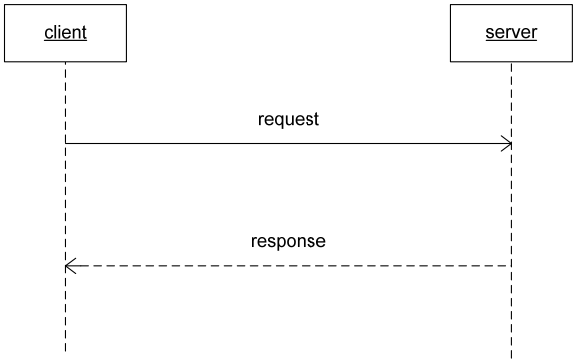
Remote-Objekte

Remote-Objekte

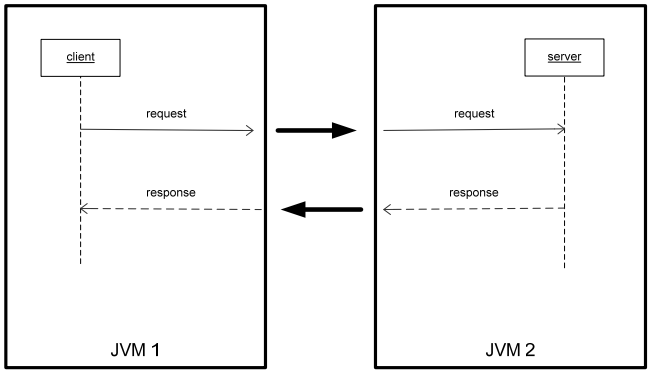
- Motivation
- Grundarchitektur
- Implementierung von Remote-Objekten
- Parameterübergabe
- Callbacks
- Distributed Garbage Collection
- Verteilung und Nachladen von Code

Motivation

- Methodenaufruf bei einer VM
  - Objekte innerhalb einer JVM



- Methodenaufruf über VM Grenzen hinweg
  - Objekte sind auf mehreren JVMs (Rechnern) verteilt



Probleme bei Remote-Objekten

- Objektreferenzen:
  - Was ist eine Remote-Referenz?
  - Wie finde ich ein Objekt?
  - Wie spreche ich Objekte an?
- Methodenaufruf:
  - Wie wird die Sprungadresse für die Methode ermittelt?
- Parameterübergabe und Rückgabewerte:
  - Wie werden Parameterwerte und Rückgabewerte verschickt?
- Objekterzeugung:
  - Wie kann der Client ein Remote-Objekt erzeugen?
- Garbage Collection:
  - Wann kann ein Objekt am Server frei gegeben werden?
- Laden der Klassen:
  - Von wo wird die Klasse für ein vom Client benötigtes Objekt geladen?

Remote-Objekte

Motivation

Grundarchitektur

Implementierung von Remote-Objekten

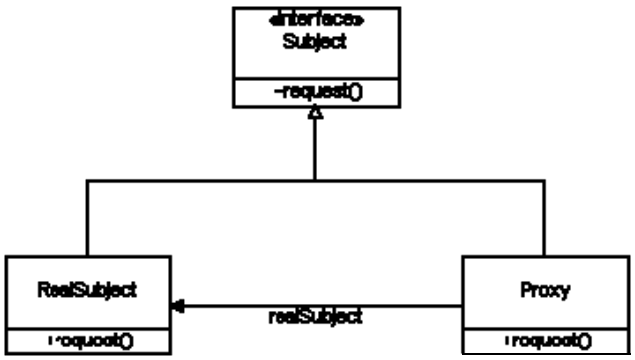
Parameterübergabe

Callbacks

Distributed Garbage Collection

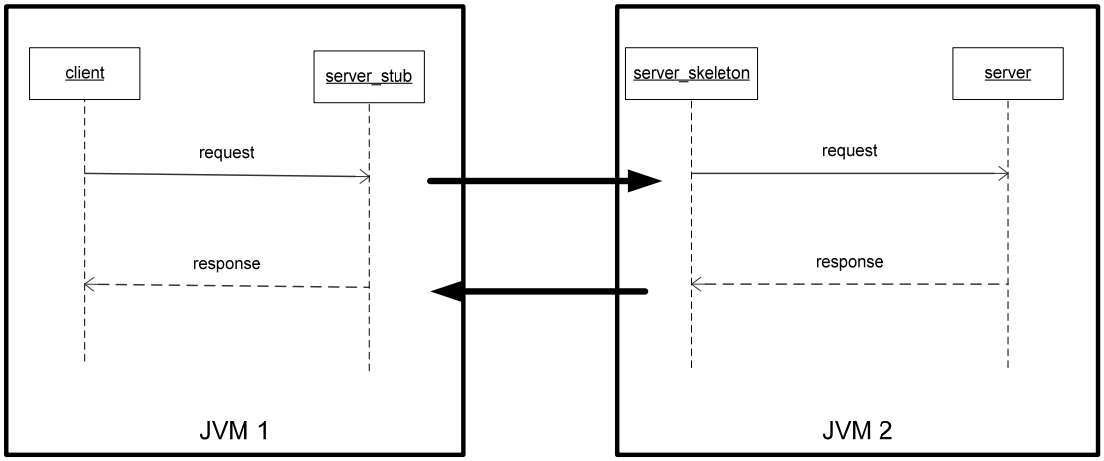
Verteilung und Nachladen von Code

Proxy-Pattern



- Interface:
  - definiert eine allgemeine Schnittstelle, die Client sieht und Server implementieren muss
- Implementierung für den Server
  - hat Interface entsprechend zu implementieren
- Proxy für den Client
  - wird vom Client angesprochen
  - implementiert die Schnittstelle, d.h. stellt sich dar wie ein wirkliches Objekt
  - delegiert alle Requests an die wirkliche Objektimplementierung

