Nr.: FIN-010-2010

Grundlagen und Einsatz von Jini für serviceorientierte Architekturen

Veit Köppen, Andreas Meier,
Michael Soffner, Norbert Siegmund

AG Datenbanken

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Technical report

Nr.: FIN-010-2010

Grundlagen und Einsatz von Jini für serviceorientierte Architekturen

Veit Köppen, Andreas Meier,
Michael Soffner, Norbert Siegmund

AG Datenbanken

Technical report (Internet)
Elektronische Zeitschriftenreihe
der Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg
ISSN 1869-5078

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Impressum (§ 5 TMG)

Herausgeber:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Der Dekan

Verantwortlich für diese Ausgabe:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Veit Köppen
Postfach 4120
39016 Magdeburg
E-Mail: veit.koeppen@iti.cs.uni-magdeburg.de

http://www.cs.uni-magdeburg.de/Technical_reports.html
Technical report (Internet)
ISSN 1869-5078

Redaktionsschluss: 15.12.2010

Bezug: Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Dekanat

Technical Report – Fakultät für Informatik
No. 10-2010

Grundlagen und Einsatz von Jini für serviceorientierte Architekturen

Veit Köppen · Andreas Meier · Michael Soffner · Norbert Siegmund

erste Version: 28. Juni 2010 / finale Version: 15. Dezember 2010

Zusammenfassung Verteilte Softwaresysteme bilden in der heutigen IT-Landschaft wesentliche Bestandteile. Aufgrund ihrer vernetzten Architektur sind sie geeignet, spezialisierte Systeme miteinander zu verbinden, um somit Aufgaben effektiv und effizient lösen zu können. Neben dem klassischen Client-Server-Paradigma [TS03] stellen die *serviceorientierten Architekturen* (SOA) einen weiteren Vertreter verteilter Softwaresysteme dar. Durch ihre Abstraktion der Funktionalität in Dienste eignen sie sich insbesondere zur Beschreibung von Geschäftsprozessen, sind aber auch in der Lage, andere IT-Bereiche abzudecken. Das auf Java basierende Framework *Jini* wurde zur Entwicklung von SOAs geschaffen und ist Gegenstand dieser Arbeit. Nach der Vorstellung grundlegender Prinzipien von Jini, wird die Entwicklung einer serviceorienterten Architektur mit Jini anhand eines kleinen Beispiels vermittelt. Diese Applikation ist in der Lage, Grafiken von einem Gerät an andere Netzwerkteilnehmer zu streamen. Dabei werden sowohl die eigentliche Programmierung als auch Stärken und Schwächen von Jini beleuchtet, sodass ein fundierter Überblick über diese Technologie entsteht. Darauf aufbauend erfolgt abschließend eine Einschätzung, wann der Einsatz von Jini sinnvoll ist und welche Alternativen sich gegebenenfalls anbieten.

Ansprechpartner:
V. Köppen
Otto-von-Guericke Universität Magdeburg
Universitätsplatz 2
39106 Magdeburg, Germany
E-Mail: veit.koeppen@ovgu.de

1 Einleitung

Im Zuge der Entwicklung von hochkomplexen Netzwerkanwendungen hat sich die Softwaretechnik kontinuierlich von low-level Implementierungen über Middlewares hin zu den serviceorientierten Architekturen (SOA) gewandelt. Mit dieser Entwicklung soll dem Fakt Rechnung getragen werden, dass verteilte Softwaresysteme einerseits immer komplexer werden, aber andererseits die Forderung nach einer möglichst losen Kopplung dieser Bestandteile besteht. Im Rahmen des vom Bundesministerium für Bildung und Forschung (BMBF) geförderten Projektes *Virtuelle und Erweiterte Realität für höchste Sicherheit in eingebetteten Systemen* (ViERforES)[1] ist es im Teilprojekt *Interoperabilität* notwendig, Erfahrungen in der Nutzung von SOAs zu sammeln. Dazu sollen vor allem die Realisierbarkeit, die Verwendbarkeit aber auch die Stärken und Schwächen dieser verteilten Architekturen untersucht werden. Für ViERforES besteht dabei ein besonderes Interesse in einfachen Entwickungsmethoden als auch die Erzielung einer große Flexibilität für sich ständig ändernde Netzwerkstrukturen, sodass Systemen vom eingebetteten Bereich bis hin zu Servern interagieren können.

Ein Ansatz, der diese Forderungen zu erfüllen scheint, ist die serviceorientierte Jini-Technologie[2], die mittlerweile im Projekt *River* der Apache Software Foundation weiterentwickelt wird. Ende der 90er Jahre präsentierte Sun Microsystems ein auf Java basierendes Framework, das eine hochflexible Vernetzung von Geräten bieten soll: Jini [New00a, New06a]. Ziel dieses Frameworks ist es, den Java-Gedanken, *write once, run anywhere*, derartig zu erweitern, dass viele Java-basierende Geräte miteinander kommunizieren können, ohne sich um Feinheiten der Kommunikation kümmern zu müssen. Vielmehr sollen auch die Implementierungsdetails der Geräte verborgen und stattdessen einfach zu nutzende Dienste angeboten werden. Neben diesem Vorteil liegt die Stärke von Jini auch in der hohen Robustheit und Flexibilität, was dynamische Netzwerke aus nur kurzzeitig verfügbaren Geräten mit einschließt. So können Geräte, die Dienste anbieten oder nutzen, schnell ein Netzwerk bilden, welches auch Ausfälle von Teilkomponenten erkennen und adäquat darauf reagieren kann [ASW^{+}99].[3]

Im Rahmen des Teilprojekt Interoperabilität wurde ein serviceorientierter Ansatz basierend auf der Jini-Technologie im Kontext der Sicherstellung der Interoperabilität verwendet. Es konnte gezeigt werden, dass sowohl reale Systeme als auch virtuelle Systeme über eine solche Architektur verbunden werden können [KSSS09]. Darüber hinaus konnten am Anwendungsszenario eines Flughafen Logistik Hubs Konzepte für die dynamische Anpassung von Services gezeigt werden [SPS^{+}09]. Dies bietet die Möglichkeit auf Systemausfälle zu reagieren und dynamisch neue Geräte in das Jini-Netzwerk zu integrieren. In diesem technischen Report werden die technischen Details beschrieben, die die Grundlagen der oben genannten Arbeiten bilden. Dabei werden zunächst allgemein die Anforderungen und die Funktionsweise der Jini-Technologie beschrieben. Anschließend wird eine Implementierung exemplarisch vorgestellt und die Stärken und Schwächen der Jini-Technologie zusammenfassend herausgearbeitet. Den Abschluss bildet die Betrachtung alternativer Technologien zur Realisierung verteilter Softwaresysteme sowie das Fazit mit einem Ausblick auf zukünftige Arbeiten.

---

[1] ViERforES, http://vierfores.de, letzter Zugriff: 10.12.2010

[2] Apache River, http://incubator.apache.org/river/index.html, letzter Zugriff: 10.12.2010

[3] Jini Architecture Specification, http://www.jini.org/wiki/Jini_Architecture_Specification, letzter Zugriff: 10.12.2010

2 Grundlagen der Jini-Architektur

Im folgenden Kapitel werden die Grundlagen der Jini-Architektur vorgestellt. Dazu wird zuerst die Zielsetzung von Jini wiedergegeben und erläutert, wie Jini gängigen Netzwerkproblemen begegnen kann. Allerdings stellt Jini auch grundlegenden Anforderungen an das Netzwerk, die im Anschluss dargelegt werden. Abschließend wird die Funktionsweise von Jini genauer besprochen.

2.1 Zielsetzung von Jini

Jinis Ziel ist es, dynamische, robuste und leistungsfähige Netzwerke zu ermöglichen, die sowohl kleine eingebettete Geräte wie Kaffeemaschinen, PDAs und Sensormodule als auch leistungsfähigere Geräte wie PCs, Workstations oder gar Server umfassen können. Dazu wurden in Jini Mechanismen implementiert, um den sogenannten *acht Trugschlüssen verteilter Anwendungen* [New06b] zu begegnen. Diese Trugschlüsse stellen Voraussetzungen an das Netzwerk dar, die häufig bei der Entwicklung von Netzwerkanwendungen angenommen werden, aber grundsätzlich nicht garantiert werden können. Die acht Trugschlüsse lauten folgendermaßen:

1. Das Netzwerk ist immer verfügbar.
2. Die Bandbreite ist unendlich hoch.
3. Das Netzwerk ist sicher.
4. Der Netzwerkaufbau ändert sich nicht.
5. Es gibt nur einen Administrator.
6. Es fallen keine Transportkosten an.
7. Das Netzwerk ist homogen.
8. Die Wartezeit ist null.

Jini integriert verschiedene Konzepte, um den Nachteilen, die durch die Annahme dieser Trugschlüsse entstehen, zu begegnen. So ist die Verwendung von Kompression oder Verschlüsselung möglich, um eine schnelle und trotzdem sichere Netzwerkanbindung zu ermöglichen. Ferner benötigt ein Jini-Netzwerk nahezu keine Administration, sofern vom einmaligen Bereitstellen eines zentralen Servers zur Dienstfindung und zum Download benötigter Softwarekomponenten abgesehen wird. Durch die Auslegung als SOA nutzen zudem alle Bestandteile Dienste und sind bei der Nutzung unabhängig davon, wie die Dienste implementiert wurden und ob das Netzwerk aus homogenen Teilnehmern besteht.

Trotz dieser Zielsetzung und der diversen Mechanismen, die ein, auch im Hinblick auf die Plattform unabhängiges Implementierungsmodell suggerieren, stellt Jini gewisse Anforderungen an die Laufzeit- und Netzwerkumgebung. Da diese den Einsatzbereich, vor allem im eingebetteten Bereich, stark einschränken können, ist eine detailliertere Betrachtung dieser Voraussetzungen notwendig.

2.2 Anforderungen von Jini

Jini basiert auf dem Java-eigenen Ansatz zur Entwicklung verteilter Anwendungen, der als *Remote Method Invocation* (RMI) [Gro01] bekannt ist. RMI setzt, neben dem Fakt,

dass in Java implementiert werden muss, auch zwingend eine konfigurierte TCP/IP-Verbindung voraus. Besitzen Entwickler außerdem das Wissen über die grundlegende Struktur des Netzwerks, so sind sie in der Lage, die verwendeten Dienste miteinander zu verbinden. Sobald diese Verbindung aufgebaut ist, kümmert sich RMI selbständig um die Kommunikation zwischen den verteilten Softwarebestandteilen. Entfernte Methodenaufrufe an verteilte Objekte werden dabei automatisch in Datenströme verpackt, sodass sich diese Aufrufe aus Entwicklersicht nicht von Aufrufen auf lokalen Objekten unterscheiden.

