NetObjects
Dynamische Proxy-Architektur für Jini

Stephan Preuß
Universität Rostock, Fachbereich Informatik
Lehrstuhl für Informations- und Kommunikationsdienste
spr@informatik.uni-rostock.de
http://wwwtec.informatik.uni-rostock.de/IuK/staff/spr.html

Abstract. Anfang des Jahres 1999 wurde Jini (Java Intelligent Network Infrastructure) von Sun Microsystems als universelle Plattform für Netzwerkdienste und als Basis spontan vernetzter Communities vorgestellt. Im Bereich kleiner mobiler Geräte sowie in der Automation birgt die Jini-Technologie, vor dem Hintergrund konfigurationsfreier Bereitstellung von Diensten im Netzwerk, ein enormes Entwicklungs- und Innovationspotential. Dem entgegen steht allerdings die bei Jini verwendete Implementierungsstrategie, die durch die enge Verknüpfung mit Java 2 und RMI (Remote Method Invocation) den Einsatz auf relevanten Plattformen wie Personal Java oder Embedded Java und Systemen mit eingeschränkten Ressourcen fast unmöglich macht. Dieser Artikel befaßt sich mit Methoden, Jini auf Klein- und Kleinstgeräten sowie in der Automation einsetzbar zu machen. Es wird das Konzept von NetObjects vorgestellt, die eine schlanke, einfach zu handhabende Technologie zur Ausführung von Methoden entfernter Java-Objekte implementieren. Im Vordergrund steht dabei die Einschränkung der Funktionalität auf das für Jini erforderliche Minimum. NetObjects sind optimiert für die Ausführung in Umgebungen mit begrenzten Ressourcen, einfachste Programmierung und maximale Transparenz.

Stichworte: spontane Vernetzung, Netzwerkdienste, remote objects, Java, Jini

Klassifikation (CR 1998): C.2.2, C.2.3, C.2.4

1 Einführung und Motivation

Jini (Java Intelligent Network Infrastructure) ist eine Java-basierte Technologie zur automatischen Bereitstellung und Nutzung von Netzwerkdiensten [Mic99b, Edw99]. Konzepte wie Registration-Leasing und Event-Generation sowie die Plattformunabhängigkeit durch Java-Implementierung machen Jini ge-

eignet als Grundlage spontan vernetzter Communities [AWO+99, Pre00]. Technologien der spontanen Vernetzung, auch als Adhoc Networking bezeichnet, minimieren den Konfigurationsaufwand bei der Integration von Geräten oder Diensten in Netzwerkgemeinschaften. Besonders interessante Einsatzgebiete sind daher der Bereich des Mobile Computings mit hohen Anforderungen an die Konfigurationsdynamik sowie der Bereich der Automation mit einer großen Anzahl zu konfigurierender Geräte. Beiden Bereichen gemeinsam ist der Einsatz zweckoptimierter Systeme, die aufgrund von Größenbeschränkung (PDAs, Mobiltelephone) oder Kosten (Sensoren, Aktoren) mit begrenzten technischen Ressourcen ausgestattet sind. Im Widerspruch dazu setzen einige Technologien der spontanen Vernetzung leistungsfähige Universalsysteme voraus, unter der Annahme, daß in absehbarer Zeit auch Embedded Systems die Leistungsfähigkeit heutiger Arbeitsplatzrechner erreichen. Hier soll Jini genauer untersucht werden.

In diesem Artikel wird das Konzept von NetObjects erläutert, das dazu beiträgt, Jini an leistungsschwächere Systeme anzupassen und somit breiter einsetzbar zu machen. NetObjects implementieren eine Technik zum Ausführen von Methoden auf entfernten, in anderen JVMs (Java Virtual Machines) befindlichen Objekten auf Basis dynamisch erzeugter Proxy-Klassen. Design-Ziele sind dabei minimale Komplexität, maximale Transparenz und angepaßte Funktionalität.

