Universität
LÜNEBURG

Diplomarbeit

Christian Hager

Entwurf und Implementierung eines verteilten Systems mit Jini und JavaSpaces

Fakultät III - Umwelt und Technik
Fachbereich Automatisierungstechnik

Faculty III – Environment and Technology
Department of Automation Technology

Christian Hager

Entwurf und Implementierung eines verteilten Systems mit Jini und JavaSpaces

WS 2006/07

Diplomarbeit eingereicht im Rahmen der Diplomprüfung
im Studiengang Ingenieur-Informatik
am Fachbereich Automatisierungstechnik
der Fakultät für Umwelt und Technik
der Universität Lüneburg

Betreuender Prüfer : Prof. Dr. rer. nat. Dipl.-Inform. Helmut Faasch
Zweitprüfer : Prof. Dr.-Ing. Dipl.-Inform. Eckhard C. Bollow

Lüneburg, den 05.03.2007

Zusammenfassung

Verfasser: Christian Hager

Thema der Diplomarbeit: Entwurf und Implementierung eines verteilten Systems mit Jini und JavaSpaces

Diese Arbeit beschreibt die Entwicklung eines Systems zur verteilten Bearbeitung rechenintensiver Probleme der Firma Analytic Company GmbH. Aufwendige Prozesse sollen dabei parallel auf mehreren Rechnern abgearbeitet werden. Im Ergebnis soll die im Rahmen dieser Arbeit entwickelte Anwendung zu einer schnelleren und flexibleren Abarbeitung rechenintensiver Probleme und besseren Nutzung der bestehen Infrastruktur der Firma führen.

Abstract

Author: Christian Hager

Thesis Title: Design and Implementation of a distributed System using Jini and JavaSpaces

This paper describes the development of a system for the distributed computation of compute intensive Problems of Analytic Company GmbH. Therefore complex and time-consuming processes should be computed parallel on various computers of the company. The result of this paper should lead to an application, which improves the computation of time-consuming problems, in terms of speed and flexibility. On top of this the developed application should lead to an improved utilization of available resources of the company.

Danksagung

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich bei der Anfertigung dieser Diplomarbeit unterstützt haben.

Besonderer Dank gilt meinen beiden Betreuern, Herrn Prof. Dr. Helmut Faasch, der mich durch seine hilfreichen Anregungen immer wieder unterstützt hat, sowie Herrn Prof. Dr. Eckhard C. Bollow für seine Bereitschaft, als Zweitprüfer zu fungieren.

Weiterhin möchte ich mich bei Herrn Ingo Meyer bedanken, der mir aufgrund seines technischen Know-Hows während der Phase der Implementierung eine große Hilfe war und viele wertvolle Hinweise gegeben hat.

Ein herzliches Dankeschön geht außerdem an meine Freundin Anja, die mich während der gesamten Diplomarbeitsphase moralisch unterstützt hat und mir bei Formulierungsproblemen und beim Korrekturlesen zur Seite stand.

Zuletzt möchte ich mich noch bei meinen Eltern und meiner Familie bedanken, die mir dieses Studium und damit auch diese Diplomarbeit überhaupt erst ermöglicht haben.

Erklärung zur Diplomarbeit

Name : Hager
Vorname : Christian
Matr.-Nr. : 1156062
Studiengang : Diplom Ingenieur-Informatik

An den Prüfungsausschuss
des Fachbereichs Automatisierungstechnik
der Universität Lüneburg
Volgershall 1

21339 Lüneburg

Ich versichere, dass ich diese Diplomarbeit – bzw. meinen Teil, der als Gruppenarbeit vergebenen Diplomarbeit – selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Lüneburg, den 05.03.2007

Unterschrift

Inhaltsverzeichnis

Zusammenfassung .... **III**
Abstract **III**
Danksagung **IV**
Erklärung zur Diplomarbeit .... **V**
Inhaltsverzeichnis .... **VI**
Abbildungsverzeichnis .... **IX**
Tabellenverzeichnis .... **X**
Abkürzungsverzeichnis .... **XI**
1 Einleitung **1**
1.1 Motivation 1
1.2 Problemstellung und Idee 2
1.3 Einsatzgebiet 3
1.4 Konventionen und Aufbau der Arbeit 3
2 Analyse **5**
2.1 Beschreibung des Szenarios 5
2.2 Funktionale Anforderungen 5
2.3 Nicht-funktionale Anforderungen 5
2.3.1 Performance (Leistung) 5
2.3.2 Availability (Verfügbarkeit) 5
2.3.3 Security (Sicherheit) 6
2.3.4 Scalability (Skalierbarkeit) und Extensibility (Erweiterbarkeit) 6
2.3.5 Manageability (Überwachbarkeit) 6
2.3.6 Maintainability (Wartbarkeit) 6
3 Technische Grundlagen **7**
3.1 Grundlagen verteilter Systeme und Anwendungen 7
3.1.1 Transparenz 8
3.1.2 Ressourcenteilung 9
3.1.3 Parallelität 9
3.1.4 Skalierbarkeit 9
3.2 Die Organisation verteilter Systeme 9
3.3 Das OSI-Referenzmodell 10
3.4 Objektserialisierung zum Datenaustausch 12

3.5 Entwurfsmuster 13
3.5.1 Das Replicated-Worker-Pattern 13
3.5.2 Das Command-Pattern....14
3.5.3 Das Proxy-Pattern....15
3.6 Remote Method Invocation 16
3.6.1 Stub und Skeleton....17
3.6.2 Registry....17
3.6.3 Übertragung 18
3.6.4 Serialisierung 18
3.7 Jeri 19
3.8 Jini....19
3.8.1 Jini Lookup....20
3.8.2 Services registrieren....21
3.8.3 Discovery - Services finden....22
3.8.4 Leasing....22
3.9 Linda und Tuple-Spaces 22
3.10 JavaSpaces....23
3.10.1 Die Vorteile von JavaSpaces 24
3.10.2 Entries....25
3.10.3 Entries schreiben, lesen und entnehmen....26
3.10.4 Leasing....28
3.10.5 Transaktionen 29
3.11 Eclipse Rich Client Platform....30
3.12 Verwandte Projekte und Abgrenzung....31
4 Design und Realisierung 32
4.1 Gesamtkonzept....32
4.2 Systemkonfiguration....34
4.2.1 Master 34
4.2.2 Worker....34
4.2.3 Middleware....34
4.3 Verwendete Software....36
4.4 Die Klasse TaskEntry....36
4.5 Die Klasse ResultEntry....38
4.6 Entwicklung der Worker-Anwendung 38

4.6.1 Die Klasse GenericWorker 39
4.6.2 Entwurf der Benutzeroberfläche 43
4.7 Entwicklung der Masteranwendung 44
4.7.1 Die Klasse Master 45
4.7.2 Die Klasse JobRunner 45
4.7.3 Watermarking 48
4.7.4 Entwurf der Benutzeroberfläche 50
4.7.5 Der Extension Point Jobs 53
4.8 Die Entwicklung der Jobs 54
4.8.1 Der Demo Job 56
4.8.2 Programmieranweisung für weitere Jobs 58
5 Bedienung und Installation 59
5.1 Installation des JavaSpace 59
5.2 Installation der Worker 61
5.3 Installation des Masters 61
6 Entwicklungsstand 62
7 Leistungs- und Skalierbarkeitsanalyse 63
8 Zusammenfassung und Ausblick 67
8.1 Zusammenfassung 67
8.2 Ausblick 67
Literaturverzeichnis 69
A Inhalt der CD 72

Abbildungsverzeichnis

Abbildung 3.1 Middleware in verteilten Systemen [Tanenbaum & van Steen, 2003, S. 19] 9
Abbildung 3.2 Das Replicated-Worker-Pattern 13
Abbildung 3.3 Das Command-Pattern [Gamma et al., 1994, S.236] 14
Abbildung 3.4 Das Proxy-Pattern [Gamma et al., 1994, S.209] 15
Abbildung 3.5 Kommunikation bei RMI 17
Abbildung 3.6 Jini Service Registration 21
Abbildung 3.7 Das JavaSpace-Interface 27
Abbildung 4.1 Das Gesamtkonzept der Anwendung 33
Abbildung 4.2 Die Klasse TaskEntry 37
Abbildung 4.3 Die Klasse ResultEntry 38
Abbildung 4.4 Benutzeroberfläche des Workers 43
Abbildung 4.5 Worker im Systray 44
Abbildung 4.6 Der Master im Überblick 45
Abbildung 4.7 Die ControlCenter-Perspektive des Masters 50
Abbildung 4.8 Die Maintenance-Perspektive des Masters 52
Abbildung 4.9 Die WorkerMonitor-Perspektive des Masters 53
Abbildung 4.10 Aufbau eines Jobs 55
Abbildung 4.12 Das DistributedJob-Interface 56
Abbildung 4.13 Klassendiagramm des Demo-Jobs 57
Abbildung 5.1 Der Installer des Blitz JavaSpace 59
Abbildung 7.1 Gesamtrechenzeiten im Grid und auf einem Einzelrechner für eine Rechenzeit pro Task von 1s bis 300s 64
Abbildung 7.2 Gesamtrechenzeiten im Grid und auf einem Einzelrechner für eine Rechenzeit pro Task von 10ms bis 500ms 64
Abbildung 7.3 Verhältnis der Rechenzeiten im Grid zu den Rechenzeiten auf einem Einzelrechner 65
Abbildung 7.4 Rechenzeit im Grid bei variabler Workerzahl 66

Tabellenverzeichnis

Tabelle 3.1 Verschiedene Formen der Transparenz in verteilten Systemen [ISO10746, 1996] 8
Tabelle 3.2 The eight fallacies of distributed computing [Deutsch, 2007] 20
Tabelle 7.1 Ergebnisse des ersten Experimentes mit 8 Workern, 50 Tasks und variabler Rechenzeit der Tasks..63
Tabelle 7.2 Ergebnisse des zweiten Experiments mit einer Rechenzeit von 5000ms pro Task, 50 Tasks und variabler Workerzahl 65

Abkürzungsverzeichnis

Abb.	Abbildung
API	Application Programming Interface
BOINC	Berkeley Open Infrastructure for Network Computing
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
DCOM	Distributed Component Object Model
GUI	Graphical User Interface
IDE	Integrated Development Environment
IP	Internet Protocol
JERI	Jini Extensible Remote Invocation
JRE	Java Runtime Environment
OSGi	Open Services Gateway Initiative
OSI	Open Systems Interconnection
RCP	Rich Client Platform
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SWT	Standard Widget Toolkit
TCP	Transmission Control Protocol
UDP	Universal Datagram Protocol
URL	Uniform Resource Locator
VM	Virtual Machine

1 Einleitung

1.1 Motivation

Bei Analytic Company GmbH werden Rechenprozesse jeglicher Art und Aufwendigkeit momentan jeweils auf einem einzelnen Rechner bearbeitet. Diese Rechner sind je nach Rechenaufwand der Prozesse über lange Zeiträume belegt. Deshalb wäre es wünschenswert, rechenintensive Prozesse über mehrere Rechner zu verteilen, um Rechenzeiten zu verkürzen und eine flexiblere Bearbeitung aufwendiger Berechnungen zu gewährleisten. Um dieses Ziel zu erreichen könnte neue, teure und leistungsfähigere Hardware angeschafft werden.

Alternativ dazu ist aber auch die Realisierung der Unternehmensanwendungen als verteiltes System denkbar und sinnvoll. Ein Vorteil des verteilten Systems sind vor allem geringe Kosten für neu zu beschaffende Hardware, da die benötigten Netzwerkkomponenten ohnehin bereits vorhanden sind. Weiterhin wird die Performance aufwendiger Prozesse durch ein verteiltes System wesentlich verbessert. So können aufwendige Prozesse in einem verteilten System wesentlich schneller als auf einzelnen Rechnern fertig gestellt werden.

Ein weiterer negativer Aspekt des bisherigen Systems ist die schlechte Ausnutzung der bestehenden Ressourcen. Mit einem verteilten System könnten zum Beispiel die Entwicklerrechner der Firma bei freien Kapazitäten[1] von anderen Prozessen genutzt werden.

Problematisch ist außerdem die geringe Zuverlässigkeit des derzeitigen Systems. Stürzt ein Prozess während der Bearbeitung ab, so ist die gesamte bereits erfolgte Berechnung verloren. Dies stellt bei langwierigen Prozessen ein erhebliches Problem dar. So konnten in der Vergangenheit bei Systemabstürzen wichtige Berechnungen mitunter nicht termingerecht fertig gestellt werden. Deshalb wäre eine erhöhte Zuverlässigkeit des Systems, genauso wie die schnellere und somit flexiblere Bearbeitung der Geschäftsprozesse, ebenfalls von Vorteil.

[1] vor allem nachts und am Wochenende

Jini bietet in Verbindung mit JavaSpaces einige Ansätze, um diese Ziele zu erreichen. Durch die Nutzung eines JavaSpace könnten lange laufende Prozesse durch die Aufteilung in mehrere Teilprobleme parallel bearbeitet werden. Dadurch wird, wie oben angesprochen, die Performance erhöht ohne dabei hohe Kosten für neue Hardware zu verursachen. Außerdem kann durch den Einsatz der beiden genannten Technologien dem System die Möglichkeit zur Selbstheilung gegeben werden. Fällt ein Rechner aus übernehmen andere automatisch dessen Aufgaben bis er wieder verfügbar ist. Dies würde zu einer Erhöhung der Verfügbarkeit des Systems führen.

1.2 Problemstellung und Idee

Ziel dieser Arbeit ist es, die Nutzung der bestehenden Infrastruktur durch den Einsatz der Java-basierten Technologien Jini und JavaSpaces zu optimieren.
Durch die Verteilung rechenintensiver Prozesse über mehrere Rechner soll außerdem der Zeitaufwand zur Bearbeitung minimiert werden. Weiterhin wird die Zuverlässigkeit des Systems durch eine geeignete Implementierung erhöht. Bei Ausfall eines Rechners übernehmen andere verfügbare Rechner automatisch dessen Aufgaben. Ein weiteres Ziel dieser Arbeit ist es, eine möglichst einfache Erweiterbarkeit des zu implementierenden Systems zu erreichen. Ist die Rechenleistung im Netzwerk nicht mehr ausreichend, sollen ohne großen Aufwand weitere Rechner hinzugefügt werden können. Diese integrieren sich dann möglichst automatisch[2] in das Netzwerk. Die einzelnen Rechner des Netzwerks übernehmen dabei jedoch nicht spezifische Aufgaben, sondern sind in der Lage jede Art von Problem zu berechnen.
Das Ziel der Arbeit ist somit der Entwurf und die Implementierung eines Systems zur Verteilung von Rechenprozessen über ein Netzwerk nach dem Vorbild des BOINC Projektes [BOINC, 2007] der kalifornischen Universität in Berkeley.

[2] Jini

1.3 Einsatzgebiet

Die Anwendung soll im lokalen Netzwerk der Firma zum Einsatz kommen, um dort Probleme aller Art zu berechnen. Speziell alle Entwicklerrechner der Firma sollen in Zukunft eine Worker-Anwendung enthalten, so dass ihre Rechenleistung bei freien Kapazitäten vom Master genutzt werden kann. Dies wird vor allem über Nacht, zu den Pausenzeiten und am Wochenende der Fall sein.

1.4 Konventionen und Aufbau der Arbeit

Im Folgenden sollen nun einige kurze Erläuterungen zum Aufbau der Arbeit gegeben werden.
Kapitel 2 befasst sich mit der Analyse der Anforderungen an die zu entwickelnde Software. Dabei werden nach einer allgemeinen Beschreibung des Szenarios sowohl funktionale als auch nicht-funktionale Anforderungen analysiert. Kapitel 3 beschäftigt sich mit den für die Entwicklung der Anwendung notwendigen technischen Grundlagen. Dabei werden zuerst einige Grundlagen verteilter Systeme erläutert. Im Anschluss erfolgen die Beschreibung einiger verwendeter Entwurfsmuster und die Erläuterung grundlegender, in der Applikation verwendeter, Middleware-Technologien für verteilte Systeme. In diesem Abschnitt erfolgt auch die Beschreibung der Jini-Technologie und des JavaSpace. Danach wird kurz auf die Eclipse Rich Client Platform eingegangen, da sowohl die Master- als auch die Worker-Anwendung auf dieser Plattform aufbauen. Zum Abschluss von Kapitel 3 werden dann einige verwandte Projekte vorgestellt. Kapitel 4 beschäftigt sich mit der Realisierung der Anwendung. Dabei wird nach einer Beschreibung der Systemkonfiguration und einem Überblick über das Gesamtsystem, die Entwicklung der Worker-Anwendung, die Entwicklung der Master-Anwendung und die Entwicklung der Jobs beschrieben. In Kapitel 5 wird die Bedienung und Installation des Systems beschrieben und Kapitel 6 beschreibt den aktuellen Entwicklungsstand der Software. Im 7. Kapitel erfolgt dann die Darstellung der Ergebnisse einer am fertigen System durchgeführten Leistungs- und Skalierbarkeitsanalyse. Kapitel 8 schließt die Arbeit mit einer Zusammenfassung und einem Ausblick ab. Hier werden unter anderem auch

einige Ansatzpunkte für eine Weiterentwicklung des Systems beziehungsweise für Folgeprojekte dargelegt.

2 Analyse

Dieses Kapitel befasst sich mit einer fachlichen Analyse des für diese Arbeit gewählten Fachgebietes. Zu Beginn erfolgt eine allgemeine Beschreibung des Szenarios. Im Anschluss werden sowohl funktionale als auch nicht-funktionale Anforderungen an die zu entwickelnde Software ermittelt.

2.1 Beschreibung des Szenarios

Ziel dieser Arbeit ist es Rechenprozesse der Firma Analytic Company GmbH verteilt in deren Netzwerk zu berechnen. Damit soll zum einen die Berechnung oben genannter Prozesse beschleunigt und zum anderen die bestehende Hardware besser ausgenutzt werden.

2.2 Funktionale Anforderungen

Die funktionalen Anforderungen an die zu entwickelnde Anwendung stellen sich wie folgt dar. Die zu entwickelnde Anwendung muss von den Entwicklern der Firma programmierte Probleme aufnehmen und diese verteilt im Netzwerk berechnen können.

2.3 Nicht-funktionale Anforderungen

2.3.1 Performance (Leistung)

Das Softwaresystem ist so auszulegen, dass bei entsprechenden Teil-Problemen eine im Vergleich zu Einzelrechnern wesentlich verbesserte Leistung gewährleistet ist.

2.3.2 Availability (Verfügbarkeit)

Das Softwaresystem ist außerdem so auszulegen, dass eine größtmögliche Verfügbarkeit gewährleistet ist. Es sollte somit Fähigkeiten zur Selbstheilung haben. Dies bedeutet, dass Teile des Systems automatisch die Aufgaben ausgefallener Teile übernehmen.

