

Praktische Einführung in Jini*

Andreas Zeidler†

Marco Gruteser‡

May 26, 1999

Bezaubernde Jini

Sun hat Jini (das Akronym steht für "Java Intelligent Network Infrastructure") mit großem Trara am 25. Januar offiziell vorgestellt. Was macht Jini denn nun so bezaubernd? Die Gesamtarchitektur wurde von der iX bereits in der Ausgabe 11/98 vorgestellt und hier gehen wir nur auf die für diesen Artikel wichtigsten Punkte ein.

Stellen wir uns vor: Sie kaufen einen Drucker, schließen ihn an und...warten. Nach einigen Sekunden schauen Sie auf Ihren Desktop und finden dort ein neues Drucker-Icon vor. Ein "Klick", ein Fenster öffnet sich und zeigt den Status des Druckers an: "Bereit". Sie starten Ihre Textverarbeitung und wollen einen Testausdruck machen. Im Druckerauswahl-Menü findet sich der neue Drucker! Ein Testausdruck gelingt auf Anhieb: Ohne Treiberinstallation...aber dank Jini.

Macht von außen nicht viel her, der Wert liegt in der Lampe

Von dieser Vision ist Jini ein gutes Stück entfernt, wirklich bezaubernd ist allerdings, daß der Jini Technologie anmerkt, daß sich die verantwortlichen Entwickler grundlegende und gute Gedanken über das System gemacht haben. Der Aufbau ist klar und einfach strukturiert. Vergleicht man die Größe der gepackten Datei mit der von z.B. JDK1.2 (ca. 20MB), kann man fast enttäuscht sein: Lediglich magere 2.5MB gilt es, über das Netz zu quetschen. Der wahre Wert eröffnet sich nach dem Auspacken.

Jini wurde konsequent auf die Verwendung von Java und Netzen ausgelegt. Die Annahme dahinter ist, daß das Internet und Java zukünftig überall zu finden sein werden, in Kühlschränken genauso wie in Festplatten. Jini macht sich den objektorientierten Ansatz von Java zunutze und verbindet ihn mit den Eigenschaften, die RMI (Remote Method Invocation) für die verteilte Programmierung bietet. Zusammen ergibt es das, was Sun als "typed network" bezeichnet: In Jini werden sowohl Daten als auch Code über das Netz bewegt, alles unter der strengen Typisierung von Java. Konsequenterweise wird bei Jini alles (also Hard- und Software) als "Objekt" angesehen und letztendlich gleich behandelt. Jini dient dabei als Abstraktionsstufe und vereint die Sichten auf unterschiedliche Komponenten.

Der goldene Schuß

Jini soll genau ins Schwarze treffen: Eine Infrastruktur für unterschiedlichste Hard- und Software bieten und das ohne langwierige Konfigurationsorgien. Sun nennt es "spontaneous networking". Hardware wird ans Netz angeschlossen und kann ohne Aufwand sofort benutzt werden. Das gleiche gilt auch für Software: Man kann sich leicht die jinifähige Textverarbeitung vorstellen, die zur Rechtschreibprüfung ein jinifähiges Wörterbuch konsultiert. Die Jini-Welt kennt nur noch Dienstanbieter und Dienstanwender. Damit der Drucker ohne Treiberinstallation funktionieren kann, verwendet Jini die Möglichkeit, Code über das Netz in Dienstanwender zu "injizieren". Der Drucker lädt seinen "Proxy" (z.B. einen Java-RMI-Stub) zum Klienten und bietet ihm so eine wohldefinierte Schnittstelle an. Der Klient ruft Methoden des Proxies auf, und dieser kümmert sich um die korrekte Behandlung der Daten. Wenn der Dienst nicht mehr benötigt wird, wird der Proxy verworfen.

*Redaktionell überarbeitet veröffentlicht in der iX 4/99 im Heise Verlag

†az@informatik.tu-darmstadt.de

‡gruteser@rbg.informatik.tu-darmstadt.de

Infrastruktur und Programmiermodell

Damit Jini zaubern kann, wurde das Java-Programmiermodell ergänzt und zu Java und RMI kamen einige Infrastrukturkomponenten hinzu. Dienste (in Jini ist eigentlich alles ein Dienst) werden in die Lage versetzt, einander und vor allem auch den sog. Lookup-Service (LUS) ohne Kenntnis des Netzes zu finden ("bootstrapping"). Als Voraussetzung brauchen sie lediglich eine Adresse, damit sie im Netz identifizierbar sind, eine Java-Virtual-Machine (JVM) und etwas Speicher. Den Rest besorgt die Jini-Infrastruktur: Dienste melden sich über ein "Discovery&Join-Protokoll" bei einem (ihnen meist vorher nicht bekannten) LUS an, dieser sorgt nötigenfalls für die Konfiguration, und schon können die Dienste verwendet werden. Dienstnutzer finden Diensteanbieter ebenfalls über den LUS: Sie durchsuchen diesen nach einem Interface, das z.B. eine Rechtschreibprüfung implementiert ("Lookup"). Werden sie fündig, bekommen sie vom LUS ein "Proxy-Objekt", das als Stellvertreter des Dienstes die gesamte Kommunikation mit dem eigentlichen Dienst übernimmt. Wie diese wiederum realisiert ist, ist Sache des Dienstes. Für den Dienstnutzer scheint alles lokal vorhanden zu sein. Der Lookup-Service ist dabei die zentrale Anlaufstelle, um Diensteanbieter und Dienstnutzer zusammenzuführen.

