Diplomarbeit

Integration nicht-Java-fähiger Dienstleister in eine Jini-Infrastruktur

Hanno Müller
Henning-Wulf-Weg 12
22529 Hamburg
kontakt@hanno.de

Juni 2001

Erstbetreuung: Prof. Dr. Winfried Lamersdorf
Zweitbetreuung: Dr. Norman Hendrich

Fachbereich Informatik
Arbeitsbereich Verteilte Systeme
Universität Hamburg
Vogt-Kölln-Straße 30
22527 Hamburg

Erklärung:

Hiermit versichere ich, diese Arbeit selbstständig und unter ausschließlicher Zuhilfenahme der in der Arbeit aufgeführten Hilfsmittel erstellt zu haben.

Hamburg, den 29. Juni 2001

Hanno Müller

Zusammenfassung

Die gegenwärtig weiter anhaltende Minituarisierung und Kostenreduktion von Hardware zeigt deutlich in eine Richtung: In naher Zukunft werden leistungsfähige Rechner als allgegenwärtige Technologie ein alltäglicher Bestandteil von bisher „dummen“ Haushalts- und Bürogeräten werden.

Die Vernetzung dieser Geräte wird neue Anforderungen an die dazu nötige Middleware-Software stellen, damit auch unbedarfte Anwender solche Geräte problemlos und ohne technische Ausbildung durchführen können. Die Industrie steht mit mehreren Vorschlägen zur Standardisierung solcher Middleware in den Startlöchern.

Jini ist ein vielversprechender Kandidat der Firma Sun, der die Interoperabilität zwischen Geräten verschiedener Hersteller sichern und dabei die Installation von Geräte-Treibern überflüssig machen will.

Bei der Evaluation des Jini-Konzepts zeigte sich allerdings ein wesentlicher Schwachpunkt: Die Entwickler setzen voraus, dass alle beteiligten Geräte eine vollständige Java Umgebung beinhalten.

Diese Arbeit stellt eine vollständige Implementation eines Dienstleisters vor, der die wesentlichen Jini-Netzwerkprotokolle beherrscht, selbst aber keine Java Virtual Machine enthält.

„Ach, lieber Sohn“, antwortete Alaeddins Mutter, „ich habe auch nicht einen einzigen Bissen Brot; du hast gestern Abend den wenigen Vorrat, der noch zu Hause war, aufgegessen. Aber gedulde dich einen Augenblick, so werde ich dir bald etwas bringen. Ich habe etwas Baumwolle gesponnen, diese will ich verkaufen, um Brot und einiges zum Mittagessen anzuschaffen.“ – „Liebe Mutter“, erwiderte Alaeddin, „hebe deine Baumwolle für ein anderes Mal auf und gib mir die Lampe, die ich gestern mitbrachte. Ich will sie verkaufen, und vielleicht löse ich so viel daraus, daß wir Frühstück und Mittagessen, und am Ende gar noch etwas für den Abend bestreiten können.“

Alaeddins Mutter holte die Lampe und sagte zu ihrem Sohne: „Da hast du sie, sie ist aber sehr schmutzig. Ich will sie ein wenig putzen, dann wird sie schon etwas mehr gelten.“ Sie nahm Wasser und feinen Sand, um sie blank zu machen, aber kaum hatte sie angefangen, die Lampe zu reiben, als augenblicklich in Gegenwart ihres Sohnes ein scheußlicher Geist von riesenhafter Gestalt vor ihr aufstand und mit einer Donnerstimme zu ihr sprach: „Was willst du? Ich bin bereit, dir zu gehorchen als dein Sklave und als Sklave aller derer, welche die Lampe in der Hand haben, sowohl ich, als die anderen Sklaven der Lampe.“

(Gustav Weil: „Alaeddin und die Wunderlampe“, aus *Tausend und eine Nacht: Arabische Erzählungen*, 1865)

Inhaltsverzeichnis

1 Einleitung **2**
- 1.1 Einführung: Der Computer wird unsichtbar . . . 2
  - 1.1.1 Die Verkleinerung schreitet voran . . . 2
  - 1.1.2 Computer dringen in neue Anwendungsbereiche und Geräteklassen vor . . . 3
  - 1.1.3 Das Dilemma des Personal Computers . . . 4
  - 1.1.4 Der Computer wird Bestandteil des Haushalts . . . 4
  - 1.1.5 Vernetzung des Haushalts . . . 6
  - 1.1.6 Auswirkungen der Entwicklung auf Bürotechnik . . . 7
  - 1.1.7 Die Suche nach neuen Marktsegmenten . . . 7
  - 1.1.8 Die Vision des allgegenwärtigen Computers . . . 8
  - 1.1.9 Warten auf die Basistechnologie . . . 9
- 1.2 Überblick über die Arbeit . . . 10

2 Verteilte Systeme und das Middleware-Konzept **11**
- 2.1 Motivation . . . 11
- 2.2 Anforderungen an den vernetzten Haushalt . . . 11
- 2.3 Haushaltsgeräte als verteiltes System . . . 14
  - 2.3.1 Klassische Herausforderungen und Lösungswege beim Einsatz offener verteilter Systeme . . . 14
- 2.4 Middleware . . . 17
  - 2.4.1 Typische Komponenten einer Middleware . . . 17
- 2.5 Ad Hoc Vernetzung . . . 21
  - 2.5.1 DHCP . . . 22
  - 2.5.2 TCP/IP Multicast Protokoll . . . 22
- 2.6 Die Java Systemumgebung als Middleware . . . 23
  - 2.6.1 Einführung in Java . . . 23
  - 2.6.2 Die Java Systemumgebung . . . 24
  - 2.6.3 Die Java Umgebung aus Middleware-Sichtweise . . . 27
  - 2.6.4 Probleme der Java Umgebung . . . 28
  - 2.6.5 Ein reduziertes Java . . . 29

3 Kommerzielle Ad-Hoc Netze im Vergleich **31**
- 3.1 Jini . . . 31
  - 3.1.1 Einführung in Jini . . . 32
  - 3.1.2 Voraussetzungen für den Einsatz . . . 32

3.1.3 Dienstbeschreibung und -attribute in Jini 33
3.1.4 Partitionierung in Namensräume 33
3.1.5 Dienstsuche und -vermittlung, Leasing 34
3.1.6 Verteilte Events, Leasing 36
3.1.7 Ad Hoc Vernetzung in Jini, Discovery Protokoll 36
3.1.8 Stellvertreter-Objekte als Treiberprogramme 36
3.2 Universal Plug and Play 37
3.2.1 Einführung in UPnP 38
3.2.2 Anforderungen für den Einsatz 39
3.2.3 Suche nach Dienstleistern 39
3.2.4 Diensttyp, -beschreibung und -Attribute 40
3.2.5 Dienstnutzung ohne Treiber 40
3.2.6 Verteilte Events in UPnP 41
3.2.7 HTML Schnittstelle zum Dienstleister 41
3.3 Bluetooth 42
3.3.1 Der Bluetooth Funk-Standard 43
3.3.2 Dienstleister-Profile 43
3.3.3 Dienstbeschreibung, -Attribute und nutzende Applikation . . . 44
3.3.4 Suche nach Dienstleistern 45
3.3.5 Dienstnutzung 45
3.4 Fazit des Vergleichs 46

4 Eine Protokoll-Erweiterung für Jini 47
4.1 Problemstellung und Lösungsweg 47
4.2 Die Jini Basisprotokolle 48
4.2.1 Überblick 48
4.2.2 Discovery: Auffinden des Jini LUS 48
4.2.3 Join: Anmelden eines Dienstleisters im LUS 55
4.2.4 Lookup: Suchen nach Dienstleistern 57
4.2.5 Nutzung des Dienstleisters via RMI-Proxy 58
4.3 Ein modifiziertes Jini Protokoll ohne Dienstleister-JVM 59
4.3.1 Serialisierung und Java RMI in den Basisprotokollen aus Sicht des Dienstleisters 59
4.3.2 Das Problem Serialisierung und Java RMI 60
4.3.3 Verzicht auf Java RMI im Dienstleister 61
4.3.4 Das Stellvertreter-Objekt als Treiber 61
4.3.5 Vorcompilierte Treiber im Gerät 61
4.3.6 Modifiziertes Discovery und Join Protokoll 63
4.3.7 Leasing und Attribut-Modifikation ohne Java RMI 67

5 Eine Implementation in Soft- und Hardware 68
5.1 Vorgehensweise 68
5.2 Implementation des Protokolls in ANSI-C 68
5.2.1 Die Programmiersprache C, das UNIX Betriebssystem und POSIX 68
5.2.2 Komponenten der C-Bibliothek für Jini (mit Threads) 70
5.2.3 Die ersten Prototypen 72

5.3 Implementation des Protokolls für eine spezielle Hardware 73
5.3.1 Suche nach einem geeigneten Gerät 73
5.3.2 Komponenten der C-Bibliothek für Jini (ohne Threads) 74
5.3.3 Jini in einer Webcam 75

6 Zusammenfassung und Ausblick **77**
6.1 Zusammenfassung 77
6.2 Ein alternativer Ansatz: Das Jini Surrogate Project 77
6.3 Ausblick 78

A Standard-Interface Lookup Service **79**

Kapitel 1

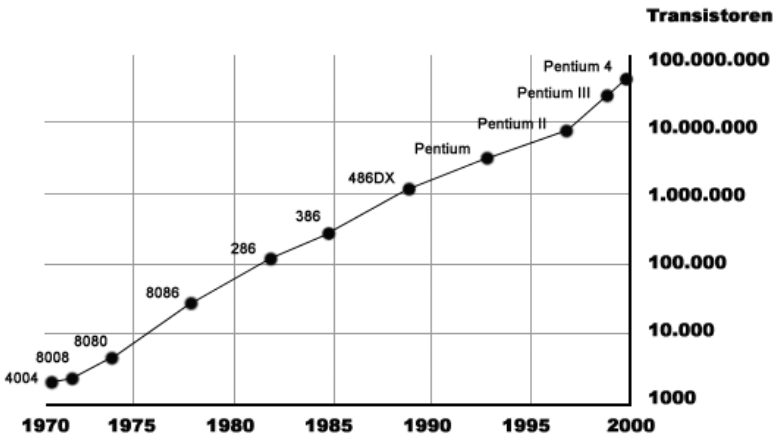
Einleitung

1.1 Einführung: Der Computer wird unsichtbar

1.1.1 Die Verkleinerung schreitet voran

Miniaturisierung ist, rückblickend betrachtet, das wohl wichtigste und auch auffälligste Merkmal in der kurzen Geschichte des Computers. Bereits die ersten industriell verwertbaren Transistoren sorgten für eine starke Verkleinerung von Geräten, die zuvor noch Röhren- oder Relaisschaltungen enthielten. Das Zusammenfassen mehrerer Transistoren zu einem einzigen Chip war der nächste Schritt zur Verkleinerung vormals großflächiger elektronischer Aufbauten.

Kurz nach der Erfindung des Mikrochips in den 60er Jahren beobachtete Gordon E. Moore einen quasi-konstanten Faktor in der ständigen Verbesserung bei der Verkleinerung der Schaltstrukturen auf einem Chip. Er stellte 1965 die These auf, dass sich in den nächsten Jahren alle 12 Monate die Anzahl Transistoren je Chip verdoppeln werde [Moo65].



Moores Gesetz, in einer Darstellung durch Moores Arbeitgeber Intel in [Int00].

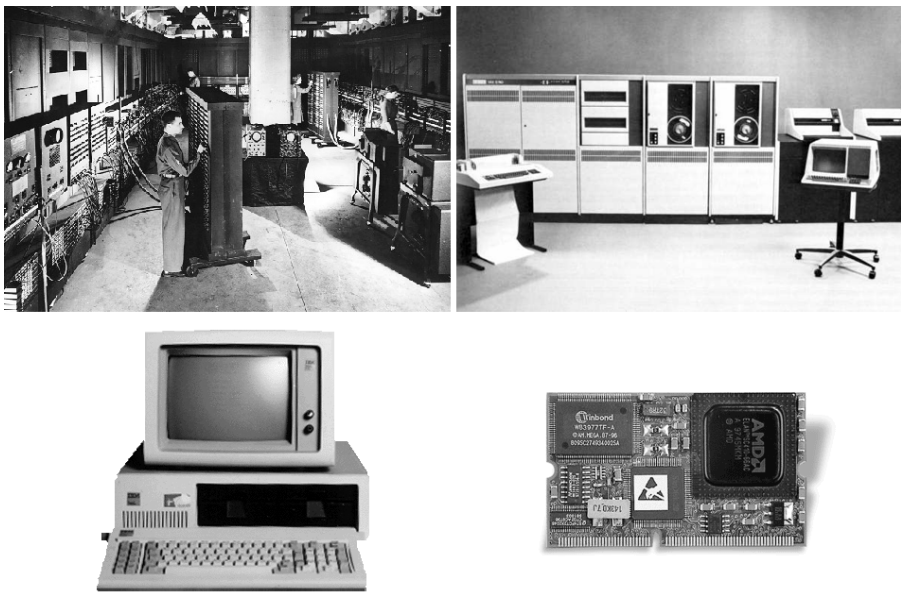
Nachdem Moore kurze Zeit danach den Zeitraum auf 18 Monate korrigierte, ist *Moores Gesetz* seitdem tatsächlich bis heute gültig geblieben. Immer wieder wurden

physikalische, technische und praktische Grenzen bei der Entwicklung vermutet und doch fand sich bisher immer wieder eine Innovation, die die Minituarisierung fortführte.

Heute werden wieder mögliche Grenzen vorhergesagt, vermutlich ist in wenigen Jahren Moores Gesetz ungültig und die Verkleinerung wird in Zukunft deutlich langsamer verlaufen. Aber noch ist es nicht so weit und wir befinden uns weiterhin in einer Periode stetigen technischen Fortschritts der Informationstechnologie-Hardware. Moore selbst sagt in [Moo00] voraus, dass dies noch mindestens 20 Jahre so bleiben wird.

Es gilt somit:

- CPUs, RAM und Speichermedien werden immer leistungsfähiger bei weniger Platzbedarf zu geringeren Kosten.



Die ständige Verkleinerung des Computers: ENIAC *(1946), einer der ersten Computer überhaupt, beanspruchte den Platz einer ganzen Halle. Die* MicroVax 700 *(1977) hatte das Format eines Büroschranks. Der* IBM Personal Computer *(1981) benötigt als Stellfläche einen Schreibtisch, aktuelle Embedded Controller wie der* JumpTec DIMM-PC *(1998) haben das Format einer Streichholzschachtel. Der DIMM-PC bietet mehr Rechenleistung als die drei zuvor genannten Computersysteme zusammengenommen.*

1.1.2 Computer dringen in neue Anwendungsbereiche und Geräteklassen vor

Trotz dieser Entwicklung hat sich der traditionelle Arbeitsplatz-Computer von außen betrachtet in den letzten zwei Jahrzehnten kaum verändert. Er ist etwa gleich groß geblieben, wird dafür aber von Generation zu Generation leistungsfähiger und enthält immer mehr Teilkomponenten.

Diese Komponenten sind so klein geworden, dass andererseits Geräte, für die dies

noch vor kurzem technisch oder ökonomisch undenkbar war, nun mit relativ leistungsfähigen Prozessoren ausgestattet werden können.

Die Rechen- und Speicherleistung eines Bürocomputers aus Mitte der 80er Jahre passt heute, nur 15 Jahre später, in eine Streichholzschachtel. Entwickler, die für ihr Produkt etwas weniger Leistung benötigen, können bereits auf vollständige Rechnersysteme aus Prozessor, Speicher und Ein-/Ausgabe im Format von wenigen Quadratzentimentern zurückgreifen.

Dank des Preisverfalls durch die Massenproduktion solcher Standardkomponenten dringt die Technologie, die vor wenigen Jahren noch einen Computer ausmachte, in ganz andere, bisher ungewohnte Bereiche vor. Moderne Fernseher, Telefone, Mikrowellenherde, Videokameras, Stereoanlagen, Waschmaschinen, Heizungssteuerungen, Armbanduhren oder Kraftfahrzeuge enthalten längst komplexe, programmierbare Schaltungen – kleine Computer.

1.1.3 Das Dilemma des Personal Computers

Mit der Verkleinerung der Hardware hat sich auch die Philosophie der Computernutzung verändert. Frühe Systeme waren riesig und teuer, Bedienzeit war kostbar und folgerichtig war lange Zeit der Computer ein zentralistisch organisiertes, knappes Gut. Ende der 70er Jahre kamen dann die ersten Personal Computer auf den Markt, jeder Angestellte und wenige Jahre später auch jeder Privatmann hatte nun seinen eigenständigen, dezentralen Rechner unter eigener Kontrolle.

So ein PC ist ein *All in one* Problemlöser und versucht, möglichst viele Anwendungen in einem Gerät zu vereinen. Ein moderner PC bietet unter anderem Textverarbeitung und Tabellenkalkulation, Kontoführung und Finanzbuchhaltung, Spiele, Grafikdesign und Photoretusche, CAD, Musik von CD oder MP3 und Videos von DVD, Radio- und TV-Empfang, Musik- und Videobearbeitung, Internet-Zugang zu Diensten wie World Wide Web oder E-Mail...

Für diese unterschiedlichen Anwendungen wurden dem Gerät immer mehr Hardware-Komponenten hinzugefügt und die Software immer mächtiger und umfangreicher. Im Ergebnis ist damit aus dem Problemlöser selbst immer mehr ein Problem geworden: Ein unbedarfter Laie wird von der Komplexität des modernen PC abgeschreckt. Immer mehr mögliche Hardware-Konflikte der Komponenten lassen den Anfänger verzweifeln; die Bedienung der für den eigentlichen Zweck überqualifizierten Software verlangt lange Lernzeiten.

1.1.4 Der Computer wird Bestandteil des Haushalts

Um diesem Dilemma des durch seine nunmehr unendlichen Möglichkeiten viel zu komplizierten Personal Computers zu begegnen, existieren zwei sehr ähnliche Lösungsvorschläge.

Der Computer wird zum Haushaltsgerät: Die Entwicklung von spezialisierten, auf Computertechnologie basierenden Geräten, die einem einzelnen Anwendungszweck dienen.

Durch Verzicht auf den Ballast eines Komplett-PC lässt sich das Gerät sowohl in seiner Hardware als auch in seiner Bedienung so stark vereinfachen, dass auch

Ein Webpad*-Prototyp der Firma Frontpath (2000).*

Laien damit ohne langwierige Lernphase umgehen können. Da nur noch sehr wenige Komponenten benötigt werden, lassen sich solche Geräte preiswerter und in einem besonders handlichen, praktischen Format realisieren.
Ein Beispiel ist das Konzept des *Webpad*, ein Mini-Computer mit Stiftbedienung im Format eines Schreibtabletts, der als problemloser Internet-Zugang für Dienste wie WWW oder E-Mail dienen soll. Webpad-Produkte werden seit einigen Jahren angekündigt, noch hat jedoch keines davon einen Erfolg im Massenmarkt erreicht.

Ein PDA *der Firma Palm (1998).*

Ein anderer Vertreter ist der *Personal Digital Assistant*, ein batteriebetriebener, digitaler Terminplaner, typischerweise im Hemdentaschen-Format. Mit reduzierter Hardware und auf seine Aufgabe optimierter Bedienung ordnet sich das Design dieses Kleinstcomputers vollständig seinem Anwendungszweck unter. PDA-Produkte sind seit etwa fünf Jahren ein großer Markterfolg und erreichen Käufer, die keine oder nur geringe Kenntnisse im Umgang mit Computern haben.

Das Haushaltsgerät wird zum Computer: Das Hinzufügen von Computertechnologie zu vorhandenen, dem Anwender vertrauten Geräten.
Wie bereits beschrieben, werden die Komponenten eines Computers immer kleiner und preiswerter; die Entwickler verwenden bereits jetzt häufig vollständige Kleinstcomputer (*Embedded Controller*) zur Steuerung ganz alltäglicher Geräte wie z. B. Fernseher.
Es ist nur noch ein kleiner Schritt, solche „intelligenten Haushaltsgeräte“ mit zusätzlichen Diensten auszustatten. Ein Beispiel sind moderne Fernseher mit Videotext, integriertem TV-Programm und im Gerät gespeicherter, am Bildschirm abrufbarer Bedienungsanleitung. Fügt man einige wenige Komponenten hinzu, kann man z. B. aus dem Fernseher ein vollwertiges Surf-Terminal für das Internet machen.

Bei beiden Lösungen spricht die IT-Industrie von *Intelligent Appliances*, um den Unterschied zum traditionellen Ansatz deutlich zu machen. So ein *intelligentes Haushaltsgerät* ist von außen nicht mehr als Computer erkennbar und der Anwender eines solchen Geräts käme nie auf den Gedanken, dass dieses Gerät technologisch mit seinem Arbeitsplatz-PC vergleichbar wäre.

Der Computer passt sich damit voll an seinen Anwendungszweck und an die Bedürfnisse des Anwenders an, *er wird „unsichtbar“*.

1.1.5 Vernetzung des Haushalts

Es sind nun mehr und mehr solcher „intelligenten“ Geräte im Haushalt vereint, jedes von ihnen ein kleiner Computer mit zusätzlichen Diensten, doch autonom. Eine Arbeitsteilung zwischen Geräten mit ähnlichen Aufgaben erscheint intuitiv sinnvoll, verlangt jedoch nach einer Kommunikationsmöglichkeit.

Auch die Bedienung der vorhandenen Appliances kann von einer Vernetzung des Haushalts profitieren, zum Beispiel um verschiedene Geräte über eine einzelne Anwender-Schnittstelle zentral zu steuern.

Vernetzte Küchengeräte, Produktstudien von Thaliaproducts / SunBeam Technologies (2000). Im Uhrzeigersinn: Wecker *mit integrierter Status-Anzeige der am Hausnetz angeschlossenen Geräte. Zentrale, abwaschbare* Kontroll-Konsole *für alle angeschlossenen Geräte, hier zusammen mit* Brotbackmaschine, Friteuse *und* Reiskocher. Rauch- und Feueralarm, Heizdecke *und* Kaffeemaschine.

Die Entwickler müssen bisher für autonome Geräte typische Ein- / Ausgabe-Komponenten in jedes Gerät einzeln einsetzen und setzen dabei zueinander unterschiedliche Bedienkonzepte um. Wer einen Fernseher, einen Satelliten-Tuner, einen Videorecorder, einen DVD-Player und eine Stereo-Anlage besitzt, wird mit mindestens ebenso vielen Fernbedienungen, Display-Typen und Benutzerschnittstellen-Konzepten konfrontiert, die erst einmal erlernt werden müssen.

Das Bedienkonzept des Videorecorders ist beispielsweise seit Jahrzehnten auf dem Markt, jede Geräte-Generation wird mit einer neuen, verbesserten Benutzerschnittstelle geliefert, und doch können die Mehrheit der Menschen nur den geringsten Teil der Möglichkeiten nutzen, weil sie den nötigen Lernaufwand meiden.

Es ist naheliegend, dass eine Kommunikation *zwischen* den Geräten hier Besserung schaffen könnte, so dass eine bevorzugte Benutzerschnittstelle für alle Geräte des Haushalts genutzt wird: Der Fernseher kann auch den Status einer Heizungssteuerung anzeigen, über Titel und Spielzeit der gerade von der Stereo-Anlage gespielten CD Auskunft geben, das TV-Programm der nächsten Woche über das Netz laden und die Sendezeit eines vom Anwender gewünschten Spielfilms an den Videorecorder übertragen, er kann auf das leere Papierfach im Faxgerät hinweisen oder Wartungsintervalle des Druckers anmahnen – um einige Möglichkeiten zu nennen.

Ebenfalls ergeben sich zusätzliche Möglichkeiten aus der Vernetzung wie die zentrale Steuerung sich wiederholender Schaltvorgänge. Oder der Fernzugriff auf die vernetzten Geräte von außen, z. B. kann ein defektes Gerät vom Kundendienst aus der Entfernung überprüft werden und der Techniker macht sich erst dann auf den Weg zum Kunden, wenn es sich tatsächlich um ein nicht-triviales Problem handelt.

1.1.6 Auswirkungen der Entwicklung auf Bürotechnik

Alle bisher beschriebenen Szenarien beziehen sich auf den privaten Heimbereich. Natürlich gibt es bereits sehr viel länger computergesteuerte Bürogeräte, die über ein im Gebäude verlegtes Netzwerk miteinander kommunizieren.

Viele der zuvor genannten Funktionen existieren im professionellen Bereich bereits. Aber sie sind teuer, kompliziert und betreuungsaufwendig: Bereits kleine Firmennetzwerke verlangen einen Vollzeit-Administrator, der sich einzig um die vernetzten Geräte kümmert.

Ein wesentlicher Grund dafür ist, dass bislang noch kein Hersteller übergreifender Standard für die *einfache* Vernetzung von Büromaschinen existiert.

Die Anforderung an einen Standard für vernetzte Bürotechnik ist dabei ganz ähnlich wie an einen Standard für vernetzte Haushaltsgeräte und von der Lösung des Problems im Heimbereich werden auch Büro-Anwender profitieren.

1.1.7 Die Suche nach neuen Marktsegmenten

Die Informationstechnologie-Industrie setzt große Hoffnungen auf diese Trends, denn sie befürchtet eine schwere Absatz-Krise, die sich bereits heute abzeichnet: Es fehlt die *Killerapplikation* der Zukunft.

Mit der „Killerapplikation“ bezeichnet man im IT-Marketing eine Anwendung, die jeder Nutzer *unbedingt* haben will. Sie gibt Anwendern einen ausreichenden Grund, sich einen Computer anzuschaffen oder einen vorhandenen, angesichts der neuen Anforderungen aber nicht mehr ausreichenden und somit veralteten Rechner zu ersetzen.

Der Büro- und Heimbereich war während der vergangenen 15 Jahre beispielsweise hauptsächlich durch Textverarbeitung, Tabellenkalkulation, elektronische Kontoführung, E-Mail und Internet-Anbindung geprägt.

Dieser Markt ist nach jüngsten Erhebungen in Kürze gesättigt. Fast jeder, der so einen „traditionellen“ Büro- / Arbeitsplatz-Computer benötigt, besitzt ihn bereits. Und

da aktuelle Systeme mit typischer Heimbüro-Software unterfordert sind, werden diese Anwender ohne einen „neuen Killer“ am Markt ihre Geräte erst in einigen Jahren ersetzen.

Die Industrie sucht also eine Möglichkeit, die übriggebliebenen Käufer zu erreichen, die sich *eigentlich* keinen Computer anschaffen würden oder bisher keinen Grund dafür sehen.

Eine Technologie, die einen vernetzten Haushalt und die Kommunikation zwischen Bürogeräten vereinfacht, könnte so einen neuen Markt schaffen und die IT-Industrie rechnet sich in diesem Bereich große Chancen aus.

Prototyp eines Kühlschranks mit integriertem Internet-Terminal und Rezeptdatenbank, Fujitsu ICL (1999). Praktisch identische Konzepte wurden von Electrolux (Kühlschrank) und Samsung (Mikrowellenherd) vorgestellt.

1.1.8 Die Vision des allgegenwärtigen Computers

Es ist aus den beschriebenen Trends heraus bereits abzusehen, dass wir uns kurz- bis mittelfristig *in ständiger Begleitung von Computertechnologie* befinden werden: Entweder von computergesteuerten Geräten, die wir selbst mitführen, oder von solchen, die wir an unserem Aufenthaltsort bereits vorfinden.

Ebenfalls werden alle diese Geräte mittel- bis langfristig in der Lage sein, miteinander Informationen auszutauschen, sei es über gemeinsame Nutzung einer Infrastruktur wie LAN und Internet oder über Kurzstrecken-Kommunikation via Funk, Infrarot oder Direktkontakt.

Mark Weiser prägte hierfür den Begriff des *Ubiquitous Computing* in [Wei91]. Seine Vision ist, dass alle diese Geräte gemeinsam wie ein einziger, *allgegenwärtiger Computer* als Werkzeuge ihres jeweilig gerade anwesenden Anwenders handeln können:

> „Das Ziel von Ubiquitous Computing ist es, die Bedienung von Computern zu verbessern, indem eine große Zahl Computer in der unmittelbaren physischen Umgebung verfügbar werden, sie jedoch für den Anwender unsichtbar bleiben.“ [Wei93]

Im Zusammenhang mit der zuvor beschriebenen Minituarisierung und dem Preisverfall der Technologie rechnet Weiser damit, dass sich in einem typischen Wohn- oder Arbeitsraum bald Hunderte von computergesteuerten Geräten befinden werden; von der Heizung über die Lichtanlage bis zur Informations-Displaytafel. Anwender

werden in seinem Szenario mit Hilfe von *Pads*, elektronischen Schreibtafeln in beliebigen Formaten, mit diesem allgegenwärtigen Computernetzwerk kommunizieren.

> „Pads unterscheiden sich in einem bedeutenden Punkt von konventionellen portablen Computern. Portable Computer folgen ihren Nutzer überall hin; dagegen wäre ein Pad ein Fehlschlag, wenn man es von Ort zu Ort tragen müsste. Pads sollen – ähnlich wie Schmierpapier – ersetzbar sein, an jedem Ort wird man eines aufheben und einsetzen können, die Geräte sind nicht individualisiert und ein einzelnes hat für sich keine spezielle Bedeutung.“ [Wei91]

In diesem Szenario würde der Mensch seine Aufgabe mitnehmen, nicht jedoch das Werkzeug: Die für die Arbeit nötigen Informationen lassen sich durch Vernetzung auch an anderen Orten durch den Anwender abrufen.

Mobiltelefone, PDAs oder Webpads gehen heute erste Schritte in die beschriebene Richtung, sind allerdings noch nicht „unsichtbar“ genug, um den Forderungen Weisers zu genügen. Er selbst rechnet erst frühestens für 2010 damit, dass die Voraussetzungen für Ubiquitous Computing erfüllt sein könnten.

1.1.9 Warten auf die Basistechnologie

Wie in den vorhergehenden Abschnitten gezeigt wurde, existieren bereits zahlreiche Ideen, Hoffnungen, Visionen und auch ökonomische Zwänge für eine einfache Vernetzung alltäglicher Technologie. Was ihnen fehlt, ist das Fundament, auf das sie bauen können.

Es gibt verschiedene Ansätze einzelner Hersteller, doch noch hat sich die Industrie nicht auf einen allgemein akzeptierten Standard einigen können. Bislang stockt die Entwicklung an dieser Stelle.

