MPI auf Basis von RESTful HTTP

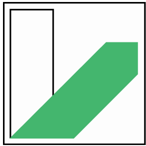
Daniel Mohr

Bayreuth Reports on Parallel and Distributed Systems

No. 6, January 2013

University of Bayreuth
Department of Mathematics, Physics and Computer Science
Applied Computer Science 2 – Parallel and Distributed Systems
95440 Bayreuth
Germany

Phone: +49 921 55 7701
Fax: +49 921 55 7702
E-Mail: brpds@ai2.uni-bayreuth.de



Fakultät Mathematik, Physik, Informatik
Lehrstuhl für Angewandte Informatik II
Prof. Dr. Thomas Rauber

Masterarbeit
zur Erlangung des Grades Master of Science (M.Sc.)

MPI auf Basis von RESTful HTTP

Daniel Mohr, B.Sc.

27. Januar 2013

Betreut durch: Dr. Matthias Korch
Dipl.-Inf. Marvin Ferber

Matrikelnummer: 1132469
Studiengang Computer Science (Master)
E-Mail: daniel.mohr@gmx.com
Adresse: Kunigundenstr. 28, 80802 München

Inhaltsverzeichnis

Vorwort 9

I. Theoretischer Teil 13

1. RESTful HTTP im MPI-Kontext 15

1.1. HTTP als Protokoll . . . 15
1.2. REST als Architekturstil . . . 17
1.3. Umsetzbarkeit der REST-Prinzipien für MPI auf Basis von HTTP . . . 18

2. Rückblick auf das Master-Projekt 23

2.1. Konzept . . . 23
2.2. Umsetzung . . . 26
2.3. Ergebnisse . . . 29

3. Anforderungen an ein MPI-System zur Nutzung in internetähnlichen Rechnernetzen 31

4. Vorhandene MPI-Implementierungen 37

4.1. Open MPI . . . 37
4.1.1. Wurzeln . . . 38
4.1.2. System . . . 41
4.2. MPICH . . . 43
4.3. MPJ/IBIS . . . 45
4.3.1. IBIS-Basissystem . . . 45
4.3.2. MPI-Schicht . . . 47
4.4. Weitere Implementierungen . . . 48
4.4.1. HeteroMPI . . . 48
4.4.2. MPICH-G2 . . . 49

4.4.3. IMPI 51
4.5. Erkenntnisse 52

II. Implementierung **55**

5. Basis-Implementierung **59**
5.1. Startphase 59
5.2. Ausführung des parallelen Programms 64

6. Umsetzung der REST-Prinzipien **71**
6.1. Ressourcen mit eindeutiger Identifikation 72
6.2. Standardmethoden 75
6.2.1. Push-Richtung 76
6.2.2. Pull-Richtung 81
6.3. Hypermedia 86

7. Bewertung der Implementierung **89**
7.1. Erfüllung der Anforderungen 89
7.2. Relevanz von REST für MPI 94

8. Optimierungen und Erweiterungen **97**
8.1. Kollektive Kommunikationsoperationen 97
8.2. Firewall-Problematik 99

9. Benchmarks und Messergebnisse **103**

10. Anwendungsbeispiel: ODE-Verfahren **111**

Fazit **115**

Abstract

This work deals with the development of an MPI implementation which uses RESTful HTTP for communication. In contrast to common MPI systems whose main focus lies on their use in homogeneous and tight coupled computer systems, such an approach offers the possibility to utilize parallel programs, which are realized with MPI, in an environment with characeristics similar to those of the World Wide Web. The work is based on an existing prototype implementation of such a system which has some conceptual insufficiencies. Beside theoretical considerations of RESTful HTTP in an MPI environment this base system is revised fundamentaly und REST conformity is established. After an evaluation of this implementation and an analysis of possible optimizations there are some performance measurements and the description of an example usage of the system in a real-world scientific computing problem.

Zusammenfassung

Diese Arbeit beschäftigt sich mit der Entwicklung einer MPI-Implementierung, welche zur Kommunikation RESTful HTTP einsetzt. Im Kontrast zu verbreiteten MPI-Systemen, deren hauptsächlicher Fokus auf dem Einsatz in homogenen und eng gekoppelten Rechnersystemen liegt, bietet ein solcher Ansatz die Möglichkeit, mit MPI umgesetzte parallele Programme in einem Umfeld zu betreiben, dessen Charakteristiken denjenigen des World Wide Web gleichen. Die Arbeit baut auf einer bereits bestehenden prototypischen Implementierung eines solchen Systems auf, die jedoch konzeptionelle Mängel aufweist. Neben der theoretischen Betrachtung von RESTful HTTP im MPI-Umfeld wird darauf aufbauend dieses Basissystem grundlegend überarbeitet und REST-Konformität hergestellt. Nach einer Bewertung dieser Implementierung und einer Analyse von möglichen Optimierungen erfolgen Performance-Messungen und der exemplarische Einsatz des Systems für ein reales Problem des wissenschaftlichen Rechnens.

Vorwort

Der *Message Passing Interface* Standard, kurz *MPI*, stellt nach wie vor eine der meist genutzten Technologien im Kontext des parallelen Rechnens in verteilten Adressräumen dar. Eingesetzt werden die gebräuchlichen Implementierungen dieses Standards jedoch fast ausschließlich auf eng gekoppelten und sehr homogenen Computersystemen, wie Clustern oder dedizierten Parallelrechnern. Die meisten großen MPI-Software-Pakete sind historisch gewachsen oder bauen auf Implementierungen auf, welche ihre Wurzeln – ebenso wie der MPI-Standard selbst – am Anfang der neunziger Jahre des letzten Jahrhunderts haben. Zu diesem Zeitpunkt waren ausreichend schnelle Netze und große Rechenkapazitäten, die parallelen Programmen für verteilte Adressräume überhaupt erst einen praktischen Nutzen verleihen, örtlich nur sehr lokal, nämlich meist nur an Forschungseinrichtungen und Universitäten, sowie bei größeren Unternehmen, verfügbar. Schlagworte wie Portabilität, Heterogenität und lose Kopplung spielten somit für Entwicklungen im MPI-Umfeld, schon allein historisch begründet, eine eher untergeordnete Rolle.

Der technologische Fortschritt im Kontext des Internets und des Heimcomputers innerhalb der letzten Dekade lässt zum heutigen Zeitpunkt jedoch eine neue Sichtweise auf das parallele Rechnen mit MPI zu. Die Bandbreiten im Internet und die verfügbaren Rechenkapazitäten in Rechnerpools an Hochschulen, der IT bereits kleiner Unternehmen und sogar bei Heimcomputern in privaten Haushalten, machen klassische MPI-Anwendungen in einem für diese völlig neuartigen Umfeld denkbar. Hierdurch ergeben sich jedoch auch gänzlich neue Anforderungen an ein MPI-System, welche die klassischen Implementierungen nur sehr eingeschränkt erfüllen können.

Die vorliegende Arbeit baut auf einem Master-Projekt auf, welches im Wintersemester 2011/2012 an der Universität Bayreuth vom Autor durchgeführt wurde. Dieses behandelte die *Implementierung einer MPI-Bibliothek auf Basis von RESTful Webservices* [49]. Vorgestellt wurde eine MPI-Implementierung, mit welcher versucht wurde, ausgehend vom Architekturstil REST, konsequent den Anforderungen einer portablen MPI-Lösung gerecht zu werden, die sich auch für Rechnernetze mit internetähnlichen Charakteristiken

eignet. Als Basis diente hierzu eine bereits vorhandene MPI-Lösung, welche komplett in Java realisiert ist und somit bereits per se ein hohes Maß an Portabilität aufweist: *MPJ Express*. Diese Software wurde mithilfe des *JAX-RS*-Frameworks *JBoss RESTEasy* um die Möglichkeit erweitert, seine Interprozesskommunikation komplett über das HTTP-Protokoll abzuwickeln. Einen Rückblick auf die Ausführungen des Master-Projekts und den Stand der geleisteten Implementierung, sowie den daraus hervorgegangenen Ergebnissen liefert Kapitel 2. Hervorzuheben ist, dass das Projekt einen sehr praktischen Charakter ohne weitreichendes theoretisches Fundament hatte. Die vorgestellte Implementierung ist deswegen hauptsächlich als Proof-of-Concept zu betrachten.

Ziel dieser Arbeit ist deswegen zum einen, im ersten Teil eine theoretische Basis zur Realisierung von MPI-Kommunikation über RESTful HTTP zu schaffen und im zweiten Teil die Implementierung entsprechend anzupassen. Zunächst wird hierzu in Kapitel 1 die Wahl der Technologiekombination REST und HTTP hinterfragt, die beiden Begriffe analysiert und eine mögliche Umsetzung der Grundprinzipien von REST formuliert. Nach dem erwähnten Rückblick auf das Master-Projekt in Kapitel 2, erfolgt in Kapitel 3 eine Analyse der Anforderungen, welche eine verteilte Anwendung, wie ein MPI-System, hat, die in einem internetähnlichen Umfeld eingesetzt wird. Den Abschluss des theoretischen Teils bildet in Kapitel 4 die Betrachtung ausgewählter vorhandener MPI-Implementierungen und die Abgrenzung des entwickelten Systems zu diesen.

Zum anderen wird die konkrete Realisierung der im ersten Teil theoretisch formulierten Umsetzungsmöglichkeiten der REST-Prinzipien dargestellt. Hierfür muss zunächst eine detaillierte Betrachtung des Basissystems in Kapitel 5 erfolgen. Diese geht über die Ausführungen in der Ausarbeitung des Master-Projekts hinaus. Darauf aufbauend leistet Kapitel 6 die Darstellung von Implementierungsdetails zu den umgesetzten Prinzipien der REST-konformen Architektur des entwickelten Systems. Die aufgestellten Anforderungen finden in Kapitel 7 Betrachtung. Dabei geht es einerseits darum, zu zeigen, welche der Anforderungen vom entwickelten System durch bloße Herstellung von REST-Konformität erfüllt werden können. Andererseits ist die Bewertung des Nutzens des Architekturstils REST im MPI-Umfeld Gegenstand dieses Kapitels. Einzelne darüber hinausgehende Optimierungen und Erweiterungen der Implementierung werden in Kapitel 8 betrachtet. Zur Bewertung der Performance des Systems erfolgt in Kapitel 9 die Darstellung von synthetischen Benchmarkergebnissen im Vergleich zur Projekt-Implementierung und einer Referenz-MPI-Implementierung. Den Abschluss des zweiten Teils dieser Arbeit bildet Kapitel 10. Dieses behandelt den Einsatz der realisierten Implementierung für ein reales Problem des wissenschaftlichen Rechnens: Die

Lösung von Systemen gewöhnlicher Differentialgleichungen mithilfe eines eingebetteten Runge-Kutta-Verfahrens. Auch hier werden Performancemessungen präsentiert.

Teil I.

Theoretischer Teil

1. RESTful HTTP im MPI-Kontext

Im Folgenden soll zunächst ein Einblick in die Überlegungen gegeben werden, welche der Idee, den Architekturstil REST auf der einen Seite und das Netzwerkprotokoll HTTP auf der anderen als Basis einer Implementierung des MPI-Standards einzusetzen, zugrunde liegen. Hierzu erfolgt in den Abschnitten 1.1 und 1.2 eine theoretische Aufarbeitung der Hintergründe und Qualitäten von HTTP, sowie derer von REST, jeweils bezogen auf das MPI-Umfeld. Ein Ziel dieser beiden Abschnitte ist, zu verdeutlichen, wie gut sich die beiden Technologien aus theoretischer Sicht für das Message Passing eignen. Im Anschluss daran wird die konkrete Umsetzbarkeit der fünf Grundprinzipien einer REST-konformen Architektur als Facetten der MPI-Kommunikation analysiert.

1.1. HTTP als Protokoll

Das Hypertext Transfer Protocol wird im Vorwort seiner Spezifikation in der Version 1.1 [9] folgendermaßen charakterisiert:

> The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, protocol which can be used for many tasks beyond its use for hypertext [...].

Dies ist offensichtlich eine sehr allgemein gehaltene Einordnung. Daraus hervor geht jedoch bereits an dieser Stelle eine relativ universale angedachte Einsetzbarkeit von HTTP. Alle Strukturen des Protokolls sind in einer Weise spezifiziert, dass sie für eine Vielfalt von Anwendungen ihre Gültigkeit behalten, solange diese nur grob der Charakterisierung „distributed, collaborative, hypermedia information system“ entsprechen. Hervorzuheben ist an dieser Stelle vor allem, dass auch ein MPI-System im Grunde dieser Struktur folgt: Die verteilte Ausführung einer Anwendung („distributed“) ist die klassische Grundmotivation zum Einsatz von Message Passing. Auch die Zusammenarbeit der verschiedenen Prozesse („collaborative“) ist durch den Austausch von Nachrichten jedem

nicht-trivialen MPI-Programm automatisch innewohnend. Lediglich die Verwendung von Verlinkungen zwischen Ressourcen zur Fortführung des Applikationsstatus („hypermedia“) ist kein ohne Weiteres sichtbares Element eines klassischen MPI-Systems. In Abschnitt 1.3 soll jedoch aufgezeigt werden, dass sich auch dieses Prinzip theoretisch auf MPI-Kommunikation übertragen lässt.

Die Basis von HTTP ist ein simples Anfrage-Antwort-Schema, aufbauend auf wenigen sogenannten HTTP-Verben, welche für Operationen stehen, die auf Ressourcen ausgeführt werden können. Beispielsweise ruft ein Client ein `GET` auf der Ressource `http://server.org/ressource` auf. Hierbei kann dieser unter anderem über den Mechanismus der *Content Negotiation* festlegen, welches Repräsentationsformat einer möglichen Antwort er akzeptiert. Falls nun die angefragte Ressource existiert und in einem passenden Format vorliegt, antwortet der Server mit einer Nachricht, die neben einigen weiteren Header-Informationen den Status `HTTP/1.1 200 OK` und die Daten der Ressource selbst enthält.

Fakt ist, dass HTTP auf Anwendungsebene für das, was allgemein als Internet bezeichnet wird, als hauptsächliches Kommunikationsprotokoll eingesetzt wird. Die Anzeige von Webseiten in Browsern und das Navigieren durch Hyperlinks basiert fast ausschließlich auf HTTP. Die Nutzung eines Browsers zum Abruf von Informationen ist zweifelsohne die Anwendung, für die das Internet in der Masse eingesetzt wird. Aus diesem Grund ist die technische Infrastruktur des World Wide Web größtenteils darauf ausgerichtet, HTTP-Kommunikation möglichst effizient zu ermöglichen. Technologien wie transparentes Caching ermöglichen die Beschleunigung der Verarbeitung und Beantwortung von HTTP-Anfragen, ohne dass dies der Anwender explizit berücksichtigen muss. Allein der in einem solch großen Umfang stattfindende Einsatz des Hypertext Transfer Protocols stellt somit einen gewichtigen Grund dar, warum Anwendungen, welche sich im Internet – oder in einem ähnlich strukturierten Netz – bewegen, von der Nutzung von HTTP profitieren können. Ein Szenario, welches auf das hier zu entwickelnde System offensichtlich zutrifft.

Ein weiterer nicht zu vernachlässigender Vorteil beim Einsatz von HTTP ist die automatisch gute Verträglichkeit mit vielen vorhandenen Firewalls, welche in weit verteilten Anwendungen potentiell Kommunikation blockieren könnten. Da der für HTTP genutzte Port in Firewalls, die an das Internet angeschlossen sind, häufig geöffnet ist, um die Nutzung eines Webbrowsers zu ermöglichen, muss an dieser Stelle – zumindest für ausgehende Verbindungen – oft nichts zusätzlich konfiguriert werden.

Im folgenden Abschnitt wird erläutert, wieso der Architekturstil REST zur Implementierung des Systems ausgewählt wurde. In diesem Kontext sei hervorgehoben, dass hierfür die Wahl von HTTP als grundlegendes Protokoll auf Anwendungsebene als alternativlos angesehen werden kann. REST ist zwar theoretisch unabhängig von einem Protokoll, wird jedoch zum einen ausgehend von HTTP definiert und zum anderen in keiner dem Autor bekannten Anwendung ohne dieses eingesetzt. Außerdem setzen alle Frameworks zur Implementierung von RESTful Webservices, ebenso wie das hier eingesetzte RESTEasy von JBoss, auf die Kommunikation per HTTP.

1.2. REST als Architekturstil

Der Begriff REST steht für *Representational State Transfer* und geht zurück auf Roy Fielding, welcher im Jahr 2000 den damit von ihm benannten Architekturstil als grundlegenden Erfolgsfaktor des World Wide Web identifizierte [26]. Heute wird von REST hauptsächlich im Kontext von Webservices gesprochen, zusammen mit anderen Begriffen aus diesem Umfeld, wie beispielsweise SOAP. Häufig stehen sich in Diskussionen über die Vor- und Nachteile von REST und SOAP diese direkt als Kontrahenten gegenüber. Ein solcher Vergleich hat jedoch wenig Sinn, da die beiden Begriffe auf völlig verschiedenen Ebenen anzusiedeln sind: REST bezeichnet einen bloßen Architekturstil, also eine theoretische Anhäufung von Regeln, Prinzipien und Vorgehensweisen ohne direkte Verknüpfung zu einer konkreten Technologie. Die auf XML basierende Webservice-Technik SOAP dagegen ist ein exakt spezifiziertes Kommunikationsprotokoll, welches zusammen mit der Beschreibungssprache WSDL direkt eingesetzt werden kann. Insofern ist der Begriff eines RESTful Webservice wesentlich weniger genau definiert und kann auf eine Vielzahl von Implementierungsvarianten zutreffen als der eines SOAP-Webservice, welcher beispielsweise immer auf der Kommunikation mit auf eine bestimmte Weise strukturierten XML-Dokumenten aufbaut. Das kann jedoch auch direkt als Vorteil von REST gesehen werden, da es durch diese Freiheit möglich ist, durch die Bindung an bestimmte Technologien aufgezwungene Nachteile, wie beispielsweise den relativ hohen durch XML implizierten Kommunikationsoverhead von SOAP, zu vermeiden. Andererseits ist hervorzuheben, dass der sehr abstrakte Charakter von REST bereits dazu geführt hat, dass es viele als RESTful bezeichnete Dienste im Web gibt, welche es streng genommen nicht sind.

Laut Fielding ist das World Wide Web eine konkrete Ausprägung von REST unter Verwendung von HTTP als Kommunikationsprotokoll. Er identifiziert – aus technischer

Sicht – REST als die Basis für den großen Erfolg der Architektur des Webs. In [26] sind die Gründe hierfür detailliert dargelegt. Aus der Arbeit Fieldings geht hervor, dass eine verteilte Anwendung, welche in einem dem Internet sehr ähnlichen Umfeld oder dem Internet selbst läuft und HTTP auf Anwendungsebene zur Kommunikation einsetzt, möglichst nach den Prinzipien von REST aufgebaut sein sollte. Auf diese Weise können vorhandene Strukturen optimal ausgenutzt werden und die Anwendung skaliert automatisch ähnlich gut wie das Web selbst. Die hier entwickelte Software hat abstrakt betrachtet genau den beschriebenen Charakter, womit eine Architektur, die den REST-Prinzipien folgt – aufbauend auf den stimmigen Schlussfolgerungen Fieldings – als optimal für das System erscheint.

Hervorzuheben ist außerdem, dass RESTful HTTP – wie bereits in 1.1 erwähnt – im Moment die einzig praktikable Ausprägung des Architekturstils darstellt. Fielding ließ die erkannten Prinzipien von REST darüber hinaus direkt als Mitautor in die Spezifikation des HTTP-1.1-Standards [9] einfließen. Hieraus ergibt sich, schon allein aus der Entstehungsgeschichte von REST und HTTP 1.1, eine optimale Kombination dieser, da beide Einflüsse des jeweils anderen beinhalten und teilweise aus diesem hervorgehen.

1.3. Umsetzbarkeit der REST-Prinzipien für MPI auf Basis von HTTP

Nach der Darstellung der Gründe für den Einsatz von RESTful HTTP erfolgt nun eine Analyse der fünf durch Fielding definierten Prinzipien des Architekturstils REST in Bezug auf ihre theoretische Umsetzbarkeit in einem MPI-System, welches über HTTP kommuniziert. Hierbei werden bereits konkrete Architekturentscheidungen formuliert, auf die in der in Teil II beschriebenen konkreten Implementierung Bezug genommen wird.

Prinzip 1 (Ressourcen mit eindeutiger Identifikation)**.** *Eine Architektur, die RESTful ist, beinhaltet verschiedene Ressourcen. Außerdem müssen diese Ressourcen eindeutig identifiziert und damit angesprochen werden können.*

Nach Tilkov sollten alle Objekte einer Implementierung, die „es wert sind“ [56], als Ressourcen exponiert werden. Der Wert eines Objekts und damit seine Eignung zur Exponierung ist dabei anwendungsabhängig und bestimmt sich vor allem darüber, ob es Teil von externer Kommunikation sein muss. Als grundlegende Ressource eines MPI-Systems

lässt sich somit offensichtlich eine MPI-Nachricht identifizieren. Eine solche bildet die Grundlage jeder Kommunikationsoperation von MPI auf der Anwendungsebene.

Eine eindeutige Identifikation von Ressourcen ist bei HTTP naheliegend und relativ einfach durch das Konzept des *Uniform Resource Identifiers (URI)* [7] möglich. Durch diesen hat man einen globalen Namensraum mit unendlich vielen Adressen zur Verfügung. Konkret auf MPI bezogen, muss festgelegt werden, wie die URI einer Nachricht aufgebaut wird. Hierzu ist es notwendig, diejenigen Merkmale zu identifizieren, die eine Nachricht eindeutig machen. Bei MPI sind dies der Sender und der Empfänger der Nachricht zusammen mit einem vergebenen Tag und die Nummer des Kommunikator-Kontextes. Aufbauend darauf wird die URI einer MPI-Nachricht im hier dargestellten System folgendermaßen definiert:

```
http://<prozess>/REST-MPI/mpi/messages/<sender>/<receiver>/<context>/<tag>
```

Den Tag als letztes Element des Pfades zu definieren hat den Hintergrund, dass MPI bei der Angabe desselben beim Empfangen einer Nachricht ein Wildcard-Element erlaubt, womit gezielt beliebige Tags akzeptiert werden können. Dies kann in obiger Struktur auf einfache Art und Weise durch Weglassen des letzten Elements der URI auf die entsprechende abstrakte Ressource abgebildet werden. Die Reihenfolge der restlichen Elemente ist im Prinzip beliebig und kann in der Tat im entwickelten System flexibel geändert werden.

Prinzip 2 (Standardmethoden)**.** *Jede Ressource stellt nach außen hin die gleiche Schnittstelle zur Verfügung.*

Dies bedeutet, dass die Funktionen, die auf exponierten Ressourcen eines RESTful Webservice aufgerufen werden können, immer die gleichen sind. Impliziert wird dabei jedoch nicht, dass jede Funktion, welche die Schnittstelle anbietet, auch für jede Ressource sinnvoll ist oder überhaupt benötigt wird, sondern nur, dass diese verfügbar ist und keine über die einheitliche Schnittstelle hinausgehenden Funktionen angeboten werden. Ein solches Konzept macht Mechanismen zur Erlangung von Informationen über die angebotenen Schnittstellen von Services, wie beispielsweise das bei SOAP eingesetzte WSDL, größtenteils überflüssig.

Wird, wie hier, HTTP als grundlegendes Protokoll für REST eingesetzt, beinhaltet die einheitliche Schnittstelle, die jede Ressource anbietet, die bereits erwähnten HTTP-Verben, wie zum Beispiel `GET`, `PUT`, `POST` und `DELETE`. Für die hier vorliegenden Ressourcen, die bei Prinzip 1 definierte Exponierung auf Nachrichtenebene, sind lediglich

zwei dieser Verben interessant: `GET` und `PUT`. Das erste für *Pull-*, das zweite für *Push-*Kommunikation zwischen zwei MPI-Prozessen. An dieser Stelle sei erwähnt, dass alle dem Autor bekannten MPI-Implementierungen lediglich eine dieser beiden Richtungen, nämlich die Push-Variante, einsetzen. Neben aktivem Senden auch aktives Empfangen – die Pull-Richtung – im entwickelten System zuzulassen, ist eine Entscheidung, die getroffen wird, um möglichst viele Facetten einer RESTful Architektur auszunutzen. Darüber hinaus ist es gerade das HTTP-`GET`, das am meisten von vorhandener Infrastruktur, wie transparentem Caching, profitieren kann.[1]

Konkret wird im Prinzip ein `GET` auf der URI einer bestimmten Nachricht, durch eine Variante von *MPI-Receive* angestoßen, ein `PUT` durch eine Variante von *MPI-Send*. Da sich beide Aufrufe für eine Nachricht überschneiden können, muss natürlich eine Art Synchronisierung der beiden Richtungen stattfinden. Näheres hierzu ist in Teil II zu finden.

Prinzip 3 (Unterschiedliche Repräsentationen)**.** *Jede Ressource kann mehrere verschiedene Repräsentationen haben.*

Ein einfaches Beispiel zur Veranschaulichung des Nutzens von verschiedenen Repräsentationen ein und derselben Ressource ist das einer Artikelliste eines Webshops. Angenommen eine solche Liste ist als Ressource exponiert. Ein Client könnte nun genau die Repräsentation anfordern, die er für die aktuelle Anwendung benötigt. Denkbare Repräsentationen wären beispielsweise eine strukturierte XML-Datei zur programmatischen Verarbeitung, eine HTML-Version zur Darstellung in einem Webbrowser und eine PDF-Datei zur Druckansicht der Artikelliste. Bei der Benutzung von HTTP ist ein solches gezieltes Abfragen einer Repräsentation durch den Mechanismus der *Content Negotiation* unter Angabe des gewünschten *MIME-Types* möglich.

Bezogen auf MPI ist die Umsetzung dieses Prinzips weniger offensichtlich. Die Java-Objekte, die im entwickelten System MPI-Nachrichten repräsentieren, werden zur Übertragung serialisiert und als Byte-Array in den *HTTP-Entity-Body* eingebettet. Zwar wäre es auch hier möglich, die gleiche Nachricht zum Beispiel als XML bereit zu stellen, der praktische Nutzen dieser zusätzlichen Funktionalität wäre jedoch fraglich.

Ein Problem, für dessen Lösung das Prinzip 3 jedoch sinnvoll genutzt werden könnte, entspringt der gewünschten Heterogenität des entwickelten Systems: Es kann durchaus vorkommen, dass mehrere Prozesse, die Teil eines MPI-Verbunds sind, unter verschiede-

[1] Details zu der erhöhten Komplexität der Implementierung und den damit verbundenen Problemen, welche das Einführen einer zweiten Kommunikationsrichtung provoziert, werden in Teil II dieser Arbeit behandelt.

nen Versionen der *Java Virtual Machine* laufen. In diesem Fall wäre das Deserialisieren von in anderen Prozessen serialisierten Objekten mit der implementierten Methode teilweise nicht möglich. Deswegen wäre es denkbar, bei einer HTTP-Anfrage ein passendes Serialisierungsformat anzugeben und das Nachrichten-Objekt entsprechend zu verarbeiten.[2]

Prinzip 4 (Statuslose Kommunikation)**.** *Der Applikationsstatus wird entweder vom Client gehalten oder in einen Ressourcenstatus umgewandelt.*

Hintergrund dieses Prinzips ist zum einen die Erhöhung der Skalierbarkeit eines RESTful Webservice. Ohne die Notwendigkeit für jeden Client Sessioninformationen zu speichern, macht sich ein Service in gewisser Weise unabhängig von der Clientanzahl. Zum anderen führt dieses Prinzip auch zu einer loseren Kopplung zwischen Client und Server, was gerade in weit verteilten Anwendungen von Vorteil ist.

Im entwickelten System stellt jeder MPI-Prozess einen RESTful Webservice zur Verfügung und interagiert mit denjenigen der anderen Prozesse. Somit ist jeder Prozess gleichzeitig Server und Client. Aufgrund dieser fehlenden Trennung der beiden möglichen Rollen, ist Prinzip 4 nicht sinnvoll anwendbar, beziehungsweise würde die forcierte Anwendung desselben keinen Unterschied in der Gestaltung der Kommunikation ausmachen.

Prinzip 5 (Hypermedia)**.** *Der Applikationszustand wird durch Verknüpfungen fortgeführt.*

Fielding nennt eines der seiner Meinung nach wichtigsten Prinzipien [27] von REST *Hypermedia as the engine of application state (HATEOAS)* [26]. Hintergrund dieses Konzepts ist das Ziel, dass ein Client ohne weitere Informationen, lediglich mit der Kenntnis der URI eines Einstiegspunktes, mit dem RESTful Webservice eines Servers interagieren kann. Informationen über die Identifikation von zusätzlichen Ressourcen werden als Verknüpfungen vom Server bereitgestellt oder sind in den übertragenen Daten kodiert.

Streng genommen widerspricht die in Prinzip 1 definierte URI-Struktur für MPI-Nachrichten diesem Konzept, falls angenommen wird, dass diese bereits vor der Programmausführung einem Client bekannt ist. Nach [27] dürfte sich in diesem Fall die angebotene API nicht RESTful nennen. Dieses Problem lässt sich jedoch durch ein einfaches Werkzeug lösen: Dem Mechanismus der URI-Templates [3]. Mit dessen Hilfe lässt

[2] Diese Funktionalität ist in der aktuellen Version des Systems nicht implementiert.

sich eine URI-Struktur definieren, die der dargestellten gleicht. Diese kann den Clients am Eintrittspunkt des Services in einem *HTTP-Link-Header* übermittelt werden. Somit sind keine weiteren *off-the-band* Informationen notwendig und die Clients können auf Basis der Vorlage die entsprechenden konkreten URIs von MPI-Nachrichten konstruieren. Denkbar wäre durch einen solchen Mechanismus sogar die Verwendung verschiedener Pfadstrukturen auf unterschiedlichen Servern des gleichen MPI-Verbunds.

2. Rückblick auf das Master-Projekt

Wie bereits im Vorwort erwähnt, basiert diese Arbeit auf einem Master-Projekt mit dem Titel *Implementierung einer MPI-Bibliothek auf Basis von RESTful Webservices* [49], das im Wintersemester 2011/2012 an der Universität Bayreuth durchgeführt wurde. Die in diesem Zusammenhang entwickelte Software bildet die Basis der hier dargestellten Implementierung und soll deswegen im Folgenden rückblickend betrachtet werden. Hierzu wird zunächst auf das zugrunde liegende Konzept eingegangen. Auf die anschließende Darstellung der konkreten Umsetzung folgt ein Einblick in die erzielten Ergebnisse. Die Betrachtung erfolgt dabei auf einer relativ abstrakten Ebene. Konkrete Implementierungsdetails auf Höhe des Programmcodes werden nicht dargelegt. Für tiefere Ausführungen hierzu sei einerseits erneut auf die Ausarbeitung des Master-Projekts [49] und andererseits besonders auf Kapitel 5 in Teil II dieser Arbeit verwiesen. Ein Ziel des aktuellen Kapitels ist, eine minimale Vergleichsbasis für Teile dieser Arbeit zu schaffen, in denen aufgezeigt wird, inwiefern sich die aktuelle Implementierung des Systems von der Version des Master-Projekts unterscheidet.

2.1. Konzept

Ziel des Projekts war es, ein funktionierendes MPI-System zu realisieren, das möglichst portabel ist und zur Kommunikation lediglich das HTTP-Protokoll in Form von RESTful Webservices verwendet. Wie bereits im Vorwort erwähnt, war eine Maxime des Projekts, möglichst effektiv eine gewisse Funktionalität zu erreichen. Da der Ansatz, RESTful HTTP für MPI einzusetzen, ein völlig neuer war und aus diesem Grund auf keine Erfahrungswerte zurückgegriffen werden konnte, war die Entscheidung, in einem ersten Schritt ein Proof-of-Concept-System ohne weitreichende theoretische Basis zu implementieren, von pragmatischer Natur. Auf diese Weise konnte frühzeitig die Idee, die genannten Technologien miteinander zu kombinieren, evaluiert und eine Aussage darüber getroffen werden, ob ein solcher Ansatz überhaupt Potential für eine tiefere Ausarbeitung bietet.

Im Allgemeinen ist es nicht selbstverständlich für eine solch neuartige Architektur, oh-

ne eine tiefere theoretische Behandlung der Hintergründe, einen aussagekräftigen Prototypen entwickeln zu können. Ein solches Vorgehen kann schnell zu grundlegend falschen Designentscheidungen führen, welche, nachdem sie ans Licht kommen, aufwändig korrigiert werden müssen – eventuell bis hin zu einem gezwungenen Neubeginn der Implementierung von Grund auf.

Glücklicherweise konnte jedoch das Risiko, solch grundlegend falsche Entscheidungen zu treffen, dadurch minimiert werden, dass die Möglichkeit bestand, vorhandene Softwarekomponenten relativ einfach miteinander zu verbinden und damit ohne großen konzeptionellen Aufwand die Architektur des Gesamtsystems festzulegen. Im einzelnen konnte das komplett in Java geschriebene MPI-System *MPJ Express* innerhalb eines *JBoss Application Servers* direkt mit *RESTEasy*, einer Implementierung des *JAX-RS*-Standards, kombiniert werden. Auf diese Weise konnten Synergieeffekte genutzt werden und die zusätzlich zu den vorhandenen Teilen notwendige Implementierung befand sich auf einer Basis, welche zunächst keiner weiteren Hinterfragung bedurfte, da die einzelnen Komponenten für sich innerhalb ihrer eigenen Domäne durchaus als ausgereift und führend bezeichnet werden konnten.

Ein Vergleich der verschiedenen verfügbaren MPI-Systeme für Java findet sich ebenfalls in [49]. Die Wahl von MPJ Express im Kontext der zu entwickelnden Anwendung präsentiert sich aus diesem Vergleich hervorgehend als alternativlos. Darüber hinaus erlaubt seine Schichtenarchitektur, welche in Abbildung 2.1 dargestellt ist, eine relativ unkomplizierte Erweiterung der Implementierung mit, für diese, neuen Kommunikationstechnologien wie RESTful HTTP.

