In diesem Kursteil

- Kursübersicht
  - Einführung in Jini (Kurzwiederholung)
  - Vergleich RMI mit Jini
- Jini - Übersicht
  - Service Registration
  - Service Auffinden (Lookup & Discovery)
    - Unicast Discovery
    - Multicast Discovery
  - Einem Lookup Service beitreten
    - JoinManager
    - ServiceRegistrar.register()
- Service Lookup - Nutzen eines Dienstes
  - Einstellungen Client / Server-seitig
- Wichtige Jini Klassen in Beispielen
  - LookupDiscovery
  - LookupLocator
- Vollständige C/S Programmbeispiele
  - Ein einfaches Client / Server System
  - Client / Server mit dynamischem Lookup
  - Jini ohne RMI - Ein Proxy Beispiel

Jini Praxis

.

1.1. Kursübersicht

In diesem Modul besprechen wir

- die grundlegenden Einstellungen, die nötig sind, um ein Jini oder JavaSpaces Programm auf einer Windows Maschine zum Laufen zu bringen.
- einfache praktische Beispiele zu Jini und JavaSpaces

In diesem Kurs verwenden wir Sun Implementationen von Jini.

1.1.1. Voraussetzungen

Sie sollten vertraut sein mit den Konzepten der objektorientierten Programmierung im Allgemeinen und der Java Programmiersprache im Speziellen.

1.1.1.1. Lernziele

Nach dem Durcharbeiten dieses Moduls sollten Sie in der Lage sein:

- Jini Programme zu kreieren und zu modifizieren.

1.1.1.2. Benötigte Software

Sie benötigen ein aktuelles JDK plus Windows plus
- JiniSDK (dieses umfasst auch bereits JavaSpaces)
- JDK

Die meisten Programme funktionieren nur, falls Sie in einem Netzwerk arbeiten. Windows muss zuerst den TCP/IP Stack laden, sonst funktionieren einige der Beispiele nicht!

Sie können das Laden des Stacks erzwingen, indem Sie die Host Tabelle in Win/System32/Drivers/etc modifizieren und sich eine fixe IP Adresse geben.

Sonst sollte das Meiste problemlos funktionieren. Beachten Sie aber, dass einige der Programme unter Jini1_0 entwickelt wurden und jetzt hier unter 1.1 laufen. Daher treten beim Übersetzen einige Warnungen auf!

Da sicher auch mal ein 1.2, ... kommt, wird dieser Zustand anhalten!

Die meisten Batch Dateien wurden aber so ausgelegt, dass Sie minimal angepasst werden müssen!

1.1.2. Einführung in Jini™

Jini™ ist eine Java™ Erweiterung, welche das Netzwerk als einen Verbund von aktiven und passiven Services betrachtet. Eines der Ziele von Jini ist die spontane Vernetzung von intelligenten Geräten.

Das Modell ist sehr flexibel und gestattet auch die Anbindung kleinster Geräte, also ohne JVM, mittels Proxies.

Die Jini Technologie macht ein Netzwerk dynamisch, indem es das plug-and-play von Geräten gestattet, auf Serviceebene, nicht bloss als Harrdware-Device. Jini stellt Mechanismen für das Hinzufügen und das Entfernen von Geräten zu / vom Netzwerk, dynamisch und ohne die Notwendigkeit jedes Gerät speziell zu konfigurieren.

Der zentrale Mechanismus von einem Jini System ist der Lookup Service, welcher Geräte und Services auf dem Netzwerk registriert und verfügbar macht. Der Lookup Service ist auch der Angelpunkt für den Benutzer des Systems.

Sobald ein Jini Device ins Netzwerk eingefügt wird, versucht dieses den Lookup Service zu finden (Discovery) und sich bzw. seinen Service zu registrieren (Join). Beim Registrieren stellt das Device ein Interface zur Verfügung (übermittelt dieses an den Lookup Dienst), mit dessen Hilfe auf seine Service(s) und Attribute zugegriffen werden kann. Attribute helfen bei der Suche nach einem passenden Service. Danach steht das Gerät / der Service zur netzwerkweiten Nutzung bereit.

Die Nutzung besteht aus dem Suchen (discovery), dem Lookup und Abfragen des / der angebotenen Services und schliesslich dem Aufruf des Interfaces. Dieses Interface kann sein:

- ein RMI Stub
- eine ausführbare Datei
- ein Wrapper umd eine bestehende Applikation, die dadurch für das Jini Netzwerk nutzbar gemacht werden kann.

Grundsätzlich stellt der Code im Lookup Dienst einen Proxy dar, mit dessen Hilfe der Dienst benutzbar gemacht wird. Geräte benötigen somit keine speziellen Driver mehr, das proxy Objekt auf dem Lookup Server stellt sozusagen den Driver dar.

Jini gestattet es somit Föderationen von Devices zu definieren, Ansammlungen von Diensten, welche selber autonom bleiben (Client - Client Computing). Es können mehrere Lokup Services vorhanden sein, die miteinander zusammenarbeiten. Die Dienste werden in der Regel zu Gruppen zusammengefasst, womit Sie leichter ansprechbar und auffindbar werden.

1.1.2.1. Was hat Jini mit dem Telefon zu tun?

Jini Architekt Jim Waldo betont in seinen Vorträgen immer wieder, dass vor vielen Jahren die Installation eines Telefons recht komplex war, dass heute aber jeder einfach ein Kabel einstecken kann und schon ist das Telefon betriebsbereit, sofern eine Nummer vorhanden ist.

Jini versucht eine dynamische Konfiguration der Netzwerkgeräte, also so etwas wie eine spontane Vernetzung. In diesem Sinne stellt Jini den "Klingelton" (gemäss Sun) auf dem Netzwerk zur Verfügung. Ein Gerät, welches ins Jini Netzwerk eingebunden wird, wird automatisch von den anderen Geräten am Netzwerk erkannt und kann somit auch benutzt werden, ohne manuelle Konfiguration.

Jini besteht aus Klassenbibliotheken und einigen Server Programmen. Jini verwendet RPC, remote procedure calls, in Java (=RMI). Jini benötigt etwa 400K, inklusive aller Bibliotheken, Serverprogramme und der virtuellen Maschine.

Verschiedene Firmen (Siemens z.B.) statten ihre Geräte heute bereits mit Jini Software aus.

Die fünf Grundprinzipien von Jini sind:
1) Discovery
2) Lookup
3) Leasing
4) Remote Ereignisse
5) Transaktionen

Als *Discovery* wird das Auffinden und Anbinden von Gemeinschaften innerhalb des Netzwerks bezeichnet.

Lookup bestimmt, auf welche Art und 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 fungieren und das Suchen und Auffinden von Diensten ermöglichen kann. Die Funktionsweise von Lookup ist jedoch komplizierter als die eines einfachen Namensservers. 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 aus Jini Sicht nützlicher) als ein einfacher Zeichenketten-basierter Namensdienst.

Leasing ist eine der wichtigsten Konzepte von Jini, da es so ausgiebig genutzt wird. Die Leasing Technik ermöglicht Jinis Fähigkeit zur "Selbstheilung". Sie sorgt dafür, dass sich eine Gemeinschaft nach Ausfällen an ihr beteiligter Hauptdienste wieder regenerieren kann. Ausserdem sorgt Leasing dafür, dass sich langlebige Dienste (wie beispielsweise Lookup) nicht mit Informationen über ihre Gemeinschaften immer weiter aufblähen. Ohne Leasing könnte das Wachstum solcher langlebiger Dienste ausufern.

Remote-Ereignisse werden von Jini verwendet, um Diensten das gegenseitige Benachrichtigen über ihren Zustand zu ermöglichen. Da Lokup selbst ein Dienst ist, kann es Remote-Ereignisse zur benachrichtigung interessierter Beteiligter über Änderungen der

innerhalb einer Gemeinschaft verfügbaren Dienste verwenden. Jinis Modell für Remote Ereignisse ähnelt dem Ereignismodell von Java Beans, stimmt jedoch nicht mit ihm überein.

Mit Hilfe von *Transaktionen* kann Jini erreichen, dass Berechnungen unter Beteiligung mehrerer Dienste sicher durchgeführt werden können. Damit ist gemeint, 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 ermöglicht Dienste mit grosser Robustheit und Wiederstandsfähigkeit in bezug auf Netzwerkfehler. Das Jini Modell für Transaktionen ähnelt zwar dem in der Datenbankprogrammierung verwendeten Transaktionsmodell, weist jedoch erhebliche interne Unterschiede auf.

1.1.2.2. Vergleich von RMI mit Jini

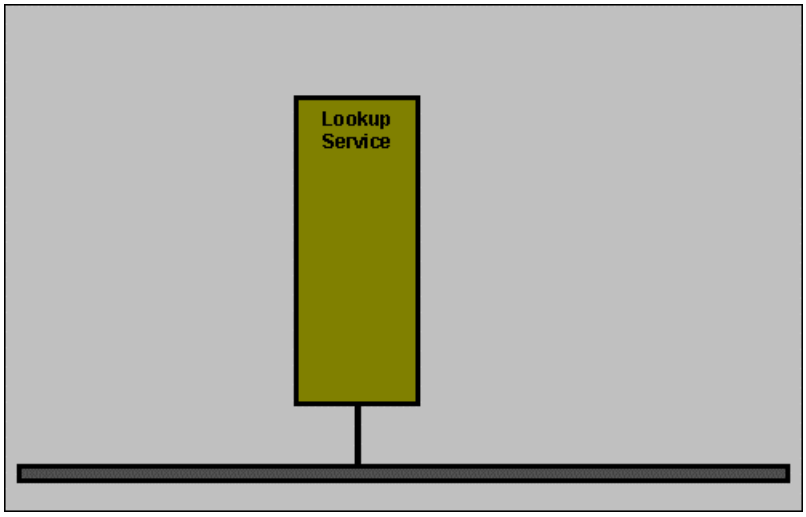
Java RMI	Jini
RMI Clients verwenden die Klasse `Naming.Lookup()` um den benötigten RMI Dienst zu finden.	Jini Clients benutzen den Discovery Prozess, um Jini Lookup Services zu finden. Die Discovery benutzt Multicasting Requests an vordefinierte Adressen oder Ports.
Informationen über andere Service Provider werden in der RMI Registry abgespeichert.	In Jini wird der Dienst, welcher die Informationen über andere Service Anbieter speichert, Jini Lookup Service genannt.
Der RMI Client muss die RMI Registry kennen. Die selbe Regel gilt für den RMI Server.	Der Jini Client sucht den Jini Service ohne den Service Host zu kennen.
Der Client hängt von einem bestimmten Service Provider ab.	Der Client ist unabhängig von einem bestimmten Service provider. Dieses Schema ist also toleranter gegenüber variablen Service providern.
Der RMI proxy-stub wird strikt angewandt.	Das Jini Proxy Konzept ist im wesentlichen Protokoll-unabhängig. Es hängt sicher nicht von einem generierten Stub ab. Der proxy muss sich darum kümmern einen Stub zu erhalten, unter Umständen einen RMI Stub.
RMI kennt keine Transaktionen, verteilte Ereignissteuerung oder Leasing.	Jini kennt Transaktionen, verteilte Ereignissteuerung und Leasing

