

Remote Method Invocation (RMI)

Hier soll ein Überblick¹ über die RMI-Technologie der Java 2 Plattform gegeben werden.

Inhaltsverzeichnis

1.Überblick.....	3
2.RMI Architektur.....	6
2.1.Überblick.....	6
2.2.RMI Layer.....	8
3.Namensgebung.....	10
4.Beispielanwendung	11
4.1.Interface.....	11
4.2.Klassenimplementierung.....	12
4.3.Stub und Skeleton.....	14
4.4.Host Server	14

¹ Basis: <http://java.sun.com/developer/onlineTraining/rmi/RMI.html>

4.5.Client	15
4.6.Installation und Start	17
5.Parameter und Ergebnisse von RMI.....	19
6.Verteilung und Installation von RMI Software.....	24

1. Überblick

Mit RMI ist es in Java möglich, **Objektorientierte Verteilte Systeme** zu entwickeln. Dabei ist die Idee die gleiche wie bei CORBA: der Zugriff auf Objekte von Clients aus auf Server, die die Objekte verwalten.

Die Designer von RMI wollten es ermöglichen, dass der Java-Programmierer Objekte und Klassen verwenden kann, unabhängig von der JVM, die für die Verwaltung davon zuständig ist.

Dazu muss eine Abbildung gefunden werden, die definiert, wie Objekte und Klassen auf unterschiedliche JVMen verlagert werden. Dazu ist erforderlich zu definieren, wie **Objekte** sich **verhalten** bzgl.:

- Instanziierung von Objekten,
- Fehlerbehandlung,
- Speicherverwaltung und
- Parameterübergabe.

Daraus sind folgende **Unterschiede** zur Verwendung von lokalen Objekten entstanden:

	<i>lokales Objekt</i>	<i>entferntes Objekt</i>
Objekt Definition	Ein lokales Objekt ist definiert durch eine Java Klasse	Das Verhalten eines entfernten Objektes ist definiert durch ein Interface , das vom Interface <code>Remote</code> abgeleitet ist.
Objekt Implementierung	Ein lokales Objekt ist implementiert durch eine Java Klasse	Das Verhalten eines entfernten Objektes ist implementiert durch eine Klasse , die die remote-Schnittstelle implementiert .
Objekt Erzeugung	Eine neue Instanz eines lokalen Objektes wird durch den new-Operator erzeugt.	Eine neue Instanz eines entfernten Objektes wird auf dem Server mit new erzeugt. Ein Client kann ein entferntes Objekt nicht direkt erzeugen. (Ausnahme: Java2 Remote Objekt Activation)
Objekt Zugriff	Auf ein lokales Objekt wird direkt über eine Referenzvariable zugegriffen.	Auf ein remote Objekt wird über eine Referenzvariable zugegriffen, die auf eine Proxystub Implementierung des remote Interface zeigt.
Referenzen	In einer einzigen JVM zeigt eine Referenz immer auf das Objekt im Heap .	Eine Remotereferenz ist ein Zeiger auf ein Proxyobjekt (Stub) im lokalen Heap . Dieser Stub enthält die Informationen, wie auf das Remoteobjekt im Heap der entfernten JVM zugegriffen werden kann.

