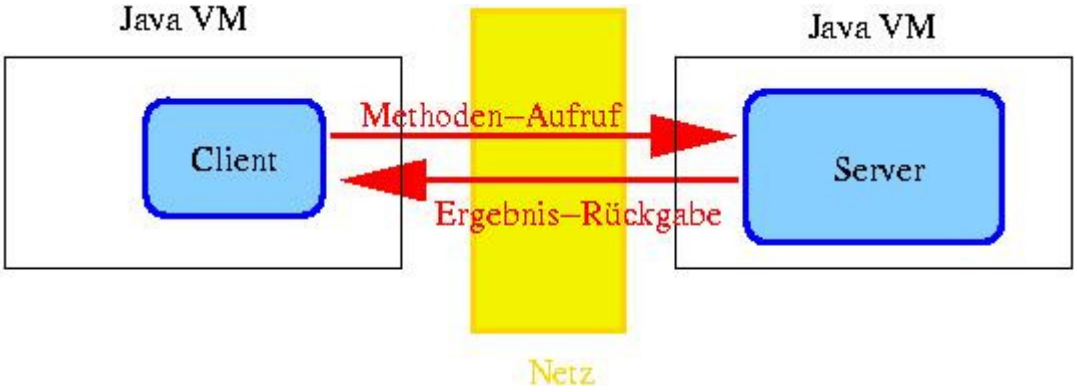
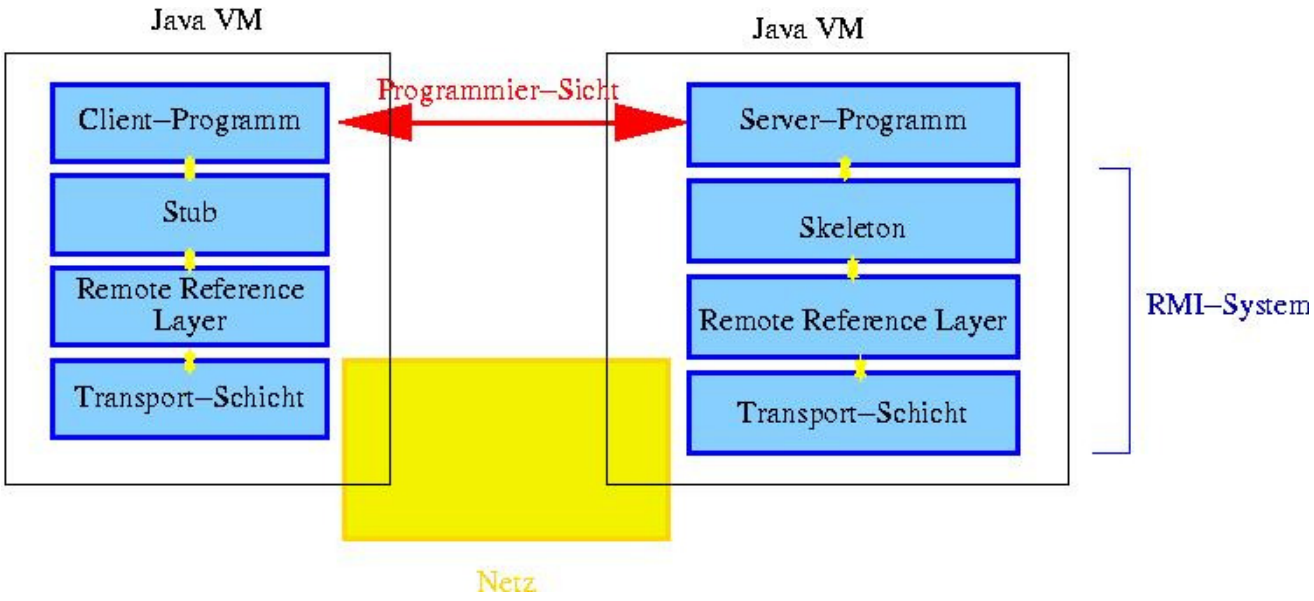
Einführung in RMI (Java Remote Method Invocation)

1 Das RMI Programmiermodell



Normalerweise macht man in Java nur Methodenaufrufe innerhalb einer einzelnen JVM. Objekte einer JVM können allerdings auch RMI Methoden von Objekten in einer entfernten JVM aufrufen. Dabei können diese JVMs können auf verschiedenen Rechnern im Netz laufen. Realisierung: Erzeugen eines Stellvertreters (Client-Stub) des entfernten Objekts (Server-Skeleton) in der entsprechenden lokalen JVM. Stub und Skeleton kommunizieren miteinander.