1.2. Übersicht

1.2.1. Service Registration

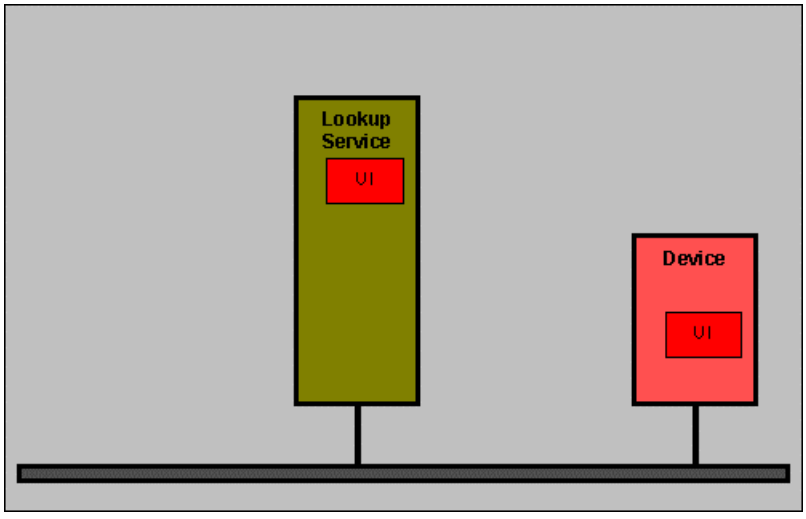
Jini's dynamische Natur resultiert aus dem cleveren Einsatz von mobilem Code. Schauen wir uns erst einmal an, was passiert, wenn ein Gerät in ein Jini Netzwerk eingefügt wird.

Ein aktives Jini Netzwerk benutzt einen Server, den *Jini Lookup Service*. Falls ein Gerät in ein Jini Netzwerk eingefügt wird, verwendet es die Jini Klassenbibliotheken, um den Lookup Dienst zu entdecken, "discover".



Lookup Service

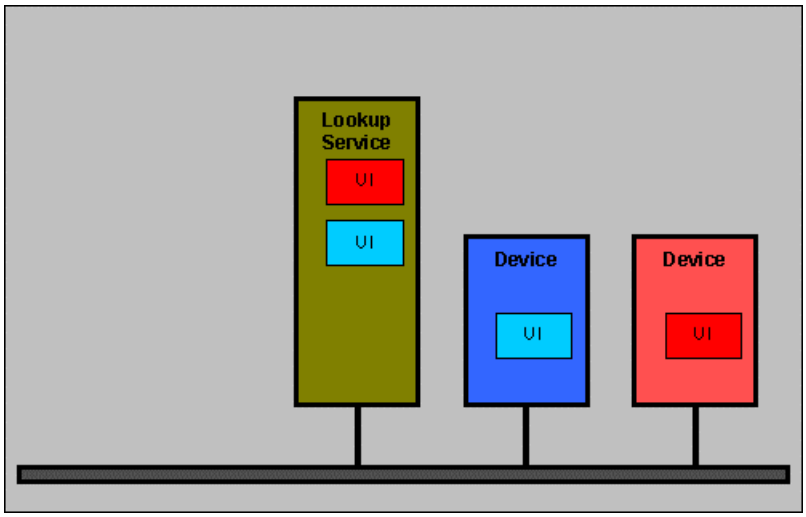
Falls das Gerät irgend einen Dienst anzubieten hat, werden diese /dieser beim Lookup Service eingetragen.



Registration

Als Teil der Registratur kopiert das Device jene Klassen zum Lookup Server, die benötigt werden, um die Dienste des Devices nutzen zu können.

Daher weiss der Lookup Service nicht nur, dass ein Service vorhanden ist, sondern er verfügt auch über Klassen, mit deren Hilfe der Service genutzt werden kann. Die Skizze zeigt wie ein Jini Netzwerk aussehen könnte, nachdem zwei Devices angemeldet wurden.



Service Devices

Ein Servive tragendes Device könnte beispielsweise ein sharable DVD sein, eine Videokamera, eine Mikrowelle, ein Fernseher oder eine automatische Sprinkleranlage. Jede Art Geräte, die irgendwelche Dienste über das Jini Netzwerk offerieren kann, kann seine Dienste anbieten.

1.2.2. Finden eines Lookup Services

Falls Sie einen Dienst entwickelt haben und diesen in einem Jini Netzwerk anbieten möchten, benötigen Sie einen Lookup Server, um Ihren Dienst bekanntgeben zu können. Daher müssen Sie / Ihr Dienst als erster Schritt einen Lookup Dienst finden (Discovery) und sich registrieren.

Discovery geschieht auf zwei Arten:
- entweder mit Unicasting oder
- mit Multicasting

1.2.2.1. Unicast Discovery

Damit Sie eine Unicast Discovery durchführen können, müssen Sie den Host (IP Adresse) und Port des Lookup Services kennen. Das Device sendet ein Unicast Discovery Protokoll Paket an diese Adresse. Der Lookup Server sendet daraufhin ein Unicast Announcement Paket an das Device, um anzuzeigen, dass der Service erkannt und akzeptiert wurde.

Im Jini Framework wird dazu die Klasse `net.jini.core.discovery.LookupLocator` eingesetzt. Als Parameter wird die IP Adresse und der Port in Form einer Jini URL angegeben. Das Format dieser Jini URL ist: jini://Device_IP:Port_NR.

Nach dem Konstruieren des `LookupLocator` Objekts können wir dessen Methode `getRegistrar()` aufrufen, um eine Objektdarstellung des Lookup Services zu erhalten. Dieses wird als `ServiceRegistrar` bezeichnet (Lookup wäre sinnvoller).

Den Rest des Registrierens auf dem Lookup Service geschieht durch das `ServiceRegistrar` Objekt. Alle Details des Unicast Discovery Protocolls werden in dieser Klasse `LookupLocator` zusammengefasst, so dass wir uns darüber keine weiteren Gedanken machen müssen.

Das folgende Programmfragment zeigt, wie das ServiceRegistrar Objekt bestimmt werden kann, mittels Unicast Discovery, durch Angabe einer Jini URL. Falls die Port Nummer fehlt, wird als Standardwert 4160 angenommen.

```
// Instanzieren der LookupLocator Klasse mit einer Jini URL
// Standard Port ist 4160

LookupLocator lookupLocator = new LookupLocator(JiniURL);

// Ausführen der Unicast Discovery zum Lookup Server derJini URL
// Ergebnis: ein Registrar Objekt, welches den Lookup Service repräsentiert

ServiceRegistrar registrar = lookupLocator.getRegistrar();
```

1.2.2.1.1. Ein vollständiges Beispiel - Unicast Discovery

```
package unicastdiscovery;

      import net.jini.core.discovery.LookupLocator;
      import net.jini.core.lookup.ServiceRegistrar;

/**
 * Title:        Unicast Discovery
 * Description:  Einfaches Beispiel, welches zeigt,
 * wie mit Unicast (Angabe der Jini URL: jini://host:port)
 * ein Lookup Service gefunden wird.
 * Copyright:    Copyright (c) J.M.Joller
 * @author J.M.Joller
 * @version 1.0
 */
public class UnicastDiscovery{

      String JiniURL;
      public UnicastDiscovery(String jiniURL){
            JiniURL = jiniURL;
      }

      public UnicastDiscovery(String IP , int port){
            JiniURL = "jini://"+IP+port;
      }
      public ServiceRegistrar startUnicastDiscovery()throws Exception{
      // instanziert ein LookupLocator Objekt
      // zum Lookup Service
      // zur IP und Port Nummer
      // Default Port : 4160
            LookupLocator lookupLocator = new LookupLocator(JiniURL);

            // Aufruf der Methode getRegistrar()
            // des LookupLocator Objekts, um mit
            // Unicast Discovery den Lookup Service zu suchen
            // (IP, Port).
            // Die Methode liefert ein Registrar Objekt
            // Dieses stellt den Lookup Service dar.
            ServiceRegistrar registrar = lookupLocator.getRegistrar();
```

```
        return registrar;
    }



    public static void main(String args[]){
           String jiniURL;
          System.out.println("[UnicastDiscovery]Start");
          UnicastDiscovery ud = null;

          if(args.length < 1){
                jiniURL = "jini://127.0.0.1:4160";
                ud = new UnicastDiscovery("jini://127.0.0.1:4160");
          }
          else
          {
                ud= new UnicastDiscovery(args[0]);
                jiniURL = args[0];
          }
          System.out.println("[UnicastDiscovery]Jini URL: "+jiniURL);
          try{
                ServiceRegistrar reg = ud.startUnicastDiscovery();
                System.out.println("[UnicastDiscovery]Die Jini ServiceID
                                                    :"+reg.getServiceID());
          }catch(Exception e){
                System.out.println("[UnicastDiscovery]Error :"+e);
                e.printStackTrace();
          }
                System.out.println("[UnicastDiscovery]Ende");
          }
    }
```

1.2.2.2. Multicast Discovery

Falls die Lokation des Lookup Services nicht bekannt ist, wird mittels Multicast Discovery gesucht. Dabei wird eine TTL (Time to Live: Standardwert 15 dh. es werden maximal 15 Gateways überquert) so angegeben, dass sich die Suche auf eine "lokale" Umgebung beschränkt. Allerdings muss das Netzwerk Multicasting gestatten, da sonst Jini nicht funktionieren kann! Typischerweise sind die meisten Routern nicht multicastingfähig.

Multicasting ist so etwas wie kontrolliertes Broadcasting: die Multicast Discovery Pakete werden an bestimmte Gruppen von Maschinen gesandt. Sobald ein Lookup Service ein solches Multicast Discovery Paket empfängt, sendet er ein ein Muticast Announcement Paket ins Netzwerk. Dieses findet hoffentlich seinen Weg zum neuen Service.

Der Lookup Service agiert zu diesem Zeitpunkt als `DiscoveryListener`. Als nächstes wird ein `Discovery Event` an den `DiscoveryListener` gesandt. Dieser stellt dann ein Array von `ServiceRegistrar` Objekten zur Verfügung, also grob eine Liste mit den zur Verfügung stehenden Services. Dabei können gleich mehrere Lookup Services auf die erste Multicasting Anfrage antworten. Es liegt dann beim Device, bei welchen es sich registrieren will.

Ein typisches Programm mit Multicast Discovery besitzt folgenden Aufbau:

1) kreieren eines `LookupDiscovery` Objekt
2) kreieren eines `DiscoveryListener` Objekts

3) Aufruf der `addDiscoveryListener()` Methode.
   Damit hört das `DiscoveryListener` Objekt auf Ankündigungspakete der Lookup Services.
4) Aufruf der `discovered()` Methode des `DiscoveryListener` Objekts.
5) Falls ein Service nicht mehr registriert sein soll, wird die `discard()` Methode aufgerufen.