Als Schwerpunkt dieser Arbeit soll die von der Firma Sun vorgeschlagene *Jini* Infrastruktur vorgestellt werden. Weitere Technologien, die in dieser Arbeit zum Vergleich herangezogen werden sollen, sind *Universal Plug & Play* von Microsoft und *Bluetooth.*

Es wird sich in dieser Betrachtung zeigen, dass die Hardware-Anforderungen von Jini bislang noch höher sind als in der Industrie für den Masseneinsatz zur Zeit verfügbar oder ökonomisch sinnvoll ist. Solange die Hardware für Jini also *noch* nicht klein genug ist, wird ein Integrationspfad für Geräte mit zu geringer Leistung benötigt.

Ziel dieser Arbeit war es, eine solche Integration für Jini zu ermöglichen.

1.2 Überblick über die Arbeit

Im einleitenden Kapitel 1 wurde aufgezeigt, dass aufgrund der fortschreitenden Hardware-Entwicklung und weiterer Trends mittelfristig mit der Technologie für vernetzte Alltagsgegenstände wie Haushaltsgeräte zu rechnen ist, als ein Zwischenschritt zur langfristigen Vision des allgegenwärtigen, unsichtbaren Computers.

Kapitel 2 beschreibt die Grundlagen von Middleware und Ad Hoc Vernetzung; mit einem besonderen Schwerpunkt auf der Java Systemumgebung, welche selbst große Teile einer Middleware implementiert. Jini als Erweiterung zu Java stellt die noch fehlenden Komponenten zur Verfügung

Kapitel 3 zeigt die wichtigsten Konzepte von Jini auf und vergleicht sie mit zwei anderen kommerziellen Ad Hoc Netz-Technologien: Universal Plug & Play und Bluetooth. Als wesentlicher Schwachpunkt von Jini wird sich die Voraussetzung einer Java Virtual Machine (JVM) in zu vernetzenden Geräten herausstellen.

Kapitel 4 betrachtet die Jini Standard-Basisprotokolle im Detail und stellt fest, warum diese eine JVM in allen beteiligten Geräten zwingend notwendig machen. Es wird eine Modifikation der Protokolle erarbeitet, die Geräten ohne JVM einen Zugang zu Jini ermöglichen.

Kapitel 5 stellt die Implementation dieses modifizierten Protokolls vor und erläutert die Technologien, die dabei zum Einsatz kamen. Zur Demonstration des Konzepts dient eine Webcam, deren Systemumgebung zu leistungsschwach für eine JVM ist.

Kapitel 6 erläutert abschließend die Vor- und Nachteile des gewählten Lösungswegs. Es wird gezeigt, warum Sun sich schließlich für einen sehr ähnlichen, letztendlich aber allgemeineren Integrationspfad für nicht-javafähige Geräte entschieden hat.

Kapitel 2

Verteilte Systeme und das Middleware-Konzept

2.1 Motivation

Die Einleitung gab einen kurzen Überblick über die Evolution der Computer-Hardware sowie über neue ökonomische Interesse und Zwänge der Industrie und wie sich in diesen Entwicklungen ein Weg zum vernetzten Haushalt aufzeichnet.

Bevor im darauf folgenden Kapitel einige kommerzielle Lösungsvorschläge für dieses Problem verglichen werden, soll dieses Kapitel zunächst die Anforderungen und theoretischen Grundlagen dieser Konzepte aufzeigen und einige Basistechnologien vorstellen.

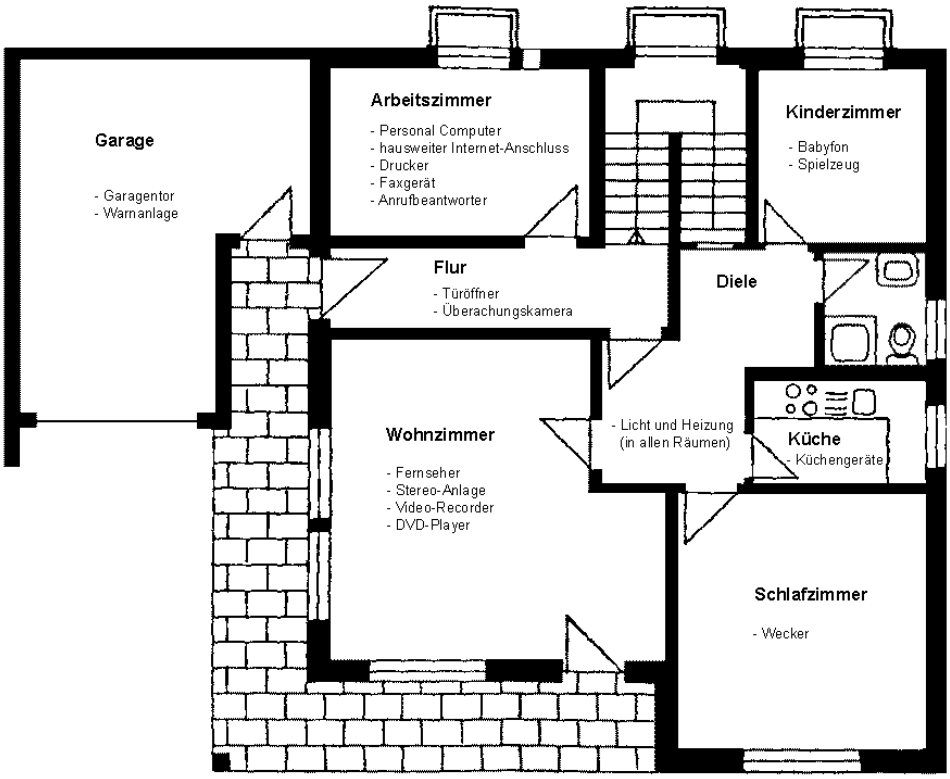
2.2 Anforderungen an den vernetzten Haushalt

Die bislang vorhandenen, meist im Büro-Umfeld genutzten Netzwerktechnologien haben sich in der Vergangenheit als zu kompliziert in Einrichtung und Bedienung gezeigt, sie wären also ungeeignet für einen vernetzten Haushalt, dessen Anwender Technik-Laien sind (und bleiben wollen).

Aus Sicht des Anwenders ergeben sich aus dieser Erfahrung heraus einige naheliegende Anforderungen, die die bisherigen Lösungen nicht oder nur teilweise erfüllen konnten:

Unproblematische Einrichtung und Betrieb

Der Anwender interessiert sich nicht für die Vernetzung, er will aber trotzdem auch ohne Hilfe eines Fachmanns in der Lage sein, die gewünschten Geräte zu vernetzen. Planung, Aufbau und Betrieb eines solchen Netzwerks muss auch einem Laien möglich sein. Er verlangt echtes *Plug & Play* – das Gerät „einschalten und verwenden“ – und will sich nicht mit der Vergabe von Netzwerkadressen oder der Installation von Treibersoftware beschäftigen.



Für die Vision des vernetzten Haushalts setzt die Industrie auf die Vernetzung zahlreicher alltäglicher Geräte. Ein Anwender wird verlangen, dass diese Weiterentwicklung vertrauter Technologie keinen zusätzlichen Aufwand bei Einrichtung und Bedienung bedeutet.

Unproblematische Änderung

Das Hinzufügen oder Entfernen von Geräten in einem vorhandenen Netz darf nicht zu Mehrarbeit für den Anwender führen; Besucher müssen in der Lage sein, mitgebrachte Geräte problemlos einzusetzen, etwa als Hotelgast, der von seinem Laptop aus ein Dokument am Laserdrucker der Rezeption ausdrucken will.

Sicherheit

Das vernetzte System darf weder von innen noch von außen angreifbar sein. Mitgebrachte Geräte dürfen z. B. nicht die vertraulichen Dokumente des Gastgebers von der Festplatte des lokalen PCs lesen. Ebenso will der Anwender nicht, dass z. B. der Kundendienst eines Herstellers, der per Netz auf ein lokales Gerät zugreift, Zugriff auf andere Geräte erhält.

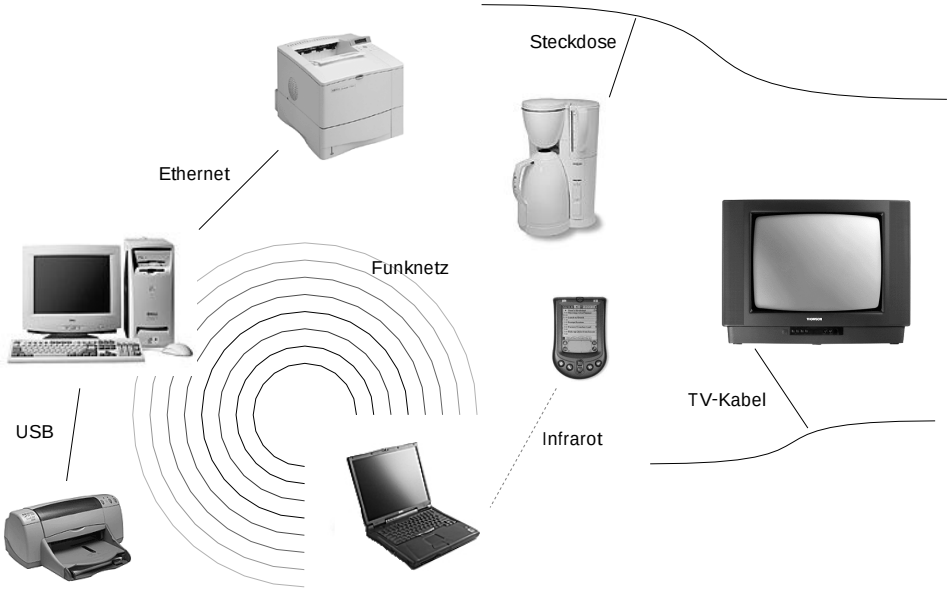
Ausfallsicherheit

Das Haushaltsnetz darf bei Ausfall einer einzelnen Komponente nicht plötzlich seinen Nutzen verlieren. Ausfälle müssen für den Anwender leicht zu erkennen und behebbar sein oder automatisch behoben werden.

Unterstützung aller Geräteklassen durch alle Hersteller

Der Anwender ist nicht daran interessiert, für verschiedene Geräte jeweils verschiedene, zueinander inkompatible Einzelnetze mit verschiedenen, inkompatiblen Schnittstellen aufzubauen.

Ein entsprechender Standard muss deshalb im Idealfall Unterstützung durch alle relevanten Gerätehersteller finden und er muss sich in möglichst allen Geräteklassen implementieren lassen – egal, ob das Gerät von einem simplen Ein-Chip-Controller oder von einem vollwertigen Computer gesteuert wird.



Verschiedene Geräteklassen verwenden verschiedene Wege der Datenübertragung. Anders als stationäre Haushalts- und Büromaschinen bevorzugen mobile Geräte einen drahtlosen Kommunikationsweg.

Daraus folgt als weitere Anforderung die Unterstützung von konkurrierenden oder veralteten Standards und ein Integrationspfad für Geräte, die von sich aus nicht die technischen Voraussetzungen mitbringen, etwa weil sie zu wenig Speicher oder zu wenig Rechenkapazität bereitstellen.

Geringe Kosten

Die Geräte dürfen durch die neue Möglichkeit der Vernetzung nicht deutlich teurer werden als zuvor. Im Idealfall bleibt der Preis nahezu gleich und die neue Schnittstelle wird vom Kunden bereits mitgekauft, bevor er sich entscheidet, sie zu nutzen.

2.3 Haushaltsgeräte als verteiltes System

Verteilte Systeme sind seit Jahren Gegenstand der Forschung in der Informatik und im Vergleich zeigt sich, dass es sich beim vernetzten Haushalt um einen Spezialfall eines *offenen verteilten Systems* (vergleich [CDK01]) handelt, das um die Möglichkeit der *Ad Hoc Vernetzung* (Spontanious Networking) erweitert wurde:

2.3.1 Klassische Herausforderungen und Lösungswege beim Einsatz offener verteilter Systeme

Heterogenität

Geräte verschiedener Hersteller, Geräteklassen und Produktgenerationen sind in einem Netz zu vereinen.

Neben der Einigung auf ein gemeinsames *Binärdatenformat* bei der Repräsentation von Daten (Reihenfolge der Bits, Länge und Format der Datenworte, digitale Darstellung von Adressen, Uhrzeiten oder Zeichenketten) ist ein von den Teilnehmern verstandenes, gemeinsames *Protokoll* notwendig.

Die Wahl des Protokolls wird wiederum durch die gewünschten *Datenübertragungswege* eingeschränkt, da beispielsweise nicht für alle Geräte eine Vernetzung via Kabel möglich ist, andererseits jedoch nicht jedes Protokoll auf drahtlosem Wege einsetzbar ist.

Offenheit

Das Protokoll bzw. die *Programmier-Schnittstelle* (API) dazu muss *dokumentiert* sein, um eine Implementation oder Erweiterung des Systems durch Außenstehende zu erlauben.

Die *Standarddienste* des Systems müssen allgemein spezifiziert werden, um allen Teilnehmern den Zugriff darauf zu ermöglichen und alternative Implementationen dieser Dienste zu erlauben.

Diese Dokumentation muss im Rahmen einer erschwinglichen *Lizensierung* verfügbar sein. Ein *Konformitätstest* ist notwendig, um die Produkte verschiedener Hersteller auf korrekte Implementation zu prüfen.

Sicherheit

Jedes verteilte System macht sich – gerade wegen seiner Kommunikationsschnittstellen – angreifbar. „Der einzige wirkliche Schutz vor Angriffen aus dem Netz ist es, das System nicht zu vernetzen“.

Trotzdem gibt es Möglichkeiten, Angriffe zu erschweren und das System in einem sinnvollen Maße abzuschotten. Dazu zählen insbesondere *Verschlüsselung* der Datenübertragung, sowie die *Authentifizierung* der Kommunikationspartner mit darauf aufbauender anschließender *Autorisierung* beim Zugriff auf Resourcen. Beispiele sind der Einsatz von *digitalen Signaturen* für Datenpakete und für mobile Code-Fragmente und das *Sandkasten*-Prinzip (Sandbox), bei dem fremder Code in einer kontrollierten, eingeschränkten Umgebung ausgeführt wird.

Skalierbarkeit

Mit dem Wachstum des Systems steigt auch der Verwaltungs- und Kommunikationsaufwand zwischen den Komponenten. Der daraus resultierende steigende *Resourcen- und Geschwindigkeitsverlust* darf einzelne Teilnehmer nicht überfordern. Er muss *auf lange Sicht* abschätzbar in einem vom Benutzer akzeptieren Bereich bleiben und durch Erweiterungen von Hard- und Software auszugleichen sein.

Insbesondere der *Adressbereich* ist so zu wählen, dass er ausreichend *Reserven* bietet. Das in den 70er Jahren entwickelte TCP/IP-Protokoll verwendet beispielsweise einen 32 Bit breiten Adressbereich, was zum damaligen Zeitpunkt als mehr als genügend erschien. Durch das rasante Wachstum des Internets wird aber in wenigen Jahren ein Umstieg auf ein neues Adressformat nötig werden. Die Einführung des Nachfolgeprotokolls mit 128 Bit breiten Adressen bedeutet einen hohen Umstellungsaufwand, der durch Vorausplanung von Anfang an hätte vermieden werden können.

Ausfallsicherheit

Wenn die Komponenten einer Applikation über mehrere Netzknoten verteilt sind, kann der Ausfall einzelner Knoten oder einzelner Netzstrecken schwerwiegende Folgen für das verteilte System haben.

Dementsprechend müssen besondere Maßnahmen für die *Erkennung und Beseitigung von Teilausfällen* getroffen werden.

Eine Möglichkeit ist *Redundanz*, die Bereitstellung von mehrfachen Instanzen eines für die Applikation wichtigen Dienstes auf verschiedenen Netzknoten, zusammen mit *Replikation* wichtiger Daten zwischen diesen Instanzen, um stets einen konsistenten Datenstand zu sichern.

Nebenläufigkeit

Die Probleme, die bei nebenläufiger Ausführung von Aktivitäten entstehen können, sind aus der theoretischen Informatik hinreichend bekannt und lassen sich durch *Synchronisation von Prozessen* bereits bei der Planung des Gesamtsystems beweisbar ausschließen, etwa durch den Einsatz von Einsatz von Techniken zum *gegenseitigen Ausschluss* (Mutex, Semaphore), zur *sicheren Verwaltung von Ressourcen* (Monitore), und zur *Synchronisation* von Prozessen (geteilte Speicherbereiche)

Ein für verteilte Systeme spezielleres Problem der nebenläufigen Ausführung stellt die Einigung auf eine gemeinsame Zeitbasis dar. Da jedes Gerät eine eigene physikalische Uhrzeit führt, muss für den Einsatz eines zeitbasierten Protokolls zunächst eine *Zeit-Synchronisation* aller beteiligten Komponenten erfolgen.

Eine Möglichkeit ist es, durch Nachrichtenaustausch unter Einrechnung der Übermittlungszeit eine für das verteilte System netzweit gültige *Standardzeit* durchzusetzen. Ein einfacherer Weg ist der Verzicht auf absolute Uhrzeiten und stattdessen nur noch *relative Zeitpunkte* im Rahmen des Protokolls zu übermitteln.

Transparenz

Anwender und Applikations-Entwickler sollen das verteilte System als ein Ganzes begreifen und nutzen können, nicht als eine Sammlung einzelner Komponenten. Dies

wurde zuvor am Beispiel für den vernetzten Haushalt aus Anwendersicht bereits unter dem Begriff „unproblematisch“ zusammengefasst.

In [CDK01] wird das ANSA Referenzhandbuch [ANSA89] mit mehreren Typen der Transparenz zitiert und teilweise angepasst:

Zugriffstransparenz ermöglicht mit identischen Operationen den Zugriff auf lokale und auf entfernte Resourcen.

Lokationstransparenz erlaubt den Zugriff auf Resourcen ohne Kenntnis ihres Standorts.

Nebenläufigkeitstransparenz macht die nebenläufige Ausführung mehrere Prozesse möglich, die ohne gegenseitige Störung auf geteilte Resourcen zugreifen können.

Replikationstransparenz erlaubt den Betrieb mehrerer Instanzen einer Resource (zur Steigerung der Zuverlässigkeit und Performanz des Gesamtsystems), ohne dass der Anwender oder Entwickler davon Kenntnis nehmen muss.

Ausfallstransparenz ermöglicht es, vor Anwender und Entwickler Teilausfälle des Systems zu verstecken und die vollständige Erledigung der gestellten Aufgaben trotzdem zu sichern.

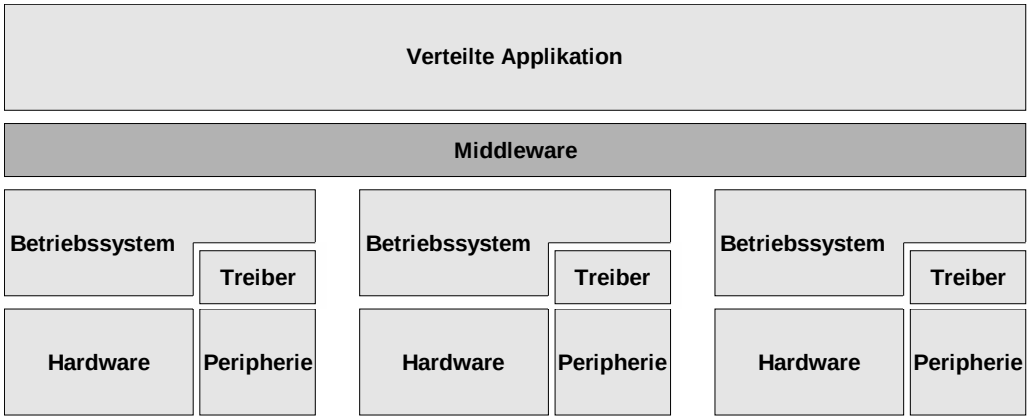
Mobilitätstransparenz lässt es zu, dass sich Dienstleister und Klienten innerhalb des Systems bewegen können, ohne dass dadurch die Funktion Gesamtsystem beeinträchtigt würde.

Performanztransparenz erlaubt es, die Performanz des Systems durch Rekonfiguration zu verändern, um es an eine geänderte Auslastung anzupassen.

Skalierungstransparenz ermöglicht das Wachstum des Systems, ohne das dafür Änderung an seiner Struktur oder den zu Grunde liegenden Algorithmen nötig werden.

2.4 Middleware

Die zur Realisierung eines offenen verteilten Systems benötigte Software-Architektur wird als *Middleware* bezeichnet. Der Begriff wurde von Bernstein in [Ber93] geprägt und bezeichnet eine zusätzliche Software-Schicht *zwischen* Betriebssystem und Applikation.



Die Middleware-Schicht abstrahiert von Hardware und Betriebssystem verschiedener Systemumgebungen für die Anwendung durch eine verteilte Applikation.

Wie die Betriebssystem-Schicht den Zugriff auf Hardware in eine Standard-API abbildet, abstrahiert die Middleware unterschiedliche Eigenschaften von Betriebssystemen und Rechnerarchitekturen, um die zuvor genannten Forderungen an ein verteiltes System zu erfüllen.

2.4.1 Typische Komponenten einer Middleware

Datenrepräsentation und -übertragung

Für die Basistypen digitaler Kommunikation haben sich bereits eine Reihe von informellen Standards durchgesetzt.

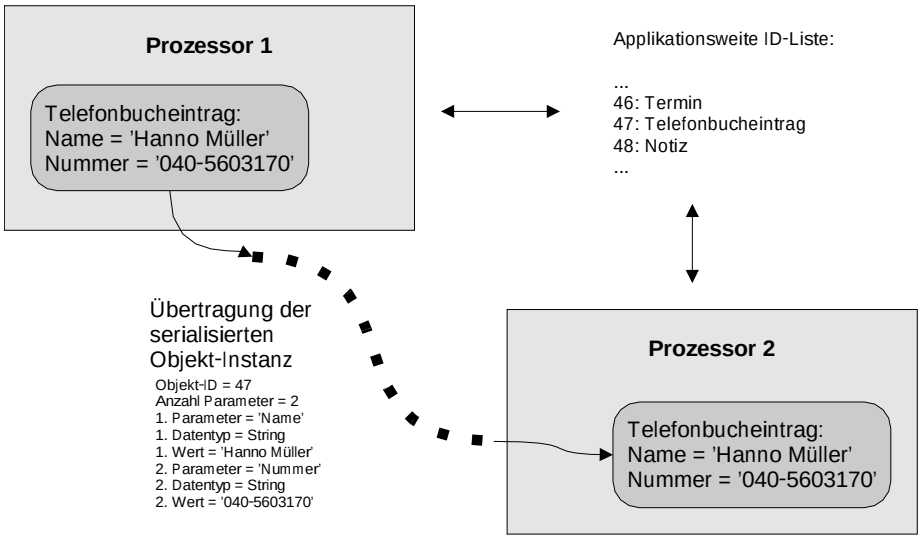
Mit dem TCP/IP-Protokoll hat sich die *Most Significant Byte Order* als Standard-Binärdarstellung von ganzzahligen Datenworten für Netzprotokolle durchgesetzt, sie wird deshalb auch *Network Byte Order* genannt. Einige der auf dem Markt verfügbaren Rechnerarchitekturen verwenden intern eine andere Zahlendarstellung (zum Beispiel die Prozessorfamilie der Firma Intel) und es ist Aufgabe der Middleware, diese Formate ineinander zu konvertieren.

Entsprechend werden für Zeichenketten *ASCII*- oder *Unicode* und für Fließkommazahlen die *IEEE*-Standard-Binärdarstellung verwendet.

Interessanter ist das für Middleware typische *Marshalling*-Verfahren für die Übertragung zusammengesetzter Datentypen und Objekte.

Sind die Reihenfolge und Datentypen der in einem zusammengesetzten Typ verwendeten Datenworte bekannt, genügt es, eine dem Datentyp eindeutig zugeordnete Kennung (ID) und die Liste der Werte zu übertragen.

In einem objektorientierten Umfeld bezeichnen diese ID und die Werte gemeinsam eine Instanz eines Objekts in der Laufzeitumgebung und werden als *Serialisierung* der Objektinstanz bezeichnet. Der empfangende Prozessor kann aus diesen Informationen eine identische *Kopie* dieser Objektinstanz anlegen und anhand des enthaltenen Status die Ausführung fortsetzen. Somit können ganze Software-Komponenten während der Erfüllung ihrer Aufgabe von Prozessor zu Prozessor wandern.



Serialisierung erlaubt die Übertragung von Objektinstanzen zwischen zwei Prozessoren einer verteilten Applikation. Sender und Empfänger greifen auf eine applikationsweit gültige Liste von Objekt-IDs zurück.

Ist der empfangende Prozessor zusätzlich in der Lage, bei Empfang von noch unbekannten Objekt-IDs den zur Ausführung benötigten Programmcode nachzuladen, sind alle Voraussetzungen für die *Code-Migration* von Objekten erfüllt.

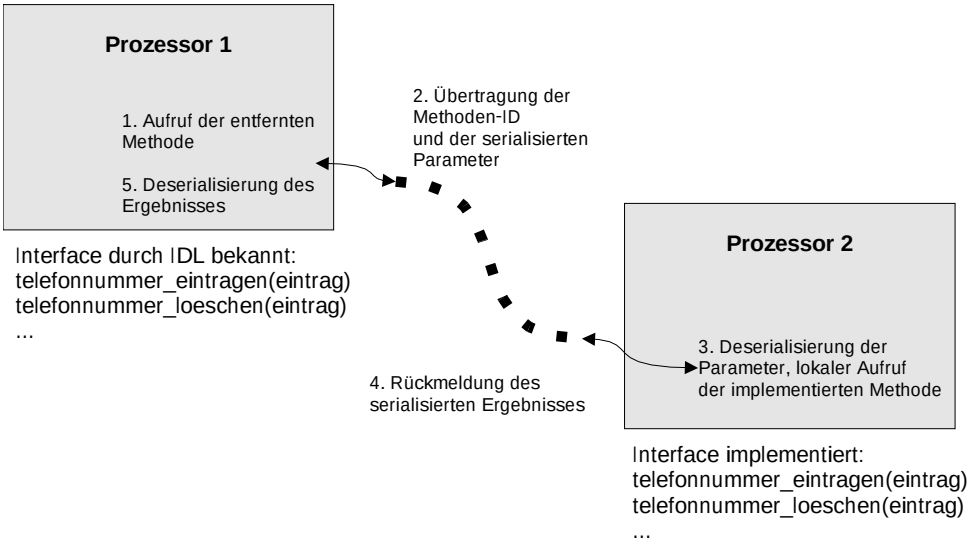
RMI und Dienstbeschreibung

Als Kommunikationsmechanismus zwischen den Prozessoren in verteilten Systemen dient der *Remote Procedure Call*, beziehungsweise *Remote Method Invocation* im objektorientierten Umfeld.

Hierfür nimmt der aufrufende Prozessor eine Verbindung zum entfernten Prozessor auf, teilt ihm eine eindeutige Referenz des Dienstleisters und der aufzurufenden Methode mit, überträgt anschließend die Parameter in serialisierter Form und empfängt das serialisierte Ergebnis.

Hierfür ist zunächst eine *Dienstbeschreibung* notwendig, die üblicherweise in einer *Interface Definition Language* notiert wird. Anhand der Beschreibung in der IDL kennt der Dienstnehmer die Schnittstellen zum Dienstleister.

Die durch die IDL beschriebene Schnittstelle stellt somit eine Vereinbarung zwischen Dienstleister und -nehmer dar. Sie lässt dem Entwickler des Dienstleisters freie Hand, wie genau letztendlich die Implementation aussieht - sie muss nach außen nur



Durch die Interface Definition Language können Dienstnehmer die Methoden eines entfernten Dienstleisters aufrufen, ohne die Details der Implementation zu kennen. Remote Method Invocation *sorgt für die korrekte Übermittlung der aufzurufenden Methode, der Parameter und der Ergebnisse zwischen den beiden entfernten Prozessoren.*

die vereinbarte Schnittstelle erfüllen.

Middleware-Komponenten sind deshalb in aller Regel nur über ihre IDL standardisiert und verschiedene Entwickler können so konkurrierende Implementationen herstellen, die beispielsweise intern unterschiedliche Datenbanken oder verschieden effiziente Algorithmen verwenden.

Einige IDL-Varianten erlauben es, verschiedene Generationen einer Schnittstelle zu definieren. So wird es möglich, kompatible Weiterentwicklungen oder Obermengen eines Dienstleisters zu definieren und auch verschiedene Stufen einer Schnittstelle zu implementieren.

Aus der IDL erzeugt ein *IDL-Compiler* eine Anbindung an die verwendete Programmiersprache, wobei die eigentliche, durch die Middleware vorgegebene Kommunikation vom Entwickler gekapselt wird. Durch diesen Schritt können Dienstleister und Dienstnehmer in völlig verschiedenen Programmiersprachen auf ganz unterschiedlichen Rechnersystemen implementiert werden, sofern ein IDL-Compiler für diese Plattform verfügbar ist.

Ein Beispiel für diesen Ansatz ist die weit verbreitete CORBA-Middleware, die IDL-Compiler für fast alle gängigen Systemumgebungen anbietet.

Namensdienst

Die für einen RMI-Aufruf nötige eindeutige Referenz zum Dienstleister kann beispielsweise aus der physikalischen Netzwerkadresse des Prozessors und aus einer ID der darauf laufenden Dienstleister-Instanz bestehen.

Anwender und Entwickler einer verteilten Applikation werden gegenüber so einer überlicherweise nur aus Zahlen bestehenden Adresse jedoch einen eindeutigen Namen

bevorzugen.

Die Aufgabe des *Namensdienstes* ist es, solche Namen in Referenzen abzubilden. Ein Namensdienst erlaubt es damit einem Dienstleister auch, seine Adresse zu ändern, also z. B. zwischen Netzknoten des verteilten Systems zu wandern.

Eine weitere Anforderung ist die Aufteilung des verteilten Systems in *Namensräume*, etwa um verschiedene Dienstleister-Typen zu gruppieren oder auch nur um die räumliche Verteilung der Dienstleister abzubilden. Im Szenario des vernetzten Haushalts z. B. Teilnetze wie „Wohnzimmer“, „Küche“ und „Büro“.

Ein Namensdienst kann dabei mehrere Namensräume verwalten. Bei Anfragen bzgl. fremder Namensräume wird er den entfernten Namensdienst konsultieren und das Ergebnis temporär zwischenspeichern. Um die Ausfallsicherheit dieses für die verteilte Applikation lebensnotwendingen Dienstes zu erhöhen, können mehrere redundante Namensdienste einen Namensraum verwalten.

Ein Beispiel ist der im Internet verwendete *Domain Name Service* (DNS), er ist der wohl zur Zeit bekannteste Namensdienst. Er bildet Rechnernamen wie `vsys1.informatik.uni-hamburg.de` in numerische IP-Adressen um und verwaltet die *Domänen* genannten Namensräume.

