



## ***Netzprogrammierung Verteilte Objekte in Java RMI II***

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



# Überblick

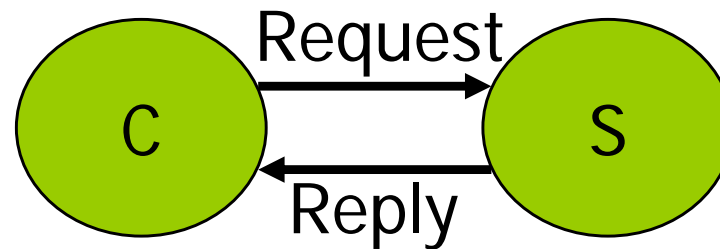
---

1. Callbacks
2. Nebenläufigkeit
3. Multiserver
4. Objektaktivierung

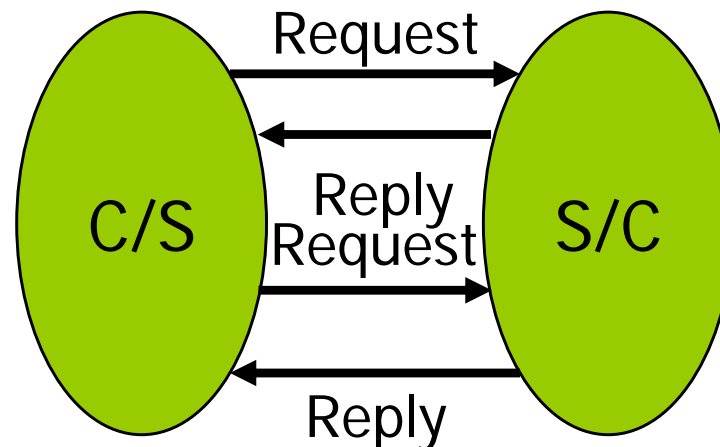


## Callbacks

- 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 ;
}
```

```
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);
    }
}
```

```
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

```
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 { }
```

```
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);
        }
    }
}
```

```
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);
}
```

```
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"); } } }
```

```
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()); }
}
}
```



## Applets und RMI

# RMI Zugriff aus Applets heraus

---

- Applets können auch auf RMI Objekte zugreifen
- Interaktion wie beim normalen RMI
- Zugriff in der Regel aber auf den Rechner beschränkt, von dem die HTML Seite geladen wurde

- Ein einfacher Zahlenaddierer

```
package adder;
import java.rmi.*;
public interface Adder extends Remote {
    public int add(int a, int b) throws RemoteException ;
}
```

- Einfache Implementierung:

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

```
public class AdderImpl extends UnicastRemoteObject implements  
    Adder {
```

```
    public AdderImpl() throws RemoteException { super(); }  
    public int add(int a, int b) throws RemoteException { return a+b;  
    }
```

```
    public static void main(String[] argv) {  
        try {  
            AdderImpl ai=new AdderImpl();  
            Naming.rebind("rmi://localhost/adder",ai);  
        } catch (Exception e) {  
            System.out.println(e.getMessage());  
        }  
    }  
}
```

## Beispiel/3

- HTML Seite

```
<head>
```

```
<title>RMI Adding...</title>
```

```
</head>
```

```
<body>
```

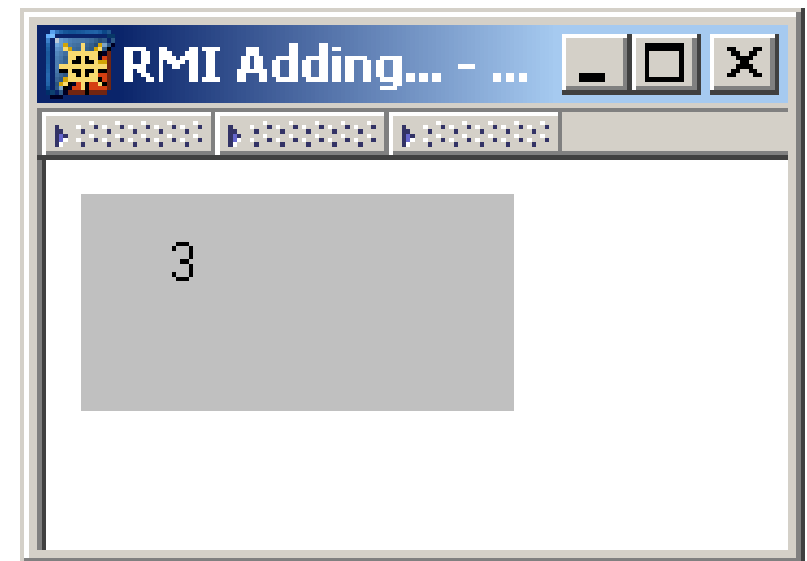
```
<applet codebase="/" code="addder.AdderApplet"
    width="100" height="50">
```

```
<param name="a" value="1">
```

```
<param name="b" value="2">
```

```
</applet>
```

```
</body>
```