1.2.2.2.1. Ein vollständiges Beispiel - Multicast Discovery

```
package multicastdiscovery;

 import net.jini.core.lookup.ServiceRegistrar;
 import net.jini.core.lookup.ServiceID;
 import net.jini.core.discovery.LookupLocator;
 import net.jini.discovery.LookupDiscovery;
 import net.jini.discovery.DiscoveryListener;
 import net.jini.discovery.DiscoveryEvent;
 import java.io.*;
 import java.rmi.*;
 import java.rmi.server.*;
 import java.util.Vector;

/**
 * Title:        Multicast Discovery
 * Description:  Einfaches Multicast Suchen eines Lookup Services
 * Dieser kann auch erst nach dem Starten dieses
 * Programms aktiv werden (das Programm such eine
 * Weile).
 * Copyright:    Copyright (c) J.M.Joller
 * Company:      Joller-Voss
 * @author J.M.Joller
 * @version 1.0
 */
 /**
  * Das Beispiel verwendet die neusten Jini Klassen!
  * Das programm funktioniert auch als Standalone Programm.
  * Der Aufruf der getLocators() Methode sammelt alle
  * LookupLocator Objekte,
  * bestimmt daraus die Registrar Objekte und alle Services
  */

 public class LookupServiceFind implements DiscoveryListener {
   public static Vector v = new Vector();

   public static synchronized void findAllServices() {
    try {
      if(System.getSecurityManager() == null)
        System.setSecurityManager(new RMISecurityManager());

        // leeren des Vectors um neue Objekte aufnehmen zu können
        v.removeAllElements();

        // kreiere ein LookupDiscovery Objekt
        // Dieses informiert das DiscoveryListener Objekt
        // über neue Lookup Services
        // oder discarded Services
```

```
        LookupDiscovery ld = new
                           LookupDiscovery(LookupDiscovery.NO_GROUPS);

        // Instaieren der LookupServiceFind Klasse als Listener Objekt
        // Falls ein neuer Lookup Service gefunden wird,
        // wird die discovered() Methode aufgerufen

        ld.addDiscoveryListener(new LookupServiceFind());

        // wir könnten auch gleich
        // LookupDiscovery.ALL_GROUPS angeben
        ld.setGroups(LookupDiscovery.ALL_GROUPS);

        // Jetzt brauchen wir etwas Zeit zum Sammeln
        Thread.currentThread().sleep(60000L);
      } catch (Exception e) {
        System.out.println("[LookupServiceFind]Exception:" + e);
      }
    }

     public static void main (String[] args) {
      System.out.println("[MulticastDiscovery]Start");
      findAllServices();
      System.out.println("[MulticastDiscovery]Ende");
     }
     public void discovered(DiscoveryEvent de) {
         try {
            // aufruf von getRegistrar() des LookupLocator
            // Dies führt zu einer Unicast Discovery
            // Damit erhalten wir ein Registar Objekt
            // vom Lookup Service
            ServiceRegistrar[] registrars = de.getRegistrars();

            // Bestimmen der Service ID des Lookup Services getServicerID()
            // des Registrar Objekt (steht für den Lookup Service)

            for (int i = 0; i < registrars.length; ++i) {
                ServiceID id = registrars[i].getServiceID();
                LookupLocator lookupLocator = registrars[i].getLocator();
                v.addElement(lookupLocator);
                String host = lookupLocator.getHost();
                int port = lookupLocator.getPort();

                System.out.println("[MulticastDiscovery]Jini URL: Host
                                                  Port  ServiceID    ");
                System.out.println("[LookupServiceFind]jini://" + host +
                                  ":" + port + ", " + id.toString());
            }
         } catch (Exception e) {
            System.out.println("[LookupServiceFind]Exception:" + e);
         }
     }

     public void discarded(DiscoveryEvent de) {
      // in Echtzeitsystemen könnte man diese Methode einsetzen
        // um bei Nichtverfügbarkeit eines Services Alternativen
        // (alternative Dienste) zu suchen
     }

     public static LookupLocator [] getLocators() {
      // Suche alle Services
      findAllServices();
```

```
      LookupLocator[] locators = new LookupLocator[v.size()];
      // kopiere die Elemente
      v.copyInto(locators);
      // Sende die Objekte
      return locators;
     }
 }
```

1.2.2.2.2. Beispielausgabe

```
MulticastDiscovery]Start
[MulticastDiscovery]Jini URL: Host   Port  ServiceID
[LookupServiceFind]jini://ztnw293:4160, 92652574-ccfd-4790-a2bd-
e3a5c4f7c93e
[MulticastDiscovery]Ende
```

1.2.3. Einem Lookup Service beitreten

Nachdem ein Lookup Service gefunden wurde, muss sich das neue Device anmelden, dem Lookup Service beitreten. Dies geschieht in einer bestimmten Gruppe, da die Lookup Services jeweils zu einer Gruppe gehören. Dabei wird das Service Objekt des neuen Devices an den Lookup Service übermittelt (sein Proxy, Stub, ...). Die Gruppe wird einfach durch die darin enthaltenen Dienste charakterisiert. Beispielsweise könnte eine PrintService Gruppe, eine Scanner Gruppe ... definiert werden.

Gruppen sind, soweit nichts anderes festgelegt wird, public. Die aktuelle Jini Implementation sucht jeweils alle Lookup Services, auch die nicht-öffentlichen.

Die folgenden Klassen spielen beim Registrieren / Joinen eines neuen Dienstes in einem bestehenden Jini Netzwerk die entscheidende Rolle:

- **net.jini.core.lookup.ServiceItem:**
  Diese Klasse kapselt die Dienste der Service Objekte, die ein device anbietet. Ein Device kann mehrere Dienste anbieten, beispielsweise falls es sich um ein multifunktionales gerät handelt. Das `ServiceItem` Objekt ist nur im Zusammenhang mit einem `ServiceRegistrar` Objekt einsetzbar. Ein `ServiceItem` Objekt ist sozusagen ein verpakter Service im Lookup Service.

Datenfelder	
`Entry`[]	`attributeSets` Attribute des Services
java.lang.Object	`service` Ein Service Objekt
`ServiceID`	`serviceID` Eine Service ID, oder null falls der Service noch nicht registriert ist

Konstruktor
`ServiceItem`(`ServiceID` serviceID, java.lang.Object service, `Entry`[] attrSets)

- **net.jini.core.entry.Entry:**
  In Jini kann ein device seine Dienste zusammen mit einem Set von Objekten registrieren. Diese helfen den Service eindeutig zu charakterisieren: man nennt sie deswegen Service *Attribute*.

  Attribute müssen serialisierbar sein. Bei einer Servicesuche werden sie verwendet, um den passenden Dienst zu finden (durch Objektvergleich). Die Attribute müssen serialisierbar sein, weil sie vom Device zum Lookup Service transportiert werden müssen.

  Der Vergleich beim Suchen geschieht direkt auf der Stufe marshalled Objekte.

  Das Jini Interface **net.jini.core.entry.Entry** ist also zentral für das Registrieren und Auffinden von Diensten. Attribute können beispielsweise sein: **Type des Service, Service Location, Service Provider, Service Einsatz,** usw. Jini bietet auch eine

abstrakte Klasse an: **net.jini.entry.AbstactEntry,** die ihrerseits das Entry Interface implementiert und nützliche Funktionen zur Verfügung stellt, wie beispielsweise `equals(), toString()` usw.

Ein Programmfragment für den Einsatz der Entry Klasse:

```
public  class DiskAttr extends
net.jini.lookup.AbstractEntry {

 // Disk Kapazität : Integer Objekt
public Integer capacity = null;

 // Hersteller
public String brandName = null;

 // Disk Status: lockjed / unlocked
public Boolean status = null;
}
```

Alle Methoden und Felder dieser Klasse müssen public sein, da sie den Dienst charakterisieren! Jini speichert das Serviceobjekt zusammen mit den Attributen mit Hilfe der Serialisierung. Daher müssen die Felder `public` sein - nicht `static`, `final` oder `transient`. Auch Basisdatentypen sind daher als Attribute nicht erlaubt, ausser man verwendet die Wrapperklassen, wie im Beispiel oben.

- **net.jini.core.lookup.ServiceRegistration**
  Bei jede Registratur eines Services bei einem Lookup Service liefert dieser ein `ServiceRegistration` Objekt zurück. Dieses liefert Informationen über die Registration:
  1. eine Methode, mit deren Hilfe das `Lease` Objekt zum Service Objekt bestimmt und erhalten werden kann (`Lease` ist ein Jini Interface). Damit kann ein Service auf einfache Art und Weise sein Dienstangebot (Lease) erneuern oder canceln.
  2. eine Methode, mit deren Hilfe die `ServiceID` bestimmt werden kann. Diese identifiziert eindeutig die Dienste beim Lookup Service. Die ID erlaubt es auf einfache Art und Weise den Dienst zu unterhalten. Die ID ist auch eindeutig und garantiert somit, dass ein Dienst nicht mehrfach registriert wird. Dies erhöht die Zuverlässigkeit des Jini Netzwerks.
  3. Methoden, mit deren Hilfe neue Attribute hinzugefügt oder entfernt werden können.

Methoden	
void	`addAttributes`(`Entry`[] attrSets) fügt das Attributset (falls noch nicht vorhanden) zum registrierten Serviceitem hinzu.
`Lease`	`getLease`() liefert ein Lease Objekt. Damit kann man erneuern / canceln.
`ServiceID`	`getServiceID`() liefert die Service ID dieses Dienstes.
void	`modifyAttributes`(`Entry`[] attrSetTemplates, `Entry`[] attrSets) modifiziert das vorhandene Attributset.
void	`setAttributes`(`Entry`[] attrSets) löscht existierende Attribute und setzt die neuen.

- **net.jini.core.lookup.ServiceTemplate**
  Der Suchprozess für bestimmte Dienste im Jini Netzwerk basiert auf Templates. Diese werden in Jini mit Hilfe der `ServiceTemplate` Klasse beschrieben. Ein Jini Template besteht aus den drei Slots:
  1) `ServiceID`
  2) Class Typen (Java Klassen)
  3) `Entry` Typen (Name, Lokation)

Datenfelder	
`Entry`[]	`attributeSetTemplates` Attribute-Set Templates, die überprüft werden können, oder `null`.
`ServiceID`	`serviceID` Service ID des gesuchten Service oder `null`.
java.lang.Class[]	`serviceTypes` Gesuchter Service Typus oder `null`.

Konstruktor
`ServiceTemplate`(`ServiceID` serviceID, java.lang.Class[] serviceTypes, `Entry`[] attrSetTemplates)

Die `Entry` Klasse ist der Supertyp, die Oberklasse aller Einträge, die in einem Lookup Service abgespeichert werden können. Jedes Feld eines Entry Objekts muss public sein. Basisdatentypen müssen gewrapped sein. Beim Serialisieren werden die einzelnen Felder separat serialisiert. Damit kann man die Suchkriterien leicher überprüfen.