Der JBoss Application Server[1] ist ein Anwendungsserver, der nach dem JavaEE-Standard [6] implementiert ist. Dieser enthält – bereits aufgrund der Spezifikation – neben einem Webserver einen Servlet-Container, in welchem nahezu beliebiger Java-Code ausgeführt werden kann. Die Entscheidung, den Application Server von JBoss anderen JavaEE-Anwendungsservern, wie beispielsweise dem GlassFish-Server von Oracle [21], vorzuziehen, wurde deshalb getroffen, da in ersterem mit RESTEasy bereits eine vollständige Implementierung des JavaEE-Standards für RESTful Webservices – JAX-RS – enthalten ist.

Der erwähnte Servlet-Container bietet im Kontext des angestrebten Systems die Möglichkeit, die vollständige MPI-Implementierung als Webanwendung innerhalb des Applikationsservers zu deployen und diesem somit alle Vorteile eines solchen Servers zugänglich zu machen. Im Einzelnen sind dies unter anderem Management- und Informationsfunk-

[1] Verwendete Version: 7.1.1.

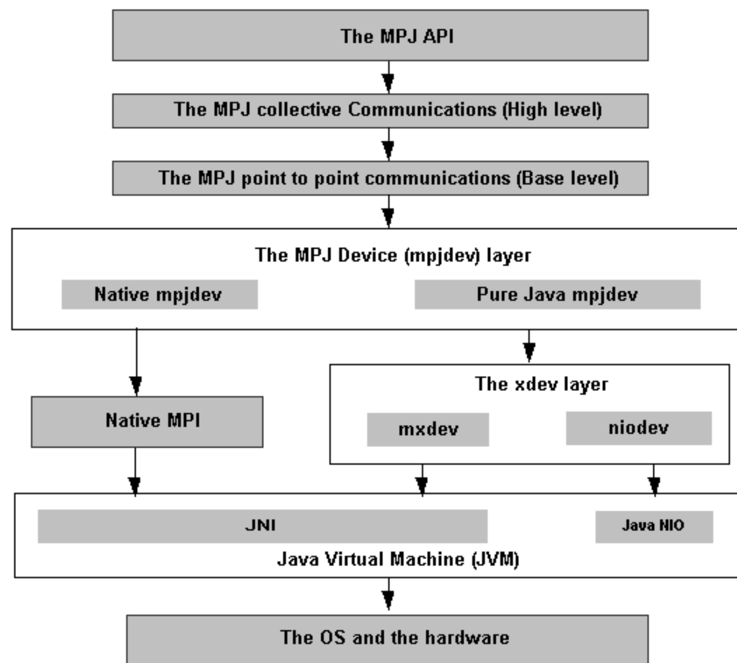


Abbildung 2.1.: Architektur von MPJ Express

tionen, wie Monitoring und Logging, und Stabilitäts-, Sicherheits- und Kommunikationsfunktionen. Letztere sind in der Ausprägung des bereits erwähnten RESTEasy-Systems für die vorliegende Implementierung besonders wichtig.

Konkret findet sich der Ansatzpunkt für die erfolgte Erweiterung von MPJ Express in der Schichtendarstellung in Abbildung 2.1 auf Höhe des sogenannten *xdev layer*. Diese Ebene beinhaltet in der offiziellen Ausgabe von MPJ Express[2] momentan drei verschiedene Implementierungen sogenannter *Devices*, welche jeweils Ausprägungen konkreter Technologien zur Interprozesskommunikation darstellen. Es liegen Lösungen zur Nutzung von Myrinet- und Ethernet-Netzwerken, sowie eine Umsetzung auf Basis von Java-Threads – zur Anwendung von MPI auf einem einzigen (Multicore-)Rechner – vor. Um nun die Möglichkeit zu bieten, RESTful HTTP zur Interprozesskommunikation zu benutzen, wurde ein weiteres solches Device, das *RESTDevice*, implementiert. Dieses greift direkt auf die Dienste von RESTEasy zu, welche vom umgebenden Anwendungsserver zur Verfügung gestellt werden.

[2]Verwendete Version: 0.38.

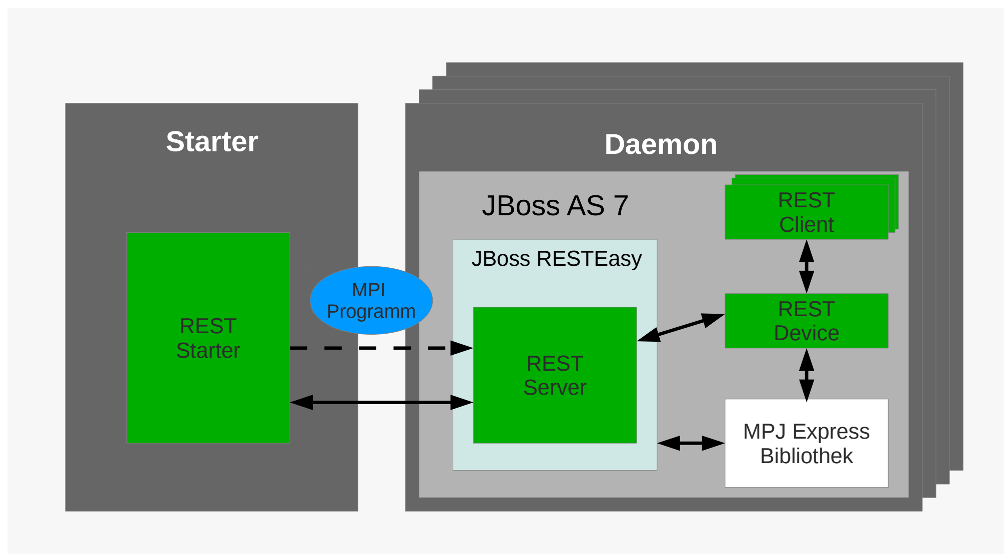


Abbildung 2.2.: Architektur des entwickelten Systems

2.2. Umsetzung

Die durchgehende Verwendung von Java als Programmiersprache der Wahl und das Vorliegen der einzelnen zu verbindenden Systeme in dieser erwies sich als großer Vorteil bei der Implementierung. Auf diese Weise konnten die Einzelkomponenten ohne die Notwendigkeit von Adapterfunktionen direkt miteinander kombiniert werden. Ein Überblick über die realisierte Systemarchitektur findet sich in Abbildung 2.2. Hervorzuheben ist, dass in der Tat alle Komponenten des zugrunde liegenden MPI-Systems innerhalb des JBoss Application Servers angesiedelt sind. Dieser stellt somit eine Art Daemon dar, welcher auf jedem teilnehmenden Rechner des MPI-Verbunds ausgeführt wird.

Konkret ist die zentrale Komponente zur Abwicklung der Kommunikation über RESTful HTTP die nach dem Singleton-Design-Pattern implementierte Klasse `RESTDevice`. HTTP-Nachrichten werden durch den `RESTServer` – ebenfalls eine Singleton-Klasse, welche den eigentlichen Webservice mit Hilfe der angebotenen Funktionen von RESTEasy implementiert – an die jeweilige Instanz des `RESTDevice` zur Verarbeitung weitergegeben. Außerdem stellt das `RESTDevice` die Schnittstelle zur abstrakteren *MPJDev*-Schicht von MPJ Express (siehe Abbildung 2.1) bereit, indem sie die grundlegenden Operationen zum blockierenden und nichtblockierenden asynchronen und synchronen Senden und zum blockierenden und nichtblockierenden Empfangen implementiert. Beim Senden tritt das `RESTDevice` in der Rolle eines HTTP-Clients auf. Diese Funktionalität ist in der Ver-

sion des Master-Projekts durch die Nutzung von sogenannten *Client-Proxy*-Objekten, welche ebenfalls von RESTEasy bereitgestellt werden, realisiert.

Eine Sonderrolle spielt die Klasse `RESTStarter`, welche nur in der Startphase des Systems in Aktion tritt. Ihre Aufgabe ist zum einen die Initialisierung, welche unter anderem eine Art Handshake mit allen teilnehmenden Prozessen zur Überprüfung ihrer Erreichbarkeit beinhaltet. Zum anderen überträgt der `RESTStarter` das letztendliche MPI-Programm an alle erfolgreich validierten Prozesse und stößt dort dessen Ausführung an. Ausgeführt wird er – wie bei MPI üblich – über ein Shell-Skript (oder dessen Äquivalent auf einem Nicht-Unix-System) namens `mpjrun`. Der korrekte Aufruf lautet:

```
mpjrun -dev <device> -np <#processes> <program>.[class|jar]
```

Dabei wird das zu verwendende Device – hier `restdev` –, die Anzahl an Prozessen, sowie das mit der MPJ-API geschriebene MPI-Programm angegeben. Die Verwendung eines Skripts zum Start des parallelen Programms widerspricht zum Teil der geforderten Plattformunabhängigkeit des Systems. Diese Option wird jedoch aus Gründen einer leichteren Bedienung des Systems für Benutzer mit MPI-Erfahrung trotzdem angeboten. Abgesehen davon kann die Ausführung des MPI-Programms mit ähnlichen Parametern ebenfalls durch direktes Starten der `main`-Methode der Java-Klasse `RESTStarter` angestoßen werden.

Der von jedem Server angebotene RESTful Webservice implementiert in der Version des Master-Projekts das folgende, unter Verwendung der von RESTEasy bereitgestellten Annotationen definierte, Interface:

```
@Path("/mpi")
public interface RESTResource {

        @GET
        @Produces("text/plain")
        public String getNodeInformation();

        @PUT
        @Path("init")
        @Consumes("text/plain")
        public String initNode(String nodeId);

        @PUT
        @Path("processes")
        @Consumes("text/plain")
        public String receiveProcesses(String processes);
```

```
        @POST
        @Path("programs")
        @Consumes("text/plain")
        public String createProgram(byte[] file);

        @POST
        @Path("start")
        @Produces("text/plain")
        public String startProgram();

        @PUT
        @Path("messages/arrivequeue")
        @Consumes("text/plain")
        public String transferSendRequest(byte[] sendRequest);

        @PUT
        @Path("messages/synchronous")
        @Consumes("text/plain")
        public String signalSynchronousSend(byte[] key);
}
```

Die ersten fünf Methoden werden lediglich in der Startphase des Systems benötigt. Die letztendliche Kommunikation wird über die Ressource `mpi/messages/arrivequeue` und somit die Methode `transferSendRequest()` abgewickelt. Im Falle einer synchronen MPI-Kommunikationsoperation wird zusätzlich `signalSynchronousSend()` verwendet.

Hervorzuheben ist an dieser Stelle die Tatsache, dass durch das obige Interface zwar offenbar per RESTEasy eine HTTP-Schnittstelle zum Zugriff auf das MPI-System angeboten wird, die Struktur derselben aber nur in sehr begrenztem Umfang den in 1.3 aufgeführten REST-Prinzipien folgt. Im Grunde wird lediglich durch die Nutzung von HTTP implizit Prinzip 2 (Standardmethoden) und auf die in 1.3 geschilderte Weise zum Teil Prinzip 4 (Statuslose Kommunikation) erfüllt. Darüber hinaus wird durch die Entscheidung, die letztendliche Interprozesskommunikation über eine Ressource abzuwickeln, auf welcher nur ein HTTP-`PUT` ausgeführt werden kann, die Kommunikation auf die Push-Richtung begrenzt. Ein eigentlich aus Gründen optimaler Ausnutzung vorhandener Infrastruktur wünschenswertes `GET` wird an keiner Stelle verwendet. Aus diesen Gründen kann der im Master-Projekt implementierte Webservice streng genommen nicht als RESTful bezeichnet werden. Ziel der vorliegenden Arbeit ist deshalb unter anderem, diese Missstände zu korrigieren und einen *echten* RESTful Webservice zu realisieren.

2.3. Ergebnisse

Als das wichtigste Ergebnis des Master-Projekts lässt sich zunächst hervorheben, dass ein funktionsfähiges, auf reinem Java basierendes MPI-System realisiert wurde, welches zur Kommunikation nur das HTTP-Protokoll einsetzt. Durch erfolgreiches Ausführen der in MPJ Express integrierten Test Suite mit dem fertigen System wurde die Funktionstüchtigkeit aller beinhalteten MPI-Funktionen verifiziert. Somit wurde eine Möglichkeit geschaffen, mit der MPJ API implementierte parallele Programme in einem internetähnlichen Umfeld auf heterogenen Rechnern und weit verteilt auszuführen.

Abgesehen davon interessiert, obwohl die Optimierung dieser Facette nicht zu den Kernzielen des Projekts gehörte, die Performance des Systems im Vergleich zu anderen Implementierungen oder Kommunikationstechnologien. Um an dieser Stelle eine Aussage zu ermöglichen, wurde das entwickelte System, also MPJ Express innerhalb eines JBoss Application Servers mit der neu implementierten RESTDevice, welche per RESTEasy eine HTTP-Kommunikation realisiert, verglichen mit dem originalen MPJ Express System in der Cluster-Konfiguration auf Basis der vorhandenen Java-NIO-Device, welche auf Socket-Kommunikation setzt. Zur Messung der Leistung der verschiedenen MPI-Operationen wurde die Section 1 der *The Java Grande Forum MPJ Benchmarks* [20] verwendet. Die Testplattform bestand aus drei Desktop-Rechnern, welche verschiedene Linux-Versionen (Ubuntu 10.04 64bit, 10.10 32bit und 11.10 64bit) und verschieden schnelle Hardware (1,66Ghz Single Core Intel Atom mit 2GB, 2,4GHz Dual Core Intel Core i5 mit 3GB RAM und 2Ghz Intel Core2Duo mit 4GB RAM) einsetzten. Gesteuert wurde der Testaufbau von einem vierten Rechner aus. Als Verbindungsnetzwerk wurde ein 100MBit Fast Ethernet verwendet.

In Abbildung 2.3 und 2.4 sind exemplarisch die Messkurven für die beiden Operationen *Broadcast* und *Gather* dargestellt. Bei steigender Nachrichtengröße wurde die jeweils in einem bestimmten Zeitintervall übertragene Anzahl an Bytes gemessen. Die Abszissen sind dabei jeweils logarithmisch skaliert. Wie bereits zu erwarten war, liegt der Datendurchsatz der HTTP-Implementierung im Allgemeinen unter demjenigen der Socket-Variante. Der Grund hierfür ist der zusätzliche Overhead einer HTTP-Verbindung, welcher sich gerade bei sehr kleinen Nachrichtengrößen und der dadurch sehr hohen Anzahl an aufgebauten Verbindungen pro Zeitintervall deutlich zeigt. Bei sehr großen Nachrichten liegt die Performance dagegen nahezu auf dem Niveau der NIODevice. Für weitere Benchmarkergebnisse und zusätzliche Details sei an dieser Stelle erneut auf die Ausarbeitung zum Master-Projekt [49] verwiesen.

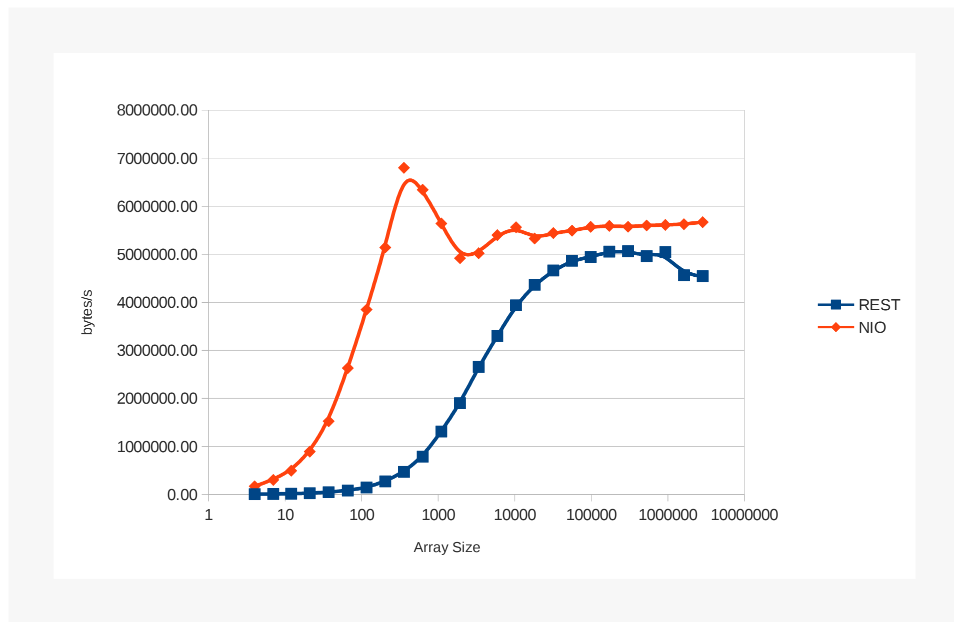


Abbildung 2.3.: Java Grande Forum MPJ Benchmarks, Section 1, Broadcast Double

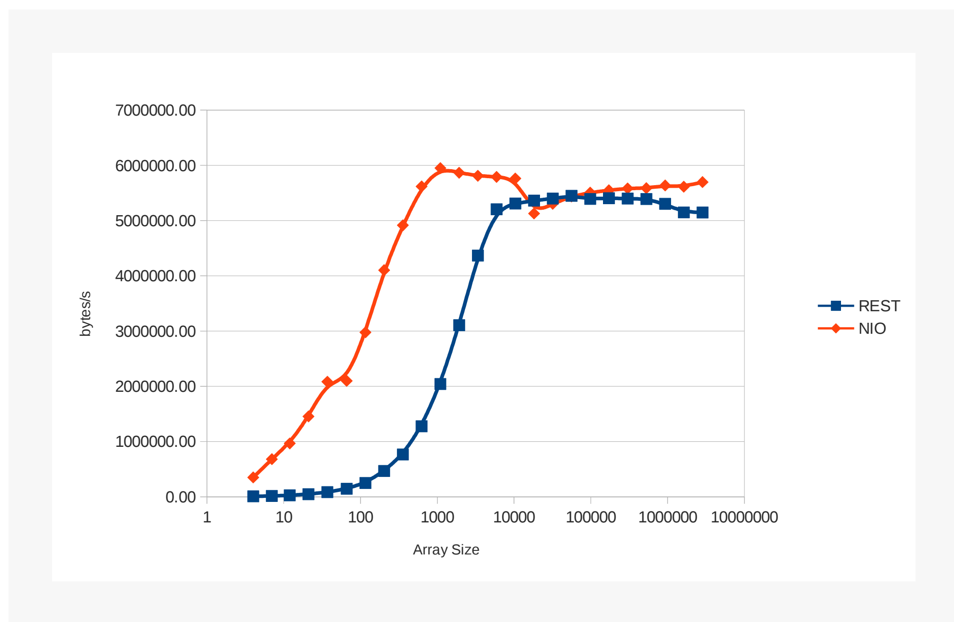


Abbildung 2.4.: Java Grande Forum MPJ Benchmarks, Section 1, Gather Double

3. Anforderungen an ein MPI-System zur Nutzung in internetähnlichen Rechnernetzen

In Kapitel 2 wurde ein System betrachtet, bei dessen Konzeption und anschließender Implementierung äußerst pragmatische Ziele, wie beispielsweise der Wunsch nach einem möglichst zeitnah lauffähigen Proof-of-Concept, verfolgt wurden. Kapitel 1 schuf durch eine Analyse der REST-Prinzipien und deren konkreter Umsetzbarkeit bereits einen ersten Teil der theoretischen Basis, die für eine konzeptionell mehr ausgereifte Implementierung notwendig ist. An diese Darlegungen soll nun durch die Formulierung von Anforderungen an ein MPI-System, welches in einem internetähnlichen Umfeld eingesetzt werden kann, angeknüpft und damit eine weitere große Lücke in den theoretischen Grundlagen der Implementierung aus dem Master-Projekt geschlossen werden.

Um solche Anforderungen aufstellen zu können, muss zunächst der Begriff des internetähnlichen Umfelds näher gefasst werden. Damit ist ein Anwendungsumfeld, also eine auf verschiedenen Systemebenen, wie Hardwarearchitektur, Softwareplattform und Netzwerktopologie, definierte Ausprägung eines verteilten Computersystems gemeint, welches ähnliche Eigenschaften hat, wie das World Wide Web. Um vor allem diejenigen konkreten Eigenschaften, die für das betrachtete System relevant sind, zu identifizieren, kann Bezug auf die bereits im Jahre 1994 von Peter Deutsch formulierten sieben Irrtümer des verteilten Rechnens [22] genommen werden, zu welchen James Goslig später einen achten hinzufügte [37]. Die postulierten acht Annahmen erweisen sich, falls sie für die Entwicklung eines verteilten Systems als Randbedingungen festgelegt werden, im Allgemeinen als falsch. Im Folgenden sollen nun diese Trugschlüsse einzeln betrachtet und jeweils daraus Anforderungen für die zu entwickelnde Anwendung abgeleitet werden.

Irrtum 1. *Das Netzwerk ist verlässlich („The network is reliable")*

Diese Annahme zielt auf die Frage ab, ob das zugrunde liegende Netzwerk stabil ist oder ob mögliche Ausfälle in Form von Verbindungsabbrüchen, verloren gegangenen

Nachrichten und Ähnliches in Betracht gezogen werden müssen. Für ein MPI-System, das in einem eng gekoppelten Rechencluster verwendet wird, lässt sich bis zu einer gewissen Größe argumentieren, dass etwaige Netzwerkausfälle vernachlässigt werden können, da diese nur im äußerst seltenen Fall eines Hardwareversagens auftreten können. Dehnt man sein Sichtfeld jedoch auf einen internetähnlichen Maßstab aus, muss die Antwort zweifelsohne gegenteilig ausfallen. Hier gehören abgebrochene Verbindungen und zeitweilig nicht erreichbare Server zur Normalität. Die Tatsache, dass jedem Laien diese Defizite des World Wide Webs bereits bei der alltäglichen Benutzung eines Webbrowsers klar ersichtlich sind, machen weitere Beweise für diese These obsolet. Ein MPI-System, welches im beschriebenen Umfeld eingesetzt werden soll, muss also mit solchen Schwierigkeiten rechnen. In diesem Kontext lässt sich Anforderung 1 formulieren.

Anforderung 1. *Das System ist tolerant gegenüber Netzwerkfehlern*

Irrtum 2 befasst sich mit den Übertragungs- und Reaktionszeiten zwischen Ressourcen in einem Netzwerk.

Irrtum 2. *Es gibt keine Latenzzeit („Latency is zero“)*

Auch diese Annahme lässt sich durch alltägliche Erfahrungen beim Benutzen eines Webbrowsers zum Abruf von Internetseiten mühelos als falsch identifizieren. Die Zeit, bis ein Webserver auf eine Verbindungsanfrage mit einer Antwort reagiert, variiert je nach Ziel der Anfrage, der Tageszeit und der Anzahl der Menschen, welche die lokale Internetverbindung gleichzeitig nutzen, stark. Ebenso schwankt die letztendlich erzielte Übertragungsgeschwindigkeit bei der Übermittlung der angefragten Daten. Daraus ergibt sich folgende Anforderung:

Anforderung 2. *Das System ist tolerant gegenüber variierenden Latenzzeiten*

Deutschs dritter erkannter Trugschluss bezieht sich auf eine Eigenschaft von Netzwerken, welche, wie Rotem-Gal-Oz richtig erkennt [23], in den letzten Jahren im Vergleich zu anderen solchen direkt messbaren Merkmalen, die größten Veränderungen und Verbesserungen erfährt: Die Bandbreite.

Irrtum 3. *Die Bandbreite ist unbegrenzt („Bandwidth is infinite“)*

Im Kontext der Frage, für welche Herausforderungen ein MPI-System in den in dieser Arbeit beschriebenen Umfängen gerüstet sein muss, spielt die letztendliche Bandbreite des Verbindungsnetzwerks eine große Rolle wenn es darum geht, wie groß die Datenmengen sind, die mit den verschickten Nachrichten des parallelen Programms übertragen

werden. Bei datenintensiven Anwendungen, muss das zugrunde liegende System entsprechende Maßnahmen, wie beispielsweise Kompression, bereit halten, um mit schwankenden verfügbaren Netzwerkbandbreiten bestmöglich umgehen zu können. An dieser Stelle lässt sich jedoch auch feststellen, dass die verfügbare Bandbreite bei MPI lediglich Einfluss auf die Performance und nicht auf die allgemeine Funktionsfähigkeit des Systems hat. Insofern ist die wie folgt formulierte Anforderung als weniger hart zu verstehen als manch andere.

Anforderung 3. *Das System kann mit sich verändernden Bandbreiten umgehen*

Anwendungen, die Teil eines öffentlichen Netzwerks sind, müssen in Betracht ziehen, Ziel von Angriffen verschiedener Motivation zu werden. Das Abfangen von übertragenen Daten, das widerrechtliche Benutzen von Diensten und gezieltes Lahmlegen des Systems sind nur einige Beispiele möglicher Attacken. In diesem Kontext formuliert Deutsch seinen nächsten Irrtum.

Irrtum 4. *Das Netzwerk ist sicher („The network is secure“)*

In Bezug auf das MPI-Umfeld spielt Netzwerksicherheit traditionell eine untergeordnete bis nicht vorhandene Rolle. Wie bereits dargelegt, wird MPI bis heute hauptsächlich in eng gekoppelten Netzwerken eingesetzt, die darüber hinaus in den meisten Fällen nicht öffentlich sind, beziehungsweise keine Anbindung an ein öffentliches Netz wie das Internet haben. Dadurch wird natürlich die Anzahl potentieller Angreifer auf ein Minimum reduziert, wodurch das Thema der Sicherheit kaum noch ernstzunehmende Relevanz hat. Sobald jedoch durch ein System, wie das hier zu entwickelnde, der Einsatz von MPI in internetähnlichen Netzen möglich wird, darf Netzwerksicherheit nicht mehr ausgeblendet werden. Aus diesem Grund ergibt sich die nächste Anforderung folgendermaßen:

Anforderung 4. *Das System stellt Mechanismen bereit, die erhöhte Netzwerksicherheit ermöglichen*

Der nächste Irrtum, den Deutsch identifiziert, bezieht sich auf die physische Struktur des Netzwerks:

Irrtum 5. *Die Netzwerktopologie ändert sich nicht („Topology doesn't change“)*

Hintergrund dieser fälschlichen Annahme ist die Tatsache, dass sich die Struktur eines Netzwerks außerhalb eines Labormaßstabs durchaus kurzfristig verändern kann. Nach Rotem-Gal-Oz [23] sind die Hauptursachen für sich verändernde Netzwerktopologie auf

Serverseite das administrative Hinzufügen und Entfernen von Servern und Diensten, sowie der spontane Ausfall von Netzwerkbestandteilen. Auf Clientseite identifiziert er das wiederholte Hinzufügen und Entfernen von Arbeitsplatzrechnern und Notebooks, sowie von neuartigen mobilen Geräten als Auslöser. Für das betrachtete System ist, wenn auch beide Varianten möglich sind, bevorzugt die clientseitige Argumentation von Belang. Diese Geräte sind es, für die die zu entwickelnde Anwendung betont das Rechnen per MPI ermöglichen soll. Somit lässt sich folgende Anforderung festhalten:

Anforderung 5. *Das System ist unabhängig von sich ändernder Netzwerktopologie*

Irrtum 6 spiegelt eine Haupteigenschaft des Wesens des Internets wider.

Irrtum 6. *Es gibt nur einen Administrator („There is one administrator“)*

Dieser Irrtum ist für eine Anwendung in einem internetähnlichen Umfeld ohne weiteres verifizierbar. Es ist offensichtlich, dass eine grundlegende Facette des World Wide Webs die Tatsache ist, dass es eben keine zentrale Stelle gibt, die das Netzwerk von Rechnern administriert. Jedes Teilnetzwerk und jeder einzelne Clientrechner hat seinen eigenen Verantwortlichen, der bestimmt, auf welche Art mit dem Rest des Netzwerks interagiert wird. Bezogen auf MPI ist die hauptsächliche Herausforderung, ein System zu entwickeln, das möglichst unabhängig von den Eigenheiten einzelner Rechner und Teilnetze und damit relativ universell einsetzbar ist. Es muss also eine gewisse *Portabilität* gewährleisten. Ein Deployment auf sehr unterschiedlicher Hardware und Systemkonfiguration muss möglich sein. Darüber hinaus muss die Installation des Systems möglichst einfach sein, da der Grad des Vorwissens des jeweiligen Benutzers (Administrators) nicht einschätzbar ist.

Anforderung 6. *Das System ist möglichst unabhängig von der Hardware- und Softwarekonfiguration der Prozessrechner und möglichst einfach installierbar*

Der vorletzte von Deutsch identifizierte Irrtum kann auf zweierlei Arten interpretiert werden. Er lautet:

Irrtum 7. *Es gibt keine Übertragungskosten („Transport cost is zero“)*

Wie ebenfalls Rotem-Gal-Oz [23] richtig erkennt, ist bei diesem Punkt nicht vollständig klar, was Deutsch meint. Einerseits können mit *Kosten* aus der Informatiksicht Zeit- und Datenaufwände gemeint sein, die in einem System oder bestimmten Algorithmus anfallen. Bezogen auf das verteilte Rechnen beim Betrieb eines MPI-Systems lässt sich

dies zum Beispiel unmittelbar mit dem Serialisieren von Nachrichten verbinden, welches durchaus zusätzliche Rechenzeit in Anspruch nimmt und somit nicht *kostenlos* ist. Dies hat, zusammen mit ähnlichen Maßnahmen zur Ermöglichung von Netzwerkkommunikation, Einfluss auf die Performance des Systems. Solche Maßnahmen sind unabhängig von Eigenschaften des Netzwerks, wie Latenz (Anforderung 2) und Bandbreite (Anforderung 3), und begründen deshalb eine davon abzugrenzende Anforderung.

Auf der anderen Seite können auch wirtschaftliche Kosten, also reale Geldbeträge, die beim Betrieb eines Netzwerks für Hardware und Dienstleistungen anfallen, gemeint sein. Diese Kosten werden zwar beim betrachteten System implizit dadurch minimiert, dass es betont auf vorhandener Infrastruktur eingesetzt werden kann, bilden jedoch aus Implementierungssicht keine Einschränkung. Die Nebenbedingung wird vielmehr bereits durch die Architektur erfüllt. Aus diesem Grund wird bei der Formulierung der nächsten Anforderung nur die erste Sichtweise betrachtet.

Anforderung 7. *Das System setzt effiziente Mechanismen ein, Netzwerkkommunikation vorzubereiten.*

Deutschs letzter Irrtum adressiert direkt eine Haupteigenschaft, mit der sich das hier betrachtete System von traditionellen MPI-Implementierungen abgrenzt:

Irrtum 8. *Das Netzwerk ist homogen („The network is homogeneous“)*

Das Internet und somit auch internetähnliche Netze haben unter anderem die Kerneigenschaft, nicht homogen zu sein. Somit ist dieser Irrtum im angedachten Einsatzumfeld des zu entwickelnden Systems ebenfalls ohne weiteres zu verifizieren. Das Merkmal Homogenität handelt von der Ähnlichkeit unterschiedlicher Netzsegmente, also verschiedener Verbindungsgeschwindigkeiten, Bandbreiten, Latenzen und Netzwerkarten, wie Ethernet, WLAN und proprietären Hochgeschwindigkeitsnetzen. Über all diese Facetten hinweg soll das zu betrachtende System arbeiten ohne diese im Einzelnen zu kennen. Darüber hinaus geht es jedoch auch um die Hardwarekonfiguration und somit potentiellen Geschwindigkeit der beteiligten Rechner. Um diese effizient Ausnutzen zu können, muss ein Mechanismus zur Lastbalancierung zur Verfügung stehen. Als weitere Anforderung ergibt sich somit:

Anforderung 8. *Das System kann über heterogene Netze hinweg kommunizieren und nutzt die Kapazitäten heterogener Rechner effizient.*

Rückblickend auf die hier diskutierten Irrtümer von Deutsch und die daraus abgeleiteten Anforderungen für das zu entwickelnde System, ist eine interessante Tatsache

erkennbar: Deutschs Irrtümer beschreiben genau diejenigen Eigenschaften, welche das hier betrachtete System von traditionellen MPI-Varianten und deren Einsatzgebiet unterscheiden soll. Für ein homogenes, eng gekoppeltes, zentral administriertes und nichtöffentliches Netzwerk, das für MPI genutzt wird, sind die Annahmen von Deutsch offensichtlich keine Irrtümer. Inwieweit die aufgestellten Anforderungen letztendlich durch konsequente Umsetzung der REST-Prinzipien erfüllt werden können, wird in Kapitel 7 analysiert.

4. Vorhandene MPI-Implementierungen

In diesem Kapitel soll ein Überblick über die momentan verfügbaren, in großem Umfang verwendeten, MPI-Implementierungen gegeben werden. Eine solche Sondierung des Angebots an entsprechenden Systemen verfolgt dabei zweierlei Ziele. Erstens geht es darum, einen Einblick zu ermöglichen, welche Realisierungen tatsächlich große Praxisrelevanz haben und welche Fähigkeiten diese besitzen. Zweitens ermöglicht nur die genaue Betrachtung bestehender Implementierungen eine klare Abgrenzung des zu entwickelnden Systems von diesen.

Zweifelsohne existiert inzwischen eine kaum überblickbare Fülle an MPI-Implementierungen, welche zum Teil nur ganz spezielle Anwendungsgebiete haben oder reine Forschungsimplementierungen mit wenig Praxisbezug sind. Einige wenige, besonders interessante Vertreter dieser beiden Kategorien, werden in Abschnitt 4.4 behandelt. Zuvor soll jedoch das Augenmerk zunächst auf die beiden größten und am meisten eingesetzten MPI-Varianten gelegt werden: Abschnitt 4.1 befasst sich mit *Open MPI*, Abschnitt 4.2 mit *MPICH*. Ein weiterer Abschnitt dieses Kapitels – 4.3 – widmet sich einer zwar nicht sehr stark verbreiteten aber gerade im Kontext der vorliegenden Arbeit besonders interessanten Implementierung des MPI-Standards: *MPJ/IBIS*.

