Universität Freiburg (Schweiz)

Departement für Informatik Rue Faucigny 2 CH–1700 Fribourg

Seminararbeit in Informatik (3. Studienjahr)

Jini™ und JavaSpaces™

Jutta Langel
jutta.langel@unifr.ch

Verantwortlicher: Prof. Jacques Pasquier–Rocha
Assistent: Patrik Fuhrer

September 2002

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einführung

1.1 Das Ziel dieser Arbeit

Distributed Programming wird in letzter Zeit immer beliebter. Im Rahmen dieser Seminararbeit wird die Netzwerktechnologie Jini™ vorgestellt. Dabei werden zwei Ziele verfolgt:

Zum einen soll eine Einführung in Jini geboten werden. Dieses Dokument vereinigt und greift das Wesentliche aus den momentan erhältlichen Büchern und Artikeln über Jini heraus (Abschnitt 2.1). Der Leser soll einen knappen, aber ausreichenden Überblick bekommen und kann bei Detailfragen die im Literaturverzeichnis 5 aufgeführten Titel konsultieren. Ein Tutorial mit einfachen Beispielen stellt die Technologien Jini (Abschnitte 2.2 und 3.2) und JavaSpaces (Kapitel 3) vor.

Zum anderen wird in Kapitel 4 gezeigt, wie Netzwerkprogrammierung im Kontext einer parallelen Berechnung genutzt werden kann. Nachdem das Prinzip und die Funktionsweise der Jini–Technologie und der darauf aufbauenden JavaSpaces–Technologie erklärt wurde, bietet dieses Kapitel eine fortgeschrittene Anwendung der Technologien. Einer ersten Beschreibung der Anwendung im Abschnitt 4.1 folgen detaillierte Ausführungen zum Quellcode (Abschnitt 4.2). Das Kapitel schliesst mit dem Abschnitt 4.3, in dem die Performanz der gezeigten parallelen Anwendung analysiert wird.

Jini bietet die Möglichkeit, verteilte Software in der Java™–Programmiersprache zu schreiben. Deshalb sollte der Leser darin Kenntnisse haben, oder zumindest in einer objekt–orientierten Programmiersprache. Diese Arbeit baut auf den Basis-Ideen des *Distributed Programming* auf. Eine Vorstellung von der Funktionsweise von RMI sind von grosser Hilfe. Dazu ist das Buch [Mahmoud00] sehr zu empfehlen.

1.2 Notationen und Konventionen

Dieses Dokument ist in Kapitel aufgeteilt. Kapitel sind in Abschnitte und Unterabschnitte untergliedert.

In jedem Kapitel sind die Abbildungen durchnummeriert. Mit *Abbildung i.j* wird auf eine Abbildung j im Kapitel i Bezug genommen. Die Fussnoten sind innerhalb eines Kapitels durchgehend durchnummeriert, mit jedem Kapitel wird wieder bei eins angefangen. Wird auf ein Werk aus dem Literaturverzeichnis verwiesen, so geschieht das durch die

Nennung des Namen des Autors gefolgt von dem Jahr der Publikation des Werkes. Bei mehreren Autoren werden alle genannt. Webseiten sind mit Monat und Jahr ihrer Konsultierung versehen. Das Literaturverzeichnis ist alphabetisch geordnet, nach den Namen der Autoren. Anschliessend finden sich die Webseiten.

URLs, Erweiterungen von Dateinamen, UNIX–Befehle, Klassennamen und Variablen, sowie Java–Quellcode sind in `Schreibmaschinenschrift` angeführt.

```
Java-Quellcode und UNIX-Befehle erscheinen zudem immer eingerückt.
UNIX-Befehle, die in diesem Dokument mehrzeilig abgebildet sind,
müssen trotzdem einzeilig ausgeführt werden.
```

Java™, Jini™ und JavaSpaces™ sind eingetragene Marken von Sun Microsystems, Inc.

1.3 Verwendete Hardware und Software

1.3.1 Hardware

Sämtliche hier vorgestellte Programme wurden in einem Solaris–Netzwerk (Geschwindigkeit: 100 MBits/Sekunde, full–duplex) ausgeführt. Es besteht zur einen Hälfte aus UltraSPARC-IIi–Maschinen mit 440-MHz–Prozessoren und 128 MB RAM. Die andere Hälfte sind UltraSPARC-IIe–Maschinen, deren Prozessoren eine Frequenz von 500 MHz haben und die mit 512 MB RAM arbeiten.

Das erste Experiment der Performanz–Analyse (siehe Unterabschnitt 4.3.1) wurde mit den UltraSPARC-IIe–Maschinen begonnen, die ersten zehn Worker liefen auf diesen Maschinen. Ab dem elften Worker kamen die UltraSPARC-IIi–Maschinen hinzu. Für das zweite Experiment (siehe Unterabschnitt 4.3.2) wurden nur UltraSPARC-IIi–Maschinen verwendet.

1.3.2 Software

Es wurde mit Java in der Version 1.3.1_01 und Jini in der Version 1.2 gearbeitet.

Dieses Dokument wurde mit LaTeX (Version 3.14159) gesetzt. Die Umsetzung als `PDF`–Datei geschah mit `pdflatex`. Dank `LaTeX2HTML` (Version 2K.1beta) konnte aus dem LaTeX–Code eine `HTML`–Repräsentation dieses Dokuments geschaffen werden. Für die Diagramme des Abschnitts 4.3 wurde Excel97 benutzt, alle übrigen Diagramme wurden, falls nicht anders angegeben, mit Dia[1] in der Version 0.88.1 erstellt. Die Screenshots in Anhang A wurden mit XV in der Version 3.10a erstellt.

1.4 Jini™– eine Netzwerktechnologie

Jini™ist eine Netzwerktechnologie, die es – basierend auf der Java™–Technologie – ermöglicht, Geräte untereinander zu verknüpfen. In einem Jini–Netzwerk, ist alles ein Dienst (Service). Hardware und Software werden demzufolge lediglich als zwei verschiedene Weisen, einen Service zu implementieren, angesehen. Dies ist eines der Hauptmerkmale

[1] weitere Informationen zu Dia unter `http://www.lysator.liu.se/~alla/dia`

von Jini. Damit die Services interagieren können, stellt die Jini–Technologie einfache Mechanismen zur Verfügung, die es erlauben, aus Computern, Peripheriegeräten und Software vernetzte Gemeinschaften – auch Service–Communitys genannt – aufzubauen.

Da Jini sowohl Hardware wie auch Software als Services behandelt, vereinfacht sich ihre Bereitstellung innerhalb eines Netzwerkes sehr. Unter den Begriff „Services“ fallen alle Nutzungsmöglichkeiten von Applikationen, Datenbanken, Betriebssystemen, Servern, mobilen Geräten, Druckern, Massenspeichern, Handheld–Geräten sowie zahllose andere Services, auf die über das Netzwerk zugegriffen werden kann.

Nicht nur die Unterscheidung zwischen Hardware und Software wird von Jini aufgehoben. Da Jini auf der Java–Technologie basiert, ist die Plattform–Unabhängigkeit gewährleistet. Die von Jini genutzte Infrastruktur kann sowohl drahtgebunden als auch drahtlos sein und zur Kommunikation kann ein beliebiges Protokoll gewählt werden. Aus nahe liegenden Gründen benutzt Jini standardmässig Java Remote Method Invocation (RMI), um ausführbaren Code als Objekt über das Netzwerk zu verschicken. RMI basiert auf mobilen Objekten, die Interfaces implementieren, die auch anderen Objekten bekannt sind. So kann eine Kommunikation mit Objekten, die auf anderen Maschinen beherbergt sind, stattfinden. Jini erweitert die Idee hinter RMI insofern, als ein Ort zur Verfügung gestellt wird, an dem die Objekte ihre Services anbieten oder andere Services finden können.

Kapitel 2

Jini

2.1 Organisation einer Service–Community

Die Service–Community ist eine dynamische Gemeinschaft von Hardware und Software. Um miteinander kommunizieren zu können, müssen diese Services bei einem Auskunftsdienst (Lookup–Service) registriert sein. Der Lookup–Service ist ein Verzeichnis aller registrierten Dienste, die somit bekannt geben, dass sie über das Netzwerk erreichbar sind. Wenn sich ein Gerät auf das Netzwerk aufschaltet, sucht es zu aller erst den Lookup–Service. Dieser Vorgang nennt sich „Discovery“. Hat es den Lookup–Service gefunden, so meldet es sich bei ihm an und teilt diesem die von ihm zur Verfügung gestellten Dienste mit („Joining“). Dieser Vorgang läuft vollkommen automatisch ab und bedarf keines menschlichen Eingreifens, was ein weiteres Hauptmerkmal von Jini ist. Der Lookup–Service ist gleichzeitig auch eine Vermittlungsstelle für die Dienste, die in der Service–Community erhältlich sind. Will ein Teilnehmer einen Dienst anfordern, so wendet er sich an den Lookup–Service und fragt nach einem Service, der das gewünschte Interface implementiert. Der Lookup–Service findet den richtigen Service heraus und die beiden Teilnehmer werden dann über RMI untereinander vermittelt. Im Folgenden wird dieser Prozess detaillierter, und anhand eines einfachen Beispiels erklärt. Schon hier sei auf die Abbildung 2.2 auf Seite 22 hingewiesen, die zur Veranschaulichung dieses Ablaufs zu Rate gezogen werden kann.

2.1.1 Discovery

Unsere Jini–Community ist im allereinfachsten Zustand: Genau eine Instanz des Lookup–Service befindet sich im Netzwerk. Ob dieser auf einem PC oder einem IBM Grosscomputer läuft, ist egal. Nun schalten wir dem Netzwerk einen jini–tauglichen Drucker hinzu, der seinen Dienst anbieten will. Um dem Netzwerk als Dienstleister beizutreten, muss der Drucker den Lookup–Service finden. Dazu benutzt er ein Discovery–Protokoll.

Es gibt zwei verschiedene Discovery–Methoden, Unicast–Discovery und Multicast–Discovery.

2.1.1.1 Unicast–Discovery

Ein Service, der schon den „Aufenthaltsort“ (d. h. die Adresse des Hosts und die Nummer des Ports) des Lookup–Services kennt und Kontakt mit diesem aufnehmen möchten, benutzt Unicast–Discovery. Das Unicast–Discovery–Protokoll basiert auf der URL des zu kontaktierenden Services. Daher ist das Unicast–Discovery–Protokoll eher ein Wiederverbindungsprotokoll als ein Protokoll, das einen Service ausfindig macht.

2.1.1.2 Multicast–Discovery

Bei Multicast–Discovery versendet der suchende Service eine Anfrage, die durch das Multicast–Request–Protokoll festgelegt ist. Alle Lookup–Services, die sich im angegebenen Multicast–Radius befinden, werden auf diese Weise im aktuellen Jini–Netzwerk entdeckt. Der suchende Service kann Gruppen angeben, denen er sich anschliessen und deren Lookup–Services er deshalb per Multicast–Discovery finden möchte. Gruppen bieten eine logische Unterteilung des Jini–Netzwerks und dienen zu Discovery–Zwecken. Nach erfolgreicher Multicast–Discovery erhält der Service – unser Drucker – von jedem entdeckten Lookup–Service eine Referenz, um seinen jeweiligen Gruppen beizutreten. Diese Lookup–Service–Referenz ist ein Java–Proxy–Objekt, das auch „registrar“ genannt wird. Es führt sich selbständig in der Java Virtual Machine des Druckers aus und teilt diesem auf diese Weise alle wichtigen Informationen über den Aufenthaltsort (Host und Port) des Lookup–Services mit.

2.1.2 Joining

Jetzt hat unser Drucker mit Hilfe einer der beiden beschriebenen Discovery–Methoden die Lage des Lookup–Services festgestellt. Die Registrierung („Joining“) beim Lookup–Service besteht aus der Übersendung eines Proxy–Objektes an den Lookup–Service. Dieses Proxy–Objekt gibt Auskunft über die Eigenschaften des Service, den der Drucker anbietet. Ein Proxy–Objekt ist ein Beauftragter oder Bevollmächtigter, der weiss, wie die vom Service angebotene Arbeit ausgeführt werden muss und kann auch, wenn die auszuführende Arbeit es erfordert, eine Rückverbindung mit dem Service selbst herstellen.

2.1.3 Clients

Nun fügen wir dem Netzwerk in diesem Zustand – ein Lookup–Service, bei dem ein Drucker seinen Dienst registriert hat – einen Laptop mit einem Jini–tauglichen Textverarbeitungsprogramm hinzu. Das Textverarbeitungsprogramm stellt im Netzwerk einen Client dar, der bei einem Druckbefehl den von unserem Drucker angebotenen Service benötigt. Um herauszufinden, ob mindestens ein solcher Service im Netzwerk existiert, muss der Client zuerst Kontakt zum Lookup–Service herstellen. Das geschieht wieder auf der Basis einer der zuvor angeführten Discovery–Methoden.

2.1.4 Service–Proxys

Nachdem der Client dem Lookup–Service mitgeteilt hat, welche Art von Service er benötigt, und falls beim Lookup–Service ein solcher Service registriert ist, erhält der Client eine Kopie des entsprechenden Proxy–Objekts. Dabei handelt es sich um ein selbständiges

Java–Objekt, das in der Java Virtual Machine des Client instanziert wird. Die Informationen über die Java–Klassen, die benötigt werden, um das Proxy–Objekt vollkommen handlungsfähig zu machen, werden dynamisch über das Netzwerk geladen. Dieses System ist sehr mächtig, da der Client lediglich mit einem lokalen Proxy–Objekt interagiert, in dem festgelegt ist, wo die Dienstleistung ausgeführt werden soll. Dafür gibt es vier Möglichkeiten:

1. Der Proxy agiert lokal, in der Virtual Machine des Clients. In diesem Fall bietet das Proxy–Objekt selbst den Service an.

2. Der Proxy stellt eine Verbindung zu dem Computer her, der den Service beherbergt. Dort wird die Dienstleistung vollständig ausgeführt.

3. Der Proxy kontaktiert über das Netzwerk eine andere, für die Ausführung des Service besonders geeignete Maschine, die zwischen der Client–VM und dem remoten Dienstleistungsanbieter steht.

4. Der Proxy veranlasst, dass ein Teil der Dienstleistung lokal in der Virtual Machine des Clients ausgeführt wird. Der verbleibende Teil der auszuführenden Aufgabe wird auf eine entfernte Maschine verlagert, entweder auf die des Dienstleistungsanbieters selbst oder auf eine andere. Diese Implementierungsart wird „Smart Proxy“ genannt.

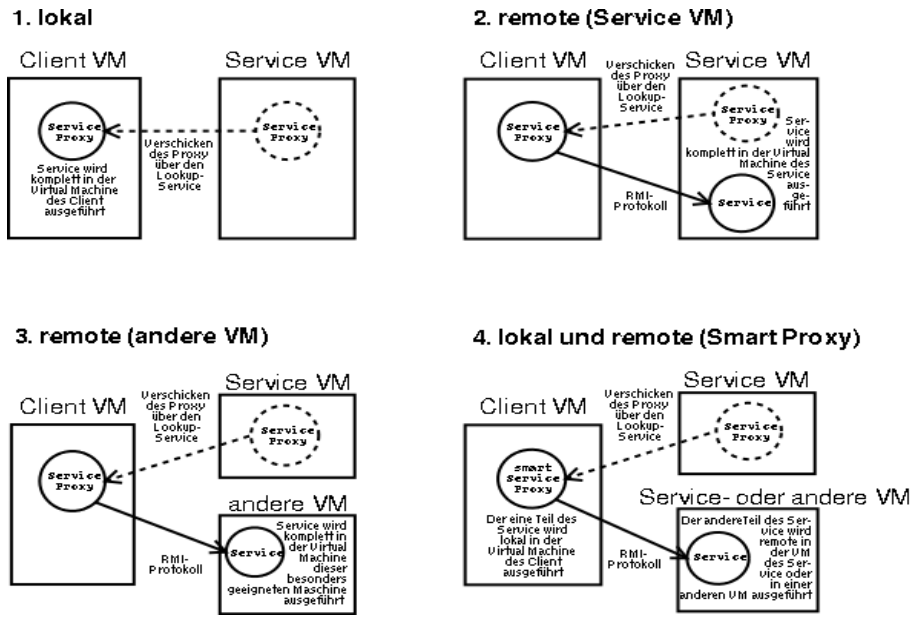

Abbildung 2.1: Vier Möglichkeiten, wie ein Proxy agieren kann

Für jegliche Art der Netzwerk–Kommunikation ist das Proxy–Objekt zuständig. Der Client braucht sich um nichts zu kümmern. Im Fall unseres Druck–Service muss der Vorgang des Ausdruckens natürlich auf einem Drucker stattfinden. Die dafür benötigte Rechenleistung kann aber an einem anderen Ort ausgeführt werden.

