Netzprogrammierung
5. Verteilte Objekte in Java RMI

Prof. Dr.-Ing. Robert Tolksdorf
Freie Universität Berlin
Institut für Informatik
Netzbasierte Informationssysteme
mailto: tolk@inf.fu-berlin.de
http://www.robert-tolksdorf.de

Überblick

1. Verteilte Objekte / RMI
2. Objektreferenzen
3. Callbacks
4. Code nachladen
5. Serialisierung
6. Threads und RMI

Verteilte Objekte / RMI

Verteilte Objekte

- RPC übertragen auf OO-Welt:
  Entfernter Methodenaufruf
- Interaktionsmuster in OO-Sprachen:
  - Objekte tauschen Mitteilungen aus
  - Beim Empfang einer Mitteilung führt ein Objekt eine Methode aus und schickt eventuelle Ergebnisse
  - Modell sagt nichts über Verteilung aus
- Rollen der Partner
  - Aufrufer – Aufgerufener
  - Dienstnutzer – Dienstanforderer
  - Client – Server
- Verteilte Objekte in Java:
  Remote Method Invocation, RMI

RMI vs. Sockets

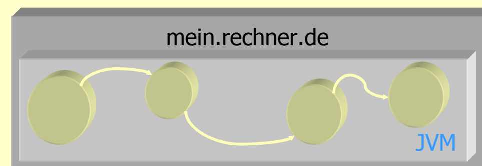
- RMI
  - Höheres Abstraktionsniveau
  - Übermittlung getypter Daten
  - Klassenübermittlung
  - …
- Sockets
  - Kleinster gemeinsamer Nenner des Internets
  - Übermittlung ungetypter Byteströme
  - Effizienter
  - …

Lokale vs. verteilte Objekte

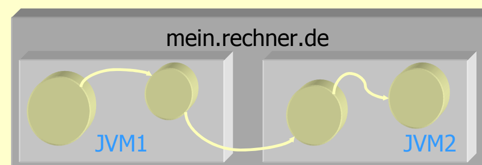
Lokales Objektmodell	*Verteiltes Objektmodell*
Aufruf an Objekten	Aufruf an Interfaces
Parameter und Ergebnisse als Referenzen	Parameter und Ergebnisse als Kopien
Alle Objekte fallen zusammen aus	Einzelne Objekte fallen aus
Keine Fehlersemantik	Komplizierte Fehlersemantik (Referenzintegrität, Netzfehler, Sicherheit etc.)
...	

Lokale vs. verteilte Java-Programme

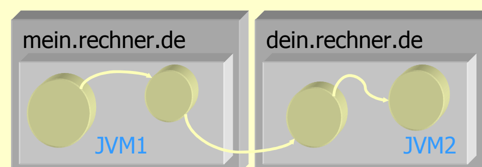
- Ein Java Programm arbeitet in einer virtuellen Java Maschine (JVM)



- Zwischen JVMs können mit *Remote Method Invocation* Methoden an Objekten aufgerufen werden



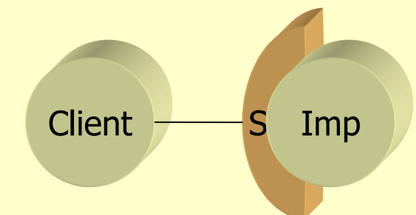
- JVMs können auf unterschiedlichen Internet-Rechnern laufen



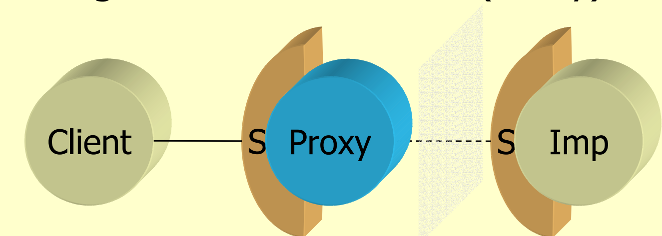
- Sie müssen es aber nicht...