2.3.3 Security (Sicherheit)

Da der Quellcode der Firma Analytic Company GmbH deren Kapital darstellt, muss eine größtmögliche Sicherheit des Codes gewährleistet werden. Da der Code in Form von Tasks im Netzwerk zirkuliert, sind geeignete Maßnahmen zu ergreifen, um den Code vor unberechtigtem Zugriff sowohl durch Mitarbeiter der Firma als auch externe Personen zu schützen.

2.3.4 Scalability (Skalierbarkeit) und Extensibility (Erweiterbarkeit)

Das System soll durch geeignete Maßnahmen so ausgelegt sein, dass eine hohe Skalierbarkeit sowie eine einfache Erweiterbarkeit gewährleistet sind. Ist die Rechenleistung im Netzwerk nicht mehr ausreichend, sollen ohne großen Aufwand neue Rechner integriert werden können. Das bedeutet unter anderem auch, dass sich neue Rechner, nachdem die Anwendung installiert wurde, weitestgehend automatisch in das Netzwerk integrieren.

2.3.5 Manageability (Überwachbarkeit)

Zur Überwachung sollen System-Logs auf einem zentralen Rechner gesammelt und zur Auswertung angezeigt werden. Worker-spezifische Nachrichten sollen zusätzlich auf einer grafischen Oberfläche auf dem Worker angezeigt werden.

2.3.6 Maintainability (Wartbarkeit)

Der Wartungsaufwand der Anwendung soll so gering wie möglich gehalten werden. Bei Implementierung zusätzlicher Probleme zur Bearbeitung im System soll kein Update der Worker nötig sein. Stattdessen enthalten die Tasks die Implementierung des zu ihrer Berechnung nötigen Quellcode selbst.

3 Technische Grundlagen

Zur Entwicklung eines Systems zur verteilten Berechnung unterschiedlicher Tasks ist es nötig, einige technische Grundlagen verteilter Systeme zu verstehen. In diesem Kapitel werden deshalb zunächst die theoretischen Grundlagen verteilter Systeme erläutert und im Anschluss die für das Projekt relevanten Technologien beschrieben.

3.1 Grundlagen verteilter Systeme und Anwendungen

Was charakterisiert also eine verteilte Anwendung beziehungsweise ein verteiltes System? Laut [Deutschmann et al., 2004] definiert sich eine verteilte Anwendung wie folgt. *„Eine verteilte Anwendung besteht aus mehreren nebenläufigen, untereinander kommunizierenden Prozessen, welche gemeinsam die vom Anwender geforderte Leistung erbringen.“* Ein verteiltes System definiert sich laut [Tanenbaum & van Steen, 2003] wie folgt. *„Ein verteiltes System ist eine Menge voneinander unabhängiger Computer, die dem Benutzer wie ein einzelnes, kohärentes System erscheinen.“* Eine ähnliche Definition liefert [Deutschmann et al., 2004]. *„Ein verteiltes System ist ein Hard- und Software-System, dessen Daten und Funktionseinheiten auf mehrere zu einem Netz zusammengeschlossene Rechner verteilt sind. Für den Benutzer sollte die Verteilung möglichst nicht sichtbar sein.“*

Ein verteiltes System hat demnach, laut beider Definitionen, zwei Aspekte. Es besteht zum einen aus mehreren autonomen homogenen oder heterogenen Rechnern. Dem Benutzer hingegen wird vorgespiegelt er hätte es mit einem einzigen System zu tun. Die Unterschiede zwischen den einzelnen Systemen sowie die Kommunikation zwischen diesen, bleiben dem Benutzer verborgen.

Als direkte Konsequenz aus der Verwendung voneinander unabhängiger Rechner ergibt sich laut Tanenbaum, dass verteilte Systeme relativ einfach zu erweitern oder zu skalieren sind.

Ein verteiltes System hat demnach die in den folgenden Abschnitten beschriebenen charakteristischen Eigenschaften.

3.1.1 Transparenz

Ein Charakteristika, welches bereits aus Tanenbaums Definition resultiert, ist die Tatsache, dass verteilte Systeme verbergen, dass sie aus mehreren Rechnern bestehen. Die Tatsache, dass sich verteilte Systeme dem Benutzer somit als ein System darstellen bezeichnet man auch als Transparenz.
Der internationale Standard „Open Distributed Processing Reference Model" (ISO/IEC 10746) der ISO, aus [ISO, 1996], definiert zum Thema Transparenz das Folgende. *„Distribution transparencies enable complexities associated with system distribution to be hidden from applications where they are irrelevant to their purpose."* Transparenz in verteilten Systemen versteckt demnach Schwierigkeiten, die durch die Verteilung entstehen und für den Nutzer irrelevant sind. Tabelle 3.1 zeigt die verschiedenen Aspekte eines verteilten Systems auf die das Konzept der Transparenz angewendet werden kann.

Transparenz	**Beschreibung**
Zugriffstransparenz	Verbirgt Unterschiede im Zugriff auf Ressourcen
Positionstransparenz	Verbirgt den Ort einer Ressource
Migrationstransparenz	Verbirgt, dass sich die Position einer Ressource ändern kann
Relokationstransparenz	Verbirgt, dass sich die Position einer Ressource ändern kann, während sie verwendet wird
Replikationstransparenz	Verbirgt, dass eine Ressource repliziert ist
Nebenläufigkeitstransparenz	Verbirgt, dass eine Ressource von mehreren konkurrierenden Benutzer gleichzeitig genutzt werden kann
Fehlertransparenz	Verbirgt den Ausfall und die Wiederherstellung von Ressourcen
Persistenztransparenz	Verbirgt, ob eine Ressource sich im Speicher oder auf einer Festplatte befindet

Tabelle 3.1 Verschiedene Formen der Transparenz in verteilten Systemen [ISO10746, 1996]

3.1.2 Ressourcenteilung

Verteilte Systeme teilen sich aufgrund ihrer Struktur häufig diverse Ressourcen. Dies können beispielsweise Drucker, Massenspeicher, Datenbanken oder andere Geräte oder Dienste sein. Ein gutes Beispiel für einen solchen Dienst ist der in der im Rahmen dieser Arbeit eingesetzte JavaSpace, welcher in Abschnitt 3.10 näher beschrieben wird.

3.1.3 Parallelität

Resultierend aus der Struktur verteilter Systeme und Anwendungen laufen Prozesse innerhalb dieser gleichzeitig ab.

3.1.4 Skalierbarkeit

Eine wichtige Charakteristik eines verteilten Systems ist seine Skalierbarkeit. Um die Skalierbarkeit eines verteilten Systems zu erhöhen, wird neben anderen die Methode der Verteilung verwendet. Dabei wird eine Komponente in kleinere Teile zerlegt und diese im Anschluss über das gesamte System verteilt. Damit lässt sich mitunter die Leistung eines verteilten Systems wesentlich verbessern.

3.2 Die Organisation verteilter Systeme

Um heterogene Netzwerke zu unterstützen, gleichzeitig aber den Eindruck eines einzigen Systems zu erwecken, organisieren sich verteilte Systeme häufig mittels einer Middleware-Schicht.

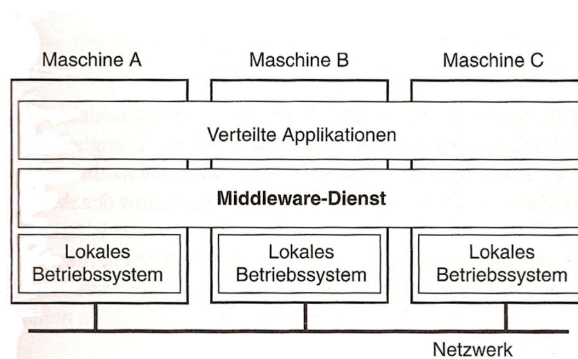

Abbildung 3.1 Middleware in verteilten Systemen [Tanenbaum & van Steen, 2003, S. 19]

Abbildung 3.1 zeigt die logische Positionierung dieser Schicht zwischen der darüber liegenden Applikationsebene und der darunter liegenden Ebene aus Betriebssystemen. Im Fall der im Rahmen dieser Arbeit entwickelten Anwendung, fungiert Jini in Verbindung mit dem JavaSpace als Middleware.

Zusammenfassend lässt sich also festhalten, dass verteilte Systeme aus autonomen Rechnern bestehen, sich aber nach außen hin als kohärentes System präsentieren. Verteilte Systeme erleichtern die Integration verschiedener Anwendungen auf verschiedenen Rechnern des Systems. Bei korrektem Design sind verteilte Systeme gut zu skalieren. Nachteile sind die häufig komplexere Software, die verschlechterte Leistung sowie eine oft schwächere Sicherheit.

3.3 Das OSI-Referenzmodell

Die Basis eines jeden verteilten Systems ist die Kommunikation über das Netzwerk. Diese lässt sich am Besten anhand des OSI–Referenzmodelles[3] der International Organization for Standardization beschreiben. Die Architektur des OSI-Modelles basiert auf sieben aufeinander aufbauenden Schichten, welche von der Kommunikation im Netzwerk durchlaufen werden und die im Folgenden näher erläutert werden sollen.

Anwendungsschicht (Application Layer): Der ursprüngliche Zweck der Anwendungsschicht war es, mehrere Standard-Netzwerkapplikationen, wie beispielsweise E-Mail, Dateitransfer oder Terminal-Emulationen aufzunehmen. Laut [Tanenbaum & van Steen, 2003] ist die Anwendungsschicht heute allerdings zu einem Container für alle Applikationen und Protokolle geworden, die sich nicht in darunter liegende Schichten einordnen lassen. Aus Sicht des OSI-Modelles handelt es sich deshalb bei fast allen verteilten Systemen einfach um Applikationen. So ist auch die hier entwickelte Anwendung in die Anwendungsschicht einzuordnen.

---

[3] vgl. [ISO, 1994]

Darstellungsschicht (Presentation Layer): Die Darstellungsschicht sorgt für eine einheitliche Bereitstellung von Daten für die Anwendungsschicht und ermöglicht damit den syntaktisch korrekten Datenaustausch zwischen unterschiedlichen Systemen. Falls erforderlich fungiert die Darstellungsschicht dabei als Übersetzer zwischen verschiedenen Datenformaten.

Sitzungsschicht (Session Layer): Die Sitzungsschicht stellt Dienste für einen organisierten und synchronisierten Datenaustausch zur Verfügung. So werden in lange Übertragungen Prüfpunkte eingefügt, damit im Falle eines Absturzes nur die Rückkehr zum letzten Prüfpunkt notwendig ist und nicht die gesamte Übertragung wiederholt werden muss. Im praktischen Einsatz existiert diese Schicht, wie zum Beispiel in der Internet Protokollfolge, meist nicht.

Transportschicht (Transport Layer): Die Transportschicht stellt eine vollständige Ende-zu-Ende Kommunikation zwischen zwei Prozessen her, so dass sich die darüber liegenden anwendungsorientierten Protokolle nicht um die Eigenschaften des Kommunikationsnetzes kümmern brauchen. Die bekanntesten Protokolle der Transportschicht sind TCP und UDP.

Vermittlungsschicht (Network Layer): Die Vermittlungsschicht ist für das so genannte Routing der Datenpakete zuständig. Als Routing bezeichnet man die Aufgabe, den besten Weg für ein Datenpaket von A nach B zu finden. Die kürzeste Route ist dabei allerdings nicht immer die Beste. Abhängig vom Verkehrsaufkommen kann ein längerer Weg deshalb mitunter besser als ein kürzerer sein. Einige Routingalgorithmen versuchen deshalb sich an variierende Verkehrsaufkommen anzupassen. Das momentan gebräuchlichste Protokoll dieser Schicht ist das verbindungslose IP-Protokoll.

Sicherungsschicht (Data Link Layer): Die Sicherungsschicht ist dafür zuständig bei der Kommunikation auftretende Fehler zu erkennen und falls möglich zu korrigieren. Dafür werden Bits in so genannte Frames gruppiert. Am Anfang und am Ende eines jeden Frames werden spezielle Bit-Muster

angefügt, um diesen zu markieren. Um die korrekte Übertragung der Frames sicherzustellen, hängt der Sender jedem Frame zusätzlich eine Prüfsumme an. Der Empfänger berechnet ebenfalls die Prüfsumme und vergleicht sie mit der Angehängten, um festzustellen, ob der Frame fehlerfrei übertragen wurde. Im Fehlerfall fordert der Empfänger den Sender auf den Frame erneut zu senden.

Bitübertragungsschicht (Physical Layer): Aufgabe der Bitübertragungsschicht ist die physikalische Übertragung roher Bits, also logischer Nullen und Einsen über ein Medium. Wichtige Fragestellungen die sich dabei ergeben sind unter anderem wie viel Volt für 0 und wie viel Volt für 1 verwendet werden, welche Dauer ein Bit hat und ob eine bidirektionale Übertragung möglich ist. Darüber hinaus sind die Art der physikalischen Verbindung, also beispielsweise die Form des Netzwerksteckers, sowie die Anzahl der verwendeten Pins und ihre Belegung interessant. Das Protokoll der Bitübertragungsschicht ist somit für die Standardisierung der elektrischen, mechanischen und Signalschnittstellen verantwortlich. Es stellt also sicher, dass ein gesendetes 0-Bit eines Rechners auf anderen Maschinen auch als 0-Bit und nicht etwa als 1-Bit empfangen wird.

3.4 Objektserialisierung zum Datenaustausch

Java und auch einige andere Programmiersprachen, zum Beispiel C++ erlauben es Objekte in ihrem aktuellen Zustand zu serialisieren, um sie dann in serialisierter Form zu speichern oder über das Netzwerk zu versenden. Später können diese serialisierten Objekte in der gleichen oder einer anderen VM wieder mit ihren ursprünglichen Werten hergestellt werden.
Serialisierbar sind grundsätzlich Objekte von Klassen, die das Marker-Interface java.io.Serializeable implementieren und deren Felder serialisierbar sind, sowie alle primitiven Datentypen. Ausnahmen bilden Objekte wie zum Beispiel java.io.FileInputStream, da es keinen Sinn macht einen Stream zu serialisieren.

Grundsätzlich existieren zwei Arten der Serialisierung. Zum einen gibt es die binäre Objektserialisierung, welche eine binäre Repräsentation des Objektes erzeugt und andererseits die Serialisierung Mittels XML.

3.5 Entwurfsmuster

In diesem Abschnitt sollen die drei wichtigsten, bei der Entwicklung der Anwendung verwendeten Entwurfsmuster näher erläutert werden. Dabei handelt es sich im Einzelnen um das Replicated-Worker-Pattern, das Command-Pattern und das Proxy-Pattern.

3.5.1 Das Replicated-Worker-Pattern

Das Replicated-Worker-Pattern[4], auch Master-Worker- oder Supervisor-Worker-Pattern genannt, ist ein einfaches aber mächtiges Entwurfsmuster. Das Replicated-Worker-Pattern umfasst in der Regel einen Master und n Worker. Der Master teilt ein Gesamtproblem in mehrere Teilprobleme auf, die dann von den Workern bearbeitet werden. Nach Bearbeitung eines Teilproblems, geben die Worker das Ergebnis der Berechnung an den Master, der diese dann zu einem Gesamtergebnis zusammenfügt, zurück. Abbildung 3.2 verdeutlicht die grundlegende Struktur des Replicated-Worker-Patterns.

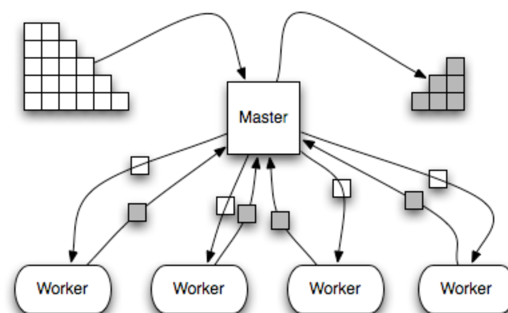

Abbildung 3.2 Das Replicated-Worker-Pattern

[4] vgl. [Freeman et al., 1999, S. 153-165]

Vor Einsatz des Replicated-Worker-Patterns sollte in jedem Fall der Computation/Communication-Ratio geprüft werden. Dieser Wert macht deutlich ob die parallele, verteilte Berechnung des Problems lohnenswert ist oder ob die Berechnung besser auf einem einzelnen Rechner durchgeführt werden sollte. Je größer der Rechenaufwand im Vergleich zum Kommunikationsaufwand ist, desto mehr Sinn macht der Einsatz des Replicated-Worker-Patterns. Normalerweise kann ein Problem umso schneller gelöst werden, je mehr Worker an der Lösung beteiligt sind. Es gibt aber auch einige Probleme, bei denen die Kommunikation einen großen bis sehr großen Anteil am Gesamtaufwand der Berechnung einnimmt. Diese Probleme können auf einem Einzelrechner meist schneller berechnet werden. Eine typische Anwendung, bei der die Kommunikation einen sehr großen Anteil einnimmt, ist die Berechnung eines Mandelbrots.

3.5.2 Das Command-Pattern

Mit Hilfe des Command-Patterns[5] kann einer Applikation eine Operation übergeben werden, ohne das diese über den Inhalt der Operation bescheid weiß. In Verbindung mit dem im vorangegangen Abschnitt beschriebenen Replicated-Worker-Pattern bedeutet das, dass die Worker-Anwendung vom Master ein Objekt zur Ausführung bekommt, ohne zu Wissen um welches spezielle Objekt es sich handelt. Die Tasks des Masters implementieren dazu das Command-Interface aus Abbildung 3.3.

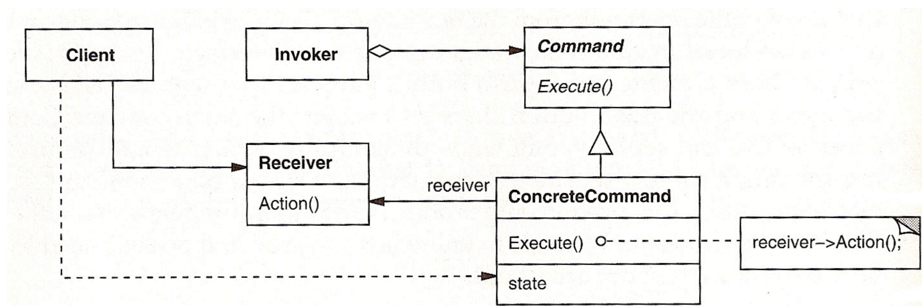

Abbildung 3.3 Das Command-Pattern [Gamma et al., 1994, S.236]

[5] vgl. [Gamma et al., 1994, S.233-S.242]

Wenn die Worker eines der Task-Objekte erhalten, führen sie nur die Execute-Methode dieses Objektes aus. Somit kann die Implementierung zusammen mit der Aufgabe übertragen werden. Der wesentliche Vorteil dabei ist, dass die Worker durch das Command-Pattern unabhängig von der Aufgabe werden. Das heißt sie müssen bei Implementierung neuer Jobs für das verteilte System nicht mehr angepasst werden, was wiederum eine erhebliche Einsparung an Administrationsaufwand bedeutet.

