F Überblick über die 5. Übung

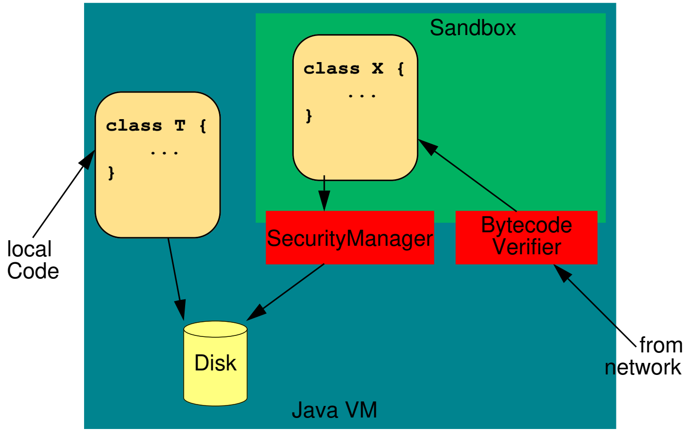
- SecurityManager
- RMI

F.1 Java Sicherheit

- Sandboxing
- Bytecode Verifier
- Security Manager
- Implementierung der Java-Bibliotheken

1 Sandboxing

- Klassen, die vom Klassenlader geladen wurden, werden innerhalb einer *Sandbox* ausgeführt



2 SecurityManager

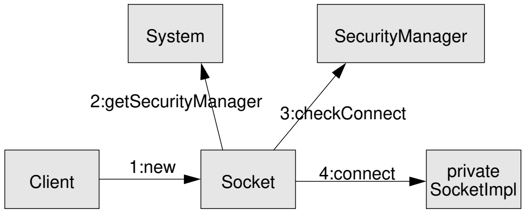
- Prüft, ob eine Klasse eine Operation durchführen darf.
- `System.getSecurityManager()` liefert den globalen SecurityManager zurück
- Überprüfungen müssen von der geschützten Klasse selbst durchgeführt werden

```
SecurityManager security = System.getSecurityManager();
if (security != null) {
    security.checkConnect(address.getHostAddress(), port);
}
```

- `System.setSecurityManager()` installiert einen globalen SecurityManager
  - diese Methode kann (normalerweise) nur einmal aufgerufen werden!

3 SecurityManager und Sockets

- Beispiel Netzwerkkommunikation: Erzeugung eines neuen Sockets:



4 Beispiel SecurityManager

```
public class SimpleSecurityManager extends SecurityManager
{
    public void checkRead(String s) {
        if (...) {
            throw new SecurityException("checkread");
        }
    }
}
```


5 Was wird durch einen SecurityManager geschützt?

- Zugriffe auf das lokale Dateisystem
- Zugriffe auf das Betriebssystem
  - Ausführen von Programmen
  - Lesen von System Informationen
- Zugriffe auf das Netzwerk
- Thread Manipulationen
- Erzeugung von *factory* Objekten (Socket Implementierung)
- JVM: Linken von native code, Verlassen des Interpreters, Erzeugung eines Klassenladers
- Erzeugung von Fenstern
- ...

6 AppletSecurityManager

Security Checks	AppletSecurityManager
checkCreateClassLoader	nicht erlaubt
checkConnect	nur erlaubt, wenn die URL des Klassenladers der Klasse den Zielrechner enthält
checkExit	nicht erlaubt
checkExec	nicht erlaubt

F.2 Fernmethodenaufrufe
Remote Method Invocation (RMI)

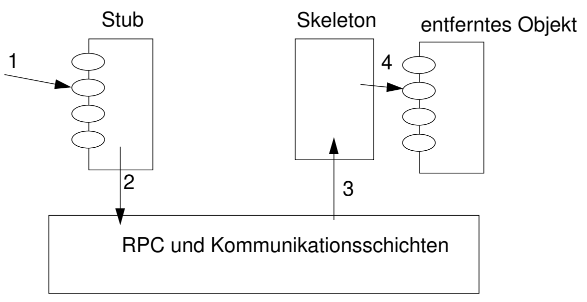
- ermöglicht Abstraktion in einem verteilten System

Socket-Kommunikation	keine Abstraktion
↓ Fernaufrufe (RPC)	prozedurale Abstraktion
↓ Fernmethodenaufrufe (RMI)	Abstraktion auf Objektebene

- Fernmethodenaufrufe: Aufruf von Methoden an Objekten auf anderen Rechnern
- Transparente Objektreferenzen zu entfernten Objekten

1 Überblick

- Stub: Stellvertreter (Proxy) des entfernten Objekts.
- Skeleton: Ruft die Methoden am entfernten Objekt auf.