Schnittstellen

- Objektschnittstellen definieren Methoden des Objekts



- Unterschiedliche Implementierungen für gleiche Schnittstelle S
- Modulschnittstellen des RPC werden auf Objektschnittstellen abgebildet
- Aufrufweiterleitung durch Stellvertreter (Proxy)



Lokales Zählerobjekt

- Lokale nutzbares Objekt:

```
class LocalCounterImplementation {
    int counter;
    public LocalCounter() {  }
    public void add(Integer i) {
      counter+=i.intValue();
    }
    public Integer value() {
      return(new Integer(counter));
    }
```

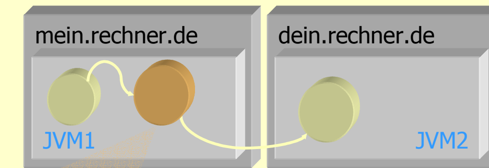
- Könnte auch folgende Schnittstelle implementieren:

```
public interface LocalCounter {
  /* Addieren */
  public void add(Integer i);
  /* Abfragen */
  public Integer value();
}
```

- Aufruf: c.add(new Integer(10));

Entferntes Zählerobjekt

- Auf Aufruferseite wird mit einem Interface gesprochen:

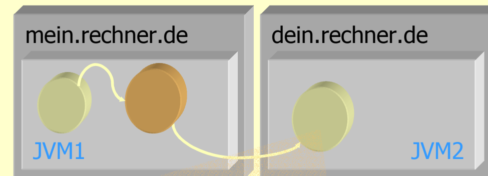



```
public interface Counter extends java.rmi.Remote {
  /* Addieren */
  public void add(Integer i) throws java.rmi.RemoteException;
  /* Abfragen */
  public Integer value() throws java.rmi.RemoteException;
}
```

- Schnittstelle macht Fehlermöglichkeit explizit

RMI: Erbringerseite

- Ein Serverobjekt implementiert das Interface:




```
import java.rmi.*;
public class CounterServer extends
                              java.rmi.server.UnicastRemoteObject
  implements Counter {
    int counter;
    public CounterServer() throws java.rmi.RemoteException {  }
    public void add(Integer i) throws java.rmi.RemoteException {
        counter+=i.intValue();
    }
    public Integer value() throws java.rmi.RemoteException {
        return(new Integer(counter));
    }
```


Aufruf eines entfernten Objekts

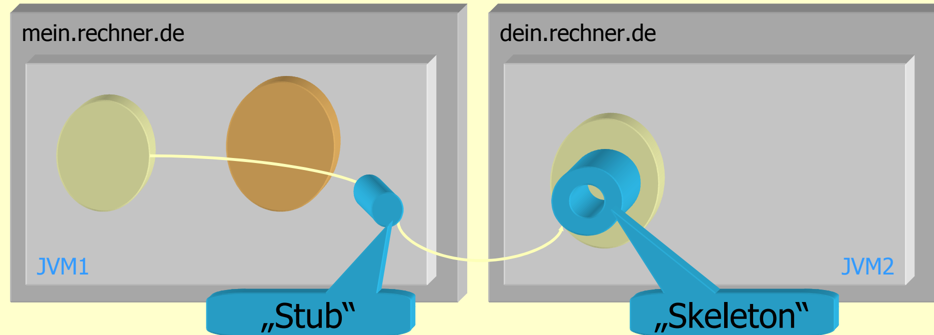
- Aufruf durch Aufruf einer Methode an Interface
- Tatsächlich wird damit eine Methode am Proxy aufgerufen:

```
Counter c;
...
 try {
     c.add(new Integer(10));
     System.out.println((c.value()).intValue());
 } catch (Exception e) {
     System.err.println(e.getMessage());
 }
```

- *Unterschied: Fehlermöglichkeit*

rmic: Compiler für Verbindungsstücke

- „Verbindungsstücke" werden automatisch erzeugt



- rmic ServerKlasse erzeugt ServerKlasse_stub.class und ServerKlasse_skel.class

Compilieren und ausführen

- Compilieren:

```
>javac Counter.java
>javac CounterServer.java
>rmic CounterServer
>javac CounterUser.java
```

- In 4 JVMs ausführen:

>rmiregistry	>java CounterServer	>java CounterUser 10
		>java CounterUser 20

Objektreferenzen

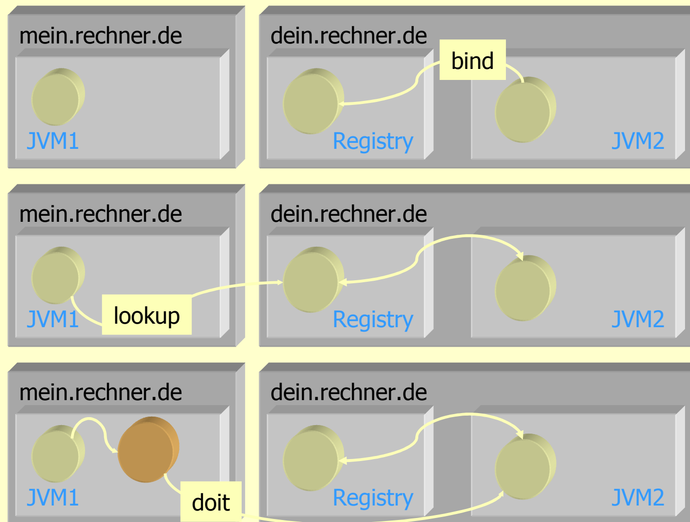
Objektreferenzen

```
Counter c;
...
 try {
     c.add(new Integer(10));
```

- Woher kennt der Aufrufer eigentlich das entfernte Objekt?
- Genauer: Wohin schickt der Proxy eigentlich den Methodenaufruf?
- RMIRegistry: „Verzeichnisdienst" für Objekte

Referenzen auf entfernte Objekte

- *Registry* Objekt liefert Referenzen auf Objekte



Beispiel: Server-Programm meldet sich an

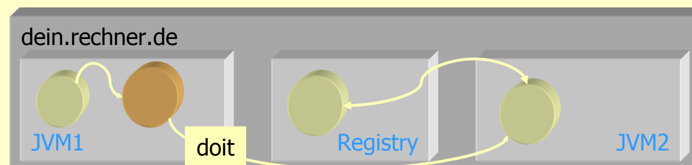
```
public class CounterServer extends java.rmi.server.UnicastRemoteObject
  implements Counter {
  //...
 public static void main(String argv[]) {
    System.setSecurityManager(new RMISecurityManager());
    try {
      CounterServer c = new CounterServer();
      Naming.bind("rmi://"+
             (InetAddress.getLocalHost()).getHostName() + "/Teilnehmer",c);
    } catch (Exception e) {
      System.err.println(e.getMessage());
    }
  }
}
```

- URL-Schema: rmi://*rechner*/*pfad*

Client Programm erhält Referenz

```
import java.net.*;
public class CounterUser {
  public static void main(String argv[]) {
    try {
      Counter c = (Counter)java.rmi.Naming.lookup("rmi://" +
            (InetAddress.getLocalHost()).getHostName() + "/Teilnehmer");
      c.add(new Integer(10));
      System.out.println((c.value()).intValue());
    } catch (Exception e) {
      System.err.println(e.getMessage());
    }
  }
}
```

Hier nur lokale Registry benutzt



rmiregistry

- JDK Programm rmiregistry ist RMI-Objekt für Interface java.rmi.registry.Registry
- Methoden
  - void bind(String name, Remote obj)
    void rebind(String name, Remote obj)
    Binden eines Objektes unter einem Namen
  - Remote lookup(String name)
    Referenz auf gebundenen Objekt erfragen
  - void unbind(String name)
    Bindung löschen
  - String[] list()
    Bindungen abfragen

java.rmi.Naming

- Woher bekommt man eigentlich die Referenz auf ein Registry-Objekt?
- Registry-Objekt der lokalen Maschine über statische Methoden der Klasse java.rmi.Naming zugänglich
  - void java.rmi.Naming.bind(String name, Remote obj)
  - void java.rmi.Naming. rebind(String name, Remote obj)
  - Remote java.rmi.Naming. lookup(String name)
  - void java.rmi.Naming. unbind(String name)
  - String[] java.rmi.Naming. list()

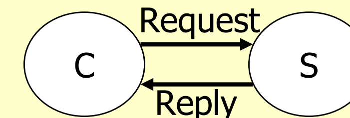
java.rmi.registry.LocateRegistry

- Woher bekommt man eigentlich die Referenz auf ein Registry-Objekt?
- Registry-Objekte auf entfernten Maschinen über statische Methoden der Klasse java.rmi.Naming zugänglich
  - Registry getRegistry()
    Lokale Registry auf Port
    java.rmi.registry.Registry.REGISTRY_PORT:
  - Registry getRegistry(int port)
    Lokale Registry auf einem anderen Port
  - Registry getRegistry(String host)
    Registry auf Rechner host
  - Registry getRegistry(String host, int port)
    Registry auf Rechner host, Port port