1.1 Jini und Ressourcen

Als Jini zu Beginn des Jahres 1999 der Öffentlichkeit präsentiert wurde, ist es zum einen aufgrund des vielversprechenden Service-Konzeptes, zum anderen aufgrund der scheinbar schlanken Implementierung beworben worden. Code mit einer Größe von ca. 48K sollte es ermöglichen, Dienste in eine Jini-Community zu integrieren. Tatsächlich ist es möglich, ausschließlich mit Klassen der `net.jini.core`-Hierarchie an einer Jini-Community zu partizipieren, jedoch ohne die Dynamik und Flexibilität, die das System eigentlich ausmachen sollte. Diese erhält man erst durch den Einsatz von Klassen des Jini-Extension-Paketes. Ein Bestandteil davon ist z.B. `net.jini.discovery`, das die Basis-Klassen für Multicast-Discovery bereitstellt. Damit erhält ein neu in die Community eintretendes Mitglied die Möglichkeit, ohne vorherige Konfiguration, einen LookupService, also die Schaltzentrale der Community, zu finden und somit Dienste zu nutzen oder anzubieten.

Obwohl der LookupService der eigentliche Träger der Jini-Technologie ist, ist er nicht Bestandteil der Jini-Implementierung. Jini stellt lediglich die Interface-Definitionen zur Interaktion mit dem LookupService zur Verfügung. Diese Aussage trifft ebenso auf Schlüsselkomponenten wie Leases und Events zu. Da in einer Community wenigstens ein LookupService existent sein muß, kann sein Anteil am Code nicht vernachlässigt werden.

Die aktuelle Jini-Implementierung ist durch die Verwendung der RMI-Klassen (Remote Method Invocation) `java.rmi.RemoteException` und `java.rmi.MarshalledObject` praktisch untrennbar mit RMI verbunden. Außerdem benötigt der LookupService den RMI-Daemon (rmid).

Vergleicht man die eben gemachten Aussagen mit den Code-Größen in Ta-

belle 1, stellt man fest, daß Jini in der jetzigen Form für den Einsatz in Geräten mit begrenzten Ressourcen praktisch ungeeignet ist.

Paket	Größe	Klassenanzahl
Jini-Core	31K	41
Jini-Ext	68K	50
LookupService	262K	112
RMI	404K	200
$\sum$	765K	403

Die Basis dieser Werte bilden Jini1.0 und JDK1.2.2_05.

Tabelle 1. Code-Größen erforderlicher Pakete.

Das wird noch deutlicher, wenn man bedenkt, daß die Kernvoraussetzung für die Funktionsfähigkeit des Systems erst mit einem Großteil der Java 2-Runtime-Umgebung geschaffen ist.

Der nächste Abschnitt verdeutlicht, daß keine der bisher für ressourcenarme oder eingebettete Systeme verfügbaren Java-Implementierungen die Anforderungen von Jini erfüllt.

1.2 Jini und Java

Nachfolgend werden einige Bemerkungen zur Eignung bzw. Nichteignung existenter Java-Implementierungen als Ausführungsplattform von Jini gemacht.

JDK1.2 bildet die Grundlage der Jini-Implementierung und erfüllt in der Konsequenz alle Bedingungen.

JDK1.1 ist nicht geeignet als Plattform für Jini, da Jini auf Klassen in den Paketen `java.util, java.security, java.rmi` zurckgreift, die erst mit dem JDK1.2 eingeführt wurden.

Personal Java 1.2 basiert auf dem JDK1.1.8 und integriert zusätzliche Merkmale von Java 2 wie das feingranulare Sicherheitsmodell und das erweiterte `java.util`-Paket. Nicht enthalten ist die Java 2-RMI-Funktionalität [Mic00b].

