Lehrstuhl für Rechnertechnik und Rechnerorganisation

Effiziente Methoden für global verteiltes wissenschaftliches Rechnen

Philipp Drum

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. H. M. Gerndt

Prüfer der Dissertation:

1. Univ.-Prof. Dr. A. Bode

2. Univ.-Prof. Dr. E. Jessen

Die Dissertation wurde am 19. März 2001 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 29. Mai 2001 angenommen.

Kurzfassung

Für langlaufende, parallele wissenschaftliche Anwendungen mit schwacher Synchronisation bieten Metacomputer-Architekturen sehr hohe Rechenleistung bei geringen zusätzlichen Kosten.

Aktuelle Systeme verwenden bei der Koppelung von Rechenressourcen dedizierte Hochleistungsnetzwerke für Anwendungen mit feingranularer Kommunikation, oder sie sind auf lose gekoppelte parallele Anwendungen beschränkt, um mit architekturbedingten hohen Kommunikationslatenzen und schwankenden Kommunikationsdurchsätzen umzugehen.

Im Rahmen dieser Arbeit zeigt eine Analyse der Methoden von existierenden Metacomputern die Schwäche für kommerzielle Nutzer. Deren bestehende Anwendungen benötigen aufwändige, kostenintensive Anpassungen, um die Rechenleistungen komplexer Metacomputer nützen zu können. Hohe Anpassungskosten implizieren eine Abhängigkeit der Software an die Metacomputing-Umgebung. Dieses wirtschaftliche Risiko reduziert die Akzeptanz von Metacomputern in der Industrie.

Als Ergebnis dieser Analyse wird eine mit der Programmiersprache Java implementierte Metacomputing-Umgebung entwickelt, die eine Lösung dieser Probleme bereitstellt. Hierzu wird eine neue und effiziente Methode der Prozessverteilung vorstellt. Diese basiert auf einer Optimierung der Kommunikationsleistung der parallelen Anwendung durch eine Platzierung kommunikationsintensiver Prozesse auf gut angebundene Rechnerknoten. Zusammen mit der Fähigkeit, auf Veränderungen der Kommunikationsleistung des Verbindungsnetzwerkes dynamisch zu reagieren, ermöglicht das System eine anwendungstransparente Optimierung der Gesamtlaufzeit. Durch den threadbasierten Aufbau und die angebotenen Schnittstellen zur externen Steuerung des Systems wird eine sehr einfache Anpassung bestehender Anwendungen ermöglicht. Die Leistungsfähigkeit wird anhand einer Reihe von Modellanwendungen verifiziert.

Der Einsatz im industriellen Umfeld wird anhand eines Produktes zum Entwurf- und Konstruktionsprozess der Firma Tecoplan AG, die 'Virtuelle Werkstatt', evaluiert. Basierend auf einer Voxeltechnologie profitiert die rechenintensive Anwendung zur Einbau- und Paßform-Simulation von dem Geschwindigkeitsgewinn durch die parallele Ausführung der Kollisionsberechnung und durch die plattformunabhängige Prozessverteilung der vorgestellten Middleware.

Danksagung

Hiermit möchte ich mich bei den Personen bedanken, die diese Arbeit durch ihr Wirken erst ermöglicht haben. An erster Stelle möchte ich mich bei Herrn Prof. Dr. Arndt Bode für seine großzügige Unterstützung und für die hervorragenden Arbeitsbedingungen bedanken, die ein optimales Umfeld für effiziente Forschung und freundliches Arbeitsklima ergeben. Auch Herrn Prof. Dr. Eike Jessen möchte ich herzlich für die Übernahme des Zweitgutachtens und für seine wertvolle fachliche Hilfe danken.

Allen Mitarbeitern des Lehrstuhl für Rechnertechnik und Rechnerorganisation möchte ich für die angenehme Arbeitsumgebung danken. Besonderer Dank geht hierbei an das Sekretariat des LRR, das mich im Kampf mit den Formularen sehr unterstützt hat.

Für fachliche und nicht-fachliche Diskussionen möchte ich einer langen Reihe von Personen herzlich danken: Dr. Markus Leberecht, Markus Lindermeier, Dr. Michael May, Dr. Günther Rackl, Martin Schulz, Klaus Tilk, Dr. Jörg Trinitis, Dr. Matthias Weidmann, Dr. Ivan Zoraja, sowie meinen Gruppenleitern Dr. Peter Luksch und Dr. Thomas Ludwig.

Der Firma Tecoplan AG sowie dem FORTWIHR möchte ich für die Zusammenarbeit und die Bereitstellung aller möglichen Ressourcen danken.

Wichtige Anregungen und interessante Diskussionen verdanke ich “meinen Azubis” und Studenten.

Während dieser Zeit haben verschiedene Personen versucht, teilweise erfolgreich, mich von der Arbeit abzuhalten. Dafür besonderen Dank an Ralf Jost, Dominik Orth, johnc, Stefan Petry, Ralf Storm, Ralf Westram und sehr vielen anderen.

Weiterhin möchte ich meiner Familie ganz herzlich für die Unterstützung danken.

Für ihre nahezu endlose Geduld und für zahlreiche “verarbeitete” Wochenenden möchte ich meiner Lebensbegleiterin Sabine Nicklas sehr herzlich danken.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Verteiltes und paralleles Rechnen in Metacomputing-Umgebungen . . . 1
1.2 Bedeutung der Netzinfrastruktur . . . 3
1.3 Metacomputer-Strukturen . . . 4
1.4 Metacomputer in der Anwendung . . . 5
1.5 Aufbau der Arbeit . . . 6

2 Techniken für weitverteilte Systeme **9**
2.1 Historische Entwicklung und Motivation . . . 9
2.2 Anforderungen für weitverteilte Systeme . . . 12
2.3 Klassifizierung von Metacomputern . . . 17
2.3.1 Modelle verteilten Rechnens . . . 17
2.3.2 Unterscheidungskriterien von Metacomputern . . . 20
2.3.3 Terminologie im Bereich Metacomputing . . . 24
2.4 Verbreitete Techniken zur verteilten Ausführung . . . 26
2.4.1 Dienste . . . 26
2.4.2 Verfügbare Techniken . . . 28
2.5 Sicherheitsaspekte . . . 33
2.6 Zusammenfassung . . . 37

3 Metacomputer-Systeme **39**
3.1 Metacomputer-Systeme . . . 39
3.1.1 Projekte zur Nutzung von trivialer Parallelität . . . 40
3.1.2 Java basierte Client/Server Umgebungen . . . 43
3.1.3 Atlas . . . 46
3.1.4 Legion . . . 48
3.1.5 Globus . . . 52
3.1.6 Weitere Systeme . . . 56
3.2 Zusammenfassung und Bewertung . . . 59
3.2.1 Vor- und Nachteile . . . 59
3.2.2 Anforderungen für kommerzielle Systeme . . . 62

4 LsD - Ein System zur Optimierung von verteilter Programmausführung **65**

4.1 Motivation 65
4.2 Theoretischer Ansatz zur Prozessverteilung 69
4.3 Klassifikation der Zielumgebung 73
4.4 Architektur und Implementierung 74
4.5 Java 87
4.5.1 Spracheigenschaften 88
4.5.2 Laufzeitverhalten 90
4.5.3 Java für wissenschaftlich-technisches Rechnen 93
4.5.4 Zusammenfassung 95
4.6 Experimentierumgebung 96
4.7 Zusammenfassung 106

5 Einsatz und Evaluierung der Anforderungen im industriellen Umfeld 109
5.1 Digitales Prototyping in der Automobilindustrie 109
5.1.1 Virtuelle Montagen 110
5.1.2 Die “Virtuelle Werkstatt” 112
5.1.3 Leistungssteigerung durch Parallelisierung 114
5.2 Die parallele Version der “Virtuellen Werkstatt” – viw/p 115
5.2.1 Modulanalyse 115
5.2.2 Parallelisierung der “Virtuellen Werkstatt” 118
5.3 Ergebnis der Analyse 121

6 Zusammenfassung und Ausblick 125
6.1 Zusammenfassung 125
6.2 Ausblick 128

Literaturverzeichnis 129

Definitionen 145

Index 146

Abbildungsverzeichnis

1.1 Metacomputer als vermittelnde Schicht zwischen Anwendung und Hardware . . . 4
1.2 Das LSD-System als “Middle”-Middleware . . . 7

2.1 Allgemeiner Aufbau eines Metacomputers . . . 11
2.2 Parallele Programmierbibliotheken . . . 18
2.3 Objektorientiertes Modell . . . 18
2.4 Metacomputer-Modell . . . 19
2.5 Passive und aktive Metacomputing-Mechanismen . . . 20
2.6 Gemeinsamer Objektraum bei *JavaSpace* . . . 29
2.7 Technik des entfernten Methodenaufrufs . . . 31
2.8 Klassifizierung von Informationen des MIMO Systems. . . . 33
2.9 Mögliche Angriffspunkte auf ein verteiltes Metacomputing-System . 35

3.1 Die Javelin Architektur . . . 44
3.2 Die JET Architektur . . . 46
3.3 Fibonacci Implementierung mit Cilk . . . 47
3.4 Format einer *LOID* . . . 49
3.5 Prinzip der Nutzung fremder Dienste unter *Globus* . . . 56

4.1 Schichtenmodell bei LsD Benutzung . . . 67
4.2 Suboptimaler Algorithmus zur Prozessverteilung . . . 72
4.3 Module des *LsD-Systems* im Überblick . . . 76
4.4 Knotenverwaltung innerhalb des *LsD-Systems* . . . 78
4.5 Schema eines LsD Knotens . . . 79
4.6 Prozesskapselung . . . 79
4.7 Kapselung von LsD Jobs . . . 80
4.8 Latenztabellen . . . 81
4.9 Algorithmus zur Bestimmung eines Zielrechners . . . 82
4.10 Kommunikationsstruktur einer LsD Anwendung . . . 83
4.11 Anwendungsbezogene Kommunikation im LsD-System . . . 84
4.12 Instrumentierung des *LsD-Systems* zur Beobachtung mit MIMO . . 85
4.13 Instrumentierung des NodeDaemon mit MIMO Adaptern . . . 86
4.14 Instrumentierung einer *LsD* Applikation mit *MIMO* Adaptern . . . 86
4.15 Das Prinzip der Java Plattform: Die Virtuelle Maschine und das API 89

4.16 Das Java Sicherheitsmodell in der JDK Version 1.2. 91
4.17 Ausführung eines Java Programms 92
4.18 Kommunikationsstruktur einer LsD Anwendung 97
4.19 Code zur Zeitmessung 99
4.20 Fortwährende “Ping” Versuche zu Rechner “sonar”. 100
4.21 Aufbau der Experimentierumgebungen. 101
4.22 Skalierbarkeit der Architektur 102
4.23 Vergleich der Anwendung A_1, optimiert und zufällige Verteilung . . 102
4.24 Vergleich der Anwendung A_2, optimiert und zufällige Verteilung . . 103
4.25 Vergleich der Anwendung A_3, optimiert und zufällige Verteilung . . 103
4.26 Vergleich der Anwendung A_4, optimiert und zufällige Verteilung . . 104
4.27 Der Speedup der optimierten Anwendungen A_{1-4} in Abhängigkeit der Latenzzeit 106

5.1 Darstellung eines Fahrzeugs durch die *Virtuelle Werkstatt* 111
5.2 Schichten der *Virtuellen Werkstatt* 113
5.3 Externer Zugang zur *Virtuellen Werkstatt* 114
5.4 Aufbau der sequentiellen *Virtuellen Werkstatt* 116
5.5 Aufgabe des Job-Servers innerhalb der *Virtuellen Werkstatt* 117
5.6 Konvertierung einer Modellrepräsentation in das Modell der *Virtuellen Werkstatt* 118
5.7 Analyse der Methodenaufrufe bezüglich ihrer verbrauchten Rechenzeit. 119
5.8 Interaktion der modifizierten Module der *Virtuellen Werkstatt* 120
5.9 Partitionierung und Berechnung von zwei Jobs innerhalb der parallelen *Virtuellen Werkstatt viw/p* 122
5.10 Struktur einer weitverteilen Berechnung der *Virtuellen Werkstatt* . . 123

Tabellenverzeichnis

2.1 Zusammenhänge der Klassifizierungsmerkmale 24

3.1 Metacomputing-Systeme und deren Techniken im Vergleich 60
3.2 Einsatzmöglichkeit verschiedener Systeme 62

4.1 Parallelitätsmodell 68
4.2 Systeme im Vergleich 69
4.3 Geeignete Einsatzmöglichkeit des *LsD-Systems* in verschiedenen Umgebungen 75
4.4 Die Zuordnung von MIMO MLM Schichten zu LsD 87
4.5 Die Rechner der Testumgebung 100
4.6 Gemessene Geschwindigkeitssteigerungen 105

5.1 Messergebnisse der parallelen “Virtuellen Werkstatt” (1) 121
5.2 Messergebnisse der parallelen “Virtuellen Werkstatt” (2) 121

1. Einleitung

1.1 Verteiltes und paralleles Rechnen in Metacomputing-Umgebungen

Das Gebiet des wissenschaftlich-technischen Hochleistungsrechnens ist, wie viele andere Bereiche der Informatik, Trendwenden und Schwankungen in der Forschungsrichtung unterworfen. Der Erfolg neuer Methoden oder neuer Architekturen kann vielmals erst in den darauffolgenden Jahren abgeschätzt werden. In der gegenwärtigen Situation sind sogenannte "Metacomputer" Gegenstand der Forschung und Grundlage kontroverser Diskussionen über deren zukünftige Architektur und Ausrichtung. Diese Arbeit beschäftigt sich mit dieser neuen Form des verteilten Rechnens und stellt dabei ein neues, leistungsfähiges Konzept zur weitverteilten Ausführung vor, das einfachen Zugang zur praktischen Anwendung und Methoden zur effizienten weitverteilten Ausführung bietet.

In der Entwicklung von dedizierten Hochleistungsrechnern ist eine Dezentralisierung zu beobachten. Waren früher zentrale Vektorrechner, die Instruktionen auf mehrere Daten angewandt haben, das am häufigsten benutzte Mittel, um wissenschaftlich-technische Probleme zu lösen, sind heute verteilte Strukturen anzutreffen. Beispiele sind hier Mehrprozessorsysteme und Multirechnersysteme, die mehrere Instruktionsströme auf mehrere Datenströme anwenden, sowie auf Verbünde von Arbeitsplatzrechnern , die über ein Netzwerk gekoppelt eine Aufgabe bearbeiten. Diese Klassifikation nach Instruktionsströmen und Datenströmen geht auf FLYNN [51] zurück, der Rechner nach SIMD (Single Instruction, Multiple Data) und MIMD (Multiple Instruction, Multiple Data) unterschied (die Mischformen sind für das parallele Rechnen unbedeutend). Diese Dezentralisierung hat in den heute existierenden Metacomputern, die in die MIMD Kategorie eingeordnet werden, die extremste Form angenommen. Obwohl die Programmierparadigmen sich dieser Entwicklung mit Werkzeugen wie PVM/MPI oder RPC angepasst haben, erfordern Metacomputer zusätzliche Konzepte, die aufgrund der Lokalität der

Ressourcen bisher nicht erforderlich waren. Diese Konzepte und ihre Methoden werden in dieser Arbeit erläutert und bewertet.

Mit der möglichen Benutzung weitverteilter Ressourcen zur Lösung großer Probleme setzt sich auch eine höhere allgemeine Verfügbarkeit fort. Parallelrechner waren früher nur für eine sehr eingeschränkte Anzahl von Nutzern verfügbar. Mit der Verwirklichung von gekoppelten Arbeitsplatzrechnern sind auch Plattformen mit hoher Leistung einer breiteren Allgemeinheit zugänglich. Dies ist insbesondere für mittelständische Unternehmen ein wichtiger wirtschaftlicher Faktor. So ist im November 2000, erstmals in Deutschland, ein Verbund aus Linux Rechnern der Universität Chemnitz unter den 500 schnellsten Rechnern der Welt [163]. Metacomputer versprechen hinsichtlich der Ökonomie diesen Trend fortzusetzen, gehorchen allerdings, was die Umsetzung in die Praxis betrifft, eigenen Gesetzen, die von vielen Faktoren abhängig sind.

Anwendungen, die eine sehr hohe Rechenleistung benötigen, kommen traditionell aus dem Bereich des wissenschaftlich-technischen Rechnens. Beispielhafte Anwendungen sind hier Optimalsteuerprobleme, Strömungssimulationen oder strukturmechanische Simulationen. Diese Anwendungsklasse wird in diesem Rahmen im Fokus des Interesses stehen. Dies hat seinen Grund vornehmlich in der "Skalierbarkeit der Anforderungen" dieses Gebietes: Benutzer von Hochleistungsrechnern modellieren ihr Problem notwendigerweise nach der zu vertretenden Rechenzeit und nicht nach der gewünschten Genauigkeit. Steht eine erhöhte Rechenleistung zur Verfügung, werden die Modelle angepasst (durch Erhöhung der Gitterpunkte, Hinzunahme von Randbedingungen und ähnlichen, von der Berechnungsart abhängigen Maßnahmen). Diese hochgenauen Berechnungen mit ausreichender Rechenleistung zu versorgen diente als Motivation für die Koppelung von mehreren Hochleistungsrechnern zu einem Metacomputer.

Die effektive Leistung eines Metacomputers ist jedoch sehr von der Art der Anwendung abhängig. Diese Abhängigkeit ist prinzipiell bereits von anderen Hochleistungsarchitekturen bekannt. Auch Vektorrechner, Multirechnersysteme, Mehrprozessorsysteme[1] und Verbünde von Arbeitsplatzrechnern bevorzugen bestimmte Applikationen, beziehungsweise deren Berechnungen oder Kommunikationsmuster. Im Unterschied hierzu ist jedoch bei Metacomputing-Architekturen die Ausführungsplattform im Allgemeinen nicht bekannt. Zudem limitieren die beschränkte Bandbreite und die hohen Latenzzeiten des Verbindungsnetzwerkes die Möglichkeiten der Parallelisierung, welches wiederum die effiziente Ausführung nur für Anwendungen mit hoher Parallelität erlaubt.

Die Kommunikation in parallelen oder verteilten Anwendungen ist einer der wichtigsten Faktoren für die Effizienz einer Parallelisierung. Aufgrund der Tatsache, dass die Rechenleistung von Prozessoren in stärkerem Maße zunimmt als die Kommunikationsleistung, wird der Nachrichtenaustausch zwischen parallel ausgeführten Prozessen teurer. Das impliziert besonders bei Metacomputern eine hohe Parallelität der einzelnen Aufgaben mit relativ wenig Kommunikationszeit, um eine

[1]Mehrprozessorsysteme (SMP, Symmetric Multiprocessor) besitzen gemeinsamen Hauptspeicher, der global adressiert werden kann, Multirechnersysteme adressieren nur lokalen Speicher.

annehmbare Effizienz zu erreichen. Zu beachten ist auch, dass bei Metacomputern, aufgrund der sehr hohen Anzahl an Ressourcen, eine effiziente Auslastung nicht den Stellenwert hat wie in anderen dedizierten Hochleistungsrechnern. Hier kann an erster Stelle des Interesses nicht eine möglichst hohe, gleichmäßige Auslastung der Prozessoren stehen, sondern eine kurze Rechenzeit (auf Kosten sehr vieler, ineffizient genutzter Prozessoren) oder die Möglichkeit, besonders große Probleme zu berechnen. Auch tendieren Metacomputer-Architekturen im Allgemeinen zu einer Gruppenbildung, also einer Anzahl von Rechnern mit einer zueinander guten Netzanbindung, die zu einer oder mehreren ebensolchen Gruppen eine weniger gut ausgebaute Verbindung unterhält. Hierfür eignen sich bestimmte Anwendungen besonders (beispielsweise "coupled fields"-Algorithmen), dies sollte von der Metacomputer-Umgebung erkannt werden und die Prozessverteilung dementsprechend vorgenommen werden.

1.2 Bedeutung der Netzinfrastruktur

Eine Technologie, die mit der Entwicklung von Metacomputer-Architekturen eng verknüpft ist, beziehungsweise sie erst ermöglicht hat, ist die Entwicklung der Netzwerke. Bereits in Multirechnersystemen mit sehr hoher Prozessorenanzahl ist ein erheblicher Aufwand in die Entwicklung der Prozessor zu Prozessor Verbindung in Form von *Crossbars* oder *Hypercube* Topologien investiert worden, um eine schnelle interne Kommunikation zu ermöglichen. In Verbünden von Arbeitsplatzrechnern kommen Technologien wie Gigabit Ethernet, SCI (Scalable Coherent Interface) oder Myrinet zum Einsatz, welche die Latenzzeiten der Kommunikation in Rechnernetzen von Millisekunden auf wenige Mikrosekunden senken. Die Latenzzeit ist hierbei die Zeit von der Vergabe bis zur Erledigung eines Transportauftrages kleiner Größe auf der Ebene der Dienstschnittstelle drei des OSI Schichtenmodells ([150], siehe auch Definition 4.1). Bei Metacomputing-Architekturen ist eine Lösung des Latenzzeitproblems durch Hardwarelösungen auf absehbare Zeit nicht in Sicht. Die Latenzzeiten sind hier um Größenordnungen höher und die Bandbreiten kleiner als in sämtlichen anderen Hochleistungsrechenarchitekturen. Dies hat eine starke Beschränkung in den geeigneten Anwendungen zur Folge.

Der enorme Erfolg des *Internets* und dessen schnelle Ausbreitung in den letzten zehn Jahren hatte den schnellen Ausbau eines globalen Verbindungsnetzes zur Folge. Zusammen mit der oben angesprochenen Dezentralisierung von Ressourcen bei der parallelen Abarbeitung kann die Entstehung von Metacomputern als ein Effekt dieser beiden Faktoren verstanden werden. Das Verbindungsnetzwerk ist dabei durch seine, relativ zu anderen Hochleistungsrechnern, niedrigen Leistungswerte zu einer gleichwertigen Komponente eines Gesamtsystems geworden, die bei der Ermittlung von Leistungsdaten mit berücksichtigt und optimiert werden muss ("The net *is* the computer" lautet ein passender Werbeslogan eines Herstellers). Erschwerend kommt hinzu, dass im Gegensatz zu Prozessor- oder Busleistung die Netzleistung keine konstante Größe ist. Die dynamische Beobachtung der Netztopologie

wird eine wichtige Dienstleistung einer Metacomputer-Umgebung sein müssen, um eine effiziente Ausführung zu ermöglichen.

1.3 Metacomputer-Strukturen

Metacomputer können allgemein als eine Dienste vermittelnde Schicht zwischen Ressourcen aller Art und der Anwendungen gesehen werden. Ähnlich der Struktur innerhalb eines Rechners, indem zwischen der Anwendung und der Hardware das Betriebssystem Dienste vermittelt, stellt eine Metacomputer-Architektur verteilte Dienste für verteilte Anwendungen zur Verfügung. Der Übergang zwischen einer verteilten Anwendung auf Multirechnersystemen oder in einem lokalen Netzwerk zu einer global verteilten Anwendung ist dabei fließend. Der Aufbau einer solchen Middleware ist bei allen Systemen in lokale und globale Komponenten unterteilt. Lokale Komponenten sind auf den beteiligten Rechnern für die Ausführung und Verwaltung der Anwendung installiert, globale Komponenten verbinden diese zu einem Gesamtsystem (siehe Abbildung 1.1).

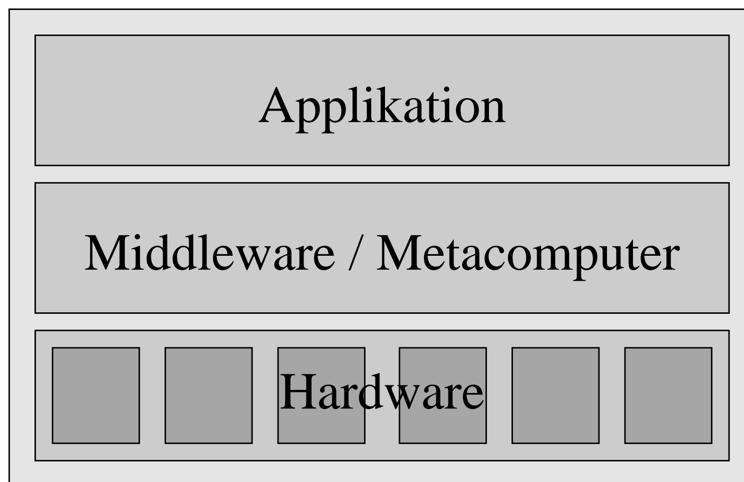

Abbildung 1.1: Metacomputer als vermittelnde Schicht zwischen Anwendung und Hardware

Innerhalb dieses prinzipiellen Aufbaus unterscheiden sich Metacomputer sowohl von den eingesetzten Techniken als auch von der Mächtigkeit der angebotenen Dienste. In Kapitel 2 werden aktuelle Metacomputer-Umgebungen analysiert. Als grundlegende und praxisorientierte Unterscheidungskriterien bei Metacomputern bieten sich die für die Anwendung verfügbaren Dienste oder auch die anvisierte Parallelität der Anwendung an, die oft durch die Merkmale einer Metacomputer-Architektur vorbestimmt ist.

Einfache Metacomputer-Architekturen realisieren eine zentrale Komponente, die Probleme mit sehr hoher Parallelität partitioniert und eine hohe Anzahl an Teilbe-

rechnungen an Rechner verschickt. Diese Form beruht häufig auf freiwilliger Basis der teilnehmenden Rechner und bietet, von der Prozessverteilung abgesehen, keine weiteren Dienste an. Zentrale Strukturen, die weitere Dienste anbieten, wie Kommunikationsmöglichkeiten der verteilten Anwendung oder intelligente Lastverteilung, bilden eine weitere Stufe der Entwicklung bei Metacomputern. Die höchste Stufe in der Komplexität erreichen dezentrale Architekturen, die verteilte, dynamische Ressourcen verwalten und die Informationen und Zugriff zu diesen der Anwendung zur Verfügung stellen. Hinzu kommen verschiedene Methoden, die es den Teilen der Anwendung erlauben, miteinander zu kommunizieren oder über einen gemeinsamen Objektraum zu verfügen.

In [55] stellen FOSTER und Kesselman das Ziel zukünftiger Metacomputing-Infrastrukturen vor, das Anwendungen eine nahtlose Benutzung global verteilter Ressourcen ermöglicht, ohne dass die Anwendung mit deren Verwaltung betroffen ist. Die Handhabung eines solchen Metacomputers ist an die transparente Benutzung von Elektrizitätsnetzen angelehnt. Bis zu der Verwirklichung einer Metacomputer-Umgebung, die sich der Anwendung als ein globales Betriebssystem darstellt, sind jedoch noch Entwicklungen wie einheitliche Protokolle, umfangreichere Dienste zur Ressourcenerkennung, Zuteilung und entfernter Benutzung sowie Werkzeuge zur Beobachtung, Fehlersuche und Leistungsanalyse notwendig, die auch in der Lage sein müssen, Informationen auszutauschen. Diese Integration der vorhandenen Techniken wird einer der zentralen Forschungspunkte der zukünftigen Entwicklung sein müssen.

Weiterhin gilt, dass eine hohe Abstraktionsebene, die der Metacomputer dem Benutzer anbietet, im Allgemeinen mit einer Leistungseinbuße bezahlt werden muss. Hierbei wird der physikalische Standort von Ressourcen oder von verteilten Objekten vor dem Benutzer verborgen, dadurch können ineffiziente Zugriffsmuster entstehen, die sich bei Kommunikationen mit hoher Latenz negativ auf die Gesamtlaufzeit auswirken können [172]. Hier sind geeignete Zugriffsmethoden und Optimierungsverfahren zu entwickeln.

1.4 Metacomputer in der Anwendung

Projekte aus dem universitären Bereich, wie sie *Globus* [63] und *Legion* [73] verwirklichen, zeigen, dass sich Metacomputing-Architekturen für das wissenschaftlich-technische Rechnen nutzen lassen. Hier können verteilte Ressourcen der Universitäten und anderer Institute zur Berechnung großer Probleme und zur Geschwindigkeitssteigerung eingesetzt werden. Auch durch die Koppelung mehrerer dedizierter Hochleistungsrechenanlagen sind bisher sehr hohe Rechenleistungen erzielt worden [12]. Diese werden im wissenschaftlich-technischen Bereich vornehmlich für Berechnungen der sogenannten "Grand Challenges"[145] eingesetzt.

Die Umsetzung dieser Projekte für Unternehmen mit hohem Rechenbedarf hat eine große wirtschaftliche Bedeutung (das hier auftretende Sicherheitsproblem wird in Abschnitt 2.5 behandelt). Hier hat die Anschaffung von Rechnerarchitekturen mit

hoher Rechenleistung großen Einfluss auf die betriebswirtschaftliche Planung. Eine kostengünstige Nutzung der im Intranet und Extranet verfügbaren Ressourcen durch eine transparente Koppelung der Architekturen zu einem Metacomputer kann hier einen entscheidenden Wettbewerbsvorteil bedeuten. Der Einsatz eines Metacomputers, der die lokalen Gegebenheiten ausnützt und dabei die Entwicklungskosten für parallele Software durch einen vereinheitlichten Zugriff auf die Hardware senkt, kann für mittelständische Unternehmen einen Technologievorsprung bewirken, der in der industriellen Praxis auch zukünftige Leistungsanforderungen erfüllt.

Hierbei ist ein Mittelweg erforderlich, der Anwendern und Entwicklern einen nahtlosen Zugang zu hoher, verteilter Rechenleistung ermöglicht, ohne einerseits mit der Handhabung von Basistechniken wie Prozessplatzierung oder Kommunikationsbibliotheken konfrontiert zu sein, und ohne andererseits Zeit in die aufwändige Installation und Verwaltung einer der benannten großen universitären Metacomputing-Projekte zu investieren. Eine solche Umgebung ermöglicht kürzere, kostengünstige Entwicklungszyklen, die auch kleinen und mittleren Unternehmen mit begrenzten finanziellen Möglichkeiten für Hardwareanschaffungen hohe Rechenleistungen ermöglicht.

Einen Lösungsansatz hierfür stellt das in Kapitel 4 beschriebene System *LsD*[2] vor. *LsD* implementiert eine leichtgewichtige Metacomputing-Middleware, die eine Koppelung von Rechenressourcen auf der Basis der plattformübergreifenden Programmiersprache Java erlaubt. Hierbei standen geringe Entwicklungskosten für die Anwendung, hohe Portabilität und anwendungstransparente Optimierung durch die Middleware im Vordergrund. Eine Umgebung mit diesen Merkmalen, die ohne hohen Installations- und Implementierungsaufwand Zugang zu hohen Rechenleistungen ermöglicht, ist bisher nicht entwickelt worden. Abbildung 1.2 zeigt das *LsD-System* als Lösung zwischen einfachen und aufwändigen Metacomputer-Installationen.

1.5 Aufbau der Arbeit

In dieser Arbeit wird ein Überblick über effiziente Methoden zur Realisierung von Metacomputern gegeben und es werden Systeme analysiert, die diese Methoden implementieren. Als ein Ergebnis dieser Analyse wird eine neue Methode entwickelt, die die Abarbeitung paralleler Anwendungen optimiert, indem sie Kommunikationsparameter bei der Verteilung berücksichtigt.

Im anschließenden Kapitel wird auf die Grundlagen von Metacomputer-Systemen eingegangen, wobei die für die globale Ausführung bedeutenden Techniken im Vordergrund stehen. Es werden Unterscheidungskriterien für Metacomputer-Systeme formuliert und die verwendete Terminologie definiert. Aus diesen Merkmalen leitet sich eine Klassifikation der Systeme ab. Abschnitt 2.5 geht detaillierter auf die Sicherheitsproblematik in globalen Netzen ein. In Kapitel 3 werden existiernde

[2] *L*atency *s*ensitive *D*istribution, Prozessverteilung unter Berücksichtigung der Kommunikationslatenzen der rechnenden Knoten.

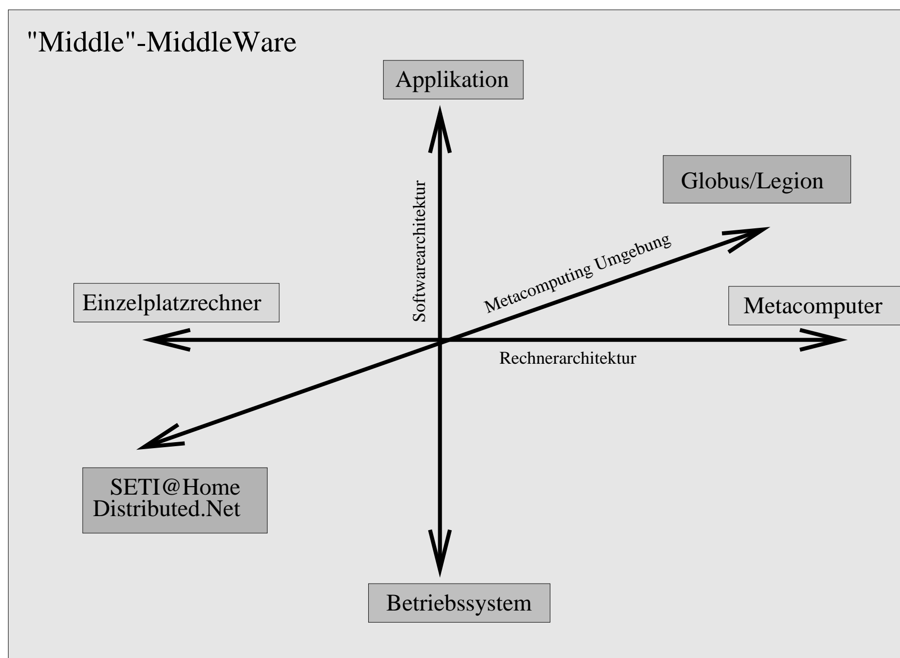

Abbildung 1.2: Das LSD-System als “Middle”-Middleware

Metacomputer-Projekte diskutiert. Abschliessend wird eine Zusammenfassung gegeben und Einschränkungen der vorhandenen Systeme für kommerzielle Nutzer beschrieben, die zur Motivation der Entwicklung des *LsD-Systems* führen.

In Kapitel 4 wird das *LsD-System* als leichtgewichtige und effiziente Metacomputing-Umgebung erläutert. Mit dem *LsD-System* wird durch eine optimierende Verteilung von Java Threads eine neue Methode vorgestellt, die besonders für mittelständische Firmen und für Institute eine kostengünstige und gleichzeitig leistungsstarke Alternative zu aufwändigen Metacomputer-Installationen bietet. Es wird detailliert auf die Prozessverteilung, die Architektur und die Implementierung eingegangen. In Abschnitt 4.6 wird die Leistungsfähigkeit anhand praxisrelevanter, synthetischer Benchmarks verifiziert. Anschließend wird in Kapitel 5 das *LsD-System* anhand der Rechnerumgebung eines Unternehmens mit großem Bedarf an Rechenleistung verifiziert. Hierzu wird der Rechenbedarf eines Beispielunternehmens, der *Tecoplan AG*, untersucht, sowie die Anwendung auf parallele Abarbeitung analysiert. Kapitel 6 fasst die Analyse der gegenwärtigen Metacomputing-Architekturen und die Einführung des Java-basierten *LsD-Systems* für Unternehmen mit hohem Bedarf an Rechenleistung zusammen und gibt einen Ausblick auf mögliche zukünftige Entwicklungen und Anforderungen für Metacomputer.

2. Techniken für weitverteilte Systeme

Dieser Abschnitt gibt eine Einführung zu der Entwicklung von Metacomputern und formuliert eine Reihe von Anforderung, die für diese Systeme gelten. Darauf basierend wird eine Ordnung abgeleitet, nach der bestehenden Systeme klassifiziert werden können. Im Abschnitt 2.4 wird auf Techniken und zum Teil auch auf Werkzeuge eingegangen, die in Zusammenhang mit Metacomputer-Systemen von Bedeutung sind. Anschließend wird wegen der stark wachsenden Bedeutung auf das Thema Sicherheit eingegangen. Aktuelle Metacomputer-Systeme werden in Kapitel 3 systematisch dargestellt.

2.1 Historische Entwicklung und Motivation

Entwicklung

Metacomputer können als eine Fortsetzung der Entwicklungen von Hochleistungsrechnern und der zunehmenden Vernetzung von Einzelplatzrechnern im Zuge des Wachstums des Internets gesehen werden. Kombiniert man diese Entwicklungen, erhält man als logische Konsequenz einen virtuellen Hochleistungsrechner, der globale verteilte Ressourcen über ein weltweites Netz Anwendern zugänglich macht. Die Art und Weise, diese Zugänge zu den Ressourcen zu ermöglichen, variieren erheblich und stellen die Motivation für die Breite der verschiedenen Entwicklungen dar, die sich in der Anzahl der aktuellen Metacomputing-Systeme widerspiegelt.

Um die Leistung von dedizierten Hochleistungsrechnern durch andere, finanziell ökonomischere Modelle zu ergänzen, stehen am Beginn dieser Entwicklung (1990) Kommunikationenbibliotheken wie PVM [154] , MPI [116] und `nxlib` [147]. Diese Programmiermodelle ermöglichen Entwicklern einer auf mehrere Rechner verteilten Anwendung die einfache Kommunikation durch Basisfunktionen wie `send`, `receive` und den Funktionalitäten von Barrieren, Gruppenbildungen und Broadcast. Bei diesen Bibliotheken wird lediglich die Kommunikation über eine Schnittstelle abgewickelt, Metacomputer bieten zusätzliche Abstraktionsebenen an, die sich

zumeist auf diese Kommunikationsbibliotheken oder auf entsprechende Eigenentwicklungen stützen. Um einen transparenten Zugang zu einem verteilten System zu ermöglichen, sind bei komplexeren Metacomputing-Systemen mittlerweile eine große Anzahl an Diensten verfügbar, die zum einen der Applikation angeboten werden, zum anderen innerhalb des Systems zur Verwaltung benutzt werden. Die Notwendigkeit der meisten dieser Dienste lassen sich aus den Anforderungen aus Abschnitt 2.2 ableiten, deren Verwirklichung dient dann zum Teil der Klassifizierung in Abschnitt 2.3. Aufgrund der Tatsache, dass die meisten Funktionalitäten für verteilte Systeme, wie entfernter Dateizugriff, Nutzen von entfernten Geräten usw, bereits existieren, ist es häufig der Fall, dass zur Nutzung dieser innerhalb einer verteilten Umgebung die Integration und Verwaltung dieser Dienste die Hauptaufgabe des Metacomputing-Systems ist. Die Analogie zwischen einer sequentiellen Anwendung und einem Betriebssystem zu einer verteilten Anwendung und einem Metacomputing-System liegt nahe und wird die zukünftige Entwicklungsrichtung bestimmen.

Der Begriff "Metacomputing" wurde erstmals in diesem Sinne von LARRY SMARR für den nahtlosen Zugang von Arbeitsplatzrechnern zu Supercomputern gebraucht [144]:

> "The computing resources transparently available to the user via this [national] network have been called a metacomputer. The metacomputer is a network of heterogeneous, computational resources linked by software in such a way that they can be used as easily as a personal computer."

Dieser Begriff etablierte sich während der letzten Jahre als Synonym für das Erreichen sehr hoher Rechenleistung durch das Koppeln von Hochleistungsrechnern an verteilten Orten.

Das primäre Ziel von diesen zusammengeschlossenen Superrechnern war das Berechnen der sogenannten "Grand Challenges" [145] im wissenschaftlich-technischen Bereich gewesen, da die hierfür benötigte Rechenleistung von Einzelsystemen oder von Parallelrechnern nicht ausreichte. Inzwischen ist die Breite der möglichen Anwendung für Metacomputer gestiegen, und auch die Terminologie hat sich dieser Entwicklung durch verfeinerte Bedeutungen, die jedoch im wissenschaftlichem Sinn nicht konsequent angewandt wird, angepasst. Siehe hierzu Abschnitt 2.3, die Definitionen halten sich weitgehenst an die Bedeutungen im wissenschaftlichen Kontext und ergänzen sie, wo es notwendig erscheint.

Rechenleistung

Zu Beginn der Entwicklung von Software zur Nutzung von mehreren Arbeitsplatzrechnern als Parallelrechner stand das Entwickeln und Testen von Parallelrechnersoftware im Vordergrund, Produktionsläufe wurden weiterhin auf dedizierten Parallelrechnern vorgenommen. Inzwischen ist es jedoch gängig, diese Verbünde aus

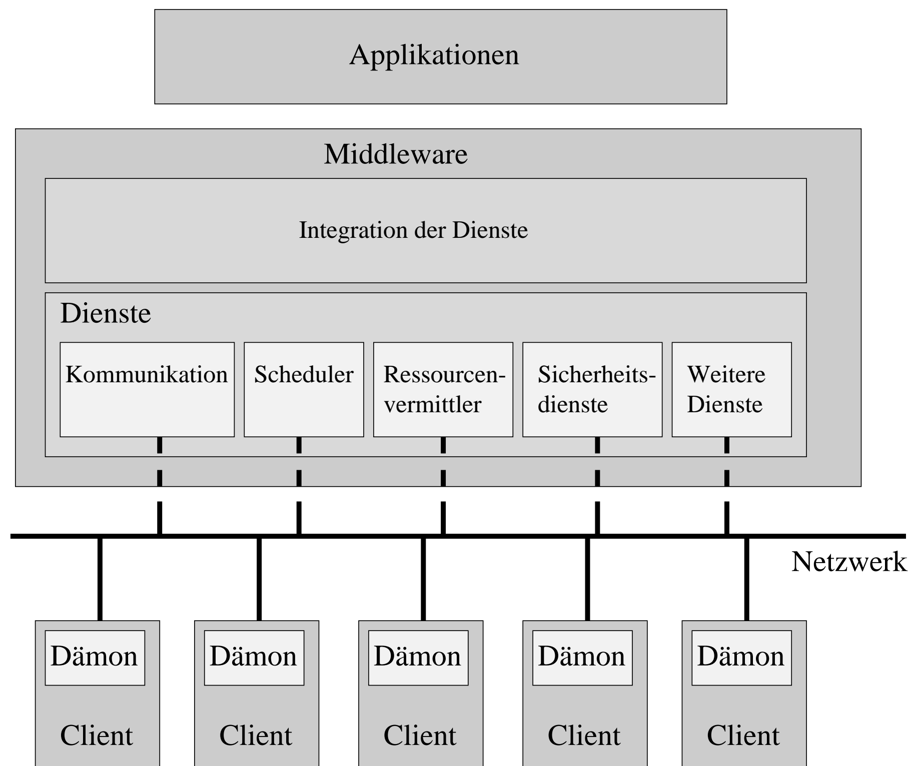

Abbildung 2.1: Allgemeiner Aufbau eines Metacomputers

Arbeitsplatzrechnern als leistungsschwächere, aber ökonomischere Variante eines Hochleistungsrechners zu verwenden[1]. Metacomputer-Architekturen setzen diese Linie auf einer, in Bezug auf die Leistung und die zur Verfügung stehenden Ressourcen, höheren Ebene fort. Sie bieten nicht nur eine Abstraktion der Leistung des Systems, wie es Verbünde aus Arbeitsplatzrechnern dem Benutzer vermitteln, an, sondern erzeugen durch zusätzliche Dienste einen Übergang zu einem vollständigen virtuellen Rechner. Der prinzipielle Aufbau einer Middleware, die zwischen Anwendung und Metacomputer platziert ist und diese Dienste zur Verfügung stellt, ist in Abbildung 2.1 zu sehen.

Bei der Steigerung der Rechenleistung von Parallelrechnern geschieht dies primär durch Erhöhung der Anzahl der vorhandenen Prozessoren und zieht einen Ausbau der darunterliegenden Transportleistung für eine höhere Skalierbarkeit nach sich. Diese Rechenleistung kann durch entsprechende Übersetzer zu einem relativ hohen Anteil genutzt werden[2]. Die Entwicklung für Anwendungsprogramme und Werkzeuge zur effizienten Nutzung durch Anwender hinkt dabei der Hardware-

[1] Neuere, sehr große Installationen von Verbünden aus Arbeitsplatzrechnern übertreffen auch die Leistung von massiv parallelen Hochleistungsrechnern. Ein Überblick kann aus [163] gewonnen werden.

[2] Verhältnis zwischen theoretischer Spitzenleistung und anhaltender Rechenleistung (peak-, sustained performance). Als Beispiel sei die Hitachi SR 8000 - F1/112 des Leibniz Rechenzentrums (LRZ) in München genannt, die bei 896 Prozessoren eine Spitzenleistung von 1344 GFlops, eine

entwicklung traditionell hinterher [109]. Bei der Metacomputer-Entwicklung stellt sich die Lage anders dar: Die weltweit vorhandene Rechenleistung zu steigern ist nicht Ziel der Entwicklung. Angesichts der Anzahl der am Internet angeschlossenen Rechner[3] zusätzlich zu den Superrechnern in Firmen und Instituten ist die theoretisch verfügbare Rechenleistung allen gegenwärtigen Anforderungen gewachsen. Hier gilt es eine Infrastruktur zu entwickeln, um diese Rechenleistung auch zu nutzen. Hierfür kann eine Reihe an Entwicklungen für traditionelle Superrechner und Verbünden von Arbeitsplatzrechnern hilfreich sein (Kommunikationsoptimierungen, Zugriffsmechanismen auf entfernte Ressourcen), aber ein Großteil der vorhandenen Forschung muss um wichtige Punkte erweitert werden, um den neuen Gegebenheiten gerecht zu werden. Die zukünftige Nutzung von verteilten Ressourcen in großem Maßstab wird sich demnach an der Entwicklung von Metacomputing-Umgebungen entscheiden, die trotz der vorhandenen Probleme einen transparenten, fließenden Zugang zu diesen Ressourcen ermöglichen und gleichzeitig den durch paralleles Rechnen möglichen Leistungsgewinn bis zur Applikation transportieren. Die Anforderungen, die für diese Entwicklung nötig sind, sind in Abschnitt 2.2 detailliert beschrieben. Besonders wird auf Merkmale eingegangen werden, die im Vergleich zu herkömmlichen Hochleistungsrechnern eine andere Lösung erfordern.

2.2 Anforderungen für weitverteilte Systeme

Die meisten Herausforderungen, die sich in dem Bereich des parallelen und verteilten Rechnens stellen, sind auch für weitverteilte Systeme von Bedeutung. Auch einige Aspekte aus der Betriebssytementwicklung, wie das Scheduling und der Zugriff auf dynamisch hinzugefügte Peripheriegeräte, lassen sich auf Metacomputer übertragen. Wie sich jedoch aus den folgenden diskutierten Punkten sehen lässt, ergeben sich durch die verteilte Architektur verlagerte Probleme, die mit herkömmlichen Lösungen nur partiell oder unzureichend gelöst werden können.

Zudem wird ein wichtiger Basisdienst einer Middleware sein, Informationen über die Konfiguration eines Systems einzuholen, nach Art dieser Informationen Entscheidungen über die eigene Konfiguration zu treffen und diese an Anwendungen weiterzugeben, beziehungsweise diese über geänderte Konfigurationen zu unterrichten. Dieses notwendige Verhalten eines Metacomputers war bei herkömmlichen Hochleistungsrechnern nicht, und bei lokalen Verbünden von mehreren Rechnern nur in sehr beschränktem Maße (meist in Zusammenhang mit Lastbalanzierung) zu finden. Einzuholende Informationen können hier sein:

- Rechnerarchitektur (Art der CPU, Anzahl, Zwischenspeicherausbau)
- Betriebssystem

anhaltende Leistung von 450 GFlops erreicht (LRZ eigene Messprogramme). Diese Effizienz gilt als gut.

[3] Im Juli 2000 zählte das *Internet Software Consortium* (`http://www.isc.org`) 93 Millionen Rechner, deren Internet Adresse (IP) ein Name zugeordnet war.

- Betriebssystemversion
- Speicherausbau
- Netzwerkbandbreite
- Latenzzeiten
- Kommunikationsprotokolle
- Netztopologie
- Verfügbare Peripheriegeräte
- Zugriffsrechte
- Lastwerte

Im folgenden werden eine Reihe möglicher Anforderungen, die ein System erfüllen kann, diskutiert und mögliche Lösungen aus den vorgestellten Metacomputing-Projekten vorweggenommen.

Fehlertoleranz

In Standard-Rechnerarchitekturen sind nur wenige funktionell unabhängige Komponenten vorhanden (meist Peripheriegeräte, deren Funktionsausfall größtenteils für den restlichen Betrieb ignoriert werden können). Dies führt dazu, dass bei Ausfall einer Komponente (CPU, Speicher, Bussystem) das gesamte System den Betrieb einstellen muss. Fehlertoleranzmechanismen sind hier selten notwendig. Erst bei größeren Systemen bis zu Supercomputern wird dies aufwändig durch Techniken wie Prozessreplikation und transaktionsorientierte Verarbeitung erreicht. Dies ist meistens bei Systemen von kommerziellen Anbietern der Fall, die Datenbanken oder Netzdienste anbieten. Im wissenschaftlich-technischen Bereich ist dies eher selten, hier wird im Fehlerfall die Applikation neu aufgesetzt. Aufgrund der hohen Fehleranfälligkeit bei Metacomputern, bedingt durch die hohe Anzahl an beteiligten Knoten und Netzkomponenten, ist es hier notwendig, ebensolche Maßnahmen zu realisieren, damit langlaufende und aufwändig aufzusetzende Applikationen nicht neu gestartet werden müssen. Demnach gilt es bei fehlertoleranten Systemen zwei Mechanismen zu realisieren: Zum einen müssen Fehler erkannt werden, zum anderen deren Auswirkungen minimal gehalten werden. Siehe hierzu auch Abschnitt 3.1.2.2.

Heterogenität

Die Heterogenität bei weitverteilter Architektur ist auf mehreren Ebenen gegenwärtig. Es existieren verschiedene Geräte, Rechnerhardware, Betriebssysteme und Programmiersprachen, welche unterschiedlich angesprochen werden müssen. Hier bieten sich zwei Lösungen an, die auch gegenwärtig in mehreren Systemen vorhanden sind. Zum einen ist dies eine weitere Zwischenschicht auf den Rechenknoten der jeweiligen Umgebung, auf denen

der lokale Teil der jeweiligen Anwendung aufsitzt. In den meisten Fällen ist dies eine virtuelle Maschine für die Programmiersprache Java, welche den Java Bytecode in Maschinencode übersetzt. Die andere Möglichkeit ist ein zusätzlicher Zwischenschritt, der das Programm auf dem Zielrechner aufsetzt. Hierzu gehört die Übersetzung, Installation und die Zuteilung der entsprechenden Benutzerrechte. Diese Möglichkeit ist entsprechend aufwändiger und benötigt einen Benutzerzugang zu dem Rechner um interaktiv die Installation vornehmen zu können, was die Administration erschwert und oft unmöglich macht.

Dynamische Struktur

Im Unterschied zu fest installierter Hardware, die ihre Konfiguration bezüglich angeschlossener Geräte und installierter Software selten ändert, ist die Architekur eines Metacomputers sehr dynamisch. Auch zur Laufzeit von Applikationen können weitere Rechner zu dem Gesamtsystem hinzugefügt oder daraus entfernt werden. Auch lässt sich auf Rechnern, deren Konfiguration mehrere laufende Applikationen erlaubt, die Last nur mit einer begrenzten zeitlichen Auflösung bestimmen. Dasselbe gilt für angeschlossene Peripheriegeräte. Dieses dynamische Verhalten stellt große Anforderungen an das Ressourcenverwaltungssystem des Metacomputers, sowie im Falle von Architekturveränderungen zur Laufzeit an den Fehlertoleranzmechanismus und das System der Informationsverwaltung.

Vorhersehbare Laufzeit

Trotz der dynamischen Struktur eines Metacomputers ist es wünschenswert, Aussagen über Laufzeit im Vorhinein treffen zu können (Ausnahmen sind in Abschnitt 3.1.1 beschrieben). Solche Vorhersagen, die vor allem bei langlaufenden Anwendungen aus dem wissenschaftlich-technischen Bereich von großer Bedeutung sind, sind bereits bei Parallelrechnern seit geraumer Zeit Forschungsgegenstand und schwer zu bestimmen [138]. Bei variierender Knotenanzahl und unterschiedlicher Rechenkapazitäten in Metacomputern wird dies auch zukünftig eine große Herausforderung bleiben. Ansätze zur Vorhersage findet man in [29], diese werden allerdings nur zur optimalen Prozessverteilung genutzt, nicht zur Laufzeitvorhersage.

