Laborprotokoll

RMI

Systemtechnik Labor
4BHIT 2015/16, Gruppe X

Colakovic Zeljko

Version 1.0

Note:
Betreuer: M.BORKO

Begonnen am 22. April 2016
Beendet am 28. April 2016

Inhaltsverzeichnis

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.

1.4 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

2 Ergebnisse

Folgende Schritte wurden auf einem Rechner mit Windows 10 durchgeführt.

Der vollständige Code ist auf github unter folgendem Link vorzufinden: *https://github.com/zcolakovic-tgm/SYT.git*

2.1 RMI-Tutorial

2.1.1Durchführung der Policy

Die Durchführung des Tutorials dient zur Vertiefung von RMI (Remote Method Invocation) in Java. Zu Anfang wird der Code runtergeladen *(siehe Punkt 2.1.2)* und in <*Eclipse*> eingebunden. Um den Code verwenden zu können müssen die Rechte in *java.policy* gewährt werden. Das File befindet sich im Installationsordner *Java (C:\Program Files\Java\jre1.8.0_60\lib\security\).* Folgende Code-Zeile muss geändert werden *grant codeBase*. Hier wird nach folgendem Schema sein der Ordnerangeben für den die Rechte freigegeben werden soll. Auf dem Folgenden Bild werden die Rechte für alle Ordner die in *C:/Users/* liegen gewährt (/- bedeutet alles weitere was in diesem Ordner liegt).

```
1
2  // Standard extensions get all permissions by default
3
4  grant codeBase "file:/C:/Users/-" {
5          permission java.security.AllPermission;
6  };
```

Falls sich im Java-Ordner mehrere Java-Library befinden, kann man mit folgenden Schritten überprüfen welches von <*Eclipse*> verwendet wird.

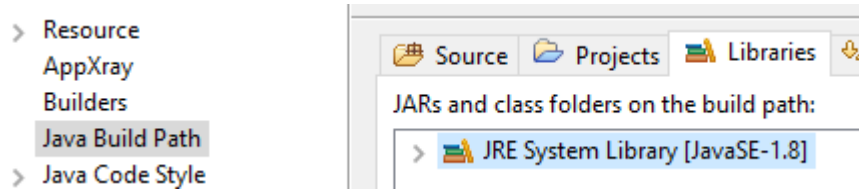
- Überprüfung der Java-Version durch den Befehl *java –version* in der Console

```
C:\Users\zeljk_000>java -version
java version "1.8.0_60"
Java(TM) SE Runtime Environment (build 1.8.0_60-b27)
Java HotSpot(TM) 64-Bit Server VM (build 25.60-b23, mixed mode)
```

- Überprüfung der Java-Version durch den Befehl *javac –version* in der Console

```
C:\Users\zeljk_000>javac -version
javac 1.8.0_60
```

- Überprüfung der Java-Version in <*Eclipse*> (Rechtsklick auf ein Projekt – Properties – Java Build Path – Libraries)




```
String name = "Compute";
// Remote-Obj
Registry registry = LocateRegistry.getRegistry(args[0]);
Compute comp = (Compute) registry.lookup(name);
// Pi.java aufrufen mit der Anzahl an Nachkommerstellen
Pi task = new Pi(Integer.parseInt(args[1]));
BigDecimal pi = comp.executeTask(task);
System.out.println(pi);
```

2.1.2Code

Der Code wurde von folgendem Link übernommen: *https://github.com/mborko/code-examples/tree/master/java/rmiTutorial* und ist auch auf *https://github.com/zeljkocolakovic/SYT.git* vorzufinden.

Es wurde aus dem Tutorial drei Packages übernommen *(client/compute/engine).*

2.1.2.1Package-client

Im Client Package befinden sich 2 Java-Klassen zu einem die *Pi.java* die dafür da ist um die Berechnungen durchzuführen und zum anderen *ComputePi.java* die die Klasse *Pi.java* aufruft und ein Remote-Obj kreiert.

2.1.2.2–Package-compute

Besteht aus 2 Java-Klassen *Compute.java* und *Task.java,* wobei es sich bei beiden um *<<inerface>>* handelt.

2.1.2.3Package-engine

Das Package *engine* beinhaltet den Server *ComputeEngine.java* der ein Remote-Obejkt entgegen nimmt und verarbeitet.