2.1.5 Leasing

Ein Client muss nicht immer mit dem gleichen Dienstleister zusammenarbeiten. Bei der nochmaligen Ausführung eines Druckbefehls kann das Textverarbeitungsprogramm den Dienst eines anderen Druckers in Anspruch nehmen, der zwischenzeitlich der Jini–Community beigetreten ist und bessere Charakteristika vorweisen kann, z. B. näher am Standort des Laptop gelegen. Bisweilen ist es sogar notwendig während der Ausführung eines Services den Anbieter zu wechseln, wenn beispielsweise der Anfangsdienstleister gecrasht ist. Dies ist möglich, da eine Jini–Community selbstheilend ist – ein weiteres Hauptmerkmal der Jini–Technologie:

Wie im Unterabschnitt 2.1.2 beschrieben, muss ein Dienstleister seinen angebotenen Service beim Lookup–Service eintragen. Der Lookup–Service hält diese Registrierung für einen bestimmte Zeitspanne – der Leasing–Zeit – aufrecht. Auch wenn ein Client eine Anfrage an den Lookup–Service stellt, wird eine Leasing–Zeit etabliert. Client, bzw. Service müssen sich in regelmässigen Abständen beim Lookup–Service zurückmelden, um zu signalisieren, dass sie noch existent sind und um die Registrierung aufrecht zu erhalten. Tritt bei einem Service ein Software-, Hardware-, oder Netzwerk–Fehler auf, so ist es ihm unmöglich, sich zurückzumelden. Seine Registrierung läuft aus. Nun weiss der Lookup–Service, dass hinter dem dazugehörigen Proxy–Objekt ein nicht dienstbereiter Service steht. So wird potentiellen Clients dieses Proxy–Objekt nicht mehr vermittelt werden und der dahinter stehende Service ist nicht mehr Mitglied des Jini–Netzwerkes. Trotzdem kann er später wieder beitreten, wenn er wieder voll hergestellt ist. Die angefangene Aufgabe – das kann z. B. ein Service sein, der aus mehreren einzelnen Services besteht – konnte also nicht fertig gestellt werden, nur k der n Services sind vollständig ausgeführt, $k < n$. Dieses teilweise Versagen kann zu nicht determinierten Systemzuständen führen, die auf einige Anwendungen desaströse Auswirkungen haben können. Für eine korrekte Fortführung der Aufgabe bei einem anderen Dienstleister, muss in den letzten stabilen Status vor dem Abbruch zurückgekehrt werden. Jini stellt deshalb einen Transaktionen–Koordinator zur Verfügung. Dieser bietet lediglich das Management in solchen Situationen an, die Implementierungsdetails müssen von den jeweiligen Services übernommen werden.

Nicht nur Services und Clients müssen sich beim Lookup–Service zurückmelden. Auch ein Lookup–Service muss in regelmässigen Abständen die anderen Services von seiner Existenz informieren. Basierend auf dem Multicast–Announcement–Protokoll versendet ein Lookup–Service Pakete, um seine Lage bekannt zu geben. Bei Versagen des Multicast–Request–Protokolls, können Services und Clients immer noch über die sie umgebenden Lookup–Services informiert werden.

2.1.6 Synchronisation

Da Clients und Services im Normalfall auf ganz unterschiedlichen Maschinen laufen, wissen sie nichts voneinander und es gibt somit auch keine implizite Synchronisation zwischen den beiden Parteien. Es kann also passieren, dass ein Client nach einem Service sucht, und die Suche fehlschlägt, da kein passender Service vorhanden ist. Also verlässt der Client das Jini–Netzwerk und prompt registriert sich der gesuchte Service beim Lookup–Service ein. Um dem vorzubeugen, bietet Jini einen remoten Ereignis–Mechanismus (remote event mechanism) an: Der Client trägt seine Interessen beim Lookup–Service ein. Dieser Eintrag ist von begrenzter Dauer. Während dieser Zeitspanne wird der Client informiert, falls ein

von ihm gewünschter Service später ins Netzwerk hinzukommt. Dies verhindert ein "busy waiting" des Clients.

2.1.7 Service–Upgrades

Eine neue Version eines Services kann sich bei allen Lookup–Services eintragen, während die alte Version noch in Kraft ist und agiert. Die Lookup–Services werde also von neuen Proxy–Objekten dieses upgegradeten Services besiedelt. Ein Client, der jetzt diesen Service sucht, wird die neueste Proxy–Version mit der neuen Implementierung des Services bekommen. Nichtsdestotrotz haben Clients, die noch die alten Proxy-Objekte haben, Zugriff auf die vorherige Implementierung des Services. Mit der Zeit entsorgen die Clients ihre alten Proxy–Objekte, weil sie sie nicht mehr benötigen, so dass zu einem gegebenen Zeitpunkt jeder Client, der diesen Service benutzt, über die neue Version verfügt. Irgendwann wird dann die alte Implementierung dieses Services aus dem Netzwerk genommen. Sollte trotzdem noch ein Client einen alten Proxy haben und versuchen, auf ihn zuzugreifen, so wird der eine Exception erhalten und muss noch einmal ein Lookup starten.

2.2 Tutorial

2.2.1 Installation

Dieser Abschnitt zeigt, wie eine Arbeitsumgebung eingerichtet werden muss, damit die Jini–Services, die mit der Jini–Technologie von Sun zur Verfügung gestellt werden, funktionieren. Um Jini benutzen zu können, wird die Java 2 Standard Edition – Plattform benötigt, da Jini auf RMI und einige Java 2 – Klassen basiert ist, die erst mit Java 2 eingeführt wurden.

Nach der Registratur und der Akzeptanz der SCSL (Sun Community Source License) kann Jini unter der URL

```
http://www.sun.com/communitysource/jini/download.html
```

kostenlos heruntergeladen werden. Für eine funktionstüchtige Jini–Umgebung ist das „Jini Technology Starter Kit" notwendig. Es umfasst die Bibliotheken der Jini–Klassen, die Jini–Spezifikationen, die Services, die zur Infrastruktur der Jini–Technologie gehören – einschliesslich der JavaSpaces^{TM}–Technology – sowie die `javadoc`–Dokumentation der Jini–Klassen und Beispielcode. Der Quellcode dieser Arbeit wurde unter der Version 1.2 entwickelt. Die momentan aktuellste Version des Starter Kit ist 1.2.1. Die herunterladbare Datei für das Jini Technology Starter Kit heisst `jini-1_2_1-src.zip` und entpackt sich in das Verzeichnis `jini-1_2_1`. Es ist eine einfache gezippte Datei, die mit den meisten unzipping–tools oder mit dem Befehl

```
jar xvf jini-1_2_1-src.zip
```

entpackt werden kann. Im Verzeichnis `jini-1_2_1` befinden sich sodann folgende Dateien und Unterverzeichnisse:

`index.html` Dies ist die Startseite der gesamten Dokumentation der Jini–Installation. Sie bietet Links zu allen wichtigen Informationsquellen innerhalb des Verzeichnis `jini-1_2_1`.

`doc/` In diesem Unterverzeichnis sind die `javadoc`–Dokumentation, Erklärungen zu den mitgelieferten Beispielen, Lizenz–Dateien und Release–Angaben zu finden.

`example/` Hier liegen die Dateien, die benötigt werden, um den Beispiel–Code auszuführen.

`lib/` Dieses Unterverzeichnis beherbergt alle `jar`–Dateien.

`policy/` Die Jini–Services, die mit dem Starter Kit geliefert werden, benutzen die policy–Dateien aus diesem Unterverzeichnis.

`source/` Der Quellcode, aus dem die binären Klassen erzeugt werden, befindet sich hier.

Jini ist an der Universität Freiburg (Schweiz) unter Solaris im Verzeichnis `/usr/local/jini` installiert[1]. Die folgenden Beispiele beziehen sich immer auf diese Installation. Für das Windows–Beispiel wird angenommen, dass Jini im Verzeichnis `C:\files\jini` liegt.

2.2.2 Einrichten der Arbeitsumgebung – der Classpath

Das Jini Technology Starter Kit besteht aus 3 Teilen, zu denen jeweils eine `jar`–Datei aus dem lib/–Unterverzeichnis gehört: `jini-core.jar` (Grundausstattung der Jini–Technologie), `jini-ext.jar` (Erweiterung der Jini–Technologie, inklusive JavaSpaces) und `sun-util.jar` (zusätzliche Dienstprogramme). Um korrekt mit der Jini–Technologie programmieren zu können, müssen diese drei Dateien im Classpath stehen. Dafür gibt es zwei Methoden: Entweder wird beim Aufruf der Java–Applikation der Classpath mit Hilfe des `-cp`–Parameters übergeben oder die CLASSPATH–Umgebungsvariable wird entsprechend gesetzt. Im Weiteren wird immer die erste Methode angewandt werden.

2.2.3 Sicherheit – die `policy`–Datei

Jeder Jini–Service hat eine `java.security.policy`–Eigenschaft, die über die angegebene `policy`–Datei festgelegt wird. Eine `policy`–Datei dient zu Sicherheitszwecken, denn in ihr steht, welche Rechte einem Programm bei seiner Ausführung zustehen. Da ein Client in einem Jini–Netzwerk Proxys von Services erhält, die bisweilen in der Java Virtual Machine des Clients Programme ausführen, muss der Client deren Rechte festlegen, um sich vor eventuellem Missbrauch zu schützen. Im policy–Unterverzeichnis sind die `policy`–Dateien zu finden, die die Services des Starter Kit benutzen. Jedoch muss je nach Netzwerkumgebung eine daran angepasste `policy`–Datei geschrieben werden. Das folgende Beispiel zeigt den Inhalt einer `policy`–Datei, die einem Programm alle Rechte gibt. Es ist sehr gefährlich, eine solche `policy.all`–Datei in einem unsicheren Netzwerk zu verwenden, für Test–Zwecke ist sie jedoch sehr nützlich.

```
grant {
  permission java.security.AllPermission "", "";
};
```

[1] Stand: August 2002

2.2.4 Jini–Framework

Suns Implementierung der Jini–Technologie bringt eine Basis-Ausstattung von Services mit sich, die man das Jini–Framework nennt. Darin sind enthalten:

reggie	ein Lookup–Service
mahalo	ein Transaktionen–Manager–Service
outrigger	zwei JavaSpaces–Services
fiddler	ein Lookup–Discovery–Service
norm	ein Service, der die Leasing–Zeit erneuert
mercury	ein Event–Mailbox–Service

Für die Beispiele im Abschnitt 2.3 müssen lediglich ein HTTP–Server, der RMI–Aktivierungsdämon *rmid* (siehe Unterabschnitt 2.2.6), und der Lookup–Service *reggie* in dieser Reihenfolge gestartet werden. Für die JavaSpaces–Technologie aus Kapitel 3 und 4 wird zusätzlich noch *mahalo* und *outrigger* benötigt. Das Starten dieser Service wird nun erklärt werden. Dabei ist es wichtig zu wissen, dass es ratsam ist, in einer Test–Umgebung eventuelle Log–Dateien immer zu entfernen. In einer Produktionsumgebung hingegen wünscht man Effekt des Neustarts am dem Punkt, an dem die Maschine gecrasht ist.

2.2.5 Starten des HTTP–Service

Wenn ein Client einen Service benötigt, wird ihm das entsprechende Proxy–Objekt zugesandt. In der von Sun zur Verfügung gestellten Jini–Implementierung sind diese Proxys RMI–Stubs. Sie stellen die Rückverbindung mit dem Service her. Damit RMI korrekt funktioniert, wird ein Mechanismus benötigt, der es ermöglicht, dem Client diejenigen `class`–Dateien zu liefern, die er nicht in seinem Classpath hat, aber die für die Ausführung des Programms erforderlich sind. Dies geschieht mittels eines Web–Servers, auf den die benötigten Dateien gelegt werden. Die `java.rmi.server.codebase`–Eigenschaft muss entsprechend gesetzt werden, um dem Client mitzuteilen, wo er die `class`–Dateien erhalten kann.

Um die Grundausstattung an Jini–Services zu benutzen, müssen Clients `jar`–Dateien herunterladen, die im `lib`–Verzeichnis der Jini–Installation zu finden sind. Für den Download dieser Dateien wird ein HTTP–Service gebraucht.

Ein einfacher, in Java geschriebener – und daher plattformunabhängiger – HTTP–Server ist Teil des Starter Kit. Um ihn auf dem Port 8080 zu starten, genügt dieses Kommando:
Unix:

```
java -jar /usr/local/jini/lib/tools.jar -port 8080 -dir /usr/local/jini/lib -verbose
```

Windows:[2]

```
java -jar C:\files\jini\lib\tools.jar -port 8080 -dir C:\files\jini\lib -verbose
```

[2]Von nun an wird nur noch die Unix–Version angegeben werden, da es offensichtlich ist, dass lediglich die Repräsentation des Pfades eine andere ist.

2.2.6 Starten des RMI–Aktivierungsdämon *rmid*

Reggie, *mahalo*, *outrigger*, *fiddler*, *mercury* und *norm* benötigen allesamt den RMI–Aktivierungsdämon. Auf einer Maschine, auf der einer dieser Services läuft, muss auch der RMI–Aktivierungsdämon *rmid* gestartet worden sein:

```
rmid -J-Djava.security.policy=/usr/local/jini/policy/policy.all -log ~/rmid_log
```

Wenn der Dämon gestartet wird, legt er das unter `log` angegebenen Verzeichnis an, wohin er seine Log–Dateien schreibt. Ist dieser Parameter nicht gegeben, legt der Dämon im aktuellen Verzeichnis ein Verzeichnis mit dem Namen log an. Wird der Dämon von Neuem gestartet, so liest er alle Informationen aus dem Log–Verzeichnis, um im alten Zustand weiterarbeiten zu können. Dabei startet er selbständig auch alle Services, die sich zuvor bei ihm registriert haben, von Neuem.

Reggie ist z. B. ein Service, der sich bei Aktivierungsdämon registrieren kann. Nur aktivierbare Services können sich beim RMI–Aktivierungsdämon registrieren, sie laufen von da an unter der Kontrolle von *rmid*. Dies macht sie persistent, fortdauernd im Sinne des automatischen Wiederstarts im Fall eines Crashs. Dank der oben erklärten Log–Eigenschaft, genügt es, bei einem Maschinen–Crash *rmid* wieder zu starten, was den Wiederstart aller dort registrierten aktivierbaren Services mit sich bringt.

Um einen aktivierbaren Service zu töten, muss *rmid* getötet werden, sowie alle aktivierbaren Services, das Log–Verzeichnis des RMI–Aktivierungsdämon muss entfernt werden, eventuell auch die Log–Verzeichnisse der aktivierbaren Services. Dann erst kann *rmid* neu gestartet werden – diesmal werden keine Services automatisch gestartet, da das Log–Verzeichnis keine Informationen beinhaltet – und die übrigen aktivierbaren Services, die man ursprünglich behalten wollte. Der RMI–Aktivierungsdämon läuft bis er getötet wird, daher ist es ratsam, ihn in einer eigenen Shell laufen zu lassen.

Um zu verhindern, dass für jeden Service, der sich beim Aktivierungsdämon registriert eine eigene Java Virtual Machine instanziert wird, empfiehlt es sich, nach dem Start des Dämon eine gemeinsame Virtual Machine zur Verfügung zu stellen. Wie dies funktioniert wird im Unterabschnitt 2.2.7 erklärt.

2.2.7 Starten der Shared Java Virtual Machine

Um nicht für jeden beim RMI–Aktivierungsdämon registrierten Service eine Instanz der Java Virtual Machine zu haben, stellt man eine einzige gemeinsame Virtual Machine zur Verfügung, die sich alle Services teilen:

```
java -jar /usr/local/jini/lib/create.jar http://HTTPServerHostname:8080/create-dl.jar
/usr/local/jini/policy/policy.all ~/sjvm_log
-Djava.security.policy=/usr/local/jini/policy/policy.all -verbose
```

2.2.8 Starten des Jini Lookup–Service *reggie*

Der Lookup–Service kann mit folgender (einzeiligen!) Kommandozeile gestartet werden:

```
java -Djava.security.policy=/usr/local/jini/policy/policy.all
-jar /usr/local/jini/lib/reggie.jar http://HTTPServerHostname:8080/reggie-dl.jar
/usr/local/jini/policy/policy.all ~/reggie_log public
```

Das erste Argument, die `-D`–Option, ist die `policy`–Datei, unter der dieser Service beim *rmid* registriert wird. Das folgende Argument, in der `-jar`–Option zu finden, gibt an, in welche `jar`–Datei *reggie* gepackt wurde. Wenn Clients den Lookup–Service kontaktieren, erhalten sie von ihm einen Proxy. Diese `jar`–Datei ist über den HTTP–Server, den wir zuvor gestartet haben, zugänglich. `HTTPServerHostname` im dritten Argument muss also durch den Namen des Computers, auf dem der HTTP–Server läuft, ersetzt werden. Der vierte Parameter bezeichnet die `policy`–Datei mit der das Kommando selbst ausgeführt wird. Als nächstes Argument wird das Log–Verzeichnis angegeben, dorthin wird *reggie* seine Log–Dateien schreiben und beim Wiederstart aus diesem Verzeichnis lesen. Schliesslich wird als letztes Argument die Gruppe bestimmt, für die dieser Lookup–Service zuständig ist. Es können auch mehrere Gruppen sein.

