

# Verteilte Anwendungen

Sven Eric Panitz

30. Mai 2021

## Inhaltsverzeichnis

<b>1</b>	<b>RMI</b>	<b>1</b>
1.1	Remote-Dienste . . . . .	2
1.2	RMI Server . . . . .	3
1.3	RMI Client . . . . .	4
1.4	Serialisierbarkeit . . . . .	5
<b>2</b>	<b>Verteilte Spielanwendung</b>	<b>5</b>
2.1	Spiel-GUI . . . . .	7
2.1.1	Animation . . . . .	7
2.1.2	Ballspielclient . . . . .	8
<b>3</b>	<b>Aufgaben</b>	<b>10</b>

## 1 RMI

Ziel dieses Lehrbrief soll es zeigen, wie man

Java bietet die Möglichkeit, Programme verteilt auf verschiedenen Rechnern laufen zu lassen. Es läuft dabei auf mehreren Rechnern eine Instanz der virtuellen Maschine. Es ist dann möglich, von der einen Maschine Methoden auf Objekten, die auf der anderen Maschinen existieren, aufzurufen. Dieses Prinzip wird RMI bezeichnet. Es steht für *remote method invocation*. Hierzu müssen die Parameter der aufgerufenen Methode über ein Netzwerk an die entfernte virtuelle Maschine geschickt werden und das Ergebnis der Methode ebenso wieder über das Netzwerk an die aufrufende Methode zurückgeschickt werden. RMI realisiert ein Client-Server Modell.

## Java: RMI

```
package name.panitz.rmi;

import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.UnicastRemoteObject;

import java.util.*;
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import static java.awt.event.KeyEvent.*;

public interface RMI{
```

### 1.1 Remote-Dienste

Schnittstellen sind in der Modellierung einer RMI Anwendung von entscheidender Bedeutung. Über eine Schnittstelle definiert der Server, welche Funktionalität er anbietet. Eine Schnittstelle, die definiert, was ein RMI Server für Methoden zur Verfügung stellt, leitet von der Schnittstelle `java.rmi.Remote` ab.

Beginnen wir mit einer ersten einfachen Anwendung. Unser Server soll eine Methode `getDate` anbieten, die das aktuelle Datum mit Uhrzeit vom Server zurückgibt. Dieses läßt sich einfach definieren:

## Java: RMI

```
interface GetDate extends Remote {
    Date getDate() throws RemoteException;
}
```

Jede Methode, die ein RMI Server über das Netzwerk als entfernte Methode anbietet, muß in seiner `throws`-Klausel die Ausnahmeklasse `RemoteException` auflisten.

Eine Schnittstelle, die `Remote` erweitert läßt sich wie jede normale Schnittstelle in Java erweitern. Es muß auf keinsten Weise berücksichtigt werden, daß die Methoden über ein Netzwerk von entfernten virtuellen Maschinen aufgerufen werden. Für unser erstes Beispiel genügt folgende naheliegende Implementierung.

## Java: RMI

```
class GetDateImpl implements GetDate {
    public Date getDate() {return new Date();}
}
```

## 1.2 RMI Server

Mit der Schnittstelle `GetDate` haben wir spezifiziert, welche Funktionalität übers Netz angeboten werden soll. Die Implementierung setzt diese Funktionalität um. Jetzt brauchen wir noch ein Programm, das diese Funktionalität auch im Netz bekannt macht und anbietet; entsprechende Methodenaufrufe aus dem Netz entgegennimmt, ausführt und das Ergebnis an die aufrufende Maschine zurücksendet. Hierzu ist eine Serveranwendung zu schreiben. Diese benötigt eine Instanz der implementierenden Klasse, und muß diese für die Benutzung unter einem Namen für das Netzwerk registrieren. Hierzu gibt es in Paketen `java.rmi.registry` bzw. `java.rmi.server` drei entscheidene Klassen:

- **UnicastRemoteObject**: die statische Methode `exportObject` dieser Klasse wandelt ein Objekt in ein Instanz um, die geeignet ist, über das Netz aufgerufen zu werden. Dahinter verbirgt sich tatsächlich unter der Hand, der ganze Code, der für die Vorbereitung der Kommunikation notwendig ist.
- **LocateRegistry**: hier kann mit der statischen Methode `getRegistry` eine Instanz der auf dem Server laufenden Registrierungsumgebung für RMI Objekte erhalten werden.
- **Registry**: Objekte dieser Klasse haben die Methode `bind`, in der Remote Objekte in der RMI Umgebung bekannt gemacht werden können. Hierbei bindet man einen beliebigen Namen an das Remote Objekt. Unter diesen Namen können Client-Applikationen auf das Remote-Objekt zugreifen.

Für unser kleines erstes Beispiel einer Servers, der die Zeit angeben kann, ergibt das folgenden Code. Zunächst wird eine Instanz der implementierenden Klasse erzeugt. Diese wird als ein Remote Objekt exportiert, dann wird die RMI-Registry geholt, auf der dann das Objekt bekannt gemacht wird.