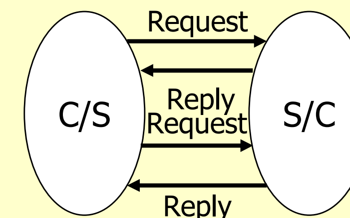
Callbacks

Callbacks

- Zwischen Client und Server-Objekt ist eventuell ein komplexeres Protokoll notwendig
- Client/Server folgt Anfrage/Antwort Protokoll



- Für Anfrage/Nachfrage/Antwort Protokoll muss Client selber zum Server werden



Callbacks

- Nachfrage: *Callback* Methode des Clienten Objekts
- In RMI:
  - Auch Client wird ein Server-Objekt
  - Übergibt Referenz auf sich bei Anfrage

Schnittstellen

- Server:

```
public interface RMIPongServerInterface
      extends java.rmi.Remote {
      public void pong(RMIPongClientInterface theClient)
              throws java.rmi.RemoteException ;
}
```

- Client:

```
public interface RMIPongClientInterface
     extends java.rmi.Remote {
     public boolean more()
              throws java.rmi.RemoteException ;
}
```


Server

```
class RMIPongServer extends java.rmi.server.UnicastRemoteObject
   implements RMIPongServerInterface {

   RMIPongServer() throws java.rmi.RemoteException  {}

   public void pong(RMIPongClientInterface theClient)
                  throws java.rmi.RemoteException {
         while (theClient.more()) { System.out.println("Pong"); }
   }

   public static void main(String[] argv) throws
                  java.rmi.RemoteException,java.rmi.AlreadyBoundException,
                  java.net.MalformedURLException {
         RMIPongServer pongServer = new RMIPongServer();
         java.rmi.Naming.bind("rmi://localhost/pong",pongServer);
   }
}
```


Client

```
class RMIPongClient extends java.rmi.server.UnicastRemoteObject
   implements RMIPongClientInterface {
   int moreCounter = 0;
   RMIPongClient() throws java.rmi.RemoteException  {}

   public boolean more() throws java.rmi.RemoteException {
         return ((moreCounter++)<5);
   }

   public static void main(String[] argv)  throws java.rmi.RemoteException,
            java.rmi.NotBoundException, java.net.MalformedURLException {
         RMIPongClient pongClient = new RMIPongClient();
         RMIPongServerInterface pongServer =
               (RMIPongServerInterface)
                     java.rmi.Naming.lookup("rmi://localhost/pong");
         pongServer.pong(pongClient);
   }
}
```

Beispiel: Chat System

Chat Server mit RMI

- Aufgabe: Schreiben Sie ein System, mit dem „gechattet" werden kann
- Idee: Es gibt ein Server-Objekt, das folgende Methoden beherrscht:
  - void register(ChatClient aClient, String name)
    Client registrieren (erhält dann alle Mitteilungen)
  - void unregister(ChatClient aClient)
    Client abmelden
  - void tell(String message)
    Mitteilung an alle
  - public String[] history()
    Alle bisherigen Mitteilungen erfragen

Chat Server mit RMI

- Client Objekte sollen sich anmelden und erhalten einen Callback Aufruf von
  void show(String message)
  beim Vorliegen einer neuen Mitteilung

Schnittstellen

```
public interface ChatServer extends java.rmi.Remote {

  public void register(ChatClient aClient, String name) throws
    java.rmi.RemoteException;
  public void unregister(ChatClient aClient) throws java.rmi.RemoteException;
  public void tell(String message) throws java.rmi.RemoteException;
  public String[] history() throws java.rmi.RemoteException;
}


public interface ChatClient extends java.rmi.Remote {
  public void show(String message) throws java.rmi.RemoteException;
}
```

Server

```
import java.rmi.*;
import java.net.*;
import java.util.*;

public class Server extends java.rmi.server.UnicastRemoteObject
implements ChatServer {
  Vector messages = new Vector();
  Vector clients = new Vector();

  public Server() throws java.rmi.RemoteException {}
```


