu diesem Zweck wird im Falle unbekannter Präfixlänge eine Netzmaske von /64 angenommen, wie sie bei IPv6 für alle LAN-Segmente Verwendung findet. Größere Präfixe (z.B. /48) sind aufgrund des hierarchischen Aufbaus des Adressraums stets mit Routing verbunden, können also folglich gar nicht einem einzelnen, direkt anliegenden physikalischen Netzsegment zugeordnet werden.

Derzeit spielt die IPv6-Präfixlänge innerhalb von Open MPI de facto keine Rolle, dennoch sollte ihr korrekter Wert im Interesse sauberer Programmierung ermittelt werden, sobald dafür eine geeignete Betriebssystemschnittstelle zur Verfügung steht. Höhere Schichten könnten sich dann darauf verlassen, daß die Information in `opal_if_t` gültig ist und ggf. Topologiedaten daraus ableiten.

4.2 Portierung des OOBs

4.2.1 Technische Aspekte der IPv6-Erweiterung

Die in Kapitel 3.2 vorgestellte und auf IPv4-basierende Out-Of-Band-Kommunikation (OOB) ist der Transportkanal des Open Run-Time Environments. Er wird zwischen Headnode-Prozess und allen beteiligten Peers etabliert, um die Ausführung von Programmen zu überwachen, zu unterstützen und im Falle des Austauschs von Adressinformationen sogar erst zu ermöglichen.

Der TCP-Code des OOBs hat viele Gemeinsamkeiten mit dem des BTLs, kommt aber bei der Programmausführung zeitlich eher zum tragen und wurde deshalb zuerst um IPv6-Unterstützung erweitert. Ferner ist ORTE auch ohne BTL lauffähig, die OOB-Modifikation kann also bereits einzeln getestet werden, bevor die gesamte Open-MPI-Bibliothek IPv6-fähig ist.

Ausgangspunkt war auch hierbei wieder pseudo-mechanisches Ersetzen von `struct sockaddr_in` durch `struct sockaddr_in6` unter Verwendung von `OPAL_WANT_IPV6`, um die geforderte SUSv2-Kompatibilität zu erhalten. Der resultierende Code ist zwar zumindest unter Linux lauffähig, unterstützt aber keine IPv6-Verbindungen, da sämtliche Netzwerkfunktionen noch ausschließlich IPv4 beherrschen. Offenkundig wird das Problem bei `socket()`:

```
mca_oob_tcp_component.tcp_listen_sd = socket(AF_INET, SOCK_STREAM, 0);
```

Diese Zeile erzeugt einen Socket, der später für eingehende Verbindungen genutzt werden soll. Die Angabe `AF_INET` signalisiert dabei dem Betriebssystem, daß ausschließlich IPv4-Anfragen bedient werden. Im Falle von IPv4-mapped-IPv6-Adressen wäre es möglich, mittels `AF_INET6` sowohl IPv4 als auch IPv6 zu unterstützen, dies ist aber aufgrund des vorgegebenen Verzichts auf eben diese Adressen nicht möglich.

Die modulare Komponentenarchitektur von Open MPI bietet hier den Ausweg, zwei getrennte Komponenten zu erzeugen: in Ergänzung zur bestehenden tcp-Komponente würde eine eigenständige tcp6-Implementierung die neue Adressklasse bedienen, indem der existierende Code `AF_INET` verwendet, der neue hingegen `AF_INET6`. Diese Variante wurde im Zuge der Diplomarbeit kurzzeitig verfolgt, erwies sich aber als wenig sinnvoll: Beide Komponenten sind sehr ähnlich, so daß große Mengen duplizierten Programmtextes entstehen würden – eine klare Verletzung des Don't-Repeat-Yourself-Prinzips und auch seitens der Open-MPI-Entwickler nicht erwünscht.

Als mögliche Lösung wurde das Instanzieren separater Module einer einzigen tcp-Komponente diskutiert. Dies vermeidet zwar identischen Code, hat aber genauso wie der ursprüngliche Ansatz einen entscheidenden Nachteil: der übergeordnete Runtime-Messaging-Layer (RML) ist zum gegenwärtigen Zeitpunkt nicht in der Lage, Nachrichten über verschiedene OOB-Module oder -Komponenten zu senden[33], wie es im Falle multipler tcp-Instanzen erforderlich wäre. Das Implementieren einer RML-Routing-Tabelle würde zu weitreichenden Änderungen im OpenRTE führen, da bisher weder konzeptionelle Vorarbeit noch konkrete Programmunterstützung zur Lösung dieser Frage existieren und ferner eine solch umfassende Modifikation der Akzeptanz der IPv6-Erweiterung bei anderen Projektteilnehmern wenig dienlich wäre.

Da das Verhältnis zwischen OOB und RML nach Aussage des zuständigen Entwicklers nicht klar definiert[22] ist und Änderungen im Zuge der IPv6-Erweiterung nach Möglichkeit auf die bestehende TCP-Codebasis beschränkt bleiben sollten, scheiden folglich alle Lösungen aus, die mehr als eine OOB-TCP-Instanz verwenden.

Damit ist klar, daß ein einzelnes OOB-TCP-Modul sowohl IPv4 als auch IPv6 beherrschen muß und somit die Frage zum Ausgangspunkt zurückkehrt. Wenn es nicht möglich ist, beide Adressfamilien mit einem Socket zu bedienen, bleibt als offensichtlicher Ausweg, zwei getrennte Sockets für `AF_INET` und `AF_INET6` zu verwenden. Die dafür notwendigen Änderungen am bestehenden Code gliedern sich wie folgt:

1. Erweiterung der OOB-TCP-Komponente um einen zweiten Socket sowie ggf. zusätzlicher socket-spezifischer Datenstrukturen.

2. Parameterisierung aller relevanten OOB-TCP-Funktionen, die bisher mit einem Socket gearbeitet haben, so daß sie gleichermaßen den neuen Socket bedienen können.

3. Erweiterung des Event-Handlings, um auch auf Ereignisse des zweiten Sockets zu reagieren.

Beginnend mit `orte/mca/oob/tcp/oob_tcp.h` wird der darin deklarierte `struct mca_oob_tcp_component_t` um zwei zusätzliche Einträge ergänzt: `tcp6_listen_sd` als Listen-Socket für eingehende IPv6-Verbindungen sowie `tcp6_listen_port` für die zugehörige IPv6-Portnummer, da auch sie sich üblicherweise von der IPv4-Portnummer unterscheidet:

```
/**
 *  OOB TCP Component
*/
struct mca_oob_tcp_component_t {
    [..]
    int                tcp_listen_sd;
    unsigned short     tcp_listen_port;
#if OPAL_WANT_IPV6
    int                tcp6_listen_sd;
    unsigned short     tcp6_listen_port;
#endif
}
```

Analog zum bestehenden `tcp_listen_sd` wird der neue Socket initialisiert

```
    /* initialize state */
    mca_oob_tcp_component.tcp_shutdown = false;
    mca_oob_tcp_component.tcp_listen_sd = -1;
#if OPAL_WANT_IPV6
    mca_oob_tcp_component.tcp6_listen_sd = -1;
#endif
```

[22] Aus [33]: *You know, we never did much of a communications layer design for OpenRTE.*

```
if (AF_INET6 == af_family) {
    int flg = 0;
    if (setsockopt (*target_sd, IPPROTO_IPV6, IPV6_V6ONLY,
        &flg, sizeof (flg)) < 0) {
        opal_output(0, "unable to disable v4-mapped addresses\n");
    }
}
```

Abbildung 5: IPv4-mapped-IPv6-Adressen sind nach Implementierungsvorgaben unzulässig, ihre Verwendung muß daher explizit ausgeschlossen werden.

und im Anschluß konfiguriert:

```
    /* create a listen socket */
    mca_oob_tcp_create_listen(&mca_oob_tcp_component.tcp_listen_sd, AF_INET);
#if OPAL_WANT_IPV6
    mca_oob_tcp_create_listen(&mca_oob_tcp_component.tcp6_listen_sd, AF_INET6);
```

Man erkennt hierbei, daß die Funktion `mca_oob_tcp_create_listen()` die im zweiten Schritt benötigte Parameterisierung bereits erfahren hat: der erste Parameter beschreibt den Socket, auf dem operiert werden soll, der zweite spezifiziert die Adressfamilie, die für diesen Socket zu konfigurieren ist. Innerhalb von

```
/*
 * Create a listen socket and bind to all interfaces
 */
static int mca_oob_tcp_create_listen(int *target_sd, uint16_t af_family)
```

passiert wenig Spektakuläres, zwei Dinge sind jedoch erwähnenswert: da `mca_oob_tcp_create_listen()` einen Socket an alle Netzwerkkarten bindet, indem für `bind()` jede beliebige Adresse erlaubt wird, müssen IPv4-mapped-IPv6-Adressen auf dem IPv6-Socket explizit deaktiviert werden. Der Code in Abbildung 5 leistet dies und illustriert ferner exemplarisch die generelle Vorgehensweise zur Implementierung adressfamilienunabhängiger Funktionen.

Zweite Besonderheit ist der in Abbildung 6 dargestellte Programmtext. Er steht stellvertretend für den dritten Teil der Erweiterung, die Änderung des Event-Handlings. Im gesamten Open-MPI-Projekt werden ereignisgesteuerte Callback-Funktionen genutzt, um nicht blockierend auf das Eintreten eines Zustands warten zu müssen und somit bereits mit anderen Aufgaben fortfahren zu können, solange beispielsweise das Betriebssystem noch mit dem Verbindungsaufbau beschäftigt ist. Auch die Arbeit mit mehreren parallelen Verbindungen wird auf diese Weise realisiert, indem bei Aktivität eines Sockets die OPAL-Event-Library die zugeordneten Event-Handler-Funktion aufruft.

Der gezeigte Code definiert sowohl für IPv4 als auch IPv6 gleichermaßen `mca_oob_tcp_recv_handler()` als gemeinsamen Callback für Ereignisse des jeweiligen Sockets,

```
#if OPAL_WANT_IPV6
    if (AF_INET == af_family) {
        opal_event_set(
            &mca_oob_tcp_component.tcp_recv_event,
            *target_sd,
            OPAL_EV_READ|OPAL_EV_PERSIST,
            mca_oob_tcp_recv_handler,
            0);
    opal_event_add(&mca_oob_tcp_component.tcp_recv_event, 0);
    }

    if (AF_INET6 == af_family) {
        opal_event_set(
            &mca_oob_tcp_component.tcp6_recv_event,
            *target_sd,
            OPAL_EV_READ|OPAL_EV_PERSIST,
            mca_oob_tcp_recv_handler,
            0);
        opal_event_add(&mca_oob_tcp_component.tcp6_recv_event, 0);
    }
#else
    opal_event_set(
        &mca_oob_tcp_component.tcp_recv_event,
        *target_sd,
        OPAL_EV_READ|OPAL_EV_PERSIST,
        mca_oob_tcp_recv_handler,
        0);
    opal_event_add(&mca_oob_tcp_component.tcp_recv_event, 0);
#endif
```

Abbildung 6: Auszug aus `mca_oob_tcp_create_listen()`: Erweiterung des Event-Handlings für Ereignisse auf dem zusätzlichen IPv6-Socket.

```
    struct mca_oob_tcp_component_t {
    [..]
    opal_event_t       tcp_send_event; /* event structure for IPv4 sends */
    opal_event_t       tcp_recv_event; /* event structure for IPv4 recvs */
#if OPAL_WANT_IPV6
    opal_event_t       tcp6_send_event; /* event structure for IPv6 sends */
    opal_event_t       tcp6_recv_event; /* event structure for IPv6 recvs */
#endif
    [..]
    }
```

Abbildung 7: Erweiterung der OOB-TCP-Komponente um Strukturen für Ereignisse des IPv6-Sockets.

allerdings ist dafür die in Abbildung 7 dargestellte zusätzliche Modifikation von `struct mca_oob_tcp_component_t` erforderlich.

Die OPAL-Event-Library übermittelt bei einem Callback neben einigen internen Daten auch den Socket, auf dem das Ereignis eingetreten ist. Diese Information ist für nachfolgende Funktionen äußerst nützlich, da sie fortan nicht mehr explizit zwischen IPv4- und IPv6-Socket unterscheiden müssen sondern ihnen mittels Parameter der zu verwendende Socket direkt übergeben werden kann. Es ist offensichtlich, daß die in Schritt 2 skizzierte Parameterisierung fest einprogrammierten Angaben wie `mca_oob_tcp_component.tcp_listen_sd` vorzuziehen ist.