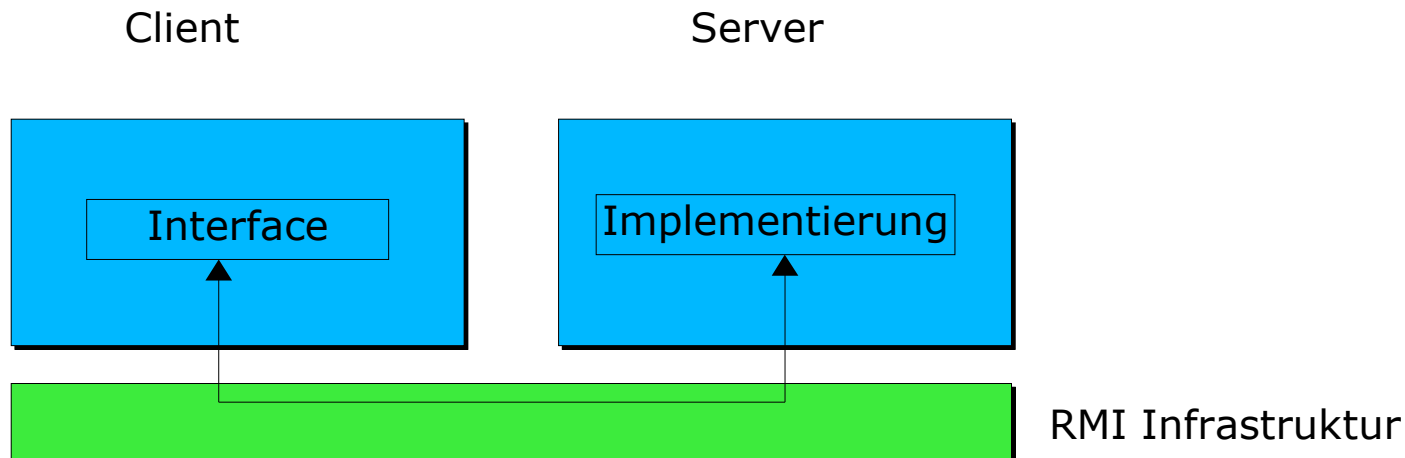
	<i>lokales Objekt</i>	<i>entferntes Objekt</i>
Aktive Referenzen	In einer einzigen JVM ist ein Objekt so lange am leben , wie es eine Referenz auf es gibt.	In einer verteilten Java Umgebung kann eine entfernte JVM „abstürzen“ oder die Netzverbindung kann unterbrochen sein. Ein Remoteobjekt wird als aktiv angesehen, wenn eine Remotereferenz existiert und ein Zugriff darüber innerhalb einer gewissen Zeit (lease period) stattgefunden hat. Wenn alle Remotereferenzen explizit gelöscht wurden oder für alle die „lease period“ abgelaufen ist, wird das Objekt dem Garbage Collector zugänglich gemacht.
Finalizer	Wenn ein Objekt eine finalizer Methode hat, wird sie aufgerufen, bevor der Garbage Collector aktiv wird.	Hat ein remote Objekt ein unreferenced Interface , wird die unreferenced Methode der Schnittstelle aufgerufen, bevor der remote Garbage Collector aktiv wird.
Ausnahmen	Der Java Compiler verlangt, dass alle Exceptions abgefangen werden.	RMI verlangt, dass alle <code>RemoteException</code> abgefangen werden.

2.RMI Architektur

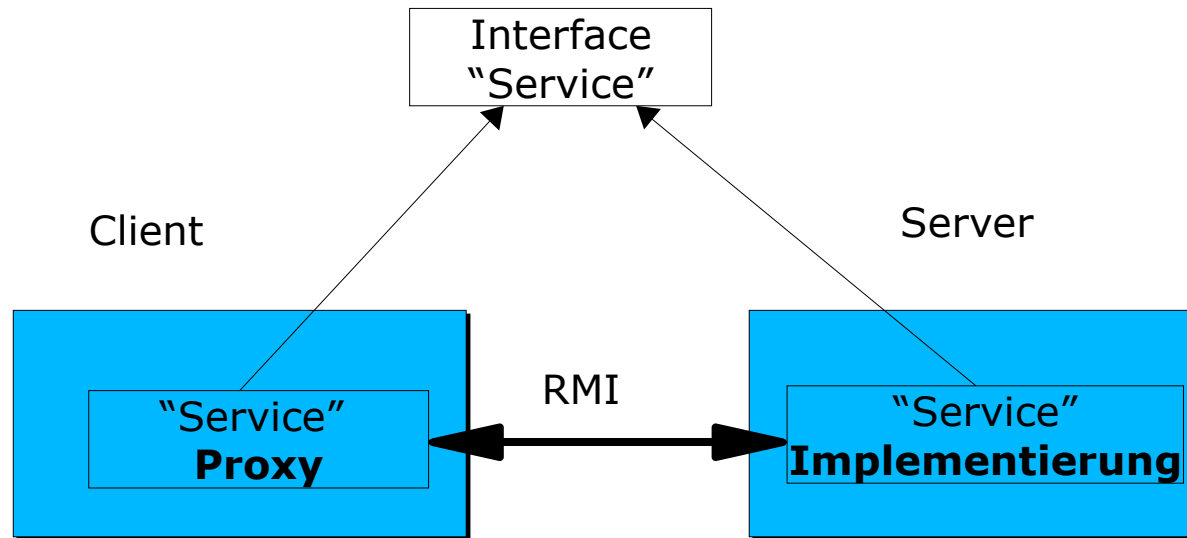
Das **Grundprinzip** von RMI ist die Trennung von der **Definition des Verhaltens** eines entfernten Objektes und dessen **Implementierung**.