2.2.9 Starten des Transaktionen–Manager–Service *mahalo*

Dieser Service wird von anderen Services (wie z. B. dem JavaSpaces–Service) benutzt, die viele einzelne Operationen wie eine einzige atomare Operation behandeln wollen. Wie auch *reggie* ist *mahalo* ein aktivierbarer Service und wird auf die gleiche Weise aufgerufen. Im folgenden Beispiel wurde dem Service als zweites Argument lediglich noch der Name „TransactionManager“ mit auf den Weg gegeben. Alle anderen Argumente entsprechen in ihren Funktionen denen des Lookup–Service.

```
java -Djava.security.policy=/usr/local/jini/policy/policy.all
-Dcom.sun.jini.mahalo.managerName=TransactionManager
-jar /usr/local/jini/lib/mahalo.jar http://HTTPServerHostname:8080/mahalo-dl.jar
/usr/local/jini/policy/policy.all ~/mahalo_log public
```

2.2.10 Starten des JavaSpaces–Service *outrigger*

Es gibt zwei verschieden JavaSpaces–Services, den `TransientSpace` und den `FrontEndSpace`.

2.2.10.1 TransientSpace

Der `TransientSpace` ist ein „vergänglicher“ Service. Bei einem Crash bleiben die Daten nicht erhalten und der Neustart beginnt bei Null. Dieser JavaSpaces–Service ist kein aktivierbarer Service, daher nimmt er als Parameter lediglich eine `policy`–Datei und bekommt die Download–Adresse über die `java.rmi.server.codebase`–Eigenschaft übergeben. Mit dem dritten Argument wird der Name des Service angegeben, der `-jar`–Parameter gibt die Datei an, in der der `TransientSpace` zu finden ist und zum Schluss folgt optional die Gruppe. Argumente wie das Log–Verzeichnis sind nicht notwendig.

```
java -Djava.security.policy=/usr/local/jini/policy/policy.all
-Djava.rmi.server.codebase=http://HTTPServerHostname:8080/outrigger-dl.jar
-Dcom.sun.jini.outrigger.spaceName=JavaSpaces
-jar /usr/local/jini/lib/transient-outrigger.jar public
```

2.2.10.2 FrontEndSpace

Der `FrontEndSpace` hingegen ist ein fortdauernder Service, der im Falle eines Crash den Neustart mit den gespeicherten Daten beginnt. Deshalb entsprechen die Funktionen seiner Argumente exakt denen des TransactionManager *mahalo*.

```
java -Djava.security.policy=/usr/local/jini/policy/policy.all
-Dcom.sun.jini.outrigger.spaceName=JavaSpaces  -jar /usr/local/jini/lib/outrigger.jar
http://HTTPServerHostname:8080/outrigger-dl.jar
/usr/local/jini/policy/policy.all ~/outrigger_log public
```

2.2.11 Starten des Jini–Services–Browser

Zur Visualisierung der in der Jini–Community vorhanden Services existiert ein Tool, das wie folgt gestartet werden kann:

```
java -cp .:/usr/local/jini/lib/jini-examples.jar
-Djava.security.policy=/usr/local/jini/policy/policy.all
-Djava.rmi.server.codebase=http://HTTPServerHostname:8080/jini-examples-dl.jar
com.sun.jini.example.browser.Browser -admin
```

2.3 Ein einfaches Beispiel: ein Addierer

Wie in 2.1.4 zu sehen war, gibt es verschiedene Arten von Proxys. Hinter jedem Proxy steht eine andere Implementierung des entsprechenden Service. Die ersten beiden dieser Service–Architekturen sollen hier vorgestellt werden.

Ein Client sucht einen Service, der zwei Zahlen miteinander addieren kann und das Ergebnis zurück liefert. Eine Möglichkeit ist, dass die dafür notwendige Arbeit lokal in der Virtual Machine des Clients ausgeführt wird, eine andere, dass dafür die Virtual Machine des Service–Anbieters benutzt wird.

Das folgende Beispiel ist so konstruiert, dass für beide Service–Implementierungen ein eigenes Verzeichnis vorgesehen ist: `ser` für die lokale Version und `rem` für die Variante, bei der die Arbeit remote verrichtet wird. Die Namen der Verzeichnisse stimmen mit den Namen der packages überein. Beide Verzeichnisse sind nach der gleichen Struktur aufgebaut: Im Arbeitsverzeichnis befinden sich die drei Dateien mit dem Java–Quellcode: `AdderClient.java`, `AdderServiceInterface.java` und `AdderServiceImpl.java`. Dort liegen auch die Skripte, die den HTTP–Server starten, sowie die Skripte, die der Kompilierung und Ausführung der Programme dienen. Des weiteren gibt es ein Verzeichnis `client/` für die ausführbare Anwendung des Client und ein Verzeichnis `service/` gleichen Zwecks für den Service. Beim Kompilieren des Clients werden die erzeugten `class`–Dateien in das Verzeichnis `./client/name_des_packages/`[3] geschrieben. Ebenso findet man die ausführbaren Dateien des Services nach seiner Kompilierung im Verzeichnis `./service/name_des_packages/`. Diese Aufteilung dient der strikten Trennung zwischen Service und Client. Sowohl Client als auch Service kennen das Interface, das zur Implementierung des Addierers führt – `AdderServiceInterface.java`. Beide benötigen es in ihren Programmen. Seine `class`–Datei muss sowohl dem Service als auch dem Client zur Verfügung stehen und ist daher in beiden Verzeichnissen zu finden.

[3] `./client/ser/` bzw. `./client/rem/`

Zur Erinnerung sei noch einmal gesagt, dass die in den Unterabschnitten 2.2.5, 2.2.6, 2.2.7 und 2.2.8 beschriebenen Services gestartet werden müssen, damit der Addierer funktioniert.

Der Client ist nicht von der jeweiligen Service–Implementierung berührt. Daher wird zuerst das Client–Programm präsentiert.

2.3.1 Der Client

Der Client sucht nach einem Service, der ein ihm bekanntes Interface implementiert – das AdderServiceInterface.

```
package ser;

import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.lookup.ServiceDiscoveryManager;
import java.rmi.*;
import java.io.*;

public class AdderClient implements Runnable {
  protected ServiceTemplate template;
  protected boolean stop = false;

  public AdderClient() throws IOException {
    Class[] types = { AdderServiceInterface.class };
    template = new ServiceTemplate(null, types, null);
    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new RMISecurityManager());
    }
    LookupDiscoveryManager ldm =
      new LookupDiscoveryManager(new String[] { "" }, null, null);
    ServiceDiscoveryManager sdm =
      new ServiceDiscoveryManager(ldm, null);
    try {
      ServiceItem o = sdm.lookup(template, null, 20000);
      if(o==null){
        System.out.println("A problem has occured.");
      } else {
        System.out.println("Got a matching service");
      }
      System.out.println("The adder says that 3 + 5 = "+
        ((AdderServiceInterface)(o.service)).add(3,5));
      stop = true;
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
```

```
    /* This thread keeps the VM from existing while discovery.*/
    public void run() {
      while (!stop) {
        try {
          Thread.sleep(1000);
        } catch (InterruptedException ex) {
          ex.printStackTrace();
        }
      }
    }
  }
```

Mittels des `ServiceTemplate` legt der Client fest, welchen Service er sucht, das `template` muss auf das AdderServiceInterface passen. Nachdem ein `RMISecurityManager` für den Download der Stubs installiert wurde, wird ein `LookupDiscoveryManager` instanziert, der nach einem Lookup–Service sucht. Dies passiert, indem er dem `ServiceDiscoveryManager` übergeben wird, der mit seiner Hilfe alle beim gefundenen Lookup–Service registrierten Services kennenlernt. Der `ServiceDiscoveryManager` sucht dann zwanzig Sekunden lang nach einem Service, der mit dem `template` übereinstimmt. Das gefundene Proxy–Objekt wird auf ein `ServiceItem` geschrieben, auf dem der gewünschte Service ausgeführt wird.

Nun wird noch ein Hauptprogramm hinzugefügt, um den AdderClient–Thread zu aktivieren:

```
  public static void main(String args[]) {
    try {
      AdderClient ac = new AdderClient();
      new Thread(ac).start();
    } catch (IOException ex) {
      System.out.println("Couldn't create client: " + ex.getMessage());
    }
  }
```

Dieser (einzeilige!) Befehl führt die für den AdderClient notwendige Kompilierung aus:

```
javac -classpath /usr/local/jini/lib/jini-core.jar:/usr/local/jini/lib/jini-ext.jar:
/usr/local/jini/lib/sun-util.jar  -d ./client AdderServiceInterface.java AdderClient.java
```

Wie auf Seite 16 erklärt, benötigt der Client die `class`–Datei des Interfaces. Deshalb wird `AdderServiceInterface.java` mitkompiliert. Das jeweilige Interface wird im Unterabschnitt 2.3.2 für die lokale Variante, bzw. im Unterabschnitt 2.3.3 für eine remote Implementierung, vorgestellt werden.

Bevor der Client gestartet werden kann, braucht er noch einen HTTP–Server für die Jini–interne Kommunikation. Die Portnummer wird 8086 sein:

```
java -jar /usr/local/jini/lib/tools.jar -port 8086 -dir /usr/local/jini/lib -verbose
```

Um den AdderClient zu starten, genügt die folgende Kommandozeile:

```
java -cp /usr/local/jini/lib/jini-core.jar:/usr/local/jini/lib/jini-ext.jar:
/usr/local/jini/lib/sun-util.jar:./client
 -Djava.rmi.server.codebase=http://HTTPServerHostname:8085/sdm-dl.jar
 -Djava.security.policy=/usr/local/jini/policy/policy.all AdderClient
```

2.3.2 Der Service und sein Interface – lokal

Empfängt der Client einen Proxy, der lokal agiert, so bietet das Proxy–Objekt selbst den Service an.

In diesem Fall handelt es sich um ein herkömmliches Interface, das eine Methode zur Addierung zweier Zahlen anbietet.

```
public interface AdderServiceInterface {
  public int add(int a, int b);
}
```

Diese Methode `add` wird von der Klasse `AdderServiceImpl` implementiert. Da der dieser Klasse entsprechende Proxy seine Arbeit vollständig in der Virtual Machine des Client verrichtet, genügt es, wenn `Serializable` implementiert wird.

```
package ser;

import net.jini.lookup.ServiceIDListener;
import net.jini.lookup.JoinManager;
import net.jini.discovery.LookupDiscoveryManager;
import java.rmi.*;
import java.io.*;

public class AdderServiceImpl implements Serializable, AdderServiceInterface {
  public AdderServiceImpl() {}
  public int add(int a, int b) {
    return (a+b);
  }
}
```

Um den Service jini-tauglich zu machen, wird ein Wrapper benötigt, der sich um die Veröffentlichung des Addierers kümmert:

```
class AdderServiceWrapper implements Runnable {

  public AdderServiceWrapper() throws IOException {
    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new RMISecurityManager());
    }
    LookupDiscoveryManager ldm =
      new LookupDiscoveryManager(new String[] { "" }, null, null);
    JoinManager dispatcher =
      new JoinManager(new AdderServiceImpl(), null, (ServiceIDListener)null , ldm, null);
    System.out.println("Object bound in the LookupService !");
  }
}
```

Aus Sicherheitsgründen muss zuerst ein `RMISecurityManager` installiert werden. Danach folgt, wie auch beim Client, ein `LookupDiscoveryManager`, der für den `JoinManager` arbeitet. Der `JoinManager` registriert eine Instanz unseres Service (`AdderServiceImpl`) beim Lookup–Service.

Damit der ausführende Thread während der Discovery nicht existiert, wird er zu Beginn erst einmal schlafen gelegt:

```
public void run() {
  while (true) {
    try {
      Thread.sleep(1000);
    } catch (InterruptedException ex){
      ex.printStackTrace();
    }
  }
}
```

Fehlt nur noch ein Hauptprogramm, das wie folgt aussehen könnte:

```
public static void main(String args[]) {
  try {
    AdderServiceWrapper asw = new AdderServiceWrapper();
    new Thread(asw).start();
  } catch (IOException ex) {
    System.out.println("Couldn't create adder: " + ex.getMessage());
  }
}
```

Kompilieren des Service, starten des dazugehörenden HTTP–Server und ausführen des Service geschieht mit diesen drei Kommandozeilen:

```
javac -classpath /usr/local/jini/lib/jini-core.jar:
/usr/local/jini/lib/jini-ext.jar:/usr/local/jini/lib/sun-util.jar
-d ./service AdderServiceInterface.java AdderServiceImpl.java

java -jar /usr/local/jini/lib/tools.jar -port 8085 -dir ./service -verbose

java -cp /usr/local/jini/lib/jini-core.jar:/usr/local/jini/lib/jini-ext.jar:
/usr/local/jini/lib/sun-util.jar:./service
-Djava.rmi.server.codebase=http://HTTPServerHostname:8085/
-Djava.security.policy=/usr/local/jini/policy/policy.all AdderServiceWrapper
```

2.3.3 Der Service und sein Interface – remote

Da ein RMI–Stub ein Java–Objekt ist, kann es direkt als Service–Proxy benutzt werden. Der Client erhält einen Proxy vom Lookup–Service. Dieser Proxy ist nichts anderes als ein RMI–Stub, das mit der Implementierung des Services, die auf der Virtual Machine des Dienstleisters läuft, kommuniziert. Das Service–Programm wird vollständig remote ausgeführt.

Deshalb muss das Service–Interface verändert werden, es muss die Eigenschaften eines Remote–Interface bekommen und `java.rmi.Remote` erweitern. Des weiteren muss die angebotene Methode eine `java.rmi.RemoteException` werfen.

```
public interface AdderServiceInterface extends java.rmi.Remote {
  public int add(int a, int b) throws java.rmi.RemoteException;
}
```

Die Klasse, die die `add`–Methode implementiert, hat sich entsprechend verändert. Wie bisher implementiert sie das `AdderServiceInterface`, jedoch nicht mehr `Serializable`. Nun wird `java.rmi.server.UnicastRemoteObject` erweitert.

```
package rem;

import net.jini.lookup.ServiceIDListener;
import net.jini.lookup.JoinManager;
import net.jini.discovery.LookupDiscoveryManager;
import java.rmi.*;
import java.io.*;

public class AdderServiceImpl extends java.rmi.server.UnicastRemoteObject
                              implements AdderServiceInterface {
  public AdderServiceImpl() throws java.rmi.RemoteException {}
  public int add(int a, int b) throws java.rmi.RemoteException {
    return (a+b);
  }
}
```

Die Wrapper–Klasse und folglich auch das Hauptprogramm sowie die `run`–Methode bleiben gleich, an der Veröffentlichung des Service ändert sich nichts.

Auch das Kompilieren läuft auf die gleiche Art und Weise wie beim Service, der lokal agiert, ab:

```
javac -classpath /usr/local/jini/lib/jini-core.jar:
/usr/local/jini/lib/jini-ext.jar:/usr/local/jini/lib/sun-util.jar
-d ./service AdderServiceInterface.java AdderServiceImpl.java
```

Nun gilt es, das RMI–Stub, das als Proxy dienen wird, zu erzeugen. Dazu wird der RMI–Compiler `rmic` benutzt:

```
rmic -v1.2 -classpath ./service -d ./service rem.AdderServiceImpl
```

Die `-v1.2`–Option gibt an, dass nur das Stub und nicht das Skeleton hergestellt werden soll. In früheren Versionen von Java war noch beides nötig, doch ab Version 1.2 genügt das Stub. Per `-classpath` wird der Pfad mitgeteilt , in dem die `class`–Dateien zu finden sind, die als Input dienen. Der `-d`–Parameter informiert über das Verzeichnis, in das die generierten Dateien zu schreiben sind. Zum Schluss steht das Objekt für das ein Stub erzeugt werden soll. Unser `AdderServiceImpl` befindet sich im package `rem`.

Damit der Addierer vom Client gefunden werden kann, muss er allen Interessenten zur Verfügung gestellt werden. Dazu müssen der HTTP–Server und der Service genau wie bei der lokalen Variante gestartet werden:

```
java -jar /usr/local/jini/lib/tools.jar -port 8085 -dir ./service -verbose

java -cp /usr/local/jini/lib/jini-core.jar:/usr/local/jini/lib/jini-ext.jar:
/usr/local/jini/lib/sun-util.jar:./service
-Djava.rmi.server.codebase=http://HTTPServerHostname:8085/
-Djava.security.policy=/usr/local/jini/policy/policy.all AdderServiceWrapper
```

2.4 Zusammenfassung

Im ersten Abschnitt dieses Kapitels wurde der Arbeitsablauf innerhalb einer Jini–Service–Community erklärt. Es wurde dargelegt, welche Schritte ein Dienstleister zu tun hat, damit sein Service veröffentlicht wird. Auch das Vorgehen eines Clients, der einen Service sucht, wurde gezeigt. Dieser Arbeitsverlauf ist in Abbildung 2.2 noch einmal zur Veranschaulichung dargestellt.