Rechner- und Institutsautonomie

Rechner eines Metacomputing-Systems stehen meist nicht unter der Verwaltung eines einzelnen Administrators. Dies impliziert Schwierigkeiten in der Verwaltung des Gesamtsystems. Es können keine vereinheitlichten Sicherheitsbestimmungen für alle Knoten bestimmt werden. Diese Verwaltung muss auf Knotenseite erfolgen. Auch eine einheitliche Installation von Hard- und Software kann nicht angenommen werden. Dies erweitert die dynamische Struktur um eine weitere Komponente, die berücksichtigt werden muss. Auf bestimmte, institutsabhängige Besonderheiten, wie beispielsweise Firewalls oder Gateways, muss ebenfalls eingegangen werden können. Da dies sicherheitsrelevante Gebiete eines Instituts sind, wird hier in der Regel eine direkte

Zusammenarbeit der Administratoren notwendig sein. Kommunikation über vorher vereinbarte getunnelte Verbindungen können hier eine Lösung sein.

Performanz

Die hier aufgezählten Anforderungen benötigen einen sehr hohen Implementierungs- und Verwaltungsaufwand bei der Verwirklichung einer transparenten Metacomputing-Umgebung. Werden alle Punkte berücksichtigt und versieht man die Middleware mit zusätzlicher Redundanz zur Fehlertoleranz und den notwendigen Protokollen, um die Sicherheit und Integrität der Software zu wahren, ergeben sich signifikante Leistungseinbußen durch die Verwaltung dieser Dienste. Um eines der Hauptziele, Leistungssteigerung der Anwendung, zu erreichen, sind hier Optimierungen auf Ebene der Middleware unerlässlich, damit die potenzielle Leistung einer solchen Umgebung bis zur Anwendung transportiert werden kann. Eine solche Optimierung kann sein, die Verwaltung vor Aufsetzen der Applikation weitgehenst abzuschliessen, um den Protokollaufwand zur Laufzeit möglichst gering zu halten.

Portabilität

Aufgrund der Heterogenität der in obigem Punkt aufgezählten Komponenten sollten sowohl die eingesetzte Middleware als auch die Anwendung selber portabel sein. Dies ist unter herkömmlichen Programmiersprachen sehr aufwändig und benötigt hohen Entwicklungsaufwand. Mit der plattformübergreifenden Programmiersprache Java und der darunter liegenden Java virtuellen Maschine (*JVM*) wird größtmögliche Portabilität erreicht, ohne auf Basisfunktionalitäten zum jeweiligen Betriebssystem verzichten zu müssen.

Sicherheit

Einer der bedeutendsten Faktoren, besonders im Hinblick auf die Zukunft des Internets, ist bei global kommunizierenden Programmen die Sicherheit. Hierauf wird detailliert in Abschnitt 2.5 eingegangen.

Skalierbarkeit

Auf die Bedeutung der Skalierbarkeit in Zusammenhang mit Metacomputern wird in der folgenden Definition eingegangen:

Definition 2.1 *Skalierbarkeit*

Skalierbarkeit heißt die Fähigkeit eines Systems, bei Variation eines quantitativen Parameters die wesentlichen Systemeigenschaften aufrecht zu erhalten. □

Bei Parallelrechnern bedeutet dies, bei Hinzunahme von Komponenten wird die Effizienz (Def. 2.12) aufrecht erhalten. Unter Komponenten versteht man in diesem Zusammenhang meist Prozessoren oder allgemein Knoten (siehe Definition 2.13). Innerhalb eines Systems können aber auch viele andere Ressourcen können einen Flaschenhals bilden, der die Skalierbarkeit in Bezug auf die Leistung beschränkt. Dies kann die Netzverbindung, Dateizugriff eines einzelnen Knotens oder auch eine generelle Überlastung eines Knotens

bei einem zentralistischen Ansatz sein. Gründe für einen Flaschenhals sind meistens Master/Client Architekturen, bei denen zu einem bestimmten Zeitpunkt die Anzahl der Clients die Netz- oder Verwaltungskapazitäten des Masters übersteigen. Hochskalierbare Metacomputer-Systeme sind daher verteilt administriert und ohne zentrale Komponente. Dies erhöht jedoch die Komplexität des Systems und benötigt zusätzlichen Aufwand, meistens in Form teurer Kommunikation.

Lastverteilung

Die Lastbalanzierung nimmt in Metacomputing-Systemen einen anderen Stellenwert ein im Vergleich zu sonstigen verteilten Architekturen. Mechanismen zur Lastverteilung werden bei Metacomputern vor allem dann eingesetzt, wenn größere Lastungleichheiten vermieden werden sollen [114], da der langsamste Prozess die Gesamtlaufzeit einer Metacomputing-Anwendung bestimmt. Lastverteilungsalgorithmen sind aufgrund ihrer Komplexität ein Gebiet intensiver Forschung, die verschiedene Ziele (hoher Durchsatz, Fehlertoleranz, Laufzeitminimierung) verfolgen können. Im Zusammenhang mit Metacomputing bedeutet dies jedoch vornehmlich, den schnellsten beteiligten Knoten mit dem rechenaufwändigsten Prozess zu versorgen, da im Normalfall genügend viele Knoten zur Verfügung stehen. Im weiteren Verlauf dieser Arbeit wird Lastverteilung unter diesem vereinfachten Aspekt betrachtet.

Im Falle des vorgestellten *LsD-Systems* wird von der Annahme ausgegangen, dass teure Kommunikation eine laufzeitentscheidende Bedeutung hat, daher wird der Prozess mit der meisten Kommunikation auf den Knoten mit der besten Netzanbindung gebracht, und damit ein Lastausgleich im Sinne von kurzer Gesamtlaufzeit erreicht.

Vertraulichkeit

Die Vertraulichkeit, ein Teilaspekt der Sicherheit, muss bei der Kommunikation in öffentlichen Netzen gewährleistet sein. Daten, die der Geheimhaltung bedürfen, wie beispielsweise die Konstruktionsdaten der in Kapitel 5 vorgestellten Firma, müssen dabei für die Übertragung verschlüsselt werden. Entsprechende kryptographische Verfahren sind verfügbar und lassen sich problemlos, teilweise für die Anwendung transparent, integrieren.

Eine weitere entscheidende Rolle in der Architektur eines Metacomputers, um die oben genannten Anforderungen umzusetzen, spielen Techniken zur Informationsgewinnung, Mechanismen die Informationen einholen über die Art der angeschlossenen Ressourcen (Peripheriegeräte), Rechnerhardware (Anzahl der Prozessoren, Geschwindigkeit der Prozessoren, aktuelle Last, Größe des Hauptspeichers), Netzverbindungen (Bandbreite, Latenz, Robustheit der Verbindung) sowie der Konfiguration und Verfügbarkeit der angeschlossenen Gastrechner.

Diese Informationen regelmäßig und zuverlässig einzuholen und dynamisch zur Laufzeit Applikationen zu vermitteln, erfordert eine sehr komplexe und flexible Schicht. Der vorgestellte Ansatz in Kapitel 4 verbirgt diese Information durch das Konzept von prozessverarbeitenden Einheiten, mit dem physikalisch vorhandene

Prozessoren verborgen werden (siehe hierzu auch Definition eines *LsD* Knotens, Definition 2.14).

Andere Systeme implementieren hier aufwändige Mechanismen, um diese Information einzuholen und auf dem aktuellen Stand zu halten (siehe Abschnitt 3.1.3, 3.1.4 und 3.1.5) und um die Applikation von den physikalischen Gegebenheiten der angeschlossenen Rechner abzukapseln.

2.3 Klassifizierung von Metacomputern

2.3.1 Modelle verteilten Rechnens

Eine allgemeine Klassifikation, die ein System in die Metacomputing-Klasse einordnet oder es davon trennt, kann nicht ohne Einschränkungen aufgestellt werden. Dies liegt zum einen an der großen Breite der Funktionen, die von verschiedenen Systemen angeboten werden, zum anderen an der Tatsache, dass die Anzahl der Knoten oft einen Metacomputer ausmachen, dies aber nicht für eine Klassifizierung hinreichend ist. Weiterhin gilt umgekehrt, dass sehr große Systeme mit sehr hohen Knotenanzahlen, wie in Abschnitt 3.1.1 beschrieben, nicht mit den Anforderung aus 2.2 schritthalten, aber trotzdem in der Literatur als Metacomputer bezeichnet werden.

Es existiert eine Hierarchie aus drei Ebenen, die sich im Laufe der Entwicklung von parallelen Anwendungen herausgestellt hat:

- Parallele Programmierumgebungen
- Objektorientierte verteilte Systeme
- Metacomputing-Systeme

Während die Abstraktionsebene hierbei ansteigt, wird die spezifische Leistung pro Knoten durch Einführung neuer Zwischenschichten und Protokollebenen geringer. Eine Implementierung von Metacomputing-Funktionalitäten auf tieferen Ebenen, beispielsweise auf Betriebssystemebene oder in der Hardware, ist gegenwärtig nicht angedacht. Diese Möglichkeit könnte jedoch in Zukunft eine effiziente Methode sein, Leistungsverluste durch Zwischenschichten zu verhindern. Parallele Programmierumgebungen stellen den Anwendungen Funktionen zur Verfügung, die es ihnen ermöglichen, mit verteilten Komponenten zu kommunizieren. Hierbei wird die Netzwerkschicht vor der Anwendung verborgen, es ist jedoch keine Schicht vorhanden, die weitere Dienste oder Informationen anbietet, die parallele Programmierbibliothek befindet sich auf der Ebene des Betriebssystems (siehe Abbildung 2.2). Die angebotenen Funktionalitäten bestehen meist aus `send` und `receive` zum Versenden von Nachrichten, `barrier` zur Synchronisation und `broadcast` für Funktionen zur Gruppenbildung. Diese Implementierungen setzen in der Regel auf TCP/IP

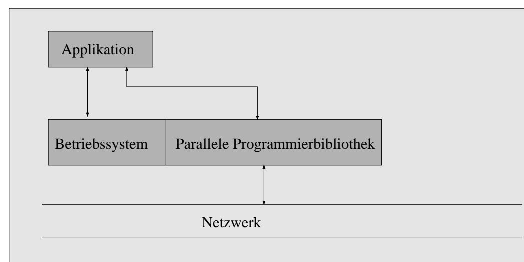

Abbildung 2.2: Parallele Programmierbibliotheken

oder UDP [151] auf. Diese Umgebungen sind für parallele Programmierung universal einsetzbar und sehr effizient, allerdings auch aufwändig in der Entwicklung und (Programmier-) fehleranfällig.

Die zweite Ebene bilden objektorientierte, verteilte Umgebungen. Diese Schicht, die auch als Middleware bezeichnet wird, sorgt für Lokalitätstransparenz von Teilen der Anwendungen, die in Objekten gekapselt sind. Hierbei werden Stellvertreter der Objekte auf den Clients installiert, die über eine lokale Vermittlungsinstanz die Kommunikation zu dem Zielobjekt herstellen. Die physikalische Adresse des Zielobjektes wird dabei von einem Broker festgestellt (Abbildung 2.3).

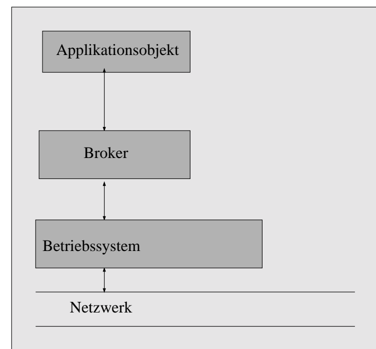

Abbildung 2.3: Objektorientiertes Modell

Durch dieses Modell läuft eine Anwendung in einem Orts- und Betriebssystem unabhängigen Umfeld, die explizite Kommunikation wird dabei verborgen, der Entwickler spricht im Sinne der Objektorientierung nur die Objekte an. Der dabei entstehende zusätzliche Aufwand bei der Kommunikation und der Verwaltung der Objekte hat Leistungseinbußen zur Folge und daher bisher dazu geführt, dass Anwendungen des wissenschaftlich-technischen Hochleistungsrechnens selten solche Modelle verwenden. Hierfür sind in der Regel die eingesetzten Ressourcen zu gering und es fehlen Dienste zur Prozessverwaltung. Auch kann die verborgene Kommunikation zu ungewollten Engpässen führen, da dies bei wissenschaftlich-technischen

Anwendungen oft der laufzeitbeschränkende Faktor ist. Als Kommunikationswerkzeug werden parallele Programmierbibliotheken verwendet.

Metacomputing-Modelle (Abbildung 2.4) haben die höchste Abstraktionsstufe bezüglich den darunterliegenden Schichten. Hier wird versucht, alle Ressourcen, die von parallelen Anwendungen benötigt werden, unter einer Schicht zu verbergen. Dies erlaubt die Nutzung von entfernten Rechnern und dedizierten Supercomputern und der daran angeschlossenen Peripheriegeräte durch die Bereitstellung der dazu notwendigen Dienste aus Abschnitt 2.2. Durch die hohe Komplexität solcher Systeme sind jedoch die zu bewältigenden Probleme noch teilweise ungelöst oder noch nicht in einem einzelnen System integriert. Hier gilt es nach dem Stand der Forschung, den benötigten Anwendungen und den daraus resultierenden Anforderungen ein optimales System zu bestimmen.

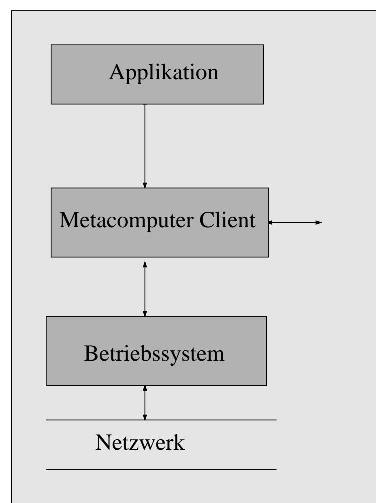

Abbildung 2.4: Metacomputer-Modell

Metacomputing-Systeme nutzen sowohl objektorientierte verteilte Systeme als auch direkt parallele Programmierumgebungen, um Anwendungen auf die zur Verfügung stehenden Ressourcen zu verteilen. Hier wird der Anwendung eine zusätzliche Schicht zur Kapselung angeboten, die es ermöglicht dynamische Ressourcen, die erst zur Laufzeit bestimmt werden, zu nutzen. Als entscheidendes Kriterium gilt hier die Fähigkeit, einen Mechanismus zur Prozessverteilung und entfernten Prozessausführung anzubieten.

Darüber hinaus lassen sich bei Metacomputer-Diensten aktive und passive Dienste unterscheiden (in Abbildung 2.5 illustriert). Aktive Dienste führen notwendige Aktionen ($A(D_2, R)$) aus oder veranlassen sie direkt. Das können Aktionen zur Prozessmigration, Ressourcenallokation oder zur Informationsgewinnung sein. Passive Dienste werden von der Anwendung benutzt, um selber Aktionen aus-

zuführen ($A(I,R)$). Sie kann dabei Informationen von dem Metacomputer einfordern ((I,R)) oder dessen Dienste zur Ausführung verwenden [4].

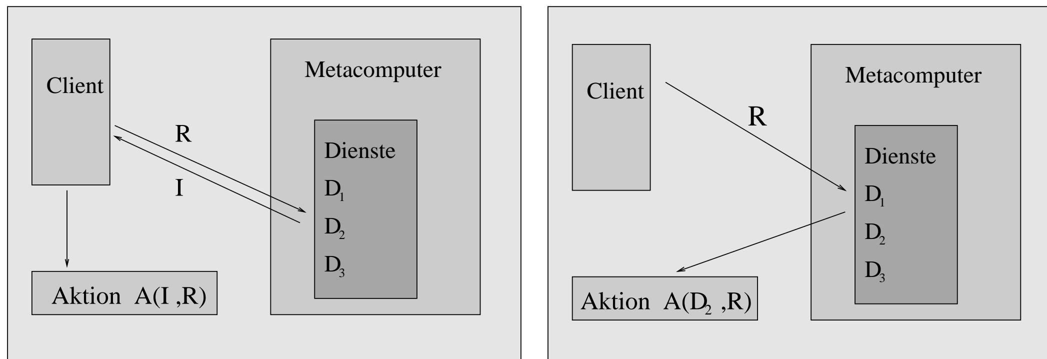

Abbildung 2.5: Passive (links) und aktive (rechts) Metacomputing-Mechanismen

2.3.2 Unterscheidungskriterien von Metacomputern

Unabhängig von der Struktur der Anwendung lassen sich Metacomputing-Systeme in Client/Server Systeme und in dezentrale, verteilte Systeme unterteilen.

Client/Server Systeme

Diese Systeme realisieren die Verwaltung der Clients an einem zentralen Punkt, dem Server. Dieser kennt alle Clients und nimmt Anfragen zur Vermittlung der Ressourcen entgegen. Oft wird auch ein eigener Prozess für die Zuteilung von Anfragen auf freie Knoten implementiert, dieser *Broker* ist meist auf dem Rechner der zentralen Komponente angesiedelt, doch auch eine Trennung ist hier möglich. Hauptmerkmal ist die zentrale Verwaltung der einzelnen Dienste. Vertreter sind in 3.1.1 und 3.1.2 beschrieben.

Diese Systeme haben eine einfache Struktur und sind bezüglich Fehleranfälligkeit bei der Implementierung und Abwicklung der Kommunikation robuster, haben jedoch, wie alle Client/Server Systeme eine höhere Ausfallwahrscheinlichkeit aufgrund eines *Single Point of Failure* bei dem Server. Zudem wird die Skalierbarkeit des Gesamtsystems von der Leistungsfähigkeit des Servers bestimmt.

Verteilte Systeme

Verteilte Systeme haben keine zentralen Komponenten, die Dienste koordinieren oder Informationen sammeln. Dies erfordert, je nach Anzahl und Vollständigkeit

[4] Analog zu dieser Unterteilung unterscheidet man auf dem Gebiet der Lastverteilung anwendungsintegrierte Lastverteilung und globale Lastverteilung

der Realisierung der Anforderungen, eine komplexe Struktur, welche meistens dadurch erreicht wird, dass die Dienste in modulare Einheiten unterteilt werden, und diese auf den Clients ausgeführt werden. Durch den Informationsaustausch untereinander wird hier eine Gesamtsicht auf das System gebildet, das dann als Informationsgrundlage für die einzelnen Dienste genutzt wird. Beispielhafte Vertreter sind in Abschnitt 3.1.4 und 3.1.5 genannt.

Es lassen sich aus dieser grundlegenden Unterscheidung durch Architekturmerkmale weitere Kriterien ableiten, die sich aus den oben genannten implizieren und daher die vorhandenen Systeme oft, aber nicht zwangsläufig, in dieselben Gruppen einteilen.

Hierzu sind einigen Definitionen notwendig:

Definition 2.2 *Verteiltes System*

> Ein Rechensystem heißt verteilt, wenn in seinem Betrieb der Gesamtzustand des Systems nicht einhellig erkennbar ist. □

Definition 2.3 *Weitverteiltes System*

> Ein verteiltes System ist weitverteilt, wenn die Zeitspannen einer nennenswerten Änderung innerhalb einer Komponente und die Laufzeit zwischen den Komponenten sich erheblich unterscheiden. □

Von entscheidender Bedeutung ist hierbei, dass ein verteiltes System keinen Dienst zur Prozessverteilung gemäß Definition 2.15 benötigt. Dies legt die Grundlage für alle hier besprochenen Systeme fest, die alle dieser Definition entsprechen. Eine exaktere Einordnung ermöglichen folgende Definitionen:

Definition 2.4 *Parallele Anwendung*

> Eine Menge von parallel ausgeführten Prozessen $P = \{p_1, \ldots, p_n\}$ mit $n > 1$, die gemeinsam an einer Aufgabe arbeiten, sind eine parallele Anwendung. □

Definition 2.5 *Virtueller Hochleistungsrechner*

> Ein virtueller Hochleistungsrechner besteht aus mehreren, an verschiedenen Orten installierten Hochleistungsrechnern, die über ein Netzwerk gekoppelt sind, wobei eine Software die Illusion eines aus Benutzersicht einzelnen Rechners realisiert. □

Definition 2.6 *Metacomputer*

> Eine Menge von selbstständigen, möglicherweise verschiedenartigen, Computern, die durch ein Softwaresystem Benutzeranwendungen wie ein einzelner Rechner ausführen können, heisst Metacomputer. □

Die Ressourcen dieser Computer sollen bei Bearbeitung des Benutzerauftrages möglichst günstig verwendet werden.

Definition 2.7 *Webcomputer*

> Ein Webcomputer ist definiert als eine sehr hohe Anzahl an, möglicherweise verschiedenartigen, Computern, die, zum Teil zeitlich begrenzt, an ein Netzwerk angebunden sind und durch eine Software gemeinsam eine verteilte Anwendung ausführen. □

Allen diesen Definitionen ist eine Softwareschicht gemeinsam, die die lokalen Ressourcen nutzt und deren Funktionalitäten der verteilten Anwendung zur Verfügung stellt. Diese Softwareschicht ist der Hauptpunkt dieser Diskussion und gegenwärtig der Forschungsgegenstand bei der Nutzung verteilter Ressourcen.

Virtuelle Hochleistungsrechner sind meist ein Verbund aus Parallelrechnern, die in Instituten oder Hochschulen installiert sind und vornehmlich im Bereich von wissenschaftlich-technischen Anwendungen benutzt werden [42, 132, 118].

Um komplexe Metacomputer von Projekten mit sehr ausgeprägter und einfacher Client/Server Architektur zu unterscheiden (Abschnitt 3.1.1), wird Definition 2.9 eingeführt, die solche Projekte, deren Rechenkapazität zu großen Teilen aus Heimrechnern von Internetbenutzern auf freiwilliger Basis besteht [135, 134] von ersteren trennt.

Analog zu den obigen Architekturdefinitionen gilt:

Definition 2.8 *Metacomputing*

> Metacomputing ist das Rechnen von Benutzeranwendungen auf Metacomputern. □

Definition 2.9 *Webcomputing*

> Webcomputing ist das Rechnen von Benutzeranwendungen auf Webcomputern. □

Eine weitere Architektur, der in jüngster Zeit zum Gegenstand der Forschung geworden ist, hat die Transparenz eines Metacomputers im Fokus. Hier soll nicht nur die Leistung des Gesamtsystems im Vordergrund stehen, sondern seine nahtlose Benutzung. Analog zu der Verfügbarkeit Strom aus dem Elektrizitätsnetz sollen durch diese Architekturen, *Computational Grids*, zukünftig verteilte Anwendungen ohne Interaktion durch den Benutzer global verteilt auf Hochleistungsrechnern berechnet werden können.

Nach FOSTER und KESSELMAN ist ein *Computational Grid* wie folgt definiert [55]:

Definition 2.10 *Computational Grid*

> Ein *Computational Grid* ist eine Infrastruktur aus Hardware und Software, die eine Steigerung der Rechenleistung ermöglicht, indem sie konsistenten, zuverlässigen, allgegenwärtigen und kostengünstigen Zugang zu Hochleistungsrechenanlagen bietet. □

Dies kann aufgrund der Forderung eines vollkommen nahtlosen Übergangs (die Infrastruktur vermittelt den Eindruck eines einzelnen Rechners, den der Benutzer bedient) zu verteilten Ressourcen als eine Weiterentwicklung eines Metacomputers gelten.

Definition 2.11 *starke/schwache Parallelität*

> Die Parallelität ist ein Maß für die Anzahl von CPU Operationen, die abgearbeitet werden, ohne in Kommunikation überzugehen. Starke Parallelität heißt eine hohe Anzahl an CPU Operationen, schwache Parallelität wenig CPU Operationen bis Kommunikation stattfindet.
>
> Ist starke Parallelität besonders ausgeprägt, spricht man auch von trivialer Parallelität. □

Alternativ dazu wird auch von der Granularität der Parallelität gesprochen. Grobe Granularität entspricht einer starken Parallelität, feine Granularität entspricht schwacher Parallelität.

Nun lassen sich aus den Modellen für verteiltes Rechnen Implikationen ableiten, die eine weitere Klassifizierung ermöglichen.

Anwendungen mit starker/schwacher Parallelität

Anwendungen mit starker Parallelität, die sich in unabhängige Teilaufgaben mit langer Rechenzeit unterteilen lassen, eignen sich für Client/Server Systeme und für Webcomputer Systeme. Diese Architekturen unterstützen oft nur indirekte Kommunikation über den Server oder durch aufwändige Vermittlungsmechanismen, um den Rechner des Kommunikationspartners ausfindig zu machen. Webcomputer unterstützen in der Regel keine Client zu Client Kommunikation.

Anwendungen mit schwacher Parallelität, deren Teilaufgaben oft miteinander kommunizieren müssen, sind für Client/Server Systeme meist nicht geeignet, da die Kommunikationszeit die Gesamtlaufzeit verlängert. Webcomputer schließen diese Anwendung gänzlich aus.

Hierfür sind Architekturen notwendig, die eine schnelle Punkt zu Punkt Kommunikation erlauben.

Anwendungen des wissenschaftlich-technischen Rechnens gehören in der Regel in diesen Bereich, da hier die Randgebiete der zu berechnenden Gitter, Matrizen oder der entsprechenden Partitionierung zwischen den jeweiligen Knoten ausgetauscht werden müssen.

Hier sind nur Metacomputer mit expliziter Punkt zu Punkt Kommunikation geeignet.

Diese Einteilungen sind in Tabelle 2.1 zusammengefasst. Eine entsprechende Tabelle mit den untersuchten Systemen findet sich in Abschnitt 3.1.

Weiterhin können noch folgende bedeutende Merkmale als Unterscheidungskriterien dienen, die sich jedoch nicht als orthogonale Klassifizierungsmerkmale eignen,

System	Parallelität	Anwendung
Metacomputer	schwache Parallelität	universelle Anwendungen
Client/Server Systeme	starke Parallelität	Client/Server Anwendungen
Webcomputer	starke Parallelität	Anwendungen mit trivialer Parallelität

Tabelle 2.1: Zusammenhänge der Klassifizierungsmerkmale

da die Breite der verfügbaren Systeme und der damit angebotenen Dienste keine klare und eindeutige Unterscheidung ermöglichen:

- Einfache Verteilung von Rechenlast oder von Prozessen.
- Reaktive Verteilung von Rechenlast oder von Prozessen.
- Realisieren von höheren Diensten zur eigenen Verwaltung.
- Realisieren von Diensten für die Anwendung.
- Realisieren von Diensten ohne Prozessverteilung.

2.3.3 Terminologie im Bereich Metacomputing

In diesem Abschnitt werden einige technische Grundlagen bezüglich der Terminologie, aufbauend auf den vorherigen Definitionen zum verteilten Rechnen, bei verteilten System festgelegt. Vorausgreifend auf Kapitel 4 werden auch dort benötigte und verwendete Begriffe definiert, diese sind in dem jeweiligen Kontext namentlich hervorgehoben und dementsprechend zu verstehen. Dieses gilt es bei Begriffen mit breiterer Bedeutung zu berücksichtigen.

Ein entscheidender Begriff bei der Einschätzung der Leistungsfähigkeit einer parallelen Anwendung ist die Effizienz, sie gilt als Maß für die Auslastung der Prozessoren.

Definition 2.12 *Effizienz*

Sei A eine parallele Anwendung, die auf einem Prozessor in der Zeit T_1, und auf N gleichen Prozessoren in der Zeit T_N ausgeführt wird. Dann beträgt die Effizienz E der parallelen Anwendung:

$$\text{Effizienz} \qquad E = \frac{\frac{T_1}{T_N}}{N}$$

□

Der Geschwindigkeitszuwachs (*Speedup*) S, mit $S = \frac{T_1}{T_n}$, wird oft auch relativ zu der Zeit des besten sequentiellen Algorithmus, T_s, angegeben, mit $S = \frac{T_s}{T_n}$. Dies erscheint in diesem Zusammenhang, bei den Problemgrößen beziehungsweise bei

den Laufzeiten von Metacomputing-Umgebungen nicht sinnvoll, da Anwendungen für Metacomputer aufgrund ihrer hohen Ressourcenanforderungen oft nicht auf sequentiellen Rechenanlagen berechnet werden können. Im Falle der Messungen aus Kapitel 4 werden als Referenzwerte Ausführungszeiten von parallelen Programmen (dort ohne Optimierungen) verwendet, da hier der Erfolg der Optimierung eines parallelen Programmes validiert werden soll. Dort werden ebenfalls die Begriffe Geschwindigkeitszuwachs und Speedup verwendet.

Während bei Parallelrechnern und bei Verbünden aus Arbeitsplatzrechnern die Bestimmung der parallelen Ausführungseinheiten im Normalfall mit der Anzahl der Prozessoren im Gesamtsystem identisch ist, ist dies bei Metacomputern nicht der Fall. Existieren im System Rechner, die über mehrere Prozessoren verfügen, was in der Regel der Fall sein wird, ist abzuwägen, welche Einheit als Maß für die parallele Ausführung verwendet wird. Hierzu wird der Begriff der Prozessoren durch Knoten ersetzt.

Definition 2.13 *Knoten*

> Ein Knoten innerhalb eines Metacomputers ist eine abstrakte Maschine, die einen Prozess einer Applikation bearbeiten kann. □

Hieraus folgt direkt die Definition eines Knotens innerhalb des *LsD-Systems*:

Definition 2.14 *LsD-System Knoten*

> Ein *LsD-System* Knoten ist ein Knoten, die eine Java Virtuelle Maschine ausführen kann. □

Mit dieser Definition gilt eine $m : n$ Beziehung von *LsD-System* Knoten zu Prozessoren. Die Anwendung auf einem *LsD-System* Knoten kann mit mehreren Prozessoren abgearbeitet werden, in diesem Fall übernimmt das Betriebssystem als Zwischenschicht die Verteilung der Instruktionen auf die Prozessoren, der unwahrscheinlichere und ungünstigere Fall ist die Platzierung von mehreren *LsD-System* Knoten auf einen Prozessor.

Definition 2.15 *Prozess*

> Ein Prozess ist eine Instanz eines Programms, welches von einem Betriebssystem ausgeführt wird (vgl. [149]). □

Ein Prozess ist demnach per definitionem nicht an einen Prozessor gebunden. Es wird jedoch in dem Rahmen von Metacomputern davon ausgegangen, dass ein Prozess genau einem Prozessor zugeordnet ist. Im allgemeinen Fall wird ansonsten von Knoten die Rede sein.

Weiterhin werden die im Zusammenhang mit dem *LsD-System* wichtigen Begriffe von Threads und Jobs bestimmt.

Definition 2.16 *Thread*

> Ein Thread ist ein leichtgewichtiger Prozess, der seinen Adressraum mit anderen Threads teilen kann. □

Definition 2.17 *Job*

> Ein Job innerhalb des *LsD-Systems* ist ein Java-Thread, der ortsunabhängig von einer Java Virtuellen Maschine ausgeführt werden kann. □

Im Rahmen des *LsD-Systems* wird zudem von *LsD* Prozessen die Rede sein, da hier eine Unterscheidung zwischen einem von der Anwendung übergebenen Objekt zur Ausführung (Job) und von demselben Objekt in der Ausführung nötig ist. Zwischen diesen beiden Zuständen wird eine Konvertierung vorgenommen, um eine Migration der Objekte und deren Ausführung zu unterstützen.

Die Definition von Thread ist der geläufigen Definition aus der Literatur entnommen [164]. Aufgrund der Tatsache, dass keine einheitliche Verwendung für den Begriff Job existiert, wird hier davon ausgegangen, dass obige Definition im Rahmen des *LsD-Systems* gültig ist. Dies ist nötig, da ein *LsD-System* Job nur mit einer Java Virtuellen Maschine in Zusammenhang steht. Damit ist eine Abstrahierung von Prozessoren und Threads erreicht, die in diesem Fall die Semantik eines Jobs ausreichend beschreibt.

2.4 Verbreitete Techniken zur verteilten Ausführung

Techniken zur verteilten Ausführung, die von Metacomputern benutzt werden, erstrecken sich über alle Bereiche des parallelen und verteilten Rechnens.

Daher wird hier zunächst eine Anzahl von Diensten benannt, die in Metacomputern realisiert sind und sich im wesentlichen an die in Abschnitt 2.2 formulierten Anforderungen halten. Anschließend wird eine Auswahl an grundlegenden Techniken behandelt. Hierbei wird nicht auf integrierte Dienste eingegangen, diese werden in Abschnitt 3 bei den jeweiligen Systemen besprochen.

2.4.1 Dienste

Die von Metacomputing-Systemen implementierten Dienste sind zum großen Teil bereits in anderen Umgebungen des verteilten oder parallelen Rechnens zu finden. Diese Methoden sind teilweise in Metacomputer-Systemen in angepasster Form (beispielsweise *Cilk* in *Atlas*, GSS in *Globus*, oder in schwächerer Form MATLAB in *NetSolve* (siehe jeweils dort, Abschnitt 3.1)) integriert, werden von diesen als externe Dienste angeboten, oder werden intern von diesen benutzt.

Am ausgeprägtesten ist der “Metacomputer als Dienstleister” in dem *Globus* System. Doch nahezu alle Systeme stellen der Anwendung oder dem System selber

Mechanismen zur verteilten Ausführung zur Verfügung, die hier untersucht werden sollen.

Aufgrund der Komplexität der folgenden Dienste sind deren Funktionen in der Regel nicht orthogonal und es ergeben sich dabei Überschneidungen.

Prozessverteilung
Dieser Dienst übernimmt eine der Grundfunktionen von Metacomputern, die Verteilung von Last auf mehrere Knoten. Dies kann implizit durch das Erzeugen von Prozessen, Threads oder Funktionen in der Anwendung geschehen, die dann für die Anwendung transparent verteilt werden, oder explizit durch einen Aufruf zur Verteilung in der Anwendung geschehen. Wie auch bei der Kommunikation bietet der implizite Ansatz eine höhere Abstraktionsebene an, der den Entwicklungsaufwand reduzieren kann. Zusätzlich zur Verteilung können hier auch Optimierungsmethoden angewandt werden, die je nach Anwendung geeignete Zielarchitekturen auswählen.

Kommunikation
Fast alle Metacomputer-Architekturen bieten Kommunikationsdienste an. Diese können, analog zur Prozessverteilung, implizit oder explizit sein. Im wissenschaftlich-technischen Rechnen sind explizite Kommunikationsaufrufe wegen der Geschwindigkeitsvorteile üblich, doch auch hier gibt es Bestrebungen, objektorientierte Ansätze ohne Leistungsverlust einzusetzen [174, 173, 111]. Bei impliziten Kommunikationsmodellen wird im allgemeinen kein Dienst als solcher angeboten, die Kommunikation wird hier innerhalb von Methodenzugriffen auf entfernte Objekte abgewickelt.

Lastverteilung
Die Aufgaben der Lastverteilung oder Lastbalanzierung[5] erfordern innerhalb von Metacomputing-Umgebungen eine gleichmäßige Verteilung von Prozessen auf Knoten (siehe hierzu auch Bemerkung zur Lastverteilung in 2.2). Es sollte dabei gelten, dass kein Knoten frei sein darf, wenn ein Auftrag zur Berechnung bereit steht (nach [114]). Die Lastverteilung sollte bei Metacomputer-Systemen mit sehr hoher Knotenanzahl unproblematisch sein, da pro Knoten ein Prozess zugeordnet werden kann. Als zusätzliche Optimierung können, aufgrund der möglichen unterschiedlichen Leistungen verschiedener Knoten, langlaufende Teilaufgaben an Prozessoren mit hoher Leistung gebunden werden.

Prozessmigration
Die Prozessmigration bietet durch Sicherungspunkterzeugung [148] oder im Falle der Java Virtuellen Maschine (JVM) durch persistente Speicherung von Threads die Möglichkeit, Prozesse auf andere Knoten zu transferieren. Die Gründe hierfür können lokal nicht verfügbare Ressourcen oder der Spezialfall dessen, eine Lastungleichheit, sein.

[5] Bei der Lastbalanzierung wird im Unterschied zur Lastverteilung ein zusätzliches Kriterium zur Balanzierung der Last gefordert.

Informationsgewinnung

Metacomputer-Systeme benötigen einen Mechanismus, der den Zustand der verteilten Ressourcen kennt und diesen anderen Diensten oder der Anwendung vermitteln kann. Da komplexe Metacomputer-Systeme in der Regel aufgrund unzureichender Skalierbarkeit über keine zentrale Komponente zur Informationsgewinnung verfügen, müssen diese in der Lage sein, neue Komponenten zur Laufzeit zu erkennen und die diesbezügliche Informationen zu bekannten Komponenten vermitteln. Die in Frage kommenden Informationen sind in Abschnitt 2.2 benannt.

Sicherheit

Die Dienste zur Sicherheit umfassen Identifikation, Authentifikation und sichere Kommunikation, siehe hierzu Abschnitt 2.5.

Visualisierung

Visualisierungsdienste verfügen über die Möglichkeit, entfernte Daten zu aktuellen Berechnungen zur Laufzeit der Anwendung abzugreifen und diese zu einer entsprechend ausgestatteten lokalen Komponente zu senden und darzustellen. Dies können Berechnungsergebnisse der Anwendung sein, der Programmablauf oder auch der Systemzustand des Metacomputers. Aufgrund der hohen Anforderungen an die Bandbreite für visuelle Informationen sind hier bezüglich der Platzierung der beteiligten Prozesse entsprechende Informationen zur Netztopologie zu berücksichtigen.

Beobachtung

Integrierte Dienste zur Beobachtung sowohl der Anwendung als auch des Metacomputer-Systems ermöglichen in komplexen verteilten Umgebungen das Analysieren von Laufzeitverhalten, Fehlern und Leistungsengpässen. Angesichts der hohen Fehleranfälligkeit von parallelen Programmen und der hohen Entwicklungskosten kann dieser Aspekt bei der parallelen Programmierung in heutigen Systemen als unzureichend bezeichnet werden. Angesichts der Tatsache, dass sich diese Entwicklungsprobleme bei Metacomputern verstärken werden, sind hier hohe Anforderungen an die Werkzeuge zur Beobachtung und zur Manipulation der Anwendung nötig.

2.4.2 Verfügbare Techniken

Die verfügbaren Techniken für Metacomputer haben meistens ihren Ursprung in parallelen oder verteilten Umgebungen, ohne die Notwendigkeiten einer Integration mit anderen Diensten. Aus diesem Ursprung ergibt sich auch die sehr große Anzahl der bestehenden Projekte. Um zu verdeutlichen, dass die für Metacomputer-Projekte notwendigen Werkzeuge und Mechanismen bereits vorhanden sind, werden anschließend einige Mechanismen aufgezählt und deren Funktion erläutert.

Gemeinsame Objekträume

JavaSpace [58, 171] ist ein gemeinsamer Objektraum für verteilte Anwendungen, der von dem *Linda* Tupelspace abgeleitet worden ist [28] .

Eine Anzahl von Prozessen kommuniziert hier über den Fluss von Objekten in und aus einem oder mehreren Räumen (Spaces). Ein Raum ist dabei eine Halde für persistente Objekte, die man speichern und austauschen kann. Prozesse kommunizieren hier nicht direkt, sondern nur über den Datenfluss des Raumes. Auch die Synchronisation mehrerer Prozesse erfolgt über den Objektraum. Operationen auf dem *JavaSpace* sind take, read und write [6].

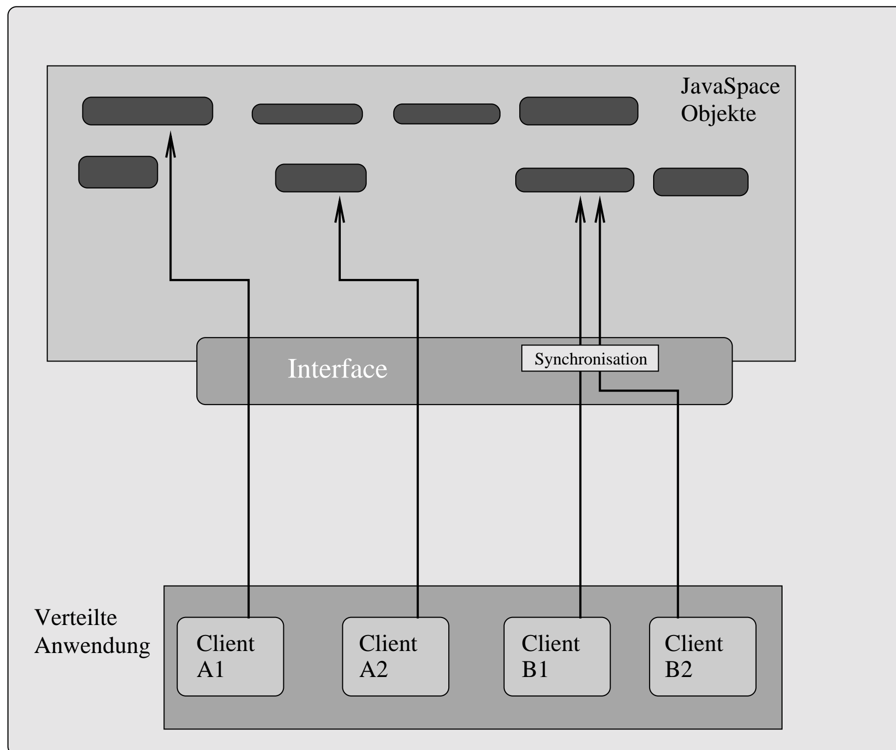

Abbildung 2.6: Gemeinsamer Objektraum bei *JavaSpace*

Objekte im *JavaSpace* sind passiv, das heißt, sie lassen sich nicht in dem Objektraum modifizieren. Hierzu müssen sie explizit entnommen, geändert und gespeichert werden.

Obwohl die Kernfunktionalität von *JavaSpace* (read, write, take) sehr einfach gestaltet ist, sind semantisch ungewöhnliche Methoden möglich:

- Durch die sehr lose Koppelung über den Speicher sind kommunizierende Prozesse möglich, die sich in ihrer Laufzeit zeitlich nicht überlappen.

[6] Alle Operatoren auf dem *JavaSpace* sind atomar

- Der Zugriff auf Objekte erfolgt nicht über Speicheradressen oder Bezeichner, sondern assoziativ[7].

Auf diese Weise realisiert *JavaSpace* gemeinsamen weitverteilten Speicher.

Für technisch-wissenschaftliches Hochleistungsrechnen ist *JavaSpace* nicht geeignet. Schnelle direkte Kommunikation ist nicht vorgesehen. Probleme mit sehr grobgranularer Parallelität lassen sich jedoch einfach und elegant (durch Producer/Consumer Architektur) lösen (siehe Abbildung 2.6). Vorstellbar sind auch sehr lang laufende Applikationen, deren beteiligte Knoten nur zeitweise mit der Anwendungen arbeiten.

CORBA (Common Object Request Broker Architecture) ist ein Standard der Object Management Group [122], der einen Rahmen für verteilte Objekte vorgibt. Die Objektschnittstellen innerhalb von CORBA werden mit Hilfe von IDL (Interface Specification Language) formuliert. Der IDL Übersetzer erzeugt für diverse Sprachen einen Coderahmen, mit dem die Implementierung die Objekte ansprechen kann. Dieser Coderahmen umfasst Skeleton und Stub, die die lokale Objektrepräsentation darstellen, über die die eigentliche Kommunikation stattfindet.

Dadurch kann die Implementierung der Anwendung plattformübergreifend ablaufen. Weiterhin definiert der CORBA Standard Methoden zur Lokalisierung von Objekten, eine Garbage Collection und ein einheitliches Datenformat für die Kommunikation zwischen verschiedenen CORBA konformen Implementierungen verschiedener Hersteller.

Der Grund für das Fehlen von CORBA Anwendungen im Bereich des wissenschaftlichen Rechnens liegt hauptsächlich in dem hohen Verwaltungsaufwand von CORBA Objekten. Metacomputing-Projekte mit gemeinsamen Objekträumen realisieren hier proprietäre Mechanismen (beispielsweise *Legion* mit dem *Legion* Objekt Modell).

DCOM (Distributed Component Object Model) ist das verteilte Objektmodell der Firma Microsoft, welches aus dem Component Object Model, das nur lokale Objekte verwaltet, entwickelt wurde [36].

Während der CORBA IDL Übersetzer Quellcode für verschiedene Programmiersprachen erzeugt, generiert DCOM aus der Schnittstellen Definition MIDL (Microsoft IDL) ein binäres Format, in dem die Funktionsdeklarationen der Objekte beschrieben sind. Weiterhin sind DCOM Objekte keine Objekte im klassischen Sinn, sondern Komponenten, die aus verschiedenen Objekten bestehen und diese eine gemeinsame Schnittstelle anbieten. Auch stehen die Funktionen einer Objektlokalisation und eine Garbage Collection zur Verfügung.

In Anbetracht der proprietären Umsetzung von DCOM und der Beschränkung auf Betriebssysteme von Microsoft ist DCOM für wissenschaftlich-technisches Rechnen nicht geeignet.

[7] Hierzu wird ein Muster für das Zielobjekt erstellt und mit den bekannten Informationen gefüllt. Stimmen diese Informationen mit Informationen eines Objektes aus dem *JavaSpace* überein, werden die freigelassenen Felder mit denen des *JavaSpace* gefüllt

Kommunikationsbibliotheken

PVM und MPI [154, 116] bilden die Kommunikationsgrundlagen für die meisten parallelen Programmierumgebungen. Ihre Funktionalitäten sind in allen Metacomputing-Umgebungen, die explizite Kommunikation verwenden, in ähnlicher Form vorhanden.

Alternativ zu expliziter oder impliziter Kommunikation sind entfernte Methodenaufrufe dazwischen angesiedelt. Hierdurch kann der Entwickler Funktionen oder Objekte auf entfernten Rechnern ansprechen ohne die expliziten Kommunikationsaufrufe verwalten zu müssen, ist sich dabei aber der stattfindenden Kommunikation bewusst. Diese Kommunikationsart wird mit lokalen Stellvertretern (Proxy) der entfernten Objekte oder Funktionen realisiert, die den Aufruf mit den entsprechenden Parametern,wenn erforderlich konvertiert, an ein Gegenstück auf einem entfernten Rechner weiterleiten (siehe Abbildung 2.7). In diese Gruppe gehören die klassischen RPC (Remote Procedure Call)[79] und Java RMI (Remote Method Invocation)[117].

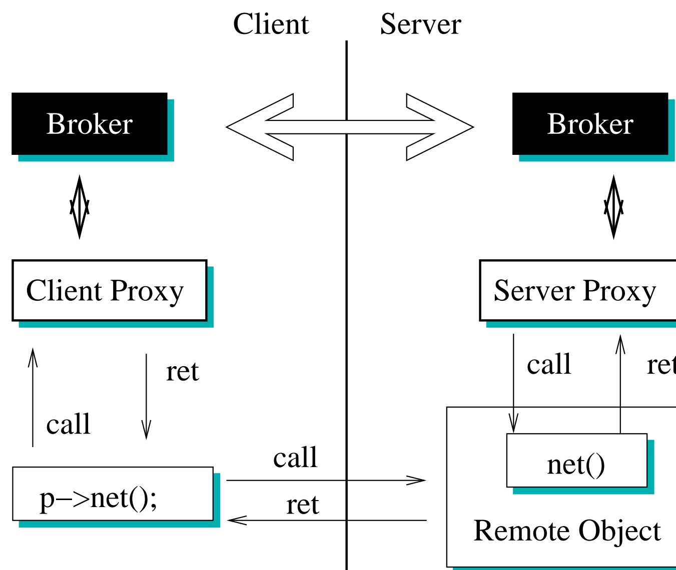

Abbildung 2.7: Technik des entfernten Methodenaufrufs

Werkzeuge zur Beobachtung

Werkzeuge zur Beobachtung paralleler Anwendungen, die sich transparent in die Metacomputer-Infrastruktur einbinden sind gegenwärtig ein Schwerpunkt der Forschung. Dabei sind Werkzeuge zur Systemüberwachung in vielen Metacomputer-Umgebungen vorhanden, diese beschränken sich jedoch auf die Überwachung der Clients (*Alert* unter *Legion*). Umfassender ist das Beobachtungswerkzeug des Metacomputer-Systems Legion *HBM* (Globus Heartbeat Monitor)[64]. Hiermit ist

auch die Überwachung von Anwendungen möglich, indem die Bibliothek HBMCL (HBM Client Library) zu der Anwendung gebunden wird. Diese liefert Informationen an einen lokal laufenden Prozess *HBMLM* (HBM Local Monitor), die dieser wiederum an externe *HBMDC* (HBM Data Collector) sendet. Hiermit lassen sich sowohl das *Globus* System, als auch die Anwendung überwachen.

Einen generischen Ansatz bietet *OMIS* (On-Line Monitoring Interface Specification) [109] , das einen Standard zur Kommunikation von Werkzeugen definiert. Hierdurch wird eine einheitliche Schnittstelle zwischen Anwendungen und einer Vielzahl an Werkzeugen (zur Leistungsanalyse [21], Visualisierung, Fehlersuche [176], Lastbalanzierung) angeboten, die allen Werkzeugen eine konsistente Sicht auf die Anwendung ermöglicht. Im Gegensatz zu den bestehenden Lösungen in Metacomputer-Architekturen ist dieser Ansatz nicht proprietär und ermöglicht zudem eine Werkzeug zu Werkzeug Kommunikation [110] und die Steuerung der Anwendung. Eine *OMIS* konforme Implementierung (*OCM*, OMIS Compliant Monitor) findet sich in [175, 108].

Einen ähnlichen, generischen Ansatz, verfolgt *MIMO* (Middleware Monitor) [128], der die Entwicklung und Beobachtung von Middleware-Anwendungen erlaubt. Hierzu teilt *MIMO* die Anwendung[8] in sechs Schichten ein und klassifiziert die Informationen dementsprechend. Bei *MIMO* angemeldete Werkzeuge können daraufhin Informationen aus den sie betreffenden Schichten von der Anwendung oder von anderen Werkzeugen erhalten. Dafür sind zwei Mechanismen vorgesehen. Zum einen können Adapter, innerhalb der Anwendung implementiert, sich bei dem MIMO System anmelden und Informationen übermitteln, zum anderen können Bibliotheken instrumentiert werden, die wiederum mit der Anwendung gebunden werden. Zusammen mit *MIVIS* (MIMO Visualizer), einem Werkzeug auf MIMO Basis, wird in Abschnitt 4.4 das vorgestellte *LsD-System* beobachtet. Die dem *MIMO* Schichtenmodell zugeordneten Schichten des *LsD-Systems* sind in Tabelle 4.4 zu sehen, die Instrumentierung über Adapter in Abbildung 4.12.

Weitere Techniken im Überblick

WebOS[23] ist eine Integration verschiedener Dienste für weitverteilt laufende Applikationen. Neben anderen Diensten bietet es einen Sicherheitsmechanismus, *CRISIS* und ein verteiltes Dateisystem, *WebFS*[165, 13]. *CRISIS* implementiert ein redundantes Identifizierungs- und Authentifizierungs System basierend auf dem Austausch öffentlicher Schlüssel und feingranularer, dezentraler Zugriffskontrollen auf lokale Domänen. *WebFS* bietet ein globales, konsistentes Dateisystem auf der Grundlage des Namensraumes von HTTP.

Condor [19, 156] ist ein Ressourcenmanagementsystem, das den Stapelbetrieb von Aufträgen auf unbenutzten Arbeitsplatzrechner erlaubt. Der Zuteilungsmechanismus ist prioritätengesteuert (interaktive Benutzer haben die höchste Priorität) und

[8]In diesem speziellen Zusammenhang wird mit Anwendung die Middleware, bzw. das Metacomputer-System gemeint und nicht die Anwendung eines Benutzers, die innerhalb eines Metacomputers läuft.

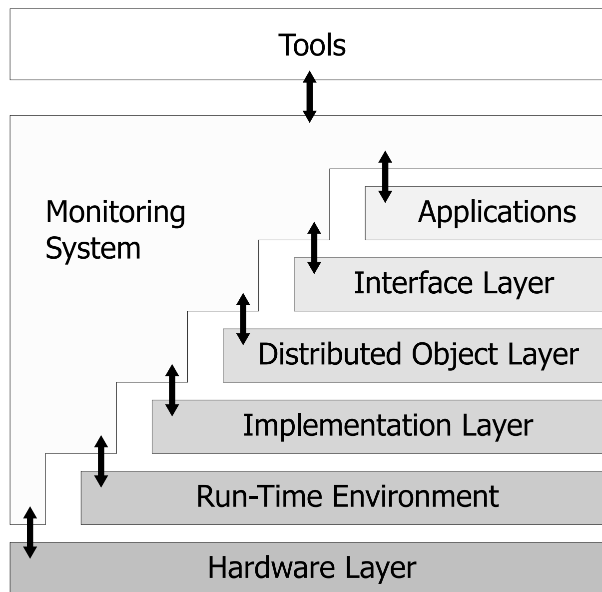

Abbildung 2.8: Klassifizierung von Informationen des MIMO Systems.

lässt eine Migration von Prozessen durch Sicherheitspunkterzeugung bei Verletzung der Prioritäten zu.

Cilk [14] ist eine Programmiersprache auf C Basis, die die Generierung von verschiebbaren, parallelen Prozessen erlaubt. *Cilk* wird von dem Metacomputer-System *Atlas* verwendet (siehe dort, Abschnitt 3.1.3).