Die Fixierung auf einen vorhandenen TCP/IP-Stack nebst passend konfigurierter Verbindung sowie die notwendige Kenntnis über die Netzwerkstruktur erschweren die Schaffung von heterogenen, flexiblen Netzwerkanwendungen. Ein Ziel der Entwicklung von Jini 2.0, das 2003 von Sun veröffentlicht wurde, war es daher, einige dieser Limitierungen aufzuheben. So wurde mit *Jini Extensible Remote Invocation* (JERI)[4] [New06c] eine Technik integriert, die nicht nur eine Kommunikation über TCP/IP erlaubt, sondern auch die Verwendung anderer Kommunikationsprotokolle ermöglicht. So existieren JERI-Implementierungen, die eine Übertragung via Firewire (IEEE 1394) oder auch eine Anbindung an eine *Common Object Request Broker Architecture* (CORBA) realisieren. Die Anforderung, dass eine genutzte Kommunikationsschnittstelle korrekt konfiguriert wurde, gilt dagegen weiterhin. Im Fall von TCP/IP wird somit eine gültige IP-Adresse und eine korrekt konfigurierte Subnetzmaske vorausgesetzt. Dieser Umstand erschwert den Aufbau eines flexiblen Netzwerks, wobei durch Techniken wie das *Dynamic Host Configuration Protocol* (DHCP) eine automatische Konfiguration von Netzwerkgeräten erfolgen kann.

Die Zusammenführung der Dienste geschieht in Jini 2.0 wahlweise wie bei RMI unter Kenntnis des Netzwerks, insbesondere unter Benutzung von Netzwerkadressen, oder auch völlig autonom. Dazu finden sich Dienste in einem Jini-Netzwerk mittels spezieller Broadcast-Nachrichten. Die autonome Dienstfindung setzt aber voraus, dass zentrale Netzwerkkomponenten wie Router, Gateways oder Firewalls diese Nachrichten auch passieren lassen, was im Allgemeinen nicht garantiert werden kann. Ferner wird eine Art Server benötigt, der als Vermittlung zwischen den Jini-Diensten fungiert.

Der vermeintlich größte Nachteil ist die Voraussetzung, dass die serviceorientierte Architektur in Java entwickelt werden muss. Eingebettete Systeme werden aufgrund ihrer starken Ressourcenbeschränkung häufig sehr hardwarenah in C/C++ oder vereinzelt auch in Assembler entwickelt. Java dagegen ist zwar eine leistungsfähige und einfache Sprache, setzt aber ein gewisses Maß an Systemresourcen voraus. Um solche stark limitierten Geräte trotzdem in einem Jini-Netzwerk nutzen zu können, wurden die so genannten *Surrogates* [Kum02] entwickelt. Unterstützt durch eine passende Programmierschnittstelle können nicht Java-fähige Geräte in das Netzwerk eingebunden werden, indem ein anderes Gerät ein Surrogate, einen Stellvertreter, bereitstellt. Dieser Stellvertreter stellt eine Brücke zwischen verschiedenen Kommunikationsprotokollen dar, indem er Nachrichten zwischen dem Jini-Netzwerk und dem Zielgerät in das jeweils benötige Protokoll übersetzt. Dabei muss allerdings beachtet werden, dass Netzwerkeigenschaften, wie Flexibilität, Robustheit oder auch Performance, die Jini garantieren will, auch durch das Surrogate erhalten bleiben müssen.

[4] Jini Extensible Remote Invocation, http://www.artima.com/intv/jeri.html, letzter Zugriff: 10.12.2010

2.3 Funktionsweise von Jini

Zum einfacheren Verständnis wird im Folgenden immer von einer TCP/IP-Verbindung ausgegangen. Andere Kommunikationsprotokolle können in der Regel ähnlich genutzt werden, unterscheiden sich aber in einigen Bestandteilen, wie zum Beispiel in der Adressierung von Netzwerkgeräten.

Dienste werden bei Jini über zwei Bestandteile definiert:

1. Ein Java-Interface spezifiziert den Dienst und seine verfügbaren Methoden.
2. Eine oder mehrere Java-Klassen implementieren als Classdateien diese Spezifikation.

Das Interface eines Dienstes muss dem Nutzer eines Dienstes zur Verfügung stehen, da nur darüber der benötigte Dienst ausfindig gemacht und genutzt werden kann. Zudem stellt das Interface die nutzbaren Methoden in Form gängiger Java-Methoden bereit und erlaubt somit den Aufruf von Methoden des Dienstes durch Clients. Die Implementierung kann dagegen dynamisch von einem *Classserver*, der vergleichbar zu einem Webserver ist, bezogen werden.

Da Jini-Dienste anderen Softwarekomponenten Funktionen zur Verfügung bereitstellen, werden sie auch als *Server* bezeichnet. Die Nutzer von Jini-Diensten werden der klassischen Nomenklatur folgend als *Client* bezeichnet. In Abbildung 1 ist die allgemeine Funktionsweise eines Clients im Jini-Netzwerk dargestellt. Die Unterteilung in die Phasen Dienstsuche und Dienstnutzung wird im Folgenden näher erläutert.

2.3.1 Dienstsuche

Zur Verbindung von Komponenten auf Basis einer SOA, müssen zuerst die Dienste gefunden werden, die anschließend miteinander interagieren sollen. Für die Dienstsuche bietet Jini zwei Mechanismen an: *Unicast-Discovery* und *Broadcast/Multicast-Discovery*. Unicast-Discovery setzt ähnlich wie bei RMI eine Kenntnis des Netzwerks voraus. Dazu wird direkt unter Angabe einer Geräteadresse, zum Beispiel eine IP-Adresse, eine Verbindung zu dem Gerät hergestellt, dass den Dienst zur Verfügung stellt. Anschließend können die Methoden des Dienstes wie lokale Methoden genutzt werden.

Die im Rahmen des ViERforES-Projektes wesentlich interessantere Variante ist dagegen die Suche nach Diensten via Broadcast-Discovery. Grundlage dieses Mechanismusses ist ein so genannter *Lookup-Service*, ein Classserver und eine Datenkommunikation per User Datagram Protocol (UDP).

Lookup-Services bilden einen der grundlegenden Bausteine jedes multicast-basierten Jini-Netzwerks, da sie die notwendigen Informationen über alle bekannten Dienste speichern und aktuell halten. Ausgeführt auf zentralen Computern, erlauben sie es den Jini-Servern, sich bei ihnen mit *Proxyobjekten* zu registrieren. Diese Proxyobjekte werden bei der Jini-Architektur dazu verwendet, Methodenaufrufe vom Client über das Netzwerk an die eigentlichen Jini-Server weiter zu reichen. Durch Jinis konsequentes Design sind die Lookup-Services selbst echte Jini-Dienste und besitzen daher auch dieselben Eigenschaften wie andere Dienste.

Soll ein Client (vgl. Abb. 1) einen bestimmten Dienst im Netzwerk nutzen, so sendet er zuerst auf einem definierten Port eine Anfrage nach Lookup-Services via UDP in das gesamte Netzwerk. Ein oder mehrere Lookup-Services hören auf diesem Port und warten kontinuierlich auf die Ankunft einer solchen Nachricht. Als Antwort auf

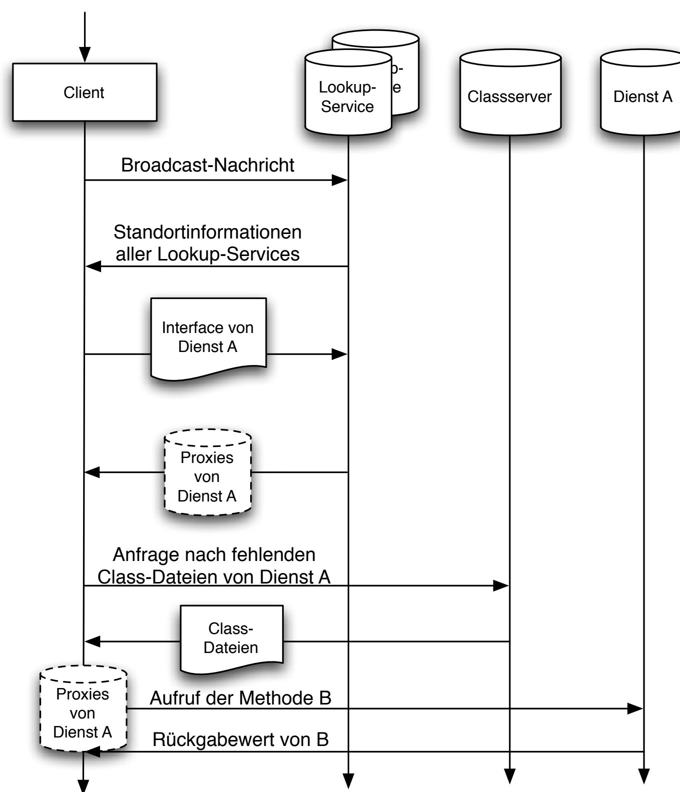

Abb. 1 Allgemeine Funktionsweise eines Jini-Clients

diese Anfrage senden sie ihre Netzwerkadresse an den Client. Dieser kann daraufhin die Lookup-Services via TCP/IP kontaktieren, sodass eine direkte Kommunikation zwischen dem Client und einem Lookup-Service möglich ist. Durch Angabe des Java-Interfaces des gewünschten Dienstes, sendet der Client eine Anfrage nach Implementierungen dieses Dienstes an die Lookup-Services. Diese antworten mit einer Liste aller Proxyobjekte, die den gewünschten Dienst implementieren.