Genauere Schichten-Darstellung:



Die Remote Reference Layer muss entfernte Referenzen (Referenzen auf entfernte Objekte) auf Rechnernamen und Objekte abbilden.

2 RMI Programmierung

2.1 Interface java.rmi.Remote

Entfernte Methoden werden durch das Interface java.rmi.Remote definiert:

```
import java.rmi.*;

public interface Hello extends Remote {
  public String sayHello (String language) throws
RemoteException;
}
```

Eine RemoteException kann z.B. auftreten, wenn das entfernte Objekt nicht mehr verfügbar ist oder wenn das Netz gestört ist.

2.2 Server: Implementierung durch java.rmi.server.UnicastRemoteObject

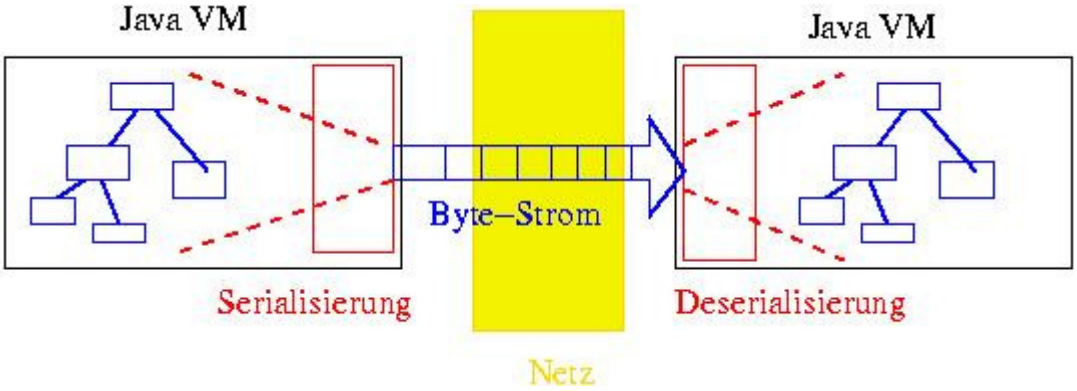
Server-Programme implementieren das Interface Remote und erweitern java.rmi.server.UnicastRemoteObject, welches die wichtigsten Methoden für die Verwendung von RMI bereitstellt.

```
import java.rmi.*;
import java.rmi.server.*;
public class HelloImpl
extends UnicastRemoteObject
implements Hello
{
  public String sayHello(String language) throws
RemoteException {
    String lang = language.toLowerCase();
    switch (lang.charAt(0)) {
      case 'd': return "Hallo!";
      case 'f': return "Salut!";
      case 'i': return "Ciao!";
      default: return "Hi!";
    }
  }
}
```

3 Serialisierung

Methoden-Parameter können auf zwei Arten verschickt werden:

1. Das entfernte Objekt wird als Remote Reference transportiert (das Objekt selbst "verlässt" dabei die JVM nicht, java.rmi.Remote)
2. Das entfernte Objekt wird als Kopie übertragen (entspricht einem "call by value", java.lang.Serializable)



Um Objekt-Kopien zu anderen JVMs zu übertragen, müssen die Datenstrukturen beim Sender in einen Datenstrom umgewandelt werden und beim Sender aus dem transportierten Datenstrom die Datenstruktur rekonstruiert werden. Dies leistet die

```
public class MobileObject implements java.io.Serializable {
  ...
}
```

Serialisierungs-API in Java für den Programmierer ganz einfach:

- Alle Grundtypen und die meisten Klassen des JDK implementieren das Interface Serializable.
- Felder von nicht-serialisierbaren (d.h. nicht das Interface Serializable implementierenden) Oberklassen und Komponenten-Klassen einer serialisierbaren Klasse werden durch deren Default-Konstruktor initialisiert und nicht mitübertragen.
- Instanzen von nicht-serialisierbaren Klassen in einem zu serialisierenden Objekt-Graph führen zu einer NotSerializableException.