Embedded Java 1.1 basiert auf dem JDK1.1.7 ohne Erweiterungen aus Java 2. Die meisten Pakete sind optional, um eine bessere Skalierbarkeit zu gewährleisten [Mic99a]. Für die weitere Entwicklung von Embedded Java hat Sun Microsystems die Integration von Jini angekündigt [Mic00c].

Java 2 MicroEdition (J2ME) ist verbunden mit einer neuen schlanken JVM (KVM), die allerdings in ihrer Funktionalität eingeschränkt ist [Mic00a]. Nachteilig für dynamische Umgebungen ist der zweistufige Verifikationsprozeß, der zum Teil außerhalb der Virtual Machine (offline) abläuft. Die Klassenbibliothek der J2ME implementiert ein proprietäres Netzwerkkonzept, das nicht zu `java.net` kompatibel ist. Weiterhin sind keine applikationsspezifischen ClassLoader zugelassen. Techniken wie RMI o.ä. können aus diesen Gründen nicht implementiert werden.

Kaffe ist kompatibel zu Personal Java 1.1, realisiert somit nur den Funktionsumfang des JDK1.1 und unterliegt daher denselben Einschränkungen [Tra99].

2 NetObjects

NetObjects implementieren eine Technologie zur Ausführung von Methoden entfernter Java-Objekte. Grundlage dafür bildet die *dynamische* Erzeugung *transienter* Proxy-Klassen, über deren Instanzen eine Interaktion mit dem entfernten Objekt realisiert wird. Der Funktionsumfang ist an die durch Jini gestellten Randbedingungen angepaßt, was eine sehr effiziente und kompakte Umsetzung erlaubt und somit auch den Einsatz auf Systemen mit eingeschränkten Ressourcen unterstützt.

2.1 Funktionsweise

Ziel von NetObjects ist die transparente Ausführung von Methoden von Objekten, die sich in entfernten JVMs befinden. Transparenz bedeutet in diesem Fall, daß weder Programmierer noch Nutzer besondere Aktionen zur Compile- bzw. Laufzeit auszuführen haben, um das System lauffähig zu machen. Solche Aktionen wären z.B. die Erzeugung von Proxy-Klassen zur Compile-Zeit oder der Einsatz eines HTTP-Servers zum dynamischen Nachladen von Klassen. NetObjects repräsentieren ein in sich geschlossenes Ausführungsmodell, das auf der in Abbildung 1 dargestellten Klassen-Metamorphose und dem Java-Reflection-API basiert.

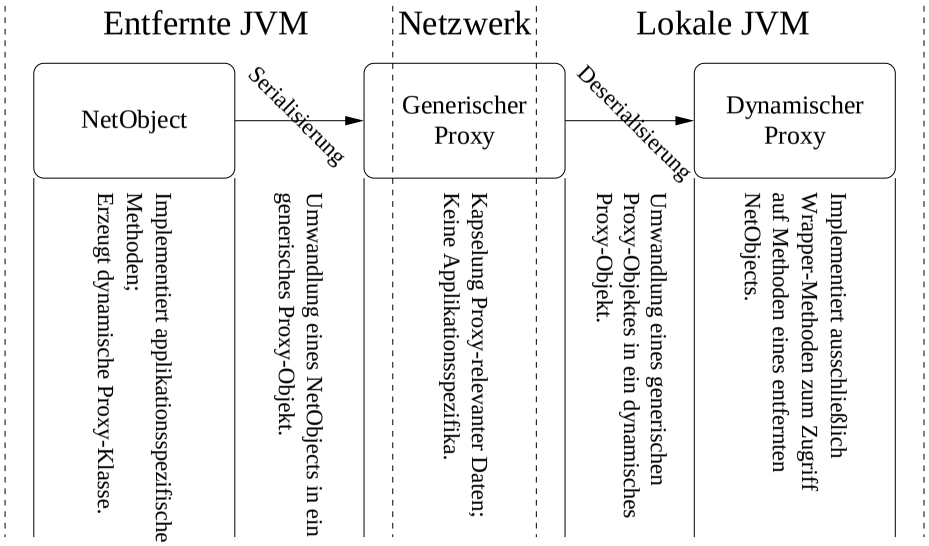


Abbildung 1. Klassen-Metamorphose.

