
Laborprotokoll

Verteilte Objekte mit RMI

Systemtechnik Labor
4CHITT 2015/16, Gruppe A

Gabriel Frassl

Version 1.0

Note:

Betreuer: M.Borko

Begonnen am 19. Februar 2016

Begonnen am 25. Februar 2016

Inhalt

1 Einführung	3
1.1 Ziele	3
1.2 Voraussetzungen	3
1.3 Aufgabenstellung.....	3
2 Quellen	10
2 Ergebnisse	4
2.1 Theorie	4
2.2 Policy Files erstellen.....	4
2.3 Java-RMI Tutorial	5
2.4 Beispiel 2	7
Middleware	8

1 Einführung

Verteilte Objekte haben bestimmte Grunderfordernisse, die mittels implementierten Middlewares leicht verwendet werden können. Das Verständnis hinter diesen Mechanismen ist aber notwendig, um funktionale Anforderungen entsprechend sicher und stabil implementieren zu können.

1.1 Ziele

Diese Übung gibt eine einfache Einführung in die Verwendung von verteilten Objekten mittels Java RMI. Es wird speziell Augenmerk auf die Referenzverwaltung sowie Serialisierung von Objekten gelegt. Es soll dabei eine einfache verteilte Applikation in Java implementiert werden.

1.2 Voraussetzungen

- Grundlagen Java und Software-Tests
- Grundlagen zu verteilten Systemen und Netzwerkverbindungen
- Grundlegendes Verständnis von nebenläufigen Prozessen

1.3 Aufgabenstellung

Folgen Sie dem offiziellen Java-RMI Tutorial, um eine einfache Implementierung des PI-Calculators zu realisieren. Beachten Sie dabei die notwendigen Schritte der Sicherheitseinstellungen (SecurityManager) sowie die Verwendung des RemoteInterfaces und der RemoteException.

Implementieren Sie ein Command-Pattern [2] mittels RMI und übertragen Sie die Aufgaben/Berechnungen an den Server. Sie können am Client entscheiden, welche Aufgaben der Server übernehmen soll. Die Erweiterung dieser Aufgabe wäre ein Callback-Interface auf der Client-Seite, die nach Beendigung der Aufgabe eine entsprechende Rückmeldung an den Client zurück senden soll. Somit hat der Client auch ein RemoteObject, welches aber nicht in der Registry eingetragen wird sondern beim Aufruf mittels Referenz an den Server übergeben wird.

2 Ergebnisse

2.1 Theorie

RMI- Remote Method Invokation stellt Methoden zur Verfügung um Objekte eines Programmes für andere Programme zur Verfügung zu stellen beziehungsweise darauf zuzugreifen, und ermöglicht so also das Verteilen von Objekten.

2.2 Policy Files erstellen

Da in den beiden folgenden Beispielen die Client und Serverprogramme mit installierten Security Managern arbeiten sollten im Vorhinein die benötigten policyfiles erstellt werden. Diese sollten so eingestellt sein das die lokalen Klassen eines Programmes, also am Client die Clientklassen und am Server die Serverklassen, alle Rechte erhalten.

Für Klassen einer anderen Lokation werden keine Rechte verliehen. Diese Einstellungen sind notwendig um zu verhindern dass der Client schädliche Software auf dem Server ausführt.

policy für Server

```
grant codeBase "file:/home/ann/src/" {  
    permission java.security.AllPermission;  
};
```

policy für Client

```
grant codeBase "file:/home/jones/src/" {  
    permission java.security.AllPermission;  
};
```

2.3 Java-RMI Tutorial

Ziel des Tutorials ist es eine Serveranwendung zu erstellen die ein Objekt mit einer einfachen Methode **executeTask** als Remote Objekt zur Verfügung stellt. Nun soll ebenfalls ein Clientprogramm geschrieben werden welches dem Server einen Task übergibt und sich somit von diesem PI berechnen lässt.

Middleware

Zuerst müssen zwei Interfaces erstellt werden die die Kommunikation zwischen Client und Server ermöglichen. Diese beiden Interfaces müssen sowohl auf Client als auch auf Server vorhanden sein.

Als erstes wird das Interface erstellt welches im Server Objekt implementiert wird.

```
package compute;

import java.rmi.Remote;

public interface Compute extends Remote {
    <T> T executeTask(Task<T> t) throws RemoteException;
}
```

Durch das **extend** von Remote kann das Interface später als Remote Objekt zur Verfügung gestellt werden. Der Client kann dann die darin enthaltene Methode **executeTask**(Inhalt später bei Serverseite) aufrufen. Als Parameter muss ein „Task“ übergeben werden. Das Interface dafür muss ebenfalls erstellt werden und sieht wie folgt aus.

```
package compute;

public interface Task<T> {
    T execute();
}
```

Sie enthält nur eine Methode execute welche wir später beim Erstellen eines Tasks benötigen.

