In diesem Kapitel:

- *Warum Jini? Die Vision*
- *Installation und Test der Installation*
- *Was ist eigentlich so schwierig an der Netzwerkprogrammierung?*
- *Neue Konzepte*
- *Das Jini Modell*
- *Zielsetzungen*
- *Was ist Jini nicht?*
- *Die fünf Grundprinzipien*

1. *Jini Übersicht*

1.1. Einleitung – Die Vision von Jini

Obwohl Java für die Unterhaltungselektronik entwickelt wurde, wird es im allgemeinen als Werkzeug für das Erstellen von Applets erachtet. Doch die ursprüngliche Vision - die Möglichkeit des Austauschens von Code zwischen Geräten unabhängig von der jeweils verwendeten CPU, die hohe Sicherheit, Kompaktheit usw. - steht heute im Vordergrund.

Diese Vision erfordert Mechanismen, die wir normalerweise nicht mit Desktop Computern in Zusammenhang bringen:

- die Software Infrastruktur dieser Geräte muss äusserst **robust** sein. Toaster und intelligente Waschmaschinen oder Geschirrspühler dürfen nicht mit einer Fehlermeldung wie "Abbrechen, Wiederholen, Ignorieren?" abbrechen. Die Software muss das Entwickeln zuverlässiger Systeme nicht nur ermöglichen, sondern auch fördern.
- die Geräte müssen echtes und müheloses **Plug & Play** unterstützen. Sie sollten die Internet Äquivalenz des Telefons sein: Sie schliessen sie einfach an, und schon funktioniert sie. Dieser Bedarf an Plug & Play bedingt einige spezielle Anforderungen. Zum ersten müssen die Geräte einfach verwendbar sein. Analog zum Telefon verfügen typische Endbenutzergeräte über eine beschränkte Anzahl von Schnittstellen. Selbstverständlich wird nicht jedes Gerät im Haushalt über eine Maus und einen hochauflösenden Bildschirm verfügen. Dies ist häufig auch gar nicht wünschenswert, da es die Geräte unnötig verkomplizieren würde. Zum zweiten muss die Handhabung dieser Geräte einfach sein. Wir möchten sie anschliessen und benutzen, ohne zunächst IP Adressen zu konfigurieren, Gateways und Router einzurichten, Treiber zu installieren (bzw. zu deinstallieren) usw. Aktualisierungen spielen in diesem Zusammenhang eine wichtige Rolle - wenn die Aktualisierungen von Software für die Fernsehgeräte in einem grossen Hotel eigens von einem Administrator vorgenommen werden muss, wird dies möglicherweise gar nicht erfolgen.
- im Zeitalter des Internets müssen Software-Systeme **erweiterbar** sein. Auch das Erstellen der Software einzelner Geräte wie beispielsweise der CPU einer Microwelle kann bereits recht aufwendig sein, doch bei vernetzten Geräten, die in der Lage sein müssen, mit beliebig vielen gleichwertigen Geräten innerhalb des Internet zu kommunizieren, vervielfachen sich die potentiellen Probleme. Noch problematischer ist möglicherweise die Tatsache, dass Entwickler die Eigenschaften künftiger Geräte nicht vorhersehen können. Wir erwarten von unseren Geräten und Software-Diensten, dass diese ohne erhebliche Anpassungen des Netzwerkes zusammen arbeiten können.
- Geräte dieser Art bilden **spontane** Gemeinschaften : stellen Sie sich eine Digitalkamera vor, die mit einem Farbdrucker verbunden wird. Wir wollen einfach nur die

Schnappschüsse ausdrucken, ohne den Geräten die Daten des jeweils anderen vorher anzugeben. Häufig würde der Aufwand für das wechselseitige Abstimmen der Geräte den zu erwartenden Vorteil der gemeinsamen Verwendung zunichte machen. Genau deshalb schrecken wir davor zurück, vorhandene Netzwerke kurzfristig umzukonfigurieren. Der Umgang mit Netzwerken könnte weitaus flexibler und weniger festgelegt erfolgen, als dies gegenwärtig der Fall ist.

Der Vision zahlreiche Geräte und Softwaredienste, die einfach und zuverlässig zusammenarbeiten, haben sich bereits zuvor mehrere Forscher und Manager der IT-Branche angenommen. Mark Weiser vom Xerox Palo Alto Research Center bezeichnete diese Vision als "ubiquitäre Datenverwaltung". Damit ist die sofortige Verfügbarkeit und Verwendbarkeit an Netzwerken angeschlossener Geräte gemeint. Bill Joy, einer der Gründer von Sun und Entwickler des Virtual Memory Managers von Berkely Unix (seine Diplomarbeit) ging davon aus, dass auch künftig traditionelle Desktop-Computer verwendet, diese jedoch durch intelligente "Vorrichtungen" in Haushalten und Fahrzeugen ergänzt werden.

Mit dieser Vision begann bei Sun ein Entwicklungsteam mit der Entwicklung einer Infrastruktur, die Java zur vollen Entfaltung bringen sollte - unter Beibehaltung der Grundprinzipien von Java:

- Zuverlässigkeit
- Handhabbarkeit
- Erweiterbarkeit
- Spontaneität

Das Projekt erhielt die Bezeichnung Jini (es musste mit J wie Java anfangen). Chefentwickler war Jim Waldo, der bei der Entwicklung von RPC, CORBA, RMI beteiligt war, sich also bestens auskennt.Ann Wollrath aus dem RMI Team und Ken Arnold (Transaktions- und Speichermodelle für Jini) sowie Bob Scheifer, der einst das X -Konsortium (X/Open) leitete und im Jini Projekt für Lookup- und Discovery-Protokolle zuständig war, vervollständigten das Jini Kernteam.

Es gibt verschiedene Programmiermodelle, die der Lösung des Problems der verrteilten Datenverarbeitung dienen sollten. Die Jini-Designer hätten auf einem dieser Modelle aufbauen können - beispielswiese der temporalen Logik, schwachkonsistenten Datenbanken oder Agenten - um sich auf diese Weise den Zielsetzungen für Jini zu nähern. Doch Systeme dieser Art erfordern zumindest einen völlig neuen gedanklichen Ansatz, da ihre Programmiermodelle so radikal sind.

Die Grundbegriffe von Jini sind jedem Java-Programmierer bereits bekannt. Jini ergänzt Java um einige Vorrichtungen, die erforderlich sind, um Java die Welt der einfachen verteilten Datenverarbeitung zu eröffnen, und ermöglicht Entwicklern den Einstieg ohne aufwendiges Verinnerlichen grundlegend neuer Programmiermodelle.

1.2. Installation von Jini

Auf dem Server befindet sich eine Version der Jini Software. Sie können die aktuelle Version auch vom Java Site runterladen : http://java.sun.com/products/jini .

1.2.1. Grundinstallation

Sinnvollerweise installieren Sie die Software im Root Verzeichnis (Unzip mit -d : alle Subverzeichnisse werden korrekt angelegt.

Im Verzeichnis sollte sich eine bat-Datei befinden:

```
@echo off
set CLASSPATH=c:\jini1_0_1\lib\jini-core.jar;c:\jini1_0_1\lib\jini-ext.jar;c:\jini1_0_1\lib\sun-util.jar;%CLASSPATH%
Rem
java -cp c:\jini1_0_1\lib\jini-examples.jar com.sun.jini.example.service.StartService
```

mit der das Jini Konfigurationsprogramm gestartet werden kann.

Die CLASSPATH Variable sollte auch auf das Lib Verzeichnis von Jini verweisen. Sie müssen diese in der Systemumgebung passend ergänzen.

1.2.2. Die Laufzeitdienste von Jini starten

Zur Laufzeit erfordert Jini eine gewisse Netzwerkinfrastruktur. Bei verteilter Datenverarbeitung würden diese Dienste vermutlich über einen dafür vorgesehenen Rechner angeboten oder aus Gründen der Redundanz mehrere Server eingesetzt. Um Jini effektiv nutzen zu können, benötigen Sie innerhalb ihres LAN zumindest eine aktive Instanz dieser Dienste. Andere Dienste sollten auf allen oder zumindest mehreren Hosts implementiert sein. Zum Entwickeln kann man die Dienste auch lokal installieren.

Die Jini Software wird mit den für das Verwenden von Jini grundlegenden Diensten ausgeliefert und umfasst darüber hinaus einige optionale Dienste. In der Liste unten sind die unterschiedlichen für das Ausführen von Jini-Anwendungen innerhalb des Netzwerkes erforderlichen Dienste aufgeführt. Zusätzlich umfasst das Jini-Paket zwei weitere Dienste - den Jini-Transaktionsmanager und den "Speicherdienst" JavaSpaces - diese sind für den Einsatz von Jini nicht unbedingt erforderlich.

Jini erfordert folgende Dienste:

- einen einfachen Web-Server.
  Jini erfordert eine solche Vorrichtung, da das Herunterladen von RMI-Code mit Hilfe des HTTP-Protokolls erfolgt. Jini wird bereits mit einem äusserst einfachen HTTP-Server ausgeliefert, der ausreichend ist, um Anwendungen mit dem benötigten Code zu versorgen. Es ist üblich, auf jedem Host, der anderen Anwendungen herunterladbaren Code zur Verfügung stellen soll, einen HTTP-Server zu betreiben.

- den RMI-Aktivierungs-Daemon:
  Trotz seines furchteinflössenden Namens ist der Aktivierungs-Dienst sehr einfach verwendbar und stellt einen äusserst nützlichen Bestandteil der Java-Infrastruktur dar. Dieser Prozess ermöglicht Objekten, die nur selten verwendet werden, gewissermassen "einzuschlafen", um bei Bedarf automatisch geweckt zu werden. Eine solche Vorrichtung wird beispielsweise bei der Programmierung von Remote-Systemen häufig verwendet, wenn mit langlebigen Server-Objekten gearbeitet wird, die nur selten zum Einsatz kommen. Der RMI-Aktivierungs-Daemon reguliert das Wechseln zwischen den aktiven und inaktiven Zuständen solcher Objekte und wird von den andern grundlegenden Jini-Laufzeitdiensten ausgiebig genutzt. Der Aktivierungs-Daemon muss zumindest auf jedem Host installiert werden, auf dem ein Lookup-Dienst (s.u.) beschrieben wird.

- einen Lookup-Dienst:
  Sie werden feststellen, dass der Lookup-Dienst die eigentliche Schlüsselkomponente von Jini darstellt. Er dient der Verwaltung der gegenwärtig aktiven Jini-Dienste, die innerhalb eines LAN verfügbar sind. Sun bietet eine eigene Implementierung eines Lookup-Dienstes für Jini an. Aus der Jini-Dokumentation geht hervor, dass der mit dem JDK ausgelieferte RMI-Registrationsserver als Lookup-Dienst verwendet werden kann. Dies ist zwar möglich, jedoch nicht empfehlenswert - der mit Jini ausgelieferte Lookup-Dienst ist weitaus leistungsfähiger.

Das klingt nach viel Arbeit, ist es aber nicht:
Jini wird mit einem einfachen Verwaltungsprogramm, mit grafischer Oberfläche, ausgeliefert. Allerdings kann man alle Dienste auch direkt starten.

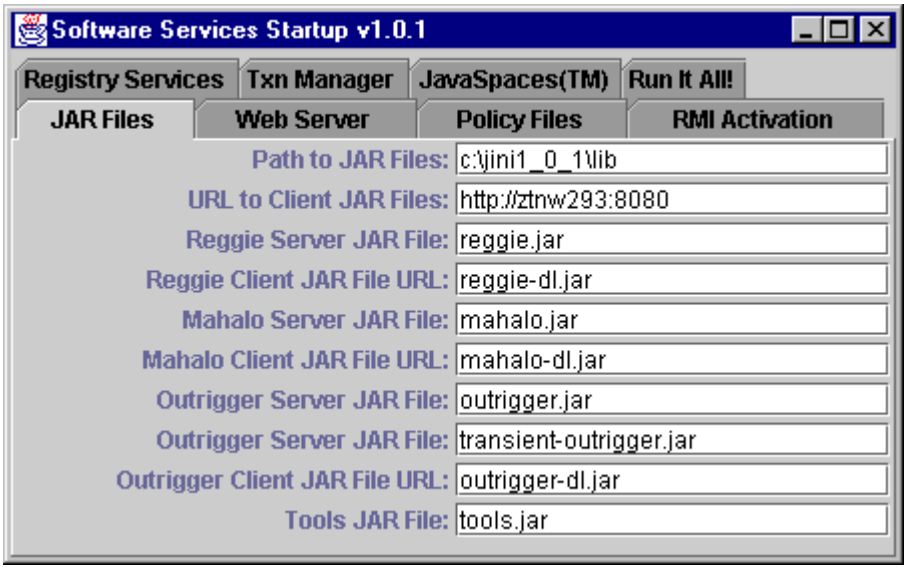
1.2.2.1. Die erforderlichen Dienste mit Hilfe des GUI starten

Wie bereits erwähnt, wird eine bat Datei mitgeliefert, die oben im Skript auch angezeigt wird. Falls Sie gerne tippen, können Sie auch Befehle direkt eintippen:

```
java -cp c:\jini1_0_1\lib\jini-examples.jar
com.sun.jini.example.service.StartService
```

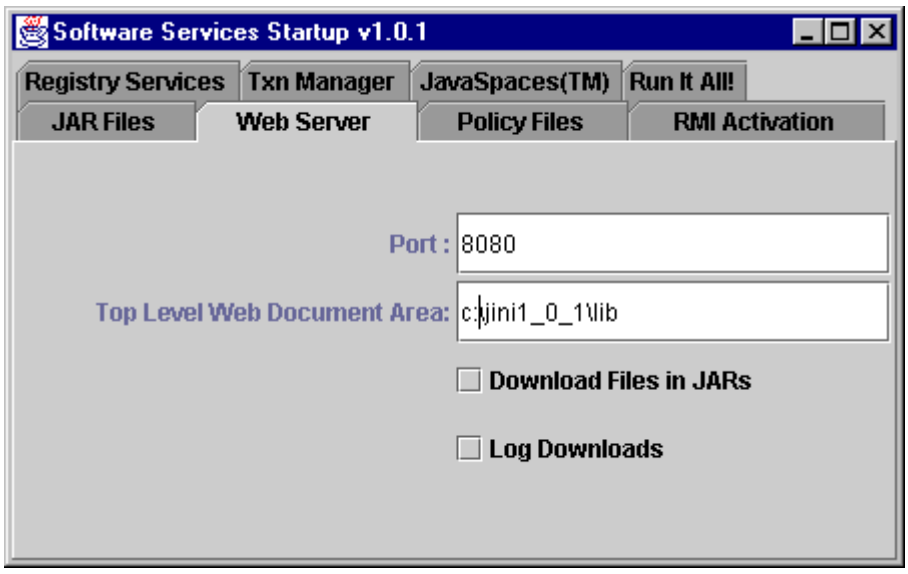
Sie müssen das Verzeichnis gegebenenfalls Ihrer Installation anpassen.

Sie sollten dann folgendes Bild sehen:

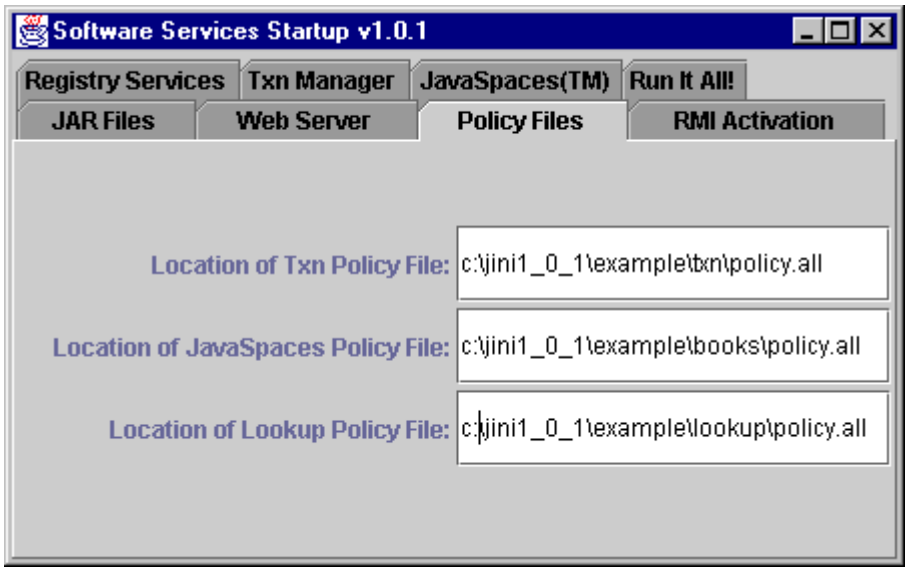


Sie müssen den Pfad auf die jar-Dateien gegebenenfalls anpassen. Die andern Angaben sollten korrekt sein.

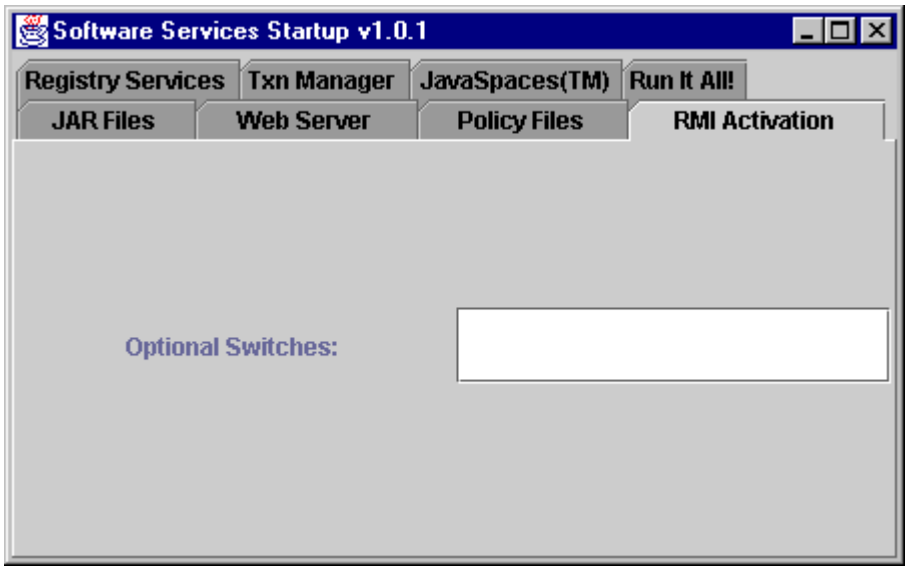
Und nun geht's los: als erstes starten wir den Web-Server. Dazu wechseln wir zur Registerkarte "Web Server". Auch hier müssen Sie gegebenenfalls den Pfad und den Port Ihren Gegebenheiten anpassen.



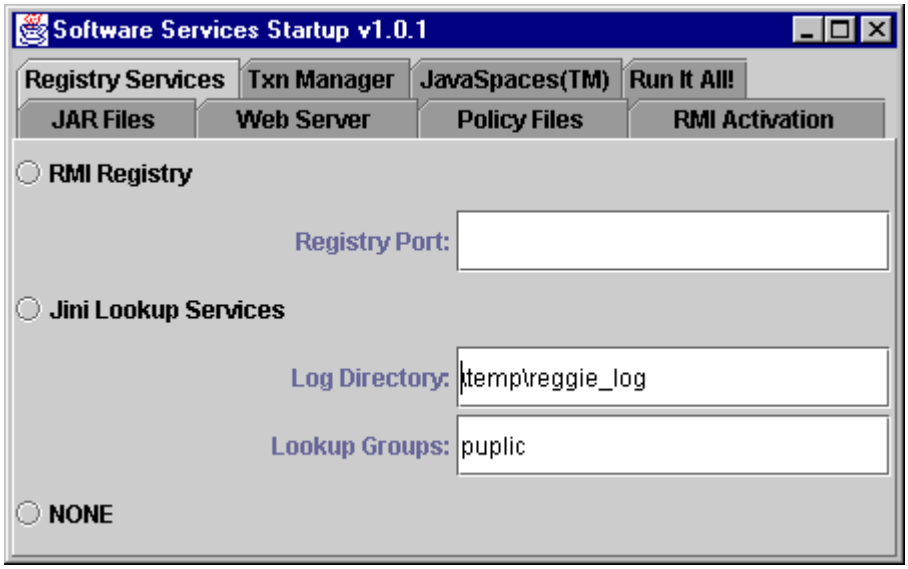
Nachdem wir diese Angaben angepasst haben, wechseln wir zur Karteikarte "Policy Files" und führen dort die entsprechenden Anpassungen durch.



Die nächste Registerkarte namens "RMI Activation" steuert das Starten des RMI Aktivierungsdaemons. Da wir keine Switches setzen wollen, brauchen wir keinerlei Änderungen vorzunehmen.

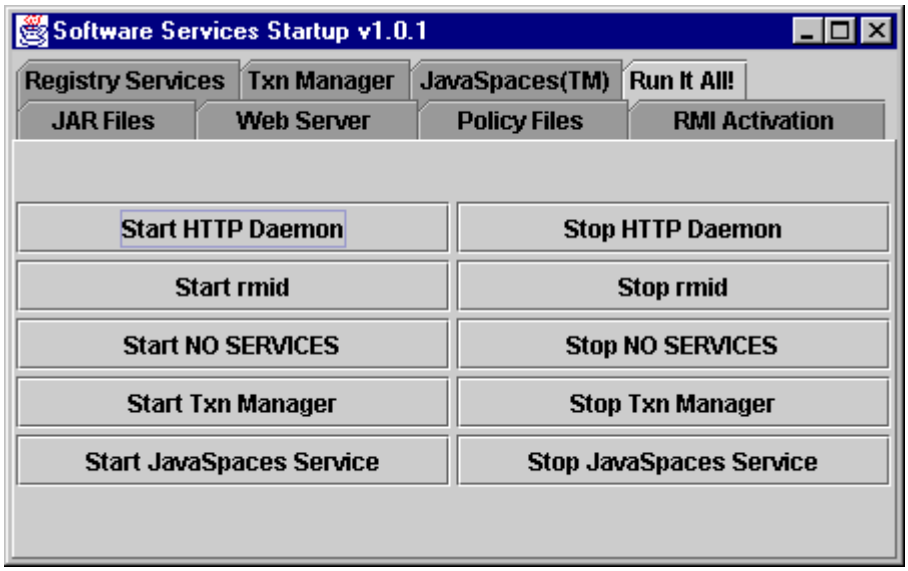


Auf der nächsten Registerkarte namens "Registry Services" passen wir wieder das Verzeichnis den konkreten Gegebenheiten an. Das Einzige was Sie eventuell anpassen müssen ist auch hier das Verzeichnis.



Die beiden Registerkarten - TRANSACTION MANAGER und JAVA SPACES - lassen wir im Moment weg, weil wir sie nicht benötigen.

Jetzt können wir die einzelnen Dienste starten:
unter "Run It All" stehen Start- und Stop- Knöpfe für die eben konfigurierten Dienste zur Verfügung.



1.2.2.2. Erforderliche Dienste von der Befehlszeile aus starten

Sie können alle Dienste auch direkt, ohne GUI starten. Als Beispiel betrachten wir den Web-Server:

```
java -jar c:\jini1_0_1\lib\tools.jar
     -port 8080 -dir \jini1_0_1\lib
     -verbose
```

Durch die Angabe von verbose sieht man, ob die jar-Dateien mt Hilfe des Web-Browsers heruntergeladen werden.

1.2.3. Starten eines Beispielprogrammes

Damit wir sicher sind, dass die Installation korrekt ist, starten wir ein einfaches Beispielprogramm, welches mitgeliefert wurde. Das Beispiel ist eine einfache Browser Anwendung. Bevor das Programm gestartet wird, müssen die eben besprochenen Dienste gestartet werden. Dabei muss für den Registry Service ein Service ausgewählt werden. Im Start Menü muss auch der Reggie Dienst ("Start NO SERVICES") gestartet werden.

Hier die Starterdatei: (start_browser.bat)

```
@echo off
Rem  Start_Browser.Bat
java -cp c:\jini1_0_1\lib\jini-examples.jar
-Djava.security.policy=c:\jini1_0_1\example\browser\policy
-Djava.rmi.server.codebase=http://127.0.0.1:8080/jini-
examples-dl.jar
com.sun.jini.example.browser.Browser
```

Der Browser überprüft das Netzwerk und sucht nach Jini Diensten. Es dauert in der Regel eine Weile, bis sich die Applikation meldet.

Falls Sie die Services korrekt gestartet haben, sollte der Browser mindestens einen der eben gestarteten Dienste finden.

1.2.4. Wenn was schief läuft - mögliche Fehler und deren Behebung

Da Jini einige Dienste benötigt, kann auch einiges schiefgehen. Die meisten Fehler treten wegen Konfigurationsproblemen auf oder Security (Policy) Problemen, speziell weil ab Java 1.2 die Zugriffsrechte mit dem Policytool gesetzt und in eine Datei abgespeichert werden können. Das Policytool ist aber eher nutzlos, wenigstens aus meiner Sicht, da keine Liste der möglichen Parameter angezeigt wird. Und da liegt in der Regel das Problem: man muss zuerst in den Spezifikationen die möglichen Properties nachschauen; ohne diese Liste kann man gar nichts machen.

1.2.4.1. Class Not Found (zum Beispiel Stub class)

Beim Testen von RMI und Jini Applikationen tritt häufig diese Fehlermeldung auf. Diese Fehlermeldung 'ClassNotFoundException' wird zum Beispiel geworfen, wenn eine Applikation ein serialisiertes Objekt von einem andern Programm erhält, aber die Implementation dieses Objekts (das Classfile) lokal nicht vorhanden ist.

Dafür gibt es drei Gründe:

1. **Codebase Probleme**
   meistens hat dieser Fehler etwas mit der CODEBASE zu tun. Der Server - das Programm, welches das Objekt sendet - muss die Codebase setzen, um dem Programm mitzuteilen, wo das Classfile gefunden werden kann.
   Falls Sie Kontrolle über das Server Programm haben und nicht einfach von irgend einem Web Objekte serialisiert herunterladen, sollten Sie die Definition der Codebase prüfen und allenfalls korrekt setzen (siehe weiter unten für zusätzliche Informationen zu CODEBASE).
2. **der Security Manager ist nicht installiert**
   Java Programme können entfernte Objekte (Class Dateien) nur dann laden, wenn eine Security Manager definiert wurde. Falls der Security Manager fehlt, können Classdateien

lediglich lokal geladen werden / lediglich lokale Classdateien geladen werden.
Typischerweise muss der Security Manager `java.rmi.RMISecurityManager` geladen werden, zum Beispiel mit Hilfe einer Property Spezifikation.

3. **der HTTP Server exportiert nicht korrekt**
   falls CODEBASE korrekt gesetzt ist, der Security Manager korrekt definiert wurde und immer noch kein Classfile gefunden wird, könnte es sein, dass der HTTP Server nicht korrekt konfiguriert wurde.
   Damit der (Mini-) Webserver von Sun (im Jini Kit) Dateien korrekt findet, müssen die Verzeichnisse folgendermassen definiert sein:
   zum Finden des Classfiles `joller.uster.Beispiel` (.class) muss das Classfile im Verzeichnis `root\joller\uster` vorhanden sein, wobei `root` das Webroot ist.
   Die Codebase wäre für diesen Fall http://mein_host:port/ ; (zum Beispiel http://ztnw293.hta.fhz.ch:8080/). Falls Sie auf Ihrem PC wie ich auf meinem mehrere IP Adressen haben, können Sie mit dem Testprogramm `TestIP` rasch nachschauen, wie java.net die IP-Adressen sieht. RMI und Jini sind in Java geschrieben, verwenden also lediglich Basisklassen:

```
localhost/127.0.0.1
ztnw293.hta.fhz.ch/62.2.88.96
client88-96.hispeed.ch/62.2.88.96
ztnw293.ztl.ch/147.88.74.117
```

   speziell zu beachten ist der Slash am Ende!
   Falls Sie Ihre Classdateien in eine Jar geziggt haben, können Sie das Archiv angeben als: http://mein_host:port/mein_jar_file.jar.

1.2.4.2. AccessControlException

Dieser Fehler ist noch häufiger, speziell bei Anfängern. Die Ursachen können sein:

1. **unkorrrekte Policy Spezifikation**
   Die Hauptursache für AccessControlExceptions ist eine falsche oder keine Policy- Datei. Sobald Sie einen Security Manager installiert haben, und Jini tut dies automatisch. kann das Programm sogut wie nichts ohne klar definierte Zugriffsrechte tun (nicht wie im Falle der RMI Programme, die, wie Sie sicher noch wissen, den Security Manager in der Regel am Programmanfang definieren, und wo der Security Manager auch leicht unterdrückt werden kann). Der Scurity Manager kontrolliert, was herunter geladen werden kann *und* was mit lokalem Code angefangen werden kann.
   Die Zugriffsrechte werden mit Hilfe einer Policy Datei spezifiziert, welche als Property an die Programme mitgeteilt wird. Damit Jini sauber arbeiten kann, benötigen seine Dienste die `DiscoveryPermission` für die Gruppen, welche gesucht werden, oder `AllPermission`, falls Security keine Rolle spielt. Für produktive Systeme ist dies natürlich keine Lösung!

1.2.4.3. Proxy Object is null

Dieser Fehler ist zum Teil nur im Stack Trace ersichtlich. Die Ursachen können sein:

1. **Service existiert nicht**
   es gibt zwei Gründe für diese Fehlermeldung des Lookup Services.
   Falls man eine einfache Form der `lookup`() Methode verwendet (jene, welche die "hits" nicht berücksichtigt), kann es vorkommen, dass Sie obige Fehlermeldung erhalten, weil kein registrierter Dienst mit den Search Kriterien übereinstimmt..

2. **Code wird nicht korrekt herunter geladen**
   Falls der Service korrekt registriert ist, kann es immer noch sein, dass entweder die Codebase oder die Policy nicht korrekt definiert sind.
   Falls Sie zum Beispiel eine komplexere Form der `lookup`() Methode einsetzen, welche die maximale Anzahl Übereinstimmungen beim Search berücksichtigt, und das Programm ein `ServiceItem` Objekt erhält aber das Proxy Objekt null ist, dann ist dies ein Hinweis auf eine falsche Konfiguration
   Dies impliziert in der Regel, dass der Client nicht in der Lage ist, den Code (die Class Datei) herunter zu laden und damit das `Proxy` Objekt nicht aufbauen kann. Dies ist wieder ein CODEBASE Fehler oder eine falsch konfigurierte Security Policy, oder Ihre Applikation setzt die Codebase nicht korrekt, oder die Codebase zeigt irgendwo hin, aber nicht dorthin, wo sie sollte.

1.2.4.4. No lookup services found

Die Fehlermeldung hat verschiedene Gründe:

1. **Lookup Service gehört nicht in die aktuelle Gruppe**
   stellen Sie sicher, dass mindestend ein Lookup Service im Netzwerk zur aktuellen Gruppe gehört.
   Meistens verwendet man einfach eine einzige Gruppe "`public`", deren Namen eine lere Zeichenkette ist. Damit der Registrierungsdienst, 'reggie' genannt, dieser Gruppe angehört muss die Zeichenkette `'public'` als Parameter angegeben werden (nicht "").
2. **Lookup Proxy wird nicht korrekt exportiert**
   Clients und Server greifen mit Hilfe von Proxyobjekten auf die Lookup Services zu. Daher muss der Lookup Service Proxy vorhanden sein. Der Lookup Dienst muss die Codebase Variable korrekt setzen und das Proxy Objekt muss auf dem Web Server erhältlich sein, im Falle der Sun Implementation ist dies das Jar Archiv *reggie-dl.ja.r*
3. **Lookup Service kann nicht starten, weil der Activation Daemon nicht läuft**
   Sun's Implementation des Lookup Service basiert auf dem RMI Activation Daemon. Das `rmid` Programm muss *korrekt* laufen, und das kann eine Weile dauern. In der Regel muss man nach dem Starten des `rmid` einige Sekunden warten, bevor der Lookup Service gestartet werden kann.
4. **Security Problem verunmöglichen das Herunterladen des Lookup Proxy**
   Damit ein Java Programm Class Dateien herunter laden darf, muss ein Security Manager installiert sein. Verifizieren Sie das Policy File und den Security Manager..
5. **Sie haben Probleme mit dem StartServices GUI**
   Das GUI Programm zum Starten der Dienste wies einen Fehler auf: bei Angabe einer URL wandelte das Programm die Slashes ('/' in Backslashes um '\', weil das Programm erkannte, dass es auf Windows lief. In einer URL Angabe sollten aber Slashes verwendet werden. Sie sehen normalerweise den Aufruf, der durch das GUI generiert wird. Falls dieser Aufruf falsch ist, setzen Sie einfach ein lookup.bat Start-Batch Skript auf. Sonst kann das Proxy Objekt für den Lookup Dienst nicht herunter geladen werden und die `LookupDiscovery` Klasse somit nicht aufgerufen werden, da die Deserialisierung des Registrierungsproxys nicht funktioniert.

1.2.4.5. Leases werden bei Verwendung des LeaseRenewalManager nicht erneuert

Dieser Fehler in Anwendungsprogrammen lässt sich in der Regel leicht geheben.

1. **renewFor statt renewUntil (lease.ANY in renewUntil)**
   Meistens werden Leases durch Instanzierung eines `LeaseRenewalManager` gesetzt

und erneuert, indem man die Methode `renewUtil`() mit dem Parameter `Lease.ANY` aufruft. Dadurch wird dauernd das Leasing erneuert, bis der `LeaseRenewalManager` verschwindet, was passiert, falls der Client aktiv wird.

Ein Problem besteht im Verwechseln der Methoden `renewUntil`() mit `renewFor`(). Die `renewFor`() Methode erneuert nicht dauernd; der Parameter `Lease.ANY` to bewirkt nicht.

1.2.4.6. Sie erhalten keine RemoteEvents

Mögliche Fehlerursache:

1. **der Listener wurde nicht korrekt exportiert**
   falls Sie den RemoteEventListener korrekt implementiert haben (Unterklasse von `UnicastRemoteObject`, etc.) aber der Programmcode nie von einem remote Event aufgerufen werden, dann könnte es sein, dass die Listener Klasse nicht korrekt exportiert wird.

   Eventuell setzt Ihr Programm die CODEBASE nicht korrekt und das Programm findet deswegen die Klassendateien nicht. Dies kann an der Konfiguration des Web-Servers liegen. Normalerweise tritt dann auch ein "Class Not Found" wie oben beschrieben auf.

1.2.4.7. Lookup Services werfen RemoteException

Wie können Sie in diesem Falle vorgehen?

1. **Sie haben nicht aufgeräumt**
   Falls Sie ein Objekt mit Hillfe des Proxys herunter geladen haben, geschieht kommunikationsseitig nichts mehr oder nicht mehr viel. Sie können damit aber auch nicht informiert werden, ob der Lookup Service noch am Leben ist oder eben nicht. Daher kann es passieren, dass die Kommunikation schief läuft, weil ein Lookup Service gestorben ist. Versuche ihn weiter zu verwenden führen dann zu einer `RemoteException`. Falls dies passiert, sollten Sie im Programm eine Aufräumroutine starten. Dies kann mit Hilfe der `discard`() des LookupDiscovery Objekts geschehen.

1.2.4.8. Performance Probleme in Services

Dafür kann es natürlich beliebige Gründe geben. Hier sind einige der üblichen; aber es sind auch viele andere Gründe möglich:

1. **Aufruf von remote Methoden in einem separaten Thread**
   Falls ein Service eine remote Methode aufruft - beispielsweise einen RemoteEventListener in einem Client - dann wird diese Methode solange blockiert sein, bis der Client seinen Aufruf abgeschlossen hat. Falls der Client sich aufhängt oder der Aufruf lange dauert, müssen Sie mit grösseren Verzögerungen, im schlimmsten Fall Blockierungen rechnen.

