Remote Method Invocation RMI (Aufruf entfernter Methoden) ist der Aufruf einer Methode eines entfernten Java-Objekts und realisiert die Java-eigene Art eines sog. Remote Procedure Call RPC. Entfernt bedeutet dabei, dass sich das Objekt in einer anderen Virtuellen Maschine befinden kann, die ihrerseits auf einem entfernten Rechner oder auf dem lokalen Rechner laufen kann. Dabei sieht der Aufruf für das aufrufende Objekt (bzw. dessen Programmierer) genauso aus, wie ein lokaler Aufruf, es müssen jedoch besondere Ausnahmen abgefangen werden, die zum Beispiel einen Verbindungsabbruch signalisieren können.

Inhaltsverzeichnis

1. Einleitung....2
1.1 RMI-Kommunikationsablauf....2
1.3 Stub und Marshaling....3
1.4 Namensdienst....4
2. Bank-Applikation....4
2.1 RMI mit Stub generiert mit dem rmic-Compiler....6
2.2 RMI mit Stub ohne den rmic-Compiler....8
3. Zusammenfassung....10
Literatur....10

1. Einleitung

Für die Applikationsentwicklung ist es häufig von entscheidender Bedeutung auf Daten und Dienste zuzugreifen, die sich auf anderen Rechnern in einem Netzwerkverbund befinden. Insbesondere der Zugriff auf entfernte Dienste ist für viele Anwendungen wichtig.

RMI stellt hierzu einen Mechanismus zur so genannten verteilten Applikationsprogrammierung bereit, d.h. es ermöglicht dem Applikationsprogrammierer auf entfernte Objekte zuzugreifen und deren Dienste zu nutzen. Als entfernte Objekte (Remote Objects) werden Objekte bezeichnet, die sich auf der gleichen oder einer anderen JVM befinden. Die Plattformunabhängigkeit der Java-Architektur lässt RMI zu einem mächtigen Werkzeug werden, da es auf einfache Weise ermöglicht, Programmcode im Netzwerk zu verteilen, auch über verschiedene Plattformen hinweg.

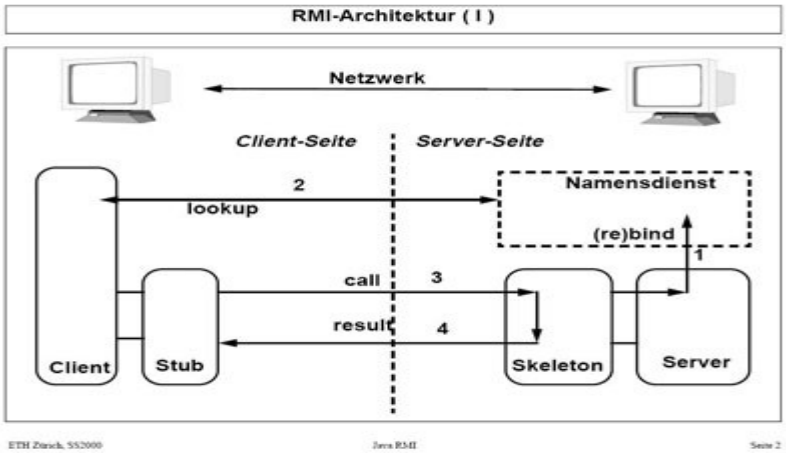
Die Übertragung von reinen Daten im Netzwerk ist mit dem Zugriff auf die Transportschicht relativ einfach zu erledigen. Das Verteilen von Programmcode dagegen und damit der Zugriff auf die Dienste entfernter Rechner erfordert allerdings ein aufwendiges Kommunikationsprotokoll, dass sich um die Datenerstellung und Kommunikationssteuerung kümmert. Das direkte Ziel von RMI ist es, den Applikationsprogrammierer hierbei zu unterstützen und ihm die aufwendige und fehleranfällige Implementierung eines solchen Protokolls abzunehmen.

RMI implementiert mehrere Protokolle zwischen denen der Applikationsprogrammierer wählen kann, um das für seinen Anwendungszweck passende auszuwählen, ohne sich um die Details der Protokolle zu kümmern:

- RMI IIOP zur Integration in CORBA
- JRMP (Java Remote Method Protocol) ist das Standardprotokoll für RMI
- RMI über HTTP getunnelt zur Umgehung von Firewalls
- Verschlüsselt über SSL zur sicheren Kommunikation

1.1 RMI-Kommunikationsablauf

Den Ablauf einer RMI-Kommunikation zeigt die folgende Abbildung (Quelle: ETH Zürich):



1. Der Server registriert ein Remote Object bei der RMI-Registry unter einem eindeutigen Namen.

2. Der Client schaut bei der RMI-Registry unter diesem Namen nach und bekommt eine Objektreferenz, die seinem Remote Interface entsprechen muss.

3. Der Client ruft eine Methode aus der Objektreferenz auf (dass diese Methode existiert, wird durch das Remote Interface garantiert).

4. Der Server gibt dem Client die Rückgabewerte dieses Aufrufes, oder der Client bekommt eine Fehlermeldung (z. B. bei einem Verbindungsabbruch).

Ist der Zugriff auf ein Objekt auf einem anderen Computer möglich, können Methoden des Remote-Objekts aufgerufen werden. Dazu müssen die Methodenparameter irgendwie zum anderen Computer gelangen, das Remote-Objekt ist über die Ausführung der Methode zu informieren und der Rückgabewert der Methode muss zurückgeliefert werden. In der RMI-Terminologie heisst das Objekt, das den Remote-Aufruf ausführt, Client-Objekt. Das Remote-Objekt heisst Server-Objekt.

Der Remote-Methodenaufruf erfolgt wie der Aufruf einer gewöhnlichen Methode in Java, jedoch ist diese gekapselt in einem Ersatzobjekt namens Stub (analog einem Proxy). Der Stub befindet sich auf dem Computer des Client-Objekts. Der Stub verpackt die im Remote-Methodenaufruf verwendeten Parameter in einer geräteunabhängigen Codierung. Die Zahlen werden beispielsweise im Big-Endian Format gesendet. Die Parameter in ein Format zu konvertieren, das sich für den Transport von einer virtuellen Maschine zu einer anderen eignet, wird Marshaling genannt.

1.3 Stub und Marshaling

Der Stub (ab Version 1.5 nicht mehr notwendig) sendet Informationen an den Server. Auf der Seite des Server-Objekts führt ein Empfänger (früher Skeleton, ab Version 1.1 nicht mehr notwendig) folgende Aktionen durch: er entpackt die Parameter; er sucht den Standort des aufzurufenden Objekts; er ruft die gewünschte Methode auf; er übernimmt den Rückgabewert oder die Ausnahme des Aufrufs und verpackt sie per Marshaling; er sendet das Paket, das aus den per Marshaling verpackten Rückgabedaten besteht, an den Stub des Client-Objekts.

Dieser Vorgang ist komplex, läuft aber vollkommen automatisch ab und ist zu einem gewissen Mass für den Programmierer transparent. Die Entwickler des RMI haben hart daran gearbeitet, den Remote-Objekten das gleiche Erscheinungsbild zu geben wie lokalen Objekten. Die Remote-Objekte werden genau wie lokale Objekte in die Garbage Collection einbezogen!

Wird ein Remote-Objekt als Parameter oder als Rückgabewert einer Remote-Methode an ein anderes Programm übergeben, muss dieses Programm in der Lage sein, das zugeordnete Stub-Objekt zu behandeln, d.h. es muss den Code für die Stub-Klasse

besitzen oder diesen Code dynamisch laden. Dieser Vorgang ist ähnlich dem Klassenladen von Applets. Wenn ein Programm neuen Code von einem anderen Netzwerkstandort lädt, sind Sicherheitsfragen zu beachten. Aus diesem Grund muss in RMI-Client Applikationen unter Umständen ein Sicherheitsmanager eingesetzt werden.
Vor dem Erscheinen der Java 2 Standard Edition (J2SE) in Version 1.5.0 war der Stub eine mit dem RMI-Compiler rmic erzeugte Klasse. Seit der Version 1.5 ist es nicht mehr notwendig den RMI-Compiler rmic aufzurufen. Das Erstellen des Stubs wird von der Java Virtual Machine übernommen.