- Anfrage: Objekt ID, Methoden ID, Parameter

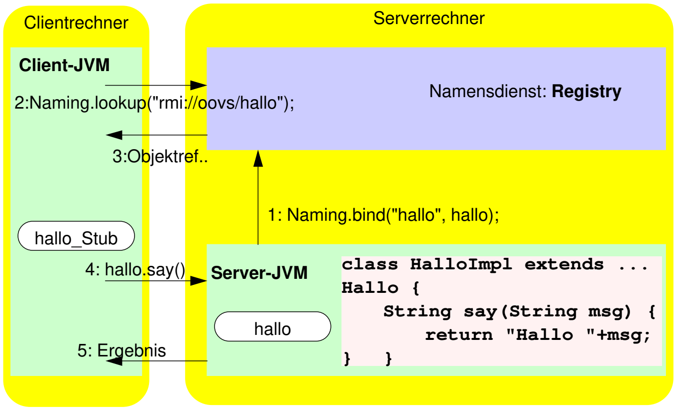
2 Einführung

- *Remote Object* (entferntes Objekt): Ein Objekt welches aus einer anderen JVM herraus genutzt werden kann
- *Remote Interface*: Beschreibt die Methoden eines entfernten Objekts.
- Ein *remote Interface* muss von **`java.rmi.Remote`** abgeleitet sein.
- Zugriffe auf ein entferntes Objekt sind nur über *remote Interfaces* möglich.
- Die Klasse eines entfernten Objekts muss mindestens ein *remote Interface* implementieren.
- Entfernte Objekte können selbst wieder entfernte Objekte nutzen.
- Parameter werden wie folgt übergeben:
  - ◆ by value: Bei allen Parametern die keine entfernten Objekte sind
  - ◆ by reference: Bei Parametern welche **`java.rmi.Remote`** implementieren

3 lokaler vs. fern Methodenaufruf

- jeder Aufruf kann eine **`RemoteException`** auslösen.
  - ◆ dabei weiß der Aufrufer nicht ob die Methode komplett, teilweise, oder garnicht ausgeführt wurde
- Entfernte Objekte können nur über Intefaces angesprochen werden.
- Normale Objekte werden kopiert anstatt eine Referenz darauf zu übergeben.

4 RMI Operation



5 Registry (1)

- Die *Registry* übernimmt als Namesdienst die Abbildung von Objektnamen auf Objektreferenzen.
- Der Zugriff erfolgt über die Klasse **`java.rmi.Naming`**:
  - **`void Naming.bind(String name, Remote obj)`**
    registriert ein Objekt unter einem eindeutigen Namen,
    falls das Objekt bereits registriert ist wird eine Exception ausgelöst
  - **`void Naming.rebind(String name, Remote obj)`**
    registriert ein Objekt unter einem eindeutigen Namen,
    falls das Objekt bereits registriert ist wird der alte Eintrag gelöscht
  - **`Remote Naming.lookup(String name)`**
    liefert die Objektreferenz zu einem gegebenen Namen
  - **`void Naming.unbind(String name)`**
    löscht den Namenseintrag
- Die Registrierung ist nur bei der Registry auf dem aktuellen Rechner möglich!