   Die Lösung dieses Problems besteht darin, remote Methoden in zeitlich kurzen Threads abarbeiten zu lassen.

1.2.4.9. Leases werden nichtkorrekt behandelt, falls ein Service reaktiviert wird

Dafür können viele Ursachen schuldig sein. Eine der üblichen ist:

1. Probleme mit dem Lease Serialisierungsformat
   Sie müssen zum Beispiel darauf achten, dass das Zeitformat in den Fällen, in denen Sie persistente Daten verwalten, also Objekte auf die Harddisk speichern und später wieder

lesen, als absolute Zeiten gespeichert werden, nicht relativ zu einer Zeitreferenz.

Das Format kann man mit Hilfe der `setSerialForm`() Methode des `Lease` Interfaces setzen, um dieses Problem zu vermeiden.

1.2.4.10. Ihre Event Registratur "verschwindet"

1. **Event Handlers werfen Runtime Exceptions**
   Die "reggie" Lookup Service Implementation wird jeden Event Handler aus der Registrierung löschen, falls dieser eine Runtime Exception wirft.
   Es liegt also an Ihnen, dafür zu sorgen, dass sich solche Ausnahmen nicht ausdehnen können. Dies umfasst zum Beispiel `NullPointerException`. RMI wirft diese Exception, es wird aber eine `RemoteException` angezeigt. Diese wird an die Reggie weitergeleitet. Die Reggie wird dadurch den Event Handler entfernen.

1.3. Was ist eigentlich so schwierig beim Umgang mit Netzwerken?

Damit die Stärken der neuen Art von Netzwerken dautlicher wird, bietet es sich an, zuerst auf die Probleme der Netzwerkprogrammierung zu verweisen und Schwachstellen aufzuzeigen.

1.3.1. Klassische Vernetzung

Ein Grossteil der Geschichte der Programmierung vernetzter Systeme handelt von Versuchen, das Netzwerk und seine Komplexität zu verbergen. Bei der Betrachtung der "klassischen" Arbeit in Infrastrukturen vernetzter Systeme erkennen wir einige grundlegende Themen. Zum ersten bieten sie eine Möglichkeit an, Daten für deren Verarbeitung zu verschieben. Zum zweiten ermöglichen Sie uns, innerhalb eines beschränkten Raumes anzugeben, was mit den Daten geschehen soll.

Die Möglichkeit des Verschiebens von Daten vor deren Verarbeitung ist erforderlich, da wir auch davon ausgehen können, dass der für die jeweilige Operation benötigte Code überall innerhalb des Netzwerkes verfügbar ist. Die Tatsache, dass der Code für eine bestimmte Plattform *übersetzt und installiert* werden muss, erschwert es, den Code überall ausführen lassen zu können. Daher ist es weitaus einfacher, die Daten zum Code zu verschieben. Doch auch das Verschieben der Daten wirft Probleme auf, da die Rechner innerhalb des Netzwerkes mit unterschiedlichen Notationen für die Byte-Reihenfolge, die Fliesskommazahlen, die Länge von Dateien .... arbeiten. Dafür wurden Werkzeuge wie XDR (External Data Representation, ein Protokoll von Sun, das Mitte der 80er Jahre für das Erwirken der Portabilität von Daten verwendet wurde) definiert. Damit wollte man die Unterschiede der Netzwerk-Infrastruktur verbergen: das Datenformat wurde einheitlich oder konnte einheitlich gemacht werden, die Daten konnten überall im Netzwerk verarbeitet werden.

Was nach der Übertragung der Daten geschehen soll, wurde häufig über einen Remote Procedure Call (RPC) oder mit Hilfe eines Systems für Remote Objekte angegeben. RPC Systeme versuchen, (dem Programmierer) bei Aufrufen durch Funktionen anderer Rechner vorzuspielen, es würde sich um Aufrufe einer lokalen Funktion innerhalb des gleichen Adressraumes handeln. Systeme für Remote-Objekte wie beispielsweise CORBA oder DCOM erhöhen die Ebene des Programmierens vom Funktionsaufruf der Methoden von Objekten, orientieren sich jedoch nach wie vor an der Sematik lokaler Aufrufe. *Qualitativ* unterscheiden sie sich nicht von der RPC Technologie der letzten 20 Jahre.

Für diese Versuche, das Netzwerk zu verbergen, gibt es einen ausnehmend guten Grund: die Programmierer sind von jeher an das Verwenden grundlegender Techniken der lokalen Programmierung gewöhnt. Sie *wissen*, wie zuverlässige Einzelprogramme entwickelt werden können. Das Hinzufügen spezialisierter Programmiermodelle erfordert lediglich Zeitaufwand für das Erlernen und verursacht Kopfschmerzen, nicht eingehaltene Fristen und Ärger zu Hause.

All diese Systeme sollen das Netzwerk *transparent* gestalten und es somit aus der Sicht des Programmierers möglichst einfach "verschwinden" lassen.

1.3.2. Die Unübersichtlichkeit des Netzwerkes

Leider erweist sich diese Vereinfachung häufig als übertrieben. Diese Systeme gehen implizit von der Annahme aus, dass das Einführen einer Netzwerkverbindung zwischen zwei Software - Komponenten sich nicht auf die Korrektheit des Programms, sondern lediglich auf dessen Leistung auswirkt. Tatsächlich wird das Verbergen der Herkunft eines bestimmten Objekts in traditionellen Systemen für verteilte Objekte häufig als wünschenswert angesehen - ähnlich wie das Verbergen der Implementation einer Funktion hinter deren Schnittstelle in der objektorientierten Programmierung gewünscht wird. Das Verhalten einer Komponente im Falle einer Störung innerhalb des Netzwerks oder bei geringer Leistung wird als weiterer Aspekt ihrer Implementierung betrachtet und nicht mit ihrer Schnittstelle oder ihrer Zusammenarbeit mit dem Rest des Systems in Verbindung gebracht. Es hat sich erwiesen, dass die grösste Schwierigkeit beim Aufbau zuverlässiger verteilter Systeme nicht im Zusammenhang mit deren grundlegenden Aufgaben auftreten: das Verpacken von Daten in portable Formen, der Aufruf von Remote Prozeduren innerhalb des Netzwerkes usw. Als besonders heikel haben sich jene Aspekte des Netzwerks erwiesen, die vom Programmierer *nicht* vernachlässigt werden können - die Tatsache, dass die für den Zugriff auf eine Remote-Resource erforderliche Zeitspanne weitaus grösser sein kann, als dies beim lokalen Zugriff auf die gleiche Resource der Fall wäre, der Umstand, dass innerhalb von Netzwerken in Situationen Fehlfunktionen auftreten, die auf Einzelsystemen einwandfrei verarbeitet werden können, und das Faktum, dass vernetzte Systeme für teilweise Ausfälle von Berechnungen anfällig sind, die das System in einem unbestimmten Zustand versetzen können.

Der Informatiker Peter Deutsch (ex XEROX PARC, Sun) hat vor vielen Jahren bereits die grundlegenden Probleme erkannt und als "Die sieben Irrtümer der verteilten Datenverarbeitung" zusammen getragen:
"Praktisch jeder, der mit der Entwicklung einer verteilten Anwendung beginnt, geht anfänglich von den folgenden sieben Annahmen aus, die sich langfristig als falsch erweisen und *erheblichen Ärger sowie schmerzhafte Lernprozesse* zur Folge haben können.

1. **das Netzwerk ist zuverlässig**
2. **die Zugriffsverzögerung ist vernachlässigbar**
3. **die Bandbreite ist unbegrenzt**
4. **das Netzwerk ist sicher**
5. **die Topologie bleibt unverändert**
6. **es gibt einen Administrator**
7. **die Übertragunskosten sind vernachlässigbar"**

1.3.2.1. Leistung und Zugriffsverzögerung

Es versteht sich von selbst, dass der Zugriff auf eine Resource - eine Datei oedr ein Objekt - über ein Netzwerk langsamer erfogt als lokal. Wir alle wissen dies, und uns ist auch bekannt, dass diese Abweichungen der Zugriffszeit häufig mehrere Grössenordnungen betragen

können und gross genug sind, um einen Unterschied auszumachen, der eher *qualitativer* als *quantitativer* Art ist.

Möglicherweise denken Sie gerade: "Wozu die ganze Aufregung? Der Zugriff auf einen Cache-Speicher ist um Grössenordnungen schneller als jener auf Hauptspeicher, der Zugriff auf Hauptspeicher erfolgt weitaus schneller als jener auf die Festplatte. Handelt es sich nicht einfach nur um eine weitere Leistungseinbusse?"

Bei vernetzten Systemen spielt dieser Faktor jedoch eine wesentliche Rolle, da die variierende Leistung es häufig unmöglich macht, *fehlerhafte* Komponenten von *langsamen* zu unterscheiden. Der Hauptspeicher ist zwar wesentlich schneller als eine Festplatte, doch die Leistung beider Komponenten bewegt sich innerhalb weitestgehend kalkulierbaren Grenzen. Doch jeder, der bereits mit einem Web-Browser gearbeitet hat, kann Ihnen bestätigen, dass die Leistung des Netzwerks bereits innerhalb kurzer Zeitspannen extreme Schwankungen aufweisen kann. Die Zugriffsverzögerungen innerhalb lokaler Systeme üben auf ein Programm einen nicht auch nur annähernd so grossen Einflus aus, wie der Zeitaufwand beim Zugriff auf Remote- Resourcen.

Bei Systemen wie RPC und CORBA werden diese Aspekte der Leistung bislang gar nicht berücksichtigt. Das heisst nicht, dass die Netzwerke durch irgendwelche Zaubertricks um das Hundertfache beschleunigt werden sollten, damit die unterschiedlichen Zugriffszeiten keine Rolle mehr spielen. Dies ist selbstverständlich nicht möglich. Doch die Leistung und Zugriffsverzögerungen werden in den Programmiermodellen von RPC und CORBA in keiner Weise berücksichtigt - dieser Aspekt wird einfach gänzlich vernachlässigt und als "verdeckter" Faktor der Implementierung behandelt, den der Programmierer zu berücksichtigen hat.

Die Tatsache, dass es *möglich* ist, diese Unterschiede während des Entwickelns zu vernachlässigen, (da Remote-Zugriffe lokalen weitestgehend nachempfunden sind), bedingt, dass Entwürfe entstehen können, die nicht nur über äusserst schlechte Leistung verfügen, sondern möglicherweise nicht einmal in der Lage sind, mit Veränderungen der Leistung umzugehen, wie sie in verteilten Systemen auftreten. Wenn die Schnittslellen zwischen Objekten innerhalb eines Systems erstellt werden, ohne solche "Details" der Implementierung wie den Ort bzw. die Entfernung dabei zu berücksichtigen, ist es sehr wahrscheinlich, dass das System nicht besonders robust wird und seine Kommunikationsstrukturen es in indiskutabler Weise verlangsamen.

Statt die "Einfachheit" von Komponenten bis in ein spätes Stadium der Entwicklung zu vernachlässigen, müssen wir diese strukturellen Einzelheiten schon beim Entwurf der Schnittstellen für die Kommunikation zwischen den Objekten berücksichtigen.

1.3.2.2. Neue Arten von Fehlern

In vernetzten Systemen kann es zu Sörungen kommen, die bei Einzelsystemen nicht auftreten:
- Router können ausfallen
- Benutzer können - absichtlich oder versehentlich - ein Ethernet-Kabel aus der Wand ziehen
- und wichtige Rechner mit Namens- und Datenbank- Servern können abstürzen.

Code, der auf Einzelrechnern geschrieben wurde, ist auf die Handhabung der innerhalb einer vernetzten Umgebung auftretenden neuen Arten von Fehlern nicht vorbereitet.

Als einfaches Beispiel betrachten wir in diesem Zusammenhang eine Funktion, die dem Lesen einer Datei von der Festplatte dient. Dieses Code-Fragment wird möglicherweise innerhalb einer grossen Anwendung (etwa einer Tabellenkalkulation) vielfach verwendet und wurde seinem ursprünglichen Zweck entsprechend optimal programmiert. Es verfügt über ausgefeilte Mechanismen für die Erkennung und Handhabung von Fehlern jeder Art, von der Überlastung des Dateisystems bis zur Verletzung von Berechtigungen.

Stellen Sie sich vor, dass die gleiche Anwendung in einer neuen Situation ausgeführt wird, in der sich die Datei auf einer Festplatte eines am Netzwerk angeschlossenen Datei-Servers befindet. Der für den Umgang mit lokalen Fehlern aller Art ausgelegte Code wird nun auch mit Netzwerkfehlern konfrontiert. Der Datei-Server könnte abstürzen oder aus einem andern Grund unerreichbar werden. Im günstigsten Fall könnte die Anwendung die Störung so interpretieren, als würde es sich um einen lokalen Fehler handeln - beispielsweise wäre es möglich, dass eine "Server-unerreichbar" Bedingung als Verletzung einer Berechtigung eingestuft wird. Doch sie könnte ebensogut auf unerwartete Weise abstürzen. Selbstverständlich könnte die Anwendung auf Netzwerkfehler nicht allzugut reagieren. Falls der Programmierer diese Probleme mit einbezieht, könnte das Programm nach einem alternativen Datei-Server fragen, versuchen die Kommunikation erneut aufzubauen oder den Benutzer involvieren.

Die Fehlervorkehrung im verteilten System ist deswegen so schwierig, weil es viel schwieriger oder sogar unmöglich ist, festzustellen *was* die Ursache für das Problem ist.

Falls dies aber nicht möglich ist, ist eine "Verberge das Netzwerk" Vorgehensweise dem Problem viel angepasster.

1.3.2.3. Konkurrenzierende Zugriffe und Konsistenz

Die beiden Unterschiede zwischen lokalen und verteilten Systemen - Leistungs- und Zugriffs-Verzögerung sowie verschieden Arten von Fehlern - sind wichtig, aber nicht annähernd so verherend wie die Probleme mit konkurrierenden Zugriffen und Konsistenz, die bei der verteilten Datenverarbeitung auftreten. Innerhalb lokaler Umgebungen treten Fehler in Software- Komponenten (zum Beispiel Abstürzen einer Anwendung) entweder ganz oder gar nicht auf. Wenn ein vollständiger Rechner ausfällt, können Sie auf ihm, unabhängig davon, was im umgebenden Netzwerk geschieht, keine Arbeiten mehr durchführen.

Bei verteilten Systemen sieht dies jedoch anders aus, da die Möglichkeit besteht, dass einzelne Komponenten ausfallen, ohne dass andere Komponenten darauf adäquat reagieren oder auch nur darüber informiert sind, dass ein Fehler aufgetreten ist. Ein solches Vorkommen wird als ein *teilweiser* Ausfall bezeichnet und ist besonders tückisch. Teilweise Ausfälle bedingen einen Grossteil der in der verteilten Datenverarbeitung auftretenden Probleme. *Totalausfälle,* die in rein lokalen Systemen eher auftreten, sind *einfacher* handhabbar, da das System zumindest einen bekannten Zustand einnimmt.

1.4. Neue Konzepte der verteilten Datenverarbeitung

Java ist zwar kein Wundermittel, stellt aber in Bezug auf die Lösung der Schwierigkeiten bei der Programmierung verteilter Systeme jedoch einen gewaltigen Fortschritt dar. Java bietet für eine Vielzahl der in klassischen Systemen auftretenden Problemen Lösungen an. Weitere Schwierigkeiten der klassischen Systeme treten bei Verwendung von Java erst gar nicht auf. Dies ergibt sich aus der Tatsache, dass Java den Vorteil hat, ein sprachorientiertes System zu sein:

für die gesamte Plattform - von Laufzeit JVMs über die Klassenbibliotheken, den Bytecode-Verifizierer bis hin zur Sicherheitsarchitektur - werden Objekte in Java Bytecode erfasst. Dies trägt erheblich zur Vereinfachung diverser Aspekte des Erstellens verteilter Systeme bei. Im Gegensatz zu Systemen wie CORBA oder DCOM, bei denen Verteilung lediglich auf einen oder mehrere Sprachen aufgesetzt wird, geht der Java Ansatz verteilter Datenverarbeitung von der grundsätzlichen Verwendung einer *bestimmten* Sprache aus, welche mit einem speziellen Datenformat und einem bestimmten Codeformat arbeitet.

Das Verschieben von Daten stellt beispielsweise kein Problem mehr dar. Die Grösse und das Format grundlegender Typen wie Integer, Floating-Point, String, Felder und von Objekten sind in der Java Spezifikation festgelegt. Es sind keine komplizierten Umwandlungen zwischen den Formaten unterschiedlicher Rechner erforderlich. Für C gilt dies beispielsweise nicht, da die Grösse und das Format grundlegender Typen in dieser Sprache nicht festgelegt werden, um portable Formate für Daten zu erzeugen.

Java bestimmt nicht nur das Format grundlegender Typen, sondern sogar komplexer Objekte. Dies beinhaltet solche Feinheiten wie Subklassifizierung, Attribute für das Schützen von Methoden und Signaturen von Datentypen. Durch diese Verkehrssprache für Daten erübrigt sich die Verwendung von Programmen wie XDR für das Umwandeln von Daten in portable Formate.

Vor der Verwendung von Java wurde in der verteilten Datenverarbeitung ausserdem davon ausgegangen, dass Daten zum Code verschoben werden müssen, da die umgekehrte Vorgehensweise nicht praktikabel erschien. Mit Java kann Code genauso einfach von Rechner zu Rechner übertragen werden wie Daten. Dies gilt sogar dann, wenn die Rechner mit unterschiedlichen Betriebsystemen oder CPUs arbeiten. Das Java Bytecode Format ist das universelle Austauschformat für ausführbaren Code. Die Sicherheitsmechanismen von Java gewährleisten ausserdem, dass Code unmittelbar, nachdem er auf den Rechner verschoben wurde, ausgeführt werden kann, und zwar mit einem Grad der Sicherheit, der bei andern Ansätzen nicht gewährleistet werden kann. Obgleich das Betriebsystem möglicherweise das dynamische Verknüpfen von C-Code mit Hilfe von gemeinsam verwendeten Objektdateien oder Dynamic Link Libraries (dll's) erlauben, gehen Javas Möglichkeiten für sicheres dynamisches Laden weit über die Möglichkeiten anderer Umgebungen hinaus.

Mit Hilfe von Sicherheitsrichtlinien erlaubt Java beispielsweise das präzise Regulieren der Verwendung der Resourcen von Rechnern. Code kann digital signiert werden, um Entwickler von Klassen kryptografisch sicher anzugeben. Vermutlich noch wichtiger ist Javas hohe Typensicherheit, die gemeinsam mit dem Code-Verifizierer gewährleistet, dass fehlerhafter Code nicht ausgeführt wird, damit beim Ausfühen des von remote Rechnern geladener Code nicht der gesamte Prozess gefährdet wird. Bei der Verwendung von C kann es beispielsweise - sogar recht einfach - vorkommen, dass dynamisch geladener Code auf einen Null-Zeiger verweist und somit einen Feldindex ausserhalb des gültigen Bereichs angibt oder den Heap beschädigt. Dadurch kann die Ausführung des ansonsten langfristig laufenden Servers beendet werden. In Java ist das Verweisen auf nicht verwendete Daten nicht möglich, und es kann keine Referenz auf ein Objekt angegeben werden, auf das der Zugriff nicht explizit erlaubt wurde. Ausserdem kann eine Funktion in C willkürliche und möglicherweise unerwartete Werte für das Signalisieren von Fehlern liefern. In Java müssen Exceptions als Teil der Signatur der Methoden deklariert werden, damit Programme, die den Code verwenden, *gezwungen* werden, die Fehlermeldungen zu verarbeiten. Diese Sicherheitsattribute von Java beschränken die Fähigkeit dynamisch geladenen Codes, das System zu beschädigen.

Die Möglichkeit des Austausches von Code zwischen Rechnern eröffnet völlig neue Perspektiven. Server können nun von dazu berechtigten Clients dynamisch aktualisiert und verwaltet werden. Der Code von Agenten kann sich innerhalb des Netzwerkes bewegen und effektiv arbeiten, indem er sich direkt zu den benötigten Daten bewegt. Systeme, die auf der Grundlage streng festgelegter statischer Schnittstellen konstruiert werden, können durch neue Implementierungen jener Schnittstellen bei Bedarf dynamisch erweitert werden.

Für eine Vielzahl von Problemen klassischer Systeme bietet Java neuartige Lösungen an. Einige von ihnen wurden sogar gänzlich gelöst. Doch wie sieht es mit den wirklich schwerwiegenden Problemen verteilter Datenverarbeitung aus, die weiter oben angesprochen wurden?

Um zu verstehen, wie Java mit diesen harten Nüssen fertig wird, müssen wir uns zunächst mit dem Java Modell der strikt typisierten Datenverarbeitung vertraut machen.Wegzaubern kann auch Java diese bereits erwähnten Schwierigkeiten nicht, doch sein Modell der verteilten Datenverarbeitung unterscheidet explizit zwischen lokaler und remote Datenverarbeitung und bietet Programmierern jeweils Werkzeuge für deren Bewerkstelligung an.

1.4.1. Die Notwendigkeit strikter Typisierung

Im vorherigen Abschnitt wurde Javas Fähigkeit des Sendens von Code an laufende Prozesse bei entsprechendem Bedarf ausführlich erörtert. Doch ist dies wirklich nötig? Können Umgebungen, in denen Code sich frei bewegen kann, überhaupt noch verstanden und verwaltet werden? Als sprachbasierter Ansatz ist Java in der Lage, die Vorteile zu nutzen, welche strikte Typisierung, echte Objektorientierung und Polymorphismus in die lokale Datenverarbeitung einbringen und auf das Internet übertragen.

Bei der Remote Methode Invocation (RMI) wird die Festlegung von Datentypen der Programmiersprache Java verwendet, um die Schnittstelle für die Kommunikation mit Remote Geräten zu beschreiben. Sie scheint auf den ersten Blick nebensächlich zu sein, ist jedoch äusserst wichtig - lokale Datentypen, die Sie beim deklarieren einer Java Klasse oder Schnittstelle erstellen und auch die in den Java Bibliotheken vorhandenen Datentypen werden mit Hilfe der gleichen Mechanismen festgelegt, die auch für die der Kommunikation mit Remote Geräten dienenden Datentypen verwendet werden. Dies bedeutet, dass ein Protokoll für die Kommunikation mit remote Servern eine Java Schnittstelle ist, die mit Hilfe einer Klasse implementiert werden kann.

In Java sieht dies, wie Sie aus RMI wissen, wie folgt aus:

```
public interface RemoreServer extends Remote {
     public int getLength(String s) throws RemoteException:
}
```

In diesem Beispiel wird ein "Protokoll" für die Kommunikation zwischen zwei remote Prozessen festgelegt - es reguliert die Art der Interaktion zwischen den beiden Prozessen und bestimmt unter anderem, was gesendet wird, was Sie tun können und was Sie empfangen könnten. Dieses Protokoll wird selbstverständlich zur Laufzeit auf der Grundlage von Protokollen unterer Schichten wie TCP/IP implementiert - RMI ist für die Handhabung aller "Bequemlichkeits-" Aufgaben des Einrichtens und Beseitigens von Sockets usw. zuständig - doch die hier angegebene Definition umfasst die Spezifikation der Kommunikation zweier Prozesse auf der *Anwendungsebene*. Zusammenfassend ergibt sich aus dem Beispiel, dass ein Client eine einzelne Nachricht senden kann, die eine Zeichenkette an einen remote Server

überträgt. Als Antwort auf diese Nachricht kann der Server einen Integer Wert zurücksenden. Doch - und dies ist wohl das Grundprinzip von RMI - dieses Beispiel definiert ausserdem in der Programmiersprache Java eine erstklassige Schnittstelle, die von Klassen implementiert, durch multiple Vererbung von Schnittstellen mit andern Schnittstellen komponiert, als Parameter übergeben, von Methoden zurückgegeben und zur Laufzeit überprüft werden kann.

Andere Systeme wie beispielsweise die Interface Definition Language (IDL) von CORBA stellen das für die Kommunikation mit Diensten verwendete remote Protokoll als Datentyp der zu Grunde liegenden Sprache dar. Der Unterschied besteht darin, dass dieser Datentyp das Protokoll bei RMI vollständig festlegt. Es repräsentiert "das Wesentliche" der Definition des Protokolls und kann mit Java frei manipuliert werden, um das Protokoll zu ändern und zu erweitern. Systeme wie IDL verwenden hingegen einige externe Ausdrücke des Protokolls, um "das Wesentliche" des Protokolls zu beschreiben. Dabei könnte es sich beispielsweise um eine in einem sprachneutralen Format verfasste Protokolldefinition handeln. Das Protokoll wird als Java Typ formuliert, und der Java-Typ bzw. der Typ einer andern Sprache, spiegelt lediglich die Protokolldefinition wieder. Es handelt sich nicht um die Protokolldefinition selbst. Veränderungen des Java Typs, die sich aus einer IDL Spezifikation durch Subklassifizierung, Komposition oder Reflexion ergeben, beeinflussen das für die Kommunikation verwendete Protokoll nicht und fördern Unterbrechungen Ihres Codes.

Auch dies könnte nach einer spitzfindigen Unterscheidung klingen, obgleich deren Bedeutung kaum überbewertet werden kann. Indem Protokolldefinitionen in modernen objektorientierten Sprachen als Objekte erster Klasse behandelt werden, können wir die gesamte Leistungsfähigkeit der Objektorientierung, die in einzelnen Adressräumen so vorzüglich funktioniert, auf die Welt der verteilten Datenverarbeitung übertragen.

Auch Polymorphismus ist in dieser Welt möglich. Beispielsweise kann ich nun remote Schnittstellen definieren, welche Unterschnittstellen anderer remote Schnittstellen darstellen. Die üblichen Regeln können hier angewendet werden - ich kann die allgemeine Schnittstelle weiterleiten, bei Bedarf zur speziellen Schnittstelle übertragen und sie reflektieren oder `instanceof` zur Laufzeit verwenden, um festzustellen, mit welcher Art von Einheit ich mich unterhalte. Genauso wie Schnittstellen mehrere Implementierungen haben können, kann ich über beliebig viele Java-Klassen verfügen, die eine bestimmte remote Schnittstelle implementieren. Dabei sorgt der Compiler dafür, dass ich die entsprechenden Anforderungen einhalte und die benötigten Methoden mit den korrekten Signaturen implementiere. Analog zur Möglichkeit der Implementierung mehrerer lokaler Schnittstellen in einzelnen Klassen kann ich sogar Klassen entwickeln, die mehrere remote Schnittstellen implementieren.

1.4.1.1. Angewandte Remote-Polymorphie

Noch radikaler als das Unterstützen von Polymorphie in Remote-Schnittstellen ist die Unterstützung von vollwertiger Polymorphie für Typen von Objekten, die als Parameter übertragen oder als Ergebnisse von Remote-Methoden durch RMI zurückgegeben werden. betrachten Sie in diesem Zusammenhang das folgende Code-Beispiel:

```
public interface MaxtrixSolver extends Remote {
     public Matrix crossProduct(Matrix m1, Matrix m2)
          throws RemoteException;
}
```

Dieser Code definiert eine Remote-Schnittstelle für Matrizenberechnungen. Das Remote-Objekt, welches diese Schnittstelle implementiert, wird vermutlich auf einem Hochleistungsprozessor ausgeführt, der die Matrizenberechnung so schnell durchführen kann,

dass die für das Senden der Parameter und für das Empfangen des Ergebnisses erforderliche Zeit akzeptabel ist. Die Schnittstelle verwendet zwei Instanzen des Typs `Matrix`, berechnet deren Kreuzprodukt und liefert das Ergebnis als `Matrix` zurück.

Offensichtlich erwartet der Entwickler dieser Remote-Schnittstelle von Aufrufern, `Matrix`-Objekte zu senden und zu empfangen und die Sematik dieser Klasse zu kennen. Gesetzt der Fall, ich wäre ein Ingenieur für Signalverarbeitung und hätte meine eigene spezielle Klasse für die Verarbeitung von Matrizen entwickelt, die nur wenige Datenelemente umfasst. Meine Implementierung mit der Bezeichnung `SparseMatrix` ist weitaus platzsparender als die standardmässige Klasse `Matrix`, welche ich subklassifiziere, um sie überall in meinem Programm zu verwenden. Ich möchte in der Lage sein, die Vorteile des Remote-Objekts für Matrizenberechnungen zu nutzen, welches von einer andern Gruppe entwickelt wurde und auf einem sehr schnellen Server ausgeführt wird. Das Remote-Objekt, welches die Funktionen für die Berechnungen implementiert, kennt jedoch lediglich die Klasse `Matrix` und hat niemals etwas von meiner `SparseMatrix` gehört.

Im rein lokalen Fall würde mir die Polymorphie des Systems der Java-Typen selbstverständlich das Durchführen meines Vorhabens ermöglichen: ich könnte `SparseMatrix` in Code einfügen, der einfach `Matrix` erwartet, und die Funktion `crossProduct`() würde trotzdem einwandfrei funktionieren, da sie lediglich den `Matrix` "Teil" meiner Klasse verwendet. Eine geschickt formulierte `crossProduct`() Funktion wird für einen seiner Eingabeparameter sogar `clone`() aufrufen, damit ein neues Objekt des gleichen Typs für das Ergebnis verwendet wird, damit der Typ des Rückgabewerts mit dem Typ eines der Eingabewerte übereinstimmt.

Doch was geschieht im Fall der Remote-Verarbeitung?
Wenn Sie RMI verwenden, läuft alles genauso ab wie im lokalen Fall! Sogar wenn das Objekt auf dem Remote-Server für die Berechnung `SparseMatrix` gar nicht kennt und der Code für jene Klassen auf dem entsprechenden Rechner gar nicht vorhanden ist, kann RMI Instanzen von `SparseMatrix` verarbeiten und die Daten in diesen Instanzen übertragen. RMI wird sogar den Code dieser Klassen übertragen, falls dies erforderlich sein sollte. Es wird sicherstellen, dass `SparseMatrix` eine aktuelle Subklasse von `Matrix` ist und alle für die sichere und zuverlässige verwendung erforderlichen Methoden implementiert, wo `Matrix` auch immer noch verwendet wird.

Das soeben behandelte Beispiel ist äusserst praxisorientiert. Dynamischer, mobiler Code wird nicht nur für avangardistische Anwendungen wie beispielsweise aktive Agenten benötigt. Er kann auch zur Erweiterung der Möglichkeiten objektorientierter Programmierung innerhalb vernetzter Umgebungen einen wertvollen Beitrag leisten.

1.4.1.2. Entfernung bezieht sich auf die Schnittstelle und nicht auf die Implementierung

Den kurzen Code-Beispielen des vorigen Abschnitts können Sie ausserdem die weiteren Vorteile von RMI entnehmen. Die "Entferntheit" eines Objekts - seine Fähigkeit, von anderen Adressräumen aus verwendet zu werden - ist ein Bestandteil der Schnittstelle des Objekts. Dies ist ein weiterer kleiner Aspekt mit vielfältigen Auswirkungen. Er veranlasst den Entwickler, die Entferntheit von Anfang an einzuplanen, statt sie im nachhinein hinzuzufügen. Diese Vorgabe der RMI-Designer bedingt, dass Programmierer von vernherein über die Kommunikationsmuster nachdenken müssen, die ihre Anwendungen unterstützen werden. Die Entferntheit einer Methode muss deklariert werden. Sie ist kein verborgener

Bestandteil der Implementierung des Objekts. Wie alles andere innerhalb des Systems der Datentypen von Java kann die Entferntheit eines Objekts zur Übersetzungszeit strikt typisiert und zur Laufzeit überwacht werden.

RMI erfordert, dass alle Schnittstellen, die ein Protokoll für Remote-Kommunikation festlegen, die Schnittstelle `java.rmi.Remote` erweitern.Diese Schnittstelle ist allen für die Kommunikation mit Remote-Geräten verwendbaren Objekten gemeinsam. Daher ist es unabhängig von der Entferntheit einfach, zur Lauf- oder Übersetzungszeit Tests vorzunehmen. Diese Objekte verfügen über andere Semantiken als einfache lokale Objekte - beispielsweise ist ihre Auffassung von Gleichheit andersartig. Gleichheit ist davon abhängig, ob zwei Objekte, die `Remote` implementieren, sich auf das gleiche Remote-Objekt beziehen, welches sich auf einem bestimmten Server befindet, und nicht davon, ob die *Objekte* identisch sind. Wenn Sie mit einer Referenz mit einem Remote-Objekt arbeiten, beziehen Sie sich dabei auf einen lokalen Proxy des Remote Objekts, der als Stub bezeichnet wird. Wenn Sie zwei solche Referenzen vergleichen, ist es uninteressant, ob sich diese auf den gleichen lokalen Stub beziehen - wichtig ist, ob jene Stubs sich auf das gleiche Remote-Objekt auf einem andern Rechner beziehen. Die Auffassung von Gleichheit wurde demnach für die Handhabung von Remote-Kommunikation erweitert.

Besonders wichtig ist, dass jede aufrufbare Remote-Methode eines Remote-Objekts aufgrund der Deklaration die Exception `java.rmi.RemoteExeption` handhaben kann. Gemeinsam mit ihren Subklassen bestimmt diese Exception die Fehlermodi, welche in einer vernetzten Umgebung auftreten können. Durch das explizite Deklarieren dieser Störungen innerhalb der Methodensignatur veranlasst RMI Verwender von Remote-Objekten, den für die Handhabung mit diesen zusätzlichen Fehlern erforderlichen Code zu implementieren. Da Java Exceptions als Teil der Methodensignatur deklariert, muss jeglicher Code, der den Remote-Code aufruft, entweder auf die Fehlerbedingungen, welche die Remote-Exception darstellen, entsprechend reagieren oder an Code weiterleiten, der mit ihnen umgehen kann. Der Compiler erlaubt dem Programmierer nicht, die Bedingung zu vernachlässigen.

Selbstverständlich ist RMI kein Allheilmittel und kann weder alle innerhalb verteilter Systeme auftretender Probleme endgültig lösen noch dafür sorgen, dass das Konstruieren verteilter Systeme genauso einfach ist wie der Aufbau lokaler. Es stellt jedoch einen Schnritt in die richtige Richtung dar. RMI verwendet das Java System für Datentypen, unterscheidet semantisch jedoch eindeutig zwischen lokalen und Remote-Objekten. Es zwingt Entwickler, jene Fehlermodi zu berücksichtigen, die in verteilten Umgebungen auftreten können und veranlasst sie, sich über die Verteilung von vornherein Gedanken zu machen, statt sie als Einzelheit der Implementierung zu verbergen. RMI erhöht die Leistungsfähigkeit der Programmiersprache Java und nutzt die Vorteile strikter Typisierung, indem es diese auf die Netzwerkdomäne ausweitet. RMI beseitigt die Probleme der Portabilität von Daten. Ausserdem ermöglicht RMI durch das Unterstützen des sicheren und zuverlässigen Verschiebens von Code die Nutzung der Vorteile der objektorientierten Programmierung über die Grenzen des Netzwerks hinaus, indem es die Verwendung von flexiblem und erweiterbarem Remote-Code unterstützt. Es bietet Entwicklern mehrere leistungsfähige Werkzeuge, mit denen diese die Probleme, welche beim Erstellen verteilter Anwendungen auftreten, lösen können.

Selbstverständlich ist Java für seinen mobilen Code berühmt, und zwar nicht nur im Zusammenhang mit RMI. In der folgenden Zusammenstellung wird die Verwendung mobilen Codes mit Hilfe von RMI mit dessen sonstiger Verwendung innerhalb von Java verglichen.