Prinzipiell gibt es zwei Möglichkeiten einem Lookup Service beizutreten.
1) mit Hilfe der `JoinManager` Hilfsklasse
2) durch Auflisten der im Jini Netzwerk vorhandenen Dienste und direktes Eintragen bei diesen (`Registrar register()` Methode).

Falls Sie einen Dienst ohne RMI implementieren wollen, muss die Serverseite dauernd aktiv bleiben. Falls Sie RMI verwenden reicht es, dass Sie den Dienst mit Hilfe eines Stubs bei einem RMI Lookup Dienst (Registry oder Activation Daemon) registrieren. Danach kann das Programm beendet werden, die RMI Activation gestattet es diesen Dienst bei Bedarf wieder zu aktivieren, da der Stub ja genau weiss, wo das Objekt steckt bzw. die Klassendatei, sofern Ihre Serverklasse die Klasse `java.rmi.UnicastRemoteObject` (im Falle von RMIRegistry), bzw. `java.rmi.activation.Activatable` (im Falle von RMI Activation Daemon) implementiert.

1.2.3.1. Einsatz der `JoinManager` Hilfsklasse

Die JoinManager Klasse ist eine Hilfsklasse, welche Lookup Services sucht und verfolgt, registriert einen Service und führt die Attribute nach.

Die Klasse besitzt mehrere Konstruktoren. Der einfachste ist:

```
JoinManager(java.lang.Object obj,
            Entry[] attrSets,
            ServiceIDListener callback,
            LeaseRenewalManager leaseMgr)
```

Dieser enthält Angaben zum Service und die Attribute der `Entry`. Falls ein neuer Dienst gefunden wird, kann der Service mittels Aufruf der Methode `serviceIDNotify()` des `ServiceIDListener` informiert werden. Falls der Service nicht an weiteren Services im Netz interessiert ist, kann man dieses Argument auf `null` setzen.

Auch das `leaseMgr` Objekt kann auf `null` gesetzt werden. In diesen Fall wird es einfach kreiert sobald es benötigt wird.

Dieser Konstruktor initiiert eine Suche nach Services, die zur Gruppe "public" gehören. Diese Gruppe wird durch die leere Zeichenkette "" definiert.

```
JoinManager joinMgr = new JoinManager(obj, null, null, null);
joinMgr.setGroups(LookupDiscovery.ALL_GROUPS);
```

Mit dem zweiten Konstruktor ist man flexibler:

```
JoinManager (java.lang.Object serviceobject,
            Entry[] attrSets,
            java.lang.String[] groups,
            LookupLocator [] locators,
            ServiceIDListener callback,
            LeaseRenewalManager leaseMgr)
```

Er enthält eine Multicast Gruppe und `LookupLocator` Objekte. Damit kann man Locators, die zu einer bestimmten Gruppe gehören mittels Nulticasting finden / suchen.

```
JoinManager joinMgr = new JoinManager(serviceobject, null,
                      LookupDiscovery.ALL_GROUPS,
                      null, null, null);
```

Ein weiterer Konstruktor gestattet die Angabe einer ServiceID. Dieser Konstruktor kann auch eingesetzt werden, um Dienste zu aktualisieren.

```
JoinManager(net.jini.core.lookup.ServiceID ServiceID,
            java.lang.Object obj,
            net.jini.core.entry.Entry[] attrSets,
            java.lang.String[] groups,
            net.jini.core.discovery.
            LookupLocator[] locators,
            LeaseRenewalManager leaseMgr)
```

1.2.3.2. Ein Berechnungsdienst (Server) Beispiel

Um einige Konzepte illustrieren zu können, betrachten wir einen Berechnungs-Service, der ein GUI an den Client liefern kann (man kann das GUI auch direkt nutzen), mit dem der Service dann genutzt werden kann.

Im einzelnen müssen wir folgende Aktivitäten ausführen:

a) ein Service Interface definieren, da wir RMI einsetzen wollen.
b) ein GUI Frame definieren, welches vom Berechnungs-Service dem Client zugesandt wird und mit dem der Client dann den Berechnungs-Service nutzen kann.
c) ein Berechnungs-Service implementieren und
d) diesen registrieren, mit dem Join Manager.

Und hier die Programmfragmente im Einzelnen:

- das Interface `Rechner`

```
package joinmanagerberechnungsserver;

import java.awt.Frame;
import java.rmi.*;

/**
 * Kontrakt Interface, welches die Dienste
 * der Berechnungsserver Klasse anzeigt.
**/

public interface Rechner extends Remote {
  public Frame getCalculator() throws RemoteException;
}
```

- GUI Aufruf im Berechnungs-Service

```
// Rechner mit GUI
public Frame getCalculator() throws RemoteException {
       Frame f = new BerechnungsFrame();
       f.setSize(200,150);
       return f;
}
```

- Berechnungs-Service

```
BerechnungsService calculatorService = new BerechnungsService();
String hostName ="localhost";
System.out.println("[BerechnungsService]Binden");
Naming.rebind("rmi://"+hostName+"/BerechnungsService",calculatorServi
ce);
System.out.println("[BerechnungsService]wurde an die RMIRegistry
gebunden");
Object serviceStub =
Naming.lookup("rmi://"+hostName+"/BerechnungsService");
System.out.println("[BerechnungsService]Aufruf des Join Manager");
// Join manager damit der Berechnungsserver Teil der Föderation wird
JoinManager joinManager = new JoinManager(serviceStub,
   attributes,calculatorService, new LeaseRenewalManager());
);
```

Damit steht unser Service bereit. Beim Anmelden müssen wir die RMIRegistry (für diesen Service) UND den RMI Activation Daemon (für Jini) starten, neben dem Web Server.

Nun kann dieser Dienst über das Netzwerk von beliebigen Clients genutzt werden.

Eine Schwachstelle ist die Verwendung des LeaseManagers. Dieser soll dafür sorgen, dass der Dienst periodisch wieder angemeldet wird:

in Jini Netzwerken gelten die Registrationen lediglich für eine limitierte Lease Dauer. Danach muss der Service erneuert werden. Dies kann automatisch geschehen, muss aber eben vorgesehen werden.

Der Konstruktor des `LeaseManager` sieht folgendermassen aus:

```
LeaseRenewalManager ( net.jini.core.lease.Lease lease,
                      long expiration_in_milli_seconds,
                                LeaseListener listener)
```

Das `LeaseListener` Objekt im Konstruktor agiert als Callback Objekt: es besitzt eine `notify()` Methode. Diese wird aufgerufen, falls das Leasing erneuert werden muss oder nicht mehr erneuert wird. Für diese Aktivität kann auch ein separater Thread bereitgestellt werden.

1.2.3.3. Joining mit Hilfe der register() Methode des ServiceRegistrar's

Nun implementieren wir den selben Berechnungs-Service mit einem anderen Registrationsmechanismus. Die Methode unterscheidet sich grundlegend:

1) wir bestimmen zuerst eine Liste der Lookup Services auf dem Netzwerk
2) dann kreieren wir ein `ServiceItem` Objekt, welches unser Berechnungs-Service Objekt und Attribute kapselt.
3) das `ServiceItem` Objekt registriert dieses bei allen Lookup Services aus unserer Liste.

Um die Lookup Services zu finden, setzen wir die Multicast Recovery ein. Das Programm, welches wir oben besprochen haben, enthält eine statische Methode, `getLocators()`, welche unsere Aufgabe bestens erfüllen kann.

```
public static void main(String[] args) {
    try {
      System.out.println("[BerechnungsService]Start");
      System.setSecurityManager(new RMISecurityManager());

      Entry[] attributes = new Entry[1];
      attributes[0] = new Name("RegistrarBerechnungsService");
      System.out.println("[BerechnungsService]LookupLocator :
                                                 getLocators()");
      LookupLocator[] lookupLocators =   LookupServiceFind.getLocators();
      Name name = new Name("RegistrarBerechnungsService");
      Entry []attribs = new Entry[1];
      attribs[0] = name;
      RegistrarBerechnungsService calculatorService = new
                                        RegistrarBerechnungsService();
      String hostName ="localhost";
      Naming.rebind("rmi://"+hostName+
                  "/RegistrarBerechnungsService",calculatorService);
      System.out.println("[BerechnungsService]Service Objekt wurde
                                    an die RMIRegistry gebunden");
      Object serviceStub =
            Naming.lookup("rmi://"+hostName+"/BerechnungsService");
      ServiceItem serviceItem = new ServiceItem(null,
      serviceStub,attribs);
```

```
      for(int i=0;i<lookupLocators.length;i++) {
        ServiceRegistrar registrar = lookupLocators[i].getRegistrar();
        ServiceRegistration registration = registrar.register(serviceItem ,
                                                              Lease.ANY);
        Lease lease = registration.getLease();
        System.out.println("[BerechnungsService]Lease Expiration
                                  :"+lease.getExpiration()+" milli secs");
        System.out.println("[BerechnungsService]Service ID
                                       :"+registration.getServiceID());
      }

    } catch (Exception e) {
      System.err.println("[BerechnungsService]Exception:" + e);
    }
  }
```

Ein registrierter Dienst in einem Jini Netzwerk ist nur für eine beschränkte Zeit registriert. Danach läuft das Leasing ab. Das Programm muss das leasing erneuern, bevor die Leasingzeit abgelaufen ist. Sonst wird der Garbage Collector das Objekt entfernen.

In der Klasse **com.sun.jini.tck.reggie.CreateLookup** wird festgelegt, dass ein Objekt maximal 5 Minuten geleased wird. Das macht in praktischen Systemen wenig Sinn. Jetzt gibt es mehrere Möglichkeiten diese Leasingzeit zu erhöhen, beispielsweise mittels eines Parameters beim Leasinggeber (Reggie Lookup siehe oben).

Eine andere Möglichkeit besteht darin, einen Thread zu bauen, der die Leasingdauer automatisch erneuert.