4.1. Open MPI

Das Open MPI Projekt ist eine noch relativ junge Anstrengung, eine General-Purpose-Implementierung des MPI-Standards bereitzustellen. Das System wird nach dem Open Source Prinzip entwickelt und gibt sich selbst den Anspruch, eine *Next Generation MPI Implementation* [31] zu sein. Open MPI ging aus drei älteren, eigenständigen MPI-Implementierungen – FT-MPI der University of Tennessee, LA-MPI des Los Alamos National Laboratory und LAM/MPI der Indiana University – hervor und verspricht, die besten Ideen dieser drei Systeme in sich zu vereinen [54]. Außerdem flossen noch

Elemente des PACX-MPI der Universität Stuttgart ein. Aus diesem Grund soll nun im Folgenden zunächst auf die Haupteigenschaften und Zielsetzungen dieser vier MPI-Varianten eingegangen werden. Im Anschluss daran kann das daraus hervorgegangene System analysiert werden.

4.1.1. Wurzeln

Im Folgenden werden die vier MPI-Implementierungen betrachtet, welche als die Wurzeln von Open MPI angesehen werden können. Die Entwicklung dieser Systeme wird zwar aufgrund der Open MPI betreffenden Anstrengungen im Allgemeinen nicht fortgeführt, sie können jedoch wegen des teilweise sehr ausgeprägten zugrunde liegenden Entwicklungsaufwandes durchaus als eigenständig betrachtet werden.

FT-MPI

Der Fokus von FT-MPI [25], einer Abkürzung für *Fault Tolerant Message Passing Interface*, liegt, wie zu erwarten, auf der Realisierung einer gewissen Fehlertoleranz. Die Entwickler führen zu diesem Zweck über den MPI-Standard hinaus gehende Zustände für den MPI-Communicator ein, welche diverse Fehlerzustände des Systems repräsentieren. Diese können abgefragt und anschließend durch spezielle Operationen, wie einen Neuaufbau des Communicators, die Lauffähigkeit des aktuellen Programms wiederhergestellt werden. Es handelt sich somit um eine Behandlung von Fehlerzuständen auf Applikationsebene. Eine solche Erweiterung des MPI-Standards ist insofern problematisch, dass zum einen zur Benutzung der Fehlertoleranzfunktionen von FT-MPI bestimmte Bedienweisen erlernt werden müssen und zum anderen für FT-MPI geschriebene Programme nicht zusammen mit anderen MPI-Implementierungen lauffähig sind.

LA-MPI

Die MPI-Variante des Los Alamos National Laboratory (LA-MPI) [4] legt seinen Fokus ebenso auf Fehlertoleranz. Die Behandlung dieser erfolgt jedoch zunächst betont auf der Transport- und Sicherungsschicht des OSI-Schichtenmodells [40] im Gegensatz zur Behandlung auf Anwendungsebene bei FT-MPI [1]. Zur Realisierung dieser Netzwerk-Fehlertoleranz – im Kontrast zu einer Prozess-Fehlertoleranz in anderen Systemen –

[1] LA-MPI adressiert zwar eine Behandlung von Netzwerkfehlern auf allen Schichten – auch auf Anwendungsebene –, konzentriert wurde die Entwicklung jedoch zunächst auf die beiden angegebenen. Eine Veröffentlichung darüber hinausgehender Funktionalitäten vor der Eingliederung des Systems in das Open MPI-Projekt erfolgte nicht.

praktiziert LA-MPI eine strikte Abkehr vom TCP/IP-Protokoll [10], welches im Grunde ähnliche Funktionalitäten bietet. Stattdessen implementiert LA-MPI ein eigenes sogenanntes *checksum/retransmission protocol* [5], welches zum einen eine garantierte Datenintegrität zusichert und zum anderen die Möglichkeit bietet, im laufenden Betrieb ein eventuell fehlerhaftes Netzwerkinterface gegen ein weiteres im System vorhandenes auszuwechseln. Wie der Name des verwendeten Protokolls bereits andeutet, basiert die grundlegende Strategie darauf, Prüfsummen (auf Byte-Ebene) zu erzeugen und diese zwischen Sender und Empfänger abzugleichen. Wird auf diese Weise ein Fehler erkannt, können die entsprechenden Daten gezielt erneut übertragen werden. Auf diese Weise realisiert LA-MPI eine Fehlertoleranz, die vollkommen vom Anwender verborgen bleibt. Das hat den großen Vorteil gegenüber anderer Strategien, dass die entsprechenden MPI-Programme zur Nutzung dieser Funktionalität keiner speziellen Anpassung bedürfen.

Da in allen LA-MPI betreffenden Veröffentlichungen stets ebenso der Aspekt der Performance betont wird, soll dieser Punkt hier ebenfalls nicht unerwähnt bleiben: Die veröffentlichten Benchmarkergebnisse von LA-MPI [5] zeigen im Vergleich zu einem MPICH-System eine grobe Verdopplung der Latenzzeiten. Die erzielte Punkt-zu-Punkt-Bandbreite bleibt von den zusätzlichen Mechanismen zur Fehlertoleranz jedoch nahezu unangetastet und kann im Falle von mehreren vorhandenen Netzwerkinterfaces sogar signifikant erhöht werden[2].

LAM/MPI

Die Indiana University liefert mit LAM/MPI das dritte System, aus dem Open MPI hervorgeht. Die Abkürzung LAM steht dabei für *Local Area Multicomputer* und bezeichnet an sich ein Teilsystem der Parallelrechnerumgebung Trollius [16], welches verschiedene Dienste und Schnittstellen zum verteilten Rechnen in sich vereint und sogar teilweise als eigenständiges Betriebssystem für Parallelrechner verwendbar ist. LAM übernimmt dabei die Aufgabe, sogenannte *Out of the box (OTB)* Computer – gemeint sind damit eigenständige Workstations, also keine dedizierten Parallelrechner (*Inside the box*) – zu bedienen. LAM/MPI ist eine bestimmte Ausprägung von LAM, die eine Programmierschnittstelle für das System nach dem MPI-Standard anbietet. Eine andere solche Ausprägung von LAM ist eine in [16] beschriebene Implementierung einer PVM-Schnittstelle [55], welche jedoch allem Anschein nach nicht veröffentlicht wurde.

[2]Dieses Phänomen ist darauf zurückzuführen, dass durch die spezielle Implementierung der Kommunikationsoperationen innerhalb von LA-MPI als Nebeneffekt die parallele Nutzung mehrerer Netzwerkinterfaces eines Systems relativ einfach möglich ist. Eine solche Funktionalität bietet MPICH nicht.

An Features hat LAM/MPI vieles mit manch anderen MPI-Implementierungen gemein. Eine Erwähnung der üblichen Funktionen ist an dieser Stelle überflüssig. Der interessierte Leser sei diesbezüglich auf die Projektseite [39] verwiesen.

Eine Besonderheit von LAM/MPI im Vergleich zu anderen Implementierungen und gleichzeitig die hauptsächliche Funktion, welche es in Open MPI einbringt, ist jedoch die integrierte Checkpoint/Restart-Implementierung [8]. Diese nutzt das Berkeley Lab Checkpoint/Restart-System (BLCR) [24], um eine Möglichkeit bereitzustellen, beispielsweise aus Fehlerzuständen heraus die Ausführung eines parallelen Programms fortzuführen. Hierzu wird zu bestimmten Zeitpunkten koordiniert der Zustand aller beteiligten Prozesse gespeichert. Tritt ein Fehler ein, kann der Programmablauf ab dem zeitlich letzten konsistenten gespeicherten Zustand fortgesetzt werden. BLCR ist auf eine Weise in LAM/MPI integriert, dass das Checkpoint/Restart-System für die MPI-Applikation als transparent erscheint. Hierdurch ist keine Anpassung des auszuführenden Programms zur Nutzung dieser Funktionalität nötig. Vielmehr wird das Anlegen von Checkpoints und das Neustarten des Programmablaufs auf der Ebene des LAM/MPI-Laufzeitsystems von außerhalb des parallelen Programms gesteuert. Geplante Unterbrechungen der Programmausführung für beispielsweise Wartungsarbeiten sind somit ebenso möglich wie spontane Neustarts aufgrund von Fehlerfällen.

PACX-MPI

Das vierte eigenständige Projekt, das zur Basis von Open MPI gehört, ist PACX-MPI der Universität Stuttgart [2]. PACX steht dabei für *Parallel Computer Extension*. Der Fokus des Systems liegt darauf, Interoperabilität zwischen mehreren in sich geschlossenen Parallelrechnern zu ermöglichen. Auf diese Weise lassen sich beispielsweise zwei Clustersysteme, die zwar jeweils auf ein Hochgeschwindigkeitsnetzwerk aufbauen, aber untereinander mit einer langsameren Verbindung gekoppelt sind, zur gemeinsamen Ausführung eines MPI-Programms nutzen.

PACX-MPI verfolgt ebenso wie LAM/MPI das Konzept, transparent gegenüber der Anwendung zu sein. Somit ist zur Verwendung der Funktionalitäten des Systems keine Anpassung des Quellcodes des MPI-Programms notwendig. PACX-MPI funktioniert in Form einer zusätzlichen Schicht zwischen dem parallelen Programm und einer traditionellen MPI-Implementierung wie MPICH. Nur das Starten gestaltet sich geringfügig anders. Auf jedem teilnehmenden Parallelrechner – damit sind in sich abgeschlossene Rechensysteme, wie lokale Cluster, gemeint – wird zusätzlich zu den Prozessen, die das MPI-Programm ausführen, jeweils ein weiterer Prozess gestartet, der die Kommunikati-

on zwischen den verschiedenen Parallelrechnern abwickelt.

4.1.2. System

In 4.1.1 wurden die Wurzeln von Open MPI – die vier MPI-Implementierungen FT-MPI, LA-MPI, LAM/MPI und PACX-MPI – näher beleuchtet. Nun soll betrachtet werden, welche Ausprägung die interessanten Elemente dieser Systeme in der Gesamtimplementierung von Open MPI bis jetzt erfahren haben. Hierzu wird ein Überblick über die Architektur des Systems gegeben und einige ausgewählte Features vertieft analysiert. Eine kurze Bewertung der Fähigkeiten von Open MPI im Kontext der in Kapitel 3 aufgestellten Anforderungen bildet den Abschluss dieses Abschnitts.

Hinter der Entwicklung von Open MPI steht zum Teil eine ähnliche Motivation wie bei dem in dieser Arbeit dargestellten System [31]. Auch hier wurden die Herausforderungen erkannt, die sich auftun, wenn MPI in Strukturen, wie weit verteilten und heterogenen Computersystemen, eingesetzt werden soll und durch die in diesem Umfeld erzielten technischen Errungenschaften im letzten Jahrzehnt auch großflächig eingesetzt werden kann. Die Entwickler der im vorigen Abschnitt betrachteten Systeme erkannten offensichtlich außerdem, dass ihre eigenen Implementierungen zwar jeweils einzelne Teilaspekte dieser Herausforderungen befriedigen konnten, jedoch keine Realisierung des MPI-Standards existierte, die alle sich ergebenden Anforderungen abdeckte. Aus diesem Grund wurde beschlossen, die Ideen und Erfahrungen der Einzelsysteme in einer von Grund auf neuen MPI-Implementierung zu aggregieren und zu erweitern. Das Resultat ist Open MPI.

Die Architektur von Open MPI basiert auf einem Komponentenkonzept mit einer dreistufigen Hierarchie [31]. Die sogenannte *MPI Component Architecture (MCA)* bildet dabei das Grundgerüst und stellt Verwaltungsdienste für die anderen Ebenen bereit. Die *Component Frameworks* bieten ebenfalls solche Dienste an, jedoch jeweils ausgerichtet auf die Module einzelner übergeordneter Funktionsbereiche. Die genannten Module stellen als eigenständige Softwarepakete mit wohldefinierten Interfaces die dritte Ebene der Hierarchie und die letztendliche Implementierung der einzelnen Funktionen dar. Sie können dabei zur Laufzeit mit anderen Modulen verbunden werden. Zwei Beispiele für Funktionsbereiche sind die Point-to-point Transport Schicht und die Collective Communication Schicht. Open MPI ist komplett in C implementiert. Hierbei wurde durch spezielle Konstrukte eine Objektorientierung im Stil von C++ nachgeahmt. Die Verwendung einer Komponentenarchitektur birgt viele konzeptionelle Vorteile. Beispielsweise können hierdurch Teilfunktionen und deren Implementierungen sehr einfach ausgetauscht wer-

den. Eine Erweiterung des Systems durch neue Module, ohne den Rest des Quellcodes verändern oder überhaupt neu kompilieren zu müssen, ist ebenfalls möglich. Das Favorisieren einer Komponentenarchitektur gegenüber eines monolithischen Designs in einer performancezentrierten Anwendung wie MPI wirft jedoch die Frage auf, wie groß der dadurch eingeführte Overhead ist. In [14] wurde gezeigt, dass dieser bei unter einem Prozent, sowohl in der Latenz, als auch in der erzielten Bandbreite, liegt. Der Overhead kann also als vernachlässigbar angesehen werden.

Die beiden Eigenschaften des Open MPI Systems, die nun näher betrachtet werden sollen, sind – auch in Hinblick auf den Beitrag, den die in 4.1.1 besprochenen Implementierungen in Open MPI leisten – die Unterstützung von Heterogenität und Fehlertoleranz. Die Integration dieser beiden Funktionalitäten in das System ist dabei bis heute unterschiedlich weit fortgeschritten[3].

Heterogenität wird auf den Ebenen Prozessor-Heterogenität, Netzwerk-Heterogenität und Binär-Heterogenität unterstützt [32]. Eine wichtige Rolle in diesem Zusammenhang spielt ein Nebenprojekt von Open MPI: Die Laufzeitumgebung OpenRTE (*Open Run Time Environment*) [19]. Diese ermöglicht beispielsweise den Startprozess eines MPI-Programms auf heterogenen Systemen. OpenRTE registriert die Eigenschaften der einzelnen Rechenknoten, wie Prozessorkonfiguration, vorhandene Netzwerkschnittstellen und zugrunde liegende Systemarchitektur. Anschließend wird eine optimale Konfiguration bestimmt und den Prozessen mitgeteilt. Über das Laufzeitsystem hinaus trägt die Komponentenarchitektur von Open MPI zur Ermöglichung von Heterogenität bei. So sind beispielsweise verschiedene Module für verschiedene Arten von Verbindungsnetzwerken verfügbar, die beliebig gegeneinander ausgetauscht werden können, da sie gemeinsame Schnittstellen des entsprechenden Component Frameworks implementieren.

Zur Unterstützung von Fehlertoleranz ist bis jetzt nur ein Mechanismus der beschriebenen Systeme implementiert: Checkpoint/Restart-Funktionalität nach dem Vorbild von LAM/MPI [16]. Zwar sind auch andere Fehlertoleranzfunktionen, wie Nachrichten-Logging, Datenverlässlichkeit und Netzwerk-Fehlertoleranz geplant [53], diese befinden sich jedoch offensichtlich noch in der Entwicklung. Das implementierte Checkpoint/Restart-System in Open MPI unterstützt sowohl synchrones, als auch asynchrones Checkpointing, das heißt programmatisch gesteuert über eine angebotene API oder per Kommandozeile von außerhalb des parallelen Programms [38]. Hierfür wurden vier zusätzliche

[3] Die tatsächlich enthaltenen Funktionalitäten in der aktuellen Version von Open MPI (1.6.3) sind äußerst spärlich dokumentiert und können deshalb nicht erschöpfend dargelegt werden. Aus diesem Grund werden an dieser Stelle nur die dokumentierten oder in wissenschaftlichen Publikationen dargestellten Umsetzungen aufgeführt.

Component Frameworks eingeführt: Ein *Snapshot Coordinator*, ein *File Manager*, ein *Checkpoint/Restart Coordination Protocol* und ein *Checkpoint/Restart Service*. Für weitere Details sei auf [38] verwiesen.

In Bezug auf die in Kapitel 3 aufgestellten Anforderungen kann resümiert werden, dass Open MPI bereits sehr viele Herausforderungen eines MPI-Systems für den Einsatz in internetähnlichen Netzen meistert. Lösungen für manche der Anforderungen, die noch nicht erfüllt werden, scheinen zumindest in Planung zu sein. Ein Beispiel hierfür wäre die Implementierung von weiteren Fehlertoleranzfunktionen um beispielsweise Anforderung 1 in Gänze zu erfüllen. Nichtsdestotrotz sind Mechanismen, die letztendlich nötig wären, um wirklich einen praktischen Einsatz im am meisten heterogenen und am weitesten verteilten Netzwerk – dem Internet – zu ermöglichen, nicht vorhanden und, aus der offiziellen Dokumentation zu folgern, auch nicht angedacht. Gerade Anforderungen wie Sicherheit (Anforderung 4) und größtmögliche Portabilität (Anforderung 6) sind an dieser Stelle ausschlaggebend. Als hauptsächliches Einsatzgebiet für ein System wie Open MPI lässt sich nach wie vor ein nicht-öffentliches, größtenteils trotzdem homogenes und zumindest in Teilbereichen eng gekoppeltes Netzwerk identifizieren.

4.2. MPICH

Die neben Open MPI zweifelsohne populärste MPI-Implementierung ist MPICH. Im Gegensatz zu ersterem kann das Projekt jedoch auf eine wesentlich längere Tradition zurückblicken. Die Entwicklung startete bereits im Jahr 1992 [50]. Das System basierte zunächst auf den *Chameleon Parallel Programming Tools* [33], einer Bibliothek zur parallelen Programmierung, aus welcher sich außerdem der Namensbestandteil „CH“ von MPICH ableitet. Im Jahr 2001 erfolgte eine Umbenennung des Projekts in MPICH2, da ab diesem Zeitpunkt der MPI-Standard in der Version 2 Einzug in die Implementierung hielt. Momentan befindet sich das System kurz vor der Veröffentlichung des finalen Release mit der Versionsnummer 3. Damit wird der 2012 veröffentlichte MPI-3.0-Standard [28] implementiert und das Projekt benennt sich zurück in MPICH [50].

Für eine umfangreiche Ausführung zu den Anfängen von MPICH, der ersten Architektur des Systems und der Performance der Implementierung bis zum Jahre 1996 sei auf [34] verwiesen. Der Fokus dieses Abschnitts soll jedoch auf den Fähigkeiten der aktuell verfügbaren Version von MPICH2 liegen, die gegenüber der ersten Version des Systems ein prinzipiell anderes Design realisiert [46]. Durch den Einzug des MPI-2-Standards in MPICH hinzugekommene Funktionen beinhalten zum Beispiel dynamisches Prozessma-

nagement und einseitige Kommunikationsoperationen. Ebenso wie in [46][4] sind in dieser Arbeit jedoch hauptsächlich die Features aus dem MPI-Standard der Version 1, also alle grundlegenden MPI-Funktionalitäten von Belang.

Ein Zentrales Konzept von MPICH2 ist das sogenannte *Abstract Device Interface (ADI)*. Dieses dient im Grunde dazu, eine Schnittstelle zur Implementierung von Kommunikationsfunktionalitäten basierend auf beliebigen Netzwerktechnologien zu bieten. Eine direkte Implementierung der ADI-Funktionen auf Basis der Hardwareschicht würde aufgrund dann nicht vorhandener Zwischenschichten die maximale Performance bieten. Weil das ADI jedoch auf einer sehr abstrakten konzeptionellen Ebene angesiedelt ist, stellt sich der direkte Implementierungsaufwand dieser Schnittstelle für eine gegebene Netzwerkhardware als sehr groß dar [11]. Um dies etwas zu vereinfachen, existiert darüber hinaus, eine konzeptionelle Ebene tiefer, das weniger abstrakte sogenannte *Channel Interface (CH)*, welches für viele Kommunikationsfunktionalitäten benötigten generischen Code bereits beinhaltet und somit nunmehr die Implementierung ein paar weniger Funktionen zur Integration neuer Netwerkinterfaces erfordert. An sogenannten *Channels*, also Kommunikationskanälen, die das Channel Interface implementieren, existieren beispielsweise eine TCP-Socket- und eine Shared-Memory-Variante.

Der Fokus von MPICH2 liegt allem Anschein nach darauf, ein möglichst performantes und stabiles MPI-System zur Verfügung zu stellen. Der entscheidende Unterschied des zugrunde liegenden Konzepts zu demjenigen von Open MPI ist die Tatsache, dass offensichtlich nicht versucht wird, ein vergleichbar breites Angebot an Funktionalitäten zu integrieren. Stattdessen scheinen sich die Entwickler von MPICH mehr auf die Kernkompetenzen dieser traditionsreichen Implementierung zu konzentrieren. Auf der offiziellen Webseite des Projekts wird zwar MPICH2 unter anderem mit dem Attribut „widely portable“ beworben, dies bezieht sich jedoch schlichtweg darauf, dass sich das in C++ geschriebene System auf vielen verschiedenen Plattformen kompilieren lässt. Eine darüber hinaus gehende Portabilität, wie sie beispielsweise eine auf reinem Java basierende Implementierung bietet, wird nicht erzielt.

Die einzige im Kontext dieser Arbeit besonders interessante Funktionalität von MPICH2 stammt aus dem Themengebiet der Fehlertoleranz. Das bereits in Bezug auf seine Verwendung in LAM/MPI erwähnte BLCR – eine Checkpoint/Restart-Implemen-

[4]Diese Publikation stellt die Grundlage für den Großteil der Ausführungen im Rest dieses Abschnitts dar. Davon abweichende Quellen sind entsprechend gekennzeichnet. Da die Auswahl an wissenschaftlichen Publikationen zum Design und zur Implementierung von MPICH2 alles andere als umfangreich ist, muss auf Erwähnungen des Systems in Dokumenten zurückgegriffen werden, welche sich im Grunde auf andere Themen konzentrieren.

tierung – kann in MPICH2 ebenfalls zum Abwickeln von Fehlerzuständen genutzt werden und ist in dieses integriert. Da diese Funktion jedoch die einzige Charakteristik von MPICH2 ist, die im Bezug auf den Einsatz einer MPI-Implementierung in einem internetähnlichen Netzwerk dieser dafür nötige Fähigkeiten verleiht, kann abschließend deutlich resümiert werden, dass sich MPICH2 für einen solchen Einsatz in der aktuellen Version nicht eignet.

4.3. MPJ/IBIS

Eine im Vergleich zu den bisher betrachteten und den meisten am Markt verfügbaren MPI-Varianten außergewöhnliche Implementierung eines Message Passing Interface Standards ist MPJ/IBIS [15] der Freien Universität Amsterdam. Dieses basiert auf der gänzlich in Java geschriebenen Grid-Programmierumgebung IBIS [51], welche ebenfalls an der Freien Universität Amsterdam entwickelt wurde. Da dieses der letztendlichen MPI-Implementierung zugrunde liegende System für die komplette Abwicklung der Kommunikation des ersteren zuständig ist, soll in 4.3.1 nun zunächst die IBIS-Umgebung betrachtet werden. Im Anschluss daran wird die darauf aufbauende MPJ/IBIS-Schicht analysiert.

4.3.1. IBIS-Basissystem[5]

Die grundlegende Motivation der Entwickler zur Erschaffung von IBIS stellte die Erkenntnis dar, dass laut diesen keine Programmierumgebung für Grids existierte, welche die drei Eigenschaften Portabilität, Flexibilität und hohe Effizienz vereinte. Sie erkannten ebenfalls, dass traditionelles MPI zwar hocheffizient, aber nicht auf die Charakteristiken von Grids – Zusammenschlüsse von zum Teil weit verteilten und heterogenen Rechenressourcen – ausgerichtet sei. Aus diesem Grund wurde ein eigenes System entwickelt, das zum einen durch den Einsatz von reinem Java eine hohe Portabilität und zum anderen durch ausgefeilte Kommunikationsmechanismen und spezialisierte Implementierungen für bestimmte Umgebungen in Form des sogenannten *Ibis Portability Layer (IPL)* ermöglichen soll.

Eine Grundidee des Designs von IBIS ist die Flexibilität, zwar vorhandene hocheffiziente Komponenten des zugrunde liegenden Rechensystems, wie proprietäre Hochgeschwindigkeitsnetze oder optimierte Java Compiler durch spezialisierte Implementie-

[5]Dieser Abschnitt basiert größtenteils auf den Ausführungen in 51, davon abweichende Quellen sind entsprechend gekennzeichnet.

rungen aktiv zu nutzen. Jedoch kann ebenfalls jederzeit auf Standardmethoden, wie das TCP-Protokoll oder den Sun Java Compiler zurückgegriffen werden. Diese Anpassungsfähigkeit wird dadurch ermöglicht, dass der IPL zur Laufzeit, über den in Java vorhandenen Mechanismus des dynamischen Ladens von Klassen, entsprechende Komponenten verwenden kann. Über definierte Schnittstellen zu darüberliegenden Schichten, wie beispielsweise der im nachfolgenden Abschnitt betrachteten MPI-Schicht, wird der IPL über die benötigten Fähigkeiten, wie zum Beispiel geordnete Nachrichtenübertragung, informiert und kann dadurch die optimalen Module für die aktuelle Situation auswählen und laden. Um beispielsweise verschiedene Verbindungsnetzwerke unterstützen zu können, setzt IBIS unter anderem auf ein anderes System der Freien Universität Amsterdam zur Ermöglichung von Portabilität über verschiedene solche Netzwerke hinweg: Panda [13]. Darüber hinaus ist ebenfalls die Benutzung von solchen Netzwerken durch das Aufsetzen auf eine eventuell vorhandene MPI-Implementierung auf dem System möglich. In solchen Fällen kann außerdem das von IBIS implementierte *zero-copy* Protokoll zur Optimierung der Serialisierung von Objekten zum Einsatz kommen. Dafür werden spezielle Eigenschaften des Verbindungsnetzwerks ausgenutzt, um ein zusätzliches Puffern von Daten vermeiden zu können.

Zur Ermöglichung von Netzwerkheterogenität sieht IBIS die Option vor, zur Laufzeit mehrere verschiedene Implementierungen des IPL laden zu können. So kann beispielsweise ein Teil der verbundenen Rechenknoten über Fast Ethernet kommunizieren und ein anderer Teil über ein Myrinet-Netz. Darüber hinaus abstrahiert IBIS von der traditionellen Sichtweise auf Prozesse: Alle Prozesse bekommen einen sogenannten *Ibis Identifier* zugewiesen, unabhängig davon, ob diese sich im gleichen Shared Memory System befinden oder auf örtlich weit voneinander entfernten Systemen. Diese Identifier werden dabei über eine zentrale Registrierung verwaltet.

Ein weiteres wichtiges Konzept des IBIS-Designs sind die sogenannten *send ports* und *receive ports*. Dabei handelt es sich um Abstraktionen von Kommunikationsendpunkten, welche zusammen eine unidirektionale Verbindung realisieren. Diese verbindungsorientierte Architektur ermöglicht ein Streaming von Daten und dadurch gleichzeitiges Bearbeiten und Senden von großen Datenmengen, da kein Warten auf den Abschluss der Vorbereitung einer vollständigen Nachricht nötig ist, weil bereits fertig verarbeitete Nachrichtenbestandteile sofort übertragen werden können. Zur abstrakten Modellierung von kollektiven Übertragungsoperationen wie Multicast oder Many-To-One ist es außerdem möglich, beispielsweise mehrere Ports des einen Typs mit einem einzigen Port des anderen Typs direkt zu verbinden.

4.3.2. MPI-Schicht[6]

Das nun zu betrachtende MPJ/IBIS ermöglicht die Verwendung der Grid-Programmierumgebung IBIS für MPI-Programme. Dabei wird die im Jahr 2000 veröffentlichte MPJ API [18] – eine Variante des MPI-Standards für Java – implementiert.

MPJ/IBIS setzt direkt auf dem in 4.3.1 beschriebenen IPL auf. Die Architektur hat die folgende, in drei Ebenen eingeteilte Struktur: Mit dem IPL interagiert der *Ibis communication layer*, welcher low-level Kommunikationsoperationen ausführt und beispielsweise dazu die beschriebenen send ports und receive ports verwendet. Darauf aufbauend zeigt sich der *base communication layer*. Dieser implementiert die aus MPI bekannten Basiskommunikationsfunktionen, wie blockierendes und nicht blockierendes Senden und Empfangen. Von dieser Schicht abstrahiert nochmals der *collective communication layer*, der, wie sein Name schon vermuten lässt, kollektive Kommunikationsalgorithmen realisiert und diese auf Basis der Operationen der darunterliegenden Schicht abbildet.

Die Realisierung einer Nachricht erfolgt in der Form eines *MPJObject*. Dies ist ein Objekt, das aus einem Header, welcher Metadaten wie den Nachrichten-Tag und -Context enthält, und einem Datenteil besteht. Diese Objekte werden auf der Seite des sendenden Prozesses direkt an einen send port des IPL übergeben. Da es sich offensichtlich um eine ausschließliche Push-Kommunikation handelt – das heißt, ohne Ausnahme initiiert der sendende Prozess den Transfer –, ist an dieser Stelle kein weiterer Verwaltungsaufwand nötig. Auf der Empfängerseite dagegen, muss mindestens eine Warteschlange verwaltet werden, in die Nachrichten eingereiht werden können, die nicht erwartet – also von einem bereits abgesetzten *Receive* antizipiert – wurden. Eine im Kontext des in der vorliegenden Arbeit zu entwickelnden Systems interessante Eigenschaft der MPJ/IBIS-Implementierung ist die Tatsache, dass die nicht-blockierenden Kommunikationsoperationen des base communication layers auf Basis ihrer blockierenden Äquivalente umgesetzt sind. Die in dieser Arbeit als Grundlage verwendete MPI-Implementierung MPJ Express realisiert dies in umgekehrter Richtung. Die Motivation, die hinter dieser Designentscheidung liegt, wäre hier interessant, ist jedoch unglücklicherweise an keiner Stelle dokumentiert.

Betrachtet man die Fähigkeiten von MPJ/IBIS und der zugrunde liegenden IBIS-Umgebung im Kontext der in Kapitel 3 aufgestellten Anforderungen, muss als Fazit gezogen werden, dass nahezu alle erfüllt werden. Lediglich Anforderung 4 (Sicherheit) wird nicht Rechnung getragen. Da MPJ/IBIS jedoch offensichtlich in seiner Zielsetzung und

[6] Dieser Abschnitt basiert größtenteils auf den Ausführungen in 15, davon abweichende Quellen sind entsprechend gekennzeichnet.

seine Fähigkeiten sehr nahe an ein derartiges System herankommt, das versucht wird, in dieser Arbeit zu entwickeln, wird dieses als Leistungsreferenz angesehen und dessen Performance in Kapitel 9 als Vergleich für diejenige des entwickelten Systems verwendet. Hervorzuheben ist jedoch bereits an dieser Stelle, dass es sich bei IBIS abermals um ein äußerst komplexes Gebilde handelt, in das offensichtlich sehr viel Entwicklungszeit geflossen ist. Die in Teil II dieser Arbeit beschriebene Implementierung verwendet dagegen ein betont, sowohl aus konzeptioneller, als auch implementierungstechnischer Sicht einfaches Kommunikationskonzept – RESTful HTTP – um ähnliche Funktionalitäten, wie sie IBIS bietet, zu erreichen.

4.4. Weitere Implementierungen

In Abschnitt 4 wurden zum einen die zwei wahrscheinlich populärsten MPI-Implementierungen – Open MPI und MPICH – betrachtet und zum anderen eine weitere, zwar weniger weit verbreitete, aber gerade im Kontext dieser Arbeit interessante Variante analysiert. Neben diesen drei Beispielen und den im Abschnitt über die Wurzeln von Open MPI beleuchteten Implementierungen sind noch eine Vielzahl weitere, größtenteils auf bestimmte Charakteristiken der Umgebung oder angestrebter Berechnungen ausgerichtete Implementierungen des Message Passing Standards vorhanden. Im Folgenden werden exemplarisch drei dieser MPI-Varianten betrachtet. Die ausgesuchten Systeme behandeln jeweils einen bestimmten eingegrenzten Forschungsbereich bezüglich MPI. Die erste betrachtete Variante in diesem Abschnitt, HeteroMPI, beschäftigt sich mit dem Problem, unter Angabe bestimmter Charakteristiken der auszuführenden Berechnungen, die vorhandenen Ressourcen eines zugrunde liegenden heterogenen Netzes von Prozessen optimal zu verteilen. Ebenfalls auf heterogene Umgebungen, im Speziellen Grids, ist MPICH-G2 spezialisiert. Dieses nutzt jedoch zur Erfüllung seiner Aufgaben die Grid-Programmierumgebung Globus. Den Abschluss bildet IMPI, ein System, das es mehreren verschiedenen MPI-Implementierungen ermöglichen soll, innerhalb eines parallelen Programms zusammen zu arbeiten.

4.4.1. HeteroMPI[7]

Die Motivation hinter der Entwicklung von HeteroMPI folgt der Erkenntnis, dass die einzelnen Knoten eines Netzwerks aus heterogenen Rechnern offensichtlich eine unter-

[7]Dieser Abschnitt basiert größtenteils auf den Ausführungen in 45, davon abweichende Quellen sind entsprechend gekennzeichnet.

schiedliche Leistung bieten. Aus diesem Grund erscheint es als angebracht, die vom auszuführenden MPI-Programm auferlegte Last entsprechend ungleichmäßig zu verteilen. Um eine solch intelligente Lastverteilung zu ermöglichen, ist die Kenntnis von zwei unterschiedlichen Gruppen von Metriken notwendig, die im traditionellen MPI-Workflow nicht zum Einsatz kommen.

Zum einen müssen bestimmte Charakteristiken des auszuführenden parallelen Programms bekannt sein. HeteroMPI betrachtet dabei Werte, wie die Anzahl der Prozesse, das Volumen der Berechnungen des Programms, das Volumen der übertragenen Daten und die Art der Interaktion zwischen den Prozessen. Beschrieben werden diese Werte bei HeteroMPI durch eine eigene sogenannte *performance model language*, die auf der Programmiersprache mpC [44] aufbaut.