2.1.Überblick

Die **Definition** eines Remoteservice wird durch ein Java **Interface** programmiert; die **Implementierung** wird serverseitig durch eine Java **Klasse** realisiert.



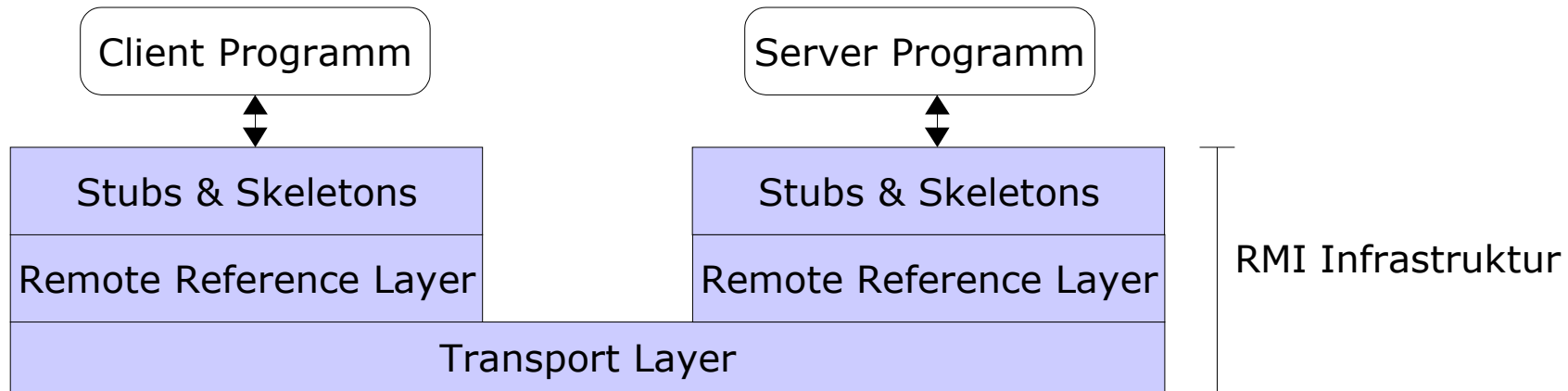
In Java enthält ein **Interface keinen ausführbaren Kode**. Deshalb stellt RMI zwei Klassen bereit, die das selbe Interface implementieren; eine Klasse mit dem Verhalten, die auf dem Server läuft; die andere läuft als Proxy für den Remoteservice auf dem Client:



Der Client macht einen Aufruf einer Methode des Proxyobjektes, RMI sendet die Anfrage an die JVM des Server, wo das Serviceobjekt die Anfrage ausführt und das Ergebnis zurücksendet.

2.2.RMI Layer

Die Realisierung der RMI Architektur basiert auf einem Ebenenkonzept mit 3 Layern:



Damit kann jeder Layer geändert werden, ohne das Gesamtsystem ändern zu müssen. So kann z.B. der Transport Layer auf UDP/IP erweitert werden, ohne Änderungen am Rest vornehmen zu müssen.

Stub und Skeleton Layern

Dieser Layer bildet die **Schnittstelle zum Programmierer**: er **interpretiert Methodenaufrufe** des Client von Interface Referenzvariablen und leitet sie an den Remote Referenz Layer weiter.

Remote Referenz Layer

Hier wird die **Verbindung von Referenzvariablen** zu Remote Service Objekten hergestellt.

Transport Layer

Der Transport Layer ist zuständig für die **Verbindungen von einer JVM zu einer anderen**. Alle diese Verbindungen sind **stream-orientierte TCP/IP** Netzverbindungen.

Dies gilt auch , wenn zwei JVMen auf dem selben Rechner laufen.

3. Namensgebung

RMI nutzt einen Nameservice, der vom Client benutzt wird, um das Remoteobjekt finden zu können. Der Nameservice, der zu einem RMI System gehört, ist `rmiregistry`, der auf jedem Rechner laufen muss, auf dem ein RMI Server ein Objekt verwaltet. Der **Defaultport**, der dazu verwendet wird ist **1099**.

Wenn ein **Serverprogramm gestartet** wird und ein **RMI Objekt** erzeugt wird, wird es beim **RMI System registriert** und ein **Listening-Service gestartet**, der auf Verbindung zu dem Objekt wartet.