1.4 Namensdienst

Entfernte Objekte können zwar auch von einem bereits im Programm bekannten entfernten Objekt bereitgestellt werden, für die erste Verbindungsaufnahme werden aber die Adresse des Servers und ein Bezeichner (ein RMI-URL) benötigt. Für den Bezeichner liefert ein Namensdienst auf dem Server eine Referenz auf das entfernte Objekt zurück. Damit dies funktioniert, muss das entfernte Objekt im Server sich unter diesem Namen zuvor beim Namensdienst registriert haben. Der RMI-Namensdienst wird über statische Methoden der Klasse `java.rmi.Naming` und ab der Version 1.5 über `java.rmi.registry.Registry` angesprochen. Der Namensdienst ist als eigenständiges Programm implementiert und wird RMI Registry (`rmiregistry`) genannt.

2. Bank-Applikation

Bevor RMI an einem praktischen Beispiel eingesetzt wird, ist es eine Voraussetzung, die fundamentalen Design Prinzipien zu verstehen:

- Java benützt Garbage Collection im Lebenszyklus eines Objekts. In einer Remote-Objekt Architektur sollte dies ebenfalls möglich sein.

- Java benützt das Exception-Handling zum Benachrichtigen von Fehlern. Objektkommunikation über ein Netzwerk kann sehr fehleranfällig sein, deshalb muss dem Exception-Handling eine grosse Bedeutung zugemessen werden.

- Java benützt Schnittstellen, die entsprechend implementiert werden müssen. In einer Remote-Objekt Architektur, die auf einer delegierten Verarbeitung über ein Netzwerk basiert, bestimmen Schnittstellen die Funktionalität von Server-Objekten. Dies ist vergleichbar mit RPC und der Common Object Request Broker Architecture Interface Definition Language (CORBA IDL), bei denen die Schnittstelle die Funktionalität beschreibt, unabhängig von der Implementation.

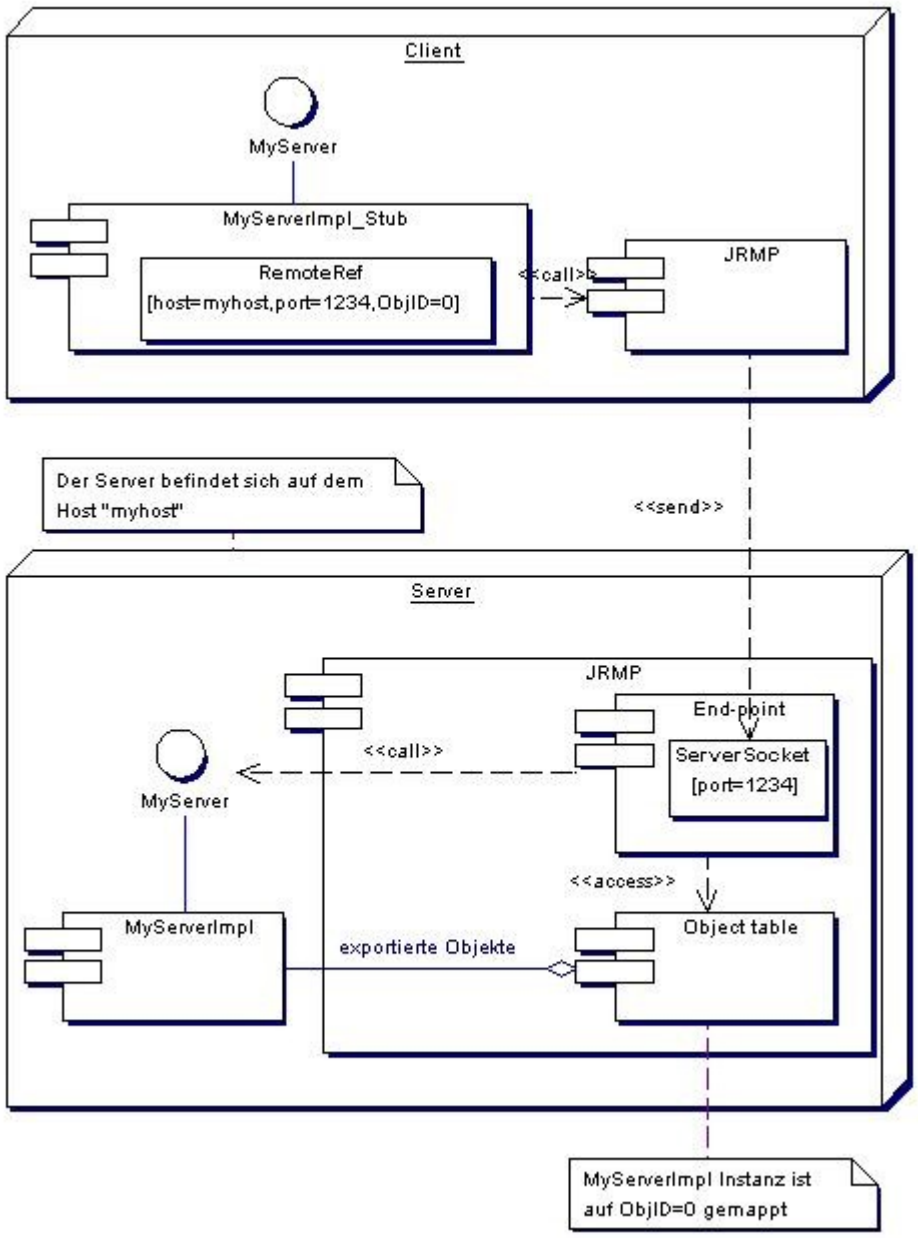
- Objekte in Java werden über Methoden angesprochen. Kommuniziert das Client- mit dem Server-Objekt über ein Netzwerk, kann ein Proxy die Illusion vermitteln, dass das Server-Objekt wie ein gewöhnliches Java-Objekt angesprochen wird (Stub).

- Java erlaubt Entwicklern den Klassenlader-Mechanismus einzusetzen (Reflection). RMI-Implementierungen können diesen Mechanismus anwenden, um Klassen dynamisch zu laden (dynamic classloading).

Diese Prinzipien haben einen direkten Einfluss darauf, wie RMI arbeitet. Jedoch bestehen wichtige Unterschiede zwischen der RMI-Programmierung und der normalen Java-

Programmierung (gilt nicht für die Methodenaufrufe!), wie das Beispiel (nach der Architektur) zeigt.

Es gibt viele kleine Details die erkannt werden müssen, bevor RMI wirklich verstanden wird. Deshalb dazu ein Blick in die RMI/JRMP Architektur:

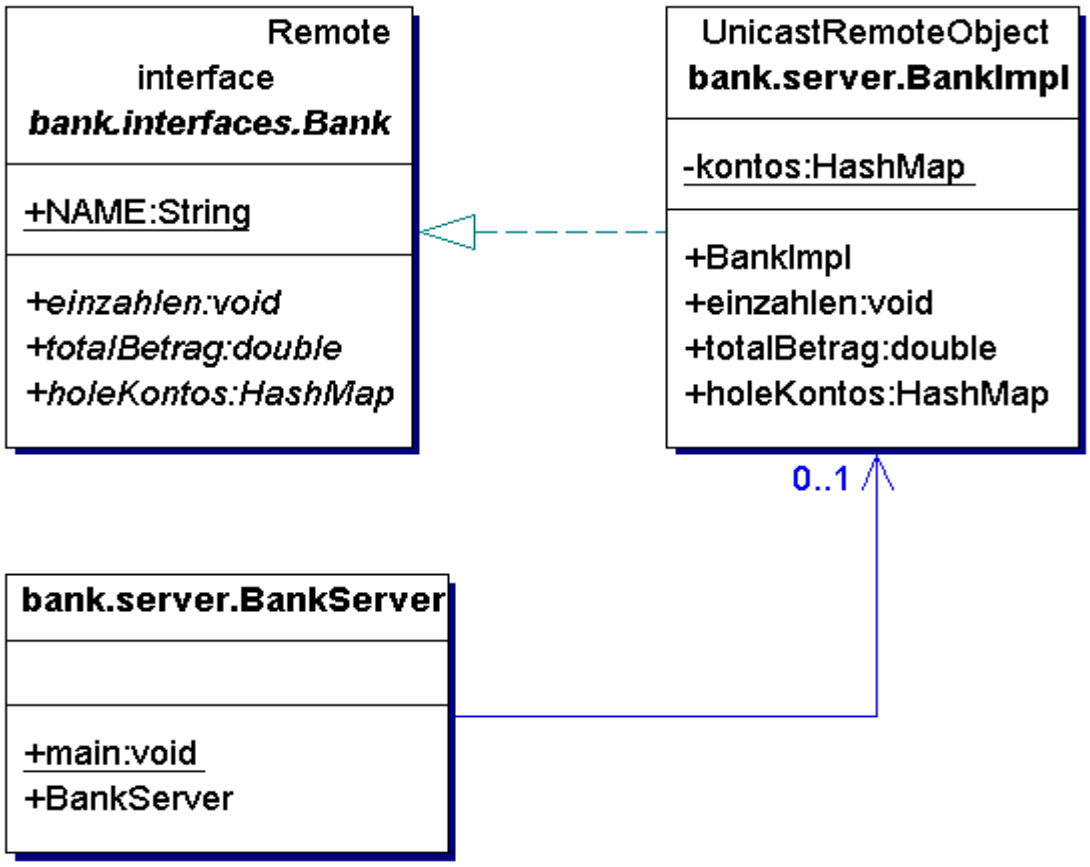


Die Benutzeroberfläche der Bank-Applikation sieht folgendermassen aus:

Die Bank-Applikation ist eine RMI-Applikation, die es dem Benutzer (Client) erlaubt, Geld auf ein Konto einzuzahlen (`einzahlen`) und den Kontostand abzufragen (`totalBetrag`). Die dazu notwendigen Klassen und deren teilweise Implementation zeigen die folgenden Ausführungen (zuerst für Java 2 Standard Edition (J2SE) in Version 1.4 und kleiner und danach für die Version 1.5 und grösser).

2.1 RMI mit Stub generiert mit dem rmic-Compiler

Das Klassendiagramm auf Server-Seite sieht folgendermassen aus:



Die Schnittstelle der Bank-Applikation zeigt das folgende Listing:

```
package bank.interfaces;

import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.HashMap;

/**
 * Schnittstelle (Vetrag) für Server und Client.
 * Sie besitzt zwei Methoden, die von aussen aufgerufen
 * werden können.
 *
 * @author Bau / letzte Änderung: 04.01.2007
 */
public interface Bank extends Remote {

  public static final String NAME = "bank";
  /**
   * Übernimmt den Betrag des Aufrufers und zahlt ihn aufs Konto.
   * @param  id: die Identifikation des Clients
   * @param  betrag: der einzuzahlende Geldbetrag
   * @exception RemoteException falls Fehler auftreten
   */
  public void einzahlen(String id,double betrag) throws RemoteException;
  /**
   * Gibt den Kontostand an den Aufrufer zurück.
   * @param  id: die Identifikation des Clients
   * @return kontostand
   * @exception RemoteException falls Fehler auftreten
   */
  public double totalBetrag(String id) throws RemoteException;
  /**
   * Holt alle Kontos und gibt diese an den Aufrufer zurück.
   * @return Kontos
   * @exception RemoteException falls Fehler auftreten
   */
  public HashMap holeKontos() throws RemoteException;
}
```

Die Implementation auf Server- und Client-Seite zeigen die folgenden Code-Ausschnitte:

Server

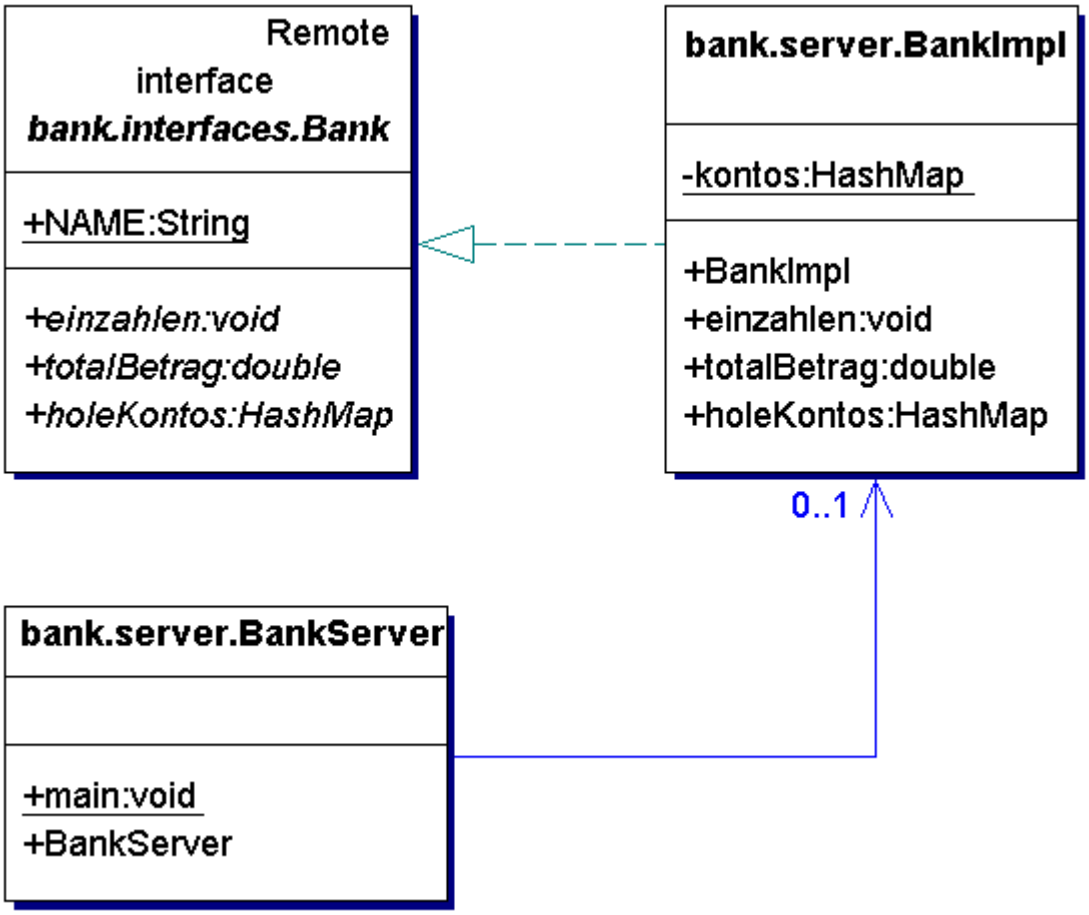
```
public BankServer() {
  try {
    BankImpl server = new BankImpl();
    Naming.rebind(Bank.NAME, server);
  }
  catch (Exception e) {
    e.printStackTrace(System.err);
  }
}
```

Client

```
private Bank server = null;
...
try {
  // Lokalisieren des Remote-Objekts
  server = (Bank) Naming.lookup
("rmi://localhost/" +   Bank.NAME);
}
catch (Exception e) {
  e.printStackTrace(System.err);
}
...
```

2.2 RMI mit Stub ohne den rmic-Compiler

Vor dem Erscheinen der Java 2 Standard Edition (J2SE) in Version 1.5.0 war der Stub eine mit dem RMI-Compiler rmic erzeugte Klasse (vergl. Kapitel 2.1). Seit der Version 1.5 ist es nicht mehr notwendig den RMI-Compiler rmic aufzurufen. Das Erstellen des Stubs wird von der Java Virtual Machine übernommen. Das ergibt folgendes Klassendiagramm:



Die Implementation der Schnittstelle (`BankImpl`) braucht nicht mehr von der Klasse `UnicastRemoteObject` abgeleitet zu werden. Die Registrierung beim Namensdienst erfolgt nicht mehr über die Klasse `Naming` sondern über die Klasse `Registry`. Die Generierung des Stub wird von der VM übernommen. Das Listing der Klasse `BankServer` sieht folgendermassen aus:

```
public class BankServer {
  /**
   * Startet den Server als stand-alone Application.
   * @param args
   */
  public static void main(String[] args) throws Exception {
    new BankServer();
  }

  /** Erzeugt ein Remote-Objekt und registriert es in der RMI-Registry.
  */
  public BankServer() {
```

```
    try {
      // Erzeugt das Remote-Objekt
      BankImpl server = new BankImpl();
      // Registriert das Server-Objekt
      // Anstelle von bind wird rebind benutzt.
      // Dies ist nützlich bei einem Restart des Servers,
      // er überschreibt damit die alte Bindung.
      Bank stub = (Bank) UnicastRemoteObject.exportObject(server, 0);
      Registry registry = LocateRegistry.getRegistry();
      registry.rebind(Bank.NAME, stub);
      System.out.println("Der Server ist gestartet und registriert");
    }
    catch (Exception e) {
      System.out.println("Das Objekt konnte nicht erzeugt werden");
      e.printStackTrace(System.err);
    }
  }
}
```

Auf Client-Seite ändert sich die Anfrage an den Namensdienst folgendermassen:

```
    try {
      // Lokalisieren des Remote-Objekts
      ResourceBundle bundle = ResourceBundle.getBundle("host");
      Registry registry =
        LocateRegistry.getRegistry(bundle.getString("HOSTNAME"));
      server = (Bank) registry.lookup(Bank.NAME);
    }
    catch (Exception e) {
      System.out.println("Das Objekt konnte nicht erzeugt werden");
      e.printStackTrace(System.err);
    }
```

3. Zusammenfassung

Remote Method Invocation (entfernter Methodenaufruf) ist der Aufruf einer Methode eines entfernten Java-Objekts und realisiert die Java-eigene Art eines Remote Procedure Call (RPC). Entfernt bedeutet dabei, dass sich das Objekt in einer anderen Virtuellen Maschine befinden kann (die ihrerseits auf einem entfernten Rechner oder auf dem lokalen Rechner laufen kann). Dabei sieht der Aufruf für das aufrufende Objekt genauso aus, wie ein lokaler Aufruf, es müssen jedoch besondere Ausnahmen abgefangen werden, die zum Beispiel einen Verbindungsabbruch signalisieren können.