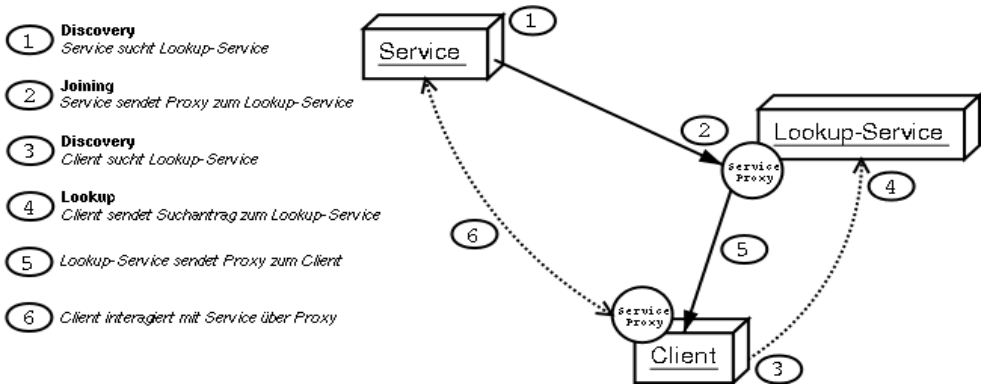

Abbildung 2.2: Der Arbeitsablauf in einer Jini–Community

Zu den in Abschnitt 2.1 vorgestellten Interaktionen findet man detailliertere Informationen in [Li00]. Besonders empfehlenswert sind der Überblick von S. 133 – 143, sowie die Kapitel 5 und 6 über Discovery- und Join–Protokolle, bzw. über den Jini–Lookup–Service.

Weiterführende Erklärungen zum Einrichten der Arbeitsumgebung (Tutorial, Abschnitt 2.2) finden sich in [OaksWong00, Kapitel 2]. Allerdings ist manches etwas veraltet, da dort mit der Version 1.0.1 des Jini Technology Starter Kit gearbeitet wird. Für aktuellere Informationen sei auf die Homepage von Sun verwiesen. Die Seite

`http://developer.java.sun.com/developer/technicalArticles/jini/javaspaces/index.html`

bietet viele Informationen, speziell auch für *mahalo* und *outrigger*.

Im Abschnitt 2.3 wurden kurz zwei Möglichkeiten, Jini–Services zu implementieren, vorgestellt. [Li00, Kapitel 12] geht mehr ins Detail und zeigt noch weitere Implementierungsmöglichkeiten auf.

Kapitel 3

JavaSpaces^{TM}– ein besonderer Jini–Service

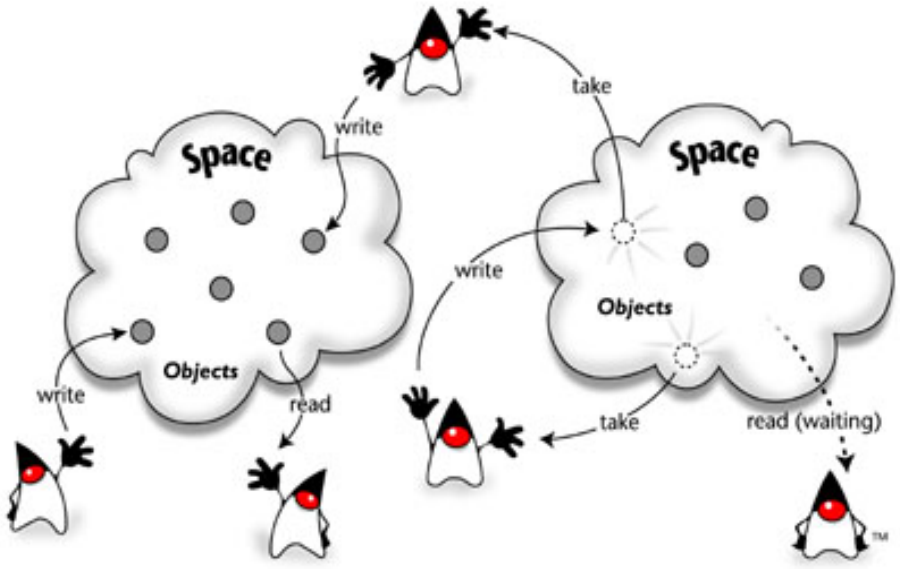

Abbildung 3.1: Koordination und Operationen in JavaSpaces (aus [JSTechnology])

3.1 Die JavaSpaces^{TM}–Technologie

Die JavaSpaces–Technologie basiert auf der Jini–Technologie, sie ist eigentlich nur ein spezieller Jini–Service, genau so wie der zuvor betrachtete Addierer. Deshalb wird auch das gesamte Jini–Framework benötigt, um mit der JavaSpaces–Technologie arbeiten zu können. Im Verständnis der JavaSpaces–Technologie ist eine Applikation eine Sammlung von Prozessen. Diese Prozesse kooperieren, indem Objekte einen oder mehrere sogenannte *Spaces* betreten und wieder verlassen. Damit wird die Erstellung von verteilter Softwa-

re erleichtert, denn die Prozesse sind nur sehr schwach miteinander verbunden. Jegliche Kommunikation sowie die – besonders bei verteilten Systemen – schwierige Synchronisation der Aktivitäten geschieht mit Hilfe des Space.

3.1.1 Was ist ein JavaSpace?

Ein Space ist ein fortdauerndes Objekte–Lager, das von verschiedenen Prozessen gemeinsam genutzt wird und über das Netzwerk zugänglich ist. Des weiteren dient ein Space zum Austausch von Objekten, denn die Prozesse kommunizieren nicht direkt miteinander, sondern die Koordination geschieht mittels des Space. Wie in Abbildung 3.1 zu sehen ist, führen die Prozesse drei einfache Operationen aus:

write: Sie schreiben Objekte in den Space.

take: Sie nehmen Objekte aus dem Space heraus.

read: Sie lesen Objekte, die sich im Space befinden.

Solange sich ein Objekt im Space befindet, ist es passiv. Um ein Objekt zu verändern, oder ein Methode auf ihm aufzurufen, muss ein Prozess das Objekt explizit aus dem Space entfernen, es eventuell verändern und danach wieder zurücklegen.

Die JavaSpaces sind aus den „Tuple–Spaces“ entstanden, die im Zusammenhang mit der Programmiersprache „Linda“ im Jahr 1982 das Licht der Welt erblickten. Weitere Informationen zu den Tuple–Spaces und der Geschichte der JavaSpaces findet man in [FreemanHupferArnold99, Foreword].

3.1.2 Die Eigenschaften der Spaces

Die JavaSpaces–Technologie ist aufgrund seiner Spaces ein einfaches und zugleich mächtiges Tool. Die Eigenschaften der Spaces leiten sich aus den Eigenschaften der Jini–Technologie her:

Spaces sind Gemeinschaftsräume: Spaces können über das Netzwerk betreten werden. Sie sind frei verfügbarer gemeinsamer Speicherplatz, in dem der gleichzeitige Zugriff geregelt ist.

Spaces sind fortdauernd: Spaces bieten zuverlässigen Lagerraum für Objekte. Befindet sich ein Objekt erst einmal in einem Space, so bleibt es dort, bis es ausdrücklich entfernt wird, oder bis seine Leasing–Zeit verstrichen ist.

Spaces sind verbindend: Objekte werden in Spaces per *associative lookup* gefunden. Unwichtig ist, wie das Objekt heisst, wer es geschaffen hat, oder wo es gespeichert ist. Lediglich sein Inhalt ist von Interesse, das passende Objekt wird über Template–Matching gefunden.

Spaces sind sicher durch Transaktionen: Spaces unterstützen atomare Operationen. Entweder wird die gesamte Operation, die aus einzelnen Unter–Operationen bestehen kann, ausgeführt, oder gar nicht. Transaktionen können sich auch über mehrere Spaces erstrecken.

Spaces dienen dem Austausch ausführbarer Programme: Ein Objekt in einem Space ist passiv. Sobald es jedoch aus dem Space herausgenommen wurde, können seine Methoden aufgerufen werden und / oder es verändern.

3.2 Ein einfaches Beispiel: Hello World

Mit Hilfe dieses einführenden Beispiels sollen die Grundzüge der JavaSpaces–Programmierung gezeigt werden. Detaillierte Informationen lassen sich in [FreemanHupferArnold99] finden.

3.2.1 Entrys

Ein Space beherbergt Einträge – Entrys. Die Idee hinter den Entrys ist sehr einfach: Ein Entry ist ein Objekt, das das `net.jini.core.entry.Entry`–Interface implementiert. Unser Entry hat ein einziges Feld, nämlich den Inhalt der Nachricht:

```
package helloWorld;

public class Message implements net.jini.core.entry.Entry {
  public String content;
  public Message() {
  }
}
```

Nun wird ein `Message`–Entry instanziert und mit „Hello World“ gefüllt:

```
Message msg = new Message();
msg.content = "Hello World";
```

3.2.2 Spaces

Dieser Entry soll nun in den Space geschrieben werden. Dafür wird ein Space–Objekt benötigt, das die Methode `getSpace` der Klasse `SpaceAccessor` liefert. `SpaceAccessor` ist eine Möglichkeit von vielen, ein Space–Objekt zu bekommen:

```
package helloWorld;

import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.lookup.ServiceItem;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.lookup.ServiceDiscoveryManager;
import net.jini.space.JavaSpace;
import net.jini.lookup.entry.Name;
import net.jini.core.entry.Entry;

import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;

public class SpaceAccessor {

  protected static ServiceTemplate template;

  public static JavaSpace getSpace(String spaceName) {

    /* create a new template with the given name */
    Entry[] attr = new Entry[1];
    attr[0] = new Name(spaceName);
```

```
    Class[] types = { JavaSpace.class };
    template = new ServiceTemplate(null, types, attr);

    JavaSpace space = null;

    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new RMISecurityManager());
    }

    try {
      LookupDiscoveryManager ldm =
        new LookupDiscoveryManager(new String[] { "" }, null, null);

      ServiceDiscoveryManager sdm =
        new ServiceDiscoveryManager(ldm, null);

      ServiceItem o = sdm.lookup(template, null, 20000);
      if(o==null){
        System.out.println("A problem has occured.");
      } else {
        System.out.println("Got a matching service");
      }
      space = ((JavaSpace)o.service);
    } catch(Exception e) {
      e.printStackTrace();
    }
    return space;
  }

  public static JavaSpace getSpace() {
    return getSpace("JavaSpaces");
  }
}
```

Zwei Varianten der Methode `getSpace` werden angeboten. Die erste nimmt als Argument den Namen des Space, auf den man Zugriff haben möchte. Die parameterlose Version liefert einen Space mit dem Namen „JavaSpaces“, welches die Standardbezeichnung für einen Space ist, dem kein Name zugeordnet wurde. Der `SpaceAccessor` findet den JavaSpace–Service auf die gleiche Weise, wie der `AdderClient` in Unterabschnitt 2.3.1: Nach der Installation eines `RMISecurityManager` wird mit einem `LookupDiscoveryManager` und einem `ServiceDiscoveryManager` nach einem Service gesucht, der das `JavaSpace`–Interface implementiert. So wird ein Space, der im Grunde auch nur ein Jini–Service ist, gefunden.

Folgende Zeile liefert einen Space, mit dem danach durch die Anwendung einiger Basisoperationen interagiert werden soll:

```
JavaSpace space = SpaceAccessor.getSpace();
```

3.2.3 Die Operation `write`

Die `write`–Methode legt die Kopie eines Entry in den Space. Wird `write` mehrmals hintereinander mit demselben Entry aufgerufen, so werden mehrere Kopien dieses Entry in den Space befördert.

```
space.write(msg, null, Lease.FOREVER);
```

Hiermit wurde unser „Hello World“–Entry in den Space geschrieben, ohne dass dabei eine Transaktion verwendet wurde. Ihm wurde eine Leasing–Zeit mit auf den Weg gegeben, die nie abläuft.

3.2.4 Die Operation `read`

Da nun unser Entry im Space existiert, kann ihn jeder Prozess, der Zugriff auf diesen Space hat, lesen. Um einen Entry zu lesen, wird ein Template, eine Schablone, benutzt. Dieses Template ist ein Entry, bei dem ein oder mehrere Felder auf `null` gesetzt sind. Ein Entry passt auf ein Template, wenn er den gleichen Typ (oder einen Untertyp) wie das Template hat und wenn alle Felder, die nicht auf `null` gesetzt wurden, genau übereinstimmen. Die `null`–Felder agieren als Wildcards und eignen sich für jeden Wert. Ein Template eines bestimmten Entry wird ganz einfach angefertigt:

```
Message template = new Message();
```

Es ist wichtig zu wissen, dass das `content`–Feld dieses Templates auf `null` gesetzt ist, da Java alle nicht initialisierten Objekte bei ihrer Erschaffung mit `null` initialisiert. Da wir ein Template haben, dessen `content`–Feld eine Wildcard enthält, wird jeder `Message`–Entry im Space darauf passen, egal welche Nachricht er enthält. Die `read`–Operation liefert die Kopie eines im Space vorhandenen passenden Entry:

```
Message result = (Message)space.read(template, null, Long.MAX_VALUE);
```

Da `read` lediglich einen `Entry` zurückgibt, muss er wieder auf `Message` „gecastet“ werden. Die obige `read`–Operation wurde ohne Transaktion durchgeführt, und es wurde so lange wie irgendwie möglich (`Long.MAX_VALUE`) auf einen passenden Entry gewartet. Um den Inhalt unseres Entrys auszudrucken genügt diese Zeile:

```
System.out.println(result.content);
```

Es ist nicht weiter erstaunlich, das Ergebnis ist:

```
Hello World
```

3.2.5 Die Operation `take`

Die `take`–Operation funktioniert genauso wie die `read`–Operation. Der einzige Unterschied ist, dass mit

```
Message result = (Message)space.take(template, null, Long.MAX_VALUE);
```

der Entry, der als passend herausgefunden wurde, aus dem Space herausgenommen wird. Wir erhalten jedoch den gleichen Ausdruck wie zuvor.

3.2.6 Serialisierung und `snapshot`

Jeder Entry ist ein lokales JavaSpace–Objekt. Der Space, der sich remote auf irgendeiner Maschine befindet, kennt den Entry als einen Proxy, und nicht als Referenz zu einem Objekt, das sich auf einer anderen Maschine befindet. Der `SpaceAccessor` liefert im Grunde auch keinen Space zurück, sondern „nur“ einen Space–Proxy, über den die zuvor beschriebenen Operationen ausgeführt werden. Dieser Proxy ist auch dafür verantwortlich, dass die Entrys richtig im Space ankommen, bzw. dass sie korrekt aus dem Space herausgenommen werden. In Abbildung 3.2 ist dargestellt, wie dieser Prozess vor sich geht:

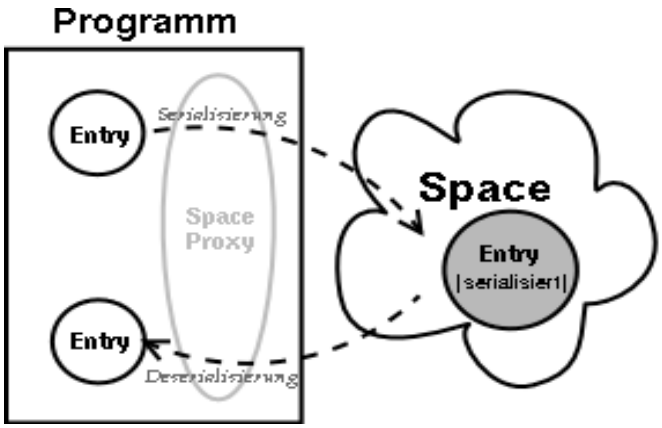

Abbildung 3.2: Serialisierung und Deserialisierung von Entrys

Wird ein Entry in den Space geschrieben, so muss der Proxy sich um dessen Serialisierung[1] kümmern. Im Space ist der Entry dann in seiner serialisierten Form gespeichert. Bei einer `read`- oder `take`–Operation erhält das Programm den serialisierten Entry aus dem Space. Der Space–Proxy kümmert sich um die Deserialisierung, damit die Methoden einen „normalen“ Entry zur Verfügung haben, mit dem sie arbeiten können.

Für `read` und `take` sind Templates erforderlich, um den passenden Entry im Space zu finden. Auch die Templates müssen die Serialisierung des Space–Proxy durchlaufen, damit sie im Space in serialisierter Form auftauchen und sie Feld für Feld mit den Entrys verglichen werden können. Zwei Felder stimmen überein, wenn die Bytes ihrer serialisierten Form übereinstimmen. Deshalb führt kein Weg an der Serialisierung der Templates vorbei.

Benutzt man einen Entry wieder und wieder, ohne daran Veränderungen vorzunehmen, erzeugt man unnötige Kosten, denn der Entry wird immer von Neuem serialisiert. In Schleifen kommt es häufig vor, das gleiche Template mehrere Male hintereinander in `read`- oder `take`–Operationen zu verwenden. Um einer Verschwendung von Rechenleistung durch unnötige Serialisierungen vorzubeugen, bietet die JavaSpaces API eine `snapshot`–Methode an, die einen Entry als Argument nimmt und einen „Schnappschuss“ davon

[1] Serialisierung besteht darin, ein Objekt in einen Bytestrom (*byte stream*) umzuwandeln. So kann es in einer Datei gespeichert oder über das Netzwerk verschickt und anschliessend wieder hergestellt werden – als Kopie des Originals.

zurückgibt. Dieses „Photo“ hat die serialisierte Form des Original–Entrys und kann nun für Operationen im Space gebraucht werden.