Der **Client** benutzt die **Methode** `lookup()` der Klasse `Naming`, um die **RMI Registry** nach einem Objekt **abzufragen**. In `lookup()` wird eine **URL** angegeben, um den Server und den Service zu spezifizieren:

```
rmi://<host>[:<port>]/<service>
```

Die Portangabe ist nur erforderlich, wenn nicht der Defaultport 1099 verwendet wird.

4. Beispielanwendung

Im folgenden wird als Beispiel ein einfacher **Taschenrechner** (wie bei RPC) programmiert. Dabei stellt eine Server das Objekt für den Rechner bereit. Ein Client verwendet das Remoteobjekt, um rechnen zu können.

Die prinzipielle Vorgehensweise beim Realisieren einer RMI-Anwendung ist wie folgt:

1. Programmieren des Javakodes für das **Interface**
2. Programmieren des Javakodes für die **Implementierung** der Klassen
3. **Generieren** der Sup und Skeleton Dateien aus der Implementierung der Klassen
4. **Programmieren** des **Remote Service** des Hostprogramms
5. Programmieren des **Clientprogramms**
6. **Installation** und **Starten** des RMI Systems

4.1. Interface

Das Interface `Calculator` beschreibt die Eigenschaften des Remote Service:

```
$ cat Calculator.java
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;

}
$
$ javac Calculator.java
$
```

dadurch wird es ein Remote Interface

dadurch wird eine Remote Exception erzeugt

4.2. Klassenimplementierung

Jetzt wird die **Implementierung** des **Remote Service** realisiert als Klasse `CalculatorImpl`:

Beispielanwendung

```
$ cat CalculatorImpl.java
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;
    }
}
$
```

Durch den expliziten Konstruktor wird wegen `super()` die Objektinitialisierung auf dem Server durch den Code von `UnicastRemoteObject` ausgeführt.

Jetzt ist die Realisierung der Server-Klasse fertig und man kann die Stubs und Skeletons erzeugen.

4.3.Stub und Skeleton

Das **Generieren** der Stubs und Skeletons wird mit dem **rmi Compiler** vorgenommen:

```
$ rmic CalculatorImpl
$ ls -l *Sk* *St*
-rw-r--r--    1 as      users      2260   4. Mär 12:40 CalculatorImpl_Skel.class
-rw-r--r--    1 as      users      4450   4. Mär 12:40 CalculatorImpl_Stub.class
$
```

rmic hat mehrere Optionen, u.a. auch eine, so dass die java Quellen der Stubs und Skeleton nicht automatisch gelöscht werden.

```
$ rmic -?
```

4.4.Host Server

Der Remote Service braucht noch ein „Hauptprogramm“ (engl. „service must be hosted“):

```
$ cat CalculatorServer.java
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();
    }
}
$
```

4.5.Client

Der Client ist einfach: ein Remote Objekt wird erzeugt und verwendet. Der meiste Code ist das Abfangen der Exception.

Beispielanwendung

```
$ cat CalculatorClient.java
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);  
    }  
}  
$
```

4.6. Installation und Start

Jetzt ist alles fertig und die Komponenten können gestartet werden:

- RMI Registry
- Server
- Client

```
$ rmiregistry &  
[1] 9612  
$ java CalculatorServer &
```

Beispielanwendung

```
[2] 9623  
$ java CalculatorClient  
1  
9  
18  
3  
$
```

Was muss gemacht werden, damit Client und Server auf unterschiedlichen Rechnern laufen?

5.Parameter und Ergebnisse von RMI

Parameter werden in einer Single JVM Umgebung per „**call by value**“ übergeben:

bei einem Methodenaufruf wird vom aktuellen Parameter**wert** eine Kopie erzeugt und auf dem Stack abgelegt. Wird der Parameter innerhalb des Methodenrumpfes verwendet, so wird immer auf die Kopie auf dem Stack zugegriffen.

Ergebnisse werden analog behandelt.

Diese „Mechanik“ wird sowohl für primitive Typen als auch für Objekte (über ihre Referenzvariable) verwendet.

