

# ***Compute Engine***

**© 1999 by gruppe 2: gerhard engelbrecht (9725485)  
georg grossmann (9701270)  
wolfgang leitner (9273819)  
andreas razz (9425322)  
gerd balon (9550098)**

<b>Inhalt:</b>	<b>Seite</b>
<b>1. Aufgabenstellung</b>	<b>3</b>
<b>2. RMI (Remote Method Invocation)</b>	<b>3</b>
2.1. Was ist RMI ?	3
2.2. Java und RMI	3
2.3. Stubs und Skeletons	4
2.4. Technischer Ablauf	4
<b>3. Compute Engine Implementation</b>	<b>4</b>
3.1. Allgemeine Deklarationen	4
3.2. Die Serverseite	5
3.3. Die Clientseite	6
3.4. Der auszuführende Task	7
<b>4. Ausführung des Programms</b>	<b>8</b>
4.1. Übersetzung (Compiling)	8
4.2. Ausführung des Programms auf einem Rechner in zwei Prozessen	8
4.3. Ausführung des Programms auf einem Server und einem Client	9

## 1. Aufgabenstellung

Es soll eine "Compute Engine" implementiert werden. Die "Compute Engine" soll auf einem Server laufen, welche beliebige Tasks, welche von Clients über das Internet geschickt werden, ausführen soll. Die Implementation soll mit Hilfe des RMI-Mechanismus realisiert werden. Der Zweck einer "Compute Engine" ist, daß komplexe Rechengänge nicht lokal ausgeführt werden müssen und somit den lokalen Rechner nicht blockieren, sondern, daß sie auf einem leistungsstarken Server ausgeführt werden können. In unserem Beispiel liegt der Schwerpunkt auf der Implementation des RMI-Mechanismus.

## 2. RMI (Remote Method Invocation)

### 2.1. Was ist RMI ?

Die RMI (Remote Method Invocation) ist Java's Basismechanismus, um verteilte Applikationen zu entwickeln. Eine verteilte Applikation besteht aus mehreren Komponenten, die auf unterschiedlichen Rechnern ausgeführt werden und die miteinander kommunizieren. RMI realisiert auf hoher Abstraktionsebene die Kommunikation zwischen verteilten Objekten bzw. den Transfer von Objekten zwischen verschiedenen Komponenten einer verteilten Applikation.

Ein *remote object* ist ein Objekt, dessen Methoden von einer anderen JVM (Java Virtual Machine) - eventuell auf einem anderen Rechner - aufgerufen werden können. Damit man auf dieses Objekt auch zugreifen kann muß dieses Objekt mit einem *remote interface* ausgestattet sein. Diese Schnittstelle beschreibt die öffentlichen Eigenschaften (Methoden) des Objektes, damit ein Client weiß, wie auf das Objekt zugegriffen werden kann. Man könnte ein dieses Interface mit einem C++ Header-File einer Klasse vergleichen. Dieser Mechanismus erfüllt auch das Konzept der Kapselung.

Unter der RMI (Remote Method Invocation) versteht man den konkreten Aufruf einer Methode eines *remote interface* für ein *remote object*. Ein RMI ist syntaktische mit einem normalen (lokalen) Methodenaufruf ident.

### 2.2. Java und RMI

Damit ein Client eine RMI durchführen kann braucht man eine Verwaltung für alle von einem Server der Öffentlichkeit zur Verfügung gestellten ("remoten") Objekten. Diese Verwaltung wird durch die sog. *rmiregistry* zur Verfügung gestellt. Das *rmiregistry* muß in einem eigenen Prozeß laufen, damit ein Server Objekte Bereitstellen kann. Ein Client kann sich somit bei dieser *rmiregistry* "erkundigen", ob und wie er auf ein Objekt des Servers zugreifen kann. Der Client erhält von der *rmiregistry* des Servers unter der Verwendung eines bestimmten Namens eine Referenz auf ein *remote object*. Mit dieser

Referenz erfolgt die Kommunikation mit einem *remote object* und damit können auch Methodenaufrufe (RMIs) ausgeführt werden. Alle Details der Kommunikation sind im RMI Mechanismus versteckt und basieren auf einem URL-Protokoll (HTTP, FTP, etc.)

