Seminar - Programmierung verteilter Systeme

Thema: *Jini*™

Autor: Andreas Tonder
Betreuer: Frank Huch

26. Mai 2003

Inhaltsverzeichnis

1 Einleitung

Verteilte Systeme - insbesondere Netzwerke - spielen in der Welt der Informatik eine bedeutende Rolle. Jedoch werden Programmierer, die sich mit derartigen Systemen auseinandersetzen, mit einer Reihe von Schwierigkeiten und Problemen konfrontiert, die bei lokalen Systemen meistens nicht auftreten oder auftreten können. Solche Probleme sind z.B. ([1], Kapitel 2):

- Verzögerungen bei der Datenübertragung
- Ausfälle von Rechnern im Netzwerk
- Berücksichtigung von Nebenläufigkeit mit Fehlern wie Dead- und Lifelocks
- verteilte Speicherverwaltung

Aufgrund dessen benötigt man geeignete Netzwerk-Infrastrukturen und programmiersprachliche Hilfsmittel, um diese Probleme in den Griff zu bekommen.
Die Jini-Technologie, die im folgenden vorgestellt werden soll, stellt eine eigene Netzwerk-Infrastruktur und Programmierwerkzeuge auf Java-Basis bereit, und versucht dadurch einem Programmierer das Lösen der oben erwähnten Probleme einfacher zu machen.

2 Grundlagen

2.1 Java

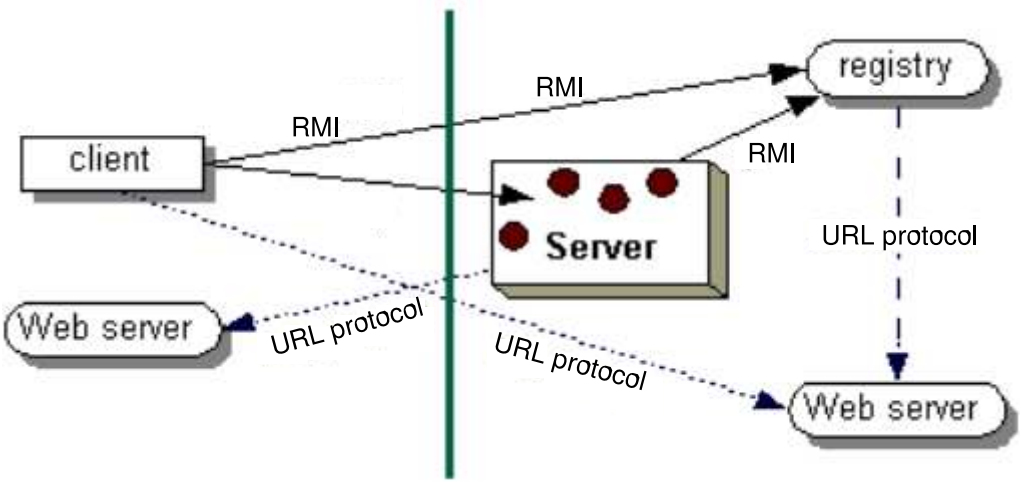
Java ist eine Programmiersprache, die speziell für die verteilte Umgebung des Internets entwickelt wurde. Sie kann benutzt werden, um eigenständige Applikationen zu programmieren, die auf einem lokalen Rechner oder auf Server bzw. Client in einem Netzwerk laufen.
Hauptaspekte von Java:

- Java-Programme sind portabel. Ein Quellprogramm wird in sog. *Bytecode* kompiliert, der auf jedem Rechner ausgeführt werden kann, der eine *Java Virtual Machine* (JVM) besitzt. Eine JVM interpretiert den Bytecode.
- Java ist objekt-orientiert.
- Java orientiert sich sehr nah an C++.

Grundlagen und Beispiele zu Java werden in [4] vermittelt.

2.2 Remote Method Invocation (RMI)

RMI ermöglicht das Ausführen von Java-Anwendungen innerhalb unterschiedlicher JVMs, wobei sich Anwendungen untereinander verständigen und sogar auf unterschiedlichen Hosts ausgeführt werden können. Ein Client-Programm kann damit eine oder mehrere Methoden eines Objekts aufrufen, welches sich auf einem anderen Rechner befindet (Remote-Objekt). Die Verwaltung und Registrierung von Remote-Objekten übernimmt dabei die sog. `rmiregistry`.
Die folgende Abbildung veranschaulicht die Arbeitsweise von RMI etwas detailierter:



Die Abbildung zeigt eine verteilte RMI Anwendung, welche die Registrierung benutzt, um an eine Referenz eines Remote-Objekts zu gelangen. Der Server wendet sich an die `rmiregistry`, um ein Remote-Objekt an einen Namen zu binden. Der Client kann nun die `rmiregistry` fragen, ob sich ein Remote-Objekt mit einem bestimmten Namen registriert hat. Wenn ja, dann erhält der Client das gewünschte Objekt. Die Abbildung zeigt ausserdem, dass das RMI-System einen existierenden Web-Server benutzt, um Bytecode der Java-Klassen zwischen Server und Client auszutauschen, falls dieses notwendig ist. Weiterführende Betrachtungen zum RMI-Mechanismus findet man in [7].