Dies verdeutlicht nochmals (zur Wiederholung) folgendes Javaprogramm:

```
$ cat parameteruebergabe.java
// Demonstriert call by value
public class parameteruebergabe {

    public static class myInt {
        public int value;
        public myInt (int x) {
            value = x;
        }
    }
}
```

Parameter und Ergebnisse von RMI

```
static void incl(int x) {    // einfacher Typ
    x++;
    System.out.println("in incl x="+x);
}

static void inc2(myInt x) { // Referenz
    x.value = x.value+1;    // Wert wird geändert
    System.out.println("in inc2 x="+x.value);
}

static void inc3(myInt x) { // Referenz
    x = new myInt(x.value+1); // x zeigt auf neues Objekt
    System.out.println("in inc3 x="+x.value);
}

public static void main(String args[]) {
    int z=1;
    myInt i= new myInt(1);

    System.out.println("vor incl z=" + z);
    incl(z);
    System.out.println("nach incl z=" + z);
    System.out.println();
}
```

Parameter und Ergebnisse von RMI

```
System.out.println("vor inc2 i=" + i.value);  
inc2(i);  
System.out.println("nach inc2 i=" + i.value);  
System.out.println();
```

```
System.out.println("vor inc3 i=" + i.value);  
inc3(i);  
System.out.println("nach inc3 i=" + i.value);
```

```
}
```

```
}
```

```
$
```

Was wird ausgegeben?

Innerhalb einer **RMI Umgebung** wird diese Mechanik nur für **primitive Typen** verwendet:

Wird ein **primitiver Typ als Parameter einer Remote-Methode** verwendet, so wird die **lokale JVM eine Kopie** des Wertes erstellen und an die **Remote JVM senden** (Ergebnisse analog).

Objekte in einer **RMI Umgebung** werden anders behandelt:

Wird ein **Objekt als Parameter einer Remote-Methode** verwendet, so sendet die **lokale JVM das Objekt** (**nicht** die Referenz) an die **Remote JVM** (Ergebnis analog).

Die ist notwendig, das ja die Remote JVM mit der lokalen Referenz nichts anfangen könnte.

Kompliziert wird es, wenn ein Objekt Referenzen auf andere Objekte enthält (graphartige Objektstruktur). Dann muss die lokale JVM das Objekt **serialisieren** zum die Remote JVM muss es deserialisieren, d.h. die Struktur wieder in seinem Heap aufbauen.

Die kann bei grossen Objekten langsam werden, deshalb gilt bei RMI Anwendungen (wie bei CORBA):

besser **mehrere kleine selbstbeschreibende Klassen** und immer nur das notwendige als Parameter mitgeben, als alles in eine komplexe Klasse packen und die dann verwenden.

Eine dritte Möglichkeit bilden in RMI Systemen die **Remote Objekt Parameter**.

Normalerweise verwendet ein Client die RMI Registry, um eine Remote Referenz zu erhalten. Ein Remote Methodenaufruf kann aber selbst eine solche Referenz zurück liefern, die dann vom Client weiter verwendet werden kann.

Im folgende Beispiel stellt ein Service (BankManager) eine Methode (getAccount()) zur Verfügung, die eine Referenz auf ein Remote Objekt zurück liefert:

```
BankManager bm;
Account      a;
try {
    bm = (BankManager) Naming.lookup("rmi://BankServer/BankManagerService");
    a  = bm.getAccount( "jGuru" );
    // Code that uses the account
}
catch (RemoteException re) { }
```

In der Implementierung von getAccount wird eine (lokale) Referenz auf das remote Objekt zurück geliefert:

```
public Account
    getAccount(String accountName) {
    // Code to find the matching account
    AccountImpl ai =
        // return reference from search
    return ai;
}
```

6. Verteilung und Installation von RMI Software

Um ein RMI zu installieren, müssen die Klassen-Files von Client und Server aus erreichbar sein.

Der Classloader des Server benötigt folgende Class Files:

- Remote Service Interface Definition
- Remote Service Implementierung
- Stubs und Skeletons
- Server Klassen der Anwendung

Der Classloader des Client benötigt folgende Class Files:

- Remote Service Interface Definition
- Stubs und Skeletons
- Server Klassen von Objekten, die der Client verwendet
- Client Klassen der Anwendung

Diese Klassen können entweder „händisch“ auf Client und Server kopiert werden, oder aber von einem zentralen Server geladen werden.

Verteilung und Installation von RMI Software

Dazu wird bei grossen Anwendungen meist ein eigener „Bootstrap Loader“ für Clients und Server programmiert.

(vgl. <http://java.sun.com/developer/onlineTraining/rmi/exercises/BootstrapExample/index.html>)