Eine Domäne gehört einer Firma oder Organisation, in diesem Fall die Domäne `informatik.uni-hamburg.de` dem Fachbereich Informatik der Universität Hamburg in Deutschland. Domänen dürfen hierarchisch gegliederte Subdomänen enthalten, hier die Subdomäne Fachbereich Informatik, die der Domäne der Universität zugeordnet ist. Der Fachbereich Mathematik betreut davon getrennt seine eigene Subdomäne `math.uni-hamburg.de` und betreibt für diese einen eigenen DNS-Server.

Dienstevermittlung

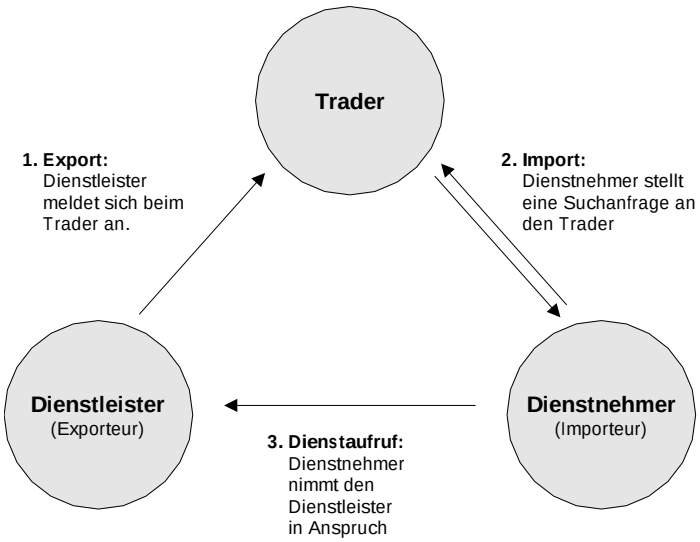
In einem verteilten System können mehrere verschiedene Objekt-Instanzen einen von einem Nutzer gewünschten Dienst anbieten, seien es mehrere redundante Instanzen der gleichen Implementation oder konkurrierende Implementation verschiedener Entwickler, die diesen Dienst mit jeweils anderen Leistungsmerkmalen bzw. *Attributen* anbieten.

So könnte beispielsweise einer der beiden im Netz verfügbaren Drucker monochrome Ergebnisse liefern, diese aber sehr schnell und zu geringen Seitenkosten, während der zweite Drucker farbig, aber langsam und teuer arbeitet.

Ein *Vermittlungsdienst* (engl. *Trader*) kann solche Dienstleister anhand der vom Dienstnehmer gewünschten Attribute vermitteln. Im Beispiel wird die Suche nach „ein Drucker“, „ein schneller Drucker“ oder „ein farbiger Drucker“ jeweils ein unterschiedliches Ergebnisse liefern.

Im Fall der ersten Suchanfrage ist das Ergebnis nicht deterministisch. Wenn nur ein einzelnes Ergebnis verlangt wurde, bleibt es dem Vermittlungsdienst überlassen, welches der mehreren möglichen Ergebnisse er zurückliefert.

Der Vorgang einer Dienste-Vermittlung ist durch das sog. *Trader-Dreieck* abgebildet:



Die Beziehung der an einer Dienste-Vermittlung beteiligten Parteien stellt das Trader-Dreieck *dar.*

Export: Ein Dienstleister meldet sich beim Vermittlungsdienst an und teilt diesem seinen Namen, seine Adresse, seinen Diensttyp und seine Leistungsmerkmale mit.

Import: Ein Dienstnehmer fragt beim Vermittlungsdienst nach einem Dienstleister eines bestimmten Diensttyps an und schränkt die Suche gegebenenfalls durch die zusätzliche Angabe von *Constraints*, einer Liste von gewünschten Leistungsmerkmalen, ein.

Dienstaufruf: Der Dienstnehmer nimmt eine Verbindung zum gefundenen Dienstleister auf, ruft die gewünschte Dienstleistung via RMI auf und übergibt dazu per Serialisierung die benötigten Parameter. Soweit der Aufruf ein sofortiges Ergebnis zurückliefert, wird dieses ebenfalls per Serialisierung abschließend an den Dienstnehmer gesendet.

2.5 Ad Hoc Vernetzung

Bevor ein verteiltes System seine Aufgabe erfüllen kann, müssen neben der physischen Installation von Geräten und Verkabelung weitere Voraussetzungen auf Systemseite erfüllt sein:

- Alle Geräte benötigen eine eindeutige Netzwerk-Adresse.
- Alle Geräte müssen den/die Namensdienst(e) kennen.
- Alle Geräte müssen den/die Vermittlungsdienst(e) kennen.

Erst danach kann mit Hilfe des Vermittlungsdienstes die Suche nach Dienstleistern beginnen.

In einem Firmennetz werden diese vorbereitenden Einstellungen heute noch häufig durch einen System-Administrator von Hand vorgenommen. Es ist jedoch zu viel verlangt, dem Heim-Anwender die Einrichtung dieser Voraussetzungen zu überlassen – und auch im geschäftlichen Umfeld wird eine Vereinfachung des Netzaufbaus gerne angenommen.

Ein Netzsystem, das *Ad Hoc Vernetzung* (Spontaneous Networking) anbietet, übernimmt diese Einrichtung automatisch und erlaubt dem Anwender das bereits geforderte „Einschalten und Verwenden“ in Bezug auf die Inbetriebnahme seiner zu vernetzenden Geräte.

Die zur Zeit am häufigsten anzutreffende Form der Vernetzung ist Ethernet in Kombination mit dem TCP/IP Protokoll. Die folgenden beiden Beispiele sollen Möglichkeiten der Ad Hoc Vernetzung in einer solchen Umgebung aufzeigen.

2.5.1 DHCP

Das *Dynamic Host Configuration Protocol* [Dro97-1] ermöglicht es, in einer Ethernet-Umgebung automatisch IP-Adressen für TCP/IP zu vergeben. Es verwendet hierfür ein einfaches Broadcast-Protokoll auf Netzwerkschicht.

Für DHCP muss im Netzwerk bereits ein DHCP-Server installiert sein, der die Vergabe der Adressen koordiniert. Er wartet auf Ethernet-Broadcast-Anfragen von Geräten aus dem Netz und vergleicht die Ethernet-Hardware-Adresse des Absender mit einer Liste der Geräte, die er bereits kennt und denen er bereits eine IP-Adresse zugeteilt hat.

Die 48 Bit breite Ethernet-Hardware-Adresse ist eine weltweit eindeutige ID der Netzwerkkarte, die bereits zum Zeitpunkt der Herstellung vergeben wird. (Es ist zwar inzwischen bei einigen Geräten möglich, diese ID nachträglich zu ändern, doch fordern der ursprüngliche Standard und alle Ethernet-Implementationen, dass innerhalb eines Subnetzes niemals zwei identische IDs vorkommen.)

Soweit nötig, vergibt der DHCP-Server nun dem hinzugefügten Gerät eine eindeutige IP-Adresse. Der Server verwaltet dafür einen vorkonfigurierten Adressbereich. Über Erweiterungen des DHCP-Protokolls kann der Server auch weitere Informationen wie die IP-Adresse von Gateway und DNS-Server an das Gerät weiterleiten, eine Arbeitsstation ohne eigenen Massenspeicher kann auch gleich das für die Inbetriebnahme nötige Startprogramm vom DHCP-Server erhalten [Dro97-2].

Ein DHCP-Server verwendet *Leasing*: Vergebene Adressen verfallen nach einer Zeit von einigen Minuten bis Stunden und das Gerät muss die Adresse erneut anfordern. Bei Ausfall oder Entfernen eines Geräts aus dem Netz bleibt die IP-Adresse so nicht blockiert und wird für andere Geräte wieder frei.

2.5.2 TCP/IP Multicast Protokoll

Ethernet Broadcast funktioniert nur innerhalb eines Ethernet Netzsegmentes. Soll ein Protokoll zur Suche nach Dienstleistern über dieses Segment hinausgehen, bietet sich das *Multicast* Protokoll als Lösungsweg an [Dee89].

Multicast-Datenpakete können ebenso wie Ethernet Broadcast nach nur einmaliger Versendung von beliebig vielen Netzteilnehmern empfangen werden – anders als

Unicast- bzw. Punkt-zu-Punkt-Verbindungen, die hierfür an mehrfache Empfänger jeweils eine individuelle Kopie der Daten senden müssten.

Die Empfänger von Multicast-Daten melden sich explizit beim lokalen Router für den Empfang einer solchen Übertragung im Netzwerk an – [Goy98] vergleicht dies mit Radiohörern, die ihr Radio auf die gewünschte Frequenz eines Senders einstellen. Die „Frequenz“ ist eine Kombination aus der *Multicast-Gruppe*, einer IP-Adresse aus den im TCP/IP-Standard für Multicast reservierten Adressbereichen, und einer Port-Nummer.

Der Router des lokalen Netzwerks wird über eine Anmeldung eines Teilnehmers aus einer Multicast-Gruppe informiert und kann so Multicast-Übertragungen von externen Sendern ins lokale Netz weiterleiten oder lokale Sender mit externen Netzen verbinden.

Multicast ist ein ungesicherter Übertragungsweg. Datagramme können – unbemerkt für den Absender – auf dem Weg zu dem oder den Empfänger(n) verloren gehen. Anhand einer Prüfsumme wird der Empfänger zwar defekte Datagramme verwerfen, die während des Versands zerstört wurden, doch eine darüber hinausgehende Fehlerkorrektur ist nicht vorgesehen.

Demgegenüber ist in TCP/IP-Unicast nach erfolgtem Verbindungsaufbau eine protokoll-eigene Fehlerkorrektur in der Lage, typische Übertragungsfehler auszugleichen und erkennt einen Verbindungsabbruch, den die darunter liegenden Schichten beiden Seiten mitteilen.

2.6 Die Java Systemumgebung als Middleware

Nachdem einige Grundeigenschaften und -elemente einer Middleware beschrieben wurden, soll an dieser Stelle die *Java* Systemumgebung der Firma Sun vorgestellt werden. Obwohl Jini, das eigentlich Thema dieser Arbeit, auf Java basiert, kann bereits Java für sich allein genommen teilweise als Middleware bezeichnet werden. Jini vervollständigt und verbessert diese Eigenschaften noch weiter, wie später dargestellt wird.

2.6.1 Einführung in Java

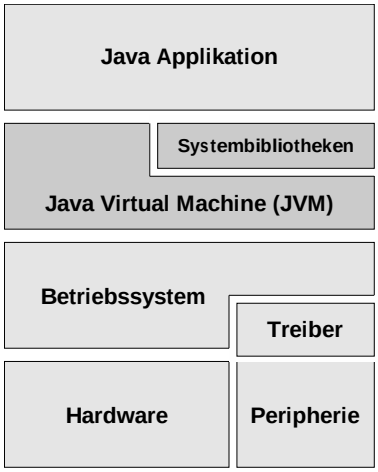
Die Entwicklung von Java begann Ende 1990 als ein Sun-internes Forschungsprojekt, ursprünglich unter dem Namen „Oak“. Das Ziel war eine Systemumgebung für den Einsatz in Set-Top-Boxen und speziell auch in chip-gesteuerten Haushaltsgeräten.

Man wollte die Nachteile der für diesen Zweck traditionell verwendeten Sprachen Assembler oder C überwinden und eine stark vereinfachte, bequemere Programmiersprache schaffen und gleichzeitig die Software-Entwicklung unabhängiger von der benutzten Hardware machen.

Hieraus entstanden die Programmiersprache Java und ein virtueller Java Prozessor, die zusammen mit den standardisierten Laufzeitbibliotheken die Java Systemumgebung bilden.

Das Set-Top-Box Projekt wurde von Sun bald darauf wieder fallen gelassen, doch zu diesem Zeitpunkt begann bereits der Siegeszug des World Wide Web. Das WWW wurde schnell zur bislang erfolgreichsten verteilten Applikation und als Middleware

bot sich aus Sicht von Sun die eigene Entwicklung Java an: Eine einfache, plattformunabhängige Programmiersprache mit einem Laufzeitsystem, das von der darunter liegenden Hardware abstrahiert.



Die Java Systemumgebung abstrahiert von der Hardware, auf der die Java Virtual Machine ausgeführt wird.

Als Proof of Concept hierfür entstand 1995 der HotJava Browser, ein WWW-Browserprototyp, der vorführte, wie man auf einfache Weise sicher und plattformunabhängig Code von anderen Servern laden und lokal ausführen kann.

Mit der kurz darauf folgenden Unterstützung dieser *Java-Applets* durch den zu diesem Zeitpunkt noch populärsten WWW-Browser *Netscape* fand Java schließlich seinen Durchbruch als Programmierumgebung für interaktive Anwendungen auf Seite des Klienten im World Wide Web. Inzwischen unterstützen alle bedeutenden Browser die Ausführung von Java-Applets.

Erst sehr viel später wuchs die Bedeutung von Java auch auf Seiten des Servers. Unabhängig von Applets und WWW ist Java nun auch eine eigenständige Systemumgebung speziell im Bereich der verteilen Systeme geworden.

2.6.2 Die Java Systemumgebung

Die Java Umgebung wird von Sun selbst als „einfach, objektorientiert, verteilt, robust, sicher, plattformunabhängig, portabel, schnell, Multithread-fähig und dynamisch“ charakterisiert. Sie besteht aus drei Komponenten, um diese Eigenschaften zu implementieren:

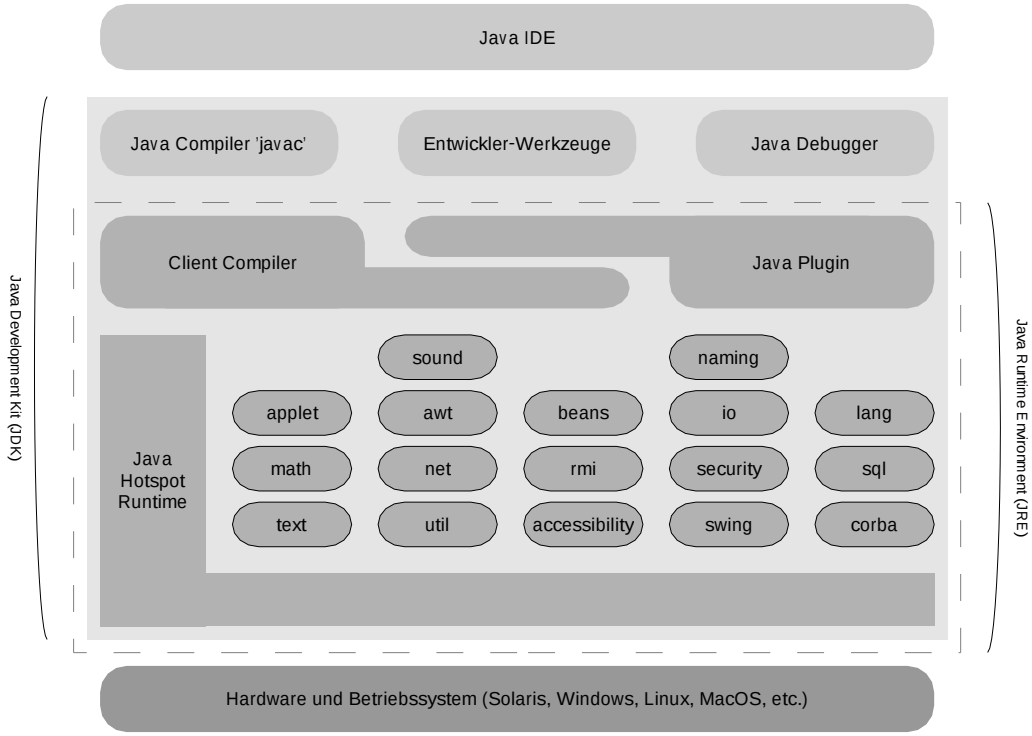
Die Programmiersprache Java

Die Programmiersprache Java – die gemeint ist, wenn man nur den Begriff Java verwendet – nimmt Anleihen bei den Sprachen C, C++ und Smalltalk und stellt damit fortgeschrittenen Programmierer sofort vertraute Bedingungen bereit. Und da viele

der komplizierteren oder umständlichen Eigenschaften dieser Vorgänger nicht in Java übernommen wurden, ist Java für Anfänger leichter zu erlernen.

Anders als C++, in dem der Einsatz der objektorientierten Programmierung (OOP) eine Option ist, forciert Java durch seine Sprach-Syntax das OOP-Paradigma. Die Syntax verpflichtet den Entwickler außerdem zu mehr Selbstdisziplin durch die erzwungene Fehlerbehandlung mit *Exception-Handling.*

Eine ständig wiederkehrende Fehlerquelle in C- und C++-Programmen ist die fehlerhafte Freigabe nicht mehr benötigten, zur Laufzeit reservierten Speichers – und die Suche nach so einem Fehler ist sehr aufwendig und in großen Applikationen manchmal sogar aussichtslos.



Die Komponenten des Java Software Development Kits (JDK), Illustration nach [CWH00].

In Java wird darauf verzichtet, stattdessen übernimmt der *Garbage Collector* die Freigabe von alloziierten Speicherbereichen, wenn die Applikation diese nicht mehr benötigt. Der GC ist für mit C und C++ vertraute Entwickler eine bedeutende Arbeitserleichterung.

Die Binärdarstellung und Breite von Datentypen in Java ist plattformübergreifend standardisiert: Ein `int` ist zum Beispiel auf jeder Hardware, auf dem die Applikation ausgeführt wird, eine vorzeichenbehaftete 32 Bit Zahl in Network Byte Order.

Die Java Standard-Bibliotheken

Der Sprachumfang von Java ist klein und schnell erlernbar. Die Möglichkeiten des Entwicklers sind aber durch die Standard-Bibliotheken vorgegeben, die mit der Java Umgebung mitgeliefert werden.

Sie bieten eine äußerst umfangreiche API für den plattformunabhängigen Zugriff auf das System, zum Beispiel für den Zugriff auf Massenspeicher, für Netzwerkkommunikation oder für die Interaktion mit dem Anwender über eine grafische Benutzeroberfläche.

Die Java Virtual Machine

Ein Java Quelltext wird durch den Compiler in *Java Byte Code* übersetzt. Diesen Binärcode kann die CPU des Zielsystems nicht selbst ausführen, als Interpreter dient die *Java Virtual Machine* (JVM).

Für jede zu unterstützende Rechner-Plattform muss deshalb zunächst eine JVM implementiert werden, die als Abstraktionsschicht zwischen Java Applikation und Hardware steht.

Daraus folgt, dass compilierter Java Byte Code direkt über das Netz von einer JVM zu einer anderen übertragen und dort ausgeführt werden kann, ohne Rücksicht auf die Plattform, auf der die JVM arbeitet. Zur Verhinderung der Ausführung von Schadensroutinen durch fremden Code dient deshalb das *Sandbox*-Konzept der JVM.

Dem auszuführenden Programm kann so der Zugriff auf die unterliegende Hardware eingeschränkt werden, wobei dies in feinen Stufen für unterschiedliche Zugriffsmöglichkeiten kontrollierbar ist. Z. B. Lese- und Schreibrechte auf einzelne Verzeichnisse oder Erlaubnis der Netzwerkkommunkation mit externen Servern.

Mobiler Java Byte Code kann durch den Autor digital signiert werden. Die Vorgaben an die Sicherheitsbeschränkungen der Sandbox lassen sich anhand der Authentifizierung durch solche Signaturen automatisch steuern; so dass man beispielsweise dem externen Code eines bestimmten, vertrauenswürdigen Herstellers grundsätzlich mehr Zugriff auf das System gewähren kann.

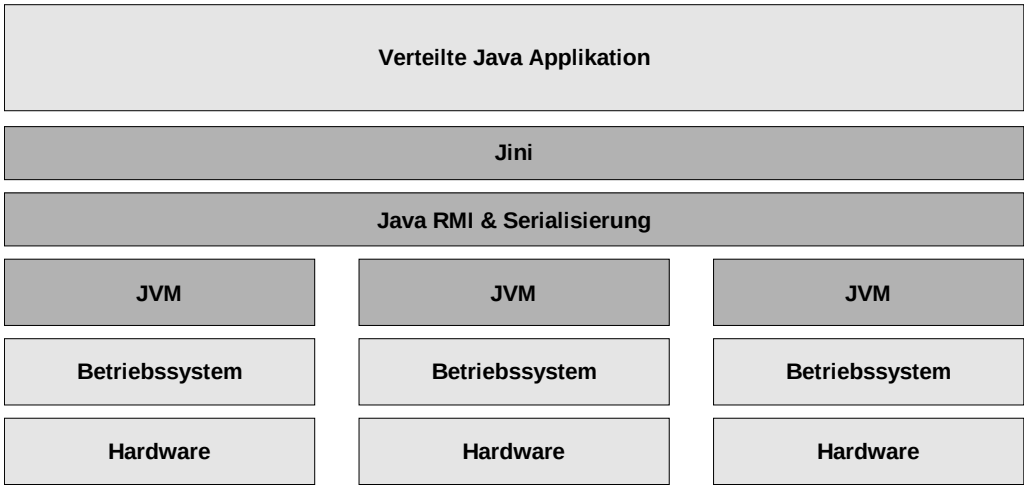
Die JVM ist nicht abhängig von der Programmiersprache Java, es existieren alternative Compiler, die zum Beispiel Python oder C in Java Byte Code übersetzen. Trotzdem ist die Java Umgebung so stark an die in Java implementierten Standard-Bibliotheken gebunden, dass der Einsatz von Java die erste Wahl bleibt und die alternativen Sprachen nur eine Nebenrolle spielen.

Implementationen der JVM existieren für zahlreiche Prozessoren, trotzdem gibt es bislang noch keine komplette JVM „in silicon“. Ein CPU-Prototyp von Sun kann nur einige JVM-Befehle direkt ausführen, bisher wird die virtuelle Maschine noch immer als Softwarelösung implementiert.

Als Interpreter ist die JVM konzeptbedingt grundsätzlich langsamer als Code, der direkt für die Zielplattform übersetzt wurde. Ein Weg zur Beschleunigung ist das *HotSpot* Verfahren, das zur Laufzeit häufig ausgeführte Code-Fragmente (sogenannte HotSpots) erkennt und den Java Byte Code wiederum in optimierten Binärcode für die Ziel-CPU übersetzt.

2.6.3 Die Java Umgebung aus Middleware-Sichtweise

Die Java Umgebung soll nun mit den zuvor erläuterten Eigenschaften und Werkzeugen einer Middleware verglichen werden; dabei werden Sprachelemente und Komponenten gezeigt, die zur Implementation dieser Eigenschaften dienen.



Die Java Systemumgebung dient in einer verteilten Java Applikation als Middleware-Schicht.

Datenrepräsentation und -übertragung in heterogenen Netzen

Die Plattformunabhängigkeit ist durch die Verwendung von Java Byte Code, JVM und standardisierte Binär-Darstellung der Datentypen gegeben. Die Java Systemumgebung macht alle beteiligten Zielplattformen im heterogenen Netz für die Applikation „gleich“ und dient so als zusätzliche Abstraktionsschicht zwischen Hardware und verteilter Applikation. (Und befindet sich damit genau an der Stelle, an der zuvor bereits bei der Definition des Begriffs die Middleware-Schicht positioniert wurde).

Serialisierung und Code-Migration

Zusammengesetzte Datentypen sind im objektorientierten Java als Objekte zu implementieren. Der Java Compiler übersetzt jede Klasse einer Applikation in ein eigenes Code-Fragment und schreibt den dazugehörigen Binärcode in eine individuelle Datei mit dem Namen der Klasse.

Für die Markierung serialisierbarer Klassen bietet Java eigene Sprachelemente und der Compiler fügt automatisch die dazu intern benötigten Schreib- und Lesemethoden hinzu. Gleichzeitig vergibt der Compiler jeder Klasse eine eindeutige ID, die entweder aus dem Quelltext errechnet oder vom Entwickler der Klasse vorgegeben wird.

Wenn die JVM zur Laufzeit ein Objekt deserialisieren soll, dessen ID sie noch nicht kennt, wird sie ihren *Classpath* prüfen. Der Klassenpfad enthält eine Liste aller Verzeichnisse, in denen die JVM vorcompilierte Klassen finden kann. Er darf auch

URLs enthalten und erlaubt somit das Nachladen von externen Code-Fragmenten aus entfernten Servern.

RMI und Dienstbeschreibung

Der Java-Syntax erlaubt es mit dem `Interface` Schlüsselwort, eine Schnittstelle unabhängig von der Implementation einer Klasse zu definieren. Verschiedene Klassen können diese Schnittstelle später auf unterschiedliche Weise realisieren, für eine aufrufende Applikation sind diese unterschiedlichen Implementation transparent austauschbar.

Objektorientierte Weitervererbung ist auch für ein Java-Interface erlaubt, so dass abwärts-kompatible Weiterentwicklungen einer Schnittstelle möglich ist. Anders als das bereits erwähnte CORBA benötigt die Java Umgebung somit keine getrennte Interface Definition Language, die IDL ist in der Java Sprachsyntax selbst bereits enthalten.

Für die Unterstützung von Remote Method Invocation benötigt Java die Hilfe des RMI-Compilers, der aus einer Interface-Definition automatisch zwei Klassen generiert [Phi98]. Die *Skeleton*-Klasse liegt lokal vor und bietet das Gerüst, dem der Entwickler die Implementation hinzufügt. Die *Stub*-Klasse dient auf dem entfernten Prozessor als Stellvertreter des lokal ausgeführten Objekts. Die Methodenaufrufe zur Kommunikation zwischen Skeleton und Stub erzeugt der RMI-Compiler automatisch und transparent für den Entwickler [Sun98].

Namensdienst und Dienstevermittlung

Als Namensdienst in der Java Umgebung dient die *RMI Registry*. Dienstleister können sich hier mit einem selbstgewählten Namen anmelden, über den ihn Dienstleister später finden können.

Die Java RMI Registry stellt jedoch keinen Vermittlungsdienst wie zuvor beschrieben dar. Ad Hoc Vernetzung ist ebenfalls nicht möglich, alle beteiligen Parteien müssen die Adressen von RMI Registry und Dienstleistern bereits kennen.

Eine entsprechende Erweiterung der Java Umgebung liefert erst Jini, dessen Konzept und Komponenten später dargestellt werden.

2.6.4 Probleme der Java Umgebung

Jede zusätzliche Abstraktionsschicht in einem System bedeutet zusätzliche Kosten. Im Falle von Java erscheinen diese Kosten durch die beschriebenen Vorteile der Umgebung gerechtfertigt. Trotzdem können einige dieser Zusatzkosten den Einsatz von Java für ein Projekt unmöglich machen:

Platzbedarf

Die JVM, die vorcompilierten Standard-Bibliotheken, RMI Registry und weitere Standardkomponenten einer Java Laufzeitumgebung benötigen zusammen viel Platz auf dem Massenspeicher des Zielsystems.

Die Java Runtime Edition 2 v1.2.2 für intel-basierte Linux-Systeme belegt beispielsweise 30 Megabyte, von denen etwa die Hälfte auf die komprimierten Java Bi-

bliotheken und -Standard-Applikationen fällt. Unkomprimiert füllen die Bibliotheken ihrerseits 26 Megabyte.

Speicherbedarf

Der Garbage Collector vereinfacht zwar die Programmierung enorm, sorgt jedoch auch dafür, dass dynamisch reservierte Speicherbereiche erst sehr viel später als möglich freigegeben werden. Zusammen mit den zahlreichen Klassen aus den Bibliotheken, die eine JVM im Speicher halten muss, ergibt sich für eine JVM zur Laufzeit ein enormer Speicherbedarf.

Rechenleistung

Wie zuvor dargestellt, ist die interpretierte Ausführung der Applikation durch die JVM langsamer als direkt für die Zielplattform compilierter Binärcode. Durch den Speicherbedarf der JVM wird dieses Problem noch verschlimmert.

Die Hot-Spot Methode konnte dies zwar bereits stark verbessern, doch trotzdem gilt, dass eine Java Applikation für die gleiche Aufgabe weit mehr Rechenleistung verlangt als ein vergleichbares Programm in C. Sogar interpretierte Skriptsprachen wie Perl oder Python können eine Java Applikation überholen, da hier zahlreiche Standardaufgaben wie das Sortieren von Daten bereits als schnelle, für die CPU compilierte Methoden fertig übersetzt vorliegen, die die JVM erst noch in Java Byte Code interpretieren muss.

Komplexe Implementation

Java wurde von Sun als plattformunabhängige Umgebung lanciert. Doch tatsächlich ist die Aufgabe der JVM so umfangreich und komplex, dass bisher nur wenige Rechnerarchitekturen ausreichend und fehlerfrei unterstützt werden. Sun selbst bietet seine eigene Java-Implementation derzeit nur für drei Betriebssysteme (Solaris, Linux und Windows) auf zwei Prozessoren (Sparc und Intel) an.

2.6.5 Ein reduziertes Java

Um den genannten Problemen der Java Systemumgebung – zu groß, zu langsam, zu hoher Resourcenbedarf – zu begegnen, stellte Sun von 1997 bis 2000 in kurzer Folge mehrere Konzepte für Embedded Java Lösungen vor, die schließlich in der *Java 2 Micro Edition* (J2ME) mündeten [Sun00-1].

Für diesen Zweck wurden die drei Komponenten der Java Systemumgebung jeweils aufwärtskompatibel vereinfacht und in ihrem Umfang reduziert.

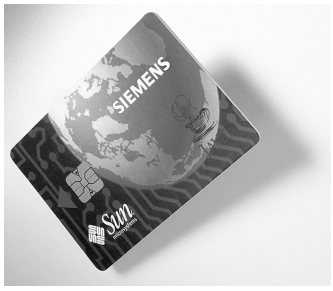
- Der Syntax der Programmiersprache Java ist gleich geblieben, einige Schlüsselworte wurden gestrichen.
- Die reduzierte KVM Spezifikation beschreibt die Mindestvoraussetzung, die eine virtuelle Maschine für die J2ME Umgebung erfüllen muss. Das „K“ steht für Kilobyte und beschreibt die Größenordnung des für die KVM erwarteten Speicherbedarfs zur Laufzeit (im Gegensatz zu den Megabytes einer Standard-JVM).

Der iButton *der Firma Dallas Semiconductors (2000) ist eine Embedded Java Hardware-Lösung im Knopf-Format, hier verarbeitet zu einem Ring. Er bietet je nach Ausführung bis zu 64 KByte ROM und 134 KByte RAM für die auszuführende Applikation und verwendet eine stark eingeschränkte JVM.*

Die KVM verzichtet unter anderem auf einige Basistypen, verwendet eine vereinfachte interne Speicherverwaltung und einen weniger leistungsfähigen Garbage Collector.