2.1.2.4Ergebnis

Zum starten des Programm muss zuerst die Klasse *ComputeEnige.java* und danach die Klasse *ComputePi.java.* Zu beachten ist, dass die Klasse *ComputePi.java* mit 2 Parameter durchzuführen ist (Server IP und Anzahl der Nachkommerstellen).

```
<terminated> ComputePi (1) |  ComputeEngine (1) [Java Application]
3.14159                       ComputeEngine bound
```

2.2 Erweiterungen

2.2.1Durchführung

Zu der Aufgabenstellung *Command-Pattern* ist unter folgendem Link ein Musterbeispiel vorzufinden: *https://github.com/mborko/code-examples.git*

2.2.2Code

2.2.2.1Package-calculation

Das Package *calculation* beinhaltet 2 Java-Klassen zum einen die Klasse *CalcEuler.java* und zum anderen die klasse *Calculation.java* wobei es sich jedoch nur um ein <<*interface*>> handelt. Die Klasse *CalcEuler.java* implementiert dieses <<*interface*>> mit den Methoden calculate und result.

```
public interface Calculation {

    /*
     * Startet den Prozess zum Berechnen
     *
     * @since 2016-04-26
     */
    void calculate();

    /*
     * Gibt das verarbeiterte Ergebnis zurück
     *
     * @since 2016-04-26
     */
    BigDecimal getResult();
```

2.2.2.2Package-callback

Das *PackageCallback* beinhaltet zwei Java-Klassen *Callback.java* ein <<*interface*>> und eine Klasse *CallbackCalc.java* die das Interface mit der Methode *execute()* implementiert. Es handelt sich dabei um eine einfache Methode die das errechnete Ergebnis zurückliefert.

2.2.2.3Package-client

Die im Package *Client.java* beinhaltet eine einfache Methode zur Verbindung mit dem Server.

```
// Register Obj
Registry registry = LocateRegistry.getRegistry(1234);

// Erstellt Service Obj
DoSomethingService uRemoteObject = (DoSomethingService) regist
System.out.println("Service found");

// Usereingabe
int numb = Integer.parseInt(args[1]);

// Generiert die Berechnung
Calculation calculation = new CalcEuler(numb);

//Generiert den Callback
Callback callback = new CallbackCalc();
Callback cuc = (Callback) UnicastRemoteObject.exportObject(callback, 0);

// Command vorbereiten
Command command = new CommandCalc(calculation, cuc);

uRemoteObject.doSomething(command);
```

2.2.2.4Package-remoteService

Die *DoSomethingService.java*, wobei es sich um ein *<<interface>>* handelt, sorgt dafür, dass der erhaltene *Command* ausgeführt wird.

2.2.2.5Package-server

Der Server sorgt für den Verbindungsaufbau in der Java-Klasse

Server.java

```
// Verbindungsobjekt
Service uRemoteObject = new Service();

// Stub das verbundene Obj
DoSomethingService stub = (DoSomethingService) UnicastRemoteObject.exportObject(uRemo
Registry registry = LocateRegistry.createRegistry(1234);

// Registriert den Service
registry.rebind("Service", stub);
System.out.println("Service bound! Press Enter to terminate ...");

// Schaltet sich ab sobald Enter eingelesen wird
while ( System.in.read() != '\n' );
    UnicastRemoteObject.unexportObject(uRemoteObject, true);

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

Die *Service.java* implementiert *DoSomethingService* ein Object und führt *execute()* aus.

2.2.2.6Package-server.commands

Bei der Klasse *Command.java* handelt sich wieder um ein *<<interface>>* mit der Methode *execute()*. In *CommandCalc.java* implementiert *Command.java* und überschreibt die *execute()* Methode in der sie die Berechnung aufruft. Zu dem sendet die Methode noch das sie aufgerufen wurde *(Callback)*.

```java
/*
 * Ruft den Callback auf
 *
 * @since 2016-04-26
 */
@Override
public void execute() {
    System.out.println("CalculationCommand called!");
    calculation.calculate();
    try {
        callback.execute(calculation.getResult());
    } catch (RemoteException e) {
        e.printStackTrace();
    }
}
```

2.2.2.7Ergebnis

Zum starten des Programm muss zuerst die Klasse *Server.java* und danach die Klasse *Clint.java.* Zu beachten ist, dass die Klasse *Client.java* mit 2 Parameter durchzuführen ist (Server IP und Anzahl der Nachkommerstellen).

```
Server [Java Application] C:\Program Files\Java\jre1.8.0_60\bin\javaw.
Service bound! Press Enter to terminate ...
CalculationCommand called!

<terminated> Client [Java Application]
Service found
2.708337
```

2.4 Zeitaufwand

Datum	Aufgabe	Zeit
Von 22.04.2016 Bis 28.04.2016	Tutorial	2 Stunden
Von 22.04.2016 Bis 28.04.2016	Erweiterungen Callback & Command Pattern	6 Stunden
Von 22.04.2016 Bis 28.04.2016	Protokoll	2 Stunden