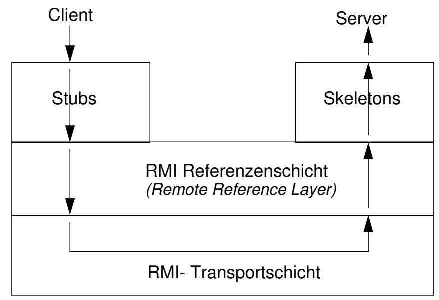
5 Registry (2)

- Einen bestimmten Port verwenden:
  - Die Registry nimmt TCP/IP-Verbindungen an Port 1099 entgegen.
  - als Parameter kann jedoch ein anderer Port angegeben werden
    z.B. 10412:
    `rmiregistry 10412`
  - Wenn eine Registry an einem bestimmten Port verwendet werden soll, so muss die URL, die bei **`bind/rebind/unbind/lookup`** verwendet wird diesen Port enthalten:

```
Naming.rebind("rmi://faui40:10412/hallo", hallo)

Naming.lookup("rmi://faui40:10412/hallo")
```


6 Systemarchitektur

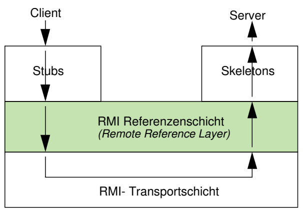


7 Stubs und Skeletons

- Stub (auf Clientseite) - implementiert das *remote interface*
  1. erhält einen **`ObjectOutputStream`** von der RMI - Referenzenschicht
  2. schreibt die Parameter in diesen Strom
  3. teilt der RMI - Referezenschicht mit, die Methode aufzurufen
  4. holt einen **`ObjectInputStream`** von der RMI - Referezenschicht
  5. liest das Rückgabeobjekt aus diesem Strom
  6. liefert das Rückgabeobjekt an den Aufrufer
- Skeleton (auf Serverseite)
  1. erhält einen **`ObjectInputStream`** von der RMI - Referezenschicht
  2. liest die Parameter aus diesem Strom
  3. ruft die Methode am implementierten Objekt auf.
  4. holt einen **`ObjectOutputStream`** von der RMI - Referezenschich
  5. schreibt das Rückgabeobjekt in diesem Strom.

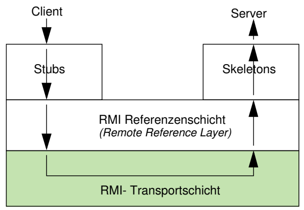
8 Referenzenschicht (Remote Reference Layer)

- verantwortlich für das Aufräumen (Garbage Collection) der entfernten Objekte
- implementiert die Aufrufsemantik, z.B.:
  - unicast, Punkt-zu-Punkt
  - Aufruf an einem replizierten Objekt
  - Strategien zum Wiederaufbau der Verbindung nach einer Unterbrechung

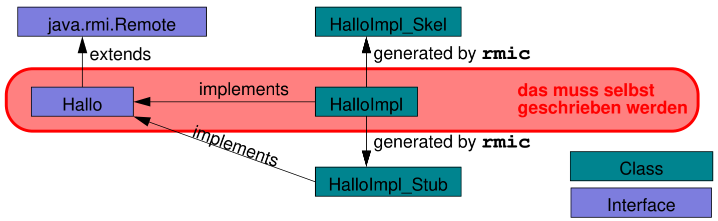


9 Transportschicht

- verantwortlich für die Datenübertragung zwischen den Rechnern
- aktuelle Implementierung basiert auf TCP/IP Sockets
- verschiedene Transportmechanismen sind denkbar



10 Interface und Implementierung



- Der Programmierer schreibt das *remote Interface* **`Hallo`** und die Implementierung **`HalloImpl`**.
- Der Stellvertreter (Stub) **`HalloImpl_Stub`** und das Skeleton **`HalloImpl_Skel`** werden duch **`rmic`** aus der Implementierung **`HalloImpl`** generiert.
- **`RMIClassLoader`**: lädt die Klassen der Parameter und des Rückgabeobjekts.