Alle Arten der Verwendung mobilen Codes, mobiler Daten und strikter Typenprüfung für die Gewährleistung der Robustheit, ebnen den Weg für den Übergang zu echten dynamisch verteilten Systemen, bei denen das "System" aus diversen Komponenten besteht, die dem Erreichen eines gemeinsamen Zwecks dienen. Dies unterscheidet es von herkömmlichen vernetzten Systemen, bei denen mit "System" die einzelnen Komponenten gemeint sind, die in recht statischer Weise miteinander verbunden sind.

1.4.1.2.1. Die Verwendung mobilen Codes in anderen Zusammenhängen

Hier eine kurze Zusammenstellung möglicher mobiler Code Anwendungen:

- Applets
  dies ist die bekannteste Verwendung von Java. Applets können herunter geladen werden und nach Gebrauch wieder entfernt werden.
- Agenten
  Agenten sind kleine autonome Code-Segmente, die sich im Internet bewegen, um nach gewünschten Daten zu suchen. Ein Reise Agent könnte beispielsweise verwendet werden, um innerhalb des Internets die besten Möglichkeiten für einen Flug nach Las Vegas zu ermitteln. Autonome Agenten haben die Eigenschaft, dass sie nach dem Verlassen des "Geburtsrechners" auch weiterleben können, wenn dieser ausgeschaltet wird.
- RMI
  RMI ist kein eigentliches verteiltes System, stellt aber die Basis für die Entwicklung solcher Systeme dar. RMI kann für das automatische Installieren von Anwendungen oder das Erstellen von Agentensystemen eingesetzt werden.
- Jini
  die Möglichkeiten von Jini haben wir zwar bislang noch nicht ausführlich genug untersucht, doch Jini baut auf die Fähigkeiten von RMI auf und verwendet mobilen Code als Möglichkeit für die Wartung, die Gewährleistung der Erweiterbarkeit und zur Vereinfachung der Verwaltung vernetzter Geräte und Dienste. Da Jini auf RMI aufsetzt und ausserdem alle Möglichkeiten der Programmiersprache Java ausschöpft, sind die Vorteile des Verschiebens von Code auch in Jini Verfügbar.

Java ist die erste verbreitete und kommerziell durchsetzungsfähige System, welches vielfältige Möglichkeiten der Nutzung mobilen Codes bietet.

1.4.1.3. Eigenschaften dynamisch verteilter Systeme

Als Zusammenfassung dieses Teils, hier eine Liste der wesentlichen Eigenschaften dynamisch verteilter Systeme:

- dynamisch verteilte Systeme können aus sehr vielen einzelnen Geräten bestehen
- zur Gewährleistung der Ausbaufähigkeit muss ihr Code auf einer Vielzahl von Rechnern vorhanden sein. Damit der Code für zahlreiche Geräte verfügbar ist, muss er quasi allgegenwärtig sein. Eine derartige ubiquitäre Verfügbarkeit setzt eine einfache Installierbarkeit des Codes voraus.
- nach deren Einrichtung sollen dynamisch verteilte Systeme möglicherweise langfrsitig betrieben werden, unter Umständen sogar monate- oder jahrelang. Daher nüssen sie robust und zuverlässig sein und Fehler "selbstständig" beheben können.
- jedes langfristig betriebene System muss in der Lage sein, an neue Netzwerktopologien angepasst zu werden. Neue Geräte müssen installiert werden können, und gelegentlich sind Aktualisierungen sowohl der Schnittstellen als auch ihrer Implementierungen

erforderlich. Dynamisch verteilte Systeme müssen deshalb anpassungs- und ausbaufähig sein.

All diese Eigenschaften deuten auf eine Art von Software hin, bei der das System aus zahlreichen unabhängigen Einheiten aufgebaut wird, die sich in der Gegenwart anderer Komponenten spezifisch verhalten. Gemeinsam bilden all diese Komponenten ein smbiotisches, selbstheilendes System, welches grösser ist als die Summe seiner Teile.

Literatur Hinweis: http://www.sun.com/research/technical-reports/1994/abstract-29.html
(A Note on Distributed Computing : Samuel C. Kendall, Jim Waldo, Ann Wollrath and Geoff Wyant)

1.5. Das Jini-Modell

Nachdem wir nun quasi als Review nochmals einige Gründe für Java als Basis für Verteilte Systeme wiederholt haben, wollen wir uns auf Jini konzentrieren. Die Vision von Jini ist ganz einfach: Sie können ein beliebiges Jini-fähiges Gerät - ob es sich nun um eine Digitalkamera, einen Drucker, einen PDA oder ein Mobiltelefon handelt - an einem TCP/IP Netzwerk anschliessen und sind daraufhin automatisch in der Lage, zahlreiche andere Jini-fähige Geräte in ihrer Nachbarschaft zu erkennen und zu benutzen. Neu sind Ansätze, auch ohne TCP/IP auszukommen. Dies war bereits in der Spezifikation keine Voraussetzung. Jede innerhalb des Netzwerks verfügbare Resource ist auch von Ihrem Jini-fähigen Gerät aus verfügbar, als sei es unmittelbar daran angeschlossen oder als sei das Gerät speziell für die Verwendung der Resource programmiert. Um die "Netzwerk-Gemeinschaft" um ein neues Gerät zu ergänzen, braucht es lediglich daran angeschlossen zu werden.

1.6. Die Zielsetzungen bei der Entwicklung von Jini

In diesem Abschnitt werden wir uns mit den Zielsetzungen bei der Entwicklung von Jini beschäftigen. Damit sind jeneBereiche gemeint, die Entwickler von Jini als besonders wichtig erachteten.

1.6.1. Einfachheit

Bill Joy, einer der geistigen Väter von Java und Jini, sagte einmal "Grosse erfolgreiche Systeme beginnen als kleine erfolgreiche Systeme". Diese Philosophie trägt erheblich zur Motivation des Jini Projekts bei. Wenn man Java kennt, kennt man im Grunde genommen Jini auch schon. Jini nutzt die grundlegenden Konzepte von Java - insbesondere jene, die im Zusammenhang mit der verteilten Datenverarbeitung stehen, die wir oben besprochen haben. Jini fügt lediglich eine dünne Schicht hinzu, um Geräte und Dienste innerhalb des Netzwerkes die Zusammenarbeit zu erleichtern.

Folgendes ist wichtig: Jini dient im wesentlichen der Verbindung von Diensten und nimmt keinen Einfluss darauf, welche Dienste im einzelnen angeboten werden oder wie diese Dienste gestaltet sind bzw. sich verhalten. Jini-Dienste brauchen nicht in Java geschrieben zu werden. Es ist lediglich erforderlich, dass sich irgendwo innerhalb des Netzwerks etwas Code befindet, der in Java geschrieben wurde und an den Mechanismen teilnehmen kann, die Jini für das Auffinden anderer Jini-Geräte und -Dienste verwendet.

Bislang habe ich die Begriffe "Gerät" und "Dienst" nicht scharf unterschieden. Aus der Sicht von Jini ist tatsächlich alles ein Dienst - sogar auch ein Gerät wie ein Scanner, Drucker oder Telefon. In Form einer objektorientierten Metapher kann man sagen, dass alles in der Welt, sogar Hardware-Geräte, anhand der jeweiligen Schnittstelle zur Umgebung verstanden werden kann. Da diese Schnittstellen den von ihnen angebotenen Diensten entsprechen, verwendet Jini den Begriff "Dienst" explizit für Einheiten innerhalb des Netzwerks, die von andern Jini-Teilnehmern verwendet werden können. Die von diesen Einheiten angebotenen Dienste können durch ein oder mehrere Hardware-Geräte oder durch reine Software-Komponenten (bzw. Kombinationen daraus) implemenriert werden (auch hier ist das Konzept der Objektorientierung erkennbar).

1.6.2. Zuverlässigkeit

Jini bietet den Diensten eine Infrastruktur für das gegenseitige Auffinden und Verwenden innerhalb des Netzwerks. Doch was bedeutet dies eigentlich? Ist Jini einfach ein Name Server wie der Domain Name Service (DNS) im Internet oder das Lightweight Directory Access Protocol (LDAP) innerhalb eines Unternehmens?
Jini weist durchaus Ähnlichkeiten mit einem Name Server auf, es stellt sogar einen Dienst für das Auffinden anderer Dienste innerhalb einer Gemeinschaft zur Verfügung. Obgleich dieser Dienst weitaus leistungsfähiger ist als traditionelle Namensdienste, wie wir sehen werden. Zwischen der Funktion von Jini und der eines einfachen Name Server gibt es jedoch zwei wesentliche Unterschiede.

- Jini unterstützt die wechselseitige Kommunikation zwischen Diensten und den Benutzern jener Dienste. Dienste können aus der Sicht anderer Jini-Teilnehmer auf sehr einfache Weise erscheinen und verschwinden. Interessierte Einheiten können automatisch benachrichtigt werden, sobald sich der Satz der zusammenarbeitenden Dienste ändert. Jini erlaubt bekannten Diensten innerhalb des Satzes der Dienste, sich fliessend zu verändern, ohne dass irgendeine Form statischer Konfiguration oder Administration erforderlich ist. Jini unterstützt auf diese Weise etwas, das als "spontane Vernetzung" bezeichnet werden könnte. Benachbarte Dienste bilden automatisch eine Gemeinschaft, ohne explizite Eingriffe des Benutzers zu erfordern. Sie müssen also keine Konfigurationsdateien bearbeiten, Name Server herunter- und wieder hochfahren, Gateways konfigurieren oder etwas Ähnliches tun, um einen Jini-Dienst zu verwenden. Sie brauchen ihn buchstäblich nur anzuschliessen, und den Rest übernimmt Jini für Sie. Darüber hinaus verfügt jeder mit einer Jini-Gemeinschaft verbundene Dienste (bzw. jedes entsprechende Gerät) über sämtlichen Code, den andere Teilnehmer der Gemeinschaft für seine Verwendung benötigen.

- Gemeinschaften von Jini-Diensten sind weitgehend selbstheilend. Dies ist eine der von Anfang an in Jini eingeplante Basiseigenschaften: Jini geht nicht davon aus, dass Netzwerke perfekt sind oder Software stets fehlerfrei arbeitet. Nötigenfalls kann das System Schäden an sich selbst reparieren. Ausserdem unterstützt Jini auf sehr natürliche Weise redundante Infrastrukturen, um die Wahrscheinlichkeit zu verringern, dass Dienste nach Ausfällen wichtiger Rechner nicht mehr verfügbar sind.

Insgesamt lassen diese Eigenschaften Jini aus der Menge der kommerziellen Infrastrukturen für verteilte Syseme herausragen. Diese Eigenschaften sorgen dafür, dass eine Jini-Gemeinschaft praktisch keiner Verwaltung bedarf. Die von Jini gebotene Möglichkeiten spontaner Vernetzung erlauben das Ändern der Konfiguration eines Netzwerks ohne Zuhilfenahme von Netzwerkverwaltern. Die Möglichkeit des Verwendens innerhalb von Jini-Gemeinschaften zuvor unbekannter Geräte bedeutet, dass keine Treiber oder sonstige Software mehr installiert werden muss, um einen bestimmten Dienst verwenden zu können. Die grundlegende Jini-Software muss selbstverständlich zuvor installiert worden sein. Die Fähigkeit des Systems zur Selbstheilung verringert ausserdem den verwaltungsaufwand und erspart Benutzern manchen Ärger. Eine zusammenarbeitende Gruppe von Jini-Diensten kann sich an Änderungen der Netzwerktopologie, Ausfällen von Diensten und Teilungen des Netzwerks auf adäquate Weise anpassen. Jini-Dienste können mit Störungen innerhalb des Netzwerks umgehen. Es könnte sein, dass sie ihre Aufgabe nicht mehr vollwertig erfüllen, sogar das Telefonnetz teilt dem Benutzer manchmal Störungen mit, doch sie werden sich bei

Störungen zumindest auf vorhersehbare Weise verhalten und sich nach einiger Zeit wiederherstellen.

1.6.3. Skalierbarkeit

Gruppen von Jini-Diensten können sich zu zusammenarbeitenden Sätzen zusammenschliessen. Bei Jini werden solche Gruppen von Diensten *Gemeinschaften* genannt. Alle Dienste innerhalb einer Gemeinschaft kennen einander und sind in der Lage, sich gegenseitig zu verwenden.

Jini-Dienste schliessen sich also zu Gemeinschaften zusammen. Doch wie gross sind solche Gemeinschaften? Welche "Zielgrösse" wurde von den Jini-Designern für eine einzelne Gemeinschaft angestrebt? Dies ist eine wichtige Frage. Falls das Jini-Design für grosse Gruppen aussgelegt ist - beispielsweise für eine Gruppe aus allen Jini-fähigen Geräten innerhalb Europas - weist es sicherlich in bezug auf die Leistung und die Komunikation erheblich andere Charaktersitiken auf, als wenn es für kleine Gruppen gedacht ist. Der Hauptaspekt ist in diesem Zusammenhang die *Skalierbarkeit* - wie Jini für unterschiedlich viele Dienste ausgelegt ist, von sehr grossen bs hin zu sehr kleinen Gemeinschaften.

Skalierbarkeit wird von Jini durch *Föderieren* gewährleistet. Föderieren ist die Fähigkeit von Jini-Gemeinschaften miteinander verknüpft bzw. zu grösseren Gruppen zusammengeschlossen zu werden. Im Einzelfall stimmt die Grösse einer einzelnen Jini-gemeinschaft mit jener einer Arbeitsgruppe überein. Dies sind die von einer Gruppe mit 10 bis 100 Personen benötigten Drucker, PDAs, Mobiltelefone, Scanner, Netzwerkdienste und sonstige Geräte. Diese Konzentration auf Arbeitsgruppen ist darauf zurückzuführen, dass die meisten Leute am ehesten mit Personen zusammenarbeiten, die sich in ihrer unmittelbaren Umgebung befinden. Mit Jini ist es ganz einfach, solche Gruppen von Personen innerhalb einer Gemeinschaft zusammenzubringen und ihre Resourcen der gemeinsamen Verwendung zugänglich zu machen.

Auch wenn Sie an einer kleinen von ihrer Arbeitsgruppe verwendeten Gemeinschaft teilnehmen, benötigen Sie möglicherweise gelegentlich Zugriff auf, aus der Sicht des Netzwerks, entfernten Resourcen - beispielsweise diesen schnellen neuen Farblaserdrucker in der Marketingabteilung. Jini unterstützt den Zugriff auf Dienste anderer Gemeinschaften, indem diese zu grösseren Einheiten föderiert werden können. Sogar der Lookup-Dienst - die für das Erfassen aller Dienste innerhalb einer Gemeinschaft zuständigen Einheit - ist *selbst* ein Jini-Dienst. Daher kann sich der Lookup-Dienst einer bestimmten Gemeinschaft in anderen Gemeinschaften registrieren und sich somit den dortigen Benutzern als verwendbare Resource anbieten. Wie wir später noch sehen werden, enthält Jini sich tatsächlich selbst: viele Funktionen von Jini sind selbst Jini-Dienste, die mit Hilfe der normalen Jini-Mechanismen freigegeben und von anderen Diensten verwendet werden können.

Die Topologie solcher Gemeinschaften ist äusserst flexibel. Sobald Sie die Jini-Software installieren, wird das System selbständig Gemeinschaften erstellen, die sich an den Grenzen des Netzwerkes orientieren. Wenn beispielsweise die Entwicklungs- und Marketingabteilung mit unterschiedlichen Netzwerken arbeiten, werden diese jeweils eine eigene Jini-Gemeinschaft bilden. Wenn Sie diese Gemeinschaften miteinander föderieren wollen, brauchen Sie lediglich dafür zu sorgen, dass die Lookup-Dienste der Gemeinschaften einander bekannt sind.

1.6.3.1. Jini und Verwaltung

Jini ist für die verwaltungsfreie Verwendung ausgelegt. Trotzdem können Sie die Struktur ihres Systems selbstverständlich ihren speziellen Anforderungen gemäss gestalten.

Das Föderieren von Gemeinschaften innerhalb eines Unternehmens ist ein Beispiel für diese Art des Einwirkens: Jini ist in der Lage, Gemeinschaften selbstständig der Struktur eines Netzwerks gemäss zu gestalten, kennt den organisatorischen Aufbau des Unternehmens jedoch nicht. Diese Informationen müssen Sie Jini deshalb selbst mitteilen.

1.7. Agnostizismus in bezug auf Geräte

(zum Begriff Agnostizismus : http://home.t-online.de/home/humanist.aktion/orientie.htm)
Es wurde zwar im vorigen Abschnitt erwähnt, verdient jedoch eine ausführliche Erläuterung: in bezug auf Geräte ist Jini agnostisch. Was heisst das? Im Grund bedeutet dies, dass Jini dafür ausgelegt ist, in seinen Gemeinschaften eine breite Palette unterschiedlicher Einheiten zu unterstützen. Bei diesen "Einheiten" kann es sich um Geräte, Software oder um eine Kombination aus beidem handeln. Der Benutzer kann tatsächlich im allgemeinen nicht feststellen, worum es sich eigentlich handelt. Dies ist eines der Grundprinzipien von Jini. Um etwas zu verwenden, brauchen Sie nicht zu wissen - und Sie brauchen sich nicht einmal darum zu kümmern - ,ob es sich dabei eigentlich um Software oder Hardware handelt. Lediglich die Verwendung seiner Schnittstelle ist für Sie von Bedeutung.

Wenn ein neues Hardware- Gerät am Netzwerk angeschlossen wird, kann sich Jini an die jeweils erforderliche Rechenleistung anpassen. Jini kann mit vollwertigen PCs oder Servern umgehen, die in der Lage sind, mit mehreren JVMs gleichzeitig zu arbeiten und untereinander mit Gigabit Geschwindigkeit verbunden sind. Ebenso kann Jini mit Geräten wie PDAs und Mobiltelefonen zusammenarbeiten, die möglicherweise nur beschränkt Java-fähig sind - beispielsweise könnten sie über eine Embedded oder Personal Java Implementierung mit einem begrenzten Satz von Klassenbibliotheken verfügen.

Jini kann sogar mit Geräten umgehen, die überhaupt keine Berechnungen durchführen können.Wie wir weiter unten sehen werden, ist Jini in der Lage, Geräte mit der geringen Komplexität eines Lichtschalters einzugliedern. Dies erfordert lediglich eine andere, möglicherweise gemeinsam verwendete Resource für Berechnungen, die im Namen jenes Geräts die Protokolle der Jini-Gemeinschaft verarbeiten kann.

Ausserdem, und dies könnte etwas überraschend sein, erfordert Jini nicht einmal, dass die Geräte oder Dienste Java verstehen bzw. darin geschrieben sind! Auch in diesem Fall ist lediglich ein Java-fähiges Gerät erforderlich, welches stellvertretend als Proxy fungiert.

1.8. Was Jini nicht *ist*

Nachdem wir nun eine grobe Vorstellung haben, was Jini ist, sollten Sie noch wissen, was Jini *nicht* ist.

1.8.1. Jini ist kein Name Server

Wie gesagt ist Jini nicht nur ein Name Server. Ein Teil der Arbeitsweise von Jini - beispielsweise das Erfassen der innerhalb einer Gemeinschaft bekannten Dienste - ähnelt der eines Name Servers. Besonders die Lookup Funktionen von Jini stimmen in ihrer Funktion weitestgehend mit der entsprechenden Funktionalität eines Name Servers überein, doch Jini kann weitaus mehr. Es ist ein Modell für das Einrichten verteilter Systeme, die spontanes

Erscheinen und Verschwinden innerhalb einer Gemeinschaft unterstützen und sich nötigenfalls selbst heilen können.

1.8.2. Jini ist nicht JavaBeans

JavaBeans bietet als Beans bezeichnete Software-Komponenten eine Möglichkeit, einander zu finden, die von andern Beans angebotenen Dienste zu nutzen, sich gegenseitig zu überprüfen usw. Doch JavaBeans dient erheblich andern Zielsetzungen als Jini. Beans sind hauptsächlich für die Verwendung innerhalb eines einzelnen Adressraums gedacht: die für die Kommunikation zwischen den Beans verwendeten Mechanismen basieren auf dem direkten Aufruf von Methoden und nicht auf Remote-Protokollen. Obgleich das Beans-Modell sehr flexibel ist, ist es weitaus weniger dynamisch als Jini. Wenn ein neues Bean auf Ihrem System erscheint, sind die innerhalb Ihrer Anwendung bereits vorhandenen Beans nicht sofort darüber informiert und beginnen nicht unmittelbar mit seiner Verwendung. Sie - der Entwickler das Systems - müssen das Bean explizit mit Ihren Anwendungen verknüpfen und Verbindungen mit anderen Beans herstellen. JavaBeans wurden hauptsächlich zu Entwicklungszwecken und ausserdem für die Verwendung zur Laufzeit in einzelnen Adressräumen konzipiert. Jini ist gänzlich auf die Verwendung zur Laufzeit in mehreren Adressräumen ausgelegt. Dies soll jedoch nicht heissen, dass es sich bei Jini und JavaBeans um inkompatible Systeme handelt. Jini kann JavaBeans in mehrfacher Hinsicht förderlich sein. Mit der Verwendung des JavaBeans-Ereignismodells durch Jini und mit der Nutzung von Beans für Jini-Dienste werden wir später beschäftigen.

1.8.3. Jini ist nicht Enterprise JavaBeans (EJB)

Auch mit Enterprise JavaBeans (EJB) ist Jini nicht zu verwechseln. EJB weist oberflächlich einige Ähnlichkeiten mit Jini auf. Es dient der Eingliederung von Diensten in das Netzwerk. Enterprise Beans kann in unterschiedlichen Adressräumen verarbeitet werden und normalerweise ist dies auch der Fall. Doch auch die Zielsetzungen von EJB weichen erheblich von jenen der Jini Technologie ab. EJB dient vornehmlich dem Verbinden bestehender Unternehmensnetzwerke mit Hilfe von Java, um eine Grundlage für Unternehmensanwendungen zu schaffen. Es ist darauf ausgerichtet, das einfache Einrichten solcher Systeme zu ermöglichen, und ergänzt die innerhalb des Unternehmensnetzwerks bereits vorhandenen Dienste für Transaktionen, Benachrichtigungen und Datenbanken. EJB wird daher zumeist verwendet, um relativ statische Verbindungen zwischen den Software-Komponenten eines Unternehmens einzurichten. Solange die Struktur des Systems unverändert bleibt, ist das Umorganisieren der Verbindungen weitgehend während der Entwicklung festgelegt. Jini hingegen ist weitaus dynamischer und unterstützt sowohl das Erkennen von Diensten als auch das Aufbauen von Verbindungen zwischen ihnen zur Laufzeit.

1.8.4. Jini ist nicht RMI

Jini und RMI sind nicht das gleiche. Jini *nutzt* RMI zwar ausgiebig, besonders für die Verarbeitung mobilen Codes, stellt jedoch eine Gruppe von Diensten und Konventionen dar, die auf RMI aufsetzen. Jini-Dienste profitieren von allen Vorteilen der Fähigkeiten von Jini hinsichtlich spontaner Vernetzung und Selbstheilung. Diese Möglichkeiten könnte zwar auch eine andere RMI Anwendung bieten, obgleich dies mit einem hohen Entwicklungsaufwand verbunden wäre, sie sind jedoch kein integraler Bestandteil von RMI, und herkömmliche RMI Anwendungen weisen solche Fähigkeiten in der Regel nicht auf.

1.8.5. Jini ist kein verteiltes Betriebssystem

Ausserdem ist Jini kein verteiltes Betriebssystem. In mancher Hinsicht gehen seine Fähigkeiten weit über ein verteiltes Betriebsystem hinaus, da Teile von Jini auf

unterschiedlichen Plattformen verarbeitet werden, die zumindest über eine JVM verfügen. Andererseits ist Jini erheblich kompakter - die Fähigkeiten und Konzepte von Jini sind auf die wesentlichen Zielsetzungen dieser Technologie beschränkt. Jini soll lediglich das Erfassen und Auffinden von Diensten ermöglichen. Echte verteilte Betriebsysteme stellen zusätzliche Dienste traditioneller Betriebsysteme zur Verfügung, wie Dateizugriffe, Zeitplanung für die CPU, Anmeldung von Benutzern und vieles mehr. Dies bezieht sich jedoch auf eine beschränkte Gruppe vernetzter Rechner. Jini ermöglicht auch weitaus einfachere Geräten das Teilnehmen an Jini-Gemeinschaften, die nicht jeweils über ein vollwertiges betriebsystem verfügen müssen.

1.9. Die fünf Grundprinzipien von Jini

Die konzeptionelle Einfachheit war eines der ausdrücklichen Ziele von Jini. Als nächstes werden wir uns mit der praktischen Bedeutung dieses Aspekts beschäftigen. Sämtliche Fähigkeiten von Jini für das Unter
stützen spontan konstruierbarer, selbstheilender Gemeinschaften von Diensten basieren auf fünf Grundprinzipien. Um Jini zu verstehen, brauchen Sie lediglich diese Konzepte zu kennen.

Es handelt sich um die folgenden:

- **Discovery**
- **Lookup**
- **Leasing**
- **Remote-Ereignisse**
- **Transaktionen**

Diese Konzepte wurden in Form einer Gruppe von Software-Bibliotheken und Konventionen implementiert, die von dem für das Aufbauen und Betreiben von Jini-Gemeinschaften zuständigen Code verwendet werden.

Als *Discovery* wird der für das Auffinden und Anbinden von Gemeinschaften innerhalb des Netzwerks verwendete Prozess bezeichnet. Discovery ist jener Teil von Jini, der die Fähigkeit des Systems zum spontanen Aufbau von Gemeinsamkeiten bedingt.

Lookup bestimmt, auf welche Weise der zur Verwendung eines bestimmten Dienstes benötigte Code an Teilnehmer übertragen wird, die den Dienst nutzen wollen. Jede Jini-Gemeinschaft verfügt über einen Lookup-Dienst, der als ihr Verzeichnisdienst fungiert und das Suchen und Auffinden von Diensten ermöglicht. Die Funktionsweise von Lookup ist jedoch komplizierter als die eines einfachen Name Server. Ein Name Server stellt lediglich Verknüpfungen zwischen Objekten und Namen her, die Lookup-Funktionen von Jini sind jedoch für die Hierarchie der Datentypen von Java ausgelegt. Daher kann eine Lookup-Suche auf dem Typ eines Objekts basieren und sogar die durch Vererbung entstandenen Verhältnisse zwischen Objekten berücksichtigen. Eine solche Funktionalität ist um einiges komplexer und im Zusammenhang mit Jini nützlicher als ein einfacher Zeichenketten-basierter Namensdienst.

Leasing ist eines der wichtigsten Konzepte von Jini, da es so ausgiebig genutzt wird. Die Leasing-Technik ermöglicht Jinis Fähigkeiten zur Selbstheilung. Sie sorgt dafür, dass sich eine Gemeinschaft nach Ausfällen an der ihr beteiligten Hauptdienste wieder regenerieren kann. Ausserdem sorgt Leasing dafür, dass sich langlebige Dienste wie beispielsweise

Lookup nicht Informationen über ihre Gemeinschaften immer wieder "aufblähen". Ohne Leasing könnte das Wachstum solcher langlebigen Dienste ausufern.

Remote-Ereignisse werden von Jini verwendet, um Diensten das gegenseitige Benachrichtigen über ihren Zustand zu ermöglichen. Da Lookup selbst ein Dienst ist, kann es Remote-Ereignisse zur Benachrichtigung interessierter Beteiligter über Änderungen der innerhalb einer Gemeinschaft verfügbaren Dienste zu verwenden. Jinis Modell für Remote-Ereignisse ähnelt dem Ereignismodell von JavaBeans, stimmt jedoch nicht mit ihm überein. Mit diesem Thema werden wir uns später noch befassen. Im Grunde müssen Remote-Ereignisse alle im vorherigen Kapitel erörteten Semantiken der "Entferntheit" aufweisen und ausserdem eine Hauptforderung von Jini erfüllen, welche darin besteht, dass jede Einheit jeden beliebigen Ereignistyp empfangen kann.

Mit Hilfe von *Transaktionen* kann Jini erreichen, dass Berechnungen unter Beteiligung mehrerer Dienste sicher ausgeführt werden können. Damit meine ich, dass der Aufrufer darüber informiert wird, ob die Berechnungen entweder vollständig oder gar nicht verarbeitet wurden. In beiden Fällen befindet sich das System in einem bekannten Zustand. Jinis Transaktionsmodell schützt vor den Tücken teilweiser Ausfälle innerhalb verteilter Systeme, hilft bei der Vermeidung und Handhabung von Problemen mit der Gleichzeitigkeit, welche in verteilten Systemen auftreten können und die wir bereits weiter vorne kennen gelernt haben, und ermöglicht Dienste mit grosser Robustheit und Widerstandsfähigkeit in bezug auf Netzwerkfehler. Das Jini-Modell für Transaktionen ähnelt zwar dem in der Datenbankprogrammierung häufig verwendeten "klassischen" Transaktionsmodell, weist jedoch intern erhebliche Unterschiede auf.

Der Rest dieses Kapitels umfasst einen Überblick über diese Grundprinzipien und zeigt auf, wie diese Konzepte in der Praxis ineinandergreifen. Am Ende dieses Kapitels werden Sie mit den wesentlichen Prinzipien vertraut sein, auf denen Jini aufgebaut ist. In den folgenden Kapiteln / Abschnitten werden wir das konzeptionelle Wissen vertiefen, und Sie haben die Möglichkeit, die für das Erstellen funktionierender Jini-Dienste und -Anwendungen notwendigen Programmierkenntnisse zu erwerben.

1.9.1. Discovery

Bevor eine Jini-fähige Einheit - entweder ein Dienst oder eine Anwendung - andere Jini-Dienste nutzen kann, müssen ihr eine oder mehrere Jini-Gemeinschaften bekannt sein. Dies kann die Einheit durch das Auffinden eines Lookup-Dienstes bewerkstelligen, der die gemeinsam verwendetetn Resourcen innerhalb einer Gemeinschaft verwaltet - dieser Vorgang des Auffindens des verfügbaren Lookup-Dienstes wird als *Discovery* bezeichnet. Das Discovery-Protokoll von Jini ermöglicht Jini-fähigem Code das Auffinden einer Jini-Gemeinschaft. Sobald eine Jini-Gemeinschaft ermittelt wurde, kann die Jini-fähige Einheit eine Folge von Konventionen durchlaufen, welche als *Teilnahmeprotokoll* bezeichnet wird, um ein Teil dieser Gemeinschaft zu werden und der Gemeinschaft Dienste anzubieten.

Zwischen Gemeinschaften und Lookup-Diensten besteht nicht grundsätzlich ein Eins-zu-Eins Verhältnis. Innerhalb eines Netzwerks kann ein Lookup-Dienst ohne weiteres mehreren Gemeinschaften dienen, und einzelne Gemeinschaften können durchaus mehrere Lookup-Server umfassen. Lookup-Dienste werden explizit von Administratoren gestartet. Dies ist in der Tat der *einzige* Jini-Dienst, der für das Aufbauen von Jini-Gemeinschaften erforderlich ist.

1.9.1.1. Das Discovery-Protokoll

Es gibt nur ein einziges Discovery-Protokoll - Jini unterstützt mehrere für unterschiedliche Situationen:

- das Multicast Request Protocol wird bei der Aktivierung einer Anwendung bzw. eines Dienstes verwendet, um etwaige benachbarte aktive Lookup-Dienste zu ermitteln.

- das Multicast Announcement Protocol wird von Lookup-Diensten zur Ankündigung des eigenen Vorhandenseins benutzt. Sobald ein neuer Lookup-Dienst innerhalb einer bestehenden Gemeinschaft gestartet wird, wird jeder interessierte Beteiligte darüber via Multicast Announcement Protocol darüber informiert.

- das Unicast Discovery Protocol wird verwendet, wenn einer Anwendung bzw. einem Dienst der zu benutzende Lookup-Dienst bereits bekannt ist. Mit Hilfe des Unicast Discovery Protocol kann direkt mit einem Lookup-Dienst kommuniziert werden, der nicht unbedingt zum lokalen Netzwerk gehören muss, sofern der Name des Lookup-Dienstes bekannt ist. Die Benennung von Lookup-Diensten erfolgt in URL-Syntax, wobei das Protokoll als `jini` spezifiziert wird (`jini://minijini.hta.fhz.ch` spezifiziert beispielsweise den auf dem Host `minijini.hta.fhz.ch` auf dem Standardport ausgeführten Lookup-Server). Unicast-Lookup wird zum Einrichten statischer Verbindungen zwischen Diensten und Lookup-Diensten verwendet. Auf diese Weise können beispielsweise Lookup-Dienste explizit föderiert werden.

Als Ergebnis des Discovery-Prozesses erhält die jeweilige Einheit- das Objekt, welches den Prozess initiiert hat - eine oder mehrere Referenzen auf die Lookup-Dienste der angegebenen Gemeinschaft. Mit Hilfe dieser Referenzen können Dienste die von ihnen angebotenen Möglichkeiten bekanntmachen, damit sowohl Clients als auch Dienste die innerhalb einer Gemeinschaft verfügbaren Dienste ermitteln können. Mit Hilfe von Lookup-Diensten können noch einige andere Operationen durchegführt werden. Beispielsweise können Ereignisse gezielt angefordert werden. Weitere Einzelheiten über die Möglichkeiten der Verwendung von Lookup-Diensten werden imAbschnitt zu Lookup später erörtert.

Einige Anwendungen wie beispielsweise Browser verwenden Referenzen zu Lookup-Servern zwar für das Anzeigen einer Liste der verfügbaren Dienste, doch häufig handelt es sich bei der den Discovery-Prozess auslösenden Einheit um einen Jini-Dienst, der sich selbst den anderen Mitgliedern der Gemeinschaft zur Verfügung stellen will. Der Vorgang der Bereitstellung eines Dienstes für die Verwendung durch andere Mitglieder einer Gemeinschaft wird als *Teilnahme* an einer Gemeinschaft bezeichnet. Ein Dienst, der an einer Gemeinschaft teilnehmen will, führt ein Discovery durch, um die Lookup-Dienste der Gemeinschaft zu ermitteln und nimmt an ihr anschliessend mit Hilfe der entsprechenden Operationen auf die vom Discovery-Vorgang zurückgegebenen Referenzen auf die Lookup-Dienste jener Gemeinschaft teil. Auch zu diesem Aspekt finden Sie weiter hinten weitere Informationen. Um zu gewährleisten, dass sich Dienste einwandfrei verhalten, erfordert Jini für die Teilnahme eines Dienstes an einer Gemeinschaft die Verwendung des Teilnahmeprotokolls. Im Sinne des Netzwerks handelt es sich dabei eigentlich nicht um ein Protokoll, sondern eher um eine Gruppe von Konventionen, die Dienste einhalten sollten.

1.9.1.2. Unterstützung mehrerer Gemeinschaften

Jini-Gemeinschaften können mit Namen versehen werden, die in den Jini-APIs als "Gruppen" bezeichnet werden. Für den Discovery-Vorgang kann ein Dienst oder eine Anwendung die aufzufindende Gruppe angeben. Die Discovery-Protokolle antworten darauf mit allen

Mitgliedern jener Gruppen. Lookup- und auch andere Dienste können Mitglieder mehrerer Gruppen sein.

Im allgemeinen brauchen Sie zwischen "Gruppen" und "Gemeinschaften" nicht zu unterscheiden - Gruppen sind lediglich die Bezeichnungen, welche Jini innerhalb seiner APIs verwendet, um Gemeinschaften anzugeben und zu repräsentieren. Der wichtigste Unterschied zwischen Gemeinschaften und Gruppen besteht darin, dass die Gruppennamen unterschieldlicher Gemeinschaften aufgrund der Trennung von Netzwerken übereinstimmen können - die Gruppe "public" bei hta.fhz.ch hat beispielsweise nichts mit der gleichnamigen Gruppe bei fhbb.ch zu tun. Trotz identischer Bezeichnungen können sich diese Namen je nach Kontext auf unterschiedliche Gemeinschaften beziehen. Gruppennamen sind also weder global eindeutig noch grundsätzlich global verfügbar. Trotzdem brauchen Sie zwischen Gruppen und Gemeinschaften in der Regel nicht zu unterscheiden.