1.2.3.4. Leasingzeiten

Damit wir die Dienste in einem Jini Netzwerk besser verwalten können, sollten wir uns mit dem Leasing Konzept von Jini befassen. In der Klasse (von Sun, nicht von Jini), welche den Reggie Service implementiert, **com.sun.jini.tck.reggie.RegistrarImpl**, wird das **RegistrarAdmin** Interface implementiert. Dieses enthält fest einprogrammiert die Leasingzeit von 5 Minuten. Neuere Versionen sind flexibler und enthalten einen Parameter, mit dem die Leasingzeit gesteuert werden kann (siehe Beispiele weiter unten). Hier geht es darum, zu zeigen, wie mit einem Thread das Leasing dauernd erneuert werden kann.

```
package leasetimethread;

import net.jini.core.lookup.*;
import net.jini.core.discovery.*;
//import com.sun.jini.tck.reggie.*;
//com.sun.jini.tck.reggie.RegistrarAdmin
import com.sun.jini.reggie.*;
import net.jini.admin.*;

/**
 * Title:        JiniLeasing - Thread für die Erneuerung
 * Description:
 * Copyright:    Copyright (c) J.M.Joller
 * Company:      Joller-Voss
 * @author J.M.Joller
 * @version 1.0
 */
public class LeaseTimeThread {
```

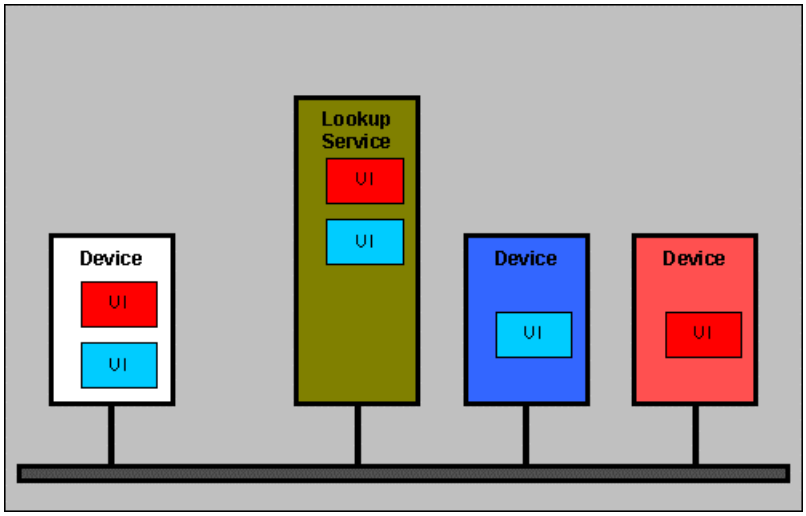
```
  public static void main(String args[]){
    if(args.length <3){
        System.out.println("Usage: java
                                  -Djava.security.policy=policy_file
                                   LeaseTimeThread \n\t\t IP:JiniPort
                                   service_lease event_lease");
    }
    try{
      LookupLocator locator = new LookupLocator("jini://"+args[0] );
      ServiceRegistrar registrar = locator.getRegistrar();
      // Changing the lease time - use RegistrarAdmin
      net.jini.admin.Administrable admin =
                          (net.jini.admin.Administrable)registrar;
      com.sun.jini.reggie.RegistrarAdmin adminProxy = null;
      adminProxy = (com.sun.jini.reggie.RegistrarAdmin) admin.getAdmin();

      long serviceLease = Long.parseLong(args[1]);
      long eventLease = Long.parseLong(args[2]);

      adminProxy.setMinMaxServiceLease(serviceLease);
      adminProxy.setMinMaxEventLease(eventLease);
    } catch(Exception ie){
      System.out.println("Casting Exception");
    }
  }
}
```

1.3. Service Lookup - Nutzen von Jini Diensten

Wie findet nun ein Gerät einen bestimmten Jini Dienst im Jini Netzwerk?

Das neue Gerät verwendet die Jini Klassenbibliothek, um den Lookup Dienst zu finden. Dann fragt es diesen, welche Dienste auf dem Netzwerk angeboten werden.

Der Lookup Dienst sendet dem Gerät die Verbindungsklassen für jeden Dienst, als Teil der Antwort auf die Anfrage.



Der Benutzer des neuen Gerätes oder das Gerät selber kann irgend eine der Verbindungsklassen auswählen, beispielsweise ein GUI. Falls die Verbindungsklasse feststellt, dass der Benutzer / das Device einen Service verlangt, dann kontaktiert es den Service, von dem es ursprünglich stammt, und startet den Service, empfängt eine Antwort und leitet diese an den Benutzer / das Device weiter (Proxy, Stub, Skeleton Mechanismus).

1.3.1. Beweglicher Code

Der eigentlich spezielle Teil bei Jini ist der, dass der *ausführbare* Code, mit dessen Hilfe der Service genutzt werden kann, an den (potentiellen) Benutzer des Dienstes *übermittelt* wird.

Der Client braucht also im voraus keinerlei Kenntnisse über den Service zu haben. Services brauchen nicht notwendigerweise Consumer Devices sein. Denkbar sind auch Serverprogramme oder die Anzeige eines Getränkeautomaten.

Ein Server Programm, welches sich an einem Jini Netzwerk beteiligt, bezeichnet man als Jini *Service*, nicht als Jini Server. Eigentlich stellt Jini auch einen Übergang von Client / Server zu Client / Client dar. Jeder Teilnehmer in einem Jini Netzwerk kann Dienste *anbieten* und Dienste *nutzen*.

1.3.2. Ein Client für den Berechnungsdienst

Here we finally create a client class to test and access the JINI service that we have developed and registered with Jini.

```
public static void main(String args[ ]) {
  try {
    Rechner rechner;
    Entry[] attributes=new Entry[5];
    LookupLocator lookup= new LookupLocator ("jini://localhost");
    ServiceID id;
```

```
    ServiceRegistrar registrar;
    try {
      registrar  = lookup.getRegistrar();
    ServiceTemplate template;
    Frame f;
    // Dienst
    attributes[0] = new Name("Berechnungsdienst");
    // Service Template mit den Attributen
    template = new ServiceTemplate(null, null, attributes);
    // Discovery
    // bestimme ServiceRegistrar
    // Lookup mit dem Template
    // Suchbegriff ist ein Name
    try {
      rechner = (Rechner) registrar.lookup(template);
      f = rechner.getCalculator();
      f.setVisible(true);
    } catch (RemoteException rmie) {rmie.printStackTrace();}

    } catch(MalformedURLException urle) { urle.printStackTrace();
    } catch(ClassNotFoundException cnfe) { cnfe.printStackTrace();}

    } catch (IOException ioe) {ioe.printStackTrace();}

  }
```

1.3.3. Einstellungen und Lokation der Klassen Bibliotheken

Server	Client	Beschreibung
jini.jar	-	Core Jini Interfaces und Klassen
Reggie.jar	reggie-dl.jar	Jini Lookup Service Implementation
Mahalo.jar	mahalo-dl.jar	Jini Transaction Implementation
Outrigger.jar, transient-outrigger.jar	outrigger-dl.jar	JavaSpaces Implementation
jini-examples.jar	jini-examples-dl.jar	Lookup Browser und Service GUI
space-examples.jar	space-examples-dl.jar	JavaSpaces Beispiele
tools.jar	-	ClassServer, DepencyChecker, SerializaionChecker
sun-util.jar	-	Development Utilities
pro.zip	-	ODI Klassen, für Outrigger

1.4. Wichtige Jini Klassen im Detail

Um diese Dienste besser verstehen zu können betrachten wir im Folgenden die wichtigsten Klassen. Wir haben diese teils oben bereits eingesetzt. Hier geht es darum die Klassen genauer oder besser zu verstehen.

1.4.1. Die LookupDiscovery Klasse

Wie funktioniert der Lookup Service auf Gruppennamen Basis?

Diese Klasse gestattet es Instanzen des Lookup Services imNetzwerk zu finden, mit Hilfe des Gruppennamens. Die Klasse agiert als einfache Schnittstelle zum Multicast Request und den Announcement Protokollen.

Datenfelder	
`static java.lang.String[]`	`ALL_GROUPS` versuche alle erreichbaren Lookup Services zu finden.
`static java.lang.String[]`	`NO_GROUPS` Mach nichts (stoppe jegliche Suchprozesse).

Konstruktoren
`LookupDiscovery(java.lang.String[] groups)` Konstuiere ein neues Lookup Discovery Objekt und versuche in denangegebenen Gruppen einen Lookup Dienst zu finden.

Methoden	
`void`	`addDiscoveryListener(DiscoveryListener l)` registrieren eines Listeners, der sich für DiscoveryEvent Notifikation interessiert.
`void`	`addGroups(java.lang.String[] newGroups)` hinzufügen von Gruppe.
`void`	`discard(ServiceRegistrar reg)` eliminiere den Registrar aus dem Set der bereits gefundenen Registrare.
`void`	`finalize()`
`java.lang.String[]`	`getGroups()` liefert die Gruppennamen dieser LookupDiscovery Instanz.
`void`	`removeDiscoveryListener(DiscoveryListener l)` löschen eines Listenereintrages (siehe oben).
`void`	`removeGroups(java.lang.String[] oldGroups)` löschen eines Sets von Gruppen aus dem Such-Set.
`void`	`setGroups(java.lang.String[] newGroups)` setzen einer neuen Suchgruppe.
`void`	`terminate()` Beende den Suchprozess (siehe Beispiel).

Jede Instanz dieser Klasse initialisiert eine Suche nach Lookup Services. Diese Suche wird solange fortgesetzt wie das Objekt aktiv ist (oder bis die `terminate()` Methode aufgerufen wird).

Immer wenn ein brauchbarer Lookup Service gefunden wurde, ruft das `LookupDiscovery` Objekt die Methode `discovered()` für jedes registrierte `DiscoveryListener` Interface auf, mit einem `DiscoveryEvent,` welches genau ein `ServiceRegistrar` Objekt pro gefundenen Lookup Service enthält.

Das *`DiscoveryListener`* Interface muss von allen Objekten implementiert werden, die von einem `LookupDiscovery` Objekt benachrichtigt werden wollen, falls das Objekt verfügbar (`discovered()`)oder zur Zeit oder dauerhaft nicht mehr verfügbar (`discarded()`) ist. Die Objekte, welche dieses Interface implementieren, müssen sich beim `LookupDiscovery` Objekt mittels `addDiscovery()` registrieren, damit sie auf dem aktuellen Stand gehalten werden können. Falls ein Dienst ausfällt, wird in der Regel die `discarded()` Methode nicht gleich aufgerufen, weil das `LookupDirectory` Objekt nicht dauernd nachfragt (nicht polling ist). Dies geschieht erst, sobald eine Anfrage an das Dienstobjekt eine `RemoteException` empfängt. Dann mus der Servicerequester die `discarded()` Methode seines `LookupDiscovery` Objekts aufrufen. Dieses leitet die Meldung an die anderen Listener weiter.

Ein *`DiscoveryEvent`* Objekt ist ein `Event` Objekt, welches mittels `DiscoveryListener` Interface den interessierten Objekten im Jini Netzwerk mitteilt, dass ein oder mehrere `ServiceRegistrar` Objekte vom `LookupDiscovery` Objekt entdeckt (discovered) wurden oder verschwinden. Das Objekt besitzt lediglich eine Methode, `getRegistrars()`, welche ein Array mit `ServiceRegistrar` Objekten liefert. Jedes `ServiceRegistrar` Objekt kann genau auf einen `LookupService` zugreifen.

Ein *`ServiceRegistrar`* definiert das Interface zu einem Lookup Service Das Interface ist kein remote Interface. Jede Implementation des Lookup Service exportiert Proxy Objekte, die das `ServiceRegistrar` Interface lokal beim Client implementieren, mit einem implementationsspezifischen Protokoll, um mit dem aktuellen remote Server kommunizieren zu können.

Sobald das `LookupDiscovery` Objekt feststellt, dass qualifizierte Lookup Services verschwunden sind, ruft es die Methode `discarded()` für jeden `DiscoveryListener` auf, immer mit der selben Information, dass er Dienst nicht mehr zur Verfügung steht.