11 ein einfaches Beispiel

1. Remote Interface
2. Server
3. Serverinitialisierung
4. Client
5. Starten des Systems

11 Beispiel - Remote-Interface

- Remote Interface *Bank*:

```
public interface Bank extends java.rmi.Remote {
    void deposit(Money amount, Account account)
                                throws java.rmi.RemoteException;
}
```

- Remote Interface *Account*:

```
public interface Account extends java.rmi.Remote {
        void deposit(Money amount)
                                throws java.rmi.RemoteException;
}
```

- Parameter *Money*:

```
public class Money implements java.io.Serializable {
    private float value;
    public Money(float value) { this.value = value; }
}
```


11 Beispiel - Remote-Interface

- jedes entfernte Objekt muss ein Interface implementieren, das alle Methoden enthält, die das Objekt seinen Clients anbieten soll
- das Interface muss von `java.rmi.Remote` erben
- alle Methoden können `java.rmi.RemoteException` werfen
- alle Parameter und Rückgabewerte müssen
  - Serializable sein (d.h. sie müssen das Interface `java.io.Serializable` implementieren)
  - oder sie müssen entfernte Objekte sein.

11 Beispiel - Server

- Der Server implementiert das Interface.
- Methoden brauchen keine `RemoteException` zu werfen.
- Zwei Alternativen zur Erzeugung eines Remote-Objektes
  - Der Server wird von `UnicastRemoteObject` abgeleitet

```
import java.rmi.server.*;
public class BankImpl extends UnicastRemoteObject implements Bank
{
        public BankImpl () throws java.rmi.RemoteException {...}

        public void deposit(Money amount, Account account)
                                throws java.rmi.RemoteException {
                account.deposit(amount);
        }
}
```

  - oder es wird mit `exportObject` ein Remote-Objekt erzeugt:

```
public class BankImpl implements Bank {... }
Remote bank = UnicastRemoteObject.exportObject(new BankImpl())
```

11 Beispiel - Serverinitialisierung

- Das Remote-Objekt muss mit **bind** oder **rebind** in die Registry eingetragen werden.

```
Naming.bind("rmi://faui40b:10412/bank", bank);
```

Protokoll Rechnername Objektname

- RMISecurityManager:
  - ohne SecurityManager lädt der RMI Classloader keine Klassen
  - Server-JVM muss einen SecurityManager setzen, z.B.

```
System.setSecurityManager(new RMISecurityManager());
```

  - **RMISecurityManager** ist ähnlich **AppletSecurityManager**

11 Beispiel - Client

- Der Client kann sich mit **lookup** eine Referenz auf das entfernte Objekt (den Server) besorgen.
- Beispiel:

```
public class Client {
    public static void main (String argv[]) throws
                            java.net.MalformedURLException,
                            java.rmi.NotBoundException,
                            java.rmi.RemoteException  {
        Bank bank= (Bank)java.rmi.Naming.lookup
                            ("rmi://faui40u:10412/bank");
        Account account = new AccountImpl();
        bank.deposit(new Money(10), account);
    }
}
```


11 Beispiel - System starten

- Klassen-Pfad setzen, damit **rmic** und java (server) die Klassen finden

```
setenv CLASSPATH /proj/i4mw/felser/aufgabe4
```

- Stubs und Skeletons erzeugen

```
rmic -d ${CLASSPATH} BankImpl
```

- Auf dem Serverrechner die Registry starten

```
rmiregistry 10412&
```

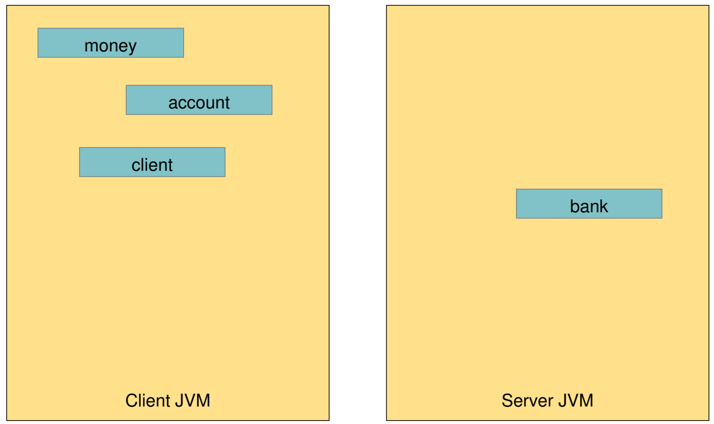
- Auf dem Serverrechner das Serverobjekt starten

```
java -Djava.rmi.server.codebase=
                file:///proj/i4mw/felser/aufgabe4/ BankImpl
```