2.3.2 Dienstnutzung

Nachdem der Client eine Anfrage an die Lookup-Services nach Diensten gestellt und eine Liste der vorhandenen Dienste erhalten hat, kann er sich nun an diese direkt wenden. Dazu lädt der Client in einem ersten Schritt fehlende Classdateien vom zentralen Classserver. Dieser Server stellt somit die notwendigen Implementierungen der Dienste bereit und verbessert dadurch die Kommunikationsstruktur im Netzwerk. Im Fall, dass der optionale Classserver nicht verfügbar ist, müssen die Clients dagegen die Implementierungen der zu nutzenden Dienste bereits besitzen. Sobald der Client die Implementierungen der Dienste besitzt, können die Methoden des Dienst-Interfaces ohne Einschränkungen genutzt werden. Methodenaufrufe, Parameterübergaben oder die

Nutzung von Rückgabewerten verhalten sich für Entwickler genauso wie bei der Nutzung von Methoden lokal vorhandener Objekte. Jini, beziehungsweise JERI, übernimmt dazu vollständig die Kommunikation zwischen Client und adressierten Diensten. Dazu nutzt es die Proxyobjekte der Dienste. Diese Stellvertreter stellen eine Referenz auf das reale Dienstobjekt dar, welches sich auf dem entsprechenden Gerät, dass den Dienst bereitstellt, befindet. Beim Aufruf einer Dienstmethode konvertiert JERI mithilfe der Referenz die Parameter und sendet alle nötigen Informationen an das reale Dienstobjekt im Netzwerk. Werden Objekte als Parameter übergeben, so serialisiert JERI diese selbständig. Diese Funktionalität setzt aber voraus, dass die Objekte das von Java bereits bekannte *Serializable-Interface* implementieren. Auf der Serverseite konvertiert JERI die empfangenen Daten zurück, um den eigentlichen Methodenaufruf ausführen zu können. Liefert eine Methode einen Rückgabewert, so schreibt JERI diesen Wert in einen Datenstrom, schickt diesen an den Client und konvertiert dort wieder die Daten in das Ursprungsformat.

Entwicklern eines Clients bleiben dabei die technischen Details verborgen. Sie können sich vollständig auf die eigentliche Aufgabe konzentrieren, was die Programmierung verteilter Applikationen wesentlich erleichtert. Die klassische Socketprogrammierung vermag durch ein schlankeres Protokoll zwar eine höhere Leistung erzielen, die Softwareentwicklung verläuft dagegen aber nicht annähernd so komfortabel und einfach, wie es mit Jini der Fall ist.

2.3.3 Dienstregistrierung

Die Dienstregistrierung erfolgt grundsätzlich ähnlich. In einem ersten Schritt sendet der Jini-Server eine Broadcastnachricht und wartet auf die Antwort der Lookup-Services. Sobald diese eintrifft, überträgt der Dienst ein Proxyobjekt an die Lookup-Services und registriert sich dort mit einem *Lease*. Leases repräsentieren eine bestimmte Zeitdauer und definieren, wie lange eine Dienstregistrierung bei einem Lookup-Service gültig sein soll. Diese Angabe ist aber für den Lookup-Service nicht bindend, denn er kann selbständig entscheiden, welches Zeitkontingent er einem Dienst zugesteht. Läuft ein Lease aus, dass heißt die verbleibende Zeitdauer bis zum Ablauf der Registrierung nähert sich gegen null, so hat der Dienst die Möglichkeit, diesen Lease und somit seine Registrierung beim Lookup-Service zu verlängern.

Der Leasemechanismus soll garantieren, dass Dienste, die nicht mehr verfügbar sind, auch nicht mehr im Lookup-Service gespeichert werden müssen. Allerdings ist dieser Mechanismus aufwändiger zu implementieren als eine selbstständige Abmeldung der Dienste beim Lookup-Service. Im Gegenzug besitzt diese Variante aber den großen Vorteil, dass Dienste, die zum Beispiel durch einen Softwarefehler nicht mehr verfügbar sind, automatisch nach einiger Zeit aus dem Netz verschwinden. Dies erhöht zum einen die Robustheit, aber auch die Leistung des Netzes, da abwesende Dienste von anderen Netzwerkteilnehmern nicht mehr berücksichtigt werden müssen. Folgerichtig stellt sich dadurch die Frage, wie groß der optimale Lease sein sollte, was generell stark anwendungsabhängig ist.

Standardmäßig werden Leases mit einer Länge von 3600 Sekunden ausgestellt, aber es lassen sich auch eigene Zeitwerte definieren. Kurze Leases sorgen dafür, dass das Netz schneller auf Ausfälle reagieren kann, erhöhen aber im Gegenzug die Netzwerklast, da ständig die Registrierungen verlängert werden müssen. Insofern sind solche Lease-Längen vor allem dann sinnvoll, wenn das Netz ständigen Änderungen unterlegen ist.

Beispielsweise ist für Umgebungen, in denen mobile Geräte verwendet werden, eine kürzere Leasedauer sinnvoll.

Lange Leases verringern dagegen die Netzwerklast, erhöhen aber im gleichen Maße die Trägheit des Netzes, da Dienstausfälle erst recht spät bemerkt werden. Sie bieten sich daher vor allem für statische Netze an, die keinen großen Änderungen unterliegen.

2.3.4 Weitere Besonderheiten von Jini

Abgesehen vom Leasemechanismus bietet Jini aber noch weitere Besonderheiten. So ist es auf Basis von Policydateien in der Lage, Richtlinien für die Nutzer der Dienste zu definieren und die Nutzbarkeit von Applikationen einzuschränken. Soll die Sicherheit noch weiter verbessert werden, kann als Übertragungskanal eine verschlüsselte Verbindung auf Basis des SSL/TLS-Standards aufgebaut werden. Dieser Schritt erhöht aber den Implementierungsaufwand und steigert zusätzlich die nötige Rechenleistung, da jeder Teilnehmer an diesem Netzwerk möglichst schnell ver- und entschlüsseln muss, was für eingebettete Systeme eine Limitierung darstellt.

Für Systeme mit begrenztem Arbeitsspeicher, die aber trotzdem mehrere Dienste anbieten müssen, wurde das Konzept der *Activation*[5] [New00b] geschaffen. Es erlaubt Computersystemen auf ihnen laufende Serverobjekte temporär abzuschalten und auf die Festplatte auszulagern. Sobald diese Objekte benötigt werden, werden diese von der Festplatte in den Arbeitsspeicher geladen und stehen somit wieder als Dienst zur Verfügung.

Ferner bietet Jini auch ein einfaches Konzept, um Transaktionen[6] [New00c, OW01] zu realisieren. Dieses wird vor allem vom *JavaSpaces-Dienst* genutzt, welcher verteilt gespeicherte Objekte im Netzwerk ermöglicht. Transaktionen können auch von eigenen Diensten genutzt werden, wobei aber zu beachten ist, dass diese nicht die Leistungsfähigkeit von Transaktionen im Datenbankumfeld erreichen.

3 Beispiel für eine Dienstimplementierung mittels Jini

Um ein Verständnis für die Entwicklung verteilter Systeme mit Hilfe von Jini zu vermitteln, wird im Folgenden ein kleines Anwendungsbeispiel entwickelt, das stufenweise erweitert wird. Neben diesem Aspekt steht zudem das Erkennen von potenziellen Stärken und Schwächen von Jini im Vordergrund dieses Abschnitts.

Bestandteil des ersten Entwicklungsschritts ist ein eingebettetes System, welches über ein Farbdisplay mit 2,1" Bildschirmdiagonale verfügt. Die restlichen Ausstattungsmerkmale sind embedded-typisch minimalistisch ausgelegt. So verfügt das System über einen mit 14,7456 MHz getakteten AT90can128 Prozessor, der im Embedded Bereich eine hohe Verbreitung besitzt. Unterstützt wird er durch 4 KB SRAM, 128 KB ROM und zwei *Universal Synchronous/Asynchronous Receiver Transmitter* (USARTs), die zur Kommunikation mit anderen Systemen gedacht sind. Eine dieser USARTs kann als RS232-Schnittstelle von außen angesprochen werden.

[5] Activation, http://www.tigernt.com/onlineDoc/jini_tutorial/Activation.html, letzter Zugriff: 10.12.2010

[6] Transactions, http://www.tigernt.com/onlineDoc/jini_tutorial/Transaction.html, letzter Zugriff: 10.12.2010

Da dieses System durch seine Rechenleistung limitiert ist, aber andererseits ein Farbdisplay besitzt, wurde der Fokus der Applikation auf das Display gelegt. Die entwickelte Software baut auf dem selbst entwickelten *Display-Service* auf. Dieser Dienst soll es erlauben am Client produzierte Informationen im gesamten Netzwerk auf verschiedenen Displays darzustellen. Ein Display-Service erhält dabei Bildinformationen vom Client und stellt diese entsprechend seiner eigenen Dienstimplementierung auf einem Display dar. Auf diese Weise kann zum Beispiel eine Grafik von einem System auf viele verschiedene Geräte gestreamt werden. Die Abbildung 2 veranschaulicht den grundsätzlichen Aufbau.

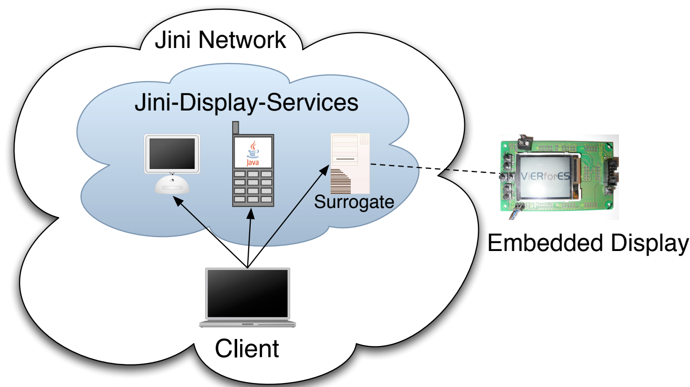

Abb. 2 Schematische Darstellung des Display-Service-Beispiels

3.1 Dienstspezifikation

Der Client kennt anfangs nur das Interface des Display-Services, denn die eigentlichen Implementierungen dieses Dienstes sind ihm noch unbekannt. Im Gegenzug sichern die Implementierungen des Dienstes auf den verschiedenen Geräten zu, sämtliche Methoden des Interfaces zu beinhalten.