Zum anderen werden die Eigenschaften des eingesetzten Rechnernetzes beschrieben. Dieses Modell enthält dabei Werte, wie die Anzahl der Prozessoren der einzelnen Knoten, die Ausführungsgeschwindigkeit, die Skalierbarkeit der Kommunikationsschicht und die Punkt-zu-Punkt Übertragungsgeschwindigkeit von Daten. Ein Teil dieser Werte wird dabei durch die gezielte Ausführung von Microbenchmarks bestimmt.

Nachdem diese beiden Performance-Modelle aufgebaut wurden, verwendet HeteroMPI einen einfachen Mapping-Algorithmus, der Arbeitspakete an die verfügbaren Prozesse verteilt. Im Grunde wird versucht, zuerst große Arbeitspakete schnellen Prozessen und nach und nach immer kleinere langsameren zuzuordnen.

Die Benutzung der durch HeteroMPI eingeführten Funktionalitäten erfolgt auf Applikationsebene. Hierzu erweitert das System die API des MPI-Standards um weitere Funktionen. Dies hat die offensichtlichen Nachteile, dass zur Nutzung der Fähigkeiten von HeteroMPI zum einen das MPI-Programm angepasst werden muss und zum anderen es dadurch nicht mehr mit anderen MPI-Implementierungen kompatibel ist.

4.4.2. MPICH-G2[8]

Ebenso wie HeteroMPI, ist MPICH-G2 ausgerichtet auf die Anwendung von MPI in Grid-Umgebungen. Dieses unternimmt jedoch den Versuch, die eingeführte Komplexität zur effizienten Verwaltung heterogener Ressourcen – ähnlich wie das in 4.3 behandelte MPJ/IBIS – komplett transparent zu gestalten. Zu diesem Zweck setzt die Implementierung auf eine bereits vorhandene Programmierumgebung für Grid-Systeme namens Globus auf.

[8]Dieser Abschnitt basiert größtenteils auf den Ausführungen in 42, davon abweichende Quellen sind entsprechend gekennzeichnet.

MPICH-G2 basiert, wie der Name bereits vermuten lässt, auf dem in 4.2 behandelten MPICH. Dieses wurde angepasst, um zur Kommunikation die Dienste von Globus benutzen zu können. Darüber hinaus bietet letzteres auch weitere Services an, welche verwendet werden. Ein Beispiel eines solchen Service ist die Abwicklung einer Authentifizierung zwischen den beteiligten Prozessen eines Rechnernetzes beim Start der Ausführung eines verteilten Programms. Schon allein dieses Feature macht MPICH-G2 im Kontext dieser Arbeit interessant. Alle bisher betrachteten MPI-Implementierungen sehen keinerlei Sicherheitskonzept vor.

Nach einer erfolgten Authentifizierung kann die Ausführung des MPI-Programms gestartet werden. MPICH-G2 nutzt hierfür eine besondere Form des traditionellen *Machine File*s von MPI. Neben den zu verwendenden Hosts können auch zusätzliche Informationen, wie die Anzahl verfügbarer CPUs und die Menge an Arbeitsspeicher, sowie Umgebungsvariablen angegeben werden. Die zur Verfügung gestellten Daten können anschließend vom Globus-System zur Ressourcenallokation benutzt werden.

Globus hat unter anderem die Fähigkeit, das jeweils schnellste verfügbare Verbindungsnetzwerk zwischen zwei Prozessen zu verwenden. Dabei greift es unter anderem auf eventuell vorhandene sogenannte *Vendor*-MPI-Implementierungen, also proprietäre, auf dedizierten Parallelrechnern laufende, Hersteller-MPIs zurück.

Wie oben erwähnt, ist MPICH-G2 durch die Verwendung von Globus in der Lage, völlig automatisiert Heterogenität zu verwalten und diese somit für den Benutzer transparent zu gestalten. Zur (optionalen) Optimierung einiger Parameter ist es jedoch notwendig, vom Benutzer bereitgestellte Informationen zu akquirieren. MPICH-G2 bietet hierzu außerdem eine Schnittstelle auf Applikationsebene an, über welche beispielsweise Informationen über die vorhandene Netzwerktopologie an das System übergeben werden können. Auf diese Weise können zum Beispiel kollektive Kommunikationsoperationen optimal auf die gegebenen Umgebungsbedingungen abgestimmt werden. Bemerkenswert an diesem Feature von MPICH-G2 ist die Tatsache, dass zur Realisierung der MPI-Standard nicht erweitert werden musste. Es wurden also keine neuen Funktionen eingeführt, sondern bereits vorhandene, im Standard definierte, geschickt verwendet.

Im Großen und Ganzen bietet MPICH-G2 ähnliche Funktionalitäten wie das bereits betrachtete MPJ/IBIS. In einigen Bereichen – allen voran demjenigen der Sicherheit – gehen diese sogar darüber hinaus. In einem solchen Vergleich und ebenfalls im Kontext dieser Arbeit unter Berücksichtigung der in Kapitel 3 aufgestellten Anforderungen zeigen sich aber zwei entscheidende Nachteile von MPICH-G2. Zum einen ist es durch seine in C (und C++) implementierte Basis MPICH ebenso wenig portabel wie die meisten

anderen hier analysierten Systeme. Zum anderen ist seine Implementierung durch die Verwendung des bereits in sich sehr vielfältigen Globus-Systems noch wesentlich komplexer als diejenige von MPJ/IBIS – und diese Komplexität wurde bereits bei letzterem als Nachteil erkannt.

4.4.3. IMPI[9]

Eine bestimmte Form von Heterogenität wird von IMPI adressiert. Dieses versucht, eine Zusammenarbeit zwischen mehreren, lokal begrenzten und für sich homogenen MPI-Installationen innerhalb eines MPI-Programms zu ermöglichen. Auf diese Weise sollen einerseits die Performancevorteile von optimierten, auf die jeweils verwendete Hardware ausgerichteten MPI-Implementierungen genutzt, andererseits jedoch die begrenzten Kapazitäten dieser vervielfacht werden.

IMPI beschreibt dabei einen Standard, welcher zunächst auf Basis des bereits vorgestellten LAM/MPI [16] implementiert wurde. Ein Bestandteil des Standards – die IMPI-Server-Spezifikation – erfuhr darüber hinaus eine eigenständige Umsetzung [48], da diese als einziges Element des Standards unabhängig von einer konkreten MPI-Implementierung ist.

Der Standard definiert dabei vier verschiedene Funktionseinheiten innerhalb eines zusammengeschlossenen MPI-Systems. Mit dem bereits erwähnten *Server* wird ein zentraler Steuerungsmechanismus etabliert, welcher den Großteil der Verwaltung des IMPI-Verbunds übernimmt. Jedes lokale MPI-System, beziehungsweise jede gestartete eigenständige Instanz einer MPI-Implementierung stellt einen *Client*, der als Stellvertreter für die jeweilige Instanz dient. Ein solcher Client verwaltet mehrere *Hosts*, also einzelne Rechner, welche wiederum sogenannte *Procs* beinhalten, die letztendlichen Prozesse.

Die Grundidee der Architektur von IMPI besteht darin, in jedem lokalen MPI-System einen globalen Kommunikator `MPI_COMM_WORLD` zu erzeugen, welcher auf Applikationsebene ebenfalls die Kommunikation mit entfernten Prozessen zulässt. Hierzu wird nach Austausch einiger Verwaltungsinformationen – mit dem Server als zentraler Datenquelle und -senke – ein vollständig verbundenes TCP/IP-Socket-Netz zwischen allen beteiligten Hosts aufgebaut. Die jeweilige konkrete Umsetzung des IMPI-Standards in den mitwirkenden MPI-Implementierungen konstruiert anschließend einen solchen erwähnten globalen Kommunikator und bildet die darüber ansprechbaren Prozesse für den Benutzer vollständig transparent auf lokale und entfernte Ressourcen ab.

[9] Dieser Abschnitt basiert größtenteils auf den Ausführungen in 41, davon abweichende Quellen sind entsprechend gekennzeichnet.

IMPI bewerkstelligt mit relativ geringer Komplexität die Option, MPI in einer Umgebung, die eine bestimmte Form von Heterogenität aufweist, zu verwenden. Diese geringe Komplexität lässt sich als größter Vorteil des Systems gegenüber anderen, zum Teil bereits hier vorgestellten Ansätzen werten. Eine erhöhte Portabilität wird darüber hinaus durch die Verwendung von C++ zur Implementierung nicht erreicht, die möglichen Einsatzmöglichkeiten durch die Einschränkung auf lokale MPI-Installationen mit integrierten IMPI-Fähigkeiten sogar noch eingeschränkt. Auf der Performanceseite kann die in [41] dargestellte Implementierung des IMPI-Standards nicht mit anderen Systemen, die vergleichbare Funktionalitäten bieten, konkurrieren. Dies war im Übrigen, wie ebenfalls in obigem Artikel erwähnt wird, auch kein Ziel des Projektes und wurde bewusst in Kauf genommen. Vielmehr ging es offensichtlich darum, ein Proof-of-Concept zu entwickeln.

4.5. Erkenntnisse

In Betrachtung der in diesem Kapitel vorgestellten und in Tabelle 4.1 nochmals aufgeführten MPI-Implementierungen – und dies gilt sicherlich auch noch für einige weitere – kann die Erkenntnis formuliert werden, dass eine Vielzahl an Entwicklern und Forschungsgruppen im MPI-Umfeld die Herausforderungen und Chancen für den Message Passing Interface Standard in einer zunehmend vernetzten und mit enormen Rechenkapazitäten aufwartenden Welt, wie der unseren, zu einem großen Teil erkannt haben. In zahlreichen Ansätzen wird versucht, diese neuartigen Strukturen möglichst effizient zu nutzen. Für eine Übersicht und Evaluierung weiterer Implementierungsvarianten des MPI-Standards, darunter beispielsweise auch verschiedene Python-Implementierungen – Sprachausprägungen von MPI, die in dieser Arbeit bisher nicht betrachtet wurden – sei auf [35] verwiesen.

Der Fokus der verschiedenen vorgestellten Varianten liegt dabei auf der Optimierung unterschiedlicher Charakteristiken. Betrachtet wurden zwar nur MPI-Implementierungen, welche unter der Annahme eines internetähnlichen Netzes als Grundlage naheliegend erscheinen – also keine hochoptimierten proprietären Hersteller-MPIs –, jedoch auch diese Strömung des MPI-Universums beinhaltet sehr unterschiedliche Zielsetzungen.

HeteroMPI beispielsweise ist stark ausgerichtet auf möglichst hohe Performance in einem solchen Netz. Hierzu wird sogar der MPI-Standard erweitert – eine sehr starke Einschränkung der Flexibilität und des möglichen Einsatzbereichs. MPICH-G2 dagegen möchte möglichst flexibel sein und erreicht dieses Ziel durch den Einsatz einer äußerst komplexen Grid-Programmierumgebung als Basis. Open MPI und MPICH, die beiden

	Sprache	Portabilität	Heterogenität	Fehlertoleranz
Open MPI	C	niedrig	Prozessor Netzwerk binär	Checkpoint/ Restart
MPICH2	C++	niedrig	-	Checkpoint/ Restart
MPJ/IBIS	Java	hoch	Prozessor Netzwerk	-
HeteroMPI	C	niedrig	algorithmisch	-
MPICH-G2	C	niedrig	Prozessor Netzwerk binär	Globus fault detection
IMPI	C++	niedrig	Kooperation von in sich homogenen Clustern	-
Projekt-Implementierung	Java	hoch	Prozessor Netzwerk	-

Tabelle 4.1.: Übersicht der vorgestellten MPI-Implementierungen

größten betrachteten MPI-Systeme – groß im Sinne von verbreitet und full-featured –, versuchen, möglichst die besten und effizientesten Features aller dieser Ausprägungen in sich zu vereinen. Die Komplexität dieser ist hierdurch jedoch ebenfalls enorm, obwohl gleichzeitg noch nicht alle Herausforderungen der geschilderten Umgebung gemeistert werden können. Keines der analysierten Systeme erfüllt sowohl alle formulierten Anforderungen, ist höchst performant, als auch flexibel einsetzbar. Dies stellt einen der Gründe dafür dar, im Kontext dieser Arbeit eine weitere MPI-Implementierung in der Reihe der analysierten zu entwickeln und den verfolgten Ansatz ebenso nach seinen letztendlichen Fähigkeiten und Charakteristiken zu bewerten.

Teil II.

Implementierung

Nachdem in Teil I eine theoretische Grundlage zur Entwicklung einer MPI-Implementierung auf Basis von RESTful HTTP geschaffen wurde, können nun im Folgenden Details zur tatsächlichen Realisierung dieser betrachtet werden. Die hier dargestellte Umsetzung baut auf der in [49] analysierten Implementierung auf. Deren Hauptbestandteil, das RESTDevice, wird unter anderem in seinen Einzelheiten zur Vollständigkeit in Kapitel 5 nochmals betrachtet. An dieser Stelle sei in Erinnerung gerufen, dass Hauptgegenstand der vorliegenden Arbeit ist, eben dieser grundlegenden Realisierung eines MPI-Systems, das versucht, RESTful HTTP zur Kommunikation zu verwenden, zum einen einen theoretischen Unterbau zu geben und zum anderen, sie entsprechend dieser Basis zu überarbeiten. Dem bisher nicht konkret betrachteten Bestandteil dieser Zielsetzung – der Überarbeitung der vorhandenen Implementierung auf Basis der REST-Prinzipien – ist Kapitel 6 gewidmet. In Kapitel 7 wird darauf aufbauend die Frage geklärt, welche der Anforderungen aus 3 letztendlich durch diese Architektur erfüllt werden können und welche über die reine Umsetzung von REST-Konformität hinausgehende Strukturen erfordern würden.

Ein weiteres Kapitel dieses Teils beschäftigt sich mit Optimierungen und Erweiterungen des Systems, die möglich werden, nachdem die bloße Umsetzung der REST-Prinzipien durchgeführt wurde. Hierfür werden besondere Fähigkeiten des Architekturstils REST ausgenutzt um beispielhaft eine der kollektiven Kommunikationsoperation – den MPI-Broadcast – zu optimieren. Darüber hinaus widmet sich das Kapitel dem nun möglichen Umgang mit teils, beispielsweise aufgrund von Firewalls, blockierten Kommunikationskanälen zwischen einzelnen Prozessen.

Synthetische Benchmarks und Messergebnisse beim Einsatz des vorgestellten Systems im Vergleich zur Implementierung aus dem Master-Projekt [49] und des MPJ/IBIS-Systems [15] werden in Kapitel 9 betrachtet. In Kapitel 10 dagegen wird aufgezeigt, dass das entwickelte System nicht nur Forschungsrelevanz hat und in synthetischen Leistungsmessungen eine gewisse Performance erzielt, sondern auch für reale Probleme und Algorithmen erfolgreich eingesetzt werden kann. Zu diesem Zweck wird die Ausführung eines MPI-Programms beschrieben, welches ein eingebettetes Runge-Kutta-Verfahren zur Lösung von Systemen gewöhnlicher Differanzialgleichungen nutzt. Dabei erfolgt ein Vergleich der Leistung auf Basis des entwickelten Systems zum selben Algorithmus, ausgeführt auf Basis einer MPICH2- Implementierung und MPJ/IBIS.

5. Basis-Implementierung

Die Intention dieses Kapitels besteht darin, das in Kapitel 2 konzeptionell betrachtete System, auf welcher die vorliegende Arbeit basiert, auf Implementierungsebene darzustellen. Durch das Zeichnen eines dann vollständigen Bildes des zugrunde liegenden Systems wird eine klare Trennung der geleisteten Implementierungsarbeit des Master-Projekts [49] und derjenigen dieser Masterarbeit möglich. Letztere wird darauf aufbauend in Kapitel 6 analysiert. Das Augenmerk liegt in diesem Kapitel vorwiegend auf einer möglichst vollständigen Darstellung der Implementierung, welche tiefer geht, als diejenige in [49]. Eine solche umfangreiche Analyse ist notwendig, um die Unterschiede zwischen den beiden Implementierungsvarianten in Gänze erfassen zu können. Es wird sich zeigen, dass diese Unterschiede hauptsächlich in der Umsetzung des `RESTDevice` zu finden sind. Der typische Ablauf der Ausführung eines MPI-Programms mithilfe des Systems, wie sie in Kapitel 2 und tiefer in [49] dargestellt wurde, stellt die Leitlinie der folgenden Abschnitte dar.

5.1. Startphase

Mit dem Begriff *Startphase* ist der zeitliche Abschnitt bei der Ausführung eines MPI-Programms mit dem System gemeint, der sich vom Starten von `mpirun` durch den Benutzer bis zum Beginn des Ablaufs des letztendlichen parallelen Programms erstreckt. Den Schlusspunkt dieser Phase stellt typischerweise die erstmalige Ausführung von `MPI.Init()` dar.

Zunächst ist es notwendig, auf allen Rechnern, die im Folgenden an der Ausführung des parallelen Programms teilnehmen sollen, ein Daemon-Programm zu starten. Dieses dient sowohl als Server und bietet in dieser Funktion einen RESTful Webservice nach außen hin an, als auch als Client unter Verwendung der RESTful Webservices anderer Daemons. Es ist realisiert in Form eines JBoss Application Servers, in dessen JavaEE Servlet Container der entsprechende Webservice und HTTP-Client ausgeführt wird. Ein großer Vorteil bei der Verwendung des JBoss AS als Daemon ist die Tatsache, dass dieser eben-

falls in reinem Java implementiert ist und somit ohne Anpassung auf allen Plattformen, für die eine Java Virtual Machine verfügbar ist, ausgeführt werden kann. Zur Implementierung des RESTful Webservice wurde ein Framework verwendet, welches ebenfalls von JBoss stammt: RESTEasy. Im einzelnen wird das in Abschnitt 2.2 dargestellte Interface `RESTResource` implementiert, dessen Methoden durch RESTEasy-Elemente annotiert sind und dadurch auf HTTP-Ressourcen abgebildet werden. Die Klasse, die die Implementierung des Interfaces vornimmt heißt `RESTServer`. Eine Instanz dieser Klasse wird von der `RESTApplication`, einer Klasse, die von `javax.ws.rs.core.Application`, eine der zentralen Klassen des JAX-RS-Standards, ableitet, bei der Initialisierung des Servlet-Containers des Application Servers als Singleton-Servlet erzeugt. Im Konstruktor von `RESTServer` geschieht die letztendliche Instantiierung der für die zu beschreibende Implementierung zentralen Singleton-Klasse `RESTDevice`, auf welche während des Ablaufs des MPI-Programms vom `RESTServer` zur Abwicklung eingegangener MPI-Nachrichten zugegriffen wird. Bevor dies jedoch in Abschnitt 5.2 näher betrachtet wird, sollen nun zunächst die Abläufe auf dem Kontrollrechner nach erfolgreichem Start der beschriebenen Daemons analysiert werden.

Zur Ausführung eines mit der API von MPJ Express – diese entspricht der in [17] definierten mpiJava-API – geschriebenen MPI-Programms führt der Benutzer auf der Kommandozeile das Programm `mpirun` aus. Dieses ist ein Bash-Skript, das lediglich als Wrapper zur Ausführung der `main`-Methode der Java-Klasse `MPJRun` von MPJ Express dient. Dass ein solches plattformspezifisches Skript eingesetzt wird, ist lediglich Komfortgründen geschuldet, die sich aus der Motivation ableiten, die Bedienung des Systems möglichst ähnlich zu derjenigen etablierter MPI-Implementierungen, wie MPICH, zu gestalten. Es bietet keinerlei zusätzliche Funktionalität und kann im Prinzip auch übergangen werden. Die dem `mpirun`-Skript übergebenen Argumente werden direkt an die Java-Klasse weitergegeben. Im Einzelnen sind dies das zu verwendende Device – zum Beispiel `-dev niodev` für die Java NIO Socket Implementierung, `-dev mxdev` für die Myrinet Implementierung oder `-dev restdev` für die mit dem hier vorgestellten System eingeführte RESTful HTTP Implementierung –, die Anzahl der teilnehmenden Prozesse (`-n <#processes>`) und das `machines` File, eine Datei, welche eine Liste mit den Host-Adressen der teilnehmenden Prozesse enthält. Im Falle der `restdev` beinhaltet letzteres die Wurzel-URIs der RESTful Webservices der beteiligten Prozesse.

Der größte Teil der Programmlogik der Klasse `MPJRun` befindet sich innerhalb ihres Konstruktors, welcher die bereits erwähnten Programmargumente übergeben bekommt. Nach einer Auswertung dieser, kann die Entscheidung getroffen werden, welche konkrete

Device-Implementierung benutzt werden soll. Die weiteren Beschreibungen gelten für den Fall, dass im weiteren Programmablauf die Klasse `RESTDevice` initialisiert wird. Für Informationen zu den mit MPJ Express mitgelieferten Device-Implementierungen, sei auf [12] verwiesen. Fordert der Benutzer also die Verwendung der RESTful Implementierung, wird eine Instanz der Klasse `RESTStarter` erzeugt, deren Konstruktor Aspekte der nun verarbeiteten Programmargumente, wie beispielsweise das `machines` File und die Anzahl der teilnehmenden Prozesse, übergeben werden. Die Logik von `MPJRun` endet direkt nach der Erzeugung dieses Objekts, der Programmablauf wird in der Klasse `RESTStarter` fortgeführt.

Die Aufgaben des `RESTStarter` sind das Einlesen des `machines` Files, das Erzeugen der HTTP-Client-Objekte, die Durchführung eines Handshakes mit allen beteiligten Prozessen, der Broadcast der validierten Host-Adressen und das letztendliche Einlesen und Starten des MPI-Programms. Wie bereits erwähnt, enthält das sogenannte `machines` File im Falle der Verwendung der RESTful Implementierung die Wurzeladressen der Webservices der teilnehmenden Prozesse in Form von URIs. Eine solche URI hat zum Beispiel die Struktur `http://<ip-adresse>:<port>/REST-MPI/mpi`. Das Einlesen dieser Datei erfolgt in der Methode `addNodes(String machinesFile)` der Klasse `RESTStarter` auf einfache Art und Weise mithilfe eines `java.io.FileReader`-Objekts. Mit `#` markierte Zeilen werden dabei als Kommentare interpretiert und aus diesem Grund bei der Auswertung übersprungen. Die eingelesenen Adressen erfahren im Anschluss eine Überführung in `java.net.URL`-Objekte, welche gespeichert werden.

In einem nächsten Schritt erfolgt die Instantiierung derjenigen Objekte, die in dieser Version der Implementierung alleinig die Rolle der jedem Daemon zugehörigen HTTP-Clients erfüllen. Zur Realisierung der Client-Funktionalität kommen sogenannte Proxy-Objekte zum Einsatz, die vom RESTEasy-Framework zur Verfügung gestellt werden. Die Technologie der Proxy-Objekte ist auf Seite des Benutzens derselben gegenüber anderen Ansätzen sehr komfortabel. Ein solches Objekt offeriert nach außen hin dasselbe Interface, das der dazugehörige RESTful Webservice implementiert. Hierdurch gestalten sich clientseitige HTTP-Anfragen als simple Methodenaufrufe auf dem entsprechenden Proxy-Objekt des zu kontaktierenden Servers. Alle nötigen Operationen, wie das Aufbauen einer TCP-Verbindung zum Server, das Formulieren und Absetzen einer HTTP-Anfrage und das Verarbeiten der entsprechenden Antwort, geschehen innerhalb des Frameworks ohne Beeinflussung durch den Benutzer. Auf der anderen Seite ist naheliegend, dass ein solcher Ansatz natürlich sehr wenig Flexibilität erlaubt. Der Benutzer hat keinerlei Handhabe über die vielfältigen Aspekte eines HTTP-Requests.

Dies ist unter anderem ein Grund dafür, dass in der Implementierung dieser Masterarbeit, wie in 6 beschrieben wird, ein anderer Ansatz zum Einsatz kommt. Die Erzeugung eben dieser Proxy-Objekte erfolgt in der Version des Master-Projekts in der Methode `createClients()` der Klasse `RESTStarter`. Es wird für jede im vorherigen Schritt eingelesene Host-Adresse ein Client-Proxy-Objekt instantiiert und in einer HashMap abgelegt, deren Schlüssel die zugehörigen URIs darstellen. Hierdurch kann im Folgenden auf effiziente Weise das zu einer URI gehörende Proxy-Objekt zum Absetzen von HTTP-Requests gefunden werden.

Die erzeugten Client-Proxy-Objekte können nun direkt verwendet werden und finden ihren ersten Einsatz in der nächsten Stufe der Startphase. Ziel des nun folgenden Schrittes ist, zum einen die Erreichbarkeit der entfernten Daemons und deren Bereitschaft zur Interaktion zu verifizieren und zum anderen an diese jeweils eine vollständige Liste der Host-Adressen der teilnehmenden und erfolgreich verifizierten Prozesse zu übermitteln. Außerdem bekommen alle Prozesse in dieser Phase vom `RESTStarter` eine eindeutige Identifikationsnummer in Form eines zufällig erzeugten *Universally Unique Identifiers (UUID)* zugeteilt. Die zwei beschriebenen Stufen werden in der Methode `initNodes()` abgewickelt. Zunächst wird versucht, für alle verfügbaren Hosts deren bereitgestellte HTTP-Ressource `/init` relativ zur Wurzel-URI des Webservices durch ein HTTP-PUT zu kontaktieren. Im Body dieses Requests wird die vorher erzeugte zufällige UUID übertragen. Das geschieht implizit durch Aufruf der entsprechenden Methode `initNode(String uuid)` des Proxy-Objekts mit der UUID als Argument. Hier sei erneut auf die äußerst komfortable Verwendung einer auf die beschriebene Weise exponierten Schnittstelle eines RESTful Webservice hingewiesen. Reagiert der angesprochene Service, wird in der zugehörigen HTTP-Response der String `OK` ebenfalls im Body der Nachricht übertragen[1]. Nach erfolgter Verifizierung der teilnehmenden Prozesse, wird im `RESTStarter` eine Liste derjenigen Hosts erstellt, die reagiert haben. Diese Liste enthält auch die eindeutigen Identifikationsnummern der Prozesse. Über die Methode `receiveProcesses(String list)` der Client-Proxy-Objekte kann im Anschluss ein HTTP-PUT auf der Ressource `/processes` ausgelöst und auf diese Weise die erzeugte Liste an alle entfernten Rechner übertragen werden.

Der letzte Schritt der Startphase besteht darin, das parallele Programm einzulesen, es

[1] Dem geneigten Leser mag an dieser Stelle zum ersten Mal auffallen, dass die geschilderte Weise, den zur Verfügung gestellten Webservice zu benutzen, sowohl in der Wahl des HTTP-Verbs (PUT), als auch im Umgang mit dem Rückgabewert, weder auf Client-, noch auf Serverseite, den in Kapitel 1 geschilderten REST-Prinzipien genügt. Die Analyse und Korrektur solcher konzeptioneller Fehler ist Gegenstand der vorliegenden Arbeit und wird in Kapitel 6 dargestellt. Im Folgenden wird deshalb zunächst auf weitere diesbezügliche Reflexionen zugrunde liegender Designentscheidungen verzichtet.

an alle teilnehmenden Prozesse zu übertragen und dort zu starten. Hierfür ist auf Seite des Kontrollrechners innerhalb der Klasse `RESTStarter` die Methode `startProgram()` und auf Seite der einzelnen Prozesse die Methode `createProgram()` der Klasse `RESTServer` zuständig. Letztere ist über die HTTP-Ressource `/programs` durch ein HTTP-POST ausführbar. Im Einzelnen wird im `RESTStarter` zunächst das beim Start von `mpirun` angegebene `jar`-Archiv als `byte`-Array eingelesen. Dieses dient im Anschluss als Argument beim Aufruf der Methode `createProgram()` auf den entsprechenden Client-Proxy-Objekten der teilnehmenden Prozesse. Erneut übernehmen an dieser Stelle die restliche Abwicklung des resultierenden HTTP-Requests die Interna von RESTEasy. Dazu gehört beispielsweise das Einbetten des `byte`-Arrays in den HTTP-Body desselben. Somit sind alle Aufgaben des Kontrollrechners innerhalb der Startphase abgearbeitet. In den `RESTServer`-Instanzen der Prozesse muss nun noch das parallele Programm empfangen und gestartet werden. Das Unmarshalling des HTTP-Bodys mit dem zuvor verpackten `byte`-Array als Resultat übernimmt erneut RESTEasy. Aufgabe der Methode `createProgram()` auf Serverseite ist nun zunächst, diesen Byte-Strom erneut als `jar`-Archiv im lokalen Dateisystem abzulegen. Die Entscheidung eines solchen vorübergehenden lokalen Persistierens des MPI-Programms wurde getroffen, um im weiteren Ablauf die Möglichkeit zu bieten, ebenfalls ein bereits lokal abgelegtes Programm, welches eventuell auf einem anderen Weg an die Prozesse verteilt wurde, zu laden und gleichzeitig die Schnittstelle der Funktion, welche das Starten übernimmt, möglichst simpel zu halten. Darüber hinaus ist der hierfür eventuell zusätzlich nötige Aufwand des Speicherns und erneuten Ladens während der Startphase gegenüber der typischen anschließenden Ausführungszeit des MPI-Programms vernachlässigbar.

Die letzte Programmzeile der Methode `createProgram()` ruft die bereits angedeutete Funktion `startProgram()` auf. Diese ist im Übrigen zur Realisierung der oben geschilderten Flexibilität in der Verteilung des parallelen Programms ebenfalls als optional verwendbare HTTP-Ressource unter dem Pfad `/start` exponiert. Sie übernimmt lediglich das Starten eines neuen Threads, in welchem das MPI-Programm ausgeführt wird. Die `run()`-Methode dieses Threads erstellt in einem ersten Schritt ein Objekt der Klasse `java.net.URLClassLoader`, welchem der aktuelle `ContextClassLoader` als *parent* zugewiesen wird. Dieser `URLClassLoader` lädt das parallele Programm zusammen mit der MPJ-Express-Library und eventuell noch zusätzlich benötigten Bibliotheken. Nachdem die `main()`-Methode des parallelen Programms lokalisiert wurde, erfolgt eine Interaktion mit der bereits im Konstruktor des `RESTServer` erzeugten `RESTDevice`: Da der `main()`-Methode unter anderem der Rang des jeweiligen Prozesses und die Gesamtzahl der

Prozesse als Argumente mitgeteilt werden müssen, bedarf es nun der Ermittlung dieser Werte. Die zuvor verteilte Liste der Prozess-Hostadressen mit den zugeordneten `UUID`s liegt in jeder verteilt vorhandenen `RESTDevice`-Instanz in derselben Sortierung vor. Der Rang wird daraus abgeleitet, an welcher Stelle dieser sortierten Liste der aktuelle Prozess steht. Die letzte Aktion der Klasse `RESTServer` innerhalb der Startphase besteht im letztendlichen Aufrufen der `main()`-Methode des MPI-Programms im erzeugten Thread.

5.2. Ausführung des parallelen Programms

Ab dem Zeitpunkt des Aufrufens der `main()`-Methode des parallelen Programms innerhalb der teilnehmenden Prozesse, ist die weitere Ausführung unabhängig vom Kontrollrechner. Es existiert in der aktuellen Version von MPJ Express kein separates Laufzeitsystem, wie es beispielsweise in Open MPI [54] vorhanden ist, welches Dienste wie Monitoring oder manuelle Prozesskontrolle bietet. Die Implementierung eines solchen war außerdem weder Gegenstand des Master-Projekts, noch dieser Arbeit. Die Prozesse agieren somit bei der Ausführung des MPI-Programms autark. Eine Intention bei der Gestaltung der Abarbeitung des Programms ist unter anderem, dass keine weitere Kontrolle durch den Benutzer notwendig ist.

Die zentrale Komponente in dieser Phase ist die verwendete konkrete Device-Implementierung – im Kontext der vorliegenden Arbeit das `RESTDevice`. Alle von der mpiJava-API [17] und somit auch von der im parallelen Programm importierten MPJ Express Bibliothek bereitgestellten MPI-Funktionen werden mithilfe der in Abbildung 2.1 dargestellten Schichtenarchitektur von MPJ Express letztendlich auf ein paar wenige Kommunikationsoperationen innerhalb der Device-Implementierung heruntergebrochen. So können beispielsweise auf einer relativ hohen Abstraktionsebene angesiedelte kollektive Operationen, wie *Broadcast* oder *Reduce*, durch die Kombination von sehr einfachen Sende- und Empfangsfunktionen realisiert werden. Die in einer Device-Implementierung umzusetzenden Funktionen sind, wie bereits in [49] dargestellt wurde, zum einen blockierendes und nicht blockierendes asynchrones und synchrones Senden (`send()`, `isend()`, `ssend()` und `issend()`), sowie blockierendes und nicht blockierendes Empfangen (`recv()` und `irecv()`). Zum anderen sind grundlegende Initialisierungs- (`init()`), Finalisierungs- (`finish()`) und Sondierungsfunktionen (`probe()` und `iprobe()`) zu implementieren.

Die Implementierung des Master-Projekts [49] setzt vollständig auf sogenannte *Push*-Kommunikation. Damit gemeint sind Kommunikationsoperationen, welche durchwegs

vom Sender der jeweiligen Daten initiiert werden. Davon abzugrenzen ist *Pull*-Kommunikation, welche von einer Initiierung von der Empfängerseite ausgeht. Reine Push-Kommunikation ist zwar die traditionell in MPI-Implementierungen eingesetzte Variante – dem Autor ist kein MPI-System bekannt, das hiervon abweicht –, wie in Kapitel 6 jedoch dargestellt wird, genügt dieses Konzept nicht einer nach REST-Prinzipien gestalteten Realisierung und muss deswegen überdacht werden. Im Folgenden wird aber zunächst die Push-Variante der Projekt-Version des Systems dargestellt.