- Die Systembibliotheken der J2ME sind in ihrem Leistungsumfang stark reduziert. So wurden zahlreiche bequeme, aber triviale und deshalb nicht unbedingt notwendige Methoden der Basisklassen entfernt, deren Aufgabe ein Entwickler nun bei Bedarf selbst implementieren muss. Die Fehlerbehandlung wurde vereinfacht.

Ein bedeutender Schnitt der J2ME Spezifikation stellt der Verzicht auf Java RMI und Serialisierung dar, der alle drei Komponenten der Systemumgebung betrifft. Da die im nächsten Kapitel besprochene Jini Middleware Umgebung grundlegend auf diesen Technologien basiert, ist es zunächst nicht möglich, eine Jini Applikation direkt in J2ME zu implementieren.



Die Java Smart Card Spezifikation definiert eine reduzierte Embedded Java Variante für den Einsatz in Kreditkarten. Hier ein Prototyp der Firma Siemens (1998)

Einen auch für J2ME möglichen Lösungsansatz wird das übernächste Kapitel vorstellen.

Kapitel 3

Kommerzielle Ad-Hoc Netze im Vergleich

Mehrere Vorschläge bemühen sich darum, Ad-Hoc Vernetzung zu realisieren. Im Folgenden soll speziell untersucht werden, in wie weit die jeweiligen Middleware-Konzepte hinter *Jini*, *Universal Plug and Play* und *Bluetooth* einen Verzicht auf Treiber möglich machen.

3.1 Jini

Die Entwicklung von Jini begann 1997 als internes, experimentelles Projekt bei Sun. Ziel war es, das einst mit „Oak“ ursprünglich angestrebte Einsatzgebiet des Java-Systems in Haushalts- und Unterhaltungselektronik näher zu erforschen.

Das Jini Logo: Die Wunderlampe.

1999 wurde Jini der Öffentlichkeit vorgestellt und von Sun bereits zur Veröffentlichung der ersten Beta-Version auf Messen als marktreife Technologie für den vernetzten Haushalt präsentiert. Der Name ist kein Akronym, sondern ein Wortspiel mit „Java“ und dem englischen Wort „Genie“, dem Flaschengeist aus den Sagen aus 1000 und 1 Nacht; das Jini Logo zeigt eine Wunderlampe.

Jini war eines der ersten größeren Projekte, für das Sun im Rahmen des Marketings den *Open Source* Ansatz wählte, um die Verbreitung der Technologie zu sichern. Es wurde eine *Jini Community* ins Leben gerufen, die unabhängig von Sun entscheiden kann (größtenteils jedoch noch immer aus Sun-Mitarbeiten besteht). Die *Sun Community Source License* [Sun00-2] erlaubt jedem Zugriff auf die Quelltexte von Jini,

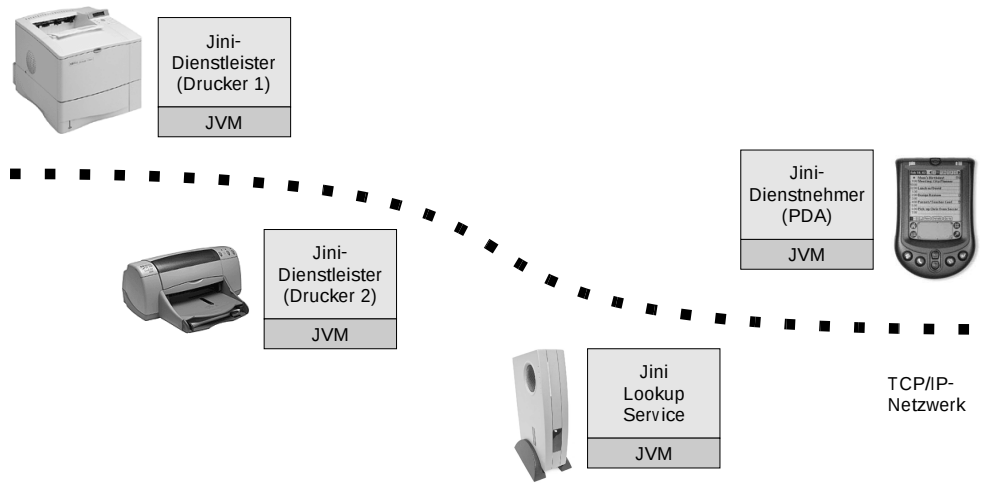
verlangt jedoch Lizenzen für den Einsatz der Technologie in kommerziellen Projekten.

3.1.1 Einführung in Jini

Jini greift in erster Linie auf das bereits im vorherigen Kapitel beschriebene Middleware-Fundament von Java zurück. Die Komponenten zur Dienstbeschreibung und zur Kommunikation zwischen Dienstleister und -nehmer sind damit durch die Java Systemumgebung bereits vorhanden.

Darauf aufbauend werden durch Jini die noch fehlenden Komponenten zur Dienstsuche und -vermittlung eingeführt und ein Programmiermodell zur Behandlung *entfernter Ereignisse* innerhalb der verteilten Applikation zur Verfügung gestellt.

Ein eigenes Protokoll dient zur Suche nach dem Dienstvermittler im lokalen Netz und erlaubt somit die Ad Hoc Vernetzung der beteiligten Geräte. Das *Leasing*-Konzept in Jini soll die Verwaltung der Diestnehmer- / Dienstnutzer-Beziehung vereinfachen und eine „Selbstheilung“ des Netzes bei Ausfall einzelner Knoten sichern.



An einem Jini-Netzwerk beteiligte Geräte benötigen eine JVM und einen Anschluss an ein TCP/IP Netzwerk. Zur Vermittlung der Dienstleistungen zwischen Jini-fähigen Geräten dient der Lookup Service, selbst ebenfalls ein Jini Dienstleister. Er kann auf einem dedizierten Rechner im Netz arbeiten oder zusätzlich in einem anderen Gerät integriert sein, etwa in einem Drucker.

3.1.2 Voraussetzungen für den Einsatz

Folgende Voraussetzungen gelten für den Einsatz von Jini:

Die Geräte verwenden TCP/IP. Die Basisprotokolle, die fest in den Quelltexten der Kern-Klassen von Jini verankert sind, wurden bislang nur für Netze auf Basis von TCP/IP spezifiziert. Sun sieht in der Unterstützung anderer Kommunikationswege zwar kein Hindernis, hat selbst jedoch noch keine anderen Protokolle implemen-

tiert und auch nicht die dazu nötigen Schnittstellen in den Kern-Klassen vorbereitet.

Das TCP/IP Netz ist installiert. Die Adressen der einzelnen Geräte sind bereits zugewiesen, ein funktionierendes Gateway und ein DNS-Dienst sind bereits aktiv – die Ad Hoc Vernetzung von Jini beginnt erst eine Schicht darüber. Sun schlägt deshalb als Bootstrapping für das eigentliche TCP/IP Netz die Verwendung von DHCP vor.

Die Geräte enthalten eine JVM. Zur Ausführung der Basisprotokolle ist Java RMI notwendig, was wiederum nur eine Java Virtual Machine mit Zugriff auf alle Bibliotheken leisten kann. Ein vollwertiges Jini-fähiges Gerät wird deshalb selbst eine JVM enthalten.

Ist dies nicht der Fall, müssen andere JVMs aus dem Netzwerk in Vertretung für dieses Gerät agieren. In den folgenden Kapiteln wird ein solcher Ansatz beschrieben.

Sun bezeichnet Hardware, die diese Voraussetzung nicht erfüllen kann, als „Legacy Devices“, also als veraltete Technologie. Damit wird impliziert, dass solche Produkte mittelfristig vom Markt verschwinden werden, und man sich als Jini Entwickler deshalb nicht mit ihnen beschäftigen muss.

3.1.3 Dienstbeschreibung und -attribute in Jini

Als Interface Definition Language für Jini dient die Java Programmiersprache selbst. Wie bereits zuvor beschrieben, erlaubt das Java-Schlüsselwort `Interface` die implementations-unabhängige Beschreibung eines Dienstleisters.

Zur Darstellung von *Attributen* einer spezifischen Instanz eines Dienstleisters dienen beliebige serialisierbare Java-Objekte. Es ist damit gleichzeitig möglich, serialisierte *Java-Applets* oder GUI-Elemente als Attribute einem Dienstleister zuzuordnen – ein Jini-Gerät kann damit seine eigene Benutzeroberfläche mitführen, ohne dass ein nutzendes Jini-Terminal diesen Dienstleistertyp zuvor kennen muss.

Sun definiert Standard-Attribute, z. B. `net.jini.lookup.entry.Name` für einen Namen des Dienstleisters, `net.jini.lookup.entry.Address` für eine Postadresse oder `net.jini.lookup.entry.Location` für die Position eines Dienstleisters innerhalb eines Gebäudes. Ansonsten ist die Wahl der gewünschten Attribute dem Entwickler frei überlassen.

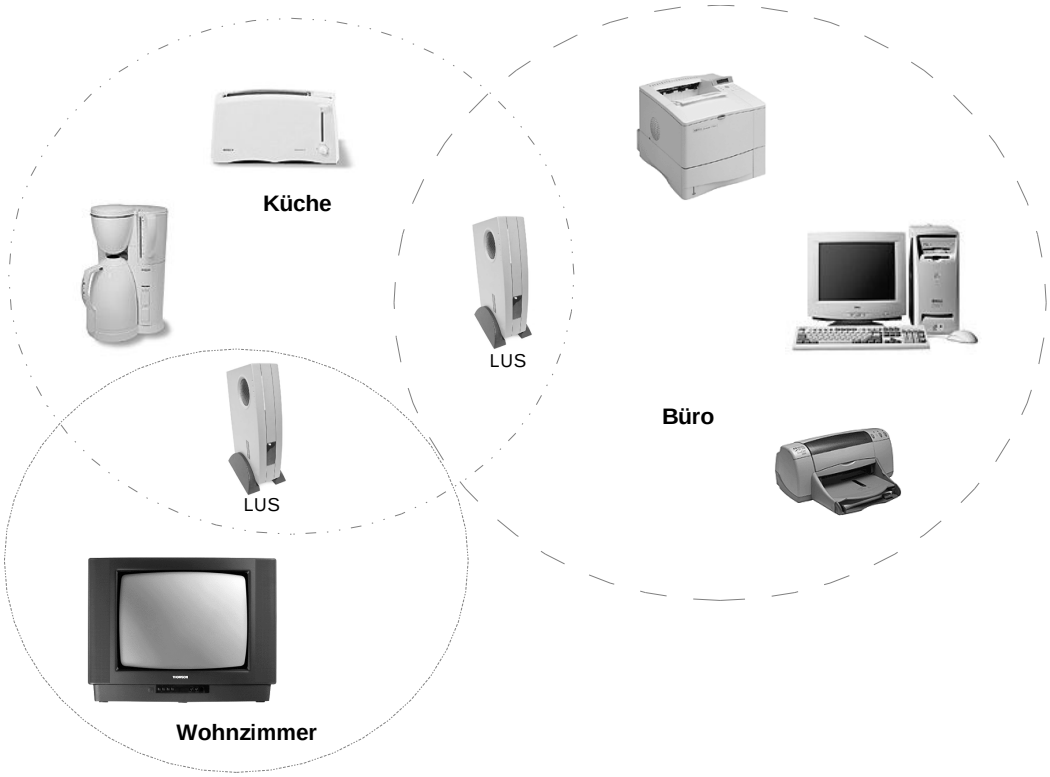
Sun rechnet damit, dass sich die verschiedenen Hersteller einzelner Geräteklassen auf Standard-Dienstbeschreibungen und dazugehörige Attribut-Typen einigen werden. Jini steht an dieser Stelle an einem „Henne und Ei“-Problem, denn solange diese Standards nicht existieren, bleibt das Interesse der Hersteller gering, ihre Produkte mit dieser Technologie auszurüsten.

In Zusammenarbeit mit den Mitgliedern der Jini Community entstanden deshalb eine Reihe von Standard-Dienstbeschreibungen, darunter für Drucker und Massenspeicher.

3.1.4 Partitionierung in Namensräume

Eine Netzgruppe aus mehreren Jini-fähigen Netzknoten wird als „Jini Community“ oder auch – wieder in Anlehnung an den Flaschengeist – als „Djinn“ bezeichnet.

Eine solche Jini-Netzgruppe trägt einen eindeutigen Namen, die der Anwender der verteilten Applikation frei wählen kann, zum Beispiel „Wohnzimmer“, „Küche“ und „Büro“. Dienstleister melden sich für eine oder mehrere Netzgruppen an und eine Suche durch Dienstnehmer kann auf einzelne Gruppen beschränkt werden.



Ein Jini LUS kann mehrere Namensräume verwalten und mehrere LUS können für einen Namensraum zuständig sein.

3.1.5 Dienstsuche und -vermittlung, Leasing

Die Teilnehmer an einer Jini-Netzgruppe stehen unter der Verwaltung des *Lookup Service* (LUS), er übernimmt in Jini die Rolle des Traders. Er selbst ist ein Jini-Dienstleister, dessen Java Schnittstelle durch Sun in der Jini-Spezifikation in `net.jini.core.lookup.ServiceRegistrar` vorgegeben wird.

Sun liefert zusammen mit Jini eine eigene Beispiel-Implementation des LUS namens „Reggie“ mit, die intern auf JavaSpaces als Repository aufbaut [Sun00-3]. Anderen Entwicklern ist es freigestellt, eigene LUS-Implementationen auf Basis anderer Object-Stores zu verwenden.

Zur Anmeldung beim LUS übergibt der Dienstleister:

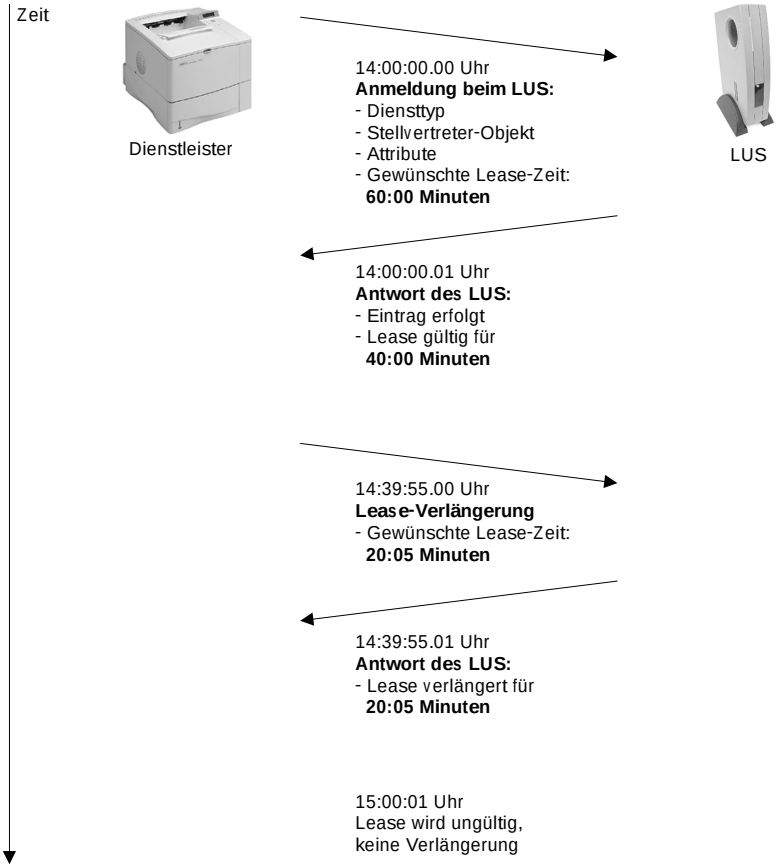
- Die Dienstbeschreibung in Form eines Java-Interfaces.
- Den Verweis auf ein *Stellvertreter*-Objekt (RMI-Proxy), das im Rahmen des RMI-Protokolls auf Seite des Dienstnehmerns die Kommunikation mit dem Dienstleis-

ter kapselt.

- Eine beliebige Zahl von Attribut-Objektinstanzen.
- Die Dauer der gewünschten *Lease*-Zeit.

Der LUS speichert serialisierte Instanzen dieser Objekte in seinem Object-Store und antwortet dem Dienstleister mit einer eindeutigen ID und der durch den LUS gewährten Lease-Zeit, die kleiner oder gleich der beantragten Zeit ist.

Es ist Aufgabe des Dienstnehmers, vor Ablauf dieser Zeit sein Lease zu verlängern. Bleibt diese Verlängerung aus, entfernt der LUS die Informationen über den Dienstleister aus seinem Object-Store und der Eintrag wird damit ungültig.



Nach Anmeldung im Lookup Service wird ein Lease vergeben, das vom Gerät gegebenenfalls während der Laufzeit des Dienstleisters ständig verlängert werden muss.

Sun sieht hierin einen Weg zur Selbstheilung der verteilten Anwendung, da der LUS einen defekten Netzknoten automatisch entfernt und bei Wiederherstellung der Verbindung der betroffene Dienstleister selbst für eine Neuanmeldung sorgt.

Bei der Suche nach einem passenden Dienstleister stellt die suchende Partei ihre Anfrage an einen oder mehrere lokal erreichbare LUS. Die Suchanfrage besteht aus:

- Einer Dienstbeschreibung des gesuchten Dienstleisters in Form eines Java Interfaces.

- Einer Liste der Namen der zu durchsuchenden Netzgruppen.
- Einer Liste von Attributen, die die Suche einschränken sollen, zum Beispiel zur Suche nach Dienstleistern eines bestimmten Herstellers.

Attribute lassen nur einen Vergleich auf Gleichheit von Werten zu, die Jini-Spezifikation sieht keine weitergehenden Vergleiche von Attribut-Werten wie „kleiner als“ vor.

Der LUS liefert – je nach gewünschtem Ergebniskontext – einen oder mehrere der gefundenen Dienstleister zurück. Der Dienstnehmer erhält vom LUS eine serialisierte Instanz eines RMI-Stellvertreter-Objekts und kann darüber die Kommunikation mit dem Dienstleister aufnehmen.

Es bleibt dem Entwickler des Dienstleisters frei, auch für die Beziehung zwischen Dienstleister und Dienstnehmer das Leasing-Verfahren zu verwenden.

3.1.6 Verteilte Events, Leasing

Zur asynchronen Kommunikation zwischen verteilten Knoten der Applikation bietet Jini ein eigenes, einfaches Konzept der *verteilten Events* an.

Ein Java-Objekt, das als Event-Empfänger das Interface `net.jini.core.event.RemoteEventListener` implementiert, bietet fortan eine Methode `notify`, die das sendende Objekt via Java RMI aufgerufen kann. Durch die Verwendung eines eigenen Threads bei der Ausführung der `notify` Methode ist die Asynchronität gewährleistet.

Event-Empfänger können sich bei Sendern registrieren und dabei ihr Interesse für spezifische Event-Typen äußern. Als Ergebnis einer Registrierung erhält der Empfänger erneut ein Lease vom Sender, das er vor Ablauf der Leasing-Zeit erneuern muss.

3.1.7 Ad Hoc Vernetzung in Jini, Discovery Protokoll

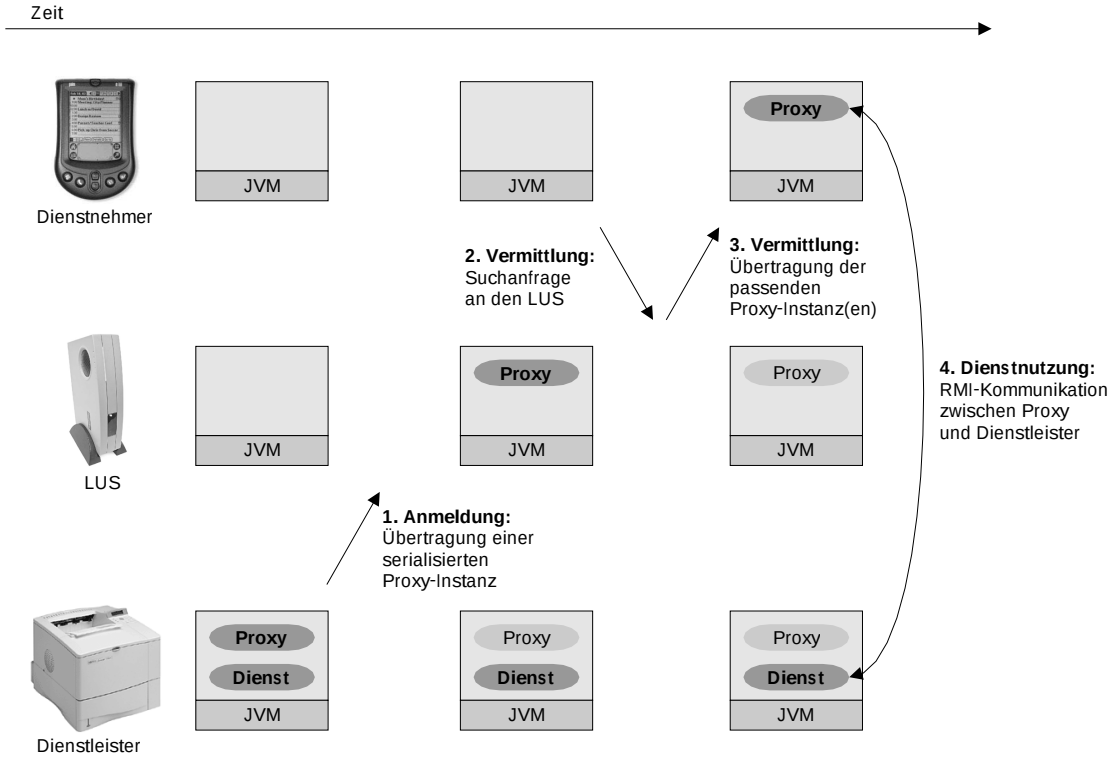
Zwei der Voraussetzungen für Ad Hoc Vernetzung sind für Jini bereits genannt worden: Die Zuständigkeit für die Vergabe einer *eindeutigen Netzwerk-Adresse* gibt Sun an andere ab, schlägt aber DHCP als adäquaten Lösungsweg vor; als *Namensdienst* dienen DNS und Java RMI-Daemon.

Zur *Suche des Vermittlungsdienstes* dient in Jini das *Discovery* Protokoll. Es basiert auf TCP/IP Multicast Kommunkation und wird im folgenden Kapitel im Detail vorgestellt.

Mit Hilfe dieses Basisprotokolls können Jini-Teilnehmer alle LUS im lokalen Netz finden, ohne dass ein Administrator-Eingriff notwendig wäre. Hinzukommende LUS-Instanzen im Netz werden automatisch im Djinn integriert, ausgefallene LUS automatisch erkannt und entfernt.

3.1.8 Stellvertreter-Objekte als Treiberprogramme

Nach der erfolgreichen Vermittlung von Dienstleistern an Dienstnehmer durch den LUS wird dem Dienstnehmer eine serialisierte Instanz eines Stellvertreter-Objekts des Dienstleisters übergeben.



Der Lookup Service legt das serialisierte Stellvertreter-Objekt des Dienstleisters in einem lokalen Objectstore ab und stellt es suchenden Dienstnehmern auf Anfrage zur Verfügung.

Der Code dieses Stellvertreters wird vom Prozessor des Dienstnehmers ausgeführt. Im Regelfall wird der Stellvertreter ein einfacher RMI-Proxy ohne jegliche Eigenintelligenz sein, der für jeden lokalen Methoden-Aufruf lediglich die Kommunikation mit der entfernten Dienstleister-Instanz abwickelt, dies bedeutet keine signifikanten Zusatzkosten bei der Ausführung für die Dienstnehmer-CPU.

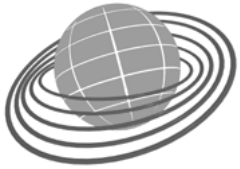
Der Entwickler des Dienstleisters kann aber auch Teilfunktionen des Dienstleisters schon im Stellvertreter-Objekt implementieren. Je mehr die Dienstnehmer-CPU an zusätzlicher Arbeit für die Vorbereitung von Parametern und die Nachbereitung von Ergebniswerten aufbringen muss, desto mehr gleicht das Stellvertreter-Objekt in Jini einem klassischen *Geräte-Treiberprogramm*.

Mit Blick auf Ziel-Geräte wie Drucker, Kameras, Scanner kann man Jini deshalb auch als eine Infrastruktur bezeichnen, die es einem Hersteller ermöglicht, *den Geräte-Treiber gleich im Gerät unterzubringen*.

Innerhalb von Jini bringt ein Gerät so seine Dienstbeschreibung, seinen Treiber und gegebenenfalls auch die dazugehörige Benutzeroberfläche bereits mit und installiert diese bei Bedarf automatisch auf dem Anwender-Terminal – ohne dass die Hilfe eines Administrators notwendig wäre.

3.2 Universal Plug and Play

Microsoft stellte Ende 1998 als Antwort auf Jini die Universal Plug and Play Technologie (UPnP) vor. Das Projekt entstand in enger Kooperation mit Hewlett-Packard und Intel und wurde bald darauf von den Partnern als direkte Konkurrenz zur Sun-Technologie platziert.



Microsoft beruft sich in den eigenen Unterlagen zu UPnP bei der Namensgebung direkt auf die *Plug and Play* genannte Technologie des eigenen Betriebssystems Windows 95 [MS01].

Tatsächlich hat UPnP aus technischer Sicht hiermit keine Gemeinsamkeiten: Windows PnP besteht im Wesentlichen aus einer vorgeschriebenen Geräte-Kennung für in einen Computer einzubauende Hardware und einer Treiberdatenbank zur Suche nach der richtigen Treiber-Software mit anschließender, weitgehend manueller Installation.

UPnP soll dagegen nicht die Treibersuche für neue Hardware im Computer vereinfachen, sondern die Kommunikation zwischen Geräten im Netzwerk standardisieren, und das unter „Verzicht auf Gerätetreiber, stattdessen werden allgemeine Protokolle verwendet“ [MS00-1].

Ebenso wie Sun bemüht man sich um einen möglichst offenen Entwicklungsprozess, hier unter der Leitung des UPnP Forums, eines formal von den ursprünglichen UPnP-Entwicklerfirmen unabhängigen Komitees, dem Firmen ohne Mitgliedsgebühren beitreten können. Im UPnP Forum besitzt Microsoft allerdings einige Sonderrechte und -pflichten, die den direkten Einfluss auf die weitere Entwicklung sichern.

UPnP setzt auf vorhandene offene Standards auf, darüber hinaus notwendige, eigens entwickelte Protokolle und Standards wurden in Form von RFC-Drafts offengelegt.

3.2.1 Einführung in UPnP

Die Zielsetzung von Jini und UPnP ist im Wesentlichen gleich: Ad Hoc Vernetzung von Geräten in einem lokalen Netz unter Verzicht auf die Installation von Treiber-Software durch den Nutzer.

UPnP unterscheidet zwischen *Geräten* und den ihnen zugeordneten *Diensten* und definiert für beide verschiedene Arten der Beschreibung und des Zugriffs; die Details dieser Unterscheidung sind für diese Betrachtung allerdings nebensächlich.

Die Spezifikation [MS00-2] teilt die Dienstnutzung via UPnP in fünf Schritte ein, dazu zusätzlich ein gegebenenfalls notwendiger, vorbereitender Schritt 0

- 0 (Addressing)
- 1 Discovery
- 2 Description
- 3 Control
- 4 Eventing
- 5 Presentation

3.2.2 Anforderungen für den Einsatz

Die UPnP Basisprotokolle bestehen aus dem Austausch einfacher Klartextnachrichten, formuliert in einer XML-konformen Syntax. Zur Übertragung dieser Nachrichten dient HTTP.

Obwohl HTTP ursprünglich für TCP/IP Unicast Verbindungen spezifiziert wurde, lässt UPnP offen, welche Art der Verbindung oder Vernetzung zwischen zwei Geräten besteht. Der Austausch von reinem Klartext lässt sich auf einfache Weise auch auf andere Technologien übertragen, Microsoft hat UPnP beispielsweise auch für den Universal Serial Bus (USB) spezifiziert.

Die Anforderungen an ein UPnP-Gerät sind somit sehr gering: Es muss in Verbindung zur Außenwelt stehen und über diese Verbindung UPnP-konforme Nachrichten entgegennehmen und versenden können. Im Folgenden wird weiterhin TCP/IP als das zu Grunde liegende Netzwerk angenommen.

Anders als Jini geht UPnP nicht von bereits fertig konfigurierten Adressen aus. Geräte ohne Netzwerk-Einstellungen sollen für diesen vorbereitenden, *Adressing* genannten Schritt ein einfaches Verfahren zur dezentralen, automatischen Zuordnung von IP-Adressen im lokalen Netz unter Verzicht auf DHCP [Che00] verwenden.

3.2.3 Suche nach Dienstleistern

Im Gegensatz zu Jini gibt es in einem UPnP Netz keinen dedizierten Trader zur Vermittlung von Dienstleistern. Stattdessen sucht jeder Dienstnehmer (in UPnP auch *Control Point* genannt) während der *Discovery*-Phase über das *Simple Services Discovery Protocol* (SDDP) auf sich allein gestellt nach Dienstleistern.

Allerdings kann ein Dienstleister gemäß Spezifikation in Vertretung für andere Dienstleister antworten: Für entfernte Dienstleister, die keine Nachrichten des lokalen Netzes empfangen können, oder für interne Dienstleister, die beispielsweise via USB nur indirekt an das Netzwerk angeschlossen sind.

Dienstnehmer erfahren auf zwei Arten von Dienstleistern im Netz:

Advertise: Geräte senden bei Neuanschluss an das Netzwerk und danach weiter in regelmäßigen Abständen ein Multicast-Datagramm mit Informationen über ihre(n) Dienst(e). Als Protokoll dient eine für Multicast modifizerte HTTP-Variante [GS00].

Dienstnehmer werten diese Multicast-HTTP Nachrichten aus und fordern bei Interesse via Unicast eine Dienstbeschreibung vom Absender.

Search: Klienten versenden via Multicast-HTTP eine Anfrage nach einem Dienstleister-Typ anhand einer vom UPnP-Forum standardisierten Typenbezeichnung oder nach einem konkreten Dienstleister anhand seiner Kennung.
Alle auf diese Beschreibung passenden Dienstleister antworten dem Dienstnehmer via Unicast und teilen ihm ihre Adresse und ihre konkrete Dienstbeschreibung mit.

Die *Description*-Phase bezeichnet den Empfang und die Auswertung dieser Beschreibungen. Anhand der Dienstbeschreibung entscheidet der Dienstnehmer, ob er den angebotenen Dienst nutzen wird.