Wie kann ein Dienste feststellen, an welcher Gemeinschaft er teilnehmen sollte? Zumeist halten Dienste einfach nach der Standardgruppe Ausschau, welche mit einer leeren Zeichenkette benannt und - per Konvention - als "öffentliche" Gemeinschaft behandelt wird. Anschliessend verwenden sie die Multicast-Protokolle, um Verbindungen mit allen gefundenen Lookuo-Diensten aufzubauen. Die Multicast-Protokolle wurden so konzipiert, dass der Discovery-Prozesslediglich die innerhalb des lokalen Netzwerks vorhandenen Lookup-Dienste berücksichtigt, damit nicht das gesamte Internet mit Paketen des Discovery-Protokolls überflutet wird.

Gelegentlich ist es erforderlich, dass ein Dienst an einer nichtstandardmässigen Gemeinschaft teilnimmt. Ein Entwicklungslabor könnte beispielsweise neue Dienste innerhalb einer experimentellen Gemeinschaft testen, die sich innerhalb des gleichen Teilnetzes befindet, in dem sich auch die Gemeinschaft "Produktion" befindet, welche als Standard deklariert sein könnte. Diese beiden Gemeinschaften können Resourcen gemeinsam nutzen, wenn Dienste an beiden Gemeinschaften teilnehmen, und sogar Lookup-Dienste gemeinsam verwenden, falls die entsprechenden Lookup-Dienste Mitglieder beider Gemeinschaften sind. Um der nichtstandardmässigen experimentellen Gemeinschaft beizutreten, würde Dienste für das Auffinden der Lookup-Dienste, die zu dieser Gemeinschaft gehören, den Gruppennamen "experimentell" benutzen.

In andern Fällen könnte es erforderlich sein, dass ein Dienst lediglich explizit mit einigen bestimmten Lookup-Diensten Verbindungen aufbaut - dabei könnte es sich beispielsweise um einen speziellen Lookup-Dienst handeln, der Systemverwaltern vorgehalten ist und besonderen Berechtigungen und Beschränkungen unterliegt. In diesem Fall könnte ein privilegierter Sicherheitsdienst Unicast-Discovery verwenden, um ausschliesslich die Verbindung mit jenem speziellen Lookup-Dienst herzustellen. Der Lookup-Dienst und der Sicherungsdienst können sich dabei in unterschiedlichen Netzwerken befinden. Die Möglichkeit des Verwendens des Unicast Discovery Protocol gibt Jini seine Flexibilität beim Strukturieren von Gemeinschaften.

Später werden wir diverse Einzelheiten der unterschiedlichen Discovery-Protokolle erörtern; doch deren praktische verwendung ist denkbar einfach. Die meisten Anwendungen und Dienste brauchen lediglich die Schnittstelle `DiscoveryListener` zu implementieren und einige einfache Aspekte für die Verwaltung zu berücksichtigen, um Discovery verwenden und benachbarte Jini-Gemeinschaften erkennen zu können.

1.9.2. Lookup

Während der Discovery Prozess für das Auffinden von Lookup-Diensten zuständig ist, bezieht sich *Lookup* auf die eigentliche Verwendung der Lookup-Dienste. Stellen Sie sich den Lookup-Dienst wie einen Name-Server vor - im Grunde ist er ein (zumeist langfristig betriebener) Vorgang zur Erfassung aller einer Jini-Gemeinschaft beigetretener Dienste. Im Gegensatz zu traditionellen Name-Servern, die gespeicherten Objekten Zeichenketten als Bezeichnungen zuordnen, unterstützt der Jini-Lookup Dienst weitreichendere Möglichkeiten. Sie können den Jini-Lookup-Dienst nach bestimmten Typen von Objekten durchsuchen und, falls der Lookup-Dienst die Typsemantik von Java versteht, sogar gezielt nach Superklassen oder übergeordneten Schnittstellen gespeicherter Objekte suchen. Nachdem der Discovery-Prozess erfolgreich einen Lookup-Dienst ermittelt hat, liefert er Ihnen eine Referenz auf ein Objekt, das die Lookup-Schnittstellen implementiert. Diese Schnittstelle heisst `ServiceRegistrar`, obgleich die Bezeichnung `Lookup` wohl erheblich plausibler gewesen wäre.

Wie bei LDAP bzw. NIS (Network Information Service) bleiben Ihnen die Interna der Implementation des Lookup-Dienstes verborgen. Dabei könnte es sich um eine einfache Zuordnungstabelle handeln, die regelmässig auf der Festplatte gespeichert wird, oder auch um einen äusserst leistungsfähigen Verzeichnisdienst mit blitzschnellen Lookups und protokollierter, persistenter Speicherung. Als Benutzer eines Lookup-Dienstes brauchen Sie lediglich zu wissen, dass das von Discovery zurückgegebene Objekte `ServiceRegistrar` implementiert und sich daher mit einem auf einem der Rechner innerhalb des Netzwerks ausgeführten Lookup-Dienst verständigen kann. Die Einzelheiten dieser Kommunikation sind Ihnen nicht bekannt - es könnten Java RMI, herkömmliche Sockets oder auch Rauchsignale verwendet werden. Es sind also zahlreiche, sehr unterschiedliche Implementierungen des Lookup-Dienstes realisierbar, deren Einzelheiten sich stets hinter dem von Ihnen verwendeten Objekt verbergen, welches die Lookup-Schnittstelle implementiert.

1.9.2.1. Einen Dienst bekanntgeben

Nachdem Sie nun den Discovery-Prozess abgeschlossen und eines oder mehrere solcher Lookup-Objekte erhalten, was können Sie nun mit ihnen anfangen? Im Prinzip können Sie sich den Lookup-Dienst als Einrichtung vorstellen, die für das Verwalten einer Liste von Dienstelementen zuständig ist. Jedes Dienstelement umfasst ein Objekt, welches andere Teilnehmer der Gemeinschaft herunterladen können, um den Dienst zu verwenden. Ausserdem gehört zu jedem Dienstelement eine Liste mit Attributen, die zur Beschreibung des Dienstes benutzt werden. Beispielsweise könnte ein Lookup-Dienst drei Dienstelemente besitzen, jedes mit einem Proxy und mehreren Attributen, die den Dienst beschreiben. Das Proxy-Objekt kann von Clients heruntergeladen werden, die den Dienst verwenden möchten. Obgleich ein bestimmter Client möglicherweise nichts über die Implementierung eines bestimmten Dienstes weiss, kann er den Proxy dieses Dienstes benutzen, um mit ihm umzugehen.

Um sich innerhalb einer Gemeinschaft zur Verfügung zu stellen, nimmt ein Dienst z.B. ein Drucker, an allen vom Discovery-Prozess angegebenen Lookup-Diensten teil. Die Schnittstelle `ServiceRegistrar` umfasst eine Methode namens `register`(), welche ihm die Teilnahme ermöglicht.

1.9.2.1.1. Tip : treten Sie allen Lookup-Diensten Ihrer Gemeinschaft bei

Beachten Sie, dass eine bestimmte Gemeinschaft, die einem eindeutigen Gruppennamen innerhalb eines Netzwerks entspricht, mehrere Lookup-Dienste enthalten kann. Aus Gründen der Fehlertoleranz könnten diverse redundante Lookup-Dienste aktiv sein.

Ein Dienst, der an einer Gemeinschaft teilnehmen soll, nimmt normalerweise an allen Lookup-Diensten teil, die jene Gemeinschaft unterstützen, damit sich die einzelnen Lookup-Dienste bei Ausfällen gegenseitig ersetzen können. Falls keine besonderen Anforderungen gegeben sind oder der Gültigkeitsbereich Ihres Dienstes aus sonstigen Gründen explizit beschränkt werden muss, sollten Sie im allgemeinen an allen vom Discovery-Prozess aufgefundenen Diensten teilnehmen.

Beim Ausrufen der Methode register() übergeben Sie ein Dienstelementobjekt als Parameter. Als Attribute verwenden Sie Objekte, welche den Dienst beschreiben sollen. Zu diesem Zweck werden von Jini einige standardisierte Attribute angeboten - Dienstname, Position des Dienstes, Kommentare usw. Falls Sie Ihr Programm mit Jini-Dienste zumVerkauf anbieten, verfügen Sie möglicherweise sogar über eine eigene Gruppe standardisierter Attribute für zusätzliche Informationen über die Dienste.

1.9.2.2. Herunterladbare Proxies

Das wichtigste am Dienstelement ist dessen Eindeutigkeit in bezug auf Ihren Dienst. Im Dienstelement bieten Sie ein frei serialisierbares Java-Objekt an, das als Dienst-Proxy bezeichnet wird. Sobald eine andere Einheit - ob dies nun ein Jini-Dienst oder einfach eine Anwendung ist, die Ihren Dienst nutzen möchte - Ihr Dienstelement auffindet, wird dieses Proxy-Objekt in dessen JVM kopiert, damit zur Nutzung Ihres Dienstes Aufrufe darauf ausgeführt werden können.

Dieses Konzept herunterladbarer Dienst-Proxies ermöglicht Jini seine Fähigkeit, Dienste und Geräte ohne explizite Installation von Treibern oder sonstiger Software zu verwenden. Die Dienste stellen den Code, über den der Zugriff aus sie erfolgt, selbst zur Verfügung. Ein Drucker bietet beispielsweise einen Proxy an, der diesen speziellen Drucker steuern kann und ein Scanner stellt einen Proxy zur Verfügung, der zur Steuerung dieses Scanners in der Lage ist. Um einen Dienst zu verwenden, lädt eine Anwendung den entsprechenden Proxy herunter und benutzt ihn, ohne eine Implementierung oder die Art seiner verständigung mit dem eigentlichen Gerät bzw. Prozess kennen zu müssen.

In mancher Hinsicht ähneln Jini-Proxies Java-Applets: Applets ermöglichen das verwaltungsfreie Erlangen und Verwenden einer Anwendung, und Jini-Proxies erlauben das verwaltungsfreie Erlangen und Nutzen benötigter Dienste oder Geräte mit Hilfe der Kommunikation über eine spezielle "Bedienungslogik". Doch Applets dienen zumeist unmittelbar dem Benutzer - sie erscheinen normalerweise auf Anfrage des Benutzers mit entsprechender Benutzeroberfläche auf Webseiten. Jini-Proxies wurden hingegen dafür konzipiert, aufgefunden, heruntergeladen und von Programmen verwendet zu werden. Ini-Proxies können als sichere Netzwerkgerätetreiber aufgefasst werden, die bei Bedarf auf Anfrage von Clients heruntergeladen werden können.

Die Einzelheiten der Verständigung des Proxy-Objekts mit Ihrem Dienst obliegt vollends Ihre Verantwortung. Dabei können Sie sich an einigen üblichen Szenarien orientieren.

- das heruntergeladene Proxy-Objekt führt den Dienst selbst aus.
  In diesem Fall übernimmt das an Verwender des Dienstes gesendete Objekt alle vom Dienst erwarteten Aufgaben, die Bezeichnung "Proxy" ist dann eher unpassend. Diese Vorgehensweise ist angebracht, wenn der Dienst rein durch Software implementiert wird und die Verwendung externer Ressourcen nicht erforderlich ist. Beispielsweise könnte ein Übersetzungsdienst als Ganzes in Java implementiert und Verwendern zum vollständigen Herunterladen und Ausführen angeboten werden. In diesem Fall ist die Durchführung der Aufgabe keine Verständigungdes Codes mit externen Prozessen erforderlich.
- beim heruntergeladenen Objekt handelt es sich um einen RMI-Stub für die Verständigung mit einem Remote-Dienst.
  Diese Methode ist üblich, wenn es einen zentralen Prozess gibt, der den Dienst implememtiert. Dies könnte beispielsweise ein IMAP-Mail-Dienst sein. IMAP - das Internet Mail Access Protocol - ist eine Möglichkeit der zentralen Speicherung und Verwaltung der E-Mails einer ganzen Gruppe mit Hilfe eines IMAP-Server-Prozesses. Dieser E-Mail-Dienst könnte Jini-Anwendungen das Senden und Empfangen von E-Mails unter verwendung der Übertragungsmechanismen der E-Mail-Datenbank des IMAP-Servers ermöglichen, ohne sie mit den Einzelheiten von IMAP zu konfrontieren. In diesem Fall würde der E-Mail-Dienst Clients ein kleines Objekt zum Herunterladen anbieten, welches einige Methoden umfasst, etwa für das Senden neuer Nachrichten oder für das Anfordern von Listen mit E-Mails. Dieses Objekt würde mittels RMI mit dem "echten" IMAP-Server kommunizieren, der auf einem der Server innerhalb des netzwerks betrieben wird. Auf diese Weise können Jini-Anwendungen und -Dienste einfach ein kleines Proxy-Objekt herunterladen, das die benötigten E-Mail-Schnittstellen implementiert, während die eigentliche Verarbeitung der Nachrichten sicher auf einem Server im Hintergrund erfolgt. Bei diesem Ansatz ist der Jini-Proxy einfach der für das Objekt automatisch erzeugte RMI-Stub.

- das heruntergeladene Objekt verwendet für die Verständigung mit dem Dienst ein privates Kommunikationsprotokoll. Diese Vorgehensweise wird vornehmlich in zwei Fällen gewählt. Im ersten Fall sind bestehende Software-Dienste am Vorgang beteiligt. Jini eignet sich hervorragend, um bestehende, nicht Java-fähige Systeme für Java verfügbar zu machen. Dazu braucht ein Entwickler lediglich das "Bindeglied" zu erstellen - das kleine herunterladbare Java-Objekt, welches die gewünschte Schnittstelle zum bestehenden Dienst implementiert. Dieses Proxy-Objekt kommuniziert daraufhin mit den vom bestehenden System erwarteten Protokollen - Sockets, proprietäre Datenbankanweisungen usw. und stellt ausserdem eine echte Java-Schnittstelle zur Verfügung, die von Jini-Diensten auf einfache Weise verwendet werden kann. Der zweite Anwendungsbereich dieser Vorgehensweise liegt im Anbieten der Funktionalitäten eines Geräts als Dienst. In diesem Fall fungiert der Proxy im wesentlichen als herunterladbarer Gerätetreiber. Wenn es sich bei dem Gerät um einen Drucker handelt, stellt sich das Proxy-Objekt aus der Sicht des Verwenders als "druckerartige" Schnittstelle dar, welche alle zur Verständigung mit dem Drucker erforderlichen proprietären Protokolle implementiert. Die Einzelheiten dieser Implementierung bleiben dem Verwender verborgen.

Das Proxy-Objekt ist der eigentliche Schlüssel zur Fähigkeit von Jini, Diensten und Geräten das Verwalten des für den eigenen Betrieb erforderlichen Code zu ermöglichen. Der auf die Implementierung eines Dienstes zugeschnittene Proxy, welcher zumindest von Entwickler des Dienstes geschrieben wird, kann bei Bedarf von Nutzern des Dienstes heruntergeladen werden. Der Code dieses heruntergeladenen Objekts wird vom Nutzer sicher ausgeführt. Der Verwender weiss nicht einmal, wie der Code des speziellen ihn mit dem Dienst verbindenden

Proxys überhaupt implementiert ist. Der Nutzer eines Druckerdienstes braucht lediglich zu wissen, dass der Proxy die Schnittstelle `printer` implementiert - die Einzelheiten der eigentlichen Kommunikation mit dem speziellen Drucker sind für ihn unerheblich. Ein neu erstellter Dienst veröffentlicht hier seinen Dienst-Proxy über die beiden von ihm gefundenen Lookup-Dienste.

1.9.2.3. Einen Dienst suchen

Wir haben uns bereits damit beschäftigt, wie Dienste an Gemeinschaften teilnehmen, um sich darin zur Nutzung anzubieten. Als nächstes werden wir betrachten, wie Nutzer von Diensten die innerhalb einer Gemeinschaft verfügbaren Dienste auffinden und benutzen können.

Sobald ein Nutzer (bei dem es sich selbst um einen Dienst handeln kann, der einer Gemeinschaft beigetreten ist) über eine Referenz auf einen Lookup-Dienst verfügt, kann er alle Dienstelemente durchsuchen, um die gewünschten Dienste zu ermitteln. Jini bietet mehrere Möglichkeiten des Suchens - Suche kann auf dem Typ des in jedem Dienstelement enthaltenen, herunterladbaren Proxy-Objekts, auf dem eindeutigen Kennzeichner des Dienstes (falls Sie ihn kennen) oder auf den in jedem Dienstelement enthaltenen Attributen basieren. Dieses Auffinden des gewünschten Dienstes ist der wichtigste Vorgang von Lookup. (Es wird Sie nicht überraschen, dass die Schnittstelle `ServiceRegistrar` über eine Methode namens `lookup`() verfügt, die genau dafür zuständig ist.) Unabhängig von der gewählten Vorgehensweise handelt es sich bei dem nach dem Angeben der Suchparameter und dem Aufruf von `lookup`() zurückgegebenen Wert um das Proxy-Objekt des Dienstelements. Nach dem Herunterladen des Proxys verwendet der Client ihn als "Schaltpult" für das direkte Kommunizieren mit dem "Hintergrundmechanismus" des Dienstes, bei dem es sich in der Regel um einen langlebigen Prozess oder um ein mit dem Netzwerk verbundenes Gerät handelt. Die Komplexität des Proxys kann den jeweiligen Anforderungen angepasst werden. Einige "intelligente Proxies" könnten beispielsweise private Kommunikationsprotokolle für den Hintergrundmechanismus implementieren. Andere könnten einfache RMI-generierte Stubs sein, die für die Verständigung mit einem Remote-Objekt im Hintergrundmechanismus verwendet wird.

Nehmen wir einmal an, dass Sie die Software für eine Digitalkamera entwickeln. Die Kamera stellt sich selbst als Dienst zur Verfügung (z.B. mit einer Schnittstelle `DigitalKamera`, welche die Methoden `Schnappschuss`() und `BildHerauskopieren`() enthält), und Sie wollen das direkte Drucken von Bildern von der Kamera aus auf einen beliebigen gerade verfügbaren Drucker ermöglichen. Dazu stellen Sie Ihre Dienste zunächst jedem Interessierten zur Verfügung, indem Sie den bereits erörterten Teilnahmevorgang durchführen. Um jeden verfügbaren Drucker verwenden zu können, fungiert die Kamera als Verwender eines Druckdienstes.

In diesem Fall ist es am wahscheinlichsten, dass die Kamera alle Lookuo-Dienste ihrer Gemeinschaft durchsucht, um Dienstelemente zu ermitteln, welche die Schnittstelle `Printer` implementieren. Die Lookup-Dienste könnten darauf mit einem oder mehreren Proxies antworten, welche diese Schnittstelle implementieren. Daraufhin könnte die Kamera dem Benutzer ermöglichen, sich auf der LCD Anzeige anhand der entsprechenden Namen, Orte oder Kommentare, die den jeweiligen Attributen entnommen sind, einen der verfügbaren Drucker auszusuchen. Wenn der Benutzer ein Bild mit einem bestimmten Drucker druckt, wird dazu die `Printer`-Methode des Proxy-Objekts aufgerufen, welche die Ausgabe an den Drucker sendet.

1.9.2.4. Einheitliche Schnittstellen, unterschiedliche Implementierungen

Möglicherweise haben Sie sich bereits die folgende Frage gestellt: "woher wusste die Kamera, dass sie nach Dienstelementen suchen soll, welche die `Printer` Schnittstelle implementieren?". Selbstverständlich müssen Dienste und Anwendungen zumindest *ein wenig* mit der Semantik der von ihnen aufgerufenen Schnittstellen vertraut sein. Als Entwickler der Software für die Digitalkamera brauchen Sie zwar nicht genau zu wissen, wie die `Printer`-Schnittstelle funktioniert, doch Sie benötigen zumindest eine allgemeine Vorstellung von ihrer Wirkungsweise - sie sendet Daten an den Drucker. Wenn Sie auf eine Schnittstelle mit dem unverständlichen Namen wie beispielsweise `KQW100PD` stossen würden, hätten Sie keine Ahnung, was diese bewirken könnte bzw. welcher Parameter an ihre Methoden zu übergeben und welche Rückgabewerte zu erwarten sind. In einem solchen Fall ist es praktisch unmöglich, diese unbekannte Schnittstelle sinnvoll zu verwenden. Aus diesem Grund implementieren die meisten Jini-Dienste wohlbekannte Schnittstellen und erwarten dies auch von anderen Diensten innerhalb einer gemeinschaft. Nur auf diese Weise können sie sofort nach deren Auffinden wissen, wie sie mit solchen Diensten kommunizieren können.

Gemeinsam mit seinen Partnern und der umfassenderen Gemeinschaft der Jini-Benutzer arbeitet Sun an einer Gruppe allgemeiner Schnitstellen für Drucker, Scanner, Mobiltelefonie, Speicherdienste und anderer üblichen Netzwerkgeräte und -dienste. Die Entwicklung ist noch in vollem Gange. Wann immer dies möglich ist, sollten Entwickler neuer Dienste standardisierte Schnittstellen verwenden, um Jini-Diensten die gegenseitige verwendung zu ermöglichen.

Wenn der Benutzer Ihnen mitteilen kann, wie mit ihr umzugehen ist, können Sie selbstverständlich auch eine unbekannte Schnittstelle verwenden. Aus diesem Grunde unterstützt Jini das Speichern willkürlich serialisierter Java-Objekte als Attribute in Dienstelementen - einschliesslich vollständiger Benutzerschnittstellen. Es wurde bereits erwähnt, dass Nutzer von Diensten auf der Grundlage von Attributen oder basierend auf dem Typ des Dienst-Proxys suchen kann. Eine Digitalkamera könnte beispielsweise nach allen Diensten innerhalb ihrer Gemeinschaft suchen, die über ein Attribut des Typs `java.awt.Component` verfügt. Bei solchen Attributen handelt es sich um Benutzerschnittstellen, die dem Benutzerr angezeigt werden können, um ihm dann das "manuelle" Interagieren mit dem Dienst zu ermöglichen, auch wenn dem Code der Kamera nicht bekannt ist, wie er sich mit diesem Dienst verständigen kann. Daher kann ein Benutzer den Dienst `KQW100PD` möglicherweise an der Benutzerschnittstelle erkennen und ihn direkt mit der Kamera steuern, obgleich dies mit der Software der kamera nicht möglich wäre.

Im Rahmen der Erläuterungen des Dienst-Proxys habe ich bereits einiges über herunterladbaren Code gesagt. Ausserdem sollte erwähnt werden, dass die Jini-Lookup-Infrastruktur selbst diesen Mechanismus verwendet, um Jini-Diensten das Kommunizieren mit unterschiedlichen Lookup-Implementierungen zu ermöglichen. Während der Durchführung des Discovery-Vorgangs antworten alle verfügbaren Lookup-Dienste mit einem Dienst-Proxy, der die Verständigung mit der jeweiligen Implementierung des Lookup-Dienstes ermöglicht. Die gleichen Vorrrichtungen, die das wahlfreie Verwenden von Jini-Diensten unabhängig von der jeweiligen Implementierung und von den für die Kommunikation benutzten Mechanismen erlauben, werden auch von Lookup genutzt.

1.9.3. Leasing

Bislang haben wir mit Discovery und Lookup jene mechanismen kennen gelernt, die in Jini das spontane Formieren von aus Diensten bestehenden Gemeinschaften und das Austauschen von Code zur Realisierung der Verständigung zwischen Diensten ermöglicht. In welcher Weise die Stabilität solcher Gemeinschaften sowie deren Fähigkeiten zur Selbstheilung und zur Bewältigung von (unvermeidlichen) Störungen innerhalb des Netzwerks, Systemabstürzen und Software-Fehlern gewährleistet werden, haben wir bisher noch nicht kennen gelernt. Der Aspekt der Zuverlässigkeit ist besonders wichtig, wenn Software-Systeme mit geringerem oder gar keinem Wartungsaufwand langfristig existieren sollen, wie es bei Jini-Gemeinschaften der Fall ist. Von den Systemen wird erwartet, dass sie monate- oder sogar jahrelang einwandfrei funktionieren.

Erwägen Sie das folgende Beispiel: Ihr Dienst für Digitalkameras nimmt an einer Gemeinschaft teil, indem er sich bei einem Lookup-Dienst jener Gemeinschaft registriert. Vermutlich geschieht dies jeweils, nachdem die Kamera in ihren vernetzten Anschlussadapter eingesetzt oder an einem Computer angeschlossen wird, der mit dem Netzwerk verbunden ist. Die Kamera gibt ihre Verfügbarkeit bekannt, und alles ist in bester Ordnung. Dies stimmt jedoch nur, bis der Benutzer die Kamera einfach aus dem Anschlussadapter herausnimmt, ohne sie vorher auszuschalten. Was geschieht dann? Den andern Mitgliedern der Gemeinschaft stellt sich diese Situation als klassischer teilweiser Ausfall dar - sie können möglicherweise nicht einmal ermitteln, ob der Remote-Host, an dem die Kamera angeschlossen ist, heruntergefahren wurde, lediglich langsam antwortet, ob die Software der Kamera abgestürzt ist oder die Kamera mit einem Hammer zerschmettert wurde. Doch gleichgültig aus welchem Grund die Verbindung unterbrochen wurde, das Gerät hatte wegen der plötzlichen Unterbrechnung der Verbindung der Kamera mit dem Netzwerk keine Möglichkeit, sich abzumelden.

Aus dem Unterbrechen der Netzwerkanbindung durch den Benutzer ohne vorheriges reguläres Abschalten ergibt sich ohne einige Vorkehrungen eine "veraltete" Regsitrierung im Lokup-Dienst der Gemeinschaft. Diensten, welche die Kamera verwenden möchten, wird diese noch immer als verfügbar gemeldet, obgleich dies nicht immer der Fall ist. Noch schwerwiegender sind die im Lookup-Dienst selbst auftretenden Probleme - wenn die Registrierungen von Diensten niemals richtig auf den neusten Stand gebracht werden, kommt es zu einer Anhäufung veralteter Einträge, die den Lookup-Dienst zunehmend belasten.

Solche Anhäufungen sind ein schwerwiegendes Problem langlebiger verteilter Systeme. Es kann unmöglich gewährleistet werden, dass kein Dienst abstürzt oder vom Netzwerk abgekopplet wird, bevor er seine Registrierung aufgehoben hat.

Im beschriebenen Fall beansprucht die Kamera eine *Resource* innerhalb des Lookup-Dienstes. Der Lookup-Dienst verwendet einen Teil seiner möglicherweise knappen oder teuren Speicherkapazität und Rechnerleistung, um die Registrierung der Kamera zu verwalten. Wenn bei Jini für das Reservieren der Resourcen ein traditioneller Ansatz gewählt worden wäre, würde die Registrierung einfach aktiv bleiben, bis sie beendet wird oder ein Systemverwalter die Protokolle durcharbeitet und Registrierungen veralteter Dienste löscht.

Diese Lösung läuft jedoch allem zuwider, was wir von Jini erwarten. Zum ersten ist eine Selbstheilung des Systems nicht gewährleistet: teilweise Ausfälle werden nicht festgestellt und bewältigt, und Dienste, die Resourcen anderer Dienste stellvertretend bereithalten, können grenzenlos anwachsen. Zum zweiten, was wohl noch schlimmer ist, erfordert diese Lösung explizit Eingriffe durch Systemverwalter.

1.9.3.1. Resourcen zeitabhängig reservieren

Zur Bewältigung solcher Schwierigkeiten verwendet Jini eine Technik namens *Leasing*. Leasing basiert auf dem Prinzip, dass der Zugriff auf eine Resource nicht für eine unbegrenzte Zeitspanne gewährt wird. Statt dessen wird dem Verwendet die Ressource für eine bestimmte Zeitspanne "geliehen". Jini-Leases stellen an Nutzer von Ressourcen bei längerfristiger Nutzung die Anforderung regelmässiger Bestätigung des tatsächlichen Bedarfs.

Jini-Leases funktionieren fast genauso wie im "richtigen Leben". Jini-Leases können vom Anbieter des Leases abgelehnt und vom Abnehmer verlängert werden. Ohne Verlängerung laufen Leases zu einem festgelegten Zeitraum ab. Leases können ausserdem vorzeitig beendet werden. Im Gegensatz zum realen Leben werden von Jini dafür keine Strafgebühren erhoben. Schliesslich können leases noch ausgehandelt werden, doch wie im richtigen Leben hat dabei der Anbieter das letzte Wort.

Leases bieten eine konsistente Möglichkeit für das Entfernen ungenutzter oder nicht mehr benötigter Informationen, um mit den vorhandenen Ressourcen schonend umzugehen.: sobald ein Dienst beabsichtigt oder unbeabsichtigt nicht mehr verfügbar ist, ohne seine Registrierung aufzuheben, werden seine Leases nach einiger Zeit ablaufen, und der Dienst wird vergessen. Leases werden für den Lookup-Dienst und, wie wir später sehen werden, auch für andere Aspekte des Systems ausgiebig genutzt. Deshlab ist ein Verständnis des effektiven Nutzens von Leases für das Erstellen und Verwenden von Jini-Diensten wichtig.

Einer der grossen Vorteile des leasings liegt darin, dass es die Wahrscheinlichkeit des Absturzes des gesamten Systems auf ein Minimum reduziert: das System werhälz sich vorsichtig. Falls Sie versäumt haben, die Verwaltung von Leases zu implementieren oder ein Programmfehler dafür sorgt, dass Leases nicht mehr verlängert werden, wird Ihr unzuverlässiger Code einfach aus der Gemeinschaft ausgeschlossen, ohne weitreichende Schäden anzurichten. Jini vereinheitlicht die Handhabung bestimmter Programmfehler, die Leases werden einfach nicht mehr verlängert, bei Netzwerkfehlern und Abstürzen einzelner Rechner. Der Anbieter einer Resource erkennt lediglich, dass der lease eines Dienstes abgelaufen und der Dienst nun nicht mehr verfügbar ist.

Der zweite grosse Vorteil des Leasings besteht darin, dass es Mitgliedern einer Jini-Gemeinschaft eine praktisch einwandfreie persistente Speicherung ermöglicht. Systemverwalter brauchen sich nicht mehr durch Protokolle durchzuarbeiten, um festzustellen, welche Dienste aktiv bzw. inaktiv sind und welche Dienste veraltete Daten im System verteilt haben. Nach einer gewissen Zeit wird die Gemeinschaft erkennen, welche Ressourcen nicht mehr verwendet werden, und diese von veralteten Daten befreien. Dies gleicht der Beseitigung von Viren durch Antikörper. Es wäre selbstverständlich ganz hervorragend, wenn wir sämtliche Rückstände nicht mehr erwünschter Anwendungen soweit nicht verwendbare Treiber und veraltete Aktualisierungen des Betriebsystems so einfach von unseren PCs entfernen könnten.

1.9.3.2. Leasing durch Dritte

Es gibt einen wichtigen Unterschied zwischen Jini-Leases und den meisten Leases im realen Leben: Jini-Leases wurden so gestaltet, dass auch Dritte stellvertretend für andere Leases abschliessen können. Der Dritte durchläuft den gesamten Vorgang des Leases, der lease selbst wird jedoch im Namen des anschliessenden verwenders abgeschlossen. Dies ist mit dem Fall

vergleichbar, dass Ihre Schwiegermutter in ihrem Namen ein hübsches Strandhaus mietet, welches Sie anschliessend beziehen.

1.9.3.3. Hinweis : eine Analogie zum Leasing

Eine Bevollmächtigung stellt in diesem Zusammenhang eine weitaus trefferendere Analogie dar. Wenn Sie jemandem eine Vollmacht erteilen, berechtigen Sie ihn damit, in Ihrem Namen Verträge abzuschliessen. Obgleich der bevollmächtigte unterzeichnet, sind Sie es, der an den Vertrag gebunden ist.

Bei Jini kann es kaum, wenn überhaupt, vorkommen, dass ein skrupelloser Dritter in Ihrem Namen einen Lease abschliesst und anschliessend selbst davon profitiert. Leasing durch Dritte dient demnach annähernd ausschliesslich der Entlastung des Leasingnehmers.

Was nützt diese Vorgehensweise? Sie hat mehrere Vorteile. Vor allem kann der Leasinggeber entlastet werden. Wenn ich mich ausschliesslich der Entwicklung meines Dienstes widmen und mich nicht um die Leasing-APIs, Verlängerungen von Leases, Ablaufen von Leases usw. kümmern möchte, kann ich den Vorgang der Verlängerung von Leases an einen Dritten delegieren. Ich beauftrage den Dritten für eine festgelegte Zeitspanne oder bis ich eine entsprechende Nachricht sende, mit dem Verlängern meines Leases in meinem Namen fortzufahren. Anschliessend kümmert sich der Dritte um die Kommunikation mit dem Leasinggeber, das Aushandeln der Dauer des Leases und seiner rechtzeitigen Verlängerung. Sobald der Leasinggeber meinen Lease aus irgendwelchen Gründen nicht mehr verlängert, wird mir dies vom Dritten mitgeteilt. Dieser Fall bestätigt die soeben erwähnte Analogie der Bevollmächtigung - der gesamte Vorgang dient der Entlastung des Leasingnehmers.

Das Beauftragen Dritter ist auch dann günstig, wenn der einen Lease verwendende Dienst langfristig existieren, jedoch nur selten aktiv sein soll. Ein Dienst, der monatlich Sichererungskopien einer Festplatte anlegt, braucht beispielsweise nur einmal im Monat ausgeführt zu werden. Ein solcher Dienst könnte für die restliche Zeit einfach in einen "Schlafzustand" versetzt werden. Dies kann mit der RMI-*Aktivierungsstruktur* bewerkstelligt werden, welche das automatische persistente Speichern von Java-Objekten und deren spätere Reaktivierung ermöglicht. Die Lebensspanne eines solchen Objekts könnte sehr lange sein, möglicherweise Jahre, die Dauer seines Leases mit dem Lookup-Dienst können jedoch Grössenordnungen kleiner sein, etwa im Minutenbereich. Wenn der monatliche Sicherungsdienst alle zehn Minuten aufwachen müsste, um seinen Lease zu verlängern, würde das dauernde Reaktivieren und Speichern des Dienstes erhebliche Ressourcen verschlingen. Der dazu erforderliche Aufwand würde möglicherweise sogar jenen der eigentlichen monatlichen Aufgabe des Dienstes übersteigen.

Leasing durch Dritte ist hier das Mittel der Wahl. Der monatliche Sicherungsdienst delegiert das Verlängern seines Leases an einen Dritten. Bei diesem Dritten kann es sich allenfalls um einen Dienst handeln, der die Bereitschaft signalisiert hat, langfristige Verlängerungen von Leases stellvertretend für andere Dienste zu übernehmen. Aus diesem Grund wird der Dienst zur Verlängerung von Leases vermutlich von mehreren langlebigen Diensten gemeinsam verwendet. Dieser Dritte arbeitet genauso, wie es im vorherigen Beispiel beschrieben wurde: er verwaltet Listen von Leases und verlängert diese, bevor sie ablaufen. Ausserdem teilt er den Leasingnehmern mit, wenn ein Lease nicht mehr verlängert werden kann.