Die Anzahl der zu implementierenden Kommunikationsoperationen lässt sich dadurch weiter reduzieren, dass alle blockierenden Funktionen auf einfache Weise durch ihre jeweiligen nicht blockierenden Äquivalente abgebildet werden können. Der MPI-Standard [28] sieht vor, dass nicht blockierende Kommunikationsfunktionen ein Objekt zurückgeben, welches eine Wartefunktion anbietet. Diese kann zu dem Zweck aufgerufen werden, um gezielt auf die Fertigstellung der zugehörigen Operation zu warten. In diesem Kontext gestaltet sich die Implementierung einer blockierenden Kommunikationsoperation insofern, dass die nicht blockierende Variante und im Anschluss die entsprechende Wartefunktion ausgeführt wird. Somit sind im Folgenden nur die nicht blockierenden Versionen der Funktionen zu betrachten.

Ein Programmierparadigma von MPI ist, dass zu jedem Aufruf einer Sendefunktion ein Aufruf einer Empfangsfunktion auf der Gegenseite gehört. Bei der Benutzung von kollektiven Operationen geschieht dies implizit durch die Ausführung derselben Funktion von allen beteiligten Prozessen unter Angabe von Datenquellen und -senken. Daraus leitet sich ab, dass grundsätzlich zwei Fälle im chronologischen Zusammenspiel zwischen Sende- und Empfangsprozessen betrachtet werden müssen. Entweder der Aufruf der Sendeoperation erfolgt vor oder nach dem Aufruf der Empfangsoperation auf der Gegenseite. Da, wie bereits erwähnt, reine Push-Kommunikation eingesetzt wird, sind in der Tat nur diese beiden Fälle zu behandeln. Der erste Fall ist dabei der komplexere, da die versendete Nachricht auf Empfängerseite zunächst zwischengespeichert werden muss, bevor sie nach einem Aufruf der zugehörigen Empfangsoperation in den Empfangspuffer des Benutzers kopiert werden kann. Beim zweiten Fall ist dieser zusätzliche Zwischenschritt nicht notwendig; der Empfangspuffer ist sofort verfügbar.

Jede Instanz der Klasse `RESTDevice` verwaltet zwei Puffer, die `arrQueue` (*“arrive queue”*) und die `recvQueue` (*“receive queue”*). Ersterer enthält Nachrichten, die vom entsprechenden Prozess empfangen, jedoch noch nicht mit einer abgesetzten Empfangsoperation abgeglichen wurden, da noch keine solche existiert. In dieser Funktion entspricht die `arrQueue` dem oben beschriebenen Zwischenspeicher im ersten zu betrachten-

den Fall. Der zweite Puffer speichert die noch nicht abgeschlossenen Empfangsanfragen des Prozesses und kommt somit hauptsächlich in Fall zwei zum Tragen. Die Implementierung der beiden Puffer wurde vollständig von der mit MPJ Express mitgelieferten Klasse `NIODevice`, welche Socket-Kommunikation realisiert, übernommen. Gegen konkurrierenden Zugriff werden die beiden Puffer durch die Verwendung von Semaphoren geschützt.

Senden

Auf Senderseite zu betrachten sind die Methoden `isend()` (nicht blockierendes asynchrones Senden) und `issend()` (nicht blockierendes synchrones Senden) der Singleton-Klasse `RESTDevice`. Ihre blockierenden Pendants sind, wie oben beschrieben, auf diesen aufbauend implementiert. Die vollständigen Signaturen der beiden Methoden lauten:

```
public Request isend(Buffer buf, ProcessID destID, int tag, int context)
```

und

```
public Request issend(Buffer buf, ProcessID destID, int tag, int context)
```

Die Parameter haben dabei folgende Bedeutung:

- `buf`: Sendepuffer, enthält die zu sendenden Daten.
- `destID`: Identifikationsnummer des Sendeziels.
- `tag`: MPI-Nachrichten-Tag.
- `context`: MPI-Kommunikator-Kontext.

Der notwendige Programmcode der Funktion `isend()` ist eine echte Teilmenge der Funktion `issend()`. Letztere vollführt lediglich im Anschluss zusätzliche Aktionen zur Synchronisierung. Diese sind weiter unten beschrieben. Im Folgenden sei jedoch zunächst die gemeinsame – und für `isend()` vollständige – Funktion der beiden Methoden beleuchtet.

Zunächst wird überprüft, ob überhaupt eine Netzwerkkommunikation notwendig ist, das heißt, ob sich die Ziel-Identifikationsnummer von derjenigen des aktuellen Prozesses unterscheidet. Ist dies nicht der Fall – Sender und Empfänger sind identisch – kann die Sendeanfrage lokal abgewickelt werden. Hierzu wird die `recvQueue` des aktuellen Prozesses direkt auf einen bereits abgesetzten Empfangswunsch kontrolliert. Die

Zuordnung von Nachrichten zu Empfangsanfragen geschieht über die drei Merkmale Identifikationsnummer des sendenden oder empfangenden Prozesses, Nachrichten-Tag und Kommunikator-Kontext. Die Methode `rem()` der beiden verwalteten Puffer-Objekte nimmt diese Merkmale als Parameter entgegen und gibt, falls vorhanden, das nächste passende `RESTRecvRequest`-Objekt zurück. In diesem sind Empfangs- oder Sendepuffer referenziert. Ist ein Empfangswunsch (passendes `RESTRecvRequest`-Objekt) vorhanden, können die zu übertragenden Daten direkt in den dort referenzierten Empfangspuffer eingefügt werden. Wurde jedoch noch keine passende Empfangsoperation aufgerufen, wird die Nachricht als neues `RESTRecvRequest`-Objekt, das den Inhalt des Sendepuffers enthält, in die `arrQueue` des aktuellen Prozesses eingefügt.

Der interessantere Fall ist aber derjenige, der eintritt, wenn das Ziel der Sendeanfrage tatsächlich ein entfernter Prozess ist. In dieser Variante werden die Daten letztendlich als HTTP-Nachricht an den RESTful Webservice des anderen Prozesses übertragen. Um das zu ermöglichen, sind jedoch vorher noch einige Schritte notwendig. Damit die zu übertragende Nachricht in einen HTTP-Request eingebettet werden kann, muss diese zunächst in einer serialisierbaren Form vorliegen. Unglücklicherweise ist das unmittelbar verfügbare `RESTSendRequest`-Objekt nicht serialisierbar, da der darin enthaltene Puffer eine Instanz der MPJ Express Klasse `mpjbuf.Buffer` ist und diese intern ein `java.nio.ByteBuffer`-Objekt verwendet. Aus diesem Grund wurde eine neue Klasse – `RESTTransferObject` – erstellt, die im Konstruktor einen `RESTSendRequest` entgegennimmt und die darin enthaltenen Puffer-Bestandteile in simple `byte`-Arrays umwandelt. Diese können problemlos serialisiert werden. Zur Übertragung wird somit aus dem verfügbaren `RESTSendRequest`-Objekt ein `RESTTransferObject` erstellt.

Zur Realisierung des nicht blockierenden Charakters der momentan betrachteten Funktion `isend()` muss der Kontrollfluss an dieser Stelle aufgespalten werden. Hierfür wird ein neuer Thread erstellt, welcher sich im Folgenden um die weitere Abwicklung des Sendevorgangs kümmert. Im ursprünglichen Thread wird das noch nicht vollständig abgewickelte `RESTSendrequest`-Objekt von der Funktion zurückgegeben. Der neue Thread erhält, neben der Übergabe des erstellten `RESTTransferObject`, ebenfalls Zugriff auf dieses, um nach Abschluss des Sendevorgangs seinen Status auf *fertiggestellt* setzen zu können.

Der nun parallel ablaufende Thread überführt zunächst das `RESTTransferObject` selbst in ein `byte`-Array. Dies geschieht mithilfe der Java-Klasse `ByteArrayOutputStream`, welche in der Methode `toByteArray()` verwendet wird. Im nächsten Schritt erfolgt bereits die Übermittlung des HTTP-Requests durch erneute Verwendung eines Client-

Proxy-Objekts. Die Methode `transferSendRequest()` setzt ein HTTP-PUT auf der Ressource `/messages/arrivequeue` ab, dessen Body das serialisierte `RESTTransferObject` enthält. Dieses wird auf Seite des Webservices von RESTEasy erneut in ein `byte`-Array überführt und der Methode `transferSendRequest()` der `RESTServer`-Klasse als Argument übergeben. Diese deserialisiert das `RESTTransferObject` mithilfe der Methode `toObject()`, welche einen `ByteArrayInputStream` verwendet, und gibt das resultierende Objekt an die `RESTDevice` des Prozesses durch deren Methode `enqueueSendRequest()` weiter.

Die Methode `enqueueSendRequest()` übernimmt die Aufgabe, die Nachricht entweder in die `arrQueue` des Prozesses einzureihen, oder, falls bereits ein passender Empfangswunsch abgesetzt wurde, diesen abzuschließen. Hierzu wird zunächst die `recvQueue` auf einen solchen hin kontrolliert. Wird ein entsprechender `RESTRecvRequest` gefunden, können die übertragenen Daten direkt in den Empfangspuffer dieses übernommen werden. Ein entsprechendes Setzen des Status dieses Requests auf *fertiggestellt* und ein Benachrichtigen eventuell darauf wartender Threads durch eine notify-Methode, bildet hier – im Falle eines asynchronen Sendens – den Abschluss der Abwicklung dieser Nachricht. Wurde kein passender Empfangswunsch gefunden, wird ein neues `RESTRecvRequest` erzeugt, welches den Inhalt des Sendepuffers enthält und dieses in die `arrQueue` des Prozesses eingefügt.

Im Fall eines synchronisierten Sendens entscheidet der Rückgabewert der Methode `enqueueSendRequest()` auf Empfängerseite, ob auf Senderseite das Übertragen der Nachricht als abgeschlossen aufgefasst wird. Synchronisiertes Senden bei MPI bedeutet, dass dieses Abschließen erst passieren darf, wenn auf der Gegenseite eine passende Empfangsoperation aufgerufen wurde. Falls das bereits beim Empfang der Nachricht zutrifft, findet `enqueueSendRequest()` den entsprechenden `RESTRecvRequest` in der `recvQueue`. Der Rückgabewert ist ein String, der die jeweilige Information enthält und von RESTEasy als HTTP-Response an den sendenden Prozess zurückübertragen wird. Letzterer kann anschließend das synchronisierte Senden abschließen, oder, falls bei der Ankunft der Nachricht beim Empfangsprozess noch keine Empfangsanfrage vorlag, es zunächst zwischenspeichern. Hierfür dient die Hash-Map `waitingSynchronousSendRequests`.

Die Übertragung der Daten einer MPI-Nachricht in den letztendlichen Empfangspuffer kann an genau zwei Stellen erfolgen. Die erste Variante – das direkte Abschließen des Sendevorgangs in der Methode `enqueueSendRequest()` beim Vorliegen einer bereits abgesetzten Empfangsanfrage – wurde soeben betrachtet. Kann jedoch beim Ankommen der Nachricht am Empfangsprozess kein passender `RESTRecvRequest` in der `recvQueue`

gefunden werden, ist eine direkte Übertragung in einen Empfangspuffer nicht möglich. In diesem Fall erfolgt, wie bereits erwähnt, ein Einfügen der Nachricht in die `arrQueue`. Die Methode `enqueueSendRequest()` kehrt mit einem entsprechenden Rückgabewert zurück.

Empfangen

Um die letztendliche Abwicklung einer so zwischengespeicherten Nachricht – und somit die zweite Variante der Übertragung einer Nachricht in den Empfangspuffer – nachzuvollziehen, muss das Augenmerk darauf gerichtet werden, was passiert, wenn im parallelen Programm eine Empfangsoperation aufgerufen wird. Wie bereits dargestellt, wird eine solche am Ende auf die Funktion `irecv()` der verwendeten Device-Implementierung abgebildet. Diejenige der `RESTDevice` wird nun behandelt.

Im Umgang mit den beiden Puffern der `RESTDevice` gestaltet sich `irecv()` komplementär zu den Sendemethoden. Die Funktion liest aus der `arrQueue` und schreibt in die `recvQueue`. Zunächst wird erstere auf bereits über das Netzwerk empfangene oder vom selben Prozess dort abgelegte Nachrichten überprüft. Wird eine solche gefunden, kann sie durch Übertragung der Daten in den Empfangspuffer fertig abgewickelt werden und die Methode kehrt zurück. Entstammt eine so abgeschlossene Nachricht jedoch einem synchronisierten Senden – dies ist durch ein Flag innerhalb des Request-Objekts spezifiziert – muss noch eine Rückmeldung an den sendenden Prozess erfolgen. Zu diesem Zweck wird ein neuer HTTP-Request initiiert, welcher über die im RESTful Webservice exponierte Methode `signalSynchronousSend()` unter der Ressource `/messages/synchronous` den Abschluss des Sendeprozesses anstößt. Im Einzelnen wird auf Sendeseite vom `RESTServer` der übertragene Identifikationsschlüssel der Nachricht direkt an die Methode `signalSynchronousSend()` der `RESTDevice` weitergegeben. Dort kann das entsprechende Request-Objekt aus der Hash-Map entnommen, sein Status aktualisiert und darauf wartende Threads aufgeweckt werden.

Sollte beim Aufruf von `irecv()` jedoch noch keine passende Nachricht eingegangen sein, schlägt das Entnehmen einer solchen aus der `arrQueue` fehl. In diesem Fall wird ein neues `RESTRecvRequest`-Objekt erzeugt und dieses in die `recvQueue` eingefügt. Im Anschluss beendet sich die Funktion.

6. Umsetzung der REST-Prinzipien

Ziel dieses Kapitels ist es, Implementierungsdetails der im Zuge der vorliegenden Arbeit entwickelten Version des Systems aufzuzeigen und diese von den in Kapitel 5 dargestellten abzugrenzen. Wie bereits angedeutet wurde, hat die Projekt-Implementierung allem voran Defizite in der konsequenten Umsetzung der REST-Prinzipien. Nach einer detaillierten Betrachtung derselben lässt sich sogar in Frage stellen, ob der darin realisierte Webservice überhaupt rechtmäßig das Prädikat *RESTful* tragen darf. Die Behebung dieser sowohl konzeptionellen, als auch daraus folgend in der Implementierung zu findenden Fehler und die daraus resultierenden neuen Programmstrukturen werden im Folgenden behandelt.

Bei der Beschreibung der sich vom Master-Projekt unterscheidenden Implementierungsdetails des entwickelten Systems wird ein anderes Vorgehen als in Kapitel 5 gewählt. Statt einen typischen Programmablauf nachzuvollziehen, bilden die in Kapitel 1 analysierten REST-Prinzipien die Ausgangspunkte für Darstellungen von Code-Details. Der Grund für ein solches Vorgehen ist die Tatsache, dass die konsequente und korrekte Umsetzung eben dieser REST-Prinzipien das hauptsächliche Motiv dafür ist, die Implementierung des Master-Projekts im Zuge der vorliegenden Arbeit zu revidieren.

In Kapitel 1 wird bereits angedeutet, wie eine Realisierung der einzelnen Prinzipien im betrachteten System aussehen könnte. Die theoretische Umsetzbarkeit variiert dabei sehr stark. Beispielsweise wird bereits dort erkannt, dass Prinzip 4 (Statuslose Kommunikation) je nach Sichtweise, entweder keine zusätzlichen Maßnahmen in der Implementierung erfordert und damit implizit erfüllt ist, oder nicht sinnvoll anwendbar ist.

Auch Prinzip 3 (Unterschiedliche Repräsentationen) wirft einige Fragen auf. Die naheliegende Realisierungsvariante für dieses – das zusätzliche Anbieten der als `byte`-Array serialisierten MPI-Nachrichten in weiteren Formaten, wie XML oder HTML, beispielsweise zum Zweck einer menschenlesbaren Repräsentation – lässt sich im MPI-Umfeld als nicht praxisrelevant ausschließen. Vorgeschlagen wird jedoch diesbezüglich eine weitere Möglichkeit, das Prinzip umzusetzen: Es könnten für verschiedene Versionen der Java Virtual Machine unterschiedliche Serialisierungsoptionen angeboten werden. Für

eine wirklich plattformunabhängige Implementierung, die bereit für den Produktionseinsatz ist, müsste eine solche Möglichkeit bestehen. Da die Einbindung verschiedener Serialisierungsmechanismen jedoch einerseits auf Grund ihrer technischen Komplexität im Rahmen der vorliegenden Arbeit nicht abgedeckt werden kann und sie andererseits wegen ihrer theoretisch relativ einfach überblickbaren Struktur wahrscheinlich keine weiteren wissenschaftlichen Erkenntnisse hervorbringen würde, wird das REST-Prinzip der unterschiedlichen Repräsentationen zunächst ebenfalls vernachlässigt. Eine Umsetzung einer solchen Funktionalität wird aber für zukünftige Versionen des Systems nahegelegt.

Die Umsetzung von zwei der fünf REST-Prinzipien erfordert somit zunächst keine Implementierungsarbeit. Wie die restlichen drei Prinzipien – Ressourcen mit eindeutiger Identifikation, Standardmethoden und Hypermedia – Einzug in das System halten, soll nun im Folgenden betrachtet werden.

6.1. Ressourcen mit eindeutiger Identifikation

In Abschnitt 2.2 wurde das Interface `RESTResource` präsentiert. Durch dieses werden alle durch HTTP erreichbaren Ressourcen des RESTful Webservice definiert. Dazu werden relative URIs auf Java-Methoden abgebildet. Die in 2.2 dargestellte Version des Master-Projekts enthält offensichtlich eine exakt festgelegte und damit starre Anzahl solcher Ressourcen. Im Kontrast dazu hat das Interface in der Version der Masterarbeit folgendes Aussehen:

```
@Path("/mpi")
public interface RESTResource {

        String pathTemplate = "messages/{source}/{destination}/{context}";

        @GET
        @Path(pathTemplate + "/{tag}")
        @Produces("text/plain")
        public Response getMessage(@PathParam("source") String src,
                        @PathParam("destination") String dest,
                        @PathParam("context") String context,
                        @PathParam("tag") String tag,
                        @QueryParam("recvid") int recvid,
                        @QueryParam("frecv") int frecv,
                        @QueryParam("fsend") int fsend);

        @GET
```

```
@Path(pathTemplate)
@Produces("text/plain")
public Response getMessageAnyTag(@PathParam("source") String src,
                @PathParam("destination") String dest,
                @PathParam("context") String context,
                @QueryParam("recvid") int recvid,
                @QueryParam("frecv") int frecv,
                @QueryParam("fsend") int fsend);

@PUT
@Path(pathTemplate + "/{tag}")
@Consumes("text/plain")
public Response putMessage(@PathParam("source") String src,
                @PathParam("destination") String dest,
                @PathParam("context") String context,
                @PathParam("tag") String tag,
                byte[] message);

@PUT
@Path(pathTemplate +"/{tag}/signal")
@Consumes("text/plain")
public Response signalResource(@PathParam("source") String src,
                @PathParam("destination") String dest,
                @PathParam("context") String context,
                @PathParam("tag") String tag,
                String body);

@GET
@Produces("text/plain")
public String getNodeInformation();

@POST
@Path("start")
@Produces("text/plain")
public String startProgram();

@PUT
@Path("init")
@Consumes("text/plain")
public Response initNode(String nodeId);

@PUT
@Path("processes")
@Consumes("text/plain")
```

```
        public String receiveProcesses(String processes);

        @POST
        @Path("programs")
        @Consumes("text/plain")
        public String createProgram(byte[] file);

}
```

Die letzten fünf Methoden des Interfaces `RESTResource` finden sich in nahezu derselben Struktur ebenfalls in der Version des Master-Projekts und werden hier nur der Vollständigkeit wegen erneut aufgeführt. Sie werden lediglich in der in 5.1 beschriebenen Startphase des Systems verwendet, welche durch die Überarbeitung des Systems kaum nennenswerte Änderungen erfahren hat. Insofern sind die Ausführungen aus Abschnitt 5.1 größtenteils ebenfalls für die Version der Masterarbeit gültig. Lediglich die Methode `initNode()` wurde zur Realisierung des REST-Prinzips *Hypermedia* angepasst. Näheres hierzu ist in Abschnitt 6.3 zu finden.

An dieser Stelle von Belang sind aber die ersten vier Methoden der Schnittstelle. Diese ersetzen zum einen die starre Ressource `/messages/arrivequeue`, über welche in der alten Variante der vollständige MPI-Nachrichtenaustausch abgewickelt wird. Zum anderen erfährt darüber hinaus die ebenfalls starre Ressource `/messages/synchronous` eine Erneuerung. Im Kontext des REST-Prinzips *Ressourcen mit eindeutiger Identifikation* ist die eingeführte Pfadstruktur der exponierten Ressourcen, welche auf diese vier neuen Methoden abgebildet werden, besonders interessant. Diese sind nun nicht mehr starr, sondern können durch den JAX-RS-Mechanismus der `@PathParam`- und `@QueryParam`-Annotationen sehr vielfältige Ausprägungen annehmen. Hierzu wird zum einen der `@Path`-Annotation der jeweiligen Methode ein URI-Template [3] übergeben, das in geschweiften Klammern diverse variable Pfadbestandteile enthält. Diese können anschließend mithilfe von `@PathParam` in die Parameterliste der Methode injiziert werden. Zum anderen geschieht das gleiche mit Parametern, die mit dem Trennzeichen `?` vom Client an die angefragte URI angehängt werden, durch die Verwendung von `@QueryParam`.

Wie in Kapitel 1 dargelegt wurde, geschieht die Exponierung von Ressourcen in der neuen Version des Systems auf der Ebene der MPI-Nachrichten. Das bedeutet, dass jeder Nachricht, sowohl auf Sender-, als auch auf Empfängerseite, eine URI und somit eine HTTP-Ressource zugeordnet wird. Die eindeutige Identifikation einer solchen Nachricht geschieht über die Kombination ihrer Merkmale *Quelle*, *Ziel*, *Kontext* und *Tag*. Eine

daraus aufgebaute URI hat folgende allgemeine, bereits in Kapitel 1 eingeführte, Form:

```
http://<prozess>/REST-MPI/mpi/messages/<sender>/<receiver>/<context>/<tag>
```

Der letzte Teil dieser URI kann zum Ausdruck des MPI-Wilcard-Elements `ANY_TAG` bei einer `GET`-Operation weggelassen werden. Aus diesem Grund muss der Tag das letzte Element des Pfades darstellen. Eine Ausnahme von dieser Regel bildet diejenige Ressource, die auf die Methode `signalResource()` abgebildet wird. Diese definiert als letztes Pfadelement das Wort `signal`. Die entsprechende URI wird jedoch nur von einem empfangenden Prozess nach Erhalt einer synchronisierten Nachricht über ein HTTP-`PUT` angesprochen. Zu diesem Zeitpunkt ist der Tag der ursprünglichen Nachricht immer bekannt. Ein Wildcard-Element und somit das Weglassen des Tag-Pfadbestandteils ist somit nicht notwendig.

Die auf die beschriebene Weise aufgebauten URIs werden an den entsprechenden Stellen von den HTTP-Clients auf Basis der Inhalte der vorliegenden Request-Objekte konstruiert. Die weitere Abwicklung auf Serverseite, angefangen mit der Ausführung der Methoden, auf welche die Ressourcen abgebildet werden, wird unter anderem in Abschnitt 6.2 dargestellt.

6.2. Standardmethoden

Wie bereits mehrmals erwähnt, setzt die Implementierung des Master-Projekts – ebenso wie die gängigen klassischen MPI-Implementierungen – durchgängig auf reine Push-Kommunikation. MPI-Nachrichten werden ohne Ausnahme über den Aufruf eines HTTP-`POST` auf einer einzigen zentralen Ressource realisiert. Aufgrund von technischen Optimierungen in vorhandenen Infrastrukturen, die beispielsweise dem Internet zu Grunde liegen, ist es jedoch gerade die Nutzung von HTTP-`GET`, die einem RESTful Ansatz, gegenüber anderen Webservice-Lösungen, einen klaren Performancevorteil verschafft. Technologien, wie transparentes Caching, sind hierfür die technische Grundlage. Darüber hinaus lassen sich in einem als RESTful bezeichneten Webservice exponierte Ressourcen, wie die hier vorliegenden MPI-Nachrichten, viel besser mit den zugrunde liegenden Intentionen des REST-Prinzips *Standardmethoden* vereinen, wenn diese ebenfalls per `GET` ansprechbar sind. Aus diesen Gründen ist es notwendig, zusätzlich zur vorhandenen Push-Kommunikation, ebenfalls Pull-Anfragen zu erlauben. Bei letzteren wird die Netzwerkkommunikation vom empfangenden Prozess initiiert. Dies ähnelt zu einem großen

Teil beispielsweise der Kommunikationsstruktur eines Webbrowsers und erscheint schon deswegen als angebrachter für internetähnliche Netzwerke.

An dem in Abschnitt 6.1 gezeigten Interface `RESTResource`, welches für die Definition der exponierten Ressourcen und der darauf angewandten Methoden zuständig ist, lässt sich bereits die neu eingeführte Möglichkeit des Einsatzes beider Kommunikationsrichtungen erkennen. Die beiden Methoden `getMessage()` und `putMessage()` wickeln Anfragen an die strukturell gleichen Ressourcen ab. Erstere behandelt `GET`-Requests, die zweite Anfragen, welche durch ein HTTP-`PUT` ausgelöst werden. Die Methode `getMessageAnyTag()` ist lediglich eine Abwandlung von `getMessage()`, die verwendet wird, falls eine Empfangsanfrage abgesetzt wird, die das Tag-Wildcard-Element enthält. In diesem Fall ist ein geringfügig anderes Vorgehen nötig. An dieser Stelle sei bereits betont, dass die Einführung der zweiten Kommunikationsrichtung die Komplexität der Device-Implementierung stark erhöht. Aus diesem Grund können die Anstrengungen, die den hier beschriebenen Implementierungsdetails zugrunde liegen, als zentrales Arbeitspaket bei der Entwicklung des vorliegenden Systems angesehen werden.

6.2.1. Push-Richtung

Zunächst wird die Push-Richtung der Interprozesskommunikation betrachtet. Dieses Vorgehen wird gewählt, da die im Folgenden behandelten Strukturen am stärksten an die Implementierung des Master-Projekts erinnern. So wird der gedankliche Übergang zu einer zweiten Kommunikationsrichtung erleichtert. Auch hier ist es so, dass letztendlich alle Sendeoperationen auf die Methode `isend()` der `RESTDevice` heruntergebrochen werden. Aus diesem Grund ist es naheliegend, erneut diese Funktion zuerst zu betrachten. Der Ablauf einer solchen Sendeoperation ist in Abbildung 6.1 auf der Device-Ebene dargestellt.

Der Code zur Abwicklung einer Sendeoperation innerhalb desselben Prozesses, das heißt ohne Notwendigkeit von echter Netzwerkkommunikation, ist identisch zu demjenigen aus der Projekt-Implementierung. Muss jedoch eine Verbindung zu einem entfernten Prozess aufgebaut werden, unterscheiden sich die Umsetzungen stark. Die hauptsächliche konzeptionelle Änderung ist dergestalt, dass nach Aufruf einer Sendeoperation der Prozess nicht sofort von sich aus eine Netzwerkverbindung initiiert, sondern eventuell eine passende bereits eingegangene Empfangsanfrage beantwortet und deren aufgebaute Verbindung zur Übertragung nutzt.

Ein Kernelement dieses Prozesses ist eine auf den ersten Blick relativ komplex aufgebaute Datenstruktur namens `availableSendRequests`. Diese ist eine Instanz von

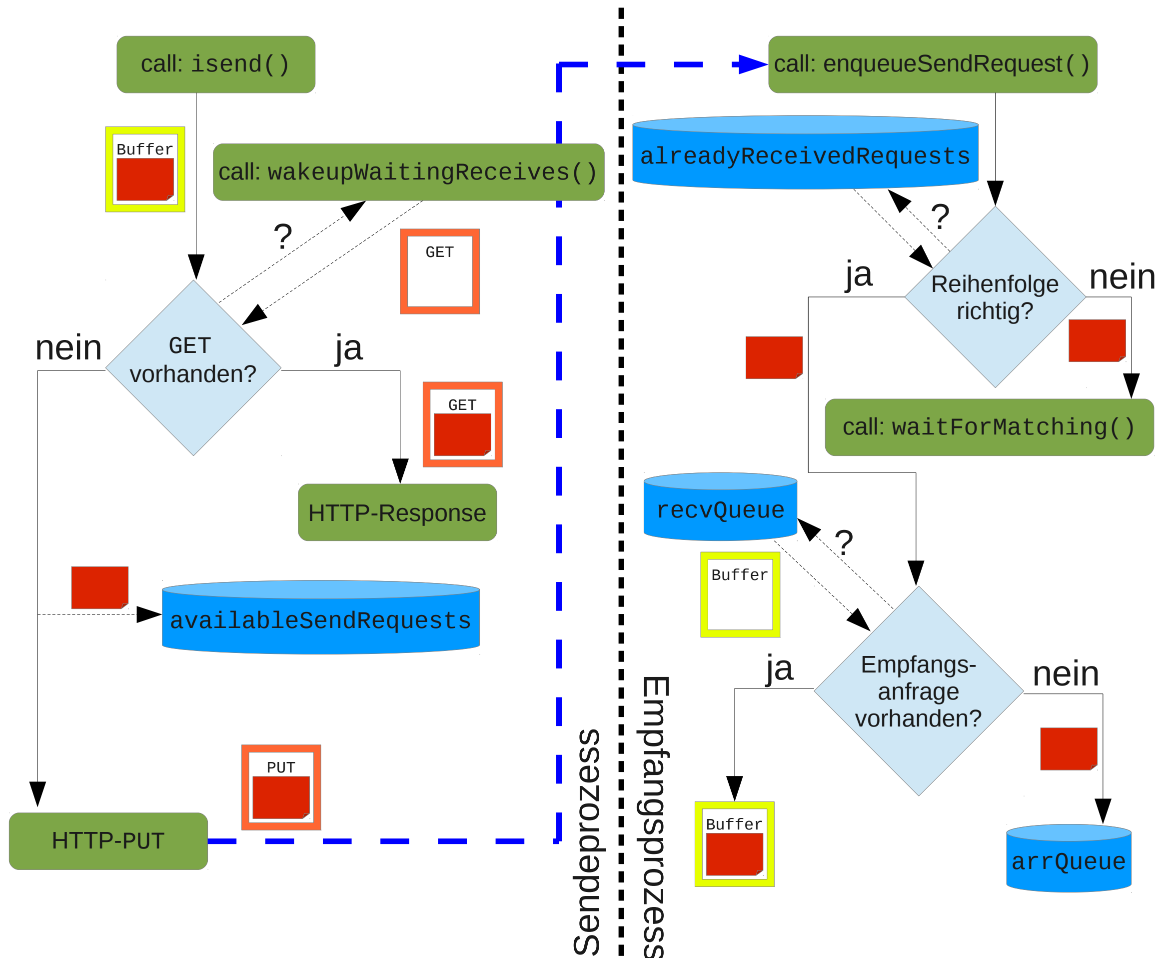


Abbildung 6.1.: Ablauf einer Sendeoperation auf Ebene der Device-Implementierung. Die von der Server-Klasse vollführten Operationen zur Entgegennahme und Weiterleitung der Nachricht sind nicht explizit dargestellt. Aktionen: Grün, Datenstrukturen: Blau, Sende- und Empfangspuffer: Gelb, HTTP-Requests: Orange.

HashMap<UUID, HashMap<RESTRequestKey, LinkedList<RESTTransferObject>>>. Sie ermöglicht den effizienten Zugriff auf eine Liste von zu versendenden Nachrichten (RESTTransferObject) unter Angabe der Identifikationsnummer des Zielprozesses und des RESTRequestKey der Nachricht. Die Klasse des letzteren übernimmt die Kapselung der Identifikationsmerkmale einer MPI-Nachricht – Zielprozess, Kontext und Tag. Die Methode isend() legt zu versendende Nachrichten zunächst in dieser Datenstruktur ab.

Der nächste Schritt ist ein Aufruf der Funktion wakeUpWaitingReceives(). Diese übernimmt die Aufgabe, passende wartende und bereits eingegangene Empfangsanfragen aufzufinden, die Nachrichtenobjekte an sie zu übergeben und sie anzustoßen, ihre Ausführung fortzusetzen. Zu diesem Zweck konstruiert die Methode zwei RESTRequestKeys – einen, der die komplette Signatur der Nachricht wiedergibt und einen für den Fall eines gesetzten ANY_TAG-Wildcard-Elements, der nur den Zielprozess und den Kontext berücksichtigt. Diese beiden Identifikationsobjekte werden anschließend genutzt, die Liste der eingegangenen Empfangsanfragen des Zielprozesses zu durchsuchen. Im Fall von gefundenen Anfragen für beide Objekte, wird diejenige ausgewählt und beantwortet, die zuerst einging.

Findet wakeUpWaitingReceives() keine passende Empfangsanfrage, kann letztendlich die Initiierung einer Push-Kommunikationsoperation erfolgen. Diese Aufgabe übernimmt aufgrund der Notwendigkeit einer asynchronen Ausführung ein eigener Thread der Klasse HttpRequestThread. Das Konzept der Client-Proxy-Objekte, wie es in der Implementierung des Master-Projekts eingesetzt wurde, findet keine Verwendung mehr, da dieses sehr wenig Einfluss auf die relativ vielfältigen Mechanismen eines HTTP-Requests zulässt. Stattdessen wird direkt die Open-Source-Bibliothek *Apache HttpClient*[1] [30] eingesetzt, auf welcher im Übrigen das Client-Framework von RESTEasy intern auch aufbaut. Diese erlaubt beispielsweise zur Performanceoptimierung Optionen, wie die Deaktivierung des *Stale Connection Checks*, eine Operation, die automatisch serverseitig beendete HTTP-Verbindungen abbaut und dafür jedesmal 15 bis 30 Millisekunden braucht [29]. Das dadurch eingeführte Risiko einer I/O Exception muss anschließend manuell behandelt werden.

