Praktikum aus Softwareentwicklung 2, Stunde 8

Lehrziele/Inhalt

1. Remoting

Remoting

Über Remoting können Objekte über JavaVMs hinweg miteinander kommunizieren. Das ist auch mit Socket-Programmierung möglich. Aber Remoting abstrahiert die Kommunikation als Methoden-aufrufe, während Sockets nur Byteströme übertragen. Bis auf *RemoteException*s die am Rufer behandelt werden müssen, sind Methodenaufrufe über Remoting gleich mit lokalen Methodenaufrufen.

Remoting arbeitet mit dem Proxy-Muster, um von der Netzwerkkommunikation zu abstrahieren. Das bedeutetet: auf der Client-Seite ist ein Proxy (*Stub*) der die Methodenaufrufe entgegennimmt und über das Netz überträgt. Auf der Server-Seite ist ein Proxy (*Skeleton*) der die Anfragen vom Netz liest und den gewünschten Methodenaufruf auf dem echten Server-Objekt macht. Hat die Methode einen Rückgabewert, dann überträgt der Server-Proxy diesen zurück an den Client-Proxy. Der Client-Proxy nimmt den Rückgabewert vom Netz und gibt ihn an den Rufer zurück. Siehe Abbildung 4.

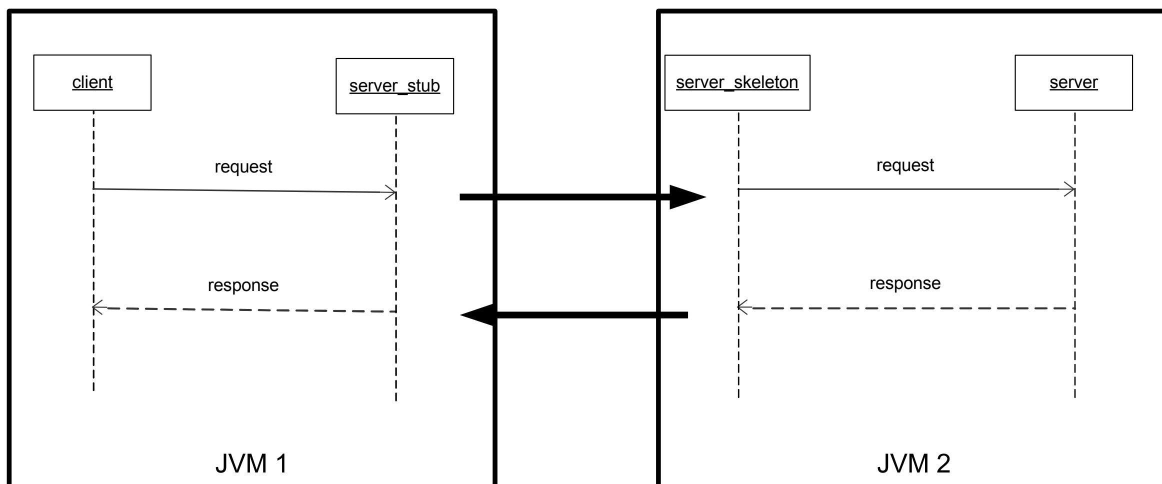

Abbildung 4) Remoting Kommunikation

Die Kommunikation zwischen den JavaVMs erfolgt über die Netzwerk-Schicht des Betriebssystems, auch wenn die VMs auf demselben Rechner laufen. Die Netzwerkverbindung läuft über TCP/IP, der Standarport ist 1099, als Kommunikationsprotokoll kann man zB: das Java Remote Method Protokol (JRMP) oder Internet Inter-ORB Protocol (IIOP) eingesetzten.

Damit man eine Methode aufrufen kann muss ein Empfänger-Objekt existieren. Das Empfänger-Objekt am Server muss also existieren solange Clients darauf zugreifen. Damit das Objekt erhalten bleibt, auch wenn es am Server keine Referenz im Programm mehr gibt, gibt es einen Remote Reference Layer der auf Server-Objekte verweist. Seit Java 2 übernimmt der Remote Reference Layer die Aufgabe vom Skeleton am Server. Abbildung 5 zeigt die Schichten der Kommunikation, inklusive Remote Reference Layer.