2.5 Sicherheitsaspekte

Mit dem Thema Computer- und Softwaresicherheit hat ein relativ neues Thema Einzug in das Gebiet des wissenschaftlich-technischen Rechnens gehalten. Dieses Thema wird in der Zukunft an Bedeutung zunehmen. Der Grund hierfür, die Bedeutung für Metacomputer-Systeme und die daraus folgenden Probleme werden in diesem Abschnitt erläutert.

In diesem Rahmen gilt die folgende Definition von Computersicherheit, die sich an die Definition von [83] hält:

Definition 2.18 *Computersicherheit*

> Computersicherheit ist die Unterbindung von nicht autorisierten Zugriffen oder der nicht autorisierten Nutzung von Computern oder Netzwerken. □

Diese allgemeine Definition kann durch Bestimmung von sicherheitsrelevanten Komponenten und deren Missbrauch (d.h. nicht autorisierte Nutzung oder nicht autorisierten Zugriff) genauer spezifiziert werden.

Zu schützende Komponenten sind Daten, Kommunikationswege und Ressourcen. Für alle drei Komponenten sind Vertraulichkeit, Integrität und Verfügbarkeit zu gewährleisten.

Drei Gründe, und vor allem deren Wechselwirkungen, machen Sicherheitsvorkehrungen für wissenschaftlich-technische Anwendungen in Zukunft unabdingbar.

Netz Bedeutung
: Im Zeitalter des Internets und der Vernetzung sind einzelne Rechner, oder einzelne dedizierte Hochleistungsrechner, nicht mehr denkbar. Die Netzverbindung ist inzwischen eine der wichtigsten Komponenten eines Rechners. Damit finden Berechnungen nicht mehr in einer vertrauenswürdigen Umgebung trusted environment ("trusted environment") statt.

Netz Entwurf
: Das Transportprotokoll im Internet, TCP/IP, ist konzeptionell, aus historischen Gründen, auf gutwillige Kommunikationsparteien ausgerichtet. Mit dem Einzug von Heimrechnern in das ursprünglich universitäre Internet werden böswillige Parteien immer wahrscheinlicher, die die Kommunikation kompromittieren. Es existiert zur Zeit keine technische Möglichkeit, dies auf Protokollebene zu verhindern [9], auch zukünftig ist dies mit der Einführung der neuen Protokollschicht IPv6 [38] nicht abzusehen. Daher können Gegenmaßnahmen nicht auf Änderung der Netzstruktur beruhen.

Netz Kommerzialisierung
: Mit der zunehmenden Verbreitung der Internets beginnt auch zunehmende kommerzielle Nutzung der Netzinfrastruktur. Die kommerzielle Nutzung des Netzes ist jedoch wegen Wahrung von Firmengeheimnissen sowie von Kundenschutz auf eine sichere Kommunikation angewiesen. In Zukunft werden auch kommerzielle Anbieter die Rechenleistungen von Metacomputer-Systemen im Rahmen von firmeninternen globalen Netzen oder auch von öffentlichen Metacomputern nutzen wollen und daher auf eine sichere Infrastruktur drängen.

Diese Gründe erfordern die Einführung von Sicherheitskonzepten nicht nur für herkömmliche sicherheitsrelevante Bereiche, sondern auch für Metacomputer-Strukturen, wenn sie von einer beabsichtigten Nutzung, wie in Definition 2.10 gefordert, ausgehen. Dies erfordert bei Metacomputern eine umfangreiche Vorsorge, da die vielfältige Nutzung und Darbietung von Diensten ein breite Angriffsfläche bietet. In Abbildung 2.9 sind mögliche Angriffspunkte auf die Kommunikationswege und Komponenten einer Metacomputer-Architektur schematisch dargestellt.

Hierbei sollte nicht übersehen werden, dass Sicherheitskonzepte, wie viele andere moderne Konzepte zur Softwareentwicklung, mit Leistungseinbußen bezahlt werden müssen.

[9] Dieses zeigen in jüngster Zeit die vergeblichen Versuche, DoS (Denial Of Service) Attacken mit technischen Hilfsmittel zu verhindern. Verhinderung durch Strafverfolgung ist aufgrund der möglichen Anonymität und der unklaren nationalen Zuständigkeit nicht durchzusetzen.

Sichere Benutzung von entfernten Ressourcen benötigt einen erhöhten Protokollaufwand bei dem Zugriff auf fremde Ressourcen und vor allem zusätzliche Kommunikation, die meistens der entscheidende Faktor für den Geschwindigkeitsgewinn einer parallelen Anwendung ist. Weiterhin kostet die Verschlüsselung der Datenpakete weitere Leistung. Daher waren Sicherheitsüberlegungen im dem Bereich des wissenschaftlich-technischen Rechnens unüblich, und angesichts des lokalen Charakters der Architekturen auch nicht notwendig.

Bei Anwendungen des wissenschaftlich-technischen Rechnens, das im allgemeinen auf hohe Rechenleistung optimiert ist, und dies insbesondere der Grund für die Nutzung von verteilter Hardware ist, ist ein Abwägen zwischen gewonnener Leistung und Sicherheit notwendig. Hier sollte die Sensibilität der Daten und der beteiligten Domänen richtungweisend sein. Anwendungen wie in Abschnitt 5 beschrieben, erfordern ein hohes Maß an Sicherheit, populäre Metacomputer-Systeme wie in Abschnitt 3.1.1.2 wenig, beziehungsweise keine Sicherheit.

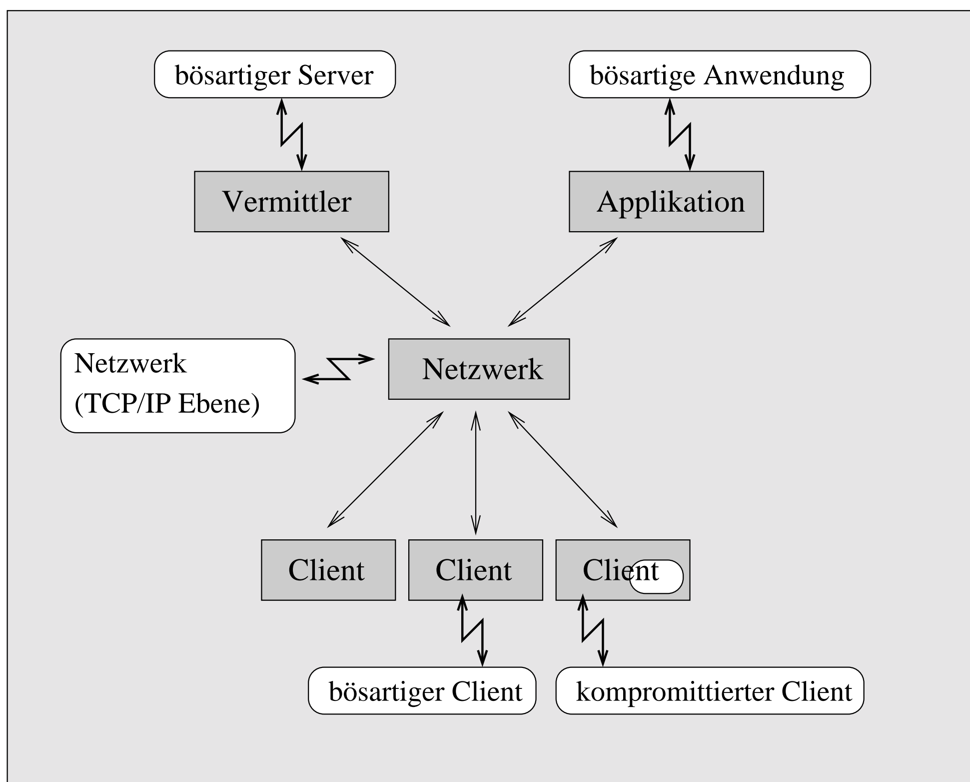

Abbildung 2.9: Mögliche Angriffspunkte auf ein verteiltes Metacomputing-System

Im folgenden werden die sicherheitsrelevanten Bereiche eines Metacomputer-Systems diskutiert und die Anforderungen an ein sicheres System formuliert. Zunächst gilt es hier die Eigenschaften des verteilten Rechnens, die für einen Sicherheitsmechanismus relevant sind, darzustellen. FOSTER [57] sieht hier folgende Kriterien als Charakteristika einer Metacomputer-Umgebung:

- Die Benutzergruppe ist verhältnismäßig groß (wissenschaftliches universitäres Umfeld) und wechselt oft.
- Sehr viele und häufig wechselnde Ressourcen.
- In Umgebungen mit langlaufenden Applikationen können die Ressourcen dynamisch zur Laufzeit wechseln.
- Auch die Kommunikation erfolgt über viele verschiedene Mechanismen, Verbindungen werden zur Laufzeit geöffnet und geschlossen, mit vorher nicht bekannten Kommunikationsendpunkten.
- Ressourcen in verschiedenen Domänen benötigen verschiedene Sicherheitsmechanismen, diese erfolgen nach verschiedenen Richtlinien.
- Verschiedene Benutzer haben auf verschiedenen Systemen verschiedene Rechte.
- Es gibt keine international gültigen Sicherheitsrichtlinien.

Diese Punkte verdeutlichen die nötige Flexibilität der verschiedenen Sicherheitsmechanismen in einer, bezüglich der teilnehmenden Domänen, heterogenen Metacomputer-Umgebung.

Daher sind herkömmliche Maßnahmen gegen Angriffe nur im Rahmen von geeigneten, domänenübergreifenden Sicherheitsrichtlinien sinnvoll. Die Konstruktion dieser Richtlinien und deren effiziente Realisierung mit herkömmlichen Mittel (Authentifikation, Identifikation, sichere Übertragung über unsichere Kommunikationswege, Verschlüsselung) ist der Fokus der wissenschaftlichen Bemühungen um Sicherheit bei Metacomputer-Umgebungen. Dass diese durchaus zu erfüllen sind, zeigen die Systeme von *Legion* und *Globus* (Abschnitt 3.1.4 und 3.1.5). Der sehr hohe Aufwand, der hier nötig ist und erbracht wurde, ist durch die aktuelle und insbesonders die zukünftige Bedeutung gerechtfertigt.

Auf obigen Punkten aufbauend wird in [57] folgende Sicherheitspolitik für die Realisierung von *Computational Grids*[55]) aufgestellt:

- Es existieren mehrere Domänen mit unterschiedlichen Richtlinien.
- Rechte auf lokalen Domänen sind auf diese beschränkt.
- Globale Rechte werden auf lokale Rechte abgebildet. Diese Abbildung ist domänenabhängig.
- Domänenübergreifenden Operationen benötigen eine gegenseitige Authentifizierung.
- Globale Authentifizierung impliziert lokale Authentifizierung.
- Zugriffsrechte werden nur lokal vergeben.

- Prozesse werden Benutzern zugeordnet und erben Teile von deren Rechten.
- Eine Berechtigung kann für mehrere Prozesse einer Domäne gültig sein.

Diese Richtlinien erlauben eine flexible lokale Sicherheitspolitik und sind geeignet, mehrere Domänen zu einer Metacomputer-Umgebung zu verbinden, die eine generelle Sicherheitspolitik beinhaltet.

2.6 Zusammenfassung

In diesem Kapitel wurde ein Überblick über die Entwicklung von Metacomputern gegeben und die Motivation hierfür aufgezeigt. Hierbei treten Anforderungen auf, die sich teilweise stark von den Anforderungen des traditionellen Parallelrechnens unterscheiden. Diese Anforderungen und Möglichkeiten zur Erfüllung wurden benannt. Zur Klassifizierung von Metacomputern wurden diese zu den verschiedenen Modellen verteilten Rechnens eingeordnet und Metacomputer nach den Anwendungen beurteilt, für die sie geeignet erscheinen. Hierbei ist der Grad der Parallelität des zu berechnenden Problems der entscheidende Faktor.

Im Hinblick auf die spätere Einführung des *LsD-Systems* wurde die verwendete Terminologie festgelegt. Die Dienste, die Metacomputer einer Anwendungen zur Verfügung stellen können, wurden aufgezählt, sowie verschiedene Techniken zu deren Realisierung genannt. Abschliessend wurde auf die besondere Bedeutung der Sicherheit eingegangen.

3. Metacomputer-Systeme

Dieser Abschnitt enthält eine systematische Übersicht über gegenwärtige Metacomputer und über Metacomputer, deren angewandte Methoden (siehe vorhergehendes Kapitel) für die Entwicklung von Bedeutung sind.

Dabei werden Systeme diskutiert, die von sehr einfachen Umgebungen bis zu komplexen und aufwändigen Systemen reichen, welche die formulierten Anforderungen weitgehenst erfüllen. In Abschnitt 3.1.6 folgt eine Aufzählung an weiteren Systemen, die interessante Merkmale aufweisen oder die ähnliche Techniken in Bezug auf das vorgestellte *LsD-System* verwenden.

Eine Zusammenfassung und Bewertung in Abschnitt 3.2 zeigt die Mängel dieser Systeme für kommerzielle Nutzer, die durch die Entwicklung des *LsD-Systems* im nachfolgenden Kapitel vermieden werden.

3.1 Metacomputer-Systeme

Die Anzahl der verteilten Systeme ist in den letzten 10 Jahren analog zu der zunehmenden Vernetzung von Computersystemen sehr stark angestiegen. Diese Zunahme gilt nicht nur für Systeme, die lokal verteilte Ressourcen umspannen, sondern auch für global verteilte Systeme. Einige dieser Architekturen werden exemplarisch in diesem Abschnitt beschrieben. Hierbei wird, wo es gegeben erscheint, teilweise auf die eingesetzten Techniken, auf die Struktur der Systeme oder auf mögliche Anwendungen eingegangen.

Die Reihenfolge hält sich hierbei an die zunehmende Komplexität der beschriebenen Systeme.

Zu Beginn werden exemplarisch ein paar Versuche zum globalen verteilten Rechnen beschrieben, die jedoch keine grundlegenden Paradigmen des Parallelrechnens verwirklichen und die auch keine echten Metacomputer-Umgebungen darstellen, die aber die Rechenleistung verdeutlicht, welche bei dem verteilten Rechnen im Internet zur Verfügung steht.

Angesichts der Bedeutung von Java in heterogenen Rechenumgebungen werden zwei in Java realisierte Client/Server Systeme diskutiert, und anschließend Metacomputer-Systeme mit komplexen Strukturen erläutert.

Jeder dieser benannten Architekturen kann dabei als eine Lösung zu bestimmten Problemen des globales Rechnens herangezogen werden. Denn hierbei zeigen sich bei der erste Gruppe, dass die Rechenleistung sich durchaus mit sehr vielen Knoten, einer grundlegenden Annahme bei dem Rechnen von Grand Challenges in globalen Netzen, skalieren lässt und sich somit die theoretisch vorhandene Rechenleistung auch praktisch nutzen lässt.

3.1.2 zeigt, dass Probleme mit der Heterogenität der Maschinen im Internet sich mit Lösungen in Java beheben lassen. Die nachfolgenden Metacomputing-Projekte zeigen, dass auch komplexe Probleme der Grand Challenges, die sich nicht mit trivialer Parallelität berechnen lassen, gelöst werden können. Desweiteren können diese Systeme als erster Schritt zu "Computational Grids" Computational Grids betrachtet werden, die diese Rechenleistung für den Benutzer transparent zur Verfügung stellen.

3.1.1 Projekte zur Nutzung von trivialer Parallelität

3.1.1.1 Distributed.Net

Distributed Net [1] [40], eine Vereinigung zur Forschung auf dem Gebiet des verteilten Rechnens, wurde 1997 gegründet. Das Ziel ist auch hier die Nutzung der Ressourcen von Privatanwendern, die nur begrenzte Zeit mit dem Internet verbunden sind.

Als Anwendungen wurden Umgebungen aufgesetzt, die unter anderem Mersenne Primzahlen berechnen und diverse (erfolgreiche) Versuche, von *RSA Security* ausgeschriebene Wettbewerbe zur Entschlüsselung von RC5 [2] Verschlüsselung mit mehreren Schlüssellängen. Hierbei wurde im Oktober 1997 ein RC5 Wettbewerb mit 56 Bit Schlüssellänge beendet (DES-I)[3],

Hierbei unterteilt ein in C++ programmierter Masterserver den Suchraum (2^{56} Schlüssel) in Blöcken zu 2^{39} Schlüssel und vergibt diese an anfragende Server. Diese unterteilen den erhaltenen Suchraum wiederum und vergeben sie an die endgültigen Clients. Deren Ergebnisse gehen den umgekehrten Weg. Ein Authentifizierungprotokoll verhindert die Ergebnisübergabe von nicht autorisierten Rechnern. Die Fehlertoleranzmechanismen bei ausfallenden Clients beschränkt sich auf ein Timeout von 12 Stunden, nach dessen Ablauf die Schlüsselblöcke erneut an andere Clients vergeben werden. Der DES-I Schlüssel wurde nach 212 Tagen gebrochen.

[1] Distributed Computing Technologies Inc

[2] RC5 ist eine Verschlüsselung von Ronald Rivest für RSA Security (http://www.rsasecurity.com/), die auf einer blockweisen Rotation und Vermischung der Eingangsdaten mit diversen Operationen beruht.

[3] DES, Data Encryption Standard, ist der Verschlüsselungsstandard der US amerikanischen Regierung.

Die addierte Rechenleistung der distributed.net Versuche steigerte sich (DES-I bis DES-III) bis zuletzt auf 160.000 Knoten der Stärke eines Intel Pentium II mit 266 Mhz, die 130 Milliarden Schlüssel in der Sekunde testen.

3.1.1.2 S.E.T.I.

Relativ hohes Aufsehen erregte ein populistischer Ansatz des SETI Institutes [140]. Dieses Institut beschäftigt sich seit November 1980 mit der Suche nach fremden Lebensformen im Universum. Hierzu analysiert es Daten, die mit einem Radioteleskop [4] aufgenommen werden, nach regelmäßigen Mustern. Da hierbei enorme Datenmengen anfallen und diese bis auf die grundlegende Partitionierung des Suchraums unabhängig voneinander berechnet werden können, bietet sich eine parallele Abarbeitung an [37].

Seit Mai 1999 ist eine Infrastruktur geschaffen worden, die es ermöglicht, auf weltweit verteilten Rechnern diese Daten zu analysieren [141].

Hierzu wird das aufgenommene Signal, das eine Bandbreite von 2.5 Mhz umfasst, in 256 10 kHz Segmente zerlegt. Diese Daten, zusammen mit zusätzlichen Verwaltungsdaten ca 250kB, werden daraufhin auf Anfrage des teilnehmenden Knotens versendet. Das Programm auf Clientseite zerlegt diese Signale mittels Fast Fourier Transformationen, um regelmäßige oder pulsierende Signale innerhalb eines 12 Sekunden Intervalls, die ein Punkt benötigt, um den Messfokus des Teleskopes zu durchqueren, zu finden. Die Rechenzeit hierfür beträgt für eine Pentium III CPU mit 600 MHz Takt um die 10 Stunden, da der Rechenaufwand um besonders schwache Signale zu lokalisieren sehr hoch ist.

Das Ergebnis dieser Analyse wird anschließend an den Server geschickt, der die nötige Nachbearbeitung übernimmt, welche unter anderem aus dem Eintrag in eine Datenbank und der Überprüfung nach segmentüberlappenden Signalen besteht.

Die Teilnahme an diesem Projekt ist freiwillig, weshalb besondere Rücksicht auf die Beanspruchung der teilnehmenden Rechenressourcen genommen wurde. Aufgrund der langen Berechnung kann diese ohne Serververbindung stattfinden, erst zur Ergebnisübergabe und Empfang eines neuen Segmentes wird erneut eine Verbindung aufgebaut. Um interaktive Benutzer nicht in ihrer Arbeit einzuschränken, ist die Berechnung in einen Bildschirmschoner integriert, der bei Nutzung des Systems unterbricht. Auf Unix Systemen läuft der Prozess mit der geringst möglichen Priorität.

Trotz dieser Einschränkungen erreicht diese verteilt rechnende Plattform aufgrund der hohen Anzahl von ca 5000 Knoten täglich, eine Rechenleistung von 24 Tera FLOPs/sek [5].

---

[4] Das Arecibo Teleskop in Puerto Rico ist das derzeit größte Radioteleskop der Welt.

[5] Dies ist der Stand im August 2000, aktuelle und detaillierte Informationen sind unter http://setiathome.ssl.berkeley.edu/totals.html zu finden.

3.1.1.3 Bewertung

Systeme wie die beiden beschriebenen bieten eine ganze Reihe von Vorteilen. Die meisten positiven Eigenschaften resultieren aus der Tatsache, dass der Aufbau einer solchen Infrastruktur und die Architektur der darauf rechnenden Applikation sehr einfach ist.

- Die Infrastruktur der Hardware beschränkt sich auf eine leistungsstarke Maschine mit guter Netzanbindung, die für die Verteilung der Rechenjobs und das Zusammenführen der Ergebnisse zuständig ist. Alle weiteren Ressourcen sind verteilt und bedürfen keiner zentralen Administration.

- Die Applikation muss eine strenge Client/Server Struktur bieten, sowie die nötige Logik um Aufgaben zu unterteilen und die Ergebnisse zu sammeln. Da die Ausfallwahrscheinlichkeit einzelner Knoten sehr hoch ist, ist ein Fehlertoleranzmechanismus oder eine hohe Redundanz der Berechnung nötig. Periodische Sicherungspunkterzeugung auf der Serverseite kann für Datensicherheit sorgen.

Diese Einfachheit sorgt für geringe Fehleranfälligkeit des Gesamtsystems und auch für schnelle Applikationsentwicklung. Dafür sprechen auch die guten und schnell erreichten Ergebnisse aus Kapitel 3.1.1.1 und 3.1.1.2 .

Zweifellos werden solche Metacomputer-Systeme auf einen kleinen Nischenmarkt beschränkt bleiben. Dies liegt nicht nur in der Applikationsarchitektur (Client/Server) begründet, sondern auch in dem Umfeld solcher Projekte. Die dabei zu berechnenden Probleme müssen folgende Merkmale aufweisen:

- Kommunikation: Nur sehr wenige Probleme lassen sich klar in Client/Server Anteil unterteilen, ohne dass zusätzliche Kommunikation notwendig ist. In den beschriebenen Projekten wäre sogar jegliche weitere Kommunikation unmöglich, da die Kommunikationspartner keine dauerhafte physikalische Verbindung unterhalten.

- Es muss ein öffentliches Interesse an der Aufgabe bestehen. Die Teilnahme ist freiwillig und beruht allein an einem Interesse für den jeweiligen Projektinhalt. Demzufolge werden Projekte, deren wissenschaftliche Bedeutung zwar hoch, aber nicht von allgemeinem Interesse ist, geringe Erfolgsaussichten haben. Auch dürfte das öffentliche Interesse an einer Berechnung für Firmen gering sein.

- Die Anzahl und Menge der zur Verfügung stehenden Ressourcen sind sowohl in der Planungsphase als auch zur tatsächlichen Laufzeit unbekannt. Daraus ergibt sich, dass keinerlei Aussagen über die genaue Laufzeit möglich sind, oder ob das Projekt im Falle sinkenden öffentlichen Interesses überhaupt erfolgreich beendet werden kann. Diese nicht zusicherbaren Eigenschaften verhindern kommerzielles Interesse in solche Metacomputer-Systeme.

- Es gibt keine zwischen Hardware und Applikation vermittelnde Schicht. Daraus ergibt sich, das jede Applikation das Problem direkt auf die Infrastruktur aufsetzten muss. Hierdurch ist für jedes Projekt eine neue Anpassung erforderlich.

Für den aufgezeigten kleinen Markt an möglichen Anwendungsgebieten, Probleme mit trivialer Parallelität, bieten Metacomputer-Systeme basierend auf opportunistischen Ressourcen [115] jedoch sehr billige und trotzdem sehr leistungsstarke Plattformen.

3.1.2 Java basierte Client/Server Umgebungen

Zwei frühe Versuche, Metacomputing auf vielen heterogenen Maschinen zu realisieren, indem sie sich auf Java Applets und das HTTP Protokoll abstützen sind Javelin [27] und JET [125, 143].

3.1.2.1 Javelin

Javelin ist eine auf Java basierende Ausführungsumgebung, die globales verteiltes Rechnen ermöglichen soll.

Entwickelt wurde das Javelin System an der University of California, Santa Barbara. Die Entwicklung ist nicht offiziell beendet worden, steht jedoch seit Mitte 1999 still. Javelin stützt sich auf eine vorhergehende Arbeit der Entwickler mit ähnlichem Schwerpunkt, *SuperWeb* [2, 3] ab.

Das Ziel von Javelin ist es, mit einfachen Standardtechniken wie javafähigen WWW-Browsern und WWW-Servern ein Netz aus Maschinen aufzubauen, die über einen (mehrere waren für die Zukunft angedacht) zentralen Server Rechenressourcen zur Verfügung stellen, bzw. benutzen. Durch den Einsatz von Standardsoftware soll die Akzeptanz erhöht werden, da die Installation sehr einfach ist und eine flexible Architektur des Gesamtsystems erlaubt. Javelin zielt auf Probleme hoher Parallelität ab, die von sehr vielen anonymisierten Maschinen mit geringen Kommunikationsbandbreiten bearbeitet werden können.

Nicht adressiert werden Sicherheit, Skalierbarkeit und Fehlertoleranz.

Die Javelin Architektur besteht aus drei Typen beteiligter Rechner, die sich nicht zwingend gegenseitig ausschließen: Broker, Clients und Hosts (siehe Bild 3.1).

Hosts sind hierbei Rechner, die Rechenleistung anzubieten haben, Clients sind Rechner, die Ressourcen suchen und Broker realisieren die zentrale Verwaltung.

Hierbei registrieren sich Rechner, die Rechenleistung benötigen bei einem Broker indem sie Ihre Applikation in einem Java Applet und einer URL übertragen. Die URL des Applets wird einem Rechner, der Ressourcen anbietet, vermittelt. Dieser führt das Applet aus und übermittelt die Ergebnisse dem Broker.

Falls eine Kommunikation zwischen den Hosts nötig ist, wird diese ebenfalls durch den Broker vermittelt. Programmiermodelle, die hierbei unterstützt werden, sind

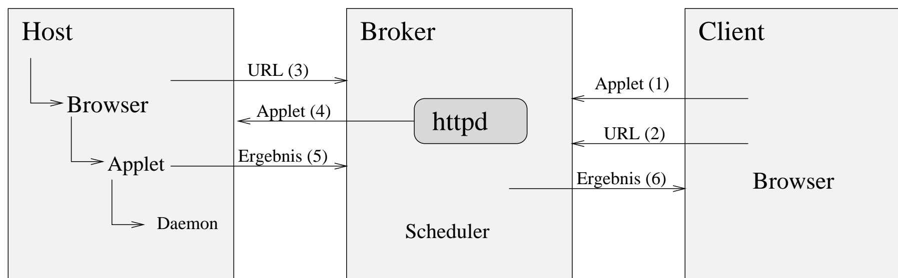

Abbildung 3.1: Die Javelin Architektur

SPMD [6] und Linda Tuple Space (siehe auch 2.4.2) [28]. Für die SPMD Programmierung werden zur Synchronisation *send()* und *receive()* und Barrieren unterstützt. Beides wird über den Broker realisiert, der die eingehenden Kommunikationsanforderungen protokolliert und gegebenenfalls weiterleitet.

Linda Operatoren (*In, Out, Barrier*) werden analog durch den Broker durchgeführt, der den Tuple Space verwaltet.

Alle in diesen Fällen benutzten Broker Funktionalitäten werden durch Servlets [7] realisiert.

Provisorische Leistungsergebnisse wurden mit Mersenne Primzahlen (die im Verhältnis zur Rechenzeit äußerst wenig Kommunikationszeit aufweisen) ermittelt. Hierbei zeigte sich, dass auf homogenen Rechnerverbünden der Geschwindigkeitsgewinn linear bleibt, solange für die Prozessverteilung genügend Rechner zur Verfügung standen. Betont wird auch hier, dass die Algorithmen zur Prozessverteilung großer Aufmerksamkeit bedürfen, da im Falle ungünstiger Verteilung, hier wurde die rechenintensivste Teilaufgabe als letztes vergeben, der Leistungsgewinn erheblich geschmälert wird.

Bei heterogenen Rechnerverbünden zeigte sich kein eindeutiges Ergebnis, was darin begründet lag, dass die Leistungsunterschiede der beteiligten Rechner sehr groß waren[8].

3.1.2.2 JET

JET basiert auf der Annahme, dass die größten Rechenressourcen nicht in Rechenzentren stehen, sondern verteilt in Büros und bei Heimanwendern auf den Schreibtischen. Diese Maschinen, die ein Großteil der Zeit (ca 90-95% [137]) ohne Rechenlast sind, gilt es in einer Infrastruktur zu verbinden und gemeinsam zu nutzen[9]. Das

[6] Single Programm, Multiple Data, ein Programmcode berechnet in mehreren Instanzen verschiedene Daten eines Problems.

[7] Programme, die auf der Serverseite ausgeführt werden.

[8] Die Gesamtlaufzeit aller Knoten sank dabei nur unwesentlich unter die Laufzeit der schnellen Knoten (für die ein Java Just-in-Time Compiler zur Verfügung stand).

[9] Die Idee, die Rechner in deren niedrigen Lastzeiten, zumeist nachts, zu nutzen, wird daher auch 'Supercomputing at night' genannt.

zugrundeliegende Modell ist demnach einfacher als bei Javelin, es wird eine strikte Master/Worker Aufteilung vorausgesetzt.

Die Kommunikation innerhalb der JET Umgebung ist UDP basiert. Eine darüber liegende Schicht sorgt für eine fehlerfreie Übertragung ohne dass eine verbindungsorientierte Kommunikation erforderlich ist.

Als hierbei zu lösende Probleme werden die in Abschnitt 2.2 genannten Themen beschrieben. Um eine Reihe dieser Probleme zu lösen, wurde JET in Java implementiert.

Ziel sind auch hier Applikationen, die sehr lange Laufzeiten und starke Parallelität aufweisen. Eine Kommunikation der Applets untereinander, also außerhalb des adressierten Master/Worker Modells ist nicht vorgesehen.

Die Server Seite von JET besteht aus einem Prozess, der in drei Threads aufgeteilt ist (Master). Ein Producer zerlegt die Applikation in kleine Teilaufgaben und übergibt sie einer Job Queue, ein Consumer Thread sammelt die Ergebnisse der Worker ein, sowie ein Provider Thread, der anfallende Informationen über die ablaufende Applikation oder den Gesamtzustand des Systems zur Verfügung stellt. Die Job Queue wird periodisch zur Sicherungspunkterzeugung auf permanenten Speicher ausgelagert.

Um das Problem hoher Latenzen zu vermeiden, implementiert JET einen Mechanismus, der auf der Client Seite (Worker) Aufträge holt und Resultate verschickt, während gleichzeitig bereits geholte Aufträge bearbeitet werden. Jede dieser Aufgaben läuft in einem Thread innerhalb des Java Applets. Durch diesen einfachen Mechanismus des *work stealings* wird zu einem begrenzten Ausmaß eine Lastbalanzierung erreicht, da schnelle bzw. gut angebundene Knoten mehr Aufträge bearbeiten können.

Für zukünftige Versionen war eine hierarische Struktur geplant, die sich in Cluster mit eigenen Job Queues unterteilt, um Engpässe in der Kommunikation zum Master bei Architekturen mit sehr vielen angeschlossenen Maschinen zu vermeiden.

Die Anzahl der Rechner in einem Metacomputing-System ist ein entscheidender Faktor bei der Fehlertoleranz. PEDROSO, SILVA und SILVA [125] berechnet hier bei nur 100 teilnehmenden Rechnern eine MTBF von ca. drei Stunden. JET implementiert hier einen Mechanismus, der durch Sicherungspunkterzeugung und fortlaufendes Protokollieren den Ausfall mehrerer Rechner kompensiert, indem die verlorenen Jobs neu verschickt werden. Weil die einzelnen Applets einer Applikation nicht miteinander kommunizieren und selber ohne Zustände sind, können die Aufträge, die während der Bearbeitung auch auf dem Server gehalten werden, neu verschickt werden, ohne dass die gesamte Applikation neu aufgesetzt werden muss.

Die Ergebnisse einer JET Testumgebung wurden mit Windows 95 Rechnern (Pentium I) als Clients/Worker und mit einer Sun Ultra-Sparc als Master (heterogener Rechenverbund) bzw. mit einem Cluster bestehend aus PentiumPro (Homogener Rechenverbund) in einer 10 Mbit Netzwerk und dem JDK 1.1[10] ermittelt.

[10] Java Development Kit, Version 1.1

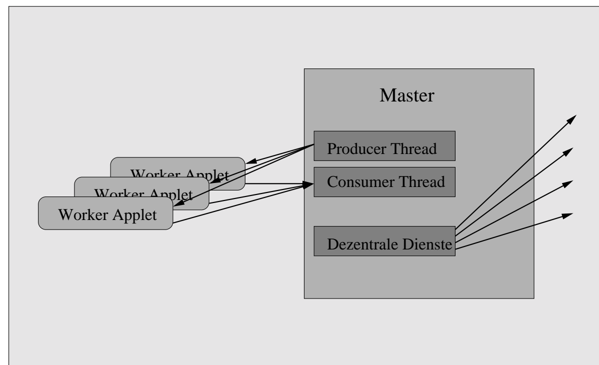

Abbildung 3.2: Die JET Architektur

Auf dem heterogenen Cluster zeigte sich bei einem N Damen Problem [11], $N = 9$, bei neun Maschinen ein Speedup von sechs.

Auf dem homogenen Cluster lag der Speedup immer über sieben bei acht teilnehmenden Maschinen.

3.1.2.3 Bewertung

Die Architekturen von Javelin und JET beschränken ihre Einsatzfähigkeit auf Applikationen mit sehr hoher Parallelität. Dies liegt an der eingeschränkten Kommunikationsfähigkeit von Java Applets, die nur Socket Verbindungen zu Rechnern aufbauen können, von denen die Applets geladen wurden. Alle Nachrichten müssen daher über den Broker vermittelt werden, was erheblichen Einfluss auf die Latenzzeiten hat. Darüber hinausgehend haben Applets weitere, durch die Sicherheitskonzepte der früheren Java Virtuellen Maschinen bedingt, Einschränkungen auf den ausführenden Rechnern (siehe 4.5.1).

Innerhalb dieser Einschränkungen zeigt sich durch die guten Leistungsergebnisse bei entsprechenden Problemen hoher Parallelität, dass solche Projekte durchaus als Erfolge für globales Rechnen gelten können.

3.1.3 Atlas

ATLAS [7] ist ein Metacomputer-System der *University of California at Berkeley* und der *University of Texas at Austin.*

Ziel der Infrastruktur ist die Ausnutzung weltweiter Rechnerressourcen in und zwischen Instituten. Hierfür werden die in Abschnitt 2.2 genannten notwendigen Eigenschaften angestrebt.

[11] Es gilt N Damen auf einem Schachbrett zu platzieren, wobei jede Dame horizontal, vertikal und diagonal keine weitere bedrohen darf.

Als Grundlage von ATLAS dienen Java und *Cilk* [14]. *Cilk* ist eine Programmiersprache, die auf C basiert und die parallele Ausführung von Threads ermöglicht. Zusammen mit einer Laufzeitumgebung, die Lastausgleich durch "work stealing"[12] betreibt, ermöglicht das System ein effiziente Ausführung mehrfädiger Programme. Dieses System ist, bis auf geschwindigkeitskritische Teile, auf die Java Plattform portiert worden.

Das ATLAS System besteht aus Manager, Clients und Compute Servern. Clients melden sich hierbei mit der Applikation bei dem Manager an, dieser sucht bei den angemeldeten Compute Servern nach freien Rechenressourcen, die die Applikation übernehmen. Ist dort der Applikationsprozess (die innerhalb der Java Virtuellen Maschine läuft) gestartet, können andere Compute Server nach dem work stealing Verfahren Threads der Applikation auf den lokalen Knoten ziehen. Das *Cilk* Ausführungsmodell unterstützt diesen Mechanismus, da *Cilk*-Programme eine baumartige Sequenz von parallel ausführbaren Threads erzeugen. Diese partiell unabhängigen Threads können auf verschiedene Knoten des ATLAS Systems ausgeführt werden. In Abbildung 3.3 ist beispielhaft eine Fibonacci Implementierung mit *Cilk* zu sehen. Hierbei können die mit `spawn` erzeugten Threads von dem *Atlas* System verteilt werden.

```
cilk int Fib (int n)
{
    if (n < 2)
      return n;
    else
    {
        int x,y;
        x = spawn Fib(n-1);
        x = spawn Fib(n-2);
        sync;
        return (x+y);
    }
}
```

Abbildung 3.3: Fibonacci Implementierung mit Cilk

Gemeinsamer Dateizugriff erfolgt durch den Zugriff über URLs mit zusätzlichen Caching-Mechanismen.

Um die Skalierbarkeit zu erhöhen, kann eine hierarische (Baum-) Struktur aus Managern aufgebaut werden, bei denen die Compute Server die Blätter des Baumes

[12] Threads werden zu Prozessoren mit wenig Last migriert.

bilden. In diesem Fall wird das work stealing durch den übergeordneten Manager übermittelt. Dies hat den Vorteil, dass migrierende Threads erst auf Knoten des lokalen Netzwerkes berechnet werden. Erst wenn hier keine Ressourcen frei sind, wird auf einen darüberliegenden Manager gewechselt.

Im Gegensatz zu den vorherigen Projekten beruht ATLAS nicht auf der Ausnutzung von trivialer Parallelität. Die Laufzeitumgebung ist als Middleware zwischen Applikation und Betriebssystem angelegt, das baumartige Ausführungsmodell kann für bestimmte Applikationen, die eine solche Struktur unterstützen, sehr effizient sein. Als Beispiel sind in [7] die Berechnung von Fibonacci Zahlen als worst-case und ein Renderprogramm für Bilder (Pov-Ray) implementiert.

Durch die adaptive Parallelisierung des work stealing wird eine hohe Skalierbarkeit erreicht, die sich auf mehrere Millionen Knoten anwenden lassen soll [7].

3.1.4 Legion

Legion [100, 71, 70] ist seit 1993 ein Projekt des *Department of Computer Science* der *University of Virginia* und mehreren Partnern. Es ist das ausgereifteste Metacomputing-System derzeit und verfügt bereits über mehrere *TestBeds* [13].

Legion bietet auf verschiedenen Plattformen transparenten Zugang zu verteilten Ressourcen an, die zusammen einen virtuellen Hochleistungsrechner darstellen. Diese Ressourcen können beispielsweise aus Rechnern, Bibliotheken, Simulationen und Kameras bestehen [33]. Auf diesen verteilten Ressourcen werden Dienste wie eine Verteilungsstrategie, Datenmanagement, Fehlertoleranz und Sicherheitsmechanismen angeboten. Das *Legion* Laufzeitsystem (LRTL, *Legion* Runtime Library [46]), das jeweils auf dem lokalen Betriebssystem aufsetzt, vermittelt dabei die von der Applikation benötigten Ressourcen und sorgt für die Einhaltung der lokalen Sicherheitsrichtlinien.

An Anforderungen aus 2.2 hat sich *Legion* die Folgenden zum Ziel erklärt:

Institutsautonomie: Ressourcen, die zur Verfügung gestellt werden, bleiben weiterhin unter lokaler Kontrolle.

Erweiterbarkeit: Dienste können von Objekten erweitert werden, da mit dem Wachsen des Systems neu benötigte Eigenschaften nicht vorhergesehen werden können.

Skalierbarkeit: Es gibt keine zentrale Ressourcenverwaltung, das *Legion* System selber ist verteilt.

Transparenz: Dem Benutzer bleibt die Komplexität des Gesamtsystems verborgen.

Eindeutiger Namensraum: *Legion* benutzt einen eindeutigen, persistenten Namensraum (*context space*).

[13] Unter dem *Legion* System laufende Verbünde von Rechnern, auf denen Applikationen aufgesetzt werden können.

Sicherheit: *Legion* implementiert ein eigenes Sicherheitssystem.

Fehlertoleranz: Während des Betriebs ausfallende Ressourcen werden durch dynamische Rekonfiguration vor dem Benutzer und der Anwendung verborgen.

Für die Entwicklung wurden folgende Bedingungen formuliert:

- Es werden keine Verwaltungsrechte auf den Gastrechnern benötigt.
- Es werden keine Teile des Betriebssystems ausgetauscht.
- *Legion* beruht auf aktuellen Netzinfrastrukturen und Protokollen.

Legion ist objektorientiert, wobei die Basisobjekte sowie Basisfunktionalitäten von Objekten wie Erstellung, Löschung und Aktivierung vorgegeben sind.

Zur Leistungssteigerung von Anwendungen bietet *Legion* zwei Möglichkeiten an: Zum einen soll gewährleistet werden, dass für serielle Programme der günstigste Knoten zur Berechnung gefunden wird. Es stehen als Leistungsmessprogramme Anwendungen aus den Spec-Benchmarks [34, 35] zur Verfügung, oder es kann die eigene Anwendung auf verschiedenen Knoten getestet werden.

Zum anderen wird die parallele Ausführung unterstützt. Hierfür werden von den *Legion* Bibliotheken die Funktionalitäten von PVM [154, 60, 80] und MPI [116] nachgebildet (vorhanden Applikationen müssen zusammen mit diesen Bibliotheken neu gebunden werden). An Programmiersprachen für die parallele Ausführung werden MPL (Mentat Programming Language, [75, 74]) und FORTRAN [45] unterstützt. Das *Legion* System ist in MPL geschrieben.

3.1.4.1 Architektur

Legion besteht aus unabhängigen Objekten, die durch gegenseitige, asynchrone Methodenaufrufe miteinander kommunizieren. Hierfür wird aus ortsunabhängigen Identifikatoren auf der Anwenderseite (LOID, *Legion* Object Identifier, siehe Abbildung 3.4 [68]) ortsabhängige Identifikatoren (LOA, *Legion* Object address) generiert, über die die Applikation auf die Objekte zugreifen kann. Diese enthalten sowohl die physikalische Adresse des Objektes, als auch einen zugehörigen RSA Schlüssel und ein Zertifikat nach *X.509* [32] . Entfernte Methodenaufrufe werden durch eine *Legion* eigene *Message Passing* Bibliothek realisiert.

Abbildung 3.4: Format einer *LOID*

Aufgrund der sehr großen Anzahl von möglichen Objekten innerhalb des Systems können Objekte automatisch in einem persistenten Zustand auf permanentem Speicher ausgelagert werden (OPR Objekt, Object Persistent Representation, [49]). Diese inaktiven Objekte können durch OPA Objekte (Object Persistent Address) lokalisiert werden, welches das ausgelagerte Objekt bei Bedarf in einen aktiven Zustand, d.h. in einen laufenden Prozess, versetzt.

Legion definiert eine Reihe an Basisklassen (Core Objects, [101]), die von jeder Anwendung benötigt werden. Diese unterteilen sich in

Hosts: Objekte von diesem Typ repräsentieren Prozessoren und implementieren das Prozessmanagement. Hierdurch wird die Heterogenität von vielen verschiedenen Knoten verborgen.

Vaults: Vaults repräsentieren persistente Speicher, die den Zustand von inaktiven Objekten halten.

Binding agents: Diese Objekte binden eine *LOID* an eine *LOA*.

Implementation Object: Diese Objekte stellen die Implementierung eines *Legion* dar, indem sie als ausführbare Datei verfügbar sind und ein zugehöriges Objekt instanziieren können.

Auch die Basisklassen von *Legion* spezifizieren nur die Funktionalität und Schnittstellen der Basisobjekte. Diese sind von Benutzern erweiterbar und auch ersetzbar.

Die Prozessverteilung innerhalb eines *Legion* Systems geschieht in den folgenden Schritten:

1. Alle Ressourcen (Hosts, Vaults) werden in einer Ressourcen Datenbank gesammelt.
2. Der Prozess, der für die Prozessverteilung zuständig ist (Scheduler) fragt in dieser Datenbank nach mehreren Möglichkeiten, bestimmte Anforderungen zu erfüllen.
3. Aus dieser Antwort werden mehrere mögliche Prozessverteilungsalternativen der Anwendung angeboten.
4. Die Anwendung (oder der Prozess, der für die Anwendung die Prozessverteilung vornimmt), kommuniziert daraufhin mit den Objekten der Ressourcen und belegt diese.
5. Diese Information wird von der Prozessverteilung zur Datenbank weitergeleitet.

Die Prozessverteilung (Scheduler) kann auch von dem Benutzer oder innerhalb der Anwendung implementiert werden. Dies kann zur Leistungssteigerung ausgenutzt werden. Die letzte Entscheidung, ob eine Ressource genutzt werden kann oder darf, liegt jedoch in dem *Legion* System immer in den Richtlinien der Ressource selber verankert. Damit wird dem Besitzer der Ressource, Instituten oder Firmen, die Autonomie gewährleistet.

3.1.4.2 Sicherheitsmechanismen

Eines der Hauptentwicklungsziele von *Legion* war die Erhaltung der Sicherheit auf den beteiligten Knoten für deren Besitzer und die Sicherheit der Daten innerhalb einer Anwendung für den Benutzer.

Der Eigentümer eines *Legion* Objektes vergibt nicht fälschbare Berechtigungen. Diese Berechtigungen werden bei einem Methodenaufruf übergeben und geprüft. Desweiteren können noch generelle Erlaubnisse an bestimmte Gruppen vergeben werden. Da Objekte häufiger indirekt über eine Kette an Aufrufen angesprochen werden, kann der Aufrufer sogenannte *Credentials*, Berechtigungslisten, dem Aufruf mitgeben. Diese Liste wird bei allen folgenden Aufrufen weitergeleitet, so dass das Zielobjekt die Prüfung auf Berechtigung vornehmen kann.

Jedes *Legion* Objekt beinhaltet eine Liste der eigenen Rechte und der Methodenaufrufe, die es erlauben kann (*access control lists*). Zur Überprüfung der Rechte eines Aufrufers kann es eine Methode implementieren, die *MayI* heißt. Alle weiteren Methodenaufrufe werden über *MayI* geprüft, ob sie die entsprechenden Rechte haben.

MayI wird in der *Legion* Laufzeitbibliothek abgearbeitet, die auch eine Standardmethode zu *MayI* anbietet. Diese Standardimplementierung kontrolliert Rechte, indem sie die Berechtigungslisten, die mit dem Aufruf übermittelt worden sind, mit den lokalen Zugriffsrechten vergleicht. Dieser Standardalgorithmus kann jedoch leicht ersetzt werden [168, 47], indem in den *LRTL* Protokollstapel neue Protokollebenen zur Laufzeit eingefügt werden können. Dadurch lassen sich Kryptografie sowie andere Methoden mit den *MayI* Funktionalitäten einbauen.

Kommunikation kann mit einem asymmetrischen Schlüsselverfahren ebenfalls innerhalb des *LRTL* abgewickelt werden. Hierbei wird der öffentliche Schlüssel aus dem Namen des Objektes abgeleitet. Somit kann ein Objekt die Nachrichten, die es verschickt, sowohl mit dem aus dem Namen des Zielobjektes erzeugten Public Key verschlüsseln, als auch seine eigene Nachricht mit dem privaten Schlüssel signieren und damit eine Authentifizierung realisieren.

Mit oben aufgeführten Techniken lassen sich in *Legion* Systemen Sicherheitsrichtlinien verwirklichen. Einzelne Domänen (Institute oder Firmen) können so Ressourcen dem *Legion* System zur Verfügung stellen und dennoch verhindern, dass fremde Objekte, die auf domäneneigenen Rechnern Ressourcen in Anspruch nehmen, Zugriff auf Administratorebene auf diese Objekte haben. Ebenso lassen sich auf diese Weise durch Rechteänderungen einzelne Rechner aus dem Ressourcenpool für eigene Anwendungen herausnehmen und sie dabei dennoch innerhalb des *Legion* Systems zu belassen. Diese Autonomie innerhalb einer Domäne soll die Akzeptanz bei Ressourcenanbietern erhöhen.

3.1.4.3 Anwendungen

Als Zielanwendungen für global verteilte Anwendungen für *Legion* werden eine Reihe von Anwendungsklassen angeführt.

- Einfache entfernte Ausführungen zusammen mit der Möglichkeit, die entfernten Programme mittels eines Werkzeuges (*remote make*) zuvor zu übersetzen. Diese Anwendungen profitieren von der objektorientierten verteilten Umgebung und der Ressourcenverwaltung. Anpassungen an die eigenen Applikationen sind dabei nur in geringem Umfang nötig.

- Anwendungen, die einen großen Suchraum unabhängig voneinander bearbeiten, können ebenfalls von der *Legion* Umgebung profitieren. Bei Problemen dieser Art ist lediglich ein Prozessverteilungsmechanismus (siehe auch Abschnitt 3.1.1.2) notwendig, der die Aufgaben über die *Legion* Ressourcen verteilt.

- Allgemeine Anwendungen, die wenig kommunizieren. Von der *University of Virginia* implementierte Beispiele sind Monte Carlo Simulationen und die Modellierung neuronaler Netzwerke.

- Mehrere verschiedene Anwendungen, die miteinander kooperieren.

Im Laufe der Entwicklung von *Legion* haben sich mehrere Institute und Firmen zu sogenannten *TestBeds* zusammengeschlossen. Innerhalb dieser *TestBeds* finden Anwendungen durch die *Legion* Infrastruktur eine Metacomputing-Umgebung vor.

Aktueller Stand ist das *Vanet TestBed*, das an der *University of Virginia* mit Institutsrechnern eine Metacomputing-Umgebung realisiert und das *Centurio TestBed* [103, 99] mit einer Konfiguration von 384 Prozessoren (256 Pentium 2, 400 MHz und 128 DEC Alpha, 533 Mhz), welches über eine Spitzenleistung von über 240 Gigaflops verfügt. Die Kommunikation erfolgt über ein zum Teil dediziertes Netzwerk von 100 Megabit und 1.28 Gigabit Myrinet.

Auf dieser Konfiguration laufende Anwendungen sind unter anderem Wetter Modellierungen, Monte-Carlo Simulationen, Simulationen zu chemischen Gasphasenabscheidungsprozessen und einigen mehr.

3.1.4.4 Bewertung

Das *Legion* Projekt ist das am weitesten fortgeschrittene Metacomputing-Projekt, das bereits eine große Anzahl an Anforderungen für weitverteiltes Rechnen erfüllt. Besonders das Thema der Ressourcen- und Datensicherheit ist weit entwickelt und genügt den Anforderungen für Domänenautonomie im kommerziellen Einsatzbereich.

Durch den Verzicht auf eine echte plattformübergreifende Programmiersprache sind im Bereich der Anwendung jedoch Einschränkungen auf die Portabilität gegeben, dieses gilt es in der Entwicklung von Applikationen zu berücksichtigen.

3.1.5 Globus

Globus [63] ist ein Projekt des *Argonne National Laboratory* und der *University of Southern California* von FOSTER und KESEELMAN [53].

Globus soll einen Schritt in die Richtung eines *Computational Grid* (siehe Definition 2.10) realisieren, indem die hierfür notwendigen Dienste entwickelt, eine prototypische Testumgebung auf diesen Diensten aufgebaut und schließlich Anwendungen diese Umgebung nutzen sollen.

Hierzu werden verteilte Ressourcen über eine Schnittstelle dem Benutzer zur Verfügung gestellt. Dabei liegt das Ziel nicht ausschließlich in einer Leistungssteigerung der Anwendung, sondern, wie es die Transparenz eines *Computational Grid* erfordert, auch in der Bereitstellung davon unabhängiger Ressourcen wie Datenbanken, Datenarchive, Visualisierungsgeräten und Instrumenten, die nicht lokal zur Verfügung stehen. Aufgrund der abstrakteren Sichtweise, die ein *Computational Grid* auf ein Metacomputing-System bietet, verfügt *Globus* über Dienste, die die Ausführung der Applikation vollkommen vor dem Anwender verbergen. Das beinhaltet sowohl den Ort der Ausführung als auch die dazu verwendete Architektur. Der Anwender braucht sich somit um die optimale Ausführungsumgebung nicht zu kümmern, das *Globus* System wählt die Architektur autonom aus, abhängig von den Attributen mit denen ein Entwickler die Anwendung versehen hat.

3.1.5.1 Dienste

Globus definiert eine abstrakte Metacomputing-Maschine, die die grundlegenden Dienste zur Verfügung stellt. Diese Dienste stehen im *Globus Toolkit* zur Verfügung, aus denen sich Dienste einer oberen Schicht zusammensetzen. Als Ziel wird eine Schnittstelle für Anwendungen geplant, AWARE (Adaptive Wide Area Resource Environment), die alle Metacomputing-Anforderungen flexibel zur Laufzeit erfüllen kann.