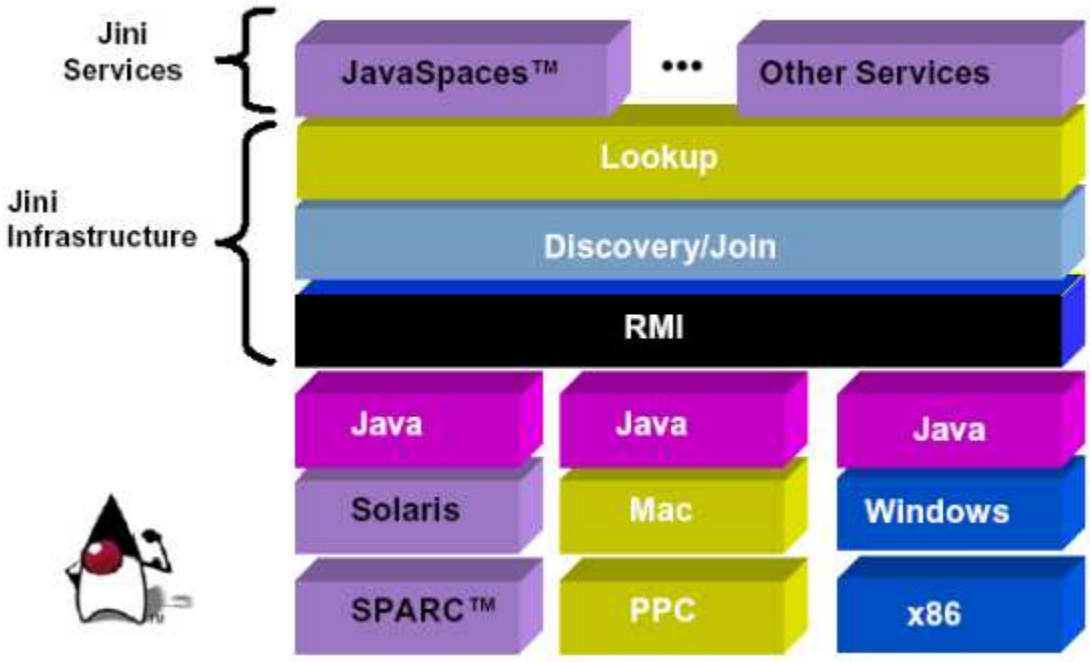
3 Jini

3.1 Was ist Jini?

Jini ist eine Technologie der Firma SUN (auf Basis der Java 2 Platform), die den Aufbau von Netzwerken durch Bereitstellung einer eigenen Infrastruktur fördern soll. Sie wurde 1999 erstmalig veröffentlicht und stellt Schnittstellen und Protokolle zur Verfügung, die die verteilte Programmierung in diesen Netzwerken unterstützt. Die Elemente dieser Netzwerke sind Software-Dienste und Hardware-Geräte oder beides zusammen, die spontan zu sog. Gemeinschaften (*federations*) zusammengefasst werden können. Hierbei sei darauf hingewiesen, dass die Hardware-Elemente nicht zwangsläufig vollwertige PCs sein müssen. Es kann sich durchaus auch um kleine, weniger leistungsstarke Geräte (wie z.B. Digitalkameras) handeln. Außerdem stellt Jini eine Art ”Selbstheilungs-Mechanismus” bereit, der das sichere Entfernen von Geräten aus solchen Gemeinschaften handhabt. Jegliche Software- und Hardware-Dienste werden durch Java-Objekte repräsentiert und das Auffinden dieser sowie der Zugriff auf diese geschieht über Java-Interfaces.

3.2 Bestandteile von Jini

Die folgende Abbildung zeigt eine Übersicht über die Architektur von Jini.



3.2.1 Infrastruktur

Die Jini-Infrastruktur beschreibt den Kern von Jini selbst und umfasst im wesentlichen folgende Punkte:

1. ein verteiltes Sicherheitssystem, das unerlaubte Zugriffe auf bestimmte Resourcen verhindert und in RMI integriert ist (wird hier nicht weiter betrachtet)
2. Discovery-/Join-Protokoll: ermöglicht Java-Objekten den Zugang und die Teilnahme an einem Jini-Netzwerk
3. den Lookup-Service, welcher Jini-Dienste verwaltet und für Applikationen oder für andere Jini-Dienste verfügbar macht

Discovery-/Join-Protokoll

Um einer Jini-Einheit (Dienst oder Anwendung) die Teilnahme an einem Netzwerk zu ermöglichen, muss diese zunächst eine Jini-Gemeinschaft finden, die diese aufnehmen kann (*Discovery*). Als Ergebnis einer Suche nach einer Jini-Gemeinschaft erhält eine Einheit eine oder mehrere Referenzen auf verfügbare Lookup-Services. Die Anmeldung an einem oder mehreren dieser Lookup-Services wird als *Join* bezeichnet.
Um einen Discovery-Vorgang zu starten, benötigt man die Klasse `LookupDiscoveryManager`, deren Konstruktor folgendermaßen aussieht:

```
LookupDiscoveryManager(String[] groups, LookupLocator[] locators,
                       DiscoveryListener listener)
```

Der erste Parameter spezifiziert, welche Jini-Gemeinschaften (repräsentiert durch Strings) nach Lookup-Services durchsucht werden sollen. Der zweite gibt an, auf welchen Hosts der Discovery-Vorgang durchgeführt werden soll. Ein `LookupLocator` ist dabei nichts anderes als eine URL, die auf einen bestimmten Host verweist. Der dritte Parameter wird verwendet, um die suchende Jini-Einheit zu benachrichtigen, falls ein Lookup-Service gefunden wurde. Der `DiscoveryListener` wird in späteren Kapiteln genauer vorgestellt.

Zur Kommunikation mit einem Lookup-Service existieren drei verschiedene Protokolle:

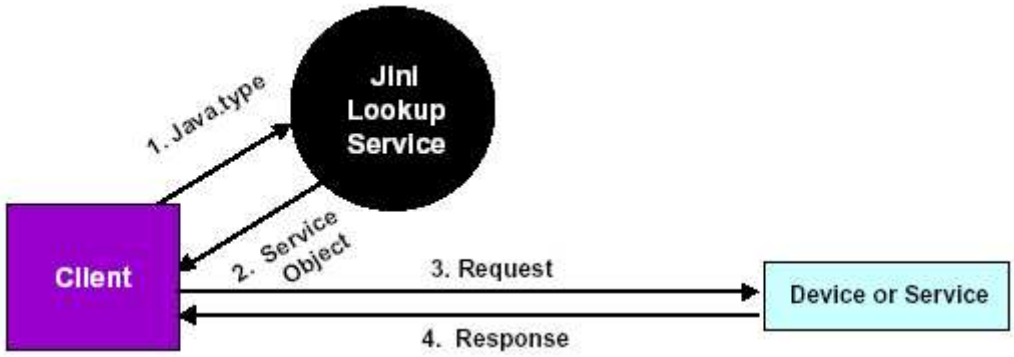
1. Multicast Request Protocol
   Dieses wird bei der Aktivierung eines Dienstes bzw. einer Applikation verwendet, um benachbarte aktive Lookup-Services zu ermitteln.

2. Multicast Announcement Protocol
   Dieses wird von Lookup-Services verwendet, die ihr eigenes Vorhandensein im Netz ankündigen möchten. Beim Starten von Lookup-Services werden alle interessierten Dienste mit Hilfe dieses Protokolls darüber informiert.

3. Unicast Discovery Protocol
   Dieses wird verwendet, falls einem Dienst bzw. einer Applikation ein bestimmter Lookup-Service schon bekannt ist, d.h. es wird direkt miteinander kommuniziert.