In Listing 1 wird die Definition des Display-Services durch das Java-Interface Display dargestellt. Eine notwendige Voraussetzung zur korrekten Spezifikation des Dienstinterfaces ist die Spezialisierung über die Java-Schnittstellen Remote und Serializable. Das Remote-Interface enthält keinerlei Funktionalität und fungiert somit lediglich als so genanntes *Markerinterface*. Wie der Name angibt, dient es zur Markierung von Klassen, um zu garantieren, dass der Dienst die allgemeinen Bedingungen, die an Jini-Dienste gestellt werden, erfüllt. Dazu gehört die Implementierung des Serializable-Interfaces, welches wiederum nur ein Markerinterface und folglich leer ist. Wird in einer Klasse die Schnittstelle Serializable implementiert, so wird eine Zusicherung gegeben, dass Objekte dieser Klasse serialisiert und wieder deserialisiert werden können. Dies ist eine wichtige Voraussetzung, denn Jini überträgt, wie bereits in Abschnitt 2.3 erwähnt wurde, Stellvertreterobjekte der Dienstklassen über das Netzwerk. Folglich müssen die-

```java
import java.io.Serializable;
import java.rmi.Remote;
import java.rmi.RemoteException;

/**
 * Das Interface, dass den Display-Service beschreibt.
 */
public interface Display extends Remote, Serializable {

    /**
     * Diese Methode liefert die Auflösung der Zeichenfläche zurück
     * @return die Aufloesung der Anzeigeflaeche
     * @throws RemoteException
     */
    public int[] getResolution() throws RemoteException;

    /**
     * Zeichnet einen Punkt auf der Zeichenflaeche
     * @param x, die x Koordinate des zu zeichnenden Punktes
     * @param y, die y Koordinate des zu zeichnenden Punktes
     * @param color, die Farbe des zu zeichnenden Punktes
     * @return ein eventueller Rueckgabewert des Services um
     *     Aenderungen am Client auszuloesen
     * @throws RemoteException
     */
    public int putPixel(int x, int y, int color) throws
        RemoteException;

    /**
     * Zeichnet ein Bild auf der Zeichenflaeche
     * @param img, das zu zeichnende Bild
     * @param width, die Breite des Bildes
     * @param height, die Hoehe des Bildes
     * @throws RemoteException
     */
    public void putImage(int[] img, int width, int height) throws
        RemoteException;
}
```

Listing 1 Definition des Display-Service

se Objekte in einen Datenstrom geschrieben und aus diesem auch wieder ausgelesen werden können, was in Java mittels Serialisierung und Deserialisierung erzielt wird.

Sämtliche Methoden, die von anderen Netzwerkteilnehmern aufgerufen werden können, müssen die Deklaration einer throws-Klausel für die RemoteException besitzen. Diese Exception wird vor allem dann ausgelöst, wenn Jini auf Probleme bei der Übertragung stößt.

Die im Display-Interface definierte Methode getResolution() ist hauptsächlich informeller Natur und wird in der Beispielimplementierung kaum genutzt. Sie liefert die Auflösung der Darstellungsfläche des Displays zurück. Da die konkrete Skalierung der Bildinformationen aber jedem Dienst selbst überlassen ist und der Client keine Konvertierungen vornimmt, ist sie nicht zwingend notwendig.

putPixel() und putImage() stellen dagegen essentielle Methoden dar. Da Display-Services immer inkrementell über Änderungen in der Darstellung informiert werden, liefert putPixel() die Koordinate und die Farbe des Punktes, der der Grafik hinzu-

gefügt wurde. Dabei ist das Protokoll so gestaltet, dass es besonders effizient Linienzüge übertragen kann, da diese den typischen Anwendungsfall im Szenario darstellen. Anstatt bei einer roten Strecke zwischen Punkt P1 (50,100) zu Punkt P2 (100,100) jeden Punkt einzeln zu übertragen, wird lediglich der Startpunkt und anschließend der Endpunkt an die Dienste gesendet. Durch diese Daten kann, unter korrekter Interpretation des Protokolls, ein Display-Service nun einfach eine Linie zeichnen. Der Punkt (-1, -1) kennzeichnet einen ungültigen Punkt im Grafikkoordinatensystem und veranlasst dadurch einen internen Reset, um zu signalisieren, dass ein neuer Linienzug, der nicht mit dem Vorherigen in Verbindung steht, gezeichnet wird.

3.2 Client zur Dienstnutzung

Ein Client ist mit Hilfe des beschriebenen Display-Interfaces in der Lage, Geräte anzusprechen, die diesen Dienst implementieren. Da die Realisierung der Grafikfunktionen im Client Standard-Java nutzt, soll an dieser Stelle lediglich eine Darstellung der Nutzung von Jini erfolgen.

In Listing 2 wird die Suche nach Display-Services gezeigt. Da die Implementierung der Suche nach Lookup-Services oder anderen Jini-Diensten aufwändig ist und zudem immer in ähnlicher Form abläuft, wurden von Sun Helferklassen geschaffen, die diese Aufgabe übernehmen. Diese Helferklassen sind die so genannten *Manager*. So kümmert sich der LookupDiscoveryManager um das Finden von Lookup-Services, in genau der Art und Weise wie es im Abschnitt 2.3.1 beschrieben wurde. Eine Instanz dieser Klasse kann an einen ServiceDiscoveryManager übergeben werden, der unter Nutzung der gefundenen Lookup-Services nach spezifischen Jini-Diensten sucht.

Weiterhin wesentlich ist die Klasse ServiceTemplate. Mit Hilfe einer Instanz dieser Klasse wird eine Vorlage erzeugt werden, die den gewünschten Dienst charakterisiert. Dazu genügt es das Display-Interface zu übergeben, um den Display-Service zu identifizieren. Mit dieser ServiceTemplate-Instanz wendet sich der ServiceDiscoveryManager an die gefundenen Lookup-Services, um Jini-Server zu finden, die den Display-Service implementieren. Das Ergebnis dieser Anfrage wird im Anschluss in Form eines ServiceItem-Arrays zurückgegeben, bei dem jedes Element ein Stellvertreterobjekt der realen Display-Services darstellt. Um die Methoden des Display-Interface nun nutzen zu können, müssen die Stellvertreterobjekte auf das Display-Interface gecasted werden. Anschließend können der normalen Java-Syntax folgend, die Methoden des Display-Service auf den Stellvertreterobjekten aufgerufen werden, als würde es sich um lokale Objekte handeln.

In der vorliegenden Implementierung wird die Suche nach Diensten in einem Thread ausgelagert. Der Aufruf der Methode lookup() kann andernfalls zum Stillstand der Applikation führen, da diese Methode blockierend ist. Dank der Verwendung von Threads wird somit eine bessere Leistung erzielt. Zudem wird die Dienstsuche zyklisch in festen Zeitintervallen wiederholt, was sich durch Threads einfach realisieren lässt.

3.3 Dienstimplementierung

Der letzte Bestandteil des Jini-Netzwerks ist die Implementierung eines Display-Services. Dazu muss eine Klasse das Display-Interface und somit die dort aufgelisteten Methoden implementieren. Dabei gilt es eine Besonderheit von Jini zu beachten. Durch die

```
...
//Verwaltet die Suche nach Lookup-Services
final LookupDiscoveryManager manager =
    new LookupDiscoveryManager(LookupDiscoveryManager.ALL_GROUPS,
        null, null);

//Verwaltet die regelmessige Suche nach Lookup-Services
final ServiceDiscoveryManager mgr =
    new ServiceDiscoveryManager(manager, new LeaseRenewalManager());

Runnable datasourcesSearchRunnable = new Runnable() {
    public void run() {
        //Erzeugt ein neues Template fuer die Suche nach Display-
            Services
        ServiceTemplate temp =
            new ServiceTemplate(null, new Class[]{Display.class},
                null);

        //Speichert die gefundenen Display-Services
        java.util.List<Display> serviceList = null;

        //Thread laeuft dauerhaft
        while (true) {
            //warte solange bis Display-Services gefunden wurden
            while (manager.getRegistrars().length == 0) {
                try {
                    //schlafe 1000 ms
                    Thread.currentThread().sleep(1000);
                } catch (InterruptedException e) {
                    System.out.println("Schlaf unterbrochen");
                }
            }

            //Frage bei allen Lookup-Services nach dem Display-
                Service an
            ServiceItem[] items = new ServiceItem[0];
            try {
                items = mgr.lookup(temp, 1, Integer.MAX_VALUE, null
                    , WAITFOR);
            } catch (Exception e) {
                e.printStackTrace();
            }

            //Erzeuge temporaere Liste
            serviceList = new ArrayList<Display>();

            //Fuege alle gefundenen Services der temporaeren Liste
                hinzu
            for (ServiceItem item : items) {
                serviceList.add((Display) item.service);
            }

            ...
```

Listing 2 Suche mit Jini nach Display-Services

Implementierung des Display-Interfaces sichern Entwickler zu, dass ein korrekter Jini-Dienst implementiert wurde. Für diese Zusicherung muss allerdings gewährleistet sein, dass die Objekte serialisierbar sind.

Bei der Serialisierung eines Objekts werden alle Attribute in einen Datenstrom geschrieben, aus dem das Objekt später wiederhergestellt werden kann. Dabei wird auch rekursiv jedes Attribut dieses Objekts serialisiert, was aber nicht in allen Fällen möglich ist. Während Strings und gewöhnliche Datenstrukturen unproblematisch sind, sind systemnahe Objekte generell nicht serialisierbar. Als systemnah sind dabei Objekte anzusehen, die eine definierte Verbindung zum darunter liegenden Computersystem haben. Sämtliche Hardwareschnittstellen wie Netzwerk- oder serielle Verbindungen, Threads, Dateien und sogar Komponenten der grafischen Oberfläche bauen auf betriebssysteminternen Funktionen auf und sind daher nicht serialisierbar. Da auf solche Objekte aber oft nicht verzichtet werden kann, wurde das Attribut *transient* geschaffen. Attribute, die mit transient versehen sind, werden als nicht serialisierbar deklariert. Listing 3 zeigt dies am Beispiel des Objekts DrawingCanvas, dass die Zeichenfläche eines Display-Service darstellt.

```
private transient DrawingCanvas canvas = new DrawingCanvas();
```

Listing 3 Verhindern der Serialisierung eines Attributs

Dieses Schlüsselwort stellt eine Anweisung an die virtuelle Maschine dar, dass das betreffende Attribut nicht serialisiert wird. Es verbleibt somit auf dem Server und wird nicht Bestandteil des Stellvertreterobjekts. Da das Stellvertreterobjekt aber lediglich als Container dient, um Methodenaufrufe und Parameter über das Netzwerk übertragen zu können, ist dies in diesem Fall nicht problematisch. Wichtig ist nur, dass sich Parameter, Rückgabewerte und Attribute, auf die vom Client aus zugriffen wird, serialisieren lassen müssen. Demzufolge ist es ausgeschlossen, eine Methode in einem Jini-Dienst zu definieren, die zum Beispiel ein Socketobjekt zurückliefert. Ist der Rückgabewert dagegen ein Array mit Integer-Werten, die innerhalb der Methode über einen Socket gelesen werden, ist dies problemlos möglich.

Unter Beachtung dieser Besonderheiten ist die Implementierung eines Jini-Dienstes relativ einfach. Allerdings muss der neue Jini-Dienst noch gestartet werden, was in Listing 4 dargestellt sind.

Die Methode main() dient zum Start des Servers. Dabei ist es wichtig, zuerst einen RMISecurityManager zu installieren. Dieser erlaubt es, dass sich der Dienst bei einem Lookup-Service registrieren darf. Anschließend wird der Server initialisiert, in dem das konkrete Stellvertreterobjekt erzeugt und beim Lookup-Service registriert wird.