3.5.3 Das Proxy-Pattern

Beim Proxy Pattern[6] wird ein Objekt durch ein so genanntes Stellvertreterobjekt repräsentiert. Das Stellvertreterobjekt nimmt dann Anfragen an das reale Objekt entgegen und leitet sie gegebenenfalls an dieses weiter. Ein Proxy-Objekt kann beispielsweise aus Effizienzgründen eingesetzt werden. So kann zum Beispiel ein Bild in einem Dokument zur Verbesserung der Ladezeit durch ein Proxy-Objekt repräsentiert werden. Das Proxy-Objekt lädt dann wenn nötig das reale Bildobjekt. Abbildung 3.4 verdeutlicht den Aufbau des Proxy-Patterns.

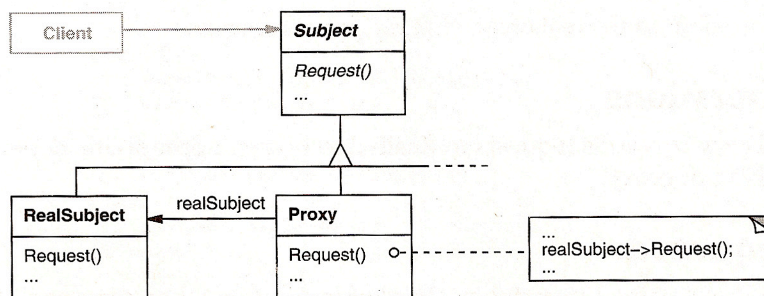

Abbildung 3.4 Das Proxy-Pattern [Gamma et al., 1994, S.209]

Außerdem ist das Proxy-Pattern in Netzwerk-Technologien wie RMI oder Jini zu finden. Bei beiden Technologien gewähren Proxy-Objekte den Zugriff auf entfernte Objekte. Die Proxy-Objekte nehmen dabei an das entfernte Objekt gerichtete Aufrufe entgegen und leiten diese an es weiter

[6] vgl. [Gamma et al., 1994, S.207-S.217]

3.6 Remote Method Invocation

Verteilte Systeme erfordern, dass Prozesse auf verschiedenen Rechnern im Netzwerk miteinander kommunizieren. Zum Zweck der grundlegenden Kommunikation stellt Java deshalb die Möglichkeit der Kommunikation über Sockets zur Verfügung. Die Kommunikation über Sockets ist flexibel und für die allgemeine Kommunikation im Netzwerk ausreichend. Jedoch erfordert eine solche Kommunikation die Entwicklung von Kommunikationsprotokollen auf Applikationsebene, was mitunter aufwendig und fehleranfällig sein kann.

Im Jahre 1984 stellten Birrel und Nelson in [Birrel & Nelson, 1984] ein Modell vor, das entfernte Methoden wie Lokale aussehen lässt. Die Verbindung zwischen Client und Server wird dabei durch so genannte Stellvertreter-objekte (engl. Proxies) hergestellt. Aufbauend auf diesem Modell entstanden verschiedene Implementierungen. So war zum Beispiel Suns RPC eine der ersten populären Implementierungen für entfernte Aufrufe in der prozeduralen Welt. RMI, dessen Funktionsweise im Anschluss näher erläutert wird, ist die Implementierung des Modells in der Programmiersprache Java. Weitere populäre Implementierungen sind DCOM[7] aus dem .NET Remoting API von Microsoft und CORBA.

Da es die Grundlage der in dieser Arbeit verwendeten Jini-Technologie darstellt, möchte ich im Folgenden näher auf RMI eingehen. Da sich RPC nicht gut auf verteilte Objektsysteme anwenden lässt, werden andere an die Semantik verteilter Objektsysteme angepasste Technologien benötigt. Das RMI System[8] der Java-Plattform stellt eine objektorientierte Variante von Suns RPC dar, mit der die Arbeit auf hoher Abstraktionsebene möglich ist. Die Kommunikation zwischen Client und Server läuft dabei wie in Abbildung 3.5 ab.

[7] später ActiveX

[8] vgl. [RMI Specification, 2006]

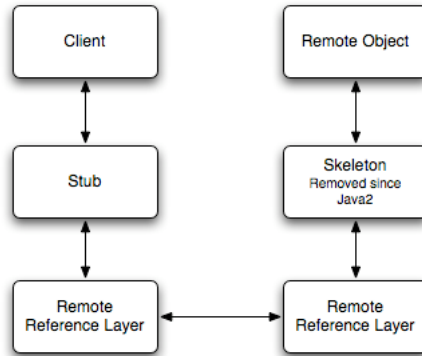

Abbildung 3.5 Kommunikation bei RMI

3.6.1 Stub und Skeleton

Um den Aufruf entfernter Methoden so transparent wie möglich zu gestalten, verwendet RMI das Proxy-Pattern, welches bereits in Abschnitt 3.5.3 beschrieben wurde. Beim Proxy-Pattern wird ein Objekt eines Kontextes in einem anderen Kontext durch ein so genanntes Proxy-Objekt repräsentiert. Das Proxy-Objekt weiß wie mit dem reellen Objekt kommuniziert werden kann und leitet Anfragen an dieses weiter.

Bei RMI ist die Stub-Klasse der Proxy und die entfernte Implementierung des Remote-Interface stellt das reale Objekt dar. Der Client richtet seine Aufrufe somit an die Stub-Klasse, welche mit dem auf dem Server befindlichen Skeleton kommuniziert. Der Skeleton ist eine auf dem Server automatisch generierte Klasse, die mit der Stub-Klasse auf dem Client kommunizieren kann. Die Skeletonklasse ihrerseits reicht die Aufrufe zur Implementierung weiter und liefert die Rückgabewerte an die Stub-Klasse zurück. Skeletons sind allerdings nur bis Java 1.1 nötig, seit Java 2 wird die Verbindung zum entfernten Service Objekt mit Hilfe des Java-Reflection-API realisiert.

3.6.2 Registry

Die RMI-Registry dient dem Auffinden von Serviceimplementierungen. Server registrieren ihre Implementierungen in der Registry und machen so das Auffinden dieser für Clients möglich. Eine Übersicht der RMI Interfaces und Klassen liefert die [RMI Specification, 2006] in Abschnitt 2.4.

3.6.3 Übertragung

Zur Übertragung verwendet RMI das RMI-Wire-Protocol. Dieses bietet die Möglichkeit RMI über Sockets oder RMI über HTTP kommunizieren zu lassen. Die Möglichkeit der Kommunikation über HTTP ist besonders für Umgebungen die durch Firewalls geschützt sind wichtig, da dort mitunter die benötigten Ports geschlossen sind. Weiterhin bietet RMI die Möglichkeit der Erweiterung über eine eigene Transportschicht. So lassen sich zum Beispiel RMI über UDP oder auch gesicherte Verbindungen mit SSL durch eine eigene Implementierung realisieren.

3.6.4 Serialisierung

Der tatsächliche Unterschied zwischen lokalen und entfernten Objekten besteht im Fehlen eines gemeinsamen Kontextes. Das bedeutet, dass die involvierten Rechner unabhängig voneinander, also mit jeweils eigenen Speicherbereichen existieren. Folglich müssen Daten beziehungsweise Objekte erst übertragen werden, wobei aufgrund des Fehlens eines gemeinsamen Speicherbereichs die Übergabe einer Referenz nicht ausreichend ist. Wie also überträgt man Objekte? Das Stichwort hier lautet Serialisierung. Zu übertragende Objekte werden, wie in Abschnitt 3.4 beschrieben, serialisiert, in serialisierter Form übertragen und beim Empfänger deserialisiert.

Ein weiterer wichtiger Punkt ist das Austauschformat. Hier gibt es ein symmetrisches sowie ein asymmetrisches Verfahren. Beim symmetrischen Verfahren wird ein neutrales Austauschformat gewählt das alle Teilnehmer verstehen. Beim asymmetrischen Verfahren überträgt jeder Client seine Daten im eigenen Format zum Server und dieser versucht die Daten dann mit verschiedenen Konvertierungsfunktionen zu erkennen. Da RMI auf der Java-Plattform aufbaut, braucht man sich darum allerdings nicht zu kümmern, denn ein Vorteil der Java-Plattform ist, dass Daten unterschiedlichster Plattformen immer gleich konvertiert werden. Folglich verwendet Java ein symmetrisches Verfahren.

3.7 Jeri

Jeri[9] wurde mit der Version 2.0 von Jini eingeführt und stellt eine Erweiterung von RMI dar. Ziel des Jeri Protokolls ist es, in erster Linie ein Übertragungsformat für eine binäre Anfrage/Antwort-Kommunikation zu schaffen. Das Protokoll hat also die Aufgabe entfernte Jini ERI Methodenaufrufe über eine TCP-Verbindung oder ähnliche verbindungs-orientierte Protokolle zu übertragen. Mit Jeri wurde somit das API vom Kommunikationsprotokoll abstrahiert, so dass nun auch andere Protokolle wie beispielsweise SOAP verwendet werden können.

3.8 Jini

1994 stellte Peter Deutsch eine Liste mit 7 Fehlern auf, die bei der Entwicklung verteilter Systeme mit hoher Wahrscheinlichkeit zu Beginn begangen werden. Diese 1997 durch John Gosling um eine zusätzliche Annahme erweiterte Liste, ist heute unter dem Namen „The eight fallacies of distributed computing" also „Die 8 Irrglauben des verteilten Rechnens" bekannt. Die Liste beinhaltet die Annahmen aus Tabelle 3.2.

Acht Irrtümer über verteilte Systeme
1. The network is reliable. / Das Netzwerk fällt nie aus.
2. Latency is zero. / Es gibt keine Latenz.
3. Bandwidth is infinite. / Die verfügbare Bandbreite ist unendlich groß.
4. The network is secure. / Das Netzwerk ist sicher.
5. Topology doesn't change. / Die Netzwerkarchitektur ändert sich nicht.
6. There is one administrator. / Es gibt einen einzigen Administrator.

[9] vgl. [Sun, 2007b]

7. Transport cost is zero. / Es fallen keine Transportkosten an.
8. The network is homogeneous. / Das Netzwerk ist homogen.

Tabelle 3.2 The eight fallacies of distributed computing [Deutsch, 2007]

Betrachtet man diese Annahmen, so wird man feststellen, dass bei der verteilten Programmierung, im Vergleich zur Anwendungsentwicklung, mit nur einem einzelnen Rechner zusätzliche Fehlerquellen entstehen.
Jini ist eine Netzwerktechnologie, die vor dem Hintergrund dieser acht Irrglauben entwickelt wurde. Jini versucht nicht diese Probleme zu ignorieren, sondern versucht Lösungen für diese Probleme zu finden. So akzeptierten die Entwickler der Jini-Technologie die Tatsache, dass Netzwerke fehlbar sind. Es können Teilnehmer aufgrund eines Hardwaredefektes oder aufgrund eines Absturzes der auf ihnen laufenden Software ausfallen oder es können Teilnehmer ausfallen weil einfach nur die Netzwerkverbindung getrennt[10] wurde. Jini trägt dieser Tatsache Rechnung, in dem es Mechanismen zur Verwaltung solcher dynamischer Umgebungen bereitstellt. Das folgende Zitat aus [JINI.ORG, 2007] liefert eine treffende Beschreibung der Jini-Technologie. *„Jini technology can be used to build adaptive network systems that are scalable, evolvable and flexible as typically required in dynamic computing environments“*
Weiterhin trägt Jini der Tatsache Rechnung, dass Netzwerke aus Rechnern mit unterschiedlichen Systemen bestehen können. Da Jini auf der Java-Plattform basiert, können somit alle von Java unterstützten Systeme in eine Jini-Community integriert werden.

3.8.1 Jini Lookup

Durch den Jini-Lookup-Service wird eine spontane Vernetzung auf Anwendungsebene ermöglicht. So können im Netzwerk veröffentlichte Jini Services automatisch von anderen Mitgliedern der Jini-Community gefunden werden.

[10] Entweder zur Systemwartung oder durch einen Defekt

Die Veröffentlichung und Verwendung der Services läuft dabei folgendermaßen ab. Um eine Jini-Community aufzubauen, müssen zuerst ein oder mehrere Lookup-Services im Netzwerk laufen. Jini liefert den Lookup-Service reggie als Teil der Standard Jini Distribution mit. Ist dieser gestartet kann er von Clients und Service Providern mittels Unicast- oder Multicast-Protokollen gefunden werden. Das so genannte Unicast-Discovery wird dabei vor allem zur Lokalisierung von Lookup-Services außerhalb des lokalen Netzes verwendet. Dies setzt aber voraus, dass die Adresse des gewünschten Lookup-Services bereits bekannt ist. Ist die Adresse des Lookup-Service nicht bekannt so kann außerdem noch das so genannte Multicast-Discovery eingesetzt werden. Dabei wird via Broadcast nach Lookup-Services gesucht. Da die meisten Router Multicastpakete aufgrund ihrer hohen Netzwerklast blockieren, funktioniert diese Methode allerdings meist nur in LANs [Newmarch 2006]. Dort allerdings ermöglicht sie, wie oben schon erwähnt, eine spontane Vernetzung auf Applikationsebene, das heißt es ist unerheblich auf welchem Rechner im Netz der Lookup-Service läuft.

3.8.2 Services registrieren

Nach Auffinden eines Lookup-Service bekommt man von diesem ein ServiceRegistrar-Objekt, welches als Proxy für den Lookup-Service fungiert. Mit der Register-Methode dieses Objektes können nun neue Services, durch Übergabe eines ServiceItems, am Lookup-Service registriert werden. Abbildung 3.6 verdeutlicht die den Registriervorgang.

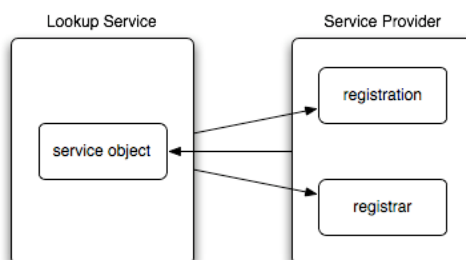

Abbildung 3.6 Jini Service Registration

3.8.3 Discovery - Services finden

Ist ein Service am Lookup-Service registriert, kann er von anderen Teilnehmern der Jini-Community gefunden werden. Der Aufruf der Lookup-Methode des ServiceRegistrars liefert ein Objekt zurück, welches als Proxy für den angeforderten Service dient. Um dem Lookup-Service mitzuteilen nach welchem Service gesucht wird, übergibt man der Lookup-Methode ein Objekt der Klasse net.jini.core.lookup.ServiceTemplate.

3.8.4 Leasing

Wie bereits oben erwähnt ignoriert Jini nicht die Tatsache, dass beim verteilten Rechnen in einem Netzwerk diverse Probleme auftreten können, sondern versucht Mechanismen zum Umgang mit einem auftretenden Problemfall bereitzustellen. Das Leasing-Konzept ist ein solcher Mechanismus. Hat ein Service-Provider einen Service am Lookup-Service registriert bekommt der Service ein Lease zugeteilt. Dieses Lease muss vom Service-Provider regelmäßig erneuert werden, um dem Lookup-Service mitzuteilen, dass der Service noch verfügbar ist. Wird das Lease nicht erneuert, wird der registrierte Service aus dem Register des Lookup-Service gelöscht, da der Lookup-Service in diesem Fall davon ausgehen kann, dass der Service entweder nicht mehr existiert oder nicht mehr erreichbar ist. Leasing ist also ein Mechanismus, um auf partielle Ausfälle eines Netzwerks zu reagieren.

3.9 Linda und Tuple-Spaces

Tuple-Spaces wurden erstmals 1982 von Dr. David Gelernter von der Yale University im Zusammenhang mit der verteilten Programmiersprache Linda implementiert. Dr. David Gelernter und Nicolas Carriero entwickelten die Programmiersprache Linda, eine Sprache zur Programmierung verteilter Systeme, auf der theoretischen Grundlage der Tuple-Spaces. Gelernter definierte damit ein grundlegendes Konzept der Interprozesskommunikation in verteilten Systemen. Das Konzept beruht auf einem assoziativen über das Netzwerk erreichbaren Speicher, auf den die vier Methoden in, rd, out und

eval[11] angewendet werden können. Damit können so genannte Tuple im Speicher abgelegt und aus dem Speicher geholt werden.
Heutige Implementierungen des Tuple-Space-Konzeptes basieren im Wesentlichen immer noch auf den von Gelernter in [Gelernter, 1985] definierten Prinzipien.

3.10 JavaSpaces

Die JavaSpaces Spezifikation[12] wurde Ende der 90er Jahre basierend auf der Jini Technologie von der Firma Sun Microsystems entwickelt. Eine JavaSpace ist demnach ein Jini-Service, welcher einen Objektspeicher darstellt. Die Begriffe JavaSpace und Space werden im Anschluss synonym verwendet.
JavaSpaces realisieren das Konzept der Tuple-Spaces in der Programmiersprache Java. Ein JavaSpace ist demnach ein gemeinsam genutzter Datenraum, der Objekte in serialisierter Form speichert. Was genau charakterisiert aber einen JavaSpace? Um das zu klären, werde ich im Folgenden die Schlüsseleigenschaften eines JavaSpace laut [Freeman & Hupfer, 1999] näher erläutern.

JavaSpaces sind geteilt: JavaSpaces sind über das Netzwerk zugänglich. Verschiedene Prozesse können gleichzeitig mit dem JavaSpace interagieren. Der JavaSpace kümmert sich dabei selbst um die Einzelheiten des gleichzeitigen Zugriffs.

JavaSpaces sind persistent: JavaSpaces bieten eine zuverlässige Möglichkeit Objekte zu speichern. Einmal im Space gespeicherte Objekte verbleiben dort, bis sie entnommen werden oder ihr Lease abläuft. Persistenz ist außerdem die wesentliche Charakteristik zur Unterstützung lose gekoppelter Protokolle, was wiederum ein wesentlicher Vorteil eines JavaSpace ist. Auf die Vorteile von JavaSpaces wird im nächsten Abschnitt näher eingegangen.