Während der Instanzierung eines NetObjects wird über das Reflection-API eine Analyse der Klasse durchgeführt. Diese ermittelt alle als `public` definierten Methoden, die in einem als Marker fungierenden `Remote`-Interface deklariert wurden. Mit Hilfe eines Java-Assemblers [Eng99] wird eine transiente Proxy-

Klasse erzeugt, die als Wrapper für die entfernt aufrufbaren Methoden dient (siehe Abschnitt 2.4).

Während der Serialisierung eines NetObjects, wird dieses durch ein entsprechend parametrisiertes Objekt einer generischen Proxy-Klasse (`NetObject-Proxy`) ersetzt. Das passiert typischerweise dann, wenn es von einer JVM in eine andere transferiert wird. Die `NetObjectProxy`-Klasse ist in der anderen JVM bekannt und deren Instanzen damit deserialisierbar. Diese Vorgehensweise ist notwendig, da die entfernte JVM bisher nicht in der Lage ist, die Klasse des transienten Proxys zu lokalisieren und zu laden. Diese Funktion übernimmt der generische Proxy unterstützt durch einen speziellen ClassLoader. Dieser Ansatz bringt zwei Vorteile mit sich: Zum einem werden alle nachfolgend von der Instanz des transienten Proxys benötigten Klassen von dessen ClassLoader geladen. Dieser wiederum kommuniziert mit dem entsprechenden NetObject, in dessen Kontext alle weiter erforderlichen Klassen zu finden und somit nachladbar sind. Zum anderen müssen in der JVM, die den Proxy-Code ausführt, nur zwei zusätzliche Klassen bekannt sein, `NetObjectProxy` und `ProxyClassLoader`. Der transiente Proxy sowie alle weiteren erforderlichen Klassen werden in diese (die lokale) JVM injiziert. Die `NetObject`-Klasse selbst muß nicht bekannt sein. Dadurch läßt sich praktisch ein ultra-leichter Client implementieren.

Abbildung 2 stellt das Ausführungsschema entfernter Methoden dar. Der transiente Proxy verpackt die Methoden-Parameter in einen Container und übergibt diesen an eine generische `execute()`-Methode, welche die Verbindung zum entfernten NetObject herstellt und nach der Methoden-Ausführung einen Container mit dem Return-Code oder einer Exception erhält. Der Umweg der Ausführung über die `execute()`-Methode der `Execute`-Klasse dient ausschließlich der Reduktion der Komplexität des dynamischen Proxys sowie dessen erzeugender Instanz des `ProxyBuilders`. Die `Execute`-Klasse wird zur Laufzeit über den `ProxyClassLoader` aus dem Kontext des NetObjects geladen.