## 2.3. Stubs und Skeletons

Ein Stub ist eine lokale Repräsentation (proxy), die ein Client verwendet, um mit *remote objects* zu kommunizieren. Nachdem ein Client bei einem Server eine Referenz auf ein *remote object* erhalten hat, wird durch das RMI-System ein Stub geladen, mit dem die Methodenaufrufe in *remote calls* auf dem Server übersetzt werden. Der Stub übernimmt das Packen/Entpacken der Argumente (marshalling/unmarshalling) und sendet/empfängt die serialisierten Daten.

Auf der Serverseite wird der *remote call* vom RMI-System empfangen und mit einem Skeleton verbunden, welches das Entpacken/Packen der Argumente auf der Serverseite übernimmt. Stubs und Skeletons werden automatisch durch den RMI-Compiler (*rmic*) anhand der Server-Implementation des *remote objects* generiert.

## 3. Compute Engine Implementation

### 3.1. Allgemeine Deklarationen

Zunächst muß ein öffentliches Interface definiert werden, in welchem ein Objekt und dessen Methoden deklariert werden. Die Implementation des Objekts wird auf der Serverseite durchgeführt. In unserem Beispiel wird ein Interface *Compute* deklariert, welches eine Methode zum Ausführen eines Tasks (*executeTask*) beinhaltet:

```
import java.rmi.Remote;
import java.rmi.RemoteExceptions;

public interface Compute extends Remote {
    Object executeTask (Task task) throws RemoteExceptions;
}
```

Der Interface wird in der Datei "Compute.java" gespeichert. Durch dieses öffentliche Interface wird festgehalten, daß das Objekt *Compute* ein *remote object* ist und daß beim Aufruf der Methode *executeTask* eventuelle Fehler (Exceptions) an die aufrufende Methode zurückgegeben werden.

Weiters muß das Objekt *Task* deklariert werden, wobei nur darauf zu achten ist, daß dieses Objekt serialisierbar ist und eine Methode hat, welche einen beliebigen Task ausführen kann. Das Interface des Objekts *Task* wird in der Datei "Task.java" gespeichert:

```
import java.io.Serializable;

public interface Task extends Serializable {
    Object execute();
}
```

Die Serialisierbarkeit des Objekts ist deshalb wichtig, weil bei der Kommunikation zwischen Server und Client die Daten in serialisierter Form

übertragen werden müssen. Der Task wird auf der Clientseite implementiert, soll aber – z.B. aus Performance-Gründen – auf einem Server ausgeführt werden. Dadurch muß die gesamte Implementation des Tasks übertragen werden. In Java wird dabei der Byte-Code eines Java-Class-Files übertragen.

### 3.2. Die Serverseite

Auf der Serverseite muß die Compute Engine vollständig implementiert werden. Sie soll ein Objekt *Compute* zur Verfügung stellen, in welchem auf der Clientseite eine Methode zum Aufruf einer Task-Abarbeitung verwendet werden kann. Die Klasse *ComputeEngine* implementiert das Interface *Compute*, wobei hier nur ein allgemeiner Konstruktor und die Methode *executeTask* implementiert werden muß. Weiters enthält die Klasse *ComputeEngine* die Methode *main*, in der die Compute Engine "gestartet" wird. Als erstes wird ein Security Manager gestartet, welcher Sicherheitsaspekte im Zusammenhang mit RMI behandelt. Ohne einen Security Manager können Stubs und class-Files nur vom lokalen Klassenpfad geladen werden.

Danach wird versucht die Instanz *server* von der Klasse *ComputeEngine* mit dem Namen des Hosts – auf welchem die Compute Engine laufen soll – zu binden. Der Host kann eine IP-Adresse eines Gerätes sein, oder der Hostname (z.B. ebola.ifs.univie.ac.at/~b9725485/) sein. Wenn der Serverprozeß lokal läuft, dann wird *localhost* eingefügt. Sollte beim "Bindeversuch" ein Fehler auftreten wird dieser in der catch-Klausel ausgegeben. Der Sourcecode dazu wird in der Datei "ComputeEngine.java" gespeichert:

```
import java.rmi.*;
import java.rmi.server.*;

public class ComputeEngine extends UnicastRemoteObject
implements Compute {

    public ComputeEngine() throws RemoteException {
        super();
    }

    public Object executeTask(Task t) {
        return t.execute();
    }

    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager
                (new RMISecurityManager());
        }
        String name = "//localhost/Compute";
        try {
            ComputeEngine server = new ComputeEngine();
            Naming.rebind(name, server);
            System.out.println("ComputeEngineregistered");
        }
        catch (Exception e) {
            System.err.println("Leider trat ein Fehler bei
```

```
}  
  }
```