Da dieser Dienst dauerhaft zur Verfügung stehen soll, darf der Server nicht terminieren. Werden keine Vorkehrungen diesbezüglich getroffen, so würde sich nach dem Initialisieren der Server wieder beenden. Eine naheliegende Lösung ist es daher, ihn in einen so genannten *Spin-Lock* zu schicken, was zum Beispiel mit einer Schleife möglich ist, deren Abbruchbedingung nie erreicht wird. Diese auch als *busy-waiting* bezeichnete Technik verbraucht dabei allerdings viel Ressourcen, was nicht zielführend ist. Sun schlägt dagegen in seiner Jini-Dokumentation vor, einen Deadlock einzusetzen. Normalerweise ist ein Deadlocks eine Erscheinung in der Programmierung, die die gegenseitige Blockade von mehreren Threads beschreibt. Blockieren Threads, so können sie nicht fortgeführt werden. Erschwerend kommt hinzu, dass diese Deadlocks oft zufällig auftreten und daher schwer zu finden sind. Allerdings wird der Deadlock an dieser Stelle bewusst eingesetzt, da er garantiert, dass das Programm nicht beendet wird und gleich-

```java
/**
* Hauptmethode startet den Display-Service
* @param args, die Argumente werden nicht gebraucht
*/
public static void main(String[] args) {
    //Setze einen SecurityManager, benoetigt fuer Jini
    System.setSecurityManager(new java.rmi.RMISecurityManager());

    //Initialisiere den Server
    new Server().init();

    //Erzeuge einen Deadlock --> Empfehlung von Sun
    Object keepAlive = new Object();
    synchronized(keepAlive) {
        try {
            //Warte auf einen Notify an den Monitor,
            //kommt aber niemals --> Deadlock
            keepAlive.wait();
        } catch(InterruptedException e) {
            // Tue nichts
        }
    }
}
```

Listing 4 Starten eines Jini-Servers

zeitig nahezu keine Systemressourcen verbraucht. Der Server läuft somit kontinuierlich durch bis das Programm vom Betriebssystem beendet wird.

3.4 Surrogates

Teil des Beispiels ist auch ein eingebettetes System, dass nicht in der Lage ist, Java-Programme auszuführen. Somit kann dieses Systeme an keinem Jini-Netzwerk direkt teilhaben, was den Anwendungsbereich einschränkt. Allerdings bietet Jini mit den Surrogates eine Möglichkeit diese Limitierung zu umgehen. Ein Surrogate ist ein Stellvertreter, der für ein System, dass an einem Jini-Netzwerk nicht direkt teilhaben kann, als Vermittler fungiert. Dieses Surrogate, welches in Java entwickelt wird, bindet sich in das Jini-Netzwerk als Client und gleichzeitig als Server ein. Über eine andere Datenverbindung, seien es rein softwareseitige oder auch hardwarebasierte Schnittstellen, baut dieses Surrogate dann eine Verbindung zum nicht Jini-fähigen Gerät auf. Wie in Abbildung 2 dargestellt, werden Nachrichten aus dem Jini-Netzwerk durch das Surrogate verarbeitet und gegebenenfalls an das nicht Jini-fähige Gerät weitergeleitet. Eine Antwort von diesem wird über das Surrogate wieder ins Jini-Netzwerk übertragen.

Jini bietet für die Surrogate-Entwicklung eine eigene API. Diese unterstützt Entwickler vor allem bei der Realisierung aufwändigerer Jini-Funktionen, wie etwa bei der Nutzung des Activation-Konzepts. Generell besteht aber keine Pflicht, die Surrogate API zur Implementierung eines Stellvertreters zu nutzen. Die Implementierung der Datenverbindung zum Gerät ist dagegen freigestellt, was insbesondere im Detail häufig Schwierigkeiten offenbart. Da Jini gewisse Anforderungen, zum Beispiel Robustheit und Geschwindigkeit, an ein verteiltes System gewährleisten will, müssen auch sämtliche Komponenten die dafür notwendigen Mechanismen unterstützen. Dies hat zur Folge,

dass auch die Verbindung zwischen Surrogate und Endgerät, sowie das Surrogate und das Endgerät selbst, entsprechend implementiert sein müssen.

Im Beispiel wurde eine Verbindung vom Surrogate zum eingebetteten System über die serielle Schnittstelle per RS232 realisiert. Um die Verbindung in einen definierten Anfangszustand zu versetzen, führen das Surrogate und das eingebettete System einen Handshake durch. Dieser Prozess wird gegebenenfalls solange wiederholt, bis eine gültige Verbindung aufgebaut werden konnte. Weiterhin sendet das Embedded System in einem definierten Zeitintervall einen Heartbeat an das Surrogate, sodass Ausfälle des einbetteten Geräts erkannt werden können. Sobald dieses Signal vom Surrogate nicht mehr erkannt wird, leitet es keine Daten mehr an das eingebettete System weiter. Dadurch soll ein Datenverlust und inkonsistenter Zustand vermieden werden. Ankommende Daten werden dann lediglich, wie auch im normalen Betrieb, in einem Cache abgelegt ohne diese an das Gerät weiterzuleiten.

Das verwendete Protokoll zwischen Surrogate und eingebettetem System ist als leichtgewichtig anzusehen. Es besteht für diese Strecke aus zwei Byte für die Farbe in 16-Bit Notation, der Angabe der x- und y-Koordinaten, die auch jeweils ein Byte benötigen, und einem Steuerbyte, um die Verarbeitung genauer kontrollieren zu können. Diese Variante nutzt interne Funktionen des Displays des eingebetteten Systems und kann somit sehr schnell Polygonzüge zeichnen ohne das alle Punkte einer Geraden übertragen werden müssen. Allerdings muss das Surrogate eine Vorverarbeitung durchführen. So müssen einerseits die Farbwerte aus einem 32-Bit Datenraum in das 16-Bit Format konvertiert werden, andererseits ist auch eine Skalierung erforderlich, da das verwendete Display eine wesentlich kleinere Auflösung als der für Desktop-Rechner gedachte Display-Client besitzt.

3.5 Erweiterungen des Beispiels

Diese Beispielapplikation wurde im Nachhinein noch um einen weiteren Dienst erweitert, um der dynamischen Struktur Rechnung zu tragen. Der neu eingeführte Dienst *Datasource* sammelt Daten von Sensoren in Form von Koordinaten- und Farbangaben. Diese Daten werden anderen Diensten oder Clients auf Anfrage zur Verfügung gestellt. Clients können diese Daten somit downloaden, was aber mit Geschwindigkeitseinbußen verbunden ist. Denn bei jeder Anfrage nach Daten, müssen diese erst durch den Datasource-Service ermittelt und anschließend zurückgesandt werden. Eine Alternative ist, den Datasource-Service die Daten kontinuierlich sammeln zu lassen und an sämtliche Dienste, die sich als Empfänger registriert haben, zu senden. Diese Variante verringert die Netzwerklast, sorgt allerdings auch dafür, dass Empfänger die Daten immer erhalten, egal ob sie diese momentan brauchen oder nicht. Der wesentlich größere Nachteil ist jedoch, dass eine Abhängigkeit zu einem anderen Dienst entstehen würde, der diese Daten empfangen kann. Um den Datasource-Service flexibel und unabhängig zu gestalten, wurde deshalb die erstgenannte Variante implementiert. Listing 5 zeigt die Definition des Datasource-Service.

Für die Implementierung des Dienstes wird ein eingebettetes System gesetzt, das einen externen, mit Logik versehenen Sensor emulieren soll. Das System verfügt über einen AVR ATMega128 Prozessor mit 8 MHz CPU-Takt, 4 KB RAM und 64 KB ROM. Besonders hervorzuheben ist die vorhandene Bluetoothschnittstelle, sowie das zur Realisierung des Bluetoothstacks benötigte Betriebssystem BTnutOS. Allerdings ist auch dieser Mikrocontroller nebst Betriebssystem nicht Java- und somit auch nicht

```java
import java.io.Serializable;
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface DatasourceService extends Remote, Serializable {

    //Gibt einen vom Sensor gelieferten Wert zurueck
    public float getValue() throws RemoteException;

    //Liefert eine Beschreibung der Datenquelle
    public String getDescription() throws RemoteException;

    //Liefert den Wertebereich
    public float[] getBounds() throws RemoteException;

    //Liefert die Farbe
    public int getColor() throws RemoteException;

}
```

Listing 5 Definition des Datasource-Services

direkt Jini-fähig, sodass wiederum ein Surrogate eingesetzt werden muss. Das Surrogate stellt dabei über die Bluetoothschnittstelle eine Verbindung zum Mikrocontroller her und erlaubt zudem die Konfiguration des Dienstes. So lässt sich der Wertebereich, die scheinbar gemessene Größe, die Farbe als auch ein Beschreibungstext definieren. Ein Problem ist dabei vor allem die Einschränkung auf maximal eine gleichzeitig Bluetoothverbindung, was durch die verwendeten Bluetoothstacks zu begründen ist. Deshalb ist das Surrogate und die Implementierung auf dem eingebetteten System so gestaltet, dass ein Mikrocontroller mehrere virtuelle Sensoren emulieren kann. Dazu wird jeder virtuelle Sensor mit einer ID versehen, sodass über diese eine eindeutige Referenzierung und das gezielte Abfragen eines Wertes möglich ist. Das eingebettete System ist allerdings nur so leistungsfähig, dass es maximal drei Sensoren fehlerfrei emulieren kann. In Umgebungen mit vielen, gleichzeitigen Anfragen versagt ansonsten der Dienst, wobei sich die Ursache für dieses Problem noch nicht genau bestimmen ließ.

Durch die Entwicklung des neuen Dienstes entsteht zudem die Anforderung, dieses mit dem Display-Service zu kombinieren. Der Client soll dazu periodisch die Daten der Sensoren abfragen, sie plotten und anschließend an sämtliche Display-Services senden. Problematisch ist in diesem Punkt aber die Einbindung des eingebetteten Systems, welches den Display-Service implementiert. Das ursprüngliche Protokoll zur Datenübertragung ist sehr hardwarenah gestaltet und auf den Zweck optimiert, Linienzüge darzustellen. Diese Bildinformationen zeichnen sich vor allem durch zusammenhängende Linien aus, sodass lediglich Start- und Endpunkte übertragen werden müssen und das System eine Verbindungslinie zwischen diesen zeichnen kann.