Lookup-Service

Ein Lookup-Service (LUS) hat die Aufgabe, die an einer Gemeinschaft teilnehmenden Dienste in einem Jini-Netzwerk zu erfassen und für andere Dienste bzw. Applikationen verfügbar zu machen. Die folgende Abbildung illustriert, wie Clients über einen LUS an einen Dienst gelangen:



Sucht eine Jini-Einheit nach einem bestimmten Service, so muss diese eine Anfrage an einen LUS stellen. Der LUS kann nach konkreten Services, Typen von Objekten (wie in der Abb. dargestellt), Superklassen, übergeordneten Schnittstellen und Attributen der vom LUS erfassten Dienstelemente durchsucht werden. Nach einer Suche erhält der Client vom LUS ein Service-Objekt, mit welchem er dann direkt (per Unicast) mit dem gewünschten Service kommunizieren kann.

Ein Suchvorgang geschieht mit Hilfe der Klasse `ServiceTemplate`, indem man eine Instanz dieser entsprechend der Suchkriterien initialisiert:

```
public class ServiceTemplate{
    public ServiceID serviceID; // Suche nach bestimmtem Service
```

```
    public Class[] serviceTypes; // Suche nach bestimmtem Objekttypen
    public Entry[] attributeSetTemplates; // Suche nach bestimmten Attributen
    // Konstruktor
    public ServiceTemplate(ServiceID serviceID,
                           java.lang.Class[] serviceTypes,
                           Entry[] attrSetTemplates)) { ... }
   ...
}
```

Anmerkung zu `Entry`:
Implementiert ein Attribut eines Dienstes dieses Marker-Interface, so kann nach diesem Attribut explizit gesucht werden.

Ein Beispiel:
Angenommen eine Applikation sucht nach einem Netzwerk-Drucker, repräsentiert durch die Klasse `Printer`.
Nach einem Discovery-Prozess hat diese eine Referenz `lusvc` des Typs `ServiceRegistrar` auf einen Lookup-Service erhalten. Um jetzt nach dem gewünschten Drucker zu suchen, muss zunächst ein Template folgendermaßen initialisiert werden:

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

Der erste sowie der dritte Parameter ist `null`, da hier keine Suche nach einem eindeutig bestimmten Service bzw. nach Attributen stattfinden soll. Stattdessen wird nach Objekten des Typs `Printer` gesucht, was durch den zweiten Parameter ausgedrückt wird.
Dann wird dieses Template der `lookup`-Methode des Lookup-Services übergeben:

```
Object o = lusvc.lookup(template);
```

War die Suche erfolgreich, so ist `o` and das erhaltene Printerobjekt gebunden und die Applikation kann versuchen dieses auf `Printer` zu casten, um dessen Schnittstelle zu benutzen.

3.2.2 Programmiermodell

Das Jini-Programmiermodell verfolgt im wesentlichen zwei Ziele. Zum einen soll es die Entwicklung von verteilten Systemen vereinfachen, zum anderen einen einheitlichen Programmierstil über alle Jini-Applikationen ermöglichen. In dieser Ausarbeitung werden folgende Aspekte betrachtet:

1. Leasing: die Zusicherung zur Nutzung eines Service oder Anmeldung bei einem Lookup-Service über einen bestimmten Zeitraum
2. Remote-Ereignisse: Benachrichtigung von Jini-Diensten oder Applikationen über bestimmte Ereignisse in einem Jini-Netzwerk
3. Transaktionen: ermöglichen die unteilbare Ausführung von nicht-atomaren Operationen

Weiterführende und hier nicht betrachtete Aspekte zum Programmiermodell von Jini, wie z.B. Landlord-Leasing und verschachtelte Transaktionen, sind in [1], [2], [6] und [8] zu finden.

Leasing

Da in Netzwerken ständig die Gefahr besteht, dass Teile ausfallen, stellt Jini einen Selbstheilungsmechanismus bereit, der solche Situationen handhabt. Das Leasing sichert die Nutzung einer bestimmten

Resource über einen bestimmten Zeitraum zu. Es bietet eine konsistente Möglichkeit, um nicht mehr benötigte oder abgestürzte Jini-Dienste aus einem Netzwerk zu entfernen.
Ein Jini-Dienst, der sich bei einem LUS registrieren möchte, gibt eine Zeitspanne an, wie lang er die Resource beim LUS nutzen möchte (muss beim Aufruf der Methode `register(ServiceItem i, long leaseTime)` des LUS geschehen). Dann hängt es vom LUS ab, ob dieser ihm diese Zeitspanne gewährt, d.h. entweder stimmt er zu oder er setzt die Zeitspanne herab.
Das folgende Beispiel zeigt wie ein Client, der sich bei einem LUS registriert, die tatsächliche Lease-Zeit seiner Registrierung (`ServiceRegistration`) in Erfahrung bringen kann.

```
ServiceRegistration reg = lookupService.register(...);
Lease lease = reg.getLease();
long duration  = lease.getExpiration()
                   - System.currentTimeMillis();
```

Will ein Jini-Dienst über einen längeren Zeitraum den LUS nutzen, so muss sichergestellt sein, dass der Lease rechtzeitig verlängert wird. Hierfür existieren zwei Möglichkeiten. Entweder sorgt der Jini-Dienst selbst dafür (z.B. in einem Thread), oder er setzt einen `LeaseRenewalManager` ein, der eigenständig dafür sorgt, dass der Lease verlängert wird, sofern der ihn beauftragende Dienst nicht abgestürzt ist.

Beispiel für die Verlängerung einer Registrierung um 10sec:

```
ServiceRegistration reg = lookupService.register(...);
 ..
Lease lease = reg.getLease();
lease.renew(10000);
```

Remote-Ereignisse

Um Java-Objekte (insbesondere Jini-Dienste) über externe Zustandsänderungen in einem Netzwerk zu informieren, werden Remote-Ereignisse verwendet. Dieses ist eine Erweiterung der Java-Standard-Ereignisse, welche den Anforderungen in verteilten Systemen gerecht werden sollen. Bei verteilten Systemen können erheblich mehr Fehler und Schwierigkeiten auftreten als bei lokalen Systemen, wie z.B.

- Verzögerung zwischen Auslösung eines Events und Benachrichtigung des Empfängers
- Vertauschung der Reihenfolge von Events, infolge von Verzögerungen im Netzwerk
- Verlust von Events im Netzwerk, ausgelöst durch einen Teilausfall

Das Jini-Ereignismodell stellt Werkzeuge bereit, um diese Probleme zu handhaben.