Wurde nun erfolgreich vom Sendeprozess ein HTTP-Request abgesetzt, muss dieser auf der Gegenseite behandelt werden. Die Ressource des RESTful Webservice, die im Fall der Push-Kommunikation der Sender durch ein HTTP-PUT anspricht, wird durch das Interface RESTResource auf die Methode putMessage abgebildet. Diese soll in den folgenden Ausführungen betrachtet werden.

[1]Verwendete Version: 4.2.2

Wie bereits mehrmals erläutert, wird das Interface `RESTResource` durch die Klasse `RESTServer` implementiert. Auf dieser Ebene gestaltet sich die Implementierung der Funktion sehr ähnlich zu ihrem Äquivalent aus der alten Version – `transferSendRequest()`. Die in Form eines `byte`-Arrays serialisierte MPI-Nachricht wird entgegengenommen, durch die bereits vorgestellte Funktion `toObject` in ein `RESTTransferObject` deserialisiert, und der Methode `enqueueSendRequest()` der `RESTDevice` übergeben. Die Funktion `transferSendRequest()` würde nach diesem Schritt bereits zurückkehren, `putMessage()` vollführt noch einige weitere Aktionen. Diese basieren jedoch zum Teil auf dem Rückgabewert von `enqueueSendRequest()` und sollen deswegen erst im Anschluss an die nun folgende Betrachtung dieser Methode erläutert werden.

Durch die Einführung der zweiten Kommunikationsrichtung und durch die Tatsache, dass die beiden Kanäle völlig asynchron und parallel arbeiten, ist es nun zu jeder Zeit möglich, dass durch unvorhersehbare Verzögerungen in der Übertragung innerhalb eines Kanals, sich Nachrichten gegenseitig zeitlich überholen. Außerdem kann es passieren, dass manche Nachrichten sowohl durch ein `PUT`, als auch durch ein `GET` übertragen werden und somit doppelt beim Empfänger ankommen. Ein Forcieren der richtigen Reihenfolge und des nur einmaligen Übertragens jeder Nachricht bereits bei der Netzwerkkommunikation, würde die beiden Übertragungskanäle stark synchronisieren und ihren parallelen Charakter in einen seriellen verwandeln. Dies hätte entscheidende Performancenachteile. Es muss jedoch trotzdem sichergestellt werden, dass weder zwei verschiedene Empfangsanfragen mit derselben duplizierten Nachricht, noch in der falschen Reihenfolge beantwortet werden. Aus diesen Gründen wird die Behandlung solcher Phänomene an anderen, wie der beispielsweise nun folgenden, Stellen vorgenommen.

Die erste Aktion der Methode `enqueueSendRequest()` bei der Abwicklung einer empfangenen Nachricht ist die Überprüfung der Hash-Map `alreadyReceivedRequests` darauf, welche die letzte eindeutige Identifikationsnummer des sendenden Prozesses war, die abgewickelt wurde. Diese Nummer – eine Ganzzahl – ist eine Member-Variable der Klasse `RESTTransferObject`, welche eindeutig für einen sendenden Prozess ist und auf diesem bei jeder Instantiierung von `RESTTransferObject` hochgezählt wird. Eine weitere Information, die `RESTTransferObject` enthält, ist die ID der letzten Nachricht, die an den empfangenden Prozess adressiert war. Mit diesen beiden Zahlen ist eine Aussage darüber möglich, ob die empfangene Nachricht bereits endgültig abgewickelt werden kann oder noch zwischengespeichert werden muss. Falls vorher noch eine andere Nachricht des sendenden Prozesses empfangen werden muss, wird das `RESTTransferObject` an die Methode `waitForMatching()` übergeben und `enqueueSendRequest()` kehrt zurück.

Innerhalb von `waitForMatching()` wird das `RESTTransferObject` einer für dessen Sendeprozess zuständigen `PriorityQueue` hinzugefügt, deren `Comparator` dafür sorgt, dass die enthaltenen Nachrichten nach den für sie jeweils vorher zu empfangenden Nachrichten-IDs sortiert werden. So wird sichergestellt, dass diejenige Nachricht, welche der Warteschlange entnommen wird, als nächste abgewickelt werden kann. Die für die jeweiligen Sendeprozesse zuständigen `PriorityQueues` sind über die Hash-Map `waitingTransfers` durch Angabe der Prozess-ID zugreifbar.

Im Fall, dass jedoch die empfangene Nachricht sofort abgewickelt werden kann, da erkannt wurde, dass die letzte Nachricht des Sendeprozesses an diesen Empfangsprozess schon bearbeitet wurde, gestaltet sich der weitere Ablauf der Methode `enqueueSendRequest()` identisch zu ihrer Variante aus der Projekt-Implementierung: Die `recvQueue` wird auf eine passende, bereits abgesetzte, Empfangsoperation überprüft. Wird eine solche gefunden, können die Daten unmittelbar in den Empfangspuffer kopiert werden. Ist noch kein solcher Empfangswunsch vorhanden, wird ein neues `RESTReceiveRequest`-Objekt erzeugt und dieses in der `arrQueue` des Prozesses abgelegt.

Nachdem eine Nachricht in diesem zweiten Fall fertig behandelt wurde, ist es möglich, dass weitere Nachrichten vom gleichen Sendeprozess existieren, die bereits zu einem früheren Zeitpunkt empfangen wurden, aber nicht abgewickelt werden konnten, da eben diese, soeben finalisierte, Nachricht vorher bearbeitet werden musste. Solche wartenden Nachrichten müssen nun erneut betrachtet werden. Zu diesem Zweck existiert die Implementierung eines weiteren Threads, der nur dafür zuständig ist, wartende Nachrichten, die zu einem bestimmten Zeitpunkt valide werden, erneut an `enqueueSendRequest()` auszuhändigen. Die Klasse `MatchingThread` enthält zur Erledigung dieser Aufgabe eine Instanz von `LinkedBlockingQueue<UUID>`, die `wakeUpQueue`. Ein Thread dieser Klasse wird bereits mit der Instantiierung der `RESTDevice` gestartet. Dieser nutzt die Methode `take()` der `wakeUpQueue`, um darauf zu warten, dass diese einen Eintrag enthält. Wird eine Nachricht, wie soeben beschrieben, durch die Funktion `enqueueSendRequest()` abgearbeitet, trägt letztere die Identifikationsnummer des zugehörigen Sendeprozesses in die `wakeUpQueue` ein. Daraufhin wacht der `MatchingThread` auf und übergibt die ID an die Methode `wakeUpWaitingTransfer()`. Innerhalb dieser wird die nächste wartende Nachricht aus der entsprechenden PriorityQueue – diese enthält, wie oben beschrieben, noch abzuwickelnde Nachrichten eines bestimmten Sendeprozesses, sortiert nach zuvor zu bearbeitenden Nachrichten-IDs – entnommen und mit dieser erneut `enqueueSendRequest()` aufgerufen. Existieren nach Bearbeitung dieser Nachricht noch immer wartende

`RESTTransferObjects`, beginnt die beschriebene Schleife erneut und `enqueueSendRequest()` wird immer wieder aufgerufen bis letztendlich die Warteschlange leer ist.

Diejenige Ressource, die im Interface `RESTResource` auf die Methode `signalResource()` abgebildet wird, wird dafür verwendet, synchronen Sendeoperationen des aktuellen Prozesses, die darauf warten, dass auf der Gegenseite eine passende Empfangsoperation aufgerufen wird, dies mitzuteilen. In der Implementierung von `signalResource()` im `RESTServer` wird ein entsprechender `RESTRequestKey` konstruiert und dieser an die Methode `signalSynchronousSend()` der `RESTDevice` weitergegeben. Dort erfolgt eine Anfrage an die Hash-Map `waitingSynchronousSendRequests`, welche den Request zurückgibt, der nun als abgeschlossen markiert werden kann. Darauf wartende Prozesse werden aufgeweckt.

6.2.2. Pull-Richtung

Das Konzept, welches die im Folgenden beschriebenen Implementierungselemente realisieren, stellt, sowohl im Vergleich zu klassischen MPI-Systemen, als auch zur Implementierung aus dem Masterprojekt, die größte Neuerung dar. Wie bereits am Anfang dieses Abschnitts erwähnt wurde, ist es zur konsequenten Umsetzung des REST-Prinzips *Standardmethoden* notwendig, die zentralen Ressourcen des Systems ebenfalls per HTTP-`GET` verfügbar zu machen. Dies impliziert jedoch in den MPI-Kommunikationsstrukturen die Einführung von sogenannter Pull-Kommunikation, also Kommunikationsoperationen, die vom Empfänger der Nachricht initiiert werden.

Bereits an den zusätzlichen Mechanismen, die zur Realisierung klassischer Push-Kommunikation unter Anwesenheit einer zweiten Kommunikationsrichtung, eingeführt werden müssen, sieht man, dass sich die Komplexität der Device-Implementierung in der Tat stark erhöht. Die Strukturen, welche letztendlich ein jederzeit absetzbares HTTP-`GET` ermöglichen, steigern diese Komplexität noch mehr.

Der grundlegende Gedanke ist, dass ein Prozess, der an einer bestimmten Stelle des parallelen Programms eine Empfangsoperation aufruft, nicht mehr untätig darauf warten muss, dass eine passende Nachricht eingeht, die Empfangsanfrage befriedigt. Vielmehr kann ein solcher Prozess nun sofort aktiv werden, von sich aus eine Netzwerkkommunikation initiieren und direkt bei einem passenden Sendeprozess die Daten anfragen und abholen. Ein solches Vorgehen hat neben verbesserter architektonischer REST-Konformität den offensichtlichen Vorteil, dass die Zeit, die der Sendeprozess eventuell noch braucht, um die Nachricht fertig zur Übertragung bereitzustellen, bereits zum Auf-

bau einer Netzwerkverbindung genutzt werden kann. Inwiefern sich dies in der Praxis auf die Performance des Systems auswirkt, wird in Kapitel 9 gezeigt.

Im Folgenden soll, ebenso wie bei der Beschreibung der Push-Kommunikationsrichtung, der Kontrollfluss im Code der Systemimplementierung, beginnend mit dem Aufruf einer Empfangsoperation bis zum Eingang der Daten im Empfangspuffer, nachvollzogen werden. Als Gegenstück zu `isend()` als grundlegende Sendeoperation, auf die alle abstrakteren Varianten abgebildet werden, findet sich auf Empfangsseite die Funktion `irecv()`. Auch hier muss im Grunde nur diese betrachtet werden. Der grundlegende Ablauf – erneut auf der Device-Ebene – ist in Abbildung 6.2 dargestellt.

Die Unterschiede zur Projekt-Implementierung fallen im Gegensatz zur Sendeseite in dieser Methode weniger drastisch aus. Im Grunde wird lediglich, falls keine passende, bereits eingegangene Nachricht in der `arrQueue` des Empfangsprozesses gefunden wurde, zusätzlich zum Einfügen eines `RESTReceiveRequest`-Objekts in die `recvQueue`, aktiv ein HTTP-Request in Richtung Sendeprozess initiiert. Diese Aufgabe übernimmt ebenfalls die Klasse `HttpRequestThread` in einem eigenen Thread. Zusätzlich zu einer reinen HTTP-`GET`-Anfrage auf der jeweiligen Ressource, welche die zu empfangende Nachricht repräsentiert, codiert der Empfangsprozess drei zusätzliche Informationen in den Request. Dies geschieht in Form von sogenannten *Query-Parametern*, also per `?`-Trennzeichen an die eigentliche URL angehängte Schlüssel-Wert-Paare. Die drei Informationen sind die ID der letzten abgeschlossenen Empfangsoperation des Prozesses (`frecv`), die ID der letzten eingegangenen Nachricht (`fsend`) und die ID der aktuellen Empfangsoperation (`recvid`). Die Kenntnis dieser drei Werte auf der Seite des Sendeprozesses ist notwendig, um einerseits beurteilen zu können, welche vorgehaltenen Nachrichten nicht mehr angefragt und folglich gelöscht werden können. Andererseits dienen diese Daten auch dazu, wartende Empfangsanfragen aufzuwecken und abzubrechen, deren zugehörige Nachricht bereits erfolgreich übertragen wurde.

Per HTTP angefragt wird eine der beiden neu eingeführten `GET`-Mappings der Schnittstelle `RESTResource` auf dem Sendeprozess. An dieser Stelle sei erneut betont, dass erstere von beiden exakt über die gleiche URI ansprechbar ist, wie die auf `putMessage()` abgebildete Ressource. Die konzeptionelle Rolle des zweiten `GET`-Mappings als Anfragepunkt für Nachrichten mit beliebigem Tag wurde ebenfalls bereits beschrieben. Die dafür zusätzlich nötigen Mechanismen in der Implementierung werden im Folgenden an den passenden Stellen erwähnt.

Zunächst sei jedoch die Implementierung der Methode `getMessage()` in der Klasse `RESTServer` betrachtet. Diese übernimmt lediglich zwei Aufgaben. Zum einen gibt sie die

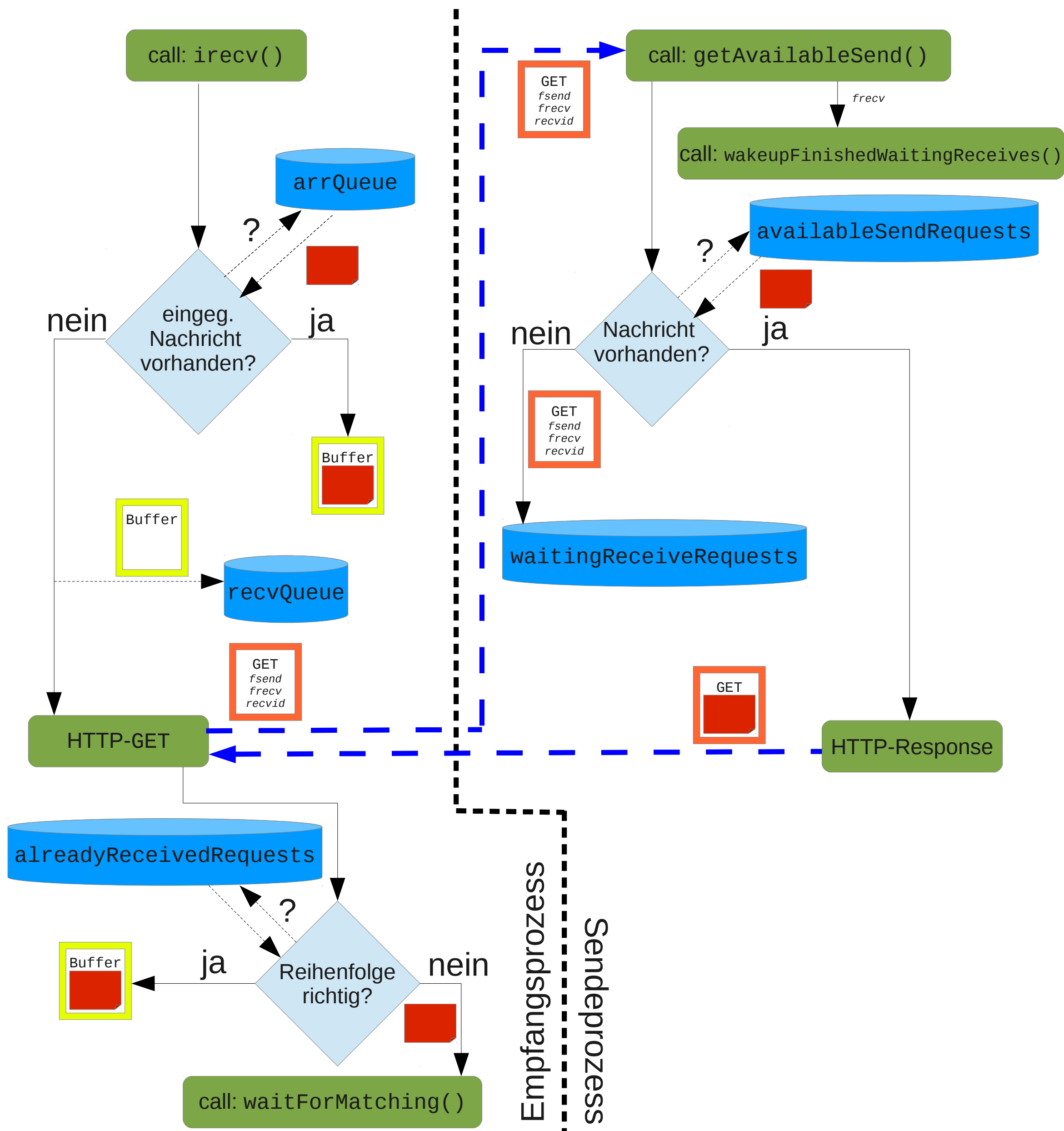


Abbildung 6.2.: Ablauf einer Empfangsoperatoin auf Ebene der Device-Implementierung. Die von der Server-Klasse vollführten Operationen zur Entgegennahme und Weiterleitung der Nachricht sind nicht explizit dargestellt. Aktionen: Grün, Datenstrukturen: Blau, Sende- und Empfangspuffer: Gelb, HTTP-Requests: Orange.

Anfrage inklusive aller eingegangenen Parameter an die Methode `getAvailableSend()` der `RESTDevice` weiter. Zum anderen konstruiert sie aufbauend auf dem Rückgabewert der letzteren eine entsprechende HTTP-Response. Logisch ist somit nun die Fortführung der Betrachtungen mit den Details von `getAvailableSend()`.

Diese Funktion kann aus architektonischer Sicht als auf der Sendeseite angesiedeltes Gegenstück zu `enqueueSendRequest()` auf der Empfangsseite gesehen werden. Zunächst wird die Methode `wakeupFinishedWaitingReceives()` aufgerufen, welche auf Basis des Wertes `frecv` Empfangsanfragen abbricht, deren zugehöriger Aufruf von `irecv()` auf der Empfangsseite bereits erfolgreich beendet wurde. Dazu wird die Datenstruktur `waitingReceiveRequests` – eine Instanz von `HashMap<UUID, LinkedList<RESTRequestKey>>` – durchsucht und die IDs der für den entsprechenden Empfangsprozess darin enthaltenen `RESTRequestKey`s mit dem Wert `frecv` abgeglichen. Ist die jeweilige ID kleiner, kann der dazugehörige Request abgebrochen werden.

Im weiteren Verlauf von `getAvailableSend()` geht es darum, eine für die jeweilige Empfangsanfrage passende Nachricht aus der oben bereits eingeführten, relativ komplexen Datenstruktur `availableSendRequests()` zu entnehmen. Hierzu muss zusätzlich zu dem Fall einer exakt durch Empfänger, Kontext und Tag spezifizierten Nachricht die Anfragevariante mit beliebigem Tag betrachtet werden. Im ersten Fall, kann einfach die für den Empfangsprozess zuständige Hash-Map mit dem hier eindeutig konstruierbaren `RESTRequestKey` angefragt und das erste Element der zurückgegebenen aufsteigend chronologisch sortierten Nachrichtenliste verwendet werden. Im Fall eines beliebigen Tags müssen jedoch alle Nachrichtenlisten aller verfügbaren `RESTRequestKey`s eines Empfangsprozesses durchsucht werden, bis eine Nachricht mit passendem Kontext gefunden wird. Dadurch, dass die Suchreihenfolge der ebenfalls in einer Hash-Map enthaltenen Hash-Maps, die `RESTRequestKey`s auf Nachrichtenlisten abbilden, aufgrund der Eigenschaften der Datenstruktur zufällig ist, wird vermieden, bestimmte Nachrichtenstrukturen implizit zu bevorzugen und eventuell mit höherer Wahrscheinlichkeit zuerst auffindbar zu machen.

Wurde nun eine passende Nachricht gefunden, besteht der nächste Schritt darin, in dem Fall, dass diese einer synchronisierten Sendeanfrage entstammt, darauf wartende Prozesse auf Sendeseite aufzuwecken. Danach wird noch ein Sonderfall behandelt, der dadurch auftreten kann, dass alle zu sendenden Nachrichten aufgrund der Möglichkeit des Fehlschlags eines HTTP-Requests, während diese aktiv versendet werden, gleichzeitig für Empfangsanfragen verfügbar gehalten werden. Dadurch kann es passieren, dass eine Nachricht, die soeben in `getAvailableSend()` ausgewählt wurde, eine Empfangsan-

frage zu befriedigen, gleichzeitig aktiv per Push-Kommunikation versendet wird und dieser Prozess noch nicht abgeschlossen wurde. Tritt ein solcher Fall ein, ist der zugehörige `RESTRequestKey` in der Datenstruktur `sendingMessages` enthalten und der ausführende Thread wird angehalten, bis das aktive Senden – erfolgreich oder nicht – beendet wurde. So kann vermieden werden, dass Nachrichten doppelt versendet werden. Sollte die Push-Kommunikation erfolgreich sein, wird die Ausführung von `getAvailableSend()` abgebrochen. Schlägt sie fehl oder war keine solche vorhanden, wird die gefundene Nachricht von der Funktion zurückgegeben und auf der Ebene des `RESTServer` in den Body der HTTP-Response eingebettet. Liegt jedoch von Anfang an noch keine passende Nachricht für die Empfangsanfrage vor, wird die Ausführung des aktuellen Threads angehalten und der zur Anfrage gehörende `RESTRequestKey` in `waitingReceiveRequests` eingefügt. Der `RESTRequestKey` fungiert darüber hinaus als Sychronisationsobjekt, was bedeutet, dass auf ihm von verschiedenen Threads die Methoden `wait()` und `notify()` aufgerufen werden.

Wie bereits erwähnt, wird nach Rückkehr der Methode `getAvailableSend()` die HTTP-Response konstruiert und im Falle einer gefundenen Nachricht diese in dessen Body eingebettet. Als dazu gehörender Response-Code wird `200 OK` zurückgegeben. Falls die Empfangsanfrage aufgrund einer bereits erfolgten Übertragung der passenden Nachricht abgebrochen wurde, bleibt dagegen der Body leer und der Status-Code lautet `204 No Content`.

Zurück auf dem Empfangsprozess, wo ursprünglich die Methode `irecv()` aufgerufen wurde, muss die in der HTTP-Response enthaltene Nachricht verarbeitet werden. Hierzu sind im Grunde die gleichen Abläufe wie in `enqueueSendRequest()` erforderlich. Zunächst wird überprüft, ob die Nachricht bereits fertig abgewickelt werden kann, oder ob dies vorher noch mit Nachrichten geschehen muss, die kleinere IDs aufweisen. In diesem Zusammenhang kann also der Fall eintreten, dass auch eine auf diesem Weg empfangene Nachricht zunächst zwischengespeichert wird. Der beschriebene `MatchingThread` übernimmt in diesem Fall auch hier die Aufgabe, die Nachricht zu gegebenem Zeitpunkt erneut der Abwicklung zu übergeben. Kann das eingegangene `RESTTransferObject` jedoch sofort verarbeitet werden, wird der passende `RESTReceiveRequest` aus der `recvQueue` entnommen und die in der Nachricht enthaltenen Daten direkt in dessen Empfangspuffer überführt. Damit ist die Empfangsoperation abgeschlossen.

6.3. Hypermedia

Das Prinzip *HATEOAS* (Hypermedia as the engine of Application State) ist ein wichtiger Bestandteil der REST-Philosophie. Dies wurde in Kapitel 1 beschrieben. Die Grundidee ist, dass ein Client, der einen RESTful Webservice benutzen will, lediglich die Einstiegs-URI desselben kennen muss. Durch in den Repräsentationsformaten der Ressourcen eingebettete Sprungmarken, werden Informationen über die Orte des Webservices gegeben, an denen eine Fortführung des Applikationszustands möglich ist.

Im entwickelten System wird der Hypermedia-Mechanismus dazu eingesetzt, die von den HTTP-Clients zu verwendende Pfadstruktur für den Zugriff auf Nachrichtenressourcen zu veröffentlichen. So wird es prinzipiell möglich, dass jeder Prozess eine unterschiedliche URI-Struktur für die Exponierung seiner Ressourcen verwendet. Die technische Umsetzung dieses Mechanismus basiert dabei zum einen auf der Technologie der URI-Templates und zum anderen auf dem HTTP-Header-Element `link`.

Jeder Prozess definiert einmalig die Struktur seiner Nachrichten-URIs relativ zur Wurzeladresse des Webservice in Form eines URI-Templates. Ein Beispiel hierfür wäre folgender String:

```
messages/{source}/{destination}/{context}/{tag}
```

Elemente in geschweiften Klammern sind dabei Variablen, die zur Laufzeit durch konkrete Werte ersetzt werden. Ein solcher String definiert im Interface `RESTResource` die Grundlage der Struktur aller exponierten Nachrichten-Ressourcen. Bis auf das `{tag}`-Element – dieses kann bei bestimmten Anfragen weggelassen werden und muss deshalb immer das letzte sein – sind alle Variablen untereinander in ihrer Position austauschbar.

Die Kommunikation der für einen Prozess auf diese Weise festgelegten URI-Struktur, erfolgt während des Startvorgangs über die Ressource `/init`. Die Methode `initNode()` der Klasse `RESTServer` fügt das entsprechende URI-Template im HTTP-Response dieser Ressource als `link`-Header-Element ein. Auf der anderen Seite der Netzwerkkommunikation – in diesem Fall auf dem Kontrollrechner – wird dieser `link`-Header weiterverarbeitet. Hierzu erfolgt zunächst die Extrahierung dieses aus der HTTP-Response. Anschließend werden diese nun von jedem teilnehmenden Prozess empfangenen URI-Templates als Teil derjenigen Liste, welche zuvor lediglich dafür zuständig war, die Zuordnung von Hostadressen zu UUIDs zu kommunizieren, an alle Prozesse verteilt. Dort angekommen, wird jeweils eine Hash-Map angelegt, die jedem Prozess des Systems seine selbst spezifizierte URI-Struktur zuordnet. Auf diese Weise kann bei jeder angestoßenen In-

terprozesskommunikation bei der Konstruktion der jeweiligen Anfrageadresse auf den korrekten Nachrichtenpfad zurückgegriffen werden.

7. Bewertung der Implementierung

Die grundlegende Motivation zur Entwicklung einer MPI-Implementierung auf Basis von RESTful HTTP besteht unter anderem darin, über die klassischen Einsatzgebiete von MPI hinausgehende Szenarien bedienen zu können. Allem voran lässt sich hier eine Ausführung von MPI-Programmen in internetähnlichen Rechnernetzen anführen. Sich daraus ergebende Anforderungen wurden in Kapitel 3 dargestellt. Nachdem das entwickelte System in seinen Einzelheiten in Kapitel 6 dargestellt wurde, ergibt sich die Möglichkeit, zu untersuchen, inwiefern diese Anforderungen durch REST-Konformität erfüllt werden können. Durch die möglichst korrekte Umsetzung der REST-Prinzipien in der realisierten Implementierung ist außerdem erstmalig eine Bewertung des Nutzens dieses Architekturstils im MPI-Umfeld möglich. Diese beiden Punkte finden hier Betrachtung.

7.1. Erfüllung der Anforderungen

Im Folgenden soll für jede der in Kapitel 3 formulierten Anforderungen eine Analyse vollzogen werden, inwiefern diese jeweils durch die beschriebene Implementierung auf Basis der Umsetzung der REST-Prinzipien erfüllt wird. Dazu werden die Anforderungen nacheinander erneut in ihrer Kurzform aufgeführt und anschließend kritisch betrachtet. Im Fall von nicht oder nicht vollständig erfüllten Anforderungen wird darüber hinaus versucht, Vorschläge zu machen, wie dieser Missstand im entwickelten System eventuell behoben werden könnte.

Anforderung 1. *Das System ist tolerant gegenüber Netzwerkfehlern*

Es kann bereits vorweg genommen werden, dass diese Anforderung durch die bloße Umsetzung von nach REST-Maßstäben korrektem RESTful HTTP nicht in Gänze erfüllt wird. Lediglich an zwei Stellen lassen sich Tendenzen in Richtung Netzwerkfehlertoleranz ausmachen. Die Kommunikation durch HTTP basiert naturgemäß auf TCP als Transportprotokoll. Dadurch zieht bereits implizit eine gewisse Toleranz gegenüber

Netzwerkfehlern, wie teilweisem Datenverlust, in das System ein. Aus diesem Grund kann die Anforderung zumindest für alle durch die Mechanismen von TCP abgedeckten Kategorien von Netzwerkfehlern als automatisch erfüllt angesehen werden.

Darüber hinaus existieren jedoch mögliche Fehlerkategorien, die nicht von einem Transportprotokoll behandelt werden können. Dazu gehört beispielsweise der Ausfall einer Kommunikationsrichtung oder die vollständige Unterbrechung der Netzwerkverbindung zwischen zwei Prozessen. Dadurch, dass das entwickelte System zeitgleich Push- und Pull-Kommunikation einsetzt und dadurch eine gewisse Redundanz in der Netzwerkkommunikation einführt, kann der Ausfall einer Kommunikationsrichtung kompensiert werden. Wird jedoch die Verbindung zwischen zwei Prozessen vollständig unterbrochen, bietet das System in der aktuellen Version keinen Mechanismus, der damit umgehen kann.

Ein solcher wäre aber auf der Grundlage von RESTful HTTP durchaus relativ einfach realisierbar. Unter der Voraussetzung, dass die plötzlich nicht mehr vorhandene direkte Verbindung zwischen zwei Prozessen durch die Umleitung der Nachrichten über einen dritten Prozess ersetzt werden kann, da dieser sowohl mit dem ersten, als auch mit dem zweiten Host kommunizieren kann, ist ein Indirektionsmechanismus denkbar. Ein solcher wird in Abschnitt 8.2 näher beschrieben.

Anforderung 2. *Das System ist tolerant gegenüber variierenden Latenzzeiten*

Es sind zwei Strategien zur Behandlung von variierenden Latenzzeiten unter der Voraussetzung, dass diese als durch das zugrunde liegende Verbindungsnetzwerk gegeben und von der Anwendung als nicht beeinflussbar angesehen werden, denkbar. Es wird außerdem davon ausgegangen, dass eine Nachricht in jedem Fall übertragen werden muss. Zum einen kann die Verbindung so lange aufrecht erhalten werden, bis die Kommunikation abgeschlossen ist, ohne darauf Rücksicht zu nehmen, wie lange dies dauert. Zum anderen ist es ebenso möglich, eine Verbindung nach einer gewissen festgesetzten Zeitspanne abzubrechen und danach erneut aufzubauen. Dies wird wiederholt, bis die Nachricht erfolgreich übertragen wurde.

Das entwickelte System verfolgt in der aktuellen Implementierung den ersten Ansatz. Es wird kein HTTP-Timeout festgelegt. So können Verbindungen beliebig lange offen gehalten werden. Eine Modifikation des Systems zur Verwendung der zweiten Variante ist jedoch ebenso denkbar und relativ einfach möglich. Variierende Latenzzeiten haben auf diese Weise in beiden Varianten lediglich einen Einfluss auf die Performance des Systems. Die Anforderung kann deswegen als erfüllt angesehen werden.

Anforderung 3. *Das System kann mit sich verändernden Bandbreiten umgehen*

Bereits bei der Formulierung dieser Anforderung in Kapitel 3 wurde vorweggenommen, dass diese als weniger gravierend betrachtet werden kann, da sich verändernde Bandbreiten ebenfalls lediglich auf die Ausführungszeit des parallelen Programms auswirken. Bei MPI-Programmen ist zwar eine möglichst schnelle Ausführung wünschenswert und auch Gegenstand von vielen Optimierungsanstrengungen in MPI-Implementierungen. Es kann jedoch im Allgemeinen davon ausgegangen werden, dass ein paralleles Programm im MPI-Umfeld keine harten Zeitgrenzen einhalten muss, wie sie beispielsweise bei Echtzeitanwendungen normal sind. Deswegen gilt die Anforderung als implizit erfüllt. Eine Optimierung der Bandbreitensensibilität des Systems wäre darüber hinaus aber zum Beispiel durch den Einsatz von Komprimierungsverfahren bei der Serialisierung von Nachrichten denkbar.

Anforderung 4. *Das System stellt Mechanismen bereit, die erhöhte Netzwerksicherheit ermöglichen*

Netzwerksicherheit ist kein Teil der REST-Prinzipien, da diese Richtlinien für die Softwarearchitektur eines Systems angeben. Sicherheit dagegen wird entweder auf Protokoll- oder auf Anwendungsebene realisiert und ist prinzipiell unabhängig von der Architektur. Aus diesem Grund führt die Umsetzung von REST-Konformität auch nicht automatisch zur Einführung von Netzwerksicherheit. Somit gilt die Anforderung als zunächst nicht erfüllt.

Abgesehen davon ist es jedoch durch den Einsatz von RESTful HTTP sehr einfach möglich, Netzwerksicherheit auf Anwendungsebene zu ermöglichen. Sicherheit in Netzwerken besteht im Grunde aus den zwei Bestandteilen Authentifizierung und Verschlüsselung. Im Kontext von MPI kann Authentifizierung so interpretiert werden, dass sich Prozesse, bevor sie an der Abarbeitung eines parallelen Programms teilnehmen dürfen, zunächst untereinander durch den Austausch ihrer sogenannten *Credentials* als vertrauenswürdig beurteilen müssen. RESTEasy bietet hierfür beispielsweise eine Implementierung von OAuth [36] an, die durch die Anpassung der Konfiguration entsprechend einfach eingesetzt werden kann.

Verschlüsselung kann ebenso einfach eingesetzt werden. Hier sind erneut zwei verschiedene Mechanismen denkbar. Zum einen kann ohne Einschränkungen auf das HTTPS-Protokoll umgestiegen werden. Auf diese Weise würde die komplette Kommunikation verschlüsselt. Zum anderen bietet RESTEasy außerdem das Prinzip der S/MIME Encryption [1] an, welche nur den HTTP-Body verschlüsselt und ebenfalls durch simple

Änderung der Konfiguration Einzug in das System erhält.