Server

```
public void register(ChatClient aClient, String name) throws
    java.rmi.RemoteException {
    clients.addElement(new Chatter(aClient,name));
    tell("** "+name+" chattet jetzt mit");
  }

public void unregister(ChatClient aClient) throws java.rmi.RemoteException {
    for (Enumeration e=clients.elements(); e.hasMoreElements();) {
      Chatter c = (Chatter) e.nextElement();
      if (c.client.equals(aClient)) {
           clients.removeElement(c);
      }
    }
  }
```


Server

```
public void tell(String message) throws
    java.rmi.RemoteException {
    for (Enumeration e=clients.elements();
    e.hasMoreElements();) {
      Chatter c=(Chatter) e.nextElement();
      c.client.show(message);
    }
    messages.addElement(message);
  }
public String[] history() throws java.rmi.RemoteException {
    String history[] = new String[messages.size()];
    messages.copyInto(history);
    return(history);
  }
```


Server

```
public static void main(String[] argv) {
    System.setSecurityManager(new RMISecurityManager());
    try {
      Server chatter = new Server();
      Naming.bind("rmi://"+  (InetAddress.getLocalHost()).getHostName() +
            "/chatter",chatter);
    } catch (Exception e) {
      System.err.println(e.getMessage());
    }
  }
}
```

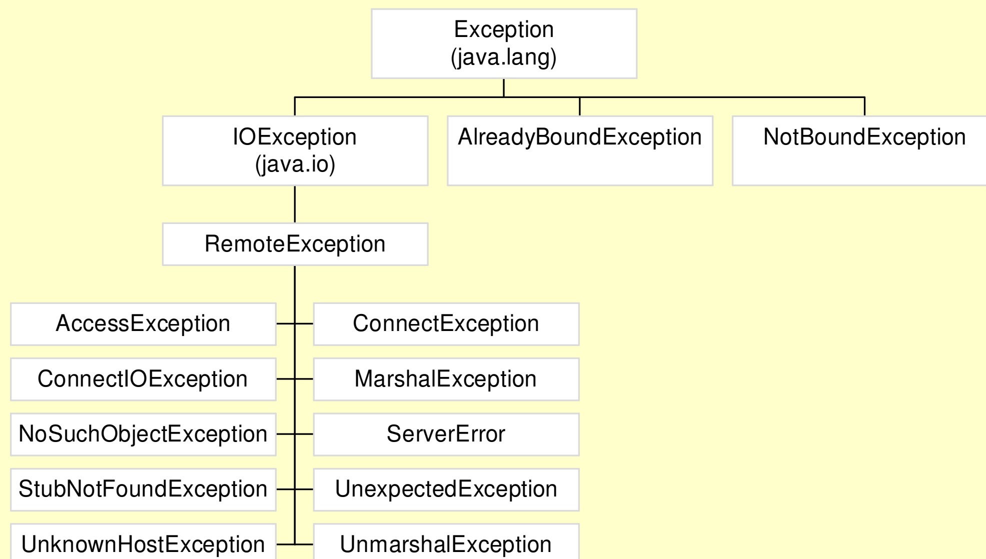
Client

```
import java.net.*;
public class Client extends java.rmi.server.UnicastRemoteObject
    implements ChatClient, java.io.Serializable  {
  TextWindow window;
  ChatServer cServer;
  public Client(String name) throws java.rmi.RemoteException,
    java.rmi.NotBoundException, java.net.MalformedURLException,
     java.net.UnknownHostException  {
    cServer = (ChatServer)java.rmi.Naming.lookup("rmi://" +
                    (InetAddress.getLocalHost()).getHostName() + "/chatter");
    window=new TextWindow();
    window.println("Willkommen zum Chatten");
    try {
      cServer.register(this,name);
      String[] history = cServer.history();
      for (int i=0; i<history.length; window.println(history[i++]));
    } catch (Exception e) {
          window.println("Konnte nicht zum Server verbinden"); }  }
```