Ein Event wird in Jini durch die Klasse `RemoteEvent` repräsentiert. Um ein Objekt für ein Ereignis zu registrieren, muss es das Interface `RemoteEventListener` implementieren und sich zudem bei einem Ereigniserzeuger anmelden. Das `RemoteEventListener`-Interface stellt die Methode `notify(RemoteEvent ev)` zur Verfügung, die von einem Ereigniserzeuger bei einem eingetretenen Event aufgerufen wird. Das Listener-Objekt benutzt diese zur Ereignisbehandlung.
Die Vorgehensweise wie Ereigniserzeuger zu programmieren sind, ist in Jini nicht durch Interfaces geregelt, sondern basiert vielmehr auf Richtlinien. Ein Ereigniserzeuger sollte eine Methode zur Verfügung stellen, mit der sich Objekte bei ihm für Events registrieren können, wie etwa

```
public EventRegistration addRemoteEventListener(RemoteEventListener l);
```

Das Jini-Objekt `EventRegistration` sollte er zur Speicherung einer Registrierung benutzen. Nach der Generierung eines neuen `RemoteEvents` sollte der Erzeuger alle registrierten Listener-Objekte durch Aufruf der `notify(...)`-Methode informieren.

Transaktionen

Ein für die verteilte Programmierung sehr interessantes Konzept stellt die Möglichkeit dar, Transaktionen durchzuführen. Eine Transaktion ist die Gruppierung von mehreren zusammenhängenden Operationen, deren Ausführung lediglich zwei Resultate zulassen: entweder werden alle Operationen erfolgreich ausgeführt oder alle schlagen fehl. D.h., ist nur eine Operation fehlerhaft, so werden alle bisher getätigten Operationen wieder rückgängig gemacht, und somit der Zustand des Systems vor Beginn der Transaktion wiederhergestellt. Transaktionen ermöglichen Datenmanipulationen, die den sogenannten ACID-Eigenschaften (ACID: Atomicity, Consistency, Isolation, Durability) genügen:

- Atomicity: atomare Ausführung von mehreren Operationen
- Consistency: nach Durchführung einer Transaktion befindet sich das System in einem definierten Zustand
- Isolation: während der Durchführung einer Transaktion sind die Änderungen an dem System für andere unsichtbar, erst nach der Ausführung können andere diese sehen
- Durability: nach Durchführung einer Transaktion können die Änderungen nicht durch eine nachfolgende Störung oder Absturz des Systems verlorengehen

Jini verwendet bei Transaktionen das sogenannte Two-Phase-Commit Protokoll. Initiiert ein Client eine Transaktion, so weist der `TransactionManager` alle Teilnehmer (diese implementieren die Schnittstelle `TransactionParticipants` mit ihren Methoden `prepare(..)`, `commit(..)`, `abort(..)`) an, ihre Berechnungen durchzuführen und temporär zu speichern (durch Aufruf der `prepare(..)`-Methode). Dieses nennt man *Precommit-Phase*. Hat ein Teilnehmer die Berechnungen durchgeführt, so teilt er dem Manager mit, ob diese erfolgreich waren oder nicht. Im Falle eines Misserfolges veranlasst der Transaktionsmanager den Abbruch aller Operationen (`abort(..)`-Methode). Sind alle Operationen erfolgreich abgelaufen, so beginnt die zweite sogenannte *Commit-Phase*.
In der Commit-Phase werden die Transaktions-Teilnehmer angewiesen, die Ergebnisse ihrer Berechnungen in den permanenten Speicher zu schreiben (durch Aufruf der `commit(..)`-Methode). Im Fehlerfall werden auch hier die Teilnehmer aufgefordert, ihre Operationen abzubrechen und den vorherigen Zustand wiederherzustellen.

Vorgehensweise für das Verwenden einer Transaktion:

1. Zunächst erlangt ein Client eine Referenz auf einen Jini-Transaktionsmanager (TM), was normalerweise durch ein Discovery und Lookup-Prozess geschieht, da es sich bei einem TM um einen Jini-Service handelt.
2. Dann ruft der Client `TransactionFactory` auf, um ein neues `Transaction`-Objekt zu erstellen, welches vom Transaktionsmanager verwendet werden kann.

3. Anschliessend gruppiert der Client die an der Transaktion teilnehmenden Operationen, indem er ihnen das `Transaction`-Objekt als Parameter beim Methodenaufruf übergibt.

4. Nach der Gruppierung ruft der Client die `commit()`-Methode des `Transaction`-Objekts auf, um die Ausführung der Operationen zu veranlassen. Der Transaktionsmanager startet dann den Two-Phase-Commit-Prozess, der dann die Kommunikation mit den Transaktionsteilnehmern (Programme, die die Operationen ausführen) beginnt.

Ein Beispiel hierzu wird im Abschnitt JavaSpaces gegeben.

3.2.3 Jini-Services

Ein Service (oder auch Dienst) in der Jini-Architekur ist eine Einheit, welche von menschlichen Benutzern, einem Programm oder anderen Services genutzt werden kann. Ein Jini-Service kann eine beliebige Funktionalität besitzen. Er muss einige wenige Voraussetzungen erfüllen, um in einem Jini-Netzwerk verfügbar zu sein:

- er muss die Verbindung zu einem TCP/IP-Netzwerk aufbauen
- er muss den oben beschriebenen Discovery-/Join Vorgang durchführen und mindestens einen LUS finden
- er muss sich bei einem LUS registrieren
- er muss Leases verlängern

Im wesentlichen gibt es drei verschiedene Hardware-Systeme auf denen ein Jini-Dienst betrieben werden kann:

- auf allgemeinen Rechnern im Netzwerk, die mittelmäßig oder viel Rechenleistung besitzen und durchaus in der Lage sind viele Dienste anzubieten
- auf Java-fähigen Geräten, die eine eingebettete JVM besitzen
- auf Geräten ohne JVM, die durch Vorschaltung eines Jini-Proxy in ein Jini-Netzwerk eingebunden werden

Grobe Vorgehensweise, um einen Dienst an einen LUS zu binden:

1. Implementierung einer inneren Klasse `Listener` zur Überprüfung auf Discovery-Ereignisse, die die Schnittstelle `DiscoveryListener` mit ihren Methoden
   `public void discovered(DiscoveryEvent ev)` und
   `public void discarded(DiscoveryEvent ev)` implementiert.
   Die Methode `discovered(...)` wird aufgerufen, falls ein oder mehrere Lookup-Services gefunden wurden (dargestellt durch ein Array des Typs `ServiceRegistrar[]`). Der Dienst sollte dafür sorgen, dass er sich bei den gewünschten LUS über die Methode
   `register(ServiceItem item, ...)` registriert.
   Die Methode `discarded(...)` wird aufgerufen, falls ein LUS explizit beendet wird.