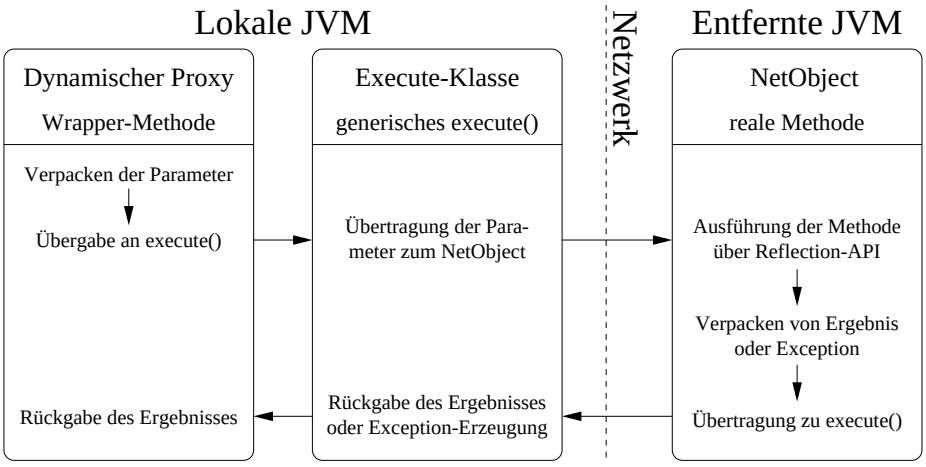


Abbildung 2. Ausführung entfernter Methoden mit NetObjects

Die augenblickliche Realisierung der NetObjects implementiert eine direkte Verbindung zwischen einem NetObject und dessen Proxy. Das impliziert die Verwendung eines Netzwerk-Ports pro NetObject und könnte sich in ressourcenkritischen Umgebungen als Nachteil herausstellen. Für spätere Implementierungen ist daher auch die Möglichkeit eines Connection-Schedulers geplant, der mit einer einzigen Instanz pro JVM alle darin existenten NetObjects bedient. Als Nebeneffekt dieses Vorgehens wird die Komplexität der NetObject-Klasse reduziert.

2.2 Parameterübergabe und Variablenzugriff

Besondere Beachtung bei der Arbeit mit entfernten NetObject-Methoden kommt der Parameterübergabe zu. Für die Übergabe von NetObjects gilt grundsätzlich Pass-by-Reference, sobald sie von einer JVM in eine andere transferiert werden, da hierbei die eine NetObject-Instanz gegen eine Proxy-Instanz getauscht wird. Für alle anderen Objekte dagegen und natürlich atomare Typen gilt Pass-by-Value, bedingt durch die Serialisierung und Deserialisierung. Ausgenommen davon sind RMI-Remote-Objects und Objekte von Klassen, die mit ähnlichen Verfahren arbeiten.

Instanz- und Klassen-Variablen von NetObjects sind nur über Access-Methoden zugreifbar, da die Variablen nicht in die Proxy-Klasse übernommen werden und somit nur im entfernten NetObject existieren. Die direkte Abbildung der Klassen- oder Instanz-Variablen in den entfernten Kontext entfällt allerdings schon aus technologischen Gründen, da Java keine effiziente Technik bietet, die einen Variablen-Zugriff indizieren und somit eine entsprechende Projektion initiieren könnte.

2.3 Garbage-Collection

NetObjects realisieren keinen Mechanismus zur verteilten Garbage-Collection. Das bedeutet, daß NetObject- und Proxy-Instanzen in ihren JVMs getrennt voneinander entsorgt werden können, wenn sie lokal nicht mehr referenziert werden. Da hier der Proxy vergleichbar mit einer Objekt-Referenz ist, ist dieser nicht weiter von Bedeutung. Die Entsorgung einer NetObject-Instanz bei noch vorhandenen Proxies (Referenzen), kann mit dem Verschwinden der entfernten JVM oder einem Abbruch der Netzwerkverbindung gleichgesetzt werden. Da diese Ereignisse in verteilten Systemen ohnehin entsprechend behandelt werden müssen, kann der Verzicht auf eine verteilte Garbage-Collection toleriert werden. Proxy-Instanzen reagieren mit einer `RemoteException` auf das Nichtvorhandensein ihres NetObjects, die von der Client-Applikation mit der Neubeschaffung eines Proxys behandelt werden müßte, da jeder NetObject-Instanz eine Proxy-Klasse zugeordnet ist.