Die so erweiterte OOB-TCP-Komponente ist damit nun prinzipiell in der Lage, sowohl IPv4- als auch IPv6-Verbindungen entgegen zu nehmen. Letztere können jedoch nicht genutzt werden, da bislang die Peers nicht über den IPv6-Listen-Socket in Kenntnis gesetzt werden. Mit Hilfe der in Anhang B.1 dargestellten Connection-Strings übergibt der Headnode-Prozess beim Starten entfernter Orte-Daemons seine lokalen Adressen sowie die Portnummer des Listen-Sockets als Kommandozeilenparameter. Der Aufbau dieser Connection-Strings ist stets gleich:

tcp://IPv4-Adresse:Portnummer;

Dem Präfix *tcp://* folgt die IPv4-Adresse, die durch einen Doppelpunkt von der Portnummer getrennt ist. Offenkundig ist es notwendig, dieses Konzept um IPv6-Fähigkeiten zu erweitern. Dafür existieren prinzipiell drei unterschiedliche Syntax-Varianten:

1. tcp://[IPv6-Adresse]:Portnummer
2. tcp6://IPv6-Adresse:Portnummer
3. tcp://IPv6-Adresse:Portnummer

Die erste Fassung entspricht der generischen Vorgehensweise in anderen IPv6-fähigen Anwendungen wie Mailservern oder Webbrowsern, die zweite ist eher an netstat angelehnt und die dritte ist üblicherweise nicht anzutreffen sondern lediglich eine theoretische

1. erzeuge IPv4-Socket
2. erzeuge IPv6-Socket
3. solange keine Verbindung existiert, wiederhole:
    a) hole neue Zieladresse
    b) wenn Zieladresse $\in$ `AF_INET`, dann verbinde via IPv4-Socket, ansonsten via IPv6-Socket
4. schließe den Socket, der nicht für die Verbindung benötigt wird

Abbildung 8: Pseudo-Algorithmus für den Callback: Da `connect()` einen zur Adresse passend konfigurierten Socket benötigt, müssen sowohl IPv6- als auch IPv4-Sockets vorgehalten werden,

Überlegung; sie erfordert, daß der Parser für Connection-Strings den Adresstyp selbst ermittelt, indem er beispielsweise nach (Doppel-)Punkten sucht und damit IPv4- von IPv6-Adressen unterscheidet.

Da es sich bei dem bereits vorliegenden Parser um eine sehr knappe aber dennoch ausreichende Implementierung handelt, würde dieser dritte Ansatz die Komplexität unnötig erhöhen. Nicht nur das, analog zur ersten Variante könnten nicht-IPv6-fähige oder alte orte-Daemons Schwierigkeiten mit IPv6-Connection-Strings bekommen, da sie zwar das gültige *tcp://*-Präfix erkennen, danach aber im Falle einer IPv6-Adresse nicht die von ihnen erwartete IPv4-Adresse vorfinden würden.

Es ist also sinnvoll, IPv6-Adressen mit einem neuen Präfix einzuleiten, das nur von IPv6-fähigen Open-MPI-Installation verstanden wird. Anhang B.2 zeigt das Resultat: alle IPv6-Adressen der OPAL-Interface-Liste beginnen mit *tcp6://*, IPv4-Adressen nachwievor mit *tcp://*. In beiden Fällen trennt der am weitesten rechts stehende Doppelpunkt die Adresse von der Portnummer, so daß auch dieser Aspekt keine Unterscheidung zwischen IPv4 und IPv6 erfordert und somit die Parser-Komplexität gering bleibt. Offenkundig unterscheiden sich die Portnummern für IPv4 und IPv6, sie repräsentieren die separaten Listen-Sockets des Headnode-Prozesses.

Ein entfernter orted wandelt nun mittels `mca_oob_tcp_parse_uri()` den Connection-String in einzelne `struct sockaddr_in6`, um sie später für den `connect()`-Verbindungsaufbau verwenden zu können. Auch hierbei entstehen durch die IPv6-Erweiterung zwei Besonderheiten: die ursprüngliche Implementierung erzeugt zuerst einen Socket, der später für ausgehende Verbindungen genutzt werden soll. Analog zur Diskussion um Listen-Sockets ist es wiederum notwendig, im Vorfeld einen zweiten Socket für IPv6-Verbindungen anzulegen und dann je nach Zieladresse den IPv4- bzw. IPv6-Socket zu verwenden. Der in Abbildung 8 skizzierte Algorithmus verdeutlicht das Problem: `connect()` erwartet, daß die übergebenen Parameter Verbindungssocket und Adress-Struct die gleiche Adressfamilie aufweisen. Da jedoch alle möglichen Zieladressen se-

quentiell innerhalb einer Schleife abgearbeitet werden, können sowohl `AF_INET`- als auch `AF_INET6`-Einträge auftreten, in deren Abhängigkeit der jeweils passende Socket zu wählen ist. Die Alternative, den Socket ebenfalls erst bei Bedarf passend zu erzeugen, wurde nicht weiter verfolgt, da die bestehende Implementierung exzessiv von Timer- und Send-Events Gebrauch macht, die mitunter bereits vor Verbindungsaufbau an den geöffneten Socket gebunden werden, um beispielsweise Nachrichten an einen Peer in einer Warteschlange aufnehmen zu können und sie dann erst bei bestehender Verbindung auszuliefern. Analog zu den Listen-Sockets wurde auch hierfür das Event-Handling auf zwei Sockets erweitert.

4.2.2 Adressauswahl in Multi-Transport-Single-Cell-Clustern

Ein wesentlicher Aspekt des aktiven Verbindungsaufbaus ist die Reihenfolge, in der die Adressen des Headnodes kontaktiert werden sollen.

Verfügt ein Headnode über mehr als eine Adresse, so enthält der von ihm erzeugte Connection-String ebenfalls mehrere Adressen, die mit Hilfe des Process-Launching-Systems als Kommandozeilenparameter für *orted* an die Peers übergeben werden. Der Parser der OOB-TCP-Komponente erzeugt daraus intern ein Adress-Array, dessen Einträge beim Verbindungsaufbau durch die Funktion `mca_oob_tcp_addr_get_next()` in einem Round-Robin-Verfahren ausgelesen werden, bis eine Verbindung erfolgreich etabliert werden konnte.

Für Single-Cell-Multicluster existiert per Definition ein gemeinsamer Transport, der alle Einzelcluster überspannt. Ein Headnode-Prozess kann nicht im Vorfeld entscheiden, welche seiner Adressen die beteiligten Peers im Callback-Verfahren kontaktieren, so daß er notwendigerweise alle verfügbaren Listen-Adressen exportiert. Der bestehende Ansatz, diese bis zum erfolgreichen Verbindungsaufbau im Round-Robin-Verfahren zu kontaktieren, führt im Multicluster-Einsatz zu Problemen: Enthält der Connection-String private IPv4-Adressen, kann ein Peer nicht zweifelsfrei entscheiden, ob es sich dabei um

- einen für ihn erreichbaren Rechner im gleichen Netz,
- einen durch Routing erreichbaren Rechner in einem anderen Netz oder
- um einen *nicht* erreichbaren Rechner in einem anderen Cluster

handelt. Aus eigener und entfernter RFC 1918-Adresse läßt sich auch mit Hilfe der Netzmaske nicht ableiten, daß der Headnode-Prozess im lokalen Netz steht: Abbildung 2 zeigt einen Multicluster-Aufbau mit identischer IPv4-Konfiguration sowie clusterübergreifender IPv6-Versorgung. Der Connection-String wird in diesem Falle private IPv4-Adressen enthalten, die für Peers innerhalb des einen Clusters eine erreichbare Adresse darstellen, für Teilnehmer des anderen Clusters aber nur scheinbar im gleichen Netz liegen. Beim Versuch, diese zu kontaktieren, wäre der Headnode-Prozess im besten Falle unerreichbar, so daß nach Ablauf eines Timeouts die nächste Adresse zum Zuge käme. Offenkundig verzögert dies den Verbindungsaufbau je nach Menge der zu probierenden Adressen, das weitaus größere Problem stellt jedoch der Versuch selbst dar: ist die gleiche

RFC 1918-Adresse sowohl im lokalen als auch im entfernten Netz vergeben, repräsentiert sie zwei unterschiedliche Rechner in verschiedenen Netzen. Während ein Verbindungsversuch innerhalb des einen Clusters erfolgreich sein wird, erreicht er im anderen Falle den falschen Rechner. Dort laufende Anwendungen könnten durch die fehlgeleitete ORTE-Kommunikation gefährdet werden, ein Verhalten, das bereits durch das in RFC 793 [6] erwähnte Robustheitsprinzip verurteilt wird und im Interesse der Netzstabilität unbedingt zu vermeiden ist: *Be conservative in what you send, be liberal in what you accept.*

Einer privaten IPv4-Adresse ist also nicht anzusehen, ob sie zielführend ist. Öffentliche IPv4-Adressen sowie IPv6-Global-Scope-Adressen stellen kein grundlegendes Problem dar, da sie einen weltweit eindeutigen Adressraum beschreiben und damit Kollisionen generell ausschließen.

Es ist deshalb Aufgabe der Funktion `mca_oob_tcp_addr_get_next()`, aus der Menge der möglichen Adressen diejenige auszuwählen, die am wahrscheinlichsten zu einem erfolgreichen Verbindungsaufbau führen wird. Insbesondere ist klar, daß RFC 1918-Adressen lediglich als letzte Alternative gewählt werden dürfen, um o.g. Probleme nach Möglichkeit zu vermeiden. Gänzlich ausschließen kann man private IPv4-Adressen nicht, da sie nachwievor bei einer Vielzahl von Single-Clustern zum Einsatz kommen und die Open-MPI-Bibliothek auch diesen Fall korrekt behandeln muß.

Wesentliche Anforderung an einen Auswahlalgorithmus ist daher seine universelle Verwendbarkeit: Er sollte gleichermaßen Multi- und Single-Cluster-Installationen beherrschen, in den verschiedensten Netzwerkszenarien nicht-zielführende Verbindungsversuche idealerweise vermeiden und dabei im Interesse der Rückwärtskompatibilität nicht zwingend IPv6 benötigen.

Beurteilt man IP-Adressen qualitativ, so ergeben sich die drei hierfür relevanten Äquivalenzklassen:

1. IPv6-Global-Scope: weltweite Eindeutigkeit.
2. IPv4-public: weltweite Eindeutigkeit.
3. IPv4-private: keine Eindeutigkeit.

Der gewünschte Auswahlalgorithmus folgt auf natürliche Weise:

1. Sortiere die Liste möglicher Zieladressen absteigend nach ihrer Qualität:
   a) öffentliche Adressklassen: IPv6, IPv4-public (Reihenfolge beliebig)
   b) IPv4-private
2. Entferne alle Adressen, für deren Äquivalenzklasse lokal kein Interface konfiguriert ist.
3. Kontaktiere die verbleibenden Adressen in der resultierenden Reihenfolge.

Anhang B.3 illustriert die Arbeitsweise des Algorithmus an einem Beispiel. Es fällt auf, daß eine Zieladresse nur dann Verwendung findet, wenn der Peer ebenfalls über eine gleichwertige Adresse verfügt. Für IPv6 ist dies einsichtig, da von einer konfigurierten Global-Scope-Adresse jede andere erreichbar sein sollte, hingegen ohne IPv6-Deployment das Betriebssystem ohnehin *No route to host* liefern würde und diese Zieladresse folglich gar nicht erst probiert werden muß. Gleiche Argumentation gilt für homogene IPv4-Strukturen und schlußendlich auch für reine IPv4-private-Umgebungen, da sich alle beteiligten Adressen innerhalb einer Äquivalenzklasse befinden. Im inhomogen Fall priorisiert der Algorithmus unkritische, da öffentliche, Adressklassen und nutzt gegebene RFC 1918-Adressen nur als letzten Ausweg.

Zu beachten ist, daß ein Peer mit ausschließlich privaten IPv4-Adressen niemals eine öffentliche IPv4-Adresse kontaktieren wird, da deren Erreichbarkeit nicht gesichert ist: die Annahme, *jeder Peer* habe *stets* mittels NAT oder einer anderen Technologie semidirekten Zugang zum öffentlichen Internet, ist unzulässig. Der Algorithmus muß also eine solche Trans-Äquivalenzklassen-Verbindung untersagen.

Im Rahmen dieser Diplomarbeit wurde lediglich ein vereinfachtes Verfahren implementiert:

1. Bevorzuge IPv6 gegenüber IPv4-public
2. Bevorzuge IPv4-public gegenüber IPv4-private

Dieser Algorithmus arbeitet für eine experimentelle IPv6-Erweiterung zufriedenstellend, insbesondere da im Versuchsaufbau alle Knoten mit IPv6 ausgestattet waren und somit aufgrund der Vorrangregel problemlos eine Verbindung hergestellt werden konnte. Das Verfahren scheitert, wenn es in gemischten IPv4-public-private-Umgebungen ohne IPv6 zum Einsatz kommt, da es o.g. Sonderfall nicht berücksichtigt: ein Compute-Node ohne NAT wird z.B. die öffentliche IPv4-Adresse des Headnodes nicht erreichen können und erst nach Ablauf eines Timeouts dessen interne, private IPv4-Adresse kontaktieren.

4.3 Portierung des Byte-Transfer-Layers

4.3.1 Grundlegende Maßnahmen der IPv6-Erweiterung

Der Byte-Transfer-Layer (BTL) ist das Transport-Framework der Open-MPI-Schicht und dient dem Austausch von MPI-Nachrichten. Im Gegensatz zum OOB, das lediglich schmalbandige Metakommunikation bereitstellt, implementieren die verschiedenen BTL-Komponenten den eigentlichen, performance-kritischen Nachrichtentransfer der MPI-Anwendung, der je nach verfügbarer Netzwerk-Hardware z.B. über Infiniband, Shared Memory oder TCP erfolgen kann.