1.9.3.4. Leasing und Delegieren durch Dritte

Wenn Sie sich mit dem AWT Ereignismodell oder auch mit der Programmierung von JavaBeans bereits auskennen, werden Sie zwischen Leasing durch Dritte und dem

"Delegations-" Ereignismodell Ähnlichkeiten feststellen. Delegieren ermöglicht Dritten den Umgang mit Ereignissen stellvertretend für andere Objekte. Dieses Modell unterscheidet sich vom ursprünglichen AWT Ereignismodell, bei dem Ereignisse jeweils von dem Objekt verarbeitet werden mussten, an die sie gesendet wurden. Das Delegieren ist weitaus flexibler und unterstützt Programmiermuster wie zentralisierte Ereignis-Manager, Verkettung von Ereignissen usw. Auch Leasing durch Dritte ermöglicht einen Grossteil dieser Vorteile. In der folgenden Randbemerkung wird erklärt, wie dies in Jini funktioniert.

1.9.3.4.1. Delegieren von Leases

Mit dem Delegieren von Leases werden wir uns noch ausführlich befassen. Zunächst werden wir uns jedoch mit einem kleinen Code-Segment beschäftigen, damit Sie ein Gefühl dafür bekommen, wie sich das Delegieren einem Programmierer darstellt, und erkennen, wie einfach seine praktische Verwendung ist.

In diesem Beispiel registriert der unten angegebene Code einen Dienst bei einem Jini-Lookup-Dienst. Das von der Registrierung zurückgegebene `ServiceRegistrar` Objekt umfasst einen Lease.

Dieser lease kann an ein Leasing-Delegate eines Dritten übergeben werden, der sich um die Verlängerung kümmert. In diesem Fall handelt es sich bei dem Leasing-Delegat um eine Klasse namens `LeaseRenewalManager`, die mit Jini in der Datei `sun-util.jar` als nicht unterstützter Code ausgeliefert wird.

```
// meinService ist das Dienstelement des anzubietenden Dienstes.
// es enthält sowohl das Proxy-Objekt des Dienstes, als auch die mit ihm assoziierten Attribute
// wir werden uns an dieser Stelle nicht mit allen Einzelheiten von ServiceItem
// auseinandersetzen

ServiceItem meinService = new ServiceItem(.....);

// Lookup ist ein ServiceRegistrar -- ein Lookup-Dienst, den ich per Discovery
// aufgefunden habe. Nun registriere ich mein Dienstelement und beantrage
// eine anfängliche Leasingdauer von fünf Minuten (in Millisekunden)

ServiceResgistration reg = lookup.register(meinService, 1000*60*5);

// da ich mich um meine Leases nicht selber kümmern möchte, verwende ich einen
// LeaseRenewalManager. Ich brauche nur einen zu erstellen und ihn damit zu beauftragen
// meine Leases stellvertretend zu verlängern. Der Parameter lease.ANY gibt an,
// dass der Lease immer wieder verlängert werden soll, bis ich dem Manager
// etwas anderes mitteile. null ist ein optionales Listener-Objekt
// das ich übergeben kann, damit es über Probleme mit der verlängerung eines Leases
// informiert wird

LeaseRenewalManager mgr = new LeaseRenewalManager();
mgr.renewUntil(reg.lease, lease.ANY, null);
```

In diesem Fall delegieren wir den Umgang mit unserem Lease an eine "Bibliotheken" Klasse namens `LeaseRenewalManager`. Die für das Verwalten unserer Leases zuständige Instanz wird mit jener JVM ausgeführt, in der sich auch der Eigner des Leases befindet. Wir

können das Verwalten des Leases jedoch auch an einen gänzlich externen Prozess delegieren - oder sogar an einen Jini-Dienst - der in einer unabhängigen JVM ausgeführt wird.

Mit den Einzelheiten dieses Themas werden wir uns später auch in Beispielen beschäftigen, doch dieses Code-Fragment sollte Ihnen einen kleinen Eindruck des Delegierens von Leases vermittelt haben.

1.9.3.5. Leasing in der Praxis

Bisher habe ich beschrieben, was Leasing ist, jedoch nicht, wie es funktioniert. In Jini erfordert jede Operation, welche die Nutzung einer Resource bewirkt, normalerweise einen Parameter, in dem die Dauer der Nutzung angegeben wird, und gibt daraufhin ein Lease-Objekt zurück. Der Lookup-Dienst vergibt beispielsweise Leases für Registrierungen von Diensten, statt sie dauerhaft zu gewähren. Die Methode `register`() des des `ServiceRegistrar`-Objekts erwartet besipielsweise ein Argument im `long integer` Format, in dem ein Dienst anzugeben hat, wie lange sein Lease dauern soll. Der Lookup-Dienst liefert daraufhin ein Ereignis des Typs `ServiceRegistration`, das Informationen über den soeben registrierten Dienst enthält. Zu den Member-Variablen des `ServiceRegistration`-Objekts gehören unter anderem ein `Lease`-Objekt. Dieses Objekt repräsentiert den Lease, den der Lookup-Dienst gewährt hat, und kann beispielsweise verwendet werden, um den Lease zu verlängern oder zu beenden.

Diese Vorgehensweise weist mehrere Vorteile auf. Zum ersten ist der "Verhandlungsvorgang" minimal - es gibt eine Verhandlungsrunde. Jemand, der einen Lease anfordert, beantragt dabei eine bestimmte Zeitspanne. Der Leasinggeber kann den lease daraufhin gänzlich ablehnen, für die beantragte Dauer genehmigen oder für eine kürzere Zeitspanne zulassen. Diese schnelle Verhandlung ist für fast alle Anwendungen ausreichend und erfordert keinen zeitaufwendigen mehrfache Austausch von Nachrichten zwischen dem Leasingnehmer und dem Leasinggeber.

Zum zweiten finden Leases stets innerhalb festgelegter Zeitspannen statt. Die Dauer eines Leases wird relativ zur gegenwärtigen Uhrzeit angegeben, statt einen fixen künftigen Zeitraum festzulegen. Warum? Um zu verstehen, warum Jini mit relativen Zeitangaben arbeitet, stellen wir uns zunächst vor, wie ein auf absoluter Zeitpunkten basierendes Leasing-System funktionieren würde.

Nehmen wir einmal an, dass Sie eine Lease beantragen, der an einem absoluten Zeitpunkt, etwa am Sonntag um 10:00 ausläuft. Diese Anfrage richten Sie an einen Prozess, der auf einem anderen Rechner irgendwo im Netzwerk abläuft, dessen Systemuhr mit der Uhr Ihres Rechners möglicherweise nicht synchronisiert ist. Möglicherweise geht der andere Rechner sogar schon davon aus, dass der beantragte Zeitpunkt des Ablaufs bereits gewesen ist. Das Synchronisieren der Systemuhren aller Rechner innerhalb eines Netzwerks ist ein kniffliges Problem.

Die von Jini verwendete Alternative - für Leases eine relative Dauer anzugeben - ist weitaus praktikabler, da sie vor erheblich variierenden Auffassungen der jeweiligen Uhrzeit zwischen den einzelnen Rechnern eines Netzwerks schützt. Sogar wenn sich die Uhrzeit des Leasinggebers um Tage von jener des Leasingnehmers unterscheidet, ergibt sich aus einer beantragten Dauer von einer Stunde eine Leasingdauer von ziemlich genau einer Stunde.

1.9.3.5.1. Hinweis - Wie präzise ist relative Zeit?

Selbstverständlich erfordert das Beantragen des leases einen gewissen Zeitaufwand. Ausserdem stimmt Ihre Auffassung von einer Stunde möglicherweise nicht mit jener des Rechners überein, uaf dem sich der Leasinggeber befindet, da dessen Systemuhr zu langsam oder zu schnell laufen könnte. Die Zeitdifferenz, welche sich während der üblichen Zeitspanne eines Leases ergibt, ist jedoch weitaus geringer als die mögliche absolute Abweichung der beiden Systemzeiten. Der Jini-Ansatz ist zwar nicht perfekt (im Zusammenhang mit der Zeit kann innerhalb eines verteilten Systems nichts perfekt sein), stellt jedoch für das Problem der Ablaufzeiten innerhalb eines verteilten Systems eine einfache und brauchbare Lösung dar.

Das von Aufrufen, die Rssourcen beanspruchen, zurückgegebene `Lease`-Objekt ist mit allen Verwaltungsinformationen versehen, die der Leasinggeber verwendet, um die vergebenen Leases zu überwachen. Wenn Sie davon ausgehen, dass es erforderlich sein könnte, diesen Lease in irgendeiner Weise zu bearbeiten, etwa zu beenden oder zu verlängern (und dies wird häufig der Fall sein), sollten Sie dieses Objekt speichern, damit es Ihnen künftig zur Verfügung steht.

Das Verwalten von Leases kann recht schwierig sein. Dies gilt insbesondere, wenn es um mehrere Leases und die entsprechenden Resourcen mit möglicherweise unterschiedlichen Ablaufzeiten geht. Der Umgang mit Leases gehört in der Tat zu den wichtigsten Angelegenheiten beim Entwickeln von Jini-Diensten. Glücklicherweise sind die Lease-Schnittstellen selbst äusserst einfach (Sie brauchen tatsächlich nur zwei Methoden zu verwenden: `renew()` und `cancel()` und die Handhabung von Leases kann weitestgehend automatisiert mit Code aus Bibliotheken durchgeführt werden. Hier betrachten wir das Automatisieren eines Grossteils der Verwaltung von Leases mit Hilfe einiger nützlicher Hilfsklassen.

1.9.3.5.2. Tip - welche Laufzeiten sind für Leases angemessen?

Bei den Erläuterungen von Leases blieb eine wichtige Frage bislang unbeantwortet:
welche Laufzeiten sind für Leases angemessen?
Die Antwort auf diese Frage ist von der jeweiligen Anwendung abhängig. Wenn ihr Dienst äusserst kurzlebig ist oder Verbindungen mit dem Netzwerk häufig aufbauen und wieder beendet, könnte eine Leasingdauer von einigen Minuten angemessen sein.

Falls Ihr Dienst hingegen aktiv und mit dem Netzwerk verbunden ist, können Sie auch längere Laufzeiten beantragen (Stunden oder sogar Tage), um den Netzwerkverkehr für die Verlängerung des leases zu reduzieren. Selbstverständlich können Sie niemals sicher sein, dass längere Leasingzeiten auch eine langsamere Selbstheilung ihrer Gemeinschaft nach einem Absturz Ihres Dienstes oder einen Ausfall des Systems bedingen. Aus diesem Grund sind für Leases in der Regel Laufzeiten von wenigen Minuten angemessen. der gegenwärtige Jini-Lookup-Dienst gewährt für Registrierungen von Diensten standardmässig lediglich Leases mit einer Laufzeit von fünf Minuten, auch wenn Anfrager längere Laufzeiten beantragen.

1.9.3.6. Leasing und Speicherbereinigung im Vergleich

Wenn Sie sich mit Garbage Collection (GC - Speicherbereinigung) bereits auskennen, sind Sie möglicherweise der Ansicht, dass Leasing im Grunde das gleiche bewirkt wie die Speicherbereinigung von Java. In einer Hinsicht stimmt dies auch: sowohl GC als auch

Leasing dienen der Bereinigung ungenutzter Resourcen eines Systems, und Javas RMI-Vorrichtungen beinhalten sogar einen verteilten Speicherbereiniger, der eine Resource innerhalb eines Netzwerks bereinigen kann, wenn keine *Remote*--Referenz darauf existiert.

Was ist also der Unterschied? Der wesentliche Unterschied besteht darin, dass Garbage Collection verwendet wird, um Resourcen zu bereinigen, auf die keine aktive Referenz eines Programms verweist, welche der Verwendung dieser Resource dienen könnte. Beim Leasing existieren normalerweise *grundsätzlich* keine Refernzen auf die entsprechenden Ressourcen. Unser Dienst für die Digitalkamera könnte sich beispielsweise in seinem Lookup-Dienst anbieten, ohne jemals von einem andern Jini-Teilnehmer verwendet zu werden. Dieser Dienst wurde twar bislang nicht verwendet, doch bedeutet dies nicht, dass er gelöscht und seine Registrierung aufgehoben werden sollte - jemand könnte ihn später verwenden wollen. Leasing eignet sich hervorragend für Ressourcen, die zur künftigen verwendung bereitgehalten werden sollen. Im Gegensatz zu GC funktioniert leasing auch dann, wenn der ursprüngliche Verwender einer Ressource diese und irgendwelche entsprechenden Resourcen nicht mehr benötigt.

Nur nebenbei hat der verteilte Speicherbereiniger von RMI Leases intern bereits von Anfang an verwendet.. RMI verwendet leases als Werkzeug, um die verteilten GC-Algorithmen robust zu implementieren und einen Grossteil jener Probleme in den Griff zu bekommen, zu deren Lösung auch Jini dient. Einer der wesentlichsten Unterschiede zwischen lokaler und Remote-Speicherbereinigung bestht beispielsweise darin, dass beim lokalen Fall alle Referenzen auf Objekte bekannt sind, sobald mit der Durchführung der Speicherbereinigung begonnen wird. Im entfernten Fall könnte ein Programm eine Referenz auf ein intern gespeichertes Objekt an einen anderen Rechner weitergegeben haben. Der verteilte Speicherbereiniger muss versuchen, zu ermitteln, ob jener andere Rechner noch immer mit der Referenz auf das Objekt arbeitet, um festzustellen, ob es beseitigt werden kann. Leasing wird als Bestandteil des verteilten Speicherbereiningungsvorgangs benutzt, um das Entfernen von Ressourcen zu ermöglichen, für die möglicherweise bereits abgestürzte Clients über Referenzen verfügen. RMI verbirgt Leases, indem es sie uasschliesslich intern verwendet. Bei Jini wird mit Leases offen beargeitet, und sie sind ein essentieller Bestandteil seines Programmiermodells.

1.9.4. Remote Ereignisse

Analog zu vielen anderen Software Komponenten innerhalb eines Systems, ob es nun verteilt oder lokal ist, müssen Jini-Dienste gelegentlich benachrichtigt, sobald für sie relevante Änderungen auftreten. Beim lokalen Programmiermodell kann es beispielsweise erforderlich sein, eine Komponente zu benachrichtigen, wenn der Benutzer die Maustaste drückt oder ein fenster schliesst.

Dies sind Beispiele für *asynchrone Benachrichtungen.* Dabei handelt es sich um Nachrichten, die direkt an eine Software-Komponente gesendet und ausserhalb des normalen Ablaufs dieser Komponente gehandhabt werden. Statt regelmässig nachfragen zu müssen, ob relevante Änderungen aufgetreten sind, kann die Komponente auf diese Weise bei wichtigen Änderungen "automatisch" benachrichtigt werden. Der asynchrone Charakter dieser Benachrichtigungen kann die Programmierung von Komponenten häufig vereinfachen - der Entwickler einer Komponente braucht ihr keinen Code für das regelmässige Überprüfen irgendeiner externen Einheit auf Änderungen hinzuzufügen.

Für asynchrone Benachrichtigungen verwendet Jini in Übereinstimmung mit der in Java üblichen Vorgehensweise sogenannte *Ereignisse*. Ein Ereignis ist ein Objekt, das

Informationen über eine externe Zustandsänderung enthält, an der eine Software-Komponente interessiert sein könnte. Bei AWT wird beispielsweise ein `MouseEvent` Ereignis gesendet, wenn sich der Zustand der Maus ändert - sobald sie bewegt bzw. eine Maustaste gedrückt oder losgelassen wird. Ereignisse werden dem System mit Hilfe von *Ereigniserzeugern* hinzugefügt, die für das Feststellen von Zustandsänderungen zuständig sind. Bei AWT gibt es einen Thread namens `AWTEventGenerator`, der diese Aufgabe erfüllt. In Java wird ein Ereignis, unmittelbar nachdem es vom System festgestellt wurde, an alle Parteien gesendet, die sich für Ereignisse der jeweiligen Art interessieren. In dieser Hinsicht funktioniert das Ereignismodell von Jini exakt genauso wie das von JavaBeans und den Java Foundation Classes verwendete herkömmliche Ereignismodell. All diese Modelle unterstützen Ereignisse und rufen Methoden von Listenern asynchron auf, sobald Ereignisse auftreten.

1.9.4.1. Remote- und lokale Ereignisse im Vergleich

Das Ereignismodell weist jedoch einige Unterschiede zum "normalen" Java-Ereignismodell auf. Wozu diese Unterscheidung? Sind die Java-Ereignismodelle nicht gut genug? Es hat sich erwiesen, dass die Java-Modelle für den ihnen zugedachten Zweck hervorragend geeignet sind: das Senden asynchroner Benachrichtigungen innerhalb einer einzelnen Java Virtual Machine. Die Welt der verteilten Datenverarbeitung sieht aber ganz anders aus und erfordert daher ein an die Art der in ihr verarbeiteten Programme angepasstes Ereignismodell.

Es gibt einige sehr wichtige Unterschiede zwischen Ereignissen, die lokal, und solchen, die verteilt übermittelt werden sollen.

- im loakeln Fall ist es weitaus einfacher, Ereignisse in der Reihenfolge ihres Auftretens zu übermitteln, da für die lokale Übermittlung von Ereignissen in der Regel eine zentrale Warteschlange verwendet wird, welche als ordnende Instanz fungiert und die Ereignisse in eine feste Reihenfolge zwingt. Da es in verteilten Systemen keinen zentralen Ereignis-Manager gibt und ausserdem beim Transportieren der Ereignisse über das Netzwerk Verzögerungen auftreten können, kann auf Systemen dieser Art ohne erhebliche Leistungseinbussen keine feste Reihenfolge gewährleistet werden. (Javas AWT-Mechanismus für die Verarbeitung von Ereignissen gewährleistet auch für lokale Benachrichtigungen über Ereignisse *keine* feste reihenfolge - im allgemeinen kann eine solche Anordnung in Vorrichtungen für die lokale Benachrichtigung über Ereignisse jedoch relativ einfach implementiert werden.)

- im lokalen Fall kommt eine Benachrichtigung über ein Ereignis stets beim Empfänger an, falls keine katastrophale Störung auftritt (wie beispielsweise ein Absturz der gesamten Anwendung). Für die verschiedenen Arten teilweiser Ausfälle, die in verteilten Systemen auftreten können, sind Einzelsysteme unempfindlich. Innerhalb verteilter Systeme können teilweise Ausfälle wie Abstürze von Rechnern oder Teilnetzen dazu führen, dass Benachrichtigungen über Ereignisse nicht beim Empfänger eintreffen.

- der Aufwand des Sendens eines lokalen Ereignisses ist im Vergleich zu dessen Handhabung in der Regel gering. Normalerweise erfolgt das "Senden" eines Ereignisses einfach durch das Aufrufen einer Methode, welche die interessierte Partei über das Auftreten des Ereignisses informiert. Die vom Empfänger als Reaktion af die Benachrichtigung durchgeführten Berechnungen werden die für das Senden des ereignisses aufgewendete Zeit wahrscheinlich geringfügig erscheinen lassen. Im entfernten Fall ist dies umgekehrt. Das Übermitteln des Ereignisses kann eine um Grössenordnungen längere Zeitspanne erfordern als im lokalen Fall und die für die Handhabung des Ereignisses aufzuwendende Zeit durchaus übersteigen. Diese erheblichen

Unterschiede der Leistung bedingen, dass verteilte Systeme so strukturiert werden sollten, dass möglichst wenige Ereignisse zu verarbeiten sind.

- im lokalen Fall kann der Versender ausserdem davon ausgehen, dass er ein Ereignis sicher an den Verwender übermitteln kann, der eine entsprechende Anfrage gestellt hat. Der entfernte Fall ist hingegen weitaus komplizierter. Der Remote-Empfänger eines Ereignisses könnte vorübergehend vom Netzwerk abgeschnitten sein. Dann wäre es in der Regel wünschenswert, dass der Versender wiederholt versucht, das Ereignis zu übermitteln. Möglicherweise ist es jedoch angebracht, das Ereignis auszusortieren, falls der Empfänger abgestürzt ist. Der Empfänger könnte auch für einige Zeit "inaktiv" und nicht in der Lage sein, das Ereignis zu verarbeiten. Der letztgenannte Fall kann durchaus häufig auftreten, wenn der Empfänger die RMI-Aktivierungsstruktur verwendet. Für ein inaktives Objekt könnte es günstig sein, wenn der versender die an es gerichtete Ereignis speichert, damit es dieses nach der regelmässig erfolgenden Reaktivierung überprüfen kann.

Im entfernten Fall ist ausserdem noch eine Reihe weiterer taktsicher Entscheidungen zu treffen - viele andere Hürden sind zu nehmen. Ist die Reihenfolge der Benachrichtigung über Ereignisse wichtig? Müssen Ereignisse unter allen Umständen übermittelt werden, wenn der Empfänger nicht erreicht werden kann?

Auf diese Fragen gibt es keine allgemeingültigen Antworten. Jede Anwendung muss hier die für sie zweckmässigen Entscheidungen treffen. Eine Online-Banking-Anwendung wird diese Fragen naturgemäss äusserst vorsichtig beantworten - die Ereignisse müssen unter allen Umständen und in korrekten Reihenfolge beim Empfänger eintreffen. Falls der Empfänger nicht empfangsbereit ist, sollen die Ereignisse protokolliert werden, bis sie erneut gesendet werden können. Andere Anwendungen stellen weniger strikte Anforderungen. Ein Online-Spiel könnte beispielsweise in der Lage sein, einige fehlende oder in falscher Reihenfolge übermittelten Ereignisse problemlos zu verkraften.

1.9.4.1.1. So nutzt Jini Ereignisse

Nachdem wir nun die Aufgaben des Jini-Ereignismodells kennengelernt haben, können wir uns damit beschäftigen, wie es funktioniert und was es nützt.

Es gibt viele Fälle, in denen ein Jini-Dienst oder eine Anwendung, die einen Jini-Dienst verwendet, das Empfangen asynchroner Benachrichtigungen über Zustandsänderungen erfordert. Im Abschnitt über Lookup wurde das Beispiel Digitalkamera beschrieben, die in der Lage sein soll, alle innerhalb seiner Jini-Gemeinschaft verfügbaren Drucker zu verwenden. Ich erwähnte, dass diese Kamera in allen auffindbaren Lookup-Diensten nach Diensten suchen würde, welche die Schnittstelle `Printer` implementieren. Dieses Beispiel geht von einer Annahme aus, die ich einfach vernachlässigte: es setzt voraus, dass die Drucker *vor* der Kamera an das Netzwerk angeschlossen werden und verfügbar sind. Was geschieht jedoch im entgegengesetzten Fall? Dann sind unmittelbar nach dem Anschliessen der Kamera keine Drucker verfügbar, da diese erst später hinzugefügt werden. Selbstverständlich wollen wir unabhängig von der reihenfolge, in der die Geräte angeschlossen wurden, in der Lage sein zu drucken.

Die Lösung besteht in der Benachrichtigung der Kamera, unmittelbar nachdem ein verwendbarer Dienst innerhalb der Gemeinschaft verfügbar wird. Die Benutzerschnittstelle einer solchen Kamera ist leicht vorstellbar. Die Schlatfläche DRUCKEN auf dem LCD ist zunächst ausgegraut. Nun schliessen Sie am Netzwerk einen Drucker an, und schon wird die

Schlatfläche DRUCKEN mit Leben gefüllt. Durch eine entsprechende Benachrichtigung hat die Kamera soeben von der Verfügbarkeit eines Druckers innerhalb des Netzwerks erfahren.

Dies ist lediglich ein Beispiel der Verwendung von Ereignissen in Jini. Offensichtlich erzeugt der Lookup-Dienst Ereignisse für interessierte Parteien, sobald Dienste erscheinen, verschwinden oder sich verändern. Doch auch andere Jini-Einheiten können Ereignisse erzeugen bzw. nutzen. Der Printer-Dienst etwa könnte andere Dienste über spezielle Zustände des Druckers informieren, beispielsweise über `OutOfPaper` (kein Papier von dem *entsprechenden* Format mehr im Fach) oder `PrinterJamed` (Papierstau generell). Ausserdem sind Ereignisse nicht auf den Austausch zwischen der bestehenden Jini-Infrastruktur und den Diensten beschränkt, sondern könnte sich auch zwischen den Diensten selbst frei bewegen.

1.9.4.1.2. Das Modell der ereignisorientierten Programmierung

Als nächstes werden wir uns dem Modell der ereignisorientierten Programmierung von Jini zuwenden. Wenn Sie sich mit der programmierung von JavaBeans oder mit den Java Foundation Classes (JFC) bereits auskennen, werden Ihnen diese APIs sehr bekannt vorkommen - und dies ist günstig, da die Programmiermodelle einander tatsächlich sehr ähnlich sind. Es gibt jedoch einige geringfügige Unterschiede, welche durch die Ausrichtung von Jini auf verteilte Datenverarbeitung bedingt sind.

Der auffälligste und am wenigsten überraschende Unterschied besteht darin, dass es sich bei der Hauptschnittstelle, die von Objekten für das Empfangen von Ereignissen verwendet wird, um eine RMI-`Remote`-Schnittstelle namen `RemoteEventListener` handelt. Entsprechend kann seine einzige Methode mit der Bezeichnung `notify`() mit Hilfe von RMI-Objekten von Objekten aufgerufen werden, die sich in anderen Adressräumen befinden uind nötigenfalls eine `RemoteException` auslösen. In der Praxis ergeben sich daraus für Sie beim Entwickeln von Klassen, welche `RemoteEventListener` implementieren, keinerlei Konsequenzen bis auf die Handhabung einiger neuer Arten von Exceptions.

Der zweitgrösste Unterschied zwischen dem "normalen" und dem Jini-Ereignismodell ist die *Kompaktheit* des Jini-Modells. Mit Kompaktheit meine ich die überraschend wenigen Klassen und Methoden. Alle Objekte, die Remote-Ereignisse empfangen sollen, implementieren eine einzige Schnittstelle mit der Bezeichnung `RemoteEventListener` mit ihrer einzigen Methode namens `notify`(). Ausserdem gibt es in Jini für aufgerufene Ereignisse lediglich die Klasse `RemoteEvent`. Bei JFC gibt es hingegen spezielle Ereignisklassen zur Beschreibung von Zustandänderungen der Maus und der Tastatur bis hin zu Änderungen des Zustands der Fenster auf dem Bildschirm. Ausserdem gibt es bei JFC für jeden dieser Ereignistypen spezielle unterstützende Listener-Schnittstellen und Adapterklassen.

Mit seinem eigenen Ereignistyp und der eigenen Art von Listener stellt Jini eine weitaus elegantere Gruppe von Klassen zur Verfügung. Jini schreibt keine grundsätzliche Vorgehensweise für das Signalisieren des Interesses an Ereignissen vor. Jini umfasst keine Schnittstelle, die eine `addRemoteListener`() Methode anbietet. Statt dessen entscheidet jede Komponente, die als Quelle von Ereignissen fungieren könnte, selbst darüber, unter welchen Umständen sie Ereignisse übermittelt, und stellt Empfängern eine eigene Möglichkeit zur Verfügung, ihr Interesse an Ereignissen zu bekunden.

1.9.4.2. Allgemeine Delegation

Wozu diese Einfachheit? Opfert Jini Ausdrucksvielfalt - die Fähigkeit äusserst spezielle Zustandsänderungen zu erfassen und mit Hilfe besonderer Ereignisklassen darzustellen - , indem jedes Ereignis als `RemoteReference` Objekt verarbeitet wird? Einbussen der Ausdrucksvielfalt treten zwar auf, doch die Kompaktheit der Schnittstellen hat sich für eine grundlegende Funktion des Jini-Modells für Remote-Ereignisse als notwendig erwiesen: die Möglichkeit des Erstellens von Delegaten, die unabhängig von der Art und Herkunft der Ereignisse mit jeder Ereignisquelle umgehen können.

Im Abschnitt über Leasing habe ich das Delegieren an Dritte bereits angesprochen und erwähnt, dass Delegation auch in den lokalen Ereignismodellen von JavaBeans und JFC eine wichtige Rolle spielt. Auch in Jini verwenden Remote-Ereignisse Delegation; diese wird jedoch auf andere Weise unterstützt als von Beans und JFC: Jini verfügt über die Fähigkeit, *allgemeine Ereignis-Listener mit Hilfe von Dritten* zu erstellen, die auf jeden Ereignistyp reagieren können.

Im folgenden werden wir uns damit auseinandersetzen, was dies bedeutet. Stellen wir uns einmal vor, dass ich eine Klasse entwickeln möchte, die JFC-Ereignisse aller Art empfangen kann, diese in einer Datei protokolliert und sie an ihre ursprünglichen Empfänger weiterleitet. Wenn ich über eine solche Klasse verfügen würde, könnte ich sie einfach als Listener für sämtliche Ereigniserzeuger innerhalb meiner JFC-Anwendung einsetzen. Die Klasse würde dazu `MouseListener, ActionListener, WindowListener` usw. implementieren.Jene Teile des Codes meiner Anwendung, die diese Ereignisse verarbeiten - indem sie auf Mausklicks usw. reagieren -, würden sich dem Drittobjekt selbst als Listener hinzufügen. Im Grunde würde der Dritte in eine Kette von Ereignisübermittlungen eingereiht werden, damit er alle Ereignisse empfangen und an die eigentlichen Listener weiterleiten kann.

Ich könnte zwar unter Verwendung des lokalen Java-Ereignismodells ein solches Konstrukt erstellen, dies wäre jedoch äusserst aufwendig. Meine Klasse für das Protokollieren der Ereignisse müsste sämtliche JFC-Ereignis-Listener-Schnittstellen mit all ihren spezialisierten Methoden implementieren und alle JFC-Ereignistypen "verstehen" können, um in der Lage zu sein, sie einwandfrei zu protokollieren. Sobald ein neuer Ereignistyp eingeführt würde, beispielsweise weil dem JFC-Toolkit ein neues Steuerelement hinzugefügt wurde, das ein spezielles Ereignis erzeugt, könnte mein Protokollierer nicht mehr funktionieren, da ihm dieser Ereignistyp nicht bekannt wäre.

Jini ist hingegen in der Lage, allgemeine Dritte zu unterstützen. "Allgemeine" bedeutet in diesem Zusammenhang, dass Drittobjekte Jini-Remote-Ereignisse verwenden, weiterleieten und speichern können, ohne spezielle Einzelheiten von ihnen zu kennen. Dies liegt daran, dass alle Jini-Ereignisse bereits einer Klasse zugehörig sind, die von Dritten verstanden wird - es wird auschliesslich mit `RemoteEvent`-Objekten gearbeitet. Die Dritten benötigen keine besonderen Finessen, um mit spezialisierten Typen umgehen zu können, und brauchen lediglich die Methode `notify`() zu implementieren, um alle gegenwärtigen und zukünftigen Jini-Ereignisse empfangen zu können.

Der wesentliche Nutzen der Verwendung Dritter als Ereignis-Listener in Jini-programmen besteht darin, dass ich ein solches Objekt, welches meiner Ereignisverarbeitung einige neue

Verhaltensweisen hinzufügt, nur einmal entwicklen brauche, um es überall verwenden zu können, wo Jini-Ereignisse erzeugt oder empfangen werden. Ich brauche nur einen solchen allgemeinen Dritten zu entwerfen, der die einzige Methode der Schnittstelle `RemoteEventListener` implementiert, um das neue Objekt überall dort verwenden zu können, wo ich ansonsten einen "normalen" Jini-Ereignis-Listener einsetzen würde.

1.9.4.3. Der Ereignis-Pipeline Anwendungsverhaltensweisen hinzufügen

Nach den vorherigen Erläuterungen über asynchrone Banchrichtigung, der Sicherung von Benachrichtigungen usw. wird es Sie möglicherweise überraschen, dass Jini-Remote-Schnittstellen nichts über die Eigenschaften der Vorgehensweisen von Diensten aussagen. Dies liegt darin, dass bei der Entwicklung von Jini davon ausgegangen wurde, dass die zugrundeliegende Infrastruktur für Ereignisse vermutlich niemals allgemein genug sein kann, um alle von unterschiedlichen Anwendungen möglicherweise gewünschtenVorgehensweisen für Benachrichtigungen zu unterstützen. Es ist leicht vorstellbar, wie eine solche API konipiert sein könnte - eine Vielzahl von Methoden mit jeweils diversen Parametern für das Steuern der Ereignisübermittlung. Ein solcher Ansatz wäre bestenfalls unhandlich und könnte trotz allem nicht sämtliche von den Subsystemen der verschiedenen Anwendungen für die Ereignisübermittlung gestellten Anforderungen erfüllen.

Entwickler müssen sich je nach Anwendung entscheiden, welche Anforderungen an die Ereignisübermittlung zu stellen sind. Zu diesem Zweck verwendet Jini statt einer API mit Hunderten von Parametern und Steuerelementen die soeben erörterten Methode des allgemeinen Delegierens. Diese Vorgehensweise erlaubt es, dem Prozess des Sendens, Speicherns und Übermittelns von Ereignissen neue Verhaltensweisen und Anforderungen hinzuzufügen.

Wenn ein bestimmter Dienst das Speichern seiner Ereignisse erfordert, damit er diese später verarbeiten kann, falls er einmal vorübergehend nicht erreichbar ist, kann er einfach ein entsprechendes Delegat implementieren, welches die gewünschte verhaltenweise aufweist. Das Delegat würde alle Remote-Ereignisse empfangen, persistent speichern und auf Anfrage weiterleiten. Dem Ereigniserzeuger erscheint das Speicherdelegat wie jeder andere Verwender von Ereignissen. Der eigentliche verwender der Ereignisse würde das Speicherdelegat als Erzeuger von Ereignissen ansehen.

Wenn ein bestimmter Dienst dafür sorgen muss, dass jedes Ereignis zuverlässig beim Empfänger eintrifft, kann er zu diesem Zweck ein Delegat für "garantierte Zustellung" implementieren, welches alle Ereignisse empfängt, persistent speichert und immer wieder versucht, das Ereignis zu übermitteln, bis der Empfänger dessen Ankunft bestätigt. Auch in diesem Fall erscheint das Delegat dem ursprünglichen Erzeuger als Verwender und dem eigentlichen Verwender als Erzeuger. Sowohl der ursprüngliche Erzeuger als Verwender als auch der eigentliche Verwender können dieses Delegat benutzen, um ihre grundlegende Vorgehensweise für das Verarbeiten von Ereignissen zu erweitern.

Die Einfachheit der Schnittstelle bedingt ausserdem, dass Delegate miteinander kombiniert werden können - Sie können beispielsweise ein protokollierungsdelegat mit einem Delegat für garantierte Zustellungen kombinieren. Durch das Kombinieren mehrerer Delegate können Sie eine "Pipeline" für die Ereignisverarbeitung konstruieren, bei der die Ergebnisse einer Station jeweils an die nächste weitergeleitet werden. Durch die einfachen Schnittstellen können die einzelnen Delegate hervorragend Hand in Hand arbeiten. Ausserdem vereinfachen es diese

Schnittstellen allen beteiligten - dem Ereigniserzeuger, dem Ereignisempfänger und auch sonstigen Interessierten - , der Ereignis-Pipeline Delegate hinzuzufügen.

Wie gesagt stellen verschiedene Anwendungen an die Vorgehensweise für die Benachrichtigung über Ereignisse unterschieldiche Anforderungen, und dies ist der wesentliche Grund für die Vorrichtungen von Jini für das Verwenden Dritter als allgemeine Ereignisdelegate. Dafür gibt es eine Reihe üblicher Anwendungsbereiche. Sie können Hilfsklassen für das Zwischenspeichern und Weiterleiten sowie für das auf einige vom jeweiligen Dienst bestimmte Parameter basierendes Filtern von Ereignissen verwenden oder auch einen "Postkaste" für Ereignisse bauen.