3.1 Transformation eines Objekts in einen Byte-Strom

Das Objekt besteht aus einem **Zustand** (Inhalt seiner Objekt- und Klassen-Variablen) und einer **Implementierung** (= class-Datei). Die class-Datei muss bekannt sein, also entweder über CLASSPATH lokal oder über das Netz. Der Zustand ist dynamisch und nur zur Laufzeit bekannt. Es wird immer ein "Schnappschuss" dieses Zustands übertragen.

3.2 Verwendungsmöglichkeiten der Serialisierung

- Der Serialisierungs-Mechanismus von Java erlaubt auch zyklische Objekt-Graphen zu serialisieren.
- Zustand kann serialisiert (als Byte-Strom) auf einen Socket geschrieben werden
  => Übertragung von Objekten zwischen JVMs.
- Der Zustand (d.h. alle wichtigen Objekte) kann serialisiert in eine Datei geschrieben werden
  => Konfigurationen, Restart-Möglichkeiten, Java Beans...

Bei allem Komfort muss man allerdings stets bedenken, dass der Protokoll-Overhead der Serialisierung zu Lasten der Effizienz geht.

Für strukturell einfache, aber große Datenmengen (z.B. Arrays von Grundtypen) kann Message-Passing (MPI[1], jcluster[2]) die bessere Wahl sein!

3.3 Unterschiede von Remote- und Serializable-Objekten

Wenn sie als Argumente oder Ergebnis-Parameter von RMI-Aufrufen vorkommen, werden

- Serializable -Objekte "by value" übertragen
- Remote -Objekte "by reference" übertragen.

Alle Felder von Remote-Objekten sind transient. Werden Remote-Objekte z.B. auf ObjectOutputStreams geschrieben, gehen die Werte der Felder verloren!

4 Server- und Client-Programm

Bis jetzt haben wir ein Remote-Objekt (HelloImpl) samt Interface (Hello), dessen Methoden von einer entfernten JVM aufgerufen werden können. Es fehlen noch:

- eine Möglichkeit zur Erzeugung von Stub (client-seitig) und Skeleton (server-seitig).
- ein Server-Programm, das ein solches Objekt erzeugt und entfernten JVMs zur Verfügung stellt.
- ein Client-Programm, das einen solchen Aufruf vornimmt.

[1] http://www.lrz-muenchen.de/services/compute/linux-cluster/cpjava/mpijava.html
[2] vip.6to23.com/jcluster/

4.1 Stub- und Skeleton-Compiler rmic

Für eine Klasse, die das Interface Remote implementiert, erzeugt der RMI-Compiler rmic die benötigten Stubs und Skeletons:

```
javac Hello.java
javac HelloImpl.java
rmic HelloImpl          # erzeugt HelloImpl_Stub.class und
                        # HelloImpl_Skel.class
```

Für Interessierte: Durch rmic -keepgenerated HelloImpl kann man die erzeugten Stub- und Skel-Klassen als Java-Quelltext anschauen.

4.2 Die Objekt-Registrierung rmiregistry

Wie bekommt ein Client Zugriff auf ein Hello-Objekt?

Wir wissen: Stub für Client, Skeleton für Server.

- Der Server muss das Skeleton unter einem (möglichst eindeutigen) Namen bei einer rmiregistry anmelden. Diese rmiregistry muss auf demselben Rechner laufen, auf dem das Remote-Objekt (Skeleton) ausgeführt wird!
- Der Client muss
    1. den URL der rmiregistry kennen, von der er einen Stub des Objekts bekommen kann (z.B. rmi://lxsrv1.lrz-muenchen.de:4711)
    2. den Namen des Objekts kennen, unter dem der Stub bei der Registry bekannt ist.