[11] schreiben, lesen, entnehmen und vergleichen
[12] vgl. [Sun, 2007]

JavaSpaces sind assoziativ: Objekte können in einem JavaSpace durch assoziative Suche gefunden werden. Es ist dabei unwichtig wer das Objekt erstellt hat oder wo es gespeichert ist. Es ist lediglich nötig ein Template[13] zu erstellen, um Objekte im JavaSpace zu finden. Die Felder des Template können dabei entweder auf bestimmte Werte gesetzt werden oder sie bleiben null, was als Wildcard fungiert.

JavaSpaces sind transaktionssicher: JavaSpaces nutzen den Jini-Transaction-Service um sicherzustellen das Operationen atomar sind. Das heißt entweder wird die gesamte Operation ausgeführt oder nichts wird ausgeführt. Transaktionen können dabei sowohl für einzelne Operationen auf einem einzelnen JavaSpace, als auch für mehrere Operationen auf einem oder mehreren JavaSpaces eingesetzt werden. Damit stellen sie ein wichtiges Mittel dar, um mit partiellem Fehlverhalten im Netzwerk umzugehen.

JavaSpaces erlauben den Austausch von ausführbaren Inhalten: In einem JavaSpace stellen die enthaltenen Objekte passive Daten dar. Liest oder nimmt ein Client aber Daten aus dem JavaSpace wird eine lokale Kopie des Objektes erstellt. Mit diesem Objekt kann dann wie mit jedem anderen lokalen Objekt verfahren werden. Es können also zum Beispiel dessen Methoden aufgerufen werden. Bei der im Rahmen dieser Arbeit entwickelten Software wird diese Eigenschaft genutzt um die Worker generisch zu gestalten. Mehr dazu folgt in Abschnitt 4.6.

3.10.1 Die Vorteile von JavaSpaces

Laut [Freeman et al., 1999, S. 16] hat die JavaSpaces Technologie die folgenden Vorteile.

[13] Ein Template ist ein Objekt der gleichen oder einer Superklasse des zu suchenden Objektes bei dem einige Attribute auf bestimmte Werte gesetzt wurden.

Die Technologie ist einfach: Die Interaktion mit einem JavaSpace baut auf nur wenigen Methoden auf und erfordert deshalb nicht das lernen komplexer APIs.

Die Technologie ist ausdrucksstark: Trotz dieser wenigen Methoden sind JavaSpaces eine mächtige Technologie, da mit Hilfe von JavaSpaces komplexe verteilte Applikationen mit relativ wenig Code erstellt werden können.

JavaSpaces unterstützen lose gekoppelte Protokolle: Durch die Entkopplung von Sender und Empfänger unterstützen JavaSpaces einfache, flexible und verlässliche Protokolle. So kann ein Empfänger Objekte eines Senders aus dem Space entnehmen lange nachdem dieser beendet wurde.

JavaSpaces erleichtern die Entwicklung von Client/Server-Systemen: Bei der Entwicklung eines Servers müssen normalerweise Funktionen wie gleichzeitiger Zugriff mehrerer Clients, persistente Speicherung und Transaktionen jedes Mal neu entwickelt werden. JavaSpaces erleichtern die Entwicklung dahingehend, dass sie alle diese Funktionen bereits bereitstellen.

3.10.2 Entries

Wie oben bereits erwähnt, können in einem JavaSpace Objekte in serialisierter Form gespeichert werden. Diese Objekte nennt man Entries. Prozesse können somit durch den Austausch von Entries über den JavaSpace kommunizieren.
Ein Entry ist dabei ein Objekt, das das Entry-Interface aus dem net.jini.core.entry Package implementiert. Das Entry-Interface ist ein so genanntes Marker-Interface[14] und wie folgt definiert.

[14] Ein Marker-Interface ist ein Interface ohne Konstanten oder Methoden und erfordert somit, außer dem `implements Entry`, in der implementierenden Klasse keine weitere Implementierung

```
package net.jini.core.entry;

public interface Entry extends java.io.Serializable {
}
```

Außer der Implementierung dieses Interface müssen Entries aber noch einige weitere Bedingungen erfüllen, um über einen JavaSpace austauschbar zu sein. Wie aus dem Entry-Interface und aus der Tatsache das Entries im Space in serialisierter Form gespeichert werden schon zu sehen ist, müssen Entries serialisierbar sein. So können beispielsweise Threads, OutputStreams oder Sockets nicht als Entries verwendet werden, da sie nicht serialisierbar sind. Weiterhin brauchen Entries den so genannten public no-arg constructor, also einen öffentlichen Konstruktor ohne Argumente. Außerdem müssen alle Attribute eines Entries als public deklariert sein. Der Grund dafür liegt darin, dass ein JavaSpace assoziativ nach Entries sucht. Wären die Attribute des Entries private, könnte also kein Vergleich zwischen Entry und Template stattfinden. Als Letztes müssen alle Attribute des Entries Objekttypen sein. Primitive Datentypen wie int, boolean, float, double müssen durch ihre zugehörigen Wrapper-Objekte, Integer, Boolean, Float, Double, ersetzt werden.

3.10.3 Entries schreiben, lesen und entnehmen

Die Arbeit mit einem JavaSpace basiert im Wesentlichen auf drei elementaren Operationen[15], die die Interaktion zwischen Applikation und JavaSpace, mit Hilfe von Entries, ermöglichen. Diese Methoden sind im JavaSpace-Interface aus dem net.jini.space Package definiert. Abbildung 3.7 zeigt dieses Interface.

---

[15] vgl. [Sun, 2007]

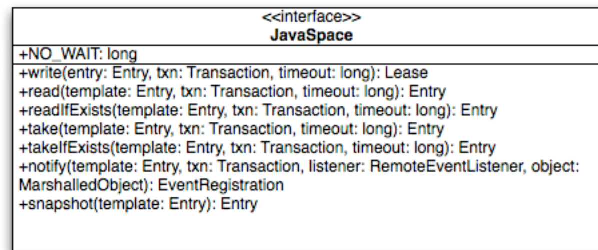

Abbildung 3.7 Das JavaSpace-Interface

- *write*: Schreibt eine Kopie des übergebenen Entries in einen JavaSpace
- *read*: Gibt einen dem übergebenen Template entsprechenden Entry zurück
- *take*: Gibt wie read einen dem Template entsprechenden Entry zurück. Der Unterschied zu read ist, dass dieser aus dem Space entfernt wird.

Mittels dieser beiden lesenden Zugriffe und des schreibenden Zugriffs können sich Prozesse im Netzwerk gegenseitig Objekte, in einem virtuellen gemeinsamen Speicher, dem JavaSpace, zugänglich machen.

Wie oben schon erwähnt können Objekte mit Hilfe der Write-Methode in Form von Entries in den Space geschrieben werden.

```
Lease write(Entry entry, Transaction txn, long lease)
      throws TransactionException, RemoteException;
```

Write hat drei Übergabeparameter. Entry ist der in den JavaSpace zu schreibende Entry. Mit txn kann, wenn nötig, ein Transaction-Objekt übergeben werden. Bei Schreibvorgängen ohne Transaktion, muss hier NULL übergeben werden. Der Parameter lease ist die angeforderte Lease-Dauer, also die angeforderte Gültigkeitsdauer, die der geschriebene Entry im Space haben soll. Auf Leasing möchte ich hier jedoch nicht weiter eingehen, da im Anschluss ein gesonderter Abschnitt über Leasing folgt.
Read und take verhalten sich anlog und sind im JavaSpace-Interface wie folgt definiert.

```
Entry read(Entry tmpl, Transaction txn, long timeout)
        throws UnusableEntryException, TransactionException,
                InterruptedException, RemoteException;

Entry take(Entry tmpl, Transaction txn, long timeout)
        throws UnusableEntryException, TransactionException,
                InterruptedException, RemoteException;
```

Der einzige Unterschied ist das Entries nach einem read im Space verbleiben und bei take aus dem Space entfernt werden. Da beide Methoden assoziativ nach Entries suchen, erfordern sie als ersten Parameter ein Template[16] des Objektes nach dem gesucht werden soll. Der Suchvorgang läuft dabei wie folgt ab. Im JavaSpace werden die Attribute des Template mit denen der enthaltenen Objekte verglichen. Ein Template gleicht dabei einem Objekt im Space, wenn die Werte ihrer Attribute übereinstimmen. NULL gilt dabei als Wildcard. Zusätzlich gleichen Templates einem Objekt, wenn sie Objekte einer Superklasse des jeweiligen Objektes sind. Die anderen Parameter der beiden Methoden sind eine Transaktion und ein Timeout, der festlegt, wie lange die Funktion wartet, ob ein passender Entry im Space auftaucht.
In vielen Fällen wird ein und dasselbe Template mehrfach verwendet. Dabei entsteht ein unnötiger Rechenaufwand in der Hinsicht, dass das Template bei jeder Verwendung erneut serialisiert werden muss. Um das zu vermeiden, stellt das JavaSpace-Interface die Methode snapshot bereit. Snapshot liefert eine so genannte Snapshot-Version des Templates. Die Snapshot-Version des Templates wird in serialisierter Form im Space gespeichert, so dass sie bei wiederholter Verwendung nicht erneut serialisiert werden muss.

3.10.4 Leasing

Leasing ist ein Konzept, um mit partiellen Ausfällen im Netzwerk umzugehen. Schreibt man einen Entry in einen JavaSpace, so bekommt dieser ein so genanntes Lease zugeteilt. Dieses Lease gibt an, wie lange der Entry gültig

[16] Ein Template ein Objekt gleichen Typs wie das gesuchte Objekt

ist, also wie lange er im Space verbleiben darf. Nach Ablauf dieses Leases wird der Entry automatisch aus dem Space entfernt. Soll der Entry weiter im Space verbleiben, muss das Lease vor Ablauf verlängert werden. Der Rückgabewert der Write-Methode aus dem letzten Abschnitt ist ein solches Lease-Objekt. Der dritte Parameter der Write-Methode gibt die Dauer des mit dem Aufruf angeforderten Leases an. Der Rückgabewert ist dann das vom Space tatsächlich vergebene Lease. Die Dauer dieses Leases muss jedoch nicht zwangsläufig mit der angeforderten Dauer übereinstimmen. Je nach Konfiguration kann der JavaSpace auch kürzere Leases vergeben. Über die Renew-Methode kann ein Lease verlängert werden. Mit der Cancel-Methode des Lease-Objektes kann der Ablauf eines Leases vorzeitig erzwungen werden.

3.10.5 Transaktionen

Transaktionen sind eine weitere Methode zum Umgang mit partiellen Ausfällen im Netzwerk. Transaktionen können mehrere Operationen in Verbindung mit JavaSpace bündeln, sodass entweder alle oder keine der Operationen ausgeführt wird. Transaktionen sind Teil von Jini und können vom Jini-Transaction-Service mittels des folgenden Aufrufs angefordert werden.

```
try
{
	long leaseTime = 1000 * 10 * 60; // 10 minutes
	Transaction.Created created = TransactionFactory.create
	(txnManager.getService (), leaseTime);
	return created.transaction;
}
catch (RemoteException e)
{
	LOG.error (e, "Network error");
	return null;
}
catch (LeaseDeniedException e)
{
```

```
        LOG.error (e, "Requested lease has been denied");
        return null;
}
```

3.11 Eclipse Rich Client Platform

Die Eclipse Rich Client Platform, welche im Folgenden kurz als Eclipse RCP bezeichnet wird, ist zwar nicht Thema der Arbeit, es soll hier aber kurz darauf eingegangen werden da sowohl die Client- als auch die Worker-Anwendung auf der Eclipse RCP aufbauen soll hier kurz darauf eingegangen werden.
Die Eclipse RCP bildet nicht nur die Grundlage der Eclipse IDE, sondern bietet seit Version 3.0 auch die Möglichkeit generische Anwendungen auf Basis des Eclipse-Frameworks zu erstellen.
Die Eclipse RCP beinhaltet unter anderem ein API zur Erstellung grafischer Benutzeroberflächen das mit dem SWT auf den nativen GUI-Elementen der verwendeten Plattform aufsetzt. Zusätzlich zu den SWT-eigenen GUI-Elementen kann außerdem auf erweiterte GUI-Elemente wie zum Beispiel diverse Editoren oder den Progress View der Eclipse IDE zurückgegriffen werden. Die Entwicklung eigener grafischer Benutzeroberflächen wird dadurch erheblich vereinfacht.
Da das Eclipse-Framework auf dem OSGi-Framework Equinox basiert, wird außerdem die modulare Entwicklung von Applikationen auf Basis von Plugins unterstützt. Das bietet unter anderem den Vorteil das Plugins in anderen Anwendungen wieder verwendet werden können.
Ein weiterer interessanter Teil des Eclipse-Frameworks ist das Eclipse-Jobs-API, welches sowohl in der Worker- als auch in der Master-Applikation Verwendung findet. Mit diesem lassen sich lang laufende Tasks kontrollieren und überwachen.
Der letzte hier erwähnte Aspekt des Eclipse-Frameworks soll die Unterstützung von so genannten Extensions sein. So bietet das Framework die Möglichkeit einer losen Kopplung von Plugins durch die Möglichkeit Extension Points zu definieren. Über diese Extension Points können andere Plugins angekoppelt werden ohne die Implementierung der Anwendung zu ändern.

Das Eclipse-Framework bietet natürlich noch weitaus mehr Funktionen. Auf diese soll hier aber nicht weiter eingegangen werden, da das Eclipse-Framework nicht den Schwerpunkt der Arbeit darstellt.

3.12 Verwandte Projekte und Abgrenzung

Das wohl bekannteste verwandte Projekt der im Rahmen dieser Arbeit entwickelten verteilten Applikation ist das BOINC[17] Projekt der kalifornischen Universität in Berkeley.
Weitere verwandte Projekte sind unter anderem das ComputeFarm[18] und das ComputeCycles[19] Projekt aus der Javawelt. Beide Projekte basieren ebenfalls auf der Verwendung eines JavaSpace zum Datenaustausch. Der Unterschied der hier entwickelten Anwendung zu diesen beiden Projekten besteht vor allem in der Integration in die OSGi Implementierung Eclipse Equinox, welche, wie in Abschnitt 3.11 beschrieben, die Basis der Eclipse Rich Client Platform bildet. Einige Aspekte dieser Integration werden in Kapitel 4 noch näher beschrieben. Ein weiterer Unterschied ist außerdem die Integration lokaler Ressourcen in den Worker.

[17] früher SETI@HOME

[18] vgl. [ComputeFarm, 2007]

[19] vgl. [ComputeCycles, 2007]

4 Design und Realisierung

In diesem Kapitel soll nun näher auf die Implementierung und Konfiguration der Anwendung eingegangen werden. Dazu wird zuerst ein Überblick über das entwickelte System gegeben. Im Anschluss daran werden die Systemkonfiguration aus Sicht der Hardware und der Konfiguration betrachtet und die einzelnen Bestandteile der Anwendung erläutert. Außerdem beinhaltet dieses Kapitel eine Programmieranweisung zur Entwicklung und Integration weiterer Probleme in Form von Jobs.

4.1 Gesamtkonzept

Das Gesamtkonzept des verteilten Systems beruht auf dem in Abschnitt 3.5.2 beschriebenen Command-Pattern in Verbindung mit dem in Abschnitt 3.5.1 beschriebenen Replicated-Worker-Pattern. Abbildung 4.1 stellt den groben Ablauf der Anwendung mit den folgenden sieben Schritten dar.

1. Der Master schreibt n Tasks des Jobs in den Space.
2. Der Worker entnimmt einen Task aus dem Space.
3. Der Worker berechnet den entnommenen Task.
4. Der Worker schreibt ein Result in den Space.
5. Die Worker bearbeiten alle weiteren im Space verfügbaren Tasks.
6. Der Master sammelt im Space auftauchende Results auf und gibt sie an den Job zurück.
7. Sind alle Results beim Job angekommen kann dieser falls nötig abschließende Berechnungen ausführen.

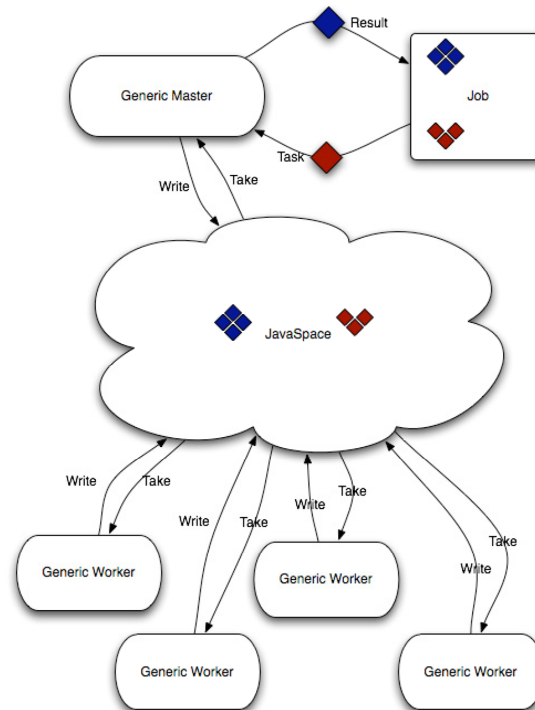

Abbildung 4.1 Das Gesamtkonzept der Anwendung

Die Jobs werden über einen dafür definierten Extension-Point an den Master angekoppelt. Durch diesen Mechanismus wird der Master unabhängig von den Jobs. Die Jobs können so einfach hinzugefügt und wieder entfernt werden ohne die Implementierung des Masters zu ändern. Der Master prüft bei jedem Start welche Jobs verfügbar sind und stellt diese dann zur Bearbeitung zur Verfügung. Im Fall dieser Implementierung bedeutet dies, dass die verfügbaren Jobs auf der Benutzeroberfläche angezeigt werden und nach Auswahl von dort gestartet werden können. Durch den modularen Aufbau der Eclipse Plattform könnten Jobs aber auch über eine nicht grafische Oberfläche gestartet werden. In diesem Fall müsste lediglich das Start-Plugin durch ein einfaches Equinox-Bundle oder ein RCP-Plugin ohne GUI ersetzt werden. Der zu startende Job könnte dann zum Beispiel über einen beim Start der Anwendung übergebenen Parameter ausgewählt werden.

4.2 Systemkonfiguration

Grundlegend lässt sich das System in die drei wesentlichen Bereiche Master, Worker und die Middleware einteilen. Diese sollen im Folgenden aus Sicht der Hardware und Konfiguration näher erläutert werden.

4.2.1 Master

Der Master ist der Rechner der die Erstellung und Verteilung der Tasks, sowie das aufsammeln der Results durchführt und überwacht. Prinzipiell ist der Master auf jedem Rechner mit Zugang zum JavaSpace lauffähig. Einzige Einschränkung sind hier die Anforderungen der Jobs an den Rechner. So kann zum Beispiel für einige Jobs eine lokale Datenbank für die Erstellung der Tasks sowie für die Verarbeitung der Results nötig sein.