Die Erweiterung der BTL/TCP-Komponente um IPv6-Unterstützung verlief weitestgehend analog zum OOB, nicht zuletzt aufgrund der großen Ähnlichkeit beider Programmteile: Im ersten Schritt wurde `struct sockaddr_in` durch `struct sockaddr_storage` ersetzt, um alle internen Datenstrukturen für 128bit-Adressen vorzubereiten. Wie beim OOB ist dies nicht ausreichend, um IPv6-Verbindungen bedienen zu können, so daß auch hier ein zweiter Listen-Socket erforderlich war. Gleichzeitig wurde das Event-Handling erweitert, um auf Ereignisse des neuen Sockets zu reagieren, nachdem alle relevanten Funktionen auf Socket- bzw. Adressfamilienparameter umgestellt wurden.

Erste Probleme entstanden durch den Wechsel von `uint32_t` zu `struct sockaddr_storage` im Zusammenhang mit den in Kapitel 3.3.2 beschriebenen Funktionen `mca_pml_base_modex_send()` und `mca_pml_base_modex_recv()` zum Austausch von Adressinformation zwischen BTL und ORTEs General-Purpose-Registry: während die ursprüngliche Implementierung lediglich einen 32bit-Integer übertragen hat, wurde nun ein üblicherweise 128 Byte langer C-Struct in der zentralen Registry hinterlegt. Zwar garantiert ORTEs Data-Switch-System (DSS) für atomare Datentypen stets das für den Peer korrekte Byte-Ordering, allerdings handelt es sich bei `struct sockaddr_storage` um ein betriebssystemspezifisches Speicherlayout, das nicht sinnvoll zwischen verschiedenen Plattformen ausgetauscht werden kann: alle Offsets des Structs fallen je nach Betriebssystem und hardwarespezifischem Alignment potenziell unterschiedlich aus, so daß beispielsweise die Port-Nummer unter x86-Solaris an einer anderen Stelle als unter x86-Linux oder ppc64-OSX zu finden ist. Ferner sind alle Konstanten wie `AF_INET` oder `AF_INET6` ebenfalls betriebssystemspezifisch, so daß das Auswerten von `ss_family` im heterogenen Einsatz ausscheidet.

Offenkundig ist es notwendig, `struct sockaddr_storage` zu serialisieren und in Form atomarer Datentypen in der Registry zu hinterlegen. Abbildung 9 zeigt den BTL/TCP-internen Datentyp `struct mca_btl_tcp_addr_t`, der für den Austausch von Adressinformationen zwischen den Peers verwendet wird.

Steht die neue Socket-API nach RFC 3493 zur Verfügung, bietet `struct in6_addr` auf allen Betriebssystemen zusammenhängend, d.h. ohne Padding, 128 Bit, die sowohl eine IPv4- als auch eine IPv6-Adresse aufnehmen können. Auf SUSv2-Systemen ohne `struct in6_addr` werden ersatzweise vier 32bit-Integer allokiert, um die geforderte Rückwärtskompatibilität zu erhalten. Zwar werden in Ermangelung jeglicher IPv6-Funktionalität in diesen Fällen lediglich die ersten 32 Bit als IPv4-Adresse interpretiert, die so erweiterten Installationen sind aber dennoch zu IPv6-fähigen ORTE-Umgebungen kompatibel,

```
/* Structure used to publish TCP connection information to peers.  */

struct mca_btl_tcp_addr_t {
    /* the following information is exchanged between different
       machines (read: byte order), so use network byte order
       for everything and don't add padding
    */
#if OPAL_WANT_IPV6
    struct in6_addr addr_inet;    /**< IPv4/IPv6 listen address > */
#else
    struct my_in6_addr {
        union {
            uint32_t u6_addr32[4];
            struct _my_in6_addr {
                struct in_addr _addr_inet;
                uint32_t _pad[3];
            } _addr__inet;
        } _union_inet;
    } addr_inet;
#endif
    in_port_t      addr_port;     /**< listen port */
    unsigned short addr_inuse;    /**< local meaning only */
    uint8_t        addr_family;   /**< AF_INET or AF_INET6 */
};
```

Abbildung 9: Interne Datenstruktur zum Repräsentieren einer IPv4/6-Adresse sowie zugehöriger Portnummer und Adressfamilie. `struct sockaddr_storage` bietet zwar ähnliche Funktionalität, ist aber nicht plattformübergreifend binärkompatibel und muß daher serialisiert werden.

da sie den notwendigen Speicherplatz für die 128 Bit langen Antworten der General-Purpose-Registry bieten.

Das `addr_family`-Feld unterscheidet zwischen IPv4 und IPv6, kann jedoch aufgrund fehlender plattformübergreifender Einheitlichkeit nicht mit `AF_INET` oder `AF_INET6` belegt werden und enthält daher die stets gleichen, BTL-eigenen Konstanten `MCA_BTL_TCP_AF_INET` bzw. `MCA_BTL_TCP_AF_INET6`:

```
#define MCA_BTL_TCP_AF_INET     0
#if OPAL_WANT_IPV6
# define MCA_BTL_TCP_AF_INET6   1
#endif
```

Im Gegensatz zum OOB benötigt die BTL/TCP-Komponente für ausgehende Verbindungen keine separaten IPv4- und IPv6-Sockets: bereits die ursprüngliche Implementierung bietet für jede Zieladresse einen eigenen Socket, der lediglich mit der zur Zieladresse passenden Adressfamilie geöffnet werden muß:

```
    if (AF_INET == btl_endpoint->endpoint_addr->addr_family) {
        af_family = AF_INET;
        addrlen = sizeof (struct sockaddr_in);
    }
#if OPAL_WANT_IPV6
    if (AF_INET6 == btl_endpoint->endpoint_addr->addr_family) {
        af_family = AF_INET6;
        addrlen = sizeof (struct sockaddr_in6);
    }
#endif

    btl_endpoint->endpoint_sd = socket(af_family, SOCK_STREAM, 0);
```

Mit Hilfe des in Abbildung 9 gezeigten `addr_inuse`-Eintrages wird geprüft, ob bereits eine Verbindung zur angegebenen Zieladresse besteht. Ist dies nicht der Fall, verwendet `connect()` den in `btl_endpoint->endpoint_sd` angegebenen Socket, um eine neue ausgehende Verbindung zu öffnen.

4.3.2 Oversubscription

In der ursprünglichen Implementierung repräsentierte eine Zieladresse auf eineindeutige Weise ein Netzwerk-Interface. War sie für einen Peer erreichbar aber nicht in Verwendung, wurde eine Verbindung geöffnet und damit das bisher vom Peer unbenutzte Remote-Interface belegt. `addr_inuse` verhinderte den Aufbau weiterer Verbindungen zu dieser Adresse und somit gleichbedeutend zu der von ihr repräsentierten Netzwerkkarte, um die verfügbare Bandbreite exklusiv einer einzelnen Verbindung zuzuordnen.

Bereits Kapitel 4.1.2 hat gezeigt, daß im Falle von Dual-Stack-Konfiguration sowie Alias-Adressen mehrere unterschiedliche Adressen pro Interface vergeben sein können

und folglich keine Bijektivität zwischen NIC und Adresse besteht. Das `addr_inuse`-Datum zeigt somit lediglich die Belegung einer einzelnen Zieladresse an, Verbindungen zu anderen Adressen auf diesem Interface sind aber weiterhin möglich. Sind parallele Verbindungen für die Kommunikation in breitbandigen Weitverkehrsnetzen (Long Fat Pipe Networks) je nach eingesetzter TCP-Implementierung ein probates Mittel zur Erhöhung des Durchsatzes [34], führen sie im LAN aufgrund des steigenden Gesamtoverheads zu einer suboptimaler Nutzung des Übertragungsmediums.

Das Problem verschärft sich noch, da bisher für jeden Eintrag in der OPAL-Interface-Liste ein eigenes BTL/TCP-Modul instanziert wurde. In der ursprünglichen Implementierung limitierte dies die Anzahl möglicher Verbindungen auf die Anzahl der verfügbaren NICs, da die Liste pro Interface nur eine Adresse beinhaltete. Mit der Erweiterung um IPv6 stehen sehr wahrscheinlich deutlich mehr Adressen als verschiedene Interfaces in der OPAL-Interface-Liste, so daß auch weitaus mehr BTL/TCP-Instanzen erzeugt werden.

Verbindungen zu einem Peer bestehen immer zwischen einem BTL/TCP-Modul und einer Zieladresse. Je mehr lokale Instanzen verfügbar sind, desto mehr Zieladressen können kontaktiert werden. Folglich führt eine hohe Anzahl lokaler Adressen zur Nutzung vieler Remote-Adressen und damit zumindest im LAN sehr wahrscheinlich zu ineffizienten Parallelverbindungen auf dem gleichen Transportmedium. Dieses Problem heißt Oversubscription (Überbuchung), ein Verhalten, das mit der alten Implementierung nicht auftreten konnte und dem daher im Zuge der IPv6-Erweiterung entgegnet werden muß.

Offenkundig ist es sinnvoll, die Anzahl der BTL/TCP-Instanzen wie in der Vergangenheit auf die Zahl der lokal verfügbaren Netzwerkkarten zu limitieren statt für jede Adresse ein eigenes Modul zu erzeugen. Die OPAL-Interface-Liste enthält bereits mit `uint16_t if_kernel_index` den Kernel-Index einer Adresse und bietet somit die relevante Information, welcher Netzwerkkarte sie zugeordnet ist. Die Anzahl unterschiedlicher Kernel-Index-Einträge entspricht dabei der Anzahl der verfügbaren Interfaces und somit der Anzahl zu erzeugender BTL/TCP-Instanzen. Die dafür notwendigen Modifikationen der Implementierung beschränken sich im Wesentlichen auf das Zählen der paarweise verschiedenen Kernel-Indexe innerhalb der OPAL-Interface-Liste sowie einer Ergänzung in `mca_btl_tcp_component_exchange()`: ist ein Interface mittels `btl_tcp_if_exclude`-Parameter zur Laufzeit deaktiviert, darf keine der ihm zugeordneten Adressen in der General-Purpose-Registry hinterlegt werden, um Verbindungsversuche anderer Peers an dieses Interface zu verhindern.

Die getroffenen Maßnahmen bewirken, lokal die Anzahl paralleler Verbindung zu einem Peer auf die Anzahl verfügbarer Netzwerk-Interfaces zu beschränken, allerdings ist es nachwievor möglich, daß mehrere dieser Verbindung auf dem selben Interface des Remote-Hosts enden und somit doch wieder durchsatzsenkende Parallelverbindung auf einem einzelnen Transportmedium auftreten. Dies ist immer dann der Fall, wenn ein entfernter Peer weniger Interfaces als der lokale Host aufweist und folglich dessen Anzahl verfügbarer NICs die Anzahl gleichzeitiger Verbindungen limitieren müßte. Eine Implementierung sollte also höchstens

min(Anzahl_lokaler_NICs, Anzahl_entfernter_NICs)

```
addrs[current_addr].addr_ifkindex = opal_ifindextokindex (index);
```

Abbildung 10: Zu jeder IP-Adresse wird ihr zugehöriger Kernel-Index exportiert. Die neue OPAL-Funktion `opal_ifindextokindex()` sucht dazu in der OPAL-Interface-Liste den OPAL-internen Index der Adresse (`index`) und gibt den passenden Kernel-Index zurück.

parallele Verbindungen zu einem Peer öffnen, um einerseits alle verfügbaren physischen Verbindungswege für Message-Striping zu nutzen, andererseits aber suboptimale Mehrfachbelegung des gleichen Transportmediums zu vermeiden. Insbesondere ist es nicht sinnvoll, parallele Verbindungen zu unterschiedlichen Zieladressen eines Peers aufzubauen, wenn sich diese auf dem gleichen Remote-Interface befinden.

Die Frage, welche Zieladressen sich ein physisches Interface teilen, ist jedoch mit der bestehenden Implementierung nicht zu beantworten, da eine IP-Adresse nicht offenbart, auf welchem Interface sie konfiguriert ist. Exportiert ein Peer neben seinen Adressen auch das jeweils zugeordnete Interface, kann Remote-Oversubscription vermieden werden.

Es ist nicht erforderlich, den vollständigen Interface-Namen zu kennen, es genügt ein numerischer Schlüssel, wie ihn der Kernel-Index zur Verfügung stellt, um entfernte NICs voneinander zu unterscheiden. Dazu wurde der in Abbildung 9 gezeigte Datentyp um `uint16_t addr_ifkindex` erweitert, so daß neben Portnummer und IP-Adresse nun auch der jeweilige Kernel-Index in der General-Purpose-Registry hinterlegt werden kann. Die dafür notwendige Änderung in `mca_btl_tcp_component_exchange` beschränkt sich auf eine zusätzliche Zeile, wie Abbildung 10 zeigt.

Es bleibt zu erwähnen, daß die im Rahmen dieser Diplomarbeit entstandene IPv6-Erweiterung den Remote-Kernel-Index nicht auswertet, wohl aber alle notwendigen Mechanismen implementiert wurden, ihn einer zukünftigen Verwendung zuzuführen.

Im Zusammenhang mit dem in Kapitel 4.2.2 dargestellten Algorithmus zur Adressauswahl wird es erforderlich sein, passende Zieladressen in Abhängigkeit der bereits verwendeten lokalen sowie entfernten Netzwerk-Interfaces zu bestimmen. Diese Aufgabe ist im Gegensatz zum OOB erheblich schwieriger, da im Interesse der Performance maximal viele parallele Verbindungen zu öffnen sind, ohne dabei Oversubscription zu provozieren.