2.4 Proxy-Erzeugung

Wie in Abschnitt 2.1 erwähnt, wird die transiente Proxy-Klasse während der Instanzierung eines NetObjects erzeugt. Im Analyseprozeß wird der Klassen-

Code über das Reflection-API nach Methoden durchsucht, die Interfaces implementieren, welche in der Vererbungshierarchie vom Marker-Interface `Remote` abgeleitet wurden. Für diese Methoden wird mit Hilfe der in [Eng99] enthaltenen Java-Assembler-Klassen eine adäquate Wrapper-Methode in die Proxy-Klasse eingefügt. Diese Wrapper-Methoden sind mit einem minimalen Umfang an Java-Bytecode-Instruktionen realisierbar. Daher ist es durchaus möglich, die bisher genutzten generischen Assembler-Klassen durch *eine* spezialisierte Klasse zu ersetzen, die nur den minimal erforderlichen Funktionsumfang realisiert.

Der `ProxyBuilder` arbeitet im gegenwärtigen Entwicklungsstand mit ca. 20% des JVM-Instruction-Sets. Unter Verzicht auf die Laufzeit-optimierten Bytecodes `[AILFD]LOAD_[0..3]` kann der Anteil auf 10% reduziert werden [Eng99, LY99]. Effektiv bedeutet das die Benutzung von 20 von ca. 200 Instruktionen.

Durch die dynamische Proxy-Generierung wird der Schritt der Erzeugung eines Proxys oder Stubs zur Compile-Zeit, wie z.B. bei RMI, überflüssig. Außerdem sind durch den eingebauten Class-Loading-Mechanismus keine zusätzlichen HTTP-Server erforderlich, um Klassen bei Bedarf nachzuladen.

2.5 Programmierung

Der Umgang mit NetObjects erfordert kein tieferes Verständnis der Funktionsweise, sondern läßt sich auf ein einfaches Schema reduzieren.

Methoden, die entfernt ausführbar sein sollen, werden in einem Interface deklariert, das vom Marker-Interface `Remote` abgeleitet ist. Die Implementierung dieser Methoden erfolgt in einer von `NetObject` abgeleiteten Klasse. Der Zugriff auf die entfernten Methoden erfolgt ausschließlich über das selbst definierte Interface.

```
public interface RemoteDate extends Remote {
  public java.util.Date getDate();
}

public class MyRemoteDate extends NetObject
implements RemoteDate {
  public java.util.Date getDate() {
    return new java.util.Date();
  }
}
```

Auch die Funktionalität vorhandener Interfaces läßt sich einfach mit der NetObject-Technik benutzen.

```
public interface RemoteMap
extends java.util.Map, Remote { }
```

Dieses Vorgehen erlaubt es, auf dem transienten Proxy-Objekt mit dem originalen Interface (hier `java.util.Map`) zu operieren.

3 NetObjects und Jini

Die Integration von NetObjects in Jini dient primär dem Zweck, Jini und RMI zu entkoppeln und das System dadurch schlanker zu gestalten.

Jini benutzt nur einen verschwindend geringen Teil der RMI-Funktionalität, sodaß der Einsatz eines Systems mit minimaler Funktionalität zu einer wesentlichen Reduzierung der Code-Größe sowie zur Verringerung der Komplexität des Gesamtsystems beiträgt. Bei entsprechend leistungsfähigen Systemen bleibt RMI als Möglichkeit der Interaktion zwischen Service-Provider und Service-Client weiterhin erhalten. In Systemen mit unzureichenden Ressourcen können NetObjects, natürlich mit entsprechend eingeschränktem Funktionsumfang, auch als allgemeine Ausführungsplattform für entfernte Methoden dienen.

Im Speziellen benutzt Jini die Klasse `java.rmi.MarshalledObject`, um in der Discovery-Phase Instanzen des LookupServices an die Discovering-Entity zu übergeben [Edw99]. Damit wird in der Folge die Verwendung des RMI-ClassLoaders impliziert. Dieser sorgt dafür, daß die für die Nutzung der entfernten Methoden benötigten Klassen von einem HTTP-Server nachgeladen werden können. Der HTTP-Server ist in der Property `java.rmi.server.codebase` der entfernten JVM spezifiziert und wird bei der Serialisierung mit in den Object-Stream geschrieben.