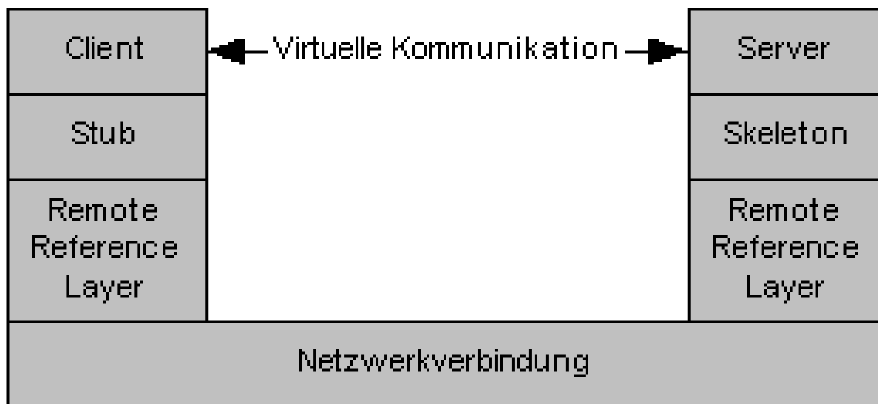

Abbildung 5) Schichten der Kommunikation bei Remoting

Objektregistrierung und Objektsuche

Eine Frage der Kommunikation zwischen zwei Objekten ist, wie finden sich die Objekte? In Java Remoting wird das über eine RMI-Registry gelöst. Server registrieren bei einer RMI-Registry die exportierten Remote-Objekte mit einem Namen; und Clients fragen über diese RMI-Registry Objekte mit dem Namen ab.

Als RMI-Registry kann man das Kommandozeilenwerkzeug *rmiregistry* benutzen oder programmatisch eine über *LocateRegistry.createRegistry* erstellen. Registrieren und suchen kann man Objekte über die Klasse *Naming*, mit den Methoden *bind* und *rebind* bzw. *lookup*.

Stub und Skeleton

Die Proxies für den Client und den Server kann man mit dem Kommandozeilenwerkzeug *rmic* erstellen. Gibt man den Parameter -*keep* an werden die erzeugten Klassen im Quellcode ausgegeben, sonst nur als class-Dateien.

Da ab Java 2 die Aufgabe des Skeletons in der Remote Reference Schicht implementiert ist erzeugt *rmic* nur Stubs. Braucht man Skeletons muss man den Parameter -*v1.1* angeben.

Nutzt man Klasse *UnicastRemoteObjekt* fällt auch die Notwendigkeit für einen Stub weg. Die Klasse *UnicastRemoteObjekt* exportiert ein Objekt und erzeugt die nötige Remote-Reference. Diese Klasse kann beerbt werden, dann wird das Objekt im Konstruktor exportiert; oder man nutzt die statische *Methode exportObject(Remote obj, int port)*, um ein beliebiges Remote-Objekt zu exportieren. Als *port* kann man *0* angeben, dann sucht Java selbst nach einem freien Port.

Parameter und Rückgabewerte

Sinnvoll werden Methodenaufrufe erst wenn man Parameter übergeben und Rückgabewerte erhalten kann. Bei Remoting werden alle serialisierbaren Datentypen als Parameter unterstützt. Das sind: die Basisdatentypen (int, boolean, double, …), Datentypen die das Interface *Serializable* implementieren und Remote-Objekte, diese implementieren das Interface *Remoting*. Für Remote-Objekte wird eine Remote-Referenz übertragen anstelle des eigentlichen Objekts. Somit können Methodenaufrufe an das echte Objekt weitergeleitet werden. Achtung, bei den direkt serialisierten Objekten wird auf der anderen Seite eine Kopie aufgebaut, Änderungen sind also lokal zu der ausführenden JavaVM. Objekte aller anderen Klassen können nicht verwendet werden.

Übergibt man Remote-Objekte an ein Server-Objekt, kann der Server Methodenaufrufe am Client machen. Die Rolle des Servers und des Clients vertauscht sich für diesen Aufruf. Das kann zum Beispiel genutzt werden um Remote-Listener am Server zu installieren.

Objektvergleich

Remote-Objekte haben eine andere Gleichheitssemantik als lokale Objekte. Der Remote Reference Layer legt jedes Mal wenn ein Objekt abgefragt wird ein neues Stub-Objekt an. Damit schlägt der Referenzvergleich (==) auf der Clientseite auch fehl, wenn es sich um dasselbe Objekt auf Serverseite handelt.

Will man feststellen ob zwei Objekt-Referenzen auf der Clientseite auf dasselbe Objekt der Serverseite zeigen muss man *equals* benutzen. Die Methode *equals* wird also in Remoting benutzt, um Referenzgleichheit am Server festzustellen. Braucht man eine Vergleichsmethode die Objektgleichheit (*equals*) am Server prüft, muss man sich eine eigene Remotemethode schreiben. Häufig wird dafür der Name *remoteEquals* benutzt.

Distributed Garbage Collection

Übergibt man ein Remote-Objekt an eine andere JavaVM, zB: als Listener oder als Arbeitsobjekt aus einer Factory, stellt sich die Frage wie lange man das reale Objekt am Leben erhalten muss. Sowohl Listener als auch Objekte die man aus einer Factory erzeugt, um sie einem Remote-Client zu übergeben speichert man Lokal nur selten. Der Garbage Collector würde diese Objekte also aufräumen. Damit die Objekte erhalten bleiben solange sie noch von Clients benötigt werden implementiert der Java Remote Reference Layer einen verteilten Garbage Collector.