Eine zukünftige BTL/TCP-Implementierung kann sich ferner vom Konzept multipler Instanzen abwenden und alle Verbindungen innerhalb eines Moduls bedienen, muß dann aber das vormals durch den PML bewerkstelligte Message-Striping in eigener Zuständigkeit betreiben, da in diesem Falle Nachrichten nicht mehr durch den PML über mehrere Instanzen verteilt werden können und somit einer einzigen Instanz zugeführt werden.

Ferner wird derzeit die Erweiterung des BTLs um Multicast-Fähigkeiten diskutiert, wenngleich dies eine Einordnung unterhalb des Point-to-Point-Layers fragwürdig erscheinen läßt. Ein zusätzlicher Point-to-Multipoint-Layer könnte jedoch spezifische Multicast-Funktionen einer BTL-Komponente aufrufen, so daß kollektive Operationen von der erweiterten Funktionalität des BTLs profitieren. Eine solche Erweiterung wird ganz sicher auch die BTL/TCP-Komponente betreffen, da insbesondere IPv6 über ausgeprägte Multicast-Fähigkeiten verfügt. Inwieweit gemischte IPv4-IPv6-Multicast-Kommunikation

benötigt wird, bleibt abzuwarten, generell unmöglich ist sie jedoch nicht.

4.4 Hostfiles

Das Open-Runtime-Environment enthält im Resource-Discovery-System einen flex-basierenden Parser zum Einlesen des Hostfiles. Der Inhalt des Hostfiles bestimmt, wieviele Prozesse auf welchen Maschinen durch orterun zu starten sind. Im Gegensatz zum Kommandozeilenparameter `-host` ist es nicht nur möglich, die Anzahl der Prozesse pro Host exakt zu bestimmen sondern auch unterschiedliche Login-Namen für verschiedene Rechner zu verwenden.

Der Code in `orte/mca/rds/hostfile` akzeptiert dabei neben Hostnames auch IPv4-Adressen in der üblichen Dotted-Decimal-Darstellung, so daß hier offenkundig eine Erweiterung um IPv6-Fähigkeiten angezeigt ist.

In Anlehnung an `ORTE_RDS_HOSTFILE_IPV4` wurde dazu eine neue Konstante `ORTE_RDS_HOSTFILE_IPV6` eingeführt und die Behandlung des IPv4-Falles auf IPv6 ausgedehnt: Wannimmer IPv4 akzeptiert wird, ist auch IPv6 erlaubt, da der Parser beide Adressfamilien intern lediglich als String repräsentiert und diesen uninterpretiert an das Process-Launching-System weiterleitet.

Um eine IPv6-Adresse zu akzeptieren, war es erforderlich, einen passenden regulären Ausdruck (regex) zu definieren. Wird er erkannt, erzeugt der Parser daraus das `ORTE_RDS_HOSTFILE_IPV6`-Token:

```
([A-Za-z0-9][A-Za-z0-9_\-]*"@")? \
([A-Fa-f0-9]{0,4}":")+[":"]*([A-Fa-f0-9]{0,4}":")+[A-Fa-f0-9]{1,4}
{
                        orte_rds_hostfile_value.sval = yytext;
                        return ORTE_RDS_HOSTFILE_IPV6;
}
```

Der reguläre Ausdruck ist nicht vollends korrekt, er akzeptiert mehrere *::* innerhalb einer IPv6-Adresse, gerade das ist aber aufgrund fehlender Eindeutigkeit nicht erlaubt. Dies ist kein praktisch relevantes Problem, da die fälschlich akzeptierte, ungültige IPv6-Adresse an das PLS weitergeleitet und dort von `getaddrinfo()` als fehlerhaft abgewiesen wird. Ferner prüft der existierende IPv4-Parser ebenfalls nicht, ob ein Oktett stets kleiner als 256 ist, ob also eine akzeptierte Adresse semantisch korrekt ist. Wie im Compilerbau ist dieser Test nachgelagerten Phasen vorbehalten, im Falle von Hostnames kann ohnehin erst mit Hilfe des DNS' die semantische Eignung festgestellt werden.

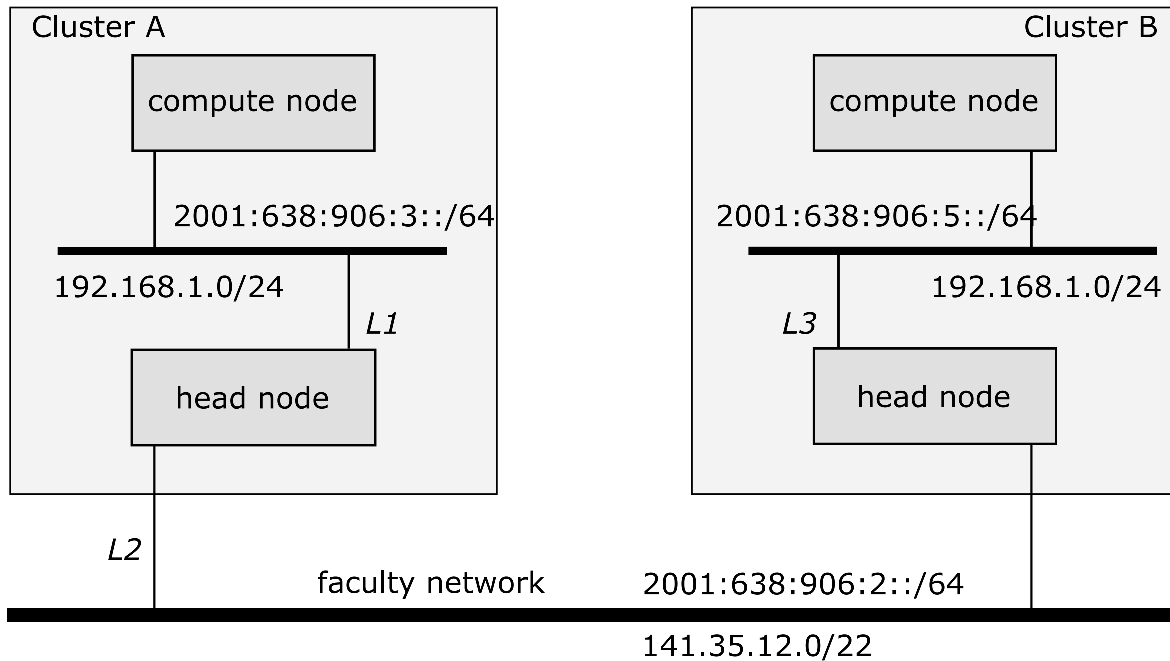

Abbildung 11: Schematische Darstellung der Multicluster-Meßstrecke. Welche Leistung kann für clusterübergreifende Kommunikation erzielt werden?

5 Leistungsbewertung

5.1 Meßvorhaben und -aufbau

Eine MPI-Implementierung muß sich neben der gebotenen Funktionalität vor allem an ihrer Geschwindigkeit messen lassen. Heutzutage ist eine Vielzahl an Benchmarks verfügbar, die je nach intendierter Anwendung verschiedene Aspekte einer Bibliothek prüfen, beispielsweise die I/O-Leistung oder auch die Skalierbarkeit kollektiver Operationen. Da die im Rahmen dieser Diplomarbeit entstandene IPv6-Erweiterung mit der BTL/TCP-Komponente Teile der performance-kritischen MPI-Schicht umfaßt, ist eine Leistungsbewertung angezeigt.

Ping-Pong-Messungen aus der Intel-MPI-Benchmarks-Suite (IMB-2.3) sollen zu Beginn zeigen, wie sich Datendurchsatz und Nachrichtenlatenz zwischen IPv4 und IPv6 unterscheiden. Diese Untersuchung wird innerhalb eines bestehenden Einzelclusters durchgeführt und dient somit als Referenzleistung, die auf direktem Wege, also ohne den Einsatz von Routing, erzielt werden kann.

Danach werden zwei Einzelcluster unter Zuhilfenahme von IPv6-Routing miteinander gekoppelt, die Leistung des Koppel-Netzes bestimmt und im Anschluß clusterübergreifender Datendurchsatz sowie Latenz ermittelt. Die erzielten Werte der IPv6-Cluster-Cluster-Kopplung werden darüber hinaus zu einer früheren Messung userspace-basierter Cluster-Cluster-Kopplung mit Hilfe von OpenVPN [35] in Bezug gesetzt, um die für die IPv6-Implementierung zu erwartenden konzeptionellen Performance-Vorteile kernelbasierter Multicluster-Lösungen zu verifizieren.

Der allen Messungen zugrunde liegende und in Abbildung 11 dargestellte Meßaufbau gliedert sich in zwei getrennte Cluster A und B, deren Headnodes über einen Cisco Catalyst 6509-Layer3-Switch miteinander verbunden sind. Innerhalb der Cluster kommt ein Netgear ProSafe GS724T-Switch zum Einsatz. Cluster A verwendet Linux-2.6.18.6 auf AMD Opteron 250-Rechnern mit Broadcom BCM5704-Netzwerkkarten, die Compute-Nodes in Cluster B hingegen nutzen Intel Pentium4 3 GHz-Prozessoren mit Intel 82547EI-Netzwerkkarten unter Linux-2.6.18.1. Sowohl der Inter-Cluster-Link (Strecke *L2*, siehe Abbildung 11) als auch die clusterinternen Verbindungen *L1* und *L3* sind als Gigabit-Ethernet ausgeführt.

5.2 Leistungsunterschiede zwischen IPv4 und IPv6 im LAN

IPv6 verfügt im Vergleich zu IPv4 über einen 20 Byte größeren Header, so daß der Einsatz von IPv6 für Message-Passing aufgrund des zusätzlichen Overheads Geschwindigkeitseinbußen erwarten läßt. Bei einer Ethernet-üblichen MTU von 1500 Byte bietet IPv4 nach Abzug von IP- und TCP-Header 1460 Byte für Nutzdaten (MSS), IPv6 hingegen lediglich 1440 Byte. Der theoretische Durchsatz-Verlust von 1,37% kann im Falle von Gigabit-Ethernet und darüber hinaus mit Hilfe von Jumbo-Frames reduziert werden, für die hier betrachtete MTU von 1500 Byte stellt er jedoch den mindestens zu erwartenden Abfall dar.

Die Messung in Abbildung 12 zeigt, daß keine gravierenden Geschwindigkeitsunterschiede zwischen IPv4 und IPv6 bestehen: beide Protokolle erreichen auf der clusterinternen Meßstrecke *L1* annähernd gleiche Werte, erst ein Blick in Anhang A.1 und A.2 offenbart die Unterschiede; IPv4 bietet eine maximale Bandbreite von $111,5$ MiB/s, IPv6 hingegen $110,0$ MiB/s. Dies entspricht einem Verlust von 1,4% und deckt sich somit weitestgehend mit der theoretischen Erwartung.

Analog zur Bandbreitenmessung vergleicht Abbildung 13 die Latenz beider Protokollfamilien. Auch hierbei sind kaum Unterschiede zu verzeichnen, insbesondere die häufig als kritisch erachtete Latenz kurzer Nachrichten zeigt keine Auffälligkeiten.

5.3 Leistungsbewertung im realen Multicluster-Betrieb

Um die Leistungfähigkeit der BTL/TCP-Komponente im echten Multicluster-Betrieb zu bewerten, wurde in Abbildung 14 neben den clusterinternen Teilstrecken *L1* und *L3* auch der erzielbare Durchsatz zwischen den beiden Headnodes ermittelt. Diese Verbindung *L2* führt über den oben genannten Cisco Catalyst-L3-Switch und erreicht mit $111,2$ MiB/s für IPv4 nahezu LAN-typische Werte, für IPv6 konnten hingegen lediglich $109,4$ MiB/s gemessen werden (vgl. dazu auch Anhang A.3 sowie A.4). Die Diskrepanz von 1,6% deutet ebenfalls auf Optimierungspotenzial hin, die jedoch mit dem Einsatz IPv6-fähiger IOS-Versionen geringer ausfallen könnte.

Um den Einfluß von Kernel-Level-Routing auf die Leistungsfähigkeit der Cluster-Cluster-Kopplung zu beurteilen, liefen die Ping-Pong-Meßdaten von einem Compute-Node in Cluster A über das Cluster-A-interne Netz *L1*, wurden vom Kernel des Headnodes über *L2* an den Kernel des Cluster-B-Headnodes versandt und schlußendlich über

L3 dem Compute-Node in Cluster B zugestellt.

Das in Anhang A.5 gezeigte Bandbreiten-Maximum der IPv6-Ende-zu-Ende-Kommunikation zweier Compute-Nodes in getrennten Clustern liegt mit $108,6$ MiB/s zwar unterhalb der Leistung aller Einzelstrecken *L1* bis *L3*, ist aber im Gegensatz zu den mit OpenVPN erzielten und ebenfalls in Abbildung 14 dargestellten 29 MiB/s um ein Vielfaches höher. Dies bestätigt die Annahme und ursprüngliche Motivation, daß Message-Passing auf Kernelebene deutliche Leistungsvorteile gegenüber userspace-basierenden Lösungen bietet.