Als Basisdienste für die abstrakte Metacomputing-Maschine sind im *Globus Toolkit* folgende Funktionalitäten definiert:

Ressourcenverwaltung: Dieser Basisdienst (GRAM, *Globus Resource Allocation Manager*) lokalisiert und initialisiert lokale Ressourcen und kennzeichnet sie als belegt für die entsprechenden Applikationen. Ein GRAM Prozess nimmt dabei Anfragen entgegen, die in einer ressourcenbeschreibenden Sprache (RSL, *Resource Specification Language*) formuliert sind. Diese Anfrage wird dann für den lokalen Ressourcenmanager (unterstützt werden unter anderem LSF, NQE, und Condor) umformuliert. Durch diesen Mechanismus ist kein Eingriff in die lokale Ressourcenverwaltung nötig, es genügt einen GRAM Prozess für jeweils einen Ressourcenverwalter aufzusetzen, welcher eine beliebige Anzahl von Knoten verwalten kann.

Prozessverwaltung: Die Prozessverwaltung erzeugt die zum Starten eines Prozesses notwendige Umgebung, erzeugt gegebenenfalls ausführbare Dateien und startet diese mit den notwendigen Parametern. Bei Prozessterminierung werden die entsprechenden Informationen weitergeleitet.

Authentifikation: Dieser Dienst verifiziert die Benutzer und die Ressourcen, um für übergeordnete Sicherheitsdienste die notwendigen Informationen bereit-

zustellen. Er richtet sich nach dem GSS (*Generic Security Service*) [106] beziehungsweise [107], der eine gegenseitige Authentifikation (Server,Client) mit Passwort oder Berechtigungslisten, sowie verschlüsselte Kommunikation und Signaturen ermöglicht. Ein allgemeines Sicherheitskonzept für Metacomputing-Architekturen ist von FOSTER in [57] beschrieben. Dieses Konzept ist in Abschnitt 2.5 beschrieben.

Kommunikation: Hier werden die für die Kommunikation notwendigen Mechanismen implementiert. Mögliche Kommunikationsarten, die als Dienst angeboten werden, sind Message Passing, remote procedure calls, verteilter gemeinsamer Speicher und lokale Kommunikation über Streams. Der *Globus* Basisdienst für die Kommunikation stützt sich hierbei auf die *Nexus* [56] Bibliothek ab. Diese bietet abstrakte Punkt zu Punkt Verbindungen, auf denen sich regelbasiert Protokolle, Datenkompression und Kryptographie definieren lassen.

Datenzugriff: Dieser Dienst bietet entfernten Zugriff auf persistenten Speicher wie Dateien, Datenbanken oder ausgelagerte Objekte. Hierfür wird auch eine Schnittstelle für Basisfunktionalitäten bereitgestellt (RIO,*remote input/output*), diese stützt sich auf ADIO (*abstract-device interface for parallel I/O*) [159] und erweitert diese um transparenten Zugriff und einen globalen Namensraum.

Information: Für die Informationsgewinnung über den aktuellen Zustand (siehe auch Abschnitt 2.2) existiert innerhalb von *Globus* MDS (*Metacomputing Directory Services*). Dieser Dienst soll durch eine einheitliche Schnittstelle allen Objekten in einer Metacomputing-Umgebung ermöglichen, Informationen aller Art über den Zustand des Systems abzufragen. MDS stützt sich hierbei auf LDAP (*Leightweight Directory Access Protocol*) ab [170, 169], welches ein Abfrage- und Änderungsprotokoll auf einer datenbankähnlichen Struktur (*Directories*) mit einem eindeutigem Namensraum definiert. In die *Directories* werden dann durch Unix Skripte Informationen abgelegt, welche dann durch *Globus* Clients abgerufen werden können. Hierzu muss ein LDAP Serverprozess und zusätzlich die benötigten Werkzeuge zum Eintragen der Informationen in die Datenbank auf jedem Client aufgesetzt werden. Die Gesamtheit aller LDAP-Server stellt das MDS System dar.

Für eine optimierte Ausführung lassen sich sowohl von Diensten, die über den Basisdiensten implementiert sind, als auch über Anwendungen Anforderungen an die zu benutzenden Ressourcen formulieren. Diese Angaben lassen sich über eine Schnittstelle abwickeln, die regelbasierte Anfragen und Kommandos akzeptiert. Somit können Anwendungen sich beispielsweise die Nutzung von Parallelrechnern oder die Abwicklung der Kommunikation über ATM Protokolle anstatt TCP zusichern lassen. Ebenso lässt sich die Eigenschaft einer Ressource (Quality of Service) zusichern, im Falle einer Verletzung dieser Zusicherung wird die Anwendung über einen Callback informiert, welche dann gegebenenfalls die Berechnungen abbrechen oder andere Ressourcen benützen kann.

Zur Optimierung der Kommunikation bietet *Globus* auf jedem beteiligten Client das Werkzeug *Gloperf* [98]. *Gloperf* misst die Bandbreite von Host zu Host Verbindungen mit einer auf *netperf*[88] basierten Bibliothek. Hierbei wird, ähnlich wie in dem *LsD-System* aus Kapitel 4, eine TCP Verbindung mit *TCP_STREAM* ausgemessen, d.h. inklusive des erhöhten Verwaltungsaufwandes für eine TCP basierte Verbindung. Bei Testmessungen über einen längeren Zeitraum ergaben sich hierbei gute Resultate bezüglich der Vorhersage der Qualität der Verbindung, auch bei seltener vorgenommenen Verbindungstests. *Gloperf* bietet zudem eine Unterteilung der sich gegenseitig prüfenden Clients durch die Vergabe von Gruppen. Dies hat den Vorteil einer geringeren Netzbelastung durch die Tests (die Netzlast steigt quadratisch mit $O(n^2)$ bei einer vollständigen Prüfung von allen Clients). Latenzbasierte Vorhersagen der Netzqualität sowie das Testen bei Bedarf (on demand), wie sie bei dem *LsD-System* vorgenommen werden können, sind für die Zukunft angedacht.

3.1.5.2 Anwendungen

Bisher wurden zwei große *Globus* Infrastrukturen aufgebaut: *I-Way*[39] und *GUSTO (Globus Ubiquitous Supercomputing Testbed).* Beide Projekte umspannen mehrere Standorte und Institute, die über verschiedene Netzwerke, teilweise dedizierte Hochgeschwindigkeitsnetze, verbunden sind. Als teilnehmende Rechner sind sowohl eine Reihe an Arbeitsplatzrechnern, als auch Parallelrechner angeschlossen.

Das *GUSTO* Testbed umfasst dabei mehrere Institute in USA, Hawai, Schweden und Deutschland mit ca. 330 angeschlossenen Knoten, die über 3600 Prozessoren mit einer Gesamtleistung von 2 TFlops/s verfügen [14].

Die für *GUSTO* entwickelten Applikationen umfassen Visualisierungen von wissenschaftlichen Simulationen, Echtzeitanalyse von Satellitendaten und verteilte Parameterstudien.

3.1.5.3 Bewertung

Im Gegensatz zu Legion, das ein objektorientiertes Programmiermodell anbietet, welches die meisten (Metacomputer-) Dienste vor dem Anwender kapselt, geht *Globus* hier den Weg, dem Anbieter eine Reihe von vorhandenen Techniken anzubieten. Die angebotenen Techniken können dann unter der Vorgabe von einheitlichen Schnittstellen genutzt werden (siehe Abbildung 3.5).

Als großer Vorteil bei der Entwicklung der oben genannten Anwendungen hat sich erwiesen, sich auf ein breites Spektrum an bereits vorhandenen Werkzeugen abzustützen. Dies ermöglicht es, bei schon bestehenden Anwendungen, die diese Werkzeuge benützen, sie mit nur geringfügigen Änderungen auf die *Globus* Umgebungen zu portieren.

[14] Stand 1998 [54]

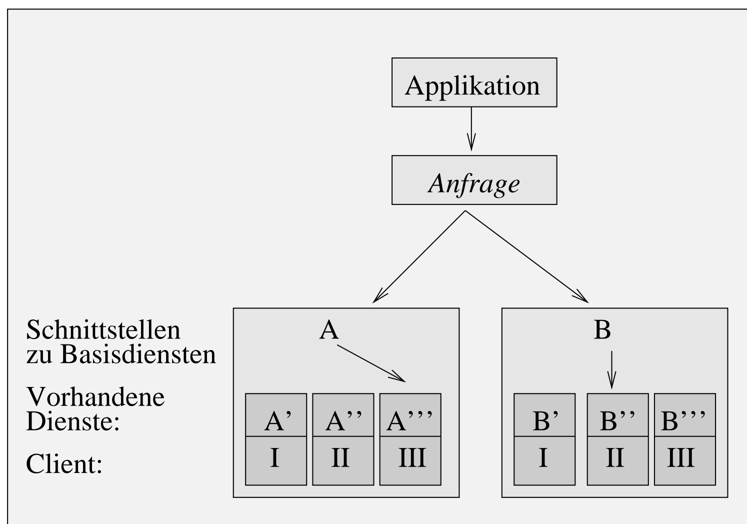

Abbildung 3.5: Prinzip der Nutzung fremder Dienste unter *Globus*

3.1.6 Weitere Systeme

3.1.6.1 HeNCE

HeNCE (Heterogeneous Network Computing Infrastucture) [8, 10] stammt bereits aus dem Jahre 1992 und kann als einer der Vorläufer einer Metacomputing-Umgebung gesehen werden. Ziel ist es, durch ein Netzwerk verbundene, heterogene Rechner gemeinsam als einen virtuellen Supercomputer nutzen zu können.

HeNCE ist eine Programmierumgebung, die es ermöglicht ein paralleles Programm graphisch darzustellen. Dabei bedeutet jeder Knoten im Graph auszuführenden Programmcode, jede Kante stellt die Abhängigkeiten dar. Wird die Ausführung des Programms anhand dieser Repräsentation innerhalb der Umgebung gestartet, wird der Graph ausgewertet und die unabhängigen Programmteile auf entfernten Ressourcen ausgeführt. Um auf heterogenen Knoten rechnen zu können, muss dabei von dem Entwickler an jedem Knoten, der auf mehreren Architekturen ausgeführt werden soll, eine entsprechende Implementierung vorliegen. Weiterhin muss der Entwickler eine Matrix zur Verfügung stellen, welche der geschätzten Last des Programmcodes auf der jeweiligen Architektur entspricht. Diese Matrix wird dann zur Prozessverteilung genutzt. Die gesamte Entwicklung kann unter der graphischen Benutzungsoberfläche erfolgen.

HeNCE setzt auf der parallelen Programmierbibliothek *PVM*[9] auf, deren lokale Prozesse die Synchronisation, Kommunikation und Prozessgenerierung übernehmen.

Als einer der ersten Vertreter von Metacomputing-Umgebungen hat dieses Projekt noch eine Reihe gravierender Einschränkungen, nämlich die geschätzten Angaben zur Last der Programmteile, der hohe Aufwand für verschiedene Architekturen, Programmcode zur Verfügung zu stellen und vor allem der sehr eingeschränkt

mögliche Parallelismus, der zum einen statisch festgelegt, zum anderen durch einen Abhängigkeitsgraph ausgedrückt werden muss. Dennoch ist damit eine verteilte Umgebung möglich, die bei geeigneten Anwendungen mit hohen Parallelismus durchaus hohe Leistungssteigerungen mit geringem Aufwand ermöglicht.

3.1.6.2 ParaWeb

ParaWeb[18], 1996 von der *University of York Department of Computer Science* entwickelt, beschreibt einen ähnlichen Ansatz wie das *LsD-System*. Auch *ParaWeb* stützt sich auf die Eigenschaft von Java, um einfache verteilte Ausführungen auf heterogenen Architekturen zu erreichen und um den Benutzer von der Applikationsinstallation, Übersetzung und der Fehlersuche zu trennen. Auch hierbei sollen die im Internet oder in lokalen Netzwerken verteilten Ressourcen genutzt werden, indem eine Umgebung zur Verfügung gestellt wird, die Java Threads auf die Maschinen verteilt und ausführt.

ParaWeb verwirklicht zwei Ansätze, um dieses Ziel zu erreichen.

JPCL (Java Parallel Class Library) ist eine Klassenbibliothek die mit einer unveränderten Java Virtuellen Maschine zusammenarbeitet. Diese bietet dem Entwickler eine Klasse `RemoteThread` an, welche das davon abgeleitete Objekt auf entfernten Rechner instanziiert, nachdem ein freier Server vermittelt wurde. Daraufhin wird der Bytecode der abgeleiteten Klasse zu diesem geschickt und ausgeführt. Als Kommunikationsmöglichkeiten stehen den Java Threads einfaches Message Passing(*send* und *receive*) zur Verfügung.

JPRS (Java Parallel Runtime System) ist eine modifizierte Java Virtuelle Maschine, die eine Obermenge der ursprünglichen JVM darstellt. Java Threads haben hier die Möglichkeit, auf entfernten Rechner instanziiert zu werden. Hierbei wird der erzeugte Bytecode auf die entfernte Maschine gesendet und dort ausgeführt. Da dies für die Anwendung transparent ist, werden auch alle von diesem Thread referenzierten Objekte als Kopie verschickt. Hierdurch ist eine parallele Ausführung von Anwendungen, die Threads benutzen, möglich, ohne diese modifizieren oder neu übersetzen zu müssen. Die Kommunikation zwischen den Threads wird hier über verteilten gemeinsamen Speicher verwirklicht, um die Semantik von Threads nicht implizit zu verändern (siehe auch Abschnitt 4.4). Um die Datenkonsistenz zwischen diesen Kopien, die auf mehreren Rechnern zu liegen kommen können, zu gewährleisten, wurde ein Release Konsistenzmodell mit Hilfe der Java Mechanismen `synchronized` implementiert.

Durch diese beiden Modelle können sowohl neue Applikationen Threads entfernt ausführen, als auch bestehende Applikationen, die bereits Threads benützen, parallel ausgeführt werden. Problematisch bei der Benutzung des *JPRS* ist jedoch, dass ungewollte, verborgene Kommunikation zu entfernten Objekten, die zudem noch durch das Konsistenzmodell synchronisiert werden müssen, zu großen Leistungs-

einbußen führen kann,welche durch das stark objektorientierte Modell verborgen und schwer zu lokalisieren sind.

Mit der parallelen Klassenbibliothek wurden in einem Netz aus 20 SparcStation 10 ein Leistungszuwachs (Speedup) von 18 bei einer Matrixmultiplikation erzielt.

3.1.6.3 NetSolve

NetSolve ist eine Entwicklung der *Universtiy of Tennessee* von DONGARRA und CASANOVA. Es ist eine Umgebung, die sich als Entwicklungsziel die Nutzung von Metacomputern für Anwendungen aus dem wissenschaftlich-technischen Bereich gesetzt hat. Hier wird der Fokus auf problembezogene Software (*Problem Solving Environment*) gesetzt. Die Schnittstelle zwischen Anwendung und verteilten Ressourcen soll für den Anwendern nicht sichtbar sein, für ein gestelltes Problem wird transparent eine geeignete Hardware gesucht und das Problem entsprechend auf die Ressourcen verteilt.

In dem *NetSolve* System ist keine Hierarchie der Clients vorgesehen. Ressourcenanfragen werden an `netagents` gestellt, die eine beliebige Teilmenge der zur Verfügung stehenden Rechenserver vermitteln können. Die Rechenserver verfügen dabei über die entsprechenden wissenschaftlich-technischen Bibliotheken (LAPACK, LINPACK, BLAS), um Anfragen lokal berechnen zu können.

Ein Problem wird hierbei dem `netagent` als 3-Tupel `(name, input, output)` angegeben, wobei `input` und `output` Listen des Typs `(Objekt, Daten)` sind. Objekte können hierbei jeden Fortran Datentyp annehmen.

Zur Problemformulierung auf dem lokalen Rechner kann hierbei sowohl eine Schnittstelle innerhalb einer Standardshell als auch eine Schnittstelle zu MATLAB benutzt werden, mithilfe derer sich beispielsweise ein lineares Gleichungssystem, dass entfernt gelöst werden soll folgendermassen formulieren lässt: $x = netsolve('ax = b', a, b)$. Der NETSOLVE AGENT sucht dann eine möglichst optimale Ressource für dieses Problem. Hierbei werden durch *NetSolve* periodisch gemessene und aktualisierte Werte von Netzlast und Serverlast zur Beurteilung herangezogen [29]. Der Server, welche die geringste geschätzte Ausführungszeit aufweisen kann, wird dann als Ressource für das Problem ausgesucht.

Die Daten für die geschätzte Ausführungszeit werden periodisch aktualisiert, hierbei wird die Ungenauigkeit durch die zeitlichen Abstände der Messungen in Kauf genommen, da eine erhöhte Last durch häufige Netzmessungen vermieden werden soll.

Fehler innerhalb des Systems können von Clients entdeckt werden und an den *netagent* weitergeleitet werden. Daraufhin können Netsolve Server neu gestartet werden. Im Falle von Ausfällen während der Ausführung von Anwendungen werden diese auf anderen Server neu gestartet. Dies wird solange wiederholt, bis das Ergebnis zur Verfügung steht.

3.1.6.4 WOS – Web Operting System

WOS [95, 162] (Web Operating System) geht einen anderen Weg, um globales Rechnen zu ermöglichen. Es wird bei diesem Projekt als gegeben angesehen, dass es angesichts der Breite möglicher Anwendung und der damit verbundenen notwendigen Dienste kein Metacomputing-System geben kann, welches diese Anforderung gegenwärtig und vor allem in Zukunft erfüllt [96].

Da es nicht ausreicht, ein System für global verteiltes Rechnen nur a priori festgelegte Dienste anbieten zu lassen, wird die Anforderung abgeleitet, dass diese Dienste dynamisch angeboten werden und dass das zugrundeliegende System diese verschiedenen Dienste über Versionen verwaltet. Die Anzahl aller dieser Dienste in verschiedenen Versionen wird `Warehouse` genannt. Darüberhinaus sind diese wegen der wachsenden Bedeutung der Kommunikation in einem solchen System als zentrale Komponente ausgelegt.

Auf jedem beteiligten Knoten implementiert *WOS* einen *WOS-Node* welcher die Client, Server und Broker Dienste anbietet. Zur Kommunikation setzen zwei Protokolle auf TCP/IP auf. Das *WOSRP* (WOS Request Protocol) übernimmt die Anfrage an die Clients um Informationen über unterstützte Versionen bestimmter Dienste einzuholen. Die Kommunikation mit diesen Diensten übernimmt *WOSP* (WOS Protocol), ein generisches Protokoll, welches Merkmale eines Dienstes zunächst erfragen und diese Informationen mittels einer Syntaxanalyse nutzen kann.

Mit diesen Eigenschaften stellt *WOS* die Mittel zur Verfügung, um einen Metacomputer für wissenschaftlich-technische Anwendungen zu ermöglichen. Es implementiert eine Schicht innerhalb eines Metacomputers, die über die eigentliche Nutzung der Ressourcen keine Annahmen macht, die Kommunikation mit diesen Ressourcen jedoch auf eine leicht zu verwaltende Schnittstelle realisiert, die auch die stetigen Wechsel in Weitverkehrsnetzen berücksichtigt.

3.2 Zusammenfassung und Bewertung

3.2.1 Vor- und Nachteile

In dem sich rasant entwickelnden Gebiet des Metacomputing kann eine abschließende Bewertung nur unvollständig sein. Der Stand der Forschung konzentriert sich zur Zeit nicht in einer homogenen Umgebung, sondern in einer Spezialisierung der Dienste, deren Kombination über eine dezentrale Verwaltung den Metacomputer bildet, dabei koordiniert die dezentrale Verwaltung die untereinander kommunizierenden Dienste.

In Tabelle 3.1 sind einige Eigenschaften der Systeme im Vergleich aufgeführt. Alle Systeme sind auf heterogenen Architekturen lauffähig, erreichen dies jedoch mit unterschiedlichen Mitteln und unterschiedlichem Aufwand. Lediglich die Systeme, die Java als Mittel für plattformunabhängige Ausführung (2. Spalte) einsetzen, erreichen dies ohne aufwändige und fehlerträchtige entfernte Übersetzung und Fehlersuche.

System	Java	Skalierbar	Dienste (i/e)	Parallelismus
distributed.net	Nein	Ja	Nein	trivialer Parallelismus
SETI	Nein	Ja	Nein	trivialer Parallelismus
Javelin	Ja	Nein	(i/-)	sehr grobkörnig
JET	Ja	Nein	Nein	sehr grobkörnig
Atlas	Ja	Ja	(i/-)	feinkörnig
Legion	Nein	Ja	(i/e)	feinkörnig
Globus	Nein	Ja	(i/e)	feinkörnig
HeNCE	Nein	Nein	Nein	grobkörnig
ParaWeb	Ja	Nein	(i/-)	(feinkörnig)
NetSolve	(Ja)	Ja	(i/e)	(feinkörnig)
WOS	–	Ja	(-/e)	(feinkörnig)

Tabelle 3.1: Metacomputing-Systeme und deren Techniken im Vergleich

Skalierbarkeit (3. Spalte) ist bei sehr komplexen Systemen und bei sehr einfachen Systeme zu finden. Dies liegt bei den sehr komplexen Systemen an der aufwändigen, dezentralen Verwaltung, bei den einfachen Systemen an der geringen Auslastung der zentralen Komponenten[15], die sich meist auf das Verteilen der Aufgaben beschränkt. Diese Komponente bildet hierbei jedoch immer einen kritischen Punkt bezüglich Ausfallsicherheit (“single point of failure”).

Prinzipiell lässt sich die Komplexität eines Metacomputer-Systems an den angebotenen Diensten abschätzen. Diese können extern sein, also Dienste, die die Anwendungen in Anspruch nehmen können, oder intern, für die eigene Verwaltung der Metacomputer (4. Spalte, “i” für ausschließlich interne Dienste, “e” für externe Dienste).

Anhand des Aufbaus des Metacomputer-Systems und seiner Merkmale zur Prozessverteilung und Kommunikationsmethode lassen sich bezüglich der notwendigen Kommunikation geeignete Anwendungen dafür bestimmen (rechte Spalte).

Allgemein lassen sich folgende Vorteile von Metacomputer-Architekturen formulieren:

Geschwindigkeitssteigerung

Verteilte Anwendungen können in mehreren Aspekten durch Metacomputing-Umgebungen gewinnen. In erster Linie ist hier die Geschwindigkeitssteigerung maßgeblich, die im wissenschaftlichen Rechnen traditionell an erster Stelle stand, denen sich die anderen Punkte unterzuord-

[15]Diese Einordnung der Skalierbarkeit ergibt sich aus den tatsächlichen Leistungsdaten, die bis jetzt keinerlei Engpässe an der Serverkomponente feststellen haben lassen, nicht aus der tatsächlichen Struktur, da diese Client/Server Systeme diesbezüglich keine skalierbare Architektur aufweisen.

nen hatten [16]. Diese wird durch die parallele Ausführung der Anwendung und durch die Wahl eines leistungsstärkeren Zielsystems erreicht. Eine indirekte Leistungssteigerung erreicht man durch die Möglichkeiten, größere Probleme lösen zu können oder mehrere Probleme einer Klasse gleichzeitig innerhalb eines Homotopieverfahrens[17] zu berechnen.

Ökonomie

Es ist weiterhin für einen Benutzer eines Metacomputers ökonomischer, hohe Rechenleistung verteilt in Anspruch zu nehmen, als lokal einen Superrechner vergleichbarer Leistung zu installieren. Äquivalent zu der Nutzung von mehreren Arbeitsplatzrechnern gilt dies auch im Maßstab von verteilt stehenden Rechnern oder von der Nutzung von mehreren, in verschiedenen Instituten aufgestellten Parallelrechnern. Dies trifft auch für angeschlossene Peripheriegeräte wie Visualisierer oder verschiedene Messgeräte zu, die lokal nicht vorhanden sind. Dies demonstriert auch *Globus* (Abschnitt 3.1.5) in [181]. Berücksichtigt man Projekte wie *SETI* (Abschnitt 3.1.1) und ähnliche, erreicht auch freiwillige Teilnahme von Heimrechnern, die nur zeitlich begrenzt an ein Netz angeschlossen sind, sehr hohe Rechenleistungen bei sehr geringen Kosten.

Softwareentwicklung

Auch auf der Ebene der Softwareentwicklung können sich Vorteile ergeben. Während die Nutzung von Parallel- oder Vektorrechnern sehr proprietäre Entwicklungsumgebungen und herstellereigene Kommunikationsmechanismen benötigen, und andererseits die Nutzung von Verbünden von Arbeitsplatzrechnern Werkzeuge und Bibliotheken mit sehr niedrigem Abstraktionsniveau erzwingen, also sehr fehlerträchtig sind, ist dies bei den meisten Metacomputer-Architekturen nicht der Fall. Hier wird oft eine klare Struktur zwischen Objekten, Prozessen und der Kommunikation erzwungen oder eine objektorientierte Umgebung gefordert. Dies macht die Portierung auf verschiedene Umgebungen einfach und verbirgt die dahinterliegende Komplexität.

Trotz dieser Vorteile, vor allem der theoretisch möglichen Geschwindigkeitssteigerung, müssen Metacomputing-Systeme ihre Tauglichkeit in kommerziellen Umgebungen oder zur Nutzung als transparenten Zugang zu Hochleistungsrechnern noch zeigen.

Das Ziel, durch eine solche Architekur große Leistungsgewinne für einzelne, darauf spezialisierte Anwendungen zu erzielen, kann als erreicht bezeichnet werden. Auch zeigen die verschiedenen Ansätze in vielen Bereichen, dass die Technik zur Realisierung zum großen Teil bereits vorhanden ist.

Es gibt bis jetzt noch kein einheitliches System, das auf alle Anwendungen abzielt, dies ist auch zukünftig nicht abzusehen und aufgrund der verschiedenen möglichen

[16] Auch nur so lässt sich der langanhaltende Erfolg von einer, hinsichtlich den Entwicklungs- und Wartungskosten, sehr teuren Programmiersprachen wie FORTRAN erklären.

[17] Berechnung mehrerer Parametersätze mit einem Datensatz.

Anwendungen auch nicht Ziel der Forschung. Daher muss man entscheiden, welche Anwendung man vorliegen hat und demnach das Metacomputer-System wählen, das die Anforderungen am weitesten erfüllt. *Legion* selber ist objektorientiert entworfen und richtet sich an objektorientierte Anwendungen. *NetSolve* ist eher ein "Problem Solving Environment", welches sich an Nutzer wendet, die ein konkretes mathematisch Problem lösen wollen, dass die lokalen Rechenkapazitäten übersteigt, ohne sich mit der eigentlichen Verteilung oder der eigentlichen Zielplattform befassen zu müssen.

BUYYA [24] kategorisiert Metacomputing-Systeme nach möglichen Anwendungszielgruppen, die anhand der Struktur des Systems bestimmt worden sind (Tabelle 3.2).

System	**Einsatz**
Javelin	Java basierte Anwendungen
JET	Java basierte Anwendungen
SETI	proprietäre Anwendung
distributed.net	proprietäre Anwendung
Legion	Objektorientierte Anwendungen
Globus	Anwendungen der Grand Challenges
NetSolve	Anwendungen aus dem Ingenieursbereich

Tabelle 3.2: Einsatzmöglichkeit verschiedener Systeme

3.2.2 Anforderungen für kommerzielle Systeme

Aus dem vorhergehenden Abschnitt lässt sich zusammenfassen, dass Metacomputer-Systeme nur teilweise für kommerzielle Anwendungen geeignet sind. Die Anforderungen in diesem Bereich werden durch hohe Leistung, Sicherheitsmechanismen und intelligente Prozessverteilung abgedeckt, doch es existieren hier weitere Anforderungen, die aus den lokalen Gegebenheiten von Firmen bestimmt werden.

- Für Firmen ist der Entwicklungsaufwand und damit die Kosten einer Anwendung für ein globales Metacomputer-System wie *Legion* oder *Globus* sehr hoch. Darüber hinaus wird die Software langfristig an das Bestehen einer solchen Infrastruktur gebunden. Diese Bindung an externe, universitäre Projekte, die nur teilweise oder überhaupt nicht beeinflusst werden können, wird als hohes unternehmerisches Risiko angesehen.

- Metacomputing-Projekte, für die Anwendungen mit niedrigeren Entwicklungskosten erstellt werden können, sind oft nicht für Applikationen des wissenschaftlich-technischen Rechnens geeignet. Anwendungen mit einfacher Client/Server Kommunikation und Aufgabenverteilung, wie sie Projekte

wie *JET* oder *Javelin* unterstützten, sind einfacher und billiger zu entwickeln oder anzupassen, jedoch gibt es wenige parallele Anwendungen mit derartig starker Parallelität.

- Die Struktur firmeninterner Rechnernetze ist oft anders ausgebildet, als dies allgemeine Metacomputer-Projekte in ihrer Kommunikation voraussetzen. Es existieren hier nicht n verschiedene Komponenten an n verschiedenen Standorten (bezügliche der Anbindung), sondern diese sind an wenigen, verteilten Orten gebündelt. Die Latenzzeiten dieser Standorte untereinander unterscheiden sich signifikant von den Latenzzeiten innerhalb dieser Standort. Dies sollte bei der Prozessverteilung berücksichtigt werden.

- Entwicklungskosten der Anwendung sind ein maßgeblicher Anteil der Finanzkalkulation bei kommerziellen Nutzern von Metacomputer-Strukturen. Diese Investitionen langfristig zu erhalten erfordert portable Software und flexible Einsatzmöglichkeiten. Daher wäre es wünschenswert, eine Anwendung auch ohne die Metacomputer-Middleware auf internen Rechnern aufsetzen zu können.

Diese Anforderungen werden von bestehenden Systemen nur unzureichend abgedeckt. Das im folgenden Kapitel vorgestellte Metacomputer-System implementiert ein für solche Infrastrukturen geeignetes Metacomputer-System, das sehr geringe Anforderungen an die Anwendung stellt und eine latenzoptimierte Verteilung von Prozessen ermöglicht. Der Mechanismus der verwendeten Technik lässt sich als Modul zur optimierten Prozessverteilung in bestehende Metacomputer-Systeme integrieren. Anwendungen, die das System zur Verteilung und Optimierung verwenden, lassen sich mit nur geringfügigen Änderungen an Umgebungen ohne das System anpassen.

4. LsD - Ein System zur Optimierung von verteilter Programmausführung in Weitverkehrsnetzen

Im folgenden Kapitel wird zu Beginn allgemein die Situation von Metacomputer-Architekturen bezüglich ihrer Anwendungen beschrieben und ihre Mängel für kleinere Installationen oder für kommerzielle Nutzer aufgezeigt. Damit wird die Entwicklung einer leichtgewichtigen, optimierenden Umgebung motiviert, die mit neuen Eigenschaften für diese Umgebungen geeignet ist. Es folgt ein Überblick über die theoretische Prozessverteilung, anschließend wird das mögliche Einsatzgebiet des entwickelten Systems, *LsD* (*L*atency *s*ensitive *D*istribution, Prozessverteilung unter Berücksichtigung der Latenzen der rechnenden Knoten), beschrieben. Es folgt eine detaillierte Beschreibung der Architektur und der Implementierung des Systems. Hierbei wird auch besonders auf die Programmiersprache Java eingegangen. Dies ist zum einen durch die unübliche wissenschaftlich-technische Umgebung motiviert, zum anderen durch den enormen Vorteil, der für das verteilte Rechnen durch die Wahl einer geeigneten Programmiersprache erzielt wird.

Die Evaluierung des *LsD-Systems* erfolgt daraufhin in zwei Schritten. In dem praktischen Schritt wird die Technik überprüft, d.h. die Annahme, dass eine intelligente Prozessverteilung, basierend auf den Kommunikationslatenzen des Netzes, die Ausführungszeit von verteilten Anwendungen verkürzen kann. In dem dann folgenden Kapitel (Kapitel 5) werden die theoretischen Annahmen über einen Einsatz im industriellen Umfeld anhand der Umgebung und der Anwendungen der Firma *Tecoplan AG* überprüft.

4.1 Motivation

Die rasante Entwicklung auf dem Gebiet der Metacomputer hat in den letzten Jahren sehr viele Systeme hervorgebracht.

Einige davon zielen auf sehr große, global verteilte Umgebungen und erfüllen einen Großteil der formulierten Bedingungen. Diese stellen jedoch implizit auch hohe An-

forderungen an die Anwendungen, die diese Dienste benutzen wollen und vielmals auch müssen.

Andere Systeme sind im Aufbau wesentlich weniger komplex, bieten dafür aber auch nur eine eingeschränkte Anzahl an Diensten an. Vor allem die Annahme über die Kommunikation der Anwendungen schränkt die Anzahl der Zielapplikationen erheblich ein, da hier im Normalfall sehr häufig Client zu Client Kommunikation notwendig ist.

Alle vorgestellten Metacomputing-Systeme bieten entweder, teilweise proprietäre[1], Technologien, die eine verteilte Infrastruktur ermöglichen, beziehungsweise unterstützen, oder sie verwirklichen eine komplette Systemumgebung, innerhalb derer die Applikation eingebettet werden muss.

Diese Situation hat für Applikationen, die eine solche Umgebung nutzen wollen, folgende zwei Möglichkeiten der Realisierung:

- Die Applikation selbst implementiert die Funktionalitäten, die für Metacomputing-Umgebungen wichtig sind.
- Die Applikation wird auf die Umgebung abgestimmt und ermöglicht so eine effiziente Nutzung der angebotenen Ressourcen. Dadurch bindet sich die Anwendung an die Umgebung, d.h. ist nur innerhalb dieser lauffähig.

Das hier vorgestellte *LsD-System* (Latency sensitive Distribution) versucht die Abhängigkeit der Anwendung zu einer bestimmten Umgebung und der aufwändigen Implementierung der Dienste zu vermeiden, indem es sich als Middle-Middleware zwischen den komplexen und aufwändigen Metacomputern und den einfachen, reinen Client/Server Architekturen platziert.

Das *LsD-System* ist eine leichtgewichtige Umgebung, die vollständig in der plattformübergreifenden, objektorientierten Programmiersprache *Java* entwickelt worden ist. Sie ist zwischen dem Betriebssystem und der Anwendung platziert, wie in Abbildung 4.1 dargestellt. Java unterstützt eine parallele Anwendung durch die Verwaltung und Verteilung von Prozessen ohne die Kommunikation zwischen diesen zu beeinträchtigen oder zusätzlichen Entwicklungsaufwand notwendig zu machen. Die an dem *LsD-System* beteiligten Rechner werden für die Anwendung transparent verwaltet. Die Verteilung der Prozesse auf diese Knoten erfolgt unter Berücksichtigung der Netzwerktopologie bezüglich der Latenzen der Kommunikationswege und sorgt auf diese Weise für eine optimale Ausführungsumgebung für die Client zu Client Kommunikationen.

Definition 4.1 *Latenz*

> Die Latenzzeit ist die Zeit, die eine Nachricht ohne Nutzdaten in einem Kommunikationssystem K von einem Knoten A zu einem Knoten B und zurück benötigt □

[1] proprietäreTechnologie ist dadurch bestimmt, dass die zum Einsatz kommenden Mittel (Software, Hardware, Kommunikationswege, Protokolle) nicht auf verbreiteten, frei zugänglichen Standards aufsetzten, sondern Eigenentwicklungen bzw. mit Kosten verbundene Fremdentwicklungen beinhalten, die die allgemeine Nutzung verhindern.

Die Merkmale des hier vorgestellten *LsD-Systems*:

- Leichtgewichtige Umgebung.
- Latenzoptimierte Ausführung.
- Automatische Prozessverteilung.
- Als Subsystem zu einer bestehenden Metacomputer-Umgebung geeignet, die vielfältigere Dienste anbietet.
- Geringe Entwicklungskosten.

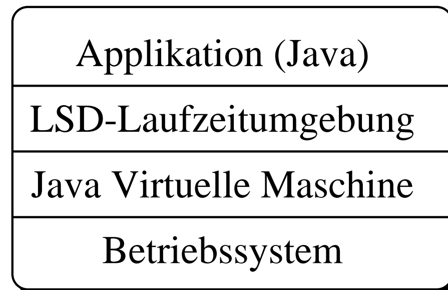

Abbildung 4.1: Schichtenmodell bei LsD Benutzung

Das *LsD-System* macht keine Annahmen über die Art der Anwendungskommunikation. Dies lässt der Anwendung die nötige Flexibilität in der Wahl eines geeigneten Protokolls um im wissenschaftlich-technischen Bereich die nötige Kommunikationsleistung zu erzielen, die an die jeweilige Aufgabe angepasst werden kann. Hier lassen sich optimierte Bibliotheken wie *JPVM* [48] verwenden oder transparente, objektorientierte Schichten wie *CORBA* implementieren. Das *LsD-System* stellt diesbezüglich einen Namensdienst zur Verfügung, der die Informationen über die Position eines Prozesses auf Anfragen zur Verfügung stellt.

Die Art der generischen Prozessverteilung und der autonomen Anwendungskommunikation hat zudem noch den Vorteil, dass sich, ist die Ausführung der Anwendung angestoßen, das *LsD-System* für die Anwendung transparent verhält. Dies hat zur Folge, dass die Anwendung auch ohne das *LsD-System*, ohne optimierte Prozessverteilung, lauffähig ist. Zudem lassen sich, da die Anwendungskommunikation autonom abläuft, auch Rechner in die Anwendung einbinden, die außerhalb der *LsD-System* Umgebung stehen.

Versorgen viele verschiedene Applikationen das *LsD-System* mit Aufgaben, und werden diese Aufgaben je nach Belastungen der einzelnen rechnenden Clients verteilt, kann man mit gewissen Einschränkungen[2] von einem Stapelverarbeitungsbetrieb sprechen.

[2]Das *LsD-System* verteilt intern die übertragenen Jobs nicht zwingend in der Reihenfolge ihres Eintreffens, sondern in der Reihenfolge, die die Applikation ihnen durch die Vergabe von Identifikatoren zuweist.

Latenzoptimierte Verteilung von Arbeitslast ist bisher in Verbünden aus Arbeitsplatzrechnern oder in dedizierten parallelen Hochleistungsrechnern nicht verwirklicht worden. Dies war bisher offensichtlich unnötig, da die Kommunikationszeiten innerhalb dieser Umgebungen sehr kurz und vor allem homogen sind[3]. Unterschiede in den Latenzzeiten sind im Vergleich zu den Rechenleistungen der einzelnen Knoten zu vernachlässigen[4]. Stand der Forschung auf diesem Gebiet ist die Prozessverteilung gemäß der vorhandenen Auslastung auf dem Zielrechner. Schwankt diese Last im Laufe der Berechnungen oder arbeitet ein interaktiver Benutzer an diesem Rechner, können die Migrationskosten [148, 113] mit der geringen Leistung oder mit der Beeinträchtigung des Benutzers abgewogen werden. In Metacomputing-Umgebungen sind diese Latenzzeiten jedoch sehr inhomogen. Zusätzlich zu der Last beziehungsweise der Rechenleistung der Knoten erhält man einen weiteren Faktor, der bei der Prozessverteilung berücksichtigt werden muss, insbesondere bei häufigen Kommunikationen mit wenig Nutzdaten. Diese Problematik innerhalb Metacomputing-Umgebungen behandelt das *LsD-System* mit einer latenzsensiblen Verteilung von Prozessen.

Die Ebene der Parallelität innerhalb des *LsD-System* ist auf einer hohen Ebene angesiedelt. Man unterscheidet hier die Granularität der Parallelität, die von Maschineninstruktionen bis zu Programm Ebene variieren kann. Dies hat direkten Einfluss auf den Entwicklungsaufwand, der bei der Anwendung zu erbringen ist. Tabelle 4.1 gibt hierüber Aufschluss.

Ebene	*Granularität*	*Entwicklungsaufwand*
Programm	hoch	gering
Prozess	hoch	hoch
Prozedur	mittel	sehr hoch
Block	niedrig	niedrig
Anweisung	sehr niedrig	keiner

Tabelle 4.1: Parallelitätsmodell

Die Verteilung von Threads durch *LsD* erfordert eine Parallelisierung, die zwischen der Prozess- und der Prozedurebene liegt. Die hohen Entwicklungskosten, die mit dieser Parallelisierungsart bisher verbunden waren, liegen zum Hauptteil an der Kommunikation und der Synchronisation der Prozesse. Dies ist jedoch bei wissenschaftlich-technischen Anwendungen unabdingbare Vorraussetzung, die sich aus dem Grundsatz der Parallelverarbeitung zur Geschwindigkeitssteigerung ergibt. Abhilfe schafft hier eine objektbasierte Schicht, die die Kommunikation vor dem Entwickler verbirgt, dieses ist jedoch in der Regel mit hohen Geschwindigkeitseinbußen verbunden, und wird deshalb in wissenschaftlich-technischen Rahmen vermieden.

[3] Als Gegenargument seien hier die großen Bemühungen der vergangenen Jahre im Bereich der Parallelrechner erwähnt, die Prozessor zu Prozessor Verbindungswege innerhalb dieser Rechner bezüglich ihrer Länge und der Anzahl der Zwischenstationen zu optimieren.

[4] D.h. ein Knoten rechnet in der Zeit des Latenzunterschiedes wenig.

Mit *LsD* vergleichbare Umgebungen sind mit Einschränkungen *Javelin* und *JET*, die ebenfalls reine Java Umgebungen sind. Die Verwaltungsstruktur ist bei diesen ebenfalls nach Client und Server unterteilt, jedoch erzwingen diese im Unterschied zu *LsD* auch der Anwendung dieses Muster auf. *ParaWeb* platziert ebenfalls Java Threads auf verteilten Rechnern, indem es eigene Mechanismen zur Prozessmigration anbietet.

Mit dem Projekt *HeNCE* hat *LsD* gemeinsam, dass auch hier zusätzlich Angaben über die Anwendung durch Angabe einer Lastmatrix gegeben werden muss, wonach dann die Verteilung durchgeführt wird.

Die Architektur des LsD Systems ähnelt dem Ausführungsmodell ATLAS aus 3.1.3 (client daemon, manager). Auch das hierarchische *work stealing* Verfahren in ATLAS bewirkt indirekt eine Verbesserung der Kommunikationszeiten durch Ausnutzung der Lokalität, da Anfragen nach zu berechnenden Threads erst die lokal naheliegenden Knoten erreichen. Diese naheliegenden Knoten können in einem lokalen Netzwerk einer Firma oder eines Institutes liegen, während übergeordnete Knoten Ressourcen außerhalb des lokalen Netzwerkes darstellen. Im Vergleich zu *LsD* stellt *Atlas* jedoch hohe Anforderungen an die Struktur der Anwendung. ATLAS-Anwendungen müssen baumartige Strukturen aufweisen, ansonsten funktioniert das *work stealing* nicht. *LsD* funktioniert prinzipiell mit jeder parallelen Anwendung, die aus kommunizierenden Threads besteht.

Diese Vergleich sind im Tabelle 4.2 zusammengefasst. Die Spalte der Dienste ist analog zu Tabelle 3.1 aufzufassen.

System	**C/S Architektur (Metacomputer)**	**C/S Architekur (Anwendung)**	**Thread Parallelismus**	**optimierte Verteilung**	**Dienste**
Javelin	Ja	Ja	(applets)	Nein	(i/-)
JET	Ja	Ja	Ja	Nein	Nein
ParaWeb	Nein	Nein	bedingt	Nein	(i/-)
HeNCE	Nein	Nein	Nein	Ja[5]	Nein
Atlas	Ja	bedingt	Ja	bedingt	(i/-)
LsD	Ja	Nein	Ja	Ja	Ja

Tabelle 4.2: Systeme im Vergleich

4.2 Theoretischer Ansatz zur Prozessverteilung

Wie im vorhergehenden Abschnitt erläutert, ist eine der zentralen Eigenschaften des *LsD-System* die optimierte Verteilung von parallelen Prozessen. Ein virtueller, verteilter Rechner K, der aus dem *LsD-System* gebildet wird, ist bestimmt durch

$$K = \{S, k_1, k_2, \ldots, k_n\},$$

wobei S das *LsD* Steuerungsmodul ist (der *MasterDaemon*, siehe Abschnitt 4.4), k ein Rechenknoten gemäß Definition 2.14, und n deren Anzahl.

Im Normalfall ist das die Kombination einer Java Virtuellen Maschine und des darin ablaufenden Prozesses des *LsD* Client. Die Anzahl der Rechenknoten ist von den physikalisch vorhandenen Prozessoren oder Maschinen unabhängig , d.h. mehrere Rechenknoten können auf einen Prozessor oder auf eine Maschine (mit eventuell mehreren vorhandenen Prozessoren) abgebildet werden.

Die Darstellung einer parallelen Anwendung wird in diesen Betrachtungen auf die laufenden Prozesse beschränkt. Die Anwendung P wird bestimmt durch

$$P = \{I, p_1, p_2, \ldots, p_n\},$$

wobei I die Initialkomponente[6] der Applikation ist und nicht verteilt wird, und p_i ein Prozess ist, der entfernt ausgeführt wird.

Weiter werden die Latenzen (siehe Definition 4.1) zwischen allen Knoten und die Verbindungshäufigkeiten zwischen den Prozessen bestimmt.

$$L(k_i, k_j) = l_{i,j} \qquad \forall \quad i,j \leq n \quad \wedge \quad i \neq j$$

$$C(p_i, p_j) = c_{i,j} \qquad \forall \quad i,j \leq m \quad \wedge \quad i \neq j$$

Hierbei gibt $L(k_i, k_j)$ die Latenzzeit von Knoten i zu Knoten j an, und $C(p_i, p_j)$ die Kommunikationshäufigkeit von Prozess i zu Prozess j. Hierbei liegen die Verbindungshäufigkeiten in einer abstrakten Metrik vor, die in keinerlei Einheit angegeben wird. Dies bedeutet keinerlei Einschränkung der Allgemeinheit, da eine Gewichtung bei der Prozessverteilung unabhängig von den angegebenen Werten ist. Dies erlaubt eine flexible Einschätzung durch den Entwickler.

Gesucht wird nun eine injektive und nicht notwendigerweise surjektive Abbildung von Prozessen zu Knoten

$$f : P \longrightarrow K,$$

so dass gilt:

$$\sum_{i,j}^{m} f(c_{i,j})c_{i,j} \quad = \quad min\left(\mathcal{V}\left(c_{i,j} l_{s,t}\right)\right) \tag{4.1}$$
$$\forall \quad i,j \leq n; \quad s,t \leq m, \quad \wedge \quad i \neq j; s \neq t$$

Dabei ist vorausgesetzt, dass für nicht kommunizierende Prozesse p_i und p_j $C(p_i, p_j) = 0$ gilt und dass $\mathcal{V}\left(c_{i,j} l_{s,t}\right)$ die Menge der Permutationen der Summen

$$\sum_{(i,j)(s,t)}^{n,m} c_{i,j} l_{s,t}$$

[6]Der Teil der Anwendung, der dem *LsD-System* die zu verteilenden Threads übergibt.

darstellt. Jeder Prozess kommt auf im Laufe der Permutationen auf jedem Knoten zu liegen. Von jeder dieser Permutationen wird die Summe der Kommunikationslatenzen gebildet. Diese Summe über alle Kommunikationslatenzen (linker Teil der Gleichung) ist damit als das Minimum aller möglichen Platzierungen von Prozessen auf Knoten (rechter Teil) festgesetzt.

Bei diesem Optimierungskriterium gilt es zwei Dinge zu beachten:

- Die Abbildung ist unabhängig von der Auslastung der Maschinen und schließt Mehrfachbelegungen von Knoten aus. Die Beschränkung lässt sich in der Praxis allerdings technisch sehr einfach umgehen, indem mehrere Client Prozesse auf einen Knoten gestartet werden.

- Die Optimierung der Kommunikationszeit ist unabhängig von dem zeitlichen Ablauf der Kommunikationen und von den Rechenphasen zwischen den Kommunikationen in der Anwendung. Diese können zu einer suboptimalen Verteilung führen, obwohl die oben formulierten Bedingungen eingehalten wurden. Dies kann in manchen Fällen innerhalb der Anwendung durch Angabe von verschobenen (überbewerteten) Parametern korrigiert werden.

Die oben genannten Permutationen sind $\|P\|$-Permutationen über $\|K\|$ ohne Wiederholungen, was einen Lösungsraum von $\frac{(\|K\|)!}{((\|K\|-\|P\|)!}$ öffnet. Dies ist ein np-vollständiges Problem und kann nicht in akzeptabler Rechenzeit gelöst werden. Hier gilt es mit einer annehmbaren Heuristik eine brauchbare Lösung zu errechnen, deren Berechnungszeit weit unterhalb der zu erwartenden Geschwindigkeitssteigerung der Optimierung liegt.

Eine Reduzierung des Suchraumes lässt sich mit der plausiblen Annahme erreichen, dass der am häufigsten kommunizierende Prozess auf dem best angebundenen Knoten zu liegen kommt, und alle weiteren Prozesse hierzu optimal zu liegen kommen. In dem Algorithmus in Abbildung 4.2, der hinzugefügte Prozesse auf den verbleibenden Knoten rotiert bis ein Minimum der Gesamtkommunikationszeit erreicht wird, wird dies realisiert.

Nicht berücksichtigt werden hier mögliche und auch wahrscheinliche Nebeneffekte, wenn früh platzierte Prozesse spätere gut angebundene Knoten blockieren und dies wegen fehlendem Rücksubstituieren nicht rückgängig gemacht werden kann[7]. Da jedoch die Prozesse mit absteigender Kommunikation verteilt werden, sind die Auswirkungen als nicht erheblich einzustufen, da aufgrund der Sortierung gilt, wenn $P(i,j) = m$ und $P(k,l) = n$ mit $n < m$ und für einen später zu verteilenden Prozess $P(k,i) = r$ gilt, dann muss $r < m$ und $r < n$ gelten.

Aufgrund der ständig vorhandenen Latenztabellen des *LsD-Systems* wird hier ein anderer Algorithmus gewählt, der von der Annahme ausgeht, dass Unterschiede der Latenzwerte nicht zufällig verteilt sind, sondern dass es Gruppen von gleich angebundenen Knoten gibt. Diese Konstellation ist eine der Zielumgebungen des *LsD-Systems* und ist häufig bei verteilten Standorten eines Instituts oder einer Firma anzutreffen. Eine weitere Annahme, die den Algorithmus in der Komplexität

[7] Führt man anschliessend eine Rücksubstitution aus, erhält man die Komplexität aus 4.1

```
Sortiere P absteigend
Sortiere K aufsteigend
Platziere Prozess P[1] auf Knoten K[1]
K = K\K[1]
P = P\P[1]

for i = 1 to |P|
    for j = 1 to |K|
        Platziere Prozess P[i] auf Knoten K[j]
        S[i] = Summe der Kommunikationen
        if     S[i] < min
        then   t = i; min = S[i];
    end

    Platziere Prozess P[i] auf Knoten K[t]
    K = K\K[t]
    P = P\P[i]
end
```

Abbildung 4.2: Suboptimaler Algorithmus zur Prozessverteilung

verringert, ist dadurch gegeben, dass zwischen Knoten (oder Gruppen von Knoten) konsistente Abhängigkeiten existieren. Dass heißt wenn

$$c_{i,j} = n \quad \wedge \quad c_{j,k} = m \quad \wedge \quad n \ll m$$

gilt, dann kann

$$c_{i,k} \gg n$$

gefolgert werden. Dies ist in der Praxis bei einen global verteilten Metacomputer nicht zwingend erforderlich, da verschiedene Vermittlungsstellen (Router) unterschiedliche Wegetabellen enthalten und diese zudem dynamisch sind. In einer begrenzten Anzahl an Standorten, wie es bei Instituten und Firmen der Fall ist, ist diese Annahme jedoch sehr wahrscheinlich.

Der diesen Bedingungen entsprechende Algorithmus ist implementiert und in Abschnitt 4.4 erläutert.

In der Literatur finden sich verschiedene Ansätze zur Optimierung der Prozessverteilung, die jedoch in der Regel keine Latenz berücksichtigen.

Das *netsolve* Projekt berücksichtigt die prognostizierte Rechenlast auf dem Zielknoten und die hierfür zur Verfügung stehende Bandbreite [29]. Daraus wird ein optimaler Zielknoten für das gestellte Problem bestimmt. Hier ist jedoch der Kommunikationsanteil des versendeten Problems und dessen Kommunikationspartner

nicht berücksichtigt. Im Falle von *LsD-System* kann eine Schnittstelle zu Werkzeugen, die die Rechenlast des Zielsystems bestimmen [130], diese fehlende Information zu dem Verteilungsalgorithmus hinzufügen.

Die Aktualisierung der Netzlastwerte im *LsD-System* kann direkt vor der Verteilung der Prozesse erfolgten. Dies bedeutet eine zusätzliche Verzögerung der Anwendung. Doch im Vergleich zu Knoten, deren Lasten sich weniger häufig ändern [29], ist dies bei Netzlasten höchstens bei dem Durchsatz zu erwarten. Die Latenzzeiten sind jedoch kurzfristigeren Schwankungen unterworfen. Da diese Schwankungen jedoch auch während der Ausführungszeit auftreten können, kann hier bei der optimalen Verteilung ein über einen längeren Zeitraum gebildeter Mittelwert diese kurzfristigen Schwankungen ausgleichen.

Das Werkzeug *Gloperf*[98] des Projektes *Globus* bietet eine Überwachung der Bandbreite und veranlasst dementsprechende Zuordnung von Ressourcen. Latenzen und Optimierung von häufig kommunizierenden Anwendungen sind zur Zeit noch nicht realisiert (siehe Abschnitt 3.1.5).

