Konzepte von Betriebssystem-Komponenten
Middleware
RMI

Mario Kiefer

21. Januar 2005

1 Einführung

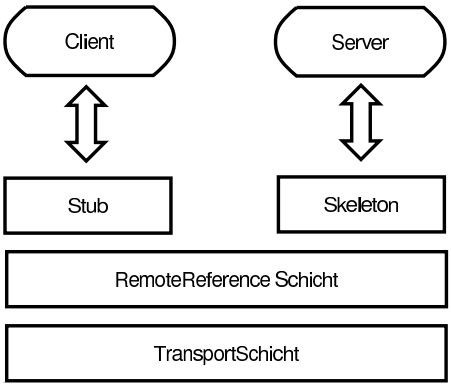
RMI (Remote Method Invocation) ermöglicht es mit relativ einfachen Mitteln verteilte Anwendungen zu erstellen. D.h. es wird ein Mechanismus zur Verfügung gestellt, mit dem es möglich ist auf Objekte zuzugreifen, die auf unterschiedlichen Systemen im Netz verteilt sind. Für den Programmierer besteht der Vorteil z.B. darin, dass er nicht zu unterscheiden braucht, ob er gerade auf ein lokales oder auf ein im Netz verteiltes Objekt zugreift. Die komplette Kommunikation mit dem System, auf dem das angeforderte Objekt liegt, wird von RMI übernommen. z.B. Erstellung eines Sockets, Datenaustausch etc.

2 RMI und Java

RMI ist bereits seit JDK 1.1 ein Teil des Sprachumfangs von Java. Es ist eine eigenständige Erweiterung und besteht aus folgenden Komponenten :

- Packet java.rmi
- Programm rmic (RMI Compiler)
- Programm rmiregistry (Registry mit Naming Service)
- Programm rmid (RMI activation system daemon)

3 Aufbau von RMI



3.1 Stub

Der Stub ist die Schnittstelle zwischen Client und der RemoteReference Schicht. Er dient als Stellvertreter für ein Objekt. Somit nimmt er alle Methodenaufrufe des Clients an das Objekt entgegen und leitet diese weiter. Dafür ist es notwendig, dass der Stub die gleichen Schnittstellen wie das Objekt implementiert.

Folgende Aufgaben übernimmt der Stub bei einem Methodenaufruf :

- Marshalling der Methoden Parameter
- Unmarshalling des Rückgabewertes oder der Exception
- Rückgabe des Wertes an den Client

Mit Marshalling wird der Vorgang bezeichnet, die Parameter des Methodenaufrufs in einen Stream zu schreiben. Die Umkehrung bezeichnet man als Unmarshalling.

3.2 Skeleton

Ein Skeleton ist das serverseitige Pendant zum Stub. Es leitet die Anfragen des Clients an das eigentliche Objekt weiter. Jedem Skeleton ist genau ein Objekt zugeordnet.

Folgende Aufgaben werden von einem Skeleton übernommen:

- Unmarshalling der Methoden Parameter
- Aufruf der Methode des Objektes
- Marshalling des Rückgabewertes oder der Exception

3.3 Stubs und Skeletons

Seit dem Java 2 SDK 1.2 existiert ein zusätzliches Stub-Protokoll, welches die Verwendung von Skeletons in reinen Java 2 Umgebungen überflüssig macht. Stubs und Skeletons werden vom RMI Compiler rmic generiert.

3.4 RemoteReference Schicht

Die RemoteReference Schicht ist für die Transport unabhängigen Funktionen von RMI zuständig, z.B. Verbindungsmanagement, Aufrufsemantik oder Garbage Collection.

In Java 1.1 werden nur Unicast (Punkt-zu-Punkt) Aufrufe unterstützt. Bevor ein Client ein verteiltes Objekt aufrufen kann, ist es erforderlich, dass es auf dem Server bereits instantiiert ist.

In Java 2 wurde das Verhalten zum Aufruf von Objekten dahingehend geändert, das RMI nun auch aktivierbare Objekte unterstützt. D.h. wird das Objekt von einem Client angefordert und ist nicht instantiiert, so instantiiert RMI das Objekt mit Hilfe des rmid. Seit Java 2 sind auch Multicast Aufrufe erlaubt. Z.b. Ein Client verschickt eine Anfrage an mehrere Server und wartet nun auf die schnellste Antwort.

Als Aufrufsemantik wird 'at-most-once' verwendet.