Tatsächlich ist Abbildung 15 zu entnehmen, daß die Ende-zu-Ende-Latenz bei IPv6-Multicluster-Kommunikation für alle gemessenen Nachrichtengrößen stets niedriger ist als die summierte Latenz aller Teilstrecken: während die Ende-zu-Ende-Kommunikation von der Paketvermittlung im IPv6-Stack des Kernels profitiert und somit unterhalb der Summenkurve liegt, enthält die Summe für jede Teilstrecke *L1* bis *L3* zusätzlich die Übergangslatenz in den Userspace, die Verarbeitungszeit der Standardlibrary sowie der MPI-Bibliothek.

Da Pakete in Userspace-basierenden Lösungen ähnlich oft den vollen Protokoll- und Bibliotheksstapel durchlaufen, liegt selbst deren theoretisch erzielbares Latenzminimum nicht deutlich unterhalb der Summenkurve. Ferner führt der fehlende Direktzugriff auf Netzwerkkarten-Interrupts zu weiteren Einbußen, wenn der Scheduler nach Eintreffen eines Pakets der Gateway-Anwendung im Userspace erst vergleichsweise spät Ausführungszeit zuteilt.

Die OpenVPN-Latenzkurve in Abbildung 15 zeigt in diesem Fall deutliche Schwankungen bei kurzen Nachrichten, da hier die zusätzliche Scheduler-Latenz im Vergleich zur Gesamtlaufzeit der Pakete stark ins Gewicht fällt. Längere Nachrichten hingegen werden durch den Kernel in einem Puffer gesammelt und dann im Gesamten verarbeitet, so daß sich betriebssystembedingte Verzögerung ausmitteln bzw. bei Gesamtlaufzeiten jenseits von 500 µs unbedeutend werden.

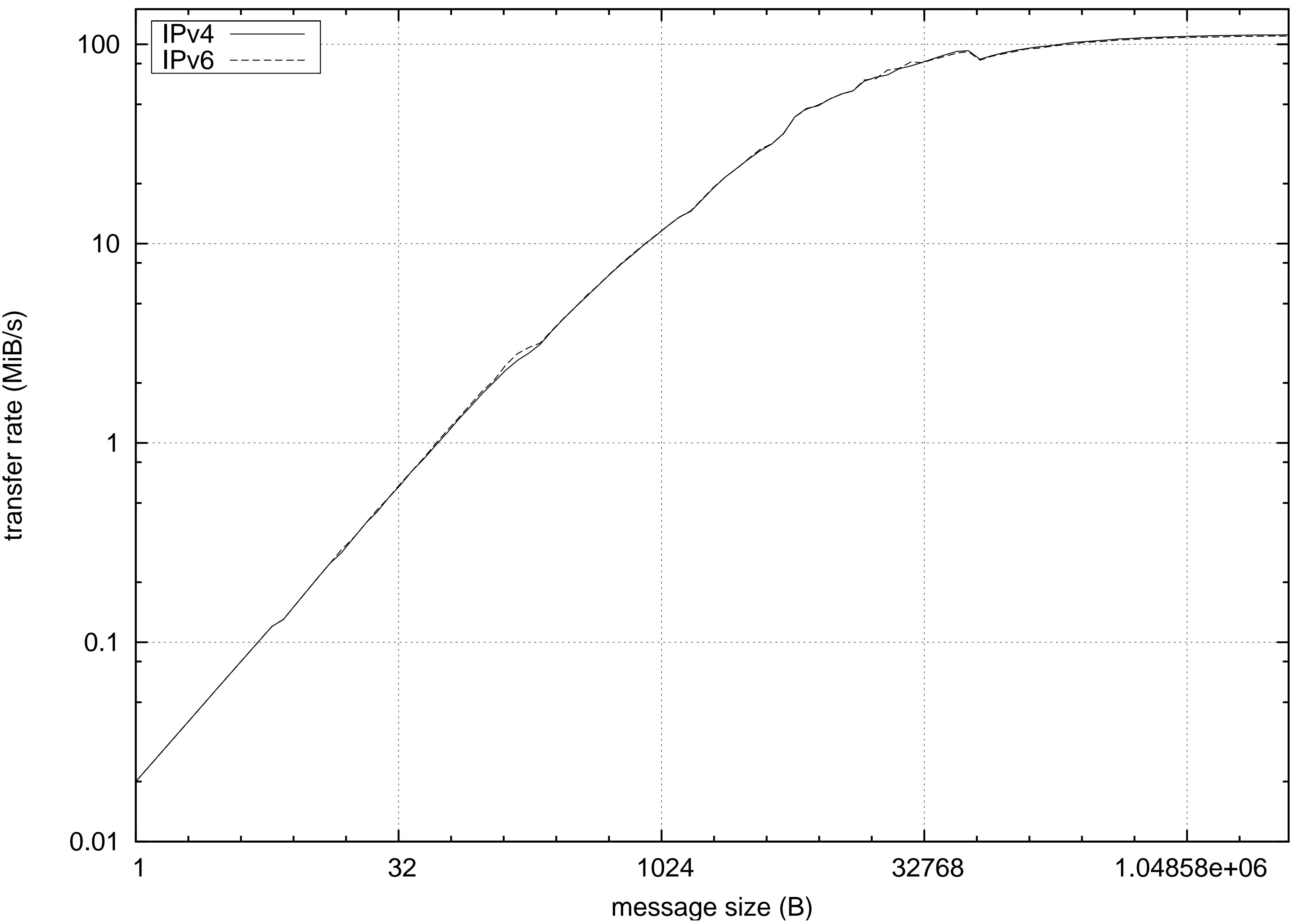

Abbildung 12: Durchsatz innerhalb eines Cluster: IPv4 erreicht 111, 5 MiB/s, IPv6 110, 0 MiB/s.

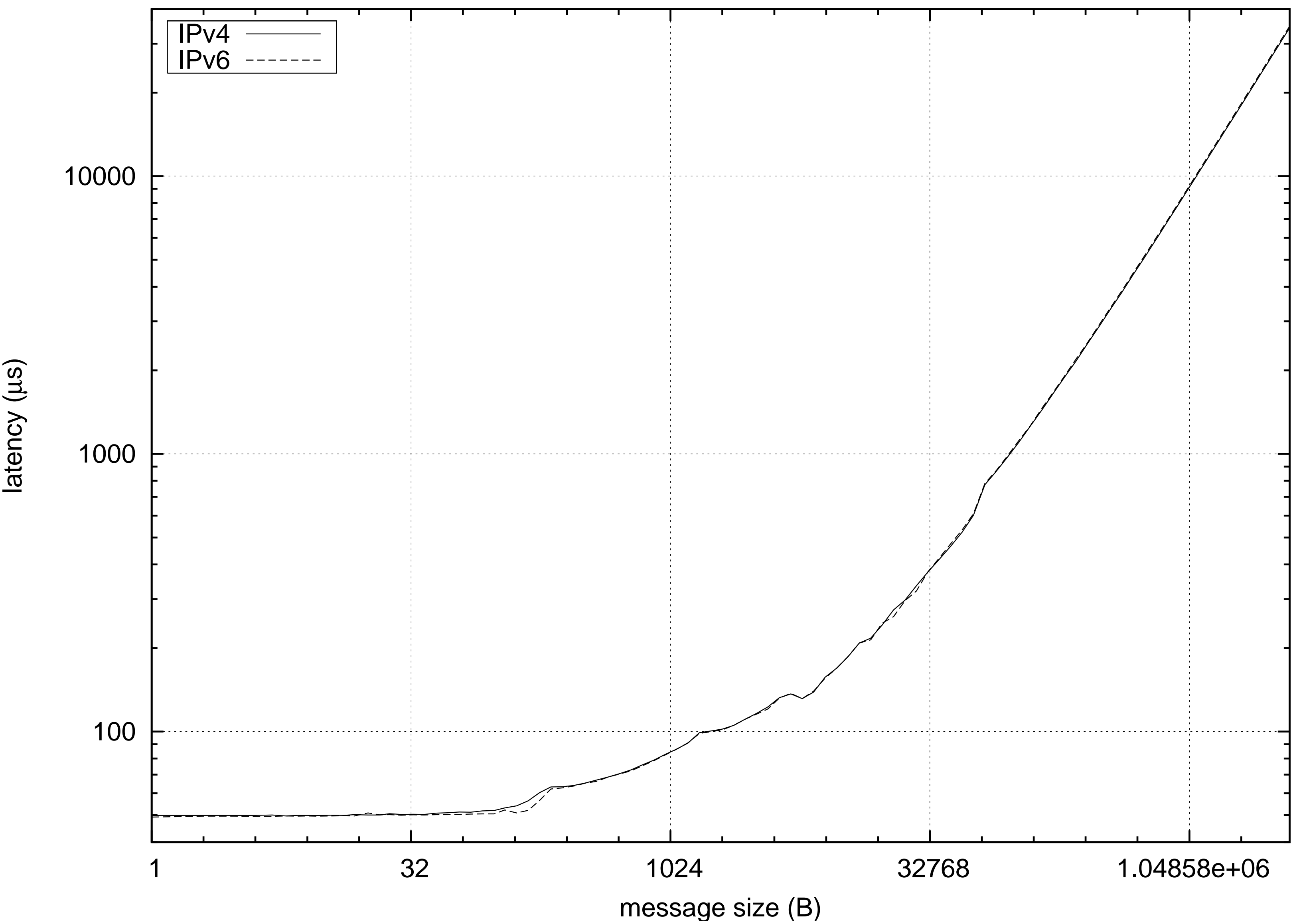

Abbildung 13: Intra-Cluster-Latenz: IPv4 und IPv6 erzielen kaum voneinander unterscheidbare Werte.

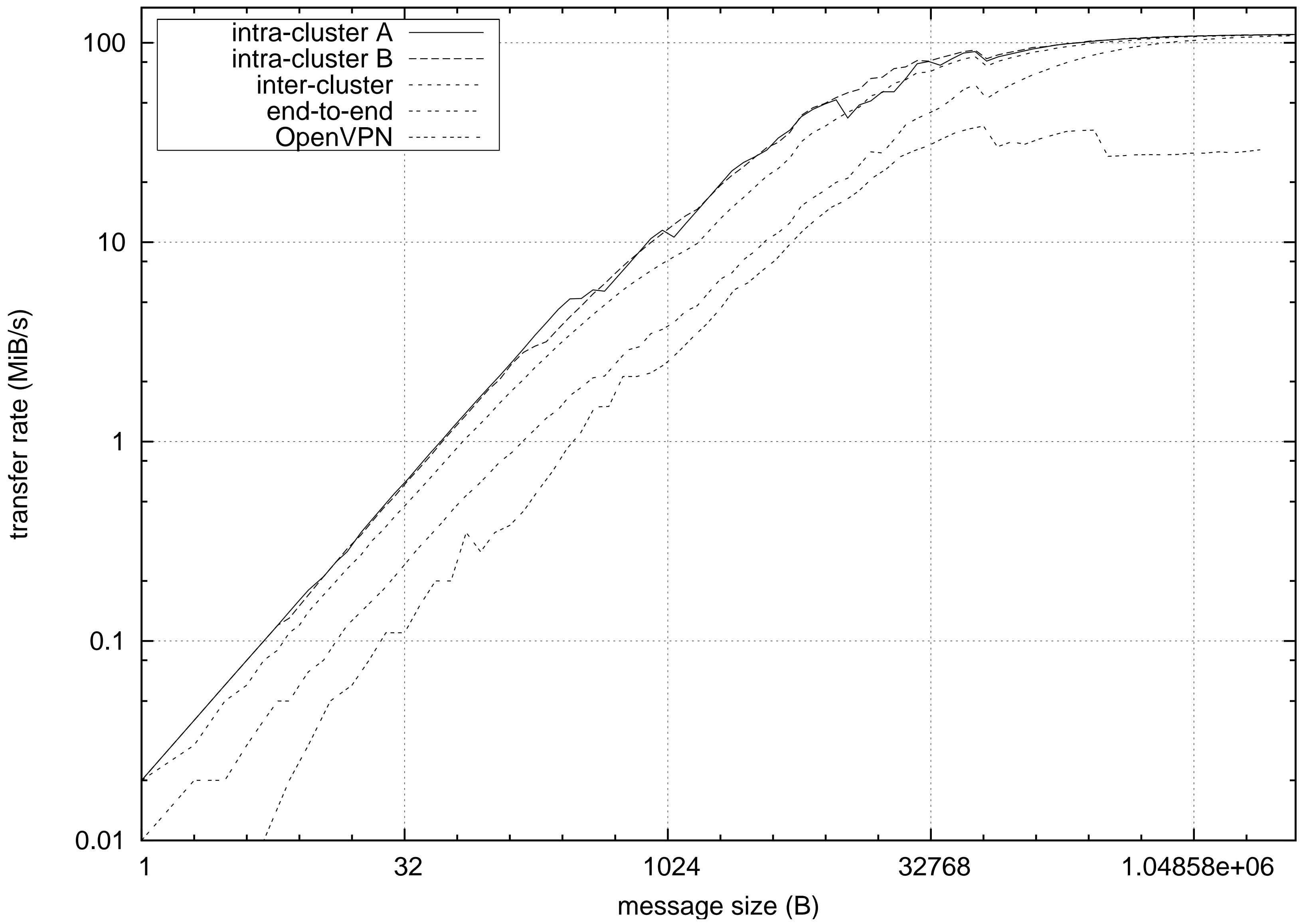

Abbildung 14: Erreichte Bandbreite innerhalb zweier Cluster A und B, zwischen den Headnodes sowie bei einer Ende-zu-Ende-Kommunikation.