Weiterhin sind die Zugriffe auf den LookupService via RMI realisiert, sodaß entfernte Zugriffe auf diesen, z.B. über die Interfaces `net.jini.core.lookup.ServiceRegistrar`, `net.jini.core.lookup.ServiceRegistration` oder `net.jini.core.lease.Lease`, eine `java.rmi.RemoteException` erzeugen können.

Der Übergang von RMI zu NetObjects in der Jini-Architektur ist also praktisch verbunden mit einer Neuimplementierung eines LookupServices, der Anpassung der Unicast-Discovery-Prozedur sowie dem Austausch der `java.rmi.RemoteException` gegen die `RemoteException` der NetObjects-Technologie. Letzteres ist der einzige Punkt, von dem auch existente Applikationen betroffen sein können, die Behandlungs-Routinen für `java.rmi.RemoteException` implementieren. Eine Variante, das zu umgehen, wäre z.B. die Exception-Behandlung auf der Ebene von `java.lang.Throwable` oder `java.lang.Exception`.

Der Einsatz von NetObjects ohne Jini ist prinzipiell auch möglich, aber nicht unbedingt sinnvoll, denn Jini bzw. dessen Lookup- und Discovery-Mechanismen ermöglichen erst die Akquisition von Proxy-Objekten. Bei RMI übernimmt die RMI-Registry die Verteilung der Proxies. Bei NetObjects wurde vor dem Hintergrund des Entwurfs für die Jini-Architektur auf eine separate Registry verzichtet, da die Funktionalität mit dem LookupService bereitgestellt wird.

4 Vergleichbare Arbeiten

Es existieren eine Reihe von Arbeiten und Studien mit dem Focus der Performance-Optimierung von RMI. So wird z.B. in [NPH99] ein leistungsoptimiertes RMI-System vorgestellt. Es ist syntaktisch kompatibel zu Java-RMI, kommt ohne native Systemaufurfe aus und erreicht Geschwindigkeitsvorteile von durch-

schnittlich 45% bei entfernten Methodenaufrufen. Bestandteil ist auch eine optimierte Objekt-Serialisierung.

Die Firma ProSyst hat mit *mBedded*^{TM} *Server* ein OSGi-konformes (Open Services Gateway initiative) Produkt auf dem Markt, das Jini als einen Service integriert. Es werden Java1.1- und Java1.2-kompatible Laufzeitumgebungen unterstützt. Der eingesetzte Jini-LookupService ist eine Neuentwicklung, die nicht als RMI-Remote-Object ausgelegt ist. Er basiert auf einem proprietären Protokoll (ProSyst Lookup Service Protocol). Dadurch wird die Abkoppelung von RMI erreicht [Pro00]. Die Kompatibilität zwischen Java1.1 und der Jini-Referenzimplementierung, die auf Java1.2 aufbaut, wird durch Hinzufügen der fehlenden Klassen aus den Paketen `java.util` und `java.rmi` erreicht.

NetObjects unterscheiden sich von den beiden angeführten Systemen in untrschiedlichen Punkten: Es ist nicht das Ziel, die Kompatibilität zu Java-RMI herzustellen bzw. dessen Funktionsumfang abzubilden oder die Leistungsfähigkeit zu steigern. Der Schwerpunkt liegt, wie in Abschnitt 1 bereits erwähnt, in der Schaffung einer Code-Größen-optimierten Ausführungsplattform für entfernte Methoden mit minimalem (angepaßtem) Funktionsumfang, um RMI in Jini ersetzen zu können und gleichzeitig einen einheitlichen, einfach zu nutzenden Service-Access-Mechanismus in ressourcenschwachen Umgebungen bereitzustellen. Kernpunkt von NetObjects ist die dynamische Proxy-Erzeugung, durch die mehr Komfort in der Programmierung und mehr Transparenz in der Nutzung erreicht wird, da ein entferntes Objekt als eine Einheit präsentiert wird und die Trennung zwischen dem realen Objekt und dessen Proxy automatisiert im Hintergrund abläuft.