4.3 Ein RMI-Server

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
public class HelloServer {

 public static void main (String[] args /* args[0]: port */) throws
RemoteException {
  int port = (args.length > 0) ? Integer.parseInt(args[0]) : 4711;
  HelloImpl obj = new HelloImpl();
  String objName = "HelloObj";

  if (System.getSecurityManager() == null) {
    System.setSecurityManager (new RMISecurityManager());
  }
  Registry registry = LocateRegistry.getRegistry (port);
  boolean bound = false;
  for (int i = 0; ! bound && i < 2; i++) {
      try {
        registry.rebind (objName, obj);
```

```
            bound = true;
            System.out.println (objName+" bound to registry, port " +
                                port + ".");
          }
          catch (RemoteException e) {
            System.out.println ("Rebinding " + objName + " failed, " +
                                "retrying ...");
            registry = LocateRegistry.createRegistry (port);
            System.out.println ("Registry started on port " + port +
                                ".");
          }
        }
        System.out.println ("HelloServer ready.");
      }
    }
```

4.4 Ein RMI-Client

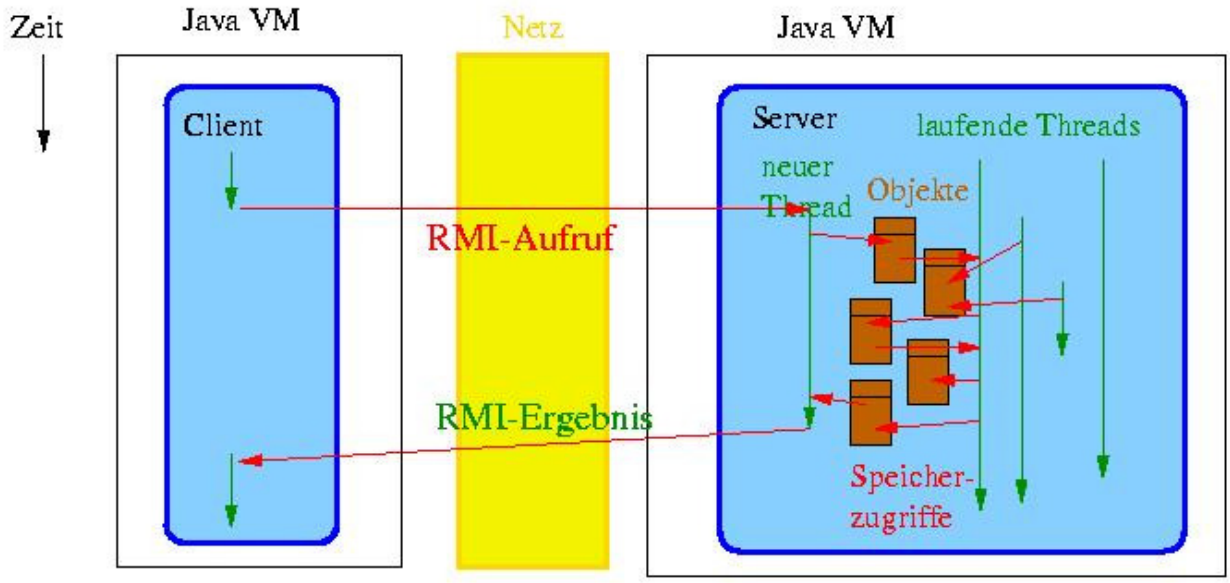
```
import java.rmi.*;
import java.rmi.server.*;