4.2.2 Worker

Gleiches gilt für die Worker welche ebenfalls prinzipiell auf jedem Rechner lauffähig sind. Spezifische Ressourcen können hier ebenfalls eine lokale Datenbank sowie die bei Analytic Company für statistische Berechnungen verwendete Scripting-Engine RServ sein. Im produktiven Einsatz soll die Worker-Anwendung auf den meisten Rechnern der Firma eingesetzt werden und bei freien Ressourcen Tasks aus dem JavaSpace berechnen. Vor allem die Entwicklerrechner der Firma sind dafür besonders geeignet da diese bis jetzt über Nacht und in Pausenzeiten ungenutzt sind.

4.2.3 Middleware

Den dritten und letzten Bereich bildet die Middleware bestehend aus dem Server mit dem JavaSpace und dem Webserver mit der Codebase.
Der zur Entwicklung dieser Anwendung sowie zum Testen verwendete JavaSpace war ein Blitz JavaSpace in Version 1.29[20] basierend auf einer Jini 2.1[21] Installation. Grundsätzlich ist aber auch die Verwendung jedes anderen

[20] vgl. [Creswell, 2007]

[21] vgl. [JINI.ORG, 2007]

der JavaSpaces Spezifikation[22] entsprechenden JavaSpace möglich. Zum Fehlerfreien Ablauf des Systems sind außerdem zwei Änderungen der Standardkonfiguration des Blitz JavaSpace nötig. Bei anderen JavaSpace Implementierungen wie zum Beispiel GigaSpaces[23] oder Outrigger sind eventuell weitere Änderungen der Konfiguration nötig. In der Konfigurationsdatei des JavaSpace mit dem Namen blitz.config muss der Parameter leaseReapInterval von 0 auf 10000 gesetzt werden. Damit wird die aktive Bereinigung des JavaSpace eingeschaltet und auf ein Zeitintervall von 10000 ms, also 10s, festgelegt. Das heißt Entries mit abgelaufenem Lease werden alle 10000 ms automatisch aus dem Space entfernt um Speicherplatz zu sparen. Außerdem muss beim Start des JavaSpace mit Hilfe des folgenden Parameters ein erweiterter Protocol-Handler angegeben werden.

```
–Djava.protocol.handler.pkgs=org.ac.net.protocols
```

Dieser ist nötig damit der Space Zugriff auf die geschützte Codebase bekommt. Ohne diesen Protocol-Handler wäre das nicht möglich da Java standardmäßig keine URLs der Form http://login:password@... verarbeiten kann. Die Jar-Datei (custom-protocols.jar) mit den zugehörigen Klassen, wird im Ordner %JAVA_HOME%\jre\lib\ext deponiert oder ihr Speicherort wird mit dem Parameter –cp beim Start des Space angegeben.

Die Codebase bildet ein passwortgeschützter Apache2 Webserver, welcher sich in Reichweite der Worker befinden muss. Der Passwortschutz soll zudem verhindern, dass Unbefugte die Jar-Dateien von der Codebase herunterladen und für eigene Zwecke einsetzen. Dazu enthält der Server ein Verzeichnis mit dem Namen codebase welches durch den folgenden Eintrag in die Konfigurationsdatei httpd.conf nur nach vorherigem Login zugänglich ist.

```
<Directory /srv/www/htdocs/codebase>
    AuthType Basic
    AuthName "Restricted Files"
```

[22] vgl. [Sun, 2007]

[23] vgl. [GigaSpaces, 2007]

```
    AuthUserFile /etc/apache2/passwords
    Require valid-user
</Directory>
```

Außerdem muss mit dem Befehl htpasswd2 die unter AuthUserFile angegebene Datei mit den Login-Daten angelegt werden. Im Anschluss können alle auf der Codebase benötigten Dateien in den Ordner codebase geladen werden. Dazu gehört unter anderem die Datei computegrid-dl.jar, welche die Class-Dateien der einzelnen Jobs enthält. Jar-Dateien auf der Codebase werden laut Konvention mit –dl für download gekennzeichnet.

4.3 Verwendete Software

An dieser Stelle soll kurz die verwendete Software erwähnt werden. Als Grundlage für die Anwendung dient die Programmiersprache Java in der Version 1.5.0. Ältere Versionen von Java können nicht verwendet werden, da Sprachfeatures von Java 1.5, wie zum Beispiel Generics und For-Each-Loops, im Source-Code eingesetzt wurden. Weiterhin wurden das Jini Starterkit in der Version 2.1 und der Blitz JavaSpace in Version 1.29 verwendet. Die Eclipse Rich Client Platform, auf der die einzelnen Anwendungen jeweils aufbauen, wurde in der Version 3.2.1 eingesetzt.

4.4 Die Klasse TaskEntry

Ein TaskEntry stellt einen logisches Teilproblem dar. Jobs teilen Probleme in Tasks, also einzelne Teilberechnungen, auf um diese verteilt und parallel berechnen zu lassen. Solche für einen Job spezifischen Tasks müssen die Klasse TaskEntry aus Abbildung 4.2 erweitern, um von den Workern als Tasks erkannt zu werden.

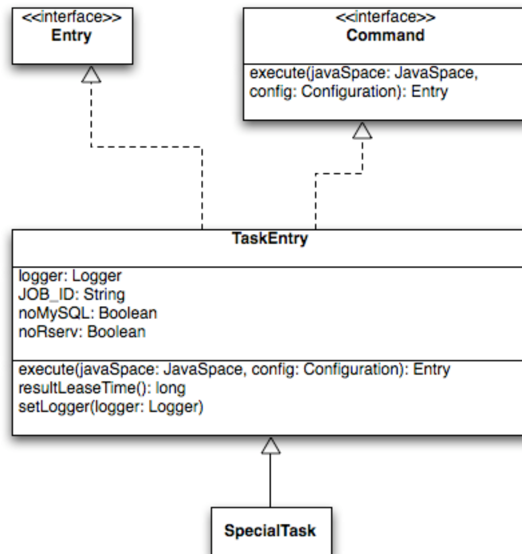

Abbildung 4.2 Die Klasse TaskEntry

Die Klasse TaskEntry implementiert zwei Interfaces. Zum einen ist dies das Entry Interface aus dem Paket net.jini.core.entry. Dieses Interface ist ein so genanntes Marker-Interface und ist nötig um einen TaskEntry in einen JavaSpace schreiben zu können. Außerdem implementiert die Klasse TaskEntry das Command-Interface, das zur Umsetzung des in Abschnitt 3.5.2 beschriebenen Command-Patterns dient. Die Execute-Methode enthält dabei die auf dem Worker auszuführende Logik. Da Worker im JavaSpace lediglich nach Objekten der Klasse TaskEntry suchen, können sie alle Tasks finden und deren Execute-Methode ausführen, ohne diese im Voraus zu kennen. Die dabei benötigten, dem Worker unbekannten Klassen, müssen auf der Codebase hinterlegt werden. Werden dem Master neue Jobs hinzugefügt ist damit kein Update der Worker notwendig.

Wie oben schon erwähnt müssen alle speziellen Tasks die Klasse TaskEntry erweitern. Normalerweise wäre es hierbei logisch die Execute-Methode der Klasse TaskEntry als abstrakte Methode zu implementieren, um damit die Unterklassen zu zwingen diese zu implementieren. Auf diese Maßnahme wurde hier jedoch bewusst verzichtet, da eine Verwendung von TaskEntry als Template zur Suche im JavaSpace sonst nicht möglich wäre. Dies ist aber nötig, damit die Worker jede Art von Task im JavaSpace finden können. Um die Unterklassen dennoch zur Implementierung beziehungsweise zum

Überschreiben der Execute-Methode zu zwingen, besteht diese nicht aus einer leeren Implementierung, sondern wirft eine Exception des Typs java.lang.RuntimeException.

4.5 Die Klasse ResultEntry

Die Klasse ResultEntry aus Abbildung 4.3 bildet das Gegenstück zur Klasse TaskEntry.

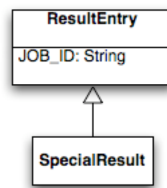

Abbildung 4.3 Die Klasse ResultEntry

Hat ein Worker die Execute-Methode eines TaskEntry ausgeführt, gibt diese ein Objekt des Typs ResultEntry zurück. Dieses Objekt enthält die Ergebnisse der Berechnung und wird vom Worker in den JavaSpace geschrieben um dort vom Master aufgesammelt zu werden. Die Klasse ResultEntry enthält außer der JOB_ID keine weitere Implementierung. Die JOB_ID dient der Identifizierung der Tasks. Der Master kann so bei gleichzeitiger Ausführung mehrerer Jobs die gesammelten Results dem jeweils richtigen Job zuordnen.
ResultEntry ist wie TaskEntry aus Abschnitt 4.4 ebenfalls nicht abstrakt da es vom Master sonst nicht als Template zum Auffinden der ResultEntry-Objekte im JavaSpace verwendet werden könnte.

4.6 Entwicklung der Worker-Anwendung

Die Worker-Anwendung bildet die Grundlage des Replicated-Worker-Patterns. Worker überwachen den JavaSpace und bearbeiten dort auftauchende Tasks. Worker sind so konzipiert, dass sie alle Arten von Tasks bearbeiten können, es sei denn die Tasks benötigen spezielle lokale Ressourcen wie zum Beispiel einen lokalen MySQL-Server oder einen lokalen R-Server. Um sicherzustellen, dass Tasks die solche Ressourcen benötigen

nur von Workern entnommen werden, die diese bereitstellen, beinhaltet die Klasse TaskEntry Flags um festzustellen welche Ressourcen benötigt, beziehungsweise nicht benötigt werden. Die Flags noMySQL und noRserv folgen einer inversen Logik. Sie geben sozusagen an, dass der Task kein MySQL oder keinen R-Server benötigt. Das resultiert daraus, dass im JavaSpace lediglich nach Objekten mit bestimmten Attributwerten gesucht werden kann. Hingegen ist es nicht möglich auf das Nichtvorhandensein eines bestimmten Attributwertes zu prüfen. Um zu ermöglichen, dass Worker mit speziellen lokalen Ressourcen auch Tasks berechnen, die diese nicht benötigen, wurde deshalb diese inverse Logik gewählt.
An dieser Stelle werde ich den Ablauf des Workers näher beschreiben.

1. Der Worker lädt die Konfigurationsdatei properties.xml mit den Konfigurationsdaten der lokalen Ressourcen.
2. Der Worker sucht via Jini-Lookup einen JavaSpace und einen TransactionManager.
3. Nachdem der JavaSpace gefunden wurde schreibt der Worker einen InfoEntry mit Statusinformationen wie IP, Rechnername und Informationen zu lokalen Ressourcen in den Space.
4. Im Anschluss daran wartet der Worker auf im Space auftauchende TaskEntries.
5. Hat der Worker einen TaskEntry im Space gefunden entnimmt er diesen und führt dessen Execute-Methode aus.
6. Ist die Excute-Methode beendet gibt sie ein Objekt des Typs ResultEntry zurück, das daraufhin vom Worker in den JavaSpace geschrieben wird.
7. Im Anschluss wartet der Worker wieder auf neue TaskEntries.

4.6.1 Die Klasse GenericWorker

Die Klasse GenericWorker realisiert die Logik des Workers. Im Folgenden möchte ich den Ablauf aus dem vorangegangenen Abschnitt anhand einiger wichtiger Ausschnitte aus der Klasse GenericWorker näher erläutern.

Im Konstruktor der Klasse erfolgen das Laden der Konfigurationsdatei sowie die Suche nach einem JavaSpace und einem TransactionManager. Zur Suche der beiden Jini-Services, wird die Helper-Klasse ServiceFinder verwendet. Im Anschluss erfolgt die Initialisierung des Jobs, der später die Tasks bearbeitet. Außerdem wird wie im letzten Abschnitt schon erwähnt ein InfoEntry in den Space geschrieben. Dieser wird über einen LeaseRenewalManager solange erneuert, bis der Worker beendet wird. Der Master kann anhand der InfoEntries im Space feststellen, wie viele Worker verfügbar, ob diese aktiv sind und welche speziellen Ressourcen sie bereitstellen.

Mittels der beiden Methoden start() und stop() kann der Worker gestartet beziehungsweise gestoppt werden. Die beiden Methoden starten oder stoppen dabei den im Konstruktor definierten Job.

Dieser Job stellt das Herzstück des Workers dar. Wie der folgende Ausschnitt aus dem Sourcecode des Workers zeigt ist es Aufgabe des Jobs TaskEntries jeweils unter Verwendung einer Transaktion aus dem Space zu entnehmen und sie zu berechnen. Wurde ein TaskEntry erfolgreich berechnet, schreibt der Worker den erstellten ResultEntry in den JavaSpace. Kommt es während der Berechnung zum Absturz des Workers oder verliert dieser die Verbindung zum JavaSpace, sorgt die verwendete Transaktion dafür, dass der entnommene Entry nicht verloren ist, sondern wieder zur Bearbeitung im Space bereitsteht.

```
LOG.info ("Started processing tasks");
while (true)
{
	LOG.trace ("Trying to get a task from space");

	Transaction txn = getTransaction ();
	if (txn == null)
	{
		LOG.fatal ("Failed obtaining a transaction");
		throw new RuntimeException ("Failed obtaining a transaction");
	}
```

```
        try
        {
                ClassLoader loader =
Thread.currentThread().getContextClassLoader ();
                Thread.currentThread ().setContextClassLoader
(TaskEntry.class.getClassLoader ());

                Entry entry = space.getService ().take (taskTmpl, txn, 30000);

                Thread.currentThread ().setContextClassLoader (loader);

                if (entry != null)
                {
                        TaskEntry task = (TaskEntry) entry;
                        task.setLogger (LOG);
                        Entry result = task.execute (space.getService (),
config);
                        space.terminate ();

                        if (result != null)
                        {
                                Object leaseTime = task.resultLeaseTime ();

                                space.getService ().write ((Entry) result, txn,
((Long) leaseTime).longValue ());
                                space.terminate ();
                        }
                }
                else
                {
                        LOG.trace ("Currently no task available");
                }
                txn.commit ();
        }
        catch (RemoteException e)
        {
                LOG.error (e, "Network error");
                e.printStackTrace ();
```

```
        }
        catch (TransactionException e)
        {
                LOG.error (e, "Transaction timed out with no commit");
                e.printStackTrace ();
        }
        catch (UnusableEntryException e)
        {
                LOG.error (e, "Problem with retrieved entry");
                e.printStackTrace ();
        }
        catch (InterruptedException e)
        {
                LOG.warn (e, "Task has been canceled");
                try
                {
                        txn.abort ();
                }
                catch (Exception e1)
                {
                        LOG.error (e1, "Failed aborting transaction");
                }
        }

        if (monitor.isCanceled ())
                break;
}
LOG.info ("Canceled Processing Tasks");
return Status.CANCEL_STATUS;
```

Da der Worker immer nach Entries gleichen Typs[24] sucht, wird hier anstatt eines normalen Templates ein Snapshot verwendet. Der folgende Codeabschnitt fordert einen solchen Snapshot vom Space an.

```
// creating template according to local resources
TaskEntry te = new TaskEntry ();
```

[24] Der Worker sucht immer nach TaskEntry-Objekten

```
if (config.MYSQL == false)
        te.noMySQL = true;
if (config.RSERV == false)
        te.noRserv = true;
taskTmpl = javaSpace.snapshot (te);
```

Die Verwendung des Snapshots hat hierbei den Vorteil, dass das Template nicht bei jeder Verwendung erneut serialisiert werden muss, sondern in serialisierter Form im Space zwischengespeichert wird.

4.6.2 Entwurf der Benutzeroberfläche

Abbildung 4.4 und Abbildung 4.5 zeigen die Benutzeroberfläche der Worker-Anwendung. Diese ist bewusst einfach gehalten, da der Worker normalerweise keine Interaktion mit dem Benutzer erfordert. Die Oberfläche besteht aus zwei wesentlichen Bestandteilen. Zum einen besteht die Oberfläche aus einem Fenster, welches Debug-Informationen ausgibt und die Steuerung das Workers ermöglicht.

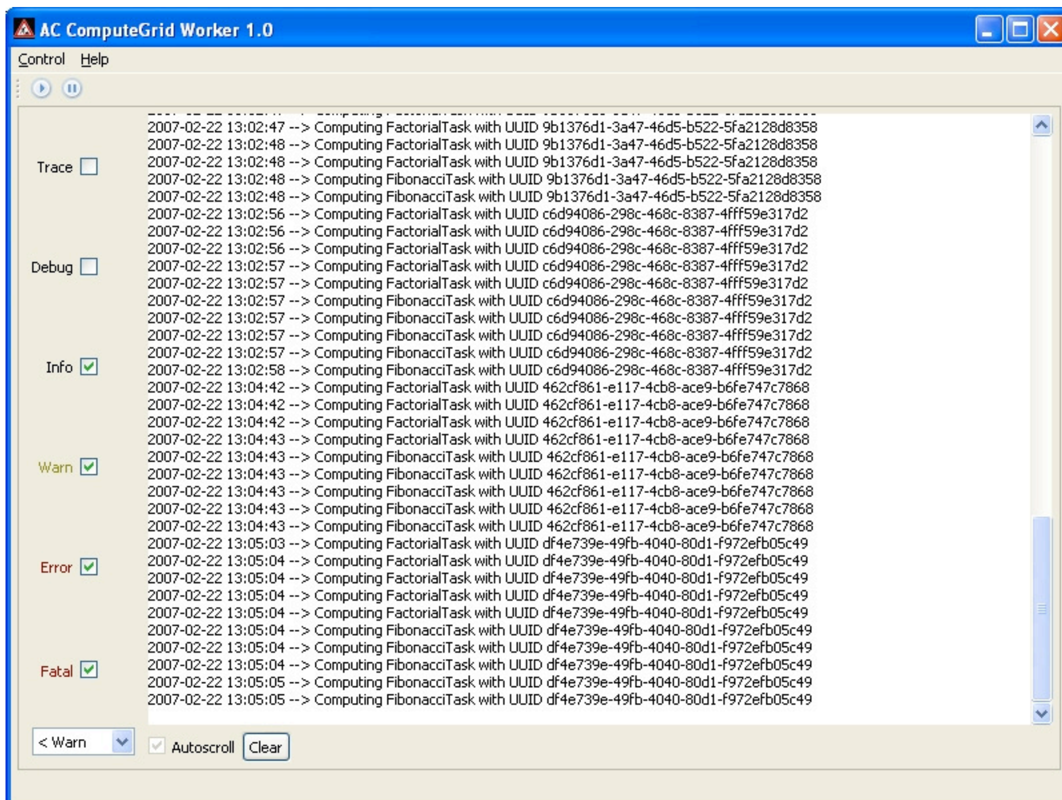

Abbildung 4.4 Benutzeroberfläche des Workers

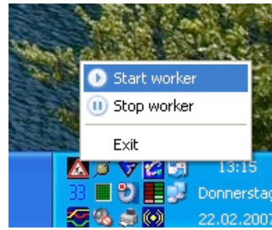

Abbildung 4.5 Worker im Systray

Dazu enthält das Fenster eine Toolbar, welche Zugriff auf die Start- und Stop-Actions des Workers ermöglicht.
Der zweite wesentliche Bestandteil ist ein Icon mit Menü im Systray des Worker-Rechners. Wird das Anwendungsfenster geschlossen, minimiert sich die Anwendung automatisch in den Systray. Über das Kontextmenü des Icons kann dann ebenfalls auf die Start- und Stop-Actions des Workers zugegriffen werden.