Auf der Client-Seite kümmert sich der sogenannte Stub um den Netzwerktransport. Der Stub muss entweder lokal oder über das Netz für den Client verfügbar sein. Vor dem Erscheinen der Java 2 Standard Edition (J2SE) in Version 1.5.0 war der Stub eine mit dem RMI-Compiler rmic erzeugte Klasse. Seit der Version 1.5 ist es nicht mehr notwendig den RMI-Compiler rmic aufzurufen. Das Erstellen des Stubs wird von der Java Virtual Machine übernommen.

Entfernte Objekte können zwar auch von einem bereits im Programm bekannten entfernten Objekt bereitgestellt werden, für die erste Verbindungsaufnahme werden aber die Adresse des Servers und ein Bezeichner (ein RMI-URL) benötigt. Für den Bezeichner liefert ein Namensdienst auf dem Server eine Referenz auf das entfernte Objekt zurück. Damit dies funktioniert, muss das entfernte Objekt im Server sich unter diesem Namen zuvor beim Namensdienst registriert haben. Der Namensdienst ist als eigenständiges Programm implementiert und wird RMI Registry genannt.

Literatur

Öberg Rickard: *Mastering RMI*. John Wiley & Sons, Inc., New York 2001

Sun Dokumentation:
http://java.sun.com/javase/6/docs/technotes/guides/rmi/index.html
http://java.sun.com/javase/6/docs/platform/rmi/spec/rmiTOC.html

Kommandozeilen Befehle RMI

- Classpath exportieren:

  ```
  export CLASSPATH=target/UhrenServer-0.0.1-SNAPSHOT.jar (Mac OS X)
  set CLASSPATH=target/UhrenServer-0.0.1-SNAPSHOT.jar (Windows)
  ```

- rmiregistry Dienst starten:

  ```
  rmiregistry 1099 (Mac OS X)
  start rmiregistry 1099 (Windows)
  ```

- Applikation aus JAR starten:

  ```
  java -jar target/UhrenServer-0.0.1-SNAPSHOT.jar
  ```

- JAR mit maven erstellen

  ```
  mvn compile package
  ```

```
    try {
      // Erzeugt das Remote-Objekt
      BankImpl server = new BankImpl();
      // Registriert das Server-Objekt
      // Anstelle von bind wird rebind benutzt.
      // Dies ist nützlich bei einem Restart des Servers,
      // er überschreibt damit die alte Bindung.
      Bank stub = (Bank) UnicastRemoteObject.exportObject(server, 0);
      Registry registry = LocateRegistry.getRegistry();
      registry.rebind(Bank.NAME, stub);
      System.out.println("Der Server ist gestartet und registriert");
    }
    catch (Exception e) {
      System.out.println("Das Objekt konnte nicht erzeugt werden");
      e.printStackTrace(System.err);
    }
}
```

bankimpl server = new bankimpl();

Erzeugt ein Remote-Objekt

```
bank stub = (bank)
unicastremoteobject.exportobject(server, 0);
```

Server-Objekt wird exportiert und in Stub exportiert

```
registry registry = locateregistry.getregistry();
```

Registry wird geholt

registry.rebind(bank.NAME, stub);

Das Remote-Objekt wird in der Registry registriert
und ist somit Bekannt für Zugriffe von Clients.

Name muss eindeutig sein.

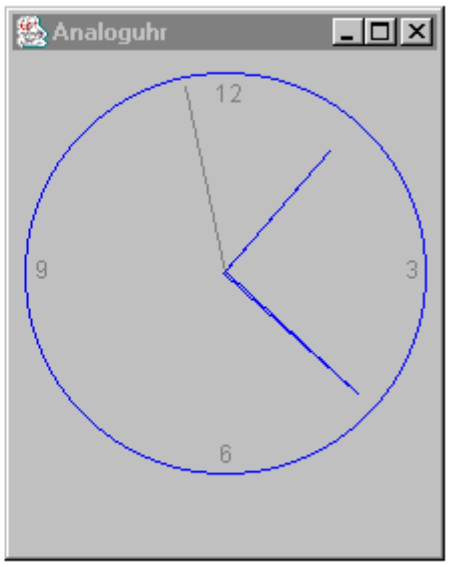
Remote Method Invocation ist der Aufruf einer Methode eines entfernten Java-Objekts und realisiert die Java-eigene Art des Remote Procedure Call. »Entfernt« bedeutet dabei, dass sich das Objekt in einer anderen Java Virtual Machine befinden kann, die ihrerseits auf einem entfernten Rechner oder auf dem lokalen Rechner laufen kann.

Inhaltsverzeichnis

1. Uhrensystem 1
1.1 Klassendiagramm 2
2. RMI-Uhrensystem 3
2.1 UhrServer 3
2.2 Clients 4
3. Aufgabe 4

1. Uhrensystem

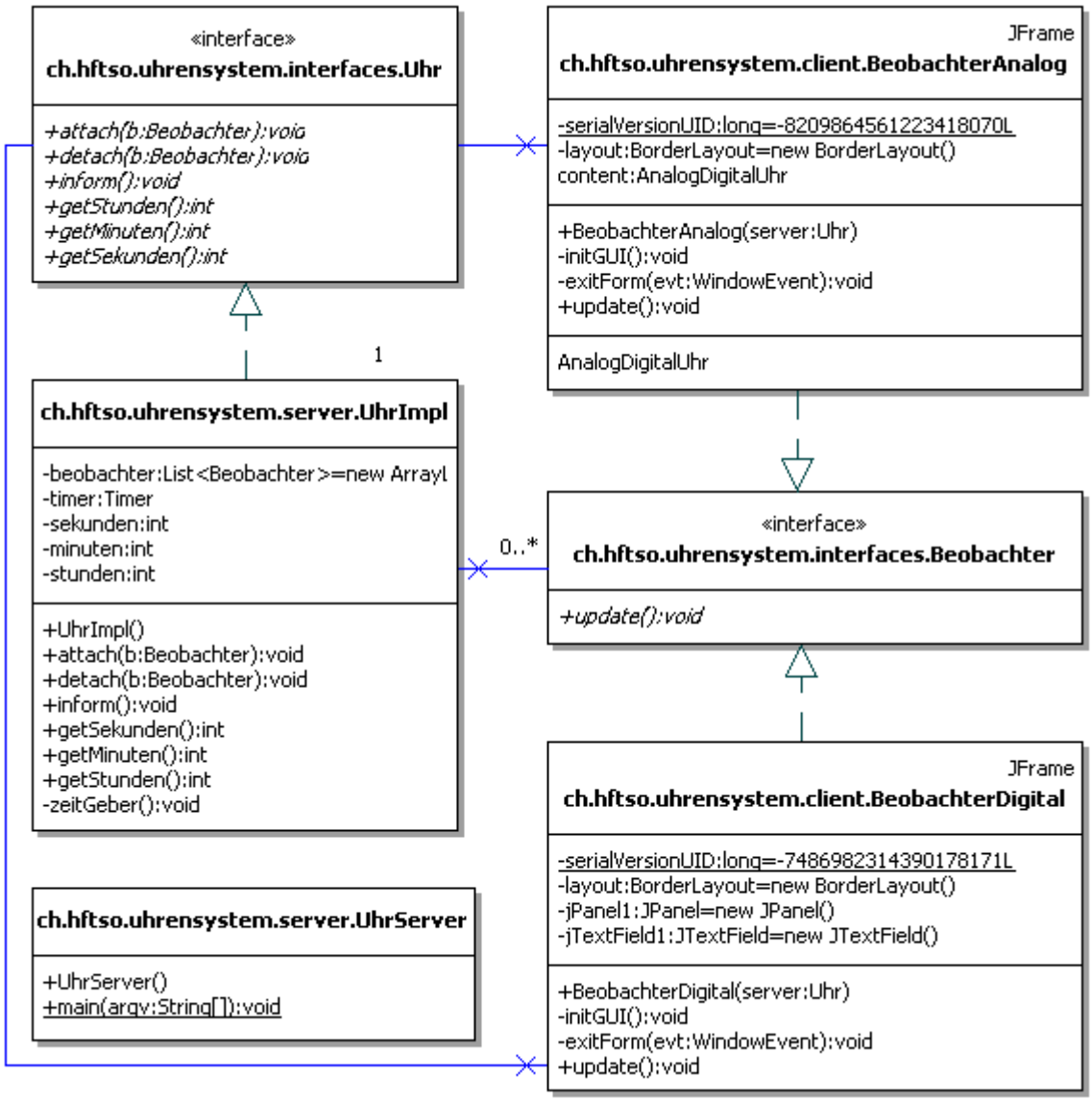
Die folgende Aufgabenbeschreibung stammt aus Informatik III: Die Applikation Uhr, die zu entwickeln ist, enthält einen Teil eines Uhrensystems, das die Uhrzeit in verschiedenen Formaten anzeigt. Die Uhrzeit muss in folgenden Formaten angezeigt werden:



Für die Anzeige der beiden Formate sind die Beobachter `BeobachterAnalog` und `BeobachterDigital` einzusetzen. Die Beobachter sollten alle 100 Millisekunden informiert werden, damit sie die aktuelle Zeit in Stunden, Minuten und Sekunden holen können.

1.1 Klassendiagramm

Das Klassendiagramm des bestehenden Uhrensystems sieht folgendermassen aus:

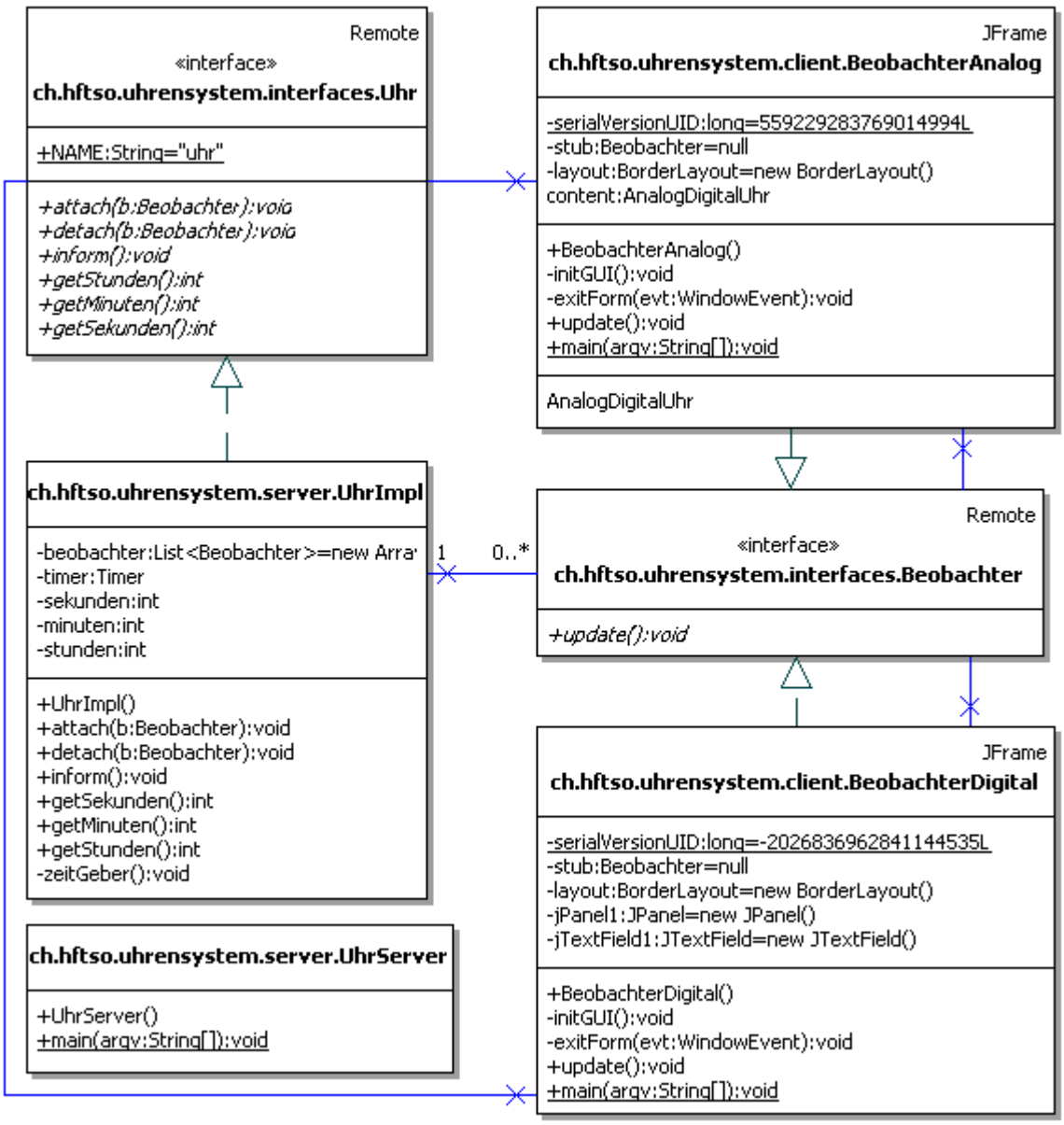


Der Konstruktor des `UhrServer` erzeugt die benötigten Objekte:

```
public UhrServer() {
  Uhr server = new UhrImpl();
  new BeobachterDigital(server);
  new BeobachterAnalog(server);
}
```

2. RMI-Uhrensystem

Das Uhrensystem aus Kapitel 1 ist in eine RMI-Applikation umzuwandeln, d.h. die Uhr wird zum Server und die beiden Beobachter werden zu Clients und müssen verteilt eingesetzt werden können. Daraus ergibt sich folgendes Klassendiagramm:



Der Server und die beiden Clients sind eigenständige Programme (`main`-Methode) und können sich an verschiedenen Orten befinden.

2.1 `UhrServer`

Der Konstruktor des `UhrServer` erzeugt das Remote-Objekt und registriert es in der RMI-Registry (`rmiregistry`):

```
public UhrServer() {
  try {
    Uhr server = new UhrImpl();
    Uhr stub = (Uhr) UnicastRemoteObject.exportObject(server, 0);
    Registry registry = LocateRegistry.getRegistry();
    registry.rebind(Uhr.NAME, stub);
    System.out.println("Der Server ist gestartet und registriert");
  }
  catch (Exception e) {
    System.out.println(e.toString());
  }
}
```

2.2 Clients

Die Clients müssen sich exportieren, damit sie der Server aufrufen kann:

```
  try {
    stub = (Beobachter) UnicastRemoteObject.exportObject(this,0);
  }
  catch (Exception ex) {
    System.out.println(ex.toString());
  }
```

Danach müssen sie sich anmelden:

```
  try {
    ResourceBundle bundle = ResourceBundle.getBundle("host");
    Registry registry =
        LocateRegistry.getRegistry(bundle.getString("HOSTNAME"));
    lnkUhr = (Uhr) registry.lookup(Uhr.NAME);
    lnkUhr.attach(stub);
    System.out.println("Der BeobachterDigital ist angemeldet");
  }
  catch (Exception ex) {
    System.out.println(ex.toString());
  }
```

3. Aufgabe

Implementieren und testen Sie das RMI-Uhrensystem gemäss Kapitel 2.

Getting Started Using Java™ RMI

This tutorial shows you the steps to follow to create a distributed version of the classic Hello World program using Java™ Remote Method Invocation (Java RMI). While you work through this example, you will probably come up with a number of related questions. You are encouraged to look for answers in the Java RMI FAQ and Preservation of Mail Lists and Documentation External to the River Project.

The distributed Hello World example uses a simple client to make a remote method invocation to a server which may be running on a remote host. The client receives the "Hello, world!" message from the server.