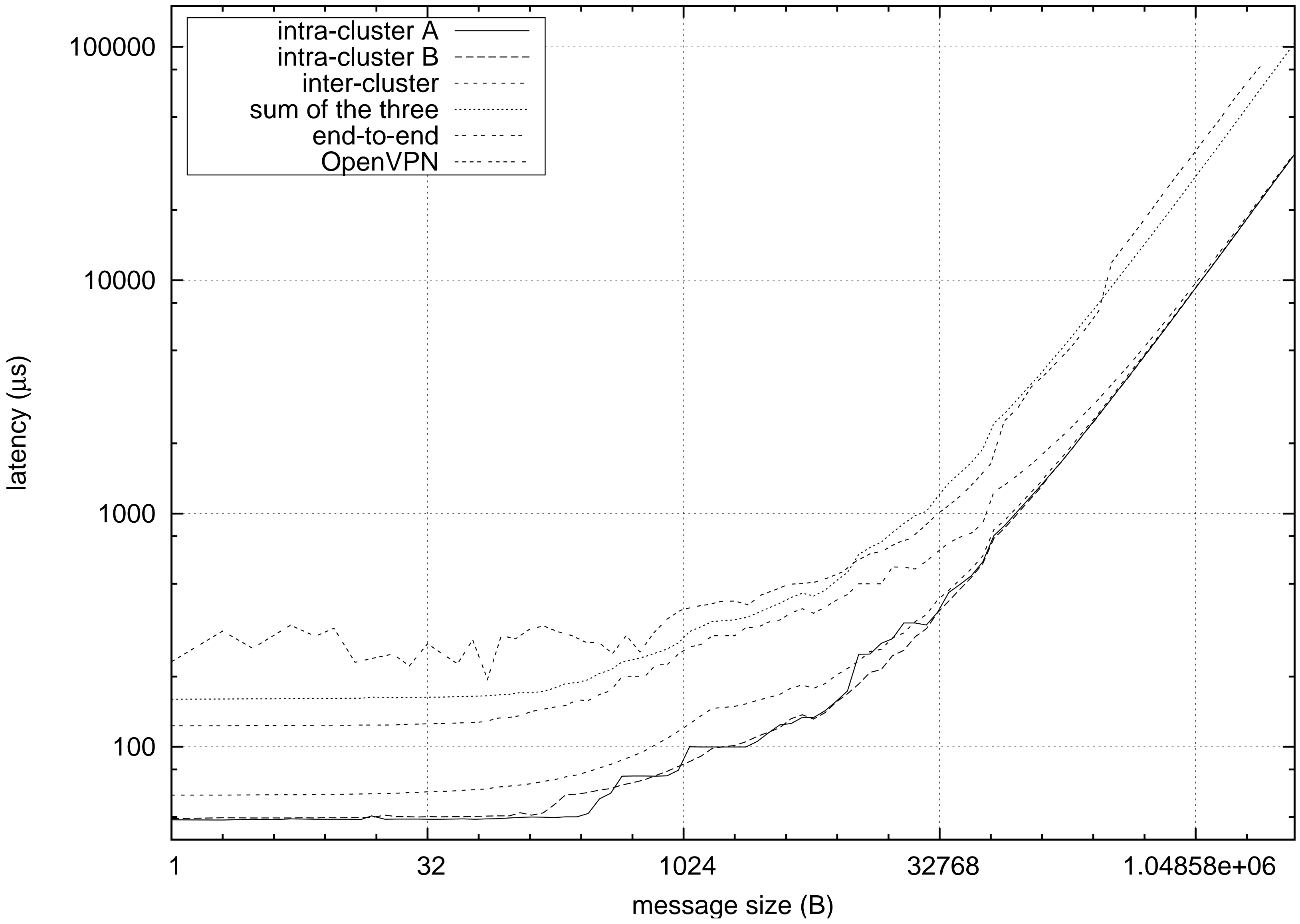

Abbildung 15: Latenz im Multicluster: kernelbasierendes Routing führt zu besseren Werten als die Summe aller Einzelstrecken.

6 Zusammenfassung und Ausblick

Gegenstand dieser Diplomarbeit war die Erweiterung des Message-Passing-Frameworks Open MPI um IPv6-Unterstützung. Da eine reine Umstellung auf IPv6 unter Verzicht von IPv4 gegenwärtig nicht praktikabel ist, sollte eine Implementierung beide Adressfamilien unterstützen.

Kommt eine derart angepaßte Bibliothek in gemischten IPv4/IPv6-Multiclustern zum Einsatz, können im Zusammenhang mit privaten IPv4-Adressen fremde Dienste gestört werden, wenn aufgrund mangelnder Eindeutigkeit Datenpakete den vermeintlich richtigen aber dennoch falschen Host erreichen. Es wurde gezeigt, daß diesem Problem durch Priorisierung von IPv6-Adressen begegnet werden kann. Ein darüber hinaus vorgeschlagener universeller Adress-Auswahlalgorithmus wurde jedoch bis dato nicht implementiert und sollte daher Gegenstand zukünftiger Untersuchungen sein.

Verfügen die Kommunikationsteilnehmer über mehrere Adressen, beispielsweise sowohl IPv4 als auch IPv6, ergeben sich zwischen ihnen häufig mehrere mögliche Verbindungen. Nutzen diese jedoch das gleiche physische Netz, würde das ansonsten durchsatzfördernde Message-Striping aufgrund des zusätzlichen Overheads Leistungseinbußen verursachen.

Zur Lösung dieses Problems wurde ein Verfahren eingeführt, das die Anzahl paralleler Verbindungen zu einem Ziel auf das Minimum verfügbarer Netzwerkkarten reduziert. Die geschaffene Implementierung berücksichtigt dabei gegenwärtig nur die senderseitige Netzwerkkonfiguration, Informationen bezüglich des Empfängers liegen zwar vor, werden aber nicht ausgewertet. Es gilt, diesen Aspekt im Interesse optimaler Ressourcennutzung zukünftig näher zu untersuchen.

Die ermittelten Leistungsdaten bestätigen die grundsätzliche Eignung von IPv6 für Message-Passing: sowohl Latenz als auch Durchsatz zeigen im LAN konkurrenzfähige Werte zu IPv4. Bei der Cluster-Cluster-Kopplung profitiert IPv6-Message-Passing von kernelbasiertem Routing und ist damit in der Lage, Gigabit-Netze im Inter-Cluster-Betrieb zu saturieren.

Die vorgestellte Implementierung beherrscht dank ihrer automatischen Adressauswahl sowohl klassische IPv4-Einzelcluster als auch dynamische IPv6-Multicluster, die abgesehen von einer einmaligen IPv6-Adressvergabe keine zusätzliche Konfiguration erfordern und damit im Gegensatz zu anderen Multicluster-Lösungen ohne weiteren Kopplungsaufwand zur Verfügung stehen.

IPv6-Message-Passing stellt daher eine leistungsstarke sowie flexible Möglichkeit dar, vorhandene verteilte Ressourcen kostengünstig miteinander zu verbinden und für das Lösen CPU-sensitiver Probleme heranzuziehen.

IPv6 bietet aufgrund seiner ausgeprägten Multicast-Fähigkeiten großes Potenzial für kollektive Operationen, das es in zukünftigen Untersuchungen auszuloten gilt. Ferner ermöglicht der hierarchische IPv6-Adressraum das automatische Erkennen von Clustergrenzen und erlaubt somit eine topologiebasierende Optimierung der Kommunikation in Multiclustern.

Literaturverzeichnis

[1] Mario Petrone and Roberto Zarrelli. Utilizing pvm in a multidomain clusters environment. In Martino et al. [36], pages 241–249.

[2] Franco Frattolillo. A pvm extension to exploit cluster grids. In Kranzlmüller et al. [37], pages 362–369.

[3] P. Srisuresh and M. Holdrege. IP Network Address Translator (NAT) Terminology and Considerations. RFC 2663 (Informational), August 1999.

[4] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[5] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301 (Proposed Standard), December 2005.

[6] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFC 3168.

[7] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. RFC 2018 (Proposed Standard), October 1996.

[8] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. RFC 2581 (Proposed Standard), April 1999. Updated by RFC 3390.

[9] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An Extension to the Selective Acknowledgement (SACK) Option for TCP. RFC 2883 (Proposed Standard), July 2000.

[10] Peter Hwang, Dawid Kurzyniec, and Vaidy S. Sunderam. Heterogeneous parallel computing across multidomain clusters. In Kranzlmüller et al. [37], pages 337–344.

[11] Thomas Beisel, Edgar Gabriel, and Michael M. Resch. An extension to mpi for distributed computing on mpps. In Marian Bubak, Jack Dongarra, and Jerzy Wasniewski, editors, *PVM/MPI*, volume 1332 of *Lecture Notes in Computer Science*, pages 75–82. Springer, 1997.

[12] Rafael Martínez-Torres. PVM-3.4.4 + IPv6: Full grid connectivity. In Martino et al. [36], pages 233–240.

[13] B. Carpenter and K. Moore. Connection of IPv6 Domains via IPv4 Clouds. RFC 3056 (Proposed Standard), February 2001.

[14] C. Huitema. An Anycast Prefix for 6to4 Relay Routers. RFC 3068 (Proposed Standard), June 2001.

[15] Lars Schneidenbach and Bettina Schnor. Migration of MPI applications to IPv6 networks. In T. Fahringer and M. H. Hamza, editors, *Proc. Parallel and Distributed Computing and Networks (PDCN 2005)*, Calgary, 2005. ACTA Press.

[16] J. Postel. Internet Protocol. RFC 791 (Standard), September 1981. Updated by RFC 1349.

[17] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460 (Draft Standard), December 1998.

[18] R. Gilligan and E. Nordmark. Transition Mechanisms for IPv6 Hosts and Routers. RFC 2893 (Proposed Standard), August 2000. Obsoleted by RFC 4213.

[19] IANA. Special-Use IPv4 Addresses. RFC 3330 (Informational), September 2002.

[20] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear. Address Allocation for Private Internets. RFC 1918 (Best Current Practice), February 1996.

[21] C. Huitema and B. Carpenter. Deprecating Site Local Addresses. RFC 3879 (Proposed Standard), September 2004.

[22] R. Hinden and B. Haberman. Unique Local IPv6 Unicast Addresses. RFC 4193 (Proposed Standard), October 2005.

[23] R. Gilligan, S. Thomson, J. Bound, J. McCann, and W. Stevens. Basic Socket Interface Extensions for IPv6. RFC 3493 (Informational), February 2003.

[24] Marc Blanchet, André Cormier, and Florent Parent. Porting applications to IPv6: simple and easy!, May 2000. `http://www.viagenie.qc.ca/en/ipv6/presentations/IPv6%20porting%20appl_v1.pdf`.

[25] Niall Richard Murphy and David Malone. *IPv6 Network Administration*. O'Reilly, Sebastopol, CA, USA, 2005.

[26] Sun Microsystems. Porting Networking Applications to the IPv6 APIs, October 1999. Formerly available at `http://www.sun.com/solaris/ipv6`, also mirrored: `http://cluster.inf-ra.uni-jena.de/~adi/porting_guide_ipv6.pdf`.

[27] Hewlett-Packard Development Company L.P. HP-UX IPv6 Porting Guide, September 2004. `http://docs.hp.com/en/netcom.html#IPv6`.

[28] Microsoft Developer Network (MSDN). IPv6 Guide for Windows Sockets Applications. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winsock/winsock/ipv6_guide_for_windows_sockets_applications_2.asp`.

[29] http://www.open-mpi.org/about/members/.

[30] http://www.open-mpi.org/community/lists/devel/2006/03/0785.php.

[31] A. Lichei T. Mehlan, T. Hoefler. Towards optimal message-striping for heterogeneous networks in Open MPI. In *Proceedings of KiCC'07, Chemnitzer Informatik Berichte*, February 2007.

[32] V. Fuller and T. Li. Classless Inter-domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan. RFC 4632 (Best Current Practice), August 2006.

[33] http://www.open-mpi.org/community/lists/devel/2006/09/1024.php.

[34] Hadrien Bullot, R. Les Cottrell, and Richard Hughes-Jones. Evaluation of advanced tcp stacks on fast long-distance production networks. *J. Grid Comput.*, 1(4):345–359, 2003.

[35] Christian Kauhaus and Dietmar Fey. Building Mini-Grid environments with Virtual Private Networks: A pragmatic approach. In *Proc. PARELEC, Bialystok, Poland, September 13–17*, pages 111–115, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[36] Beniamino Di Martino, Dieter Kranzlmüller, and Jack Dongarra, editors. *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 12th European PVM/MPI Users' Group Meeting, Sorrento, Italy, September 18-21, 2005, Proceedings*, volume 3666 of *Lecture Notes in Computer Science*. Springer, 2005.

[37] Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra, editors. *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, September 19-22, 2004, Proceedings*, volume 3241 of *Lecture Notes in Computer Science*. Springer, 2004.