public class HelloClient {

static public void main (String[] args) /*args[0]: host, args[1]: port*/
{
   String host = (args.length < 1) ? "lxsrv1.lrz-muenchen.de" : args[0];
   int port = (args.length < 2) ? 4711 : Integer.parseInt(args[1]);
   try {
     if (System.getSecurityManager() == null) {
       System.setSecurityManager (new RMISecurityManager());
      }
     Hello obj = (Hello) Naming.lookup ("rmi://" + host + ":" + port +
                                        "/" + "HelloObj");
     System.out.println (obj.sayHello("GB"));
     System.out.println (obj.sayHello("Deutsch"));
     System.out.println (obj.sayHello("italiano"));
     System.out.println (obj.sayHello("français"));
   } catch (Exception e) {
     System.out.println ("HelloClient failed, caught exception " +
                         e.getMessage());
   }
 }
}
```

5 Threads auf der Server-Seite

Jeder RMI-Aufruf erzeugt auf der Server-Seite (dort, wo die Funktion ausgeführt wird; Skeleton) einen neuen Thread. Das bedeutet, dass dort mehrere solcher Threads gleichzeitig laufen können, zusammen evtl. mit weiteren dauerhaft laufenden Threads des Servers (z.B. Garbage Collector, Compute-Server,...)



Man muss sich also in jedem Fall Gedanken über eine notwendige Synchronisation machen (mit synchronized-Blöcken, ggf. auch mit wait() und notify()).

6 Die Policy - Sicherheitsaspekte

Wenn eine benötigte class-Datei von einem öffentlichen Web-Server oder aus einem entfernten Verzeichnis(-Dienst) geholt werden soll, müssen gewisse Regelungen bzgl. Sicherheit getroffen werden.

Für diesen Zweck wird in den RMI-Programmen der RMISecurityManager gesetzt. Dieser verhält sich gemäß den Regeln, die in der Datei

```
$HOME/ .java.policy
```

angegeben werden.
Am einfachsten, aber auch am unsichersten, wäre folgendes .java.policy:

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

RMI (bzw. das zugrunde liegende Protokoll JRMP) verwendet Sockets, deshalb müssen mindestens SocketPermissions gesetzt werden, etwa:

```
grant {
 permission java.net.SocketPermission "localhost", "accept,connect,listen";
 permission java.net.SocketPermission "*.lrz-muenchen.de", "accept,connect,listen";
};
```

Die Pakete java.rmi und java.rmi.server selbst definieren keine Permissions. Daher müssen in .java.policy keine weiteren RMI-spezifischen Permissions gesetzt werden.

Einen Überblick über mögliche Permissions gibt http://java.sun.com/j2se/1.4.2/docs/guide/security/permissions.html.

Weitere Beispiele für Policy-Files z.B. in http://java.sun.com/j2se/1.4.2/docs/guide/security/PolicyFiles.html#Examples.

Zur Erstellung der Datei .java.policy kann auch das GUI-basierte policytool aus dem JDK verwendet werden. Dokumentation in http://java.sun.com/j2se/1.4.2/docs/tooldocs/solaris/policytool.html.

7 Code-Mobilität

RMI überträgt den Zustand eines Objekts, nicht aber die Implementierung der zugehörigen Klasse.

Problem**:** Bewegt man sich außerhalb eines gemeinsamen Datei-Systems (z.B. AFS), so sind Klassen i.a. beim Client nicht lokal verfügbar, d.h. nicht in dessen CLASSPATH aufgelistet. Konsequenz:

- Klassen-Implementierungen müssen zur Laufzeit des Client dynamisch aus dem Netz (z.B. von Web-Servern) nachgeladen werden.
- Serialisierung: Auf Empfängerseite werden Zustand und Implementierung zu einem gültigen Objekt zusammengefügt.

Programmiertechnisch läuft dies über den java.rmi.server.RMIClassLoader und entsprechende gesetzte Property codebase, z.B. mit