3.2.4 Diensttyp, -beschreibung und -Attribute

Die (textuellen) Bezeichner für *Geräte*- und *Diensttypen* werden durch das UPnP Forum definiert. Es folgt hieraus, dass ein Dienstnehmer nur solche Dienstleister-Typen suchen und nutzen kann, die zum Zeitpunkt seiner Entwicklung bekannt waren.

Die Beschreibung der Dienstleistung erfolgt durch den Hersteller des Geräts, sie wird in einer XML-basierten Interface Definition Language im Klartext-Format formuliert, der *UPnP Device Template Language*.

Neben allgemeinen Informationen (Hersteller, Gerätebezeichnung, Versionsnummer, Icon-Bitmap) in der *UPnP Device Description* enthält das dazugehörige *UPnP Device Template* eine Liste aller Dienstleistungen des Geräts, wiederum in Form von *Service Description* und *Service Template*. Diese beschreiben Methoden und Status-Variablen, die im Rahmen der Dienstnutzung zur Anwendung kommen können.

Ein Hersteller kann über die im Standard-Template eines bestimmten Dienstleister-Typs beschriebenen Mindestanforderungen hinaus weitere Funktionalitäten anbieten und an dieser Stelle beschreiben.

3.2.5 Dienstnutzung ohne Treiber

Nach Auswahl des gefundenen Dienstleisters erfolgt die Dienstnutzung, die *Control*-Phase.

Prinzipbedingt kann ein Dienstnehmer nur solche Dienstleister in Anspruch nehmen, deren durch das UPnP Forum vordefinierte UPnP Standard Template zum Zeitpunkt der Entwicklung bereits bekannt war.

Einen *Gerätetreiber* gibt es deshalb in dem Sinne also nicht mehr – stattdessen nur noch eine für diesen Diensttyp vordefinierte Schnittstelle, der ein Hersteller jedoch eigene Erweiterungen hinzufügen kann.

Jeder Entwickler eines Dienstnehmers wird den Zugriff über diese Schnittstelle erneut programmieren müssen. Falls der Dienstleister nur über eine geringe „Eigenintelligenz“ verfügt, kann dies durchaus den gleichen Entwicklungsaufwand wie für einen vollwertigen Treiber bedeuten.

Für den Zugriff auf die Methoden und Status-Variablen des Dienstleisters und den Empfang der Ergebnisse dient das *Simple Object Access Protocol* [W3C00]. SOAP basiert auf einer XML-konformen Syntax und erlaubt die Klartext-Übertragung von RMI-Befehlen und von serialisierten Objekt-Instanzen. Durch SOAP ist es so möglich, einen entfernten Methodenaufruf von einer vom Dienstleister völlig unterschiedlichen Systemumgebung und Programmiersprache aus auszulösen.

3.2.6 Verteilte Events in UPnP

Ähnliche wie in Jini können Geräte einander asynchron mit Hilfe von Events über Statusänderungen informieren. Obwohl die UPnP-Spezifikation der *Eventing*-Phase einen eigenen Schritt 4 zuordnet, werden Events zu jedem Zeitpunkt der Dienstnutzung eingesetzt, nicht erst streng nach der Control-Phase.

Ein Empfänger von Events muss als *Subscriber* zunächst ein Abonnement bei einem Sender anfordern, dem *Publisher*. Die Anmeldung ist zeitlich begrenzt und muss vom Subscriber gegebenfalls verlängert werden, vergleichbar mit Leasing in Jini.

Bei Änderung von Status-Variablen informiert der Publisher nacheinander alle registrierten Subscriber; zur Verringerung der Netzlast können mehrere Statusänderungen in einer einzelnen Nachricht zusammengefasst werden.

Wie zuvor dienen via HTTP übertragene Nachrichten in einer XML-konformen Syntax zur Übermittlung dieser Events.

3.2.7 HTML Schnittstelle zum Dienstleister

Die bisher beschriebenen Schritte ermöglichen die Nutzung von UPnP Dienstleistern durch programmierte Dienstnehmer.

Zusätzlich dazu sieht die *Presentation*-Phase vor, dass Dienstleister eine direkt nutzbare, in HTML formulierte Benutzerschnittstelle anbieten, die von einem üblichen WWW-Browser darstellbar ist. Über diesen Weg soll der Anwender des Geräts die Konfiguration, die Status-Überwachung und gegebenenfalls auch die Dienstnutzung mit seinem Browser erledigen können.

Insofern ist die vorherige Aussage nicht ganz korrekt – auch Geräte, deren UPnP Template noch völlig unbekannt ist, können genutzt werden, wenn sie sich prinzipiell über diese Schnittstelle bedienen lassen. Durch die starken Einschränkungen der Interaktion über HTML wird dies aber nur für wenige Geräte gelten: Ein Drucker kann beispielsweise nicht über diesen Weg den Druckauftrag der Textverarbeitung des Nutzers entgegennehmen.

3.3 Bluetooth

In Erwartung eines kommenden Booms drahtloser Kurzstrecken-Vernetzung in Büro- und Kommunikationsgeräten gründeten die Firmen Ericsson, IBM, Intel, Nokia und Toshiba bereits 1998 die *Bluetooth Special Interest Group*, um frühzeitig und Hersteller übergreifend die dazu notwendigen Standards zu spezifizieren. Unter dem Einfluss der beteiligten skandinavischen Firmen diente Wikingerkönig *Harold Blaatand* (Blauzahn) als Namensgeber, der im 10. Jahrhundert Dänemark und Norwegen einte. Die Implementation der Bluetooth-Technologie ist lizenzfrei.

Das Bluetooth Logo enthält die Initialen des Wikingerkönigs Harold Blaatand in Runenschrift.

Ziel von Bluetooth ist die drahtlose Ad-Hoc Vernetzung von Geräten in unmittelbarer Umgebung des Anwenders im Umkreis von 10 m, wobei mit Hilfe von mehreren vernetzten Basisstationen auch größere Funknetze möglich sind.

Die Entwicklung konzentriert sich speziell auf batteriebetriebende mobile Geräte. Aus diesem Grund sollte die notwendige Funk-Hardware klein, Strom sparend und in Erwartung eines Massenmarktes preisgünstig sein. Aktuelle Ein-Chip Bluetooth-Module haben dieses Ziel mit einer Größe unter 5 cm², einem Stromverbrauch unter 300 mA und erwarteten Stückkosten unter 10 US-Dollar bereits erreicht.

Typische Anwendungsszenarien sind beispielsweise:

- Drahtloser Anschluss von Tastatur und Maus an einen Computer.
- Die lokale Funk-Vernetzung zwischen Computern, Druckern und einem Internet-Router in einem Büro.
- Synchronisation zwischen PDA und Tisch-Rechner und die Datenübertragung zwischen zwei PDAs, etwa die Übertragung einer elektronischen Visitenkarte.
- Die drahtlose Verbindung der Komponenten eines Funktelefons der nächsten Generation: Handy-Modul am Gürtel, Telefon-Tastatur am Armband und Headset am Ohr.
- Einsatz des Funktelefons als Internet-Verbindung für das Laptop.
- Nutzung von Dienstleistungen regionaler Funkzellen, etwa ein öffentlicher Internet-Anschluss in einem Flughafen-Gebäude oder ein Fahrplan-Service in Umgebung einer öffentlichen Haltestelle.

Dabei müssen die beteiligten Geräte nur in Funkreichweite sein, so dass z. B. ein Funktelefon am Gürtel oder in der Jackentasche verbleiben kann, will man mit dem Laptop eine Internet-Verbindung aufnehmen. Genauso ist eine Anwendung denkbar, bei der ein noch im Koffer befindliches Laptop bei Ankunft am Heimat-Flughafen automatisch den dort angebotenen Internet-Anschluss zum Abholen von E-Mails verwendet.

3.3.1 Der Bluetooth Funk-Standard

Die Bluetooth-Funkübertragung arbeitet im 2,45 GHz Frequenzband, das weltweit für zivile Zwecke reserviert ist. Allerdings nutzen bereits andere Anwendungen diesen Bereich, darunter zum Beispiel Garagentor-Steuerungen. Zudem senden Geräte wie Mikrowellenöfen hier Störsignale.

Aus diesem Grund verwendet Bluetooth via Frequenzmultiplexing insgesamt 79 Funkkanäle im Abstand von 1 MHz, zwischen denen zwei miteinander vernetzte Geräte nach einem vorher gemeinsam festgelegten, quasi-zufälligen Muster im Takt von 1600 Hops/s wechseln. Auf diese Weise erreicht Bluetooth eine robuste Übertragung selbst in einem bereits benutzten Frequenzbereich und sichert, dass sich auch mehrere Bluetooth-Geräte nicht gegenseitig stören.

Die Bluetooth Datenübertragung erreicht standardmäßig Datenraten von 1 MBit/s bei einer Reichweite von 10 m, wobei die Spezifikation eine Erweiterung der Sendeleistung auf 100 m definiert. Mehrere Übertragungsmodi existieren für verschiedene Anwendungen, darunter beispielsweise:

- Symmetrische, synchrone Audio-Übertragung (z. B. für Telefonie) mit 64 KBit/s je Richtung.
- Symmetrische asynchrone Daten-Übertragung (z.B. für lokale Vernetzung oder Dateitransfer) mit 432.6 KBit/s je Richtung.
- Asymmetrische asynchrone Daten-Übertragung (z. B. für Web-Browsing) mit 57.6 KBit/s auf dem Hinkanal und 721 KBit/s auf dem Rückkanal.

Auf diese Übertragungs-Technologie bauen mehrere Protokoll-Schichten auf. Das *Link Manager Protocol* (LMP) übernimmt den Auf- und Abbau sowie die Sicherung einer Verbindungen, das *Logical Link and Control Adaption Protocol* (L2CAP) kontrolliert die Datenübertragung auf logischen Kanälen über diese Verbindung.

Über diesen Schichten befinden sich schließlich die Emulationen verschiedener älterer Standard-Protokolle. Einige dieser Bluetooth-Protokolle sind Audio-Übertragung zur Telefonie, Emulation einer seriellen RS232-Schnittstelle, TCP/IP basierend auf PPP oder eine Faxmodem-Emulation unter Verwendung von Hayes-AT-Befehlen.

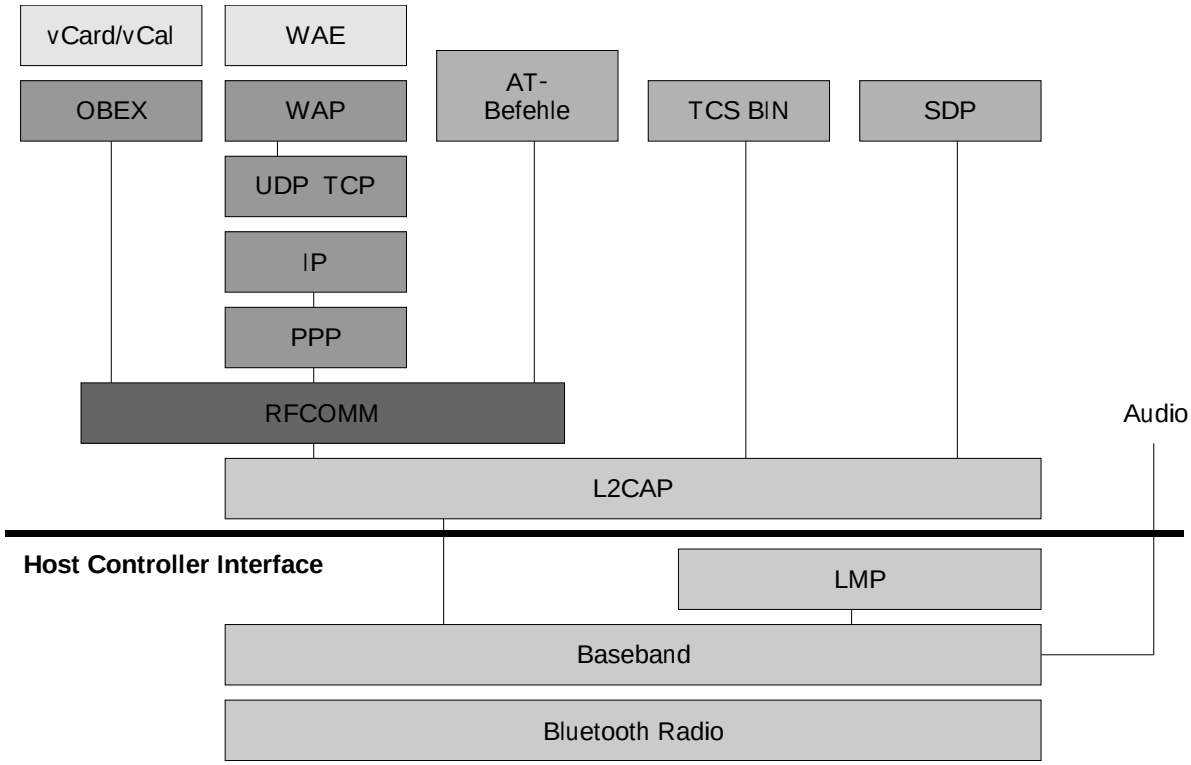
Ein lokaler Verbund aus mehreren Bluetooth-Geräten wird *Piconet* genannt. Eines der Geräte ist der *Master,* der die gemeinsame Verbindung mit bis zu 7 *Slaves* steuert. Alle Geräte eines Piconets verwenden das gleiche Frequenz-Sprung-Muster und teilen sich die vorhandene Bandbreite dieser Verbindung. Ein einzelnes Gerät kann an mehreren Piconets teilnehmen und bildet damit ein *Scatternet,* wobei es in einem Netz als Master und in einem anderen als Slave auftreten kann.

Feste Bestandteile von Bluetooth sind *Authentifizierung* der Geräte und *Autorisierung* ihrer Verbindungswünsche, um den Missbrauch von drahtlos erreichbaren Dienstleistern zu verhindern, zusammen mit einer *Verschlüsselung* der Übertragung gegen das Abhören der Daten durch Dritte.

3.3.2 Dienstleister-Profile

Die Bluetooth Special Interest Group fasst verschiedene Typen von Dienstleistern in *Profilen* zusammen, die teilweise aufeinander aufbauen.

Für ein Profil ist Hersteller übergreifend definiert, welche *Service-Attribute* zur Beschreibung des Dienstleisters zur Verfügung stehen, welche der Bluetooth-Protokolle



das Gerät verwenden darf, und wie die Dienstleistung über diese Protokolle anzusprechen ist. Bisher sind in [Blu01-2] unter anderem spezifiziert:

Serielle Datenübertragung, z. B. für den Zugriff auf einen entfernten Drucker.

LAN-Zugriff auf das im Gebäude installierte Büro-Netzwerk via TCP/IP.

Internet-Wählverbindung über ein (Funk-) Telefon.

Fax-Übertragung über ein entferntes oder gegebenenfalls von einem Funktelefon emuliertes Modem.

Audio-Übertragung zwischen Headset und Telefon oder zwischen mehreren Headsets.

Telefonie zur Verwendung eines entfernten Telefons, so dass z. B. eine Bluetooth-Tastatur am Armband oder die Adressverwaltung im PDA die Telefonnummer wählt.

Dateitransfer zwischen zwei Laptops.

Objekt-Transfer wie der Austausch von elektronischen Visitenkarten zwischen zwei PDAs.

Synchronisierung zwischen zwei Geräten, etwa zwischen PDA und Tisch-Rechner oder zwischen Laptop und File-Server.

3.3.3 Dienstbeschreibung, -Attribute und nutzende Applikation

Bluetooth spezifiziert Zahlencodes für die einzelnen Profile und ihre Attribute und gibt vor, in welchem Datenformat Attributwerte zu übertragen sind.

Eine Dienstbeschreibung und die dazugehörigen Attribute werden damit nicht im Klartext, sondern binär codiert, werden aber von allen Bluetooth-konformen Implementationen in verschiedenen Systemumgebungen verstanden.

Zu den Standard-Attributen gehören neben der Profil-ID, dem Namen des Herstellers, Gerätekennung und Versionsnummer auch:

- Eine Liste von Icons in verschiedenen Formaten und Farbtiefen, z.B. als 1-Bit-Bitmap für Monochrom-Display und mit 8-Bit Farbpalette für Farbdarstellung.
- Eine URL auf die Dokumentation der Dienstleistung im HTML-Format, die im Dienstleister-Gerät intern gespeichert sein oder sich auf dem externen Server des Herstellers befinden kann.
- Eine Liste von *vorcompilierten Anwendungsprogrammen* für verschiedene Betriebssysteme, die der Klient auf das eigene Gerät herunterladen und dort ausführen kann. Dieses Programm kann im Dienstleister-Gerät intern gespeichert sein oder sich auf dem externen Server des Herstellers befinden.

3.3.4 Suche nach Dienstleistern

Das *Service Discovery Protocol* [Blu01-1] ermöglicht die Suche nach Diensten in Sendereichweite, wobei der Dienstnehmer als *SDP Client* und der Dienstleister als *SDP Server* bezeichnet wird. Pro Bluetooth-Gerät ist maximal ein solcher SDP Server für die Vermittlung der eigenen Dienste zuständig, ein reiner Dienstnehmer kann auf den SDP Server verzichten.

SDP-Nachrichten werden direkt über L2CAP-Verbindungen übertragen und enthalten eine Liste der gewünschten Dienst-Typen in Form von Profil-IDs. Geräte, die diesem Profil entsprechen, können mit einer Liste ihrer Attribute antworten.

Der Klient kann in seiner Suchanfrage vorher die gewünschten Attribute angeben und so unnötigen Datentransfer vermeiden, indem er nur die Attribute anfordert, anhand derer er sich später für einen Dienst entscheiden will.

Bestandteil von SDP ist auch das *Service Browsing*, bei dem Dienstleistungen in hierarchischen Gruppen sortiert sind und von einem Anwender manuell durchsucht werden können, ohne vorherige Vorstellung über das gewünschte Profil oder die gesuchten Attribute.

3.3.5 Dienstnutzung

In den Bluetooth-Profilen wird die Schnittstelle zum Dienstleister vorgegeben, so dass der Entwickler eines Dienstnehmers den „Gerätetreiber“ für seine Applikation selbst programmieren muss, falls dieser nicht schon Bestandteil der Bluetooth-Implementation der von ihm gewählten Systemumgebung ist.

Bluetooth definiert kein allgemeingültiges Protokoll für entfernte Methodenaufrufe und Serialisierung, so dass jedes Profil völlig unterschiedliche Protokolle zur Kommunikation mit dem Dienstleister vorgeben kann.

Bemerkenswert ist die Möglichkeit, als Dienst-Attribut eine URL auf eine vorcompilierte Dienstnehmer-Applikationen für verschiedene Betriebssysteme zu nennen. So wird es möglich, neue Dienste von noch unbekannten Profiltypen zu verwenden ohne bereits zuvor die dazu benötigte Applikation zu kennen.

3.4 Fazit des Vergleichs

Über die Definition von allgemeinen Diensttypen und mit Hilfe eines Dienstsuche-Protokolls erlauben alle drei Technologien den Verzicht auf die Installation von Treibern.

Allen gemeinsam ist, dass die Schnittstelle zum Dienstleister zum Zeitpunkt der Entwicklung des Dienstnehmers bekannt sein muss, so dass die jeweiligen Standard-Komitees Diensttypen definieren müssen.

Für den Zugriff auf noch unbekannte Diensttypen bieten sie als Ausweg interaktive Schnittstellen an – Bluetooth in Form von vorcompilierten Programmen für verschiedene Plattformen, UPnP mit einem browser-basierten Zugang zum Gerät. Jini lässt hier die meisten Freiheiten, da es dem Klienten Applets, einen Browser-Zugang oder ebenfalls ganze Applikationen anbieten kann.

Bluetooth konzentriert sich auf die Bereitstellung von Übertragungswegen und bietet kaum Profile für höherwertige Dienstleistungen an. Obwohl dies im Rahmen der Bluetooth-Profile möglich wäre, ist bisher beispielsweise keine allgemeine *Dienstleistung Drucken* vorgesehen, sondern nur die drahtlose *Verbindung* zum Drucker definiert worden. Die Kenntnis des Drucker-Typs und die Installation eines Druckertreibers für den Client-Computer sind in einem Bluetooth-Netz immer noch notwendig.

Sowohl Jini als auch UPnP setzen hier eine Ebene höher an und stehen damit in direkter Konkurrenz zueinander. (Tatsächlich existieren sowohl *Jini over Bluetooth* als auch *UPnP over Bluetooth* Spezifikationen aus den beiden Lagern.)

Beide wollen hochwertige Dienstleistungen wie Drucken, Datenablage auf vernetzten Massenspeichern oder Terminabsprache Hersteller übergreifend standardisieren.

Der herausragende Vorteil von UPnP gegenüber Jini ist die konsequente Verwendung von Klartext-Protokollen wie HTTP und SOAP, die eine Implementation in verschiedensten Systemumgebungen ermöglicht und nur geringe Anforderungen an die Hardware stellt. Andererseits sind die Möglichkeiten der UPnP Dienstbeschreibung stark eingeschränkt und erlauben nur geringfügige Erweiterungen eines Diensttyps.

Demgegenüber bietet der in Jini benutzte Java Interface Syntax die größten Freiheiten bei der konformen Erweiterung vorhandener Diensttypen und -attribute. Jini erscheint damit als die umfassendste und flexibelste Lösung des Vergleichs.

Der bedeutendste Nachteil sind jedoch die mit der Java Systemumgebung verbundenen Kosten von Jini. Zu Beginn dieser Arbeit gab es in der Jini-Spezifikation noch kein konkretes Konzept, wie ein „Legacy Device“ ohne eigene JVM an einem Jini Netz teilnehmen kann. Das folgende Kapitel beschreibt einen Weg, dies zu ermöglichen.

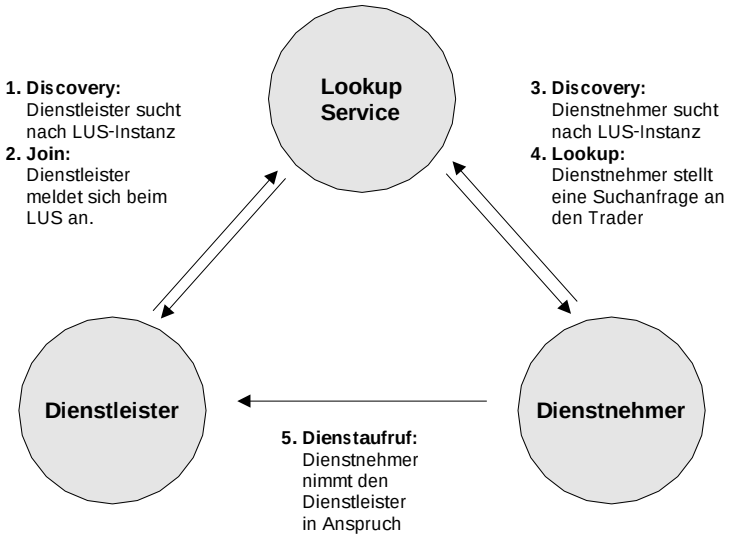
Kapitel 4

Eine Protokoll-Erweiterung für Jini

4.1 Problemstellung und Lösungsweg

Wie vorhergehend bereits beschrieben, ist in Jini ein Dienstleister nur dann in der Lage, an einem Djinn zu partizipieren, wenn er selbst eine vollständige JVM enthält.

Die genauen technischen Gründe dafür liegen im Vorgehen der Jini Basisprotokolle *Discovery*, *Lookup* und *Join*, wie nun im Detail erläutert wird.



Der Lookup Service übernimmt in Jini die Aufgabe des Traders. Die drei Jini Basisprotokolle dienen der Kommunikation mit dem LUS.

Wie sich zeigen wird, kann eine einfache Änderung dieser Protokolle die JVM im Dienstleister unnötig machen und stattdessen die des Lookup Services und des Dienstnehmers mit nutzen. Das modifizierte Protokoll wird im anschließenden Kapitel in einem funktionierenden Gerät ohne Java Systemumgebung implementiert.

4.2 Die Jini Basisprotokolle

Für den zentralen Vorgang der Dienstevermittlung in Jini treten drei Parteien miteinander in Verbindung: Dienstleister, Dienstnehmer und Lookup Service (der in Jini die Funktion des Traders übernimmt). Gemeinsam bilden sie das bereits dargestellte Trader-Dreieck.

4.2.1 Überblick

Ein neu eingeschaltetes Gerät wird zu Beginn den für seinen Djinn zuständigen LUS im lokalen Netz suchen, seine Dienstleistung dort anmelden bzw. eine gewünschte Dienstleistung im LUS-Verzeichnis auswählen und schließlich mit diesem Dienstleister in Verbindung treten.

Zur Kommunikation zwischen Dienstleister und -nehmer dient die RMI Schnittstelle der Java Systemumgebung. Hierfür greift der Dienstnehmer auf einen RMI-Proxy des Dienstleisters zurück, den er vor Dienstnutzung durch Vermittlung über den LUS erhält.

Bevor es zu einer Vermittlung des RMI-Proxy-Objekts kommen kann, müssen zunächst beide Parteien den LUS im lokale Netz finden und der Dienstleister seinen RMI-Proxy in dessen Verzeichnis anmelden.

Hierfür dienen die drei Jini Basisprotokolle [Sun99-2] Discovery, Lookup und Join.

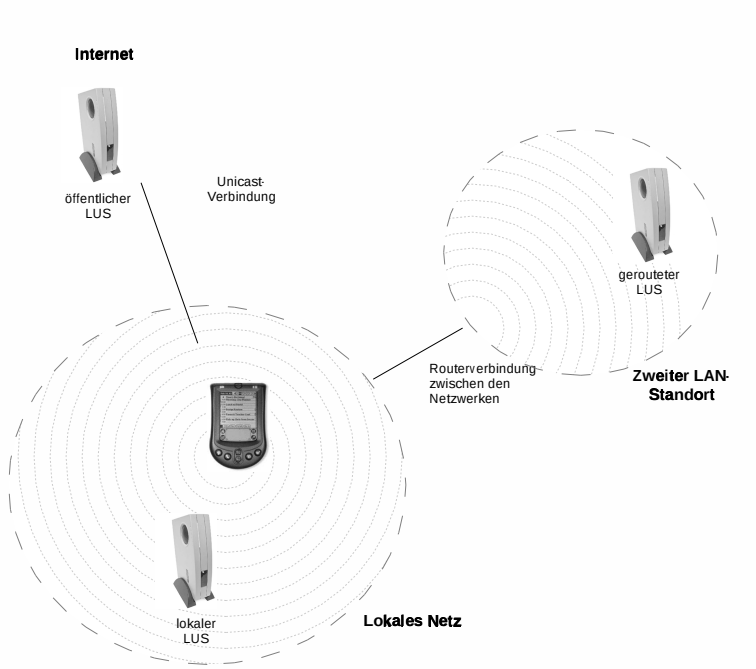
4.2.2 Discovery: Auffinden des Jini LUS

Zur Teilnahme an einem Djinn muss sich ein Jini-Gerät zunächst bei einem (oder – soweit vorhanden – mehreren) für diesen Djinn zuständigen LUS anmelden. Dieser LUS ist über ein TCP/IP-Netzwerk erreichbar.

Ist die IP-Adresse noch unbekannt, erlauben die beiden Protokolle *Multicast Request* und *Multicast Announce* das Auffinden von noch unbekannten LUS-Instanzen im lokalen Netz.

Zur Kontaktaufnahme mit einem LUS mit bereits bekannter IP-Adresse dient das *Unicast Discovery* Protokoll. Es erlaubt auch die Anmeldung von Dienstleistern in LUS-Instanzen, die sich außerhalb der Multicast-Reichweite des lokalen Netzes befinden, falls es etwa einen konzern- oder gar weltweit erreichbaren, zentralen LUS für öffentliche Dienstleister geben sollte.

Die beiden Multicast Protokolle dienen den Nachrichtenwegen vom Gerät zum LUS (Request) und vom LUS zum Gerät (Announce). Alle vernetzten Jini-Geräte empfangen diese UDP-Datagramme und reagieren darauf mit einem Unicast Discovery, falls sie von diesen Anforderungen betroffen sind.



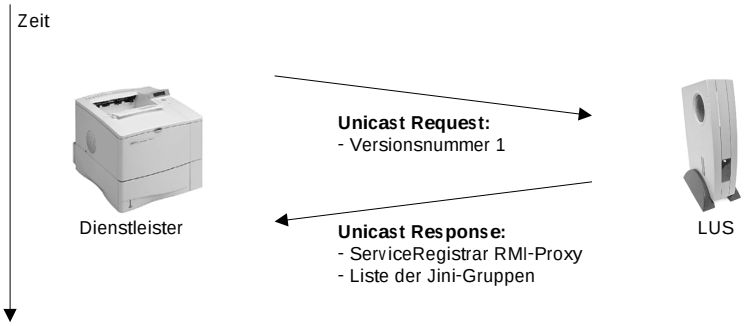
Ein Multicast Discovery findet noch unbekannte LUS Instanzen im lokalen Netz und kann – sofern eine entsprechende Routerkonfiguration vorliegt – auch weitere Netze durchsuchen. LUS Instanzen mit bereits bekannter IP-Adresse werden über das Unicast Discovery Protokoll erreicht.

Unicast Discovery

Jeder LUS nimmt TCP Unicast Verbindungen für das Unicast Discovery Protokoll an. Die Spezifikation empfiehlt hierfür einen TCP Socket Server auf dem Port 4160 [Sun99-2].

Der Unicast Discovery Protokoll-Dialog besteht aus einem einfachen Request/Response-Schema. Der Dienstnehmer teilt dem LUS mit, welche Version des Protokolls er versteht (derzeit stets Versionsnummer 1) [Sun99-2]:

```
// Service Provider -> LUS

int protoVersion = 1;                          // protocol version

ByteArrayOutputStream byteStr =
    new ByteArrayOutputStream();
DataOutputStream objStr =
    new DataOutputStream(byteStr);

objStr.writeInt(protoVersion);
byte[] requestBody = byteStr.toByteArray(); // final result
```



Für einen Unicast Discovery muss der Dienstleister die IP-Adresse des LUS bereits kennen. Die weitere Nutzung der Dienstleistung LUS für Join und Lookup erfolgt über den RMI-Proxy.