Ein Werkzeug zur Bewertung der Netzqualität bietet der Ansatz von OBRACZKA und GHEORGHIU [121], mit dem Ziel, eine Bewertungsgrundlage für Netzanwendungen zur Verfügung zu stellen. Hier werden mit dem Werkzeug *topology-d* mittels *ping* und *netperf* [88] alle Kommunikationspaare einer Gruppe von Rechnern gemessen und danach eine minimum-cost Topologie konstruiert, nach der Teile einer Anwendungen platziert werden können. Eine Einbettung dieses Systems in eine Metacomputing-Umgebung ist für das *Globus* Projekt innerhalb des MDS (*Metacomputing Directory Services*) angedacht.

4.3 Klassifikation der Zielumgebung

Der Einsatz einer parallelen Anwendung für ein Metacomputer-System hängt von vielen Faktoren ab, die in den vorhergehenden Abschnitten und Kapiteln erläutert wurden. Durch die Architektur der Anwendungen und deren Einsatzgebiet können Metacomputing-Umgebungen spezifiziert werden, die für die Anwendung am günstigsten sind. Im Falle der Spezifizierung von *LsD* wird der umgekehrte Weg gegangen: Aufgrund der Merkmale der *LsD* Umgebung und ihrer Vor- und Nachteile wird die Zielumgebung der Anwendung beschrieben und geeignete Anwendungsmerkmale definiert.

Obwohl einer Verwendung des *LsD* Systems in globalen Metacomputing-Umgebungen, die über sehr viele inhomogene Rechner verfügen und welche an sehr vielen verschiedenen Standorten positioniert sind, kein technischer Grund entgegenspricht, gibt es hier Systeme, die geeigneter sind. Dies liegt an der Client/Server Architektur des *LsD-Systems*, das nur eine bedingte Skalierbarkeit erlaubt (dies gilt für alle Client/Server basierten Metacomputer). Für ein solches Einsatzgebiet müsste *LsD* zudem mit Sicherheitsmechanismen versehen werden, die über eine

reine Verschlüsselung der Transferdaten hinausgeht[8]. *LsD* erlaubt hier die Abarbeitung von beliebigen Applikation, unterscheidet hier aber keine Zugriffsrechte. Diese müssten im Rahmen der neuen Sicherheitskonzepte von Java eingearbeitet werden.

Das *LsD-System* ist für, bezüglich der Rechneranzahl, limitierte Umgebungen im Rahmen von großen Instituten und Firmen entwickelt worden, die über verteilte Standorte verfügen. Dort lassen sich sehr große Probleme aus dem wissenschaftlich-technischen Bereich mit sehr geringem Entwicklungsaufwand durch *LsD* berechnen, ohne das die Kosten für dedizierte Parallelrechner anfallen. Das Ausführungsmodell von *LsD* unterstützt durch die latenzoptimierte Verteilung der Prozesse die verschiedenen Standorte, ohne dass diese komplexe Aufgabe in der Entwicklung der Anwendung berücksichtigt werden muss.

Zusammenfassung der spezifischen Umgebungsmerkmale:

- Limitierte Knotenanzahl.
- Keine hohen Sicherheitsanforderungen.
- Hohe Leistung notwendig.
- Geringe Entwicklungskosten.
- Unabhängige Anwendungsentwicklung. [9]
- Optimierung der Prozessverteilung aufgrund verschiedener Netzbereiche nötig.

In Tabelle 4.3 sind die für verschiedene Anwendungszenarien vorherrschenden Bedingungen zusammengefasst und die optimalen Anforderung für das *LsD-System* hervorgehoben.

4.4 Architektur und Implementierung

Das *LsD-Systems* besteht aus einer Hauptkomponente, dem *MasterDaemon* und aus einer beliebig hohen Anzahl[10] von *NodeDaemons*. Die Hauptkomponente ist für folgende Funktionalitäten verantwortlich:

- Verwaltung aller Applikationen, die sich zur Verteilung über eine Schnittstelle anmelden.

[8] Über eine solche Verschlüsselung verfügt *LsD* bisher nicht, jedoch ist die Verwirklichung dieser Funktionalität mit den Mitteln der JVM und deren Verschlüsselungalgorithmen trivial, da lediglich der Typ der eingehenden und ausgehenden Datenströme modifiziert werden muss. Erstreckt sich das benutzte Netzwerk über öffentlich zugängliche Netzbereiche, ist diese Art der Übertragung notwendig.

[9] *LsD* Anwendungen können auch ohne das *LsD-System* aufgesetzt werden, ohne dass eine aufwändige Portierung nötig ist.

[10] Diese Anzahl ist durch den vorhandenen Hauptspeicher auf dem Knoten des *MasterDaemon* beschränkt

Merkmal	Kleine Firma	Große Firma	verschiedene Standorte	Global
Rechenleistung	gering	**hoch**	**hoch**	sehr hoch
Heterogenität	gering	**mittel**	**hoch**	**sehr hoch**
Sicherheit	nein	**nein**	**ja**, bedingt	ja
Netzwerkbandbreite	sehr hoch	hoch	niedrig	niedrig
Netzwerklatenz	gering	gering	**hoch**	**sehr hoch**

Tabelle 4.3: Geeignete Einsatzmöglichkeit des *LsD-Systems* in verschiedenen Umgebungen

- Verwaltung aller Jobs (Java-Threads), die von den Applikationen über diese Schnittstelle übergeben werden.
- Verwaltung aller teilnehmenden *NodeDaemons*, die zu der verteilten Ausführung zur Verfügung stehen.
- Verwaltung und Aktualisierung einer globalen Latenztabelle.
- Berechnung des Algorithmus zur Prozessverteilung.
- Initialverteilung aller Jobs einer Applikation.
- Initiierung der verteilten Berechnung.
- Verwaltung und Aktualisierung eines Dienstes zur Lokalisation von Prozessen.

Die Komponenten, die auf den am Metacomputer beteiligten Rechnern installiert sind, die *NodeDaemons*, realisieren folgende Aufgaben:

- Entgegennahme und Verwaltung beliebiger Jobs von beliebigen Applikationen.
- Aktualisierung der lokalen Latenztabellen.
- Ausführen von einzelnen Jobs oder von allen Jobs einer Applikation.
- Versenden des aktuellen Status aller lokalen Jobs an den *MasterDaemon*.

Die Funktionalitäten werden von verschiedenen Java Klassen implementiert. Alle Module im Überblick und deren Zusammenspiel sind in Abbildung 4.3 zu sehen. Im folgenden wird auf die Aufgaben der einzelnen Module und deren gegenseitige Kommunikation eingegangen.

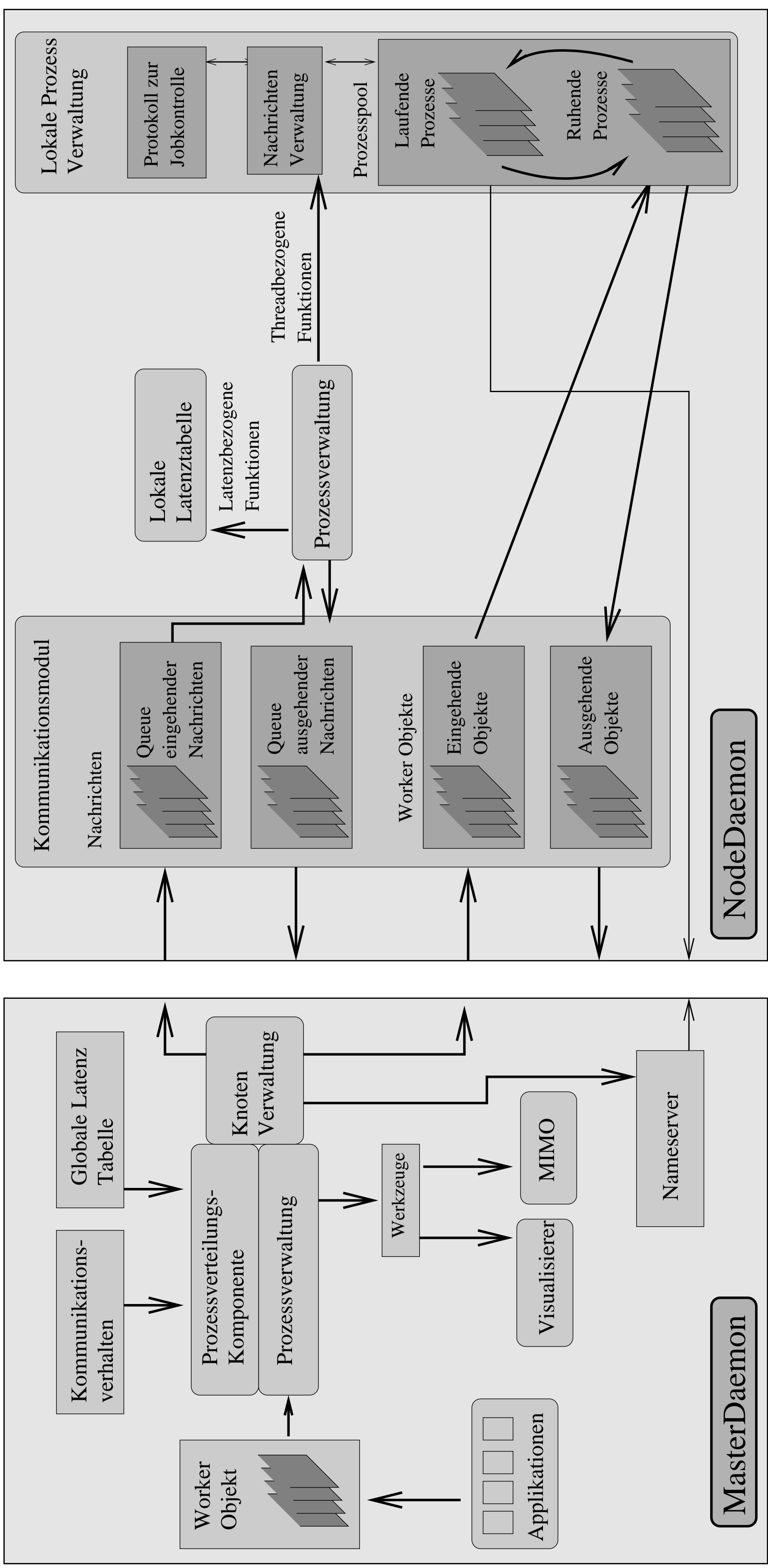

Abbildung 4.3: Module des *LsD-Systems* im Überblick

Jobverwaltung

Zentrale Komponenten innerhalb des MasterDaemons sind die Jobverwaltung und die Knotenverwaltung. Das *LsD-System* bietet Anwendungen über einen Socket eine Schnittstelle an, über die es eine Struktur zur Beschreibung der Applikation (Abbildung 4.10) sowie die zu verteilenden Prozesse der Applikation entgegennimmt. Hierzu wird ein separater Thread gestartet, der alle konnektierenden Applikationen verwaltet und die Informationen der Jobverwaltung und der Knotenverwaltung zur Verfügung stellt. Die Strukturen aller Applikation werden dabei in einer verketteten Liste verwaltet. Ist eine Applikation zur Ausführung markiert, wird der Algorithmus der Prozessverteilung gestartet und anschließend die Prozesse auf die *NodeDaemon* migriert.

Zusätzlich zu der automatischen Verteilung und Ausführung verfügt der *MasterDaemon* über eine Kommandozeilen Schnittstelle, über die in die Steuerung eingegriffen werden kann. Diese Steuerung kann zukünftig an eine entfernte grafische Benutzungsoberfläche gekoppelt werden.

Knotenverwaltung

Die Knotenverwaltung ist in einem eigenen Prozess (Java Thread) realisiert, der Verbindungsversuche vom *NodeDaemon* entgegennimmt. Die Adresse des Knotens des *MasterDaemon* wird über eine öffentlich zugängliche URL bekanntgegeben. Ist eine Anmeldung erfolgreich verlaufen, wird eine neue Struktur zur Verwaltung dieses neuen Knotens angelegt und in eine verkettete Liste aller *NodeDaemon* eingetragen. Zusätzlich werden die Informationen der lokalen Latenztabelle des neuen Knotens der globalen Latenztabelle des *MasterDaemon* hinzugefügt. Zu diesem Zeitpunkt ist der neue Knoten dem Metacomputer hinzugefügt und kann Applikationsprozesse entgegennehmen. Zur Ablaufsteuerung werden alle Verbindungen zu beteiligten Rechnern periodisch auf eingehende Nachrichten (*Events*) überprüft und diese von der entsprechenden Instanz verarbeitet. Hierzu existiert für jeden Knoten eine eigene eingehende und abgehende Nachrichtenqueue, die die Nachrichten zur weiteren Verarbeitung zwischenspeichert (Nachrichten oder *Event* gesteuerte Verarbeitung). Abbildung 4.4 zeigt schematisch die Knotenverwaltung des Moduls *NdMan*.

NodeDaemon

Die Komponente des *NodeDaemon* besteht aus den Modulen der Nachrichtenverwaltung, der Prozessverwaltung und der Steuerung. Zu dem Knoten migrierte Jobs werden in Joblisten verwaltet, die Informationen über die Prozessidentifikatoren und deren Zuordnung zu den Applikationsidentifikatoren beinhalten. Eine Liste enthält die ruhenden Jobs, die zur Abarbeitung anstehen, eine weitere Liste beinhaltet die Jobs, die sich in der Ausführung befinden. Ein migierter Prozess wird zuerst in die ruhende Jobliste eingeordnet, bis von dem *MasterDaemon* das Signal zur Ab-

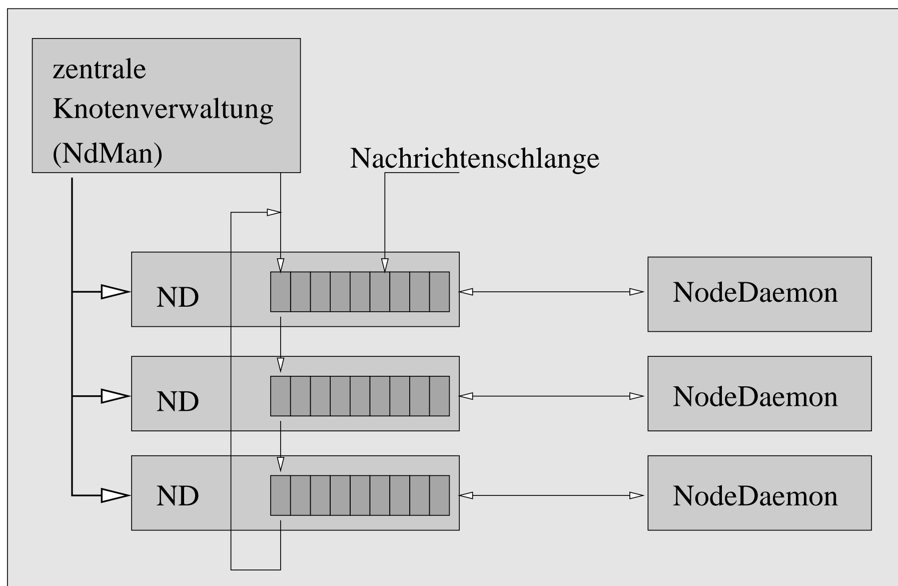

Abbildung 4.4: Knotenverwaltung innerhalb des *LsD-Systems*

arbeitung eines bestimmten Applikationsidentifikators empfangen wird. Alle Jobs, die dieser Applikation zugeordnet sind, werden daraufhin in einer Prozessstruktur gekapselt, in die Liste der laufenden Prozesse eingefügt und zur Ausführung gebracht. Auf die hier notwendige Kapselung eines Jobs in eine Prozessstruktur und die Unterscheidung zwischen diesen Zuständen wird im nächsten Abschnitt eingegangen. Ist ein Prozess abgearbeitet, sendet dieser über einen Rückrufmechanismus, der in der Schnittstelle spezifiziert ist, eine Nachricht an die Prozessverwaltung, die den Job in eine Liste der beendeten Aufträge einträgt und aus der Liste der laufenden Jobs entfernt. Neue Einträge in der Liste der beendeten Aufträge werden dem *MasterDaemon* mitgeteilt, anschließend veranlasst dieser die Löschung aus dieser Liste.

Prozesskapselung

Bearbeitungsaufträge einer Applikation werden als Jobs bezeichnet, solange sie sich in den Warteschlangen des *LsD-Systems* befinden. Werden sie gestartet und befinden sich in der Ausführung, wird in diesem Rahmen von einem *LsD* Prozess gesprochen. *LsD* Prozesse sind Threads, die eine Java Virtuelle Maschine innerhalb ihres Threadmodells[11] ausführen kann. Da die Notwendigkeit besteht, diese Auf-

[11] Das Java Threadmodell variiert mit der darunterliegenden Architektur. Es unterscheidet “green threads” und “native Threads”. Erstere sind von der JVM emulierte Threads, die nicht auf die reale Maschine abgebildet werden können, letztere sind die von der Architektur unterstützten Threads, die meistens effizient auf die Prozessor- und die Betriebssystemarchitektur abgebildet werden können. Zudem besteht bei den gängigen Implementierungen der JVM die Option, die Ausführung mit “green threads” erzwingen zu können.

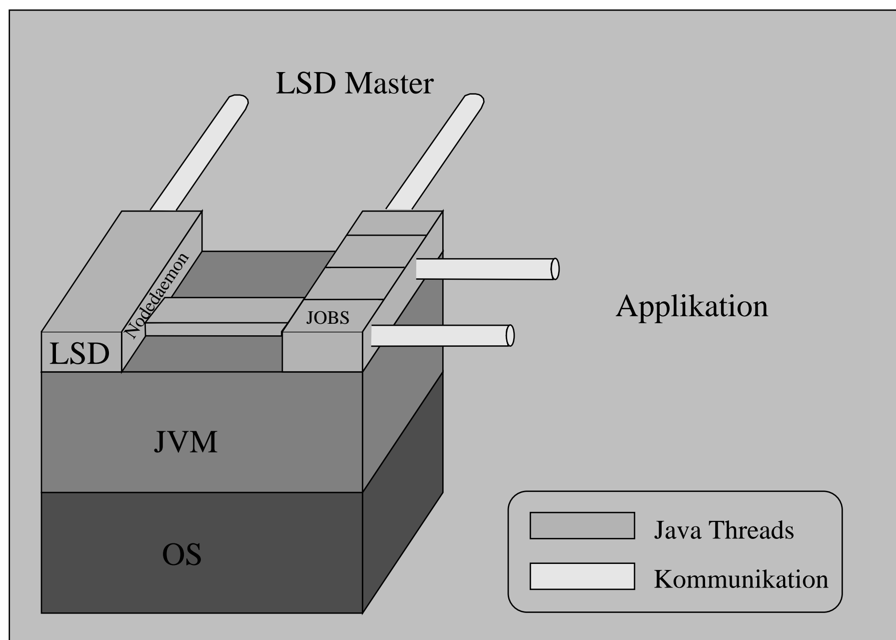

Abbildung 4.5: Schema eines LsD Knotens

träge zu anderen Knoten migrieren zu können, ist eine Kapselung erforderlich. Die Java Migration von Klassen erfolgt über das Prinzip der Serialisierung, diese unterstützt jedoch keine persistente Speicherung von Prozessinformationen, die bei dem Ablauf eines Threads anfallen und verwaltet werden müssen. Daher ist es notwendig, einen Applikationsauftrag erst zur Laufzeit und vor beziehungsweise nach der Migration mit einem Threadobjekt zu kapseln (siehe Abbildung 4.6.). Hierzu

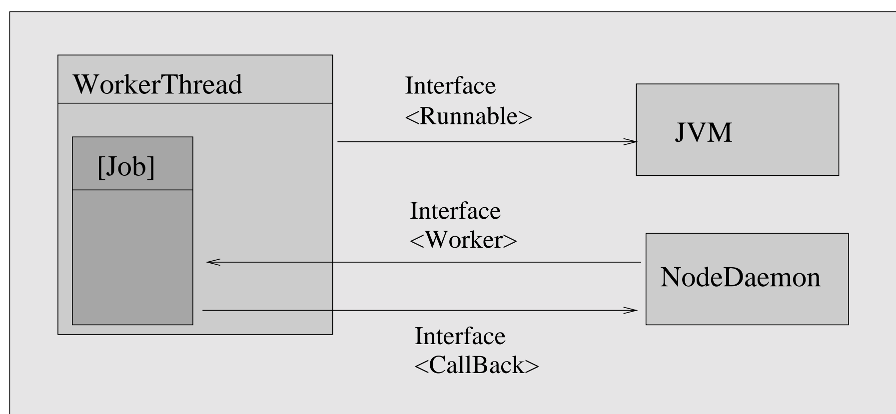

Abbildung 4.6: Prozesskapselung

müssen zwei Objektschnittstellen durch den Applikationsjob implementiert werden. Eine Schnittstelle realisiert die für einen Java Thread notwendigen Funktionalitäten

(*Interface Runnable*), die andere Schnittstelle ermöglicht dem *LsD-System* den Zugang zur Steuerung des Objekts (*Interface Worker*,diese Schnittstelle ist in Abbildung 4.7 zu sehen). Dieser Zugang ermöglicht eine Initialisierung des Prozesses, sowie eine postmortem Methode, um applikationsinterne Verwaltung zu ermöglichen (*init(); exit();*). Die Methode zum Lesen und Schreiben der Job- und Applikationsidentifikatoren ermöglichen dem *LsD-System* und der Applikation die Lokalisation der Objekte nach der Verteilung und deren Zuordnung zu Applikation beziehungsweise Knoten:

- setJobID();
- getJobID();
- setAppID();
- getAppID();

Der Zustand des Prozesses kann durch die Java Laufzeitumgebung erfragt werden, oder alternativ kann dem Prozess über die Schnittstelle *CallBack*, die von der Prozessverwaltung des *LsD-Systems* implementiert wird, eine Methode übermittelt werden, die bei Beendigung des Prozesses aufgerufen werden kann.

```
interface Worker
{
    void setJobID (int jobid);
    int  getJobID ();

    void setAppID (int appid);
    int  getAppID ();

    int start ();
    int stop ();

    int init (Object o);
    int exit ();

    void setFinishedCB(CallBack cb);

    boolean isRunning();
}
```

Abbildung 4.7: Kapselung von LsD Jobs

Das *LsD-System* bewirkt eine Modifikation des klassischen Threadmodells [146]. In diesem Modell wird angenommen, dass ein Thread ein leichtgewichtiger Prozess ist, der die Ausführungsumgebung des Initiators teilt, auch dessen Adressraum und Ressourcen und daher dessen Initialisierung und Terminierung schneller sind als bei Prozessen. Diese Annahmen sind für Threads, die über das *LsD-System* verteilt werden, nur bedingt richtig. Adressraum und Ressourcen sind weiterhin noch die des Initiators, also des Prozesses, der den Konstruktor des Java Thread aufgerufen hat, aber dieser Prozess ist nicht die ursprüngliche Applikation, sondern das Laufzeitsystem von *LsD*.

Latenzbasierte Verteilung

Die optimierte Prozessverteilung des *LsD-Systems* basiert auf einer Zuordnung der Kommunikationshäufigkeiten zwischen den verteilten Prozessen und der Kommunikationsanbindung der vorhandenen freien Knoten zueinander. Dieser Mechanismus ist für die Anwendung und für die verteilten Prozesse transparent. Prozesse der Anwendung müssen vor einer Kommunikationsanweisung lediglich einmalig den Knoten des Kommunikationspartners über einen Namensserver erfragen. Dieser Namensserver ist ein Prozess des *MasterDaemon*, der die Zuordnung von Anwenderprozessen zu Knoten hält und bei Anfragen, die das Tupel aus Jobidentifikator und Applikationsidentifikator beinhalten, mit dem Knoten des Zielprozesses, einer Hostadresse in FQDN [12] Format, antwortet.

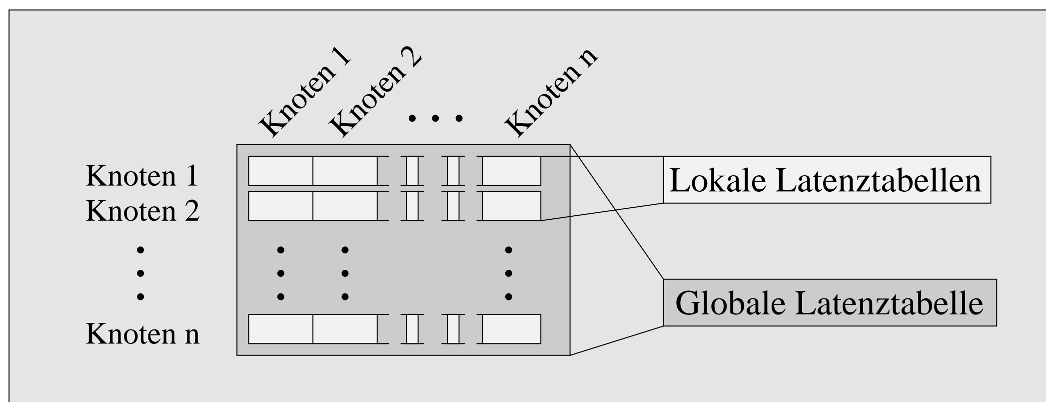

Abbildung 4.8: Latenztabellen

Sowohl der *MasterDaemon* als auch der *NodeDaemon* verfügen über Latenztabellen, die periodisch aktualisiert werden. Der *MasterDaemon* erhält dabei die Latenztabellen aller angeschlossen Knoten und bildet hieraus die globale Latenztabelle. Diese enthält sämtliche Knoten zu Knoten Verbindungen (siehe Abbildung 4.8). In einer Schleife über alle Knoten werden nun die Latenzen eines Knotens

[12] Fully Qualified Domain Name

akkumuliert, ebenso werden alle eingetragen Kommunikationswerte eines Prozesses akkumuliert. Diese beiden Listen werden sortiert und in invertierter Reihenfolge zueinander gekoppelt. Der entsprechende Algorithmus ist in Abbildung 4.9 in Pseudocode dargestellt.

$\mathcal{K}$: Menge der Knoten
$\mathcal{L}$: Latenzwerte
$\mathcal{P}$: Menge der Prozesse
$\mathcal{C}$: Kommunikationswerte

for each p in $\mathcal{P}$
    $p_{total} = \sum_{i=1}^{|P|} C(i,p)$
end

for each k in $\mathcal{K}$
    $l_{total} = \sum_{i=1}^{|K|} L(i,l)$
end

$\mathcal{P}$ = **sort** $(p, 1)$
$\mathcal{K}$ = **sort** $(l, -1)$

for each p in $\mathcal{P}$, k in $\mathcal{K}$
    bind (p, k)
end

Abbildung 4.9: Algorithmus zur Bestimmung eines Zielrechners

Die Tabelle der Häufigkeit der Anwendungskommunikation ist von dem Entwickler zur Verfügung zu stellen. Obwohl dies zusätzlichen Entwicklungsaufwand bedeutet, erscheint dies die günstigste Methode, Kommunikationsinformationen dem *LsD-Systems* zur Verfügung zu stellen. Werkzeuge zur automatisierten Extraktion von diesen Parametern sind in der benötigten Form nicht erhältlich, darüberhinaus kann durch die manuelle Angabe diese Informationen sehr exakt angeben werden, da der Entwickler zu jedem Kommunikationsaufruf diesen Wert angeben kann.

Ein weiterer Vorteil ist durch die Möglichkeit der Einflussnahme gegeben. Die Parameter liegen in einer nicht spezifizierten Metrik vor, es wird jeweils lediglich der abstrakte Wert berücksichtigt. Das ermöglicht dem Programmierer eine eigene Wertung der Wichtigkeit einer Kommunikation. So läßt sich für zeitkritische Übertragungen ein hoher, relativ überbewerteter, Betrag angeben. Dieser Wert wird bei der optimierten Verteilung als hohes Kommunikationsaufkommen bewertet und der Prozess entsprechend günstig positioniert.

```
public class JobComm
{
    LinkedList Jobs = null;
    LinkedList lockedProcesses;
    int appID;
    int [][] CommunikationArray;
    int []   jobToIndexMapper;
    int JobEntrys;
    int mode;
    /* [...] */
}
```

Abbildung 4.10: Kommunikationsstruktur einer LsD Anwendung

Die dem *LsD-System* zu übergebende Struktur ist in Abbildung 4.10 abgebildet. Zusätzlich ist die Möglichkeit gegeben, eine Auswahl an Prozessen an bestimmte Knoten zu binden, falls diese Information im Vorhinein bekannt ist. Dies ermöglicht es, Prozesse die eine starke Anbindung zu lokalen Ressourcen benötigen (beispielsweise zu einem Datenbankprozess oder Zugriff auf bestimme Massenspeicher), optimal zu platzieren. Aufgrund der zusätzlich nötigen Information (welcher dieser Knoten ist an dem *LsD-System* beteiligt) ist diese Option jedoch meist nur in Intranets praktikabel.

Zusätzlich ist in der Struktur, die dem *LsD-System* die Kommunikationshäufigkeiten mitteilt, ein Feld für die Art der Ermittlung der Latenztabelle enthalten. Über dieses Feld wird dem *LsD-System* mitgeteilt, wann diese Tabelle erneut erstellt wird. Hier steht eine Ermittlung direkt vor der Verteilung der Prozesse zur Auswahl, oder, falls eine sofortige Verteilung als vorteilhafter angesehen wird, die bis zu diesem Zeitpunkt ermittelten Werte (dies ist der voreingestellte Wert). Dieser Mechanismus gibt dem Entwickler die notwendige Flexibilität, nach anwendungsinternen Prioritäten die Prozesse zu verteilen.

Kommunikation

Das *LsD* kommuniziert über Java Sockets, die in der Regel sehr effizient direkt auf den gleichnamigen Mechanismus des darunterliegenden Betriebssystems abgebildet werden können. Als Protokoll dient *TCP* , was gegenüber des verbindungslosen Protokolls *UDP* eine Sicherungsschicht zur zuverlässigen Übertragung bietet [150]. Obwohl auf *UDP* basierte Kommunikation eine höhere Leistung bietet, da ein signifikanter Anteil an Verwaltungsaufwand eliminiert wird, werden alle internen Verbindungen durch *TCP* realisiert, da die Kommunikationssicherheit eine Schlüsselrolle für die Zuverlässigkeit eines Metacomputers spielt.

Über die Kommunikation der Anwendungsprozesse werden keine Annahmen getroffen, um der Anwendung einen größtmöglichen Freiraum zu bieten. Diese Methode ist ein effizienter Mittelweg zwischen lose gekoppelten Prozessen, wie sie beispielsweise *JavaSpace* oder *CORBA* bieten, und streng gekoppelten Prozessen, deren Kommunikationspartner nicht variabel sein können. Abbildung 4.11 zeigt den Ablauf der Kommunikation während der Abarbeitung einer Anwendung.

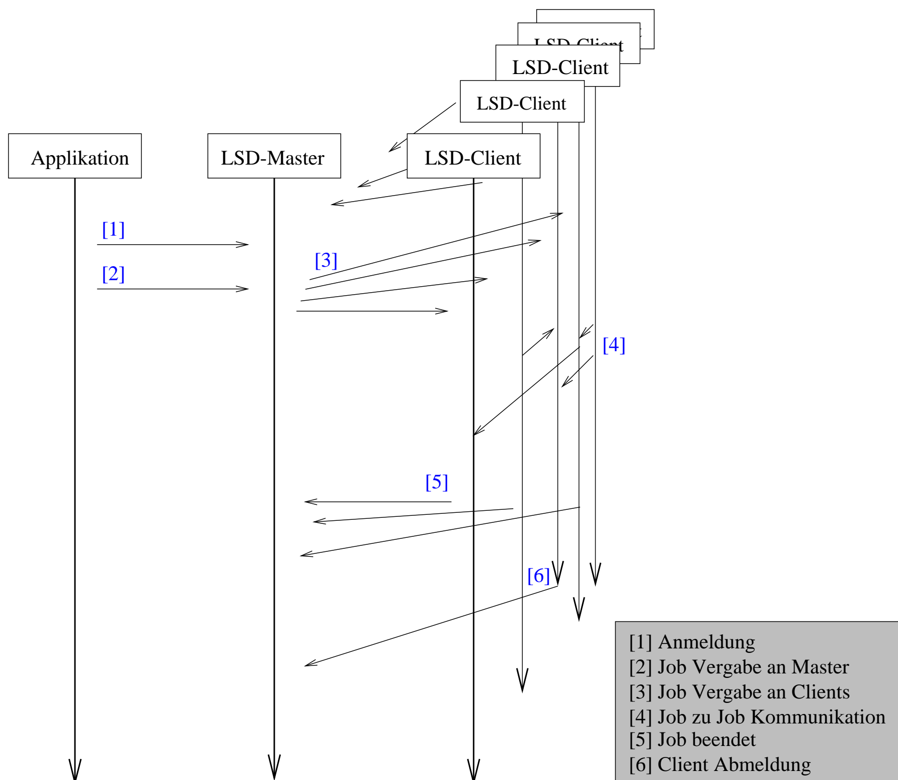

Abbildung 4.11: Anwendungsbezogene Kommunikation im LsD-System

Die Latenzmessungen der *NodeDaemon* erfolgt nicht über das übliche Werkzeug `ping`, da dies gegen den Java Gedanken der Portabilität sprechen würde. Auch eine Nachbildung eines solchen Mechanismus lässt Java aus diesen Gründen nicht zu, da `ping` keine *TCP* oder *UDP* Pakete versendet, sondern die Internet Kontroll Pakete (*ICMP*, Internet Control Message Protocol) ECHO_REQUEST versendet[13]. Erschwerend kommt hinzu, dass dieser Paketart teilweise unterschiedliche Priorität bei den Vermittlungseinheiten gewährleistet wird. Aus diesen Gründen wird eine anwendungsrelevante Methode benutzt, indem dem Internet Dienst *echo* auf Port 7 des Zielrechners ein *UDP* Paket gesendet wird. Der *echo* Dienst beantwortet dieses mit einem Paket gleichen Inhalts. Diese Antwortzeit wird in diesem Rahmen als Latenzwert betrachtet.

[13] Das konstruieren solcher Pakete würde gegen Benutzerrechte verstoßen.

Der Ansatz der Messung der Latenz innerhalb des *LsD-Systems* hat als Nachteil die im Vergleich zu externen Standardwerkzeugen wie *ping*, *traceroute* oder *netperf* verschobene Messwerte. Diese entstehen durch Leistungseinbußen aufgrund der Java Virtuellen Maschine. Diese Werte sind dennoch in sich konsistent, da sie gleichmäßig alle Werte betreffen, und als Entscheidungskriterium nur der relative zeitliche Abstand herangezogen wird. Darüberhinaus lässt sich im *LsD-System* ein externes Werkzeug zur Messung angeben, dessen Ergebnisse verwendet werden können. Dies ist jedoch aus oben genannten Portierungsgründen nicht die Standardeinstellung.

Schnittstellen

Das *LsD-System* kann anderen Werkzeugen Zugang zu den Prozess- und Knoteninformationen zur Verfügung stellen. Dies wird über die Kommandozeilen Schnittstelle ermöglicht, deren Eingabe und Ausgabe über Datenströme umgeleitet und an Werkzeuge gekoppelt werden können. Werkzeuge, die diese Schnittstelle verwenden können, sind Visualisierer, Debugger und Leistungsmesswerkzeuge sowie Metacomputing-Umgebungen, die das *LsD-System* als Werkzeug zur optimierten Prozessverteilung nutzen.

Als Beobachtungswerkzeug steht *MIMO* [128] zur Verfügung [41]. *MIMO* ist ein Werkzeug zum Beobachten von verteilten Middleware-Anwendungen (siehe Abschnitt 2.4.2). *MIMO* erhält Informationen von instrumentierten Anwendungen oder von instrumentierten Bibliotheken. Zur Beobachtung von *LsD* sind die Module *MasterDaemon* und *NodeDaemon* mit *MIMO* Adaptern implementiert worden (siehe Abbildung 4.12). Hierbei kommunizieren die *MIMO* Adapter mit dem *MIMO* System und erhalten die Informationen über die Prozessaktivitäten der Anwendung von der *LsD* Prozessverwaltung.

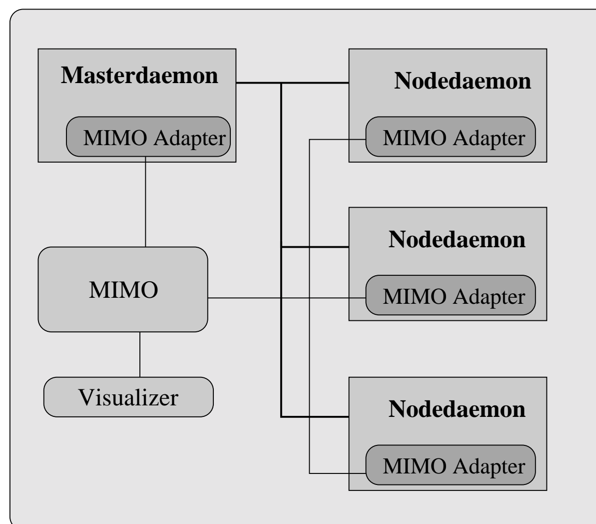

Abbildung 4.12: Instrumentierung des *LsD-Systems* zur Beobachtung mit MIMO

```
com.ooc.CORBA.Properties
                nsLocProp = com.ooc.CORBA.Properties.init(args);
adapter = new MIMOAdapter.ManualAdapter(orb,nsLocProp);
adapter.setApplName("LSD");
/* [...] */
adapter.setApplName("LSD"+W.worker.getAppID());
adapter.newThread (W.worker.getJobID(),
                  T.currentThread().getName());
/* [...] */
adapter.delThread(jobid);
```

Abbildung 4.13: Instrumentierung des NodeDaemon mit MIMO Adaptern

```
/* [...] */
private static MIMOAdapter.ManualAdapter adapter;
Properties props = System.getProperties();
props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
props.put("org.omg.CORBA.ORBSingletonClass",
         "com.ooc.CORBA.ORBSingleton");
System.setProperties(props);

String[] args = new String[0];
ORB    orb  = ORB.init(args,props);

com.ooc.CORBA.Properties nsLocProp =
                          com.ooc.CORBA.Properties.init(args);
adapter = new MIMOAdapter.ManualAdapter(orb,nsLocProp);
adapter.setApplName("LSD");
/* [...] */
adapter.interactionThreads(jobID,jjid,"GenServer");
```

Abbildung 4.14: Instrumentierung einer *LsD* Applikation mit *MIMO* Adaptern

In den Abbildungen 4.13 und 4.14 sind jeweils die vorgenommenen Instrumentierungen des *NodeDaemon* und der gestarteten Applikation dargestellt. Beide Code

Fragmente definieren einen neuen *MIMO* Adapter als *CORBA* Objekt und ordnen diesem Objekt innerhalb von *MIMO* einen Namen zu. In Abbildung 4.13 wird daraufhin bei der Generierung eines neuen Applikationsthreads der Adapter über Applikationsidentifikator und Jobidentifikator informiert. Diese Information wird zu dem *MIMO* System übertragen und ausgewertet. In dem in Abbildung 4.14 dargestellten Programm wird analog verfahren. Als Informationen werden dem *MIMO* System die Kommunikationsaufrufe des Threads mitgeteilt. Aus diesen Informationen kann *MIMO* die Gesamtkommunikation beobachten und diese dem Entwickler zur Generierung einer exakten Kommunikationsmatrix zur Verfügung stellen.

Die Zuordnung des *MIMO* Schichtenmodells zu den Ebenen innerhalb von *LsD* sind in Tabelle 4.4 zu sehen.

LsD Entity	*MIMO MLM Entity*
LsD Applikation	Application
LsD Worker Interface	Interface
LsD Jobs	Distributed Object
LsD Thread	Implementation
LsD Prozess	Runtime
Knoten	Hardware

Tabelle 4.4: Die Zuordnung von MIMO MLM Schichten zu LsD

Eine weitere Schnittstelle steht der Anwendung in der Prozessverteilung zur Verfügung. Dieses Modul kann durch dynamisches Klassennachladen überladen werden. Damit ist dem Entwickler oder einer Umgebung, die das *LsD-System* als Werkzeug benutzt, die Möglichkeit gegeben, die Prozessverteilung der Anwendungen entsprechend zu steuern.

4.5 Java

Die Programmiersprache Java ist eine verhältnismäßig junge Sprache und im Bereich des wissenschaftlich-technischen Rechnens unüblich. Deshalb wird in diesem Abschnitt auf die Gründe hierzu eingegangen und die für die Implementierung wesentlichen Eigenschaften diskutiert.

Java wurde im Jahre 1995 von Sun [155] am Markt eingeführt. Eine Reihe von Eigenschaften führte dazu, dass Java eine sehr große Akzeptanz erreichte und innerhalb von kürzester Zeit in verschiedenen Bereichen eingesetzt wurde.

Zu dem überraschenden Erfolg trug vor allem bei, dass Java sich vielen aktuell diskutierten Problemen in Form von umfangreichen Klassenbibliotheken, die fest zu der sogenannten *Java Plattform* [94] (siehe Abbildung 4.15) gehören, oder durch Berücksichtigung im Sprachumfang annahm. Hierzu gehören vor allem das dynamische Binden (Internet-Bezug) sowie Sicherheitsaspekte, Ausnahmebehandlung, automatische Speicherverwaltung (*Garbage Collection*), mehrfädige Ausführung und vor allem große Unterstützung im Bereich der verteilten Programmierung.

Zudem sind als optionale Klassenbibliotheken eine große Zahl an Paketen frei verfügbar (zum großen Teil von Sun selbst), so dass die Entwicklung von Applikationen ungewöhnlich schnell ging und relativ fehlerfrei möglich war.

Die Java Virtuelle Maschine (JVM) ermöglicht das dynamische Binden von Programmteilen (Klassen). Neben dem Laden von Klassen aus dem lokalen Dateisystem ermöglicht die Klasse `ClassLoader`, welche innerhalb des Prozesses des Bindens das Laden von Klassen zur Laufzeit realisiert, das Laden von Klassen über ein Netzwerk oder auch dynamisch erzeugte Klassen. Dieses flexible Konzept wird durch die Möglichkeit erweitert, dass die Klasse `ClassLoader` durch Anwendungen überladen werden kann, und somit individuelle Mechanismen implementierbar sind.

Als sehr großer Vorteil gegenüber anderen aktuell benutzten Sprachen *C++*, *C*, im wissenschaftlich-technischen Bereich auch *Fortran*) ist die Plattformunabhängigkeit des Java Bytecode vermarktet worden. Das Ergebnis des Java Übersetzers ist keine maschinenabhängige Objektdatei, sondern eine Zwischenform,der sogenannte Bytecode, die von jeder Java Plattform verarbeitet werden kann. Diese Verfahrensweise ist an sich nicht neu [14], aber in Kombination mit dem an Bedeutung stark anwachsenden Internet oder bei verteilter Programmierung im Intranet, in dem eine Vielzahl verschiedener Architekturen zum Zusammenspiel bewegt werden sollen, und einer stark objektorientierten Sprache lag einer der wichtigsten Gründe für den Erfolg von Java.

In den folgenden Abschnitten werden zunächst die wichtigsten Spracheigenschaften von Java erläutert, sowie anschließend auf die technischen Besonderheiten eingegangen.

4.5.1 Spracheigenschaften

Plattformunabhängig/ Netzgeeignet:

Durch die Übersetzung des Quellcodes in einen maschinenunabhängigen Bytecode sind Java Programme auf jeder Plattform , auf der eine Java Virtuelle Maschine existiert, lauffähig. Mit der nahtlosen Einbindung von Klassentypen für *URLs*, *Sockets* und weiteren wichtigen Protokollen (in `java.net.*`) in die Klassenhierarchie der Standard Java Plattform wird die verteilte Programmierung wesentlich vereinfacht. Sowohl der Kommunikationsaufbau über diese Klassen als auch die Kommunikation selber sind durch die vollständige Kapselung von der fehleranfälligen und schwer zu analysierenden Netzwerkprogrammierung entkoppelt. Davon profitieren einfache Internetanwendungen wie das Lesen und Schreiben in *URLs* oder das Versenden von Applets ebenso wie sehr komplexe verteilte Anwendungen.

[14]UCSD Pascal benutzte eine Virtuelle Maschine (P-System), der Zwischencode hieß P-Code.

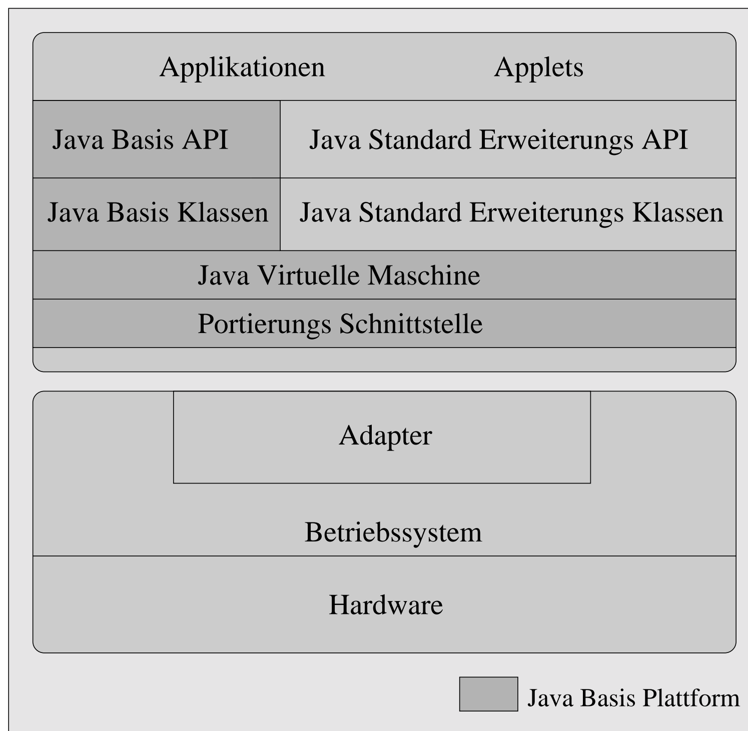

Abbildung 4.15: Das Prinzip der Java Plattform: Die Virtuelle Maschine und das API

Objektorientiert:

Als moderne Sprache bietet Java strenge Objektorientierung, Vererbung[15], Polymorphismus und Datenabstraktion. Die mit der Objektorientierung verbundenen häufigen Methodenaufrufe und deren dynamische Bindung (Polymorphismus) sind neben der Interpretation des Codes einer der wichtigsten Gründe für die Leistungsschwäche von Java und damit Hauptansatzpunkt für Optimierungen [77].

Ausnahmebehandlung:

Zu der Frage, wann und wo in einer Applikation Fehler, die während der Laufzeit aufgetreten sind, behandelt werden, bietet Java das Konzept der Ausnahmebehandlung an. Dieses trennt einerseits im Quelltext ordnungsgemäß ausgeführten Code von Code, der im Fehlerfall ausgeführt wird. Andererseits kann durch Werfen von Ausnahmenfehlern der Zeitpunkt (beziehungsweise die Programmebene) der Fehlerbehandlung bestimmt werden.

Sicherheitsmechanismen:

Schon in den ersten Versionen implementierte die Java Plattform ein Konzept

[15] Jedoch keine Mehrfachvererbung, welche mit der Einführung von *Interfaces* ersetzt wurde. *Interfaces* beschreiben lediglich das Verhalten eines Objektes und stellen keine Implementierung, somit auch keine implizierten Annahmen über diese, zur Verfügung.

zur sicheren Ausführung von fremden Programmen [16], die sogenannte *Sandbox*. In dieser hatte das auszuführende Programm nur sehr eingeschränkte Rechte auf die Ressourcen der Maschine. Realisiert wurde dieses durch eine Klasse `SecurityManager`, welche Aktionen des fremden Programmes im Voraus überprüfte. Lediglich Programme, die nicht über ein Netzwerk geladen wurden, konnten auf alle Ressourcen der JVM zurückgreifen. In der folgenden Entwicklung ([104, 105]) wurde dieser starre und unflexible Mechanismus aufgegeben und durch einen allgemeineren Ansatz ersetzt. Dieser sieht eine mögliche Überprüfung der Sicherheitsrichtlinien für jedes Programm vor, unabhängig von der Herkunft (lokal oder über ein Netzwerk geladen) . Desweiteren können Zugriffsrechte zu jeder lokalen Ressource vergeben werden, beziehungsweise diese an vorhandene digitale Signaturen koppeln. Eine genauere und auch praxisbezogene Beschreibung findet sich in [66].

Weitere Sicherheitsmechanismen setzen auf der Sprache selbst auf und beinhalten die Überprüfbarkeit der Quellen von Übersetzer und der *JVM*, der Abwesenheit von Zeigerarithmetik, einer automatischen Speicherverwaltung und einer strengen Typprüfung, welches die Hauptgründe für Sicherheitslücken in bisherigen Programmiersprachen waren. Weiterhin wird zur Laufzeit von einem *Verifizierer* die Korrektheit des Programms überprüft. Dieser Verifizierer erkennt unzulässige Formen der `.class` Dateien, ungültige Operanden (*Bytecode Verifizierer*) und Zeiger [17]. Eine nähere Beschreibung über die sicherheitsrelevanten Aufgaben der Verifizierers findet sich in [180].

4.5.2 Laufzeitverhalten

Der Einsatz von Java als Programmiersprache in einer Metacomputing-Umgebung bestimmt viele Eigenschaften dieser Umgebung. Diese Tatsache alleine verdeutlicht, dass Java neben einem reinen Werkzeug zur Applikationsentwicklung auch Konzepte bis zum Endprodukt transportiert. Die Vorteile sind die gute Wartbarkeit, (fast) keine Portierungskosten für verschiedene Plattformen, kurze Entwicklungszeit, reduzierte Fehleranfälligkeit und eine konzeptuell einfache Lösung für eine heterogene, untereinander kommunizierende Rechnerumgebung.

Diese Vorteile werden mit zum Teil erheblichen Leistungseinbußen bezahlt, welche sich gerade im wissenschaftlich-technischen Rechnen und/oder in Metacomputing-Umgebungen, wo es auf hohe Effizienz ankommt, empfindlich auswirken.

Auf technische Grundlagen wird im Folgenden eingegangen, die Konsequenzen zu den Leistungseinbußen und mögliche Lösungsansätze im wissenschaftlich-technischen Rechnen werden im nächsten Abschnitt (4.5.3) beschrieben.

[16] neben anderen sicherheitsrelevanten Funktionalitäten wie digitalen Signaturen, Verschlüsselung der Kommunikation und eine Schlüsselverwaltung.

[17] Im Falle von bösartigen Übersetzern, die unkorrekte `.class` Dateien erzeugen.

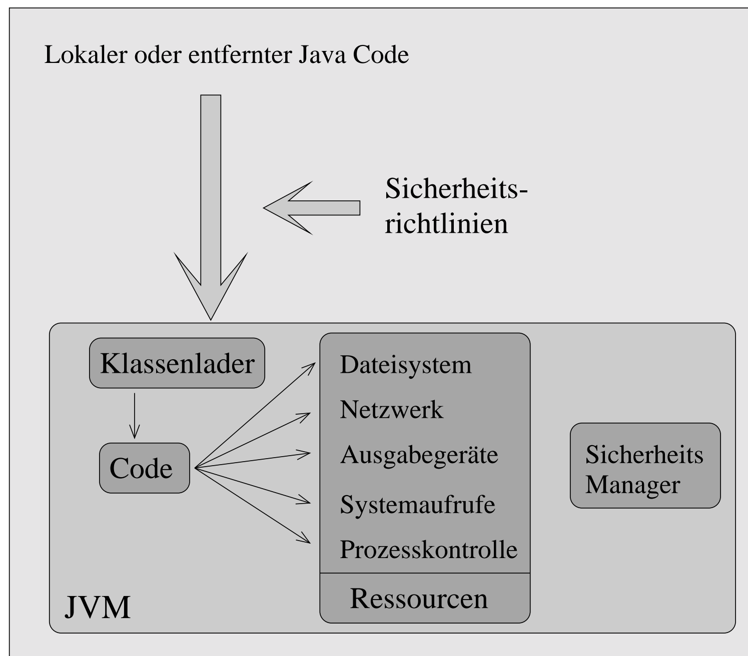

Abbildung 4.16: Das Java Sicherheitsmodell in der JDK Version 1.2.

Interpreter/JVM:

Die ursprüngliche Ausführungsumgebung von Java [153] war ausschließlich interpretativ (interpretiert wurde der Bytecode, siehe auch Bild 4.17). Dies hatte zum Teil sehr lange Ausführungszeiten zur Folge, wodurch Java den Ruf erlangte, sehr langsam zu sein.

Neben einer konventionellen Übersetzung kann alternativ zu der Interpretation des Java Bytecodes heute zur Geschwindigkeitssteigerung Übersetzung zur Laufzeit (*Just in Time Compiler*) und die *Hot Spot* Technik Anwendung finden.