Lookup Services sind Mitglieder von *Gruppen*. Immer wenn ein LookupDiscovery Objekt instanziert wird, muss die Gruppe angegeben werden, und nur Dienste dieser Gruppe werden gesucht und zurückgegeben.

Falls eine Service Anfrage an einen entdeckten Lookup Dienst eine `RemoteException` empfängt, wird erwartet, dass der Anfragende die Methode `discarded()` des `DiscoveryListener` Objekts aufgerufen wird und damit dem `LookupDiscovery` Objekt mitteilt, dass dieser Service zur Zeit nicht zur Verfügung steht. Das `DiscoveryLookup` Objekt leitet dieses Meldung dann an alle `DiscoveryListeners` weiter.

1.4.1.1. Programmbeispiel

Da das `LookupDiscovery` Objekt kontinuierlich Lookup Services sucht, auch noch nach der Instanzierung, kann es einfach gestartet werden, ohne dass die Dienste bereit sind.. Eine Gefahr in Jini Programmen ist ganz generell, dass einer der Dienste (Reggie=Lookup Dienst, RMID=RMI Daemon) noch nicht vollständig gestartet ist. Daher lohnt es sich in der Regel

beim Starten der Services nicht zu schnell nacheinander diese zu starten, sondern etwas zu warten zwischen den Starts der einzelnen Dienste.

In diesem Beispiel starten wir den Suchlauf, und er funktioniert auch, wie oben erwähnt, selbst wenn noch kein Server Dienst gestartet wurde. Damit das Programm irgend einen Dienst finden kann, müssen Sie nach oder vor dem Starten des LookupDiscoveryBeispiel den RMI Daemon und die Reggie (den Lookup Dienst) starten. Das LookupDiscoveryBeispiel Programm findet dann diesen Dienst und zeigt die Details an.

Im LookupDiscoveryBeispiel wird nach dem `join()` die `terminate()` Methode ausgeführt. da der `join()` den main-Thread veranlasst unendlich lange zu warten, und der Programmabbruch mit CTRL/C erfolgen muss, wird der Methodenaufruf `terminate()` nie ausgeführt. Aber er dient einfach als Dokumentation: so könnte man in einem produktiven Programm die Suche abbrechen.

Falls Sie nur einen Dienst starten wird das Beispielprogramm genau diesen finden: den reggie Lookup Dienst

Versuchen Sie beispielsweise Folgendes:
1) starten Sie den Web Server
2) starten Sie das LookupDiscoveryBeispiel Programm.
   Das Programm findet nichts und sucht einfach immer weiter.
3) starten Sie nun den RMI Daemon und danach die Reggie (den Lookup Server)
   Kurz danach wird das Beispielprogramm die Reggie finden und anzeigen.

1.4.1.2. Starten des Beispielprogramms

Die Reihenfolge ist gegeben durch die Nummerierung der batch Dateien auf dem Server / der CD. Die Reihenfolge muss aber nicht strikt eingehalten werden, siehe oben.

Es steht Ihnen frei, das Programm abzuändern oder im JBuilder zu testen. Beachten Sie, dass Sie Properties und die Jini Bibliothek setzen müssen, bevor Sie im JBuilder irgendetwas Sinnvolles machen können. Details können Sie den Batch Dateien entnehmen.

1.4.1.3. Programmlisting - LookupDiscovery

```
package lookupdiscoverybeispiel;

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

/**
 * Title:        Jini LookupDiscovery Klasse
 * Description:  Das Beispiel zeigt einfach und grob, wie der
 * LookupDiscovery Prozess funktioniert
 * Copyright:    Copyright (c) J.M.Joller
 * Company:      Joller-Voss
 * @author J.M.Joller
 * @version 1.0
 */

    public class LookupDiscoveryBeispiel implements DiscoveryListener
      {
      int i;
      LookupDiscovery discovery;
```

```
ServiceRegistrar[] registrars;
LookupDiscoveryBeispiel ()
  {
  try
    {
    discovery = new LookupDiscovery (LookupDiscovery.ALL_GROUPS);
    discovery.addDiscoveryListener (this);
    System.out.println ("\n[LookupDiscoveryBeispiel]Die Suche nach
                                    einem Lookup Dienst beginnt...");
    }
  catch (Exception e)
    {
    System.out.println("[DiscoveryLookupBeispiel]
                                     LookupDiscoveryBeispiel(): " + e);
    }
  }
public void discovered (DiscoveryEvent evt)
// DiscoveryListener
  {
  registrars = evt.getRegistrars ();
  System.out.println ("[LookupDiscoveryBeispiel]
                         Es wurde ein LookupService gefunden...");
  for (i = 0; i < registrars.length; i++)
    {
    System.out.println ("[LookupDiscoveryBeispiel]
                               \t\tregistrar = " + registrars[i]);
    }
  }
public void discarded (DiscoveryEvent evt)
// DiscoveryListener
  {
  registrars = evt.getRegistrars ();
  System.out.println ("\n[LookupDiscoveryBeispiel]\t\tdiscarded():");
  for (i = 0; i < registrars.length; i++)
    {
    System.out.println ("    registrar = " + registrars[i]);
    }
  }
public static void main (String[] args)
  {
  LookupDiscoveryBeispiel x;
  try
    {
    System.setSecurityManager (new RMISecurityManager ());
    lookup = new LookupDiscoveryBeispiel ();
    Thread.currentThread().join ();
    // warte unendlich lange
    //wird nicht ausgeführt
    lookup.discovery.terminate ();
    // discovery stoppen
    }
  catch (Exception e)
    {
    System.out.println ("[LookupDiscoveryBeispiel]
                         LookupDiscoveryBeispiel.main(): " + e);
    }
  }
}
```

1.4.1.4. Beispielausgabe

```
[LookupDiscoveryBeispiel]Die Suche nach einem Lookup Dienst beginnt...
[LookupDiscoveryBeispiel]Es wurde ein LookupService gefunden...
```

```
[LookupDiscoveryBeispiel]               registrar =
com.sun.jini.reggie.RegistrarProxy@570847da
```

1.4.2. Die LookupLocator Klasse

Diese Klasse ist eine Hilfsklasse, welche ein Unicast basiertes Discovery durchführt.

Datenfelder
`protected java.lang.String host` Der Name des Hosts, auf dem die Suche / Discovery durchgeführt werden soll.
`protected int port` Die Port Nummer beim Host, über die eine Suche ausgeführt werden soll.

Konstruktoren
`LookupLocator(java.lang.String url)` Konstruiere ein neues `LookupLocator` Objekt, welches an der `url` Dienste sucht.
`LookupLocator(java.lang.String host, int port)` Konstruiere ein neues `LookupLocator` Objekt, welches eine Unicast Suche beim Host `host` mit dem Port `port` startet.

Methoden
`boolean equals(java.lang.Object o)` Zwei Locators sind gleich, falls sie den selben Host und Port haben.
`java.lang.String getHost()` Liefert eine Zeichenkette, welche den Namen des Hosts, den das `LookupLocator` Objekt kontaktieren soll.
`int getPort()` Liefert die Portnummer, an den der Dienst sich binden soll.
`ServiceRegistrar getPort()` Führt eine Unicast Discovery durch und liefert das `ServiceRegistrar` Objekt für tden gegebenen Lookup Service.
`ServiceRegistrar getRegistrar()` Führt eine Unicast Discovery durch und liefert das `ServiceRegistrar` Objekt für den gegebenen Lookup Service.
`ServiceRegistrar getRegistrar(int timeout)` Führt eine Unicast Discovery durch und liefert das `ServiceRegistrar` Objekt für den gegebenen Lookup Service.
`int hashCode()`
`java.lang.String toString()` Liefert die Zeichenkettendarstellung dieses `LookupLocator`, als `jini`-Schema URL.

Jede Instanz dieser Klasse repräsentiert einen **LookUp Service.** Der Host und der Port des Services sind bekannt. Diese Klasse wird durch Angabe eines URL ähnlichen Strings angegeben: "jini://localhost:4160". Falls der Port nicht angegeben wird, wird der Standardport 4160 (an dem die Reggie / der Jini Lookup Dienst Anfragen erwartet) verwendet. Während der Instanzierung kontaktiert die Klasse host:port und baut eine Serviceverbindung auf. Dies bezeichnet man als *Unicast Discovery*.

Um mit dem Lookup Service zu kommunizieren, wird die Meldung `getRegistrar()` an die Instanz gesandt (die Methode wird aufgerufen). Als Antwort erhält man ein `ServiceRegistrar` Objekt. Mit diesem werden alle Service Anfragen an den Lookup Service abgehandelt.

1.4.2.1. Programmbeispiel

Das Programm kann fast nichts und besteht auch nur aus einigen wenigen Programmzeilen. Wir instanzieren die obige Klasse mit einem Jini-URL. Anschliessend fragen wir Host und Port ab. Diese sind aber eigentlich ja schon bekannt.

Einzig die Bestimmung des Registrars ist eine nichttriviale Methode. Diese setzt voraus, dass bereits Dienste gestartet wurden. Sonst stürtzt das Programm ab!

1.4.2.2. Starten des Beispielprogramms

Die Reihenfolge ist gegeben durch die Nummerierung der batch Dateien auf dem Server / der CD. Die Reihenfolge muss strikt eingehalten werden, siehe oben.

Es steht Ihnen frei, das Programm abzuändern oder im JBuilder zu testen. Beachten Sie, dass Sie Properties und die Jini Bibliothek setzen müssen, bevor Sie im JBuilder irgendetwas Sinnvolles machen können. Details können Sie den Batch Dateien entnehmen.