4.7 Entwicklung der Masteranwendung

Die Masteranwendung ist dafür verantwortlich ein Gesamtproblem zur verteilten Berechnung in Teilprobleme aufzuteilen. Diese Teilprobleme werden dann in Form von TaskEntries in den JavaSpace geschrieben, um dort von den Workern berechnet zu werden. Weiterhin sammelt der Master alle ResultEntries aus dem Space und fügt sie gegebenenfalls zu einer Gesamtlösung zusammen. Abbildung 4.6 zeigt den Master im Überblick.

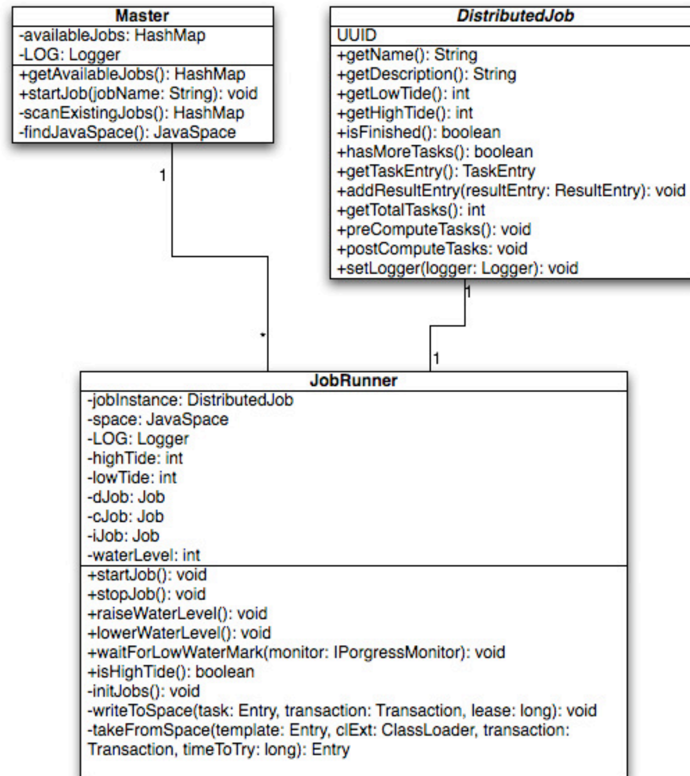

Abbildung 4.6 Der Master im Überblick

4.7.1 Die Klasse Master

Die Klasse Master ist für die Verwaltung der verfügbaren Jobs zuständig. Beim Start des Masters prüft die Methode scanExistingJobs() welche Jobs am Jobs Erweiterungspunkt registriert, also verfügbar sind. Da das scannen des Erweiterungspunktes im Abschnitt 4.7.5 detailliert beschrieben wird, soll hier nicht weiter darauf eingegangen werden. Mit Hilfe der Start-Methode des Masters können die Jobs dann unter Angabe ihrer Bezeichnung gestartet werden. Dazu wird eine Instanz der Klasse JobRunner erstellt, welche die Verteilung der Tasks und das sammeln der Results übernimmt.

4.7.2 Die Klasse JobRunner

Die Klasse JobRunner verwaltet die Abarbeitung eines Jobs. Wie auch im Worker kommt im JobRunner das Eclipse-Jobs-API zum Einsatz. Die Klasse besteht dabei im Wesentlichen aus zwei dieser Eclipse-Jobs. Dies sind zum einen ein Job zur Verteilung der Tasks und zum anderen ein Job zum

sammeln der Results. Durch diese beiden parallelen Jobs wird gewährleistet, dass gleichzeitig Tasks geschrieben und Results gesammelt werden können. Wäre dies nicht der Fall, könnte es zu einer Überfüllung des JavaSpace kommen, was zu dessen Absturz führen würde.
Der Job zur Verteilung schreibt dabei solange Tasks in den Space, bis alle Tasks des Jobs abgearbeitet sind. Um den JavaSpace so voll wie nötig zu halten, ihn gleichzeitig aber nicht zu überfüllen, wird das so genannte Watermarking-Verfahren verwendet. Dieses Verfahren wird in Abschnitt 4.7.3 beschrieben, sodass hier nicht näher darauf eingegangen werden muss.

```
while (jobInstance.hasMoreTasks ())
{
	if (monitor.isCanceled ())
	{
		LOG.warn ("Task-Distribution canceled for Job with ID: {}", 
jobInstance.UUID);
		return Status.CANCEL_STATUS;
	}

	// waiting here if space is full
	waitForLowWaterMark (monitor);

	if (monitor.isCanceled ())
	{
		LOG.warn ("Task-Distribution canceled for Job with ID: {}", 
jobInstance.UUID);
		return Status.CANCEL_STATUS;
	}
	LOG.warn ("Reached low tide. Distributing new Tasks");
	while (!isHighTide () && jobInstance.hasMoreTasks ())
	{
		writeToSpace (jobInstance.getTaskEntry (), null, 
Lease.FOREVER);
		raiseWaterLevel ();
		monitor.worked (1);
	}
}
```

Der zweite Job des JobRunners ist für das Einsammeln der fertigen Results aus dem Space zuständig. Als Template wird ein ResultEntry mit der UUID des Jobs verwendet. Dies stellt sicher, dass jeder JobRunner nur die Results seines eigenen Jobs einsammelt. Da sich das Template während der gesamten Abarbeitung des Jobs nicht ändert, wird auch hier die Snapshot-Version verwendet.

```
monitor.beginTask ("Collecting Results", jobInstance.getTotalTasks ());

ClassLoader cl = jobInstance.getClass ().getClassLoader ();
Entry resultTemplate;

// creating a template of TaskEntry to find any Task in the space no matter
what Task it is

try
{
	resultTemplate = space.snapshot (new ResultEntry (jobInstance.UUID));
}
catch (RemoteException e)
{
	throw new RuntimeException ("Can't obtain a snapshot from the space");
}

while (!jobInstance.isFinished ())
{
	if (monitor.isCanceled ())
		return Status.CANCEL_STATUS;
	ResultEntry result = null;
	try
	{
		result = (ResultEntry) takeFromSpace (resultTemplate, cl, null,
null);
	}
	catch (Exception e)
	{
```

```
                e.printStackTrace ();
        }

        if (result != null)
        {
                jobInstance.addResultEntry (result);
                monitor.worked (1);
                lowerWaterLevel ();
        }
}

jobInstance.postComputeTasks ();

setProperty (IProgressConstants.KEEP_PROPERTY, Boolean.TRUE);

return Status.OK_STATUS;
```

4.7.3 Watermarking

Um den JavaSpace nicht zu überfüllen, was dessen Absturz zur Folge hätte, gleichzeitig aber auch die Worker nicht zu unterfordern wird, wie im vorangegangenen Abschnitt schon erwähnt, die Watermarking-Technik[25] eingesetzt. Der Name ist abgeleitet von den beiden Linien die Ebbe und Flut an Stränden hinterlassen. Ausgehend davon definiert die Anwendung eine maximale und eine minimale Anzahl an Entries, die sich im Space befinden sollten. Die Festlegung dieser Grenzen erfolgt frei nach dem Motto: „So wenig wie möglich, so viel wie nötig". Die Grenzen werden im Job definiert und können über die Methoden getHightide() und getLowTide() abgerufen werden. In einer Folgeversion der Anwendung wäre auch die Kopplung dieser Grenzen an die Anzahl der verfügbaren Worker denkbar.

Der Ablauf des Watermarking gliedert sich nun wie folgt. Zuerst wird der Space solange befüllt, bis die Obergrenze erreicht ist. Die Worker bearbeiten nun solange Tasks, bis der Wasserstand im Space auf die Untergrenze

[25] vgl. [Freeman et al., 1999, S. 296]

gefallen ist. Daraufhin wird der Space vom JobRunner wieder bis zur Obergrenze gefüllt.
Die Realisierung der Technik erfolgt über die folgenden Methoden der JobRunner-Klasse.

```
public synchronized void raiseWaterLevel ()
{
	waterLevel++;
}

public synchronized void lowerWaterLevel ()
{
	waterLevel--;
	notifyAll ();
}

public synchronized void waitForLowWaterMark (IProgressMonitor monitor)
{
	while (waterLevel > lowTide)
	{
		if (monitor.isCanceled ())
			break;
		try
		{
			wait ();
		}
		catch (InterruptedException e)
		{
			// continue
		}
	}
}

public boolean isHighTide ()
{
	return waterLevel >= highTide;
}
```

Der JobRunner schreibt demnach solange TaskEntries in den JavaSpace bis die Methode isHighTide() den Wert true zurückliefert. Dabei wird bei jedem geschriebenen Entry der Wasserspiegel angehoben. Ist der Space bis zur Obergrenze gefüllt, blockiert die WaitForLowWaterMark-Methode die Verteilung weiterer Tasks solange bis der Wasserstand auf die Untergrenze gefallen ist. Sofern weitere Tasks vorhanden sind wird der Space daraufhin wieder bis zu Limit gefüllt.

4.7.4 Entwurf der Benutzeroberfläche

Die Benutzeroberfläche des Masters besteht aus drei verschiedenen Perspektiven. Die ControlCenter-Perspektive[26] wird nach dem Start der Anwendung gezeigt und bietet die verfügbaren Jobs über ein DropDown-Feld zum Starten an. Außerdem wurde der Eclipse-ProgressView in die Perspektive integriert, um dem Benutzer den aktuellen Bearbeitungsstand der Jobs anzuzeigen. Über den ProgressView kann der Job außerdem falls notwendig abgebrochen werden.

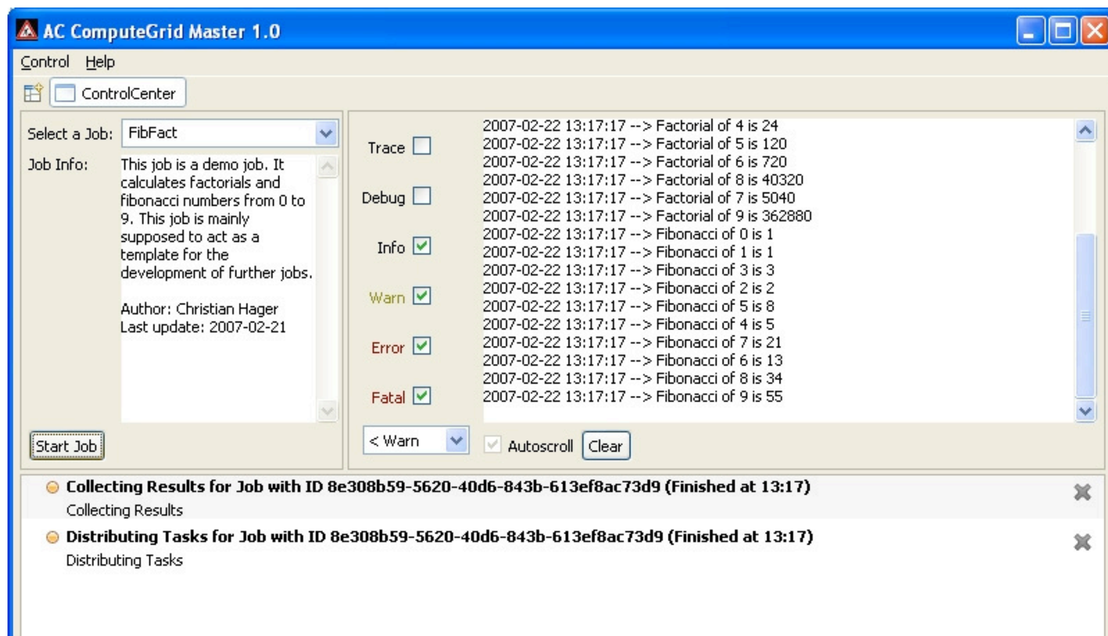

Abbildung 4.7 Die ControlCenter-Perspektive des Masters

[26] siehe Abbildung 4.7

Der ebenfalls in die ControlCenter-Perspektive integrierte DebugView zeigt dem Benutzer die Log-Meldungen des Masters an. Durch setzen der Checkboxen auf der linken Seite des Views, können die Log-Meldungen nach Level gefiltert werden.

Die Monitor-Perspektive aus Abbildung 4.9 bietet eine Übersicht über die momentan verfügbaren Worker und deren Status. Über die Maintenance-Perspektive aus Abbildung 4.8 können wenn nötig bestimmte Entries aus dem Space entfernt werden.

Zur Realisierung der Views wurde die Eclipse Rich Client Platform sowie das SWT-Framework verwendet. Das Standard Widget Toolkit bietet gegenüber Swing den Vorteil, dass die GUI-Elemente, wo dies möglich ist, auf den systemeigenen Widgets basieren. Somit erhält die Benutzeroberfläche das systemeigene Look&Feel, was die optische Integration in die jeweilige Platform erheblich verbessert.

Die hier vorgestellten Views sind alle als Eclipse RCP Views realisiert und wurden über den ExtensionPoint org.eclipse.ui.views in die Rich Client Anwendung integriert. Die folgende XML-Definition aus dem org.ac.computegrid.master.ui Plugin realisiert die Anbindung der Views des Masters an den ExtensionPoint.

```
<extension point="org.eclipse.ui.views">
	<view
		allowMultiple="false"
		class="org.ac.computegrid.master.ui.JobControlView"
		id="org.ac.computegrid.master.ui.JobControlView"
		name="Job Control View">
	</view>
	<view
		name="Progress View"
		icon=""
		category="org.eclipse.ui"
		class="org.eclipse.ui.ExtensionFactory:progressView"
		id="org.eclipse.ui.views.ProgressView">
	</view>
	<view
```

```
            class="org.ac.computegrid.master.ui.DebugView"
            id="org.ac.computegrid.master.ui.DebugView"
            name="Debug View">
      <view/>
</extension>
```

Die Anordnung der Views erfolgt über den ExtensionPoint org.eclipse.ui.perspective und die Perspective-Klasse. Die Methode createInitialLayout(IPageLayout layout) der Perspective-Klasse erstellt dafür ein Layout und fügt diesem die verschiedenen Views hinzu.

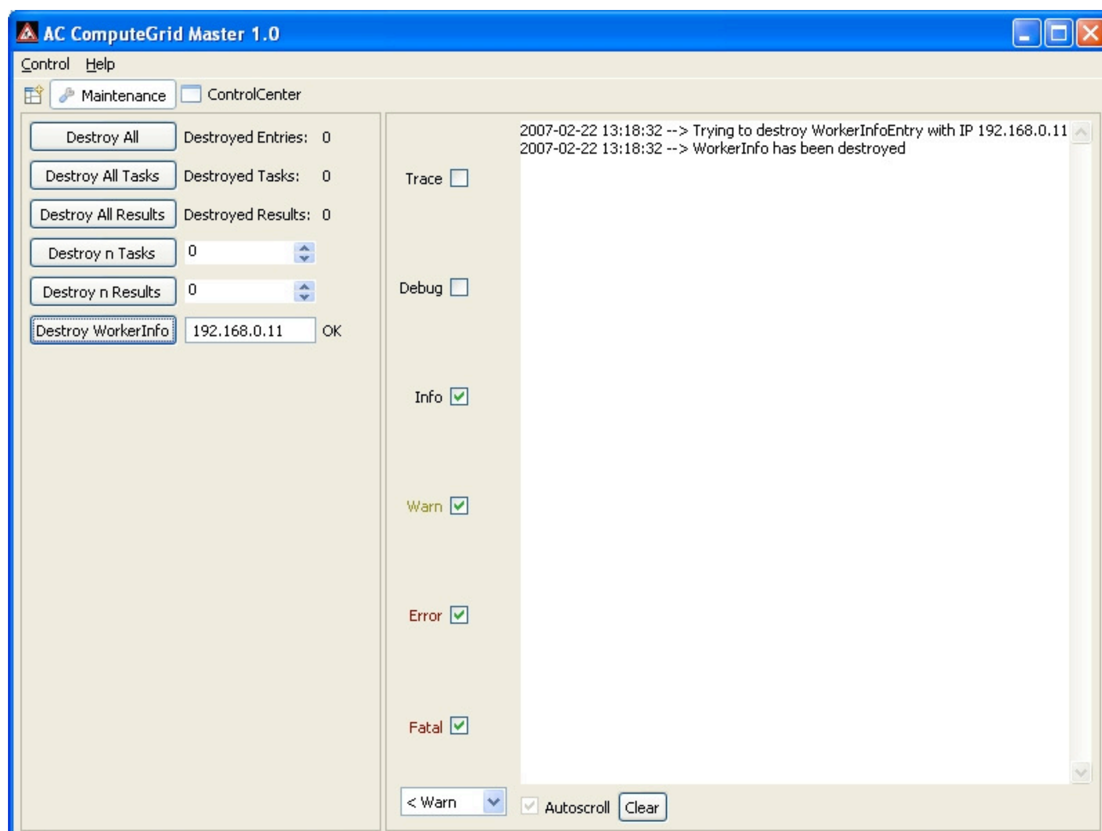

Abbildung 4.8 Die Maintenance-Perspektive des Masters

Abbildung 4.9 Die WorkerMonitor-Perspektive des Masters

4.7.5 Der Extension Point Jobs

Um die Entwicklung neuer Jobs so einfach und flexibel wie möglich zu gestalten, stellt die Master-Applikation eine Schnittstelle zur Ankopplung der Jobs in Form eines ExtensionPoints bereit. An diesen ExtensionPoint können Jobs in Form von RCP Plugins angekoppelt werden, indem sie ihn in ihrer plugin.xml-Datei erweitern. Der ExtensionPoint erwartet einen Namen für den Job und den Name der Klasse des Jobs die die Klasse DistributedJob erweitert.