Im folgenden Beispiel wird hundert Mal die Methode `read` aufgerufen. Das `MyEntry`–Template wird jedes Mal an den Space–Proxy übergeben, der es serialisiert und an den Space sendet, damit es mit den vorhandenen Entrys verglichen werden kann:

```
MyEntry template = new MyEntry();
for (int i = 0; i < 100; i++){
  MyEntry result = (MyEntry)space.read(template, null, Long.MAX_VALUE);
}
```

Da das Template sich nie verändert, ist die erneute Serialisierung bei jeder Iteration nicht notwendig. Mit `snapshot` wird die Schleife effizienter:

```
MyEntry template = new MyEntry();
Entry snapTemplate = space.snapshot(template)
for (int i = 0; i < 100; i++){
  MyEntry result = (MyEntry)space.read(snapTemplate, null, Long.MAX_VALUE);
}
```

Hier wurde eine `snapshot`–Version des Entry `template` gemacht und auf die Variable `snapTemplate` geschrieben. Da nun `snapTemplate` an `read` übergeben wird, wird der Entry nicht jedes Mal vom Space–Proxy serialisiert. Das Template wird nur ein einziges Mal serialisiert (in der `snapshot`–Operation), und nicht hundert Mal.

3.2.7 Das „Hello World“–Programm

Wenn die soeben gesehenen Code–Fragmente korrekt zusammengebaut werden, entsteht ein komplettes „Hello World“-Programm, das auf einem JavaSpace basiert:

```
package helloWorld;

import net.jini.core.lease.Lease;
import net.jini.space.JavaSpace;

public class HelloWorld {
  public static void main(String[] args) {
    try {
      Message msg = new Message();
      msg.content = "Hello World";

      JavaSpace space = SpaceAccessor.getSpace();
      space.write(msg, null, Lease.FOREVER);

      Message template = new Message();
      Message result =
        (Message)space.read(template, null, Long.MAX_VALUE);
      System.out.println(result.content);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

3.3 Kompilieren und Ausführen von JavaSpaces–Programmen

Für das korrekte Funktionieren der JavaSpaces–Technologie sind ein HTTP–Server und *rmid*, sowie die im Abschnitt 2.2 vorgestellten Jini–Services *reggie*, *mahalo* und *outrigger* notwendig. Wenn diese laufen, kann mit dem Kompilieren begonnen werden:

```
javac -classpath /usr/local/jini/lib/jini-core.jar:
/usr/local/jini/lib/jini-ext.jar:/usr/local/jini/lib/sun-util.jar:
/usr/local/jini/lib/space-examples.jar:. -d . *.java;
```

Zum bekannten Jini–Classpath kommt hier noch die `space-examples.jar`–Datei hinzu, die verschiedene Klassen der JavaSpaces–Technologie enthält. Die erzeugten `class`–Dateien werden in das aktuelle Verzeichnis geschrieben und es werden alle im Verzeichnis liegenden `java`–Dateien kompiliert.

Nun kann das „Hello World“–Programm ausgeführt werden. Dazu dient der folgende (einzeilige!) Befehl:

```
java -Djava.security.policy=/usr/local/jini/policy/policy.all
-Dmahalo.managerName=TransactionManager
-Doutrigger.spacename=JavaSpaces
-Dcom.sun.jini.lookup.groups=public
-Djava.rmi.server.codebase=http://hostname:8080/space-examples-dl.jar
-cp /usr/local/jini/lib/space-examples.jar:/usr/local/jini/lib/jini-core.jar:
/usr/local/jini/lib/jini-ext.jar:/usr/local/jini/lib/sun-util.jar:.
helloWorld/HelloWorld
```

Die `-Djava.security.policy`–Option gibt die `policy`–Datei an, die benutzt werden soll, wenn das Programm heruntergeladenen Code ausführt. Das kann z. B. ein Entry aus dem Space sein, auf dem eine Methode aufgerufen wird. Mit `-Dmahalo.managerName`, bzw. `-Doutrigger.spacename` werden die Namen des `TransactionManager` respektive des `JavaSpace` angegeben, mit denen das Programm zusammenarbeiten soll. Der `-Dcom.sun.jini.lookup.groups`–Parameter gibt die Jini–Community des Programms an. Exportiert das Programm herunterladbaren Code, muss die `-Djava.rmi.server.codebase`–Eigenschaft gesetzt werden. Dies ist z. B. der Fall, wenn ein Programm (wie unser „Hello World“–Programm) Entrys in einen JavaSpace schreibt. Denn wenn andere Programme diese Entrys lesen oder heraus nehmen, müssen sie wissen, wo sie die dazugehörigen Klassen herunterladen können. Auf diese Weise trägt jeder Entry, der in den Space geschrieben wird, diese Information in sich. Schliesslich werden noch die zur Ausführung benötigten `jar`–Dateien angegeben. Das auszuführende Programm befindet sich im package „helloWorld“.

Die anhand des „Hello World“–Programms beschriebenen Vorgänge zum Kompilieren und Ausführen können leicht an jedes JavaSpaces–Programm angepasst werden.

Kapitel 4

Eine parallele Anwendung

Die JavaSpaces–Technologie ermöglicht es, auf einen einfache Weise Anwendungen zu entwerfen und zu implementieren, die aus verschiedenen Prozessen bestehen. Diese Prozesse sind über ein Netzwerk auf mehreren Maschinen verteilt und arbeiten zusammen, um ein gemeinsames Problem zu lösen. Die JavaSpaces bieten eine sehr gute Grundlage für eine parallele Anwendung, denn sie bringen wichtige Eigenschaften mit sich:

Performanz – Leistung: Anstelle von einer CPU werden mehrere CPUs benutzt, um die Lösung des Problems zu erlangen. Dazu muss das Ausgangsproblem in viele kleine Einzelprobleme zerlegt werden. Diese werden dann auf mehrere Computer verteilt, parallel verarbeitet und schliesslich wieder zusammengefügt.[1]

Erweiterbarkeit: Bei einer Einzelanwendung hängt die Leistung von der jeweiligen Maschine ab. Ist aber eine Anwendung dafür entworfen worden, dass sie auf mehreren Prozessoren gleichzeitig laufen soll, dann ist das nicht nur eine Leistungssteigerung, sondern damit wurde eine erweiterbare Anwendung geschaffen: Falls das zu bewältigende Problem zu viel Arbeit für die damit beauftragten CPUs darstellt, so können einfach eine oder mehrere Maschinen zu dieser Gemeinschaft hinzugefügt werden, ohne dass die Anwendung ein neues Design bräuchte. Somit wurde ein *distributed–computing*–Motor geschaffen, der beliebig wachsen und schrumpfen kann, je nach Bedarf.

Fehler–Toleranz und Verfügbarkeit: Wenn eine standalone–Anwendung versagt, so bleibt die Dienstleistung so lange nicht verfügbar bis sie wieder neu gestartet wird. In einem verteilten System hingegen kann Versagen bis zu einem bestimmtem Niveau kompensiert werden, nämlich dann, wenn viele, voneinander unabhängige Prozesse gestartet wurden. Falls einige Prozesse versagen, können immer noch andere Prozesse weiterarbeiten. Damit wird die Verfügbarkeit deutlich maximiert.

Eleganz: Für die meisten Probleme ist ein verteiltes System die “natürlichste„ Art und Weise der Lösung: Es ist einfacher und eleganter, ein Design als Gemeinschaftswerk von relativ unabhängigen spezialisierten Teilstücken, jedes einzelne wird von einem individuellen Prozess verwirklicht, zu entwerfen, als ein einziges grosses, komplexes, schwer instandzuhaltendes Einzelstück zu entwerfen und zu verwirklichen.

[1] Man möchte meinen, dass eine immer grössere Anzahl von Prozessoren auch eine gleichmässige Leistungssteigerung bringt. Im Abschnitt 4.3, einer Performanz–Analyse dieser parallelen Anwendung, wird sich zeigen, dass bei zu vielen Prozessoren die Kommunikation (d. h. das abschliessende Zusammenfügen) den Gewinn, den man aus der Parallelverarbeitung erlangt hatte, wieder abschwächt.

4.1 Beschreibung der Anwendung

Ziel dieser Anwendung ist es, Parallelverarbeitung, die auf einem JavaSpace basiert, genauer zu betrachten.

4.1.1 Die ursprüngliche Anwendung

Als Ausgangspunkt diente die in [FreemanHupferArnold99, Kapitel 11] vorgestellte Anwendung, die ein Passwort per *brute–force*–Attacke knackt. Die Idee hinter dieser parallelen Anwendung ist ganz einfach: Der `CryptMaster` (oder auch Master) erhält das Passwort, das es zu knacken gilt. Anhand dieses Passworts können alle Zeichen–Kombinationen gleicher Länge erzeugt werden, die potentielle Passwörter sein könnten. Generierung und Überprüfung liegen jedoch nicht in der Hand des Master, sondern er lässt die `GenericWorker` (im Weiteren als Worker bezeichnet) diese Arbeit verrichten.

1. Dazu unterteilt der Master das Ausgangsproblem, das nämlich darin besteht, alle potentielle Passwörter auf ihre Identität mit dem Originalpasswort zu überprüfen, in mehrere Unterprobleme (`CryptTask`, im Weiteren als Task bezeichnet). Diese Unterprobleme werden in den Space geschrieben und dann von den Workern ausserhalb des Space bearbeitet.

2. Ein Worker, der ein Task aus dem Space genommen hat, findet darin eine Zeichenfolge, die das erste Wort darstellt, das er zu testen hat. Ausserdem beinhaltet ein Task eine Zahl, die dem Worker sagt, wie oft das Anfangswort zu inkrementieren ist, d. h. wie viele potentielle Passwörter er zu testen hat. Schliesslich ist auch noch das Passwort selbst Bestandteil eines Tasks, damit der Worker lokal jedes generierte potentielle Passwort auf Übereinstimmung mit dem Originalpasswort testen kann.

3. Diese Arbeit findet auf der Maschine, auf der der Worker läuft, statt.

4. Hat der Worker alle ihm zugeteilten Passwörter überprüft und dabei das gesuchte Passwort nicht gefunden, so schreibt er ein leeres `CryptResult` (im Weiteren als Resultat bezeichnet) in den Space zurück. Trifft er aber beim Testen auf das gesuchte Passwort, schreibt er sofort ein Resultat, in dem das gefundene Passwort steht, in den Space zurück.

5. Solange das Passwort noch nicht gefunden ist, nehmen die Worker immer wieder neue Tasks aus dem Space, um sie bei sich zu überprüfen.

6. Der Master entfernt die Resultate aus dem Space und stoppt die Berechnungen, sobald das gesuchte Passwort gefunden ist, indem er keine weiteren Tasks mehr herausgibt.

Abbildung 4.1 auf Seite 33 zeigt das Zusammenspiel des Masters mit den Workern.

4.1.2 Notwendige Veränderungen

Um die Performanz, die Erweiterbarkeit, die Fehler–Toleranz und die Verfügbarkeit einer parallelen Anwendung, die in einem Space agiert, besser analysieren zu können, waren

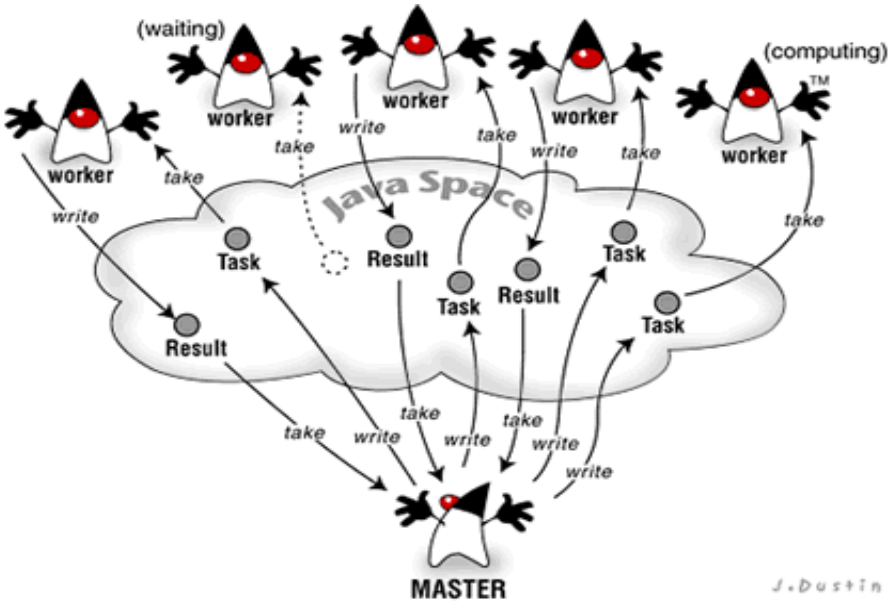

Abbildung 4.1: Übersicht der parallelen Anwendung in einem JavaSpace (aus [MakeRoom])

einige Modifikationen an der in [FreemanHupferArnold99] vorgeschlagenen Applikation vorzunehmen. Das Interesse war auf die Zeit gerichtet, die benötigt wird, um das Passwort zu knacken. Da es für solch eine Analyse wichtig ist, eine grosse Menge an Daten zur Verfügung zu haben, musste die Berechnung automatisiert werden: Statt eines Passworts können jetzt beliebig viele Passwörter gefunden werden, denn die Anwendung endet nicht nachdem das Passwort gefunden wurde, sondern stellt neue Tasks in den Space. Dafür waren sowohl auf der Seite des Master als auch bei den Workern Veränderungen vorzunehmen, damit es zu keiner „Unordnung“ im Space kommt. Im Zuge dieser Automatisierung wurde auch das GUI entfernt und der Chiffrierungsalgorithmus verändert. Der ursprüngliche rechenintensive Algorithmus wurde aufgegeben. Es findet keine Chiffrierung und Dechiffrierung mehr statt, die Zeichenfolgen werden direkt miteinander verglichen. Dies ist durch die Tatsache zu rechtfertigen, dass in möglichst kurzer Zeit möglichst viele Resultate erzeugt werden sollen. Ein rechenintensiver Algorithmus hätte die Zeit bis zum Knacken des Passworts lediglich verlängert, auf das Ergebnis aber keinerlei Auswirkungen gehabt. Aus dem gleichen Grund wurden die Zeichen, die im Passwort vorkommen dürfen, eingeschränkt. Die ursprünglich 95 druckbaren ASCII–Zeichen wurden auf die Zahlen 0 bis 9 reduziert. Als positive Nebenwirkung dieser Veränderung stellte sich heraus, dass die Inkrementierung des zu testenden Worts erheblich vereinfacht wurde.

4.2 Erläuterungen zum Quellcode

Diese Anwendung macht von der gesamten JavaSpaces API und ihren fortgeschrittenen Methoden, wie z. B. Transaktionen, Gebrauch. Soweit es im Rahmen dieses Dokuments

möglich ist, werden dazu Erklärungen geliefert. Für weitere Informationen sei aber schon jetzt auf [FreemanHupferArnold99] verwiesen.

Im Anhang A sind verschiedene Screenshots dieser Anwendung zu finden.

4.2.1 Das `Command`–Interface

Das `Command`–Interface erweitert `Entry` und hat nur eine einzige Methode, nämlich `execute`. Jedes ausführbare Task, das in den Space geschrieben wird, muss dieses Interface implementieren. Somit kann seine `execute`–Methode aufgerufen werden.

```
public interface Command extends Entry {
  public Entry execute(JavaSpace space);
}
```

4.2.2 Der `TaskEntry`

Diese Klasse ist die Basis für jedes ausführbare Task, denn sie implementiert das `Command`–Interface, was das Task zum `Entry` macht. Ausserdem erzwingt sie, dass ihre Unterklassen eine Implementierung für die `execute`–Methode bieten, denn sonst wird eine `RuntimeException` geworfen. Die Methode `resultLeaseTime` wird gebraucht, um eine Leasing–Zeit für die Resultate anzugeben, die vom Worker in den Space zurückgeschrieben werden.