1.9.4.4. Ereignisse und Leasing

Nachdem wir uns nun sowohl mit Leasing als auch mit Ereignissen beschäftigt haben, können wir uns eine berechtigte Frage stellen: was geschieht, wenn ich mich als Interessent für Ereignisse registriere und anschliessend einen Absturz erleide oder vom Netzwerk getrennt werde? Werde ich bis in alle Ewigkeit in der Liste der Empfänger von Ereignissen erfasst sein?
Die Antwort auf diese Frage ist naheliegend und besteht darin, dass Registrierungen von Ereignisempfängern beim Ereigniserzeuger geleast werden.

Wie beim Lookup-Dienst wird durch Leasing dafür gesorgt, dass sich das System selbst aufräumt. Wenn Sie sich für das Empfangen von Benachrichtigungen über Ereignisse registriert haben, müssen Sie ihr Interesse regelmässig bestätigen, um dauerhaft Ereignisse empfangen zu können. Falls Sie diesen Nachweis nicht erbringen, weil Sie entweder abgestürzt sind, Fehler aufweisen oder Probleme mit dem Netzwerk aufgetreten sind, läuft die Registrierung Ihres Interesses ab, und Sie werden aus der entsprechenden Quelle nicht mehr über Ereignisse informiert. Die meisten Ereigniserzeuger unterstützen eine Funktion in der Art von `register` und geben ein Objekt zurück, das die Schnittstelle `EventRegistration` implementiert. Diese Schnittstelle erlaubt den Zugriff auf das lease Objekt, welches die Dauer des Interesses an dem Ereignis repäsentiert.

Obschon Suns Implementierung des Jini-Lookup-Dienstes unterschiedliche Mechanismen für Registrierungsreleases für Dienste und Ereignisse anbietet, brauchen sich andere Dienste nicht an diese Vorgehensweise zu halten. Wenn Sie einen eigenen Dienst entwickeln, können Sie sich dafür entscheiden, nur einen Leasing-Mechanismus anzubieten, bei dem beispielsweise alle Leases eines Clients gleichzeitig ablaufen.

Ein Dienst sollte grundsätzlich nicht versuchen, Ereignisse an Clients zu senden, deren Registrierung abgelaufen ist. Ein solches verhalten wird unter Jini als "unanständig" aufgefasst.

Kombinieren von Ereignisdelegaten

Als nächstes werden wir uns mit einem Code-Segment beschäftigen, welches das Ineinandergreifen mehrerer Delegate beim Zusammenfügen einer Ereignis-Pipeline veranschaulicht. Dies ist ein hypothetisches Beispiel, da Jini im Moment noch keine Ereignisdelegate umfasst. Mit dem Entwickeln von Delegaten werden wir uns später befassen, doch der unten angegebene Code soll Ihnen einen Eindruck davon vermitteln, wie Delegate miteinander kombiniert werden können.

Gehen wir einmal davon aus, dass wir bereits einen Jini-Dienst entwickelt haben, der eine zuverlässige benachrichtigung über Ereignisse durch einen Lookup-Dienst erfordert. Da dieser Dienst ausserdem zumeist inaktiv ist, wäre es nicht effektiv, ihn immer wieder zu aktivieren, nur um ihn ein Ereignis empfangen zu lassen. Deshalb entscheiden wir uns für das Kombinieren einiger Delegate, die (1) gewährleiten, dass alle Ereignisse berücksichtigt werden, und (2) Ereignisse ansammeln und regelmässig senden (beispielsweise täglich oder wöchentlich).

Im wesentlichen ergibt sich dieses Verhalten daraus, dass unser neuer Dienst RemoteEventListener implementiert, um Ereignisse empfangen zu können. Auch LoggingDelegate und ReliableDeliveryDelegate sind RemoteListeners. Wenn unser neuer Dienst beim Jini-Lookup-Dienst angibt, an Ereignissen interessiert zu sein, gibtt er als Empfänger nicht sich selbst, sondern LoggingDelegate an. In gleicher Weise fordert er das LoggingDelegate auf, seine Ereignisse an ReliableDeliveryDelegate zu senden. Da jedes dieser Delegate die erforderliche Schnittstelle implementiert, kann er in jeder API verwendet werden, die RemoteEventListener empfängt.

Lassen Sie uns nun etwas genauer betrachten, wie dies funktionieren könnte:

```
ReliableDeliveryDelegate rdd =
     new ReliableDeliveryDelegate(this);
// als nächstes wollen wir ein Protokollierungsdelegate in die
// Pipeline einfügen (näher am Ereigniserzeuger). Daher
// erstellen wir ein solches Delegat und senden seine Ausgabe
// an das bereits fertige Delegat für zuverlässige Zustellung
LoggingDelegate ld = new LoggingDelegate(rdd);
// nun werden alle vom Protokollierungsdelegate empfangenen
// Ereignisse an das Delegat für zuverlässige Zustellung
// gesendet, welches sie anschliessend an uns weiterleitet.
// Der ursprüngliche Ereigniserzeuger ist der Pipeline jedoch
// noch hinzuzufügen. Lookup ist in diesem Fall ein Jini-
// Lookup-Dienst.
// Mit Hilfe der Methode register() signalisieren wir unser
// Interesse an Ereignissen, die übermittelt werden sollen.
// Statt dessen werden wir uns auf das Konstruieren der
// Pipeline konzentrieren. Der Parameter von register() sollte
// vom Typ RemoteEventListener sein, und legt fest, wohin der
// Lookup Dienst seine Ereignisse senden soll.
EventRegistration regl = lookup.register(...,rdd);
```

Damit dieser Code verwendbar wird, müssen die Delegationsklassen - `LoggingDelegate` und `ReliableDeliveryDelegate` irgendeine externe Einheit erstellen oder wiederverwenden. Wenn die Delegate ausschliesslich innerhalb des Dienstes existieren würden, von dem sie erstellt wurden, wären sie selbstverständlich nutzlos - wenn unser Prozess für jedes gesendete Ereignis aufgerufen werden müsste, um es zu protokollieren, dann könnten wir uns das Protokollierungsdelegate gleich sparen und die Ereignisse unmittelbar selbst verarbeiten. Deshalb werden diese Delegate durch das Erstellen (oder Wiederverwenden) externer Prozesse implementiert, die den Code, der sie erstellt hat, unabhängig sind. Vermutlich wird es sich bei diesem externen Prozess um einen per Lookup aufgefundenen Jini-Dienst handeln.

Mit Hilfe dieser Vorgehensweise ist es uns gelungen, die Pipeline zu konstruieren. Weitere Aspekte wie beispielsweise die Handhabung des leasings innerhalb solcher Delegate haben wir zunächst ausser acht gelassen.
Wie bereits erwähnt, sind die hier vorgestellten Delegate rein hypothetischer Art, da solche Delefate in Jini bislang nicht enthalten sind. Diese Art des Kombinierens von Funktionen wurde bei der Entwicklung von Jini jedoch explizit berücksichtigt.

1.9.4.5. Folgenummern und Transaktionen

Obgleich wir nicht ins Detail gehen wollen, werden Sie einige Aspekte von Ereignissen kennen lernen, die für Dienste wichtig sind, und welche eine sichere Zustellung von Ereignissen gewährleisten sollen. Jedes `RemoteEvent` ist mit einer *Folgenummer* versehen, die verwendet werden kann, um es in bezug auf andere Ereignisse der gleichen Ereignisquelle einzuordnen. Listener (auch Dritt-Listener), für die die verarbeitung empfangener Ereignisse in korrekter Reiehnfolge wichtig ist, können die ursprüngliche reihenfolge mit Hilfe der Ereignisnummer wiederherstellen und erkennen, ob einzelne Ereignisse nicht empfangen wurden.

Ausserdem sind Folgenummern für Jini-Transaktionen, die im nächsten Abschnitt erklärt werden, von Bedeutung. Da es sich hierbei jedoch um ein fortgeschrittenes Thema handelt, verschiebe ich die nähere Erläuterungen von Folgenummern und der Zusammenarbeit zwischen Ereignissen und Transaktionen auf später.

1.9.5. Transaktionen

Nun ist es an der Zeit, dass wir uns mit dem letzten der fünf Grundkonzepte von Jini befassen: *Transaktionen.* Leider stellen Transaktionen das wohl schwierigste Grundkonzept von Jini dar, sie werden jedoch glücklicherweise in vielen Anwendungen gar nicht benötigt.

Wir haben uns bereits mehrfach mit dem Erfordernis der Zuverlässigkeit und Robustheit innerhalb verteilter Systeme auseinandergesetzt, der vor allem die Tücken teilweiser Ausfälle im Wege stehen. Wie bereits erwähnt, wird von teilweisen Ausfällen gesprochen, wenn eine oder mehrere der für eine Operation benötigten Komponenten ausfällt. Wenn einfach *alle* an der Operation beteiligten Komponenten ausfallen würden, wäre das Wiederherstellen einfach - Ihnen wäre bekannt, dass die Operation in jeder Hinsicht fehlgeschlagen ist, und Sie könnten es anschliessend einfach erneut versuchen, und wenn alle Komponenten ihre Aufgaben erfolgreich bewältigt haben, ist selbstverständlich keinerlei Wiederherstellung nötig.

Aus der Sicht des Programmierers ist es am ungünstigsten, wenn nur Teile einer Operation erfolgreich durchgeführt werden. Lassen Sie uns in diesem Zusammenhang zunächst das klassische beispiel der Datenbankprogrammierung im lokalen Fall betrachten. Datenbanken von Banken müsen zuverlässig sein, da Millionen- und sogar Milliardenbeträge täglich mit ihnen verarbeitet werden. Ein Grossteil der finanziellen Transaktionen erfolgt in Form von Buchungen zwischen zwei Konten: das Geld wird dem einen Konto entnommen und dem andern hinzugefügt. Denken Sie einmal darüber nach, wie Sie eine solche Buchung programmieren würden. Vermutlich würden ein Code-Segment entwickeln, welches das Abbuchen des Betrags von Konto A bewirkt, und ein zweites, das dem Konto B den gleichen betrag wieder hinzufügt.

Dies erscheint trivial, bis Sie anfangen, über mögliche Störungen und Ausfälle nachzudenken. Was geschieht, wenn Ihr System abstürzt, unmittelbar nachdem Konto A um den Betrag

vermindert wurde? Konto A könnten mehrere Millionen CHF entnommen worden sein, die niemals bei Konto B angekommen sind! Das Geld ist einfach ins Nirvana entschwunden, oder auf Ihr Konto?!?

Das den Buchungsfehler verursachende Programm ist nicht dafür ausgelegt, teilweise Ausfälle zu tolereieren.

Verteilte Systeme sind für Probleme mit teilweisen Ausfällen besonders anfällig. Jedes Stadium Ihrer Operation könnte das herstellen einer Verbindung mit einer speziellen Komponente irgendwo im Netzwerk erfordern. Während der Durchführung der Operation können diese Komponenten abstürzen, oder das Netzwerk könnte instabil werden. Wie reagieren Sie darauf? Sie können immer wieder versuchen, Verbindungen mit den nicht erreichbaren Komponenten aufzubauen und schaffen dies hoffentlich, bevor Ihr System abstürzt, oder die erfolgreichen Komponenten kontaktieren und auffordern, die soeben vorgenommenen Änderungen rückgängig zu machen. Selbstverständlich kann dabei niemals ausgeschlossen werden, dass nicht alle Aufforderungen zum Rückgängigmachen erfolgreich verarbeitet werden.

1.9.5.1. Die Datenintegrität gewährleisten

In diesem Zusammenhang treten Transaktionen auf den Plan. Transaktionen sind eine Möglichkeit des Gruppierens zusammenhängender Operationen, damit diese ausschlisslich: entweder sind alle Operationen erfolgreich, oder alle schlagen fehl. In beiden Fällen nimmt das System einen bestimmten Zustand an, in dem es recht einfach ist, sich für die weitere Vorgehensweise zu entscheiden - war die Transaktion erfolgreich, so wird einfach fortgefahren, ansonsten wird sie später erneut versucht.

Transaktionen ermöglichen Datenmanipulation mit vier grundlegenden Eigenschaften, die häufig als ACID-Eigenschaften (ACID : Atomicity, Consistency, Isolation, Durability - Atomizität, Konsistenz, Unabhängigkeit, Dauerhaftigkeit):

- Atomizität:
  die innerhalb einer Transaktion gruppierten Operationen werden entweder alle erfolgreich ausgeführt oder schlagen fehl. Sie werden ausgeführt, als würde es sich um eine einzelne atomare Operation handeln.

- Konsistenz:
  nach dem Abschluss einer Transaktion sollte sich das System in einem widerspruchsfreien und verständlichen Zustand befinden. Da letztlich nur der Benutzer eines Systems übre dessen Freiheit von Widersprüchen zu befinden vermag, dienen Transaktionen vornehmlich der Gewährleistung von Konsistenz, können diese jedoch nicht garantieren.

- Isolation:
  bis sie vollständig ausgeführt sind, beeinflussen Transaktionen einander nicht. Die Auswirkungen von Transaktionen sind während ihrer Ausführung von anderen Operationen nicht feststellbar. Diese Eigenschaft sorgt dafür, dass keine Berechnungen auf der Grundlage von Daten erfolgen, denen im Fall eines Fehlschlages einer Transaktion wieder ihre ursprünglichen Werte zugewiesen werden.

- Dauerhaftigkeit:
  sobald eine Transaktion erfolgreich abgeschlossen wurde - all seine Änderungen also wirksam geworden sind - dürfen ihre Auswirkungen nicht durch eine nachfolgende

Störung oder einen Absturz verlorengehen. Die Ergebnisse der Transaktion müssen zumindest genauso persistent sein, wie die Einheiten, welche sie verwenden.

Transaktionen erfolgen durch das Koordinieren aller beteiligten Komponenten mit Hilfe einer zentralen Einheit. Dazu müssen alle Beteiligten aufeinander abgestimmt werden und ihre Zustände an den sogenannten *Transaktionsmanager* übermitteln. Da der Transaktionsmanager in bezug auf die beteiligten Komponenten eine zentrale Stellung einnimmt, verfügt er über einen "privilegierten" Betrachtungswinkel, von dem aus er alle Beteiligten kontrolliert.

In lokalen Fällen wie bei unserem Beispiel der Datenbank einer Bank übernimmt in der Regel die Datenbank selbst die Aufgabe des Transaktionsmanagers. Kommerzielle Datenbanken verfügen über umfassende Vorrichtungen, um sicherzustellen, dass Gruppen von Operationen innerhalb einer Transaktion entweder alle erfolgreich ausgeführt werden oder alle fehlgeschlagen. Die Datenbank protokolliert entsprechende Aufgaben, damit sie nach einem Absturz während der Ausführung einer Transaktion feststellen kann, ob eine Transaktion vollständig ausgeführt wurde oder fehlgeschlagen ist.

Verteilte Datenbanken - bzw. jedes System, das Komponenten umfasst, die auf mehreren Geräten ausgeführt werden - sind zweifellos etwas komplizierter. Im lokalen Fall bedingt ein Absturz der Datenbank die gleichzeitige Unterbrechnung aller an der Transaktion beteiligten Operationen. Die Datenbank kann solche Abläufe exakt protokollieren, da sämtliche Operationen innerhalb einer Transaktion die lokale Datenbank betreffen. Sie werden "innerhalb" der Datenbank selbst durchgeführt.

Innerhalb verteilter Datenbanken können Operationen auf Remote-Rechnern durchfgeführt werden, die nicht der unmittelbaren Kontrolle der Datenbank unterliegen. Es kann vorkommen, dass die Datenbank nicht feststellen kann, ob eine auf einem Remote-Rechner durchgeführte Operation erfolgreich verlaufen oder fehlgeschlagen ist. Ein solcher Fall erfordert einen etwas komplizierteren Ansatz, damit die ACID-Eigenschaft bei Veränderungen unserer Daten eingehalten werden können.

1.9.5.2. Two-Phase Commit

Das für die Verwendung in Systemen dieser Art übliche "Protokoll" wird als *Two-Phase-Commit* (2PC) bezeichnet. Der Name ist eine Anspielung auf die beiden Phasen bzw. Stadien dieses Protokolls, welche alle Beteiligten durchlaufen müssen, bevor die Transaktion als Ganzes erfolgreich verläuft oder fehlschlägt.

Als nächstes werden wir uns mit der allgemeinen Funktionsweise von Two-Phase-Commit innerhalb traditioneller Transaktionsmanager wie Datenbanken befassen. Selbstverständlich nehmen reale Datenbanken über die nachfolgend beschriebene Vorgehensweise hinaus noch diverse Optimierungen vor, doch hier geht es vor allem um das Verstehen des Prinzips von Two-Phase-Commit.

Zunächst werden alle Ergebnisse einer Gruppe von Operationen errechnet und temporär gespeichert. Anschliessend werden die zwischengespeicherten Ergebnisse vom temporären in den eigentlichen Speicher übertragen. Auf diese Weise ist gewährleistet, dass alle Operationen abgeschlossen sind - falls das System zwischendurch abstürzt, wurden noch keine Daten geändert, und das System kann versuchen, die Operationen erneut durchzuführen, sobald es wieder aktiviert wurde.

Diese einfache Definition ist zwar im Prinzip korrekt, lässt jedoch einige Einzelheiten des Protokolls ausser acht. Sobald wir uns mit Two-Phase Commit etwas eingehender beschäftigen, stossen wir auf eine diesem Protokoll eigene Terminologie. Da wir uns mit dieser Terminologie und mit einer genaueren Beschreibung der Funktionsweise von Two-Phase Commit zu einem späteren Zeitpunkt noch auseinander setzen werden, werden wir unsere einfache definition des Protokolls an dieser Stelle bereits etwas ausbauen und einige Ausdrücke kennenlernen, welche Sie während der Ausführungen über Jini-Transaktionen wiedererkennen werden.

Zunächst bündelt der Transaktionsmanager alle die Transaktionen konstituierender Operationen. Diese werden als *Teilnehmer* einer Transaktion bezeichnet. Als nächstes fordert der Manager alle Telnehmer auf, in eine *Precommit*-Phase einzutreten. Dies bedeutet, dass alle Teilnehmer ihre Berechnungen in der üblichen Weise durchführen, Änderungen jedoch nicht permanent, beispielsweise direkt in die Datenbank oder auf der Festplatte, sondern in einem temporären Berich speichern. Alle Teilnehmer teilen dem Manager mit, ob sie erfolgreich in die Precommit-Phase eingetreten sind.

In der Precommit-Phase fordert der Transaktionsmanager alle einzelnen Operationen auf, entweder abzubrechen oder teilzunehmen. Der Transaktionsmanager in der ersten Phase alle zur Transaktion gehörenden Operationen, ob sie zur Teilnahme am Commit bereit sind. nehmen wir an, dass in diesem Fall alle Operationen ihre Bereitschaft erklären.

Als nächstes fasst der Transaktionsmanager die Ergebnisse der einzelnen zum Eintreten in die Precommit-Phase aufgeforderten Teilnehmer zusammen. Sobald eine der beteiligten Operationen fehlschlägt, werden alle anderen Teilnehmer zum *Abbruch* aufgefordert. Dies bedeutet, dass sie ihre temporären Ergebnisse vergessen können und nicht in den permanenten Speicher schreiben sollen. Sind jedoch alle beteiligten Operationen erfolgreich verlaufen, dann werden die Teilnehmer zum *Commit* aufgefordert, welcher das permanente Speichern der Änderungen bewirkt. Alle Teilnehmer werden übereinstimmend entweder zum Abbruch oder zum Commit aufgefordert.

Der Transaktionsmanager ist für die einwandfreie Zusammenarbeit zwischen den teilnehmern und für deren Betreuung während des Durchlaufens der einzelnen Phasen des Vorgangs zuständig - Precommit und Abbruch bzw. Commit.

Doch was geschieht, wenn eine der Komponenten des Systems während des Two-Phase Commit Vorgangs ausfällt? Jede einzelne am Commit Prozess beteiligte Komponente, einschliesslich aller Operationen und des Transaktionsmanagers selbst, protokollieren die von ihnen erfolgreich erreichten Zustände. Sobald einer der Teilnehmer die Precommit Nachricht empfängt, legt er einen Protokolleintrag in permanentem Speicher ab, der sowohl die unveränderten Werte als auch die neuen Werte der Daten umfasst. Die unveränderten Werte werden benötigt, falls die Operation abgebrochen wird und die neuen Werte ersetzten die bisherigen, falls die gesamte Transaktion erfolgreich verläuft. Das "Merken" dieser Werte ist erforderlich, damit der gegenwärtige Zustand im Fall eines Absturzes nach dem Informieren des Transaktionsmanagers über die Bereitschaft zum Commit wiederhergestellt werden kann.

Jedesmal wenn der Transaktionsmanager eine der Antworten auf seine Aufforderungen zum Precommit erhält, erstellt er einen entsprechenden Protokolleintrag. Wenn alle Anworten die Bereitschaft der Teilnehmer zur Fortführung der Operation signalisieren, legt der Manager einen "OK"-Datensatz an und übermittelt anschlissend die Commit-Nachricht an alle Beteiligten. Falls jedoch eine der Operationen meldet, dass sie nicht zur Fortführung des

Vorgangs bereit ist, erfasst der Transaktionsmanager dies in seinem Protokoll und fordert alle Operationen zum Abbruch auf.

Das Ablegen dieser Nachricht im Protokoll des Transaktionsmanagers entspricht dem Übergang von der Phase 1 des Protokolls zur Phase 2. Falls der Manager abstürzt, bevor diese Aktualisierung des Protokolls durchgeführt wurde, kann er davon ausgehen, dass die Precommit-Phase nicht vollständig durchgeführt wurde, und sie erneut einleiten. Wenn er einen positiven Eintrag im Protokoll vorfindet, kann er die Commit-Phase des Protokolls neu starten.

Sie brauchen diesen Vorgang nicht unbedingt in allen Einzelheiten zu kennen. Diese Beschreibung bezieht sich auf die Verwendung von Two-Phase Commit durch Datenbanken, welche sich geringfügig von der Vorgehensweise von Jini unterscheidet. In diesem Zusammenhang geht es jedoch nur um ein grundsätzliches Verständnis des Protokolls, welches für das Verstehen der Funktionsweise von Transaktionen wichtig ist.

1.9.5.3. 2PC in Jini

Nachdem wir nun gesehen haben, wie es bei Datenbanken funktioniert, können wir uns damit beschäftigen wie Jini 2PC implementiert. Überraschend ist die Tatsache, dass Jini Two-Phase Commit im Grunde *gar nicht* implementiert. Das Jini-Transaktionsmodell folgt einem sehr einfachen und äusserst objektorientierten Ansatz: Jini geht davon aus, dass Two-Phase Commit lediglich ein Protokoll bzw. eine Schnittstelle ist. Weitere Einzelheiten zu diesem Thema besprechen wir noch. Die tatsächlichen Abläufe während der Ausführung des Protokolls obliegen im einzelnen der Implementierung - sobald die Methoden der Schnittstelle aufgerufen wurden.

Dieser Ansatz bedingt, dass alle Teilnehmer an einer Jini-Transaktion die Schnittstelle `TransactionParticipant` implementieren. Die Namen der Methoden dieser Schnittstelle sind dem herkömmlichen Two-Phase Commit Protokoll entlehnt: `prepare`(), `commit`() und `abort`(). Doch es handelt sich lediglich um eine Schnittstelle - Jini schreibt nicht vor, was die Teilnehmer zu tun haben, wenn eine dieser Methoden aufgerufen wird. Die Implementierung der Semantik für Transaktionen ergibt sich aus den Implementierungen der beteiligten Klassen. Weitere Einzelheiten zu diesem Thema finden Sie unten.

Im Gegensatz zum Transaktionsmanager einer Datenbank, der rigoros dafür sorgt, dass keine im Rahmen einer Transaktion durchgeführte Operationen die an sie gestellten Anforderungen unterlaufen kann, führt der Jini-Transaktionsmanager das Two-Phase Commit Protokoll lediglich stellvertretend für seine Teilnehmer durch. Es durchläuft das Protokoll, ruft die TransactionParticipant Methode aller Teilnehmer auf, behandelt sie in Abhängigkeit von den Ereignissen der Gruppe und ruft schliesslich die `commit`() bzw. `abort`() Methode aller Teilnehmer auf. Die reaktionsweise der einzelnen Teilnehmer auf diese Aufrufe der Methode liegt gänzlich in deren eigener Verantwortung.

Die Schnittstelle TransactionParticipant

Zunächst betrachten wir die Schnittstelle selbst, um zu erkennen, wie stark die Schnittstelle `TransactionParticipant` an die Terminologie und Semantik von Two-Phase Commit angelehnt ist.

```
public interface TransactionParticipant {
     public int prepare(TransactionManager mgr, long id);
```

```
     public void commit(TransactionManager mgr, long id);
     public void abort(TransactionManager mgr, long id);
public int prepareCommit(TransactionManager mgr, long id);}
```

Wenn Sie in Ihren Dienst irgendeine Operation exportieren, die im Rahmen einer Transaktion verwendet werden soll, können Sie die Schnittstelle `TransactionParticipant` implementieren. Dann wird der Transaktionsmanager die entsprechenden Methoden in der Ihnen angemessenen erscheinenden Weise implementieren, doch die Semantik Ihrer Implementierung sollte mit dem übereinstimmen, was die Clients während eines Two-Phase Commit erwarten.

Die Methode `prepare`() ermöglicht dem Transaktionsmanager, den Teilnehmer (Ihren Code) zum Übergeben in die Precommit Phase aufzufordern. Als Parameter werden der die Transaktion durchführende Transaktionsmanager und die ID der zu verarbeitenden Transaktion angegeben. Ihr Code antwortet darauf mit einem Rückgabewert, der dem Transaktionsmanager signalisiert, ob der Übergang in die Precommit Phase erfolgreich verlaufen oder ein Abbruch erfolgt ist.

Die Methode `commit`() wird vom Transaktionsmanager aufgerufen, sobald alle Teilnehmer ihm den erfolgreichen Übergang in die Precommit-Phase bestätigt haben. Falls ein Teilnehmer nicht in der Lage war, zur Precommit Phase überzugehen, wird die Methode `abort`() aufgerufen. Nach einem Aufruf von `commit`() sollte Ihre Operation die Änderungen permanent speichern, und nach einem Aufruf der Methode `abort`() sollte sie dafür sorgen, dass die ursprünglichen Werte unverändert bleiben.

Die Methode `prepareCommit`() ermöglicht dem Transaktionsmanager eine Optimierung des Vorgangs. Falls Ihr Client der letzte Teilnehmer innerhalb einer Transaktion ist und alle vorherigen Teilnehmer deren Bereitschaft zum Commit bestätigt haben, kann der Transaktionsmanager diese Methode aufrufen, damit der Commit nach Möglichkeit unmittelbar durchgeführt wird. Sie können durch den Rückgabewert einen Abbruch signalisieren, damit auch alle vorherigen Teilnehmer zum Abbruch aufgefordert werden.

Wie Sie sehen, ist Jini weitaus weniger restriktiv und potentiell erheblich unsicherer hinsichtlich der verarbeitung von Transaktionen. Jini *ermöglicht* Objekten das Teilnehmen an Transaktionen, statt sie dazu *aufzufordern.*

In der Praxis erweist sich dies als vorteilhaft: Jini-Dienste kennen die an sie in bezug auf die Integrität der Daten gestellten Anforderungen und können so entwickelt werden, dass sie lediglich den ihnen gemässen Teil der Infrastruktur für Transaktionen implmentieren. Wenn Sie beispielsweise eine Anwendung für eine Bank in Jini erstellen, können Sie eine Semantik für sichere Transaktionen implementieren, die alle Eigenschaften des Two-Phase Commit einer kommerziellen Datenbank aufweist. Falls Sie jedoch einen Online-Chat entwickeln, können Sie dafür einen erheblich einfacheren Ansatz wählen, der möglicherweise nicht die volle Integrität der verarbeiteten Daten nach Atomexplosionen oder implodierenden Sonnen gewährleistet.

1.9.5.4. Jini-Transaktionen verwenden

Glücklicherweise ist das Verwenden von Jini-Transaktionen in der Praxis aus drei Gründen recht einfach. Zum ersten werden Sie Transaktionen vermutlich nur selten benötigen. Von den bestehenden Jini-Diensten macht nur der JavaSpaves-Sicherungsdienst ausgiebigen Gebrauch

von Transaktionen. Falls Sie niemals auf Transaktionen angewiesen sind, ist der Umgang mit ihnen äusserst einfach!
Zum zweiten brauchen Sie für das Arbeiten mit Transaktionen nicht alle Einzelheiten der Mechanismen von Two-Phase Commit zu kennen, um sie als *Client* zu nutzen. Das *Anbieten* einer Operation, der es möglich sein soll, an einer Transaktion teilzunehmen, ist hingegen aufwendiger. Die Verwendung der APIs zur Durchführung von Transaktionen durch den Programmierer ist äusserst einfach: Sie brauchen nur eine Transaktion erstellen, diese an alle zu gruppierenden Operationen übergeben und die Transaktion anschliessend auffordern, mit der verarbeitung des Two-Phase Commits zu beginnen. Diese wird dann entweder erfolgreich verlaufen oder fehlschlagen.

Obgleich Jini nicht vorschreibt, wie Dienste die unterschiedlichen Phasen des Two-Phase Commit Protokolls implementieren sollen, brauchen Sie sich um die Konsistenz der Daten als reiner Client von Transaktionen keine Sorgen zu machen. Jini legt die sogenannte *Standardsemantik* für Transaktionen fest - dies ist die standardmässige Auffassung von der Verarbeitung von Transaktionen. Glücklicherweise richten sich die meisten Anwendungen danach, weil Jini eine solche Vorgehensweise unterstützt.

Die Standardsemantik wird von einer Gruppe von Klassen implementiert, welche die mit den zuvor erörterten ACID Eigenschaften übereinstimmende "normale" Transaktionssemantik zur Verfügung stellen. Beim Verwenden dieser Transaktionsklassen können Sie davon ausgehen, dass die gwünschten Eigenschaften wie Einheitlichkeit und Dauerhaftigkeit gewährleistet sind.

Wenn Sie Transaktionen verwenden, um einige Operationen unterschiedlicher Dienste zu gruppieren, können Sie für das Erstellen eines `Transaction`- Objekts mit der Standardsemantik einfach die Jini- `TransactionFactory` benutzen. Jeder Dienst, der in der Lage ist, an Transaktionen teilzunehmen, akzeptiert für seine Operationen einen `Transaction`-Parameter. Sie können einfach das von `TransactionFactory` zurückgegebene Standardobjekt übergeben, um diese Operationen der Standardsemantik für Transaktionen entsprechend zu gruppieren. Nachdem Sie die einzelnen Operationen dieser Dienste mit einem `Transaction`-Parameter aufgerufen haben, können Sie entweder die Methode `commit`() oder die Methode `abort`() des `Transaction`-Objekts aufrufen, damit die Operationen zur Verarbeitung bzw. zum Abbruch aufgefordert werden. Durch das Aufrufen von `commit`() wird der Two-Phase Commit Vorgang für alle Teilnehmer eingeleitet. Sobald alle Operationen erfolgreich abgeschlossen wurden, endet `commit`(). Falls die Transaktion nicht korrekt durchgeführt werden kann, werden alle Operationen rückgängig gemacht, und `commit`() signalisiert eine Exception namens `CannotCommitException`.

Ausserdem umfasst Jini einige Klassen, die das Implementieren der Schnittstelle TransactionParticipant in Dienste und das Gruppieren von Operationen zu Transaktionen erleichtern.

Da Transaktionen vermutlich zu jenen Aspekten von Jini gehören, mit denen Sie am wenigsten zu tun haben werden, erfolgt die eingehende Erörterung später noch genauer.

1.9.6. Blick zurück

Wir haben jetzt die fünf grundlegenden Konzepte von Jini vorgestellt:
1. Discovery

2. Lookup
3. Leasing
4. Remote-Ereignisse
5. Transaktionen

Wir haben uns mit der Bedeutung dieser Begriffe auseinandergesetzt, bislang jedoch kaum erfahren, wie in der Praxis mit ihnen programmiert werden kann.

Für das Entwickeln von Jini-Clients - Nutzer und Jini-Dienste - benötigen Sie ein grundsätzliches Verständnis von Discovery und Lookup, je nach Zielsetzung möglicherweise auch von Ereignissen und Leasing sowie in geringerem Umfang von Transaktionen.

Beim Erstellen von Jini-Diensten benötigen Sie ein grundsätzliches Verständnis der von Jini Diensten zu erfüllenden Eigenschaften, damit diese in Jini-Gemeinschaften einwandfrei integriert werden können. Dazu müssen Sie genau wissen, wie Discovery und Lookup verwendet werden, und möglicherweise ist es für Sie erforderlich, ein Ereigniserzeuger zu sein, falls Sie Clients asynchron über Änderungen Ihres Zustands benachrichtigen möchten. Das Leasing muss sorgfältig vorbereitet worden sein, und unter Umständen müssen Sie Ihren Operationen ermöglichen, an Transaktionen teilzunehmen. Glücklicherweise stellt Jini Ihnen einige nützliche Klassen zur Verfügung, die Ihnen Ihr Dasein als Dienst und auch das Überleben diverser Verantwortlichkeiten im bereich der Protokollierung erleichtern.

Jetzt werden wir uns den fünf grundlegenden Konzepten aus Sicht des Programmierers zuwenden.

Bevor wir uns jedoch mit der eigentlichen Entwicklung beschäftigen, sollten wir uns mit der praktischen Verwendung von Jini in einem Netzwerk auseinandersetzen. Nachdem wir nun einen Überblick über Jini gewonnen haben, können wir uns mit den Möglichkeiten des Implementierens von Jini in kleinen Geräten, Software Diensten und Kombinationen aus Hardware und Software beschäftigen.

1.10. Jini im Einsatz

Damit die Beispiele funktionieren, muss die Software korrekt installiert sein und vorallem die Codebase und Security (Policy Dateien) korrekt definiert sein. Wann sollte man Jini überhaupt einsetzen? Dazu müssen wir nochmals kurz auf das "Wesen" von Jini eingehen.

1.10.1. Was ist eigentlich ein Jini-Dienst?

Nachdem wir wissen, was Jini ist, können wir uns der Frage zuwenden, was es bedeutet, Jini zu "sprechen" - was sind die minimalen Anforderungen an Geräte oder Programme, die mt der Jini-Infrastruktur zusammenarbeiten sollen?

Die minimalen Anforderungen sind äusserst gering. Betrachten wir einmal, wie sich ein Gerät oder Dienst über Jini zur Verfügung stellen kann. Als Gerät oder Dienst müssen Sie bzw. ein anderes Programm in Ihrem Auftrag folgendes gewährleisten:

- Sie müssen in der Lage sein, eine Verbindung mit einem TCP / IP Netzwerk herzustellen. Ihr Gerät oder Dienst - bzw. das stellvertretend verwendete Programm - sollte über eine adresse und einen vollständigen TCP-Protokollstapel mit der Fähigkeit des Sendens und Empfangens von Multicast-Nachrichten verfügen.

- Sie müssen den Discovery-Vorgang durchführen, um mindestens einen Lookup-Dienst auffinden zu können. Nahezu alle Dienste verwenden Multicast-Discovery, und gut konzipierte Dienste können mit Hilfe des Unicast-Discovery-Protokolls in einer festgelegten Gruppe von Lookup-Diensten registriert werden.

- Sie müssen sich bei einem Lookup-Dienst registrieren und diesem ein Proxy-Objekt übergeben haben, welches Clients herunterladen kann, um den Dienst zu verwenden. Dieses Proxy-Objekt kann mit einem nachgeschaltenen Prozess oder Gerät zusammenarbeiten oder den gesamten Dienst selbst implementieren.

- ausserdem müssen Sie dafür sorgen, dass die Leases seiner Lookup-Registrierungen verlängert werden, solange der Dienst verfügbar ist.