Für das Plotten von Sensorwerten ist dies ungeeignet, da lediglich Punkte in Abhängigkeit von der Zeit übertragen werden, zwischen denen keine Verbindungslinie gezogen werden darf. Folglich muss ständig ein interner Reset für das eingebettete System ausgelöst werden, um dieses Verhalten zu unterbinden. Dies wiederum führt zu einem beträchtlich höheren Nachrichtenaufkommen, sodass viele, nahezu gleichzeitig ankommende Datenpakete nicht ausreichend schnell verarbeitet werden können. Somit ist das bestehende Protokoll des Display-Services ungeeignet, um Daten an das

eingebettete System zu übertragen. Gleichzeitig ist es aber auch nicht mögleich, den Display-Service zu ändern, da sonst sämtliche Dienste, die das Display-Service-Interface implementieren, angepasst werden müssen.

Eine Optimierung des Übertragungsprotokolls zwischen dem Surrogate und dem eingebetteten System führt zudem nicht zum gewünschten Erfolg oder ist teilweise mit einem zu hohem Implementierungsaufwand verbunden. Somit bleibt als letzte Möglichkeit zu verhindern, dass viele Nachrichten gleichzeitig bei den Display-Services ankommen. Dazu wird der Wert weiterhin vom Client geplottet, aber die zugehörigen Nachrichten mit einem gewissen Zeitversatz zueinander versandt. Dadurch ist das eingebettete System in der Lage, die Daten schnell genug zu verarbeiten, während aber die maximal mögliche Darstellungsgeschwindigkeit gesenkt wird. Für diese Beispielimplementierung ist dies aber vernachlässigbar.

Ein weiteres Anwendungsszenario stellt die Koppelung von Sensornetzwerken und virtuellen Umgebungen dar, wie es auch in [KSSS09] beschrieben ist. Eingebettete Systeme und Sensoren können dazu genutzt werden, Daten aufzunehmen und diese gegebenenfalls in geeigneter Art und Weise für die Beobachtung, Planung und Entscheidungsfindung zur Verfügung gestellt werden. Aufgrund der Vielschichtigkeit heutiger Prozesse und Produkte ist dabei jedoch auf eine für den Anwender geeignete Darstellungsumgebung zu achten. Virtuelle Realität ist hierbei ein Mittel, die Komplexität der zu steuernden Einheiten handhabbar zu gestalten. Gleichzeitig spielt die Dynamik innerhalb der Netzwerke eine wichtige Rolle. Die Koppelung von Datenersteller und Virtueller Realität mittels einer SOA ist hierbei vielversprechend.

4 Bewertung von Jini

Im Rahmen dieser Arbeit wurde Jini als Technologie zur Entwicklung verteilter Systeme erläutert, eine Beispielimplementierung vorgestellt und auf bestimmte Probleme hingewiesen. Aufbauend auf diesen Erfahrungen hat sich eine eigene Charakterisierung entwickelt, die beurteilt, wann ein Einsatz von Jini sinnvoll ist und wann dagegen nicht. Dazu wird in den folgenden Abschnitten zunächst jeweils auf die Stärken und Schwächen der Jini-Architektur eingegangen und abschließend ein Gesamteindruck formuliert.

4.1 Jinis Stärken

Die wohl größte Stärke der Jini-Technologie ist ihre Einfachheit. Unterstützt durch Javas einfache Syntax und die mächtige Bibliothek gelingt eine sehr schnelle Entwicklung, die zudem sehr robust ist. Es ist im Gegensatz zu CORBA keine andere Sprache zu erlernen und trotzdem kann partiell CORBA angebunden werden. Sämtliche Entwicklungen basieren größtenteils auf Standardprinzipien und -methoden der Java-Welt, wie etwa besondere Interfaces zum Markieren von Codeabschnitten oder die Nutzung von Serialisierung. Demzufolge ist ein leichter und schneller Einstieg in die Entwicklung Jini-basierter Applikationen möglich.

Weiterhin von Vorteil ist der hohe Funktionsreichtum, den Jini bietet. Angefangen mit einfachen Diensten lassen sich hochkomplexe Komponenten mit Transaktionen, Rechtevergabe, verteilt gespeicherten Objekten, anderen Kommunikationskanälen oder auch partiell herunterfahrbaren Dienstbestandteilen implementieren. Besonders für die

Entwicklung großer, vernetzter Anwendungen im Desktop- oder Serverbereich, die einer gewissen Netzwerkdynamik unterliegen sollen, bietet sich Jini an. Zudem lässt es sich im beschränkten Maße auch im eingebetteten Bereich verwenden, sodass selbst nicht Java-fähige Geräte über Umwege nutzbar sind.

Ein weiterer Vorteil ist die Abstraktion der gesamten Netzwerkkommunikation. Entwickler benötigen dazu weder detailliertes Wissen über Netzwerkprotokolle noch über Sockets oder Streams. Jini kümmert sich um diese Aspekte und lässt Entwickler sich auf die eigentliche Dienstfunktionalität konzentrieren, was das Ziel eines Frameworks zur Entwicklung verteilter Architekturen sein sollte.

4.2 Jinis Schwächen

Jinis größte Stärke, seine Einfachheit bedingt durch Java, ist gleichzeitig auch seine größte Schwäche. Durch die Abhängigkeit zu Java wird für einen Jini-Dienst immer zwingend eine Java-Implementierung benötigt. Dies ist für Desktopsysteme oder Server im Allgemeinen kein Problem, da für alle gängigen Betriebssysteme eine Java Virtual Machine zur Verfügung steht. Allerdings verfolgt Jini auch den Java-Gedanken *Write once, run everywhere* und möchte somit auch andere Geräteklassen, vor allem eingebettete Geräte ansprechen. Diese sind aber, wie am Beispiel ersichtlich wurde, meist weder softwareseitig in der Lage Javas Bytecode zu interpretieren, noch verfügen sie über die nötige Rechenleistung um die verlangten Java-Methoden ausführen zu können. Es gibt zwar Java Virtual Machines für Atmel Microcontroller, doch erlauben diese meist keine Objekterzeugung, besitzen keinen Garbage Collector und unterstützen selten netzwerkbasierte Schnittstellen. Demzufolge ist zur Anbindung von eingebetteten Systemen an ein Jini-Netzwerk fast immer ein Surrogate nötig, dessen Entwicklung sehr zeitaufwändig und kompliziert sein kann.

Problematisch im Zusammenhang mit Embedded Systemen ist außerdem die fehlende Echtzeitfähigkeit von Jini. Gerade kleine eingebetteten Geräte arbeiten oft in Umgebungen, in denen Deadlines von Tasks teilweise zwingend eingehalten werden müssen. Durch die Verwendung eines Garbage Collectors, dessen Ausführung nichtdeterministisch ist, und Betriebssystemen, die auch nicht auf den Echtzeiteinsatz ausgelegt sind, können diese Bedingungen nicht garantiert eingehalten werden. Dadurch bleibt Jini vorerst ein großer Anwendungsbereich verschlossen.

Ferner benötigt Jini immer konfigurierte Netzwerkschnittstellen, die somit gültige Adressen und Routing-Informationen besitzen müssen. Eine Autokonfiguration von diesen Schnittstellen unterstützt Jini nicht, sodass die Flexibilität des Netzwerks eingeschränkt wird. Dienste oder Clients, die sich neu in das Netzwerk einbinden, müssen demzufolge erst manuell oder durch andere Netzwerkdienste wie DHCP konfiguriert werden. Andere Schnittstellen, abseits von Firewire oder TCP/IP, existieren zudem bisher in keiner Referenzimplementierung, sodass die Möglichkeiten zur Verbindung der Dienste eingeschränkt sind.

Ein weiteres Problem ist der relativ große Overhead, den Jini erzeugt. Da Objekte für den Netzwerktransport komplett serialisiert werden, müssen auch relativ große Datenmengen über das Netzwerk übertragen werden. Gerade für Wireless LANs oder langsame Verbindungen kann dieser Umstand bei besonders großen Netzwerken dazu führen, dass die Geschwindigkeit des Gesamtsystems signifikant sinkt.

4.3 Gesamteindruck

Nach Abwägung aller Stärken und Schwächen ist Jini vor allem für einfache und leistungsstarke Netzwerke interessant. Durch seine Abstraktion der Kommunikation zwischen verschiedenen Geräten ist es besonders geeignet, die Entwicklung großer Applikationen im Desktop- und Serverumfeld zu vereinfachen. Komplexe oder leistungsschwache Netzwerke aus dem eingebetteten Bereich sind dagegen zwar grundsätzlich möglich, allerdings gestaltet sich die Entwicklung zu kompliziert und schwierig, falls keine Java Virtual Machine verfügbar ist. Jini ist für ein Java-Umfeld geschaffen worden und jedes Einbinden eines nicht Java-fähigen Geräts erhöht den Entwicklungsaufwand beträchtlich, da die Netzwerkprinzipien von Jini erhalten bleiben müssen. Erschwerend für die Entwicklung im eingebetteten Bereich ist weiterhin, dass keine Echtzeitfähigkeit garantiert werden kann. Demzufolge kann Jini nicht als die umfassende Lösung betrachtet werden, für die sie vorgesehen ist, was hauptsächlich durch die mangelnde Verbreitung von Java im eingebetteten Bereich verursacht wird.

5 Alternativen zu Jini

Aufgrund der dargelegten Schwächen von Jini, zum Beispiel bei der Verwendung nicht Java-fähiger Geräte, ist eine Betrachtung potenzieller Alternativen sinnvoll. Die folgenden Beispiele stellen ausnahmslos Plattformen dar, mit denen sich verteilte Systeme entwickeln und betreiben lassen. Jedes dieser Frameworks besitzt dabei genauso wie Jini individuelle Vor- und Nachteile. Folglich kann an dieser Stelle lediglich eine Empfehlung gegeben werden, wann sich welches Werkzeug am besten eignet.

5.1 CORBA - Common Object Request Broker Architecture

Die *Common Object Request Broker Architecture*[7], kurz CORBA genannt, ist sowohl Framework als auch Laufzeitumgebung zugleich und wird von der *Object Management Group* (OMG) entwickelt [RNS04, TB01]. Es ist eines der bekanntesten Middleware-Systeme für verteilte, heterogene Systeme und zeichnet sich vor allem durch seine Plattformunabhängigkeit aus. Im Gegensatz zu Jini ist es zudem nicht an eine bestimmte Programmiersprache gebunden, sodass CORBA-Implementierungen zum Beispiel für Java oder C++ existieren. Die Unabhängigkeit zur verwendeten Sprache wird dabei durch die CORBA-eigene Sprache, die *Interface Definition Language* (IDL), erreicht, die für die Spezifikation von Diensten und Schnittstellen verwendet wird. Ein IDL-Compiler erzeugt auf Basis dieser Dienstdefinition programmiersprachenspezifischen Code für die Client- und Dienstseite, der wie normaler Programmcode der gewählten Programmiersprache aufgerufen werden kann. Ein *Object Request Broker*, kurz ORB genannt, übersetzt dazu Aufrufe an entfernte Methoden und realisiert die gesamte Netzwerkkommunikation. Dabei findet immer eine Übersetzung in IDL-Code zur Laufzeit statt, um die Verbindung von Programmen, implementiert in verschiedenen Sprachen, zu ermöglichen. Dieser Umstand hat anfänglich entscheidend dazu beigetragen, dass CORBA nicht mit der Geschwindigkeit "nativer"Varianten vergleichbar war.