Beim Start der MasterAnwendung scannt diese alle über den ExtensionPoint verfügbaren Jobs, um sie dem Benutzer zur Ausführung zur Verfügung zu stellen. Der Scanprozess läuft dabei wie folgt ab. Zuerst wird mit

```
IExtensionRegistry registry = Platform.getExtensionRegistry ();
```

die Erweiterungsregistratur der Eclipse Plattform geholt. Von dieser kann mit

```
IExtensionPoint extpt =
registry.getExtensionPoint("org.ac.computegrid.master.jobs");
```

der Extension Point mit dem Namen „org.ac.computegrid.master.jobs“ angefordert werden. Der ExtensionPoint wiederum liefert einer Liste aller verfügbaren Extensions, also der verfügbaren Jobs.

```
IExtension[] job = extpt.getExtensions ()
```

Um nun die Hauptklasse des jeweiligen Jobs zu laden, benötigt man den

Name dieser Klasse. Folgender Code liefert dazu die am ExtensionPoint registrierten Attribute.

```
IConfigurationElement[] jobConf = job.getConfigurationElements ()
String name = jobConf.getAttribute ("name");
String className = jobConf.getAttribute ("className");
```

Mit Hilfe dieses Klassennamens und des Java-Reflection-APIs kann jetzt die entsprechende Klasse über den ClassLoader des den Job implementierenden Plugins geladen werden.

```
Class clazz = Platform.getBundle (job.getContributor ().getName
()).loadClass (className);
```

Das erhaltene Class-Objekt kann jetzt für die spätere Verwendung gespeichert werden. Eine Instanziierung der Klasse ist zu diesem Zeitpunkt noch nicht nötig, da noch nicht bekannt ist welchen Job der Benutzer starten möchte.
Über die Methode startJob(String jobName) können die verfügbaren Jobs jetzt instanziiert und ausgeführt werden.

4.8 Die Entwicklung der Jobs

Jobs stellen jeweils das zu lösende Gesamtproblem dar. Das Klassendiagramm in Abbildung 4.10 zeigt die für einen Job minimal benötigten Klassen. Dazu gehören zum einen die Hauptklasse des Jobs und zum anderen jeweils mindestens eine Spezialisierung der Klasse TaskEntry und der Klasse ResultEntry. Ebenso kann ein Job aber auch verschiedene Task- und ResultEntry erzeugen falls dies nötig ist.

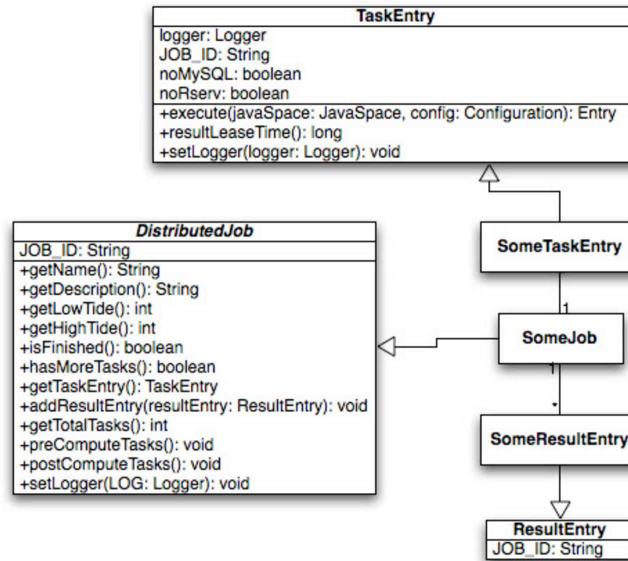

Abbildung 4.10 Aufbau eines Jobs

Alle Jobs werden in Form von Eclipse RCP Plugins erstellt und erweitern den vom Master bereitgestellten Erweiterungspunkt mit dem Namen org.ac.computegrid.master.jobs. Das erfordert die Angabe eines Namens für den Job sowie die Angabe des Namens der Hauptklasse des Jobs. Realisiert wird dies in der Datei plugin.xml des Job-Plugins durch den folgenden Bereich.

```
<extension point="org.ac.computegrid.master.jobs">
   <job className="org.ac.computegrid.jobs.fibfact.FibFactJob"
name="FibFact"/>
</extension>
```

Die Hauptklasse des Jobs, die in der Erweiterung angegeben wird, muss die Klasse DistributedJob des Masters erweitern. Abbildung 4.11 zeigt diese Klasse.

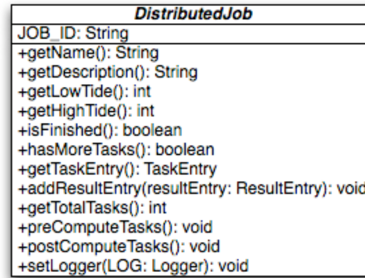

Abbildung 4.11 Das DistributedJob-Interface

Die beiden wichtigsten Methoden dieser Klasse sind getTaskEntry() und addResultEntry(). Der Master holt sich mittels getTaskEntry() neue Tasks, um sie in den JavaSpace zu schreiben und gibt mittels addResultEntry() fertige Results zur weiteren Verarbeitung an den Job zurück. Mittels der Methoden preComputeTasks() und postComputeTasks() können Aktionen zu Beginn sowie zum Abschluss des Jobs ausgeführt werden. Der Master ruft die PreComputeTasks-Methode auf bevor der erste Task des Jobs in den JavaSpace geschrieben wird. Die PostComputeTasks-Methode wird ausgeführt, wenn das letzte Result des Jobs eingesammelt wurde und kann zum Beispiel dafür verwendet werden, die einzelnen Results zu einem Gesamtergebnis zusammenzufügen. Außerdem kann mit Hilfe dieser Methode die weitere Verarbeitung des Ergebnisses initiiert werden.

4.8.1 Der Demo Job

Der Demo-Job soll die grundlegende Implementierung eines Jobs für den ComputeServer verdeutlichen und als Vorlage für die Entwicklung weiterer Jobs dienen. Als Grundlage für den Demo-Job soll hier das Beispiel aus [Freeman et al., 1999, S.170-173] dienen. Der Job berechnet die Fibonacci-Zahlen und Fakultäten der Werte 0 bis 9. Dabei wird die Berechnung eines jeden Wertes als einzelner Task im Space zur Bearbeitung zur Verfügung gestellt. Das Klassendiagramm in Abbildung 4.12 zeigt den Aufbau des Jobs. FibFactJob, die Hauptklasse des Demo-Jobs, erweitert die vom Master geforderte DistributedJob-Klasse. Weiterhin gibt es zwei TaskEntries mit den jeweils zugehörigen ResultEntries. Die beiden TaskEntries FibonacciTask und FactorialTask beinhalten jeweils die Logik zur Berechnung der Fibonacci

Zahlen beziehungsweise der Fakultäten. Die beiden ResultEntries sind lediglich Container, um die Ergebnisse der Berechnung an den Master zurückzugeben. Der Ablauf des Demo-Jobs gliedert sich in die Folgenden Punkte.

1. Als erstes generiert der Master durch Aufruf der Methode preComputeTasks() jeweils zehn Factorial- und FibonacciTasks.
2. Danach fordert der Master nacheinander alle Tasks vom Job via getTaskEntry() an und schreibt sie in den JavaSpace.
3. Die fertigen Results werden vom Master über die Methode addResultEntry() an den Job zurückgegeben.
4. Sind alle Tasks abgearbeitet und die jeweiligen Results eingesammelt führt der Master die Methode postComputeTasks() des Jobs aus welche die Ergebnisse der Berechnung über den DebugView ausgibt.

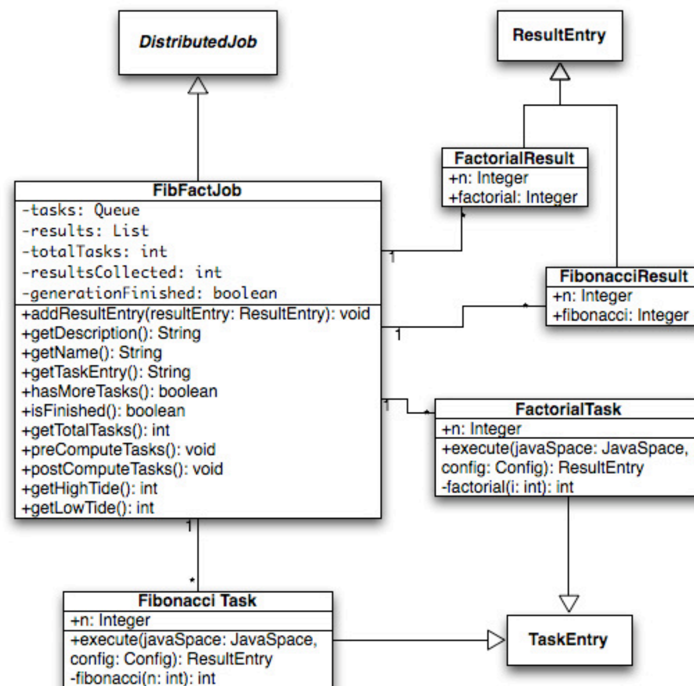

Abbildung 4.12 Klassendiagramm des Demo-Jobs

4.8.2 Programmieranweisung für weitere Jobs

In diesem Abschnitt sollen einige Punkte erwähnt werden, die bei der Entwicklung neuer Jobs zu beachten sind. Grundsätzlich gilt der Demo-Job als Vorlage für neue Jobs. Trotzdem soll hier eine Anleitung zur Entwicklung neuer Jobs in Form einiger Schritte gegeben werden.

1. Zuerst muss ein neues RCP-Plugin mit dem Namen org.ac.computegrid.jobs.<Name des Jobs> angelegt werden.
2. Danach wird im zuvor angelegten Plugin ein Package mit dem Namen org.ac.computegrid.jobs.<Name des Jobs> erstellt.
3. Jetzt kann die Implementierung der Jobklasse erfolgen. Diese muss das DistributedJob-Interface aus dem Plugin org.ac.computegrid.master implementieren.
4. Danach erfolgen die Implementierung der Task und Result-Klassen.
5. Um den neuen Job im Master verfügbar zu machen, muss das Job-Plugin den Erweiterungspunkt org.ac.computegrid.master.jobs unter Angabe eines Namens und der Jobklasse erweitern.
6. Im Anschluss muss das Job-Plugin im Configuration-Tab des Products des Masters hinzugefügt werden.
7. Als letztes müssen nun noch die neuen TaskEntry- und ResultEntry-Klassen zur Codebase hinzugefügt werden.

Nachdem diese sieben Schritte ausgeführt wurden, sollte der neue Job beim nächsten Start des Masters in der Auswahl zur Verfügung stehen.

5 Bedienung und Installation

5.1 Installation des JavaSpace

Die hier beschriebene Installation basiert auf einem Suse Linux 10.1 Server. Prinzipiell ist aber auch eine Installation auf jedem anderen System möglich. Die einzige Bedingung ist, dass eine Java VM für das entsprechende System verfügbar ist. Im Folgenden wird also ein System mit installiertem JRE in der Version 1.5 oder höher vorausgesetzt. Weiterhin wurden Jini in der Version 2.1 und Blitz in der Version 1.29 verwendet. Blitz kann aber auch durch andere JavaSpaces wie Outrigger oder GigaSpaces ersetzt werden.

Zu Beginn muss die Installation des Jini-Paketes erfolgen. Hierbei wurde die Installation via Starterkit gewählt. Das Setup des Starterkit führt durch die Installationsprozedur, sodass der Installationsprozess hier nicht näher erläutert werden muss.

Im Anschluss an Jini kann der Blitz JavaSpace mit Hilfe des Installers aus Abbildung 5.1 installiert werden. Dieser erfordert die Angabe des Installationspfades des Java JRE und des Jini-Paketes. Der Installer erstellt dann selbstständig Konfigurationsdateien und Start-Skripte für die jeweilige Plattform. Bei der hier verwendeten Linux-Distribution handelt es sich dabei um Sh-Shell Skripte.

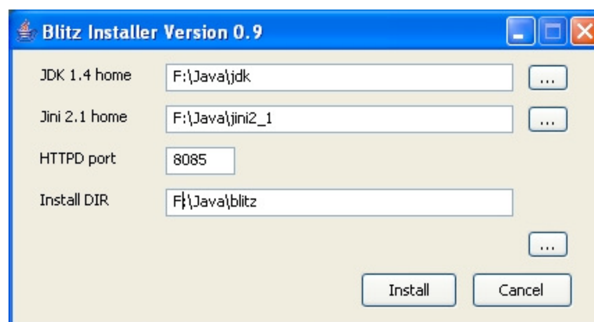

Abbildung 5.1 Der Installer des Blitz JavaSpace

Im Anschluss an die Installation sind noch einige Anpassungen der Konfigurationen und Start-Skripte an die Verwendung des Apache2 Webservers als Codebase notwendig.

In der Datei start-blitz.config aus dem config Ordner der Blitz Installation, müssen die Ports aller Codebase-Einträge auf den Port des Apache angepasst werden. Im Start-Skript blitz.sh des JavaSpace muss durch Angabe des VM-Parameters

```
-Djava.protocol.handler.pkgs=org.ac.net.protocols
```

der Protocol-Handler zur Unterstützung des Passwortschutzes des Apache Servers hinzugefügt werden. Weiterhin muss das Jar-Archiv mit der entsprechenden Handler-Klasse mit

```
-cp lib/custom-protocols.jar
```

zum Klassenpfad des JavaSpace hinzugefügt werden. Das entsprechende Jar-Archiv ist im Anschluss am angegebenen Ort zu hinterlegen. Weiterhin sind folgenden Jar-Dateien auf der Codebase zu hinterlegen.

- blitz-dl.jar
- reggie-dl.jar
- jsk-dl.jar
- mahalo-dl.jar

Als letztes sollte nun noch die aktive Leasebereinigung in der Konfiguration des JavaSpace aktiviert werden. Dazu muss der Parameter leaseReapInterval in der Datei blitz.config von 0 auf 10000 geändert werden. Dies bewirkt, dass der JavaSpace Entries mit abgelaufenen Leases einmal pro 10s bereinigt, um Speicherplatz freizugeben.
Danach ist der JavaSpace fertig konfiguriert und kann gestartet werden. Auf dem Linux Server sollte der Space vorzugsweise mit Hilfe des Befehls nohup gestartet werden. Dies bewirkt, dass der JavaSpace beim Logout nicht beendet wird. An diesem Punkt sollte der JavaSpace laufen und kann von Master und Workern verwendet werden.

5.2 Installation der Worker

Um den Worker auf dem Zielsystem zu installieren, muss dieser zuerst mit Hilfe des Eclipse Product export wizard exportiert werden. Im Anschluss müssen dem exportierten Verzeichnis noch die beiden Dateien policy.all und properties.xml hinzugefügt werden. An diesem Punkt ist der Worker lauffähig und könnte verwendet werden. Um die Installation der Worker zu vereinfachen, wurde mit Hilfe der Freeware [IzPack, 2007] zusätzlich noch ein Installationsprogramm erstellt. Mit diesem kann der Worker unter Angabe eines Installationspfades auf dem Zielsystem installiert werden.

5.3 Installation des Masters

Der Master wird genauso wie der Worker mit Hilfe des Eclipse Product export wizard exportiert. Auch hier muss im Anschluss die Datei policy.all hinzugefügt werden. Da der Master nur auf wenigen Rechnern zum Einsatz kommt, wurde hier auf die Erstellung eines Installationsprogramms verzichtet. Der exportierte Master muss somit lediglich auf das Zielsystem kopiert werden.

6 Entwicklungsstand

Der momentane Entwicklungsstand des AC ComputeServers ermöglicht Benutzern die verteilte Berechnung rechenintensiver Probleme sofern diese in verteilbare Einzelprobleme, so genannte Tasks, aufteilbar sind. Mehrere Anwender können dabei gleichzeitig Probleme bearbeiten ohne sich gegenseitig zu stören. Entwickler können Probleme in Form neuer Jobs hinzufügen. Eine Anleitung dazu wurde in Abschnitt 4.8.2 gegeben. Die Anwendung ist gut skalierbar. Bei Engpässen in Bezug auf die Rechenleistung können einfach neue Worker hinzugefügt werden.

Wie bei nahezu jeder Software gibt es natürlich auch mögliche Verbesserungen, die in kommenden Versionen vorgenommen werden können und sollten. Eine mögliche Erweiterung wäre hierbei unter anderem eine Erweiterung des Systems in Hinblick auf die Unterstützung mehrerer JavaSpaces. Dies wäre nicht nur in Hinsicht auf die Leistung der Anwendung wünschenswert, sondern auch in Hinblick auf die Ausfallssicherheit. Bei Verwendung mehrerer JavaSpaces könnte der Ausfall eines Spaces automatisch durch die anderen Spaces ausgeglichen werden. Weiterhin wäre eine Möglichkeit zur automatischen Steuerung der Worker vom Master aus sinnvoll. So sollten Worker vom Master aus gestartet beziehungsweise gestoppt werden können. Außerdem sollte eine Zeitsteuerung in die Worker integriert werden. So könnten sich die Worker auf den Entwicklerrechnern der Firma am Abend automatisch aktivieren, um über Nacht Tasks zu bearbeiten. Am Morgen, bevor die Arbeit an den Rechnern wieder aufgenommen wird, könnten sich die Worker dann wieder deaktivieren.

Um auch Pausen und Leerlaufzeiten zu nutzen wäre zusätzlich noch eine Aktivierung beziehungsweise Deaktivierung nach CPU-Auslastung denkbar.

7 Leistungs- und Skalierbarkeitsanalyse

In diesem Kapitel soll die Leistung und Skalierbarkeit der entwickelten Anwendung gemessen und analysiert werden. Dazu wurden zwei Experimente durchgeführt. Als Experimentiergrundlage diente hierbei ein Job, dessen Task-Implementierung lediglich ein Thread.sleep(...) enthält. Diese Vorgehensweise wurde gewählt, da die Testsysteme mit den Workern teilweise auf sehr unterschiedlicher Hardware aufbauten. Um gleiche Rechner zu simulieren wurde deshalb die genannte Methode zur Simulation gleicher Rechenzeit und somit gleicher Hardware verwendet.

Im ersten Experiment wurde die Anzahl der Worker konstant bei 8 Workern gehalten. Die Anzahl der zu berechnenden Tasks pro Durchgang lag bei 50. Pro Durchgang wurde nun die Rechenzeit der Tasks variiert und jeweils die Gesamtzeit für die Berechnung aller 50 Tasks gemessen. Tabelle 7.1 zeigt die Ergebnisse der Messung.