Das Java-Programmiermodell wird von Jini um einige einfache APIs erweitert; so gesellen sich zu bekannten Komponenten wie Beans oder Swing noch Leases, Transaktionen und Distributed Events. Da Jini eine allgemeine Architektur ist, sind die Strukturelemente keine festen Klassen, sondern lediglich Interfaces: Dienste implementieren sie nach ihren Bedürfnissen, werden dadurch aber gezwungen, "jini enabled" zu sein, d.h. die Art der Interaktion zwischen den Diensten wird festgelegt, nicht jedoch deren Implementierung an sich. Das Kommunikationsprotokoll zwischen einem Druckerdienst und dem Drucker kann proprietär sein, für den Benutzer ist das gleichgültig.

Diese wenigen einfachen und gut durchdachten Elemente sind es, die Jini so bezaubernd machen. Weniger ist manchmal eben mehr.

Installation

Den Geist aus der Flasche zu bekommen...

Mit ein wenig Glück ist der Aufwand, Jini lauffähig zu bekommen, relativ gering. Die größte Hürde dabei wird wahrscheinlich der Download sein. Sun hat keine europäischen Mirrors dafür eingerichtet, d.h. man muß sich das Programmpaket direkt von (<http://developer.java.sun.com/developer/products/jini/index.html>) herunterladen. Dazu ist es notwendig, sich als Entwickler bei Sun zu registrieren. Jini wird unter der neuen "Sun Community Source License" (SCSL) distribuiert, man muß einen kostenlosen Lizenzvertrag akzeptieren, hat allerdings dafür die Quellen zu Jini.

Sun bietet zwei Pakete zu Jini an:

1. Das "JiniSystem Software 1.0 Starter Kit", bestehend aus der "Jini Core Platform": Das "Core-API", die Spezifikationen, die Dokumentationen und die Quellen zu Jini
2. Die JavaSpaces (separat herunterzuladen)

Nachdem man die üblichen Informationen (Name, Firma, Email) angegeben hat, kann man sich am Download versuchen. Zumindest aus dem DFN heraus ist das jedoch tagsüber ein langwieriges Unterfangen (zwei bis drei Stunden sollte man mind. einrechnen). Oftmals hilft dann nur noch, einen Download zur nächtlichen Stunde zu starten.

Die Jini-Version 1.0 benötigt als Java-Umgebung das JDK1.2. Dieses ist leider noch nicht für alle Plattformen erhältlich und schränkt die Benutzer auf Windows95/98/NT und Solaris ab Version 2.5.1 ein. Unter Solaris wird die Version 7 empfohlen.

Jini kommt als ZIP-Files daher und installiert sich in ein Unterverzeichnis namens jini1_0. Wenn man zusätzlich noch die JavaSpaces installiert, darf man diese erst nach Jini installieren, da JavaSpaces Teile der HTML Dokumentation ersetzt und erweitert.

... ist oft nicht schwer.

Damit man einen Eindruck von Jini bekommt, liefert Sun für einige Interfaces eine "Referenzimplementierung" mit. Für diesen Artikel interessant ist davon der Lookup-Service. Da Jini auf Java-RMI aufbaut, muß man als erstes das "RMI Activation System" (rmid) starten. Dort registriert sich der LUS als aktivierbares Objekt. Im Normalfall reicht zum Start ein einfaches: (Anm: Die nachfolgenden Kommandozeilen-Beispiele beziehen sich auf die Solaris-Umgebung)

```
rmid -log <Verzeichnis>\&
```

Dabei ist darauf zu achten, daß das JDK1.2-Verzeichnis zuerst im Pfad steht. rmid legt dabei in <Verzeichnis> ein Log-Verzeichnis an oder benutzt das vorhandene, wenn eine frühere Instanz schon ein solches angelegt hat, was manchmal zu unschönen Nebeneffekten führen kann.

Jini beruht u.a. auf der Fähigkeit von RMI, Klassen über das Netz von einem HTTP-Server zu laden. Dieser wird in der Codebase-Property angegeben. Sun liefert bei Jini einen einfachen Server mit, der mittels HTTP GET auf Port 8080 .jar- oder .class-Files zur Verfügung stellt. Er findet sich in der Tools-Bibliothek von Jini (tools.jar) und wird durch

```
prompt> java -jar $JHOME/jini1_0/lib/tools.jar
        -port 8080 -dir $JHOME/jini1_0/lib/ -verbose
```

gestartet. Das im Request angegebene File (relative Pfadnamen sind möglich) muß sich im Verzeichnis befinden, das mit der -dir-Option angegeben wurde. Wenn er mit der -verbose-Option gestartet wurde, gibt der Server bei jedem Request aus, welcher Rechner welche Datei angefordert hat. Das kann sehr praktisch sein, wenn man den Weg von Code durch das Netz verfolgen will.