A Meßwerte

A.1 Intra-Cluster-Kommunikation mit IPv4

```
#---------------------------------------------------
#    Intel (R) MPI Benchmark Suite V2.3, MPI-1 part
#---------------------------------------------------
# Date       : Wed Jan  3 11:13:02 2007
# Machine    : x86_64# System     : Linux
# Release    : 2.6.18-gentoo-r6
# Version    : #1 SMP Tue Jan 2 16:59:18 CET 2007

#
# Minimum message length in bytes:   0
# Maximum message length in bytes:   4194304
#
# MPI_Datatype                   :   MPI_BYTE
# MPI_Datatype for reductions    :   MPI_FLOAT
# MPI_Op                         :   MPI_SUM
#
#

# List of Benchmarks to run:

# PingPong

#---------------------------------------------------
# Benchmarking PingPong
# #processes = 2
#---------------------------------------------------
       #bytes #repetitions      t[usec]   Mbytes/sec
            0         1000        49.58         0.00
            1         1000        49.90         0.02
            2         1000        49.91         0.04
            3         1000        49.91         0.06
            4         1000        49.95         0.08
            5         1000        50.10         0.10
            6         1000        49.71         0.12
            7         1000        49.95         0.13
            8         1000        49.96         0.15
            9         1000        49.88         0.17
           11         1000        50.01         0.21
           13         1000        49.92         0.25
           15         1000        50.23         0.28
           18         1000        50.12         0.34
           21         1000        50.06         0.40
           24         1000        50.63         0.45
           28         1000        50.30         0.53
           33         1000        50.43         0.62
           38         1000        50.30         0.72
           45         1000        50.90         0.84
           52         1000        51.06         0.97
           61         1000        51.33         1.13
           71         1000        51.28         1.32
           83         1000        51.80         1.53
           97         1000        51.99         1.78
          112         1000        53.11         2.01
          131         1000        54.04         2.31
          153         1000        56.37         2.59
          178         1000        60.22         2.82
          207         1000        63.24         3.12
          242         1000        63.29         3.65
          282         1000        63.99         4.20
          328         1000        65.29         4.79
          382         1000        66.93         5.44
```

445	1000	68.69	6.18
519	1000	70.63	7.01
604	1000	72.83	7.91
704	1000	75.94	8.84
820	1000	78.96	9.90
955	1000	82.87	10.99
1112	1000	86.45	12.27
1296	1000	91.03	13.58
1509	1000	99.17	14.51
1758	1000	100.55	16.67
2047	1000	102.09	19.12
2385	1000	105.32	21.60
2778	1000	110.86	23.90
3236	1000	116.36	26.52
3769	1000	122.80	29.27
4389	1000	132.43	31.61
5113	1000	136.60	35.70
5955	1000	131.46	43.20
6936	1000	138.93	47.61
8079	1000	156.89	49.11
9410	1000	169.19	53.04
10960	1000	185.92	56.22
12765	1000	208.39	58.42
14868	1000	217.01	65.34
17318	1000	241.89	68.28
20171	1000	274.55	70.07
23493	1000	297.44	75.33
27364	1000	335.32	77.83
31871	1000	375.20	81.01
37122	1000	417.32	84.83
43237	970	465.32	88.62
50360	832	522.10	91.99
58656	715	601.85	92.94
68319	613	775.06	84.06
79573	527	865.84	87.64
92681	452	976.70	90.50
107949	388	1105.37	93.13
125732	333	1262.01	95.01
146445	286	1440.68	96.94
170569	245	1655.42	98.26
198668	211	1895.21	99.97
231395	181	2163.14	102.02
269513	155	2497.95	102.90
313911	133	2875.87	104.10
365623	114	3320.21	105.02
425854	98	3820.08	106.31
496006	84	4430.04	106.78
577715	72	5111.44	107.79
672884	62	5922.85	108.35
783731	53	6869.44	108.80
912838	45	7967.62	109.26
1063213	39	9242.77	109.70
1238360	33	10738.41	109.98
1442360	29	12472.14	110.29
1679965	24	14489.27	110.57
1956712	21	16839.45	110.82
2279048	18	19569.67	111.06
2654483	15	22760.47	111.22
3091766	13	26485.77	111.33
3601084	11	30813.82	111.45
4194303	10	35855.36	111.56

A.2 Intra-Cluster-Kommunikation mit IPv6

```
#---------------------------------------------------
#    Intel (R) MPI Benchmark Suite V2.3, MPI-1 part
#---------------------------------------------------
# Date       : Wed Jan  3 11:14:22 2007
# Machine    : x86_64# System     : Linux
# Release    : 2.6.18-gentoo-r6
# Version    : #1 SMP Tue Jan 2 16:59:18 CET 2007

#
# Minimum message length in bytes:   0
# Maximum message length in bytes:   4194304
#
# MPI_Datatype                   :   MPI_BYTE
# MPI_Datatype for reductions    :   MPI_FLOAT
# MPI_Op                         :   MPI_SUM
#
#

# List of Benchmarks to run:

# PingPong

#---------------------------------------------------
# Benchmarking PingPong
# #processes = 2
#---------------------------------------------------
       #bytes #repetitions      t[usec]   Mbytes/sec
            0         1000        49.10         0.00
            1         1000        49.23         0.02
            2         1000        49.62         0.04
            3         1000        49.59         0.06
            4         1000        49.58         0.08
            5         1000        49.57         0.10
            6         1000        49.70         0.12
            7         1000        49.60         0.13
            8         1000        49.73         0.15
            9         1000        49.65         0.17
           11         1000        49.71         0.21
           13         1000        49.76         0.25
           15         1000        49.72         0.29
           18         1000        50.99         0.34
           21         1000        50.19         0.40
           24         1000        50.24         0.46
           28         1000        50.02         0.53
           33         1000        50.17         0.63
           38         1000        50.09         0.72
           45         1000        50.25         0.85
           52         1000        50.23         0.99
           61         1000        50.33         1.16
           71         1000        50.46         1.34
           83         1000        50.55         1.57
           97         1000        50.54         1.83
          112         1000        52.22         2.05
          131         1000        50.94         2.45
          153         1000        52.06         2.80
          178         1000        56.49         3.01
          207         1000        62.19         3.17
          242         1000        62.73         3.68
          282         1000        63.74         4.22
          328         1000        65.26         4.79
          382         1000        66.25         5.50
          445         1000        68.60         6.19
          519         1000        70.34         7.04
          604         1000        72.29         7.97
```

704	1000	75.39	8.91
820	1000	78.58	9.95
955	1000	82.43	11.05
1112	1000	86.46	12.27
1296	1000	91.17	13.56
1509	1000	98.49	14.61
1758	1000	99.99	16.77
2047	1000	101.45	19.24
2385	1000	105.27	21.61
2778	1000	110.93	23.88
3236	1000	115.59	26.70
3769	1000	120.73	29.77
4389	1000	132.09	31.69
5113	1000	137.36	35.50
5955	1000	131.42	43.21
6936	1000	139.98	47.26
8079	1000	155.63	49.51
9410	1000	168.91	53.13
10960	1000	186.01	56.19
12765	1000	208.51	58.38
14868	1000	214.13	66.22
17318	1000	245.77	67.20
20171	1000	259.45	74.14
23493	1000	296.00	75.69
27364	1000	320.77	81.35
31871	1000	376.11	80.81
37122	1000	421.45	84.00
43237	970	474.21	86.95
50360	832	532.40	90.21
58656	715	609.25	91.82
68319	613	782.66	83.25
79573	527	871.35	87.09
92681	452	987.01	89.55
107949	388	1121.35	91.81
125732	333	1266.53	94.67
146445	286	1462.37	95.50
170569	245	1668.67	97.48
198668	211	1913.20	99.03
231395	181	2202.61	100.19
269513	155	2517.58	102.09
313911	133	2913.78	102.74
365623	114	3360.67	103.75
425854	98	3869.20	104.96
496006	84	4486.51	105.43
577715	72	5182.88	106.30
672884	62	6005.91	106.85
783731	53	6953.97	107.48
912838	45	8070.41	107.87
1063213	39	9367.59	108.24
1238360	33	10886.17	108.49
1442360	29	12640.05	108.82
1679965	24	14688.44	109.07
1956712	21	17070.26	109.32
2279048	18	19848.25	109.50
2654483	15	23075.53	109.71
3091766	13	26849.23	109.82
3601084	11	31239.59	109.93
4194303	10	36354.10	110.03

A.3 Headnode-Headnode-Kommunikation via IPv4

```
#---------------------------------------------------
#    Intel (R) MPI Benchmark Suite V2.3, MPI-1 part
#---------------------------------------------------
# Date       : Wed Jan  3 15:49:11 2007
# Machine    : x86_64# System     : Linux
# Release    : 2.6.18-gentoo-r6
# Version    : #1 SMP Tue Jan 2 16:59:18 CET 2007

#
# Minimum message length in bytes:   0
# Maximum message length in bytes:   4194304
#
# MPI_Datatype                   :   MPI_BYTE
# MPI_Datatype for reductions    :   MPI_FLOAT
# MPI_Op                         :   MPI_SUM
#
#

# List of Benchmarks to run:

# PingPong

#---------------------------------------------------
# Benchmarking PingPong
# #processes = 2
#---------------------------------------------------
       #bytes #repetitions      t[usec]   Mbytes/sec
            0         1000        62.92         0.00
            1         1000        63.28         0.02
            2         1000        63.33         0.03
            3         1000        63.46         0.05
            4         1000        63.43         0.06
            5         1000        63.49         0.08
            6         1000        63.76         0.09
            7         1000        63.61         0.10
            8         1000        63.52         0.12
            9         1000        63.86         0.13
           11         1000        63.72         0.16
           13         1000        63.83         0.19
           15         1000        63.96         0.22
           18         1000        64.26         0.27
           21         1000        64.46         0.31
           24         1000        64.68         0.35
           28         1000        64.93         0.41
           33         1000        65.39         0.48
           38         1000        65.35         0.55
           45         1000        65.92         0.65
           52         1000        66.26         0.75
           61         1000        66.91         0.87
           71         1000        67.59         1.00
           83         1000        68.21         1.16
           97         1000        68.55         1.35
          112         1000        70.14         1.52
          131         1000        70.42         1.77
          153         1000        71.54         2.04
          178         1000        72.92         2.33
          207         1000        74.81         2.64
          242         1000        77.04         3.00
          282         1000        79.41         3.39
          328         1000        81.87         3.82
          382         1000        85.65         4.25
          445         1000        88.60         4.79
          519         1000        92.63         5.34
          604         1000        97.44         5.91
```

704	1000	104.03	6.45
820	1000	109.89	7.12
955	1000	118.15	7.71
1112	1000	126.45	8.39
1296	1000	136.39	9.06
1509	1000	149.04	9.66
1758	1000	150.25	11.16
2047	1000	152.15	12.83
2385	1000	154.61	14.71
2778	1000	160.73	16.48
3236	1000	165.62	18.63
3769	1000	170.42	21.09
4389	1000	180.99	23.13
5113	1000	185.33	26.31
5955	1000	179.10	31.71
6936	1000	186.15	35.53
8079	1000	206.48	37.31
9410	1000	217.15	41.33
10960	1000	232.56	44.94
12765	1000	254.70	47.80
14868	1000	264.38	53.63
17318	1000	290.07	56.94
20171	1000	321.11	59.91
23493	1000	343.92	65.14
27364	1000	381.66	68.38
31871	1000	422.77	71.89
37122	1000	464.56	76.21
43237	970	511.19	80.66
50360	832	567.78	84.59
58656	715	650.10	86.05
68319	613	849.15	76.73
79573	527	938.90	80.83
92681	452	1050.14	84.17
107949	388	1179.34	87.29
125732	333	1338.75	89.57
146445	286	1514.99	92.19
170569	245	1725.74	94.26
198668	211	1968.48	96.25
231395	181	2241.81	98.44
269513	155	2568.68	100.06
313911	133	2950.00	101.48
365623	114	3389.03	102.89
425854	98	3902.10	104.08
496006	84	4501.27	105.09
577715	72	5189.92	106.16
672884	62	6007.73	106.81
783731	53	6958.56	107.41
912838	45	8052.95	108.10
1063213	39	9339.59	108.57
1238360	33	10829.44	109.05
1442360	29	12563.33	109.49
1679965	24	14580.35	109.88
1956712	21	16931.93	110.21
2279048	18	19667.89	110.51
2654483	15	22923.70	110.43
3091766	13	26595.92	110.86
3601084	11	30931.78	111.03
4194303	10	35976.05	111.19