This tutorial has the following steps:

- Define the remote interface
- Implement the server
- Implement the client
- Compile the source files
- Start the Java RMI registry, server, and client

The files needed for this tutorial are:

- `Hello.java` - a remote interface
- `Server.java` - a remote object implementation that implements the remote interface
- `Client.java` - a simple client that invokes a method of the remote interface

Note: For the remainder of this tutorial, the terms "remote object implementation" and "implementation class" are used interchangeably to refer to the class `example.hello.Server`, which implements a remote interface.

Define the remote interface

A remote object is an instance of a class that implements a *remote interface*. A remote interface extends the interface `java.rmi.Remote` and declares a set of *remote methods*. Each *remote method* must declare `java.rmi.RemoteException` (or a superclass of `RemoteException`) in its `throws` clause, in addition to any application-specific exceptions.

Here is the interface definition for the remote interface used in this example, `example.hello.Hello`. It declares just one method, `sayHello`, which returns a string to the caller:

```
package example.hello;

import java.rmi.Remote;
import java.rmi.RemoteException;

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

Remote method invocations can fail in many additional ways compared to local method invocations (such as network-related communication problems and server problems), and remote methods will report such failures by throwing a `java.rmi.RemoteException`.

Implement the server

A "server" class, in this context, is the class which has a `main` method that creates an instance of the remote object implementation, exports the remote object, and then binds that instance to a name in a Java RMI *registry*. The class that contains this `main` method could be the implementation class itself, or another class entirely.

In this example, the `main` method for the server is defined in the class `Server` which also implements the remote interface `Hello`. The server's `main` method does the following:

- Create and export a remote object
- Register the remote object with a Java RMI registry

Here is the source code for the class `Server`. Descriptions for writing this server class follow the source code:

```
package example.hello;

import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class Server implements Hello {
```

```
    public Server() {}

    public String sayHello() {
        return "Hello, world!";
    }

    public static void main(String args[]) {

        try {
            Server obj = new Server();
            Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);

            // Bind the remote object's stub in the registry
            Registry registry = LocateRegistry.getRegistry();
            registry.bind("Hello", stub);

            System.err.println("Server ready");
        } catch (Exception e) {
            System.err.println("Server exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

The implementation class `Server` implements the remote interface `Hello`, providing an implementation for the remote method `sayHello`. The method `sayHello` does not need to declare that it throws any exception because the method implementation itself does not throw `RemoteException` nor does it throw any other checked exceptions.

Note: A class can define methods not specified in the remote interface, but those methods can only be invoked within the virtual machine running the service and cannot be invoked remotely.

Create and export a remote object

The `main` method of the server needs to create the remote object that provides the service. Additionally, the remote object must be *exported* to the Java RMI runtime so that it may receive incoming remote calls. This can be done as follows:

```
Server obj = new Server();
Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);
```

The static method `UnicastRemoteObject.exportObject` exports the supplied remote object to receive incoming remote method invocations on an anonymous TCP port and returns the stub for the remote object to pass to clients. As a result of the `exportObject` call, the runtime may begin to listen on a new server socket or may use a shared server socket to accept incoming remote calls for the remote object. The returned stub implements the same set of remote interfaces as the remote object's class and contains the host name and port over which the remote object can be contacted.

Note: As of the J2SE 5.0 release, stub classes for remote objects no longer need to be pregenerated using the `rmic` stub compiler, unless the remote object needs to support clients running in pre-5.0 VMs. If your application needs to support such clients, you will need to generate stub classes for the remote objects used in the application and deploy those stub classes for clients to download. For details on how to generate stub classes, see the tools documentation for `rmic` [Solaris, Windows]. For details on how to deploy your application along with pregenerated stub classes, see the codebase tutorial.

Register the remote object with a Java RMI registry

For a caller (client, peer, or applet) to be able to invoke a method on a remote object, that caller must first obtain a stub for the remote object. For bootstrapping, Java RMI provides a registry API for applications to bind a name to a remote object's stub and for clients to look up remote objects by name in order to obtain their stubs.

A Java RMI registry is a simplified name service that allows clients to get a reference (a stub) to a remote object. In general, a registry is used (if at all) only to locate the first remote object a client needs to use. Then, typically, that first object would in turn provide application-specific support for finding other objects. For example, the reference can be obtained as a parameter to, or a return value from, another remote method call. For a discussion on how this works, please take a look at *Applying the Factory Pattern to Java RMI*.

Once a remote object is registered on the server, callers can look up the object by name, obtain a remote object reference, and then invoke remote methods on the object.

The following code in the server obtains a stub for a registry on the local host and default registry port and then uses the registry stub to bind the name "Hello" to the remote object's stub in that registry:

```
Registry registry = LocateRegistry.getRegistry();
registry.bind("Hello", stub);
```

The static method `LocateRegistry.getRegistry` that takes no arguments returns a stub that implements the remote interface `java.rmi.registry.Registry` and sends invocations to the registry on server's local host on the default registry port of `1099`. The `bind` method is then invoked on the `registry` stub in order to bind the remote object's stub to the name "`Hello`" in the registry.

Note: The call to `LocateRegistry.getRegistry` simply returns an appropriate stub for a registry. The call does not check to see if a registry is actually running. If no registry is running on TCP port 1099 of the local host when the `bind` method is invoked, the server will fail with a `RemoteException`.

Implement the client

The client program obtains a stub for the registry on the server's host, looks up the remote object's stub by name in the registry, and then invokes

the `sayHello` method on the remote object using the stub.

Here is the source code for the client:

```
package example.hello;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {

    private Client() {}

    public static void main(String[] args) {

        String host = (args.length < 1) ? null : args[0];
        try {
            Registry registry = LocateRegistry.getRegistry(host);
            Hello stub = (Hello) registry.lookup("Hello");
            String response = stub.sayHello();
            System.out.println("response: " + response);
        } catch (Exception e) {
            System.err.println("Client exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

This client first obtains the stub for the registry by invoking the static `LocateRegistry.getRegistry` method with the hostname specified on the command line. If no hostname is specified, then `null` is used as the hostname indicating that the local host address should be used.

Next, the client invokes the remote method `lookup` on the registry stub to obtain the stub for the remote object from the server's registry.

Finally, the client invokes the `sayHello` method on the remote object's stub, which causes the following actions to happen:

- The client-side runtime opens a connection to the server using the host and port information in the remote object's stub and then serializes the call data.
- The server-side runtime accepts the incoming call, dispatches the call to the remote object, and serializes the result (the reply string "Hello, world!") to the client.
- The client-side runtime receives, deserializes, and returns the result to the caller.

The response message returned from the remote invocation on the remote object is then printed to `System.out`.

Compile the source files

The source files for this example can be compiled as follows:

```
javac -d destDir Hello.java Server.java Client.java
```

where ***destDir*** is the destination directory to put the class files in.
Note: If the server needs to support clients running on pre-5.0 VMs, then a stub class for the remote object implementation class needs to be pregenerated using the `rmic` compiler, and that stub class needs to be made available for clients to download. See the codebase tutorial for more details.

Start the Java RMI registry, server, and client

To run this example, you will need to do the following:

- Start the Java RMI registry
- Start the server
- Run the client

Start the Java RMI registry

To start the registry, run the `rmiregistry` command on the server's host. This command produces no output (when successful) and is typically run in the background. For more information, see the tools documentation for `rmiregistry` [Solaris, Windows].

For example, on the Solaris(tm) Operating System:

```
rmiregistry &
```

Or, on Windows platforms:

```
start rmiregistry
```

By default, the registry runs on TCP port 1099. To start a registry on a different port, specify the port number from the command line. For example, to start the registry on port 2001 on a Windows platform:

```
start rmiregistry 2001
```

If the registry will be running on a port other than 1099, you'll need to specify the port number in the calls to `LocateRegistry.getRegistry` in

the `Server` and `Client` classes. For example, if the registry is running on port 2001 in this example, the call to `getRegistry` in the server would be:

```
Registry registry = LocateRegistry.getRegistry(2001);
```

Start the server

To start the server, run the `Server` class using the `java` command as follows:

On the Solaris Operating System:

```
java -classpath classDir -Djava.rmi.server.codebase=file:classDir/ example.hello.Server &
```

On Windows platforms:

```
start java -classpath classDir -Djava.rmi.server.codebase=file:classDir/ example.hello.Server
```

where ***classDir*** is the root directory of the class file tree (see ***destDir*** in the section "Compiling the source files"). Setting the `java.rmi.server.codebase` system property ensures that the registry can load the remote interface definition (note that the trailing slash is important); for more information about using this property, see the codebase tutorial.

The output from the server should look like this:

```
Server ready
```

The server remains running until the process is terminated by the user (typically by killing the process).

Run the client

Once the server is ready, the client can be run as follows:

```
java  -classpath classDir example.hello.Client
```

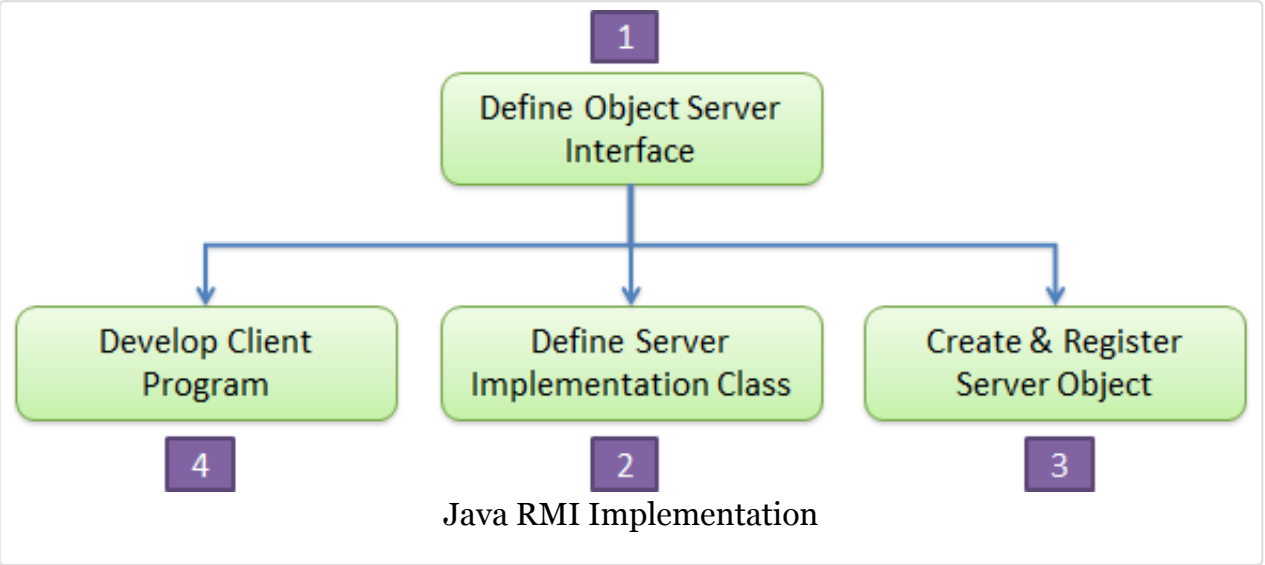
where ***classDir*** is the root directory of the class file tree (see ***destDir*** in the section "Compiling the source files").

The output from the client is the following message:

```
response: Hello, world!
```

How to Implement Java RMI?

10 Mar 2011 by Nguonly Taing



Java RMI Implementation

Implement Java RMI could be ever easier if you follow my rule strictly. You need to understand Java RMI Overview before you can catch the code.

According to the above figure, there are 4 steps need to be involved in Java RMI development. I will give you an example of creating Java RMI application dealing with Fibonacci and Factorial calculation using badly recursive algorithm. These two methods will be available for invocation at the server side. I will call the server program as PowerServer and client program as PowerClient.

Step 1: Define Object Server Interface

- Extend the java.rmi.Remote class
- Declare methods with throwing RemoteException

IPower.java

```
1  import java.rmi.*;
2  /**
3   *
4   * @author http://lycog.com
5   */
6
7  //Define interface by extending Remote class
8  public interface IPower extends Remote{
9    //Declare available methods and must throw RemoteException
10   long fibonacci(int n) throws RemoteException;
11   long factorial(int n) throws RemoteException;
12 }
```

Step 2: Define Server Implementation Class

- Extend UnicastRemoteObject
- Implement interface done in Step 1
- Throw RemoteException on constructor

PowerImplementation.java

```java
import java.rmi.RemoteException;
import java.rmi.server.ServerNotActiveException;
import java.rmi.server.UnicastRemoteObject;

/**
 *
 * @author http://lycog.com
 */
public class PowerImplementation extends UnicastRemoteObject
                implements IPower{

  //Throw RemoteException on constructor
  public PowerImplementation() throws RemoteException{

  }

  //Overwrite IPower interface
  public long fibonacci(int n) throws RemoteException{
    //Log connection from a client
    try {
      System.out.println("Got invocation from " + getClientHost());
    } catch (ServerNotActiveException ex) {
      ex.printStackTrace();
    }

    return calculateFibonacci(n);
  }

  //Overwrite IPower interface
  public long factorial(int n) throws RemoteException{
    //Log connection from a client
    try {
      System.out.println("Got invocation from " + getClientHost());
    } catch (ServerNotActiveException ex) {
      ex.printStackTrace();
    }

    return calculateFactorial(n);
  }

  //Fibonacci calculation
  private long calculateFibonacci(int n){
    if (n == 0) {
      return 0L;
    }
    if (n == 1) {
      return 1L;
    }

    return (calculateFibonacci(n - 1) + calculateFibonacci(n - 2));
  }

  //Factorial calculation
  private long calculateFactorial(int n){
    if(n<=0) return 1L;

    return n*calculateFactorial(n-1);
  }
}
```

Step 3: Create and Register Server Object

- Instantiate server object implemented in step 2
- Get reference to registry service
- Register server object to registry service

PowerServer.java

```
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

/**
 *
 * @author http://lycog.com
 */
public class PowerServer {
  public static void main(String[] args){
    try{
      //Create server object
      PowerImplementation power = new PowerImplementation();

      //Reference to registry service by creating registry service
      Registry registry = LocateRegistry.createRegistry(1099);

      //Register server object to registry with unique name
      registry.rebind("PowerObject", power);

      System.out.println("Server starts...");
    }catch(RemoteException re){
      re.printStackTrace();
    }
  }
}
```

Step 4: Develop Client Program

- Get reference to server's registry
- Locate server object from server's registry service

PowerClient.java

```
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

/**
 *
 * @author http://lycog.com
 */
public class PowerClient {
  public static void main(String[] args){
    try{
      //Get reference from server's registry
      Registry registry = LocateRegistry.getRegistry("127.0.0.1");

      //Lookup server object from server's registry
      IPower power_proxy = (IPower)registry.lookup("PowerObject");