Als letztes wird der Lookup-Service gestartet. Der CLASSPATH sollte dabei so aussehen:

```
prompt> echo $CLASSPATH
.:$JHOME/jini1_0/lib/jini-core.jar:$JHOME/jini1_0/lib/sun-util.jar:
 $JHOME/jini1_0/lib/reggie.jar:$JHOME/jini1_0/lib/reggie_dl.jar:
 $JHOME/jini1_0/lib/jini-ext.jar
```

```
prompt> java -jar -Djava.security.policy=
        $JHOME/jini1_0/example/lookup/policy.all
        $JHOME/jini1_0/lib/reggie.jar http://<hostname>:8080/reggie_dl.jar
        $JHOME/jini1_0/example/lookup/policy.all /tmp/reggie_log public
```

”Reggie” ist eine Beispiel-Implementierung von Sun und findet sich in ”reggie.jar”. In <Pfad>/policy.all findet sich eine allgemeine Sicherheits-Policy, die für das Java-Sicherheitsmodell notwendig ist (s. ”Stolperfallen”). Der Lookup-Service ist selbst als Jini-Dienst implementiert, d.h. er registriert sich bei sich selbst in der ”public”-Gruppe. Der Zustand wird in /tmp/reggie_log festgehalten.

Man sollte darauf achten, daß zwischen den Aufrufen ausreichend viel Zeit liegt, damit sich die Dienste initialisieren können (>10 Sekunden)

Im Paket ist auch ein graphischer LUS-Browser enthalten:

```
prompt> java -cp $JHOME/jini1_0/lib/jini-examples.jar
        -Djava.security.policy=$JHOME/jini1_0/example/browser/policy
        -Djava.rmi.server.codebase=
        http://<hostname>:8080/jini-examples-dl.jar
        com.sun.jini.example.browser.Browser -admin &
```

Die anderen Referenzimplementierungen von Sun (Transaction Manager und JavaSpaces) kann man ähnlich starten, werden aber zu diesem Zeitpunkt noch nicht benötigt.

Stolperfallen

Bevor man haareraufend auf Jini schimpft, hier einige mögliche Stolperfallen:

- Bei manchen Konfigurationen von Solaris scheinen Programme mit Swing-Komponenten nur lokal darstellbar zu sein. Die Darstellung auf einem Remote-Rechner führt entweder zu einem ”core dump”, oder es wird ein leerer Frame dargestellt.

- Bei Rechnern mit Windows-Betriebssystem ohne Netzanschluß startet der rmid oft nicht. Ursache ist wohl ein fehlschlagender "Name Lookup". Ein verlässlicher Workaround ist nicht bekannt. Es kann helfen, einen DFÜ-Adapter und TCP/IP zu installieren, dem DFÜ-Adapter eine feste IP-Adresse zuzuweisen und das Paar Rechnername/IP-Adresse in "hosts", bzw. "lmhosts" einzutragen, DNS muß auf jeden Fall deaktiviert sein.
- Jini verwendet eine eigene Multicast-Gruppe. Die übliche Switch-/Router-Konfiguration blockt aber Multicasts. Wird der LUS in einem anderen Netzsegment gestartet, wird man ihn in der Regel nicht sehen und verwenden können.
- Ein weiteres Problem ist oftmals mit der Verwendung von Policy-Files verbunden. Das Java-Sicherheitsmodell fordert die Zusicherung expliziter Zugriffsrechte. Beispiele dafür sind Verzeichnisse oder Ports. In der Testphase von Jini sollte man immer die mitgelieferte "policy.all" verwenden: Sie sichert maximale Zugriffsrechte zu. Für ernsthafte Anwendungen ist sie nicht geeignet. "policy.all" ist folgendermaßen aufgebaut:

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

- Die "Server-Codebase-Falle": Immer wenn ein "ClassDefNotFoundError" oder eine "ClassNotFoundException" auftritt, sollte man sich ernsthaft mit der Codebase-Property auseinandersetzen. Ist diese falsch gesetzt, findet der Client die Server-Proxy-Klasse nicht (diese wird in der Regel von der dort angegebenen Position geladen). Die Client-JVM bekommt beim Proxy-Download vom Lookup-Server mitgeteilt, von wo sie die Klasse selbst herunterladen kann. Steht hier bspw. nur ein Rechnername ohne Domäne, wird ein Klient in einer anderen Domäne die Server-Klassen nicht finden. Der Service muß also immer die (aus Sicht des Servicenutzers) korrekte Codebase angeben.

Die Arbeit mit Jini

Jini, die "Java Intelligent Network Infrastructure", dessen Kurzform im Klang an "Genie" erinnern soll, tritt an, um die zweite Runde der Java-Revolution einzuläuten. Jini erweitert das klassische Java-Paradigma um Konzepte zur Entwicklung verlässlicher verteilter Systeme. Vom "write-once, run anywhere" wird in Jini besonders das "run anywhere" betont. Ein Überblick über Jini findet sich in der iX 11/98. Dieser Artikel erklärt die Entwicklung von Anwendungen, die Jini als "intelligente" Infrastruktur nutzen. Ein Jini-System besteht aus einer Menge von Diensten, die zu Gruppen – sog. djinns – organisiert werden. Ein Dienst stellt dabei beliebige Funktionen bereit, die von anderen Diensten, Anwendungen oder Menschen verwendet werden können. Er kann in Soft- und/oder Hardware realisiert werden und an beliebigen Stellen im Netz laufen. Beispiele dafür sind ein Dienst, der eMails abschickt oder ein Druckdienst, der etwas auf einem Drucker ausgibt. Eine zentrale Aufgabe der Jini-Infrastruktur ist es Dienstanbieter und -nutzer zusammenzubringen. Wenn beispielsweise ein Drucker an ein Netz angeschlossen wird, soll das Netz den Druckdienst erkennen und dieser ohne weitere Installationsarbeiten (z.B. Treiberinstallation) für andere Komponenten im Netz verfügbar sein ("The Network is the Computer"). Jini realisiert dies mit den Discovery-/Join-Protokollen und typischerweise einem Lookup-Service (LUS).