Rechenzeit der Tasks in ms	Rechenzeit im Grid in ms	Rechenzeit auf einem Einzelrechner in ms	Gesamtrechen-zeit für 50 Tasks in ms	Kommuni-kationszeit	Verhältnis Rechenzeit/ Kommuni-kationszeit
10	3.657	504	63	3.595	0,02
25	3.656	1.254	156	3.500	0,04
50	3.907	2.504	313	3.595	0,09
100	3.704	5.005	625	3.079	0,20
250	4.063	12.505	1.563	2.501	0,62
500	4.422	25.005	3.125	1.297	2,41
1.000	7.266	50.005	6.250	1.016	6,15
5.000	35.297	250.006	31.250	4.047	7,72
10.000	70.282	500.005	62.500	7.782	8,03
30.000	210.672	1.500.005	187.500	23.172	8,09
60.000	420.610	3.000.004	375.000	45.610	8,22
120.000	840.953	6.000.006	750.000	90.953	8,25
240.000	1.680.797	12.000.005	1.500.000	180.797	8,30
300.000	2.100.781	15.000.006	1.875.000	225.781	8,30

Tabelle 7.1 Ergebnisse des ersten Experimentes mit 8 Workern, 50 Tasks und variabler Rechenzeit der Tasks

Laut [Freemann et al., 1999, S. 157] lohnt sich die parallele Berechnung im Netzwerk nur dann, wenn die Zeit zur Berechnung der Tasks im Vergleich zur Kommunikationszeit verhältnismäßig groß ist. Wie in Abbildung 7.2 zu sehen ist, lohnt sich die parallele Berechnung im Netz der Firma Analytic Company GmbH höchstens bis zu einer minimalen Rechendauer von 250ms pro Task.

Ab einer Rechendauer 100ms pro Task bietet die parallele Berechnung keinen wesentlichen Vorteil mehr und ab 50ms pro Task dauert die verteilte Berechnung sogar länger als die Berechnung auf einem einzelnen Rechner. Das Diagramm in Abbildung 7.1 zeigt hingegen das bei höheren Rechendauern der Tasks die Berechnung im Grid einen wesentlichen Vorteil bietet.

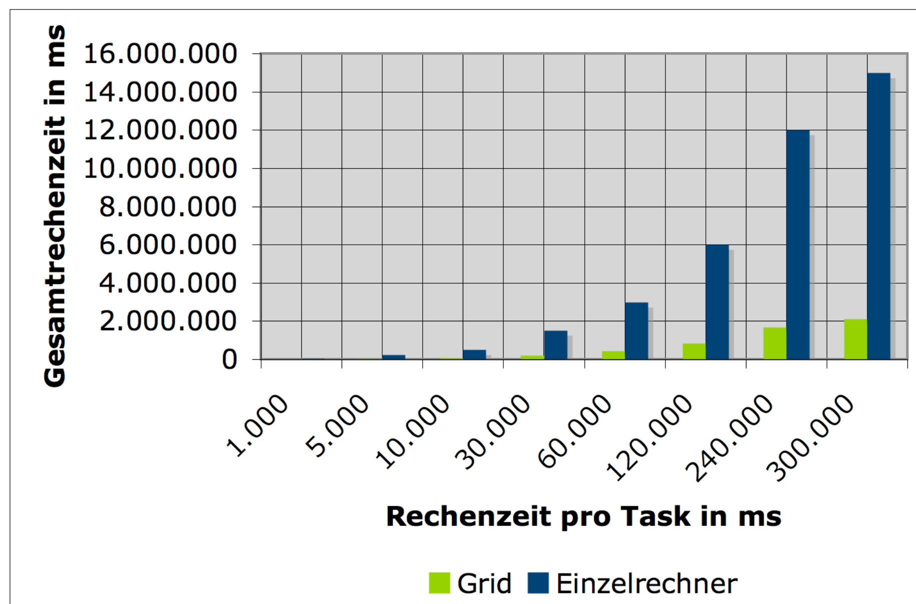

Abbildung 7.1 Gesamtrechenzeiten im Grid und auf einem Einzelrechner für eine Rechenzeit pro Task von 1s bis 300s

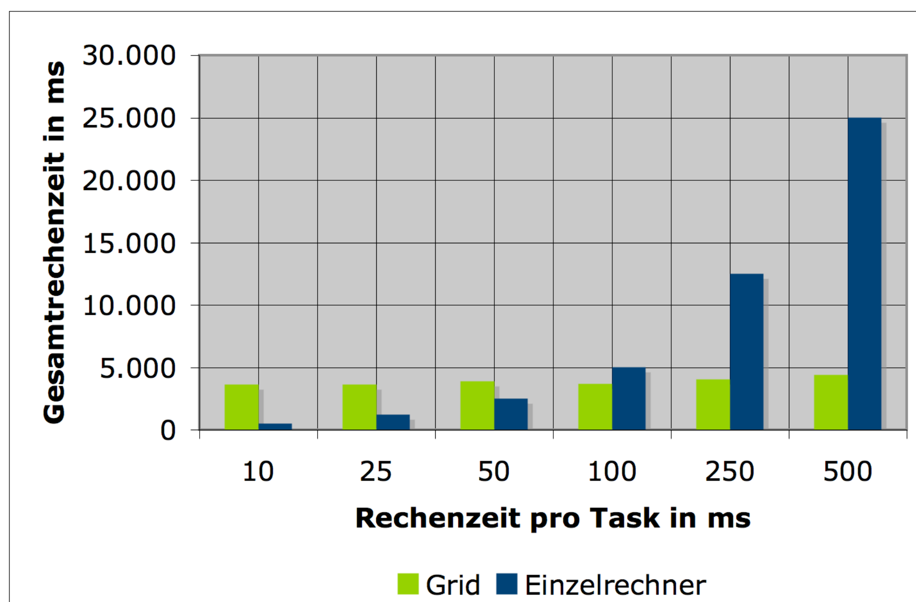

Abbildung 7.2 Gesamtrechenzeiten im Grid und auf einem Einzelrechner für eine Rechenzeit pro Task von 10ms bis 500ms

Abbildung 7.3 zeigt die Rechendauer im Grid im Vergleich zur Kommunikationszeit. Hier zeigt sich weshalb bei Taskrechenzeiten von 250ms und kleiner die Berechnung im Grid nicht mehr lohnenswert ist. Bei so kurzen Tasks nimmt die Kommunikation einen so großen Anteil der Gesamtzeit ein, dass die Berechnung insgesamt länger als auf einem einzelnen Rechner dauert.

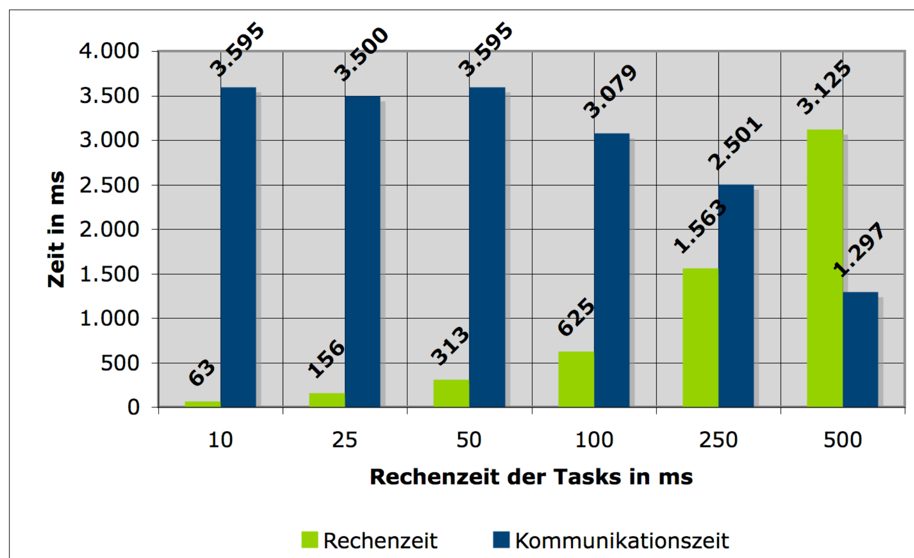

Abbildung 7.3 Verhältnis der Rechenzeiten im Grid zu den Rechenzeiten auf einem Einzelrechner

Im zweiten Experiment wurde die Rechendauer der Tasks konstant bei 5000ms gehalten und die Anzahl der Worker variiert. Tabelle 7.2 zeigt die Messergebnisse des zweiten Experiments.

Anzahl der Worker	Rechenzeit im Grid in ms	Rechenzeit auf Einzelrechner in ms	Grid/Einzelrechner in %	Verhältnis Rechenzeit/Kommu-nikationszeit
2	126.140	250.006	50,45%	2,000048
3	85.640	250.006	34,26%	3,000072
4	65.750	250.006	26,30%	4,000096
5	50.547	250.006	20,22%	5,00012
6	45.390	250.006	18,16%	6,000144
7	40.250	250.006	16,10%	7,000168
8	35.297	250.006	14,12%	8,000192

Tabelle 7.2 Ergebnisse des zweiten Experiments mit einer Rechenzeit von 5000ms pro Task, 50 Tasks und variabler Workerzahl

Abbildung 7.4 zeigt, dass es bei einer Verdopplung der Worker erwartungsgemäß zu einer annähernden Halbierung der Gesamtrechenzeit kommt.

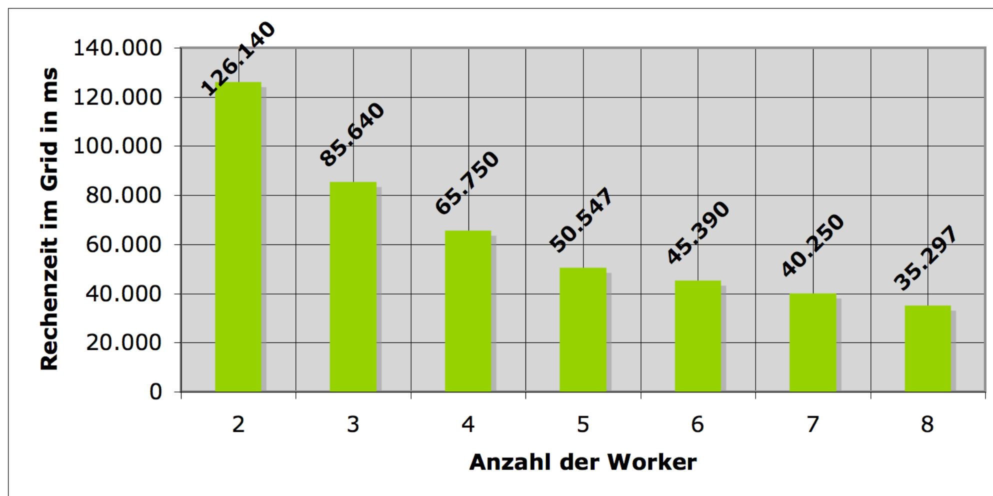

Abbildung 7.4 Rechenzeit im Grid bei variabler Workerzahl

Das Fazit der beiden Experimente ist also, dass bei ausreichender Rechendauer der Tasks, eine wesentliche Verringerung der Gesamtrechenzeit des Problems im Vergleich zur Berechnung auf einem einzelnen Rechner erreicht werden kann. Experiment 2 hat gezeigt, dass die verteilte Anwendung zusammen mit entsprechenden Tasks gut skalierbar ist.

8 Zusammenfassung und Ausblick

8.1 Zusammenfassung

Ziel dieser Arbeit war es ein verteiltes System zur parallelen Berechnung von Problemen im Netzwerk zu entwickeln. Dazu wurden in Kapitel 2 zunächst die Anforderungen analysiert. In Kapitel 3 wurde dann näher auf die technologischen Grundlagen des Systems eingegangen. Der Fokus lag dabei besonders auf den beiden im Rahmen der Entwicklung verwendeten Technologien Jini und JavaSpaces. Außerdem wurde auf die Grundlagen der beiden Technologien wie zum Beispiel Java RMI und die Serialisierung von Objekten eingegangen. Kapitel 4 beschäftigte sich dann mit Design und Realisierung der Anwendung. Dabei erfolgte zuerst eine Beschreibung des Gesamtkonzeptes das auf dem Replicated-Worker-Pattern aus [Freeman et al., 1999, S. 153-165] aufbaut. Danach wurde die Systemkonfiguration beschrieben und einzeln auf die Bestandteile des Systems sowie deren Implementierung eingegangen. Kapitel 5 erläuterte im Anschluss die zu Installation und Betrieb nötigen Schritte. So wurde dort unter anderem die Installation des JavaSpace und dessen Konfiguration beschrieben. In Kapitel 6 wurden dann der derzeitige Entwicklungsstand des Systems dargestellt und Ansatzpunkte zur Weiterentwicklung der Anwendungen gegeben. Das folgende Kapitel beschäftigte sich dann zum Abschluss noch mit einer Leistungs- und Skalierbarkeitsanalyse, die anhand zweier Experimente durchgeführt wurde.

8.2 Ausblick

Die Arbeit hat gezeigt, dass die Implementierung eines Systems zur verteilten Berechnung aufwändiger Probleme ein komplexes und vielschichtiges Thema ist. Nichtsdestotrotz steckt in der Entwicklung und Anwendung verteilter Systeme ein großes Potential.

Die Implementierung des verteilten Systems schafft die Grundlage zur verbesserten Nutzung bestehender Ressourcen der Firma sowie zu einer schnelleren Abarbeitung rechenintensiver Probleme, wo dies nötig ist.

Wie nahezu jede Anwendung bietet natürlich auch die im Rahmen dieser Arbeit entwickelte Anwendung Potential für eine Weiterentwicklung. So sollte die Applikation wie in Kapitel 6 beschrieben sowohl in Hinblick auf Ausfallsicherheit und Leistung als auch in Hinblick auf Handhabung optimiert werden. Dazu wäre, wie bereits erwähnt, der Einsatz mehrerer JavaSpaces denkbar. Mit Blick auf die Handhabung des Systems wäre eine Steuerung der Worker sowohl vom Master aus als auch auf Zeitbasis denkbar. Bei einer Fortsetzung des Projektes könnte an diesen Punkten angeknüpft werden um die Anwendung zu verbessern beziehungsweise weiterzuentwickeln.

Literaturverzeichnis

[Birrel & Nelson, 1984] Birrel, Andrew D. /Nelson, Bruce J.: ACM Transactions on Computer Systems, Ausgabe 2, Nr. 1, New York, USA: ACM Press, 1984, S. 39-59

[BOINC, 2007] BOINC: Berkeley Open Infrastructure for Network Computing: http://boinc.ssl.berkeley.edu/ (2007-02-08, 16:44)

[ComputeCycles, 2007] ComputeCycles: https://computecycles.dev.java.net/Welcome.html (2007-03-01, 1:45)

[ComputeFarm, 2007] ComputeFarm: https://computefarm.dev.java.net/ (2007-03-01, 01:42)

[Creswell, 2007] Creswell, Dan: Blitz JavaSpace: http://www.dancres.org/blitz/index.html (2007-03-01, 02:02)

[Deutsch, 2007] Deutsch, Peter: The Eight Fallacies of Distributed Computing: http://weblogs.java.net/jag/Fallacies.html (2007-01-17, 13:30)

[Deutschmann et al., 2004] Deutschmann, Jörg/Horn, Werner/Reif, Holger/Reschke, Dietrich/Schiller, Jochen H./Seitz, Jochen: Telematik: Netze - Dienste - Protokolle, 3. Auflage, München, Wien: Fachbuchverlag Leipzig im Carl Hanser Verlag, 2004

[Freeman et al., 1999] Freeman, Eric/Hupfer, Susanne/Arnold, Ken: JavaSpaces™: Principles, Patterns, and Practice, 1. Auflage, Reading, Massachusetts, USA: Addison-Wesley, 1999

[Freeman & Hupfer, 1999] Freeman, Eric/Hupfer, Susanne: Make room for JavaSpaces, Part1; Ease the development of distributed apps with JavaSpaces: http://www.javaworld.com/jw-11-1999/jw-11-jiniology.html (2007-03-03, 13:16)

[Gamma et al., 1994] Gamma, Erich/Helm, Richard/Johnson, Ralph/Vlissides, John: Design Patterns: Elements of Reusable Object-Oriented Software, 2. Auflage, Reading, Massachusetts, USA: Addison-Wesley, 1994

[Gelernter, 1985] Gelernter, David: Generative communications in Linda, in ACM Transactions on Programming Languages and Systems, Ausgabe 7, Nr.1, New York, USA: ACM Press, 1985, S. 80-112

[Gigaspaces, 2007] GigaSpaces: http://www.gigaspaces.com (2007-03-01, 02:08)

[ISO, 1996] International Organization for Standardization: ISO/IEC 10746: Open Distributed Processing Reference Model: http://standards.iso.org/ittf/PubliclyAvailableStandards/s018836_ISO_IEC_10746-2_1996(E).zip (2007-02-07, 00:01 MEZ)

[ISO, 1994] International Organization for Standardization: ISO/IEC 7498-1: Information Technology – Open Systems Interconnection – Basic Reference Model: http://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994(E).zip (2007-02-08, 11:56)

[IzPack, 2007] IzPack Java Installer: http://www.izforge.com/izpack/ (2007-02-22, 19:34)

[JINI.ORG, 2007] JINI.ORG: JINI: Introduction to Jini: http://www.jini.org/wiki/Category:Introduction_to_Jini (2007-02-08, 17:46)

[Newmarch, 2006] Newmarch, Jan: Foundations of JINI™ 2 Programming: Learn to build SOA-driven distributed systems using the new and open source Jini™ 2 technology, Berkeley, Californien, USA: Apress, 2006

[Sun, 2007] Sun Microsystems, Inc.: JavaSpaces Specification: https://java.sun.com/products/jini/2.1/doc/specs/html/js-spec.html (2007-02-07, 23:50 MEZ)

[Sun, 2007a] Sun Microsystems, Inc.: Java RMI Specification: http://java.sun.com/javase/6/docs/platform/rmi/spec/rmiTOC.html (2007-02-08, 12:07)

[Sun, 2007b] Sun Microsystems, Inc.: Jeri Specification: http://java.sun.com/products/jini/2.1/doc/specs/api/net/jini/jeri/connection/doc-files/mux.html (2007-02-08, 12:16)

[Tanenbaum & van Stehen, 2003] Tanenbaum, Andrew/van Steen, Marten: Verteilte Systeme: Grundlagen und Paradigmen, München: Pearson Studium, 2003

[Ullenboom, 2007] Ullenboom, Christian: Java ist auch eine Insel: Programmieren mit der Java Standard Edition Version 6, 6. Auflage, http://www.galileocomputing.de/openbook/javainsel6/ (2007-02-08, 12:12), Galileo Computing, 2007

A Inhalt der CD

Die beigefügte CD enthält sämtliche Sourcen der im Rahmen dieser Arbeit entwickelten Anwendung. Um eine einfache Verwendung zu gewährleisten, wird die Anwendung in Form eines Eclipse Workspaces bereitgestellt. Dieser enthält alle zum System gehörenden Plugins.
Zusätzlich enthält die CD noch diese Arbeit in elektronischer Form im Word und im PDF-Format.