      //Invoke server object's methods
      System.out.println("Fibonacci(10)=" + power_proxy.fibonacci(10));
      System.out.println("Factorial(10)=" + power_proxy.factorial(10));
    }catch(NotBoundException nbe){
      nbe.printStackTrace();
    }catch(RemoteException re){
      re.printStackTrace();
    }
  }
}
```

The **RMI** (Remote Method Invocation) is an API that provides a mechanism to create distributed application in java. The RMI allows an object to invoke methods on an object running in another JVM.

The RMI provides remote communication between the applications using two objects *stub* and*skeleton*.

Understanding stub and skeleton

RMI uses stub and skeleton object for communication with the remote object.

A **remote object** is an object whose method can be invoked from another JVM. Let's understand the stub and skeleton objects:

stub

The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the following tasks:

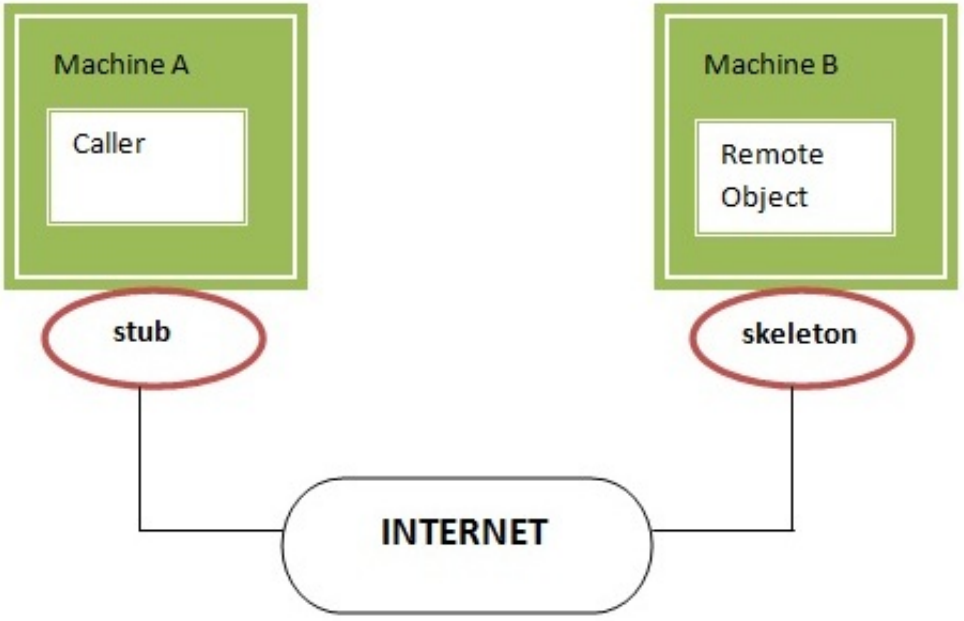
1. It initiates a connection with remote Virtual Machine (JVM),
2. It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
3. It waits for the result
4. It reads (unmarshals) the return value or exception, and
5. It finally, returns the value to the caller.

skeleton

The skeleton is an object, acts as a gateway for the server side object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks:

1. It reads the parameter for the remote method
2. It invokes the method on the actual remote object, and
3. It writes and transmits (marshals) the result to the caller.

In the Java 2 SDK, an stub protocol was introduced that eliminates the need for skeletons.



Understanding requirements for the distributed applications

If any application performs these tasks, it can be distributed application.

.

1. The application need to locate the remote method
2. It need to provide the communication with the remote objects, and
3. The application need to load the class definitions for the objects.

The RMI application have all these features, so it is called the distributed application.

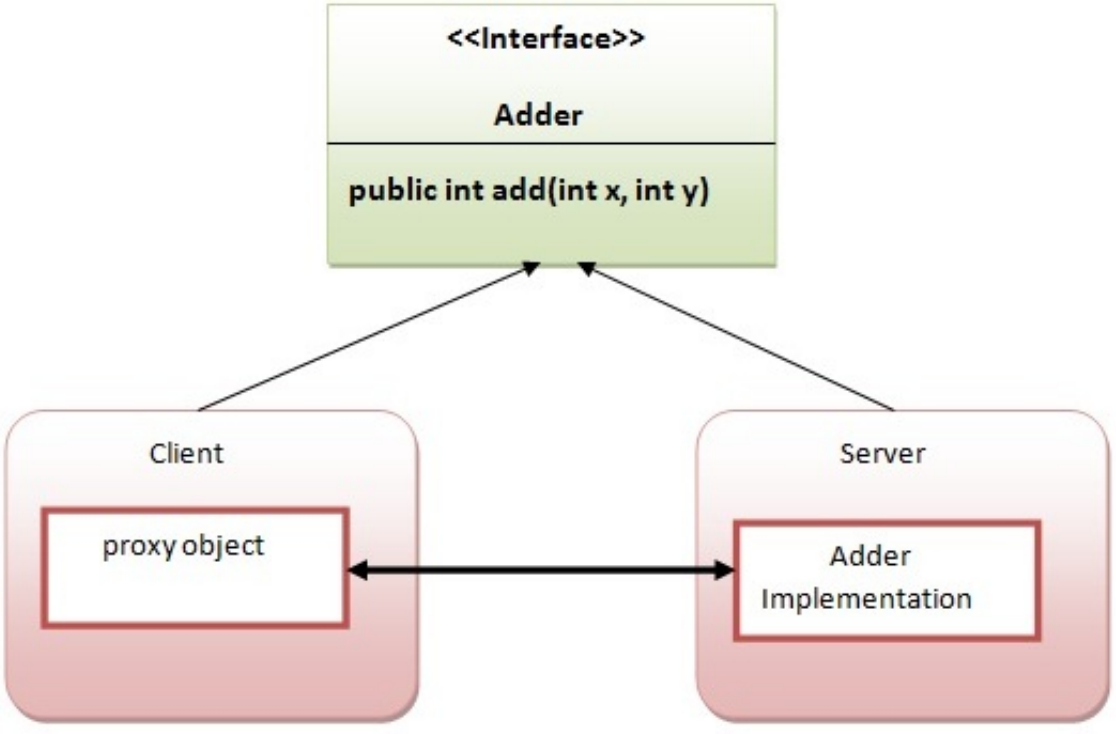
Java RMI Example

The is given the 6 steps to write the RMI program.

1. Create the remote interface
2. Provide the implementation of the remote interface
3. Compile the implementation class and create the stub and skeleton objects using the rmic tool
4. Start the registry service by rmiregistry tool
5. Create and start the remote application
6. Create and start the client application

RMI Example

In this example, we have followed all the 6 steps to create and run the rmi application. The client application need only two files, remote interface and client application. In the rmi application, both client and server interacts with the remote interface. The client application invokes methods on the proxy object, RMI sends the request to the remote JVM. The return value is sent back to the proxy object and then to the client application.



1) create the remote interface

For creating the remote interface, extend the Remote interface and declare the RemoteException with all the methods of the remote interface. Here, we are creating a remote interface that extends the Remote interface. There is only one method named add() and it declares RemoteException.

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

```
public interface Adder extends Remote{
public int add(int x,int y)throws RemoteException;
}
```

2) Provide the implementation of the remote interface

Now provide the implementation of the remote interface. For providing the implementation of the Remote interface, we need to

- Either extend the UnicastRemoteObject class,
- or use the exportObject() method of the UnicastRemoteObject class

In case, you extend the UnicastRemoteObject class, you must define a constructor that declares RemoteException.

```
import java.rmi.*;
import java.rmi.server.*;
public class AdderRemote extends UnicastRemoteObject implements Adder{
AdderRemote()throws RemoteException{
super();
}
public int add(int x,int y){return x+y;}
}
```

3) create the stub and skeleton objects using the rmic tool.

Next step is to create stub and skeleton objects using the rmi compiler. The rmic tool invokes the RMI compiler and creates stub and skeleton objects.

```
rmic AdderRemote
```

4) Start the registry service by the rmiregistry tool

Now start the registry service by using the rmiregistry tool. If you don't specify the port number, it uses a default port number. In this example, we are using the port number 5000.

```
rmiregistry 5000
```

5) Create and run the server application

Now rmi services need to be hosted in a server process. The Naming class provides methods to get and store the remote object. The Naming class provides 5 methods.

1. **public static java.rmi.Remote lookup(java.lang.String) throws java.rmi.NotBoundException, java.net.MalformedURLException, java.rmi.RemoteException;** it returns the reference of the remote object.
2. **public static void bind(java.lang.String, java.rmi.Remote) throws java.rmi.AlreadyBoundException, java.net.MalformedURLException, java.rmi.RemoteException;** it binds the remote object with the given name.
3. **public static void unbind(java.lang.String) throws java.rmi.RemoteException, java.rmi.NotBoundException, java.net.MalformedURLException;** it destroys the remote object which is bound with the given name.
4. **public static void rebind(java.lang.String, java.rmi.Remote) throws java.rmi.RemoteException, java.net.MalformedURLException;** it binds the remote object to the new name.
5. **public static java.lang.String[] list(java.lang.String) throws java.rmi.RemoteException, java.net.MalformedURLException;** it returns an array of the names of the remote objects bound in the registry.

In this example, we are binding the remote object by the name sonoo.

```
import java.rmi.*;
import java.rmi.registry.*;
public class MyServer{
public static void main(String args[]){
try{
Adder stub=new AdderRemote();
Naming.rebind("rmi://localhost:5000/sonoo",stub);
}catch(Exception e){System.out.println(e);}
}
}
```

6) Create and run the client application

At the client we are getting the stub object by the lookup() method of the Naming class and invoking the method on this object. In this example, we are running the server and client applications, in the same machine so we are using localhost. If you want to access the remote object from another machine, change the localhost to the host name (or IP address) where the remote object is located.

```
import java.rmi.*;
public class MyClient{
public static void main(String args[]){
try{
Adder stub=(Adder)Naming.lookup("rmi://localhost:5000/sonoo");
System.out.println(stub.add(34,4));
}catch(Exception e){}
}
}
```

download this example of rmi

```
For running this rmi example,

1) compile all the java files

javac *.java

2)create stub and skeleton object by rmic tool

rmic AdderRemote

3)start rmi registry in one command prompt

rmiregistry 5000

4)start the server in another command prompt

java MyServer

5)start the client application in another command prompt

java MyClient
```