Clientseite

Auf Clientseite benötigt man zunächst eine Klasse die das Interface Task implementiert. In unserem Fall die Klasse Pi

```
public class Pi implements Task<BigDecimal>, Serializable {
```

Die Klasse enthält nun mehrere Methoden zur Berechnung von Pi (siehe Code) und eben auch die eine execute Methode vom Interface Task.

```
public BigDecimal execute() {
    return computePi(digits);
}
```

Als nächstes muss noch die Klasse mit der Main Methode auf Clientseite **ComputePi** erstellt werden.

```
public class ComputePi {
    public static void main(String args[]) {
        //if (System.getSecurityManager() == null) {
        //    System.setSecurityManager(new SecurityManager());
        //}
        try {
            String name = "Compute";
            Registry registry = LocateRegistry.getRegistry(args[0]);
            Compute comp = (Compute) registry.lookup(name);
            Pi task = new Pi(Integer.parseInt(args[1]));
            BigDecimal pi = comp.executeTask(task);
            System.out.println(pi);
        } catch (Exception e) {
            System.err.println("ComputePi exception:");
            e.printStackTrace();
        }
    }
}
```

Wie im Code ersichtlich wird zuerst die Registry mit einem CLI Argument gesucht. Daraufhin wird in der Registry nach dem verteilten Objekt mit dem Namen „Compute“ gesucht und dieses als eigenes Objekt comp gespeichert. Es wird nun comp.execute mit einer Instanz der Klasse Pi aufgerufen und die daraus zurückgegebene Zahl gespeichert und ausgegeben.

Serverseite

Auf Serverseite ist nun die Klasse enthalten die das Interface Compute implementiert. Gleichzeitig beinhaltet sie auch die Main Methode die das Remote Objekt in der Registry zur Verfügung stellt.

```
public <T> T executeTask(Task<T> t) {
    return t.execute();
}
```

In der executeTask Methode wird einfach die execute Methode des übergebenen Tasks aufgerufen und zurückgegeben.

```

public static void main(String[] args) {
    //if (System.getSecurityManager() == null) {
    //    System.setSecurityManager(new SecurityManager());
    //}
    try {
        String name = "Compute";
        Compute engine = new ComputeEngine();
        Compute stub =
            (Compute) UnicastRemoteObject.exportObject(engine, 0);
        Registry registry = LocateRegistry.getRegistry();
        registry.rebind(name, stub);
        System.out.println("ComputeEngine bound");
    } catch (Exception e) {
        System.err.println("ComputeEngine exception:");
        e.printStackTrace();
    }
}

```

In der Main Methode wird ein stub erstellt welcher dann für den Client zur Verfügung gestellt wird.

Testen

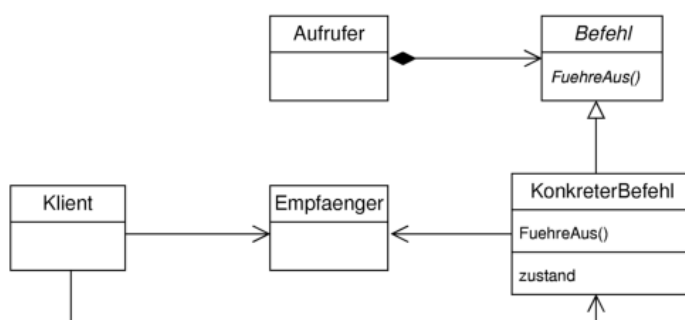
2.4 Beispiel 2

Nun wird die gerade erarbeitete Aufgabe wie folgt überarbeitet

- Sie wird mit einem Command Pattern gelöst.
- Die Registry wird vom Programm selber erstellt.

Command Pattern

Das Command Pattern ist ein Design Pattern welches den Auslöser und Ausführenden entkoppelt. Somit wird ein leichtes Weitergeben von Command Objekten ermöglicht.



Middleware

Auch hier sind wieder zwei Interfaces nötig welche sowohl auf dem Server als auch auf dem Client vorhanden sein müssen.

Zuerst wieder das Interface welches später für das verteilte Objekt verwendet wird. Dieses funktioniert genau gleich wie beim Tutorialbeispiel.

```
public interface DoSomethingService extends Remote {

    public void doSomething(Command c) throws RemoteException;

}
```

Wie man sieht wird für die doSomething Methode ein Command übergeben. Command ist das Basisinterface für alle spezifischen Commands(Command Pattern).

```
public interface Command extends Serializable {





    public void execute();

}
```

Aus diesem Interface können auf Client Seite leicht spezifische Commands erstellt werden.

Client Seite

Als erstes empfiehlt es sich spezifische Commands zu erstellen die später am Server ausgeführt werden sollen.