Eine Besonderheit von Jini ist, daß jeder Dienst einen sogenannten Proxy besitzt. Dieser Proxy besteht aus einem oder mehreren Java-Objekten, die serialisiert inklusive Programmcode an den Dienstnehmer übertragen werden. Der Typ und die Methoden [M1] der Objekte stellen dann die Schnittstelle dar, über die der Dienstnehmer den Dienst bei sich anspricht. Durch diese Form von mobilem Code, kann die Funktionalität des Dienstes teilweise oder ganz beim Dienstnehmer realisiert werden. Als weiterer nützlicher Nebeneffekt ist damit das Protokoll, über das der Proxy mit seinem Dienst kommuniziert nicht festgelegt. Dies ist besonders beim Einbinden von Geräten interessant, da der Proxy sich dann bereits bestehender Protokolle bedienen kann, um mit dem Gerät zu kommunizieren. Der Proxy kann als ein dynamischer Treiber angesehen werden, der bei Bedarf in das System geladen und nach der Verwendung wieder entfernt wird.

Ein allgemeines Szenario

Zentrale Anlaufstelle in einem Jini-System ist der LUS. An ihn wendet sich ein Dienstanbieter, um seinen Dienst im Netz zu offerieren, und ein Interessent, um an den gewünschten Dienst zu gelangen. Das Zusam-

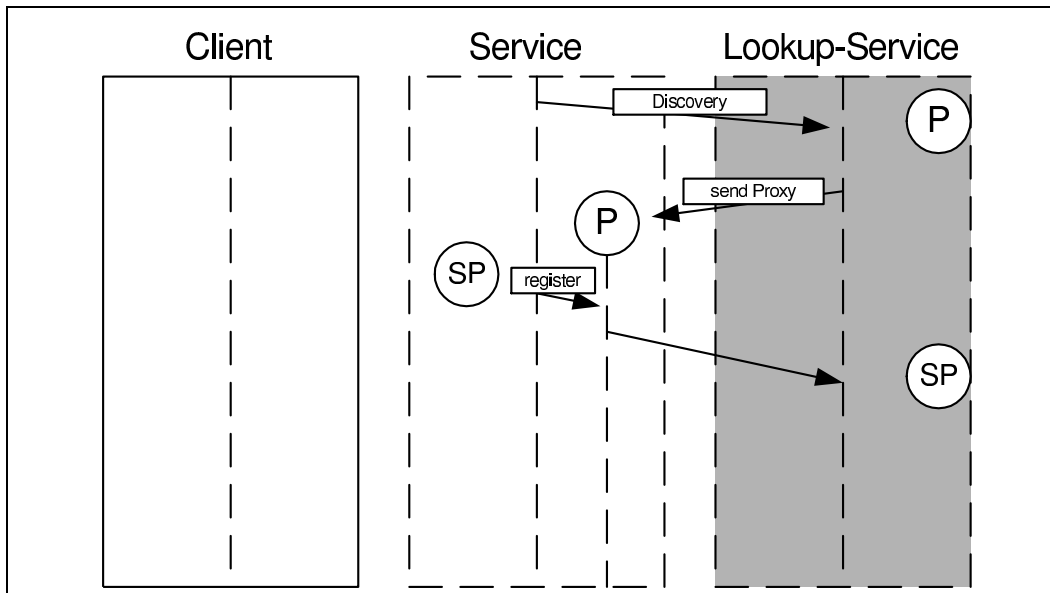


Figure 1: Dienst meldet sich an

Beispiel dieser im Netz verteilten Komponenten wird in Abbildung 1 und 2 illustriert. Die "Bootstrap"-Phase (Abbildung 1) des Diensteanbieters besteht aus folgenden Schritten:

Discovery Der Diensteanbieter sucht mittels Discovery-Protokoll nach erreichbaren LUS. Da Lookup-Services vollständige Jini-Dienste sind, bieten sie ihre Dienste ebenfalls über Proxies an. Diese werden von den gefundenen LUS an den Diensteanbieter übermittelt.

Join Der Diensteanbieter trägt seinen Dienst in die LUS ein. Dabei wird der Dienst-Proxy dort als serialisiertes Objekt hinterlegt.

Leasing Der Eintrag ist nur für eine bestimmte Zeit vereinbart. Nach Ablauf dieser Zeit wird er vom LUS automatisch entfernt. Soll der Eintrag länger bestehen bleiben, so muß der Diensteanbieter den "Lease" für den Eintrag rechtzeitig verlängern. Durch das Leasing-Konzept wird verhindert, daß der LUS nach einiger Zeit viele ungültige Einträge von Diensten, die nicht mehr verfügbar sind, enthält.

Damit ist der Dienst im Netz verfügbar. Zum Auffinden und Benutzen eines Dienstes sind folgende Schritte nötig (Abbildung 2):

Discovery Der Interessent sucht zuerst, genau wie der Diensteanbieter, via Discovery-Protokoll einen LUS und erhält dessen Proxy.

Lookup Der Interessent gibt eine Beschreibung des gesuchten Dienstes an den Proxy des LUS. Der LUS durchsucht seine Dienst-Einträge nach passenden Kandidaten und übermittelt die Proxies der gefundenen Dienste an den Dienstnehmer.