Die Java Virtuelle Maschine ist einer der Grundpfeiler von Java. Als eine abstrakte Maschine verfügt sie über einen Befehlssatz, der lesend und schreibend auf den Speicher zugreift. Der dabei auszuführende Bytecode kann interpretiert oder direkt in den Maschinencode der Zielmaschine übersetzt werden.

Multithreaded

Sowohl die JVM sieht in ihrer Spezifikation [105] bereits mehrfädige Ausführung vor, als auch die Definition der Sprache selbst [18]. Diese Einbettung von leichtgewichtigen Prozessen in die Ausführungsplattform [67] ermöglicht leichte Zugänglichkeit und effiziente Nutzung von Nebenläufigkeit.

[18] Durch das Schlüsselwort *synchronized*, welches die Ausführung unter der Kontrolle von Monitoren erzwingt.

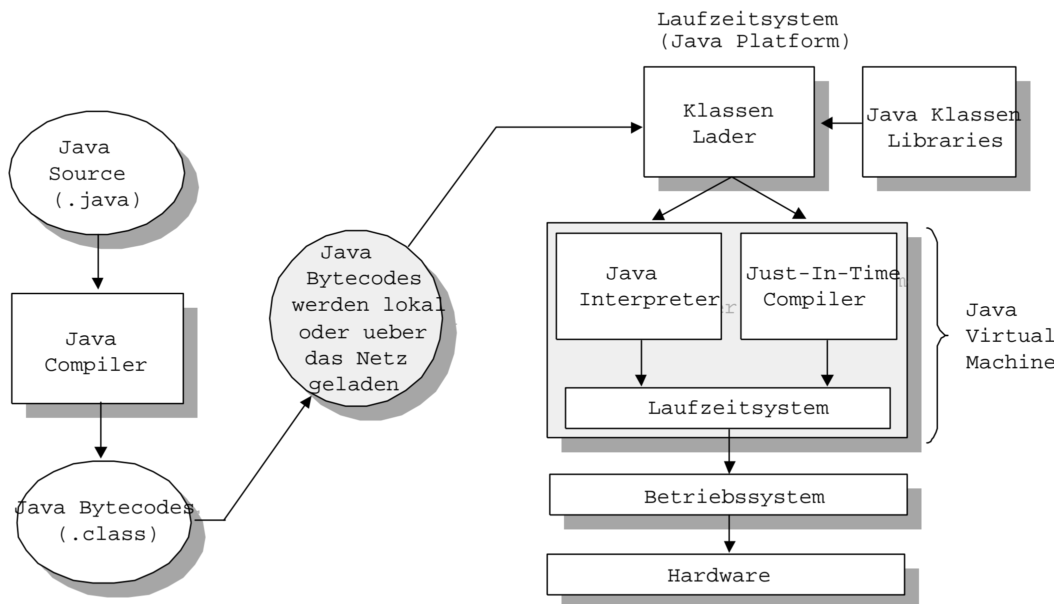

Abbildung 4.17: Ausführung eines Java Programms

Automatische Speicherverwaltung:

Java verfügt standardmäßig über eine automatische Speicherverwaltung. Diese stellt bei der Objekterzeugung Speicher zur Verfügung (ohne explizite Anforderung) und gibt diesen wieder frei, sobald dieser Speicherbereich nicht mehr referenziert wird. Einfluss kann auf die automatische Speicherverwaltung, deren Aufrufe zu nichtdeterministischen Zeitpunkten erhebliche Leistungseinbußen hervorrufen können, durch zwei Möglichkeiten genommen werden: Zum einen durch das Überladen der Methode *finalize()*, welche die automatische Speicherverwaltung vor Freigabe des Speicherplatzes für dieses Objekt aufruft, zum anderen durch explizites Aufrufen von *System.gc()*, wodurch die automatische Speicherverwaltung zwangsweise ausgeführt werden kann.

Im Normalfall durchsucht ein nebenläufiger Prozess Listen von Objekten und deren Dereferenzierungen und löst bei Überschreiten von Grenzwerten die Speicherbereinigung aus. Dennoch ist die Implementierung den Entwicklern der *JVM* für die jeweiligen Plattformen überlassen.

Neben dem Verzicht auf Zeigerarithmetik ist die automatische Speicherverwaltung einer der Hauptgründe für die leichte Programmierbarkeit und geringe Fehleranfälligkeit von Java Programmen [89].

Just in Time Compilation (*JiT*):

Unter der *JiT Compilation*, der ersten großen Leistungssteigerungsmaßnahmen durch *Sun*, versteht man die Übersetzung des Java Bytecodes zur Laufzeit in Maschinencode der Plattform [152, 86]. Die Leistungen, die damit

erzielt werden können, liegen knapp unter der Leistung der Programmiersprache C++ [174].

Hot Spot:
Die *Hot Spot* Technologie erweitert die *JiT* Übersetzung indem sie eine Optimierung der Ausführung zur Laufzeit realisiert, welche darauf beruht, nur häufig ausgeführte Programmteile zu übersetzen und diesen Teil der Codes zu optimieren. Für langlaufende Applikation, in denen Schleifen häufig durchlaufen werden, so dass sich eine Übersetzung lohnt, erreicht diese Technik sehr große Leistungsgewinne. Bei kurzen Schleifen, bei denen die Ausführungszeiten in derselben Größenordnung liegen wie die eigentliche Optimierung ist aufgrund des Verwaltungsaufwandes durch das Messen der Schleifendurchläufe und durch die zusätzlich stattfindende Übersetzung und Optimierung eher eine Verschlechterung der Leistung die Folge.

Aufgrund der Leistungseinbußen, die sich mit der Plattformunabhängigkeit, der schwergewichtigen Laufzeitumgebung und dem sehr hohen Abstraktionsniveau ergeben haben, war und ist die Leistungsanalyse- und Optimierung ein vieldiskutiertes Thema [1]. In dem nächsten Abschnitt werden die wesentlichen Einflussfaktoren auf das wissenschaftlich-technische Rechnen beschrieben.

4.5.3 Java für wissenschaftlich-technisches Rechnen

Es gibt (wissenschaftliche) Bemühungen, Java hinsichtlich der Ausführungsgeschwindigkeit im Bereich des wissenschaftlich-technischen Rechnens, oder allgemeiner, auf das Hochleistungsrechnen hin, zu optimieren.

Bedeutung erlangt hat hier das 1998 in Palo Alto in einer *Birds of a Feather* Veranstaltung gegründete JavaGrande Forum [87, 43]. Ziel dieses Forums, in dem mehrere Forscher, Anwender und Firmen vertreten sind, ist es, sogenannte Grande Anwendungen mit Java zu realisieren. Grande Anwendungen sind hierbei sehr große und komplexe Probleme, die sehr hohe Rechenleistung benötigen und lange Laufzeiten haben. Für solche Anwendungen sind effiziente Ausführungsumgebungen notwendig, wobei die Programmiersprache Java sowie die Ausführungsumgebung diesbezüglich einige Mängel aufweisen, von denen die meisten mit jeweils unterschiedlichem Aufwand entweder behoben oder auf ein akzeptables Maß reduziert werden können.

Die folgende Erörterung basiert hauptsächlich auf den Erkenntnissen und Arbeiten dieses Forums [127], sowie auf Optimierungen von *IBM* und *SUN* als auch auf diversen Arbeiten auf universitärer Seite zum Thema *High Performace Java*. Sie beziehen sich auf Spracheigenschaften von Java, die in Bezug auf wissenschaftlich-technisches Rechnen Schwachstellen darstellen.

Klassische Programmiersprachen in diesem Bereich wie *C* und insbesondere *FORTRAN*, und vermehrt in jüngster Zeit *C++*, haben in den folgenden Bereichen noch einen teilweise erheblichen Vorsprung.

Fliesskommaleistung

Die Fliesskommaleistung von Java erreicht ohne Modifikationen bei weitem nicht die Leistung von anderen Programmiersprachen. Zudem kommen noch Verhaltensweisen, die für wissenschaftlich-technisches Rechnen ungeeignet sind, hinzu. Hierzu gehören:

- Die Handhabung von komplexen Zahlen ist durch die fehlende Möglichkeit des Überladens von Operatoren und durch das Fehlen von primitiven Datentypen [19] umständlich und langsam [78].
- Durch die möglichen heterogenen Ausführungsumgebungen hat sich die Java Spezifikation darauf festgelegt, auf jeder Plattform gleiche und reproduzierbare Ergebnisse hervorzubringen[20]. Dies führt zu einer eingeschränkten Nutzung der Fliesskommaeinheiten und zu überflüssigen Rundungen bei jeden möglichen Zwischenschritten [90].
- Java ist nicht `IEEE 754` [85] konform. Diese Norm sorgt (unter vielem Anderen) durch das Setzen mehrerer Schalter *(flags)* für die Möglichkeit, den Status der letzten Berechnung abzufragen, um somit Fehler in der weiteren Programmausführung zu verhindern. In Java ist die Funktionalität dieser Schalter nicht implementiert.

Auf einen *JSR* [21] von Seiten des JavaGrande Forums wurde daraufhin das neue Schlüsselwort `strictfp` eingeführt, welches das ursprünglich Verhalten der JVM erzwingt. Ansonsten darf auch das für die jeweilige Plattform optimale Format (z.B. bei Intel kompatiblen Prozessoren das interne 80 Bit Format) und Vorgehen (z.B. bei PowerPC, MIPS und HP Prozessoren die FMA - Befehle (*fused multiply add*)) verwendet werden.

Für das Rechnen mit komplexen Zahlen gibt es zwei *JSR* Vorschläge von dem JavaGrande Forum. Zum einen ist dies die minimal invasive Java Klasse `java.lang.Complex`, IBM hat diese Klasse in ihrem Java Übersetzer realisiert [178]. Der zweite Vorschlag ist ein primitiver Datentyp `complex`, dessen Verwirklichung allerdings geringere Chancen hat, da er eine Änderung des Bytecodes oder die etwas unelegante Lösung eines Präprozessors benötigt.

Andere Quellen zu dem Thema Fliesskommaleistung in Java, vor allem von *IBM*, sind in [139] zu finden.

Feldzugriffe:

Zu jedem Feldzugriff wird zur Laufzeit überprüft, ob der Index innerhalb der Feldgrenzen liegt. Dieses Verhalten, welches im Allgemeinen sinnvoll ist und hilft, Programmierfehler zu vermeiden, sollte jedoch im Einzelfall aus Leistungsgründen abschaltbar sein.

[19] Ein Identitätsvergleich von implementierten Klassen, die komplexe Zahlen nachbilden, prüft auf Objektidentität anstatt auf Wertidentität.

[20] Dieses Konzept ist neben den technischen Mängeln auch bei vielen wissenschaftlichen Anwendungen, wie zum Beispiel Strömungssimulationen oder Crashtestsimulationen, nicht sinnvoll.

[21] *Java Specification Request*

Mehrdimensionale Felder in Java bestehen aus mehreren eindimensionalen Felder. Das hat zur Folge, dass mehrdimensionale Felder unterschiedlich lange einzelne Zeilen haben können. Dies erschwert zum einen die Optimierung auf Cache- und Pipelinegrößen, zum anderen kostet die zusätzliche Indirektion Zeit.

Remote Method Invocation:

RMI ist eine Implementierung des entfernten Methodenaufrufes in Java. Der entfernte Methodenaufruf leidet unter der Tatsache, dass er ursprünglich nicht für paralleles Hochleistungsrechnen konzipiert wurde, und dass die dabei durchgeführte Serialisierung (siehe dort) in ein maschinenunabhängiges Format der Aufrufparameter in der Regel unnötig aufwändig ist. In [126, 120] wird ein effizienter entfernter Methodenaufruf realisiert (zusammen mit einer neuen, etwas eingeschränkten Serialisierung, siehe nächster Punkt). Die dort erzielten Geschwindigkeitsgewinne und die Erweiterung auf Netze , die nicht auf TCP/IP basieren, zeigen die mögliche Leistungsfähigkeit zukünftiger, leichtgewichtigerer Versionen.

Serialisierung:

Die Serialisierung überführt Objekte in einen persistenten, maschinenunabhängigen Zustand. In der Objektdefinition geschieht dies durch das Implementieren des `Serializable` Interfaces. Hiermit lassen sich Objekte auf verschiedene Plattformen übertragen und verarbeiten. Die meisten Implementierungen beinhalten jedoch sehr aufwändige Mechanismen zur Überführung in die Byterepräsentation und viele Kopieroperationen, um alle referenzierten Objekte und Schleifenbildungen zu erkennen. Zudem wird bei diesem generellen Ansatz davon ausgegangen, dass der Objekt Code und der Objekterzeugende Code bei der Deserialisierung nicht mehr verfügbar sein kann, und deshalb ebenfalls serialisiert wird.

Für parallele Anwendungen im Hochleistungsrechnen sind diese umfassenden Maßnahmen meistens überflüssig, da diese Anwendungen in einem relativ engen (zusammenhängenden) Zeitfenster rechnen und hierin die Klasseninformationen fortwährend vorhanden sind.

Auf Vorschläge des JavaGrande Forums wird *Sun* die Art der Serialisierung in den nächsten Versionen wählbar gestalten, so dass effizientere Methoden [126, 81] zur Verfügung stehen.

4.5.4 Zusammenfassung

Java ist eine sehr flexible, moderne Sprache, die sich auch für wissenschaftlich-technisches Hochleistungsrechnen eignet. Allerdings ist hierfür der Entwicklungsaufwand höher als für Standardanwendungen. Um hohe Rechenleistung und Kommunikationsleistung zu erhalten, ist eine sorgfältige Wahl der Java Virtuellen Maschine und diverser erhältlicher Optimierungsmöglichkeiten notwendig.

Diese Erweiterungen sind nicht im Sinne der Entwicklung von Java als universelle Plattform für Anwendungen aller Art, die wenig zusätzlichen Aufwand versprechen, sondern sie zeigen das Potential der nächsten Jahre und machen die Entwicklung dafür kalkulier- und berechenbar.

Für Metacomputing-Plattformen ist mit gewissen Einschränkungen, die vor allem den hohen Speicherverbrauch betreffen, Java mit seinen plattformübergreifenden Eigenschaften geeignet.

4.6 Experimentierumgebung

Software

Die Evaluierung des *LsD-Systems* erfolgte über vier synthetische Anwendungen ($A_1, \ldots, A_4$), die verschiedene Kommunikationsmuster simulieren. Alle vier Anwendungen wurden daraufhin entwickelt, eine Evaluation bezüglich der Effizienz der Kommunikation zu ermöglichen. Der maßgebliche Bewertungsfaktor ist bei allen die Gesamtlaufzeit, als sekundärer Faktor kann die Stabilität der Laufzeit gegenüber Latenzschwankungen gelten.

Wie aus dem nächsten Abschnitt (Hardware) ersehen werden kann, ist die eingesetzte Rechner Hardware in ihrer Rechenleistung sehr inhomogen. Da die Gesamtlaufzeit der Anwendung bewertet wird, die von der Leistung der Maschinen abhängig ist, gilt es hier eine Messmethodik zu finden, die es erlaubt, diesen Faktor nicht in das Ergebnis einfließen zu lassen. Dies ist nötig, damit die Kommunikation die entscheidende Komponente der Gesamtlaufzeit bildet. Erreicht wird dies durch die einfache Methode, in den Prozessen der Mini-Benchmarks keine Berechnungen zuzulassen, die die Laufzeit entscheidend beeinflußen. Diese Prozesse wickeln lediglich die Kommunikation der simulierten Anwendung ab, ohne eigentliche Berechnungen vorzunehmen. In einem gewissen Rahmen sollten demnach gleichartige Prozesse ähnliche Laufzeiten aufweisen, unabhängig von dem Knoten, auf den sie das *LsD-Systems* migriert hat.

Alle Anwendungen kommunizieren über Java Sockets mit dem Protokoll *TCP* (siehe Abbildung 4.18). Wo es sinnvoll erscheint, werden Verbindungen nicht abgebaut und die Leerung der Kommunikationspuffer erzwungen.

Dabei sind die Kommunikationsmuster der einzelnen Anwendungen wie folgt aufgebaut:

A_1 ist eine symmetrische, strenge Client/Server Kommunikation. Alle Clients fordern in kurzen Abständen Daten von einem Server an, der daraufhin eine neue Aufgabe an den Client verschickt. Diese Kommunikationsart tritt bei wissenschaftlich-technischen Anwendungen auf, bei denen ein Server die Partitionierung des Problems berechnet und die Partitionen an Clients versendet. Weitere Anwendungen können Abfragen an Datenbanken sein, oder die in Abschnitt 3.1.1 beschriebenen Berechnungen. Die Anwendung A_1 sollte

```
Socket s = new Socket(target, port);

/* [...] */

Socket s;
ServerSocket serverSocket = new ServerSocket(listenPort);
serverSocket.setSoTimeout(timeOut);
s = serverSocket.accept();
```

Abbildung 4.18: Kommunikationsstruktur einer LsD Anwendung

von der latenzbezogenen Verteilung des *LsD-Systems* solange nicht betroffen sein, wie der Server auf einen günstigen Knoten migriert wird. In diesem Falle kann es als Referenz einer schlecht zu optimierenden Anwendung gelten, an der A_2 gemessen werden kann. Den “worst case” einer nichtoptimierten Verteilung bestimmt der Fall der Migration des Servers auf den langsamsten Knoten, die als obere Schranke der Gesamtausführungszeit angenommen werden.

Die Kommunikationsmatrix [22] dieser Anwendung ist immer in der Form

$$\begin{pmatrix} 0 & n & \cdots & n \\ n & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ n & 0 & \cdots & 0 \end{pmatrix} \tag{4.2}$$

gegeben. In den Kommunikationsmatrizen stehen die Elemente ungleich null für das jeweilige Kommunikationsaufkommen, eine null steht für keinen Verbindungsaufbau, ein Eintrag in (i, j) steht für eine Kommunikation von Knoten i zu j. Diese Matrizen entsprechen den Angaben des Applikationsentwicklers für das *LsD-System* (siehe Abschnitt 4.4).

Liegt eine Anwendung in der klassischen Client/Server Aufteilung vor, d.h. es gibt einen Prozess (Server), welcher das zu lösende Problem partitioniert und eine beliebige Anzahl an Prozessen (Clients), die eine gewisse Anzahl an Teilproblemen T_A anfordern und bearbeiten, so sind diese Probleme meist automatisch lastbalanziert, solange das Verhältnis aus Teilproblemen zu Knoten P (vgl 2.13) $G = \frac{T_A}{P}$ hinreichend hoch ist und solange die Unterschiede der Rechenleistungen der Clients in begrenztem Ausmaß bleiben. Diese Art

[22] Alle Kommunikationsmatrizen sind hier vereinfacht schematisch angegeben, auf eine genaue Wiedergabe der Matrizen, die die Benchmarks detaillierter modellieren, wird aus Gründen der Lesbarkeit verzichtet.

der Lastbalanzierung hat zur Folge, dass hier Knoten mit schlechter Anbindung weniger Aufträge berechnen können, und daher eine hohe Last auf gut angebundenen Knoten eine Verlängerung der Gesamtlaufzeit bewirkt.

A_2 ist eine asymmetrische Client/Server Kommunikation. Das Kommunikationsmuster ist zu A_1 identisch, aber es existieren hier Clients, die unterschiedlich häufig mit dem Server kommunizieren. Eine optimierte Verteilung sollte sich demnach wesentlich deutlicher als bei Anwendung A_1 auswirken, da hier die Platzierung von häufig kommunizierenden Clients auf schlecht angebundene Knoten von dem *LsD-System* vermieden werden sollte. Reale Anwendungen, die hiermit simuliert werden, sind identisch zu denen bei A_1 genannten, die Asymmetrie tritt jedoch in der Praxis häufiger auf. Dies liegt an der inhomogenen Rechnerleistung der angebundenen Knoten und an unterschiedlichen Problemgrößen.

Die Kommunikationsmatrix wurde folgendermaßen festgelegt:

$$\begin{pmatrix} 0 & n & m & n & 2m \\ n & 0 & \cdots & \cdots & 0 \\ m & \vdots & \ddots & & \vdots \\ n & \vdots & & \ddots & \vdots \\ 2m & 0 & \cdots & \cdots & 0 \end{pmatrix} \tag{4.3}$$

wobei $m > n$ gilt.

A_3 Hier wird das Auslagern von häufig miteinander kommunizierenden Knoten simuliert, die gemeinsam eine Aufgabe berechnen, dies jedoch verhältnismäßig unabhängig von einer zentralen Instanz vermögen. In der Praxis tritt dieses Schema bei “coupled fields” Algorithmen auf. Diese Anwendung wurde mit 3 Knoten und folgender Matrix gemessen (Knoten 0 ist hier die zentrale Instanz):

$$\begin{pmatrix} 0 & n & n \\ n & 0 & m \\ n & m & 0 \end{pmatrix} \tag{4.4}$$

wobei $m >> n$ gilt. Der Erhöhung der Anzahl der Knoten hat auf die Laufzeit und auf das Verteilungsprinzip der Optimierung erwartungsgemäß keinen Einfluss.

A_4 Diese Anwendung, eine Erweiterung von A_3, hat mehrere Mengen von Knoten, die untereinander häufig kommunizieren, zwischen diesen Mengen wird wenig kommuniziert. Die Anwendung soll die Auslagerung von rechenintensiven Teilen an entfernte, nicht gut angebundene Orte simulieren. Zudem wird eine zentrale Instanz simuliert, die diese Berechnung koordiniert und die

Ergebnisse empfängt. Die allgemeine Kommunikationsmatrix kann wie folgt angegeben werden:

$$\begin{pmatrix}
0 & & i & \cdots & i & & i & \cdots & i & \\
& \ddots & & & & & & & & \\
i & & n & \cdots & n & & & & & \\
\vdots & & \vdots & \ddots & \vdots & & & & & \\
i & & n & \cdots & n & & & & & \\
& & & & & \ddots & & & & \\
i & & & & & & m & \cdots & m & \\
\vdots & & & & & & \vdots & \ddots & \vdots & \\
i & & & & & & m & \cdots & m & \\
& & & & & & & & & \ddots
\end{pmatrix} \qquad (4.5)$$

Analog zu oben gilt hier $i << m, n$.

Alle Typen von Messungen wurden jeweils vier mal durchgeführt. Davon je zwei mit eingeschalteter latenzoptimierter Verteilung, und je zweimal ohne Optimierung. Bei der Verteilung ohne Optimierung wurde bei den Client/Server Simulationen darauf geachtet, sowohl die ungünstigste Variante (Server kommt auf schlecht angebundenem Knoten zu liegen) als auch eine zufällige zu messen; bei den nicht optimierten Anwendungen A_3 und A_4 wurden die Prozesse zufällig verteilt.

Die Zeitmessung erfolgte mit instrumentierten Anwendungen. Dabei wurde als Startpunkt der Messung der Aufruf durch das *LsD-System* des jeweils als zentrale Komponente angegebenen Java Thread genommen, den Endpunkt markierte das Absetzen des *CallBack cb* (siehe Abbildung 4.7). Der zum Messen eingesetzte Code (siehe Abbildung 4.19) ist eine Methode der Java Virtuellen Maschine, für die eine Genauigkeit von einer Millisekunde angegeben wird.

```
long tStart,tAll;

tStart = System.currentTimeMillis();
/* [...] */
tAll = System.currentTimeMillis() - tStart;
```

Abbildung 4.19: Code zur Zeitmessung

Hardware

Die bei den Messungen beteiligten Rechner sind in Tabelle 4.5 angegeben. Die Kommunikationslatenzzeiten der Rechner simulieren eine Intranet Umgebung. Die Latenzen zwischen den Rechnern in dem 100 MBit Netz (über Switches) sind im Durchschnitt unter 1 ms, Spitzen wurden bei ca maximal 5 ms gemessen.

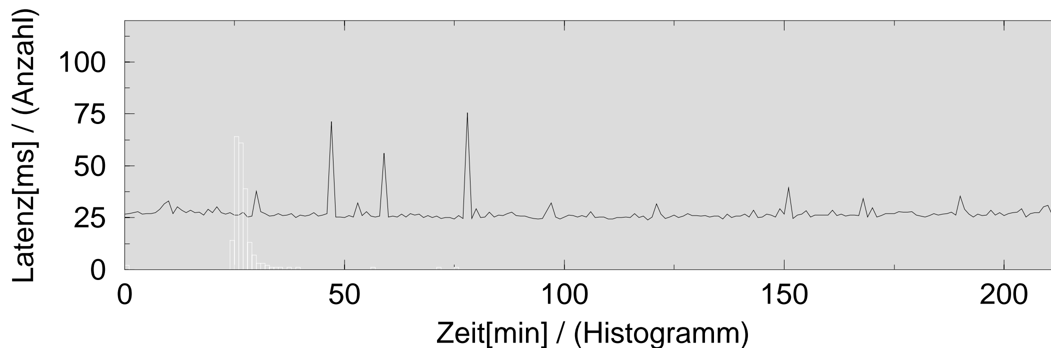

Abbildung 4.20: Fortwährende “Ping” Versuche zu Rechner “sonar”.

Die Netztopologie ist in Abbildung 4.21 (Rechner vgl. Tabelle 4.5) zu sehen; die Latenzzeit zu dem Rechner sonar beträgt zwischen 30 und 50 Millisekunden, ist somit fast konstant, was eine konsistente Messumgebung ermöglicht. Siehe hierzu Abbildung 4.20, die eine Messung der Latenz über mehrere Stunden hinweg veranschaulicht. Dabei besteht jeder Messpunkt aus dem Mittelwert dreier Versuche mittels `ping`.

Hostname	**Hstl.**	**Typ**	**CPU**	**Takt (MHz)**	**Speicher (MB)**	**uname -rs**	**Anbindung (MBit)**
atbode62	Dell	Intel	Pentium II	233	64	Linux 2.2.17	100 nistnet
atbode64	Dell	Intel	Pentium II	300	128	Linux 2.2.17	100 nistnet
sunbode1	Sun	Enterprise	4 x UltraSparc II	296	3000	SunOS 5.8	100
sunbode5	Sun	Ultra 10	1 x UltraSparc II	296	192	SunOS 5.6	100
sunbode15,16	Sun	Ultra 60	2 x UltraSparc II	296	512	SunOS 5.8	100
sunbode3,10	Sun	Ultra 10	1 x UltraSparc II	296	192	SunOS 5.8	100
sunhalle4-15	Sun	Ultra 60	2 x UltraSparc II	296	384	SunOS 5.8	100
sonar	Sun	Ultra 80	2 x UltraSparc II	450	2000	SunOS 5.8	2

Tabelle 4.5: Die Rechner der Testumgebung

Um Messungen über das Verhalten des *LsD-Systems* bei unterschiedlichen Latenzzeiten vorzunehmen, sind reale Bedingungen mit Knoten verteilt im Internet nicht sinnvoll. Latenzzeiten lassen sich hier nicht beeinflussen und Schwankungen

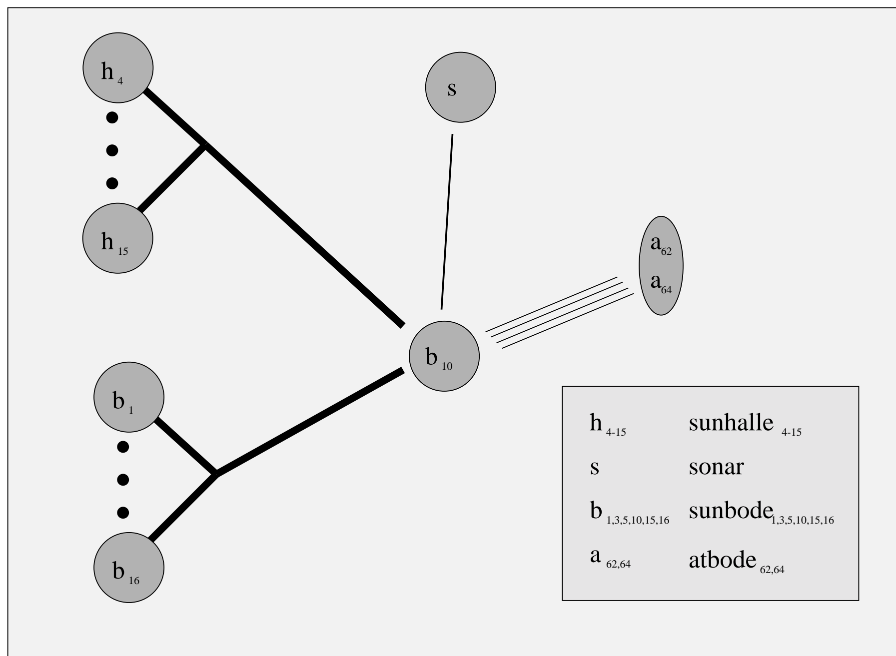

Abbildung 4.21: Aufbau der Experimentierumgebungen.

in den Verzögerungen machen Experimente nicht wiederholbar. Daher wurden Latenzzeiten mit dem Werkzeug *NIST Net*[76] (in der Version 2.0.6) emuliert. *NIST Net* ist ein Werkzeug des *National Institute of Standards and Technology* welches Eigenschaften eines Verbindungsnetzwerkes emuliert. Unter anderem können Latenzverzögerungen, Bandbreiten, Paketverluste sowie deren statistische Verteilung emuliert werden. *NIST Net* verwendet hierzu ein verändertes *Linux* Betriebssystem, indem es die Paketverwaltung des Netzwerkadapters modifiziert. Pakete werden je nach angegebenen Parametern daraufhin zurückgehalten oder verworfen.

Die Netzanbindungen der Rechner atbode62 und atbode64 sind über dieses Werkzeug gesteuert worden. Dabei wurden die Latenzzeiten entsprechend den Werten in den Tabellen eingestellt, die Standardabweichung wurde bei dem voreingestellten Wert Null belassen. Messungen haben ergeben, dass die Schwankungen innerhalb der von *NIST Net* absichtlich vorgenommenen Verzögerung weit unterhalb der üblichen Netzschwankungen sind, also vernachlässigt werden können.

Messergebnisse

Die Messergebnisse wurden gemäß den Vorgaben wie in den vorherigen Abschnitten geschildert durchgeführt. Evaluiert werden die Funktionsfähigkeit und die erzielbare Geschwindigkeitssteigerung durch das *LsD-System*. Hierzu wird in den

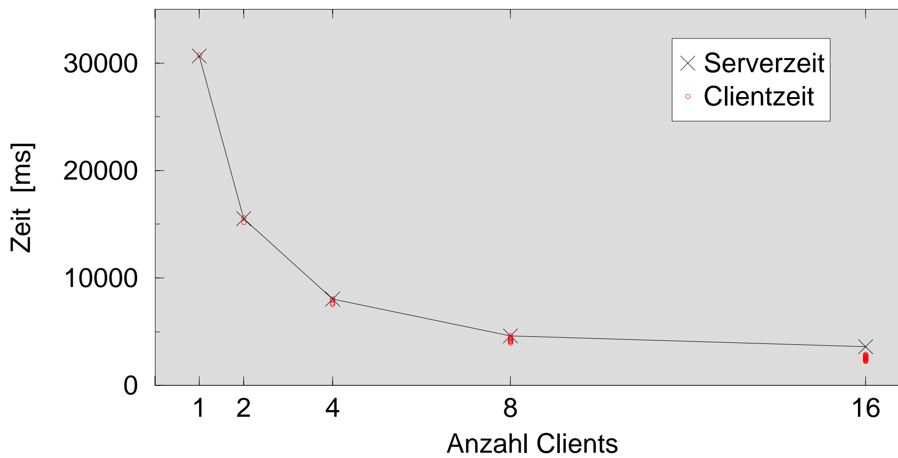

Abbildung 4.22: Skalierbarkeit der Architektur

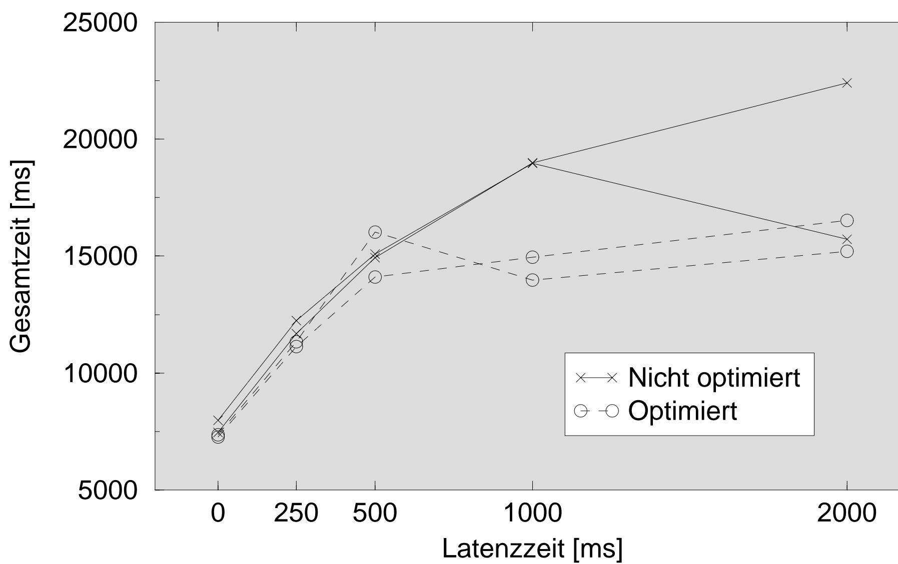

Abbildung 4.23: Vergleich der Anwendung A_1, optimiert und zufällige Verteilung

Abbildungen die erzielte Gesamtzeit mit der durchgeführten Optimierung der Prozessverteilung mit denen der zufälligen Prozessverteilung verglichen. Die angegebenen Latenzzeiten beziehen sich auf die Anbindung der mit *NIST Net* modifizierten Rechner atbode62 und atbode64.

Um ungewollte Nebeneffekte wie eine Auslastung des *LsD-Systems* oder der Rechner auszuschließen, wurde die Skalierung der Anwendung bezüglich der verteilten Prozesse und der Anzahl der beteiligten Knoten geprüft. Abbildung 4.22 zeigt die Ausführungszeiten des Servers und aller Clients von A_1. Dabei wird die Anzahl der Knoten jeweils um Faktor zwei erhöht. Erwartungsgemäß halbiert sich die Ge-

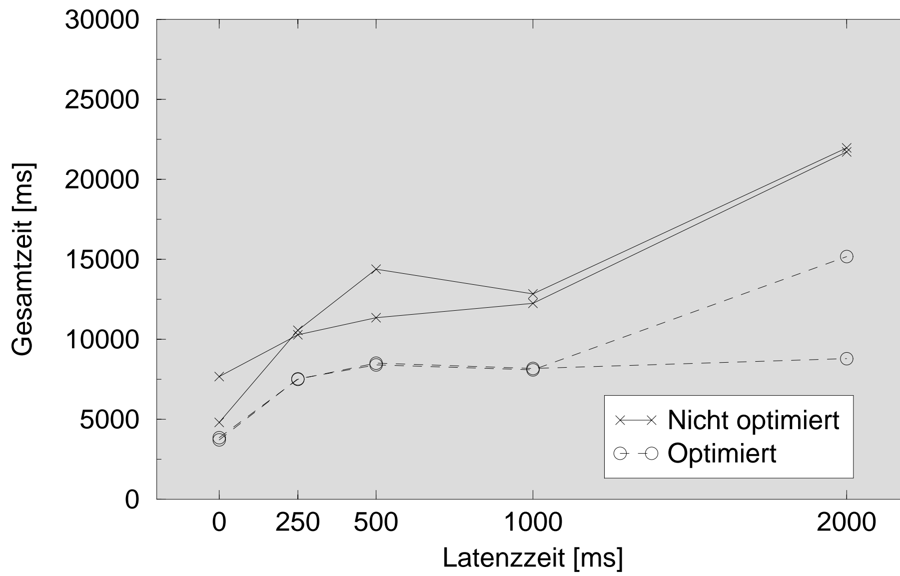

Abbildung 4.24: Vergleich der Anwendung A_2, optimiert und zufällige Verteilung

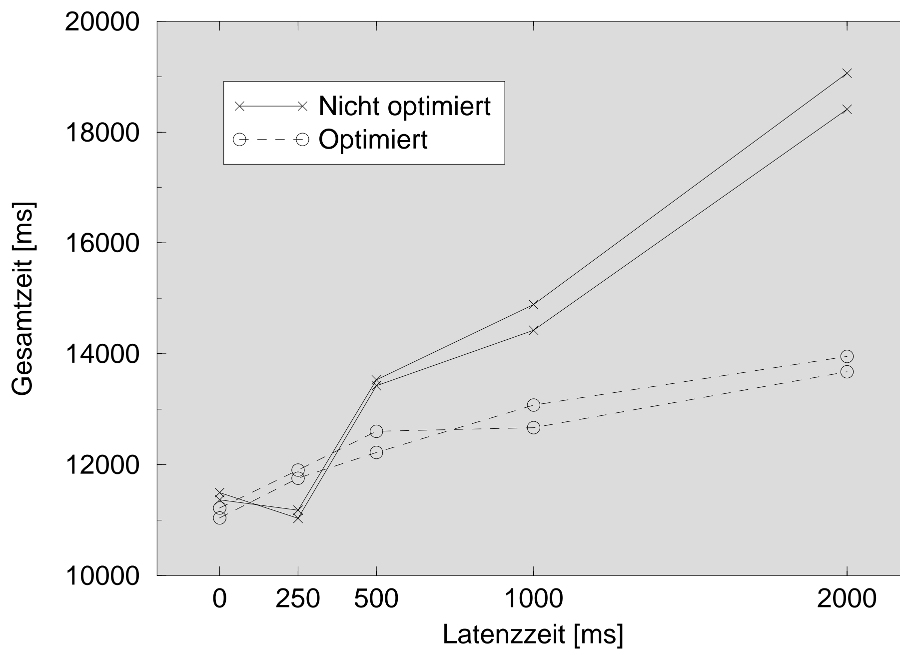

Abbildung 4.25: Vergleich der Anwendung A_3, optimiert und zufällige Verteilung

samtausführungszeit, zusätzlich sind die Ausführungszeiten der einzelnen Clients angegeben. Dieses Ergebnis lässt die Annahme zu, dass die beteiligten Komponen-

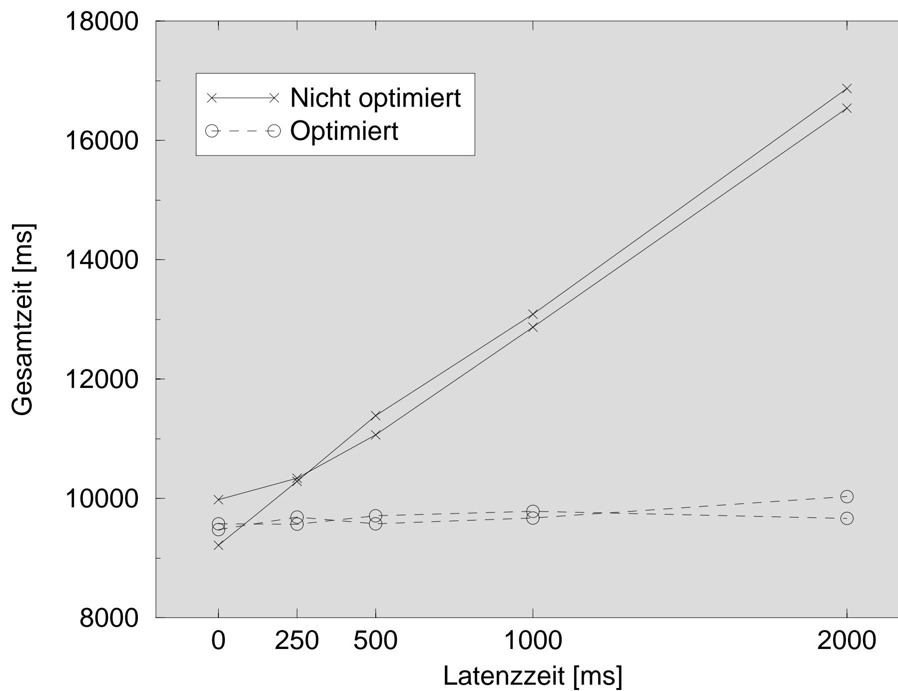

Abbildung 4.26: Vergleich der Anwendung A_4, optimiert und zufällige Verteilung

ten unterhalb ihrer Auslastungsgrenze liegen und die Unterschiede in den folgenden Messungen demnach auf die Optimierung zurückzuführen sind [23].

Abbildung 4.23 zeigt die Ausführungszeiten der Anwendung A_1 in der nicht latenzoptimierten und in der optimierten Prozessverteilung. Die Vorteile der latenzoptimierten Verteilung zeigen sich erwartungsgemäß bei den hohen Latenzen, allerdings ist hier die Gesamtlaufzeit stark von der Ankunft des zuletzt vergebenen Auftrags abhängig. Wird der letzte Auftrag von einem schlecht angebundenen Knoten bearbeitet, ergeben sich hier der Latenz entsprechend große Abweichungen, dieses Verhalten kann man an der letzten Messung (2000ms), nicht optimiert, ersehen.

In Abbildung 4.24 sind die Ergebnisse der nicht symmetrischen Client/Server Aufteilung gezeigt. Hierbei kommen bei der optimierten Verteilung die Prozesse mit den hohen Werten für die Kommunikation in keinem Fall auf den schlecht angebundenen Knoten atbode62, atbode64 und sonar zu liegen. Der Geschwindigkeitszuwachs liegt hier immer über 1.3, im besten Fall bei 1.8. Bei realen Anwendungen, die in diese Problemklasse fallen, kann man davon ausgehen, hier eine erhebliche Leistungssteigerung zu erzielen.

Anwendung A_3 beschreibt den oft vorkommenden Fall von einer Gruppe häufig kommunizierender Prozesse, die eine lose Verbindung zu einer zentralen Verwaltung beinhalten (“coupled fields” Algorithmen). Diese Messung simuliert das Aus-

[23]Es gibt darüberhinaus auch noch andere Faktoren, die Laufzeitunterschiede bewirken können, beispielsweise die unten erwähnte zufällige Erteilung des letzten Auftrages. Diese können jedoch bei langen Laufzeiten vernachlässigt werden.

lagern von Berechnungen in einem global verteilten Intranet. Die Geschwindigkeitssteigerung liegt über denen in A_1 und A_2 beobachteten. Hier wirkt sich die implizite Lastverteilung durch schnellere, gut angebundene Knoten, die mehr Aufträge bearbeiten, nicht aus. Falsch oder zufällige platzierte Prozesse erzeugen hohe Verzögerungen, die sich auf die Gesamtlaufzeit auswirken.

Die Messungen aus Anwendung A_3, die Gruppen von kommunizierenden Prozessen beschreibt, sind in Abbildung 4.26 zu sehen. Es kommen dabei die oft kommunizierenden Prozesse auf den Knoten atbode62 und atbode64, die zueinander eine schnelle Kommunikation erlauben, beziehungsweise auf den Rechner des Verbundes sunbode zu liegen. Auch hier wird ein sehr hoher Geschwindigkeitszuwachs gemessen, der in Abhängigkeit der variierten Latenz zwischen 1.0 und 1.7 liegt. Auch lässt sich sehen, dass bei keinen Latenzunterschieden die optimierte Verteilung keine Nachteile einbringt, dies gilt für alle betrachteten Anwendungen, im schlechtesten Fall ist bei A_3 0.94 gemessen worden, was auch allgemein für alle Anwendungen den niedrigsten Wert darstellt.

Die Messungen zeigen sehr deutlich, dass das *LsD-System* zusätzlich zu den in den vorherigen Abschnitten beschriebenen Vorteilen einer Middleware durch die latenzbasierte Prozessverteilung Leistungsvorteile bei verteilten Anwendungen hat. Auch im Falle geringer oder keiner Unterschiede zwischen den einzelnen Knoten konnten keine Leistungseinbußen durch Verwaltungsaufwand oder Berechnung der Zielknoten gemessen werden.

Zusätzlich sei hierzu bemerkt, dass für diese Leistungsvorteile keine verbesserte Parallelisierung, neue Algorithmen oder leistungsfähigere Hardware nötig sind. Im Vergleich zu diesen kostenintensiven Möglichkeiten der Leistungssteigerung ist die latenzbasierte Verteilung eine sehr kostengünstige und für die Anwendung transparente Alternative. Vor diesem Hintergrund sind die erzielten Leistungsgewinne als sehr hoch einzustufen.

Die Ergebnisse des Leistungszuwaches sind in Abbildung 4.27 und in Tabelle 4.6 in Abhängigkeit der eingestellten Latenzen wiedergegeben. In der Tabelle sind in der linken und unteren Spalte zusätzlich die durchschnittlichen Werte angeben.

Latenz[ms]	A_1	A_2	A_3	A_4	
0	1.054	1.649	1.027	1.007	1.184
250	1.065	1.389	**0.939**	1.071	1.116
500	0.996	1.522	1.085	1.164	1.192
1000	1.312	1.541	1.139	1.334	1.332
2000	1.202	**1.823**	1.356	1.696	1.519
	1.126	1.585	1.109	1.254	1.268

Tabelle 4.6: Gemessene Geschwindigkeitssteigerungen

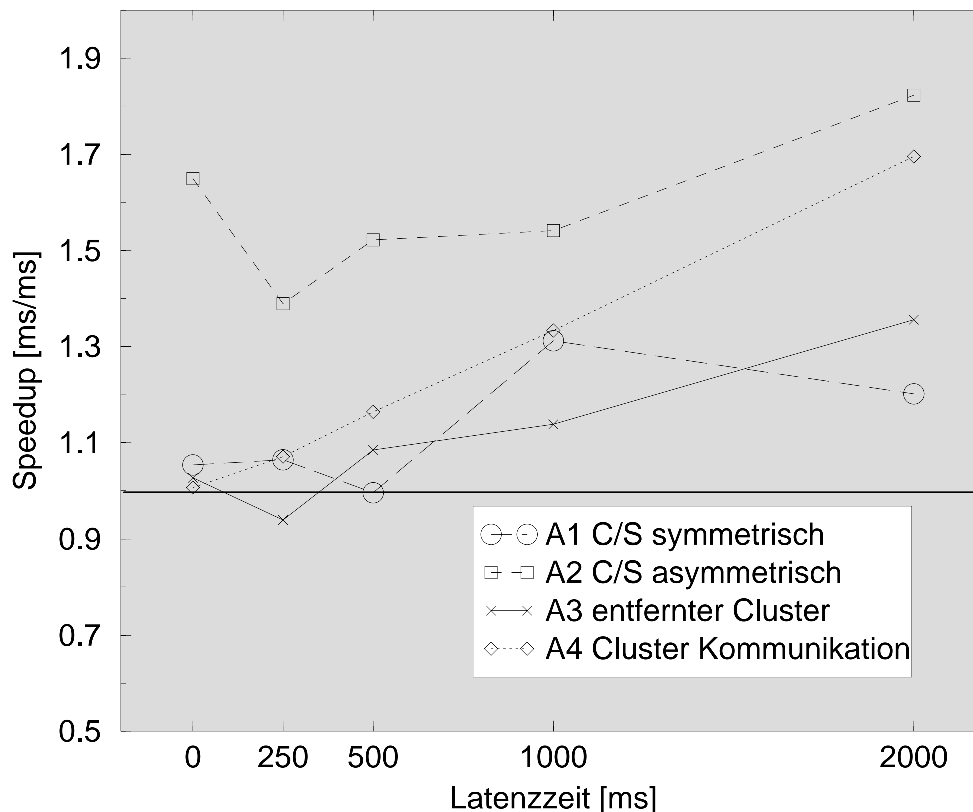

Abbildung 4.27: Der Speedup der optimierten Anwendungen A_{1-4} in Abhängigkeit der Latenzzeit

4.7 Zusammenfassung

In diesem Kapitel wurde das *LsD-System* eingeführt. *LsD* ist eine neue und effiziente Methode, Java Programme in Weitverkehrsnetzen optimiert ausführen zu lassen. Hierbei werden Java-Threads latenzoptimiert auf die angebundenen Knoten verteilt. Durch den leichtgewichtigen, plattformübergreifenden Ansatz eignet sich das *LsD-System* für Umgebungen, in denen Anpassungen bestehender Software an komplexe Metacomputer-Umgebungen wegen kostenintensiven Entwicklungen nicht realisiert werden können.

Zur Verteilung der Prozesse wurde ein effizienter Algorithmus entwickelt, der die Latenzen der beteiligten Knoten berücksichtigt. Es wurde weiterhin der Aufbau der Umgebung detailliert beschrieben und die Schnittstellen für Anwendungen und Werkzeuge angegeben. Ein eigener Abschnitt hat die Programmiersprache Java vor dem Hintergrund des wissenschaftlich-technischen Rechnens diskutiert. Mit einer Reihe von synthetischen Benchmarks, die die Anforderungen von realen Anwendungen simulieren, wurde die Leistungsfähigkeit der Umgebung nachgewiesen. Es wurden bei hohen Latenzzeiten oder bei Anwendungen mit geeigneter Struktur durchschnittliche Leistungsgewinne von 1.5 erzielt. Im günstigen Fall beträgt der

Leistungsgewinn 1.82, im ungünstigsten Fall 0.94. Alle Werte sind relativ zu der zufälligen Verteilung der parallelen Anwendung angeben.

5. Einsatz und Evaluierung der Anforderungen im industriellen Umfeld

Nachdem im vorherigen Kapitel das *LsD-System* als leichtgewichtige Metacomputer-Umgebung vorgestellt wurde und die latenzbasierte Verteilung von Prozessen sich als sinnvolle Optimierungsmethode erwiesen hat, werden in diesem Kapitel die in Abschnitt 4.3 postulierten Eigenschaften am Beispiel einer wichtigen industriellen Anwendung untersucht. Im ersten Abschnitt wird die Anwendungsumgebung beschrieben und der Wunsch nach Leistungssteigerung motiviert. Im zweiten Abschnitt wird die Anwendung für eine Nutzung des *LsD-Systems* analysiert.

5.1 Digitales Prototyping in der Automobilindustrie

Das Konzept der latenzoptimierten Prozessverteilung durch das *LsD-System* für ein global verteiltes Netzwerk soll anhand einer Analyse der Gegebenheiten in einer realen Umgebung evaluiert werden.

Hierzu werden die Ressourcen einer Firma (bezogen auf die Rechnerhardware und die Netztopologie), ihre Anwendung und die Parallelisierbarkeit der Anwendung bezüglich der Benutzung des *LsD-Systems* untersucht.

Die zugrundeliegende Software ("Virtuelle Werkstatt") der Firma Tecoplan AG stammt aus dem Bereich des wissenschaftlich-technischen Rechnens und realisiert eine voxelbasierte Einbau– und Passform-Simulation für den Bau virtueller Prototypen.

Die Firma Tecoplan AG [158] aus Ottobrunn bei München wurde 1992 gegründet und vertreibt Software zur rechnergestützten Konstruktion[1]

[1] Diese Untersuchung entstand im Rahmen des `FORTWIHR` (Bayerischer Forschungsverbund für technisch-wissenschaftliches Hochleistungsrechnen)[52] Projektes. Das Ziel des `FORTWIHR` ist es, durch interdisziplinäre Zusammenarbeit der Fachbereiche Ingenieurwissenschaft, Angewandte Mathematik und Informatik, Ergebnisse der Forschung in den bayerischen Industrieunternehmen umzusetzen [22]. Das *LsD-System* ist als Mittel zur Leistungssteigerung der "Virtuellen Werkstatt"

5.1.1 Virtuelle Montagen

Der Entwurfsprozess im Maschinenbau verlagerte sich mit der Einführung der elektronischen Datenverarbeitung zunehmend vom Zeichenbrett auf CAD (Computer Aided Design) Systeme. Heutzutage sind daher die Geometrieinformationen der Bauteile in einer elektronischen Repräsentation vorhanden und können mit CAD Systemen gespeichert, weiterverarbeitet und verwaltet werden. Einzelne Komponenten einer komplexen Struktur werden am Rechner entworfen und können daraufhin computergestützt gefertigt werden. Das Zusammenfügen zu dieser komplexen Struktur kann jedoch fehlerbehaftet sein. Dies liegt an der großen Anzahl von unterschiedlichen CAD Systemen oder an Fehler im Entwurfsprozess selber.

Um diese Fehler rechtzeitig vor Produktionsbeginn zu beseitigen, wird im Normalfall ein Prototyp, teilweise auch in kleinerem Maßstab, angefertigt. Der Bau eines Prototyps ist jedoch sehr zeitaufwändig und teuer, da zu dessen Herstellung Spezialwerkzeuge benötigt werden, die wiederum einem Entwurfs- und Konstruktionsprozess unterliegen. Erkannte Fehler werden wiederum an der Modellrepräsentation im Rechner geändert und der Prozess beginnt von vorne. Dieser Kosten- und Zeitaufwand ist auch für große Firmen nicht als trivial anzusehen, daher können aufgrund von diesbezüglich notwendigen Einsparungen im Entwicklungsprozess Qualitätseinbußen bei den fertiggestellten Produkten entstehen [133]. Entwicklungskosten und die Zeit der Entwicklungszyklen sind angesichts der Konkurrenzsituation in der industriellen Konstruktion (Automobilbau, Flugzeugbau) zukünftig mitentscheidende Faktoren für eine flexible Produktstrategie und damit den wirtschaftlichen Erfolg.