Der LUS empfängt diese Anfrage und überprüft, ob er diese Version des Unicast Discovery Protokolls unterstützt. Sun hat bisher nur die Version 1 des Protokolls spezifiziert, in dem der LUS mit folgenden Daten antwortet:

- Einen serialisierten RMI-Proxy für diesen Lookup Service, der das Jini-Interface `net.jini.core.lookup.ServiceRegistrar` implementiert (siehe Anhang). Über dieses Interface können Klienten des LUS auf seine Dienste zugreifen.
- Eine Liste aller Jini-Gruppen, für die dieser LUS zuständig ist.
- Der RMI-Proxy enthält: Die Service ID der LUS Instanz.
- Der RMI-Proxy enthält: Den Netzwerk-Namen des Rechners, auf dem die Instanz des LUS ausgeführt wird.
- Der RMI-Proxy enthält: Die Port-Nummer der LUS Instanz für das Unicast Discovery Protokoll.

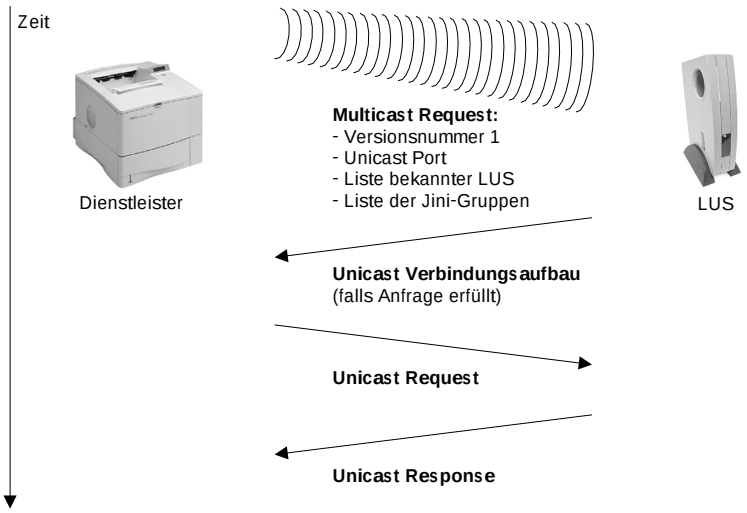
```
// LUS -> Service Provider

net.jini.core.lookup.ServiceRegistrar reg;
String[] groups;                              // groups registrar
                                              // will respond with
java.rmi.MarshalledObject obj =
    new java.rmi.MarshalledObject(reg);
ByteArrayOutputStream byteStr =
    new ByteArrayOutputStream();
ObjectOutputStream objStr =
    new ObjectOutputStream(byteStr);

objStr.writeObject(obj);
objStr.writeInt(groups.length);
for (int i = 0; i < groups.length; i++) {
    objStr.writeUTF(groups[i]);
}

byte[] responseBody = byteStr.toByteArray(); // final result
```

Die empfangenen Informationen und der RMI-Proxy werden vom Gerät unter der Service ID des Lookup Services lokal abgelegt.

Für die weiteren Schritte zur Anmeldung (*Join* Protokoll) oder Suche (*Lookup* Protokoll) eines Dienstleisters greifen die anfragenden Geräte auf diesen RMI-Proxy zurück.

Multicast Request Protocol

Das Multicast Request Protocol wird von Geräten im lokalen Netz verwendet, um vorhandene Lookup Services zu finden.



Ein von einem Multicast Request angesprochener LUS antwortet mit einem Verbindungsaufbau auf dem Unicast Port des suchenden Geräts, das dann einen Unicast Discovery auf dieser Verbindung ausführt.

Hierzu versendet das suchende Gerät Datagramme auf Port 4160 in der Multicast-Gruppe 224.0.1.85. Die Datagramme enthalten

- Die Versionsnummer des Discovery Protokolls (derzeit stets 1).
- Die Port-Nummer, auf der der Empfänger eine Unicast-Verbindung mit dem Absender des Datagramms aufnehmen kann.
- Eine Liste aller LUS-IDs, die das suchende Gerät bereits kennt und die deshalb nicht mehr auf die Anfrage antworten müssen.
- Eine Liste aller Jini-Gruppen, an denen das suchende Gerät interessiert ist.

Aus [Sun99-2]:

```
// Service Provider -> Network, LUS listening

int protoVersion = 1;                       // protocol version
int port;                                   // port to contact
String[] groups;                            // groups of interest
net.jini.core.lookup.ServiceID[] heard;     // known lookups

ByteArrayOutputStream byteStr =
    new ByteArrayOutputStream();
DataOutputStream objStr =
    new DataOutputStream(byteStr);

objStr.writeInt(protoVersion);
objStr.writeInt(port);
objStr.writeInt(heard.length);
for (int i = 0; i < heard.length; i++) {
    heard[i].writeBytes(objStr);
}
objStr.writeInt(groups.length);
for (int i = 0; i < groups.length; i++) {
    objStr.writeUTF(groups[i]);
}

byte[] packetBody = byteStr.toByteArray(); // the final result
```

Falls aufgrund der UDP-Längenbeschränkung von 512 Byte Nutzdaten je Paket nicht ausreichend Informationen in ein einzelnes Datagramm passen, kann der Absender auch mehrere Datagramme mit Teilinformationen versenden.

Alle LUS Instanzen im lokalen Netz warten in dieser Multicast-Gruppe auf Anfragen, untersuchen, ob sie davon angesprochen werden. Bei Bedarf antworten sie mit einer Unicast Verbindung zum fragenden Gerät.

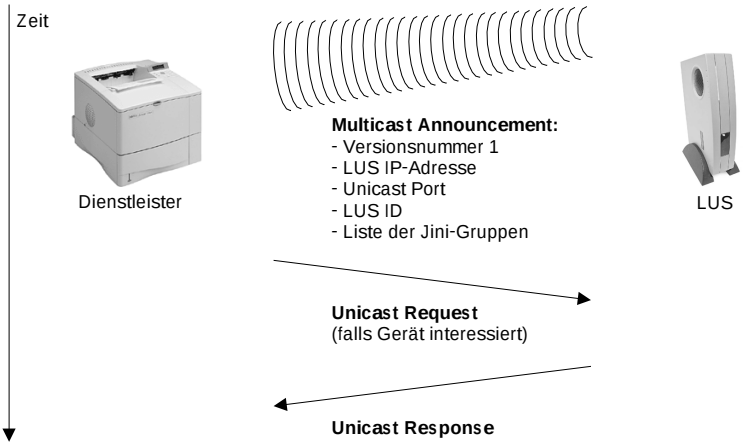
Jini-Geräte halten hierfür einen einfachen TCP Socket Server bereit, der nach externer Verbindungsaufnahme durch den LUS das Unicast Discovery Protokoll durch das Gerät initiiert.

Multicast Announcement Protocol

Ein LUS kann mit Hilfe des Multicast Announcement Protokolls seine Anwesenheit im lokalen Netz ankündigen, etwa um bereits vorhandenen Geräten die Aktivierung einer neuen LUS-Instanz mitzuteilen. Dieses Protokoll dient ebenfalls dazu, Statusänderungen einer LUS-Instanz zu vermelden.

Ein LUS versendet hierzu periodisch und im Fall von Statusänderungen einen kurzen Bericht über seinen aktuellen Zustand auf Port 4160 in der Multicast-Gruppe 224.0.1.84. Die Datagramme enthalten:

- Die Versionsnummer des Discovery Protokolls (derzeit stets 1).
- Der Netzwerk-Name des Rechners, auf dem die Instanz des LUS ausgeführt wird.
- Die Port-Nummer der LUS Instanz für das Unicast Discovery Protokoll.
- Die ID des LUS (eine 128 Bit Zahl).
- Eine Liste aller Jini-Gruppen, für die dieser LUS zuständig ist.



LUS Instanzen senden regelmäßig ein Multicast Announcement, Geräte antworten bei Interesse diesem LUS mit einem Unicast Discovery.

```
// LUS -> Network, Jini devices listening

int protoVersion = 1;                       // protocol version
String hostname;                            // LUS hostname
int port;                                   // port for unicast discovery
String[] groups;                            // groups represented by LUS
net.jini.core.lookup.ServiceID lusID;       // LUS Jini service ID

ByteArrayOutputStream byteStr =
    new ByteArrayOutputStream();
DataOutputStream objStr =
    new DataOutputStream(byteStr);

objStr.writeInt(protoVersion);
objStr.writeUTF(hostname);
lus.ID.writeBytes(objStr);
objStr.writeInt(groups.length);
for (int i = 0; i < groups.length; i++) {
    objStr.writeUTF(groups[i]);
}

byte[] packetBody = byteStr.toByteArray(); // the final result
```

Auch für die UDP-Nachrichten des Multicast Announcement Protokolls ist es möglich, nacheinander mehrere Datagramme mit Teilen der Gesamtinformationen zu-

sammenzustellen, um die Längenbeschränkung von 512 Bytes Nutzdaten je UDP-Paket zu umgehen.

Die Geräte eines Jini-Netzwerks sind stets empfangsbereit, um diese Multicast-Nachrichten der Lookup Service Instanzen entgegenzunehmen. Falls sich auf diese Weise ein LUS meldet, der für den Djinn des empfangenden Geräts zuständig ist, wird das Gerät anhand der übersandten Informationen entscheiden, ob es antwortet:

Das Gerät meldet sich bei einer ihm noch unbekannten LUS-Instanz an, sofern diese für den Djinn des Geräts zuständig ist.

Sendet jedoch ein dem Gerät bereits bekannter LUS, überprüft es, ob sich der zuletzt bekannte Status dieser LUS-Instanz verändert hat. Soweit nötig, wird es sich bei diesem LUS anmelden oder eine vorhandene Anmeldung zurücknehmen.

Anzahl und Intervalle der Multicast-Datagramme

Da Multicast prinzipbedingt keine Fehlerkorrektur und nur eine einfache Fehlererkennung für UDP-Datagramme enthält, sieht die Spezifikation der beiden Multicast-Protokolle vor, dass alle UDP-Datagramme mehrfach verschickt werden.

Die in der Spezifikation genannten Anzahl der Versuche und Intervalle haben allerdings nur Empfehlungscharakter.

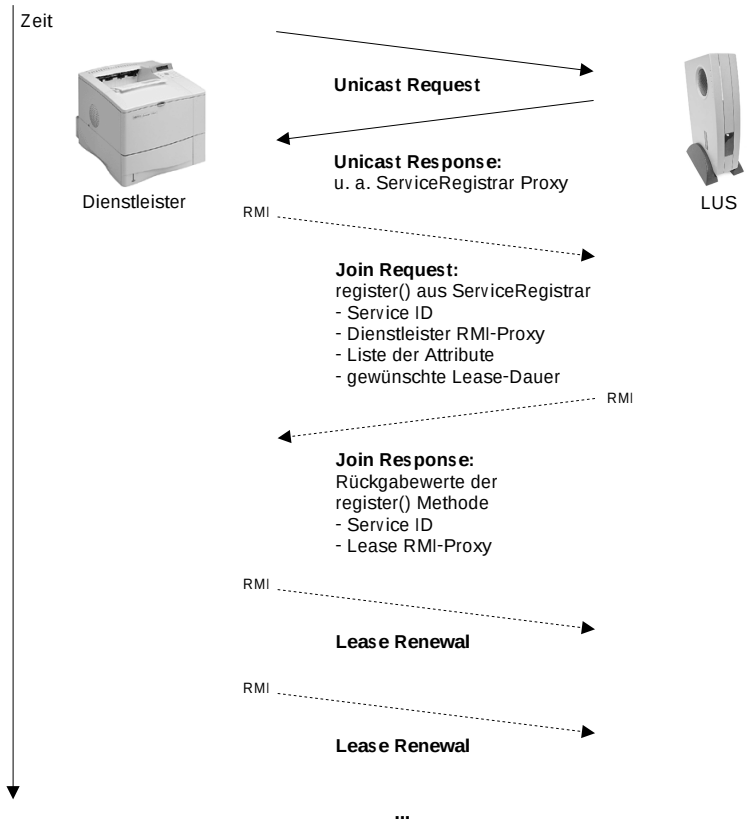
Multicast Request: Um Paketverluste auszugleichen, sind Jini-konforme Geräte angehalten, nach dem Einschalten einen Multicast Request Versuch 7 mal im Abstand von jeweils 5 Sekunden auszuführen.

Multicast Announcement: Zusätzlich zu Statusänderungen (etwa direkt nach dem Einschalten) versendet jeder LUS während seiner gesamten Lebenszeit via Multicast Announcement regelmäßig alle 120 Sekunden Informationen über seinen aktuellen Status.

Zusätzlich ist vorgesehen, dass neu aktivierte Geräte eine zufällig gewählte Zeit von bis zu 15 Sekunden warten, bevor sie mit ihrer Suche nach einem LUS im lokalen Netz beginnen.

Dies soll eine Überlastung von Netz und Empfängern vermeiden, falls beispielsweise alle Geräte einer Jini-Netzgruppe zum gleichen Zeitpunkt – etwa nach Wiederherstellen der ausgefallenen zentralen Stromzufuhr in einem größeren Gebäude – aktiv werden und sonst das lokale Netz mit gleichzeitigen Anfragen überfluten würden.

4.2.3 Join: Anmelden eines Dienstleisters im LUS



Das Join Protokoll besteht aus dem Aufruf der register() Methode des ServiceRegistrar RMI-Proxies. Die anschließende Lease-Verlängerung benutzt ebenfalls einen RMI-Proxy zur Kommunikation mit dem LUS.

Nach Abschluss des Discovery Vorgangs erfolgt die eigentliche Anmeldung des Dienstleisters im LUS. Die Methode `register()` aus der Schnittstelle `net.jini.core.lookup.ServiceRegistrar` nimmt für diesen Vorgang folgende Parameter entgegen:

```
net.jini.core.lookup.ServiceRegistration register(
  net.jini.core.lookup.ServiceItem item,
  long leaseDuration) throws RemoteException;
```

- Eine Beschreibung des zu registrierenden Dienstleisters in Form einer Instanz von `net.jini.core.lookup.ServiceItem`:

```
public class ServiceItem implements Serializable {

    public ServiceID serviceID;
    public Object service;
```

```
    public Entry[] attributeSets;
    public ServiceItem(ServiceID serviceID,
                       Object service,
                       Entry[] attrSets);
}
```

- Die gewünschte Dauer der Lease-Zeit.
- Die ServiceItem-Instanz enthält: Die Service ID des Dienstleisters (eine 128 Bit Zahl), soweit der Dienstleister bereits bei einem LUS registriert wurde. Nur ein LUS darf Jini Service IDs generieren, deshalb dient für noch unregistrierte Dienstleister die Service ID 0 zur Anmeldung.
- Die ServiceItem-Instanz enthält: Einen Verweis auf die Dienstleister-Instanz. Ein RMI-Aufruf der `register`-Methode wandelt diesen Verweis für die Übertragung zum LUS automatisch in einen RMI-Proxy des Dienstleister-Objekts um.
- Die ServiceItem-Instanz enthält: Eine Liste von Dienstleister-Attributen, die jeweils die Schnittstelle `net.jini.core.entry.Entry` implementieren.
  Ein Beispiel für ein Jini-Attribut ist die von Sun als Standard-Attribut vordefinierte Klasse `net.jini.lookup.entry.Address`:

  ```
  public class Address extends AbstractEntry {
      public String street;
      public String organization;
      public String organizationalUnit;
      public String locality;
      public String stateOrProvince;
      public String postalCode;
      public String country;

      public Address();

      public Address(String street,
                     String organization,
                     String organizationalUnit,
                     String locality,
                     String stateOrProvince,
                     String postalCode,
                     String country);
  }
  ```

  Einzige Bedingung an ein Attribut ist, dass es serialisiert werden kann (im Beispiel durch die übergeordnete Klasse AbstractEntry gegeben, die ihrerseits die Schnittstelle `java.io.Serializable` implementiert), damit es für Java RMI nutzbar ist und der Lookup Service Attribute es in einem Objectstore persistent ablegen kann.

Der Lookup Service nimmt diese Werte entgegen und legt sie persistent unter der Service ID des Dienstleisters in einem lokalen Objectstore ab. (Sun verwendet für diesen Zweck JavaSpaces in der *Reggie* Beispiel-Implementation.)

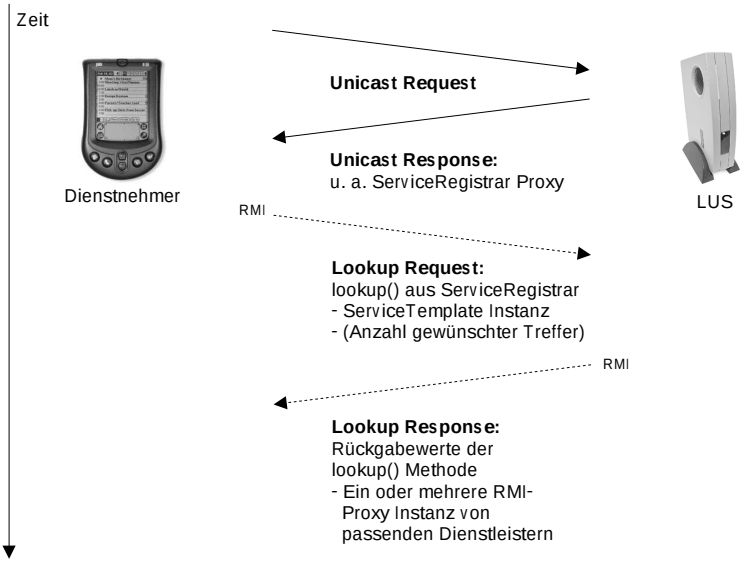
Das Ergebnis nach einer erfolgreichen Anmeldung ist eine Objekt-Instanz, die die Java-Schnittstelle `net.jini.core.lookup.ServiceRegistration` implementiert. Diese nennt dem Dienstleister:

- Die Service ID des Dienstleisters (vom LUS neu vergeben oder unverändert, sofern das Gerät bereits zuvor eine gültige ID hatte).
- Eine Instanz eines Objekts gemäß `net.jini.core.lease.Lease` mit Informationen über die durch den LUS gewährten Leasing-Konditionen. Diese Java-Schnittstelle bietet gleichzeitig mehrere Methoden an, um das Lease ohne Neuanmeldung des Dienstleisters via RMI direkt beim LUS zu verlängern oder abzubrechen.
  Der wichtigste im Lease-Objekt enthaltene Wert ist der Zeitraum bis zum Ablauf der Leasing-Zeit. Der Lookup Service kann einen kürzeren Zeitraum gewähren, als ursprünglich vom Dienstleister angefragt. Es ist dann Aufgabe des Gerätes, sein Lease rechtzeitig über das Lease-Objekt zu verlängern.

Außerdem stellt das Registrations-Objekt mehrere Methoden bereit, um die Attribut-Liste des Dienstleisters via Java RMI direkt im LUS-Objectstore zu verändern.

Der Dienstleister muss für jeden Lookup Service, an dem er sich registriert hat, eine Instanz des Registrations-Objekts aufbewahren und die Lease-Erneuerung darüber handhaben.

4.2.4 Lookup: Suchen nach Dienstleistern



Der ServiceRegistrar RMI-Proxy der LUS Instanz bietet die Methode lookup(), über die das Lookup Protokoll abgewickelt wird. Zur Formulierung der Suchanfrage dient ein ServiceTemplate, das verschiedene Möglichkeiten der Suche anbietet.

Dienstnehmer werden ebenso wie Dienstleister zunächst einen oder mehrere Lookup Services mit Hilfe des Discovery Protokolls suchen.

Die `lookup`-Methode des dabei zurückgelieferten RMI-Proxies erlaubt die Suche nach den vom Teilnehmer gewünschten Dienstleistern des Djinns:

```
Object lookup(net.jini.core.lookup.ServiceTemplate tmpl)
  throws RemoteException;

net.jini.core.lookup.ServiceMatches lookup(
  net.jini.core.lookup.ServiceTemplate tmpl,
  int maxMatches) throws RemoteException;
```

Die beiden durch Overloading unterschiedlich definierten lookup-Methoden dienen der Suche nach entweder exakt einem oder nach mehreren Treffern. Die Reihenfolge der Ergebnisse bei mehreren passenden Dienstleistern ist nicht festgelegt.

Die Klasse `net.jini.core.lookup.ServiceTemplate` beschreibt die Suchanfrage und erlaubt verschiedene Varianten der Dienstleister-Suche:

```
public class ServiceTemplate implements Serializable {

    public ServiceID serviceID;
    public Class[] serviceTypes;
    public Entry[] attributeSetTemplates;

    public ServiceTemplate(ServiceID serviceID,
   Class[] serviceTypes,
   Entry[] attrSetTemplates);
}
```

- Die Suche nach einer konkreten Service-ID. (Dieser Wert bleibt `null` falls der LUS keine Service ID vergleichen soll.)
- Der Vergleich mit verschiedenen Java Klassen. Die Suche liefert zu diesen Klassen kompatible Dienstleister zurück. Für eine typische Suchanfrage werden Java Interface Definitionen verwendet, etwa eine Standarddefinition eines Druck-Dienstleisters.
- Der Vergleich mit verschiedenen Attributen.
  Jini sieht an dieser Stelle nur einfache Identitätsvergleiche vor. Ein Template enthält entweder `null` für Eingenschaften, deren Wert für die Suche unerheblich ist, oder den konkreten Wert, der auf Identität mit vorhandenen, im LUS gespeicherten Attribut-Werten verglichen wird.

4.2.5 Nutzung des Dienstleisters via RMI-Proxy

Nach Abschluss des Discovery, Join und Lookup Vorgangs beginnt die eigentliche Dienstnutzung.

Das Ergebnis der lookup-Methode ist eine Instanz eines Stellvertreter-Objekts für den Zugriff auf den externen Dienstleister durch den Aufruf von Methoden des Stellvertreters.

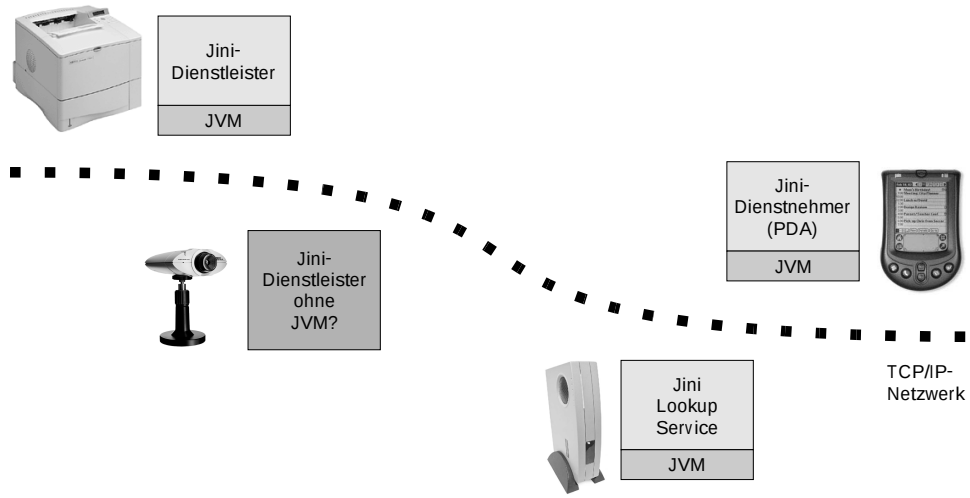
Genauer betrachtet erhält der Dienstnehmer vom LUS die Klassen-ID und den Namen eines serialisierten Stellvertreter-Objekts inklusive des zur Deserialisierung notwendigen Laufzeitstatus aller Instanz-Variablen.

Die Dienstnehmers-JVM wird im lokalen Klassenpfad nach dem Byte Code der Java-Klasse des Stellvertreter-Objekts suchen. Falls sie diesen nicht lokal bereits vorliegen hat, wird sie den Code von der Netz-URL beziehen, die als Zusatzinformation im Datenstrom einer serialisierten Objekt-Instanz enthalten ist.

Das Stellvertreter-Objekt ist im ursprünglichen Jini Szenario ein einfacher, „dummer“ RMI-Proxy, der automatisch vom Java RMI Präcompiler generiert wurde und jeden Methoden-Aufruf über das Netz direkt an die Instanz des Dienstleisters weiterreicht.

4.3 Ein modifiziertes Jini Protokoll ohne Dienstleister-JVM

Für die Protokoll-Änderung wurde davon ausgegangen, dass das Dienstleister selbst zwar TCP/IP-fähig ist, aber keine JVM enthält und somit nicht RMI unterstützt.



Angesichts der Verwendung von Java RMI in den Jini Basisprotokollen stellt sich die Frage, wie ein Dienstleister ohne JVM in ein Jini Netz zu integrieren ist.

Bereits im Unicast Discovery Protokoll und an mehreren weiteren, zentralen Stellen der Basisprotokolle verwendet Jini serialisierte Objekt-Instanzen und Aufrufe via Java RMI.

4.3.1 Serialisierung und Java RMI in den Basisprotokollen aus Sicht des Dienstleisters

Die wichtigsten Problemstellen für einen Dienstleister ohne Java Virtual Machine beim Discovery, Join und Lookup Prozess sind deshalb:

Unicast Discovery: Die Antwort des LUS auf einen Unicast Discovery besteht aus einem serialisierten RMI-Proxy der `ServiceRegistrar` Schnittstelle.
Zentrale Basis-Informationen über den LUS wie zum Beispiel seine Service ID sind nur indirekt über diesen RMI-Proxy erfragbar und nicht im Datenstrom der Protokoll-Antwort direkt lesbar.

Join: Die Anmeldung des Dienstleisters erfolgt über Verwendung des LUS RMI-Proxies.

Attribute: Die für einen Join notwendigen Attribut-Werte liegen in Klassen-Instanzen vor, die serialisiert zum LUS übertragen werden. Änderungen der Attribut-Werte von registrierten Dienstleistern erfolgen via Java RMI.

Leasing: Ein Ergebnis eines erfolgreichen Join Vorgangs ist das vom LUS vergebene Lease, das er per Serialisierung zum Dienstleister sendet. Die Verlängerung eines Leases erfolgt via Java RMI.

Dienstnutzung: Das Stellvertreter-Objekt ist ein RMI-Proxy, zur Dienstnutzung dienen entfernte Methoden-Aufrufe und die Übertragung von serialisierten Parametern in beide Richtungen.

4.3.2 Das Problem Serialisierung und Java RMI

Die Übertragung einer serialisierten Objekt-Instanz zu und der Aufruf einer Methode in einer entfernten Java Virtual Machine bedeutet den Austausch von Protokoll-Nachrichten zwischen den beiden JVMs.

Es liegt nahe, zu versuchen, eine gültige Teilmenge dieses Intra-JVM-Protokolls in einem Gerät ohne JVM zu implementieren, damit dieses mit einer Java Systemumgebung im Rahmen von Jini kommunizieren könnte.

Dies scheitert jedoch an mehreren problematischen Eigenschaften dieses Protokolls:

- Die Nachrichten basieren auf einem genau einzuhaltenden Binärformat.
  (Im Gegensatz dazu bestehen beispielsweise HTTP und SOAP aus dem Austausch einfachster ASCII-Nachrichten gemäß einer Syntax, die gewisse Freiheiten und Vereinfachungen zulässt.)
- Information über Reihenfolge und Datentypen der zusammengesetzten Nachrichten sind nicht Bestandteil der Nachricht, sondern werden von beiden JVMs indirekt zur Laufzeit aus dem Java Byte Code serialisierbarer Klassen ermittelt.
- Die Class ID einer Klasse spielt in den Protokoll-Nachrichten eine bedeutende Rolle. Zur Laufzeit können durch Erweiterungen des Klassenpfads der JVM neue Class IDs hinzukommen, so dass eine vorbereitete Liste von IDs und den dazugehörigen Klasseninformationen nie vollständig sein kann.
- Beim Empfang einer serialisierten Objekt-Instanz ist nur bekannt, dass der Datenstrom der Protokoll-Nachricht eine zum Parameter-Typ *kompatible* Instanz enthält. Es ist offen, welche konkrete Klasse verwendet wird, dies erfährt der Empfänger nur aus der ID.
  Damit kann der Empfänger nicht mehr davon ausgehen, dass eine bestimmte Reihenfolge von Werten mit bekannten Datentypen eintreffen wird.

Eine Implementation dieses Protokolls außerhalb einer JVM erscheint damit nicht gegeben.

(Sie ist prinzipiell möglich, wenn das empfangende Gerät Zugriff zu alle Java Byte Code Dateien des JVM Klassenpfads hat, diese analysieren und daraus die Informationen ableiten kann, die für Empfang und Versand der Protokoll-Nachrichten notwendig ist.

Damit entstünde außerhalb der Java Systemumgebung eine Wiederholung mehrerer zentraler Mechanismen der JVM.

Dies widerspricht jedoch dem eigentlichen Ziel dieser Arbeit – die Implementation von Jini in einer möglichst einfachen und reduzierten Hardware-Umgebung.)

4.3.3 Verzicht auf Java RMI im Dienstleister

Ein Gerät ohne JVM (oder mit reduzierter JVM, zum Beispiel ein Embedded Java System) muss auf einfache Socket-Kommunikation mit den anderen Mitgliedern des Djinns zurückgreifen.

Serialisierung und Java RMI sind jedoch ein so integraler Bestandteil von Jini, dass der Dienstleister auf die Hilfe einer externen JVM zurückgreifen muss, um die Basis-Protokolle abzuwickeln.

Der Lösungsansatz dieser Arbeit greift hierfür auf die Hilfe der beiden anderen am Trader-Dreieck beteiligten Parteien zurück. Die JVM der Lookup Service Instanz übernimmt den Anmelde-Vorgang, die JVM des Dienstnehmers wickelt die Dienstnutzung ab.

4.3.4 Das Stellvertreter-Objekt als Treiber

Das Stellvertreter-Objekt des Dienstleisters muss nicht der bereits beschriebene „dumme“ RMI-Proxy sein, der die Methodenaufrufe und -ergebnisse einfach nur zwischen den beiden Parteien durchreicht.

Ebenso kann der Stellvertreter eine gewisse „Eigenintelligenz“ mitbringen und Werte vorberechnen oder Ergebnisse aufbereiten, zum Beispiel um den benötigten Datenverkehr zu reduzieren. Oder er kann die Kommunikation mit dem Dienstleister über einfache Socket-Kommunikation abwickeln und Parameter und Ergebnisse in der Dienstnehmer-JVM aus den und in die Datenformate der Java Systemumgebung übersetzen.

Ein solcher Stellvertreter ist damit konzeptionell ein vollständiger Geräte-Treiber. Für den Dienstnehmer besteht kein Unterschied zwischen RMI-Proxy und Geräte-Treiber; die Nutzung des Stellvertreter-Objekts ist für ihn völlig transparent. Der einzige Unterschied sind möglicherweise höhere Kosten bei der Ausführung des Treibers in der Dienstnehmer-JVM.