Das war's ! Im Prinzip brauchen Sie lediglich sicherzustellen, dass ein Proxy-Objekt in einem Lookup-Dienst veröffentlicht wird, damit andere den von Ihnen angebotene Dienst nutzen können. Die soeben dargestellten Aufgaben können von der Ihren Dienst implementierenden Einheit oder stellvertretend von einem andern Programm übernommen werden.

Diese Aufforderungen müssen *grundsätzlich* erfüllt werden, um an Jini teilnehmen zu können. Selbstverständlich kann das Verhalten von Diensten durch zusätzliche Aufgaben erweitert und verbessert werden. Dienste können beispielsweise die Gruppe der während ihrer Lebensdauer aufgefundenen und wieder beendeten Lookup-Dienste oder Registrierungen von Leases oder für Benachrichtigungen über Ereignisse verwalten. Das wirklich Herausragende an Jini ist jedoch das Veröffentlichen von Proxies. Alle sonstigen Funktionen haben lediglich unterstützenden Charakter. Zweifellos ist Ihnen eine schon mehrfach erwähnte Tatsache nicht entgangen : innerhalb einer Jini-Gemeinschaft ist es möglich, statt eines Geräts oder Dienstes ein stellvertretendes Programm einzusetzen. Eine solche Verwendung Dritter ermöglicht das Steuern von Geräten wie Lichtschalter und Thermometer, die über keine eigene JVM verfügen, sowie von bestehenden Software-Diensten mit Jini.

Als nächstes beschäftigen wir uns mit den Anforderungen an Verwendern von Jini-Diensten.

- der Verwender eines Dienstes, der selbst ein Dienst sein kann, aber nicht sein muss, muss in der Lage sein, einen oder mehrere Lookup-Dienste mit Hilfe des Discovery-Vorgangs zu finden. Nahezu alle Verwender benutzen dafür den Multicast-Discovery-Prozess. In einigen Fällen könnte ein verwender über einen URL für einen *bestimmten* Lookup-Dienst verfügen und ihn mit Hilfe des Unicast-Discovery-Protokolls direkt auswählen.

- anschliessend muss der Verwender das Proxy-Objekt des zu verwendenden Dienstes erlangen. Dabei kann es sich um einen "reinen" RMI-Stub, einen "intelligenten" Proxy oder um ein einfaches serialisiertes Objekt handeln. Der Verwender kann einen geeigneten Dienst auffinden, indem er die Attribute der beim Lookup-Dienst registrierten Dienste überprüft und nach der benötigten Schnittstelle sucht, oder den Dienst mit Hilfe der entsprechenden Dienst-ID - falls diese bekannt ist.

- falls kein den jeweiligen Anforderungen entsprechender Dienst aufgefunden werden konnte, hat der Verbraucher die Möglichkeit, sich von den ihm bekannten Lookup-Diensten benachrichtigen zu lassen, sobald ein den angegebenen Kriterien entsprechender Dienst verfügbar wird.

Auch in diesem Fall ist alles im Grunde ganz einfach. Sobald der Dienst-Proxy aufgefunden ist, wird er automatisch zu dem Client heruntergeladen. Dieser verwendet ihn anschliessend wie eine zur Kompilierzeit aufgefundene Klasse.

Besonders kleine Jini-Dienste und verwender von Diensten brauchen nichts über Remote-Ereignisse oder Transaktionen zu wissen und benötigen lediglich eine minimale und einfach automatisierbare Unterstützung des Leasings. Sie müssen jedoch die Discovery-Protokolle verwenden und mit den Lookup-Diensten kommunizieren können.

1.11. Jini für eigene Geräte und Dienste verwenden

Als nächstes werden wir uns damit beschäftigen, wie Sie Ihre Geräte und vernetzten Dienste Jini-fähig machen können. Ich erwähnte bereits, dass Jini darauf ausgelegt ist, diverse Hardware-Plattformen zu unterstützen und zur Vernetzung von Hardware unterschiedlichster Art verwendet zu werden. Die Preise der vernetzten Geräte können durchaus von einigen Zehntel Euros bis hin zu mehreren Millionen Euro variieren.

Wenn Sie sich dazu entscheiden, Ihr Produkt für Jini einzurichten, ist Verständnis der Ihnen zur Verfügung stehenden Möglichkeiten entscheidend, da diese mit sehr unterschiedlichen Vor- und Nachteilen behaftet sein können. Im allgemeinen sind drei Arten des Einsatzes von Jini. In vielen Fällen ergibt sich die jeweils verwendete Möglichkeit aus der Art des anzubietenden Geräts bzw. Dienstes. Es handelt sich um die folgenden Möglichkeiten:

- das Verwenden von Jini auf einem Universalrechner.
  Mit "Universal" meine ich, dass der Rechner über eine eigene Netzwerkanbindung und ausreichend Rechnerkapazität für die Ausführung einer vollständigen Java Virtual Maschine verfügt. Ausserdem ist ihre Netzwerkanbindung zumeist langfrsitig aktiv.

- das Verwenden von Jini für Geräte mit einer eigenen eingebauten JVM. Diese Geräte unterscheiden sich von Universalrechnern durch ihre geringere Rechnerleistung, sie sind

zumeist weitaus billiger, sie sind nur gelegentlich mit dem Netzwerk verbunden und ihre Anbindung an das Netzwerk erfolgt in der Regel mit erheblich geringerer Bandbreite. Ausserdem unterstützen diese Geräte teilweise lediglich spezialisierte Varianten von Java, etwa Embedded Java oder Personal Java (oder JavaCard VM).

- das Verwenden von Jini zur Steuerung eines Geräts ohne JVM. Dabei wird ein gemeinsam verwendeter Jini-fähiger Computer für das Steuern einer Gruppe von Geräten mit nur geringfügiger oder auch gar keiner eigenen Rechnerleistung benutzt, die jedoch zumindest über einfache E/A Möglichkeiten verfügen müssen. Eine solche Nutzung ist beispielsweise für das Bedienen von Schaltern in Haushaltgeräten ideal. Die einzelnen Schalter können zu diesem Zweck mit Ein-Bit-E/A-Anschlüssen versehen (in der Praxis ist dies jeweils ein einfaches an ihnen angeschlossenes Kabel) und mit einem zentralen Jini-fähigen Computer verbunden werden, der die Kommunikation mit den Schaltern übernimmt und sie als Jini-Dienst verfügbar macht. Beachten Sie, dass der zentrale Computer in diesem Fall nicht besonders leistungsstark sein muss. Ein kleiner Thermostat an einer Wand wäre bereits ausreichend, es könnte jedoch auch ein normaler PC sein.

Im folgenden werden wir diese Szenarien etwas detaillierter besprechen.

1.11.1. Jini auf allgemeinen Systemen

In den Medien wird in Zusammenhang mit Jini zumeist vornehmlich die Eignung dieses Systems für das Verbinden kleiner Geräte hervorgehoben, doch auch für die Verwendung auf Universalrechnern ist Jini hervorragend geeignet. Es gibt unzählige Dienste, die auf solchen Rechnern ausgeführt und mit Hilfe von Jini andern Mitgliedern einer Jini-Gemeinschaft zur Verfügung gestellt werden können. Beispielsweise verfügt jeder PC über eine Festplatte, deren Speicherplatz problemlos andern Geräten zur Verfügung gestellt werden kann, etwa einer Digitalkamera, für das Speichern von Bildern, einem Handy, zum Sichern des integrierten Telefonbuches, dem Anrufbeantworter, um Nachrichten langfristig zu speichern oder auch dem Videorecorder, um Videos in digitaler Form auf der Festplatte zu speichern. Dieser Speicherdienst würde direkt vom PC angeboten werden und allen an der entsprechenden Jini-Gemeinschaft teilnehmenden Geräte das verwenden seiner Festplatte ermöglichen.

In grösserem Masstab kann Jini auf grossen Servern ausgeführt werden, um ein bequemes und robustes verwalten der Datendienste des Unternehmens zu ermöglichen. Unternehmensnetzwerke können mit Jini äusserst zuverlässig und fehlertolerant konzipiert werden. Ausserdem trägt Jini zur Verringerung des verwaltungsaufwandes bei. Stellen Sie sich in diesem Zusammenhang einen Jini-Dienst für das Zusammenstellen von Statistiken über die am Unternehmensnetzwerk beteiligten Geräte vor. Da er zur Gewährleistung der Redundanz über beliebig viele Lookup-Dienste zur Verfügung gestellt werden kann, ist er äusserst zuverlässig und erfüllt somit eine grundlegende Anforderung an die Systemverwaltung, weil gerade von Werkzeugen für die Netzwerkdiagnose erwartet wird, dass sie auch in Problemsituationen stets verfügbar sind. Dieser Dienst sollte eine Benutzerschnittstelle anbieten, mit deren Hilfe die Statistiken sämtlicher Jini-fähigen Geräte angezeigt werden können. Ausserdem könnte er auf einfache Weise mit andern Diensten zusammenarbeiten, um beispielsweise einen Kommunikationsdienst zu veranlassen, e-Mails oder gedruckte Nachrichten zu erstellen, sobald eine Festplatte voll oder ein Rechner abgestürzt ist.

Universalrechner dieser Art sind als Host für Dienste prädestiniert, die jeweils langfristig mit dem Netzwerk verbunden bleiben. Dies betrifft auch die "grundlegenden" Jini-Dienste wie

beispielsweise Lookup. Sie können besonders kostengünstig betrieben werden, da mehrere Dienste auf einem einzelnen Computer ausgeführt werden können. Ausserdem sind sie auch als Entwicklungsform sehr gut geeignet, da die Dienste auf den Rechnern entwickelt werden, von denen aus sie später in Jini-Gemeinschaften eingebunden werden.

Ferner kann der Vorteil einer vollwertigen JVM kaum überbewertet werden. Auf solchen Plattformen ausgeführte Dienste können alle von RMI gebotenen Möglichkeiten für die Kommunikation mit ihren Dienst-Proxies ausschöpfen, sobald diese zu den Clients heruntergeladen wurden. Zudem könnte RMI verwenet werden, um neuen Code zum Server *heraufzuladen* und auf diese Weise das fliegende Verwalten und Aktualisieren der auf ihm ablaufenden Dienst ermöglichen. Dienste dieser Art können auch mit Hilfe von Java Database Connectivity Software (JDBC) mit Unternehmensnetzwerken verbunden werden. Eine vollständige JVM mit den gesamten Java-Klassenbibliotheken gewährleistet ausserdem, dass praktisch sämtlicher heruntergeladener Code ausgeführt werden kann - dies kann bei der Verwendung einer Java-Variante wie Embedded Java oder Personal Java unter Umständen nicht der Fall sein.

Der Nachteil ist sicher, dass eben ein Universalrechner benötigt wird. Häufig ist dieser jedoch ohnehin verfügbar und unterliegt durch Jini nur einer geringfügigen Belastung.

Spielarten von Java

Es gibt zwei wichtige Varianten von Java, die von der Industrie angenommen wurden. Sie werden als Personal Java und Embedded Java bezeichnet und sind abgespeckte Varianten des vollständigen Java. Zwar verarbeiten beide den gleichen Bytecode wie das vollständige Java (im Gegensatz zum JavaCard Java und zur *Connected, Limited Device Configuration Specificatio*n, CLDC). Die Java Semantik stimmt vollständig überein. Es steht jedoch lediglich eine erheblich eingeschränkte Anzahl Klassenbibliotheken zur Verfügung.

Personal Java ist in erster Linie für programmierbare Geräte mit eigenen Schnittstellen gedacht - dies sind beispielsweise Handys und PDAs. Solche Geräte sind normalerweise in der Lage, Applet-Code herunterzuladen und umfassen für Benutzerschnittstellen zumindest das AWT sowie gelegentlich die Java Foundation Classes. Bei Personal Java wurden in erster Linie die unternehmensorientierten APIs weggelassen: Datenbankanbindung, CORBA-Kommunikation und (manchmal) RMI. Zum Zeitpunkt basiert PJ auf der Version Java 2.

Personal Java wird vermutlich im Jini Umfeld die am meisten eingesetzte Version sein. Glücklicherweise unterscheidet sie sich nur geringfügig vom vollwertigen Java. Ausserdem werden Personal Java Geräte, die mit Jini arbeiten sollen, nahezu ausnahmslos RMI unterstützen.

Eingebettete Java-Geräte unterliegen jedoch grösseren Einschränkungen. Glücklicherweise sind embedded Java Geräte in Jini-Umgebungen eher die Ausnahme, heute! Bei embedded Java wird ein grösserer Teil der Klassen für Benutzerschnittstellen und für den Umgang mit dem Netzwerk lediglich optional angeboten. Geräte, die auf embedded Java basieren, können im Jini Umfeld beispielsweise mit Hilfe von Proxies grafische Schnittstellen haben.

1.11.2. Jini auf Java-fähigen Geräten

Da weltweit immer mehr Geräte Java-fähig werden, wird das Verwenden kleiner Geräte mit eigenen eingebetteten JVMs zumindest üblicher. Immer mehr Geräte, wie Handys, PDAs,

Drucker, Scanner und Kameras werden bereits mit eigenen JVMs ausgeliefert. In mancher Hinsicht unterscheiden sich diese Szenarios kaum vom vorigen - in beiden Fällen können die Geräte an Netzwerke angeschlossen werden und sind zumindest in der Lage, zusätzlich zu den in ihnen bereits enthaltenen Java-Programmen auch aus dem Netzwerk heruntergeladenen Java Bytecode auszufüllen.

Trotzdem gibt es einige wesentliche Unterschiede zwischen der Ausführung von Jini auf kleinen Geräten und jener auf Universalmaschinen:

- auf kleinen Geräten kann lediglich eine spezialisierte Variante der vollständigen Java-Plattform ausgeführt werden.
- kleine Geräte können lediglich mittelbar mit dem Netzwerk verbunden werden
- kleine Geräte verfügen im allgemeinen über wesentlich weniger Rechenleistung und Netzwerkbandbreite als ihre universellen Kollegen.

Entscheidend ist der erste der soeben genannten Punkte. Es würde Ihnen wohl kaum in den Sinn kommen, den Jini-Lookup-Dienst auf einem Gerät ausführen zu lassen, das häufiger vom Netzwerk getrennt als daran angebunden ist oder lediglich über eine äusserst begrenzte Rechnerleistung verfügt. Ausserdem werden Sie den einzigen Lookuo-Dienst Ihres Netzwerks nicht einem Toaster oder einem Mikrowellengerät überlassen wollen.

1.11.2.1. Untermengen von Jini und Java

Der Umgang mit unterschiedlichen Versionen von Java stellt für kleine Geräte ein bedeutendes Problem dar. Solche zumeist billigen Geräte verfügen häufig lediglich über eine kleine Untermenge der vollständigen Java-Plattform und unterstützen das Darstellen von Benutzeroberflächen möglicherweise nur in äusserst beschränktem Umfang. Einige dieser Geräte sind zur Anzeige von Benutzerschnittstellen sogar häufig sogar *überhaupt nicht* in der Lage.

Sie brauchen nur ein wenig darüber nachzudenken, wie Jini funktioniert, um zu erkennen, warum dies en Problem darstellt. Eine grundlegende Eigenschaft von Jini-Diensten ist die Fähigkeit des Einbettens eines Teils ihrer selbst (der Dienst-Proxies) in Clients. Während des Entwickelns eines Dienst-Proxies geht der Entwickler stets von der Verfügbarkeit einiger Klassenbibliotheken auf der Client Platform aus. Wenn der Dienst-Proxy beispielsweise JFC verwendet,setzt er voraus, dass JFC auf allen Clients, die den Proxy benutzen, verfügbar ist. In Umgebungen, in denen die Verwender des Dienstes mit beschränkten Varianten von Java betrieben werden, ist dies jedoch nicht immer der Fall.

Die bereits erwähnten primären Untermengen von Java - Personal Java und Embedded Java - erbeiten jedoch mit eingeschränkteren Gruppen von Klassenbibliotheken als dies beim vollwertigen Java der Fall ist. Daher können bei der Ausführung von Dienst-Proxies in solchen Umgebungen Probleme auftreten.

1.11.2.1.1. Noch beschränktere Java-Umgebungen

Beachten Sie, dass die im vorigen Abschnitt erörterten Probleme stets durch die Verwendung von Java-Geräten mit eingeschränkten Funktionalitäten als *Verwender* von Jini-Diensten bedingt sind. Möglicherweise gehen Sie davon aus, dass ein beschränktes Java Gerät, welches lediglich einen Dienst anbietet, von diesen Schwierigkeiten nicht betroffen ist. Doch bedenken Sie, dass auch Anbieter von Diensten setst als Verwender des Lookup Dienstes

fungieren - der Lookup-Dienst lässt seinen eigenen Dienst-Proxy zu den Dienstanbietern herunter. Daher muss ein embedded Java bzw. Personal Java Gerät zumindest in der Lage sein, den Dienst-Proxy des Lookup-Dienstes herunterzuladen und auszuführen.

Es gibt jedoch noch eine andere Möglichkeit. Ein Gerät könnte mit einer *benutzerdefinierten* Untermenge der JVM arbeiten, die mit Jini zusammenarbeiten kann, ohne Code herunterladen zu können.

Ein solches Gerät würde bereits von sich aus mit den Discovery Protokollen ausgestattet sein und bereits über den für das Kommunizieren mit bestimmten Implementierungen des Jini-Lookup-Dienstes erforderlichen Code verfügen. Seine Möglichkeiten wären äusserst beschränkt - er könnte ausschliesslich innerhalb der von ihm erwarteten Umgebung verwendet werden. Nach einer veränderung der Implementierung des Lookup-Dienstes, beispielsweise weil mittlerweile ein neuerer und schnellerer Dienst angeboten wird, der über ein anderes Protokoll mit den Lookup-Nutzern kommuniziert, würde das Gerät dann nicht mehr funktionieren.

Diesem Dilemma könnte jedoch durch eine eigene spezielle Implementierung des Lookup-Dienstes begegnet werden, der lediglich den Proxy des jeweiligen Geräts verfügbar macht. Auf diese Weise könnte die erforderliche Kommunikation des Geräts mit dem Rest der Jini-Gemeinschaft beschränkt werden - es müsste lediglich in der Lage sein, einen Proxy für seinen eigenen Lookup-Dienst zur Verfügung zu stellen, damit Clients den Proxy dieses Geräts vom Lookup-Dienst erhalten können.

Der Vorteil dieser Lösung besteht darin, dass sie lediglich eine einfache strukturierte JVM mit äusserst eingeschränkten Klassenbibliotheken erfordert. Da kein Code heruntergeladen werden könnte, wäre weder ein Bytecode-Verifizierer noch ein Sicherheitsmanager notwendig, und die auf den Geräten installierten Klassenbibliotheken könnten auf das absolute Minimum reduziert werden, welches dem Gerät lediglich das Verwenden von Discovery und Lookup erlaubt.

Ein solches Gerät könnte ausschliesslich Jini-Dienste anbieten, da für das Verwenden eines Jini-Dienstes die Fähigkeit des Herunterladens seines Proxys erforderlich ist. Die damit verbundenen Einschränkungen sind erheblich, können jedoch in Einzelfällen akzeptabel sein.

1.11.2.2. Versionen

Die Schwierigkeiten mit embedded und Personal Java entspringen dem allgemeinen Problem, dafür zu sorgen, dass die Version der Host-Plattform mit der vom Dienst-Proxy erwarteten übereinstimmt. Die Lösung besteht darin, sich auf eine Reihe von Konventionen zu einigen - die Dienst-Proxies werden mit Attributen versehen, welche auf die für ihre Ausführung mindestens erforderliche Java-Funktionalität hinweisen. Ein solches Attribut würde Informationen über die benötigten Klassenbibliotheken, JVM Versionen usw. enthalten. Durch das Überprüfen der Attribute eines Dienst-proxy könnte ein Verwender dann vor dessen Benutzung feststellen, ob er überhaupt ausgeführt werden kann.

Mit Hilfe geeigneter Massnahmen kann man dafür sorgen, dass die von Ihnen erstellte Dienste auf möglichst vielen Plattformen verwendet werden können. Am wichtigsten ist dafür das Beschränken der Abhängigkeiten Ihres Proxy-Objekts auf der Host-Plattform. In der Regel impliziert dies den Verzicht auf Abhängigkeiten von Bibliotheken, die möglicherweise nicht überall präsent sind. Falls Ihr Dienst in Teile mit unterschiedlicher Funktionalität zerlegt werden kann - von denen einige lediglich von den grundlegenden und andere von weniger

verbreiteten Bibliotheken abhängig sind - kann die zusätzliche Funktionalität dem Dienst häufig in Form von Attributen hinzugefügt werden. Indem Sie im Proxy auf bedenklichen Code verzichten, gwährleisten Sie, dass Clients das Proxy-Objekt grundsätzlich herunterladen und ausführen können. Sonstiger Code - etwa Benutzerschnittstellen - kann separat bereitgestellt und Clients können bei Bedarf auf Anfrage übermittelt werden.

1.11.2.3. Verwendung von Geräte-Proxies durch Jini

Dieser dritte Ansatz wird wahrscheinlich von sehr vielen Geräten gewählt. Dies betrifft beispielsweise Drucker, Scanner sowie andere mit einem Host Computer verbundene bzw. besonders einfache und billige Geräte, wie etwa Schalter im Haushalt.

Bei diesem Ansatz ist ein Gerät, auf dem eine JVM ausgeführt werden kann, physisch mit einem oder mehreren Geräten verbunden. Es fungiert stellvertretend für die mit ihm verbundenen Geräte als Jini-*Proxy*. Dazu benutzt es die Lookup- und Discovery-Protokolle, geht mit ihren Dienst-Proxies um und verwaltet ihre Leases. In diesem Fall handelt es sich bei der Verbindung zwischen dem die JVM ausführenden und den angeschlossenen Geräten normalerweise nicht um ein Netzwerkkabel, sondern eher um eine RS-232, USB-, Firewire- oder auch eine X10 (Heimautomastisierung) Verbindung (X10 =1-Wire = Strom und Kommunikation in einem Kabel).

Für das Gerät, auf dem die JVM ausgeführt wird, kann entweder das vollwertige Java (siehe Universalcomputer) oder eine Untermenge von Java verwendet werden. Bei einem solchen Gerät kann es sich *tatsächlich* um einen Universalrechner handeln, beispielsweise um einen ausrangierten, im Keller aufbewarten PC, es könnte jedoch ebensogut ein auf die Verwendung als Jini-Steuerbox spezialisiertes Gerät benutzt werden. Stellen Sie sich beispielsweise einen Jini-Controller vor, der an Ihre Stereoanlage angeschlossen wir, um CD- und DVD-Player, Tuner, Videorecorder und Fernsehapparate Ihrer privaten Jini-Gemeinschaft zur Verfügung zu stellen. Ein noch kleineres Gerät, das ungefähr wie ein Thermostat mit Wandhalterung aussehen könnte, würde stellvertretend für das Alarmsystem Ihres Hauses oder Ihrer Wohnung oder Ihres Autos, für Lichtschalter, für Schliessvorrichtungen von Türen usw. Dienste anbieten. Unabhängig von der Form des Controllers wären die an ihm angeschlossenen Geräte innerhalb der Jini-Gemeinschaft sofort vollständig verfügbar.

Eine auf PCs ausgeführte Proxy-Anwendung ist jeweils nur kurzfristig aktiv. Eine solche Anwendung startet während des Hochfahrens des Rechners und überprüft die Plug & Play Datenbank von Windows, um festzustellen, ob neue Hardware hinzugefügt wurde. Falls ein neu hinzugefügtes Gerät wie beispielsweise ein Drucker oder ein Scanner erkannt wird, veröffentlicht die Proxy-Anwendung entsprechende Dienste, um diese neuen Geräte innerhalb des Netzwerks verfügbar zu machen.

Dieser Ansatz weist deutliche Vorteile auf. Zum ersten ermöglicht er "ältere" Hardware das Teilnehmers an Jini. Weltweit gibt es zahlreiche Drucker, Scanner und andere Geräte, die bislang nicht Java-fähig sind. Durch das Installieren entsprechender Software auf einen Host-Computer können solche Geräte der Jini-Welt zugänglich gemacht werden. Es wäre sogar eine Jini-"Umwandlungsbox" denkbar, die zwischen einem Host und einem Drucker geschaltet wird, um den Drucker Jini-fähig zu machen.

Zum zweiten ermöglicht der Proxy-Ansatz das Einfügen äusserst preisgünstiger Geräte in Jini-Gemeinschaften. Wenn wir ausschliesslich Lichtschalter mit eingebetten JVMs und 8 Mbyte Speicher verwenden könnten, würden diese vermutlich nur von wenigen gekauft werden. Wenn hingegen zahlreiche preisgünstige Geräte einen einzelnen Jini-Host

gemeinsam verwenden können und nicht von Geräten unterscheidbar sind, die mit vollwertigen JVMs arbeiten, werden solche preisgünstige Jini-Geräte möglicherweise zunehmend in Privathaushalten verwendet.

1.11.3. Anforderungen an Jini-Infrastrukturdienste

In den vorherigen Abschnitten wurden die unterschiedlichen Anforderungen an die Plattformen der Anbieter und Verwender von Jini-Diensten erörtert. Doch wie sieht es mit den Basiskomponenten der Jini-Infrastruktur selbst aus? Bedenken Sie, dass wesentliche Teile der Jini-Infrastruktur selbst aus Diensten besteht, die ebenfalls gewisse Anforderungen an die Umgebung stellen, in der sie ausgeführt werden sollen.

Für seine Dienste legt Jini Schnittstellen, jedoch keine Implementierungen fest. Eine bestimmte Implementierung eines Jini-Dienstes kann daher höheren oder auch weniger hohen Anforderungen unterliegen als eine alternative Implementierung. Beispielsweise könnte ein Lookup Dienst für eine vollwertige kommerzielle Datenbank JDBC Unterstützung erfordern, und ein anderer könnte mit einer internen Zuordnungstabelle arbeiten. Wir können an dieser Stelle zwar nicht klären, wie die Anforderungen an spezielle Implementierungen dieser Hauptdienste aussehen könnten, doch wir können uns mit deren *minimal* benötigter Funktionalität beschäftigen - dies betrifft die von den Schnittstellen dieser Komponenten vorausgesetzten Funktionen.

Jini deklariert zwei Dienste als besonders wichtig - den Lookup-Dienst und den Transaktionsmanager - der Lookup-Dienst steht jedoch eindeutig im Vordergrund. Ungeachtet irgendwelcher befremdlichen Anforderungen an die Implementierung eines speziellen Transaktionsmanagers können Sie diesen Dienst überall ausführen und brauchen dabei lediglich die bereits erörterten Richtlinien für "normale" Dienste zu beachten.

Die referenzimplementation des Lookup-Dienstes von Sun ist eigentlich für das Ausführen auf einem Universalrechner mit vollwertigem Java ausgelegt. Ein etwas beschränketerer Lookup-Dienst kann jedoch durchaus auf weitaus weniger leistungsfähigen Plattformen ausgeführt werden. Möglicherweise wollen Sie Ihre Lookup-Dienste auf einem Server ausführen, um die Dauer ihrer Inaktivität zu minimieren, doch es ist sogar möglich, einen minimalen Lookup-Dienst zu entwickeln, der in keiner Weise auf Java basiert.

Die wichtigste Anforderungen an die Host-Plattform für einen minimalen Lookup-Dienst ist die Unterstützung von IP Multicast Netzwerkverkehr über Sockets. Die Discovery Protokolle verwenden dieses Protokoll für das Auffinden von Lookup- Servern, und von Lookup-Servern wird die Fähigkeit des beantwortens entsprechender Nachrichten erwartet. Ein Lookup-Server antwortet auf solche Nachrichten mit einem serialisierten Java.Objekt, bei dem es sich um den Dienst-Proxy handelt, den Clients für die Verständigung mit dem Lookup-Dienst verwenden. Dieses Objekt kann von einem Rechner, auf dem Java-Code ausgeführt werden kann, "vorserialisiert" worden sein und somit braucht der Lookup-Dienst selbst nicht in der Lage zu sein, Java-Code auszuführen. Während seiner Verwendung durch einen Lookup-Client kann dieses vorserialisierte Objekt mit dem Lookup-Dienst auf beliebige Weise über ein benutzerdefiniertes Protokoll kommunizieren.

Analog zum "Bootstrapping" Dienst für Jini kann der Lookup-Dienst lediglich auf Plattformen ausgeführt werden, die etwas aufwendiger sind als ein Kasten mit Netzwerkanbindung.

1.11.4. Wofür sollte Jini verwendet werden?

Nachdem wir uns nun mit einigen unterschiedlichen Methoden des Einbindens Ihrer Geräte und Dienste in Jini-Gemeinschaften beschäftigt haben, können wir uns der Frage zuwenden, wofür Jini aus technologischer und ökonomischer Sicht verwendet werden sollte. Die gängisten Fälle sind im folgenden aufgelistet:

- wenn Sie Dienste anbieten wollen, um Ihren Geräten oder Software-Diensten deren Verwendung zu ermöglichen, oder bereits Jini-Dienste verfügbar sind, könnte Jini eine günstige Möglichkeit darstellen. Wenn Sie beispielweise Scanner verkaufen möchten, und auf kleine Jini-Dienste aufmerksam werden, welche das direkte Mailen oder Faxen von Bildern zulassen, können Sie diese problemlos nutzen, um die Funktionalität Ihres Scanners zu erhöhen. Indem Sie Ihren Scanner als Verwender von Jini-Diensten auftreten lassen, befähigen Sie ihn, eingescannte Vorlagen quasi direkt als E-Mail oder Fax zu versenden. Auf diese Weise machen Sie sich die von den Entwicklern dieser Dienste bereits geleistete Vorarbeit unmittelbar zunutze. Im Grunde fungieren diese Entwickler als *Ihre* Entwickler.

- falls Ihr Gerät vorhandene oder noch einzurichtende Jini-Dienste nutzen kann, ist Jini möglicherweise das Mittel der Wahl. Wenn Sie beispielsweise Drucker verkaufen und feststellen, dass zunehmend billige Jini-fähige Kameras vetriebenen werden, können Sie Ihre Produkte als einfache Lösung für das Ausdrucken digitaler Bilder anpreisen.

- falls Ihr Gerät bereits über eine eigene JVM verfügt oder normalerweise an Universalrechner angeschlossen wird, ist das Hinzufügen von Jini praktisch ein Kinderspiel. Wenn Java in oder in der Nähe von Ihrem Gerät bereits verfügbar ist, erfordert das Unterstützen von Jini nur noch einen minimalen Aufwand und der potentielle Nutzen ist enorm.

- wenn Ihr Software-Dienst auf einem Universalrechner betrieben wird, der eine JVM ausführen kann - und dies ist heututage praktisch ausnahmslos der Fall - haben Sie die Möglichkeit, ihn mit Hilfe eines als Stellvertreter fungierenden Jini-Dienstes für Jini-Gemeinschaften verfügbar zu machen.Dies ist einer der grossen Vorteile von Jini - es ist sehr einfach, ein kleines Programm zu entwickeln, welches einen kompletten Software-Dienst, ob dieser nun in Java geschrieben ist oder nicht, für andere Jini-Dienste und Geräte verfügbar macht.

- wenn Sie einen Software-Dienst, der sich auf einfache Weise in die ihn konstrituierenden Bestandteile zerlegen lässt, in eine Jini-Gemeinschaft einbinden wollen, könnte es sich als günstig erweisen, alle Komponenten einzeln als Jini-Dienste verfügbar zu machen. Ein System zur unternehmensweiten Verwaltung von Dokumenten könnte beispielsweise Komponenten umfassen, die jeweils für das Speichern, Bearbeiten bzw. Durchsuchen von Dokumenten verwendet werden oder ähnlichen Zwecken dienen. Das separate Anbieten einzelner Komponenten als Dienst birgt mehrere Vorteile. Zum ersten können Sie Komponenten bei Bedarf einfach auf andere Rechner verschieben, ohne deshalb die Konfiguration ändern zu müssen oder sonstigen Verwaltungsaufwand zu verursachen. Zum zweiten schaffen Sie sich die Möglichkeit, den Dienst redundant zur Verfügung zu stellen. Um sicherheitshalber eine zweite Instanz eines dieser Dienste zu betreiben, brauchen Sie ihn lediglich auf einen zweiten Rechner ausführen zu lassen. Ausserdem erreichen Sie durch die offenere Gestaltung Ihrer Architektur ganz nebenbei eine

hervorragende Erweiterbarkeit.

- wenn Sie ein Gerät entwickeln, das normalerweise mit anderen Geräten verbunden sind, könnte sich die Jini-Technologie für das Steuern der Verbindungen als nützlich erweisen. Die Komponenten einer Stereoanlage sind beispielsweise in der Regel durch diverse Kabel miteinander verbunden. Wäre es nicht eine feine Sache, wenn der Vorverstärker die angeschlossenen Geräte sowie deren Pegel und Eingänge kennen würde?

- ausserdem ist Jini für programmierbare Geräte häufig eine gute Lösung. Da Jini das Herunterladen von Code erlaubt, können solche Geräte innerhalb Ihrer Jini-Gemeinschaften automatisch zu steuern - stellen Sie sich beispielsweise ein MIDI-Steuerpult mit einer Benutzeroberfläche vor, die unmittelbar nach dem Anschliessen eines neuen Rythmusgeräts entsprechend aktualisiert wird.

Da Java für immer mehr Computer und andere Geräte verwendet wird und Geräte immer häufiger miteinander verbunden werden, bietet sich Jini für deren Vernetzung an. Ausserdem - und dies wird die zunehmende Verbreitung von Jini vermutlich am intensivsten fördern - ist es einfach, einen PC, auf dem Java bereits vorhanden ist, Jini hinzuzufügen.

1.11.5. Wofür ist Jini nicht geeignet?

Da Jini speziell entwickelt wurde, möglichst vielfältig verwendbar zu sein - dies reicht von Servern bis hinunter zu Lichtschaltern - gibt es nur wenige Anordnungen von Hardware, für die Jini nicht geeignet ist. Es gibt jedoch auch Argumente gegen das Ausstatten bestimmter Geräte mit der Jini Technologie

- falls Ihr Gerät *gänzlich* unabhängig von anderen Geräten verwendet wird - also nicht einmalein Kabel mit ihm verbunden ist - oder so ausgesprochen uninteressant ist, dass niemand es über das Netzwerk verwenden würde, dann ist die Nutzung von Jini möglicherweise nicht angebracht. Doch heutzutage werden gänzlich isolierte Geräte immer seltener - was würden Sie tun, wenn Sie ein so langweiliges Gerät zu verkaufen hätten?

- wenn Ihr Gerät nicht im Zusammenhang mit einer JVM betrieben wird und so billig sein soll, dass keine JVM darin eingebettet werden kann, ist Jini keine pratikable Lösung. Ganz ohne Java können Geräte nicht an Jini-Gemeinschaften teilnehmen.

1.11.6. Einige Hinweise auf Verbesserungsmöglichkeiten in Ihrem Wohnumfeld

Falls Sie ein begeisterter Anhänger hochwertiger Audio- und Videoausrüstungen für Heimanwender sind, dürfte die Website von HAVi (home audio/video interoperability) für Sie interessant sein. Havi ist ein Konsortium grosser A/V Anbieter, die sich für einige äusserst interessante Standards einsetzen. HAVi Geräte werden per Firewire miteinander verbunden. Bei Firewire handelt es sich um eine äusserst schnelle serielle Verbindung (IEEE 394 oderiLink von Sony). HAVi Geräte können nicht nur Steuerinformationen, sondern auch Audio- und Videodaten austauschen.