```
public class TaskEntry implements Command {
  public Entry execute(JavaSpace space) {
    throw new RuntimeException(
      "TaskEntry.execute() not implemented");
  }

  public long resultLeaseTime() {
      return 1000 * 10 * 60;  // 10 minutes
  }
}
```

4.2.3 Der Worker

Im Anhang A zeigt die Abbildung A.1 einen Worker, der auf Arbeit wartet. Die Basisoperation des `GenericWorker` ist sehr einfach: Er nimmt einen `TaskEntry` aus dem Space und ruft dessen `execute`–Methode. Falls diese ihm einen Entry zurückliefert und der Space noch bereit ist, Resultate aufzunehmen, schreibt der Worker das Ergebnis in den Space zurück. Danach beginnt der Worker wieder von vorne. Hier ist die `run`–Methode des Workers:

```
public void run() {

  Entry taskTmpl = null;

  try {
    taskTmpl = space.snapshot(new TaskEntry());
  } catch (RemoteException e) {
```

```
      throw new RuntimeException("Can't obtain a snapshot");
    }

    while (true) {

      Transaction txn = getTransaction();
      if (txn == null) {
        throw new RuntimeException("Can't obtain a transaction");
      }

      try {
        TaskEntry task = (TaskEntry)space.take(taskTmpl, txn, Long.MAX_VALUE);
        Entry result = task.execute(space);
        if ((result != null) && (noPillInSpace())) {
          space.write(result, txn, task.resultLeaseTime());
        }
        txn.commit();
      } catch (RemoteException e) {
        e.printStackTrace();
      } catch (TransactionException e) {
        e.printStackTrace();
      } catch (UnusableEntryException e) {
        e.printStackTrace();
      } catch (InterruptedException e) {
        try {
          txn.abort();
        } catch(Exception ex) {
          /* RemoteException or BadTransactionException
          lease expiration will take care of the transaction*/
        }
        System.out.println("Task cancelled");
      }
    }
  }
```

Zwei Besonderheiten bringt die Implementierung des Workers mit sich: einen Snapshot und eine Transaktion. Da immer wieder das gleiche Task–Template benutzt wird, können wir eine `snapshot`–Version des Templates verwenden. Damit wird eine erneute Serialisierung bei jedem `take` vermieden (vgl. Unterabschnitt 3.2.6). Um die gesamte Berechnung gegen teilweises Versagen robust zu machen, wurde vor der `take`–Operation eine Transaktion (`getTransaction`) eingeführt, die erst nach der `write`–Operation beendet wird. So wurde aus diesen beiden Operationen eine atomare Operation gemacht – entweder beide werden erfolgreich ausgeführt oder keine. Eine Transaktion ist von Nutzen, wenn der Worker ein Task aus dem Space nimmt, mit dessen Berechnung beginnt und dann crasht. Ohne Transaktion wäre dieses Task für immer verloren, da sein Resultat nie in den Space zurück geschrieben wird. Da der Space aber über alle Transaktionen informiert ist, wird er nach einer in `getTransaction` angegebenen Leasing–Zeit (hier: zwei Minuten) das Task erneut in den Space schreiben, damit das ihm innewohnende Resultat nicht verloren geht. Die `getTransaction`–Methode erzeugt mit Hilfe eines `TransactionManager` und einer Leasing–Zeit eine Transaktion:

```
public Transaction getTransaction() {
  TransactionManager mgr =
    TransactionManagerAccessor.getManager();

  try {
    long leaseTime = 1000 * 2 * 60; // two minutes
    Transaction.Created created = TransactionFactory.create(mgr, leaseTime);
    return created.transaction;
  } catch(RemoteException e) {
    e.printStackTrace();
    return null;
  } catch(LeaseDeniedException e) {
    e.printStackTrace();
    return null;
  }
}
```

Das korrekte Bestimmen der Leasing–Zeit erfordert Kenntnisse über die Beschaffenheit der Maschinen und des zur Verfügung stehenden Netzwerkes. Da eine Bearbeitungszeit, die zwei Minuten überschreitet, bei dem in Abschnitt 1.3 beschriebenen Material vom Normalfall abweichen würde, wurde hier die Leasing–Zeit auf zwei Minuten festgelegt. Bräuchte eine Maschine jedoch immer länger als die festgelegte Leasing–Zeit, so wäre ihre Arbeitszeit verschwendet: Sie schafft es nicht, innerhalb von zwei Minuten `take` und `write` auszuführen. Somit schlägt `txn.commit` in der `run`–Methode des Worker fehl, was `txn.abort` auslöst. Der Transaktionen–Manager informiert nun alle Beteiligten von der Entscheidung, die Transaktion abzubrechen und versetzt das gesamte System in den Zustand vor dem `take` zurück. Der „zu langsame“ Worker, muss mitten in seiner Arbeit unterbrechen, wird es aber nie schaffen, auch nur ein einziges Resultat zu erzeugen, da er nicht schnell genug ist.

4.2.4 Das Resultat einer Berechnung: `CryptResult`

Der Worker erhält von der `execute`–Methode ein `CryptResult` zurück, das sodann in den Space geschrieben wird um vom Master aufgesammelt zu werden. Es ist ein einfacher Entry mit zwei Feldern: dem Anfangswort des getesteten `TaskEntry` und einem Feld, in dem sich entweder das geknackte Passwort befindet oder `null`. Eben dieses Feld wird vom Master getestet und wenn es ungleich `null` ist, weiss er, dass das Passwort gefunden ist.

```
public class CryptResult implements Entry {
  public String word = null;
  public String startPoint;

  public CryptResult() {
  }
  public CryptResult(String word, String sp) {
    this.word = word;
    this.startPoint = sp;
  }
}
```

4.2.5 Das Herz der Berechnung: `CryptTask`

Die Worker suchen im Space nach Tasks vom Typ `TaskEntry`, der von der Klasse `CryptTask` erweitert wird. Diese Klasse stellt die Implementierung der `execute`–Methode zur Verfügung, die von den Workern aufgerufen wird.

```
public class CryptTask extends TaskEntry {

  public Integer tries;
  public String word;
  public String original;

  public CryptTask() {
  }
  public CryptTask(int tries, String word, String original) {
    this.tries = new Integer(tries);
    this.word = word;
    this.original = original;
  }

  public Entry execute(JavaSpace space) {
    String startPoint = word;
    PoisonPill template = new PoisonPill();

    // If a PoisonPill exists, skip the computation and return null.
    try {
      if (space.readIfExists(template, null, JavaSpace.NO_WAIT) != null) {
        return null;
      }
    } catch (Exception e) {
      ; // continue on
    }

    int num = tries.intValue();
    for (int times = 0; times < num; times++) {
      if (original.equals(word)) {
        System.out.println(">>>>>> I found it! <<<<<<<");
        CryptResult result = new CryptResult(word,startPoint);
        return result;
      }
      word = alterItNTimes(word, 1); //get the next word to encrypt
    }
    CryptResult result = new CryptResult(null, startPoint);
    return result;
  }
}
```

Ein `CryptTask` enthält drei wichtige Informationen: `tries`, eine ganze Zahl, die angibt, wie viele potentielle Passwörter innerhalb dieses Tasks erzeugt und getestet werden sollen; `word`, die Zeichenkette, mit der die Überprüfung begonnen wird und aus der die folgenden potentiellen Passwörter generiert werden; `original`, das Passwort, das geknackt werden soll. Diese drei Felder werden mittels des Konstruktor gefüllt.

In der `execute`–Methode wird zuerst überprüft, ob der Space noch bereit ist, Resultate aufzunehmen. Diese Überprüfung geschieht, indem nachgeschaut wird, ob sich

ein `PoisonPill`–Entry (siehe Seite 43) im Space befindet. Ist dies der Fall, so wird die eigentliche Arbeit gar nicht erst begonnen, sondern sofort `null` zurückgegeben. Die darauf folgende Schleife wird so oft ausgeführt, wie potentielle Passwörter in diesem Task getestet werden sollen. Zuerst wird die in `word` befindliche Zeichenkette auf Identität mit dem zu knackenden Passwort überprüft. War dieser Test erfolgreich, so wird ein `CryptResult` mit dem gefundenen Passwort zurückgegeben. Ansonsten erhöht die Methode `alterItNTimes`[2] das Passwort um eins und die Schleife beginnt von Neuem. Konnte das Passwort nicht geknackt werden, so wird ein `CryptResult` zurückgegeben, dessen `word`–Feld `null` enthält. Abbildung A.2 zeigt den Moment, in dem ein Passwort geknackt wurde.

4.2.6 Die Steuerungszentrale der Berechnung – der `CryptMaster`

Der Master hat zwei Aufgaben: Zum einen zerlegt er das Ausgangsproblem in mehrere Teilprobleme, indem er Tasks produziert und sie in den Space schreibt. Zum anderen muss er wieder zusammenpuzzeln, was er auseinander genommen hat. Dazu nimmt er die von den Workern produzierten Resultate aus dem Space heraus und schaut, ob das Passwort geknackt wurde. Für jede dieser Aufgaben gibt es einen Thread. Doch zuvor gilt es, sich mit den Hilfsmitteln des Master bekannt zu machen:

4.2.6.1 Der Konstruktor

```
public CryptMaster(String original){
  this.space = SpaceAccessor.getSpace();
  this.cryptTaskTemplate = snapshotCryptTask();
  this.cryptResultTemplate = snapshotCryptResult();
  this.poisonPillTemplate = snapshotPoisonPill();
  this.original= original;
  this.length = original.length();

  takeAll("CryptTask", false);
  takeAll("CryptResult", false);
  takeAll("PoisonPill", false);
  cleanUpTheSpace();

  this.tasks = generateTasks();
  wThread = new WriteThread();
  wThread.start();
  cThread = new CollectThread();
  cThread.start();
}
```

Mit der `SpaceAccessor.getSpace`–Methode, die im Unterabschnitt 3.2.2 bei den Erklärungen zum „Hello World“–Programm vorgestellt wurde, wird der Space gefunden, in den der Master die Tasks schreiben wird. Danach werden Snapshot–Templates von `CryptTask`, `CryptResult` und `PoisonPill` angefertigt. Exemplarisch wird hier eine der drei Methoden angeführt – die anderen beiden funktionieren auf die gleiche Weise:

[2] Die Implementierungsdetails dieser Methode sind weder im Sinne der JavaSpaces–Technologie noch für das Verständnis der Anwendung von Bewandtnis. Deshalb werden sie hier nicht aufgeführt. Für ähnliche Fälle wird ebenso verfahren werden.

```
private Entry snapshotCryptTask(){
  Entry cryptTaskTemplate = null;
  try {
    cryptTaskTemplate = space.snapshot(new CryptTask());
  } catch (RemoteException e) {
    throw new RuntimeException("Can't create a CryptTask snapshot");
  }
  return cryptTaskTemplate;
}
```

Der Parameter des Konstruktors ist das zu knackende Passwort, dessen Länge für weitere Arbeiten festgehalten wird. Bevor der Space mit neuen Tasks gefüllt werden kann, muss er „aufgeräumt“ werden. Von der vorherigen Berechnung könnten noch Entrys im Space zu finden sein, die erheblich stören und die Ergebnisse verfälschen würden. Deshalb werden zuerst alle `CryptTask`–, `CryptResult`– und `PoisonPill`–Entrys entfernt und danach alles, was sonst noch im Space herumliegt.

```
private void takeAll(String type, boolean change){
  Entry en = null;
  if (type.equals("CryptTask")){
    en = cryptTaskTemplate;
  }
  if (type.equals("CryptResult")){
    en = cryptResultTemplate;
  }
  if (type.equals("PoisonPill")){
    en = poisonPillTemplate;
  }

  while(takeIfExistsTask(en)!= null){
    if (change){
      changeWaterLevel(-1);
    }
  }
}
```

Die `takeAll`–Methode nimmt alle Entrys des angegebenen Typs aus dem Space heraus. `takeIfExistsTask` ist eine Methode, die mittels `space.takeIfExists` einen Entry aus dem Space heraus nimmt, der mit dem Parameter übereinstimmt. Dies ist jedoch nur möglich, wenn ein solcher Entry im Space vorhanden ist. Ist dies nicht der Fall, wird sofort `null` zurückgegeben, es wird nicht auf einen passenden Entry gewartet. Auf `changeWaterLevel` wird unter 4.2.6.2 eingegangen. `cleanUpTheSpace` benutzt zum Template–Matching die Wildcard `null`. Mit der Methode `generateTasks` erzeugt der Master alle Tasks, die er in den Space schreiben wird, d. h. die Teilprobleme und füllt sie in einen `Vector`. Nun ist alles vorbereitet und der `WriteThread` sowie der `CollectThread` können gestartet werden.

4.2.6.2 Watermarking – Der „Wasserstand“ im Space

Bevor die beiden Threads im Detail betrachtet werden, muss noch das Watermarking erklärt werden. Ziel ist es, die Kapazitäten des Space ideal zu nutzen und damit die An-

wendung zu verbessern: Der Space soll so leer wie möglich aber so voll wie nötig sein, d. h. in ihm sollen sich gerade so viele Tasks befinden, dass nie einer der Worker zur Untätigkeit gezwungen wird. Es werden eine obere und eine untere Schranke gewählt, die angeben wieviel Tasks höchstens, bzw. mindestens im Space sein sollen. Zu Beginn wird der Space bis zur Obergrenze mit Tasks gefüllt und dann wird gewartet, bis der „Wasserstand“ unter die Untergrenze fällt. An diesem Punkt wird der Space wieder aufgefüllt. Davon ausgehend, dass die Anzahl der Worker während der Berechnung nicht variiert, wird als untere Schranke die Anzahl der Worker gewählt (so ist für jeden Worker mindestens ein Task im Space) und als obere Schranke zwei Mal die Untergrenze plus eins.

Immer wenn der Master ein `CryptTask` in den Space schreibt, erhöht er den Wasserstand (`waterlevel`) um eins. Nimmt er ein `CryptResult` aus dem Space heraus, verringert er den Wasserstand um eins. Beides geschieht mit Hilfe der Methode `changeWaterLevel`.

```
private synchronized void changeWaterLevel(int delta){
  waterlevel += delta;
  notifyAll();
}
```

Dem `waterlevel`, der zu Beginn der Berechnung null ist, wird das entsprechende Delta (1 oder -1) hinzugefügt. Danach werden alle wartenden Threads davon benachrichtigt, dass der Wasserstand sich geändert hat.

Da in 4.2.6.3 die Methode `waitForLowWaterMark` benutzt wird, kann sie auch gleich hier erklärt werden:

```
private synchronized void waitForLowWaterMark(int value) {
  while (getWaterLevel() > value) {
    try {
      if (!getSolved()){ //wait only if the problem is not yet solved
        wait();
      }
      else return;
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

Diese Methode veranlasst den aufrufenden Thread zu warten, bis der Wasserstand unter den angegebenen Wert (`value` – in diesem Fall ist es die untere Schranke) gefallen ist. Der Thread darf jedoch nur warten, wenn das Passwort noch nicht gefunden worden ist, denn sonst riskiert er, nicht mehr wieder aufzuwachen.

4.2.6.3 Der `WriteThread`

Die Arbeit dieses Threads ist sehr einfach. Er füllt, unter Einhaltung des Wasserstandes (`waitForLowWaterMark`) den Space mit Tasks: Nachdem der Wasserstand um eins erhöht wurde, wird ein `CryptTask` per Zufallsprinzip aus dem `Vector`, der alle erzeugten Tasks enthält, entfernt und in den Space geschrieben. In Abbildung A.3 ist ein Master zu sehen, der den Space mit Tasks befüllt.

```
private class WriteThread extends Thread {
  public void run() {
    startTime = System.currentTimeMillis();
    while(true){
      waitForLowWaterMark(lowmark);
      while ((getWaterLevel()<highmark) && !tasks.isEmpty() && !getSolved()) {
        int i = generator.nextInt(Math.max(1,tasks.size()));
        if (!tasks.isEmpty()){
          CryptTask task = (CryptTask)tasks.remove(i);
          changeWaterLevel(1);
          writeTask(task);
        }
      }
    }
  }
}
```

Zu Beginn der Berechnung wird die aktuelle Zeit auf `startTime` geschrieben, damit berechnet werden kann, wie lange es gedauert hat, bis das Passwort geknackt wurde. Dann wird überprüft, ob noch genug Tasks im Space sind. Wenn ja, muss der Thread warten und kann erst dann weiterarbeiten, wenn der Wasserstand unter `lowmark` gefallen ist. Der Space wird daraufhin mit Tasks gefüllt, jedoch nur so lange, bis die obere Grenze noch nicht erreicht ist, es noch zu bearbeitende Tasks gibt und das Passwort noch nicht gefunden wurde. Um ein Task auszuwählen, wird eine Zufallszahl zwischen null (inklusive) und dem Maximum[3] von eins und der Anzahl der noch verbleibenden Tasks (exklusive) erzeugt. Falls der `Vector`, der die Tasks enthält zwischenzeitlich nicht geleert wurde (vom `CollectThread`, siehe 4.2.6.4), kann nun das von der Zufallszahl bestimmte Task aus dem `Vector` entfernt, der Wasserstand erhöht und das Task in den Space geschrieben werden. Daraufhin wird wieder von vorne begonnen und überprüft, ob noch genug Tasks im Space sind. Das Schreiben eines Tasks in den Space erfolgt mit der Methode `writeTask`, die einen `TaskEntry` als Parameter nimmt und ihn dann ohne Transaktion für immer in den Space schreibt:

```
private void writeTask(TaskEntry task) {
  try {
    space.write(task, null, Lease.FOREVER);
  } catch (RemoteException e) {
    e.printStackTrace();
  } catch (TransactionException e) {
    e.printStackTrace();
  }
}
```

4.2.6.4 Der `CollectThread`

Dieser Thread ist für das Sammeln der von den Workern erzeugten Resultate zuständig. Ist das Passwort geknackt worden, so muss der `CollectThread` alle nötigen Schritte tun,

[3] Das Maximum muss genommen werden, damit keine Exception geworfen wird, falls der `Vector` zwischenzeitlich geleert wurde. Trotzdem werden mit dieser Methode alle Tasks erfasst, da die Methode `nextInt` den ersten Parameter inklusive und den zweiten Parameter exklusive behandelt.

damit keine unnötigen Berechnungen mehr gemacht werden. Ausserdem muss er den Space aufräumen und die neue Berechnung starten. Das Finden des Passworts und das Aufräumen des Space werden in der Abbildung A.4 gezeigt.

```
private class CollectThread extends Thread {
  public void run() {
    while(true){
      CryptResult result = (CryptResult)takeTask(cryptResultTemplate);
      changeWaterLevel(-1);

      if (result.word != null) {
        stopTime = System.currentTimeMillis();
        setSolved(true);
        tasks.removeAllElements();
        addPoison();
        LogFile.write("log"+triesPerTask,""+(stopTime - startTime) / 1000.0);

        takeAll("CryptTask", true);
        takeAll("CryptResult", true);
        tasks = generateTasks();
        takeAll("PoisonPill", false);
        try{
          Thread.sleep(1000);
        } catch (Exception e){
          e.printStackTrace();
        }
        cleanUpTheSpace();

        changeWaterLevel(- getWaterLevel());

        // start the new computation
        setSolved(false);
        startTime = System.currentTimeMillis();

      }
     }
    }
  }
```