Der verteilte Garbage Collector arbeitet mit Reference Counting und einer Lease Time. Erst wenn der Referenz-Zähler auf null ist werden Objekte freigegeben. Ein Client gilt eine Zeit (Lease Time, Standard 10 Minuten) lang als aktiv, innerhalb dieser Zeit muss er sich melden, um weiter als aktiv zu gelten. Das Erneuern der Lease übernimmt der Remote Reference Layer des Clients, der Programmierer ist davon unbehelligt.

Die Lease Time kann über das System-Property *java.rmi.dgc.leaseValue* verändert werden. Je kürzer die Zeit, umso schneller werden unbenutzte Objekte freigegeben, aber die Netz-Last steigt. Einen idealen Wert gibt es nicht, normalerweise kann man den Standardwert von 10 Minuten beibehalten, in Spezialfällen muss man sich die Netz-Infrastruktur und die Objekt-Last am Server ansehen und den Wert entsprechend verändern.

Nachladen von Klassen

Übergibt man Objekte als Parameter oder Rückgabewerte die auf der Gegenstelle unbekannt sind muss Code nachgeladen werden. Zum Beispiel kennt ein Client nur das Interface eines Remote-Objekts, der Stub ist aber eventuell nur am Server bekannt. Damit der Client dennoch mit den Objekten umgehen kann muss er die Klassen nachladen.

Remoting unterstützt nachladen von Klassen, dazu muss über das System-Property *java.rmi.server.codebase* eine Url angegeben werden; und ein *SecurityManager* installiert sein. Damit der Sicherheitsmanager den Zugriff auf die Codebase erlaubt muss über eine Policy-Datei Zugriff auf den Server erlaubt sein.

Beispiel:

```
public class MyClient {
  public static void main(String[] args) {
    System.setSecurityManager(new SecurityManager());
    System.setProperty("java.security.policy", "client.policy");
    ...
```

client.policy:

```
grant {
  permission java.net.SocketPermission "server-url:1024-65535", "connect";
  permission java.net.SocketPermission "server-url:80", "connect";
  permission java.net.SocketPermission "server-url:8080", "connect";
}
```

Aufteilung der Klassen

Die Klassen können wie folgt aufgeteilt werden:

- Server: Klassen die für die Ausführung des Servers erforderlich sind
- Download-Bereich: Klassen die vom Client nachgeladen werden sollen, inklusive aller Basisklassen und Interfaces.
- Client: Klassen die unmittelbar am Client benötigt werden; und die Policy-Datei die Zugriff auf den Download-Server erlaubt.

Der Download-Bereich kann ein Web-Server sein, aber auch ein Verzeichnis auf einem FTP-Server oder ein Verzeichnis auf dem lokalen Rechner.

Beispiel Remote Calculator

Am Beispiel eines verteilten Rechners soll veranschaulicht werden wie man mit Remoting einen verteilten dienst implementieren kann. Als erstes muss eine Schnittstelle definiert werden die der Remote-Service implementieren soll, in unserem Fall ein Rechner mit den Grundrechnungsarten:

```
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;
}
```

Jede Methode im Interface muss die eine *java.rmi.RemoteException* werfen können.

Dazu eine Implementierung der Schnittstelle:

```
public class CalculatorImpl implements Calculator {
  public long add(long a, long b) { return a + b; }
  public long sub(long a, long b) { return a - b; }
  public long mul(long a, long b) { return a * b; }
  public long div(long a, long b) { return a / b; }
}
```

Auf der Server-Seite kann hier keine Exception auftreten, also kann auf die Deklaration der *RemoteException* verzichtet werden. Der Client muss dennoch mit RemoteExceptions umgehen können, zB könnte der Server durch Netzwerkprobleme unerreichbar werden.

Und einen Server der die Implementierung exportiert:

```
public class CalculatorServer {
  public static void main(String args[]) throws RemoteException,
      MalformedURLException {
```

```java
    Calculator c = new CalculatorImpl();
    Remote calcStub = UnicastRemoteObject.exportObject(c, 0);
    Naming.rebind("rmi://localhost:1099/CalculatorService", calcStub);
  }
}
```

Dieser Server benutzt eine RMI-Registry am lokalen Rechner auf Port *1099* und exportiert den von *UnicastRemoteObject* generierten Stub unter dem Namen *CalculatorService*.

Bevor der Server gestartet werden kann, muss eine RMI-Registry am lokalen Rechner gestartet werden, zB über das Kommandozeilenwerkzeugt *rmiregistry*.

Abschließend noch ein Test-Client der den Remote-Calculator benutzt:

```java
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("MalformedURLException");
      System.out.println(murle);
    } catch (RemoteException re) {
      System.out.println("RemoteException");
      System.out.println(re);
    } catch (NotBoundException nbe) {
      System.out.println("NotBoundException");
      System.out.println(nbe);
    } catch (java.lang.ArithmeticException ae) {
      System.out.println("java.lang.ArithmeticException");
      System.out.println(ae);
    }
  }
}
```