2. Erzeugung eines Objekts `item` des Typs `ServiceItem`, welches das Proxy-Objekt (Schnittstelle für Applikationen) des Dienstes enthält. Dieses wird bei der Registrierung dem LUS übergeben.

3. Erzeugung eines Objekts des Typs `LookupDiscovery` mit Angabe der zu Durchsuchenden Jini-Gemeinschaften (repräsentiert durch Strings). Mit der Methode `addDiscoveryListener` dieses Objekts muss dann der oben beschriebene `Listener` hinzugefügt werden, um den Dienst zu benachrichtigen, falls eine LUS gefunden wurde.

3.3 Ein Jini-Dienst: JavaSpaces

JavaSpaces stellt eine Möglichkeit zur verteilten Speicherung von Java-Objekten bereit und unterstützt somit die verteilte Programmierung. Es basiert größtenteils auf ein Kommunikationsmodell namens *Linda*, das von den Informatikern David Gelernter und Nicholas Carriero an der Yale Universität entwickelt wurde (siehe [5]). Dieses wird hier zunächst näher erläutert:

Linda

Linda ist ein Programmiersprachen-unabhängiges Modell zur Kreierung und Koordination von Prozessen entweder auf einem lokalen System oder in einem Netzwerk.
Es basiert auf der sogenannten ”generative communication” , d.h. Prozesse tauschen keine Nachrichten aus und teilen sich keine Variablen, sondern erzeugen Datenobjekte (sog. *Tupel*) und legen diese in einem dafür vorgesehenen Speicherplatz ab (sog. *Tupelraum*). Diese Tupel können dann von anderen Prozessen gelesen oder entfernt werden. Desweiteren können Prozesse neue Prozesse erzeugen und diese dann in einem Tupelraum ablegen.
Folgende Grundoperationen auf Tupelräumen sind vorgesehen:

- `out(t)`
  Legt ein Tupel `t` nach dessen Auswertung in dem Tupelraum ab.
  Bsp.: `out(″A″,2*2,5)` fügt das Tupel `(″A″,4,5)` ein

- `in(t)`
  Entfernt das Tupel `t` aus dem Tupelraum, falls vorhanden. Falls nicht vorhanden, wird solange blockiert bis `t` von einem anderen Prozess in den Tupelraum gelegt wurde. `t` kann auch ein Muster sein. Dann wird Pattern-Matching über den im Tupelraum vorhandenen Tupeln durchgeführt und ein passendes Tupel zurückgeliefert.
  Bsp.: `in(″A″,?var1,?var2)` entfernt das oben eingefügte Tupel und setzt `var1:=4` und `var2:=5`

- `rd(t)`
  Wie `in(t)`, nur ohne Entfernung des Tupels `t`.

- `eval(t)`
  Erzeugt einen neuen Prozess, der `t` nach dessen Auswertung im Tupelraum ablegt.

Beispiel: Dinierende Philosophen in C-Notation, siehe auch [5]

```
// Prozess eines Philosophen
// zum Essen benötigt er das linke
// und rechte Ess-Staebchen
phil(i){
  while(TRUE){
    think();
```

```
    in("chopstick", i);
    in("chopstick", (i+1) mod 5);
    eat();
    out("chopstick", i);
    out("chopstick", (i+1) mod 5);
  }
}

// Erzeuge 5 Philosophen
init(){
  for(i=0;i<5;i++){
    out("chopstick",i);
    eval(phil(i));
  }
}
```

JavaSpaces

Die Schnittstelle eines `JavaSpace` ($\approx$ Tupelraum) umfasst im wesentlichen die Grundoperationen des Linda-Modells bis auf einige Ausnahmen:

- `write(..)`$\triangleq$ `out(..)`
- `take(..)`$\triangleq$ `in(..)`
- `takeIfExists(..)`, blockiert nicht, falls das gesuchte Objekt nicht vorhanden ist
- `read(..)`$\triangleq$ `rd(..)`
- `readIfExists`, blockiert nicht, falls das gesuchte Objekt nicht vorhanden ist
- `notify(Entry template, ..)`, benachrichtigt ein Java-Objekt, falls der Eintrag, der dem Muster von `template` entspricht, in den JavaSpace geschrieben wurde (Funktionsweise des Pattern-Matchings s.u.).

Die `eval(..)`-Funktion wird von JavaSpaces selbst nicht unterstützt, da in Java bereits ein Mechanismus zur Kreierung von Prozessen vorhanden ist (Threads).

Ein Tupel in der JavaSpace-Umgebung entspricht einem Java-Objekt, welches zwei Marker-Interfaces implementiert:

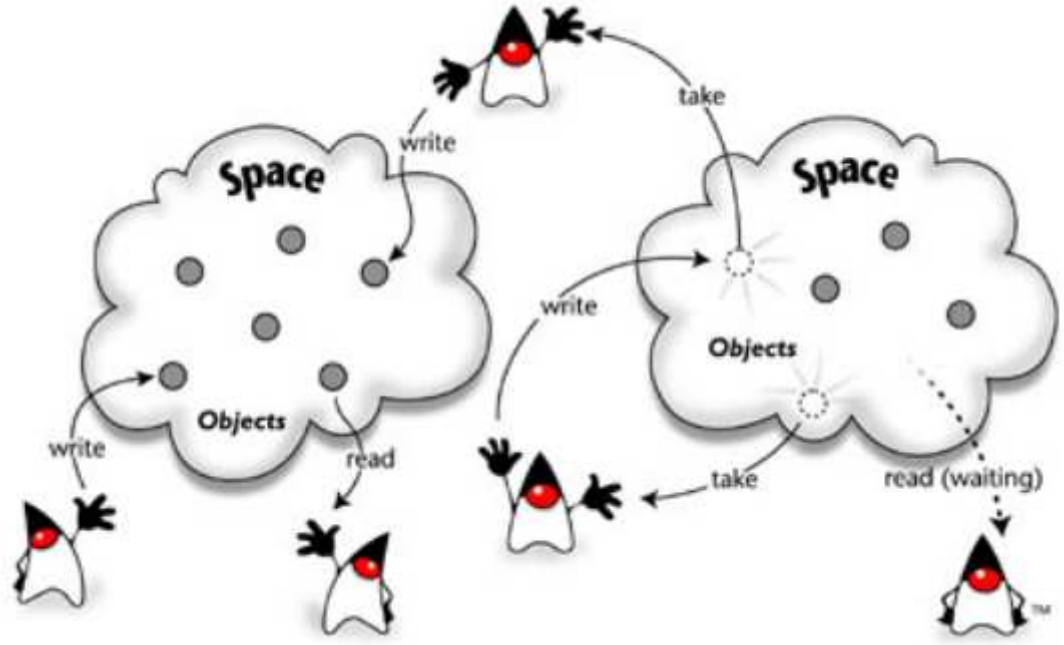
- `Serializable`, um die RMI-Kommunikation im Netzwerk zu unterstützen
- `Entry`, kennzeichnet das Java-Objekt als Tupel