# Beispiel/4 Das eigentliche Applet

```
package adder;
import java.applet.Applet;
import java.awt.Graphics;
import java.rmi.*;
public class AdderApplet extends Applet {
    Adder adder=null;
    String result="";
    public void init() {
        try {
            adder=(Adder)Naming.lookup("//"+getCodeBase().getHost()+"/adder");
            int a=Integer.parseInt(getParameter("a"));
            int b=Integer.parseInt(getParameter("b"));
            result=Integer.toString(adder.add(a,b));
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
    public void paint(Graphics g) {
        g.drawString(result,20,20);
    }
}
```



## Nebenläufigkeit

# Threads und RMI Aufrufe

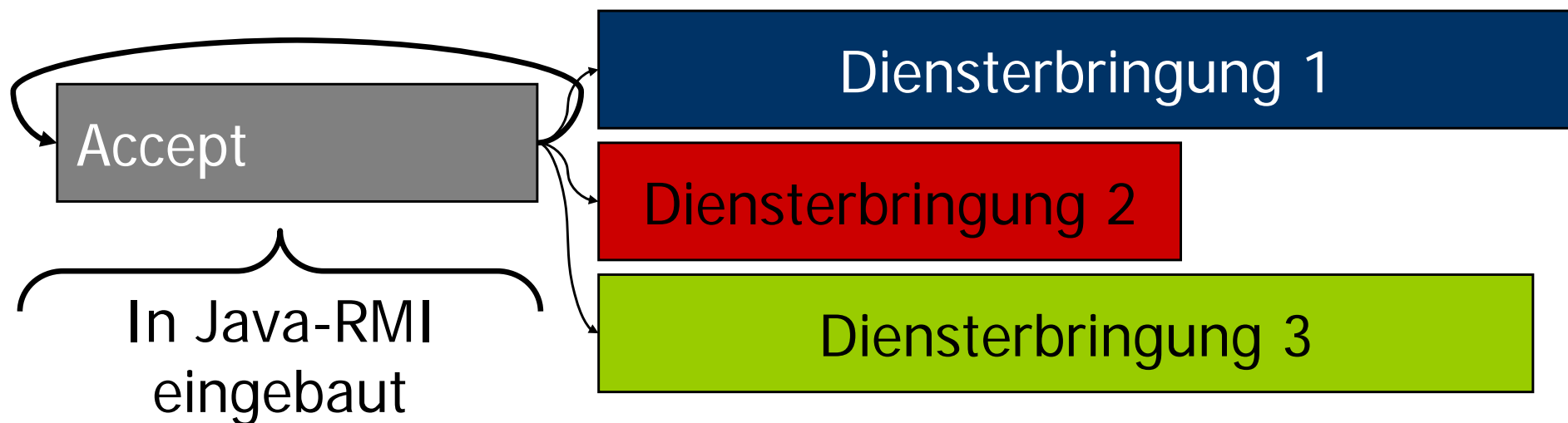
- Mehrere „gleichzeitig“ eintreffende RMI Aufrufe (lokal und mehrere Netzverbindungen) *können* in Threads arbeiten
- RMI Spezifikation:  
„**3.2 Thread Usage in Remote Method Invocations**  
A method dispatched by the RMI runtime to a remote object implementation **may or may not execute in a separate 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

# Multithreaded Server

- Das single-threaded Serverprogramm ist während der Erbringung eines Dienstes nicht in der Lage, neue Verbindungen anzunehmen:



- Abhilfe: Multithreading:



# 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

# Thread-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));
    }
}
```



## Multiserver

# Multiserver

- Ein Serverobjekt kann auch mehrere Dienste anbieten:  
*Multiserver*
- Es bietet mehrere Schnittstellen an
  - Mehrere unterschiedliche Dienste
    - Beispiel: Druckschnittstelle und Faxschnittstelle
  - Mehrere Aspekte desselben Dienstes
    - Beispiel: Funktionale Schnittstelle und Management Schnittstelle
    - RM-ODP-Part 1:  
**8.5.6 Management interfaces**  
In an ODP system, only an object can modify its own behaviour. An object may respond to requests from a management application to modify its behaviour and may, in consequence, delegate responsibility for some part of its management to the management application.

# Multiserver mit RMI

- RMI Objekte können mehrere Schnittstellen nach außen anbieten
- Management Beispiel:

```
public interface Managed extends java.rmi.Remote {  
    /* Return age of object in milliseconds */  
    public long age() throws java.rmi.RemoteException;  
    /* Terminate the server program */  
    public void shutdown() throws java.rmi.RemoteException;  
    /* Register the server */  
    public void register(String name) throws  
        java.net.MalformedURLException,  
        java.rmi.RemoteException;  
    /* Deregister the server */  
    public void deregister() throws java.rmi.RemoteException,  
        java.rmi.NotBoundException,  
        java.net.MalformedURLException;  
}
```

# ManagedCounterServer

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

public class ManagedCounterServer extends CounterServer implements Managed {
    Date startDate;
    String myName;

    public ManagedCounterServer() throws java.rmi.RemoteException {
        startDate=new Date();
    }

    /* Return age of object in milliseconds */
    public long age() throws java.rmi.RemoteException {
        return new Date().getTime()-startDate.getTime();
    }

    /* Terminate the server program */
    public void shutdown() throws java.rmi.RemoteException {
        try {
            deregister();
        } catch (Exception e) { /* Too brutal */}
        /* Too brutal... */
        System.exit(0);
    }
}
```

# ManagedCounterServer

```
/* Register the server */
public void register(String name)
    throws java.net.MalformedURLException, java.rmi.RemoteException {
    try {
        Naming.bind(name,this);
    } catch (AlreadyBoundException e) {Naming.rebind(name,this); }
    myName=name;
}

/* Deregister the server */
public void deregister() throws java.rmi.RemoteException,
                               java.rmi.NotBoundException
    java.net.MalformedURLException {
    Naming.unbind(myName);
}

public static void main(String argv[]) {
    System.setSecurityManager(new RMI SecurityManager());
    try {
        ManagedCounterServer mc = new ManagedCounterServer();
        mc.register("rmi://localhost/MyManCounter");
    } catch (Exception e) {}
}
}
```

# Shutdown

- Shutdown Programm nutzt die Managed Schnittstelle und weiß nicht, dass es sich auch um ein CounterServer handelt:

```
import java.net.*;
import java.rmi.*;
public class Shutdown {
    public static void main(String argv[]) {
        String name=argv[0];

        System.setSecurityManager(new RMISecurityManager());
        try {
            Managed mo = (Managed)java.rmi.Naming.lookup(name);
            System.out.println("Shutting down " + name +
                " after " + mo.age() + " milliseconds life.");
            mo.shutdown();
        } catch (Exception e) {
        }
    }
}
```

- Im Shutdown Programm wird immer eine Exception geworfen:

```
java Shutdown rmi://localhost/MyManCounter
```

```
Shutting down rmi://localhost/MyManCounter after 24083  
milliseconds life.
```

```
java.rmi.UnmarshalException: Error unmarshaling return  
header; nested exception is:
```

```
java.net.SocketException: Connection reset
```

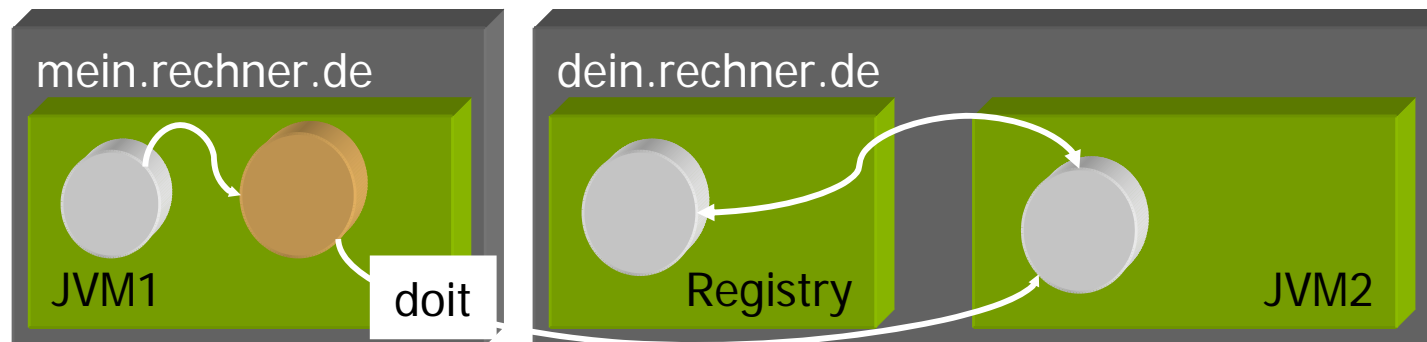
- Grund: shutdown() führt System.exit(0) aus bevor der Fernaufruf beendet ist...