Anforderung 5. *Das System ist unabhängig von sich ändernder Netzwerktopologie*

Auch diese Anforderung kann auf zwei verschiedene Weisen interpretiert werden. Das dafür entscheidende Kriterium ist der Zeitpunkt, an dem sich die bestehende Netzwerktopologie ändert. Entweder geschieht dies während, oder vor der Ausführung des parallelen Programms. Für eine Berücksichtigung sich ändernder Topologie während der Laufzeit wären sehr umfangreiche Maßnahmen nötig. Denkbar wäre beispielsweise die Einführung eines Laufzeitsystems, das eine Abstraktion zwischen MPI-Prozessen und realen Rechnern bildet und erstere dynamisch verteilen kann. Das Erkennen einer Veränderung in der Netzwerktopologie wäre ebenso eine Aufgabe dieser Anwendung. Die aktuelle Version des entwickelten Systems kann keine solche Funktionalität leisten.

In Kapitel 3 wurde der Hintergrund dieser Anforderung, Deutsch's Irrtum *Topology doesn't change* und Rotem-Gal-Oz' Erklärung dieser, dargelegt. Diesen Ausführungen folgend erscheint es als naheliegender, dass damit eine Änderung der Netzwerktopologie innerhalb des Lebenszyklus einer verteilten Anwendung und nicht innerhalb einer einzelnen Ausführung derselben gemeint ist. Erwartungsgemäß ist dies darüber hinaus der einfacher umzusetzende Fall. Das entwickelte System kann mit dieser Variante sich ändernder Topologie umgehen. Dies ist jedoch nicht unmittelbar der RESTful Architektur geschuldet. Vielmehr kann jede beliebige große *general purpose* Implementierung von MPI dies leisten. Durch Angabe eines `machines` Files bei jedem Start des Systems kann die aktuelle Topologie flexibel festgelegt werden. Insofern kann auch diese Anforderung als erfüllt angesehen werden.

Anforderung 6. *Das System ist möglichst unabhängig von der Hardware- und Softwarekonfiguration der Prozessrechner und möglichst einfach installierbar*

Diese Anforderung wird ohne Einschränkung vom entwickelten System erfüllt. Da dieses vollständig in Java implementiert wurde, ist es auf jeder Plattform ausführbar, für das eine Java Virtual Machine verfügbar ist. Die Installation gestaltet sich ebenfalls sehr einfach. Es muss nur die Verzeichnisstruktur kopiert und das Startskript des JBoss Application Servers ausgeführt werden. Lediglich auf dem Kontrollrechner ist zusätzlich das Erstellen eines `machines` Files und das Ausführen von `mpirun` notwendig. An dieser Stelle muss jedoch betont werden, dass sich diese Eigenschaften nicht auf den Einsatz einer RESTful Architektur zurückführen lassen. Vielmehr ist die Wahl von Java als Programmiersprache ein Seiteneffekt der Umsetzung einer solchen.

Anforderung 7. *Das System setzt effiziente Mechanismen ein, Netzwerkkommunikation vorzubereiten*

Gemeint ist mit dieser Anforderung die Bereitstellung von netzwerkunabhängigen Mechanismen im System, die durch Vorbereitung der Kommunikation dafür sorgen, diese möglichst effizient durchführen zu können. Als Beispiel kann hier die effiziente Serialisierung der zu versendenden Nachrichten angeführt werden. Im entwickelten System wird jede Nachricht durch die Verwendung eines `ByteStreams` in ein `byte`-Array umgewandelt. Weiter fortgeschrittene Verfahren hierfür wurden aus den in Kapitel 6 genannten Gründen im Rahmen dieser Arbeit nicht implementiert. Deshalb wird diese Anforderung von der aktuellen Version des Systems nicht erfüllt. Sie ist jedoch mit relativ hohem, aber wenig komplexem, Implementierungsaufwand durchaus erfüllbar. Durch die Verbindung mit dem REST-Prinzip *Unterschiedliche Repräsentationen* könnte bei Erfüllung der Anforderung diese darüberhinaus als Errungenschaft der RESTful Architektur gewertet werden.

Anforderung 8. *Das System kann über heterogene Netze hinweg kommunizieren und nutzt die Kapazitäten heterogener Rechner effizient*

Es liegt in der Natur des auf TCP aufsetzenden HTTP-Protokolls und dessen hauptsächlichen Einsatzes im Umfeld des World Wide Web, dass dieses Kommunikation über heterogene Netze hinweg realisiert. Diese Heterogenität wird – beim Einsatz des Internetprotokoll-Stacks – insofern eingeschränkt, dass in der aktuellen Implementierung Netzwerkgeräte ansprechbar sind, die sich innerhalb eines IP-Netzes befinden. IP-Netze und das Internet als Musterbeispiel eines solchen können jedoch in sich eine sehr heterogene Struktur haben. Insofern wird diese Anforderung ebenfalls erfüllt.

Darüber hinaus wäre es durchaus denkbar, per RESTful HTTP auch über Netzwerkgrenzen hinweg zu kommunizieren. Im Prinzip kann das HTTP-Protokoll auf Basis jeder beliebigen Transportschicht implementiert werden. Eine Evaluierung dieses, beispielsweise umgesetzt auf einem proprietären Hochgeschwindigkeitsnetz, wie Infiniband, wäre interessant. Durch einen zusätzlichen Mechanismus, der Nachrichten vom einen in das andere Netz weiterleiten kann, könnten beide verbunden und für die gemeinsame Abarbeitung eines parallelen Programms mit RESTful HTTP eingesetzt werden. Das in 4.3 dargestellte MPJ/IBIS realisiert mithilfe seines *Smartsocket*-Mechanismus [47] ein solches Prinzip.

Eine große Herausforderung stellt der zweite Teil dieser Anforderung dar: Die Lastbalancierung für Rechner mit verschiedenen potentiellen Ausführungsgeschwindigkeiten.

Um dies zu realisieren, benötigt das System Informationen über die verfügbaren Rechenkapazitäten. Das in 4.4 vorgestellte HeteroMPI geht sogar einen Schritt weiter und akquiriert zusätzlich Details zur algorithmischen Struktur des auszuführenden Programms.

Ein relativ einfach umzusetzender Mechanismus wäre die simple manuelle Angabe der Anzahl verfügbarer Prozessorkerne und damit parallel ausführbarer Tasks eines Rechners. Diese Informationen könnte dazu genutzt werden, beispielsweise einem teilnehmenden Rechner entsprechend mehrere logische Prozesse zuzuordnen und damit die Ressourcen effizienter zu nutzen. Keine der beschriebenen Mechanismen können jedoch durch bloßen Einsatz von RESTful HTTP herbeigeführt werden und sind deswegen im entwickelten System nicht vorhanden. Der zweite Teil der Anforderung gilt somit als noch nicht erfüllt.

7.2. Relevanz von REST für MPI

Es wurde gezeigt, dass einige der postulierten Anforderungen allein durch den konsequenten Einsatz von RESTful HTTP im entwickelten System erfüllt werden können. Andere dagegen erfordern weit darüber hinausgehende Konstrukte, welche in der im Zuge dieser Arbeit vorgestellten Implementierung nicht realisiert werden können.

Auffallend ist, dass es an vielen Stellen so scheint, als würden die offensichtlichen Vorteile der eingesetzten Kommunikationsarchitektur in einem internetähnlichen Umfeld allein auf den Einsatz von HTTP als Protokoll und dessen Eigenschaften zurückzuführen sein. Es fällt schwer, zu erkennen, welchen zusätzlichen Nutzen es bringt, dass diese Strukturen zusätzlich das Prädikat *RESTful* tragen. Dies ist jedoch nicht die Denkweise, die der grundlegenden Intention von REST genügt. Folgt man den Ausführungen Fieldings [26], dem Erfinder, oder vielmehr *Entdecker* des Architekturstils REST, darf dieses nicht als Zusatz zu oder Spezialform der konkreten Technologie, in der es seine Ausprägung findet, gesehen werden. Vielmehr ist REST nach Fielding die *richtige* Art, diese Technologie – hier HTTP – in einem Umfeld mit ähnlichen Anforderungen, wie dem World Wide Web, einzusetzen.

Insofern ist es, wenn man Fieldings Schlussfolgerungen zustimmt, geradezu notwendig, eine verteilte Anwendung, die sich im Internet bewegt und die Grundlagen seines Erfolgs für sich nutzen möchte, nach den REST-Prinzipien zu gestalten. Der Message Passing Interface Standard beschreibt ein System, das von sich aus wenig mit klassischen verteilten Anwendungen aus dem Internetumfeld, wie Webservices, gemeinsam hat. Die vorliegende Arbeit unternimmt den Versuch, dieses System trotz dessen im gleichen Umfeld

einzusetzen und seine Implementierung möglichst nach den dort offensichtlich erfolgreichen Strukturen, wie einer RESTful Architektur, zu gestalten. Da allein dadurch bereits einige der allgemein für verteilte Anwendungen in internetähnlichen Szenarien gültigen Anforderungen erfüllt werden, kann die Relevanz von REST für ein MPI-System in diesem Umfeld im Vergleich zu eventuell möglichen anderen Ansätzen als hoch angesetzt werden.

Dass andere Anforderungen nicht sogleich erfüllt werden, ist der einer MPI-Implementierung zwangsläufig innewohnenden Struktur geschuldet, die in ihrer ursprünglichen Ausrichtung offensichtlich nicht für einen solchen Einsatz konzipiert ist. Daraus ergeben sich spezielle Anforderungen, die durch bloßes Verwenden eines Protokolls, wie HTTP, in einer bestimmten Form, wie REST, nicht abgedeckt werden können. Betont sei darüber hinaus, dass die jeweils vorgeschlagenen Verbesserungen des Systems, dieses keineswegs weniger konform zu den REST-Prinzipien machen würde. Die Umsetzung dieser würde lediglich zusätzlichen Implementierungsaufwand bedeuten.

8. Optimierungen und Erweiterungen

Über die reine Herstellung von REST-Konformität durch Umsetzung der REST-Prinzipien, wie sie in Kapitel 6 beschrieben wird, hinaus, bietet die dadurch eingeführte Kommunikationsarchitektur zahlreiche Ansatzpunkte für mögliche Optimierungen und Erweiterungen des Systems. Im Folgenden werden zwei dieser vielfältigen Möglichkeiten betrachtet. Zunächst wird der Fokus auf Kommunikationsoperationen gelegt, die auf einer abstrakteren Ebene, als die bisher behandelten grundlegenden Sende- und Empfangsfunktionen angesiedelt sind: Die kollektiven Kommunikationsoperationen. Im einzelnen wird das Optimierungspotential der *Broadcast*-Operation exemplarisch dargelegt.

Der zweite Teil dieses Kapitels beschäftigt sich mit einer Problematik, die sich bei der Verwendung von verteilten Systemen vor allem im Umfeld des World Wide Web zeigt. Aufgrund ansonsten vorliegender hoher Sicherheitsrisiken gehört die Verwendung von Firewalls zum Schutz von einzelnen Rechnern oder Netzwerksegmenten vor böswilliger Netzwerkkommunikation zu einer sehr weit verbreiteten Praxis. Auch beim Umgang mit solchen eröffnet die umgesetzte RESTful Architektur einen gewissen Spielraum.

Es sei erwähnt, dass die hier vorgestellten Optimierungs- und Erweiterungsansätze lediglich konzeptioneller Natur sind. In der im Kontext dieser Arbeit vorgestellten Implementierung wurden diese noch nicht umgesetzt. Sich eventuell bei der Realisierung dieser Konzepte ergebende Probleme können deswegen, da sie aufgrund der Komplexität des Systems nicht vollständig vorhersagbar sind, nicht berücksichtigt werden.

8.1. Kollektive Kommunikationsoperationen

In den verbreiteten MPI-Implementierung sind bereits vielfältige Optimierungen für kollektive Kommunikationsoperationen enthalten. Der Grundansatz der meisten dieser ist die Anwendung eines passenden Explorationsalgorithmus aus der Graphentheorie zur Bestimmung einer optimalen Reihenfolge und Struktur an Sende- und Empfangsoperationen für eine gegebene kollektive Operation. So wird beispielsweise zur Realisierung einer Broadcast-Operation oft eine baumartige Nachrichtenpropagierung gewählt. In [52]

findet sich eine Evaluierung verschiedener solcher Herangehensweisen.

Auf Basis der realisierten Kommunikation mit RESTful HTTP ist jedoch ein völlig anderer Ansatz denkbar. Statt aktiv die Nachrichtenreihenfolge und Propagierungstopologie zu manipulieren, kann die Exponierung von REST-Ressourcen bei kollektiven Operationen so gestaltet werden, dass eventuell vorhandene Infrastruktur, wie Caches, automatisch optimal – und damit passiv – eingesetzt werden. Dies soll nun am Beispiel der Broadcast-Operation unter Verwendung von transparentem Caching näher erläutert werden.

Die Broadcast-Operation kann bei MPI dazu genutzt werden, den Inhalt eines Puffers von einem Prozess ausgehend an alle anderen Prozesse der gleichen Gruppe zu verteilen. Nachher liegt bei den empfangenden Prozessen jeweils eine Kopie des Puffers vor. Die entscheidende Erkenntnis an dieser Stelle ist die Tatsache, dass offensichtlich dieselbe Nachricht mehrfach – zu verschiedenen Zielen – verschickt wird.

Transparentes Caching beschreibt eine Technik, die an bestimmten Punkten einer Netzwerktopologie Kommunikationsdaten ohne Einfluss Kommunikationsteilnehmer zwischenspeichert und weiterverteilt. Wird beispielsweise eine Ressource eines entfernten Servers von einem Client angefragt und diese ist bereits auf einem Cache gespeichert, der sich topologisch zwischen Client und Server befindet, antwortet dieser Cache stellvertretend für den Server. So wird die restliche Netzwerkstrecke vom Cache zum Server bei der Kommunikation eingespart.

Prinzipiell können Ressourcen zwischengespeichert werden, die sich über mehrere Anfragen hinweg nicht verändern. Nach den REST-Prinzipien konzipierte HTTP-Kommunikation ist aufgrund dessen, dass sie Zusicherungen der HTTP-Verben, wie beispielsweise Idempotenz bei `GET`, sicherstellt, cachebar [26]. Schon allein diese Tatsache hat auch bei ordinärer Punkt-zu-Punkt-Kommunikation das Potential, die Performance in Netzwerken, die eine solche Infrastruktur enthalten, zu steigern. Im Kontext von kollektiven Operationen, wie hier bei Broadcast, sind jedoch zur effizienten Nutzung dieser Strukturen, eine Anpassung des Programmcodes notwendig.

In der aktuellen Version des Systems wird die Broadcast-Operation in der Klasse `mpi.Intracomm` implementiert. Diese bildet die Semantik auf eine Reihe von Sende- und Empfangsoperationen der zugrundeliegenden Device-Implementierung ab. Zur Abwicklung wird zwar, wie oben beschrieben, eine Baumstruktur verwendet, bei der Verwendung der `RESTDevice` werden jedoch die letztendlichen Kommunikationsoperationen auf mehrere unterschiedliche Ressourcen auf mehreren Prozessen abgebildet. Die Möglichkeit, die eine, ursprüngliche Nachricht, die vom Wurzelprozess verteilt werden soll, zu cachen

und damit den Empfangsprozessen effizienter zur Verfügung zu stellen, entfällt so.

Aus diesem Grund liegt es nahe, die Implementierung der Broadcast-Operation auf dieser Ebene anzupassen. Ein Vorschlag hierzu wäre die Einführung einer neuen Ressource, welche die zu verteilende Nachricht eindeutig identifiziert. Folgende Struktur wäre denkbar:

```
/broadcast/{communicator}/{root}/{id}
```

Die Identifikation der Nachricht erfolgt über die Kombination der Merkmale *Kommunikator*, *Wurzelprozess* und *Broadcast-ID*. Letztere wird eingeführt zur Unterscheidung mehrerer aufeinander folgender Broadcasts eines Wurzelprozesses innerhalb desselben Kommunikators. Die Ressource ist auf dem Wurzelprozess per `GET` und auf den Empfangsprozessen per `PUT` ansprechbar. Die Abwicklung solcher Anfragen muss entsprechend in der Device-Implementierung umgesetzt werden. Für anderen kollektive Operationen sind ähnliche zusätzliche Ressourcen denkbar.

Ein denkbares Szenario, in dem eine solche Umsetzung Vorteile versprechen würde, wäre der Fall, in dem der Wurzelprozess eines Broadcasts eine sehr langsame Netzwerkverbindung aufweist. Darüber hinaus befindet sich jedoch zwischen Wurzelprozess und allen Empfangsprozessen ein transparenter Cache. Die Empfangsprozesse sind besser mit dem Cache verbunden. Hier würde beim ersten `GET` eines Empfangsprozesses, oder – je nachdem, was zuerst auftritt – beim ersten `PUT` des Wurzelprozesses während der Operation die Nachricht auf dem Cache gespeichert. Kommen nun weitere `GET`s für die Ressource auf dem Wurzelprozess beim Cache an, kann letzterer sie direkt bedienen und es muss nicht mehr auf die langsame Netzwerkverbindung zurückgegriffen werden. Durch die Einführung eindeutiger HTTP-Ressourcen für jede angestoßene Broadcast-Operation wird darüber hinaus die Praxis der Cache-Invalidierung nach einer gewissen Zeit überflüssig: Die Ressourcen sind unbegrenzt gültig.

8.2. Firewall-Problematik

Ein Problem, welchem sich jede verteilte Anwendung im Internet-Umfeld stellen muss, ist die Existenz von vielfältig konfigurierten Firewalls an unvorhersehbaren Stellen der Netzwerktopologie. Durch ausschließliche Verwendung von HTTP zur Kommunikation, hat die hier vorgestellte Implementierung bereits einen großen Vorteil gegenüber anderen verteilten Systemen. HTTP-Kommunikation wird im Allgemeinen als weniger gefährlich

und vor allem kontrollierbarer als andere Protokolle angesehen und wird deswegen weniger eingeschränkt.

Es kann davon ausgegangen werden, dass auf den meisten Rechnern, die an das World Wide Web angebunden sind, zumindest ausgehende HTTP-Verbindungen nicht blockiert werden. Diese sind zum Betrieb eines Webbrowsers und zur Nutzung vieler populärer Dienste notwendig. Anders gestaltet sich die Situation mit eingehenden Verbindungen. Da im entwickelten System jeder Prozess sowohl HTTP-Client, als auch HTTP-Server ist, müssen prinzipiell auch solche Verbindungen möglich sein.

Es gibt jedoch einen Spezialfall, in dem die vorgestellte Implementierung – zumindest nach der Startphase, ab der Ausführung des parallelen Programms – zum Teil automatisch mit blockierter eingehender HTTP-Kommunikation umgehen kann. Solange jeder Prozess, bei dem diese Situation vorliegt, im Verlauf des parallelen Programms nur mit anderen Prozessen kommuniziert, die selbst eingehende Verbindungen erlauben und somit als Server fungieren können, kann jede Art der MPI-Kommunikation weiterhin vollführt werden. Solche Prozesse können selbst nicht Server sein, können also keinen Webservice bereitstellen. An sie gestellte HTTP-Requests werden nicht beantwortet. Da jedoch für Senden und Empfangen jeweils zwei verschiedene Kommunikationsrichtungen – Push- und Pull-Kommunikation – implementiert wurden, kann nach wie vor die Funktionalität als nun reiner Client erhalten werden. Ein solcher Prozess sendet per Push und empfängt per Pull, wird also stets selbst aktiv. Die einzige Funktionalität, die ein reiner Client-Prozess in der aktuellen Implementierung nicht abbilden kann, ist das Abschließen einer synchronen Sendeoperation. Die Benachrichtigung über eine angestoßene Empfangsoperation auf der Gegenseite kann momentan nur per `PUT`-Request auf eine Ressource des Sendeprozesses erfolgen. Es wären jedoch zum Umgehen dieser Problematik andere Strategien denkbar. Um auch die Startphase mit solchen Prozessen durchführen zu können, müsste diese grundlegend verändert werden. Auch hierfür sind jedoch Implementierungsvarianten durchaus vorstellbar.

Darüberhinaus kann aber auch hier unter Ausnutzung der besonderen durch RESTful HTTP eingeführten Gegebenheiten und Möglichkeiten ein anderer Ansatz zum Umgang mit vorhandenen Firewalls formuliert werden: Indirektion. Damit ist die Umleitung von Nachrichten über andere Prozesse gemeint, bevor sie den Empfangsprozess erreichen. Auf diese Weise könnte mit dem Fall umgegangen werden, dass lediglich die direkte Verbindung zwischen zwei Prozessen nicht möglich, aber eine indirekte über einen dritten verfügbar ist.

Die vorgestellte Ressourcenstruktur kann für einen solche Implementierung direkt ge-

nutzt werden. Es muss lediglich die Implementierung der Umleitungslogik erfolgen. Ein denkbarer Ansatz ist der folgende: Dadurch, dass die URI einer Ressource alle Merkmale enthält, Nachrichten zu identifizieren, können diese ohne weitere Informationen gezielt weitergeleitet werden. Erhält nun ein Prozess eine Nachricht auf einer Ressource, deren Ziel-ID nicht die eigene ist, kann dieser selbst die den zugehörigen Ziel-Host bestimmen und die Nachricht weitersenden. Ähnlich funktioniert dies mit einer Empfangsanfrage: Erhält ein Prozess ein `GET` auf einer Ressource, dessen Quellen-ID nicht seine eigene ist, fragt dieser selbst beim richtigen Sendeprozess an und gibt die danach eventuell empfangene Nachricht an den ursprünglichen Prozess weiter.

9. Benchmarks und Messergebnisse

Dieses Kapitel dient der Präsentation von Benchmark-Ergebnissen unter Einsatz des entwickelten Systems. Die im Folgenden dargestellten Werte stammen aus der im Downloadpaket von MPJ Express enthaltenen Benchmark-Suite. Diese implementiert einen Teil der *The Java Grande Forum MPJ Benchmarks* [20]. Zur Beurteilung eventueller Fortschritte der überarbeiteten Implementierung sind die Ergebnisse der gleichen Messroutinen bei Benutzung der Version des Systems aus dem Master-Projekt aufgeführt. Als drittes MPI-System wird zur Referenz das in 4.3 dargestellte MPJ/IBIS eingesetzt. Zu diesem Zweck wurde die Benchmark-Suite, welche in der ausgelieferten Variante die ebenfalls in MPJ Express implementierte mpjJava API [17] einsetzt, umgeschrieben und an die Konventionen der MPJ API [18] – diese wird von MPJ/IBIS implementiert – angepasst.

Die hier aufgeführten Benchmarks testen die grundlegenden kollektiven Kommunikationsoperationen von MPI. Hierzu wird gemessen, wie viele Daten in einer vorgegebenen Zeitspanne durch die Operation übertragen werden können. Die Funktionen werden jeweils mit zwei Formen von Eingabedaten bedient: Primitive `double`-Werte und komplexere MPI-Objekte, welche jedoch wiederum nur `double`-Werte kapseln. Als physikalisches Verbindungsnetzwerk wird ein Fast-Ethernet verwendet. Das Testsystem besteht aus folgenden Rechnern:

- AMD Phenom II, 4x3,2GHz, 8GB RAM, Ubuntu Linux 12.04 64bit
- Intel Core2Duo, 2x2,0GHz, 4GB RAM, Ubuntu Linux 12.04 64bit
- Intel Atom 330, 2x1,6Ghz, 2GB RAM, Ubuntu Linux 12.04 32bit

Betrachtet man die Abbildungen 9.1 bis 9.9, lässt sich zunächst eine offensichtliche Überlegenheit von MPJ/IBIS ausmachen. Dies entspricht jedoch den Erwartungen, da dieses System zur Erreichung einer möglichst hohen Performance sehr ausgefeilte Techniken und eigens auf heterogene Anforderungen abgestimmte auf TCP aufsetzende Protokolle einsetzt. Darüber hinaus liegt dieser durchaus beachtlichen Leistung ein weitaus