Use Der Dienstnehmer kann nun in den Proxies wie in lokalen Java-Objekten Methoden aufrufen und so den Dienst benutzen. Jegliche Kommunikation zwischen Proxy und Dienst bleibt dabei für ihn transparent.
Umsetzung

Beginnend auf der Dienst-Seite sind für die folgende Umsetzung drei Schritte notwendig:

1. Definition eines Java-Interfaces als Schnittstelle zwischen Dienst und Dienstnehmer item Implementierung der Schnittstelle und Erzeugen eines Proxy-Objekts
2. Anmelden des Dienstes

Als Beispiel dient ein Additions-Dienst, der die Berechnung auf einem anderen Rechner ausführen läßt. Da Jini auf Remote Methode Invocation (RMI) aufbaut, ist es naheliegend (aber nicht zwingend) die Kommunikation zwischen Proxy und dem eigentlichen Dienst über dieses Protokoll zu realisieren. Jini ermöglicht es,

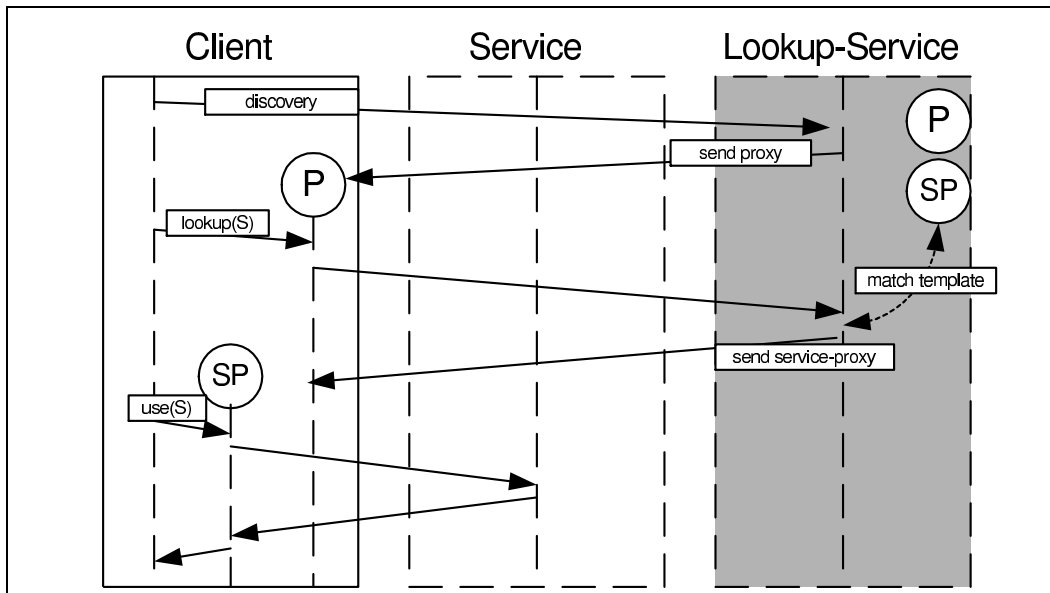


Figure 2: Klient sucht und benutzt Dienst

den von RMI erzeugten Server-Stub direkt als Proxy zu benutzen. Das Interface (Listing 1) zwischen Proxy und Dienstnehmer ist ein normales RMI-Interface, das - wie üblich - von Remote abgeleitet ist und dessen Methoden RemoteExceptions auslösen können.

```

import java.rmi.*;

public interface AddInterface extends Remote {
    public int add(int a, int b) throws RemoteException;
}

```

Listing 1

Im zweiten Schritt, muß das Interface implementiert werden (Listing). Dazu entwickeln wir einen RMI-Server, der die Methode add zur Verfügung stellt. Nach dem Kompilieren müssen mit rmic die Stub- und Skeleton-Klassen erzeugt werden.

```

import java.rmi.*;
import java.rmi.server.*;

public class AddInterface_impl extends UnicastRemoteObject
    implements AddInterface {

    public AddInterface_impl() throws RemoteException {
        super();
    }

    public int add (int a, int b) throws RemoteException {
        System.out.println ("server: " + a + "+"
            + b + "= " + (a+b) );
        return (a+b);
    }
}