Um Einträge eines JavaSpaces zu finden, die einem bestimmten Muster genügen, wird Pattern-Matching verwendet. Man benötigt dafür eine Objekt-Vorlage (Template), welche die Schnittstelle `Entry` implementiert. Ein Eintrag im JavaSpace genügt einem Template, falls die folgenden zwei Regeln erfüllt sind:

1. Der Typ des Templates stimmt mit dem des Eintrags überein oder ist ein Supertyp des Eintrags.

2. Jedes Feld des Templates genügt dem korrespondierenden Feld des Eintrags, wobei gilt:
   - Hat ein Feld des Templates den Wert `null`, dann genügt er immer dem korrespondierenden Feld des Eintrags.
   - Ist der Wert eines Feldes des Templates nicht-`null`, so genügt er dem korrespondierenden Feld des Eintrags, falls die beiden den gleichen Wert haben.

Die folgende Abbildung zeigt die JavaSpace-API:



JavaSpaces setzen die oben erwähnte Konzepte des Leasings und der Transaktionen ein. So kann man z.B. bei der `write(..)`-Methode eine Zeitspanne angeben, wie lange ein Tupel höchstens in einem Space gespeichert werden soll. Jeder Lese- und Schreiboperation kann man Transaktionen zuordnen, was beim Arbeiten mit mehreren JavaSpaces oft notwendig ist.

Beispiel: Kopieren von JavaSpace-Entries von einem JavaSpace in einen anderen mit Hilfe einer Transaktion
Durch einen Lookup bei einem LUS mit einem JavaSpace-Template erlangen wir zwei unterschiedliche Referenzen auf JavaSpaces.

```
JavaSpace s1 = (JavaSpace)lus.lookup(javaSpaceTemplate);
JavaSpace s2 = (JavaSpace)lus.lookup(javaSpaceTemplate);
```

Die Klasse `Name` ist eine Unterklasse eines `Entry`-Objekts und repräsentiert einen String, der in einen JavaSpace geschrieben werden kann. Zwei Instanzen davon schreiben wir in den ersten JavaSpace. Der `null`-Parameter gibt an, dass die Operation keiner Transaktion zugeordnet wird. Der dritte Parameter sorgt dafür, dass der Eintrag höchstens 60 Sekunden im JavaSpace gespeichert wird (Lease).

```
s1.write(new Name("Eintrag1"),null,60*1000);
s1.write(new Name("Eintrag2"),null,60*1000);
```

Wir erzeugen ein `Name`-Template, um die geschriebenen Einträge nachher wieder aus dem JavaSpace holen zu können. Man beachte, dass jedes `Name`-Objekt auf dieses Template passt (wegen der `null`-Initialisierung).

`Name nameTmpl = new Name(null);`
Jetzt erlangen wir durch einen Lookup bei einem LUS (analog wie bei den JavaSpaces) eine Referenz auf einen Transaktionsmanager.
`TransactionManager tm = (TransactionManager)lus.lookup(tmTemplate);`
Nun erzeugen wir eine Transaktion mit Hilfe der Klasse `TransactionFactory` (10 Minuten Lease).
`Transaction.Created c = TransactionFactory.create(tm,10*60*1000);`
Die Transaktion ist ein Attribut des Objektes `c` und wird von uns beim Kopiervorgang benutzt. Nach jeder Kopieraktion rufen wir `commit()` auf, um die Operation abzuschliessen. Die Konstante `NO_WAIT` bei der `take`-Methode veranlasst, dass die Methode nicht blockiert, falls kein `Name`-Objekt im JavaSpace vorhanden ist.

```
Entry e = js1.take(nameTmpl,c.transaction,JavaSpace.NO_WAIT);
js2.write(e,c.transaction,60*1000);
c.transaction.commit();
e = js1.take(nameTmpl,c.transaction,JavaSpace.NO_WAIT);
js2.write(e,c.transaction,60*1000);
c.transaction.commit();
```

Wenn alle Operationen erfolgreich waren, befinden sich die Einträge nach diesem Vorgang in dem zweiten JavaSpace.

Ausführlichere Informationen zu JavaSpaces und praktische Beispiele werden in [3] und [6] gegeben.

4 Ein vollständiges Beispiel

In diesem Abschnitt soll ein Beispiel für einen Jini-Service und einen Client, der diesen nutzt, gegeben werden. Der Jini-Dienst besteht aus einer Schnittstelle, die eine Methode zur Verfügung stellt. Diese Methode inkrementiert eine übergebene Integer-Zahl um Eins.

4.1 Beispiel-Service

4.1.1 Service-Interface

Wir definieren ein Interface, welches nachher für einen Client sichtbar wird.

```
public interface IncNumInterface {
  public int incNum(int num);
}
```

4.1.2 Service-Proxy

Diese Klasse enthält die konkrete Implementierung des Service-Interfaces. Sie implementiert die Schnittstelle `Serializable`, damit Instanzen dieser im Netzwerk von Clients heruntergeladen werden können. Die Jini-Proxies werden in der Regel auf einem Webserver abgelegt.

```
public class IncNumProxy implements IncNumInterface, Serializable {
  public IncNumProxy(){ }
  public int incNum(int num) {
```

```
    return num+1;
  }
}
```

4.1.3 Service-Wrapper

Diese Wrapper-Klasse meldet den Service bei allen verfügbaren LUS an. Sie implementiert das Interface `Runnable`, um als eigener Prozess ablaufen zu können.

```
public class IncNumService implements Runnable{
```