### 3.3. Die Clientseite

Auf der Clientseite muß eine Klasse *TuWas* implementiert werden, welche eine Methode *main* enthält. In dieser Methode wird zuerst der Security Manager gestartet, welcher wie auf der Serverseite die Sicherheitsaspekte in Zusammenhang mit RMI behandelt. Danach wird versucht mit Hilfe von *Naming.lookup* das Objekt *Compute* auf dem Server zu "finden". Dieses *remote object* wird in der Instanz von *Compute* mit dem Namen *comp* "gespeichert". Man kann nun auf *comp* (ein *remote object*) zugreifen, wie wenn dieses Objekt lokal zur Verfügung stünde.

Danach wird eine Instanz von Rechnung mit dem Namen *task* angelegt. Rechnung ist eine eigene Klasse, welche die Implementation von Task darstellt (siehe 3.4.). Danach wird in einem StringBuffer *res* der Rückgabewert von *comp.executeTask* gesichert. Der auszuführende Task wird dabei an die remote method *executeTask* übergeben. Am Ende wird das in *res* gesicherte Ergebnis der Execution ausgegeben. Sollten in diesem Teil des Programms Fehler auftreten, so werden diese in einer catch-Klausel abgefangen und es wird eine entsprechende Fehlermeldung ausgegeben. Der Sourcecode wird in der Datei "TuWas.java" gespeichert:

```
import java.rmi.*;

public class TuWas {

    public static void main(String args[]) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager
                (new RMISecurityManager());
        }

        try {
            String name = "//localhost/ComputeEngine";
            Compute comp = (Compute) Naming.lookup(name);
            Rechnung task = new Rechnung
                (Integer.parseInt(args[0]),
                 Integer.parseInt(args[1]));
            StringBuffer res =
                (StringBuffer) (comp.executeTask(task));
            System.out.println(res);
        }

        catch (Exception e) {
            System.err.println("Leider trat eine Exception
```

}

### 3.4. Der auszuführende Task

Die Klasse Rechnung ist die Implementation von Task. In dieser Klasse muß eine Methode *execute* implementiert werden, welche in diesem Beispiel eine einfache Methode *computeOperation* durchführt. In dieser Methode wird die einfache Addition durchgeführt und in einem StringBuffer zurückgegeben. Der Sourcecode wird in der Datei "Rechnen.java" gespeichert:

```
public class Rechnung implements Task {

    private int input1, input2;

    public Rechnung(int z1, int z2) {
        input1 = z1;
        input2 = z2;
    }

    public Object execute() {
        return computeRechnung(input1, input2);
    }

    public static StringBuffer computeRechnung(int x1, int x2)
    {
        int e = x1 + x2;
        StringBuffer erg_str =
            new StringBuffer("Rechnung: ");
        erg_str.append(x1);
        erg_str.append(" + ");
        erg_str.append(x2);
        erg_str.append(" = ");
        erg_str.append(e);
        return erg_str;
    }
}
```

In diesem Beispiel wird der Task in ganz einfacher Art und Weise implementiert. Man könnte bei der Implementation dieses Tasks auch äußerst umfangreich Berechnung (z.B. Wahlhochrechnungen, Wettervorhersagen, wissenschaftliche Auswertungen, statistische Analysen, Simulationen usw.) durchführen. Der Sourcecode zu dieser Implementation von Task kann lokal geschrieben und übersetzt (compiliert) werden. Bei der Abarbeitung auf dem Server wird das class-File vollständig zum Server übertragen und dort ausgeführt. Bei der Übertragung über das Internet wird ein URL-Protokoll (HTTP, FTP) verwendet, welches beim Starten des Servers und des Clients in einem Parameter (siehe 4.3.) angegeben werden muß. Wenn man die Compute Engine lokal auf einem Rechner in zwei verschiedenen Prozessen (Sessions, Tasks) startet, kann man hier direkt auf die Files zugreifen, da beide Prozesse im gleichen Filesystem arbeiten (siehe 4.2.).