- ▷  CalculationCommand.java
- ▷  Command.java
- ▷  LoginCommand.java
- ▷  RegisterCommand.java



Für die Aufgabe sollte die Berechnung von Pi implementiert werden, deswegen erstellen wir CalculationCommand.

```
public class CalculationCommand implements Command, Serializable {

    private static final long serialVersionUID = 3202369269194172790L;
    private PISCalc calc;

    /**
     * ruft die berechnungsmethoden von PISCalc auf
     */
    @Override
    public void execute() {
        System.out.println("CalculationCommand called!");
        calc = new PISCalc();
        calc.calculate();
        calc.getResult();
    }

}
```

-  ComputePi.java
-  Pi.java

Für die Berechnung von Pi wurde ein eigenes Interface und eine Klasse erstellt. Die Funktionsweise ist allerdings die selbe wie im vorherigen Beispiel(siehe Code).

Als nächstes muss natürlich noch die Hauptklasse erstellt werden welches das Remoteobjekt anspricht und diesem die Commands welche Auszuführen sind zuschickt.

```
public class Client {

    /**
     * anfragen der Registry, erstellen einer "Verbindung" zum Remote Objekt und ausführen verschiedener Comr
     * @param args cli argumente - anzahl der Stellen von Pi
     */
    public static void main(String[] args) {
        if(System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
        try {
            Registry registry = LocateRegistry.getRegistry(1233);

            DoSomethingService uRemoteObject = (DoSomethingService) registry.lookup("Service");
            System.out.println("Service found");

            Command rc = new RegisterCommand();
            Command lc = new LoginCommand();
            Command cc = new CalculationCommand();
            uRemoteObject.doSomething(rc);
            uRemoteObject.doSomething(lc);
            uRemoteObject.doSomething(cc);

        } catch (RemoteException re) {
```

Wie man an diesem Beispiel sieht ist es nun extrem leicht mehrere spezifische Commands zu erstellen und diese dem Server weiter zu geben.

Server Seite

Auf Serverseite unterscheidet sich dieses Beispiel insofern von dem Tutorial, dass die Klasse welche das Interface DoSomethingService implementiert und die Klasse welche dieses zur Verfügung stellt(Main) getrennt sind.

```
public class ServerService implements DoSomethingService {

    @Override
    public void doSomething(Command c) throws RemoteException {
        c.execute();
    }

}
```

Hier die Klasse welche DoSomethingService implementiert und angibt dass die Methode execute des übergebenen Commands aufgerufen werden soll.

Hier sieht man den Ausschnitt aus der Serverklasse in welcher ein Objekt der gerade erstellten Klasse Server Service erstellt wurde und dieses in der Registry als Remote Objekt zur Verfügung gestellt wird.

```
public static void main(String[] args) {
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new SecurityManager());
    }
    try {
        ServerService uRemoteObject = new ServerService();
        DoSomethingService stub = (DoSomethingService) UnicastRemoteObject.exportObject(uRemoteObject, 0);
        Registry registry = LocateRegistry.createRegistry(1233);
        registry.rebind("Service", stub);
        System.out.println("Service bound! Press Enter to terminate ...");

        while ( System.in.read() != '\n' );
        UnicastRemoteObject.unexportObject(uRemoteObject, true);

        System.out.println("Service unbound, System goes down ...");
    }
}
```

Des Weiteren wird auf eine Usereingabe gehört die den Service „unexportet“ und dieser somit nicht mehr erreichbar ist.

Test

-Service zur Verfügung stellen
Klasse Server starten ->

Service bound! Press Enter to terminate ...

-Client Programm ausführen
Klasse Client starten ->

```
RegisterCommand called!
LoginCommand called!
CalculationCommand called!
Pi : 3.14159
```

Hier sieht man die Ergebnisse der Commands welche am Server ausgeführt wurden.

3 Quellen

- [1] "The Java Tutorials - Trail RMI"; online: <http://docs.oracle.com/javase/tutorial/rmi/>
- [2] "Command Pattern"; Vince Huston; online: <http://vincehuston.org/dp/command.html>
- [3] "Beispiel Konstrukt für Command Pattern mit Java RMI"; Michael Borko; online: <https://github.com/mborko/code-examples/tree/master/java/rmiCommandPattern>

- [4] Github repository: <https://github.com/gfrassl-tgm/DezSys04-RMI.git>

4 Zeitaufzeichnung

Übung	Zeitaufwand geschätzt	Zeitaufwand real	Datum/Ort
Tutorial	180 min	300 min	19.02.16-Labor 24.02.16
RMI CommandPattern	200 min	240 min	Zuhause 25.02.2016
Protokollieren	120 min	140 min	Zuhause 23.02.2016
Gesamt	400min	680min	