Client

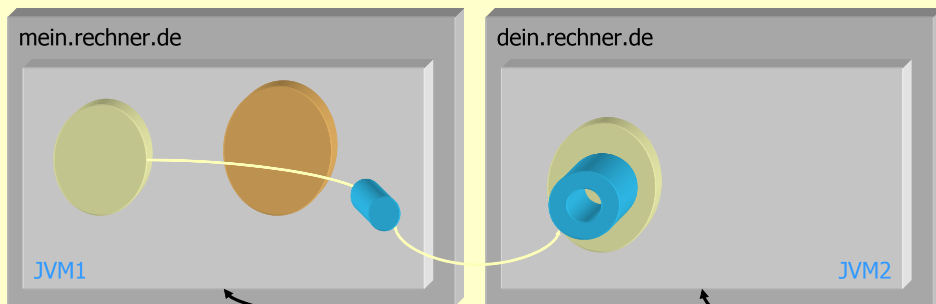
```
    public void show(String message) throws java.rmi.RemoteException {
        window.println(message);
    }
    public static void main(String[] argv) {
      try {
        String name = (argv.length>0)?argv[0]:"Chatter";
        Client cClient = new Client(name);
        while (true) {
                String message=cClient.window.getInput();
                cClient.cServer.tell(name + ": " + message);
        }
      } catch (Exception e) {
        System.err.println(e.getMessage());
      }
    }
}
```


Ausnahmen aus java.rmi



Code nachladen

Code nachladen

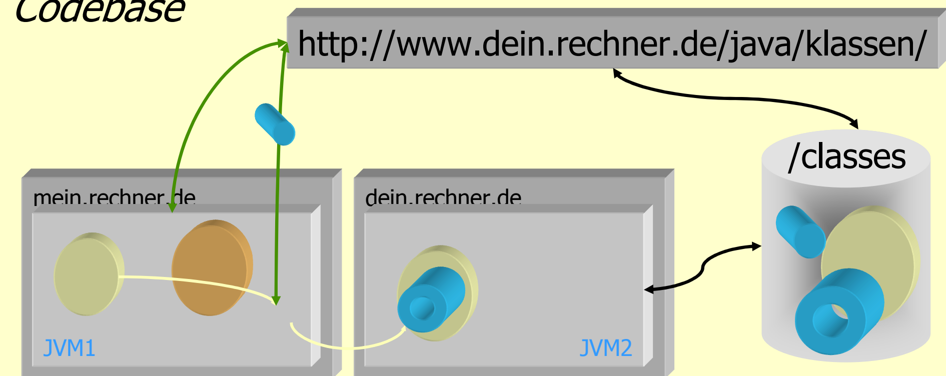


/classes

- JVM1 muss ServerKlasse_stub.class laden
- Wenn JVM1 und JVM2 im gleichen Dateisystem arbeiten, müssen sie entsprechend den CLASSPATH gesetzt haben

Nachladen über Web-Server

- Wenn JVM1 und JVM2 in getrennten Dateisystemen arbeiten nutzt CLASSPATH nichts
- ServerKlasse_stub.class muss über das Netz nachgeladen werden
- Dies geschieht durch Angabe einer Basis-URL, der *Codebase*



Die Codebase festlegen

- Eigenschaft java.rmi.server.codebase enthält URL der Codebase
- Beim Start des Serverobjekts festlegen:

```
>java –Djava.rmi.server.codebase=
   http://www.dein.rechner.de/java/klassen/
   CounterServer
```

- Im Programm festelegen:
  System.setProperty("java.rmi.server.codebase",codebase);
- Codebase wird beim Registry-Eintrag vermerkt und beim lookup an Clienten übermittelt
- Codebase
  - CLASSPATH bezeichnet „lokale“ Codebase
  - java.rmi.server.codebase entfernte Codebase

Client auch mit Codebase

- Auch Client kann Codebase anbieten (müssen):
  - Server-Objekt bietet void $m(T_1\ t_1)$ an
  - Client-Objekt ruft $m(t_2)$ auf mit T_2 als Unterklasse von T_1
  - T2.class muss geladen werden
  - Vergleichsweise mit Interfaces
- Also auch

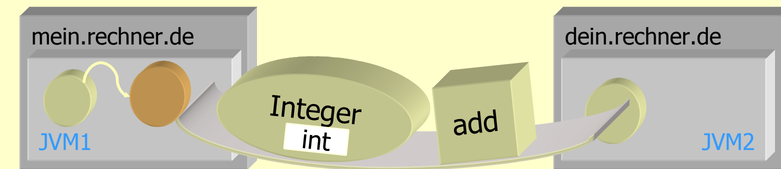
```
>java –Djava.rmi.server.codebase=
   http://www.mein.rechner.de/classes/  AClient
```