Die rechnergestützte Konstruktion und der rechnergestützte Zusammenbau von Prototypen durch numerische Simulation zu einem kompletten virtuellen Produkt können die oben beschriebenen Probleme lösen und die Entwicklungszyklen entscheidend verkürzen. Dieses virtuelle Prototyping ("digital Mockup") einer technischen Geometrie ist bereits in industriellen Produktionsumgebungen realisiert und wird erfolgreich eingesetzt.

in dem Teilprojekt 6 (Produktivitätssteigerung im digitalen Prototyping durch Interoperabilität von CAx und numerischer Simulation sowie Parallelverarbeitung) entstanden.

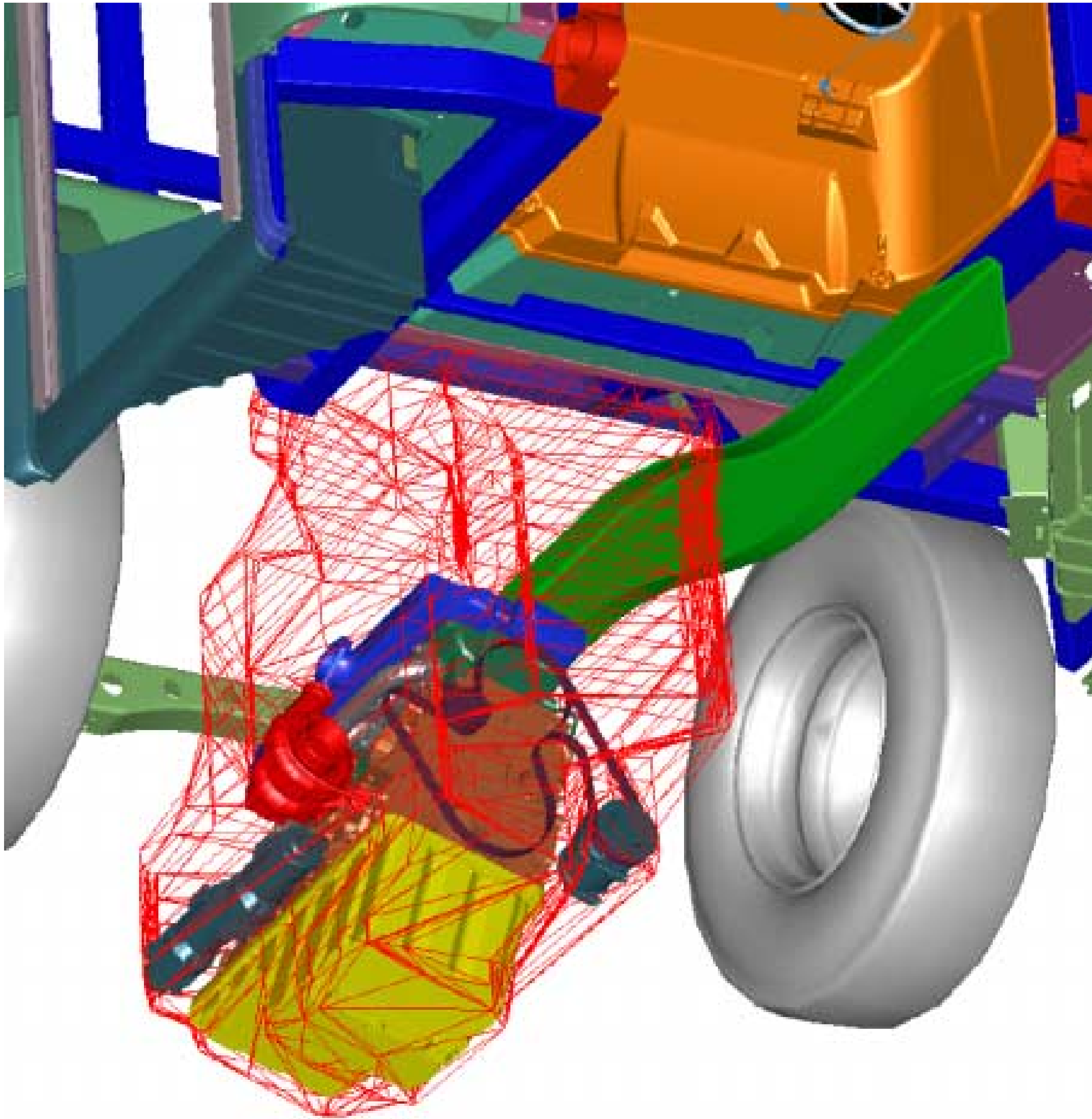

Abbildung 5.1: Darstellung eines Fahrzeugs durch die *Virtuelle Werkstatt*

5.1.2 Die “Virtuelle Werkstatt”

Einen großen Fortschritt für die virtuelle Montage bringt hier das voxelgestützte Programmsystem “Virtuelle Werkstatt” der Firma Tecoplan AG. Bauteile werden hier mit einer Repräsentation eines Modelles in Voxeltechnik[2] versehen. Diese Technik ermöglicht eine räumliche Darstellung (siehe Abbildung 5.1 als Beispiel einer Darstellung eine Automobils innerhalb der *Virtuellen Werkstatt*), mit der Einbau- und Passform-Simulationen effizient berechnet werden können.

Dem Entwickler eines Bauteils wird es ermöglicht, neu konstruierte oder veränderte Bauteile virtuell in das Produkt einzubauen. Hier können daraufhin ohne aufwändigen Prototypenbau Probleme direkt festgestellt werden. Die Dauer der Änderungszyklen eines Bauteils sind daher bei dieser Vorgehensweise auf die Rechenzeit der jeweiligen Prüfung reduziert.

Als Grundlage zur Datenverwaltung dient der *Virtuellen Werkstatt* ein *EDM* (Electronic Data Management) System, welches folgende Daten hält:

- CAD-Daten im Original-Format (beispielsweise CATIA, SLA, HOOPS, VDAFS oder STEP),
- eine Repräsentation in dem eigenen Voxel Format,
- Produkt-Strukturen (Konstruktionsinformationen),
- Mockups, für Berechnungen umgewandelte Produkt-Strukturen,
- Regel-Definitionen für Berechnungen.
- sowie alle Ergebnisse bereits durchgeführter Berechnungen.

Die Verwaltung dieser Information bildet die unterste Schicht innerhalb der *Virtuellen Werkstatt*, auf dieser Basis sind verschiedenste Funktionalitäten für den Entwurf durch ein CAD System als zweite Schicht implementiert (die *Virtuelle Werkstatt* ist sowohl als eigenständiges Softwarepaket erhältlich, das auf vorhandenen Konstruktionsdaten aufsetzten kann, als auch als modularisierte Teilpakete, die in bestehenden CAD Systeme eingebettet werden können).

In dieser Schicht stehen folgende implementierte Funktionen zur Verfügung:

Kollisionen

Hier ist eine Überprüfung auf Kollisionen von Bauteilen möglich. Es können

[2] Die Voxel-Technik teilt einen Raum in ein Gitter gleich großer, dreidimensionaler Würfel (Voxel) mit festgelegter Kantenlänge auf. Indem die Volumina, Flächen und Punkte des ursprünglichen CAD-Modells einheitlich in einfache Voxel-Modelle umgewandelt werden, verringert sich das Datenvolumen drastisch, was auch zur Folge hat, dass Untersuchungen erheblich beschleunigt werden. Durch die Kantenlänge der Voxel kann die Genauigkeit der Modelle und damit auch die der Berechnungen variiert werden. Um die räumlichen Nachbarschaftsbeziehungen der so erzeugten Modelle festzuhalten, werden diese in einer *Raumkarte* verzeichnet.

dabei zwei Bauteile geprüft werden, ein Bauteil mit allen anderen, oder alle Bauteile untereinander. Durch Aktivierung der *Epsilontik* kann angegeben werden, ab welcher Schnitt-Tiefe, -Winkel und -Länge eine Kollision angezeigt werden soll. Zudem können durch Angabe von Bewegungsmustern Kollisionen von sich bewegenden Bauteilen erkannt werden.

Distanzbereiche
Bei Angabe eines Distanzbereiches werden hier alle Tupel von Bauteilen berechnet, deren Entfernung sich innerhalb dieser Distanz bewegt.

Distanzminimum
Der geringste Abstand zwischen zwei Bauteilen wird berechnet.

Modelldifferenz
Bei Angabe zweier Bauteile wird die Differenz berechnet (Kontrolle der Versionsstände).

Nachbarschaftbestimmung
Die räumlich benachbarten Teile zu einem Bauteil werden berechnet.

Freiraumbestimmung
Zu einem vorgegebenen Quader wird der zur Verfügung stehenden Freiraum berechnet.

Auf diese zweite Schicht setzen die Visualisierungs- und CAD Systeme auf, die diese Dienste benutzen (siehe Abbildung 5.2)

Abbildung 5.2: Schichten der *Virtuellen Werkstatt*

Diese Darstellung erlaubt die Verknüpfung mit artverwandten Teilbereichen des Entwurfs, beispielsweise die Simulation des Strömungsverhaltens oder der Temperaturverteilung (CFD) oder strukturmechanische Berechnungen, für die bisher getrennte, nicht interoperable Softwarelösungen existierten. Durch eine Kombination

dieser Bereiche lassen sich enorme Vorteile bei der Konstruktion im Automobilbau beziehungsweise in der Luft- und Raumfahrt erhoffen. Hierzu müssen sich die dafür notwendigen Berechnungen jedoch in für interaktive Benutzer annehmbare Bereiche gebracht werden. Für eine einfache Kollisionsberechnung zweier Bauteile werden in der aktuellen Version mehrere Minuten Rechenzeit benötigt. Bei komplexen Berechnungen, wie beispielsweise der Bestimmung einer Volumenhülle eines variabel aufgehängten Abgasrohres, werden diese Zeiten um ein vielfaches übertroffen. Werden diese Berechnungen zukünftig noch durch die oben aufgezählten Disziplinen der Strömungsmechanik ergänzt, ist eine Berechnung mit sequentieller Abarbeitung nicht mehr denkbar.

5.1.3 Leistungssteigerung durch Parallelisierung

Als eine erste Maßnahme zur Realisierung eines global verteilten DMU-Servers (siehe 5.2.1) ist eine externe Anbindung an die *Virtuelle Werkstatt* implementiert worden. Diese stellt die Funktionalitäten des DMU-Servers beispielsweise externen Zulieferern zur Verfügung, um Bauteile bezüglich Passform zu kontrollieren [59] (siehe Abbildung 5.3).

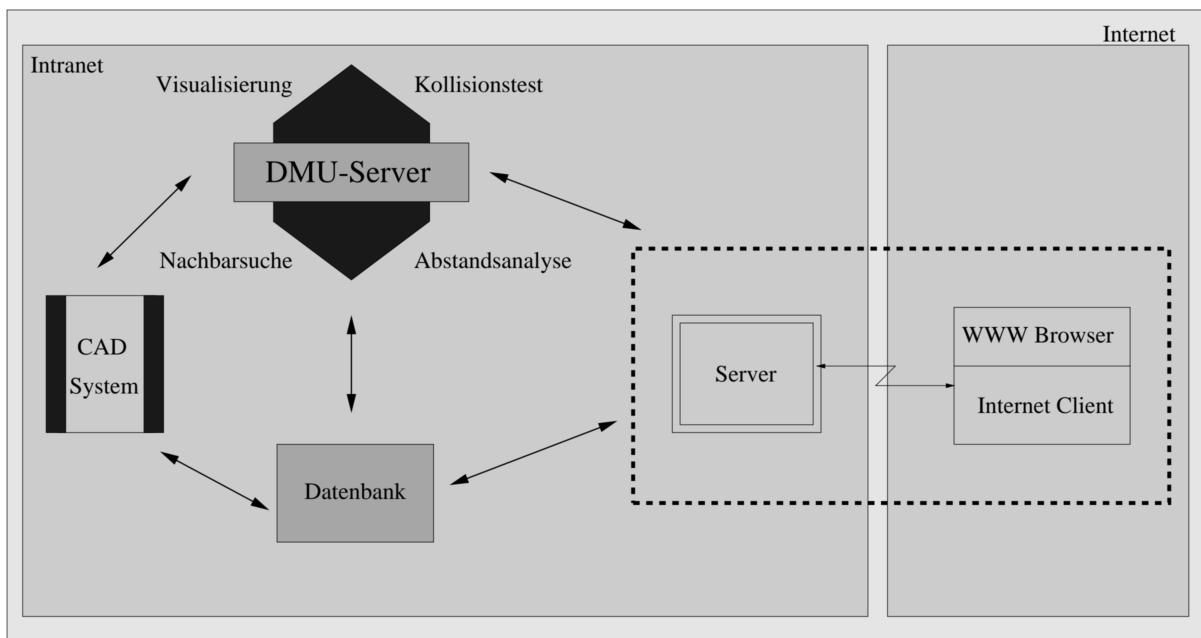

Abbildung 5.3: Externer Zugang zur *Virtuellen Werkstatt*

Eine Parallelisierung und Verteilung über ein globales Netzwerk durch das *LsD-System* würde hier eine entsprechende Leistungssteigerung bewirken.

Die heute verwendeten sequentiellen Berechnungsverfahren stoßen aufgrund der zunehmenden Komplexität der Modelle sowie der intensiveren Nutzung der Simulation durch die Entwickler an Leistungsgrenzen. Künftige Leistungsanforderungen

der industriellen Praxis können nur durch die Entwicklung paralleler und verteilter Berechnungsverfahren befriedigt werden.

Die Parallelisierung verschiedener Teile der *Virtuellen Werkstatt* lassen sich in folgende grundlegende Ziele unterteilen:

- Niedrige Antwortzeiten bei interaktiver Benutzung.
- Erhöhter Durchsatz des *DMU* Servers.
- Berechnung größerer Modelle durch verteilte Ressourcen.

Die bei einer erfolgten Parallelisierung zugrundeliegenden Techniken zur weitverteilten Ausführung sind dabei weitgehendst bereits vorhanden, wie in den vorgehenden zwei Kapiteln dargelegt worden ist. Mit dem Internet ist ein kommerziell nutzbares weltumspannendes Netzwerk mit sehr großen, verteilten Ressourcen gegeben. In Intranets stehen firmenintern ebenfalls großen Rechenressourcen zur Verfügung, welches im Falle der Firma Tecoplan AG ebenfalls zutrifft. Auch Zulieferer von Bauteilen verfügen über dezentrale Ressourcen und können mit der entwickelten Schnittstelle auf verteilte Ressourcen zugreifen. Mit der Programmiersprache Java ist ein Werkzeug vorhanden, welches das Problem der unvermeidbaren Heterogenität in solchen Netzwerken löst.

5.2 Die parallele Version der “Virtuellen Werkstatt” – viw/p

5.2.1 Modulanalyse

Für die Feststellung der Eignung der *Virtuelle Werkstatt* für das *LsD-System* muss eine Parallelisierung gefunden werden, die an das Prozessverteilungsmodell des *LsD-Systems* angepasst werden kann. Hierfür ist eine genaue Analyse der Anwendung nötig, um eine entsprechende Aufteilung an Aufgaben, die diesem Modell entsprechen, zu erreichen. Die *Virtuelle Werkstatt* liegt in der Version 4 (V4) bereits in sehr modularer Form vor. Die Module sind in Abbildung 5.4 abgebildet.

Diese Module bilden das sogenannte “DMU Framework” [92, 93], welches die Basis der Client/Server-Architektur ist. Folgende Komponenten existieren:

OBM
: bildet die Datenablage. Die Schnittstellen ermöglichen eine einfache Anbindung an bestehenden Datenbank Systeme. In der von Tecoplan realisierten Implementierung werden Objekte in dieser Datenbank über ein gemeinsames Dateisystem angesprochen [160].

Communication Platform
: Über das Modul “Communication Platform” wird die gesamte Kommunikation innerhalb des “DMU Framework” verwaltet [129]. Alle Nachrichten der

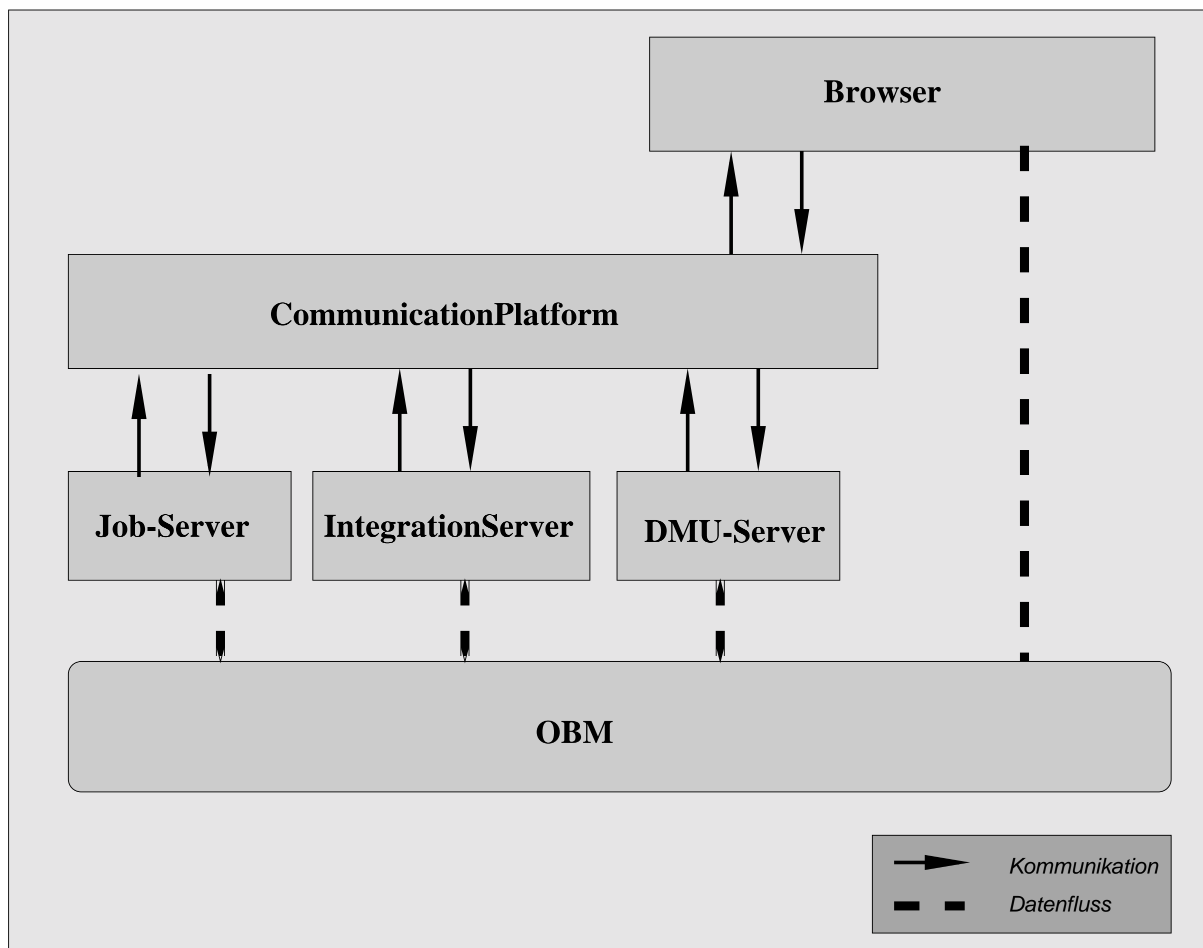

Abbildung 5.4: Aufbau der sequentiellen *Virtuellen Werkstatt*

angeschlossenen Subsysteme werden zu diesem Modul gesendet und von dort entsprechend weitergeleitet. Die aktuelle Implementierung realisiert dies mit Hilfe von RPC [79, 61]. Als zentrale Komponente neben dem Job-Server ist die "Communication Platform" auch für das Starten und Beenden der anderen Module verantwortlich.

DMU-Server

Der DMU-Server ist bezüglich der oben aufgezählten Funktionalitäten das Kernstück der Version 4. Hier werden alle aufgeführten Berechnungen ausgeführt und ein Großteil der Rechenzeit verbraucht.

Job-Server

Der Job-Server verwaltet die korrekte Abarbeitung von Anfragen an die Struktur der Bauteile. Er verwaltet Jobs in Queues, sorgt für die Ausführung zum gewünschten Zeitpunkt und ist in der Lage, Abhängigkeiten zwischen Jobs zu behandeln. In der Datenflusshierarchie der *Virtuellen Werkstatt* ist der Job-Server das zentrale Modul, hier werden alle Informationen gesam-

melt und gegebenenfalls weiterverteilt. [157, 5]. Die Aufgabe des Job-Servers ist in Abbildung 5.5 schematisch dargestellt.

Browser

Der Browser steht hier für eine beliebige Anwendung, die auf die Datensätzen und die Berechnungen der *Virtuellen Werkstatt* zugreift und die Ergebnisse verarbeitet.

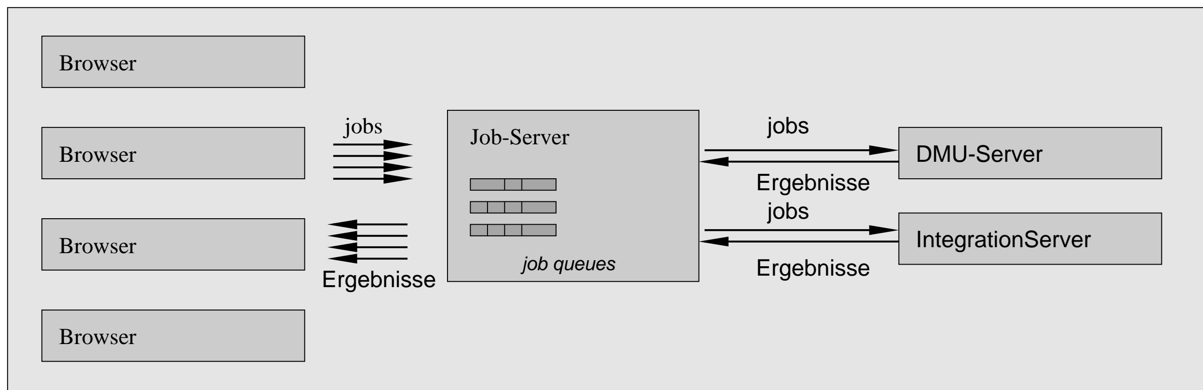

Abbildung 5.5: Aufgabe des Job-Servers innerhalb der *Virtuellen Werkstatt*

Innerhalb dieser Komponenten gestaltet sich der Ablauf einer Anfrage (beispielsweise: Kollidiert Bauteil A mit Bauteil B?) wie folgt:

Diese Anfrage wird dem JobServer übergeben, der daraufhin den Auftrag der Konvertierung der Bauteile A und B an den DMU-Server übergibt. Diese Bauteile werden nun von der externen Repräsentation (*assembly*) in die Repräsentation der *Virtuellen Werkstatt* (*Spacemaps* oder *mockup*) konvertiert. Diese Konvertierung ist in Abbildung 5.6 dargestellt. Sie erlaubt eine räumliche Auflösung der Bauteile und ist deshalb sehr effizient bei Berechnungen von Punkt zu Punkt Abständen. Auch sind hier keine Referenzen erlaubt, aufgrund der räumlichen Darstellung ist beispielsweise in der Abbildung jedes Rad einzeln dargestellt. Diese Konvertierung, Änderungen an Bauteilen ausgeschlossen, wird nur einmal vorgenommen (dabei wird das Datenvolumen sehr stark reduziert, teilweise auf ein Prozent der ursprünglichen Daten). Daraufhin definiert der Job-Server anhand der Anfrage Regeln, die diese Anfrage bearbeiten und sortiert diese in eine Warteschlange zu berechnender Aufträge ein.

Ist der DMU-Server für eine neue Anfrage bereit, wird ihm der neue Auftrag übermittelt. Dies ist mit einem einfachen nachrichtenorientiertem Protokoll realisiert (*compute job*, *kill job*, *suspend job*), welches über die “Communication Platform” vermittelt wird. Das Resultat dieser Berechnung wird dem Job-Server übergeben, der es in entsprechender Form der anfragenden Anwendung weiterleitet.

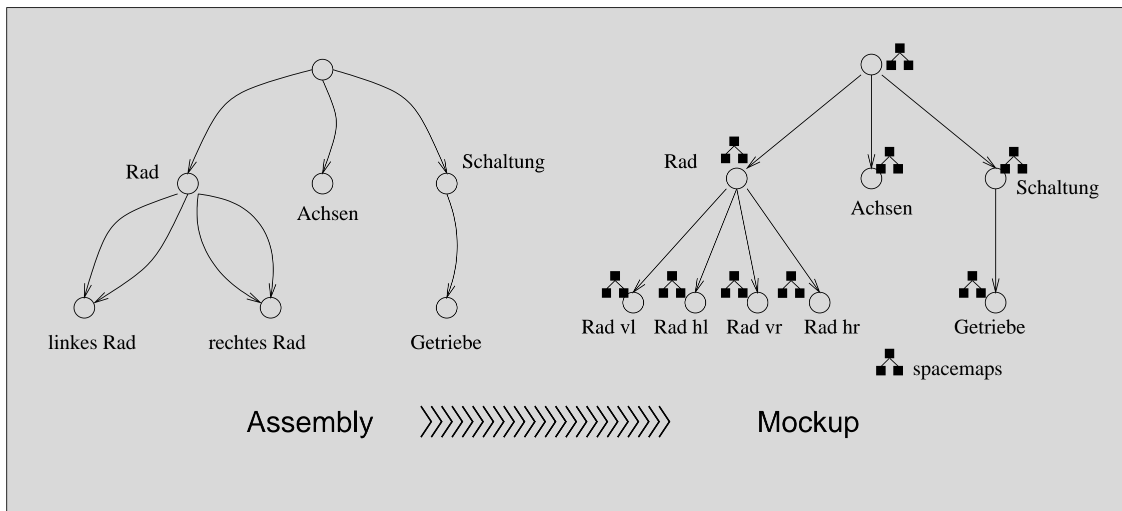

Abbildung 5.6: Konvertierung einer Modellrepräsentation in das Modell der *Virtuellen Werkstatt*

5.2.2 Parallelisierung der "Virtuellen Werkstatt"

Für eine weitverteilte Ausführung wurden aus verschiedenen evaluierten Modellen mehrere, verteilte DMU-Server, die den Rechenkern der Simulationen bilden, ausgewählt. Mehrere DMU-Server sind aufgrund weitgehender Datenunabhängigkeit und des günstigen Verhältnisses zwischen Rechenzeit und Kommunikationszeit der erfolgversprechendste Ansatz für eine Ausführung in einem Weitverkehrsnetz.

Alternative Konzepte, die in Betracht gezogen wurden, waren die Parallelisierung auf Prozedur- oder Blockebene sowie die Realisierung mehrerer Threads. Parallelisierungen auf Prozedur- oder Blockebene erfordern einen hohen Entwicklungsaufwand, da hier Veränderungen an den Basisroutinen für die Berechnungen erforderlich sind. Sonstige Programmteile sind aufgrund ihres geringen Anteils an der Gesamtrechenzeit für eine Parallelisierung nicht lohnenswert. Auf dieser Ebene der Parallelisierung ist jedoch viel Kommunikation erforderlich. Dies würde nur auf dedizierten Parallelrechner mit kurzen Latenzzeiten hohe Leistungsgewinne erbringen. Die Zielumgebung umfasst jedoch auch Verbünde von Arbeitsplatzrechnern und weitverteilte Architekturen, weswegen dieses Konzept zur Parallelisierung verworfen wurde.

Auch die Realisierung einer vielfädigen Ausführung schränkt die Ausführungsplattform stark ein, da hier große Leistungsgewinne nur bei Mehrprozessormaschinen erreicht werden können. Daher wurde eine Parallelisierung auf Modulebene, durch replizierte DM-Server, angestrebt. Dies kann auch als eine Variante der vielfädigen Ausführung gesehen werden, wobei hier jedoch einzelne Threads auf mehrere Rechner verteilt werden können.

Zur plattformunabhängigen und verteilten Ausführung der Hauptmodule der *Virtuellen Werkstatt* (DMU-Server, Job-Server, Kommunikationsplattform und darunter-

liegende Datenbank) wird eine für die Anwendung transparente Schicht, das *LsD-System*, eingefügt. Diese Schicht, die zwischen der Ausführungsplattform und der Applikation liegt, soll sowohl für die firmeninterne Rechnerarchitektur als auch allgemein für parallele und verteilte Anwendungen eine netzwerkbezogene optimale Ausführungsumgebung und Plattformunabhängigkeit bieten.

Aufgrund der Rechenzeitanalysen stellt sich heraus, dass der Großteil der Rechenzeit (ohne Berücksichtigung der Kommunikationszeit 90% der Gesamtrechenzeit eines Auftrags) in der Konvertierung der Modellrepräsentationen und in den Regelberechnungen des DMU-Servers verbraucht wird. Abbildung 5.7 zeigt hier eine Analyse bestehend aus der Gegenüberstellung der Häufigkeit einer aufgerufenen Methode zu ihrer Rechenzeit [82]. Hieraus geht hervor, dass die Rechenzeit fast gänzlich in den Methoden zur Kollisionsberechnung und Nachbarschaftsbestimmung von Bauteilen verbraucht wird.

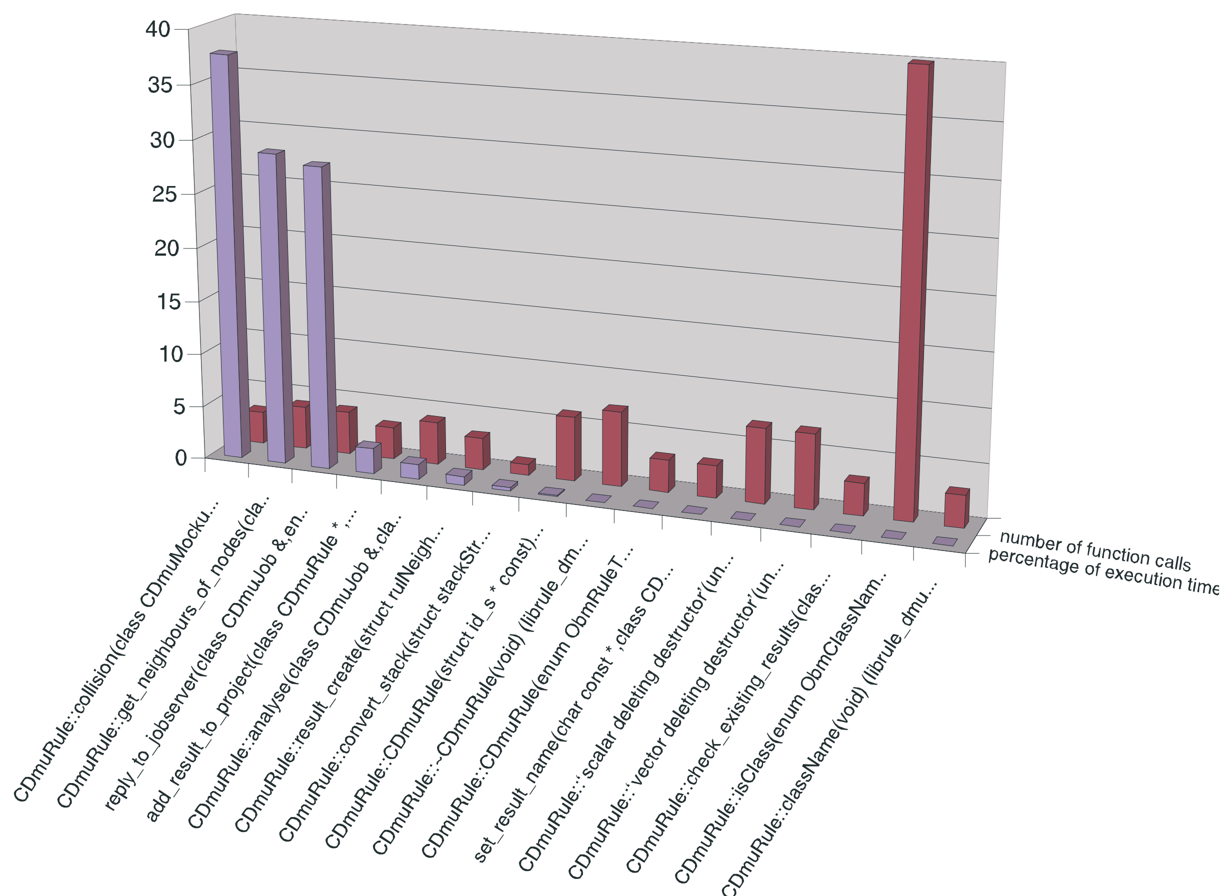

Abbildung 5.7: Analyse der Methodenaufrufe bezüglich ihrer verbrauchten Rechenzeit.

Eine Replizierung der DMU-Server und der damit verbundenen Änderung in den Verwaltungsmodulen der *Virtuellen Werkstatt* sollte in Verbindung mit dem *LsD-System* eine erfolgversprechende Parallelisierungsstrategie für eine parallele *Vir-*

tuelle Werkstatt (viw/p) sein. Hierbei kann das Modell der Threadverteilung des *LsD-Systems* gewinnbringend eingesetzt werden.

Die in diesem Rahmen der Parallelisierung betroffenen Module, die einer Modifikation bedürfen, sind die "Communication Platform", der Job-Server und der DMU-Server. Die vorgenommenen Modifikationen sind transparent für die Datenbank und die Anwendungen der Benutzer. Diese Systeme bedürfen im Rahmen der Parallelisierung keiner Änderungen. Abbildung 5.8 zeigt die Interaktion zwischen der Modulen der geänderten *Virtuellen Werkstatt*.

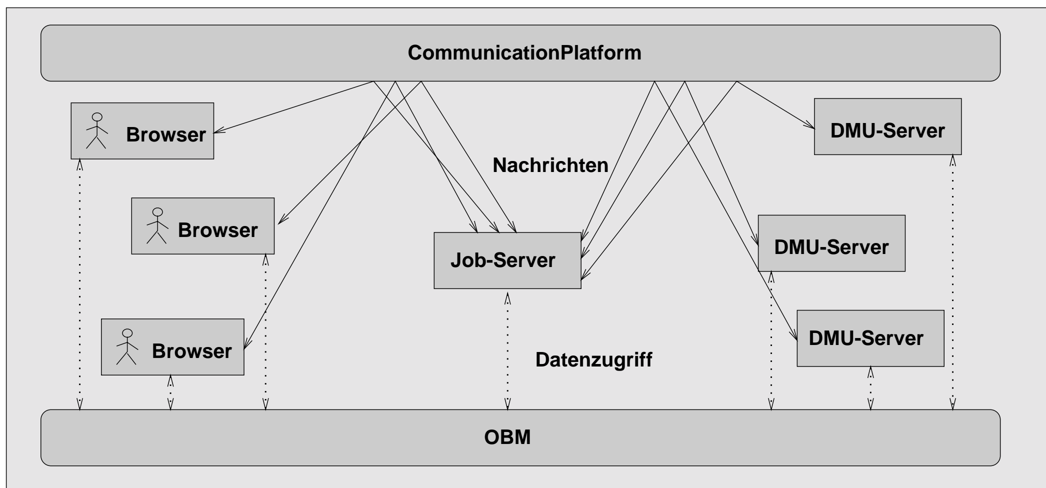

Abbildung 5.8: Interaktion der modifizierten Module der *Virtuellen Werkstatt*

In dem Modul Job-Server wurden die vorhandenen Mechanismen zur Jobverwaltung, die aus einer FIFO (First In First Out) Schlange bestanden, erweitert, so dass mehrere DMU-Server gleichzeitig mit Jobs versorgt werden können. Hierzu ist ebenfalls ein Modul auf der "Communication Platform", das Server-Management, mit einer Schnittstelle zu dem Job-Server versehen worden, damit dieser Informationen über den Status der gegenwärtig vorliegenden Aufträge abrufen kann. In der seriellen Version der *Virtuellen Werkstatt* bestehen Regeln zur Berechnung der Anfragen aus mehreren Subregeln. Diese Unterteilung der Regeln in Subregeln (*Partitioning*) kann nur der DMU-Server vornehmen. Da dies einer sinnvollen Verteilung von Jobs auf mehrere DMU-Server widersprechen würde, werden in der parallelen Version die Ergebnisse dieser Partitionierung dem Job-Server mitgeteilt, und dieser kann daraufhin diese Partitionen auf parallel arbeitende DMU-Server verteilen. Hierbei wird gleichzeitig eine Metrik zur Abschätzung des Berechnungsaufwandes vorgegeben, die dem Job-Server Informationen zur effizienteren Verteilung zur Verfügung stellt. Abbildung 5.9 zeigt diesen Ablauf der Partitionierung und Berechnung von zwei Jobs innerhalb der parallelen *Virtuellen Werkstatt* mit zwei DMU-Servern. In den Tabellen 5.1 und 5.2 sind die Messergebnisse der parallelen "Virtuellen Werkstatt" mit zwei unterschiedlichen Problemgrößen zu sehen. Gemessen wurde die Gesamtausführungszeit der parallelen Version mit einem Rech-

ner (System A) und mit drei Rechnern (System B). Die Zeiten der Module sind ebenfalls angeben.

System	t_{total} (s)	$t_{Job-Server_1}$	$t_{DMU-Server_1}$	$t_{DMU-Server_2}$
A	280	16	1	-
B	170	19	12	21

Tabelle 5.1: Messergebnisse der parallelen "Virtuellen Werkstatt" (1)

System	t_{total} (s)	$t_{Job-Server_1}$	$t_{DMU-Server_1}$	$t_{DMU-Server_2}$
A	84	5	20	-
B	46	2	7	7

Tabelle 5.2: Messergebnisse der parallelen "Virtuellen Werkstatt" (2)

5.3 Ergebnis der Analyse

Die in diesem Kapitel vorgenommene Analyse über die Einsatzmöglichkeiten des *LsD-Systems* in Firmen mittlerer Größe hat gezeigt, dass sowohl die Rahmenbedingungen der Rechnerumgebung für einen Einsatz geeignet sind, als auch die verwendete Applikation für dieses Ausführungsmodell. Existierende Metacomputer können diese Anforderungen nicht in diesem Maße erfüllen. Für die trivialen Systeme (siehe Abschnitt 3.1.1) erfordert die Anwendung "Virtuelle Werkstatt" erheblich mehr Kommunikationsdienste als diese anbieten, für große Metacomputer-Umgebungen (siehe Abschnitt 3.1.3 bis 3.1.5) wäre der Entwicklungsaufwand der Anwendung und der Wartung der Metacomputer-Infrastruktur zu aufwändig. Zwischenlösungen bieten zum einen nicht die Flexibität, basieren nicht auf Java, oder bieten keine Optimierungen für globale Ausführung an. Hier zeigt sich ein leichtgewichtiger, portabler Ansatz als Vorteil.

Für Rechnerumgebung innerhalb der Firma Tecoplan AG trifft die Annahme des *LsD-Systems* zu, dass Gruppen von zueinander gut angebundenen Rechnern existieren, die mit entfernten Gruppen gemeinsam eine Anfrage einer Applikation berechnen können. Diese Situation trifft sowohl für Tecoplan als Dienstleister [3] zu, als auch auch für die Rechnerumgebungen von Kunden aus dem Automobilbau beziehungsweise aus der Herstellung von Produkten aus der Luft– und Raumfahrt. Hier kann die latenzoptimierte Prozessverteilung des *LsD-Systems* große Geschwindigkeitssteigerungen erreichen, ohne dass auf kostenintensive Hochleistungsrechner

[3] Angeboten werden hier die Dienste einer Bauteileüberprüfung

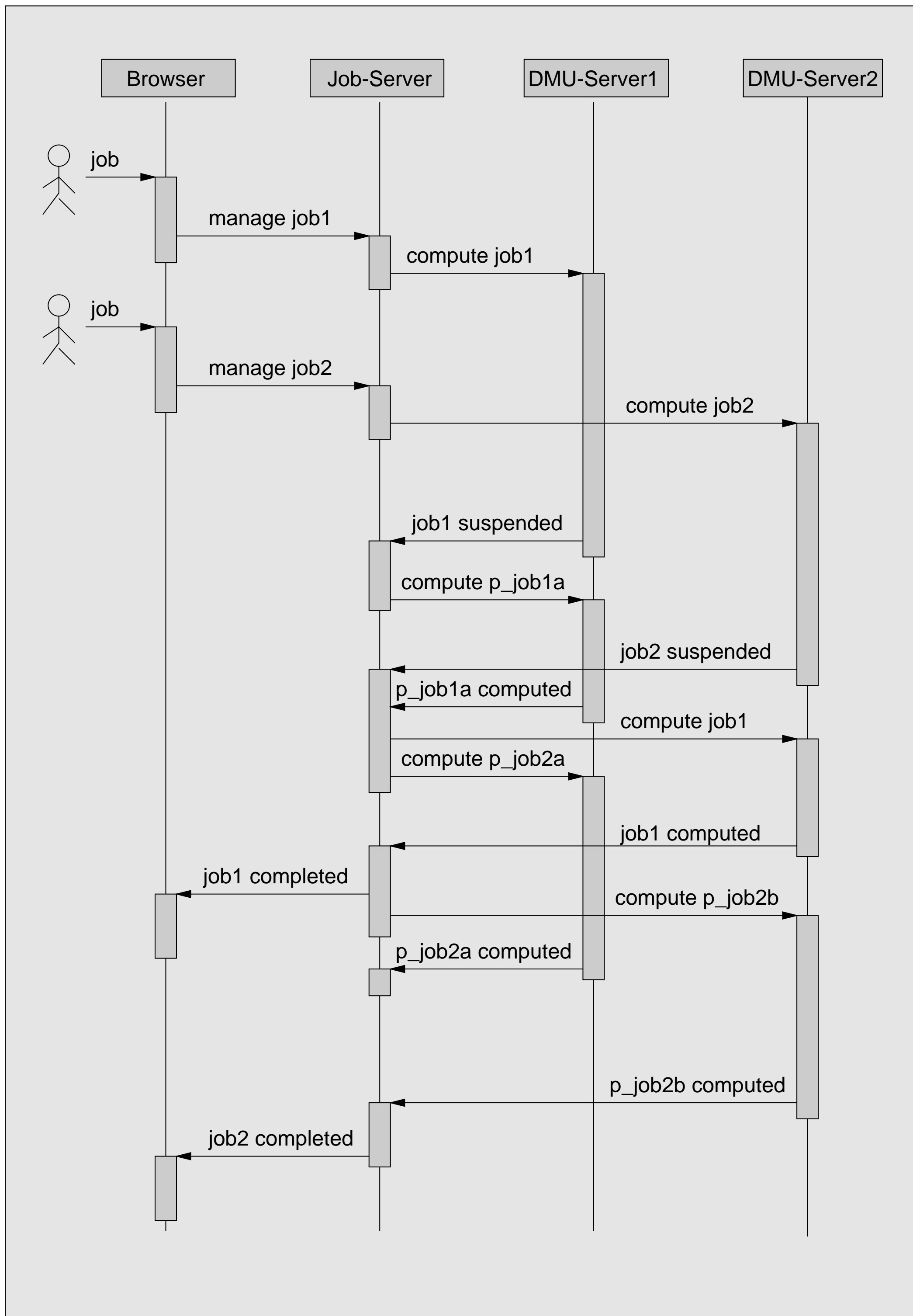

Abbildung 5.9: Partitionierung und Berechnung von zwei Jobs innerhalb der parallelen *Virtuellen Werkstatt viw/p*

zurückgegriffen werden muss. Diese Kosteneinsparung kann einen entscheidenden Wettbewerbsvorteil erbringen.

Auch die untersuchte Anwendung, die *Virtuelle Werkstatt*, erweist sich für das *LsD-System* als sehr geeignet. Nach einer durchgeführten Parallelisierung lassen sich die rechenintensivsten Teile der Anwendung, die DMU-Server, in separate Prozesse kapseln und diese auf mehreren Rechnern zur Ausführung bringen. Hierbei lässt sich die Anzahl der erzeugten Prozesse von der Auftragslast auf dem Job-Server abhängig machen und so eine Skalierung nach Last verwirklichen.

In Abbildung 5.10 ist das Szenario einer weitverteilen Berechnung der *Virtuellen Werkstatt* schematisch dargestellt.

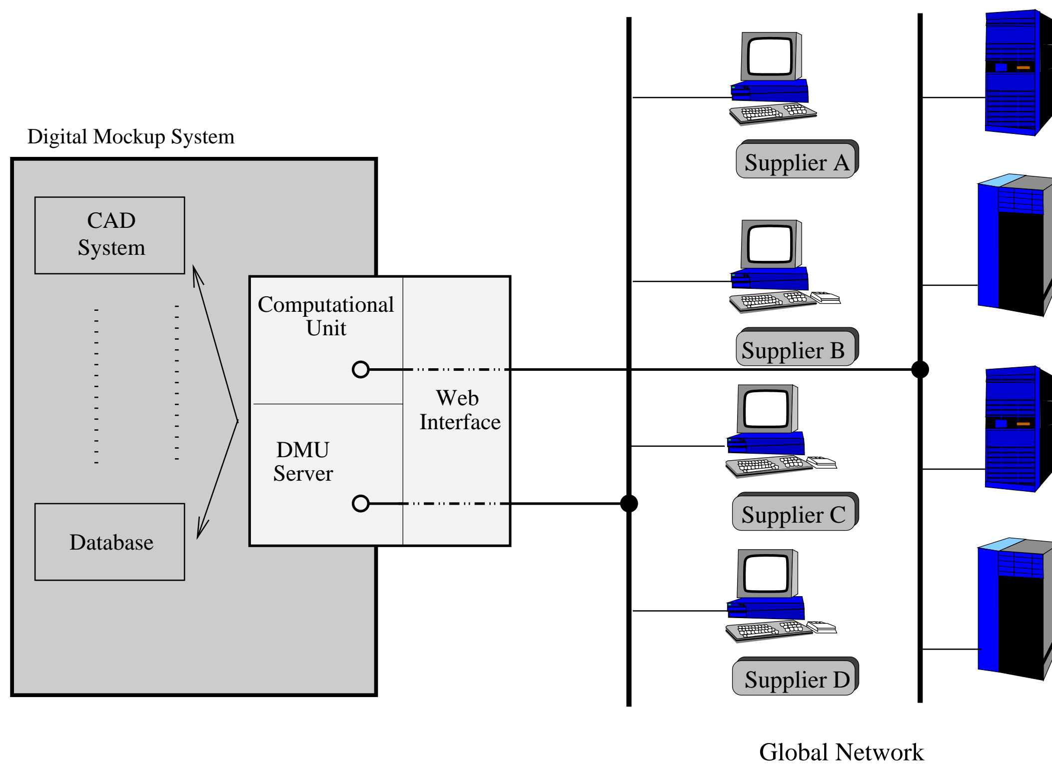

Abbildung 5.10: Struktur einer weitverteilen Berechnung der *Virtuellen Werkstatt*

Mit der Verifizierung dieser beiden Annahmen, der Rechnerumgebung und der Art der Applikation, zeigt sich, dass das *LsD-System* für wissenschaftlich-technische Probleme eine geeignete Ausführungsumgebung zur Kostenersparnis und zur Leistungssteigerung von parallelen Anwendungen darstellt.

6. Zusammenfassung und Ausblick

6.1 Zusammenfassung

In dieser Arbeit wurden die Konzepte und Technologien von sogenannten Metacomputer-Infrastrukturen für wissenschaftlich-technisches Rechnen untersucht. Die zunehmende Vernetzung hat durch das schnell wachsende Internet und der damit verbundenen Technologie auch im wissenschaftlich-technischen Bereich zu einer Globalisierung in Form von global vernetzten Ressourcen geführt.

Diese setzen die Entwicklung zu immer höherer Parallelisierung im wissenschaftlich-technischen Hochleistungsrechnen fort, indem nicht nur lokal vorhandene Ressourcen zur Leistungssteigerung hinzugezogen werden, sondern auch physikalisch entfernte Geräte und Rechner mit einbezogenen werden. Als Kommunikationsmedien dienen dabei entweder dedizierte Hochleistungsverbindungen oder das Internet. Mit der Koppelung mehrerer Hochleistungsrechner oder einer sehr hohen Anzahl von Rechnern über das Internet lassen sich so sehr hohe Rechenleistungen erzielen oder alternativ sehr große Probleme lösen. Auch die Benutzung lokal nicht vorhandender Geräte oder Rechenleistung lässt sich durch die Infrastrukturen virtualisieren. Mit der zunehmenden Rechenleistung steigt jedoch auch die Komplexität und Fehleranfälligkeit der eingesetzten Software, dies gilt zum einen für die Implementation des Metacomputers selber, zum anderen für die Applikationen.

Bei dieser extremsten Form des verteilten Rechnens werden entfernte Ressourcen zu einem virtuellen Hochleistungsrechner zusammengefasst, die über eine einheitliche Schnittstelle angesprochen und benutzt werden können. Die verfügbaren Metacomputer unterscheiden sich dabei in den Diensten, die angeboten werden und in der Technik, diese Dienste zu nutzen. Hierbei können Metacomputer unterschieden werden, die lediglich Rechenressourcen auf Basis einer entfernten Prozessausführung anbieten, bis hin zu komplexen Infrastrukturen, die eine komplette Verwaltung vieler verschiedener Dienste, wie Sicherheitsmechanismen für die An-

wendung und für den Betreiber der Ressource, entfernter Dateizugriff, Rechteverwaltung, entfernte Geräteansteuerung und Visualisierung beinhalten können.

Nach diesen Merkmalen eines Metacomputers lassen sich Anwendungen unterscheiden, die sich für existierende Metacomputer eignen. Applikationen zur Berechnung von Problemen mit trivialer Parallelität und einem hohen Parallelitätsgrad lassen sich auf vielen Architekturen berechnen, insbesondere aber auch auf einfachen Metacomputern, bei denen verteilte Prozesse keine oder nur rudimentäre (indirekte) Kommunikationsmöglichkeiten haben und bei denen neben der Rechenzeit keine weiteren Ressourcen benötigt werden. Eine weitere Klasse der Metacomputer ermöglicht über verschiedene Techniken direkte Client zu Client Kommunikation und bieten diverse Dienste an. Die am weitesten entwickelte Klasse der Metacomputer bilden Systeme, die eine Vielzahl der oben genannten Dienste realisieren und die die Kommunikation der Anwendung optional durch einen gemeinsamen Objektraum abwickeln können. Diese Systeme richten sich an langlaufende wissenschaftlich-technische Anwendungen für Berechnungen aus den Bereichen der Grand Challenges, die den hohen Installations- und Implementierungsaufwand mit hoher Funktionalität und Rechenleistung rechtfertigen.

Wie die Untersuchung der Metacomputer und deren Applikationen zeigt, ist Metacomputing keine universelle Lösung für rechenintensive Applikationen, sondern ist nur für Anwendungen sinnvoll, deren Ablaufstrukturen nicht von den Beschränkungen von weitverteilten Architekturen abhängen. Die wesentliche Einschränkung liegt in dem Kommunikationsverhalten der Anwendung. Ist die Anwendung jedoch für eine weitverteilte Ausführung geeignet, erhält man gute bis sehr gute Ergebnisse, wobei sich besonders bei sehr großen Problemen neue Möglichkeiten ergeben, die bisher ohne diese Architekturen in dieser Größenordnung nicht berechenbar waren.

Ein Großteil der existierenden Metacomputer kann in zwei Klassen eingeordnet werden. Zum einen in die Klasse der sehr einfachen Metacomputer, die Probleme trivialer Parallelität lösen, zum anderen in sehr aufwändige Metacomputer, die ein komplettes, weitverteiltes Betriebssystem realisieren.

In dieser Arbeit wurde das *LsD-System* als eine leichtgewichtige Variante eines Metacomputers vorgestellt. Das *LsD-System* realisiert eine Prozessverteilung für Applikationen zur Leistungssteigerung auf Intra- und Extranets, mit dem Fokus auf kurze Entwicklungszeiten durch geringe Änderungen der Codebasis bestehender Anwendungen. Um günstige, plattformübergreifende Anwendungsentwicklung zu gewährleisten, basiert *LsD* auf der portablen, objektorientierten Programmiersprache *Java*. Der Mechanismus der Prozessverteilung ermöglicht das automatisierte Platzieren von Applikationsprozessen, die in Java Threads eingebettet sind.

In vielen parallelen Anwendungen sind die Kommunikationszeiten der laufzeitentscheidende Faktor. Daher gilt es, besonders bei den hohen Varianzen der Latenzzeiten im Internet, den Einfluss dieses Leistungsengpasses möglichst zu minimieren. Eine zur Laufzeit erzeugte Tabelle der Latenzzeiten aller beteiligten Knoten ermöglicht eine dynamische Zuordnung von häufig kommunizierenden Prozessen zu gut angebundenen Knoten. Hiermit wird eine Optimierung zur Laufzeit erreicht,

die den häufig wechselnden Netzbedingungen von weitverzweigten Architekturen angepasst ist.