[7] CORBA FAQ, http://www.omg.org/gettingstarted/corbafaq.htm, letzter Zugriff: 10.12.2010

CORBA eignet sich durch seine Plattformunabhängigkeit für heterogene Umgebungen, wie sie in dynamischen Netzwerken oft zu finden sind. Grundsätzlich bietet es dadurch die ideale Voraussetzung zur allumfassenden Vernetzung vieler Geräte, leidet aber auch an einer fehlenden Implementierung für eingebettete Systeme. Dabei dürfte sich die Entwicklung im Verhältnis zu Jini zudem aufwändiger gestalten, da das Erlernen der IDL zur Dienstspezifikation nötig ist. Problematisch an der Verwendung der IDL im Vergleich zu Jini ist außerdem, dass sich die Spezifikation größerer Klassenstrukturen aufwändiger gestaltet, da diese in der IDL separat erfolgen muss. Bei Jini ist diese Spezifikation durch die Definition des Dienst-Interfaces dagegen automatisch vorhanden und unterliegt, abgesehen von systemnahen Objekten, die nicht übertragen werden können, keiner großen Einschränkung. Ein weiterer Nachteil, den sowohl CORBA als auch Jini besitzt, ist die Tatsache, dass eine bereits konfigurierte Verbindung vorausgesetzt wird. Für ein dynamisches Netzwerk ist daher die Nutzung von Techniken wie DHCP notwendig. Dies erhöht den Installationsaufwand zur Schaffung einer Netzwerkinfrastruktur wesentlich.

5.2 UPnP - Universal Plug and Play

Das ursprünglich von Microsoft entwickelte *Universal Plug and Play* (UPnP), dass sich mittlerweile in der Standardisierung durch das UPnP-Forum[8] befindet, ist kein klassisches Framework, wie es Jini oder CORBA darstellen. Vielmehr steckt hinter UPnP [JW03] eine Sammlung und Kombination bestehender IP-Technologien in einem Stack. Sämtliche physischen Schnittstellen, die eine IP-Kommunikation unterstützen, können durch diesen Stack für UPnP genutzt werden. Eine Besonderheit ist, dass IP-fähige Schnittstellen automatisch via DHCP konfiguriert werden können und somit die manuelle Konfiguration von Schnittstellen oder das Aufsetzen entsprechender Netzwerkdienste entfällt. Die Erkennung von neuen Diensten im Netzwerk kann anschließend vor allem auf Basis der Netzwerktechnik *Zeroconf* erfolgen, aber auch die Nutzung von Multicast-Anweisungen ist möglich. Zur eigentlichen Kommunikation zwischen verschiedenen Diensten kann anschließend auf Technologien wie HTTP, XML-Nachrichten oder SOAP gesetzt werden, die auf den verwendeten TCP- beziehungsweise UDP-Protokollen basieren.

UPnP ist durch seine Autokonfiguration und der theoretischen Möglichkeit auf sämtlichen Systemen, die einen IP-Stack besitzen, laufen zu können, ein leistungsfähiges Werkzeug. Allerdings stellt es kein echtes Framework bereit, was zwar eine gewisse Flexibilität in der Wahl der Bibliotheken bedeutet, aber somit auch keine einheitliche Komplettlösung zur Entwicklung bieten kann. Der größte Nachteil ist dagegen die Fixierung auf IP-Kommunikation, sodass sämtliche Systeme, die nicht über IP-basierte Schnittstellen verfügen, außen vor bleiben. Von dieser Einschränkung sind insbesondere viele Embedded Systeme betroffen, sodass ihre Einbindung in ein dynamisches Netzwerk schwierig ist. Erfolgt die Kommunikation dagegen nur über IP-Technologie, so stellt UPnP eines der potenziell besten Verfahren dar.

[8] UPnP Forum, http://www.upnp.org/about/default.asp, letzter Zugriff: 10.12.2010

Die Web Services[9] [Cer02, CJ02, Mel08] sind der Funktionsweise von Jini ähnlich. Im Gegensatz zu Jini sind Web Services dagegen plattformunabhängig sowie unabhängig von der gewählten Programmiersprache. So lassen sich zur Entwicklung dieser Dienste alle gängigen Programmierumgebungen nutzen, sei es Java, C/C++, Python oder .NET. Die einzigen Voraussetzungen sind eine vorhandene Laufzeitumgebung sowie ein Framework zur Unterstützung der Programmierung.

Web Services basieren auf den drei wesentlichen Säulen: UDDI, WSDL und SOAP. *Universal Description, Discovery and Integration*, kurz UDDI, stellt einen Verzeichnisdienst zur Registrierung von Web Services zur Verfügung. An ihm können sich Dienste anmelden, sodass diese von Clients automatisch gefunden werden können. Dieser Dienst ist somit eng verwandt mit dem Lookup-Service von Jini, der dieselbe Funktionalität bereitstellt.

WSDL, ein Akronym für *Web Services Description Language*, dient der Definition von Diensten. Durch eine auf XML basierende Syntax können mit ihr Datentypen, Funktionen als auch verbindungsspezifische Parameter, wie das verwendete Protokoll oder genutzte Ports für einen Dienst spezifiziert werden. Somit ist es durch die Verwendung dieser Metasprache möglich, Dienste in verschiedenen Programmiersprachen und -umgebungen zu entwickeln. Diese Funktionalität, wenn auch mit dem Nachteil der fehlenden Programmiersprachenunabhängigkeit behaftet, wird in Jini durch die Interfaces bereitgestellt. Folglich gelten für die Web Services wieder die gleichen Vor- und Nachteile, wie sie schon bei CORBA im Zusammenhang mit der IDL dargestellt wurden.

Der letzte Bestandteil ist SOAP, was früher noch mit *Simple Object Access Protocol* assoziiert wurde. Heute ist diese Bedeutung als Akronym offiziell nicht mehr gegebenen und die Bezeichnung SOAP wird nun als Eigenname angesehen. SOAP wurde in der Version 1.2 durch das World Wide Web Consortium (W3C) als Empfehlung anerkannt, ist aber kein offizieller Standard. Es existieren auch hier für alle gängigen Programmierumgebungen Frameworks, sei es für Java, C/C++, Python oder .NET, sodass unabhängig von der Programmiersprache entwickelt werden kann. SOAP realisiert dabei genauso entfernte Funktionsaufrufe wie CORBA oder Jini, setzt aber auf XML zum Datenaustausch, was bedingt durch die XML-Struktur mit einem Overhead verbunden ist. Diese XML-Nachrichten enthalten Informationen über die aufzurufenden Funktionen sowie die dafür nötigen Parameter. Zur Übertragung der XML-Daten wird häufig HTTP und TCP genutzt, wodurch vor allem auch eine Kommunikation durch Firewalls möglich ist. Durch die Unabhängigkeit zum verwendeten Netzwerkprotokoll können aber zum Beispiel auch verschlüsselte Verbindungen aufgebaut werden.

Die Web Services sind Jini sehr ähnlich, besitzen aber aufgrund des genutzten XML-Nachrichtenformates einen Overhead, der für eine hohe Netzwerklast bei vielen, gleichzeitig versandten Nachrichten sorgt. Der Vorteil, durch viele Firewalls hindurch übertragen zu können, ist für LAN-basierte Anwendungsszenarien nicht von wesentlicher Natur, stellt aber auch keinen Nachteil dar. Ähnlich wie Jini können sich die verwendeten Netzwerkgeräte allerdings nicht selber konfigurieren, sodass auch hier idealerweise ein Dienst wie DHCP zur automatischen Konfiguration eingesetzt werden sollte. Dafür gestaltet sich die Entwicklung der Web Services aber allgemein sehr

[9] Web Services Architecture Working Group, http://www.w3.org/2002/ws/arch/, letzter Zugriff: 10.12.2010

flexibel, da eine nahezu freie Wahl der Programmiersprache besteht. Allerdings werden nicht netzwerktaugliche Geräte aus dem Embedded Bereich durch die IP-basierte Kommunikation ausgeschlossen. Somit können auch die Web Services keine ganzheitliche Lösung bieten.

5.4 FAMOUSO - Family of Adaptive Middleware for autonomOUs Sentient Objects

FAMOUSO[10] [Sch09, HSKN08] ist eine an der Otto-von-Guericke-Universität Magdeburg von Michael Schulze entwickelte Middleware, die vor allem für den Einsatz im Embedded Bereich prädestiniert ist. Sie basiert auf den Ideen der *COoperating SMart devICes*, kurz COSMIC, und fügt die Möglichkeit der Adaption, zum Beispiel auf eine bestimmte Plattform, hinzu.

Das Grundprinzip hinter dieser Middleware ist, eine ereignisbasierte Kommunikation aufzubauen, die dem publisher/subscriber Konzept folgt. Dabei werden Nachrichten nicht an bestimmte Empfänger gesandt, sondern per Broadcast im ganzen Netzwerk bekannt gemacht. Unter Betrachtung der verwendeten ID innerhalb einer Nachricht kann dann jeder Netzwerkteilnehmer selbst entscheiden, ob er diese Daten verwenden will oder nicht. Diese Art der Netzwerkkommunikation erhöht zwar die Last auf dem Netzwerk, allerdings sind die Nachrichten relativ klein, sodass auch viele Nachrichten gleichzeitig übertragen werden können. Ein weiterer Vorteil dieser Variante ist, dass keine Dienstsuche wie bei allen anderen vorgestellten Alternativen nötig ist. Fraglich ist dagegen aber noch, wie gut dieses Verfahren in großen Netzwerken skaliert, wie sie im Rahmen von ViERforES beispielsweise möglich sind.

FAMOUSO ist dank seiner Adaptivität auf einer großen Anzahl an verschiedenen Hardwareplattformen lauffähig, angefangen vom 8-Bit Mikrocontroller bis hin zum 32-Bit PC/Workstation-System. Als Kommunikationsmedien werden derzeit CAN, ZigBee, AWDS und UDP-MultiCast unterstützt, sodass gängige Schnittstellen aus dem Embedded Bereich genutzt werden können. Weiterhin existiert eine API die mit C/C++, Python, .NET oder Java genutzt, als auch mit Tools wie Matlab/Simulink oder Labview verwendet werden kann.