## Objektaktivierung

# Persistente RMI Objekte

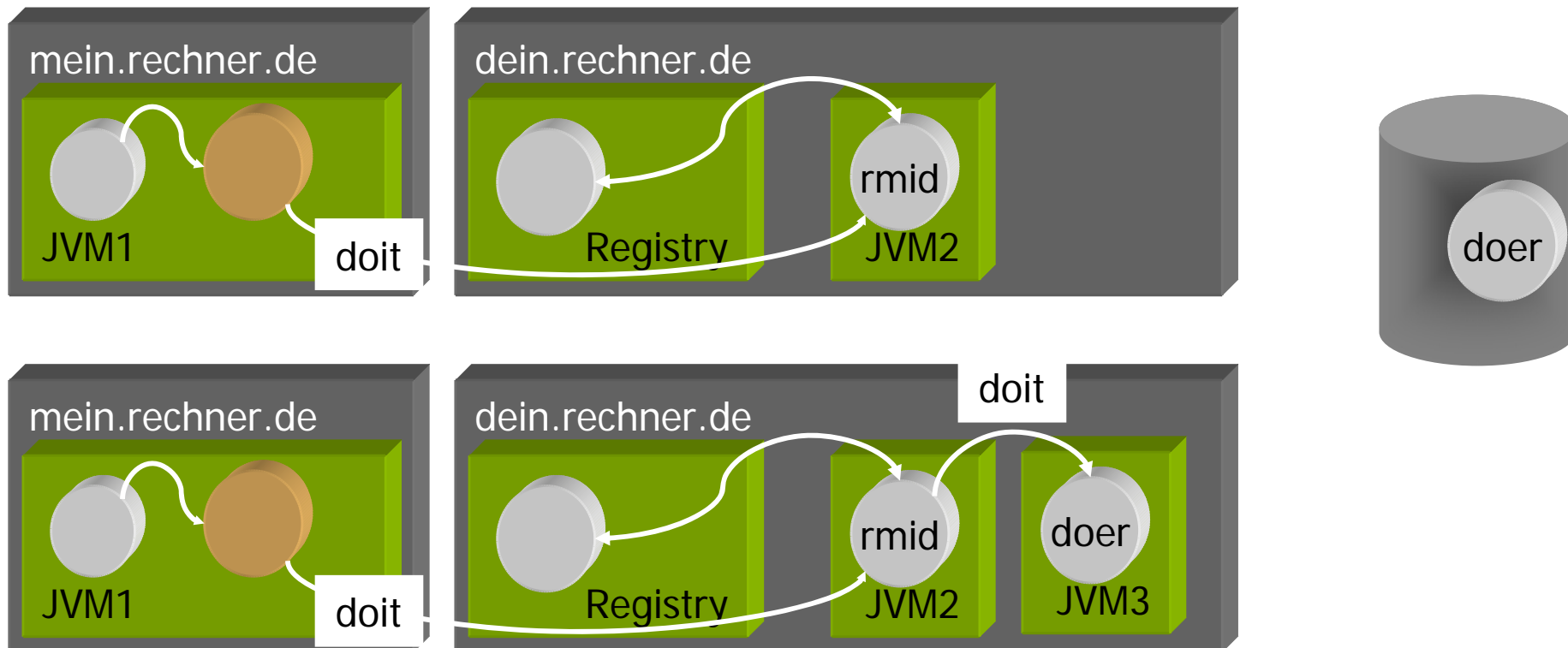
- Registrierte RMI Objekte müssen aktiv laufen um aufrufbar zu sein



- Persistenztransparenz (persistence transparency)
  - Objekte erscheinen immer zugänglich
    - Maskiert die Aktivierung und Deaktivierung von Objekten zu Klienten und zum Objekt selber
  - Durch Persistenz überlebt ein Objekt Zeiten in denen ein System nicht ausführen, speichern, kommunizieren etc. kann
  - Objekte erscheinen immer verfügbar