### Java: RMI

```
static class DateServer {
    public static void main(String args[]) throws Exception{
        var obj = new GetDateImpl();
        var stub = (GetDate) UnicastRemoteObject.exportObject(obj, 0);
        LocateRegistry.getRegistry().bind("date", stub);
        System.err.println("Server ready");
    }
}
```

Die drei bis hierhin geschriebenen Javodateien sind ganz normal mit dem `Javac` Compiler zu übersetzen. Bevor jedoch die Hauptmethode von `DateServer` gestartet werden kann, ist die RMI-Registrierung zu starten. Hierzu liefert das JDK das Programm `rmiregistry` mit. Dieses ist auf dem Servermaschine zunächst einmal zu starten. Dann ist das Programm `DateServer` zu starten. Diesem ist aber als Umgebungsparameter mitzugeben, wo der Server seine Javaklassen findet. Hier reicht der Klassenpfad nicht aus. Es ist

das Attribut `java.rmi.server.codebase` zu setzen. Insgesamt erhalten wir folgende kleine Session auf der Kommandozeile:

Shell

```
sep@pc305-3:~> rmiregistry &
[1] 5492
sep@pc305-3:~> java -Djava.rmi.server.codebase=file:///home/panitz/
↳ name.panitz.rmi.RMI$DateServer &
[2] 5502
sep@pc305-3:~/fh/java/student/classes> Server ready

sep@pc305-3:~/fh/java/student/classes>
```

### 1.3 RMI Client

Unser Server ist jetzt bereit, Methodenaufrufe zu empfangen. Diese werden von einem Clientprogramm aus unternommen. Hierzu ist ähnlich wie im Serverprogramm zu verfahren. Für die RMI-Registry des Servers ist eine repräsentierende Instanz zu erzeugen, in der ist unter dem registrierten Namen nach dem angebotenen entfernten Objekt zu fragen. Schließlich kann dieses wie ein ganz normales Javaobjekt benutzt werden. Es ist dabei nur zu beachten, daß eventuell eine Ausnahme auftritt.

Für unser Beispiel schreiben wir eine kleine GUI-Anwendung. Der meiste Code beschäftigt sich dabei mit dem GUI. Die eigentlichen Aufrufe an das auf einer anderen Maschine laufende Objekt, bestehen aus nur drei Zeilen.

Java: RMI

```
static class DateClient extends JPanel {
    JButton getButton = new JButton("get new time from server");
    JLabel l = new JLabel();
    DateClient(String host){
        add(getButton);
        add(l);
        try {
            var registry = LocateRegistry.getRegistry(host);
            var stub = (GetDate) registry.lookup("date");
            l.setText(""+stub.getDate());

            getButton.addActionListener(e->{
                try {l.setText(""+stub.getDate());}catch (Exception e1){}
            });
        }catch (Exception e) {}
    }
}
```

## Java: RMI

```
public static void main(String[] args) {  
    JFrame f = new JFrame();  
    f.add(new DateClient(args[0]));  
    f.pack();  
    f.setVisible(true);  
}  
}
```

### 1.4 Serialisierbarkeit

Um entfernte Methode aufrufen zu können, müssen die Argumente des Methodenauf-rufs über das Netz evrschickt werden und ebenso das Ergebnis zurückgeschickt werden. Hierzu müssen sich die entsprechenden Daten in eine für das verschicken geeignete Art umwandeln lassen, sie müssen serialisierbar sein. Java kennt das Konzept der Serialisier-barkeit und drückt es über die Schnittstelle `Serializable` aus. Die einige Einschränkung für RMI besteht also in der Serialisierbarkeit der in der methodensignatur beteiligten Typen.

## 2 Verteilte Spielanwendung

Da im letzten Abschnitt sich das Schreiben verteilter Anwendungen in Java als so einfach erwiesen hat, wollen wir in diesem Abschnitt eine weitere kleine verteilte Anwendung schreiben. Dieses Mal sollen die entfernten Methoden auch Parameter bekommen. Damit können Clients dem Server Daten schicken, die anderen Clients dann auch vom Server erfragen können. Somit können die Clients sich gegenseitig Nachrichten zukommen lassen, die Grundlage zum Beispiel eines jedem Chatprogramms.

Wir wollen aber keine Chatprogramm entwickeln, sondern eine kleine Anwendung, in der die Clients einen Punkt auf einer zweidimensionalen Fläche plazieren und bewegen können. Die einzelnen Clients sehen dabei nicht nur ihre eigenen Punkte sondern auch die Punkte der anderen Clients.

Die entsprechende `Remote`-Schnittstelle kommt mit drei Methoden aus:

- eine zum Erzeugen eines neuen Balls, wobei jeder Ball durch einen Namen identi-fiziert wird.
- eine zum Bewegen eines Balles mit einem bestimmten Namen.
- und eine zum Erfragen der Position eines jeden Balls, der existiert.