## 4. Ausführung des Programms

### 4.1. Übersetzung (Compiling)

Bei der Übersetzung der Sourcecodefiles in den plattformunabhängigen Java-Bytecode muß man mit Hilfe des Java-Compilers (*javac*) alle Java-Files behandeln. Auf der Serverseite werden mit "*javac ComputeEngine.java*" class-Files vom den Interfaces *Task* und *Compute* erzeugt, sowie vom "Hauptprogramm" *ComputeEngine*. Danach muß man mit Hilfe des RMI-Compilers (*rmic*) den Stub und den Skeleton der Compute Engine generieren: "*rmic ComputeEngine*". Stub und Skeleton werden zur Kommunikation zwischen Server und Client benötigt (siehe 2.3.). Danach sollten sich folgende Files (bei unserem Beispiel) auf dem Server (bzw. im Serververzeichnis) befinden:

<i>Compute.java</i>	<i>Compute.class</i>
<i>Task.java</i>	<i>Task.class</i>
<i>ComputeEngine.java</i>	<i>ComputeEngine.class</i>
<i>ComputeEngine_Stub.class</i>	<i>ComputeEngine_Skel.class</i>

Auf der Clientseite müssen nur alle Java-Files mit dem Java-Compiler übersetzt werden: "*javac \*.java*". Danach sollten sich folgende Files (bei unserem Beispiel) auf dem Client (bzw. im Clientverzeichnis) befinden:

<i>Compute.java</i>	<i>Compute.class</i>
<i>Task.java</i>	<i>Task.class</i>
<i>Rechnung.java</i>	<i>Rechnung.class</i>
<i>TuWas.java</i>	<i>TuWas.class</i>

### 4.2. Ausführung des Programms auf einem Rechner in zwei Prozessen

Als erstes muß in die *rmiregistry* auf dem Server gestartet werden. Dabei ist zu beachten, daß der Start der Registry in einer Umgebung durchgeführt werden muß, welche keinen Zugriff auf die class-Files des Servers hat. D.h. es darf weder ein die Umgebungsvariable *CLASSPATH* gesetzt sein, noch darf der Start in dem Verzeichnis erfolgen in dem die class-Files sind. Insbesondere geht es dabei um das Stub und das Skeleton-class-File, weil wenn die *rmiregistry* auf diese Files zugreifen kann, dann wird das Stub-File von einem Client nicht gefunden (selbst wenn der Parameter *codebase* auf den später noch eingegangen wird) gesetzt ist.

Im ersten Prozeß soll nun der Server gestartet werden. Zuerst sollte man die Umgebungsvariable *CLASSPATH* auf "keinen" Pfad setzen (*unset*) In weiterer Folge bezeichnen wir dieses Verzeichnis nur noch mit "*c:\Bsp\Server*" und das Clientverzeichnis mit "*c:\Bsp\Client*". Um Sicherheitsaspekte mit Hilfe RMI-Security-Managers zu behandeln benötigt man im Serververzeichnis ein Policy-File. Hier einfach "*policy*" genannt:

```
grant {
    permission java.security.AllPermission;
};
```

In diesem einfachen Policy-File werden alle "Permissions" behandelt. Alternative Policy-Files findet man im Java-Tutorial um Sicherheitsaspekte im Detail zu behandeln.

Der Aufruf des Servers erfolgt im Serververzeichnis: `"java -Djava.rmi.server.codebase=file:/c:\Bsp\Server/ -Djava.security.policy=policy ComputeEngine"` Mit Hilfe des codebase-Parameters wird angegeben, wo sich im Filesystem die Server-class-Files (insbesondere Stub und Skelton) befinden. Außerdem wird das Policy-File in einem Parameter angegeben. Wenn der Server gestartet worden ist, sollte eine Ausgabe: `"ComputeEngine registered ..."` erfolgen.