Das *LsD-System* ist durch seinen einfachen Aufbau und durch die Vermittlung von sehr hoher Rechenleistung auch für kleinere Installationen geeignet. Zudem lässt es sich auch für andere Middleware-Architekturen als optimierende Prozessverteilungskomponente nutzen. Prozessverteilungsstrategien lassen sich anwendungsabhängig zur Laufzeit hinzubinden, somit können anwendungsspezifische Bindungen zu bestimmten Rechnern ohne Leistungseinbußen gewährleistet werden. Durch die transparente Verteilung von Java Threads ist es zudem ohne Entwicklungsaufwand möglich, Applikationen auch ohne das *LsD-System* aufzusetzen, hiermit kann eine unerwünschte Bindung an externe Software vermieden werden.

Um das *LsD-System* an den Eigenschaften einer realen Zielumgebung zu validieren, wurde als Beispiel die Rechnerinfrastruktur der Firma Tecoplan AG und deren Produkt, die Software "Virtuellen Werkstatt", untersucht. Dabei bestätigten sich die Annahmen über Zielumgebungen in dieser Größenordnung, die zu dem Entwurf des *LsD-Systems* geführt haben. Auch die Parallelisierung der "Virtuellen Werkstatt" konnte mit geringem Aufwand durchgeführt werden. Hier konnte eine Prozessverteilung auf Jobebene realisiert werden, die das *LsD-System* optimal unterstützen kann. Es wurden bei drei Knoten Geschwindigkeitssteigerungen bis 1.7 erreicht. Für den Nachweis der Effizienz des *LsD-Systems* bei der latenzoptimierten Verteilung wurden synthetische Benchmarks entworfen, die die Kommunikation von realen Anwendungen simulieren. Hierbei zeigte sich, dass sich durch die optimierte Verteilung Geschwindigkeitssteigerungen bis zu 1.8 erreichen lassen. Diese Ergebnisse zeigen, dass *LsD*-ähnliche Konzepte eine sinnvolle Alternative zu bestehenden Metacomputern darstellen und diese ergänzen können.

Allgemein kann aus den Untersuchungen in dieser Arbeit geschlossen werden, dass mit der zukünftigen Verbreitung von verschiedenen Metacomputer-Architekturen eine Standardisierung für die Kommunikation zu dessen angebotenen Diensten notwendig ist. Erst hierdurch können Applikationen entwickelt werden, die ohne Bindung zu einer spezifischen Architektur Metacomputer nutzen können und damit die Entwicklungskosten in einem höheren Maße kalkulierbar werden. Auch eine Kommunikation zwischen Metacomputern wäre für eine Weiterleitung der Dienste im Sinne einer globalen Hochleistungsrechner-Infrastruktur.

Eine latenzsensible Verteilung von Prozessen durch das *LsD-System*, nach dessen Kommunikationsaufkommen gewichtet, hat sich als sinnvolle Leistungssteigerungsmaßnahme herausgestellt.

Zusammenfassend ist Metacomputing demnach nicht nur als ein Nebenprodukt der globalen Vernetzung zu sehen, sondern als eine weiter fortschreitende Spezialisierung der Rechnerlandschaft, mit der für bestimmte Applikationen sehr hohe Rechenleistungen erzielt werden kann. Für diese Applikationen kann Metacomputing ein großer Fortschritt bedeuten. Dieser Fortschritt lässt sich jedoch nicht für das wissenschaftlich-technische Hochleistungsrechnen allgemein erweitern. Für einen Großteil der Applikationen lösen Metacomputer das Problem des Hochleistungs-

rechnens nicht, da hier die Anforderungen an die Kommunikationsleistung zu hoch sind.

6.2 Ausblick

Metacomputer-Architekturen werden auch zukünftig im wissenschaftlich-technischen Bereich für geeignete Applikationen eine große Rolle spielen. Die Realisierung eines virtuellen Hochleistungsrechners, der sich dem Benutzer oder der Applikation wie ein einzelner Rechner darstellt, ist jedoch weniger ein technisches Problem als ein logistisches. Es existieren bereits heute Teillösungen der meisten der auftretenden Probleme, die bereits in einige der gegenwärtigen Metacomputer integriert sind. Eine dieser Teillösungen, die Handhabung der unterschiedlichen und hohen Latenzzeiten, wurde in dieser Arbeit vorgestellt. Die Implementierung und Integration aller dieser Teillösungen durch eine Firma oder ein Institut erfordert sehr hohen Aufwand und resultiert in einer proprietären Umgebung, die einem einheitlichen, globalem Zugriff widerspricht. Für eine Realisierung eines homogenen Zugriffs auf globale Ressourcen in dem Sinne von FOSTER [55] (*Computational Grid*) sind daher standardisierte Protokolle notwendig, um auf Dienste Metacomputer-unabhängig zugreifen zu können und um verschiedene Metacomputer-Architekturen koppeln zu können.

Mit dem *LsD-System* wurde gezeigt, dass in globalen Netzen latenzbasierte Optimierung möglich ist und dass dies zur Leistungssteigerung eingesetzt werden kann. Der modulare Aufbau ermöglicht das dynamische Binden von externen Diensten. Hier können zukünftig, für die Anwendung transparent, weitere Dienste und Werkzeuge integriert werden. Visualisierer, Debugger und Leistungsanalysewerkzeuge lassen sich zur Informationsgewinnung an dieser Schnittstelle anwenden. Zur weiteren Automatisierung der latenzbasierten Optimierung sind weiterhin Werkzeuge wünschenswert, die das Kommunikationsverhalten der Anwendungen vor dem Lauf analysieren und auf diese Weise die Verteilung beeinflussen.

Schnittstellen zu anderen Metacomputer-Architekturen für die Bereitstellung der eigenen Dienste und die Nutzung der Dienste fremder Architekturen für eigene Anwendungen wird ein weiterer Schritt sein, um die existierenden Metacomputer zu einer globalen Recheninfrastruktur zu verbinden.

Literaturverzeichnis

[1] ACM [Hrsg.]. *ACM 1998 Workshop on Java for High-Performance Network Computing*, New York, NY 10036, USA, 1998. ACM Press. Also published as *Concurrency: Practice and Experience*, **10**(11–13), September 1998, CODEN CPEXEI, ISSN 1040-3108. http://www.cs.ucsb.edu/conferences/java98/program.html.

[2] A. ALEXANDROV, M. IBEL u. K. SCHAUSER C. SCHEIMAN. SuperWeb: Towards a Global Web-Based Parallel Computing Infrastructure. In *Proceedings of the 11th International Parallel Processing Symposium (IPPS-97)*, S. 100–106, Los Alamitos, April 1–5 1997. IEEE Computer Society Press.

[3] A. D. ALEXANDROV, M. IBEL, K. E. SCHAUSER u. C. J. SCHEIMAN. SuperWeb: research issues in Java-based global computing. *Concurrency: Practice and Experience*, 9(6):535–553, Juni 1997.

[4] G. ALMES, S. KALIDINDI u. M. ZEKAUSKAS. RFC 2679: A One-Way Delay Metric for IPPM, September 1999.

[5] C. ANNESER. *Tecoplan Informatik Virtual Workshop V4: JobServer - DmuServer Kommunikationsschnittstelle*. Tecoplan AG, Ottobrunn near Munich, Germany. Tecoplan internal documentation - not published.

[6] D. ARNOLD, D. BACHMANN u. J. DONGARRA. Request Sequencing: Optimizing Communication for the Grid. In *Proceedings of Euro-Par 2000*, Bd. 1900 LNCS, S. 1213–1222, Berlin, 2000. Springer.

[7] J. E. BALDESCHWIELER, R. D. BLUMOFE u. E. A. BREWER. ATLAS: An Infrastructure for Global Computing. In *Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications*, 1996.

[8] A. BEGUELIN, J. DONGARRA, A. GEIST, R. MANCHEK u. a. HeNCE: Graphical Development Tools for Network-Based Concurrent Computing. In *Scalable High Performance Computing Conference, SHPCC-92, April 26–29, 1992, Williamsburg, Virginia*, S. 129–136, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, April 1992. IEEE Computer Society Press.

[9] A. BEGUELIN, J. DONGARRA, A. GEIST, R. MANCHEK u. a. PVM and HeNCE: Tools for Heterogeneous Network Computing. In J. S. KOWALIK

u. L. GRANDINETTI [Hrsg.], *Software for parallel computation: Proceedings of the NATO Advanced Workshop on Software for Parallel Computation, held at Cetraro, Cosenza, Italy, June 22–26, 1992*, Bd. 106 NATO ASI series. Series F, Computer and systems sciences. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1993.

[10] A. BEGUELIN, J. DONGARRA, A. GEIST, R. MANCHEK u. K. MOORE. HeNCE: a heterogeneous network computing environment. *Scientific Programming*, 3(1):49–60, Spring 1994.

[11] A. BEGUELIN, J. DONGARRA, A. GEIST, R. MANCHEK u. a. HeNCE: A User's Guide, Version 2.0. Technischer Bericht, Computer Science, University of Tennessee, Knoxville, TN 37996, 15 Juni 1994.

[12] T. BEISEL, E. GABRIEL u. M. RESCH. An Extension to MPI for Distributed Computing on MPPs. *Lecture Notes in Computer Science*, 1332(1332):75–82, 1997.

[13] E. BELANI, A. THORNTON u. M. ZHOU. Authentication and security in WebFS, Januar 1997. http:// now.cs.berkeley.edu/WebOS/security.ps.

[14] R. D. BLUMOFE, C. F. JOERG, B. C. KUSZMAUL, C. E. LEISERSON u. a. Cilk: an efficient multithreaded runtime system. *ACM SIGPLAN Notices*, 30(8):207–216, August 1995.

[15] J. C. BONGARD. The Legion System: A Novel Approach to Evolving Heterogeneity for Collective Problem Solving. In RICCARDO POLI, WOLFGANG BANZHAF, WILLIAM B. LANGDON, JULIAN F. MILLER u. a. [Hrsg.], *Genetic Programming, Proceedings of EuroGP'2000*, Bd. 1802 LNCS, S. 16–28, Edinburgh, April 2000. Springer-Verlag.

[16] S. BRADNER. RFC 1242: Benchmarking terminology for network interconnection devices, Juli 1991. Status: informational. ftp://ftp.internic.net/rfc/rfc1242.txt,ftp://ftp.math.utah.edu/pub/rfc/rfc1242.txt.

[17] T. BRÄUNL. *Parallele Programmierung*. Vieweg Verlag, Braunschweig/Wiesbaden, 1993.

[18] T. BRECHT, H. SANDHU, M. SHAN u. J. TALBOT. ParaWeb: Towards worldwide supercomputing. In *Seventh ACM SIGOPS European Workshop:System Support for WorldWide Applications*, 1996.

[19] A. BRICKER, M. LITZKOW u. M. LIVNY. Condor Technical Summary. Technischer Bericht TR 1069, Department of Computer Science, University of Wisconsin, Madison, WI, Oktober 1991.

[20] S. BROWNE, H. CASANOVA u. J. DONGARRA. Providing access to high performance computing technologies. In JERZY WASNIEWSKI, J. DONGARRA,

K. MADSEN u. D. OLESEN [Hrsg.], *Applied parallel computing: industrial-strength computation and optimization: Third International Workshop, PARA 96, Lyngby, Denmark, August 18–21, 1996: proceedings*, Bd. 1184 Lecture Notes in Computer Science, S. 123–133, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1996. Springer-Verlag.

[21] M. BUBAK, W. FUNIKA, K. ISKRA u. R. MARUSZEWSKI. On-Line Performance Monitoring Using OMIS. *Lecture Notes in Computer Science*, 1497, 1998.

[22] H.-J. BUNGARTZ, C. ZENGER u. F. DURST [Hrsg.]. *High performance scientific and engineering computing: proceedings of the International FORTWIHR Conference on HPSEC, Munich, March 16–18, 1998*, Bd. 8 Lecture Notes in Computational Science and Engineering, New York, NY, USA, 1999. Springer-Verlag Inc.

[23] R. BUYYA. *High Performance Cluster Computing: Programming and Applications*. Prentice Hall, Juni 1999.

[24] R. BUYYA. Hot Topics in Cluster Computing and the Grid, Tutorial, August 2000. European Conference on Parallel Computing, Munich, Germany.

[25] L. CAI. Untersuchung der Eignung von Java Web Computern für verteilte und parallele Anwendungen des technisch wissenschaftlichen Rechnens. Diplomarbeit, Technical University Munich, Februar 1999.

[26] D. CAMERON. *Security issues for the Internet and the World Wide Web*. Computer Technology Research Corp., 6 N. Atlantic Wharf, Charleston, SC 29401-2150, USA, 1996.

[27] P. CAPPELLO, B. O. CHRISTIANSEN, M. F. IONESCU, M. O. NEARY u. a. Javelin: Internet-Based Parallel Computing Using Java. In GEOFFREY C. FOX u. WEI LI [Hrsg.], *ACM Workshop on Java for Science and Engineering Computation*, Juni 1997.

[28] N. CARRIERO u. D. GELERNTER. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989. http://www.acm.org/pubs/toc/Abstracts/0001-0782/63337.html.

[29] H. CASANOVA u. J. DONGARRA. NetSolve: A Network-Enabled Server for Solving Computational Science Problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, Juli 1997.

[30] H. CASANOVA u. J. DONGARRA. The Use of Java in the NetSolve Project. In ACHIM SYDOW [Hrsg.], *15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics: Berlin, August 1997: proceedings*, Bd. 4 IMACS World Congress, S. 791–796, Berlin, Germany, 1997. Wissenschaft and Technik.

[31] H. CASANOVA, M. KIM, J. S. PLANK u. J. J. DONGARRA. Adaptive Scheduling for Task Farming with Grid Middleware. *The International Journal of High Performance Computing Applications*, 13(3):231–240, Juli 1999.

[32] C. C. I. T. T. RECOMMENDATION X.509. *The Directory-Authentication Framework*, 1988.

[33] S. J. CHAPIN, D. KATRAMATOS, J. KARPOVICH u. A. S. GRIMSHAW. Resource Management in Legion. Technischer Bericht CS-98-09, Department of Computer Science, University of Virginia, Februar 1998. ftp://ftp.cs.virginia.edu/pub/techreports/CS-98-09.ps.Z.

[34] STANDARD PERFORMANCE EVALUATION CORPORATION. SPEC Benchmark Suite Release, 1990.

[35] STANDARD PERFORMANCE EVALUATION CORPORATION. SPEC Benchmark Specifications, 1995.

[36] MICROSOFT CORPORATION. DCOM architecture. Technischer Bericht, Microsoft Corporation, 1998.

[37] D. K. CULLERS, IVAN R. LINSCOTT u. BERNARD M. OLIVER. Signal processing in SETI. *Communications of the ACM*, 28(11):1151–1163, November 1985. http://www.acm.org/pubs/toc/Abstracts/0001-0782/4549.html.

[38] S. DEERING u. R. HINDEN. RFC 2460: Internet Protocol, Version 6 (IPv6) Specification, Dezember 1998. Obsoletes RFC1883. Status: DRAFT STANDARD. ftp://ftp.math.utah.edu/pub/rfc/rfc1883.txt, ftp://ftp.math.utah.edu/pub/rfc/rfc2460.txt.

[39] T. A. DEFANTI, I. FOSTER, M. E. PAPKA, R. STEVENS u. T. KUHFUSS. Overview of the I-WAY: Wide-Area Visual Supercomputing. *The International Journal of Supercomputer Applications and High Performance Computing*, 10(2/3):123–131, September 1996.

[40] DISTRIBUTED NET. Distributed Net. http://www.distributed.net.

[41] P. DRUM u. G. RACKL. Applying and Monitoring Latency-Based Metacomputing Infrastructures. In P. SADAYAPPAN [Hrsg.], *Proceedings of the 2000 ICPP Workshop on Metacomputing Systems and Applications*, S. 181–188, Toronto, Canada, August 2000. IEEE Computer Society.

[42] T. EICKERMANN, J. HENRICHS, M. RESCH, R. STOY u. R. VOLPEL. Metacomputing in gigabit environments: networks, tools, and applications. *Parallel Computing*, 24(12–13):1847–1872, November 1998.

[43] EUROTOOLS. EuroTools, Java Grande Europe, Special Interest Group, 1999. http://www.irisa.fr/EuroTools/Sigs/Java.html.

[44] J. FARLEY. *Java Distributed Computing*. O'Reilly, 1998.

[45] A. J. FERRARI u. A. S. GRIMSHAW. Basic Fortran Support in Legion. Technischer Bericht CS-98-11, Department of Computer Science, University of Virginia, März 1998. ftp://ftp.cs.virginia.edu/pub/techreports/CS-98-11.ps.Z.

[46] A. J. FERRARI, M. J. LEWIS, C. L. VILES, A. NGUYEN-TUONG u. A. GRIMSHAW. Implementation of the Legion Library. Technischer Bericht CS-96-16, Department of Computer Science, University of Virginia, November 1996. ftp://ftp.cs.virginia.edu/pub/techreports/CS-96-16.ps.Z.

[47] A. FERRARI, F. KNABE, M. HUMPHREY, S. CHAPIN u. A. GRIMSHAW. A Flexible Security System for Metacomputing Environments. Technischer Bericht CS-98-36, Department of Computer Science, University of Virginia, Dezember 1998. ftp://ftp.cs.virginia.edu/pub/techreports/CS-98-36.98.36.ps.Z.

[48] A. J. FERRARI. JPVM: Network Parallel Computing in Java. Technischer Bericht CS-97-29, Department of Computer Science, University of Virginia, Dezember 1997. ftp://ftp.cs.virginia.edu/pub/techreports/CS-97-29.ps.Z.

[49] A. FERRER u. A. GRIMSHAW. Persistent Object State Management in Legion. Technischer Bericht CS-95-36, Department of Computer Science, University of Virginia, August 1995.

[50] D. FLANAGAN. *Java in a nutshell*. O´Reilly, 1996.

[51] M. FLYNN. Very High Speed Computing. *Proceedings of the IEEE*, S. 1901–1909, 1966.

[52] FORTWIHR. Bayerischer Forschungsverbund für technisch-wissenschaftliches Hochleistungsrechnen. http://www.abayfor.de/fortwihr/.

[53] I. FOSTER u. C. KESSELMAN. Globus: A Metacomputing Infrastructure Toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Juni 1997.

[54] I. FOSTER u. C. KESSELMAN. The Globus Project: A Status Report, März 1998.

[55] I. FOSTER u. C. KESSELMAN [Hrsg.]. *The Grid – Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1. Aufl., Juli 1998.

[56] I. FOSTER, C. KESSELMAN u. S. TUECKE. The Nexus Approach to Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing*, 37(1):70–82, August 1996.

[57] I. FOSTER, C. KESSELMAN, G. TSUDIK u. S. TUECKE. A Security Architecture for Computational Grids. In *Proceedings of the 5th ACM Conference on Computer and Communications Security (CCS-98)*, S. 83–92, New York, November 1998. ACM Press.

[58] E. FREEMAN, S. HUPFER u. K. ARNOLD. *JavaSpaces principles, patterns, and practice*. Addison-Wesley, Reading, MA, USA, 1999.

[59] R. FRÖMMING. Untersuchung der im Internet zur Verfügung stehenden Technologien für die Realisierung einer professionell einsetzbaren Client-Server Anwendung. Diplomarbeit, Technical University Munich, November 1997.

[60] A. GEIST, A. BEGUELIN, J. DONGARRA, W. JIANG u. a. PVM 3 User's Guide and Reference Manual. Technischer Bericht ORNL/TM-12187, Massachusetts Institute of Technology, 1994.

[61] M. GERGELEIT. *ONC RPC for Windows NT*. http://set.gmx.de/ mfg/.

[62] V. GETOV, S. FLYNN-HUMMEL u. S.MINTCHEF. A Programming Environment for High-Performance Computing in Java. In R. ALLAN, A. SIMPSON u. D. NICOLE [Hrsg.], *High Performance Computing*. Plenum Publishing, 1998.

[63] GLOBUS PROJECT. Globus, 1998. http://www.globus.org/.

[64] GLOBUS PROJEKT. The Globus Heartbeat Monitor Specification v1.0. http://www.globus.org/hbm/heartbeat_spec.html.

[65] E. FELTEN G. MCGRAW. *Java Security : hostile applets, holes & antidotes*. Wiley , New York, 1997.

[66] L. GONG, M. MUELLER, H. PRAFULLCHANDRA u. R. SCHEMERS. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2. In USENIX [Hrsg.], *USENIX Symposium on Internet Technologies and Systems Proceedings, Monterey, California, Dezember 8–11, 1997*, S. 103–112, Berkeley, CA, USA, 1997. USENIX. http://www.usenix.org/publications/library/proceedings/usits97/gong.html.

[67] J. GOSLING u. H. MCGILTON. The Java Language Environment: A White Paper. Technical Report, Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043, USA, 1996.

[68] A. GRIMSHAW u. W. WULF. Legion: Flexible support for wide-area computing. In *Seventh ACM SIGOPS European Workshop*, S. pp 205–212, September 1996.

[69] A. S. GRIMSHAW u. W. A. WULF. The Legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1):39–45, Januar 1997.

[70] A. S. GRIMSHAW, W. A. WULF, J. C. FRENCH, A. C. WEAVER u. P. F. REYNOLDS, JR. Legion: The Next Logical Step Toward a Nationwide Virtual Computer. Technischer Bericht CS-94-21, Department of Computer Science, University of Virginia, Juni 1994. ftp://ftp.cs.virginia.edu/pub/techreports/CS-94-21.ps.Z.

[71] A. S. GRIMSHAW, W. A. WULF, J. C. FRENCH, A. C. WEAVER u. P. F. REYNOLDS, JR. A Synopsis of the Legion Project. Technischer Bericht CS-94-20, Department of Computer Science, University of Virginia, Juni 1994. ftp://ftp.cs.virginia.edu/pub/techreports/CS-94-20.ps.Z.

[72] A. S. GRIMSHAW, A. NGUYEN-TUONG, M. J. LEWIS u. M. HYETT. Campus-Wide Computing: Early Results Using Legion at the University of Virginia. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):129–143, Juni 1997.

[73] A. GRIMSHAW, A. FERRARI, F. KNABE u. M. HUMPHREY. Legion: An Operating System for Wide-Area Computing. Technischer Bericht CS-99-12, Department of Computer Science, University of Virginia, März 1999. ftp://ftp.cs.virginia.edu/pub/techreports/CS-99-12.ps.Z.

[74] A. S. GRIMSHAW. The Mentat run-time system: support for medium grain parallel computation. In *Proceedings of the Fifth Distributed Memory Computing Conference*, S. 1064–1073, Charleston, SC, April 1990.

[75] A. S. GRIMSHAW. The Mentat Computation Model Data-Driven Support for Object-Oriented Parallel Processing. Technischer Bericht CS-93-30, University Of Virginia, Mai 93. ftp://uvacs.cs.virginia.edu/pub/techreports/CS-93-30.ps.Z.

[76] NIST INTERNETWORKING TECHNOLOGY GROUP. NIST Net network emulation package. Juni 2000. http://www.antd.nist.gov/itg/nistnet/.

[77] W. GU, N.A. BURNS, M.T. COLLINS u. W.Y.P. WONG. The evolution of a high-performing Java virtual machine. *IBM Systems Journal*, 39(1):135–150, 2000.

[78] E. GÜNTHNER u. M. PHILIPPSEN. Complex Numbers for Java. In *Proc. ISCOPE'99, 3rd International Symposium on Computing in Object-Oriented Parallel Environments*, Nr. 1732 Lecture Notes in Computer Science, S. 1–12, San Francisco, December 7–10, 1999. Springer-Verlag.

[79] K. G. HAMILTON. A Remote Procedure Call System. Technical Report no 70, University of Cambridge Computer Laboratory, England, Dezember 1984. Ph.D. thesis.

[80] R. R. HARPER. Interoperability of Parallel Systems: Running PVM Applications in the Legion Environment. Technischer Bericht CS-95-23, Department of Computer Science, University of Virginia, Mai 1995.

[81] B. HAUMACHER u. M. PHILIPPSEN. More Efficient Object Serialization. *Lecture Notes in Computer Science*, 1586:718–724, 1999.

[82] T. HENNE. Analysis and Implementation of a Software System for Digital Mockup Regarding the Suitability for Parallel or Distributed Execution. Diplomarbeit, Technical University Munich, April 1999.

[83] J. HOWARD. *An Analysis of Security Incidents on the Internet*. Doktorarbeit, Carnegie Mellon University, August 1998.

[84] M. HUMPHREY, F. KNABE, A. FERRARI u. A. GRIMSHAW. Accountability and Control of Process Creation in Metasystems. In *Symposium on Network and Distributed Systems Security (NDSS '00)*, S. 209–220, San Diego, CA, Februar 2000. Internet Society.

[85] IEEE COMPUTER SOCIETY STANDARDS COMMITTEE. WORKING GROUP OF THE MICROPROCESSOR STANDARDS SUBCOMMITTEE u. AMERICAN NATIONAL STANDARDS INSTITUTE. *IEEE standard for binary floating-point arithmetic*. ANSI/IEEE Std 754-1985. IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1985.

[86] K. ISHIZAKI, M. KAWAHITO, T. YASUE, M. TAKEUCHI u. a. Design, implementation, and evaluation of optimizations in a Java^{TM} Just-In-Time compiler. *Concurrency: Practice and Experience*, 12(6):457–475, Mai 2000.

[87] JAVA GRANDE FORUM. Java Grande, März 1998. http://www.javagrande.org/.

[88] R. JONES. The Netperf Homepage. http://www.netperf.org.

[89] C. JONES. *Estimating Software Costs*. McGraw-Hill, 1998.

[90] W. KAHAN u. J. D. DARCY. How Java's floating-point hurts everyone everywhere. In *Proceedings of the ACM Workshop on Java for High Performance Network Computing*, Stanford University, Juni 1998. http://www.cs.berkeley.edu/ wkahan/JAVAhurt.pdf.

[91] J. F. KARPOVICH. Support for Object Placement in Wide-Area Heterogeneous Distributed Systems. Technischer Bericht CS-96-03, Department of Computer Science, University of Virginia, Januar 1996. ftp://ftp.cs.virginia.edu/pub/techreports/CS-96-03.ps.Z.

[92] S. KLIEM. *Dmu-Framework: Getting Started*. Tecoplan AG, Ottobrunn near Munich, Germany. Tecoplan internal documentation - not published.

[93] S. KLIEM. *DmuFramework Class Documentation*. Tecoplan AG, Ottobrunn near Munich, Germany. Tecoplan internal documentation - not published.

[94] D. KRAMER. The Java Platform. Report, Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043, USA, 1996. http://java.sun.com/doc/whitePaper.Platform/JavaPlatform.doc.html.

[95] P. KROPF, J. PLAICE u. H. UNGER. Towards a Web Operating System. In *WebNet '97, Toronto*, November 1997.

[96] S. LAMINE, J. PLAICE u. P. KROPF. Problems of computing on the WEB. In A. TENTNER [Hrsg.], *High Performance Computing*, S. 296–301, 1997.

[97] H. LANGENDÖRFER u. B. SCHNOR. *Verteilte Systeme*. Carl Hanser Verlag, München, 1994.

[98] C. A. LEE, R. WOLSKI, I. FOSTER, C. KESSELMAN u. J. STEPANEK. A Network Performance Tool for Grid Environments. In ACM [Hrsg.], *SC'99: Oregon Convention Center 777 NE Martin Luther King Jr. Boulevard, Portland, Oregon, November 11–18, 1999*, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1999. ACM Press and IEEE Computer Society Press.

[99] LEGION PROJECT. Centurion TestBed. http://www.cs.virginia.edu/̃legion/centurion/Centurion.html.

[100] LEGION PROJECT. Legion Project. http://legion.virginia.edu.

[101] M. J. LEWIS u. A. GRIMSHAW. The Core Legion Object Model. Technischer Bericht CS-95-35, Department of Computer Science, University of Virginia, August 1995. ftp://ftp.cs.virginia.edu/pub/techreports/CS-95-35.ps.Z.

[102] M. J. LEWIS u. A. S. GRIMSHAW. Using Dynamic Configurability to Support Object-Orientation in Legion. Technischer Bericht CS-96-19, Department of Computer Science, University of Virginia, Dezember 1996. ftp://ftp.cs.virginia.edu/pub/techreports/CS-96-19.ps.Z.

[103] G. LINDAHL, S. J. CHAPIN, N. BEEKWILDER u. A. GRIMSHAW. Experiences with Legion on the Centurion Cluster. Technischer Bericht CS-98-27, Department of Computer Science, University of Virginia, August 1998. ftp://ftp.cs.virginia.edu/pub/techreports/CS-98-27.ps.Z.

[104] T. LINDHOLM u. F. YELLIN. *The Java Virtual Machine Specification*. Addison-Wesley, Reading/Massachusetts, 1. Aufl., 1996.

[105] T. LINDHOLM u. F. YELLIN. *The Java Virtual Machine Specification*. Addison-Wesley, Reading/Massachusetts, 2. Aufl., 1999.

[106] J. LINN. Generic Security Service Application Program Interface. Internet Request for Comment RFC 1508, Network Working Group, September 1993.

[107] J. LINN. Generic Security Service Application Program Interface, Version 2. Internet Network Working Group, Standards Track, Request for Comments: RFC 2078, Januar 1997. Obsoletes RFC 1508.

[108] T. LUDWIG, R. WISMUELLER u. M. OBERHUBER. OCM — An OMIS Compliant Monitoring System. *Lecture Notes in Computer Science*, 1156:81ff, 1996.

[109] T. LUDWIG, R. WISMUELLER, V. SUNDERAM u. A. BODE. OMIS — On-Line Monitoring Interface Specification (Version 2.0). Institutsbericht, Technische Universitaet Muenchen, Institut fuer Informatik, Juli 1997.

[110] THOMAS LUDWIG, ROLAND WISMÜLLER u. ARNDT BODE. Interoperable Tools Based on OMIS. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT-98)*, S. 155–155, New York, August 3–4 1998. ACM Press.

[111] PETER LUKSCH, URSULA MAIER, SABINE RATHMAYER, MATTHIAS WEIDMANN u. FRIEDEMANN UNGER. Sempa: Software Engineering for Parallel Scientific Computing. *IEEE Concurrency*, 5(3):64–72, Juli/September 1997.

[112] J. MAHDAVI u. V. PAXSON. RFC 2678: IPPM Metrics for Measuring Connectivity, September 1999.

[113] U. MAIER, G. STELLNER u. I. ZORAJA. Resource Allocation, Scheduling and Load Balancing based on the PVM Resource Manager. In E. H. D'HOLLANDER, G. R. JOUBERT, F. J. PETERS u. U. TROTTENBERG [Hrsg.], *Parallel Computing: Fundamentals, Applications and New Directions, Proceedings of the Conference ParCo'97, 19-22 September 1997, Bonn, Germany*, Bd. 12 Advances in Parallel Computing, S. 711–718, Amsterdam, Februar 1998. Elsevier, North-Holland.

[114] U. MAIER. *Konzepte zur Ressourcenverwaltung für wissenschaftlich-technische Anwendungen in verteilten Rechensystemen*. Dissertation, Technische Universität München, 1999.

[115] M. MAY. *Sammlung und Nutzung freier Ressourcen in Weitverkehrsnetzen*. Dissertation, Technische Universität München, 2000.

[116] MESSAGE PASSING INTERFACE FORUM. MPI: A Message-Passing Interface Standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4):159–416, April 1994.

[117] SUN MICROSYSTEMS. Java Remote Method Invocation Specification. Technischer Bericht, Sun Microsystems, 1997. http://www.javasoft.com/products/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html.

[118] LEIBNIZ RECHENZENTRUM MÜNCHEN. GTS - Gigabit Testbed Süd, Aug. 1998. http://www.lrz-muenchen.de/projekte/gigabit.

[119] JOSÉ E. MOREIRA, SAMUEL P. MIDKIFF u. MANISH GUPTA. From flop to megaflops: Java for technical computing. *ACM Transactions on Programming Languages and Systems*, 22(2):265–295, März 2000. http://www.acm.org/pubs/citations/journals/toplas/2000-22-2/p265-moreira/.

[120] C. NESTER, M. PHILIPPSEN u. B. HAUMACHER. A More Efficient RMI for Java. In *Proceedings of ACM 1999 Java Grande Conference*, S. 152–157, Juni 1999.

[121] K. OBRACZKA u. G. GHEORGHIU. The Performance of a Service for Network-Aware Applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT-98)*, S. 81–91, New York, August 1998. ACM Press.

[122] Object Management Group, Inc., 492 Old Connecticut Path, Framingham, MA 01701. *The Common Object Request Broker: Architecture and Specification*, 1.1, 1992.

[123] R. ORFALI u. D. HARKEY. *Client/Server Programming with Java and CORBA*. John Wiley & Sons, 1998.

[124] V. PAXSON, G. ALMES, J. MAHDAVI u. M. MATHIS. RFC 2330: Framework for IP Performance Metrics, Mai 1998. Status: informational. ftp://ftp.internic.net/rfc/rfc2330.txt, ftp://ftp.math.utah.edu/pub/rfc/rfc2330.txt.

[125] H. PEDROSO, L. SILVA u. J. SILVA. Web-based metacomputing with JET. *Concurrency: Practice and Experience*, 9(11):1169–1173, November 1997. Special Issue: Java for computational science and engineering — simulation and modeling II.

[126] M. PHILIPPSEN, B. HAUMACHER u. C. NESTER. More Efficient Serialization and RMI for Java. *Concurrency: Practice and Experience*, S. to appear, 2000.

[127] M. PHILIPPSEN. JavaGrande – Hochleistungsrechnen mit Java. *Informatik-Spektrum*, April 2000.

[128] G. RACKL, M. LINDERMEIER u. M. RUDORFER. MIMO An Infrastructure for Monitoring and Managing Distributed Middleware Environments. In *Middleware 2000 — IFIP/ACM International Conference on Distributed Systems Platform*, Bd. 1795 Lecture Notes in Computer Science, S. 71–87. Springer-Verlag, April 2000.

[129] M. RATHMAYER u. D. HÜBNER. *Communication Platform - Requirements Specification*. Tecoplan AG, Ottobrunn near Munich, Germany. Tecoplan internal documentation - not published.

[130] C. RÖDER. *Load Management Techniques in Distributed Heterogeneous Systems*. Dissertation, Technische Universität München, 1998.

[131] P. RECHENBERG, G. POMBERGER u. a. *Informatik Handbuch*. Carl Hanser Verlag, 1997. http://www.hanser.de.

[132] M. ROMBERG. The UNICORE Architecture: Seamless Access to Distributed Resources. In *Proc. of the Eighth IEEE Int'l Symp. on High Performance Distributed Computing (HPDC-8)*, Aug. 1999.

[133] B. ROSE. Prototypen gibts es nur noch am Rechner. *VDI-Nachrichten*, 28, Juli 1997.

[134] L. F. G. SARMENTA, S. HIRANO u. S. A. WARD. Towards Bayanihan: Building an Extensible Framework for Volunteer Computing Using Java. In ACM [Hrsg.], *ACM 1998 Workshop on Java for High-Performance Network Computing*, New York, NY 10036, USA, 1998. ACM Press. http://www.cs.ucsb.edu/conferences/java98/papers/bayanihan.ps; http://www.cs.ucsb.edu/conferences/java98/papers/bayanihan.pdf.

[135] L. F. G. SARMENTA. Bayanihan: Web-Based Volunteer Computing Using Java. *Lecture Notes in Computer Science*, 1368, 1998.

[136] B. SCHNEIER. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc, 1996.

[137] MICHAEL SCHRAGE. Piranha Processing - utilizing your down time. *HP-Cwire (Electronic Newsletter)*, August 1992.

[138] M. SCHUMANN. Automatic performance prediction to support cross development of parallel programs. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, S. 88–97. ACM Press, Mai 1996.

[139] V. SESHADRI. IBM High Performance Compiler for Java. AIXpert, IBM's magazine for AIX developers, September 1997.

[140] SETI INSTITUTE. SETI. http://www.seti-inst.edu.

[141] SETI INSTITUTE. SETI@home. http://setiathome.ssl.berkeley.edu/.

[142] N. SHAREEF, D. WANG u. R. YAGEL. Segmentation of Medical Data Using Locally Excitatory Globally Inhibitory OscillatorNetworks. In *World Congress on Neural Networks WCNN '96*, September 1996.

[143] L. SILVA, H. PEDROSO u. J. SILVA. The Design of JET: A Java Library for Embarrassingly Parallel Applications. In A. BAKKERS [Hrsg.], *Parallel Programming and Java, Proceedings of WoTUG 20*, Bd. 50 Concurrent Systems Engineering, S. 210–228, University of Twente, Netherlands, April 1997. World occam and Transputer User Group (WoTUG), IOS Press, Netherlands.

[144] L. SMARR u. C. CATLETT. Metacomputing. *Communications of the ACM*, 35(6):44–52, 1992. http://www.acm.org/pubs/articles/journals/cacm/1992-35-6/p44-smarr/p44-smarr.pdf.

[145] L. SMARR, G. MCRAY, D. DIXON u. E. LANDER. Grand challenges of computational science. In ALYCE KAPROW [Hrsg.], *Computer Graphics (SIGGRAPH '90 Panel Proceedings)*, Bd. 24, S. 1.1–1.27, August 1990.

[146] IEEE/ANSI STANDARD. Pthreads: POSIX threads standard, 1995. IEEE Standard 1003.1c-1995.

[147] G. STELLNER, A. BODE, S. LAMBERTS u. T. LUDWIG. NXLib — A Parallel Programming Environment for Workstation Clusters. *Lecture Notes in Computer Science*, 817:745ff, 1994.

[148] G. STELLNER. CoCheck: checkpointing and process migration for MPI. In IEEE [Hrsg.], *Proceedings of IPPS '96. The 10th International Parallel Processing Symposium: Honolulu, HI, USA, 15–19 April 1996*, S. 526–531, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. IEEE Computer Society Press.

[149] R. W. STEVENS. *Unix Network Programming*. Prentice Hall, ISBN 0 13 949876 1, 1990.

[150] RICHARD W STEVENS. *UNIX Network Programming*. Software Series. Prentice Hall PTR, 1990.

[151] W. RICHARD STEVENS. *TCP/IP Illustrated - The Protocols*. Addison-Wesley, Reading, MA, USA, 1994.

[152] T. SUGANUMA, T. OGASAWARA, M. TAKEUCHI, T. YASUE u. a. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[153] Sun Microsystems. *The Java Virtual Machine Specification*, 1.0 Beta, August 1995.

[154] V. SUNDERAM. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2 (4):315–339, Dezember 1990.

[155] SUN MICROSYSTEMS. Sun. http://www.sun.com/.

[156] TODD TANNENBAUM u. MICHAEL LITZKOW. The Condor distributed processing system. *Dr. Dobb's Journal of Software Tools*, 20(2):40, 42–44, 47–48, 102, Februar 1995.

[157] Tecoplan AG, Ottobrunn near Munich, Germany. *Job Server - Requirements and Specification*. Tecoplan internal documentation - not published.

[158] Tecoplan AG. *Tecoplan Homepage*. http://www.tecoplan.de, http://www.tecoplan.com.

[159] R. THAKUR, W. GROPP u. E. LUSK. An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, S. 180–187, Oktober 1996. http://www.mcs.anl.gov/ thakur/papers/adio.ps.gz.

[160] C. TODD. *OBM+: requirement analysis, definition and specification*. Tecoplan AG, Ottobrunn near Munich, Germany. Tecoplan internal documentation - not published.

[161] A. UMAR. *Distributed Computing and Client-Server Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[162] H. UNGER u. P. KROPF. Overview About the Resource Management in the WEB Operating System. In *Proceedings of the Workshop on Distributed Computing on the Web*, S. 134–140, 1998.

[163] UNIVERSITY OF TENNESSEE/MANNHEIM. TOP500 Supercomputer Sites, November 2000. http://www.top500.org/list/2000/11/.

[164] URESH VAHALIA. *UNIX Internals*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1996.

[165] A. VAHDAT, M. DAHLIN, P. EASTHAM u. T. ANDERSON. WebFS: A Global Filesystem for Fine-Grained Sharing, Oktober 1996. OSDI Works In Progress, October 1996.

[166] A. VAHDAT, P. EASTHAM, C. YOSHOKAWA, E. BELANI u. a. WebOS: Operating System Services for Wide Area Applications. Technical Report CSD-97-938, University of California, Berkeley, Mai 6, 1997.

[167] C. L. VILES u. J. C. FRENCH. Availability and Latency of World Wide Web Information Servers. In USENIX ASSOCIATION [Hrsg.], *Computing Systems, Winter, 1995.*, Bd. 8, S. 61–91, Berkeley, CA, USA, 1995. USENIX.

[168] C. VILES, M. LEWIS, A. FERRARI, A. NGUYEN-TUONG u. A. GRIMSHAW. Enabling flexibility in the legion run-time library. In H. R. ARABNIA [Hrsg.], *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, S. pp 265–274, Juli 1997.

[169] G. V. LASZEWSKI u. I. FOSTER. Usage of LDAP in Globus. ftp://ftp.globus.org/pub/globus/papers/ldap_in_globus.pdf.

[170] M. WAHL. Lightweight Directory Access Protocol, Dezember 1997. Internet RFC 2251.

[171] J. WALDO u. a. JavaSpace Specification - 1.0. Technischer Bericht, Sun Microsystems, März 1998. http://java.sun.com/products/javaspaces.

[172] J. WALDO, G. WYANT, A. WOLLRATH u. S. KENDALL. A Note on Distributed Computing. Technischer Bericht, Sun Microsystems Laboratories, 1994.

[173] M. WEIDMANN, P. DRUM, N. THOMSON u. P. LUKSCH. Towards Real World Scientific Web Computing. In SERGE DEMEYER u. JAN BOSCH [Hrsg.], *Parallel and Object Oriented Scientific Computing — ECOOP'98 Workshop Reader*, Bd. 1543 Lecture Notes in Computer Science, Berlin, 1998. Springer.

[174] M. WEIDMANN. *Objektorientiertes und verteiltes Wissenschaftliches Rechnen*. Dissertation, Technische Universität München, 1999.

[175] R. WISMÜLLER, J. TRINITIS u. T. LUDWIG. OCM: A Monitoring System for Interoperable Tools. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT-98)*, S. 1–9, New York, August 3–4 1998. ACM Press.

[176] R. WISMÜLLER. Debugging Parallel Programs Using DETOP and OMIS, Mai 1998. Vortrag auf dem Aurora Kolloquium der Universität Wien, Österreich.

[177] GARY WRIGHT u. W. RICHARD STEVENS. *TCP/IP Illustrated: Volume 2. The Implementation*. Addison-Wesley, Reading, MA, USA, 1995.

[178] P. WU, S. MIDKIFF, J. MOREIRA u. M. GUPTA. Efficient support for complex numbers in Java. In *Proceedings of the ACM Java Grande Conference, San Fransisco, CA*, 1999.

[179] W. A. WULF, C. WANG u. D. KIENZLE. A New Model of Security for Distributed Systems. Technischer Bericht CS-95-34, Department of Computer Science, University of Virginia, August 1995. ftp://ftp.cs.virginia.edu/pub/techreports/CS-95-34.ps.Z.

[180] F. YELLIN. Low Level Security in Java. In *Fourth International Conference on the World-Wide Web*, MIT, Boston, 1995. http://www.w3.org/pub/Conferences/WWW4/Papers/197/40.html.

[181] D. ZAGORODNOV, F. BERMAN u. R. WOLSKI. Application Scheduling on the Information Power Grid. Technical Report CS2000-0644, University of California, San Diego, Computer Science and Engineering, Januar 2000.

[182] A. ZOMAYA [Hrsg.]. *Parallel & Distributed Computing Handbook*. McGrawHill, 1996.

Definitionen

Skalierbarkeit 15
Verteiltes System 21
Weitverteiltes System 21
Parallele Anwendung 21
Virtueller Hochleistungsrechner 21
Metacomputer 21
Webcomputer 22
Metacomputing 22
Webcomputing 22
Computational Grid 22
starke/schwache Parallelität 23
Effizienz 24
Knoten 25
LsD-System Knoten 25
Prozess 25
Thread 25
Job 26
Computersicherheit 33
Latenz 66

Index

A

Abhängigkeitsgraph 57
Abstraktionsebene 5
Administation
  verteilt 16
Anwendungskommunikation 67
Anwendungszielgruppen 62
Applikationsidentifikatoren 77
asymmetrischen Schlüsselverfahren 51
Atlas 47
Ausführung
  mehrfädige 87
  plattformunabhängig 59
  weitverteilt 1
Ausführungsmodell 74
Ausnahmebehandlung 89
Authentifikation 36, 53
automatische Speicherverwaltung 87
Automobilbau 110
Aware 53

B

Barrieren 9
Basisklassen 50
Beobachtung 28
Betriebssystem 12
Betriebssystemversion 13
Betriebssytementwicklung 12
BLAS 58
Broadcast 9
Broker 43
Browser 117
Bytecode 57

C

CAD 110, 113
Cilk 33, 47
Clients 43
Computational Grid 22, 40, 53
Computersicherheit 33
Condor 32
CORBA 30, 67
coupled fields 3, 104
CRISIS 32
Crossbars 3

D

Datenkonsistenz 57
Datenströme 1
DCOM 30
Debugger 85
DES 40
dezentrale Architekturen 5
Dezentralisierung 1
Dienste
  aktive 19
  Globus 53
  passive 19
digital Mockup 110
Digitales Prototyping 109
Distanzbereiche 113
Distributed Net 40
DMU-Server 116
DoS 34
Dynamische Struktur **14**
dynamisches Klassennachladen . . 87

E

echo 84
ECHO REQUEST 84
EDM 112
Effizienz 24
Elektrizitätsnetz 5
entfernte Methodenaufrufe 31
Entwicklungskosten 6
Entwurfsprozess 110

F

Fehlersuche 32
Fehlertoleranz 13
  Legion 49
Feldzugriff 94
Fibonacci 47
Flaschenhals 15
Fliesskommaleistung 94
FORTRAN 49
FORTWIHR 109
Foster 22
FQDN 81
Freiraumbestimmung 113

G

Garbage Collection 30, 87
gemeinsamer Objektraum 29
gemeinsamer, weitverteilter Speicher 30
Geometrieinformationen 110
Geschwindigkeitssteigerung 61
Geschwindigkeitszuwachs 24
Gigabit Ethernet 3
Globalisierung 125
Globus 5, **52**, 61, 73
Grand Challenges 5, 10
Granularität 23
Gruppenbildung 3, 9
GUSTO 55

H

HBM 31
HBMCL 32
HBMDC 32
HBMLM 32
HeNCE 56
Heterogenität **13**
Hitachi 11
Hochleistungsrechner 1, 125
  dediziert 9
Hot Spot 91
HTTP 43
Hypercube 3

I

ICMP 84
Identifikation 36
Identifikatoren 49
IDL 30
IEEE 754 94
Informationen 12
Informationsgewinnung 28
Infrastrukturen 125
Initialkomponente 70
Institutsautonomie 48
Instruktionsstrom 1
Internet
  Hosts 12
Internets 9
IPv6 34

J

Java 6, 87, **87**
  Sicherheitsmechanismen 89
  Spracheigenschaften 88
  synchronized 57
Java Virtuelle Maschine 26
Java Thread 57, 77
Java Virtuelle Maschine . . 27, 70, 78, 99
JavaSpace 29
Javelin 43
JET 43
Job 26
  LsD 26

Job-Server 116
JPCL 57
JPRS 57
Just in Time Compiler 91
JVM 27, 90

K

Kapselung 19
Kesselman 22
Klassenbibliotheken 87
Klassifikation 17
Knoten 25
    LsD 25
Knotenverwaltung 77
Kollisionen 112
Kollisionsberechnung 114
kommerzielle Anwendungen 62
Kommunikation 2, 27
    Client/Server 96
    direkt 30
    explizit 23, 31
    Globus 54
    implizit 31
Kommunikationsaufkommen 82
Kommunikationsbibliotheken 6
Kommunikationslatenz 71
Kommunikationsmatrix 87, 97
Kommunikationsprotokolle 13
Komponenten 15
Konstruktionsprozess 110

L

Lösungsraum 71
LAPACK 58
Lastverteilung 16, 27
Lastwerte 13
Latenz **66**
Latenzen 81
Latenzschwankungen 96
Latenztabelle 77, 81
Latenzverzögerung 101
Latenzzeiten 13
Laufzeitverhalten 28
LDAP 54
Legion 5, **48**
    Basisklassen 50
Linda 29, 44
    Tupelspace 29
LINPACK 58
Linux 2, 101
LOA 49
LOID 49
LsD 74
    Events 77
    Funktionalitäten 74
    Initialverteilung 75
    Module 75
    Prozess 78
    Prozesskapselung 78
    Schnittstelle 85
    Schnittstellen 80
    Subsystem 67

M

MasterDaemon 70, 74
MayI 51
MDS 54, 73
Mersenne Primzahlen 44
Mersenne Primzahlen 40
Message Passing 49
Metacomputer 10
    Aufbau 11
    Definition 21
    Dienstleister 26
    Entwicklung 9
Metacomputing
    Klassifikation 17
Microsoft 30
Middleware
    Atlas 48
    Aufbau 11
    Basisdienst 12
MIDL 30
Migration 79
MIMD 1
MIMO 32
    CORBA 86

LsD 85
Schichtenmodell 32
MIVIS 32
Modelldifferenz 113
Modellrepräsentation 110
Monte Carlo Simulationen 52
MPI 9, 31
MPL 49
MTBF 45
Myrinet 3

N

Nachbarschaftbestimmung 113
Nachrichtenqueue 77
Nachrichtenverwaltung 77
Namensserver 81
netperf 85
NetSolve 58
Netzgeeignet 88
Netzinfrastruktur 3
Netztopologie 13
Netzwerkbandbreite 13
NIST Net 101
NodeDaemon 77

O

Object Management Group 30
Objektdatei 88
objektorientiert 89
objektorientierte Umgebungen . . . 18
Objektorientierung 89
Objektraum
gemeinsam 5
OCM 32
OMIS 32
Optimalsteuerprobleme 2

P

P-Code 88
parallele Programmierung 18
Parallelisierung 68, 114
Parallelität
grobgranular 30
starke/schwache 23
trivial 40, 43, 126
triviale 23
ParaWeb 57
Performanz 15
Peripheriegeräte 12, 13, 61
Permutationen 71
persistente Objekte 29
ping 85
Plattformunabhängigkeit 88
Portabilität 6, 15
Problem Solving Environment . . . 58
Programmiermodelle 9
Programmierumgebung
graphisch 56
Programmierumgebungen 31
Protokoll
generisch 59
Legion 49
Prototyp 110
Proxy 31
Prozess 25
Prozessidentifikatoren 77
Prozessmigration 27
Prozessverteilung 27, 50, 70, 109
Prozessverwaltung 53, 77
Public Key 51
PVM 9, 31, 49

R

RC5 40
Rechenlast 72
Rechnerumgebung 121
Rekonfiguration 49
Release Konsistenzmodell 57
Remote Method Invocation 31
Ressourcen
opportunistisch 43
Sciherheit 34
Ressourcenmanagementsystem . . . 32
RMI 31
RPC 31
RSA 40, 49

S

Scheduler 50
Scheduling 12
Serialisierung 95
SETI 41
Sicherheit 15, 28, **33**, 51
  Legion 49
Sicherheitspolitik 36
Sicherheitsrichtlinien 36, 51
Sicherungspunkterzeugung 45
SIMD 1
Skalierbarkeit 59
  Anforderungen 2
  Definition 15
  Information 15
  Legion 48
Smarr 10
Sockets 83
Speedup 24
Speicherausbau 13
Stapelbetrieb 32
Steuerungsmodul 70
Strömungssimulationen 2
strukturmechanische Simulationen . 2
Suchraum 71
Sun Microsystems 87
SuperWeb 43
System
  Client/Server 20
  verteilt 20, 21
  weitverteilt 21

T

TCP 34, 54, 84, 96
Tecoplan AG 109
TestBed 48, 52, 55
Thread 26
Threadmodell 78
traceroute 85
trusted environment 34

U

UDP 45, 84
Umgebungsmerkmale 74
Unix 41
URL 77

V

Vektorrechner 1, 2
Verbünde von Arbeitsplatzrechnern 1, 11
Verfügbarkeit 2
verteilte Ressourcen 39
Vertraulichkeit 16
Virtuelle Werkstatt 109, 112
Virtueller Hochleistungsrechner 9, 21
Visualisierer 61, 85, 128
Visualisierung 28, 126
Vorhersehbare Laufzeit 14
voxelbasiert 109
Voxeltechnik 112

W

Warehouse 59
Webcomputer 22
WebFS 32
WebOS 32
Werkzeuge 28
Wettbewerbsvorteil 6
work stealing 47
WOS 59

Z

Zertifikat 49
Zielumgebung 73
Zugriffsrechte 13