Die `run`–Methode besteht lediglich aus einer einzigen Schleife. Begonnen wird damit, ein `CryptResult` aus dem Space zu nehmen und den Wasserstand zu dekrementieren. Ist das `word`–Feld dieses Resultats `null`, wird der folgende `if`–Zweig nicht ausgeführt und das nächste Resultat wird aus dem Space genommen, denn das Passwort ist noch nicht geknackt worden. Ein `CryptResult` wird aus dem Space entfernt, indem der Methode `takeTask` das Snapshot–Template des Entry `CryptResult` übergeben wird. Es wird so lange wie möglich gewartet, dass ein passender Entry gefunden wird. Der Rückgabewert ist entweder der Entry oder `null`, falls keiner gefunden werden konnte, der mit dem Parameter übereinstimmt:

```
private Entry takeTask(Entry en) {
  Entry result = null;
  try {
    result = (Entry) space.take(en, null, Long.MAX_VALUE);
```

```
    } catch (RemoteException e) {
      e.printStackTrace();
    } catch (TransactionException e) {
      e.printStackTrace();
    } catch (UnusableEntryException e) {
      e.printStackTrace();
    } catch (InterruptedException e) {
      LogFile.write("log"+triesPerTask,"CryptMaster: Task cancelled");
    }
    return result;
  }
```

Sobald der Master jedoch bei einem Resultat auf ein `word`–Feld trifft, das ungleich `null` ist, weiss er, dass das Passwort geknackt ist. Die `execute`–Methode des `CryptTask` ist nämlich so beschaffen, dass nur wenn das Passwort gefunden wurde, im zugehörenden `CryptResult` eben dieses Passwort steht. Der Master hält sofort die Zeit an und informiert den `WriteThread` darüber, dass das Passwort geknackt ist (`setSolved(true)`). Sodann kann der `Vector`, der die noch zu berechnenden Tasks enthält, geleert werden. Bevor die für das Auffinden des Passworts benötigte Zeit in eine Log–Datei geschrieben wird, müssen noch die Worker benachrichtigt werden, dass das Passwort gefunden wurde. Dazu dient die Methode `addPoison`:

```
  private void addPoison() {
    PoisonPill poison = new PoisonPill();
    try {
      space.write(poison, null, Lease.FOREVER);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
```

Bei `PoisonPill` handelt es sich um einen einfachen Entry, der im Space die Aufgabe hat, als ein Hindernis zu agieren:

```
  public class PoisonPill implements Entry {
    public PoisonPill() {
    }
  }
```

Ein `PoisonPill` im Space verhindert, dass Ressourcen verschwendet werden. Die `execute`–Methode eines `CryptTask` überprüft zuerst, ob kein solcher Entry im Space vorhanden ist. Denn wenn ein `PoisonPill` im Space ist, ist das Passwort geknackt worden und weitere Tests sind nicht mehr notwendig. Bevor ein Worker ein Resultat in den Space zurückschreibt, schaut auch er nach, ob kein `PoisonPill` im Space zu finden ist. Somit schreibt er nicht unnötig Resultate in den Space.

Der Master räumt sodann den Space auf: Alle noch nicht berechneten Tasks sowie alle auffindbaren Resultate werden herausgenommen, der Wasserstand wird kontinuierlich verringert. Danach werden neue Tasks erzeugt (`generateTasks`) und der `Vector` wird wieder aufgefüllt. Um den nun unmittelbar bevorstehenden Neustart zu ermöglichen,

wird der `PoisonPill`–Entry aus dem Space entfernt, eine Sekunde lang gewartet und dann alles, was noch im Space zu finden ist, herausgenommen. Somit wird verhindert, dass sich alte Resultate, die zwischenzeitlich vielleicht noch in den Space geschrieben wurden, mit denen der neuen Berechnung vermischen. Nachdem der Wasserstand auf null gesetzt wurde, kann das nächste Problem in Angriff genommen werden: Der Status des Problems wird auf ungelöst gesetzt (`setSolved(false)`) und die Anfangszeit wird festgelegt. Nun schreibt der `WriteThread` wieder Tasks aus dem `Vector` in den Space, und der `CollectThread` fängt wieder an, Resultate zu sammeln.

Der `CryptMaster` wird einmal vom Hauptprogramm `CryptMain` gestartet. Dort erhält er das zu knackende Passwort. Danach läuft alles automatisch:

```
public class CryptMain {
  public static void main(String[] args){
    CryptMaster master = new CryptMaster("1499999");
  }
}
```

4.2.7 Gesamtübersicht

Das folgende Schema (Abbildung 4.2) stellt den Ablauf des Programms bildlich in sechs Schritten dar:

1. Der Master schreibt n Tasks in den Space.
2. Der Worker nimmt ein Task aus dem Space.
3. Der Worker führt lokal sein Task aus (Passwörter testen), falls kein PoisonPill im Space ist.
4. Der Worker schreibt sein Resultat in den Space und kehrt zu 2. zurück, falls kein PoisonPill im Space ist.
5. Der Master nimmt ein Resultat aus dem Space.
6. Der Master schreibt ein PoisonPill in den Space, falls das Passwort gefunden wurde. Sonst kehrt er zu 1. zurück.

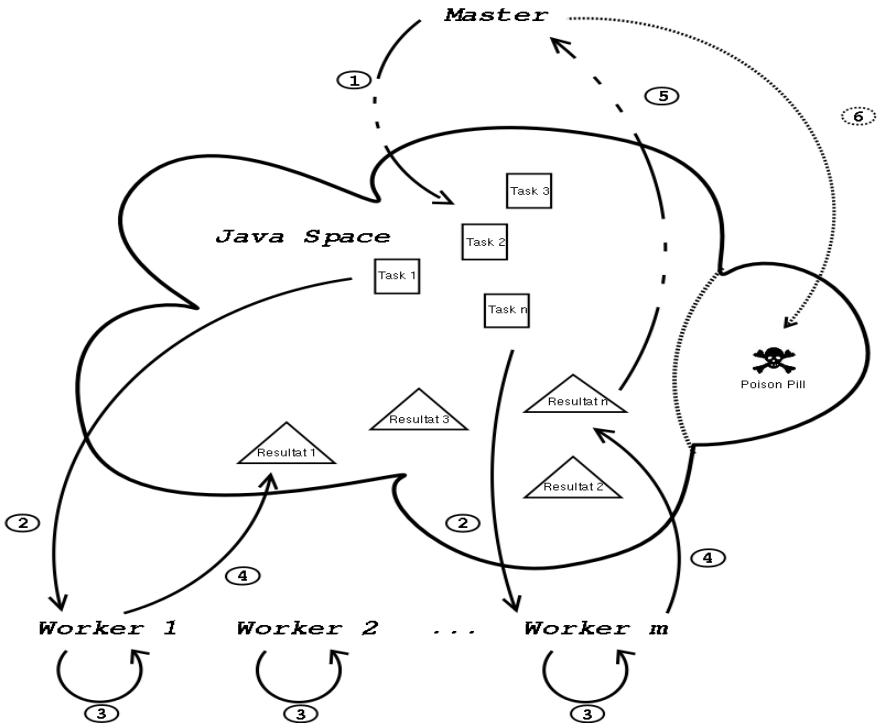

Abbildung 4.2: Interaktionen des Masters und der Worker mit dem Space

4.3 Performanz–Analyse

Im Rahmen dieser Performanz–Analyse waren zwei Parameter von Interesse: die Anzahl der Worker und die Anzahl der in einem Task zu testenden Passwörter (triesPerTask). Dies führte zu zwei Untersuchungen:

- Für das erste Experiment wurden die triesPerTask festgelegt und die Anzahl der Worker sukzessive von eins auf zwanzig erhöht.
- Die zweite Betrachtung begann mit einem triesPerTask–Wert von 600'000, der in Schritten von 15'000 verringert wurde, bis hin zu einem Wert von 1000. Fünf Worker führten die Berechnung durch.

Die Anzahl der möglichen Passwörter war bei beiden Experimenten 3'000'000. Als zu knackendes Passwort wurde 1'499'999 gewählt. Dieses Passwort musste nun 200 Mal von den Workern gefunden werden, jedes Mal wurde die benötigte Zeit vermerkt.

4.3.1 Anzahl der Worker

Innerhalb dieses Experiments stieg die Anzahl der Worker, die für das Auffinden des Passworts zur Verfügung standen, von eins auf zwanzig. Die Anzahl der in einem Task zu testenden Wörter wurde auf 60'000 fixiert. Diesen Wert erhält man, wenn man 3'000'000 (die Anzahl der möglichen Passwörter) durch 50 teilt. Denn so verfügt – selbst bei zwanzig Workern – jeder Worker über ein bisschen mehr als zwei Tasks. Um immer einen

guten „Wasserstand“ im Space zu haben, wurden die obere und die untere Schranke bei jedem zusätzlichen Worker angepasst: Untergrenze = Anzahl der Worker, Obergrenze = 2 · Untergrenze +1. So wurden zwanzig Mal 200 Werte erzeugt, aus denen jeweils der Mittelwert errechnet wurde.

Worker	Sekunden	Worker	Sekunden
1	76,06	11	17,96
2	35,54	12	15,98
3	26,47	13	18,82
4	20,55	14	18,71
5	15,79	15	18,30
6	16,66	16	19,15
7	16,18	17	18,41
8	15,87	18	15,71
9	14,67	19	20,33
10	15,82	20	20,45

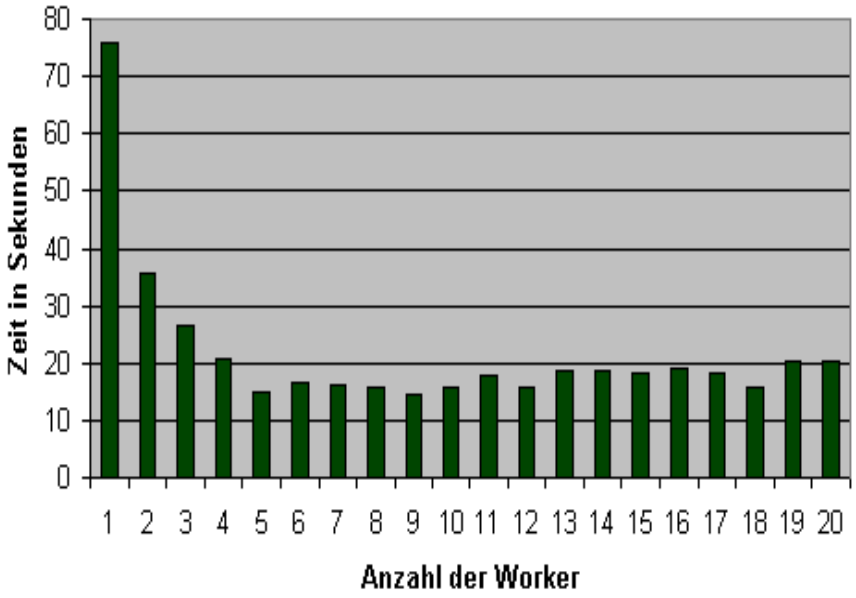

Abbildung 4.3: Erhöhung der Worker von eins auf zwanzig bei 60’000 triesPerTask

Es wird deutlich, dass ein Worker alleine sehr viel zu tun hat. Durchschnittlich benötigt er 76,06 Sekunden, um das Passwort zu knacken. Fügt man einen zweiten Worker hinzu, geht alles mehr als doppelt so schnell. Die ideale Anzahl der Worker liegt zwischen fünf und zehn, kommen noch mehr hinzu, dauert es wieder länger, bis ein Passwort gefunden ist. Jedoch steigt, wie auf Abbildung 4.3 zu sehen ist, die benötigte Zeit langsam und gleichmässig an, während sie bei einer geringen Anzahl von Workern sehr rapide sinkt.

Die Tatsache, dass die Zeit, die durchschnittlich gebraucht wird, um ein Passwort zu knacken, sich nicht kontinuierlich verringert, wenn man zusätzliche Worker hinzufügt, kann

nicht am Master liegen, denn er stellt immer die gleiche Anzahl Tasks in den Space und hat somit immer die gleiche Arbeit zu verrichten. Also muss die Ursache bei den Workern zu finden sein: Arbeitet nur ein Worker, passiert das Schreiben der Resultate sequentiell, bei zwei Workern quasi–sequentiell, die benötigte Zeit halbiert sich. Kommen aber nun zusätzliche Worker hinzu, wollen immer mehr Worker gleichzeitig auf den Space zugreifen, was zu einer Verlangsamung führt. Bei zwanzig Workern kann man zwar von einer raschen Berechnung profitieren, es wollen aber zu viele Space–Operationen gleichzeitig ausgeführt werden, was den Nutzen wieder relativiert, da der Space überlastet ist.

4.3.2 Anzahl der in einem Task zu testenden Wörter – triesPerTask

Ein Worker, der ein Task aus dem Space nimmt, hat eine bestimmte Anzahl von Wörtern auf ihre Identität mit dem Originalpasswort zu testen. Ist diese durch triesPerTask gegebene Zahl gross, werden nur wenige Tasks vom Master in den Space gestellt. Nimmt ein Worker ein Task, so ist er lange mit der Überprüfung beschäftigt, die er auf seiner eigenen Maschine, ausserhalb des Spaces, durchführt. Der Grad der Kommunikation ist sehr gering, dafür ist die Berechnungszeit für ein einzelnes Task sehr hoch. Bei einem sehr kleinen triesPerTask–Wert ist die Zeit, die gebraucht wird, um die einem Task zugeordneten Wörter zu testen, verschwindend klein. Die Anzahl der Tasks und die für das Knacken des Passworts benötigte Zeit hingegen nehmen unglaubliche Ausmasse an.

tries	**Sek.**	**tries**	**Sek.**	**tries**	**Sek.**	**tries**	**Sek.**
600’000	11,92	450’000	10,31	300’000	12,23	150’000	12,67
585’000	13,75	435’000	9,56	285’000	10,02	135’000	11,11
570’000	12,80	420’000	10,11	270’000	10,50	120’000	11,92
555’000	13,58	405’000	10,93	255’000	11,36	105’000	11,75
540’000	12,10	390’000	11,34	240’000	9,36	90’000	12,97
525’000	11,91	375’000	11,94	225’000	10,91	75’000	11,45
510’000	12,37	360’000	9,74	210’000	9,72	60’000	15,57
495’000	8,92	345’000	9,90	195’000	11,28	45’000	15,83
480’000	9,61	330’000	11,11	180’000	11,11	30’000	19,49
465’000	10,45	315’000	10,12	165’000	10,37	15’000	35,65
						1’000	584,96

Bis zu einem triesPerTask–Wert von 105’000 liegt die benötigte Zeit in etwa bei elf Sekunden (vgl. Abbildung 4.4). Danach vergrössert sie sich recht schnell. Wie in Abbildung 4.5 zu sehen ist, schiesst sie bei einem Wert von 1’000 förmlich in die Höhe.

Dieses Verhalten erklärt sich wiederum durch eine zu grosse Kommunikationsrate im Space, zu viele Operationen müssen über ihn ausgeführt werden, so dass er trotz „Watermarking“ heillos überlastet ist.

Interessant ist, dass die Zeiten von 600’000 triesPerTask bis zu 105’000 triesPerTask ungefähr übereinstimmen. Das bedeutet, dass ohne Probleme kleine triesPerTask–Werte gewählt werden können, was im Fall von teilweisem Versagen vorteilhafter ist. Der Crash einer Maschine bedeutet, dass das von ihm bearbeitete Task noch einmal berechnet werden muss. Daher ist es besser, wenn das Task nicht so gross ist, da damit Zeit gespart wird.