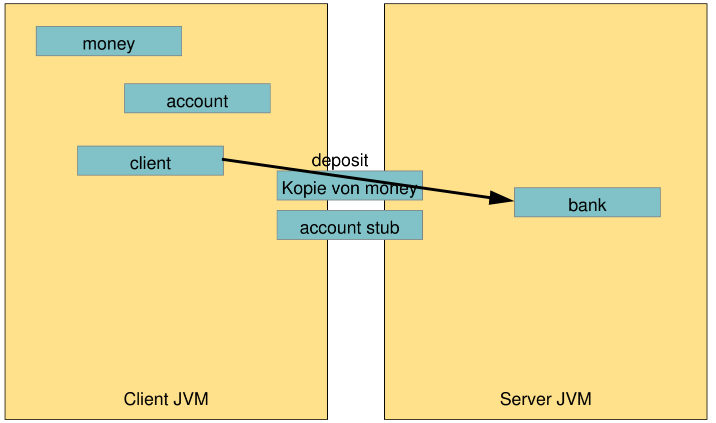
- Von jedem beliebigen Rechner aus kann der Client nun benutzt werden:

```
java Client
```

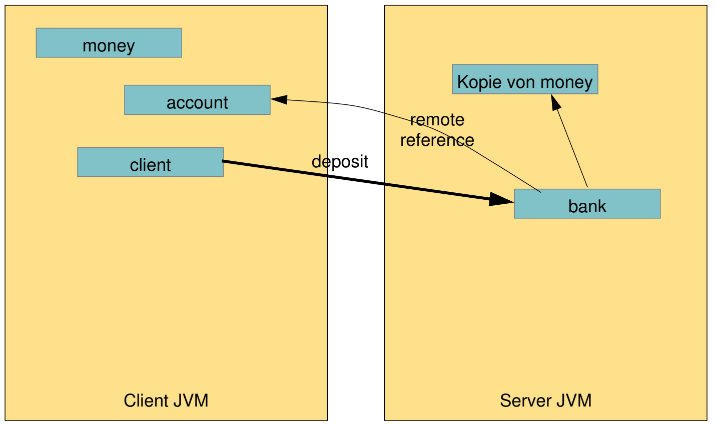

11 Interaktionen während des Fernmethodenaufrufes



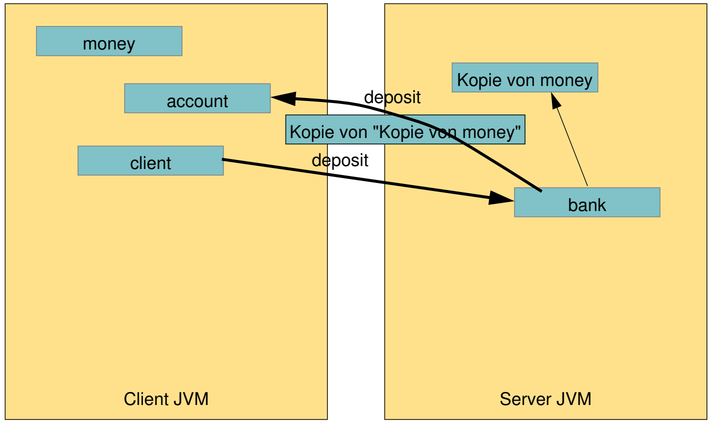
11 Interaktionen während des Fernmethodenaufrufes (2)