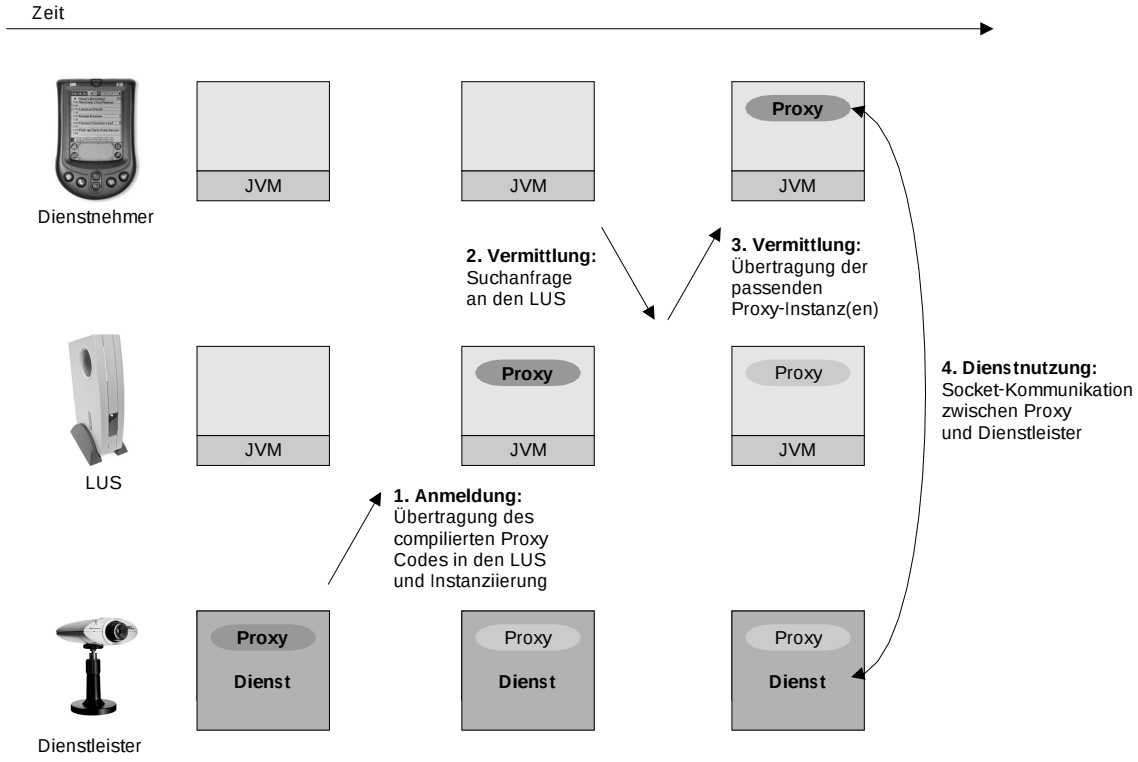
4.3.5 Vorcompilierte Treiber im Gerät

Es wäre möglich, den Java Byte Code des Treibers für ein solches Gerät im Klassenpfad des Lookup Services abzulegen. Dies widerspricht jedoch dem Jini Konzept, das für den Betrieb von Dienstleistern keine Installation von Software bzw. Treibern notwendig sein soll.

Das Gerät bringt den Java Byte Code seines Treibers stattdessen selbst mit, beispielsweise in einem freien ROM-Bereich der Firmware.

Es kann zwar diesen Code nicht selbst ausführen, aber LUS und Dienstnehmer können den Code vom Gerät abholen, wenn sie den Treiber benötigen

Die Startwerte der Instanzparameter erfragt der Treiber durch eine Eigen-Implementation des Deserialisierungs-Vorgangs. Der Treiber implementiert hierfür die Java Schnittstelle `net.jini.core.lookup.NonJavaServiceDriver`. Nach Erzeugen einer noch unvorbereiten Treiber-Instanz im LUS übernimmt die Methode



Das Stellvertreter-Objekt liegt im JVM-losen Gerät als Treiberdatei bereits. Durch die alternative Deserialisierung kann dieses in den LUS und in den Dienstnehmer übertragen und damit eine Jini-konforme Vermittlung des Dienstleisters erfolgen.

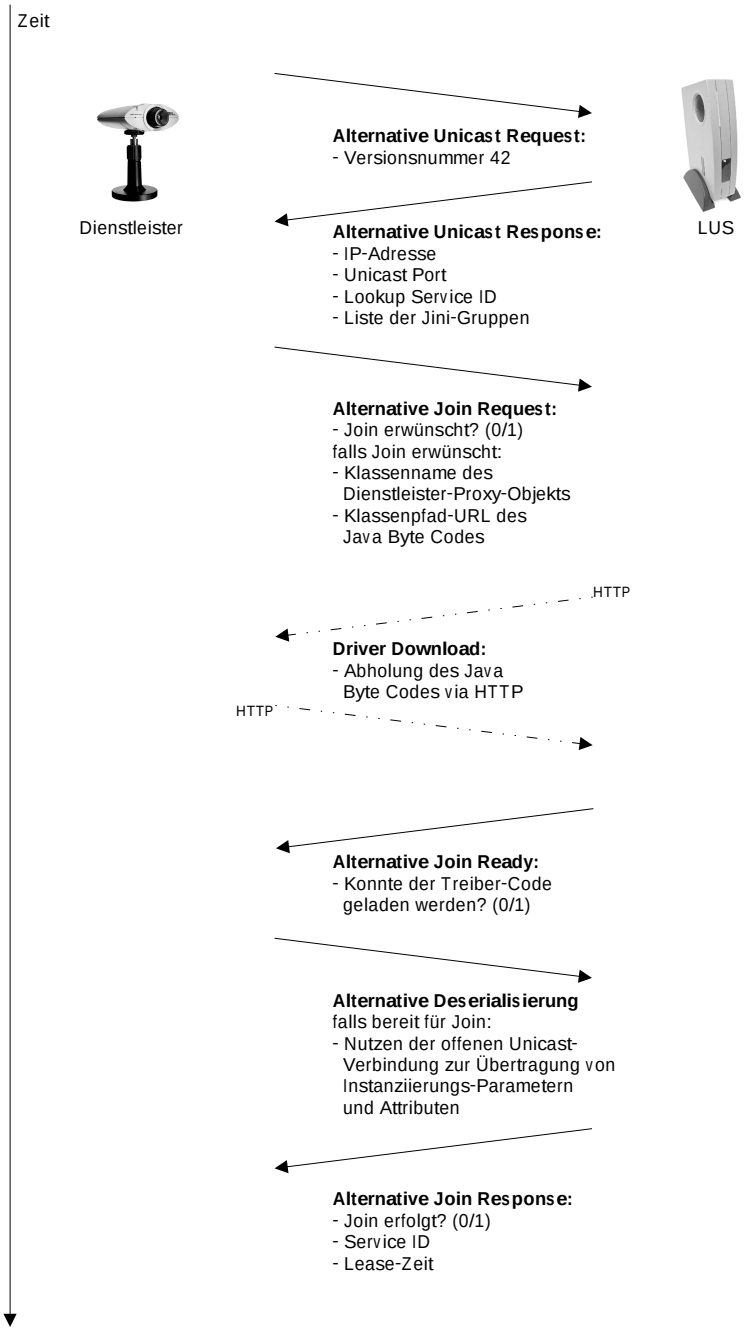
`unmarshalParameters` aus der Protokoll-Verbindung die gewünschten Parameter-Werte. Reihenfolge und Typ der im Protokoll-Dialog übertragenen Parameter bleiben dabei dem Entwickler von Gerät und Treiber überlassen.

```
public interface net.jini.core.lookup.NonJavaServiceDriver
  extends Serializable {

  public net.jini.core.Entry[] unmarshalParameters(
    InputStream is,
    OutputStream os) throws IOException;
}
```

Das Ergebnis der Methode ist eine Jini-konforme Liste von Attributen, die der LUS für die weitere Anmeldung des Treibers verwendet. (Auch die Werte dieser Attribute kann der Entwickler des Geräts zuvor über die offene Protokoll-Verbindung beim Gerät erfragen lassen.)

Ein einfacher HTTP Socket Server im Gerät liefert den Java Byte Code des Treibers auf Anfrage einer externen JVM. Der Lookup Service instanziiert den Treiber über den Classloader.

4.3.6 Modifiziertes Discovery und Join Protokoll



Das modifizierte Protokoll verwendet eine Unicast-Verbindung für den gesamten Dialog zwischen LUS und Dienstleister. Zusätzlich wird bei Bedarf eine HTTP-Verbindung aufgebaut, um den Treiber-Code aus dem Gerät zu laden.

Ein Gerät ohne JVM kann nun mit Hilfe der alternativen Deserialisierung eine Treiber-Instanz an LUS und Dienstnehmer übergeben. Die dazu noch weiter benötigten Änderungen der Jini Basisprotokolle führten zum *modifizierten Discovery und Join Protokoll*, das diese beiden Schritte in einen einzigen Protokoll-Dialog auf Basis einfacher Socket-Kommunikation zusammenfasst.

Die beiden Multicast Protokolle verzichten auf java-spezifische Eigenschaften und ihre Implementation ist ohne Änderung möglich.

Statt der Versionsnummer 1 beginnt ein Unicast Discovery Request seitens des Dienstleisters nun jedoch mit der Versionsnummer 42, um dem LUS mitzuteilen, dass dieses Gerät das modifizierte Protokoll erwartet. (Die Zahl 42 dient hierbei nur der Unterscheidung zum Original-Protokoll und wurde ohne speziellen Grund gewählt.)

```
// Service Provider -> LUS

int protoVersion = 42;                        // protocol version

ByteArrayOutputStream byteStr =
    new ByteArrayOutputStream();
DataOutputStream objStr =
    new DataOutputStream(byteStr);

objStr.writeInt(protoVersion);
byte[] requestBody = byteStr.toByteArray(); // final result
```

Ein LUS, der dieses Protokoll nicht verstehen kann, wird die Verbindung sofort abbrechen. Der modifizierte Lookup Service antwortet mit den Basisinformationen, die der Dienstleister für einen Join benötigt:

```
// LUS -> Service Provider

String hostname;                               // LUS host name
int port;                                      // LUS unicast port
net.jini.core.lookup.ServiceID lusID;          // LUS Jini service ID
String[] groups;                               // LUS group list

ByteArrayOutputStream byteStr =
    new ByteArrayOutputStream();
DataOutputStream objStr =
    new DataOutputStream(byteStr);

hostname.writeUTF(objStr);
objStr.writeInt(port);
lusID.writeBytes(objStr);

objStr.writeInt(groups.length);
for (int i = 0; i < groups.length; i++) {
    objStr.writeUTF(groups[i]);
}

byte[] responseBody = byteStr.toByteArray(); // final result
```

Der Dienstleister entscheidet anhand dieser Angaben, ob er seinen Treiber in diesem LUS anmeldet.

Bei Desinteresse seitens des Dienstleisters ist der Dialog beendet, ansonsten übergibt das Gerät dem LUS die nötigen Informationen, um den Java Byte Code des Treibers via HTTP zu laden und um dann die alternative Deserialisierung durchzuführen.

```
// Service Provider -> LUS

boolean doJoin;                                // wants to join?

net.jini.core.lookup.ServiceID serviceID;
long requestedLeaseDuration;
String driverClassname;
String httpHostname;
int httpPort;
String httpFilename;

ByteArrayOutputStream byteStr =
    new ByteArrayOutputStream();
DataOutputStream objStr =
    new DataOutputStream(byteStr);

if (doJoin) {
  objStr.writeInt(1);                             // true
  objStr.writeLong(requestedLeaseDuration);
  if (serviceID == null) {
    objStr.writeInt(0);                           // service ID = null
  } else {
    objStr.writeInt(1);
    serviceID.writeBytes(obj);
  }
  objStr.writeUTF(driverClassname);
  objStr.writeUTF(httpHostname);
  objStr.writeInt(httpPort);
  objStr.writeUTF(httpFilename);
} else {
  objStr.writeInt(0);                             // false
}

byte[] alternateJoinRequestBody = byteStr.toByteArray();
```

Der Lookup Service wird nun versuchen, den Treiber aus dem Gerät via HTTP zu laden und zu instanziieren. Diese Instanz des Treibers wird damit durch die JVM des LUS ausgeführt:

```
// LUS -> Service Provider

Socket socket;                                 // protocol connection
InputStream is = socket.getInputStream();      // socket stream
OutputStream os = socket.getOutputStream(); // socket stream

URL driverURL =
```

```
  new URL("http", httpHostname, httpPort, httpFilename)

URL urls[] = new URL[1];
urls[0] = driverURL;
URLClassLoader loader = new URLClassLoader(urls);

Object driver = null;
try{
  driver = loader.loadClass(joinreq.getDriverClassname()).
    newInstance();
} catch (Exception e) {
  // premature end of protocol
}

// ... to be continued ...
```

Eine erfolgreiche Instanziierung muss dem Dienstleister mitgeteilt werden. Anschließend tauschen Treiber und Gerät über die bereits offene Protokoll-Verbindung die Anfangs-Parameter und die Dienstleister-Attribute für den Treiber aus.

```
// ... continued ...

DataOutputStream objStr =
    new DataOutputStream(os);
net.jini.core.Entry[] driverAttributes;

if (driver == null) {

  objStr.writeInt(0);                    // failed loading driver
  // end of protocol

} else {

  objStr.writeInt(1);
  driverAttributes =
    ((net.jini.core.lookup.NonJavaServiceDriver)driver).
    unmarshalParameters(is, os);

}

// ...to be continued ...
```

Nun stehen Treiber-Instanz, -Parameter und -Attribute bereit. Der LUS kann ihn damit über seine eigene ServiceRegistrar-Schnittstelle bei sich selbst anmelden.

Bei Erfolg liefert der LUS dem Gerät abschließend die Service ID des Dienstleisters und die gewährte Lease-Zeit zurück.

```
// ... continued ...

net.jini.core.lookup.ServiceRegistration reg;

try {
```

```
  reg = register(new
          net.jini.core.lookup.
            ServiceItem(serviceID, driver, driverAttributes),
          requestedLeaseDuration);
} catch (Exception e) {
  reg = null;
}

if (reg == null) {
  objStr.writeInt(0);
} else {
  objStr.writeInt(1);
  reg.getServiceID().writeBytes(objStr);
  objStr.writeLong(
    reg.getLease.getExpiration() - System.currentTimeMillis()
  );
}
```

Die Treiber-Instanz wurde damit – dieses Mal java-konform – im LUS deserialisiert, im ObjectStore persistent abgelegt und kann wieder aus dem Speicher entfernt werden.

Mit erfolgter Registrierung im LUS steht der Geräte-Treiber ab sofort im Djinn Jini-konform zur Verfügung. Aus Sicht des Dienstnehmers hat sich bei Suche und Nutzung von Dienstleistern *nichts geändert,* der Zugriff auf Dienstleister mit und ohne JVM erfolgt transparent.

4.3.7 Leasing und Attribut-Modifikation ohne Java RMI

Mit Hilfe des modifizierten Discovery und Join Protokoll kann sich der Dienstleister im LUS anmelden; das Stellvertreter-Objekt als Geräte-Treiber macht RMI-Kommunikation vom Dienstnehmer zum Dienstleister unnötig. Es bleiben noch zwei weitere Einsatzgebiete von Java RMI, die das Gerät ohne JVM beherrschen muss: Leasing und die Modifikation von Attributen im Lookup Service.

Beides lässt sich durch eine einfache Neu-Anmeldung mit gleicher Service ID über das modifizierte Protokoll erledigen. Der Lookup Service ersetzt bei einer Neu-Anmeldung alle alten Leasing- und Attribut-Einträge dieser Service ID und vergibt ein neues Lease.

Das Gerät wird deshalb intern eine Liste aller LUS führen, bei denen es sich angemeldet hat, und sich vor Ablauf der Leasing-Zeit eines Eintrags bei diesem LUS erneut anmelden.

Kapitel 5

Eine Implementation in Soft- und Hardware

5.1 Vorgehensweise

Ziel dieser Arbeit war es, einen Jini-fähigen Dienstleister zu implementieren, der selbst auf die Java-Systemumgebung verzichten kann.

Nachdem erste Tests des modifizierten Protokolls in der *Skript-Sprache Perl* erfolgreich verliefen, entstand zunächst ein Prototyp in der *Programmiersprache C*, der auf einem Arbeitsplatzrechner unter *Linux* ausgeführt wurde. Wichtigster Bestandteil ist eine portable C-Bibliothek, die einer Dienstleister-Applikation das modifizierte Protokoll zur Verfügung stellt.

Der Arbeitsplatz-PC wäre selbst problemlos in der Lage, eine Java-Systemumgebung auszuführen. Zu Demonstrationszwecken entstand deshalb ein zweiter Prototyp auf einer Hardware, die offensichtlich zu schwach für eine Java Virtual Machine ist.

Die Suche nach einer dafür geeigneten Hardware gestaltete sich dabei schwieriger als erwartet.

5.2 Implementation des Protokolls in ANSI-C

Für den ersten vollständigen Prototyp fiel die Wahl auf die *Programmiersprache C*, um später eine Portierung auf ein Embedded System zu ermöglichen. Die Entwicklung begann auf einem x86-Arbeitsplatzrechner unter Linux, doch wurde mit Blick auf die spätere Portierung auf besondere Eigenschaften dieser Systemumgebung verzichtet und die *ANSI*- und *POSIX-Standards* als Referenz herangezogen [LKR90] [Lew92].

5.2.1 Die Programmiersprache C, das UNIX Betriebssystem und POSIX

C wurde Anfang der 70er Jahre in den Bell Laboraties als Werkzeug zur Implementation des Betriebssystems UNIX entwickelt.

Die Sprachdefinition von C ist sehr klein und überschaubar und die Programmierung eines (einfachen, nicht-optimierenden) C-Compilers stellt keine große Hürde dar, da sich die Kontrollstrukturen und Datentypen oft direkt in Prozessorbefehle und -register abbilden lassen.

> „C ist eine relativ maschinennahe Sprache. Diese Feststellung ist nicht abwertend zu sehen, sie drückt lediglich aus, dass C mit den gleichen Objekten umgeht wie die meisten Rechner selbst.“ [KR77]

Der Sprachstandard war lange Zeit allein durch das Buch der beiden C-Erfinder Kernighan und Ritchie [KR77] definiert, doch während C immer populärer wurde, fügten verschiedene Compiler-Entwickler jeweils eigene Erweiterungen hinzu, was zu einer Vielzahl leicht unterschiedlicher C-Dialekte führte und die plattform-übergreifende Software-Entwicklung erschwerte. Um diesem Problem zu begegnen, beschloss das ANSI nach mehrjähriger Arbeit 1988 einen übergreifenden *ANSI-C-Standard*, den die verschiedenen C-Compiler heute zumindest als kleinsten gemeinsamen Nenner unterstützen.

C ist eine imperative Programmiersprache mit den üblichen Kontrollstrukturen und vereint Erfahrungen aus Algol 68 und B, einer weiteren Eigenentwicklung der Bell Labs. Seine herausragendste Eigenschaft ist der vollständige Verzicht auf Typsicherheit, der „Hauptgrund dafür, dass C gleichermaßen geliebt und gehasst wird [...] die einen begeistern sich über die Flexibilität, die anderen finden es viel zu unsicher.“ [Seb93].

Jeder Datentyp lässt sich beliebig in jeden anderen umwandeln, selbst dann, wenn dies semantisch keinen Sinn ergibt. Dies gilt auch für den in C besonders wichtigen *Pointer*-Datentyp, ein Zeiger, der direkt auf Speicherbereiche des Rechners verweist. Zusammen mit Typumwandlung und *Zeigerarithmetik* (direkte Manipulation von Speicheradressen) ist ein Programmierer in der Lage, sehr hardware-nah zu arbeiten.

Gleichzeitig ist dies auch die größte Gefahr in C. Das Schreiben in einen falschen Speicherbereich durch die fehlerhafte Verwendung eines Zeigers ist auch bei professionellen Entwicklern einer der häufigsten Fehler und führt meist direkt zum Absturz der Applikation. Ein Schutz gegen solche Fehler existiert in ANSI-C nicht.

> „[Auf Zeigerarithmetik] ist keinerlei Verlass [...] Wenn Sie Glück haben, erhalten Sie offensichtlich unsinnige Resultate auf allen Maschinen. Haben Sie Pech, dann funktioniert Ihr Programm auf einer Maschine und bricht auf einer anderen unter mysteriösen Umständen in sich zusammen.“ [KR77]

Diese Erfahrungen führten bei späteren Sprachen zu strengeren Typ-Konzepten und zu Vereinfachungen wie zum Beispiel den Garbage Collector, der in Java Anwendung findet und die Fehlerquote während der Entwicklung stark verringert.

C und sein direkter Nachfolger C++ sind – auch wegen ihrer Hardware-Nähe – die erste Wahl für die Entwicklung von Betriebssystemen und Gerätetreibern und neben den verschiedenen UNIX Varianten (Solaris, AIX, BSD, NextStep, Linux) sind alle derzeit kommerziell bedeutenden Betriebssysteme (die Microsoft Windows Familie, Apple MacOS und MacOS X, BeOS, QNX...) hauptsächlich in C oder einem C-Dialekt entstanden.

C selbst besitzt einen sehr kleinen Umfang, die Möglichkeiten des Entwicklers werden sehr viel mehr durch die Programmierumgebung des verwendeten Betriebssystems, durch Bibliotheken und APIs bestimmt als durch den Sprachstandard. So sieht die Entwicklung in C unter Microsoft Windows völlig anders aus als unter einem UNIX System, selbst dann, wenn auf beiden System ANSI-konforme C-Compiler zur Verfügung stehen.

Auch die Bibliotheken der verschiedenen UNIX-Hersteller entfernten sich im Laufe ihrer Entwicklung voneinander, was die Programmierung von Applikationen für mehr als eine UNIX-Variante mehr und mehr erschwerte. Um eine gegenseitige Isolation zu verhindern, einigten sich die Hersteller auf einen gemeinsamen Standard für ihre Systeme: *POSIX*.

In POSIX werden zahlreiche Eigenschaften eines UNIX-Systems vorgegeben: Neben Bibliotheken, Systemfunktionen und Datenstrukturen auch Informationen wie Terminal-Kontrollbefehle, Zeitzonen-, Landes-, Sprachbezeichnungen etc. und auch das Verhalten von Basis-Werkzeugen wie der Shell und zahlreichen anderen typischen UNIX-Programmen.

Zusammen mit den *BSD Socket* Netzwerk-Routinen waren für die Entwicklung des Prototyps die *POSIX-Thread* Bibliotheken für C die bedeutendsten Werkzeuge aus dem POSIX-Standard.

5.2.2 Komponenten der C-Bibliothek für Jini (mit Threads)

Das modifizierte Discovery und Join Protokoll ist ein einfacher Unicast-Dialog, der sich gradlinig in C umsetzen lässt.

Aufwendiger ist die Anforderung der Jini Basisprotokolle, dass ein Dienstleister auf verschiedenartige Verbindungsversuche antworten und auch selbst in regelmässigen Abständen Protokoll-Nachrichten versenden muss.

Die C-Bibliothek des ersten Prototypen verwendet deshalb Multithreading und startet für die verschiedenen Protokoll-Aufgaben eigene Threads:

HTTP Server: Ein TCP-Unicast Server, der HTTP versteht und so einer externen Java Virtual Machine den vorcompilierten Java Byte Code des Stellvertreter-Objekts / Jini-Treibers zurückliefert.

Multicast Announcement Server: Ein UDP-Multicast Server, der die vom LUS versandte Datagramme des Multicast Announcement Protokolls empfängt und verarbeitet. Bei Interesse an dem LUS wird eine Unicast Verbindung aufgebaut und das modifizierte Discovery und Join Protokoll ausgeführt.

Multicast Request Sender: Ein Thread, der auf Veranlassung des Dienstleisters hin 7 UDP-Datagramme des Multicast Request Protokolls im Abstand von 5 Sekunden versendet.

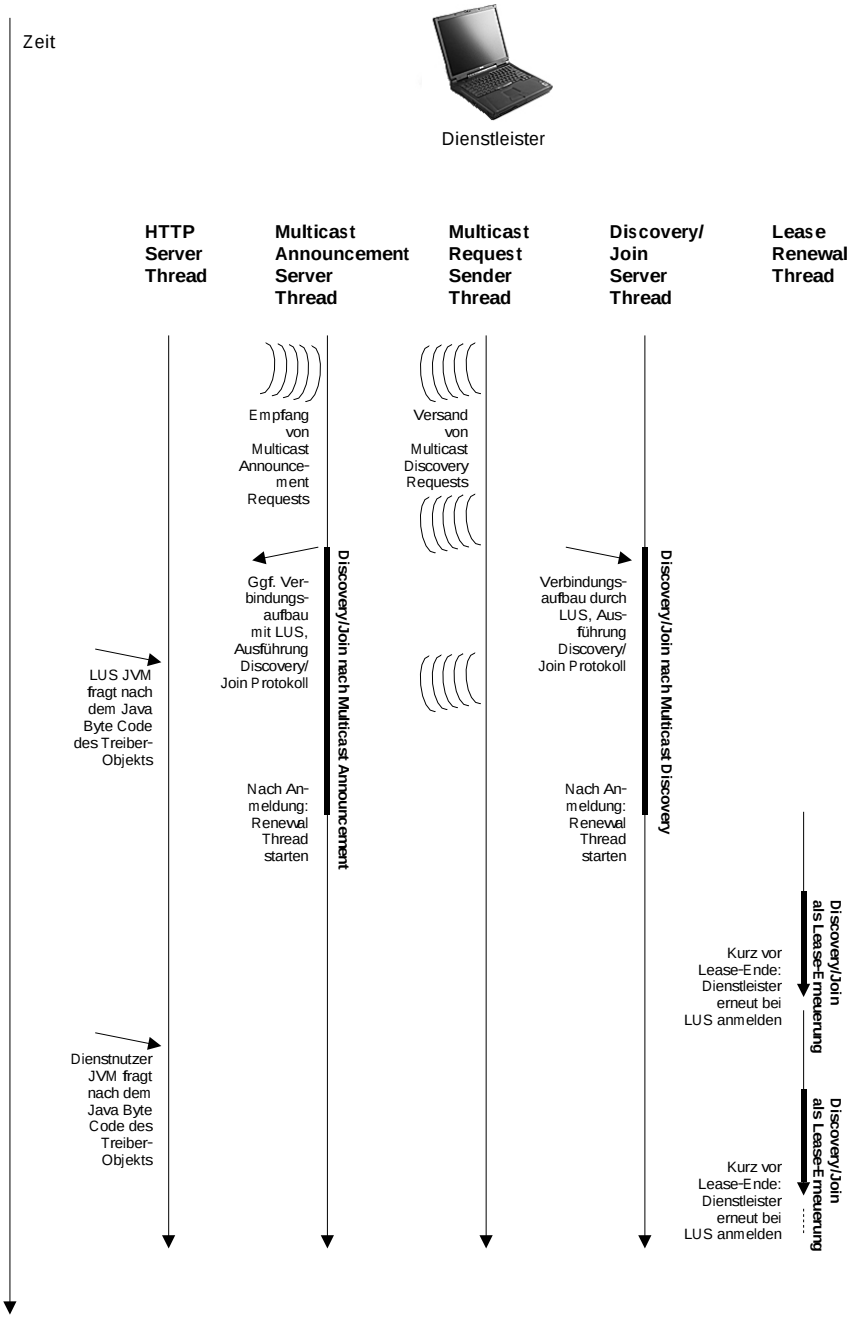
Discovery, Join Server: Ein TCP-Unicast Server, der auf Verbindungsversuche des modifizierten Discovery und Join Protokolls reagiert und den beschriebenen Protokoll-Dialog abwickelt.

Lease Renewal Thread: Zusätzlich startet jede erfolgreiche Anmeldung einen weiteren Thread, falls das vom LUS gelieferte Lease kürzer ist als die vom Dienstleister gewünschte Leasing-Zeit.

Die einzige Aufgabe des Threads besteht darin, vor Ablauf die Anmeldung zu erneuern und so ein neues Lease zu sichern.

Die verschiedenen Teil-Threads der Bibliothek greifen auf gemeinsame Daten zu, die deshalb mit Hilfe von Mutex-Semaphore gegen gleichzeitigen Zugriff verschiedener Threads geschützt sind:

Service ID: Der Dienstleister beginnt mit einer leeren Service ID (mit dem Wert 0) und erhält nach der ersten erfolgten Anmeldung eine gültige ID.



Die Jini C-Bibliothek verwendet Threads, um die Teilaufgaben der Basisprotokolle zu erledigen. In dieser Darstellung sind zwei Anmeldevorgänge parallel dargestellt, einer infolge eines Multicast Announcements, ein anderer nach einem Multicast Discovery. In beiden Fällen startet anschließend eine Lease-Erneuerung, die solange wiederholt wird, bis die gewünschte Anmelde-Zeit erreicht ist.

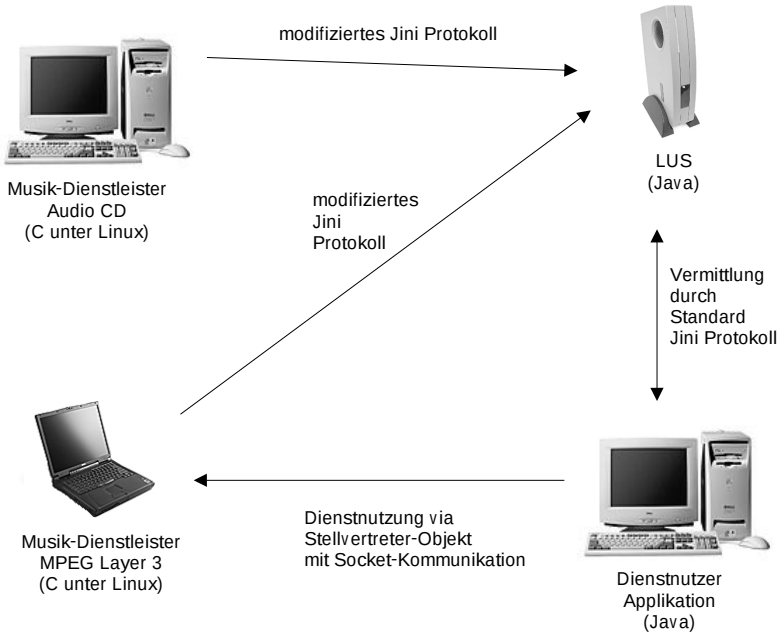
Liste der LUS Instanzen: Der Dienstleister merkt sich alle ihm bekannten Service IDs von Jini Lookup Service Instanzen, die auf frühere Multicast Request Anfragen geantwortet oder die von sich aus per Multicast Announcement auf sich aufmerksam gemacht haben.
Diese Liste findet später bei der Erstellung weiterer Datagramme für das Multicast Request Protokoll Verwendung, um unnötige Antworten bereits bekannter LUS Instanzen zu vermeiden.