```

Listing 2

Damit der Dienst verwendet werden kann, muß er im LUS registriert werden. Im Beispiel wird das von einem eigenständigen Java-Programm erledigt, das in Listing 3 abgedruckt ist. Als erstes muß ein `RMISecurityManager` installiert werden, da der Code des LUS-Proxy über das Netz geladen und dann ausgeführt werden soll. Dies ist mit dem Standard-Security-Manager nicht zulässig. Neben den eigentlichen Jini-Klassen (`net.jini-packages`) hat Sun noch eine Reihe weitere Wrapper-Klassen mitgeliefert, die Standardaufgaben wie Eintragen im LUS und Verlängern von Leases stark vereinfachen. Diese finden sich in den `com.sun.jini-packages`. Im folgenden wird von der `JoinManager`- und `LeaseRenewalManager`-Klasse Gebrauch gemacht. Der `JoinManager` verbirgt die Arbeit mit den Discovery/Join Protokollen und stellt eine einfache Möglichkeit zum Eintragen von Diensten in die LUS dar. Der `JoinManager` erwartet in seinem Konstruktor eine Instanz des Dienst-Proxy, die Attribute (zusätzliche Informationen) des Dienstes, einen `ServiceIDListener`, der die erzeugte Service ID entgegen nimmt, und ein `LeaseRenewalManager`-Objekt. Letzterer übernimmt dann selbständig die Verlängerung des Lease für den Eintrag im LUS. Im Beispiel wird eine Instanz der `AddInterface_impl` Klasse erzeugt und an den `JoinManager` übergeben. Dieser ermittelt automatisch die zugehörige Stub-Klasse und hinterlegt diese als Proxy im LUS. Jeder Jini-Dienst wird in den LUS durch einen Universal Unique Identifier (UUID) eindeutig identifiziert. Die Discovery & Join-Spezifikation fordert von Diensten, daß sie sich in allen LUS mit der gleichen Service ID registrieren und diese persistent speichern, um sie auch bei einem möglichen Neustart des Dienstes beizubehalten. Im Beispiel läßt der `JoinManager` die Service ID von einem LUS generieren und übermittelt sie dann an die `ServiceIDNotify`-Methode, wo sie ignoriert wird. Das Beispiel ist also nicht ganz Jini-konform.

```
import java.rmi.RMISecurityManager;

[...]
import net.jini.core.lookup.*;
import com.sun.jini.lookup.*;
import com.sun.jini.lease.LeaseRenewalManager;

public class JiniAddService implements ServiceIDListener {

    public void serviceIDNotify (ServiceID idIn) {
    }

    public static void main(String[] args) throws Exception {
        System.setSecurityManager(new RMISecurityManager());

        AddInterface_impl adder = new AddInterface_impl();

        JoinManager joinManager = new JoinManager(
            adder, // anzumeldendes Service Objekt
                null, // Attribute des Service
            (ServiceIDListener) new JiniAddService(),
            new LeaseRenewalManager()); //Leases erneuern

        System.out.println("JiniAddService bound in LUS");

        Thread.currentThread().sleep(100000);
    }
}
```

Listing 3

Das Programm wird mit dem Aufruf

```
java -Djava.rmi.server.codebase=http://<host>:<port>/<path>/
```

```
-Djava.security.policy=<path>/<policy-file> JiniAddService
```

gestartet. Wichtig ist dabei der abschließende Slash in der Codebase, da diese als Präfix vor die Package und Klassennamen gesetzt wird, um eine URL zu erhalten. Diese URL gibt an, von wo die class-Datei der jeweiligen Klasse aus der Sicht des Client geladen werden kann. Der Hostname sollte deshalb immer vollqualifiziert sein. Wenn eine Klasse foo.bar.puh nachgeladen werden muß, muß sie auf dem HTTP-Server unter `http://<host>:<port>/<path>/foo/bar/puh.class` zu finden sein, da der RMI-Classloader der Client-JVM über diese URL ihm unbekannte Klassen nachzuladen versucht.

Um den Dienst zu benutzen ist - wie in Listing 4 erkennbar - ein wenig mehr Aufwand zu treiben. Die wesentlichen Klassen für diese Aufgabe sind LookupLocator, ServiceRegistrar und ServiceTemplate. Der LookupLocator verbirgt die Discovery-Phase. Verwendet wird dabei das Unicast Discovery Protokoll, bei dem die Adresse eines LUS direkt angegeben werden muß. Das von ihm erhaltene ServiceRegistrar-Objekt stellt den LUS-Proxy dar. Die Beschreibung des gesuchten Dienstes erfolgt mittels der ServiceTemplate Klasse. Sie läßt als Suchkriterien eine Service ID, Java-Typen und Attribute zu. Service IDs und Attribute müssen exakt übereinstimmen, bei Java-Typen werden sowohl Klassen gleichen Typs als auch davon abgeleitete gefunden. Der Wert "null" wird als Wildcard interpretiert. Im Allgemeinen ist es sinnvoll, immer mindestens einen Typ anzugeben, damit es beim Benutzen der gefundenen Objekte nicht zu ClassCastExceptions kommen kann. Attribute können die Suche verfeinern, um die Treffermenge zu reduzieren, und Service IDs sind nützlich um einen ganz bestimmten Dienst zu finden. Im Beispiel wird nur ein Java-Typ, nämlich das AddInterface, angegeben. Die lookup-Methode des ServiceRegistrar-Objekts führt die eigentliche Suche aus. Das gefundene Objekt kann dann nach einem Typecast wie ein lokales angesprochen werden.

```
import net.jini.discovery.LookupLocator;
import net.jini.lookup.*;
import java.rmi.RMISecurityManager;

class JiniAddClient {

    public static void main(String args[]) throws Exception {
        System.setSecurityManager(new RMISecurityManager());

        LookupLocator locator=new LookupLocator("jini://localhost");
        ServiceRegistrar sr=locator.getRegistrar();

        Class[] classes=new Class[] { AddInterface.class };
        ServiceTemplate st=new ServiceTemplate( null, // ID
                                                classes, // types
                                                null); // attributes

        AddInterface service=(AddInterface)sr.lookup(st);

        int result=service.add(3,5);
    }
}
```

Listing 4

Das Programm wird gestartet mit:

```
java -Djava.security.policy=/<path>/policy.all JiniAddClient
```