Wir definieren eine `HashMap`, um die LUS-Registrierungen zu speichern. Dadurch können wir Doppelregistrierungen vermeiden. Das `ServiceItem` ist die Objekt-Repräsentation unseres Services. Wir übergeben es bei jeder Registrierung mit einem LUS. Desweiteren benötigen wir einen `LookupDiscoveryManager`, damit wir LUS auffinden können.

```
  protected HashMap regs = new HashMap();
  protected ServiceItem item;
  protected LookupDiscoveryManager manager;
```

Wir definieren eine innere Klasse zur Reaktion auf `DiscoveryEvents`. Eine Instanz dieser übergeben wir später dem `LookupDiscoveryManager`. Dieses bewirkt, dass wir über das Erscheinen neuer oder die Beendung alter LUS benachrichtigt werden.

```
  class Listener implements DiscoveryListener{
```

Diese Methode wird aufgerufen, wenn ein LUS, bei welchem unserer Service registriert ist, explizit beendet wird. Wir nutzen das Eintreten eines solchen Events zur Entfernung alter LUS-Einträge aus der `HashMap`.

```
    public void discarded(DiscoveryEvent ev) {
      ServiceRegistrar[] deadregs = ev.getRegistrars();
      for (int i = 0; i < deadregs.length; i++) {
        regs.remove(deadregs[i]);
      }
    }
```

Diese Methode wird aufgerufen, wenn ein oder mehrere LUS gefunden wurden. Durch `ev.getRegistrars()` holen wir uns die gefundenen LUS. Mit einem Aufruf von `registerWithLookup(...)` veranlassen wir, dass unser Service bei jedem der gefundenen LUS registriert wird.

```
    public void discovered(DiscoveryEvent ev) {
      ServiceRegistrar[] newregs = ev.getRegistrars();
      for (int i = 0; i < newregs.length; i++) {
        if(!regs.containsKey(newregs[i])){
          registerWithLookup(newregs[i]);
        }
      }
    }
  } // Ende Listener
```

Im Konstruktor leiten wir alle nötigen Schritte für die Registrierung unseres Services ein.

```
public IncNumService() throws IOException{
  IncNumProxy proxy = new IncNumProxy();
```

Wir packen unseren `IncNumProxy` in ein `ServiceItem`, mit dem wir uns bei einem LUS als Service registrieren können. Ausserdem erzeugen wir einen Sicherheitsmanager, damit der Proxy von Clients heruntergeladen werden kann (die Konfiguration unseres Sicherheitsmanagers geschieht über ein sog. policy-File, das beim Start unseres Programms in der Kommandozeile übergeben wird).

```
  item = new ServiceItem(null,proxy,null);
  if(System.getSecurityManager()==null){
    System.setSecurityManager(new RMISecurityManager());
  }
```

Wir installieren einen LU-Discovery Manager, um nach LUS zu suchen. Der erste Parameter steht für die Liste von Jini-Gruppen, in welchen wir suchen wollen. Der leere String steht dabei für die sog. öffentliche Gruppe. Als zweiten Parameter kann man Orte angeben, wo sich LUS befinden. Orte sind URLs repräsentiert durch die Jini-Klasse `LookupLocator`. Gibt man den Wert `null` an, so wird im gesamten Netzwerk nach LUS gesucht. Dieses wählen wir, da wir davon ausgehen, dass mindestens ein LUS in unserem lokalen Netzwerk zu finden ist. Der dritte Parameter bewirkt, dass wir über `DiscoveryEvents` benachrichtigt werden.

```
  DiscoveryListener listener = new Listener();
  manager = new LookupDiscoveryManager(new String[] {""},null,listener);
}
```

Diese Methode registriert den Service mit dem als Parameter übergebenen LUS. Das Schlüsselwort `synchronized` bewirkt, dass diese Methode nur von einem Thread zur Zeit ausgeführt werden kann (gegenseitiger Ausschluss).

```
protected synchronized void registerWithLookup(ServiceRegistrar registrar){
  ServiceRegistration reg = null;
```

Wir registrieren unseren Service (repräsentiert durch das `ServiceItem`) mit einem 60-Minuten Lease.

```
  try {
    reg = registrar.register(item, 60*60*1000);
  } catch (RemoteException e) {}
```

Wir setzen die eindeutige `ServiceID`, die vom ersten gefundenden LUS stammt. Diese benutzen wir dann für die weiteren Registrierungen bei anderen LUS, damit die ID eindeutig bleibt. Ausserdem speichern wir die Registrierung lokal in der `HashMap`.

```
  if(item.serviceID == null){
    item.serviceID = reg.getServiceID();
  }
  regs.put(registrar,reg);
} Ende registerWithLookup(...)
```

Diese Methode erzeugt eine Proxy-Instanz.

```
  protected IncNumProxy createProxy(){
    return new IncNumProxy();
  }
```

Diese Methode implementiert das Interface `Runnable`. Wir lassen unseren Thread schlafen, um die Virtual-Machine nicht zu beenden, was uns wiederum das Empfangen von Events ermöglicht. Wir könnten die Methode beispielsweise auch dazu verwenden, um Leases bei LUS zu verlängern.

```
  public void run() {
    while(true){
      try {
        Thread.sleep(1000000);
      } catch (InterruptedException e) {
      }
    }
  }
```

Im Hauptprogramm instanziieren wir einen `IncNumService`. Wir kreiieren einen neuen Thread und starten diesen. Dadurch wird die Methode `run()` (einzige Methode des Interfaces `Runnable`) implizit aufgerufen. Dann ist unser Service bereit, um `DiscoveryEvents` zu empfangen, durch die er sich bei einem LUS registrieren kann. Ist er erst einmal registriert, so ist seine Schnittstelle für Clients verfügbar.

```
  public static void main(String[] args){
    try {
      IncNumService service = new IncNumService();
      new Thread(service).start();
    } catch (IOException e) {}
  }
} // Ende IncNumService
```

4.2 Beispiel-Client

Eine Klasse eines Clients, der den `IncNumService` nutzt. Er implementiert das Interface `Runnable`, um als eigener Prozess ablaufen zu können.

```
public class IncNumClient implements Runnable {
```

Das `ServiceTemplate` und den `LookupDiscoveryManager` benötigen wir für die Suche nach einem `IncNumService`-Interface.

```
  protected ServiceTemplate template;
  protected LookupDiscoveryManager manager;
```

Wir definieren eine innere Listener-Klasse, die wir dazu benutzen, einen LUS und ein `IncNumService` zu finden. Die implementierte Methode `discarded(...)` benötigen wir nicht, da wir - als Client - keine Registrierungen bei LUS vornehmen und somit nicht auf das Verschwinden von LUS reagieren brauchen.

```
class Listener implements DiscoveryListener{
  public void discarded(DiscoveryEvent ev) {}
```