# RMI Aktivierung

- Mit RMI Aktivierung werden Objekte in einer eigenen JVM beim Aufruf gestartet
- Zuständig für Aktivierung: rmid Programm
- rmid registriert sich für das Objekt und kann es aktivieren



# Aktivierbares Serverobjekt

```
import java.rmi.*;
import java.rmi.activation.*;
public class ActivatableCounterServer extends Activatable
    implements Counter {
    int counter;
    public ActivatableCounterServer(ActivationID id,
                                    MarshalledObject data)
        throws RemoteException {
        // im Aktivierungssystem registrieren und exportieren
        super(id, 0);
    }

    public void add(Integer i) [...]
    public Integer value() [...]
}
```

# Aktivierung vorbereiten

```
import java.rmi.*;
import java.rmi.activation.*;
import java.util.*;
public class SetupCounter {
    public static void main(String[] argv) {
        try {
            System.setSecurityManager(new RMISecurityManager());
            // Ort der Policy-Datei vermerken
            Properties pr = new Properties();
            pr.put("java.security.policy",
                "M:/files/teaching/2005-06 V NP/Programme/RMI/counter/policy");
            // Umgebung für die zu startende JVM einrichten
            ActivationGroupDesc.CommandEnvironment env=null;
            ActivationGroupDesc myGroup=
                new ActivationGroupDesc(pr,env);
            // Eine ActivationGroup mit dieser Umgebung erzeugen
            ActivationGroupID aid=
                ActivationGroup.getSystem().registerGroup(myGroup);
```

# Aktivierung vorbereiten

```
String classLocation =
    "file:M:/files/teaching/2005-06%20V%20NP/Programme/RMI/counter/";
// keinerlei Parameter bei Aktivierung
MarshaledObject data=null;
ActivationDesc ad =
    new ActivationDesc(aid,"ActivatableCounterServer",
        classLocation,data);
// Interface registrieren
Counter c=(Counter)Activatable.register(ad);
// Binden
Naming.rebind("CounterServer",c);
System.exit(0);
} catch (Exception e) {
    System.out.println(e+"\n"+e.getMessage());
}
}
}
```

# Policy Datei

---

- Wieder erstmal alles erlauben:

```
grant {  
    // Allow everything for now  
    permission java.security.AllPermission;  
};
```

- Alles übersetzen

# Ausführen

- `rmiregistry`
- `rmid -J-Djava.security.policy="M:\files\teaching\2005-06 V Netzprogrammierung\Programme\RMI\counter\policy"`
- `java -Djava.security.policy="M:\files\teaching\2005-06 V Netzprogrammierung\Programme\RMI\counter\policy"`  
`SetupCounter`  
Terminiert nach Anmeldung des Objekts
- `java CounterUser`  
`10`

# rmid startet neue JVM für Server-Objekt

- Nach Anmeldung, von Client-Aufruf:

```
C:\Programme\util\system\cygwin\usr>ps -WI|grep -E "rmi|java"
```

```
2328 ? 0 14:34:11 C:\Programme\Java\jdk1.5.0_05\bin\rmiregistry.exe
```

```
3324 ? 0 14:34:20 C:\Programme\Java\jdk1.5.0_05\bin\rmid.exe
```

- Nach Client-Aufruf:

```
C:\Programme\util\system\cygwin\usr>ps -WI|grep -E "rmi|java"
```

```
2328 ? 0 14:34:11 C:\Programme\Java\jdk1.5.0_05\bin\rmiregistry.exe
```

```
3324 ? 0 14:34:20 C:\Programme\Java\jdk1.5.0_05\bin\rmid.exe
```

```
196 ? 0 14:36:39 C:\WINDOWS\system32\java.exe
```

```
2448 ? 0 14:36:41 C:\Programme\Java\jdk1.5.0_05\jre\bin\java.exe
```

- Nach Client-Aufruf:

```
C:\Programme\util\system\cygwin\usr>ps -WI|grep -E "rmi|java"
```

```
2328 ? 0 14:34:11 C:\Programme\Java\jdk1.5.0_05\bin\rmiregistry.exe
```

```
3324 ? 0 14:34:20 C:\Programme\Java\jdk1.5.0_05\bin\rmid.exe
```

```
2448 ? 0 14:36:41 C:\Programme\Java\jdk1.5.0_05\jre\bin\java.exe
```