Wir bekommen die folgende Schnittstelle:

#### Java: RMI

```
static interface BallGame extends Remote {  
    void createBall(String name) throws RemoteException;  
    void move(String name,int x,int y) throws RemoteException;  
    Map<String,Dimension> getBalls()throws RemoteException;  
}
```

Für die x-, y-position der Bälle mißbrauchen wir die Klasse Dimension, die eine 2-dimensionale Position speichern kann, auch wenn die entsprechenden Felder width und height statt x und y heißen.

Auch die Implementierung dieser Schnittstelle läßt sich relativ trivial bewerkstelligen. Ein privates Feld speichert alle Namens- Dimensionspaare in einer Abbildung. Die Methoden createBall und move manipulieren diese Abbildung.

#### Java: RMI

```
static class BallGameImpl implements BallGame {  
    private Map<String,Dimension> bs  
        = new HashMap<String,Dimension>();  
  
    public void createBall(String name){  
        bs.put(name,new Dimension(10,10));  
    }  
  
    public void move(String name,int x,int y){  
        Dimension d=bs.get(name);  
        if (d!=null) d.setSize(d.width+x,d.height+y);  
    }  
  
    public Map<String,Dimension> getBalls(){return bs;}  
}
```

Der Server für diese Anwendung sieht exakt aus, wie der Server für die Datumsanwendung zuvor.

#### Java: RMI

```
static class BallGameServer {
    public static void main(String args[]) {
        try {
            var obj = new BallGameImpl();
            var stub = (BallGame) UnicastRemoteObject.exportObject(obj, 0);
            LocateRegistry.getRegistry().bind("ballgame", stub);
            System.err.println("Server ready");
        } catch (Exception e) {
            System.err.println("Server exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

## 2.1 Spiel-GUI

Es bleibt das spannendste: der Client. Auch in diesem Beispiel wieder eine GUI-Anwendung.

### 2.1.1 Animation

#### Java: RMI

```
static interface Animation {
    public void move();
}
```

#### Java: RMI

```
static abstract class AnimatedJPanel extends JPanel implements Animation{
```

Um zeitgesteuert das Szenario der Animation zu verändern, brauchen wir einen Timer.

#### Java: RMI

```
javax.swing.Timer t;
```

Im Konstruktor wird dieser initialisiert. Als Ereignisbehandlung wird ein Ereignisbehandlungsobjekt erzeugt, das die Methode `move` aufruft, also dafür sorgt, daß die Szenerie sich weiterbewegt und das dafür sorgt, daß die Szenerie neu gezeichnet wird. Wir starten diesen Timer gleich.

#### Java: RMI

```
public AnimatedJPanel(){
    super(true);
    t = new javax.swing.Timer(29, ev->{
        move();
        repaint();
    });
    t.start();
}
```

### 2.1.2 Ballspielclient

#### Java: RMI

```
static class BallGameClient extends AnimatedJPanel {
    final String name;
    BallGame game=null;
    final int width = 400;
    final int height = 300;
```

#### Java: RMI

```
BallGameClient(String host,String n){
    this.name=n;
    setFocusable(true);
    try {
        Registry registry = LocateRegistry.getRegistry(host);
        game = (BallGame) registry.lookup("ballgame");
    } catch (Exception e) {}
```

#### Java: RMI

```
addMouseListener(new MouseAdapter(){
    public void mouseClicked(MouseEvent e){
        try{game.createBall(name);}catch(Exception e1){}
    }
});
```



#### Java: RMI

```
addKeyListener(new KeyAdapter(){
    public void keyReleased(KeyEvent e){
        try{
            switch (e.getKeyCode()){
                case VK_DOWN: game.move(name,0,2);break;
                case VK_LEFT:game.move(name,-2,0);break;
                case VK_RIGHT:game.move(name,2,0);break;
                case VK_UP:game.move(name,0,-2);break;
            }
        }catch(Exception e1){System.out.println(e);}
    }
});
}
```

#### Java: RMI

```
public void paintComponent(Graphics g){
    g.setColor(Color.WHITE);
    g.fillRect(0,0,width,height);
    try{
        for (Dimension d:game.getBalls().values()){
            g.setColor(Color.RED);
            g.fillOval(d.height,d.width,10,10);
        }
    }catch (Exception e){}
}
```

#### Java: RMI

```
public Dimension getPreferredSize(){
    return new Dimension(width,height);}
}
```

#### Java: RMI

```
public static void main(String[] args) {
    JFrame f = new JFrame();
    f.add(new BallGameClient(args[0],args[1]));
    f.pack();
    f.setVisible(true);
}
public void move(){}
}
```

### 3 Aufgaben

**Aufgabe 1** Schreiben Sie eine verteilte Spielanwendung

Java: RMI

```
}
```