Diese Methode wird aufgerufen, wenn ein oder mehrere LUS gefunden wurden. Wir durchsuchen die gefundenen LUS der Reihe nach, bis ein `IncNumInterface` gefunden wurde. Anschliessend verwenden wir die Methode `incNum()` des Service-Interfaces.

```
  public void discovered(DiscoveryEvent ev) {
    ServiceRegistrar[] newregs = ev.getRegistrars();
    if (newregs!=null) {
      for (int i = 0; i < newregs.length; i++) {
        IncNumInterface servobj =
          (IncNumInterface)lookForService(newregs[i]);
        if(servobj!=null){
          System.out.println("Incrementing 1 -- result: "
                             +servobj.incNum(1));
          System.out.println("Incrementing 100 -- result:
                             "+servobj.incNum(100));
        }
      }
    }
  }
} Ende Listener

public IncNumClient() throws IOException{
```

Wir initialisieren ein Template für die Suche nach einem `IncNumInterface`.

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

Wir installieren einen Sicherheitsmanager, damit unser Client in der Lage ist, Java-Code (in unserem Fall der `IncNumProxy`) herunterzuladen. Die weiteren Initialisierungen erledigen wir analog zum `IncNumService`.

```
  if(System.getSecurityManager()==null){
    System.setSecurityManager(new RMISecurityManager());
  }
  DiscoveryListener listener = new Listener();
  manager = new LookupDiscoveryManager(new String[] {""},null,listener);
} // Ende Konstruktor
```

Diese Methode macht einen Lookup mit Hilfe des `ServiceTemplates` bei dem als Parameter übergebenen LUS. Falls wir ein Service-Objekt finden, so geben wir dieses zurück, andernfalls `null`. zurückgeliefert.

```
protected Object lookForService(ServiceRegistrar lus){
  Object o = null;
  try {
    o = lus.lookup(template);
```

```
    } catch (RemoteException e) { }
    if(o==null)
      return null;
    else{
      return o;
    }
  }
```

Beschreibung analog zum `IncNumService`

```
  public void run() {
    while(true){
      try {
        Thread.sleep(1000000);
      } catch (InterruptedException e) {
      }
    }
  }
```

Im Hauptprogramm starten wir den Client, welcher dann auf `DiscoveryEvents` wartet.

```
  public static void main(String[] args) {
    try {
      IncNumClient client = new IncNumClient();
      new Thread(client).start();
    } catch (IOException e) {}
  }
} Ende IncNumClient
```

Ein sehr ausführliches und praxisnahes Beispiel zur Jini-Programmierung wird in [1], Kapitel 13 gegeben. Es handelt sich dabei um die Implementierung eines Druckerdienstes. Weiterführende Konzepte ,wie z.B. Wiederherstellung des Services nach einem Absturz und Leasemanagement, werden dort verwendet.

5 Zusammenfassung

Diese Seminararbeit gibt einen Einblick in die Jini-Technologie, mit deren Hilfe man Service-basierte Netzwerke erschaffen kann. Lookup-Services übernehmen die Verwaltung innerhalb eines solchen Netzwerkes. Durch Koppelung mehrerer Lookup-Services können beliebig grosse Netze aufgebaut werden.

Die Kommunikation in einem Jini-System ist sehr stark protokollorientiert. Clients kommunizieren über das Discovery-/Lookup-Protokoll mit Lookup-Services, um gewisse Dienste aufzufinden oder anzumelden.
Jini bietet desweiteren programmiertechnische Tools wie Leasing (zeitbeschränkte Resourcenvergabe), Remote-Ereignisse und Transaktionen (atomare Ausführung von mehreren Operationen). Die verteilte Speicherung von Daten wird durch JavaSpaces realisiert, welche selbst Jini-Dienste sind. Dadurch wird auch die Entwicklung von komplexen, verteilten Anwendungen möglich.

Zusammenfassend lässt sich sagen, dass Sun mit Jini einen vielversprechenden Ansatz für die Handhabung von verteilten Systemen geschaffen hat. Die standardisierte Architektur ermöglicht es, Dienstanbieter und Clients unterschiedlichster Art über gemeinsame Netzwerke zu koppeln. Dadurch, dass Jini vollständig in Java realisiert ist, können diese Dienstanbieter und Clients durchaus auf unterschiedlichen Plattformen betrieben werden.
Aus programmiertechnischer Sicht bietet Jini eine recht einfache API, da auf einem sehr hohen Abstraktionsniveau operiert wird. Typische Probleme, die bei der Programmierung verteilter Systeme auftreten (siehe Einleitung), werden meiner Ansicht nach sehr elegant gelöst (z.B. durch Transaktionen).
Nachteile von Jini liegen vor allem darin, dass es sich um eine relativ junge und daher auch noch nicht vollständig ausgereiften Technologie handelt. Kritisiert werden dabei häufig die Sicherheitsmechanismen, welche im wesentlichen auf die von Java und RMI aufsetzen, und teilweise unzureichend für Jini-Zwecke sind.
Andere Nachteile liegen in dem relativ hohen Resourcenverbrauch von Java-Programmen (Betreiben einer JVM), was den Einsatz der Technologie auf kleineren Geräten erschwert.

Verwendete Literatur

1. W. Keith Edwards, *CORE JINI*, Prentice Hall, 1999
2. H. Bader, W. Huber, *Jini^{TM}- Die intelligente Netzwerkarchitektur in Theorie und Praxis*, Addison-Wesley, 2000
3. Freeman, Hupfer, Arnold, *JavaSpaces^{TM}Principles, Patterns, and Practice*, Addison-Wesley, 2000
4. Arnold, Gosling, Holmes, *The Java Programming Language, Third Edition*, Addison-Wesley, 2000
5. N. Carriero, D. Gelernter, *Linda in Context*, Communications of the ACM, Vol.32, No.4,pp. 444-458, 1989
6. Jini Spezifikationen v.1.2, `http://wwws.sun.com/software/jini/specs/index.html`
7. RMI Spezifikationen, `http://java.sun.com/j2se/1.4.1/docs/guide/rmi/index.html`
8. Jan Newmarch's Guide to Jini Technolgies, `http://pandonia.canberra.edu.au/java/jini/tutorial/Jini.xml`
9. Jini-Community, `http://www.jini.org`