1.4.2.3. Programmlisting - LookupLocator

```
package lookuplocatorbeispiel;

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

/**
 * Title:        Einfaches LookupDiscovery Beispiel
 * Description:  Beispiel, welches die Grundfunktionen eines
 * LookupDiscovery
 * Objektes zeigt.
 * Copyright:    Copyright (c) J.M.Joller
 * Company:      Joller-Voss
 * @author J.M.Joller
 * @version 1.0
 */

    public class LookupLocatorBeispiel
      {
      public static void main (String[] args)
        {
        int iPort;
        String sHost;
        LookupLocator lookup;
        ServiceRegistrar registrar;

        try
          {
          System.setSecurityManager (new RMISecurityManager ());

          lookup = new LookupLocator ("jini://localhost");
//entspricht lookup = new LookupLocator ("jini://localhost:4160");

          sHost     = lookup.getHost ();
          iPort     = lookup.getPort ();
          registrar = lookup.getRegistrar ();

          System.out.println ();
```

```
          System.out.println ("[LookupLocatorBeispiel]LookupLocator      =
                                                              " + lookup);
          System.out.println ("[LookupLocatorBeispiel]LookupLocator.host =
                                                               " + sHost);
          System.out.println ("[LookupLocatorBeispiel]LookupLocator.port =
                                                              " + iPort);
          System.out.println ("[LookupLocatorBeispiel]ServiceRegistrar   =
                                                            " + registrar);
          System.out.println ("[LookupLocatorBeispiel]\t Locator:" +
                                              registrar.getLocator());
          System.out.println ("[LookupLocatorBeispiel]\t ServiceID:" +
                                            registrar.getServiceID());
          String [] strGroups = registrar.getGroups();
          for (int i=0; i<strGroups.length; i++)
          System.out.println ("[LookupLocatorBeispiel]\t Group["+i+"]:" +
                                                            strGroups[i]);
          }

        catch (Exception e)
          {
          System.out.println ("[LookupLocatorBeispiel]
                       LookupLocatorBeispiel.main() exception: " + e);
          }
        }
      }
```

1.4.2.4. Beispielausgabe

```
[LookupLocatorBeispiel]LookupLocator      = jini://localhost/
[LookupLocatorBeispiel]LookupLocator.host = localhost
[LookupLocatorBeispiel]LookupLocator.port = 4160
[LookupLocatorBeispiel]ServiceRegistrar   =
com.sun.jini.reggie.RegistrarProxy@886523a5
[LookupLocatorBeispiel]  Locator:jini://ztnw293/
[LookupLocatorBeispiel]  ServiceID:bc69715e-845a-4f3d-917a-ada2212cb064
[LookupLocatorBeispiel]  Group[0]:
```

1.5. Einige Client Server Beispielprogramme

Als erstes betrachten wir eine Applikation, bei der ein Applet als Attribut mitgegeben wird. Clientseitig wird , falls beim Lookup festgestellt wird, dass ein Service ein Applet ist, dieses gestartet, nachdem es mittels Web Server zum Client heruntergeladen wurde.

1.5.1. Ein einfaches Jini Client Server Programm

1.5.1.1. Das Server Interface

```
public interface ServerInterface extends Remote
      {
      public String frageAnServer() throws RemoteException;
      }
```

1.5.1.2. Das Server Programm

```
try {
      System.setSecurityManager (new RMISecurityManager ());
      // Attribute setzen:
      meinServer = new Server ();
      aeAttributes = new Entry[2];
      aeAttributes[0] = new Name ("JiniAppletServer");
      aeAttributes[1] = new ServerApplet (meinServer);
      joinmanager = new JoinManager(meinServer, aeAttributes,
            meinServer, new LeaseRenewalManager ());
      System.out.println ();
      System.out.println ("[Server]JoinManager (a)    = " + joinmanager);

      /*        Suche Jini lookup service (reggie) und gib Lokation aus
      ============================================================  */
      lookup = new LookupLocator ("jini://localhost");
      sHost = lookup.getHost ();
      iPort = lookup.getPort ();
      System.out.println ("[Server]LookupLocator      = " + lookup);
      System.out.println ("[Server]LookupLocator.host = " + sHost);
      System.out.println ("[Server]LookupLocator.port = " + iPort);


      /*        Bestimme des ServiceRegistrar Objekt des Services
                Diese agiert mit dem Lookup Service
      ============================================================  */
      registrar  = lookup.getRegistrar ();
      id  = registrar.getServiceID ();
      System.out.println ("[Server]ServiceRegistrar   = " + registrar);
      System.out.println ("[Server]ServiceID          = " + id);

      try {Thread.sleep (2000);} catch (Exception e) {}
          matches = registrar.lookup(
            new ServiceTemplate (null, null, null),
            50
            );
          System.out.println ("[Server]ServiceMatches     = " + matches);
          System.out.println ("[Server]Anzahl Matches      = "
            + matches.totalMatches);
          for (i = 0; i < matches.totalMatches; i++)
            {
            System.out.println ("[Server]ServiceItem   " + i + ": "
              + matches.items[i]);
            System.out.println ("[Server]ServiceObjekt " + i + ": "
              + matches.items[i].service);
```

```
        }
      }
    catch (Exception e)
      {
      System.out.println("[Server]Server.main(): Exception " + e);
      }
    }
```

1.5.1.3. Das Applet

```
public class ServerApplet extends Applet
  implements ActionListener, Entry
  {
  Button b;
  public ServerInterface meinServerInterface;

  public ServerApplet ()
    {
    // dieser Konstruktor muss da sein, weil sonst das Objekt
    // nicht serialisiert werden kann!!!
    }

  public ServerApplet (ServerInterface meinServerInterfaceIn)
    {
    meinServerInterface = meinServerInterfaceIn;
    }

  public void init ()
    {
    Label l;

    setLayout (new BorderLayout (0, 0));
    l = new Label ("ServerApplet (aus Jini)", Label.CENTER);
    add ("North", l);
    b = new Button ("anklicken");
    b.addActionListener (this);
    b.setFont (new Font ("Helvetica", Font.BOLD | Font.ITALIC, 24));
    add ("Center", b);
    }

  public void actionPerformed (ActionEvent evt)
    {
    if (evt.getSource () == b)
      {
      callServer ();
      }
    }

  void callServer ()
    {
    try
      {
      String strAntwort = meinServerInterface.frageAnServer();
      System.out.println ("[JiniApplet]Server-Methodenaufruf\t\t"
        + strAntwort);
      b.setLabel(strAntwort);
      b.setFont (new Font ("Helvetica", Font.BOLD | Font.ITALIC, 12));
      }
    catch (Exception e)
      {
      System.out.println ("[JiniApplet]callServer(): exception: " + e);
      }
```

```
  }

public String toString ()
  {
  return ("ServerApplet[" + meinServerInterface + "]");
  }
}
```

1.5.1.4. Der Client

```
public class Client
  {
  public static void main (String[] args)
    {
    int i, j;
    int iPort;
    String sHost;
    ClientFrame frame;
    Entry[] aeAttributes;
    LookupLocator lookup;
    ServiceID serviceID;
    ServiceMatches matches;
    ServiceRegistrar registrar;
    ServiceTemplate template;
    Applet applet;
    Object object;

    try
      {


/*    SecurityManager : wegen Codebase
      ========================================================  */
      System.setSecurityManager (new RMISecurityManager ());
/*    Suche Reggie und drucke Lokation aus
      ========================================================  */

      lookup = new LookupLocator ("jini://localhost");
      sHost = lookup.getHost ();
      iPort = lookup.getPort ();
      System.out.println ();
      System.out.println ("[JiniClient]LookupLocator      = " + lookup);
      System.out.println ("[JiniClient]LookupLocator.host = " + sHost);
      System.out.println ("[JiniClient]LookupLocator.port = " + iPort);

/*    Bestimme ServiceRegistrar (ermöglicht die Interaktion
      mit dem lookup service).
      ========================================================  */
      registrar  = lookup.getRegistrar ();
      serviceID  = registrar.getServiceID ();
      System.out.println ("[JiniClient]ServiceRegistrar   = " + registrar);
      System.out.println ("[JiniClient]ServiceID          = " + serviceID);

/*    Bestimme die Services, die beim Lookup Dienst registriert sind
      ========================================================  */
      matches = registrar.lookup
      (
        new ServiceTemplate (null, null, null), 300 );
      System.out.println ("[JiniClient]ServiceMatches     = " + matches);
      System.out.println ("[JiniClient]Anzahl Matches  = "
            + matches.totalMatches);
/*    bearbeiten der registrierten Services
      ========================================================  */
      for (i = 0; i < matches.totalMatches; i++)
      {
        System.out.println ("[JiniClient] " + i + ":   Service Item:   "
         + matches.items[i]);
        System.out.println ("[JiniClient] " + i + ":   Service ID:     "
         + matches.items[i].serviceID);
```

```
        serviceID = matches.items[i].serviceID;
        System.out.println ("[JiniClient] " + i + ":   Service Objekt: "
              + matches.items[i].service);
/*    Ausgabe der Attribute, mit denen der Service registriert ist
      ============================================================  */
      for (j = 0; j < matches.items[i].attributeSets.length; j++)
            {
      object = matches.items[i].attributeSets[j];
        System.out.println ("[JiniClient] " + i + ":" + j + ": Attribute: "
               + object);

/*      Falls die Attribute vom Applet stammen:
        Starte Frame und zeige diese an
        ============================================================  */
        if (object instanceof Applet)
        {
          System.out.println ("[JiniClient]Init Applet Attribute...");
          applet = (Applet) object;
          applet.init ();
          System.out.println ("[JiniClient]Starten des Applet...");
          frame = new ClientFrame ();
          frame.addWindowListener (frame);
          frame.setSize (300, 200);
          frame.add ("Center", applet);
          frame.show ();
        }
       }
      }
     }

        catch (Exception e)
          {
          System.out.println ("[JiniClient]Client.main() exception: " + e);
          }
        }
      }

/**************************************************************************
*    ClientFrame -- unabhängiges Fenster für das Applet
**************************************************************************/
    class ClientFrame extends Frame implements WindowListener
      {
      public void windowOpened (WindowEvent e)
        {
        }
      public void windowClosing (WindowEvent e)
        {
        System.out.println ("[JiniClient]Applet wird geschlossen.");
        System.exit (0);
        }
      public void windowClosed (WindowEvent e)         // WindowListener
        {
        }
      public void windowIconified (WindowEvent e)      // WindowListener
        {
        }
      public void windowDeiconified (WindowEvent e)    // WindowListener
        {
        }
      public void windowActivated (WindowEvent e)      // WindowListener
        {
        }
```

```
public void windowDeactivated (WindowEvent e)    // WindowListener
  {
  }
}
```

1.5.2. Client Server mit dynamischer Suche des Lookup Service

Dabei verwenden wir die Techniken, die wir oben einzeln besprochen haben. Dieses Beispiel kombiniert lediglich bekannte Programmteile, die Listings dieses Beispiels sind daher stark gekürzt.