- Codebase kann auch
  - jar-File sein
  - an mehreren Orten liegen
  - java.rmi.server.codebase=
    "http://www.mein.rechner.de/classes/
    http://www.dein.rechner.de/java/klassen/
    http://www.unser.rechner/unsereklasse.jar"

Serialisierung und Sicherheit

Objektserialisierung

- Parameter und Ergebnisse müssen übertragen werden:



- Das Erstellen einer seriellen Repräsentation eines Objekts ist die Serialisierung (ggs: Deserialisierung)
- Objektserialisierung notwendig zum Versenden oder Speichern von *Objekten*
- Instanzwerte + Klasse = Objektrepräsentation

Automatische Serialisierung

- Beispiel:

```
class Person {          class Street {
  String name;                 String name;
  Street adress;               int number;
}                              }
```

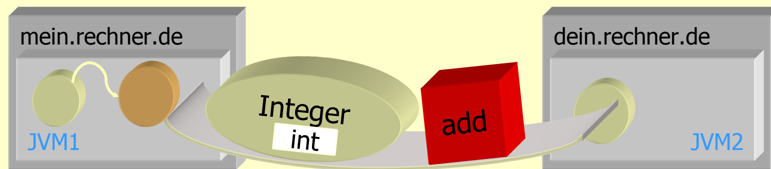
- Serialisiert: String String int
- Automatisch, wenn Klassen die Schnittstelle java.io.Serializable implementieren (lassen)
- rmic erzeugt Code für Serialisierung und Deserialisierung durch Analyse der Klassendefinitionen

Serialisierung

- implements Serializable bedeutet: Die automatische Serialisierung kann angewandt werden
- Wird von den meisten vordefinierten Klassen unterstützt, wenn möglich
- Gegenbeispiel: Ströme
- Schreibt man RMI Objekte und bewegt Objekte als Parameter oder Ergebnis, dann müssen diese Objekte als Serializable markiert sein
- Fehlerquelle: rmic hat Serialisierungscode erzeugt und man ändert danach das Objekt-Layout: Ergibt Marshalling Fehler zur Laufzeit
- Falls eigenes Mashalling notwendig, Definition von
  - private void writeObject(java.io.ObjectOutputStream out) throws IOException
  - private void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException;

Serialisierung: Klassen nachladen

- Objekte = Daten und Verhalten
- Serialisierte Java-Objekte: Datenstrom + Klasse



- Klassen zu übermittelten Objekten nachladen, falls nicht
  - vorher nachgeladen
  - vorher schon vorhanden (java.* Klassen)
- Bedrohung durch Angreifer:
  - Bildet Unterklasse von Street, ändert dabei toString()
  - Erzeugt Objekt davon
  - Übergibt Objekt als Argument beim Methodenaufruf
  - Beim Aufruf von toString() dort wird geänderter Code ausgeführt
  - Mit allen Rechten des RMI Objekts

Serialisierung: Klassen nachladen

- *SecurityManager* wird vor sicherheitsrelevanten Aktionen in verschiedenen Gruppen (Dateien, Sockets, GUI etc.) gefragt
- Ist Objekt mit Methoden wie:
  - void checkAccept(String host, int port)
  - void checkAccess(Thread t)
  - void checkAccess(ThreadGroup g)
  - void checkAwtEventQueueAccess()
  - void checkConnect(String host, int port)
  - void checkConnect(String host, int port, Object context)
  - void checkCreateClassLoader()
  - void checkDelete(String file)
  - ...
- Methoden werfen java.lang.SecurityException falls Ausführung nicht zugelassen

Security Manager

- Installation eines Security Managers mit der Methode void System.setSecurityManager(SecurityManager s)
- Fragt eventuell schon vorhandenen SecurityManager, ob das erlaubt ist
- RMI Classloader lädt Klassen von entfernten Rechnern nur dann nach, wenn SecurityManager installiert ist
- SecurityManager ist abstrakte Klasse
  - Im JDK mitgeliefert: RMISecurityManager System.setSecurityManager(new RMISecurityManager());
  - In Browsern: Eigener SecurityManager