A.4 Headnode-Headnode-Kommunikation via IPv6

```
#---------------------------------------------------
#    Intel (R) MPI Benchmark Suite V2.3, MPI-1 part
#---------------------------------------------------
# Date       : Wed Jan  3 15:47:03 2007
# Machine    : x86_64# System     : Linux
# Release    : 2.6.18-gentoo-r6
# Version    : #1 SMP Tue Jan 2 16:59:18 CET 2007

#
# Minimum message length in bytes:   0
# Maximum message length in bytes:   4194304
#
# MPI_Datatype                   :   MPI_BYTE
# MPI_Datatype for reductions    :   MPI_FLOAT
# MPI_Op                         :   MPI_SUM
#
#

# List of Benchmarks to run:

# PingPong

#---------------------------------------------------
# Benchmarking PingPong
# #processes = 2
#---------------------------------------------------
       #bytes #repetitions      t[usec]   Mbytes/sec
            0         1000        61.83         0.00
            1         1000        62.12         0.02
            2         1000        62.08         0.03
            3         1000        62.23         0.05
            4         1000        62.29         0.06
            5         1000        62.39         0.08
            6         1000        62.39         0.09
            7         1000        62.48         0.11
            8         1000        62.54         0.12
            9         1000        62.62         0.14
           11         1000        62.68         0.17
           13         1000        62.85         0.20
           15         1000        62.93         0.23
           18         1000        62.99         0.27
           21         1000        63.23         0.32
           24         1000        63.62         0.36
           28         1000        63.84         0.42
           33         1000        64.17         0.49
           38         1000        64.35         0.56
           45         1000        64.73         0.66
           52         1000        65.11         0.76
           61         1000        65.52         0.89
           71         1000        65.97         1.03
           83         1000        67.23         1.18
           97         1000        67.85         1.36
          112         1000        68.67         1.56
          131         1000        69.51         1.80
          153         1000        71.03         2.05
          178         1000        72.26         2.35
          207         1000        74.35         2.66
          242         1000        75.86         3.04
          282         1000        78.30         3.43
          328         1000        81.21         3.85
          382         1000        84.17         4.33
          445         1000        87.69         4.84
          519         1000        91.60         5.40
          604         1000        96.07         6.00
```

704	1000	102.23	6.57
820	1000	108.75	7.19
955	1000	116.62	7.81
1112	1000	125.48	8.45
1296	1000	135.96	9.09
1509	1000	146.19	9.84
1758	1000	147.58	11.36
2047	1000	149.11	13.09
2385	1000	152.71	14.89
2778	1000	158.26	16.74
3236	1000	162.81	18.96
3769	1000	167.90	21.41
4389	1000	178.92	23.39
5113	1000	184.24	26.47
5955	1000	178.25	31.86
6936	1000	186.46	35.47
8079	1000	202.79	37.99
9410	1000	216.62	41.43
10960	1000	233.25	44.81
12765	1000	256.35	47.49
14868	1000	261.85	54.15
17318	1000	292.74	56.42
20171	1000	307.66	62.53
23493	1000	343.73	65.18
27364	1000	368.60	70.80
31871	1000	426.57	71.25
37122	1000	469.52	75.40
43237	970	521.47	79.07
50360	832	579.81	82.83
58656	715	657.21	85.12
68319	613	854.08	76.29
79573	527	940.34	80.70
92681	452	1056.94	83.63
107949	388	1190.90	86.45
125732	333	1336.62	89.71
146445	286	1530.84	91.23
170569	245	1731.87	93.93
198668	211	1977.73	95.80
231395	181	2269.40	97.24
269513	155	2584.75	99.44
313911	133	2976.66	100.57
365623	114	3430.02	101.66
425854	98	3982.02	101.99
496006	84	4564.65	103.63
577715	72	5266.22	104.62
672884	62	6079.35	105.56
783731	53	7034.32	106.25
912838	45	8144.76	106.88
1063213	39	9442.48	107.38
1238360	33	10955.47	107.80
1442360	29	12724.29	108.10
1679965	24	14807.19	108.20
1956712	21	17160.05	108.74
2279048	18	19938.72	109.01
2654483	15	23176.37	109.23
3091766	13	26947.31	109.42
3601084	11	31366.91	109.49
4194303	10	36549.75	109.44

A.5 Ende-zu-Ende-Kommunikation mit IPv6

```
#---------------------------------------------------
#    Intel (R) MPI Benchmark Suite V2.3, MPI-1 part
#---------------------------------------------------
# Date       : Wed Jan  3 17:16:33 2007
# Machine    : i686# System     : Linux
# Release    : 2.6.18.1
# Version    : #1 SMP Tue Jan 2 18:11:39 CET 2007

#
# Minimum message length in bytes:   0
# Maximum message length in bytes:   4194304
#
# MPI_Datatype                   :   MPI_BYTE
# MPI_Datatype for reductions    :   MPI_FLOAT
# MPI_Op                         :   MPI_SUM
#
#

# List of Benchmarks to run:

# PingPong

#---------------------------------------------------
# Benchmarking PingPong
# #processes = 2
#---------------------------------------------------
       #bytes #repetitions      t[usec]   Mbytes/sec
            0         1000       123.10         0.00
            1         1000       123.15         0.01
            2         1000       123.16         0.02
            3         1000       123.24         0.02
            4         1000       123.37         0.03
            5         1000       123.45         0.04
            6         1000       123.57         0.05
            7         1000       123.73         0.05
            8         1000       123.66         0.06
            9         1000       123.72         0.07
           11         1000       123.81         0.08
           13         1000       123.99         0.10
           15         1000       123.96         0.12
           18         1000       124.02         0.14
           21         1000       124.32         0.16
           24         1000       124.88         0.18
           28         1000       125.10         0.21
           33         1000       125.58         0.25
           38         1000       125.73         0.29
           45         1000       126.19         0.34
           52         1000       126.82         0.39
           61         1000       127.11         0.46
           71         1000       128.64         0.53
           83         1000       132.69         0.60
           97         1000       133.77         0.69
          112         1000       135.88         0.79
          131         1000       141.86         0.88
          153         1000       144.84         1.01
          178         1000       148.16         1.15
          207         1000       150.17         1.31
          242         1000       158.81         1.45
          282         1000       157.86         1.70
          328         1000       167.65         1.87
          382         1000       174.06         2.09
          445         1000       199.66         2.13
          519         1000       199.71         2.48
          604         1000       199.73         2.88
```

704	1000	224.67	2.99
820	1000	224.65	3.48
955	1000	249.61	3.65
1112	1000	268.00	3.96
1296	1000	274.57	4.50
1509	1000	299.58	4.80
1758	1000	299.58	5.60
2047	1000	299.54	6.52
2385	1000	324.44	7.01
2778	1000	324.69	8.16
3236	1000	342.58	9.01
3769	1000	349.45	10.29
4389	1000	375.26	11.15
5113	1000	390.86	12.48
5955	1000	373.86	15.19
6936	1000	396.48	16.68
8079	1000	424.50	18.15
9410	1000	449.54	19.96
10960	1000	499.59	20.92
12765	1000	499.92	24.35
14868	1000	499.33	28.40
17318	1000	589.75	28.00
20171	1000	588.60	32.68
23493	1000	578.18	38.75
27364	1000	624.83	41.77
31871	1000	685.82	44.32
37122	1000	749.45	47.24
43237	970	791.52	52.09
50360	832	825.00	58.21
58656	715	907.52	61.64
68319	613	1240.78	52.51
79573	527	1329.82	57.07
92681	452	1447.09	61.08
107949	388	1581.96	65.08
125732	333	1736.13	69.07
146445	286	1929.83	72.37
170569	245	2133.38	76.25
198668	211	2375.26	79.77
231395	181	2665.88	82.78
269513	155	2983.86	86.14
313911	133	3372.28	88.77
365623	114	3827.64	91.10
425854	98	4339.56	93.59
496006	84	4946.85	95.62
577715	72	5645.37	97.59
672884	62	6477.10	99.07
783731	53	7416.65	100.78
912838	45	8539.77	101.94
1063213	39	9860.59	102.83
1238360	33	11347.41	104.08
1442360	29	13116.12	104.87
1679965	24	15149.60	105.75
1956712	21	17538.62	106.40
2279048	18	20317.61	106.97
2654483	15	23550.24	107.49
3091766	13	27323.54	107.91
3601084	11	31703.41	108.32
4194303	10	36842.51	108.57

B Connection-Strings

B.1 Process-Launch-Protokoll für IPv4-orteds

```
[ipc654:12261] pls:rsh: local csh: 0, local bash: 1
[ipc654:12261] pls:rsh: assuming same remote shell as local shell
[ipc654:12261] pls:rsh: remote csh: 0, remote bash: 1
[ipc654:12261] pls:rsh: final template argv:
[ipc654:12261] pls:rsh:     /usr/bin/ssh <template> orted
--bootproxy 1 --name <template> --num_procs 3 --vpid_start 0 --nodename
<template> --universe adi@ipc654:default-universe --nsreplica
"0.0.0;tcp://141.35.14.189:57985;tcp://192.168.1.1:57985" --gprreplica
"0.0.0;tcp://141.35.14.189:57985;tcp://192.168.1.1:57985"
--mpi-call-yield 0 [ipc654:12261] pls:rsh: launching on node amun4
[ipc654:12261] pls:rsh: not oversubscribed -- setting mpi_yield_when_idle to 0
[ipc654:12261] pls:rsh: amun4 is a REMOTE node

[ipc654:12261] pls:rsh: executing: /usr/bin/ssh amun4
PATH=/usr/local/openmpi/bin:$PATH ; export PATH ;
LD_LIBRARY_PATH=/usr/local/openmpi/lib:$LD_LIBRARY_PATH ; export
LD_LIBRARY_PATH ; /usr/local/openmpi/bin/orted --bootproxy 1 --name
0.0.1 --num_procs 3 --vpid_start 0 --nodename amun4 --universe
adi@ipc654:default-universe --nsreplica
"0.0.0;tcp://141.35.14.189:57985;tcp://192.168.1.1:57985" --gprreplica
"0.0.0;tcp://141.35.14.189:57985;tcp://192.168.1.1:57985"
--mpi-call-yield 0
```

B.2 Process-Launch-Protokoll für IPv4/6-fähige orteds

```
[ipc654:12299] pls:rsh: local csh: 0, local bash: 1
[ipc654:12299] pls:rsh: assuming same remote shell as local shell
[ipc654:12299] pls:rsh: remote csh: 0, remote bash: 1
[ipc654:12299] pls:rsh: final template argv:
[ipc654:12299] pls:rsh:     /usr/bin/ssh <template> orted
--bootproxy 1 --name <template> --num_procs 3 --vpid_start 0 --nodename
<template> --universe adi@ipc654:default-universe-12299 --nsreplica
"0.0.0;tcp://141.35.14.189:49290;tcp://192.168.1.1:49290;
tcp6://2001:638:906:1::1:40933;tcp6://2001:638:906:4::1:40933;
tcp6://2001:638:906:1:200:5aff:fe9e:a883:40933;
tcp6://2001:638:906:2:20e:cff:fe06:2c9b:40933;
tcp6://2001:638:f:800::906:2:40933"
--gprreplica
"0.0.0;tcp://141.35.14.189:49290;tcp://192.168.1.1:49290;
tcp6://2001:638:906:1::1:40933;tcp6://2001:638:906:4::1:40933;
tcp6://2001:638:906:1:200:5aff:fe9e:a883:40933;
tcp6://2001:638:906:2:20e:cff:fe06:2c9b:40933;
tcp6://2001:638:f:800::906:2:40933"

[ipc654:12299] pls:rsh: launching on node amun3
[ipc654:12299] pls:rsh: amun3 is a REMOTE node
[ipc654:12299] pls:rsh: executing: /usr/bin/ssh amun3
PATH=/home/racl/adi/ompi/trunk/Linux-i686/bin:$PATH ; export PATH ;
LD_LIBRARY_PATH=/home/racl/adi/ompi/trunk/Linux-i686/lib:$LD_LIBRARY_PATH ;
export LD_LIBRARY_PATH ;
/home/racl/adi/ompi/trunk/Linux-i686/bin/orted --bootproxy 1 --name
0.0.1 --num_procs 3 --vpid_start 0 --nodename amun3 --universe
adi@ipc654:default-universe-12299 --nsreplica
"0.0.0;tcp://141.35.14.189:49290;tcp://192.168.1.1:49290;
tcp6://2001:638:906:1::1:40933;tcp6://2001:638:906:4::1:40933;
tcp6://2001:638:906:1:200:5aff:fe9e:a883:40933;
tcp6://2001:638:906:2:20e:cff:fe06:2c9b:40933;
tcp6://2001:638:f:800::906:2:40933"
--gprreplica
"0.0.0;tcp://141.35.14.189:49290;tcp://192.168.1.1:49290;
tcp6://2001:638:906:1::1:40933;tcp6://2001:638:906:4::1:40933;
tcp6://2001:638:906:1:200:5aff:fe9e:a883:40933;
tcp6://2001:638:906:2:20e:cff:fe06:2c9b:40933;
tcp6://2001:638:f:800::906:2:40933"
```

B.3 Illustration des Adress-Auswahl-Algorithmus

Adressen, die der Headnode an die Peers übermittelt:

192.168.1.1	(Adressklasse *IPv4-private*)
141.35.14.189	(Adressklasse *IPv4-public*)
2001:638:906:3::1	(Adressklasse *IPv6*)

Eigene Adressen eines Peers:

192.168.1.55	(Adressklasse *IPv4-private*)
2001:638:906:5::55	(Adressklasse *IPv6*)

Start des Algorithmus:

1. Der Peer sortiert die Adressen des Headnodes nach ihrer Qualität:

2001:638:906:3::1	(Adressklasse *IPv6*, gut)
141.35.14.189	(Adressklasse *IPv4-public*, gut)
192.168.1.1	(Adressklasse *IPv4-private*, schwierig)

2. Der Peer entfernt alle Adressklassen des Headnodes, über die er nicht selbst verfügt:

2001:638:906:3::1	(Adressklasse *IPv6*, gut)
192.168.1.1	(Adressklasse *IPv4-private*, schwierig)

3. Es ergibt sich die bevorzugte Kontaktierungsreihenfolge:

   a) 2001:638:906:3::1

   b) 192.168.1.1

Erklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Jena, 27.03.2007