3.5 Transportschicht

Die Transportschicht ist für die Verbindung zwischen Client und Server verantwortlich. Hierfür werden Stream basierte TCP/IP Verbindungen verwendet. RMI benutzt als Protokoll JRMP. JRMP (Java Remote Method Protocol) ist ein properitäres Protokoll, welches HTTP und Objekt Serialisation verwendet. Seit Java 2 SDK 1.3 unterstütz RMI IIOP (Internet Inter ORB Protocol) der OMG (Object Management Group Inc.).

4 RMI Allgemein

4.1 Serialisation

> Serialization is a general purpose mechanism for taking an object and encoding it as a stream of bytes. [1]

Die Java Sprachdefinition stellt Mechanismen zur Verfügung, um primitiven Datentypen zu serialisieren. Klassen, welche nur primitive Datentypen verwenden, sind bereits serialisierbar. Alle diese Klassen nennt man offensichtlich serialisierbar. Ist eine Klasse nicht offensichtlich serialisierbar, muss der Programmierer den Code für die Serialisation zur Verfügung stellen. In [1] wird ein Algorithmus dazu vorgestellt.

Generell muss jede Klasse das Marker Interface Serialization implementieren, damit sie zur Serialisation benutzt werden kann. Dies geschieht entweder direkt, oder durch Vererbung.

4.2 Parameterübergabe

RMI verhält sich bei der Parameterübergabe anders, als es der Programmierer von lokalen Methodenaufrufen gewöhnt ist.

4.2.1 Call-by-Reference

In RMI werden nur Parameter mittels Call-by-Reference übergeben, wenn diese vom Typ Remote sind. Um auf das verteilte Objekt zugreifen zu können, lädt der Empfänger einen Stub.

4.2.2 Call-by-Value

Im Gegensatz zum normalen Aufruf in Java, werden Objekte, die nicht vom Typ Remote sind, kopiert und als Wert übergeben. Der Methodenaufruf beim Empfänger bezieht sich nur auf seine lokale Kopie. Um Objekte mittels Call-by-Value zu benutzen, müssen diese serialisierbar sein.

4.3 Dynamischen Laden von Klassen

RMI ermöglicht es automatisch Klassen nachzuladen, wenn diese lokal nicht verfügbar sind. Hierfür verwendet RMI das HTTP-Protokoll. Über die Eigenschaft (Property) java.rmi.server.codebase wird angegeben, wo die fehlenden Klassen nachgeladen werden können.

4.4 Garbage Collection

Für jedes Remote-Objekt existiert ein Referenzzähler. Dieser Zähler zeigt an, ob dieses Objekt noch von einem Client referenziert wird. Erst wenn der Zähler auf Null steht, kann das Objekt von der lokalen Garbage Collection entfernt werden. Versucht ein Client ein Referenz auf ein nicht mehr vorhandenes Remote-Objekt zu nutzen, liefert ihm der Server eine RemoteException.

4.5 RMI SecurityManager

Innerhalb der JVM werden Klassen in einer sogenannten Sandbox ausgeführt. Damit Client und Server miteinander kommunizieren können, muss es ihnen gestattet sein, Verbindungen zueinander aufzubauen.

Dies durch den RMISecurityManager erledigt. Dieser sollte bei Client und Server als erstes geladen werden.

Ab Java 1.2 wird zusätzlich über die Eigenschaft (Property) java.security.policy Angaben zur Policy gemacht. z.B. :

```
grant {
permission java.net.SocketPermission "*:1024-65535",
"listen,connect,accept,resolve";
permission java.net.SocketPermission "*:1-1023",
"connect,resolve";
};
```

5 Registry und Naming Service

5.1 Naming Service

Für jeden Client ist es wichtig zu wissen, auf welchen Server er sein Remote-Objekt findet. Dazu registriert jeder Server seine Remote-Objekte bei einem Naming Service. Der Client stellt über einen Methodenaufruf eine Anfrage an den Naming Service. Die Anfrage hat den Aufbau einer URL:

```
rmi://<host_name>
      [:<name_service_port>]
                  /<service_name>
```

5.2 Registry

Die Registry, welche beim JDK mitgeliefert wird, implementiert einen solchen Naming Service. Sie wird durch das Programm rmiregistry [port] auf der Konsole gestartet. Standardmäßig lauscht die Registry auf Port 1099.

6 Beispiel

Beispiel ist aus [1] entnommen. Es implementiert einen einfachen verteilten Taschenrechner.