Auf der Clientseite muß das Programm *TuWas* gestartet werden. Auch hier müssen die gleichen Parameter angegeben werden, wie auf der Serverseite. In diesem Fall gibt es in beide Richtungen einen Datenaustausch, weil ein bestimmter Task vom Client zum Server übertragen werden muß, und vom Server ein *remote object* benützt wird. Wenn (wie z.B. beim RMI-"Hello World" Beispiel) vom Client nur Methoden des *remote object* verwendet werden müssen hier keine derartigen Parameter angegeben werden, da der Server nichts über den Client wissen muß, da keine "Objekte" vom Client zum Server transferiert werden. Vor dem Aufruf des Clients muß im Clientprozeß die Umgebungsvariable CLASSPATH noch auf "keinen" Pfad gesetzt werden. Der Aufruf des Clients erfolgt im Clientverzeichnis: `"java -Djava.rmi.server.codebase=file:/c:\Bsp\Client/ -Djava.security.policy=policy TuWas 1 1"` Ein identes Policy-File muß auch im Clientverzeichnis befinden. Danach sollte die Ausgabe: `"1+1=2"` erfolgen.

### 4.3. Ausführung des Programms auf einem Server und einem Client

In weiterer Folge gehen wir davon aus, daß der Server und der Client jeweils auf einem Gerät im PC Labor des Hauptgebäudes (HS 27) laufen. Im Homedirectory eines jeden Users (SWE-Übungsteilnehmer) befindet sich ein Verzeichnis mit dem Namen `"public_html"`, welches mit dem HTTP-Protokoll, sofern die Zugriffsrechte stimmen, über das Internet unter der Adresse `"http://ebola.ifs.univie.ac.at/~bXXXXXXX/"` zugänglich ist. In unserem Beispiel gehen wir davon aus, daß ein User b9900001 den Serverteil in einem Verzeichnis `"/home/public_html/Bsp/Server"` auf einem Gerät mit dem Namen: `"hgXX.ifs.univie.ac.at"` (XX steht für die Gerätenummer) starten will und seine Compute Engine allgemein zugänglich machen will. Auf der anderen Seite gibt es einen User b9900002, welcher den Client bei sich starten will und die Compute Engine von User b9900001 benutzen will.

Bevor man das Programm auf zwei verschiedenen Rechnern in einem Netzwerk bzw. im Internet ausführen will, müssen im Sourcecode für den Server und im Client die Namen für einen bestimmten Host geändert werden. Will z.B. der User b9900001 auf dem Gerät Nummer 05 den Server starten, dann müssen sowohl Server als auch Client beim Hostnamen: `"hg05.ifs.univie.ac.at"` angegeben. Nach dieser Codeveränderung müssen beide User ihren Sourcecode wieder kompilieren (siehe 4.1.). Danach sollten beide User, die in 4.1. aufgelisteten Files plus das auch in 4.2. beschriebene idente Policy-File besitzen.

Zuerst muß der User b9900001 auf seiner Maschine eine *rmiregistry* starten. Beim Starten der Registry ist auch hier darauf zu achten, daß die Registry in einer Umgebung gestartet wird, in der sie keinen Zugriff auf die class-Files des Servers hat (siehe 4.2.), weil sonst der Stub auf dem Server vom Client nicht lokalisiert werden kann.

Der Start der Compute Engine im Serververzeichnis auf dem Server funktioniert wie folgt: *java -Djava.rmi.server.codebase=http://ebola.ifs.univie.ac.at/~b9900001/Bsp/Server/ -Djava.security.policy=policy ComputeEngine*

Hier wird beim codebase-Parameter ein URL-Protokoll angegeben und eine Internetadresse, wo sich die class-Files des Servers (insbesondere Stub!) befinden. Mit Hilfe des angegebenen Protokolls können die Daten nun über das Internet transferiert werden. Danach ist die Compute Engine in der *rmiregistry* des Servers registriert und kann von einem anderen Rechner benutzt werden.

Auf der Clientseite wird das Programm *TuWas* vom User b9900002 ausgeführt. Dieser ruft sein Programm im Clientverzeichnis wie folgt auf:

```
java -Djava.rmi.server.codebase  
=http://ebola.ifs.univie.ac.at/~b9900002/Bsp/Client/  
-Djava.security.policy=policy TuWas 1 1
```

Hier wird im codebase-Parameter die Internetadresse der Clientfiles angegeben, weil der Server, welcher einen Task des Clients ausführen soll, einen Zugriff auf das class-File *Rechnung* braucht. Würde man hier den codebase-Parameter nicht angeben, dann würde man eine Fehlermeldung, daß die Klasse *Rechnung* nicht gefunden wurde, bekommen. Diese Fehlermeldung wäre zwar bei der Ausführung des Clientteils zu sehen, allerdings wird diese vom Server verursacht, weil dieser keinen Zugriff auf diese Klasse hätte.