FAMOUSO ist somit vor allem dazu geeignet eingebettete Systeme ins Netzwerk einzubinden, da es einerseits typische Schnittstellen unterstützt und zum anderen auch Echtzeitfähigkeiten bietet. Die Möglichkeit solche Systeme ansprechen zu können, stellt im Vergleich zu den anderen vorgestellten Ansätzen ein Alleinstellungsmerkmal dar. Allerdings mangelt es FAMOUSO noch an der Verbreitung und einer breiteren Hardwarebasis, die sich aber durch Portierungen ergeben kann. Insofern stellt FAMOUSO eine interessante Option dar, wenn viele eingebettete Systeme in ein Netzwerk eingebunden werden sollen.

5.5 Vergleich der Ansätze

Zusammenfassend lässt sich die Entwicklung der beschriebenen Architektur und der Realisierung durch Jini mit den folgenden Punkten vergleichend darstellen. Jini ermöglicht eine unkomplizierte und schnelle Einbindung bereits entwickelter Softwareprodukte bzw. Module sofern eine Java-Unterstützung gegeben ist. Jedoch können auch

[10] FAMOUSO, http://famouso.sourceforge.net/, letzter Zugriff: 10.12.2010

nicht Java-fähige Anwendungen und Plattformen mit Hilfe der Surrogates in das Jini-Netzwerk eingebunden werden. Dies geht aber einher mit hohen Implementierungskosten. Daher ist Jini vor allem dann zu empfehlen, wenn die eingesetzten Systeme eine Java-Laufzeitumgebung bereitstellen und die Software größtenteils in Java vorliegt.

Diese Sprachbeschränkung ist bei CORBA nicht vorhanden, so dass ein wesentlich größeres Einsatzgebiet existiert. Jedoch geht dieser Vorteil zu Lasten der Performance. Ein weiterer Nachteil, der sich aus der Sprachunabhängikeit ergibt, ist das Fehlen von Implementierungen im Bereich der eingebetteten Systeme. Insgesamt betrachtet kann CORBA im Vergleich zu Jini ein breiteres Spektrum von Softwaresystemen unterstützen, was jedoch mit wesentlich mehr Aufwand und eventuellen Geschwindigkeitsnachteilen verbunden ist.

UPnP nutzt für die Kommunikation lediglich das Internet Protokoll, das für die Domäne der eingebetteten Systeme zu restriktiv ist, da diese häufig keine physische Netzwerkschnittstelle besitzen. Des Weiteren stellt UPnP kein Framework bereit, sodass ein wesentlich höherer Aufwand in der Entwicklung im Vergleich zu Jini nötig ist. Jedoch kann die Implementierung in einer beliebigen Programmiersprache erfolgen, wodurch sich die Einsatzmöglichkeiten im Vergleich zu Jini erweitern.

Web Services sind zwar vergleichbar zu Jini, jedoch führt die auf XML-basierende Datenübertragung der Web Services zu einem signifikanten Overhead. Dies ist insbesondere für resourcenbeschränkte, eingebettete Systeme ein großer Nachteil. Ein weiterer Nachteil der Web Services ist, dass ihre Entwicklung im Vergleich zu Jini relativ aufwändig ist. Existieren zudem viele heterogene Systemplattformen im Netzwerk, so muss für jede Plattform eine Anbindung an die Web Services neu programmiert werden. Somit sinkt die Wiederverwendbarkeit von Quellcode, wodurch der Aufwand zur Implementierung eines verteilten System steigt.

Die Datenübertragung unter der FAMOUSO Middleware basiert auf dem Broadcast-Prinzip. Sofern die Nachrichten klein sind, bietet sich dadurch eine effiziente Kommunikationsstruktur. Jedoch kann das Broadcast-Prinzip sowohl bei einem hohen Nachrichtenaufkommen als auch bei einer hohen Anzahl von Teilnehmern nachteilig sein. Ein Vorteil von FAMOUSO ist die große Anzahl unterstützter Sprachen, so dass der Einsatz im Bereich eingebetteter Systeme einfach ist. Sollen jedoch auch viele Systeme über das Netzwerk mit hohen Transfervolumen eingebunden werden, so bietet sich eher die Nutzung von Jini an.

6 Fazit und Ausblick

In dieser Arbeit wurde die Entwicklung einer verteilten Applikation auf Basis der Jini-Technologie vorgestellt. Jini eignet sich zur Realisierung einer serviceorientierten Architektur und erlaubt die schnelle Entwicklung von Diensten. Dabei versucht Jini eine hohe Flexibilität und Robustheit zu erreichen, ohne eine komplizierte Entwicklung zu forcieren. Dies konnte für das gewählte Beispiel größtenteils bestätigt werden.

Dabei zeigte sich vor allem im Bereich der eingebetteten Systeme, der einen Anteil von 98% aller Computersysteme ausmacht [Ten00, Tur03], dass Jini in diesem Fall durch Java limitiert wird. Viele eingebettete Systeme sind nicht in der Lage, Java Bytecode auszuführen, sodass ihre Anbindung ans Netzwerk nur umständlich realisierbar ist. Die im letzten Abschnitt vorgestellten Alternativen unterliegen zum Teil ähnlichen Problemen, wobei FAMOUSO einen Ausweg darstellen könnte. Allerdings besitzt jede

der genannten Alternativen individuelle Stärken und Schwächen, die für verschiedene Anwendungsbereiche evaluiert werden müssen. Ein kompromissloses Verfahren zur Realisierung einer serviceorientierten Architektur existiert somit nicht. Es verbleibt als eine zukünftige Forschungsaufgabe die Evaluation weiterer Frameworks für verteilte Systeme. Interessant ist zudem die Kombination verschiedener Ansätze, um zu evaluieren, ob dadurch partielle Schwächen der Frameworks ausgeglichen werden können. So wäre es denkbar, Jini nur im Desktop- und Serverumfeld einzusetzen, aber zusätzlich ein Surrogate zur Verbindung mit FAMOUSO bereitzustellen. Auf diese Weise könnten eingebettete Systeme leicht mit Desktop- oder Serversystemen vernetzt werden.

Danksagung

Diese Arbeit wurde im Rahmen des Projektes „Virtuelle und Erweiterte Realität für höchste Sicherheit und Zuverlässigkeit für Eingebettete Systeme" (ViERforES)[11] realisiert. Die Arbeiten im ViERforES-Projekt werden durch das Bundesministerium für Bildung und Forschung (BMBF) gefördert (Nr. 01IM08003C).

Literatur

[ASW+99] Ken Arnold, Robert W. Scheifler, Jim Waldo, Ann Wollrath, Robert Scheifler, and Bryan O'Sullivan. *The Jini(TM) Specification*, chapter The Jini Architecture Specification, pages 61–81. Addison-Wesley Pub (Sd), 1st edition, June 1999.

[Cer02] Ethan Cerami. *Web Services Essentials.* O'Reilly Media, Inc., 2002.

[CJ02] David A. Chappell and Tyler Jewell. *Java Web Services.* O'Reilly Media, Inc., 2002.

[Gro01] William Grosso. *Java RMI.* O'Reilly Media, October 2001.

[HSKN08] André Herms, Michael Schulze, Jörg Kaiser, and Edgar Nett. Exploiting Publish/Subscribe Communication in Wireless Mesh Networks for Industrial Scenarios. In *Proceedings of 13th IEEE International Conference on Emerging Technologies and Factory Automation (EFTA)*, pages 648–655. IEEE, 2008.

[JW03] Michael Jeronimo and Jack Weast. *UPnP Design by Example: A Software Developer's Guide to Universal Plug and Play.* Intel Press, May 2003.

[KSSS09] Veit Köppen, Norbert Siegmund, Michael Soffner, and Gunter Saake. An Architecture for Interoperability of Embedded Systems and Virtual Reality. *IETE Technical Review*, 26(5):350–356, September-October 2009.

[Kum02] S. Ilango Kumaran. *Jini Technology: An Overview*, chapter 7. Jini: A Service Network, pages 205–227. Prentice Hall PTR, 2002.

[Mel08] Ingo Melzer. *Service-orientierte Architekturen mit Web Services: Konzepte - Standards - Praxis.* Spektrum Akademischer Verlag, 3rd edition, October 2008.

[New00a] Jan Newmarch. *A Programmer's Guide to Jini Technology.* Apress, 1st edition, November 2000.

[New00b] Jan Newmarch. *A Programmer's Guide to Jini Technology*, chapter 20: Activations, pages 393–431. Apress, 1st edition, November 2000.

[New00c] Jan Newmarch. *A Programmer's Guide to Jini Technology*, chapter 16: Transactions, pages 271–294. Apress, 1st edition, November 2000.

[New06a] Jan Newmarch. *Foundations of Jini 2 Programming.* Apress, 1st edition, October 2006.

[New06b] Jan Newmarch. *Foundations of Jini 2 Programming*, chapter 1: Overview of Jini, pages 1–20. Apress, 1st edition, October 2006.

[New06c] Jan Newmarch. *Foundations of Jini 2 Programming*, chapter 10: Jini Extensible Remote Invocation, pages 117–126. Apress, 1st edition, October 2006.

[11] ViERforES, http://vierfores.de, letzter Zugriff: 10.12.2010

[OW01] Scott Oaks and Henry Wong. *Jini in a Nutshell*, chapter 9: Transaktionen, pages 157–169. O'Reilly, 2001.

[RNS04] T. Ritter, B. Neubauer, and F. Stoinski. *CORBA Komponenten. Effektives Software-Design und Programmierung*. Springer, Berlin, 1st edition, June 2004.

[Sch09] Michael Schulze. FAMOUSO - Eine adaptierbare Publish/Subscribe Middleware für ressourcenbeschränkte Systeme. *Electronic Communications of the European Association of Software Science and Technology (ECEASST)*, 17, 2009.

[SPS+09] Norbert Siegmund, Mario Pukall, Michael Soffner, Veit Köppen, and G. Saake. Using Software Product Lines for Runtime Interoperability. In *Proceedings of Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE)*, pages 1–7. ACM, JUL 2009.

[TB01] Zahir Tari and Omran Bukhres. *Fundamentals of Distributed Object Systems: The CORBA Perspective*. John Wiley & Sons, 1st edition, April 2001.

[Ten00] David Tennenhouse. Proactive Computing. *Communications of the ACM*, 43(5):43–50, 2000.

[TS03] Andrew Stuart Tanenbaum and Maarten van Steen. *Verteilte Systeme: Grundlagen und Paradigmen*. Pearson Studium, 2003.

[Tur03] Jim Turley. *The Essential Guide to Semiconductors*. Prentice Hall Press, 2003.