11 Interaktionen während des Fernmethodenaufrufes (3)

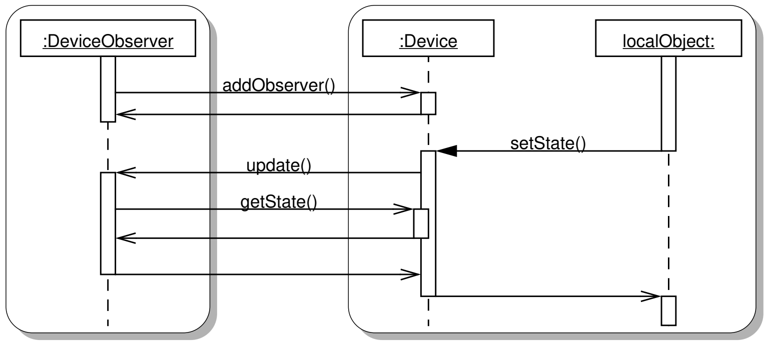


11 Interaktionen während des Fernmethodenaufrufes (4)



12 Beispiel Observer

- Überwachung eines Gerätes
- Benachrichtigung bei Zustandsänderung



12 Beispiel Observer: Remote-Interfaces

- Interface für Gerät (entspricht Observable)

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

        public Integer getState()
                    throws java.rmi.RemoteException;

        public void addObserver( DeviceObserver o )
                    throws java.rmi.RemoteException;
}
```

- Interface für Geräteüberwachung (entspricht Observer)

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

        public void update( Device d )
                    throws java.rmi.RemoteException;
}
```


12 Beispiel: Observable-Implementierung

```
public class DeviceImpl
        extends java.rmi.UnicastRemoteObject implements Device {
        private Integer state; // Zustand
        private java.util.Vector observers; // Menge der Observer
        public DeviceImpl() throws java.rmi.RemoteException {
           super();
           ... /* Initialisierung von state und observers */
        }
        // Remote-Methoden
        public Integer getState() throws java.rmi.RemoteException {
           return state;
        }
        public void synchronized addObserver( DeviceObserver o ){
           observers.addElement( o );
        }
        // lokale Methode
        public void synchronized setState( Integer s ) {
           state = s;
           ... // fuer jeden DeviceObserver o in observers
           try {
              // Remote-Aufruf an DeviceObserver
              o.update( this );
           } catch( Exception e ) { /* Fehler */ };
        }
}
```


12 Beispiel Observer: Server-Wrapper

```
public class DeviceServer {
        public static void main( String argv[] ) {
           // Security-Manager setzen
           System.setSecurityManager
               (new java.rmi.RMISecurityManager());
           try {
              // Implementierungsobjekt erzeugen
              DeviceImpl d = new DeviceImpl();
              // Objekt mit Namen anmelden
              java.rmi.Naming.rebind( "rmi://server/Device1", d );
           } catch ( Exception e ) {
              // Fehler
           }
        }
}
```


12 Beispiel Observer: Observer-Application

```
public class DeviceAppl extends java.awt.Frame
           implements DeviceObserver{
        private Integer state; // Zustand fuer paint()
        public static void main(String [] args){ new DeviceAppl(); }
        public DeviceAppl(){ super(...);setSize(..);setVisible(..);
           try { // Skeleton erzeugen
              java.rmi.server.
                     UnicastRemoteObject.exportObject( this );
              // Device suchen
              Device d = (Device)java.rmi.Naming.lookup
                     ("rmi://server/Device1" );
              state = d.getState(); // aktuellen Zustand holen
              d.addObserver( this ); // als Observer registrieren
           } catch ( Exception e ) { /* Fehler */ }
        }
        public void paint( java.awt.Graphics g ) {
           g.drawString( "Current state: " + state, ... );
        }
        // Remote-Methode
        public void update( Device d )
                 throws java.rmi.RemoteException {
           state = d.getState(); // veraenderten Zustand holen
           repaint(); // ruft paint() auf
        }
}
```

13 Zusammenfassung

- Ein entferntes Objekt muss ein “*remote Interface*” implementieren.
- Alle Methoden des Interfaces müssen eine `java.rmi.RemoteException` werfen.
- Um ein entferntes Objekt zu erzeugen kann man entweder `java.rmi.server.UnicastRemoteObject` ableiten oder die Methode `exportObject` verwenden.
- Um entfernte Objekte an der *Registry* zu registrieren oder zu suchen kann man `java.rmi.Naming` verwenden.
- Clients verwenden nur das remote interface.