5 Entwicklungsstand

Die Implementierung der NetObjects ist abgeschlossen, sie umfaßt nur neun Klassen mit einer gesamten Code-Größe von ca. 25K. Dazu kommen 100K des generischen Assemblers, die bei entsprechender Spezialisierung um mindestens 50% reduziert werden können, sodaß ein vollständiges System mit weniger als 100K realisierbar ist. Der Vergleich zu Tabelle 1 zeigt eine deutliche Reduzierung. Client-seitig werden nur ca. 6K zusätzlich benötigt, zu denen natürlich zur Laufzeit wenigstens noch der Code des transienten Proxys mit einigen 100 Byte hinzukommt.

Die Portierung von Jini auf NetObjects ist weitgehend abgeschlossen. Derzeit laufen die Arbeiten zur Implementierung des LookupServices und zur Anpassung von Jini an Java1.1.

Folgende Ziele sind mit NetObjects erreicht worden:

- wesentlich verringerter Ressourcenverbrauch gegenüber RMI,
- kein HTTP-Server erforderlich, um Klassen dynamisch nachzuladen,
- keine offline Proxy-Generierung erforderlich.

6 Ausblick

Ausgehend von der Analyse von Technologien der spontanen Vernetzung in [Pre00] wurde das gesamte System um die Fähigkeit erweitert werden, Service-Lookups ohne einen zentralen LookupService auszuführen, um die Funktionalität auch in kleinsten Communities (z.B. zwei PDAs) zu gewährleisten. Weiterhin ist geplant, das System von IP als Netzwerkschicht abzukopplen. Dadurch eröffnet sich die Möglichkeit der Verwendung fast beliebiger Netzwerkschichten (z.B. CAN, PROFIBUS).

Literatur

[AWO+99] Ken Arnold, Ann Wollrath, Brian O'Sullivan, Robert Scheifler, and Jim Waldo. *The Jini Specification*. Addison-Wesley, 1999.

[Edw99] W. Keith Edwards. *Core Jini*. Prentice Hall, 1999.

[Eng99] Joshua Engel. *Programming for the Java Virtual Machine*. Addison Wesley, 1999.

[LY99] Tim Lindholm and Frank Yellin. The Java Virtual Machine Specification. http://java.sun.com/docs/books/vmspec/, 1999.

[Mic99a] Sun Microsystems. EmbeddedJava Application Environment Specification (Version 1.1). http://java.sun.com/products/embeddedjava/spec/index.html, 1999.

[Mic99b] Sun Microsystems. Jini Connection Technology. http://java.sun.com/products/jini/, 1999.

[Mic00a] Sun Microsystems. JSR-000030 J2ME Connected, Limited Device Configuration. http://java.sun.com/aboutJava/communityprocess/first/jsr030/index.html, 2000.

[Mic00b] Sun Microsystems. PersonalJava Application Environment Specification (Version 1.2). http://java.sun.com/products/personaljava/, 2000.

[Mic00c] Sun Microsystems. Technical Overview of EmbeddedJava Technology. http://java.sun.com/products/embeddedjava/overview.html, 2000.

[NPH99] Christian Nester, Michael Phlippsen, and Bernhard Haumacher. Effizientes RMI für Java. In *Java-Informations-Tage 1999*, pages 135–148. Springer Verlag, 1999.

[Pre00] Stephan Preuß. Overview of Spontaneous Networking - Evolving Concepts and Technologies. In *Rostocker Informatik-Berichte*, volume 24, pages 113–123. Fachbereich Informatik der Universität Rostock, 2000.

[Pro00] ProSyst. mBedded Server White Paper. http://www.prosyst.com/, 2000.

[Tra99] Transvirtual. Kaffe Product Architecture. http://www.transvirtual.com/products/architecture.html, 1999.