Policies

- Policies definieren im Detail, was erlaubt ist
- Securitymanager verwenden Policies um Rechte zu ermitteln
- Policies in einer Policy-Datei definiert
- Beispielausschnitt:

```
// Standard extensions get all permissions by default
grant codeBase "file:${java.home}/lib/ext/*" {
   permission java.security.AllPermission;
};
// default permissions granted to all domains
grant {
   // allows anyone to listen on un-privileged ports
   permission java.net.SocketPermission "localhost:1024-", "listen";
   // "standard" properies that can be read by anyone
   permission java.util.PropertyPermission "java.version", "read";
 ...
```

Erlaubnis für Socketnutzung

- Damit RMI arbeiten kann, muss es Socket-Verbindungen öffnen dürfen
- Das ist eventuell durch die lokal installierte Sicherheitspolicy verboten
  - Globale Policy-Definition bei *java.home*/lib/security/java.policy
  - Nutzerspezifische Policy-Definition bei *user.home*/.java.policy
  - System.out.println(System.getProperty("user.home"));
    System.out.println(System.getProperty("java.home"));
  - C:\Documents and Settings\tolk
    C:\Program Files\Java\j2re1.4.2_04

Erlaubnis für Socketnutzung

- Eintrag in .java.policy

```
grant { permission java.net.SocketPermission
    "*", "accept,connect,listen,resolve";
};
```

  läßt beliebige Socketverbindungen zu

- Falls in anderer Datei:
  java -Djava.security.policy=/meindir/meinepolicy
- Alternative: policytool mit GUI

Nebenläufigkeit

Threads und RMI Aufrufe

- Mehrere „gleichzeitig“eintreffende RMI Aufrufe (lokal und mehrere Netzverbindungen) *können* in Threads arbeiten
- RMI Spezifikation:
  „***3.2* Thread Usage in Remote Method Invocations**
  A method dispatched by the RMI runtime to a remote object implementation may or may not execute in a separate thread. The RMI runtime makes no guarantees with respect to mapping remote object invocations to threads. Since remote method invocation on the same remote object may execute concurrently, a remote object implementation needs to make sure its implementation is thread-safe.“
- Man muß *immer* davon ausgehen, daß Methoden nebenläufig aufgerufen werden

Threads und RMI Aufrufe

- Kann immer zu Schreibkonflikten führen:

m(int p) {	a=10;	a=10;	a=10;
a=p;	b=-10;	a=20;	a=20;
b=-p;	a=20;	b=-10;	b=-20;
}	b=-20;	b=-20;	b=-10;
m(10)||m(20)	20/-20	20/-20	20/-10

- Koordination notwendig: Synchronisierung
- Konzepte dazu -> ALP IV

Sicheres CounterServer Objekt

```
import java.rmi.*;
public class CounterServer extends java.rmi.server.UnicastRemoteObject
    implements Counter {
  protected int counter;
  public CounterServer() throws java.rmi.RemoteException {  }
  public synchronized void add(Integer i) //mögl. Synchronisationsmech.
      throws java.rmi.RemoteException {
      counter+=i.intValue();
  }

  public Integer value() throws java.rmi.RemoteException {
      return(new Integer(counter));
  }
```


Zusammenfassung

Überblick

1. Verteilte Objekte / RMI
   1. Verteilte Objekte haben anderes Verhalten als lokale
   2. Kommunikation über Schnittstellen/Proxy
2. Objektreferenzen
   1. Rmiregistry als Objektverzeichnis
3. Callbacks
   1. Clientenobjekte auch als Serverobjekte anbieten
   2. Referenz per Parameter zum Rückruf an Server übergeben
4. Codenachladen
   1. Klassencode von Webserver
   2. Codebase Eigenschaft
5. Serialisierung und Sicherheit
   1. Code nachladen
   2. Serialisierbarkeit von Objekten
   3. rmic
   4. Policies
6. Threads und RMI
   1. Abarbeitung von Aufrufen kann nebenläufig geschehen

Literatur

- Sun. Java Remote Method Invocation Specification.
  http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html
- Java Remote Method Invocation Homepage
  http://java.sun.com/products/jdk/rmi/
- Dynamic code downloading using RMI
  http://java.sun.com/j2se/1.4.2/docs/guide/rmi/codebase.html
- Default Policy Implementation and Policy File Syntax
  http://java.sun.com/j2se/1.4.2/docs/guide/security/PolicyFiles.html