RMI

Verteilte Anwendungen analog zu nicht verteilten Anwendungen

(Verteilt heißt: verschiedene JVMs beteiligt!)

Design Prinzip

Trennung von Interface und Implementierung

Interface => Client

Implementierung via Stub (Proxy Design Pattern)

Implementierung => Server

Kommunikation

Client muß den Server kennen => JNDI

Server muß für Callbacks den Client kennen

Client muß sich registrieren!

Wie schreibt man eine RMI Anwendung?

Schritt 1

Service Interface schreiben

Interface

extends Remote

Methoden:

throws RemoteException

Schritt 2

Implementierung für die Server Seite schreiben

Class

extends UnicastRemoteObject

oder

Verwendung der statischen Methode UnicastRemoteObject.exportObject

Schritt 3

Erzeugen von Stubs (und Skeletons) via rmic

Schritt 4

RMI Server schreiben / übersetzen

Schritt 5

RMI Client schreiben /übersetzen

Schritt 6

rmiregistry starten (CLASSPATH!)

Schritt 7

Server starten

Schritt 8

Client starten

```
package calculator;

import java.rmi.Remote;

public interface Calculator extends 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;
}
```

```
package calculator;
import java.rmi.server.UnicastRemoteObject;

public class CalculatorImpl
extends UnicastRemoteObject
implements Calculator {

  public CalculatorImpl()
                throws java.rmi.RemoteException {
                super();
        }
        public long add(long a, long b)
                throws java.rmi.RemoteException {
                return a + b;
        }

        …

}
```

```
package calculator;
import java.rmi.Naming;

public class CalculatorServer {

  public CalculatorServer() {
    try {
      Calculator c = new CalculatorImpl();
      Naming.rebind
("rmi://localhost:1099/CalculatorService", c);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public static void main(String args[]) {
    new CalculatorServer();
  }

}
```

```
package calculator;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.net.MalformedURLException;
import java.rmi.NotBoundException;

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

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

Was muß auf dem Server verfügbar sein?

- **Remote service interface definitions**
- **Remote service implementations**
- **Skeletons for the implementation classes (JDK 1.1 only)**
- **Stubs for the implementation classes**
- **All other server classes**

Was muß auf dem Client verfügbar sein?

- **Remote service interface definitions**
- **Stubs for the remote service implementation classes**
- **Server classes for objects used by the client (such as return values)**
- **All other client classes**

Parameterübergabe

Lokale JVM

Einfache Datentypen:

call / return by value

Objekte:

Referenzen werden anstelle von Werten übergeben

RMI

Einfache Datentypen:

call / return by value

Objekte:

Objekt Serialisierung

Remote Objekte:

Proxies