# Deaktivierung

---

- Objekte teilen selber dem Aktivierungssystem mit, dass sie nicht mehr aktiv sein wollen:  
`Activatable.inactive(ActivationID id)`
- Wann soll das sein?
  - Heuristik notwendig
    - Wann sind keine Aufrufe zu erwarten?
    - Ist Aufwand Deaktivierung/Aktivierung < Aufwand des inaktiven Objekts?
- Wenn keine ankommenden Aufrufe vorliegen wird das Objekt nicht mehr nach aussen zugänglich gemacht (exportiert)
- Wenn in der Aktivierungsgruppe keine weiteren aktiven Objekte mehr sind, wird die gestartete JVM geschlossen

# Deaktivierung

```
public class ActivatableCounterServer [...]
    ActivationID id;
public ActivatableCounterServer(ActivationID id, [...])
    this.id=id; // id merken
[...]
// 1. Versuch
public Integer value() throws java.rmi.RemoteException {
    try {
        Activatable.inactive(id);
    } catch (Exception e) {
        System.err.println("Cannot deactivate");
    }
    return(new Integer(counter));
}
```

- inactive() liefert immer false, warum?
- Weil noch ein RPC aktiv ist!

- Richtig: Asynchrone Prozedur für inactive()-Aufruf:

```
public Integer value() throws java.rmi.RemoteException {
    new Thread (new Runnable() {
        public void run() {
            try {
                Thread.sleep(2000);
                Activatable.inactive(id);
            } catch (Exception e) {
                System.err.println("Cannot deactivate");
            }
        }
    }).start();
    return(new Integer(counter))
}
```



# Zustand

- Nach Deaktivierung ist Zustand verloren (JVM ist ja weg...)
- Bei Aktivierung kann ein Objekt übergeben werden, das beispielsweise auf eine Datei verweist

```
public class ActivatableCounterServer [...]
    File store;
    public ActivatableCounterServer(ActivationID id,
       _marshaledObject data)
        throws RemoteException, ClassNotFoundException,
        IOException {
        // im Aktivierungssystem registrieren und exportieren
        super(id, 0);
        this.id=id;
        store = (File)data.get();
        if (store.exists()) {
            restoreState();
        } else { counter=0; } // eigentlich unnötig...
    }
```

- Serverobjekt ist selber für Zustandssicherung und –wiederherstellung verantwortlich:

```
public void add(Integer i) throws java.rmi.RemoteException {  
    counter += i.intValue();  
    saveState();  
}
```

```
void restoreState() throws IOException, ClassNotFoundException {  
    DataInputStream dis=  
        new DataInputStream(new FileInputStream(store));  
    counter=dis.readInt();  
    dis.close();    }  
}
```

```
void saveState() {  
    try {  
        DataOutputStream dos=  
            new DataOutputStream(new FileOutputStream(store));  
        dos.writeInt(counter);  
        dos.close();  
    } catch (Exception e) {  
        throw new RuntimeException("State not saved");  
    }  
}
```

- Alternative: ObjectOutputStream-Klassen verwenden

- Bei Vorbereitung der Aktivierung Datei vorbereiten

```
public class SetupCounter {
    public static void main(String[] argv) {
        // keinerlei Parameter bei Aktivierung
        //      MarshalledObject data=null;
        MarshalledObject data =
            new MarshalledObject(
                new File("M:/files/teaching/2005-06 V
                        NP/Programme/RMI/counter/store"));
        ActivationDesc ad = new ActivationDesc(
            aid,
            "ActivatableCounterServer",
            classLocation,
            data);
    }
}
```

# Persistenztransparenz

---

- `java.rmi.activation` realisiert Persistenztransparenz:
- Persistenztransparenz (persistence transparency)
  - Objekte erscheinen immer zugänglich
    - Maskiert die Aktivierung und Deaktivierung von Objekten zu Klienten und zum Objekt selber
  - Durch Persistenz überlebt ein Objekt Zeiten in denen ein System nicht ausführen, speichern, kommunizieren etc. kann
  - Objekte erscheinen immer verfügbar



## Zusammenfassung

1. Callbacks
  1. Klienten werden von Server zurückgerufen
2. Nebenläufigkeit
  1. Beliebig viele Threads in RMI-Objekten
3. Multiserver
  1. Mehrere Schnittstellen können angeboten werden
4. Objektaktivierung
  1. Objekte können auch erst bei Aufruf aktiviert werden
  2. Deaktivierung
  3. Zustandssicherung

# 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/>
- Sun. Remote Object Activation  
<http://java.sun.com/j2se/1.4.2/docs/guide/rmi/activation.html>  
Default Policy Implementation and Policy File Syntax  
<http://java.sun.com/j2se/1.4.2/docs/guide/security/PolicyFiles.html>