Gruppen

Jini sieht vor, daß Dienste in Gruppen organisiert werden können. Jeder Dienst kann zu beliebig vielen Gruppen gehören. Beispielsweise könnte ein Drucker sich in den Gruppen "Hardware", "Drucker" und "Konferen-

zraum 2" anmelden oder es werden verschiedene Dienste zusammengefaßt weil sie gemeinsam eine Aufgabe lösen. Gruppen werden eingerichtet, indem man LUS die entsprechenden Gruppennamen zuweist. Bei Suns Referenzimplementierung müssen sie als Kommandozeilenparameter beim Start des LUS angegeben werden. Der Name "public" ist als Standard vergeben. Ein Dienstanbieter ist selbst für eine passende Registrierung in den LUS verantwortlich. Er muß seinen Dienst also in jedem ihm bekannten LUS genau dann registrieren, wenn der LUS mindestens eine Gruppe verwaltet, zu der der Dienst gehören soll. Daneben hat Sun auch ein Admin-Interface vorgesehen. Implementiert ein Dienst dieses Interface, so kann dessen Gruppenzugehörigkeit während der Laufzeit z.B. über den mitgelieferten Browser geändert werden. Bei den LUS kann man zusätzlich noch die Gruppen ändern, die sie verwalten sollen. In unserem Beispiel-Dienst (Listing 3) wurde das Gruppenkonzept durch den JoinManager verborgen. Bei dem angegebenen Aufruf registriert der JoinManager den Dienst in allen gefundenen LUS und damit auch in allen Gruppen. Indem man einen anderen Konstruktor des JoinManagers verwendet, kann man die Registrierung des Dienstes auf bestimmte Gruppen einschränken:

```
new JoinManager(proxy, null, new String[] {"Gruppe1", "Gruppe2"},
               null, new SL(), new LeaseRenewalManager());
```

Die Discovery Protokolle

Wie bereits angedeutet, unterstützt der Dienstnehmer in Listing 4 nur das Unicast Discovery Protocol. Daneben bietet Jini noch das Multicast Request Protocol und das Multicast Announcement Protocol, um LUS in einem Netz zu finden. Jini bietet eine Infrastruktur auf Dienstebene an, d.h. sie kann nicht dazu dienen IP-Adressen und Subnetz-Masken von Rechnern zu konfigurieren, sondern setzt die Konfiguration auf niedriger Ebene voraus.

Unicast Discovery Protocol

Voraussetzung ist, daß die Adresse des LUS bereits bekannt ist. Der LUS wird direkt über eine TCP-Verbindung zum Port 4160 (ein "Scherz" der Jini-Entwickler: Ergebnis der hexadezimalen Subtraktion CAFE - BABE) kontaktiert. Als Antwort überträgt der LUS sein Proxy-Objekt und seine Gruppenzugehörigkeiten.

Multicast Request Protocol

Der Einsatz von Multicast-IP ermöglicht es, LUS ohne Kenntnis ihrer konkreten IP-Adresse zu finden. Dazu werden UDP-Datagramme an die Multicast-Gruppe 224.0.1.85 und Port 4160 gesendet. Sie enthalten die Gruppennamen, an denen man interessiert ist, und Service IDs der bereits bekannten LUS. Ein LUS antwortet nur dann, wenn er noch nicht bekannt ist und zu einer der genannten Gruppen gehört. Er baut dazu eine TCP-Verbindung auf und übermittelt die Antwort wie im Unicast Discovery Protocol.

Multicast Announcement Protocol

Über das Multicast Announcement Protocol machen sich LUS im Netz bekannt. Dazu werden UDP-Datagramme an die Multicast-Gruppe 224.0.1.84:4160 geschickt. Sie enthalten die Service ID, die Netzadresse des LUS und die Gruppenzugehörigkeiten. Falls der LUS noch nicht bekannt ist und eine interessante Gruppe verwaltet, fordern interessierte Teilnehmer den LUS-Proxy über das Unicast Discovery Protocol an.

Normalerweise werden die beiden Multicast-Protokolle verwendet, um LUS im lokalen Netzsegment zu finden. Der Vorteil liegt vor allem im geringeren Konfigurationsaufwand: Der Benutzer muß in seinen Diensten und Anwendungen keine LUS-Adressen eintragen. Das verbirgt sich u.a. hinter dem Begriff "spontaneous networking". Die Multicast-Protokolle haben jedoch nur eine begrenzte Reichweite, da die meisten Router so konfiguriert sind, daß sie keine Multicast-Pakete weitergeben. Für Verbindungen über solche Netzsegmente hinaus, muß das Unicast Discovery Protocol verwendet werden.

Nach diesem theoretischen Exkurs, soll nun auch der Dienstnehmer "spontan" über die Multicast-Protokolle seinen Additionsdienst finden (Listing 5). Wesentlich für die Implementierung sind die Klasse LookupDiscovery und das DiscoveryListener-Interface. Die Klasse LookupDiscovery benutzt die Multicast-Protokolle um umliegende LUS zu finden. Welche LUS interessant sind, wird von den im Konstruktor angegebenen Gruppen bestimmt. Die Proxies der interessanten LUS werden via Event an die registrierten DiscoveryListener übergeben. Das Programm nimmt die Proxies in der discovered-Methode des DiscoveryListener-Interface entgegen. Die Methoden dieses Interfaces sollen keine komplexen Operationen