6.1 Interface

Der erste Schritt ist es, ein Interface zu erstellen, welches alle Methoden beinhaltet, die das Remote-Objekt dem Client anbietet.

```
public interface Calculator
        extends java.rmi.Remote {
        public long add(long a, long b)
                throws java.rmi.RemoteException;
        public long sub(long a, long b)
                throws java.rmi.RemoteException;
        public long mul(long a, long b)
                throws java.rmi.RemoteException;
        public long div(long a, long b)
                throws java.rmi.RemoteException;
}
```

Wie man sieht, stammt das Interface Calculator vom Objekt Remote ab. Jede Interface Methode **muss** eine RemoteException werfen können.

```
>javac Calculator.java
```

6.2 Implementierung

Im zweiten Schritt implementiert man die eigentliche Klasse

```
public class CalculatorImpl
        extends
                java.rmi.server.UnicastRemoteObject
        implements Calculator {

        // Implementations must have an
        //explicit constructor
        // in order to declare the
        //RemoteException exception
        public CalculatorImpl()
                throws java.rmi.RemoteException {
                super();
        }
        public long add(long a, long b)
                throws java.rmi.RemoteException {
                return a + b;
        }
        public long sub(long a, long b)
                throws java.rmi.RemoteException {
                return a - b;
        }
        public long mul(long a, long b)
                throws java.rmi.RemoteException {
                return a * b;
        }
```

```
        public long div(long a, long b)
                throws java.rmi.RemoteException {
                return a / b;
        }
}
```

Die Klasse CalculatorImpl implementiert einen Konstruktor mit leerer Parameterliste. Dieser Konstruktor ruft mittels super() den Konstruktor der Klasse UnicastRemoteObject auf. Dort findet die RMI Verlinkung und Initialisierung statt.

```
>javac CalculatorImpl.java
```

6.3 Stubs und Skeletons

Mit dem Programm rmic werden Stub und Skeleton erzeugt.

```
>rmic CalculatorImpl
```

Danach sollten im Arbeitsverzeichnis zusätzlich die beiden Dateien Calculator_Skel.class und Calculator_Stub.class existieren.

6.4 Server

```
import java.rmi.Naming;

public class CalculatorServer {
        public CalculatorServer() {
                try {
                        Calculator c = new CalculatorImpl();
                        Naming.rebind("rmi://localhost:1099/"+
                                        "CalculatorService", c);
                } catch (Exception e) {
                        System.out.println("Trouble: " + e);
                }
        }
        public static void main(String args[]) {
                new CalculatorServer();
        }
}
```

Der Server registriert das Remote Objekt mit der Methode Naming.rebind().

6.5 Client

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.net.MalformedURLException;
import java.rmi.NotBoundException;

public class CalculatorClient {
        public static void main(String[] args) {
                try {
                        Calculator c = (Calculator)
                          Naming.lookup(
                          "rmi://localhost/CalculatorService");
```

```
                System.out.println( c.sub(4, 3) );
                System.out.println( c.add(4, 5) );
                System.out.println( c.mul(3, 6) );
                System.out.println( c.div(9, 3) );
            }
            catch (MalformedURLException murle) {
                System.out.println();
                System.out.println(
                        "MalformedURLException");
                System.out.println(murle);
            }
            catch (RemoteException re) {
                System.out.println();
                System.out.println(
                        "RemoteException");
                System.out.println(re);
            }
            catch (NotBoundException nbe) {
                System.out.println();
                System.out.println(
                        "NotBoundException");
                System.out.println(nbe);
            }
            catch (
                java.lang.ArithmeticException ae) {
                System.out.println();
                System.out.println(
                        "java.lang.ArithmeticException");
                System.out.println(ae);
            }
        }
}
```

6.6 Testen des Beispiels

In jeweils einer eigenen Konsole die aufgeführten Befehle ausführen.

- rmiregistry
- java CalculatorServer
- java CalculatorClient

Literatur

[1] William Grosso. Java RMI. O'Reilly, 2002.

[2] jGuru
http://java.sun.com/developer/onlineTraining/rmi/RMI.html

[3] Daniel Dietterle
www.ihp-ffo.de/systems/Doc/Vorlesung/VBS1/SS%202004/%DCbung/RMIintro.pdf

[4] Jürgen Kleinöder & Franz J. Hauck. Vorlesung: Middleware. FAU Erlangen-Nürnberg & Uni. Ulm, 1997-2004.

[5] Nathalie Weiler. Vorlesung: Java RMI. Züricher Hochschule Winterthur, 2004.