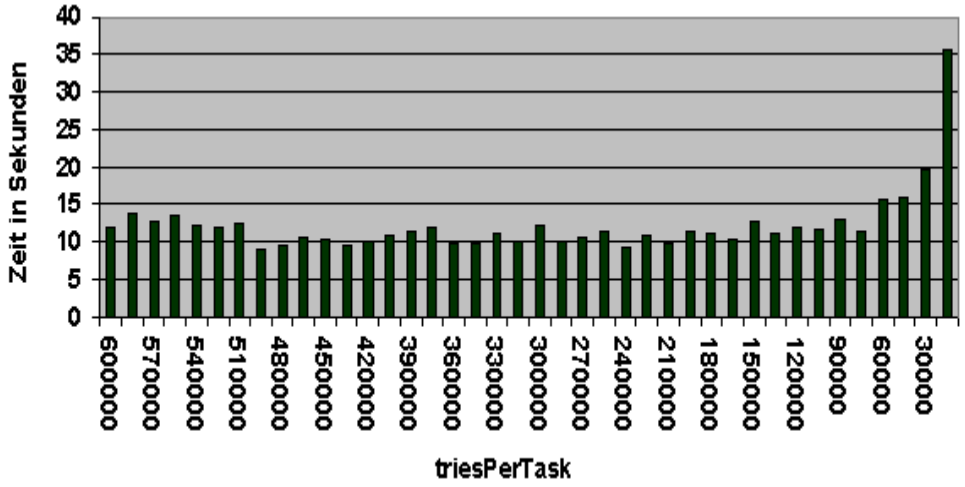

Abbildung 4.4: Verkleinerung des triesPerTask–Wertes von 600'000 auf 15'000 bei fünf Workern

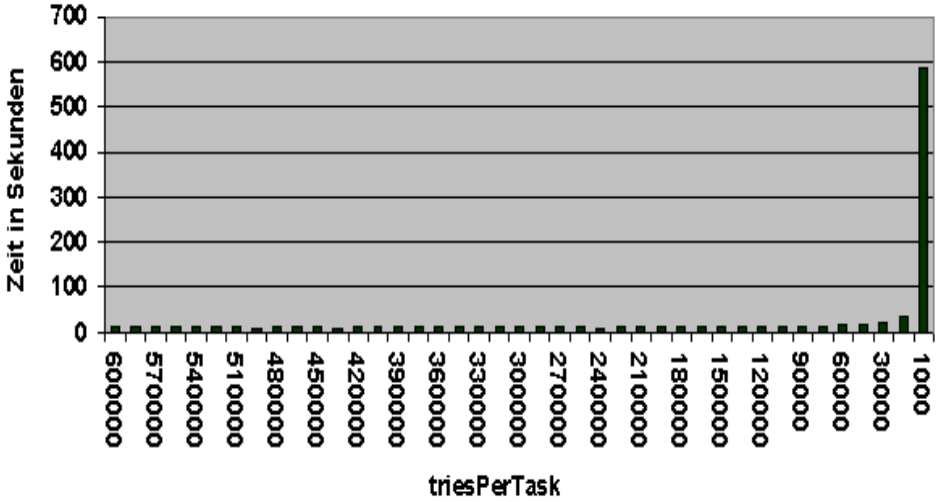

Abbildung 4.5: Verkleinerung des triesPerTask–Wertes von 600'000 auf 1'000 bei fünf Workern

4.3.3 Bilanz

Die beiden Experimente haben gezeigt, dass ein JavaSpace unter Umständen sehr sensibel reagieren kann. Sicherlich kommen die Eigenschaften der JavaSpaces–Technologie parallelen Anwendungen sehr entgegen, wie am Anfang des Kapitels erklärt. Die Kunst besteht jedoch darin, die Parameter richtig zu setzen. Der Space darf weder unterfordert werden, d. h. seine Kapazitäten nicht genügend auszunutzen, noch darf nicht zu viel von ihm zu verlangt werden, indem man zu viele Operationen auf ihm ausführt.

In diesem konkreten Fall, empfiehlt es sich mit 9 Workern und 100'000 triesPerTask zu arbeiten, um die besten Ergebnisse zu erzielen. Durchschnittlich benötigt man dann 8,6 Sekunden um ein Passwort von 3'000'000 Möglichkeiten zu knacken.

Kapitel 5

Schlussfolgerung

Diese sich über ein Jahr erstreckende Seminararbeit hat mich viel gelehrt. Es war für mich das erste Mal, an einem Projekt von dieser Grösse eigenverantwortlich zu arbeiten. Interessant war es, sich in ein unbekanntes Themengebiet selbständig einzuarbeiten, um es von Grund auf zu verstehen. Auch das Schreiben dieser Arbeit, besonders das Kapitel 2 mit seiner Einführung in Jini und dem Tutorial, waren eine gute Erfahrung. Ich musste die detaillierten Erkenntnisse, die ich im Laufe eines Jahres erworben hatte, derart zusammenfassen und formulieren, dass ein Leser, der auf diesem Gebiet kein Detailwissen hat, mich versteht. Lehrreich war für mich auch der Aspekt der selbständigen Zeiteinteilung.

Die Punkte, die ich soeben als interessant angegeben habe, finden sich aber auch unter den Problemen wieder, auf die ich während der Realisierung dieses Projekts gestossen bin. Ich habe eine lange Einarbeitungsphase in das Thema gebraucht, da man sich – will man wirklich verstehen, wie alles funktioniert – gut mit verteilten Systemen und verteilter Programmierung auskennen muss. Auch die eigenverantwortliche Zeiteinteilung war nicht einfach für mich, denn es galt zu lernen, die eigenen Fähigkeiten und Leistungsmöglichkeiten mit der zur Verfügung stehenden Zeit abzustimmen.

Der Rückblick auf dieses Projekt ist jedoch positiv, da ich dabei nicht nur Detailwissen in einem Themengebiet erlangt habe, sondern auch viel über die eigene Arbeitsweise gelernt habe.

Anhang A

Screenshots

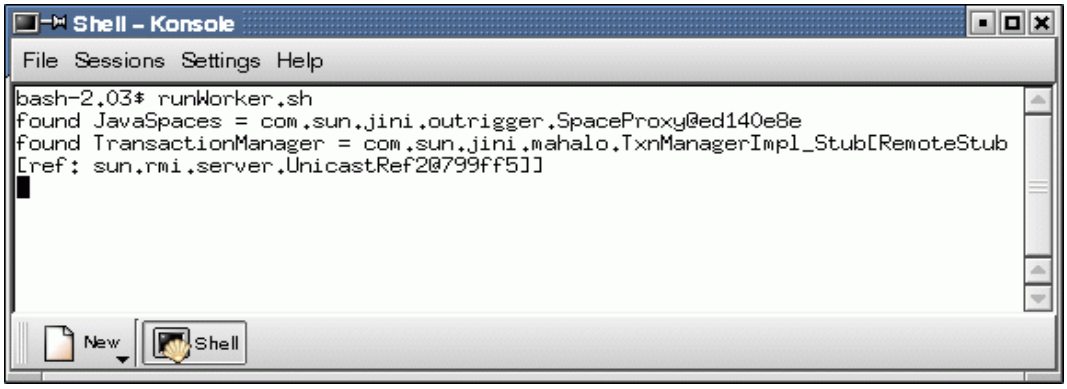

Abbildung A.1: Ein Worker, der auf Arbeit wartet

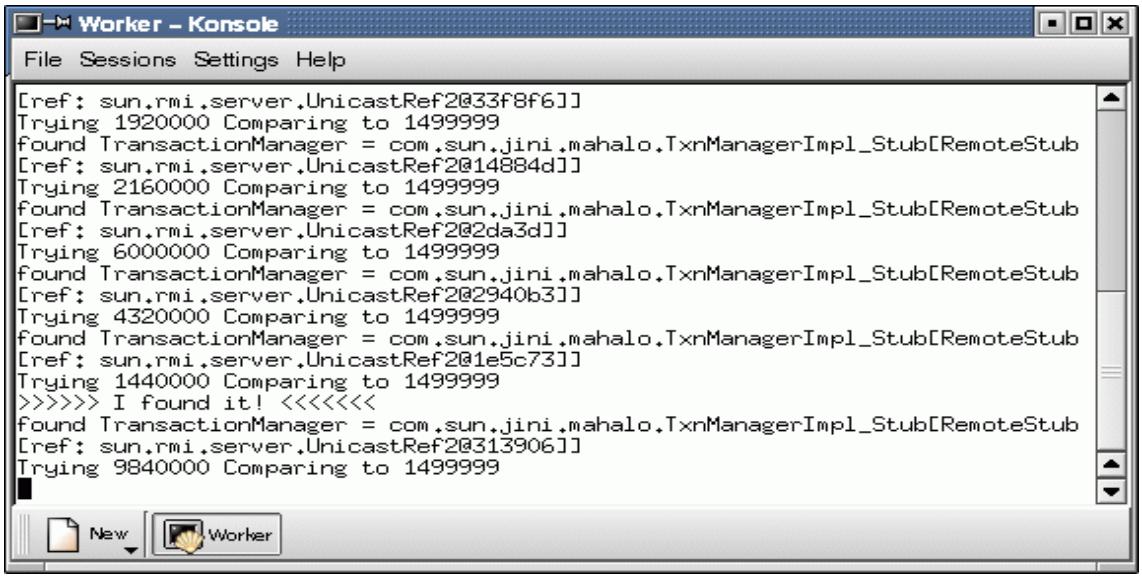

Abbildung A.2: Ein Worker, der das Passwort geknackt hat

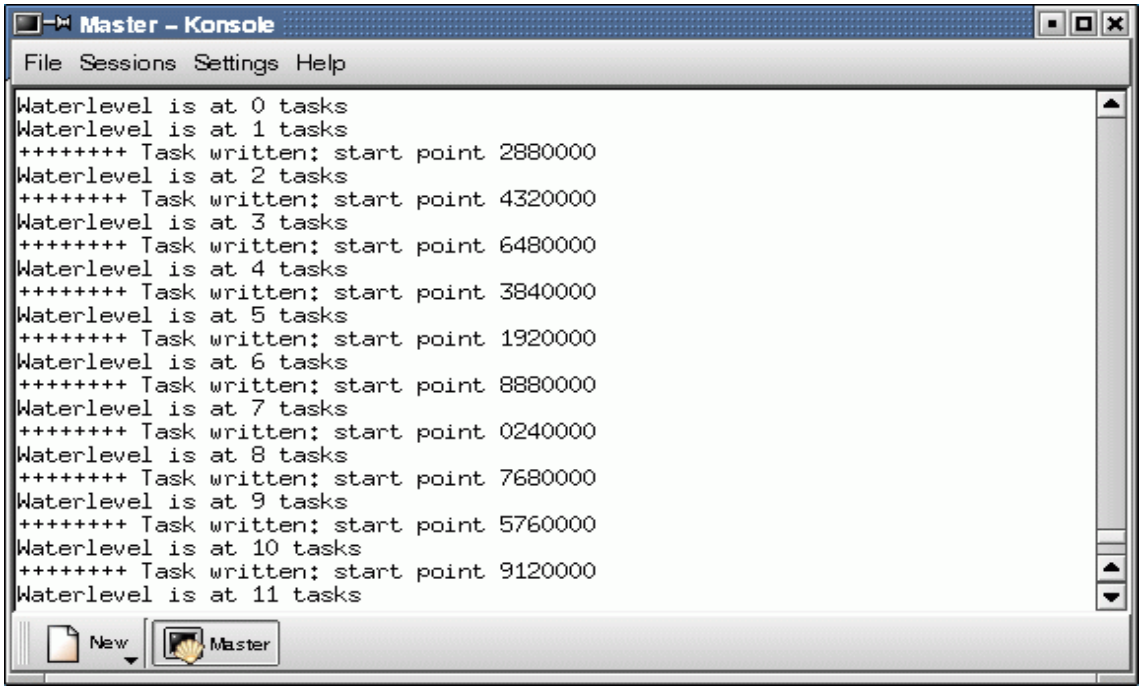

Abbildung A.3: Der Master zu Beginn der Berechnung

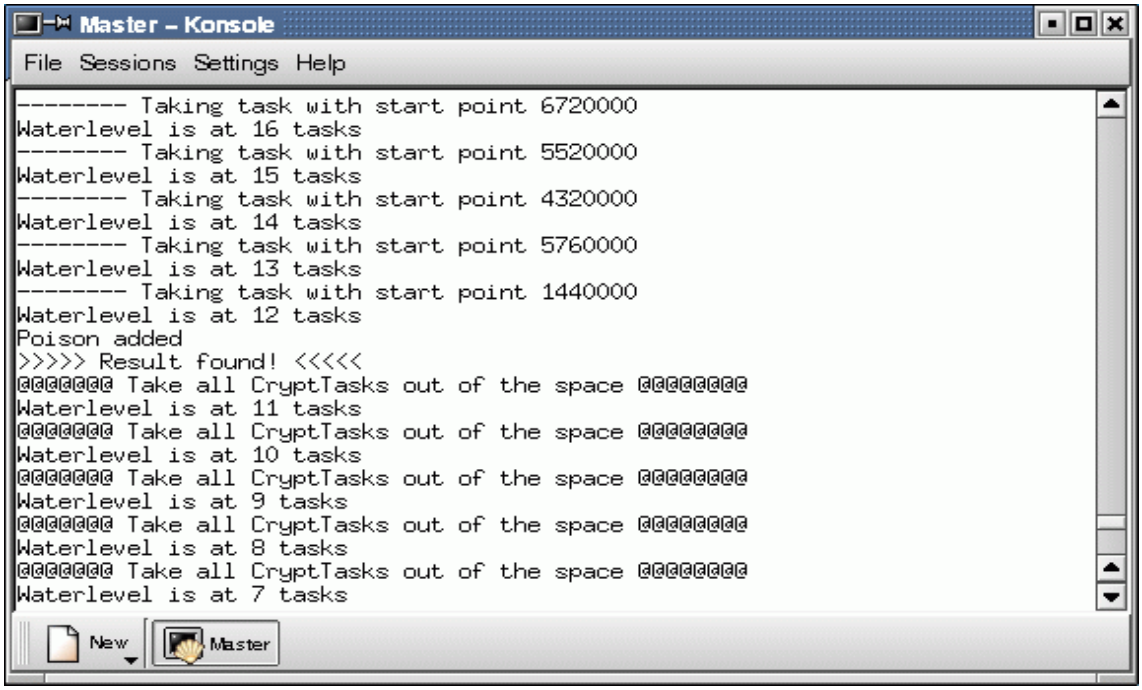

Abbildung A.4: Der Master hat das Passwort gefunden und räumt im Space auf

Anhang B

Webseite der Arbeit

Unter

`http://diuf.unifr.ch/softeng/student-projects/completed/langel/`

ist eine Kurzbeschreibung dieses Projektes zu finden, wie auch der Quellcode, die kompilierten Dateien und die Skripte, die in dieser Arbeit aufgeführt wurden. Ausserdem steht dort dieses Dokument im PDF-, PS- wie auch im HTML–Format zur Verfügung. Auch die JavaDoc–Dokumentation der in Kapitel 4 vorgestellten parallelen Anwendung ist unter dieser Adresse zu finden.

Anhang C

CD–ROM

Die diesem Dokument beigefügte CD–ROM hat folgenden Inhalt:

- Dokumentation (Verzeichnis `doku/`)

  `jini.ps` Seminararbeit im PS-Format

  `jini.pdf` Seminararbeit im PDF-Format

  `html/` Seminararbeit in HTML-Format

  `jini.tex` LaTeX–Quellcode dieser Arbeit

  `javadoc/` JavaDoc Dokumentation der parallelen Anwendung

- Dateien (Verzeichnis `dat/`)

  `bspjini.tar.gz` Beispiele zur Jini-Technologie: Quellcode, kompilierte Dateien und Skripte

  `bspjs.tar.gz` Beispiele zur JavaSpaces-Technologie: Quellcode, kompilierte Dateien und Skripte

  `crypt.tar.gz` Quellcode der parallelen Anwendung, kompilierte Dateien sowie Skripte zur Kompilation und Ausführung der Anwendung

- Webseite (Verzeichnis `web/`)

  `index.html` Webseite, die unter dem in Anhang B genannten Link zu finden ist

  `doku/` oben detailliertes Verzeichnis `doku/`, ohne `jini.tex`

  `dat/` oben detailliertes Verzeichnis `dat/`

Literaturverzeichnis

[Edwards01] W. Keith Edwards: Core Jini. 2nd ed. Java™Series. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001

[FreemanHupferArnold99] Eric Freeman, Susanne Hupfer und Ken Arnold: JavaSpaces™. Principles, Patterns, and Practice. The Jini™Technology Series. Reading, Massachusetts, USA: Addison Wesley Longman, Inc., 1999

[Li00] Sing Li: Professional Jini. Birmingham, UK: Wrox Press Ltd, 2000

[Mahmoud00] Qusay H. Mahmoud: Distributed Programming with Java. Greenwich, CT, USA: Manning Publications Co., 2000

[OaksWong00] Scott Oaks und Henry Wong: Jini™in a Nutshell. A Desktop Quick Reference. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2000

[MakeRoom] Susanne Hupfer: Make room for JavaSpaces. Part 5: JavaWorld.com `http://www.javaworld.com/javaworld/jw-06-2000/jw-0623-jiniology-p1.html` (September 2002)

[datasheet] Jini–Datasheet 0601 `http://wwws.sun.com/software/jini/whitepapers/jini-datasheet0601.pdf` (August 2002)

[developer] Jini(TM) Network Technology Developer Center `http://developer.java.sun.com/developer/products/jini/` (August 2002)

[glossary] Jini Technology Glossary `http://www.sun.com/2000-0829/jini/glossary.html` (August 2002)

[JiniHeute] Die Jini(TM) Technologie heute `http://ch.sun.com/d/products/03_jini.html` (August 2002)

[JiniHeute2] Die Jini(TM) Technologie heute (2) `http://ch.sun.com/d/products/03b_jini.html` (August 2002)

[JiniToday] The Jini(TM) Technology today `http://www.sun.com/2000-0829/jini/works.html` (August 2002)

[jini.org] Jini.org `http://www.jini.org/` (August 2002)

[JSTechnology] JavaSpaces(TM) Technology
`http://java.sun.com/products/javaspaces/` (August 2002)

[NutsBolts] The Nuts and Bolts of Compiling and Running JavaSpaces(TM) Programs
`http://developer.java.sun.com/developer/technicalArticles/jini/javaspaces/index.html` (August 2002)