beinhalten, um die Bearbeitung schnell aufeinanderfolgender Events zu ermöglichen. Daher speichert das Programm die Proxies zwischen und weckt den main-Thread zur Bearbeitung. Dort werden die Lookup-Services wie in Listing 4 nach einem AddInterface durchsucht. Sobald dieses gefunden wurde, wird die Berechnung durchgeführt und das Programm beendet. Zur Vervollständigung des DiscoveryListener-Interface muß die Methode discarded implementiert werden.

```
import java.rmi.*;
import net.jini.core.lookup.*;
import net.jini.discovery.*;

public class JiniAddClient implements DiscoveryListener {

    private static LookupDiscovery ldisc;
    private static JiniAddClient jc;
    private static ServiceRegistrar[] registrars;

    public static void main(String args[]) throws Exception {
        System.setSecurityManager(new RMISecurityManager());

        ldisc = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        jc=new JiniAddClient();
        ldisc.addDiscoveryListener(jc);

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

        boolean found=false;
        while(!found) {
            ServiceRegistrar[] regs;
            synchronized(jc) {
                jc.wait();
            }
            regs=registrars;
            for(int i=0; i<regs.length; i++) {
                Object o=regs[i].lookup(template);
                if(o == null)
                    continue;
                found=true;
                System.out.println("3 + 5 = "+ ((AddInterface)o).add(3,5));
                break;
            }
        }

        ldisc.terminate();

        public void discovered(DiscoveryEvent discEv) {
            synchronized(jc) {
                registrars=discEv.getRegistrars();
                jc.notify();
            }
        }

        public void discarded(DiscoveryEvent e) {
        }
    }
}
```

Attribute

Die Beispiele wurden bisher bewußt einfach gehalten. Dabei ist ein weiteres wichtiges Konzept von Jini vernachlässigt worden: Attribute. Einem Dienst können beim Eintrag in einen LUS Attribute zugeordnet werden, die den Dienst näher beschreiben oder Programmcode enthalten. Beispiele für beschreibende Attribute sind Herstellername, Versionsnummer des Dienstes oder ein Icon zur Darstellung in graphischen Browsern. Neben diesen allgemeinen lassen sich auch spezielle, auf bestimmte Dienstklassen abgestimmte Attribute definieren. Für einen Druckdienst könnte man so zusätzlich angeben, ob es sich um einen Laser oder Tintenstrahldrucker handelt. Standardisierte Attribute existieren (bisher) nicht.

Attribute ermöglichen es, einerseits die Anzahl der Treffer einer Suchanfrage möglichst gering zu halten, andererseits einem menschlichen Benutzer Informationen für die Auswahl des "richtigen" Dienstes anzubieten.

Realisiert werden Attribute durch Java-Objekte. Jedes public-Feld einer Klasse, die von Entry bzw. AbstractEntry abgeleitet ist, wird dabei als Attribut interpretiert.

```
public class ServiceInfo extends AbstractEntry
    implements ServiceControlled {
    [...]
    public ServiceInfo(String name, String manufacturer, String vendor,
        String version, String model, String serialNumber) {
    [...]
    }

    public String name;
    public String manufacturer;
    public String vendor;
    public String version;
    public String model;
    public String serialNumber;
}
```

Ein Entry-Objekt stellt eine Menge von Attributen dar. Da einem Dienst mehrere solche Objekte zugeordnet werden können, speichert der LUS zu jedem Dienst demnach eine Menge von Attributmengen. Dadurch, daß Attribute in für Menschen verständlicher Form angezeigt werden sollen, entstehen Probleme wie z.B. Lokalisierung. Zu deren Lösung soll u.a. das Java-Beans-Konzept beitragen.

Hier ein Beispiel für das Registrieren eines Dienstes mit Attributen:

```
Entry[] attributes = new Entry[] { new Name("JiniAddService"),
    new ServiceInfo("Jini Enabled Adder Service",
        "az", "az", "1.0beta", "", "08/15") };
```

Das attributes-Array wird dann dem JoinManager im Konstruktor übergeben. Folgendes Code-Fragment setzt zusätzlich zum Typ ein Attribut als Suchkriterium ein:

```
Entry[] info = new Entry[] {
    new ServiceInfo(null, null, null, "1.0beta", null, null) };
ServiceTemplate template = new ServiceTemplate(null, cl, info);
```

Es wird nach einem Dienst gesucht, dem mindestens ein ServiceInfo-Objekt zugeordnet ist, das genau den Eintrag "1.0beta" enthält. Der Wert "null" wird wieder als Wildcard benutzt.

Attribute können auch nach der Registrierung noch geändert werden, um Zustandsänderungen des Dienstes wiederzugeben. Beispielsweise können so Statusinformationen in Attributen in den LUS hinterlegt werden

(z.B. Fehlerzustände von Geräten). Es wird jedoch davon ausgegangen, daß Attribute eher statischen Charakter haben. Aus Effizienzgründen wird empfohlen, sie nicht öfter als 1x pro Minute zu ändern.

Da die Attribute in normalen Java-Klassen gespeichert werden, können diese auch mit Programmcode versehen werden. Damit ergibt sich eine weitere interessante Anwendungsmöglichkeit der Attribute, nämlich das Hinterlegen eines GUI zur Benutzung des Dienstes. Dieser Ansatz erlaubt es, daß andere Dienste und Programme den Dienst ganz normal über das Java-Interface ansprechen können. Zusätzlich bringt der Dienst für den Fall, daß ein menschlicher Benutzer ihn direkt verwenden will, sein eigenes GUI gleich mit.

Fazit

Die Beispiele zeigen, wie wenig Aufwand notwendig ist, einen Dienst in ein Jini-System zu integrieren. Durch die Verwendung des Jini-Programmiermodells gewinnt man eine wesentlich höhere Flexibilität gerade was die Anbindung von Geräten betrifft. Der Vorteil liegt in den dynamischen Konfigurationsmöglichkeiten von Jini. Weitere interessante Aspekte (Transaktionen, verteilte Events, GUIs, JavaSpaces) haben im Rahmen dieses Artikels keinen Platz gefunden.