1.5.2.1.1. Das Server Interface

```
import java.rmi.*;

/************************************************************************
*    ServerInterface -- remote verfügbare Methoden
************************************************************************/
    public interface ServerInterface extends Remote
      {
      public String wieGehts() throws RemoteException;
      }
```

1.5.2.1.2. Der Server

```
public class Server extends UnicastRemoteObject
      implements ServerInterface, ServiceIDListener, DiscoveryListener
  {
  LookupDiscovery discovery;
  ServiceRegistrar[] registrars;
  public Server () throws RemoteException
    {
    try
      {
      discovery = new LookupDiscovery (LookupDiscovery.ALL_GROUPS);
      discovery.addDiscoveryListener (this);

    }...
}
  public String wieGehts() throws RemoteException
  {
   return ("Mir geht's gut! Wie geht's Dir?!");
  }
public void discovered (DiscoveryEvent evt)       // DiscoveryListener
  {
    int i, j;
    ServiceID id;
    Entry[] aeAttributes;
    ServiceTemplate template;
    ServiceMatches matches;
/*  Bestimme die Registrar Objekte
    Das kann dauern
    ======================================================  */
    registrars = evt.getRegistrars ();
    System.out.println ("[Server]Es wurden "
        + registrars.length + " Registrars im Lookup Service[s]
```

```
                                                gefunden...");
    try {Thread.sleep (2000);} catch (Exception e) {}

/*  Versuche das Interface zu finden, in einem der Registrars
      ========================================================  */
      for (i = 0; i < registrars.length; i++)
      {
        id  = registrars[i].getServiceID ();
        System.out.println ("\n[Server]ServiceRegistrar   = "
            + registrars[i]);
        System.out.println ("[Server]ServiceID          = " + id);
        aeAttributes = new Entry[1];
        aeAttributes[0] = new Name ("MeinServer");
        template = new ServiceTemplate (null, null, aeAttributes);
        try
        {
      matches = registrars[i].lookup(new ServiceTemplate (null,null,null),
             50);
...    }
    public void discarded (DiscoveryEvent evt)        // DiscoveryListener
    {
      int i;
      registrars = evt.getRegistrars ();
      System.out.println ("[Server]discarded() Anzahl Registrars = "
          + registrars.length);
      for (i = 0; i < registrars.length; i++)
      {
        System.out.println ("[Server]Registrar      = " + registrars[i]);
      }
    }
    public static void main (String[] args)
    {
      Server meinServer;
      Entry[] aeAttributes;
      JoinManager joinmanager;
      System.out.println("[Server]Remote Server wird gestartet");
      try
      {
      System.setSecurityManager (new RMISecurityManager ());
      aeAttributes = new Entry[1];
      aeAttributes[0] = new Name("MeinServer");
      meinServer = new Server ();
      joinmanager = new JoinManager(meinServer,aeAttributes,
            meinServer,
            new LeaseRenewalManager ()
            );
      }
            ...

      }
```

1.5.2.2. Der Client

```
/*************************************************************************
*   Client --Suchen des Looup Servers mit Discovery
*************************************************************************/
public  class Client implements DiscoveryListener {
  LookupDiscovery discovery;
  ServiceRegistrar[] registrars;
  Client ()  {
    try {
      discovery = new LookupDiscovery (LookupDiscovery.ALL_GROUPS);
      discovery.addDiscoveryListener (this);
      } catch (Exception e)  {               }
  }
  public void discovered (DiscoveryEvent evt)        // DiscoveryListener
        {
        int i, j;
        ServiceID id;
        Entry[] aeAttributes;
        ServiceTemplate template;
        ServerInterface meinServerInterface;

        registrars = evt.getRegistrars ();

/*      Suche ServerInterface
        in einem der Registrars
        ========================================================  */
        for (i = 0; i < registrars.length; i++)
          {
          id  = registrars[i].getServiceID ();
          aeAttributes = new Entry[1];
          aeAttributes[0] = new Name ("MeinServer");
          template = new ServiceTemplate (null, null, aeAttributes);
          try     {
            meinServerInterface
              = (ServerInterface) registrars[i].lookup (template);
            if (meinServerInterface instanceof ServerInterface)
              {
              System.out.println ("[Client]Aufruf der Methode:
                  wieGehts();"+ " Antwort :"  +
                  meinServerInterface.wieGehts());
              System.exit(0);
              }
            }
}
public static void main (String[] args)
        {
       Client meinClient;
        try
          {
            System.setSecurityManager (new RMISecurityManager ());
            meinClient = new Client();
            Thread.currentThread().join ();
            meinClient.discovery.terminate ();
          }
        catch (Exception e)
          {
          System.out.println ("[Client]Client.main(): " + e);
     }
   }
}
```

1.5.3. Client Server - Jini und RMI kombiniert mit Proxy

Der Client holt sich das Proxy Objekt und benutzt dieses um den Server zu kontraktieren.
Der RMI Teil beschränkt sich auf den ersten Teil: dort wird das Proxy Objekt serialisiert zum Client befördert.

1.5.3.1.1. Das Proxy Interface

```
/*************************************************************************
*    ServerProxyInterface -- Proxy's Methoden welche remote verfügbar sind
*************************************************************************/
    public interface ServerProxyInterface
      {
      public String postVomProxy();
      }
```

1.5.3.1.2. Der Server Proxy

```
import java.io.*;
import java.net.*;
/*************************************************************************
*    ServerProxy -- Service's Proxy Class
*************************************************************************/
    public class ServerProxy
      implements ServerProxyInterface, Serializable      {
      String sHost;
      int iPort;

      public ServerProxy (String sHostIn, int iPortIn)        {
        sHost = sHostIn;
        iPort = iPortIn;
        }
      public String postVomProxy() // ServerProxyInterface
        {
        String s;
        StringBuffer sb;
        byte[] ab;
        int iCount;
        Socket socket;
        InputStream is;
        try   {
          ab = new byte[1024];
          sb = new StringBuffer ();
          socket = new Socket (sHost, iPort);
          is = socket.getInputStream ();
          while (true)
            {
            iCount = is.read (ab);
            if (iCount < 0) break;
            s = new String (ab, 0, iCount);
            System.out.println ("[Proxy]Empfang +
                  "\n\t'"+ s + "'"+
                  "\n\t(" + iCount + " Bytes)");
            sb.append (s);
            }
          return (sb.toString ());
          }
        catch (Exception e)
          {
```

```
          e.printStackTrace ();
          }
        return (null);
        }
      }
```

1.5.3.1.3. Server

```
import net.jini.core.entry.*;
import net.jini.core.lookup.*;
import net.jini.core.discovery.*;
import net.jini.lookup.entry.*;
import com.sun.jini.lookup.*;
import com.sun.jini.lease.*;
import java.net.*;
import java.io.*;

/*************************************************************************
*    Server -- Jini Server Klasse
*************************************************************************/
    public class Server
      implements ServiceIDListener, Serializable, Runnable      {
      int iPort;

      public Server (int iPortIn)        {
        Thread th;

        iPort = iPortIn;
        th = new Thread (this);
        th.start ();
        }
      public void serviceIDNotify (ServiceID sidIn)
      // ServiceIDListener
        {
        System.out.println ("[Server]ServiceID = " + sidIn);
        }
      public void run ()
        {
        String s;
        byte[] ab;
        InputStream is;
        OutputStream os;
        Socket socket;
        ServerSocket ssocket;
        try
          {
          System.out.println ("[Server]Warte an Port " + iPort);
          ssocket = new ServerSocket (iPort);
          while (true)
            {
            System.out.println ("[Server]Warte auf eine Verbindung...");
            socket = ssocket.accept ();
            System.out.println ("[Server]Verbindungsaufbau mit "
              + socket.getInetAddress () + "/" + socket.getPort ());
            is = socket.getInputStream ();
            os = socket.getOutputStream ();
            s = "Hallo, ich bin" + this+". Wer bist Du "+
                        socket.getInetAddress () + "/" + socket.getPort();
            // Socket Request empfangen eine Msg
            ab = s.getBytes ();
```

```
      System.out.println ("[Server]Sende \n\t'" + s + "'");
      os.write (ab);
      socket.close ();
      }
    }
catch (Exception e)
  {          }
}
```

```
      public static void main (String[] args)
        {
        int i;
        Server meinServer;
        ServerProxy meinServerProxy;
        LookupLocator lookup;
        Entry[] aeAttributes;
        JoinManager joinmanager;
        ServiceMatches matches;
        ServiceRegistrar registrar;

        try {

/*      Kreiere ein Service und ein Proxy Objekt und registriere beide beim
        lookup service.  Die IP Nummer des Services wird an den Proxy gegeben
        um sicherzustellen, dass dieser weiss, wo der Dienst angefordert
        werden kann.
        ============================================================  */
            System.out.println ();
            System.out.println ("[Server]main: kreiere die Services...");
            meinServer = new Server (7777);
            meinServerProxy = new ServerProxy(
              InetAddress.getLocalHost().getHostAddress(), 7777 );

            System.out.println("[Server]registriere beim Lookup Service");
            aeAttributes = new Entry[1];
            aeAttributes[0] = new Name("ServerProxy");
            joinmanager = new JoinManager
              (
              meinServerProxy,
              aeAttributes,
              meinServer,
              new LeaseRenewalManager ()
              );

/*          Itemize den aktuellen Jini Services
            ================================================================ */

            System.out.println ("[Server]main: Pause 5 Sekunden...");
            try {Thread.sleep (5000);} catch (Exception e) {}

            System.out.println ("[Server]Suchen des Lookup Service...");
            lookup = new LookupLocator ("jini://localhost");
            registrar = lookup.getRegistrar ();

            System.out.println ("[Server]main: Itemizing Services...");
            matches = registrar.lookup(new ServiceTemplate(null, null, null),
                      50);
            System.out.println ("[Server]ServiceMatches     = " + matches);
            System.out.println ("[Server]main: Anzahl Matches      = "
              + matches.totalMatches);
            for (i = 0; i < matches.totalMatches; i++)
              {
              System.out.println ("[Server]main: Service Item   " + i + ": "
                + matches.items[i]);
              System.out.println ("[Server]main: Service Objekt " + i + ": "
                + matches.items[i].service);
              }
            }

          catch (Exception e)
```

```
          {
          System.out.println("[Server] Server.main(): Exception " + e);
          }
        }
      }
```

1.5.3.1.4. Client

```
import net.jini.core.entry.*;
import net.jini.core.lookup.*;
import net.jini.core.discovery.*;
import net.jini.lookup.entry.*;
import com.sun.jini.lookup.*;
import java.rmi.*; // RMI SecurityManager

/*************************************************************************
*    Client -- The Jini Client Class
*************************************************************************/
    class Client
      {
      public static void main (String[] args)
        {
        String s;
        Entry[] aeAttributes;
        LookupLocator lookup;
        ServiceRegistrar registrar;
        ServiceTemplate template;
        ServerProxyInterface meinServerProxyInterface;

        try
          {
          /* lockerer Schutz, damit der Client die Objekte lesen kann
          ==========================================================  */
          System.setSecurityManager (new RMISecurityManager ());

          /* Jini Lookup Suche und Proxy Objekt
          ==========================================================  */

          System.out.println ();
          System.out.println ("[Client]Suche Lookup Service...");
          lookup = new LookupLocator ("jini://localhost");
          registrar  = lookup.getRegistrar ();

          System.out.println ("[Client]Hole ServerProxy...");
          aeAttributes = new Entry[1];
          aeAttributes[0] = new Name ("ServerProxy");
          template = new ServiceTemplate (null, null, aeAttributes);
          meinServerProxyInterface
            = (ServerProxyInterface) registrar.lookup (template);

          /*  Proxy implementiert das gewünschte Interface
              Aufruf der Proxy Methode
          ==========================================================  */
          if (meinServerProxyInterface instanceof ServerProxyInterface)
            {
            System.out.println ("[Client]Aufuf der Methode ueber den
                                 Proxy... ");
            s = meinServerProxyInterface.postVomProxy();
```

```
            System.out.println ("[Client]Post vom Proxy:" +
                    "\n\t'"+s+"'" );
            }
          }

        catch (Exception e)
          {
          System.out.println ("[Client]Client.main() exception: " + e);
    }
  }
}
```