```
java -Djava.rmi.server.codebase=http:// hostname:portname / ...
```

Siehe dazu z.B.:

- das Tutorial "Dynamic code downloading using RMI" http://java.sun.com/j2se/1.4.2/docs/guide/rmi/codebase.html.
- einen Artikel in JavaWorld: "Revolutionary RMI: Dynamic class loading and behavior objects" (http://www.javaworld.com/javaworld/jw-12-1998/jw-12-enterprise.html).

8 Activatable Objects

Der HelloServer erzeugt beim Start sofort ein Hello-Objekt. Dies kann für ressourcen-intensive Objekte u.U. nicht erwünscht sein. Zu diesem Zweck gibt es in Java 2 als Alternative zu **UnicastRemoteObject** die **java.rmi.activation.Activatable** Objekte zusammen mit einem RMI Activation-Daemon **rmid**. Die notwendigen Änderungen im Java-Quellcode sind dabei minimal und auf die Server-Seite beschränkt.

Einführung mit Beispielprogramm unter
http://java.sun.com/j2se/1.4.2/docs/guide/rmi/activation.html.

9 RMI - Zusammenfassung

Wichtige Klassen:

- interface java.rmi.Remote
- java.rmi.server.UnicastRemoteObject
- java.rmi.registry.LocateRegistry
- java.io.Serializable
- java.rmi.RMISecurityManager
- java.rmi.RMIClassLoader

Tools:

- rmic (Stub- und Skeleton-Compiler)
- javapolicy (mit Datei .java.policy)

10 Informationsquellen

- Java RMI Home Page: http://java.sun.com/products/jdk/rmi/index.html
- Tutorial: http://java.sun.com/j2se/1.4.2/docs/guide/rmi/getstart.doc.html
- RMI Spezifikation mit ausführlicher Dokumentation: http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html
- Ein Buch für den Überblick: J. Farley: Java Distributed Computing. O'Reilly and Associates, 1998

11 Ihre Aufgabe: Eine verteilte Anwendung mit RMI

Ziel: Ein entferntes Objekt mit einer Methode quadrat(long) erzeugen, die wenn sie von entfernten Klienten mit einer Zahl als Parameter aufgerufen wird, das Quadrat dieser Zahl zurückgibt.

Folgende Vorgehensweise wird verlangt und auch empfohlen:

1. Das Quadrat Interface (Quadrat.java) schreiben, welches das entfernte Objekt definiert.

2. Die Klasse QuadratImpl (QuadratImpl.java) schreiben welche das Interface Quadrat implementiert. Sie soll dabei eine main()-Funktion beinhalten, die den RMISecurityManager erzeugt und installiert, falls er nicht schon installiert ist, und das entfernte Objekt bei der rmiregistry registiert.

3. Das Klientenprogramm QuadratClient (QuadratClient.java) schreiben, welches zwei Argumente auf der Kommandozeile entgegen nimmt: den Namen der Servermaschine auf welcher das entfernte Objekt registriert ist und die Zahl welche quadriert werden soll.

4. Das Interface und die Klassen kompilieren.

5. Stubs und Skeletons mit rmic erstellen.

6. Starten des Namensdienstes auf dem Server:

```
> rmiregistry <PORTNR>
```

7. Starten des Servers:

```
> java –Djava.security.policy=<POLICYFILE> ServerProc
```

   Bevor Sie den Server starten, müssen Sie ein passendes Policyfile erstellt haben. Es genügt hier die unsichere Variante.

8. Auf einem Rechner den Client starten:

```
> java –Djava.security.policy=<POLICYFILE> QuadratClient
  <SERVER:PORTNR> Zahl
```

Hinweise:

- Benutzen Sie bitte Ports zwischen 2500 und 15000 um nicht die Arbeit ihrer Kollegen zu stören

- Bitte beenden Sie am Ende wieder Ihre gestartete rmiregistry!

Abgabe:

Als Abgabe wird eine funktionierende Version der Aufgaben inklusive Quellcode, eine Prozessliste der laufenden Prozesse wenn das Programm arbeitet und eine vollständige Prozessliste, wenn alle Programme beendet sind, erwartet.

Zusätzlich müssen folgende Fragen beantwortet werden:

1. Nennen sie mindestens drei Unterschiede zu CORBA.
2. Nennen sie ein Beispielszenario bei welchem der RMIClassLoader benötigt wird
3. Benötigt man mit der J2SE (Java 2 Standard Edition) ab der Version 1.5 ebenfalls den RMI-Compiler rmic? Warum?
4. Was beinhaltet das Interface java.rmi.Remote? Wieso?