Sun arbeitet mit Sony an einer Verbindung Jini-Firewire.

http://www.havi.org

1.12. Mit Jini arbeiten

In diesem Abschnitt fangen wir an, einige konkrete Jini-Dienste sowie die Client-Anwendungen für ihre Nutzung zu erstellen. Das erste Programmpaar, das wir uns ansehen werden, sind ein "Hello World" Dienst und der dazugehörige Client. Das sind die kleinsten Jini-Programme - sie verwenden Discovery und suchen einen Lookup; der Dienst veröffentlicht einen Dienst-Proxy, und der Client holt und benutzt ihn. Der Proxy für diesen Dienst ist ziemlich einfach. Er gibt auf Anfragen die Zeichenfolge "Hello, World!" zurück. In diesem Fall *ist* der Proxy gleichzeitig der Dienst, da er in der Lage ist, die Aufgabe völlig selbständig zu erledigen.

Nach diesem ersten Programm beginnen wir mit weiteren Durchgängen, um immer mehr Jini Merkmale zu zeigen.

1.12.1. Jini-Dienste verwenden

Bevor Sie anfangen können, Ihr Programm zu schreiben, müssen Sie sicherstellen, dass die erforderlichen Jini-Dienste funktionieren. Sie finden ganz vorne Einzelheiten darüber, wie Sie Ihre Umgebung so einrichten, dass Sie Jini verwenden können. Für die Beispiele dieses und der folgenden Abschnitte brauchen Sie weder einen JavaSpaces-Speicherdienst noch den Transaktionsmanager zu starten. Die Wiederholungen von "Hello World!" erfordern lediglich, dass an irgendeiner Stelle im Netzwerk ein Lookup-Dienst, ein RMI-Aktivierungs-Daemon und ein HTTP-Daemon in Betrieb sind. Der RMI-Aktivierungs-Daemon (`rmid`) muss auf demselben Host-Rechner laufen wie der Lookup-Dienst, weil die Sun-Implementierung des Lookup-Dienstes Aktivierung verwendet. Ausserdem können wir einen eigenen aktivierbaren Dienst erstellen. Auch in diesem Fall muss `rmid` auf demselben Rechner wie der Dienst laufen.

Es folgt die Prüfliste für die Einreichtung:

- konfigurieren Sie den RMI-Aktivierungs-Daemon. Das sollte auf demselben Rechner geschehen, auf dem der Lookup-Dienst laufen soll, und zwar *vor* dem Lookup-Dienst. Für das Beispiel muss der Aktivierungs-Daemon ausserdem auf demselben Host laufen wie das Beispielprogramm.

- starten Sie den RMI-Aktivierungs-Daemon. Das sollte auf demselben Rechner geschehen, auf dem der Lookup-Dienst laufen soll, und zwar *vor* dem Lookup-Dienst.

- starten Sie den Jini-Dienst. Sie sollten diesen auf einem Rechner im selben Subnetz starten, auf dem Sie die Musterprogramme betreiben wollen - weil nämlich die Multicast-Discovery-Protokolle von Jini standardmässig so konfiguriert sind, dass sie Lookup-Dienste suchen, die "in der Nähe" des Objekts laufen, das Discovery betreibt.

- starten Sie einen Jini-HTTP-Server. Diese konkrete Instanz des HTTP-Servers liefert herunterladbaren Code für den Jini-Lookup-Dienst. Er kann auf demselben Rechner wie dieser laufen. Sie können sein "Stammverzeichnis" so einrichten, dass es auf das "lib"-Verzeichnis aus der Jini-Distribution zeigt. Damit stellen Sie sicher, dass Ihr Code die Klassen aus der Datei `reggie-dl.jar` herunterladen kann, welche den Code enthalten, den Ihre Programme zur Verwendung des Lookup-Dienstes benötigen. Ausserdem

müssen alle Beispielprogramme, die wir schreiben, in der Lage sein, Code an Jini-Dienste und aneinander zu liefern. Daher müssen Sie auch einen HTTP-Dienst starten, um den Code aus diesen Beispielen zu transportieren. Wie man das macht, sehen Sie sobald wir zum Beispielcode kommen.

Das war die Checkliste. Diese Konfiguration von Diensten sollte es Ihnen ermöglichen, alle folgenden Programme zu verwenden. Sie können jedoch noch einige weitere Schritte unternehmen, die Ihnen das Leben leichter machen, wenn Sie zum Testen und zum Einsatz von Jini-Diensten kommen; das werden wir uns im nächsten Abschnitt ansehen.

Ausserdem schauen wir uns die spezifischen Anweisungen für jedes Beispiel an, wenn wir dort angelangt sind. Beachten Sie, dass die Beispiele in diesem Kapitel aus einem Jini-Dienst - der Einheit, die einen Dienst-Proxy veröffentlicht - sowie einem Jini-Dienst besteht - im Grunde einfach eine Anwendung, die mit Hilfe von Discovery und Lookup einen Dienst-Proxy sucht, der an der Verwendung interessiert ist. Häufig verwenden Dienste andere Dienste, und daher ist ein Jini-Programm oft gleichzeitig Dienst und Client. Im Interesse der Einfachheit sind die Client-Programme in diesem Zusammenhang jedoch "eigenständige" Anwendungen.

Sie können diese Programme, Lookup-Dienste, HTTP-Server, Aktivierungs-Daemons, Clients und Server zwar alle auf demselben Rechner betreiben, ich rate jedoch dringend, den nächsten Abschnitt zu lesen. Wenn Sie Jini-Anwendungen auf dieselbe Art entwickeln, auf die Sie wahrscheinlich "eigenständige" Java-Anwendungen schreiben - alles auf einem Rechner mit demselben CLASSPATH - geraten Sie wahrscheinlich in Schwierigkeiten, wen Sie die Programme auf mehreren Rechnern einsetzen. Der nächste Abschnitt enthält einige Strategien für Feinabstimmung Ihrer Umgebung, um diese Probleme zu minimieren.

1.12.2. Verteilte Anwendungen erstellen

Natürlich laufen in der "realen Welt" praktisch alle Jini-Dienste und -Clients in Umgebungen mit mehreren Rechnern. Das heisst, dass Dienste über eine beliebige Anzahl von Rechnern in einem Netzwerk verstreut sein können. Der herunterladbare Code für diese Dienste kann von einer beliebigen Anzahl von HTTP-Servern zur Verfügung gestellt werden, und Clients können von jedem Rechner aus Verbindungen zu diesen Diensten aufnehmen.

Häufig müssen Sie jedoch - aus Gründen der Bequemlichkeit oder der Notwendigkeit - Ihren Code auf einem einzigen Rechner entwickeln und testen. Dann gerät man einfach deshalb sehr leicht in Schwierigkeiten, weil man, wenn alles auf einem Rechner läuft, Teile des Codes nicht ausprobiert, die zu Abstürzen führen können, wenn die Systemkomponenten auf unterschiedlichen Host-Rechnern laufen.

Sehen wir uns ein wenig näher an, was das heisst. Für die Entwicklung der meisten Jini-Programme haben Sie drei Jini-fähige Anwendungen in Betrieb: einen Lookup-Dienst, den Dienst, den Sie gerade testen, und einen Client, der diesen Dienst einsetzt (zusätzlich zu den übrigen erforderlichen Programmen, etwa HTTP-Servern oder RMI-Aktivierungs-Daemons). Wenn all diese programme auf demselben Rechner laufen und dieselben Ressourcen benutzen (denselben CLASSPATH, denselben HTTP-Server usw.) können potentielle Probleme beim dynamischen Laden von Klassen oder hinsichtlich der Sicherheit verschleiert werden. Das Entwickeln und Testen in derartigen Umgebungen kann dazu führen, dass sich Probleme unbemerkt in Ihrem Code verbergen.

Das ist die grosse Gefahr beim nur "lokalen" Testen. Die offensichtliche anfängliche Bequemlichkeit, den einfachen Weg zu wählen, kann zu grösseren Kopfschmerzen auf dem späteren Weg führen. Auch wenn eine Umgebung mit mehreren Rechnern am Anfang mehr Arbeit macht, z.B. mehrere HTTP-Server zu starten, Sicherheitsrichtlinien aufzustellen usw., werden Sie später durch vereinfachung der Fehlersuche und des Einsatzes für diese Mühe belohnt. Aus diesem Grund ist es günstig, sich die Berücksichtigung von Umgebungen mit mehreren Rechnern zur Gewohnheit zu machen und solche Umgebungen sogar beim Entwickeln und Testen zu "simulieren".

Die "Simulation" lässt sich glücklicherweise ziemlich einfach durchführen. Es gibt einige einfache Tips für das Betreiben und Testen des Codes, um sicherzustellen, dass dieser auch in einer solchen Umgebung funktioniert, selbst wenn Sie auf einem einzigen Rechner entwickeln. Diese betreffen die am häufigsten vorkommenden Probleme in einer Mehrplatzanlage.

An folgende Tips sollten Sie sich bei der Entwicklung halten:

- starten Sie für jesed Programm, das herunterladbaren Code für andere Programme bereitstellen einen eigenen HTTP-Server
- achten Sie auf Codebase-Probleme
- richten Sie immer einen Security Manager ein
- Denken Sie an Sicherheitsrichtlinien
- legen Sie den CLASSPATH mit Bedacht fest
- überlegen Sie, ob Sie den herunterladbaren Code in einer Jar-Datei zusammenfassen

Sehen wir uns die Punkte der Reihe nach an.

1.12.2.1. Mehrere HTTP-Server verwenden

Es ist günstig, für jedes einzelne Programm, das herunterladbaren Code für andere Programme bereitstellen muss, einen eigenen HTTP-Server zu verwenden. Diese Strategie erlaubt Ihnen, den herunterladbaren Code für jedes Programm deutlichvon dem für andere Programme zu trennen. Eine Alternative, z.B. die Verwendung von eines HTTP-Servers mit einem Stammverzeichnis, das auf die oberste Ebene Ihres Java-Entwicklungsbaums zeigt, ist mit grosser Wahrscheinlichkeit bequemer, weil alle Klassen, die Sie schreiben, von dieser Stelle aus zugänglich sind, aber es verhindert, dass Sie Abhängigkeiten in Ihrem herunterladbaren Code erkennen, die dazu führen könnten, dass Ihre Anwendungen nicht funktionieren, wenn sie auf mehreren Rechnern eingesetzt werden.

Wenn Sie den gesamten herunterladbaren Code für ein Programm deutlich auf einem eigenen HTTP-Server isolieren und dieses Programm mit einer Codebase-Eigenschaft betreiben können, welche die Clients des Programms anweist, Code nur von dem betreffenden HTTP-Server zu holen, merken Sie notfalls, dass nicht der gesamte benötigte Code an einer Stelle zusammengefasst ist - die Clients beschweren sich, dass sie bestimmte erforderliche Klassen nicht laden können.

1.12.2.2. Achten Sie auf Codebase Probleme

Denken Sie daran, dass die Codebase-Eigenschaft auf einem *Server* gesetzt wird (einem Programm, das herunterladbaren Code exportiert), um den *Client* (den Nutzer dieses Codes) mitzuteilen, von welcher Stelle sie die benötigten Klassen herunterladen können. Codebase Probleme zeigen sich in den RMI Funktionen (Aktivierung etc). Deswegen kann es sinnvoll sein, erst eine RMI Verbindung zu testen, bevor Sie Jini-Probleme zu lösen versuchen.

Es gibt ein paar gute allgemeine Tips, die Sie beim Setzen der Codebase-Eigenschaft dennoch beachten sollten. Erstens sollten Sie niemals `file:`URLs verwenden. Wenn ein Server einen `file:`URL an einen Client übergibt, versucht dieser, den gesamten benötigten Code aus seinem eigenen lokalen Dateisystem herunterzuladen. Wenn Sie Ihren Client und Ihren Server auf demselben Rechner entwickeln und testen, mag Ihr Code funktionieren - da sich die Klassendateien für Client und Server an derselben Stelle befinden. Aber wenn Sie Client und Server jemals auf unterschiedlicen rechnern einsetzen, die nicht dasselbe Dateisystem nutzen, stürzt Ihr Code ab.

Ebenso sollten Sie in einer Codebase niemals "localhost" als Host-Namen wählen. "Localhost" wird für den aktuellen Host verwendet. Falls ein Server die Codebase also auf einen URL setzt, der `localhost` enthält, wertet der Client also diese Codebase aus und versucht, den Code aus *seinem* eigenen System zu laden statt aus dem des Servers. Das kann auch hier funktionieren und Sie irreführen, wenn Sie den Client und den Server auf demselben Rechner testen, was wahrscheinlich der Fall ist. Wenn Sie jedoch in Codebase-URLs konkrete Host-Namen verwenden und für jeden einen eigenen HTTP-Server betreiben, wie oben gesagt, könnten Sie potentielle Probleme beim Laden von Code schon früh erkennen.

1.12.2.3. Richten Sie stets einen Security-Manager ein

Jedes Java-Programm, das herunterladbaren Code *verwendet*, sollte durch Aufrufen von `System.setSecurityManager()` einen Security Manager einrichten. Dieser gewährleistet, dass alle Klassen, die entfernt - über eine Codebase, die über RMI bereitgestellt wird - geladen werden, keine unzulässigen Operationen durchführen.

Wenn Sie keinen Security Manager einrichten, werden nur Klassen geladen, die im CLASSPATH der Anwendung vorkommen. Wenn Sie nur lokal testen und keinen Security Manager einrichten, kann Ihr Code daher trotzdem funktionieren, weil die Klassen, die sonst heruntergeladen werden müssten, möglicherweise im CLASSPATH stehen. In einer Umgebung mit mehreren Rechnern wird Ihre Anwendung aber mit Sicherheit versagen.

1.12.2.4. Berücksichtigen Sie die Sicherheitsrichtlinien

Wenn Sie ein Programm mit einem Security Manager einsetzen, verwendet die Sicherheitsmaschinerie von Java 2 standardmässig eine Standardsicherheitsrichtlinie. Diese ist leider häufig zu restriktiv, um den Betrieb von Jini-Anwendungen zuzulassen. deshalb müssen Sie fast immer eine neue Richtliniendatei angeben, die den Start Ihres Programms ermöglicht.

In diesen Beispielen habe ich eine sehr lockere Sicherheitsrichtlinie verwendet - sie ermöglicht unseren Beispielanwendungen freien und unbeschränkten Zugriff auf alle Resourcen. Das ist zwar für das Testen "bekannten" Codes - dh. von Ihnen geschriebenen - ganz nett, eignet sich aber mit Sicherheit nicht für eine Produktionsumgebung.

1.12.2.5. Legen Sie den CLASSPATH mit Bedacht fest

Sie haben wahrscheinlich schon bemerkt, dass die meisten dieser Entwicklungstips mit dem Verhindern unerwünschter gemeinsamer Nutzung von Ressourcen zu tun haben - in erster Linie mit der gemeinsamen Nutzung von Code mit Hilfe unbeabsichtigter und belastbarer Mittel wie `file:`-URLs oder gemeinsam genutzter HTTP-Server. Eine andere Möglichkeit der gemeinsamen Code-Nutzung durch Anwendungen - die Möglichkeit, mit der die meisten von uns vertraut sind - ist der Betrieb von Anwendungen auf demselben Rechner mit demselben CLASSPATH. Wenn ein Client und ein Server Klassendateien von der Festplatte gemeinsam nutzen, ist es praktisch nicht feststellbar, auf welche speziellen Klassen diese Programme entfernt zugreifen müssen. Um die unerwartete gemeinsame Nutzung von Klassendateien zu verhindern, ziehen Sie daher möglicherweise in Betracht, völlig *ohne* CLASSPATH auszukommen. Versuchen Sie lieber dem Java Bytecode Interpreter die Option -cp mitzugeben.Damit können Sie eine Reihe von Verzeichnissen und JAR-Dateien angeben, aus denen Klassendateien geladen werden können. Selbst wenn Sie Ihren Client und Ihren Server auf demselben Rechner entwickeln, können Sie so die Klassendateien für jeden ein anderes Argument -cp für den Java Interpreter verwenden, um sicherzustellen, dass keine unerwünschten Querverbindungen zwischen Ihren Anwendungen vorkommen.

Darüber hinaus ist es eine gute Gewohnheit, *ausschliesslich* jene Klassen bereitzustellen, welche die Anwendung für ihre Aufgaben benötigt, anstatt "sicherheitshalber" alle Klassendateien. Sie können von den drei Jini-JAR Dateien im Argument -cp ausgehen (jini-core.jar, jini-ext.jar und sun-util.jar) und dann nach Bedarf Ihre eigenen speziellen Klassendateien für die Anwendungen hineinkopieren. Der Compiler ist bei der Feststellung, welche Klassendateien installiert werden müssen, eine grosse Hilfe, weil er sich beschwert, wenn er benötigte Klassen nicht finden kann.

Unter keinen Umständen sollten Sie die -dl JAR-Dateien (reggie-dl.jar etc) aus der Jini-Distribution in Ihr Klassenverzeichnis aufnehmen - diese sind zum dynamischen Herunterladen aus den Jini-Kerndiensten vorgesehen. Wenn Sie sie in Ihre Anwendungsklassen aufnehmen, garantieren Sie nur, dass Sie die Versionen dieser Dateien verwenden, die Sie zusammen mit Jini bekommen haben, und nicht die Version, die der Lookup-Dienst *erwartet* und deren Verwendung er von Ihnen *verlangt.*

1.12.2.6. Erwägen Sie das Zusammenfassen herunterladbaren Codes in einer Jar-Datei

Vielleicht die beste Methode, zu gewährleisten, dass Sie alle unerwünschten Abhängigkeiten aus Ihrem Code ausgeschlossen haben und ausschliesslich und genau den Code bereitstellen, den andere Programme herunterladen müssen, besteht in der Erstellung einer JAR-Datei, welche die Klassen enthält, die Clients zur Verwendung Ihrer Programme herunterladen müssen. Das ist die Strategie der Jini-Kerndienste - die Klassen, die der Lookup-Dienst für die Implementierung benötigt, befinden sich beispielsweise in der Datei reggie.jar, diejenigen, die Clients herunterladen können, in reggie-dl.jar. Das Stammverzeichnis des HTTP-Servers, der den herunterladbaren Code des Lookup-Dienstes exportiert, ist auf ein Verzeichnis gesetzt, das reggie-dl.jar enthält. Die Codebase-Eigenschaft für den Lookup-Dienst liefert einen URL, der den HTTP-Server benennt, welcher angibt, von wo Clients die Klassen in reggie-dl.jar herunterladen sollen.

Sie sollten sich überlegen, Ihre Klassen in Teilstücke für die Implementierung und die herunterladbaren Komponenten aufzuteilen. Die Erstellung separater JAR-Dateien für beide

Zwecke ist auch dann praktisch, wenn Sie Ihren Dienst verschieben oder den Fundort für den herunterladbaren Code des Dienstes ändern müssen.

1.12.2.7. Zusammenfassung

Diese Tips sind dazu gedacht, die Isolierung zwischen zwei Jini-Programmen zu simulieren, die auf unterschiedlichen Rechnern laufen - sie benutzen kein gemeinsames Dateisystem und keinen gemeinsamen CLASSPATH. Die Sicherheit muss berücksichtigt werden, weil die Programme den Rechner verlassen müssen, um auf ressourcen zuzugreifen. Und bei einer Anlage im realen Einsatz können Sie nicht garantieren, dass für das Netzwerk ein einziger globaler HTTP-Server vorhanden ist, der alle herunterladbaren Klassendateien enthält, sondern müssen die Trennung der Klassendateien für jeden Dienst und den Zugriff darauf über verschiedene HTTP-Server einplanen.

Die Entwicklung Ihrer Entwicklungsumgebung in dieser Art bedeutet ein wenig Arbeit, aber diese kann sich gewaltig auszahlen, wenn Sie mit der Entwicklung komplexer Dienste und Anwendungen beginnen.

Nachdem wir nun Strategien für die Entwicklung von Jini-Software untersucht haben, wollen wir jetzt daran gehen, konkreten Code zu schreiben.

1.12.3. Ein erstes Jini-Programm : Hello World

Sehen wir uns die kleinsten Jini-Programme an. Wir erstellen einen einfachen Dienst und eine Client-Anwendung, die diesen Dienst einsetzt, und untersuchen schrittweise die Funktion der beiden Programme. Zuerst brauchen wir eine Schnittstelle, etwa die im folgenden Beispiel, die definiert, was unser Dienst tun soll. Das Dienst-Proxy-Objekt implementiert diese Schnittstelle, und der Client verwendet sie jedesmal, wenn er den Lookup-Dienst durchsucht.

Listing 1-1 Hello World Service Interface

```
// dies ist die Schnittstelle, die
// der Proxy des Dienstes implementiert
package ...;
public interface HelloWorldServiceInterface {
     public String getMessage();
}
```

Diese Schnittstelle definiert eine einzige Methode: wenn ein Aufrufer um eine Nachricht bittet, wird eine Zeichenfolge zurückgegeben. Das Proxy-Objekt für den dienst implementiert diese Schnittstelle.

Betrachten wir nun unseren Dienst. in diesem einfachen Fall ist der "Dienst" in Wirklichkeit einfach das Proxy-Objekt -welches die selbständige Rückgabe einer zeichenfolge perfekt beherrscht, ohne mit einem Gerät oder Backend-Prozess zu kommunizieren.

Trotzdem benötigen wir einen Prozess, der einen Lookup-Dienst sucht, den Proxy veröffentlicht und in einem späteren Beispiel das Leasing für unseren Proxy erledigt, um zu gewährleisten, dass die Lookup-Dienste, welche das Lease halten, es nicht "wegwerfen". Dieser Prozess wird als "Mantelprozess" bezeichnet. Er besitzt eine Routine main(), die sich um das Zusammenspiel mit Jini kümmert, welches zur Veröffentlichung des Proxys erforderlich ist.

Sie müssen einige Punkte in bezug auf dieses Beispiel sowie einige Konventionen beachten, an die man sich sinnvollerweise haltet.

1. der grösste Teil des Jini-Codes befindet sich im Paket `net.jini`. Dieses Namensystem entspricht der Konvention, invertierte Domänennamen als Paketnamen zu verwenden: Aspen SmallWorks von Bill Joy, eines der Sun-Forschunglaboratorien, hat die Domäne "jini.net" Anfang 1997 registrieren lassen.

2. Jini unterteilt den Namespace des Pakets, um anzudeuten, was Kernbestandteil des Systems ist und was auf diesem Kern aufbaut. Das Paket `net.jini.core.*` enthält daher zum Kern gehörende Standardschnittstellen, die das Herz des Systems ausmachen. Diese Pakete liegen in der Datei `jini-core.jar`. Die `net.jini.*` Pakete enthalten Bibliotheken, die mit Hilfe der Kernpakete erstellt wurden: dieser Code steht in der Datei `jini-ext.jar`. Schliesslich stellt Sun in den `com.sun.jini.*` Paketen noch ein paar "Helferklassen" bereit. Diese gelten als "nicht zum Standard gehörend" und unterliegen Änderungen. Sie befinden sich in der Datei `sun-util.jar`.

Listing 1-2 HelloWorldService

```
// Hello, World
// Service-- publiziert einen proxy der
// eine auf Clientanfrage eine Zeichenkette liefert.

package corejini.chapter5;

import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceRegistration;
import java.util.HashMap;
import java.io.IOException;
import java.io.Serializable;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;

// Proxy Objekt, welches von Clients heruntergeladen wird
// Es ist serialisierbar un implementiert das
// HelloWorldServiceInterface.
class HelloWorldServiceProxy implements Serializable,
    HelloWorldServiceInterface {
    public HelloWorldServiceProxy() {
    }
    public String getMessage() {
        return "Hello, world!";
    }
}

// HelloWorldService ist die "wrapper" Klasse, die
// Service Items publiziert.
public class HelloWorldService implements Runnable {
    // 10 minute leases
    protected final int LEASE_TIME = 10 * 60 * 1000;
    protected HashMap registrations = new HashMap();
    protected ServiceItem item;
    protected LookupDiscovery disco;

    // Inner class : hört auf Clients
```

```
    class Listener implements DiscoveryListener {
        //
        public void discovered(DiscoveryEvent ev) {
            System.out.println("discovered a lookup service!");
            ServiceRegistrar[] newregs = ev.getRegistrars();
            for (int i=0 ; i<newregs.length ; i++) {
                if (!registrations.containsKey(newregs[i])) {
                    registerWithLookup(newregs[i]);
                }
            }
        }


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

    public HelloWorldService() throws IOException {
        item = new ServiceItem(null, createProxy(), null);

        // Security manager
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }


        disco = new LookupDiscovery(new String[] { "" });

        // Installieren eines Listeners.
        disco.addDiscoveryListener(new Listener());
    }

    protected HelloWorldServiceInterface createProxy() {
        return new HelloWorldServiceProxy();
    }


    synchronized void registerWithLookup(ServiceRegistrar
                                         registrar) {
        ServiceRegistration registration = null;

        try {
            registration = registrar.register(item, LEASE_TIME);
        } catch (RemoteException ex) {
            System.out.println("Couldn't register: " + ex.getMessage());
            return;
        }

if (item.serviceID == null) {
            item.serviceID = registration.getServiceID();
            System.out.println("Set serviceID to " + item.serviceID);
        }

        registrations.put(registrar, registration);
    }

    // This thread does nothing but sleep, but it
    // makes sure the VM doesn't exit.
```

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

public static void main(String args[]) {
        try {
            HelloWorldService hws =
                new HelloWorldService();
            new Thread(hws).start();
        } catch (IOException ex) {
            System.out.println("Couldn't create service: " +
                               ex.getMessage());
        }
    }
}
```

Gehen wir das Beispiel durch. Zuerst müssen wir die erforderlichen Jini-Klassen für die verwendung von Discovery und Lookup importieren. Wir werden hier eine ganze Reihe von Klassen verwenden, in erster Linie aus den Paketen net.jini.core.lookup und net.jini.discovery.

1.12.3.1. Den Dienst-Proxy erstellen

Im Anschluss an den Import sehen Sie die Klassendefinitionen für unseren Dienst-Proxy, der hier HelloWorldServiceProxy heisst. Über diese Klasse muss man einiges wissen.

- Sie werden feststellen, dass sie die Schnittstelle java.io.Serializable implementiert. Das ist ein *Erfordernis*für einen Dienst-Proxy, weil eine Instanz der Klasse "konserviert" und an jeden Lookup-Dienst gesendet werden muss, zu dem wir Kontakte aufnehmen, und dann von dort aus weiter an jeden Client. Serialisierbar zu sein, bedeutet, dass sich der Proxy als Byte-Folge speichern, an einen Socket zu einem Remote-System senden und dort rekonstruieren lässt.

- ausserdem implementiert der Dienst Proxy eine Schnittstelle, nämlich HelloWorldServiceInterface, die dem Client bekannt ist. Der Grund liegt darin, dass das Client-Programm in diesem Beispiel die Lookup-Dienste nach Dienst-Proxies durchsucht, welche diese Schnittstelle implementieren. Im allgemeinen sollten Sie versuchen, soweit wie möglich bekannte Schnittstellen zu verwenden, damit die Clients leichter erfahren, was Ihr Dienst tut. Anders gesagt: der Client muss wissen, worum er bitten soll.

- der Proxy Dienst besitzt einen argumentlosen öffentlichen Konstruktor. Das ist für alle Klassen erforderlich, sie serialisiert und deserialisiert werden.

- in diesem Beispiel wird der Proxy als Klasse der "obersten Ebene", aber nicht öffentlich, in derselben Datei wie die "Mantelanwendung" (wrapper), die ihn veröffentlicht, deklariert. Die hier verwendete Anordnung ist hübsch - denken Sie daran, dass der Client während der Kompilierzeit keinen Zugriff auf den Proxy benötigt und es daher unproblematisch ist, wenn er verborgen ist. Während der Laufzeit erhält der Client über Serialisierung und Herunterladen von Code Zugriff auf diesen Proxy. Sie können einen Proxy auch als verschachtelte innere Klasse deklarieren, mit einer Warnung: innere

Klassen enthalten eine "verborgene" Referenz auf die Instanz der Klasse, von der sie erstellt wurden. Wenn ein Objekt serialisiert wird, werden auch alle darin enthaltenen nicht transienten, nicht statischen Referenzen serialisirt, und das würde dazu führen, dass die Mantelklasse mit dem Proxy zusammengefasst würden. Das wäre mit ziemlich hoher Wahrscheinlichkeit nicht das, was Sie wollen. Wenn Sie Ihren Proxy als innere Klasse deklarieren, achten Sie darauf, ihn als *statische* innere Klasse zu deklarieren, um dieses Problem zu vermeiden. Das Schlüsselwort static für eine innere Klasse bedeutet, dass die Klasse lediglich zum Zweck des bequemen Aufbaus verschachtelt wurde, nicht zum Zweck der Laufzeitverknüpfung der verschachtelten Klasse mit äusseren Klassen. Statische innere Klassen enthalten keine "verborgene" Referenz auf die sie umgebenden Klassen, und deshalb werden die Mantelklassen (wrapper) nicht serialisiert.

1.12.3.1.1. Achtung - Serialisierung vergessen

Wenn Sie wie es manchmal vorkommt, Ihren Dienst-Proxy entweder nicht Serializable machen, oder den argumentlosen Konstruktor weglassen, kann der Compiler diesen Fehler nicht feststellen. Statt dessen bekommen Sie einen Laufzeitfehler, wenn Sie versuchen, diesen Dienst-Proxy bei einem Lookup-Dienst zu registrieren. Die genaue Exception, die dann ausgelöst wird, heisst java.io.NotSerializableException.

Die Klasse ServiceItem, welche den Dienst-Proxy enthält, der an den Registrierungsprozess übergeben wird, ist so definiert, dass sie eine Instanz des Typs Object als Proxy des Dienstes enthält. Daher können Sie Objekte beleibigen Typs übergeben - Sie sollten sich aber bewusst sein, dass Sie während der Kompilierung keine Warnung bekommen, wenn das Objekt nicht serialisierbar ist.

1.12.3.2. Die "Mantelanwendung"

Werfen wir nun einen Blick auf die Anwendung, die Lookup-Dienste sucht und den Proxy veröffentlicht. Dieses Programm hilft dem Proxy nicht bei der Ausführung seiner Dienste; es hilft ihm nur dabei, seine Jini-Verpflichtungen zu erledigen. Im Beispiel heisst die dafür zuständige Klasse HelloWorldService. Sie enthält die Funktion main(), die den Discovery-Prozess einleitet.

Es gibt hier eine weitere innere Klasse, die für das Zusammenspiel mit dem Jini-Discovery-System eingesetzt wird. Diese Klasse mit dem Namen Listener implementiert die Schnittstelle DiscoveryListener. Durch Herauslösen des Discovery-orientierten Codes in eine separate innere Klasse (anstatt die mantelklasse HelloWorldService selbst Discovery durchführen zu lassen) könnten wir den Entwurf ein wenig aufräumen und eine hübsche Trennung zwischen der für die Teilnahme an Discovery erforderlichen Arbeit und der von unseren Dienst verlangten Arbeit beibehalten, Wir kommen auch noch auf die Klasse Listener zurück und werden beschreiben, wie sie funktioniert.

Nach der Deklaration der inneren Klasse für Discovery gelangen wir zum wesentlichen Teil des Programms. Der Konstruktor für HelloWorldService tut vier Dinge. Zuerst legt er eine Instanz von ServiceItem an. Instanzen dieser Klasse werden den Lookup-Diensten während des Registrationsprozesses übergeben.

Sie sehen drei Argumente für diesen Konstruktor: das erste ist die Dienst-ID für den Dienst. Sie ist ein global eindeutiger Bezeichner für Ihren Dienst - selbst wenn Sie den Dienst zu verschiedenen Zeiten bei verschiedenen Lookup-Diensten registrieren lassen, sollte sie gleich

bleiben. Das Erfordernis, dass Dienste sich ihre Dienst-ID merken, zieht für sie einen gewissen Verwaltungsaufwand nach sich. Dieser Dienst ist nicht besonders gut gestaltet, denn er verlangt bei jedem Start eine neue Dienst-ID.

Um die Konstruktion einer ersten Dienst-ID zu erleichtern, verwendet Jini eine Konvention, nach welcher der Lookup-Dienst uns eine global eindeutige Dienst-ID zuweist, wenn Sie bei der ersten Registrierung als Dienst-ID null übergeben. Spätere Registrierungen sollten diese ID verwenden. Wir werden bald sehen, wie man das für unseren einfacen Dienst macht.

Das zweite Argument ist eine Instanz des Dienst-Proxies. Dieses Objekt wird serialisiert und sobald ein Lookup Dienst entdeckt wird, an diesen gesendet, um auf einen Client zu warten, der es haben möchte. Die Konstruktion des Proxy-Objekts ist in eine Methode createProxy() verlagert, damit das Programm klarerr strukturiert ist und spätere Subklassen die Methode überschreiben können.

Das letzte Argument, hier ebenfalls null, kann eine Liste von Attributen sein, die wir mit dem Dienst-Proxy verknüpfen wollen. Interessierte Clients könnten diese Attribute durchsuchen. Wenn Sie beispielsweise einen Drucker registrieren, könnten Sie also Attribute angegeb, die Ort und Modell des Druckers nennen.Clients könnten diese Informationen dann holen und einem Anwender anzeigen oder sie im Rahmen des Programms verwenden, um einen bestimmten Drucker zu suchen. Das Argument wird als array von Objekten eingegeben, welche die Schnittstelle Entry implementieren.

1.12.3.3. Übersetzen

Die folgende Anleitung bezieht sich auf das HelloWorld Programm, welches ursprünglich von Xerox stammt.

Verzeichnisse:
c:\Client
c:\Service
c:\Service-dl
c:\Policy

Befehl zum Übersetzen:
javac -classpath c:\jini1_0_1\lib\jini-core.jar;
c:\jini1_0_1\lib\jini-ext.jar;
c:\jini1_0_1\lib\sun-util.jar;
c:\Service
-d c:\Sevice
c:\.....<wo steht die Datei?> ...\HelloWorldServiceInterface.Java
c:\.....<wo steht die Datei?> ...\HelloWorldClient.Java

und für den Client
javac -classpath c:\jini1_0_1\lib\jini-core.jar;
c:\jini1_0_1\lib\jini-ext.jar;
c:\jini1_0_1\lib\sun-util.jar;
c:\Client
-d c:\Client
c:\.....<wo steht die Datei?> ...\HelloWorldServiceInterface.Java
c:\.....<wo steht die Datei?> ...\Client.Java

Jetzt folgend die üblichen Schritte:

1. einen HTTP Server für den Dienst starten
   java -jar c:\jini1_0_1\lib\tools.jar -dir c:\Service-dl -verbose -port 8085

2. Sicherheitsrichtliniendatei einrichten
   zum Testen kann man einfach die Security "ausschalten"
   In diesem Fall sieht die Policy Datei folgendermassen aus:
   ```
   grant {
       // jeden Zugriff auf alles gewähren
       permission java.security.AllPermission;
   }
   ```

3. ausführen
   ```
   java -cp    c:\jini1_0_1\lib\jini-core.jar;
               c:\jini1_0_1\lib\jini-ext.jar;
               c:\jini1_0_1\lib\sun-util.jar;
               c:\Service
       -Djava.rmi.server.codebase=http://ztnw293.hta.fhz.ch:8085/
       -Djava.security.policy=c:\Policy\policy
       <package>.HelloWorldService
   ```

Falls die meldung ausgegeben wird, dass ein Lookup Dienst gefunden wurde, dann können Sie den Client starten. Dieser wird mit folgender Prozedur gestartet:

```
java -cp    c:\jini1_0_1\lib\jini-core.jar;
            c:\jini1_0_1\lib\jini-ext.jar;
            c:\jini1_0_1\lib\sun-util.jar
            c:\Client
    -Djava.security.policy=c:\Policy\policy
    <package>.HelloWorldClient
```