Liste der Anmeldungen: Ein Eintrag in dieser Liste dient dem Dienstleister als Referenz für die Lease Erneuerung.

5.2.3 Die ersten Prototypen

Zur Demonstration des Konzepts entstanden auf einem Arbeitsplatz-PC mehrere C-Dienstleister, darunter ein Beispiel zur Steuerung einer Stereo-Anlage via Jini:

Zwei Computer starten je einen in C programmierten Dienstleister. Ein Tischrechner spielt Musik von einer Audio-CD, ein Laptop ohne CD-Laufwerk steuert einen MP3-Decoder.



Zwei unterschiedliche, in C implementierte Dienstleister stellen das gleiche Java Interface zur Verfügung - die dienstnutzende Java Applikation kann beide steuern.

Beide Dienstleister enthalten je einen eigenen Jini-Treiber, der ein generisches *AudioPlayerService*-Interface implementiert. So bieten die beiden völlig unterschiedlichen Geräte aus Sicht eines Jini Dienstnehmers die gleichen Funktionen: Start, Stopp, Auswahl des zu spielenden Titels.

Der in Java programmierte Dienstnehmer wird im Rahmen der Jini Infrastruktur automatisch auf neue AudioPlayerService-Dienstleister hingewiesen und der Anwender kann diese über das Netzwerk fernsteuern.

5.3 Implementation des Protokolls für eine spezielle Hardware

Die Prototypen auf einem vollwertigen Arbeitsplatz-PC waren nur zur Demonstration der Funktion des modifizierten Protokolls geeignet.

Im nächsten Schritt sollte das Protokoll in einer Hardware Anwendung finden, die offensichtlich zu leistungsschwach für eine vollwertige Java Virtual Machine ist, wie das unmodifizierte Jini sie im Dienstleister verlangt.

5.3.1 Suche nach einem geeigneten Gerät

Sun spricht in seinen Jini-Unterlagen gern von vernetzten Kaffeemaschinen und Toastern – im vollen Bewusstsein, dass es sich hierbei um eine absurde Übertreibung, aber eben auch um ein griffiges Beispiel handelt.

Eine erste Überlegung war es deshalb, Sun hier einmal beim Wort zu nehmen und tatsächlich einen Toaster oder eine Kaffeemaschine mit einem Netzwerk-Anschluss und dem modifizierten Protokoll auszustatten.

Der Küchengeräte-Hersteller *Bosch-Siemens* war auf Anfrage sofort bereit, die hausinternen Schaltpläne und Quelltexte der Microcontroller-Software ihrer Produkte zur Verfügung zu stellen.

Das überraschende Ergebnis aus diesen Unterlagen war, dass diese Geräte üblicherweise nur einfachste elektronische Regelkreise enthalten und ohne Programmierung auskommen. Wenn ein Top-Modell einmal doch einen eigenen Microcontroller verwendet, dann in aller Regel nur ein 4-Bit Modell mit kaum ausreichend Speicher, um die Protokoll-Aufgaben eines modifizierten Jini darin unterzubringen.

Um in solchen Geräten das für Jini notwendige TCP/IP Protokoll zu integrieren, ist neben einem Speicherausbau noch ein weiterer Microcontroller notwendig. Die meisten dieser Chips erlauben noch keinen Ethernet-Anschluss, sondern bieten nur PPP auf einer einfachen seriellen Schnittstelle. Zudem ist TCP/IP selbst oft nur teilweise implementiert, so wird das für Jini wichtige Multicast beispielsweise von den meisten marktüblichen TCP/IP-Microcontrollern nicht unterstützt.

Prinzipiell wäre es also möglich gewesen, dem Microcontroller eines Luxus-Toasters einen weiteren Chip für TCP/IP zur Seite zu stellen, die Toaster-Software zu ändern und das Gerät via PPP in ein Jini-Netzwerk zu integrieren. Der technische Aufwand einer solchen Hardware-Eigenentwicklung erschien im Rahmen dieser Arbeit jedoch unangebracht hoch.

Als Alternative kamen deshalb Geräte in Betracht, die bereits eine Netz-werk-Schnittstelle enthalten, beispielsweise Netzwerk-Drucker und -Drucker-Spooler, Set-top-Boxen oder Webcams.

Die Wahl fiel schließlich auf eine Internet-Kamera vom Typ *NetEye 2100* der Firma *Axis*.

Die NetEye 2100 Kamera(Axis, 2000), Vorder- und Rückanschicht.

Die Webcam enthält einen vollständigen Kleinst-Computer auf Basis einer von Axis selbst entwickelten Embedded-CPU. Der *ETRAX 100* 32 Bit Prozessor ist auf Netzwerkanwendungen spezialisiert und bietet TCP/IP via Ethernet und serielles PPP. 2 MByte Flash-ROM und 8 MByte RAM stehen für Betriebssystem und Applikationen bereit, ein Teil des Speichers steht als eine gepufferte RAM-Disk der Kamera zur Verfügung.

Die Etrax 100 Systemumgebung ist nicht leistungsstark genug für eine Java Virtual Machine.

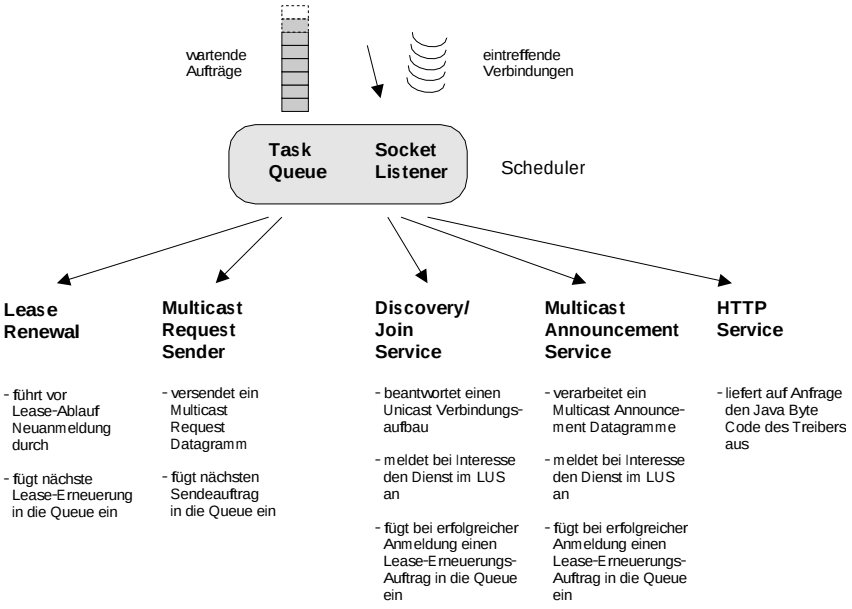
Als Betriebssystem verwendet das Gerät *MicroLinux*. Diese reduzierte Variante des Linux-Kernels ist auf den Einsatz in Embedded Systemen spezialisiert, es hat geringere Anforderungen an die einzusetzende CPU und benötigt zur Laufzeit weniger RAM.

Die Firmware der Kamera enthält neben der Software zur Bildsteuerung einen einfachen HTTP Webserver und einen FTP-Server zur Übertragung von Dateien auf das Gerät.

5.3.2 Komponenten der C-Bibliothek für Jini (ohne Threads)

MicroLinux benutzt eine vereinfachte GNU-C-Bibliothek, die mehrere Teile der ANSI- und POSIX-Standards bewusst auslässt, darunter Threads und Semaphoren. Dieser Verzicht verlangte bei der Programmierung des zweiten Prototypen eine Re-Implementierung der Jini-C-Bibliothek.

Diese zweite Fassung der Bibliothek verwendet stattdessen einen einfachen, selbst implementierten Event-Scheduler.



Der Scheduler prüft mehrere Server-Sockets auf eingehende Verbindungen und startet daraufhin ein dafür zuständiges Unterprogramm, das das Protokoll dieser Verbindungsart betreut. Zusätzlich startet er nach Ablauf eines vorgegebenen Zeitraums einzelne Aufträge wie Multicast Request und Lease Renewal.

Die Teil-Threads der ursprünglichen Jini-Bibliothek finden sich hier als kurze Unterprogramme wieder, die der Scheduler nach Bedarf ausführt. Da diese Unterprogramme jeweils nur sehr wenig Code ausführen und den Programmfluss schnell wieder zurück an den Scheduler geben, werden die einzelnen Aufgaben ausreichend zeitnah erledigt. Eine Verzögerung in der Bearbeitung der Aufträge ist hier jedoch stets möglich.

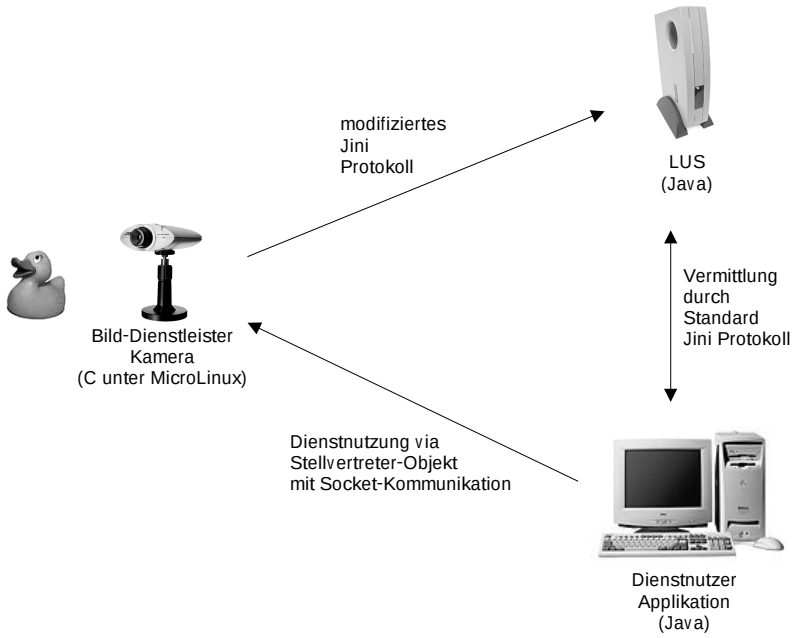
5.3.3 Jini in einer Webcam

Dank der Ähnlichkeiten zwischen Linux und MicroLinux gestaltete sich die grundsätzliche Anpassung der ursprünglichen Quellen einfacher als erwartet.

Schwierigkeiten ergaben sich nur aus der teilweise noch unvollständigen Portierung der Gnu-C-Bibliothek an die ETRAX Architektur, was zu zahlreichen Abstürzen des Kamera-Rechners während der Entwicklung führte. Durch Verzicht auf die betroffenen C-Funktionen oder durch Eigenentwicklung der nötigen Funktionalität ließen sich diese Probleme umgehen.

Das Programm zur Unterstützung des modifizierten Discovery und Join Protokolls startet in der Kamera als ein eigener Prozess. Der bereits vorhandene Webserver der Firmware dient zur Auslieferung des vorcompilierten Stellvertreter-Objekt-Codes an externe JVMs. Der dafür notwendige Java Byte Code liegt als Datei auf der RAM-Disk vor.

Die Kamera kann nun selbstständig alle LUS-Instanzen im lokalen Netz finden und den Treiber-Code dort anmelden. Der Treiber implementiert eine `CameraService` Schnittstelle, über die ein Dienstnehmer einzelne Schnappschüsse aus der Kamera aus-

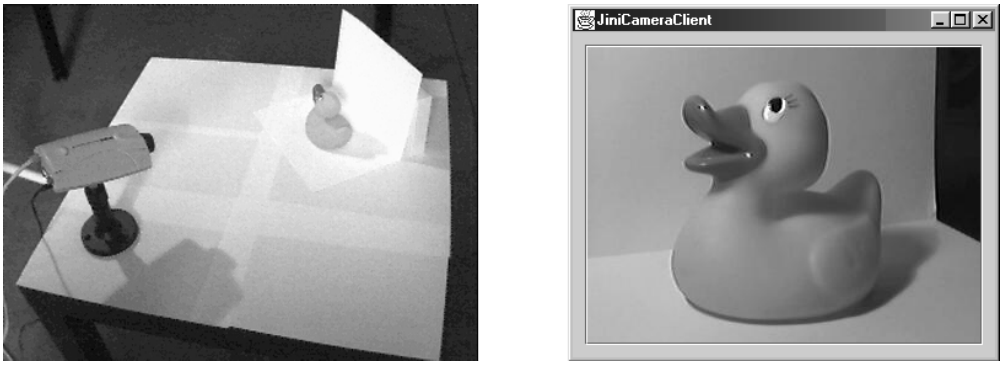


Die Kamera findet nach Anschluss an das Netz automatisch die LUS Instanz. Mit Beginn der Dienstnutzung sendet sie Bilder an den Klienten.

lesen und anzeigen kann.

Ein solcher Klient wurde in Java implementiert, er wird vom Lookup Service automatisch informiert, sobald ein Kamera-Dienstleister dem Djinn beitritt.

Nach Anschluss der Kamera im lokalen Netz beginnt dieser Klient mit kurzer Verzögerung damit, Bilder der Kamera auf dem Bildschirm zu zeigen.



Die Webcam als Jini Dienstleister.

Kapitel 6

Zusammenfassung und Ausblick

6.1 Zusammenfassung

Auslöser dieser Arbeit war die Beobachtung, dass Sun bei der Spezifikation von Jini die Integration von nicht-Java-fähigen Geräten nicht spezifiziert hatte. Eine Untersuchung der Jini Quelltexte und der Basisprotokolle ergab, dass in diesen Java RMI zum Einsatz kommt. Aufgrund der Implementationseigenschaften von RMI in der Java Virtual Machine ist es jedoch einer nicht-Java-fähigen Systemumgebung kaum möglich, einen entfernten Methodenaufruf in einer entfernten JVM auszulösen.

Es enstand im Rahmen der Arbeit ein zusätzliches Jini Basisprotokoll, das eine gerinfügige Änderung des Lookup Services notwendig macht. Die Erweiterung ist abwärtskompatibel und ermöglicht die Integration von nicht-Java-fähigen Dienstleistern in Jini.

Ein deutlicher Nachteil dieser Lösung ist, dass eine Integration von Dienstnehmern mit dem modifizierten Protokoll nicht möglich ist, da die Dienstnutzung in Jini über ein in Java Byte Code compiliertes Stellvertreter-Objekt erfolgt. Zur Ausführung dieses Codes und damit zum Zugriff auf den Dienstleister ist deshalb eine JVM weiterhin notwendig.

6.2 Ein alternativer Ansatz: Das Jini Surrogate Project

Parallel zu dieser Arbeit entstand nach Anregung der Mitglieder der offenen *Jini Community* das *Surrogate Project* um ebenfalls das Problem der Integration von nicht-Java-fähigen Geräten anzugehen.

Das modifizierte Discovery und Join Protokoll floss in die Arbeit des Surrogate Projects ein, jedoch wurde die Idee einer Basisprotokoll-Änderung zugunsten eines allgemeineren Ansatzes verworfen.

Das Ergebnis dieser Entwickling [Tho01] sieht eine zusätzliche JVM-Instanz im Jini Netzwerk vor, die von JVM-losen Geräten über ein eigenes Suchprotokoll gefunden werden kann. Neben TCP/IP ist der Surrogate auch auf andere Verbindungsarten vorbereitet, beispielsweise USB. Der Surrogate dient damit als ein *Device Bay*.

Der Einsatz des Surrogate erlaubt JVM-losen Dienstleistern *und* Dienstnehmern den Zugriff auf den Djinn ohne Veränderung der Jini Basisprotokolle oder des Lookup Services, da die Basisprotokolle unverändert bleiben.

Das wesentlich allgemeiner spezifizierte Protokoll zwischen Gerät und Surrogate geht dabei den gleichen Weg, den in dieser Arbeit beschriebene, spezialisierte Protokoll zwischen Gerät und modifiziertem LUS verwendet:

Das Gerät überträgt den Java Byte Code eines Stellvertreters in die Surrogate-JVM, der Stellvertreter übernimmt die Kommunikation mit Jini. Wie die Kommunikation zwischen Stellvertreter und Gerät aussieht, bleibt dabei dem Entwickler überlassen.

Neben den erwähnten Vorteilen dieser allgemeineren Lösung bleibt als Nachteil zu erwähnen, dass eine zusätzliche JVM und der erhöhte Aufwand bei der Ausführung weitere Kosten beim Aufbau des Djinns nach sich ziehen: Der Surrogate widerspricht der Jini Philosophie, da er zunächst vom Anwender des Netzes installiert werden muss.

6.3 Ausblick

Das Ziel eines reduzierten „Jini ohne Java“ konnte im Rahmen dieser Diplomarbeit erfüllt werden.

Doch die Erfahrung bei der Suche nach einer geeigneten Hardware für die Demonstration führte zu einer überraschenden Erkenntnis: Echte Haushaltsgeräte sind noch sehr viel stärkeren Einschränkungen unterworfen als erwartet und trotz Moores Gesetz werden kleine Haushaltsgeräte nach Meinung ihrer Entwickler auch langfristig weiterhin aus Kostengründen nur einfache 4- oder 8-Bit Embedded Controller einsetzen.

Diese technische Hürde kann den Durchbruch der vorgestellten Technologien weiterhin verzögern.

Ein Thema, das in dieser Arbeit nicht betrachtet wurde, ist die berechtigte Frage nach dem Sinn von vernetzten Reiskochern und Kaffeemaschinen: Obwohl die Industrie seit mehreren Jahren den vernetzten Haushalt als vielversprechenden zukünftigen Markt ankündigt, hat sie bisher nur wenige Konzepte vorstellen können, die einen tatsächlichen Mehrwert für den Nutzer bringen.

Dass diese Anwendungen fehlen zeigt sich auch darin, dass sowohl UPnP als auch Jini bislang nur in Prototypen oder in internen Produkten Verwendung finden, obwohl beide Gruppen für spätestens Anfang 2001 marktfertige Produkte auf dem Massenmarkt versprachen (siehe auch [Sha00]). Es existiert weiterhin ein Wirrwarr an Verbindungen, Netzwerken und Treibern.

Einzig Bluetooth beginnt jetzt, Mitte 2001, auf dem Markt in Erscheinung zu treten. Die ersten Produkte verlassen das Laborstadium, aber sie zeigen massive Kinderkrankheiten: Trotz der frühen Definition von Bluetooth-Profilen existieren bereits zueinander inkompatible Geräte.

Der vernetzte Haushalt ist somit einen ersten Schritt näher gerückt, aber noch lange nicht erreicht.

Anhang A

Standard-Interface Lookup Service

Gekürzter Quelltext von `net.jini.core.lookup.ServiceRegistrar`:

```
package net.jini.core.lookup;

import java.rmi.RemoteException;
import java.rmi.MarshalledObject;
import net.jini.core.event.*;
import net.jini.core.discovery.LookupLocator;

public interface ServiceRegistrar {

    ServiceID getServiceID();

    LookupLocator getLocator() throws RemoteException;

    String[] getGroups() throws RemoteException;

    ServiceRegistration register(ServiceItem item, long leaseDuration)
      throws RemoteException;

    Object lookup(ServiceTemplate tmpl) throws RemoteException;

    ServiceMatches lookup(ServiceTemplate tmpl, int maxMatches)
      throws RemoteException;

    EventRegistration notify(ServiceTemplate tmpl,
                             int transitions,
                             RemoteEventListener listener,
                             MarshalledObject handback,
                             long leaseDuration)
      throws RemoteException;

    Class[] getEntryClasses(ServiceTemplate tmpl) throws RemoteException;

    Object[] getFieldValues(ServiceTemplate tmpl, int setIndex, String field)
      throws NoSuchFieldException, RemoteException;
```

```
    Class[] getServiceTypes(ServiceTemplate tmpl, String prefix)
      throws RemoteException;

}
```

Literaturverzeichnis

[ANSA89] *The Advanced Network Systems Architecture (ANSA) Reference Manual*, Architecture Projects Management Ltd., Cambridge, 1989

[AKZ99] Gerd Aschemann, Roger Kehr, Andreas Zeidler: *Jini Shortcomings*, FG Datenbanken und verteilte Systeme, FB Informatik, TU Darmstadt, Vortrag zum 2. Jini Workshop Zürich, 1999, http://gemini.iti.informatik.tu-darmstadt.de/ kehr/research.html

[Atk95] Tom Atkinson: *The History Of Computing At Chemistry*, Department of Chemistry Alumni Magazine, Michigan State University, November 1995

[Bar01] Robert Baric: Studienarbeit *Jini und Mobilität: Dynamische Dienstleistung und -nutzung in mobilen Systemumgebungen*, Arbeitsbereich Verteilte Systeme, Universität Hamburg, 2001

[Bau00] Daniel Baumberger: *Intel Universal Plug and Play SDK v1.0 for Linux Technology Brief*, Revision 1.1, Intel Architecture Labs, 2000, http://developer.intel.com/ial/upnp/

[Ber93] Philip A. Bernstein: *An Architecture For Distributed System Services*, Technical Report CRL 93/6, Cambridge Research Laboratory, März 1993

[Ber96] Philip A. Bernstein: *Middleware: A Model For Distributed System Services*, Communications Of The ACM, Vol. 39 No. 2, 1996, Seite 86

[Ber99] Thomas J. Bergin: *ENIAC*, Computing History Museum, American University 1999

[Blu99] *Bluetooth Protocol Architecture*, Signal: The official newsletter of the Bluetooth Special Interest Group, Issue No. 3, November 1999, Seite 5

[Blu01-1] *Specification of the Bluetooth System 1.1: Core*, Bluetooth Special Interest Group, 2001

[Blu01-2] *Specification of the Bluetooth System 1.1: Profiles*, Bluetooth Special Interest Group, 2001

[Bir96] Kenneth P. Birman: *Building Secure And Reliable Network Applications*, Manning Publications Co., 1996

[Bro98] David Brownlee et. al.: *DEC VAX Hardware Reference*, NetBSD Documentation Project, 1998, http://www.netbsd.org/Documentation/Hardware/Machines/DEC/vax/

[But97] David R. Butenhof: *Programming with POSIX threads*, Addison Wesley Publishing Co., 1997

[CDK01] George Coulouris, Jean Dollimore, Tim Kindberg: *Distributed Systems – Concepts And Design*, 3rd Edition, Addison Wesley Publishing Co., 2001

[Che00] Stuart Cheshire: *Dynamic Configuration of IPv4 link-local addresses*, Internet Engineering Task Force Memo, 2000, http://www.ietf.org/internet-drafts/draft-ietf-zeroconf-ipv4-linklocal-03.txt

[CWH00] Mary Campione, Kathy Walrath, Alison Huml: *The Java Tutorial*, 3rd. Edition, Addison-Wesley Publishing Co., 2000

[Dee89] Steve Deering: *RFC 1112 – Host Extensions For IP Multicasting*, Stanford University 1989, http://www.ietf.org/rfc/rfc1112.txt

[Dro97-1] R. Droms: *RFC 2131 – Dynamic Host Configuration Protocol*, Bucknell University 1997, http://www.ietf.org/rfc/rfc2131.txt

[Dro97-2] R. Droms: *RFC 2132 – DHCP Options and Boot Vendor Extensions*, Bucknell University 1997, http://www.ietf.org/rfc/rfc2132.txt

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

[Fla96] David Flanagan: *Java In A Nutshell*, O'Reilly & Associates, 1996

[Ger99] Neil Gershenfeld: *When Things Start To Think*, Henry Holt & Co., 1999

[GNU99] Free Software Foundation: *The GNU C Library Reference Manual*, Edition 0.09 DRAFT for Version 2.2 Beta of the GNU C Library, 1999, http://www.gnu.org/doc/libc-manual.html

[Goy98] Juan-Mariano de Goyeneche: *Multicast Over TCP/IP*, Linux Documentation Project, 1998, http://www.linuxdoc.org/HOWTO/Multicast-HOWTO.html

[GS00] Yaron Y. Goland, Jeffrey C. Schlimmer: *Multicast and Unicast UDP HTTP Messages*, Internet Engineering Task Force Memo, 2000, http://www.upnp.org/draft-goland-http-udp-04.txt

[Hus97] Eric Huss: *The C Library Reference Guide*, Release 1, Association for Computing Machinery, University of Illinois, 1997

[Int01] Intel Corporation: *UPNP SDK v1.0 for Linux Specification*, Intel Architecture Labs, 2001, ftp://download.intel.com/ial/upnp/upnpsdkarch.pdf

[KR77] Brian W. Kernighan, Dennis M. Ritchie: *The C Programming Language*, Prentice-Hall, 1977

[KP84] Brian W. Kernighan, Rob Pike: *The Unix Programming Environment*, Prentice-Hall, 1984

[Int00] Intel Corporation: *Silicon Showcase: About Moore's Law*, Intel Corp., 2000, http://www.intel.com/research/silicon/mooreslaw.htm

[LB98] Bil Lewis, Daniel J. Berg: *Multithreaded programming with pthreads*, Sun Microsystems, 1998

[Lew92] Donald A. Lewine: *POSIX programmer's guide: writing portable UNIX programs with the POSIX.1 standard*, O'Reilly & Associates, 1992

[Lin00] Linus Torvalds et. al.: *Linux Kernel 2.2.17 Source Documentation*, 2000, http://www.kernel.org/pub/linux/kernel/v2.2/

[LKR90] Dolenc, Lemme, Keppel, Reilly: *Notes On Writing Portable Programs In C*, 8th Revision, 1995

[Moo65] Dr. Gordon E. Moore: *Cramming More Components Into Integrated Circuits*, Electronics Vol. 38, No. 8, 1965, Seite 114

[Moo00] Dr. Gordon E. Moore: *The Continuing Silicon Technology Evolution Inside The PC Platform*, Intel Corp., 2000, http://developer.intel.com/update/archive/issue2/feature.htm

[MS00-1] *Universal Plug and Play Device Architecture*, Microsoft Corp., 2000, http://www.upnp.org/resources.htm

[MS00-2] *Understanding Universal Plug and Play*, White Paper, Microsoft Corp., 2000, http://www.upnp.org/resources.htm

[MS01] *Universal Plug and Play: Background*, Microsoft Corp., 2001, http://www.upnp.org/forum/default.htm

[Nor88] Donald A. Normal: *The Design Of Everyday Things*, Basic Books Inc., 1988

[OW00] Scott Oaks, Henry Wong: *Jini In A Nutshell*, O'Reilly & Associates, 2000

[Phi98] Michael Philippsen: *RMI: Verteiltes Programmieren in Java*, Universität Karlsruhe, 1998

[RKF93] Ward Rosenberry, David Kennedy, Gerry Fischer: *Understanding DCE*, 2nd Edition, O'Reilly & Associates, 1993

[Sch00] Jochen Schiller: *Mobilkommunikation: Techniken für das allgegenwärtige Internet*, Addison Wesley Publishing Co., 2000

[Seb93] Robert W. Sebesta: *Concepts Of Programming Languages*, Benjamin / Cummings Publishing Co., 1993

[Sha00] Stephen Shankland: *Jini's Bottleneck: What is holding up Sun's much-hyped technology?*, C-NET Special Report, 15.3.2000, http://news.cnet.com/news/0-1003-201-1559726-0.html

[SHM94] John Shirley, Wei Hu, David Magid: *Guide To Writing DCE Applications*, 2nd Edition, O'Reilly & Associates, 1994

[Sri97] Prashant Sridharan: *Advanced Java Networking*, Prentice Hall PTR, 1997

[Sim95] Alan R. Simon, Tom Wheeler: *Open Client/Server Computing And Middleware*, Academic Press, 1995

[Sin97] Pradeep K. Sinha: *Distributed Operating Systems – Concepts And Design*, IEEE Press, 1997

[Sta97] Stardust Technologies: *Writing IP Multicast-Enabled Applications*, http://www.ipmulticast.com/community/whitepapers/ipmcapps.html, 1997

[Sun98] Sun Microsystems: *Java Remote Method Invocation Specification*, 1.2.2, 1998

[Sun99-1] Sun Microsystems: *Jini Architecture Specification*, 1.1 alpha, 1999

[Sun99-2] Sun Microsystems: *Jini Discovery And Join Specification*, 1.1 alpha, 1999

[Sun99-3] Sun Microsystems: *Jini Discovery Utilites Specification*, 1.1 alpha, 1999

[Sun99-4] Sun Microsystems: *Jini Device Architecture Specification*, 1.1 alpha, 1999

[Sun99-5] Sun Microsystems: *Jini Lookup Service Specification*, 1.1 alpha, 1999

[Sun00-1] Sun Microsystems: *Java 2 Platform Micro Edition (J2ME) Technology For Creating Mobile Devices*, KVM White Paper, 2000

[Sun00-2] Sun Microsystems, Richard P. Gabriel, William N. Joy: *Sun Community Source License Principles*, 2000

[Sun00-3] Sun Microsystems: *The JavaSpaces Specification*, 1.1, 2000

[Tan95] Andrew S. Tanenbaum: *Verteilte Betriebssysteme*, deutsche Ausgabe, Prentice-Hall Verlag, 1995

[Tho01] Keith B. Thompson: *Jini Surrogate Architecture Specification*, Sun Microsystems, 2001

[W3C00] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, Dave Winer: *Simple Object Access Protocol (SOAP) 1.1*, World Wide Web Consortium W3C Note, 2000, http://www.w3.org/TR/SOAP/

[WT95] Thomas Wagner, Don Towsley: *Getting Started With POSIX Threads*, Department of Computer Science, University of Massachusetts, 1995

[Wei91] Mark Weiser: *The Computer For The 21st Century*, Scientific American, 9/1991, Seite 94

[Wei93] Mark Weiser: *Hot Topics: Ubiquitous Computing*, IEEE Computer, 10/1993, Seite 71

[Wei93-2] Mark Weiser: *The World Is Not A Desktop*, ACM Interactions, 11/1993, Seite 7

[Won99] Wylie Wong: *Java wars move to consumer front*, C-NET News, 12. April 1999, http://news.cnet.com/news/0-1006-200-341023.html

[ZG99] Andreas Zeidler, Marco Gruteser: *Jini: Bezaubernde Geräte*, iX Magazin für professionelle Informationstechnik 4/1999, Seite 144

Herzlichen Dank für technische Unterstützung und interne Unterlagen:

- Bjorn Wesen, Axis
- Thomas Dorn, Bosch-Siemens Haushaltsgeräte
- Keith B. Thompson, SUN Microsystems