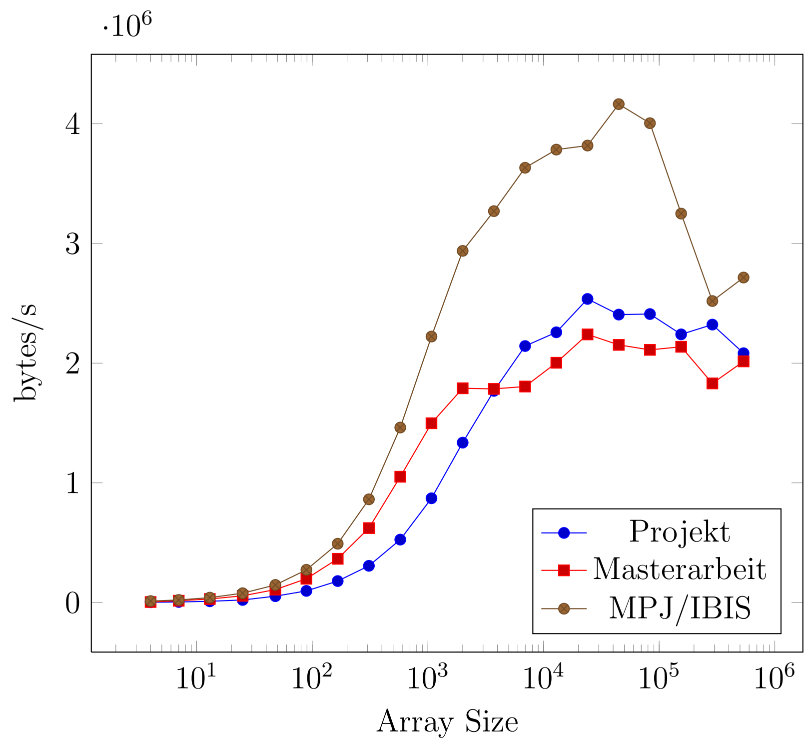


Abbildung 9.1.: All-to-all Double

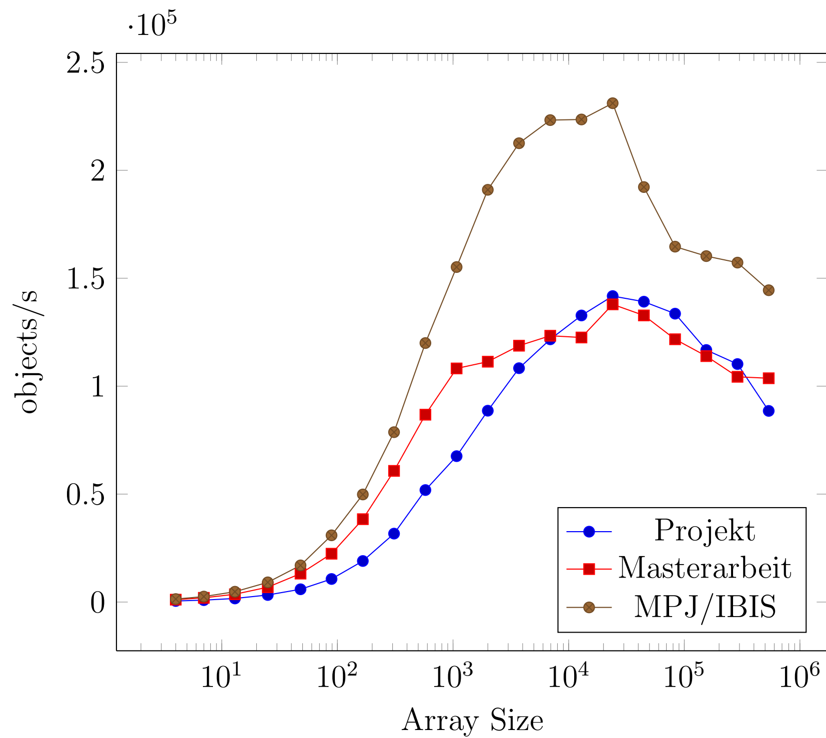


Abbildung 9.2.: All-to-all Object

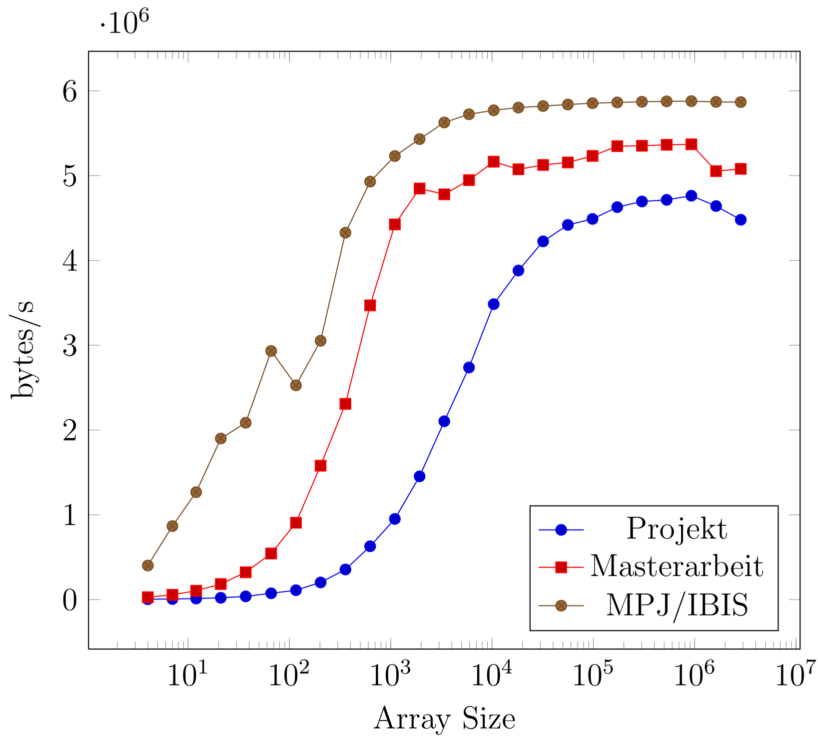


Abbildung 9.3.: Broadcast Double

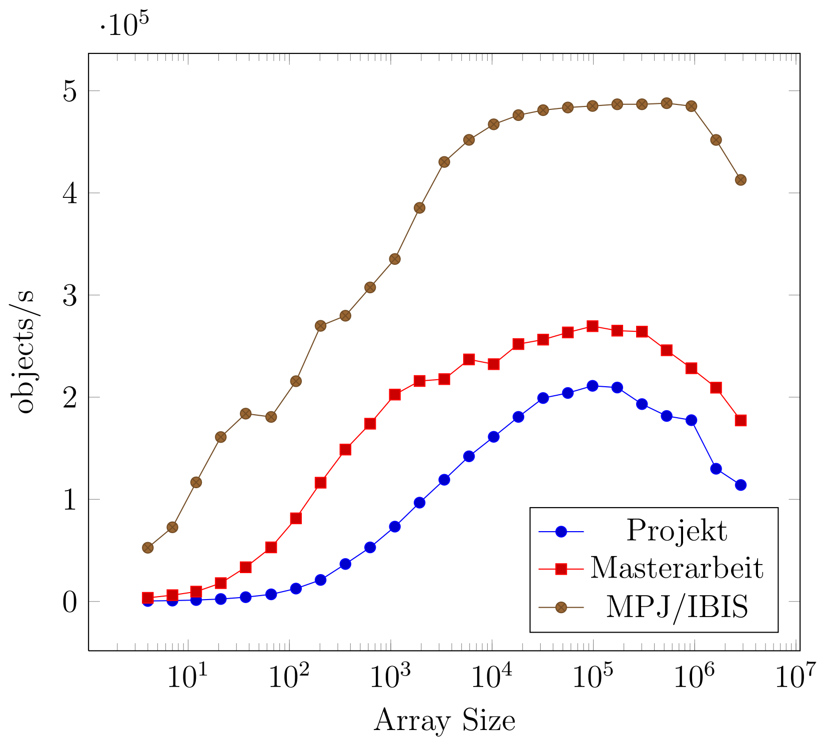


Abbildung 9.4.: Broadcast Object

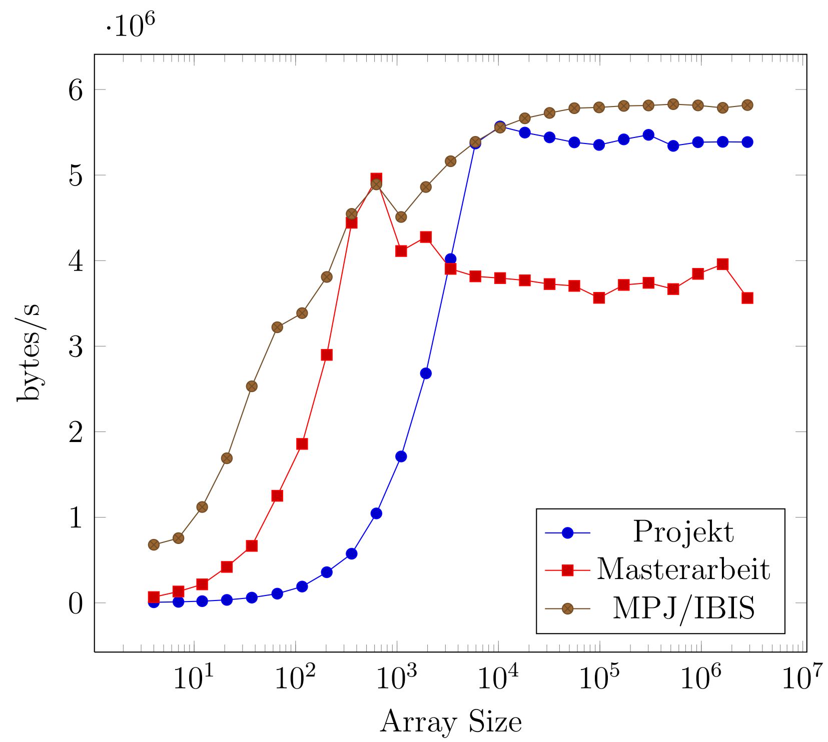


Abbildung 9.5.: Gather Double

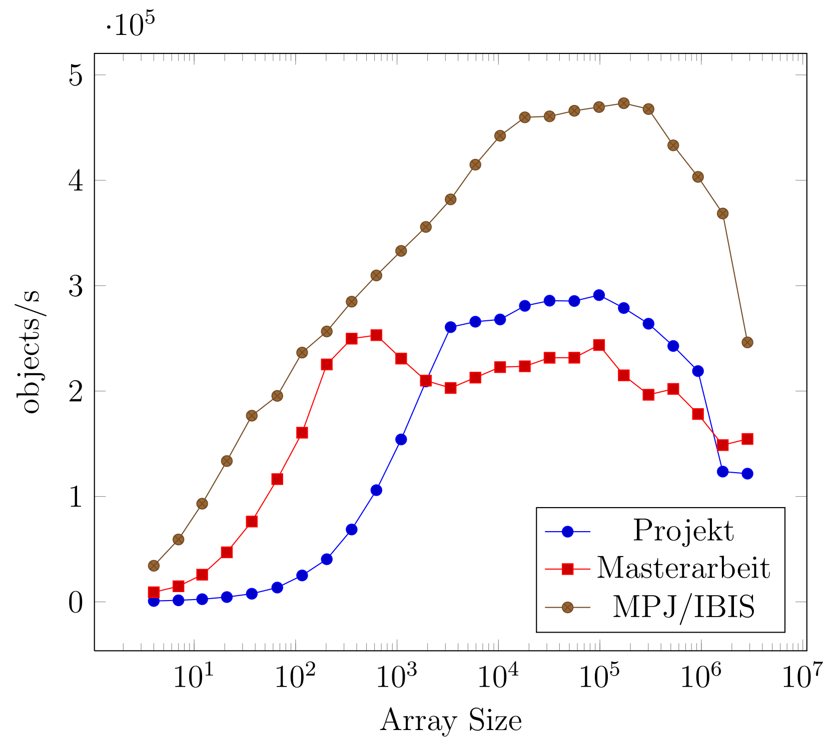


Abbildung 9.6.: Gather Object

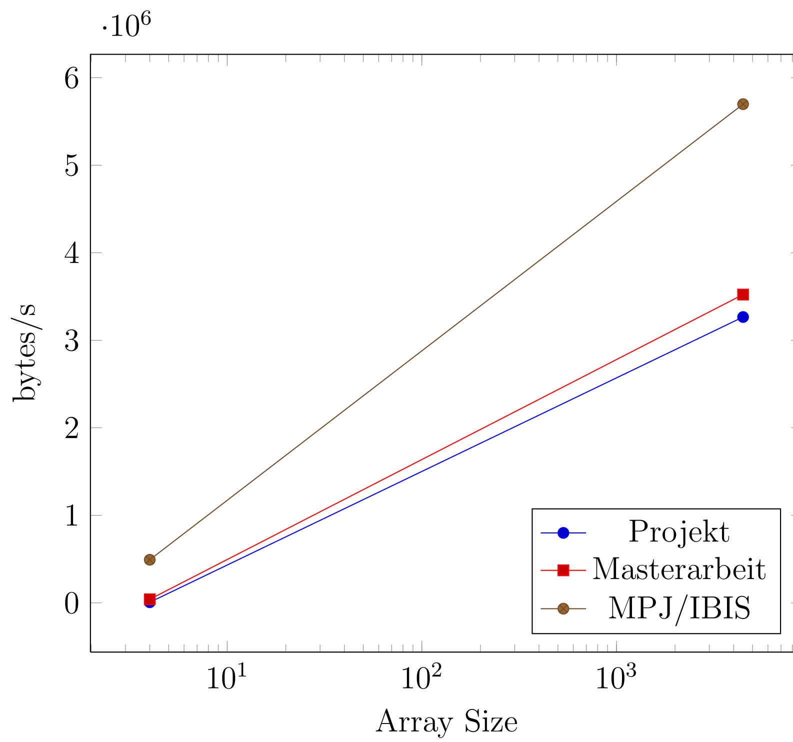


Abbildung 9.7.: Scatter Double

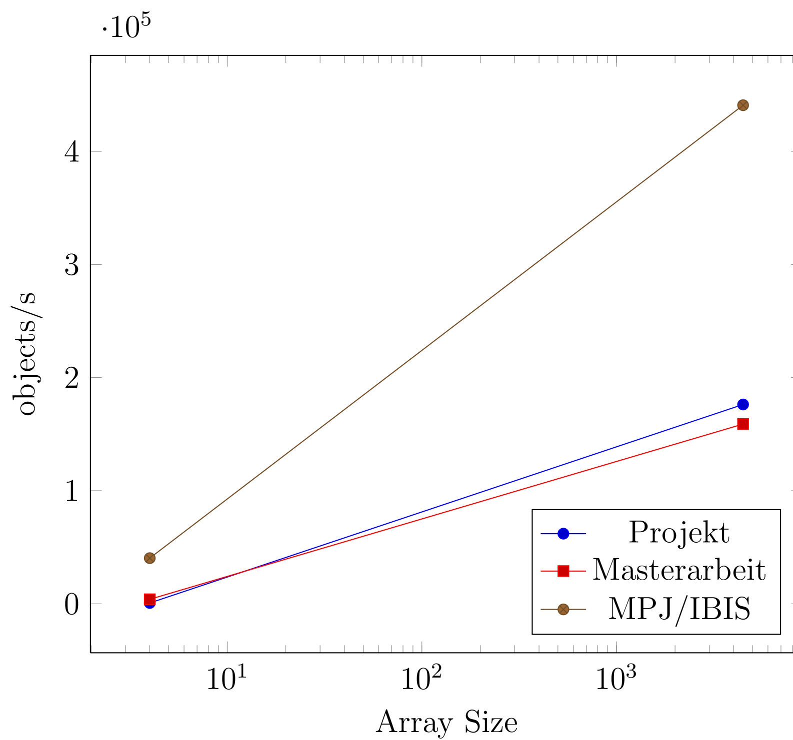


Abbildung 9.8.: Scatter Object

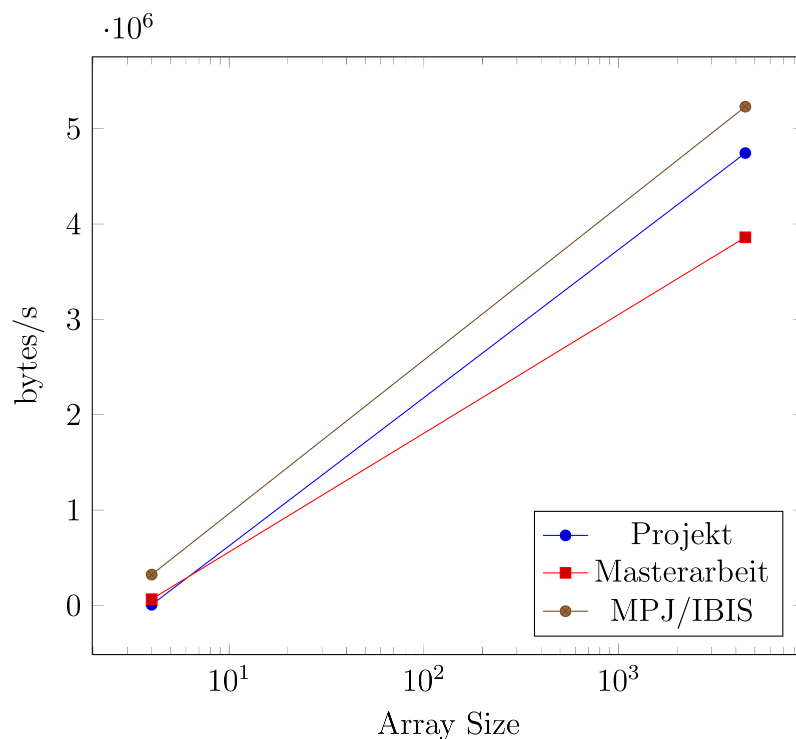


Abbildung 9.9.: Reduce Double

höherer Entwicklungsaufwand zugrunde. In diesem Kontext ist jedoch ebenfalls erstaunlich, dass zumindest die Implementierung aus dieser Masterarbeit an einigen wenigen Stellen fast an die Performance von MPJ/IBIS heran reicht.

Interessanter gestaltet sich der Vergleich der beiden Versionen des entwickelten Systems. Wie sich bereits in den Ergebnissen des Master-Projekts, welche in Kapitel 2 dargestellt werden, zeigt, macht sich der durch den Einsatz von HTTP eingeführte Overhead in der Kommunikation desto mehr bemerkbar, je kleiner die enthaltene Datenmenge der verschickten Nachrichten ist. Das Anfrage-Antwort-Schema von HTTP benötigt wesentlich mehr, nur für das Protokoll notwendige, Datenübertragung, als dies beispielsweise bei der Verwendung von reinen TCP-Sockets der Fall ist. Dieser Overhead bleibt jedoch für sich verändernde Nachrichtengrößen konstant. Aus diesem Grund ist das Verhältnis von Overhead zu Nachrichtengröße bei kleinen Nachrichten am größten. In diesem Zusammenhang auffallend ist, dass sich gerade in dem angesprochenen Bereich der kleinen Nachrichten die Performance der neuen Implementierung des Systems von derjenigen des Master-Projekts bei jeder getesteten Funktion stark abhebt. Daraus lässt sich ableiten, dass bereits die bloße Umsetzung von möglichst hoher REST-Konformität durchaus einen Vorteil beim Einsatz von HTTP als Kommunikationsprotokoll bringt. Die höhere

Performance bei kleinen Nachrichten lässt sich am stärksten auf die Einführung der zusätzlichen Pull-Kommunikation zurückführen. Durch die Nutzung des Zeitintervalls zwischen angestoßener Empfangsoperation und für das Senden fertig vorbereiteter Nachrichten für einen vorzeitigen HTTP-Verbindungsaufbau kann offensichtlich das Gewicht des HTTP-Overheads auf die Gesamtperformance in der Tat stark reduziert werden. Diese Erkenntnis lässt sich als Erfolg der vorgenommenen Implementierung werten.

Ein etwas anderes Bild zeigt sich jedoch bei den Operationen *All-to-all* (Abbildungen 9.1 und 9.2) und *Gather* (Abbildungen 9.5 und 9.6) ab ungefähr der Hälfte der logarithmiert dargestellten Abszisse, welche die Nachrichtengröße beschreibt. Ab dort fällt die neue Implementierung in der Performance hinter diejenige des Master-Projekts zurück. Dieser Effekt lässt sich nicht auf die nun vorhandene REST-Konformität zurückführen, da keine der dafür eingesetzten Maßnahmen in der Theorie bei steigender Nachrichtengröße einen negativen Einfluss auf die Performance haben dürfte. Dass der Effekt beispielsweise bei *Broadcast* (Abbildungen 9.3 und 9.4) überhaupt nicht auftritt, schränkt den Spekulationsraum über die möglichen Ursachen weiter ein. Leider kann an dieser Stelle keine endgültige Aussage darüber getroffen werden, was den Grund dieses Effektes darstellt. Jedoch ist der Ausschluss der umgesetzten REST-Prinzipien als Ursache durchaus möglich. Es wird vermutet, dass ein Implementierungsfehler in der Verwendung der vielfältigen und komplexen für die Verwaltung der zusätzlichen Kommunikationsrichtung notwendigen Datenstrukturen vorliegt. Es ist Gegenstand zukünftiger Arbeiten am entwickelten System, diesen zu beheben.

10. Anwendungsbeispiel: ODE-Verfahren

Kapitel 9 präsentiert die Ergebnisse von Performance-Messungen des entwickelten Systems bei der Ausführung von synthetischen Benchmarks. In Kontrast dazu wird im Folgenden ein reales Problem aus dem Bereich des wissenschaftlichen Rechnens betrachtet und die darin erzielte Leistung der Implementierung überprüft. Es erfolgt darüber hinaus ein Vergleich zur Ausführung des gleichen parallelen Programms auf Basis eines MPICH2-Systems einerseits und MPJ/IBIS andererseits. Intention dieses Kapitels ist zum einen der Nachweis, dass das entwickelte System ohne weiteres für reale Probleme und zur Ausführung bestehender MPI-Programme verwendet werden kann. Dadurch erhält es bereits in dieser kaum für den Produktiveinsatz optimierten Version eine gewisse Praxisrelevanz. Zum anderen soll erstmalig ein Vergleich der Performance bei der Ausführung desselben Programms, einerseits auf dem hier entwickelten System und andererseits auf einer etablierten und weit verbreiteten MPI-Implementierung, durchgeführt werden. Die Messungen mit MPJ/IBIS dienen der Refernez. Dadurch kann der Overhead eingeschätzt werden, der durch den Einsatz von Java und RESTful HTTP aus reiner Performance-Blickrichtung eingeführt wird. Dass jedoch die reine Geschwindigkeit des Systems nur am Rande eine Rolle für die Intentionen der vorliegenden Arbeit spielt, muss an dieser Stelle nicht erneut tiefer ausgeführt werden. Der Leser sei trotzdem darauf hingewiesen.

Das betrachtete Problem aus dem Bereich des wissenschaftlichen Rechnens ist die Lösung von Systemen gewöhnlicher Differentialgleichungen (*Ordinary Differantial Equations – ODE*) mithilfe eines eingebetteten Runge-Kutta-Verfahrens. Die verwendete Implementierung wird in [43] dargestellt, wo die Beobachtung gemacht wird, dass ODE-Systeme mit einer bestimmten Struktur, mit einer besonders speichereffizienten Variante eines solchen Verfahrens gelöst werden können. Der hier eingesetzte Quellcode realisiert die Basisimplementierung (Algorithmus D) eines eingebetteten Runge-Kutta-Verfahrens, auf welcher die speichereffizienten Varianten aus [43] basieren. Die Darstel-

lung der Implementierung ist nicht Gegenstand der vorliegenden Arbeit.

Das MPI-Programm, welches das beschriebene eingebettete Runge-Kutta-Verfahren realisiert, wurde vom Lehrstuhl für Angewandte Informatik II (Parallele und verteilte Systeme) der Universität Bayreuth zur Verfügung gestellt. Als Testproblem kommt eine 2D-Brusselator-Gleichung der Größe 500000[1] zum Einsatz. Die in C umgesetzte Implementierung verwendet zur Kommunikation auf Rechnern mit verteilten Adressräumen standardkonforme MPI-Funktionen und kann somit zusammen mit jedem beliebigen MPI-System eingesetzt werden, das den MPI-Standard [28] implementiert. Für die hier durchgeführten Vergleichsmessungen wurde MPICH2 in der Version 1.4.1 eingesetzt. Das bereitgestellte Programm konnte somit ohne Veränderung – auf diesem System – verwendet werden.

Zum Einsatz zusammen mit der in dieser Arbeit vorgestellten Implementierung und mit MPJ/IBIS musste jedoch das parallele Programm zum einen auf Java übersetzt und zum anderen die enthaltenen MPI-Funktionen durch ihre jeweiligen Äquivalente aus der mpiJava- [17] und MPJ-API [18] ersetzt werden. Als Fazit dieser Übersetzungsarbeit kann die Erkenntnis formuliert werden, dass eine solche sehr einfach durchführbar ist. Die Änderung der MPI-Funktionen beinhaltet hauptsächlich die Nutzung geringfügig anderer Funktionsnamen. Insofern stellt die Übersetzung des restlichen Programmcodes von C in Java die größere Herausforderung dar.

Das Testsystem besteht aus den folgenden beiden Rechnern, welche per Fast-Ethernet verbunden sind:

- AMD Phenom II, 4x3,2GHz, 8GB RAM, Ubuntu Linux 12.04 64bit
- Intel Core2Duo, 2x2,0GHz, 4GB RAM, Ubuntu Linux 12.04 64bit

	RESTful HTTP	**MPICH2**	**MPJ/IBIS**
mittlere Kernelzeit	6329	1939	1721
Standardabweichung	4521	65	49

Tabelle 10.1.: Vergleich der mittleren Kernelzeiten des eingebetteten Runge-Kutta-Verfahrens zwischen dem entwickelten System, MPICH2 und MPJ/IBIS in Millisekunden bei 78 Messungen.

Die Resultate der jeweiligen Messungen der durchschnittlichen Kernelzeit – die Zeit einer Ausführung der Schleife über die Zeitschritte des Lösungsverfahrens – sind in Tabelle 10.1 dargestellt. Dieser liegt die Messung von jeweils 78 Schleifenausführungen zugrunde. Wie bereits zu erwarten, liefert die MPICH2-Implementierung wesentlich bessere

[1] Anzahl der Gleichungen.

Ergebnisse als die RESTful HTTP Implementierung. Die durchschnittliche Kernelzeit auf Basis des entwickelten Systems ist mehr als doppelt so lang als bei der C-Variante. Der durch HTTP-Kommunikation eingeführte Overhead lässt sich auf Basis einer solchen Vergleichsplattform, auf der sich aufgrund ihrer Binär-Homogenität auch ein System wie MPICH2 einsetzen lässt, durchaus als signifikant bezeichnen. Erstaunlicher ist jedoch, dass die Performance von MPICH2 durch diejenige von MPJ/IBIS noch übertroffen wird. Dies lässt sich dadurch erklären, dass MPJ/IBIS zusätzlich zu seinem effizienten Umgang mit heterogenen Netzwerkstrukturen eine Lastbalancierung einsetzt und damit die unterschiedliche Rechenleistung der beiden Testrechner besser nutzen kann.

Ein interessantes Phänomen zeigt sich bei der Betrachtung der Standardabweichung der Kernelzeit. Diese ist in der C-Version und bei MPJ/IBIS sehr gering, die Ausführungszeiten der einzelnen Schleifendurchläufe sind nahezu identisch. Ein völlig anderes Bild zeigt sich in der RESTful HTTP-Implementierung. Hier variieren die Kernelzeiten von 3321 bis 25070 Millisekunden. Dies ist darauf zurückzuführen, dass durch Einführung einer zweiten asynchronen Kommunikationsrichtung die Übertragung und Verarbeitung der Nachrichten nicht mehr völlig deterministisch abläuft. Offensichtlich gibt es bei bestimmten Übertragungsreihenfolgen oder Anfragekonstellationen Effekte, die die Verarbeitung der Nachrichten verzögern. Hier ist ein großes Optimierungspotential für zukünftige Versionen des Systems erkennbar.

Zusammenfassend lässt sich festhalten, dass die Möglichkeit der Nutzung des entwickelten Systems für reale Probleme des wissenschaftlichen Rechnens praktisch gezeigt wurde. Die notwendige Anpassung vorhandener MPI-Programme ist geringfügig und ihre hauptsächliche Schwierigkeit liegt in der Portierung nach Java. Die erreichbare Performance beim Einsatz der realisierten Implementierung reicht auf der relativ homogenen Testplattform bei weitem nicht an diejenige heran, die Systeme wie MPICH2 bereitstellen, die für solche Umgebungen optimiert sind. Hier zeigt sich deutlich, dass aus der Perspektive der reinen Ausführungsgeschwindigkeit hochoptimierte MPI-Implementierungen auf Plattformen, auf denen sie technisch eingesetzt werden können, eindeutig zu bevorzugen sind. Das entwickelte System ist hierfür keine Alternative. Vielmehr stellt es, wie MPJ/IBIS, eine Möglichkeit dar, MPI in Umfeldern zu nutzen, in denen MPI-Systeme, wie MPICH2, nicht verwendet werden können und deswegen auch nicht zu einem Leistungsvergleich herangezogen werden können. Dass das ähnlich flexible MPJ/IBIS auch auf homogener Hardware sehr respektable Leistungen erzielt, ist darauf zurückzuführen, dass dieses im Gegensatz zum entwickelten System auch sehr komplexe Optimierungen für solche Umgebungen beinhaltet.

Fazit

Es wurde eine MPI-Implementierung vorgestellt, die in reinem Java umgesetzt ist und zur Kommunikation RESTful HTTP einsetzt. Zu diesem Zweck erfolgte eine theoretische Analyse des Architekturstils REST auf Basis des HTTP-Protokolls. Die fünf REST-Prinzipien wurden betrachtet und auf das zu entwickelnde System übertragen. Ein Rückblick auf die verfolgten Konzepte des dieser Arbeit vorausgehenden Master-Projekts [49] bildete die theoretische Grundlage für darauf aufbauende Implementierungsarbeiten. Außerdem erfolgte die Formulierung von Anforderungen und die umfangreiche Abgrenzung des Systems mithilfe eines Überblicks über vorhandene MPI-Implementierungen und deren Fähigkeiten.

Zur Beschreibung der letztendlichen Umsetzung wurde zunächst die Projekt-Implementierung detailliert betrachtet. Die darauf aufbauenden Änderungen zur konsequenten Umsetzung der REST-Prinzipien bildeten zusammen mit einer Analyse der dadurch erfüllten Anforderungen die beiden zentralen Kapitel des zweiten Teils dieser Arbeit. Auf den konzeptionellen Entwurf von möglichen Optimierungen und Erweiterung des Systems folgte die letztendliche Performancemessung mit synthetischen Benchmarks. Der Einsatz des Systems für ein reales Problem des wissenschaftlichen Rechnens und der damit einhergehende Vergleich zu einem etablierten MPI-System bildete den Abschluss.

Als zentrales Ergebnis dieser Arbeit lässt sich festhalten, dass eine MPI-Implementierung entwickelt wurde, die den Einsatz paralleler Programme, die mit MPI realisiert sind, auf einfache Art und Weise flexibler ermöglicht, als es die meisten bestehenden Systeme zulassen. Durch bloßes Starten eines Java-Programms kann mit sehr wenig Konfigurationsaufwand – es muss lediglich ein `machines`-File angelegt werden – die Ausführung eines MPI-Programms über das Internet auf Basis von sehr heterogener Hardware angestoßen werden. Zur Erreichung des gleichen Ziels mit konventionellen MPI-Systemen, wie MPICH2 oder Open MPI, sind sehr viele Schritte mehr und viel Erfahrung mit solchen Systemen notwendig. Hervorzuheben ist außerdem die Tatsache, dass eine solche Funktionalität durch relativ einfache Maßnahmen, wie die Nutzung von RESTful HTTP zur Kommunkation, herbeigeführt werden konnte. Systeme wie MPJ/IBIS bieten zwar

noch über die Fähigkeiten des entwickelten Systems hinausgehende Möglichkeiten und eine bessere Performance, hierzu war jedoch die Entwicklung des äußerst komplexen IBIS-Systems erforderlich.

Die vorliegende Arbeit leistet außerdem mit dem Einsatz von RESTful HTTP für MPI zum einen eine Evaluierung dieses Architekturstils für einen bisher nicht in diesem Kontext vorzufindenden Einsatzzweck. Zum anderen stellt auch die Verwendung des HTTP-Protokolls im MPI-Umfeld in den Ausmaßen eines voll funktionsfähigen Systems ein Novum dar.

Es konnte gezeigt werden, dass allein die reine Umsetzung der REST-Prinzipien für ein verteiltes System im Umfeld des World Wide Web Vorteile bringt. Diese sind einerseits auf der Leistungsseite zu finden. Die aktuelle Version des entwickelten Systems bietet gerade im kritischen Bereich der kleinen Nachrichten eine bessere Performance als die Variante des Master-Projekts, die sich noch nicht REST-konform nennen konnte. Andererseits – und dies ist besonders hervorzuheben – zeigen sich Effekte, die konzeptionelle Arbeit am entwickelten System und daraus folgend dessen Implementierung wesentlich vereinfachen. Dies lässt sich zunächst bereits an der konzeptionell einfachen Kommunikationsstruktur mit dem Aufruf weniger HTTP-Verben auf klar strukturierten Ressourcen-URIs erkennen. Beim Versuch, einen Teil des Optimierungspotentials des Systems zu ergründen, zeigte sich in Kapitel 8, dass die Konzepte solcher Erweiterungen sich direkt aus den Möglichkeiten der RESTful Architektur ergeben. Somit kann die Aussicht auf konzeptionell relativ einfache Optimierung des Systems proklamiert werden.

Auf Plattformen, auf denen konventionelle MPI-Systeme ohne weiteres eingesetzt werden können, also auf relativ homogenen, eng gekoppelten Rechnersystemen, kann die umgesetzte Implementierung keine mit diesen vergleichbare Performance erzielen. Weder das Übertreffen der Geschwindigkeit bestehender Systeme, noch die Verwendung in einem solchen Umfeld, sind jedoch die Intentionen des entwickelten Systems. Seine Stärken entfaltet dieses auf Basis von weit verteilten und heterogenen Rechnern, die über heterogene Netze miteinander verbunden sind. Ein Vergleich mit den klassischen MPI-Implementierungen fällt in diesem Umfeld zugunsten des hier vorgestellten Systems aus, da erstere dort nicht oder nur sehr schwierig einsetzbar sind.

Literatur

[1] B. Ramsdell u. a. *RFC 5751: Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification*. Request for Comments. Internet Engineering Task Force, 2010.

[2] Edgar Gabriel u. a. „Distributed computing in a heterogeneous computing environment“. In: *Euro PVM/MPI*. 1998.

[3] J. Gregorio u. a. *RFC 6570: URI Template*. Request for Comments. Internet Engineering Task Force, 2012.

[4] Rob T. Aulwes u. a. „Architecture of LA-MPI, a Network-Fault-Tolerant MPI“. In: *International Parallel and Distributed Processing Symposium*. 2004.

[5] Rob T. Aulwes u. a. „Network Fault Tolerance in LA-MPI“. In: *Euro PVM/MPI*. 2003.

[6] Roberto Chinnici u. a. *Java Platform, Enterprise Edition (Java EE) Specification, v6*. Specification. Sun Microsystems, Inc., 2009.

[7] Roy T. Fielding u. a. *RFC 2616: Hypertext Transfer Protocol – HTTP/1.1*. Request for Comments. The Internet Society, 1999.

[8] Sriram Sankaran u. a. „The LAM/MPI checkpoint/restart framework: System-initiated checkpointing“. In: *International Journal of High Performance Computing Applications* (2004).

[9] Tim Berners-Lee u. a. *RFC 3986: Uniform Resource Identifier (URI): Generic Syntax*. Request for Comments. The Internet Society, 2005.

[10] Vinton Cerf u. a. *RFC 675: Specification of Internet Transmission Control Program*. Request for Comments. Network Working Group, 1974.

[11] Oliver Aumage und Guillaume Mercier. „MPICH/Madeleine: a True Multi-Protocol MPI for High Performance Networks“. In: *Parallel and Distributed Processing Symposium, International* 1 (2001), 10051a. ISSN: 1530-2075.

[12] M. Baker, B. Carpenter und A. Shafi. „MPJ Express: towards thread safe Java HPC". In: *Cluster Computing, 2006 IEEE International Conference on*. IEEE. 2006, S. 1–10.

[13] H. Bal u. a. „Performance evaluation of the Orca shared-object system". In: *ACM Trans. on Computer Systems*. Bd. 16.1. 1998, S. 1–40.

[14] B. Barrett u. a. „Analysis of the Component Architecture Overhead in Open MPI". In: *Proceedings, 12th European PVM/MPI Users' Group Meeting*. Sorrento, Italy, 2005.

[15] Markus Bornemann, Rob V. van Nieuwpoort und Thilo Kielmann. „MPJ/Ibis: a flexible and efficient message passing platform for Java". In: *Proceedings, 12th European PVM/MPI Users' Group Meeting*. Sorrento, Italy, 2005, S. 217–224.

[16] Greg Burns, Raja Daoud und James Vaigl. „LAM: An Open Cluster Environment for MPI". In: *Proceedings of Supercomputing Symposium*. 1994, S. 379–386.

[17] B. Carpenter u. a. „mpijava 1.2: API Specification". In: (1999).

[18] Bryan Carpenter u. a. *MPJ: MPI-like message passing for Java*. Paper 64. Northeast Parallel Architecture Center, 2000.

[19] R. H. Castain u. a. „The Open Run-Time Environment (OpenRTE): A Transparent Multi-Cluster Environment for High-Performance Computing". In: *Proceedings, 12th European PVM/MPI Users' Group Meeting*. Sorrento, Italy, 2005.

[20] Edinburgh Parallel Computing Centre. *The Java Grande Forum MPJ Benchmarks*. University of Edinburgh. 2001. URL: `http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/mpj.html` (besucht am 08. 12. 2011).

[21] Oracle Corporation. *GlassFish*. 2012. URL: `http://glassfish.java.net/de/` (besucht am 20. 09. 2012).

[22] Peter Deutsch. *The Eight Fallacies of Distributed Computing*. 1994. URL: `http://brannigan.emporia.edu/projects/GRID/references/19sep2005/Fallacies.pdf` (besucht am 04. 10. 2012).

[23] Peter Deutsch. *The Eight Fallacies of Distributed Computing*. 2004. URL: `http://www.rgoarchitects.com/Files/fallacies.pdf` (besucht am 04. 10. 2012).

[24] Jason Duell. *The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart*. 2002.

[25] Graham E. Fagg und Jack J. Dongarra. „FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World“. In: *Proceedings of EuroPVM-MPI 2000*. Bd. 1908. Jan. 2000, S. 346–353.

[26] Roy T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Dissertation. Irvine: University of California, 2000.

[27] Roy T. Fielding. *REST APIs must be hypertext-driven*. 2008. URL: `http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven` (besucht am 02. 09. 2012).

[28] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard. Version 3.0*. Standard. University of Tennessee, 2012.

[29] Apache Software Foundation. *Connection management*. 2012. URL: `http://hc.apache.org/httpcomponents-client-ga/tutorial/html/connmgmt.html` (besucht am 10. 01. 2013).

[30] Apache Software Foundation. *HttpComponents HttpClient Overview*. 2012. URL: `http://hc.apache.org/httpcomponents-client-ga/index.html` (besucht am 10. 01. 2013).

[31] Edgar Gabriel u. a. „Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation“. In: *Proceedings, 11th European PVM/MPI Users' Group Meeting*. Budapest, Hungary, 2004, S. 97–104.

[32] Richard L. Graham u. a. „Open MPI: A High-Performance, Heterogeneous MPI“. In: *Proceedings, Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*. Barcelona, Spain, 2006.

[33] William Gropp und Barry Smith. *Users Manual for the Chameleon Parallel Programming Tools*. Argonne National Laboratory. 1993.

[34] William Gropp u. a. „A high-performance, portable implementation of the MPI message passing interface standard“. In: *Parallel Computing*. Bd. 22. 1996, S. 789–828.

[35] M. Hafeez u. a. „Survey of MPI Implementations“. In: *International Conference on Digital Information and Communication Technology and its Applications (DICTAP 2011)*. Bd. 167. Dijon, Frankreich, 2011, S. 1011–1025.

[36] E. Hammer-Lahav. *RFC 5849: The OAuth 1.0 Protocol*. Request for Comments. Internet Engineering Task Force, 2010.

[37] Ingrid van den Hoogen. *Deutsch's Fallacies, 10 Years After*. 2004. URL: `http://brannigan.emporia.edu/projects/GRID/references/19sep2005/Fallacies.pdf` (besucht am 04. 10. 2012).

[38] Joshua Hursey u. a. „The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI". In: *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Long Beach, CA, USA: IEEE Computer Society, März 2007.

[39] Trustees of Indiana University. *LAM/MPI Overview*. 2006. URL: `http://www.lam-mpi.org/about/overview/` (besucht am 01. 11. 2012).

[40] *ISO/IEC 7498-1: Information Technology – Open Systems Interconnection – Basic Reference Model: The Basic Model*. International Standard. International Organization for Standardization, 1996.

[41] J.Squyres u. a. „The interoperable message passing interface (IMPI) extensions to LAM/MPI". In: *Proceedings of the MPI Developer's Conference*. Ithica, NY, 2000.

[42] N. Karonis, B. Toonen und I. Foster. „MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface". In: *Journal of Parallel and Distributed Computing* 63 (2003), S. 551–563.

[43] M. Korch und T. Rauber. „Parallel Low-Storage Runge—Kutta Solvers for ODE Systems with Limited Access Distance". In: *International Journal of High Performance Computing Applications* 25.2 (2011), S. 236–255.

[44] A. Lastovetsky. „mpC: A Multi-Paradigm Programming Language for Massively Parallel Computers". In: *ACM SIGPLAN Notices* 31 (1996), S. 13–20.

[45] A. Lastovetsky und R. Reddy. „HeteroMPI: Towards a Message-Passing Library for Heterogeneous Networks of Computers". In: *Journal of Parallel and Distributed Computing* 66 (2006), S. 197–220.

[46] J. Liu u. a. „Design and Implementation of MPICH2 over InfiniBand with RDMA Support". In: *Proceedings of the International Parallel and Distributed Processing Symposium*. 2004.

[47] J. Maassen und H.E. Bal. „Smartsockets: solving the connectivity problems in grid computing". In: *Proceedings of the 16th international symposium on High performance distributed computing*. ACM. 2007, S. 1–10.

[48] M.D. McNally, Jeffrey M. Squyres und Andrew Lumsdaine. *A Freeware Implementation of the IMPI Server*. University of Notre Dame. 1999.

[49] Daniel Mohr. *Implementierung einer MPI-Bibliothek auf Basis von RESTful Webservices.* Master-Projekt. Universität Bayreuth, Lehrstuhl für Angewandte Informatik II, 2012.

[50] MPICH. *MPICH Overview.* 2012. URL: `http://www.mpich.org/about/overview/` (besucht am 10. 11. 2012).

[51] Rob V. van Nieuwpoort u. a. „Ibis: an Efficient Java-based Grid Programming Environment". In: *Joint ACM Java Grande - ISCOPE 2002 Conference.* Seattle, Washington, USA, 2002, S. 18–27.

[52] J. Pješivac-Grbović u. a. „Performance analysis of MPI collective operations". In: *Cluster Computing* 10.2 (2007), S. 127–143.

[53] The Open MPI Project. *FAQ: Fault tolerance for parallel MPI jobs.* 2012. URL: `http://www.open-mpi.org/faq/?category=ft` (besucht am 15. 10. 2012).

[54] The Open MPI Project. *Open MPI: Open Source High Performance Computing.* 2012. URL: `http://www.open-mpi.org/` (besucht am 15. 10. 2012).

[55] V. S. Sunderam. „PVM: A Framework for Parallel Distributed Computing". In: *Concurrency: Practice and Experience.* Bd. 2.4. 1990, S. 315–339.

[56] Stefan Tilkov. *REST und HTTP – Einsatz der Architektur des Web für Integrationsszenarien.* Heidelberg, 2009.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht sind und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

München, den 27. Januar 2013