Proxy-Pattern bei Remote-Objekten



- Stub stellt das Proxy-Objekt auf der Seite des Client dar
- Proxy leitet die Anforderungen über die VM-Grenzen an Server weiter
- Skeleton empfängt die Requests und baut eigentlichen Request an den Server auf
- Server bedient den Request
- Ergebnis wird an Skeleton zurückgegeben
- Ergebnis wird über Netzwerk an Stub weitergeleitet
- Stub gibt das Ergebnis an Client zurück

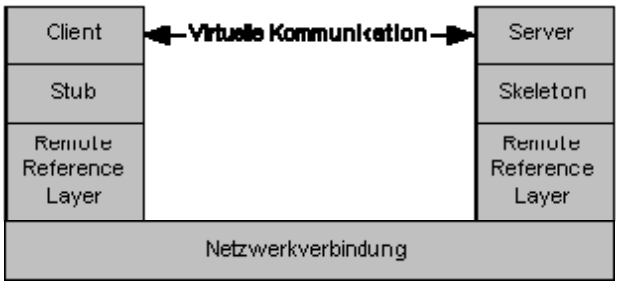
Kommunikationsarchitektur

Client und Server

Stubs und Skeletons

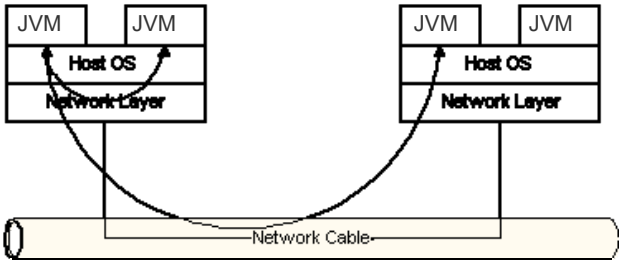
Remote Reference Layer

Netzwerkverbindung



Netzwerkverbindung

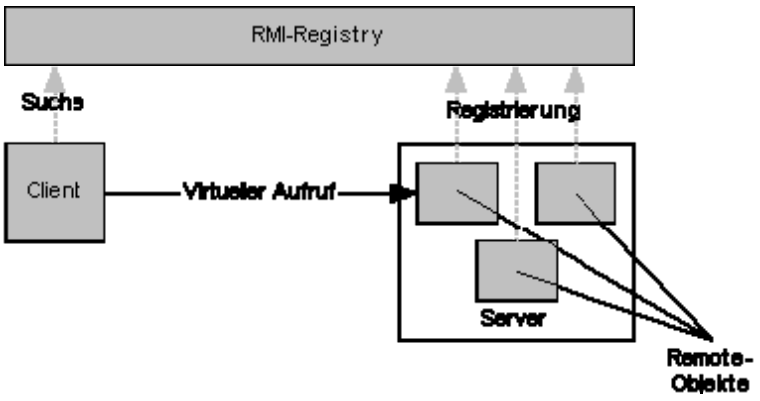
- zwischen JVMs immer über Network Layer des Host OS



Kommunikationsarchitektur: Aufgaben der Schichten

- Client und Server
  - sind die eigentlichen Anwendungsobjekte
  - kommunizieren über virtuelle Kommunikation
- Stubs und Skeletons
  - Stub (Achtung: seit Java 1.5 automatisch generiert)
    - ist das Proxy-Objekt auf der Client-Seite und stellt sich wie Server-Objekt dar
    - leitet Requests des Client an die unteren RMI-Services weiter
  - Skeleton (Achtung: seit Java 1.2 nicht mehr nötig!!!)
    - empfängt die Requests von unteren RMI-Services und
    - setzt diese für Requests an Server-Objekt um
- Remote Reference Layer
  - verwaltet die Remote-Referenzen (Klasse `RemoteRef`) der Server-Objekte
  - `RemoteRef` bietet Methode `invoke` für Methodenaufrufe abzusetzen (wird von Stubs verwendet)
  - dient zur Registrierung von Remote-Objekts
- Netzwerkverbindung
  - eigentliche Netzwerkverbindung über TCP/IP
  - stream-basierte Verbindung
  - standardmäßig auf Port 1099
  - Kommunikationsprotokoll oberhalb TCP/IP (nicht standardisiert)
    - Java Remote Method Protocol (JRMP)
    - IIOP
    - und weitere

Objektregistrierung und Objektsuche



- RMI ermöglicht
  - Registrierung von Remote-Objekten durch Server mit einer RMI-URL
  - Aufsuchen von Remote-Objekten durch Client
- Dazu dienen
  - RMI-Registrierungsprogramm **rmiregistry**
  - RMI-Service Klasse **`Naming`**

Objektregistrierung und Objektsuche: Vorgehen

- Über Service-Klasse `Naming` wird mit `bind` oder `rebind` ein Server-Objekt registriert. Es muss dabei ein eindeutiger Name für das Server-Objekt vergeben werden.

```
try {
  BankManager bm = new BankManagerImpl();
  Naming.rebind("BankService", bm);
}
  ...
```

Das Server-Objekt wird dabei unter der RMI-URL mit
`rmi://<HostComputer>:1099/BankService`
eingetragen.

- Damit hat ein Client eine eindeutige URL, um auf ein Server-Objekt zuzugreifen; dazu dient abermals die Service-Klasse `Naming`.

```
// Client
try {
  BankManager bm = Naming.lookup("rmi://bankserver.com/BankService");
  ...
}
...
```

default port 1099

Remote-Objekte

Motivation

Grundarchitektur

Implementierung von Remote-Objekten

Parameterübergabe

Callbacks

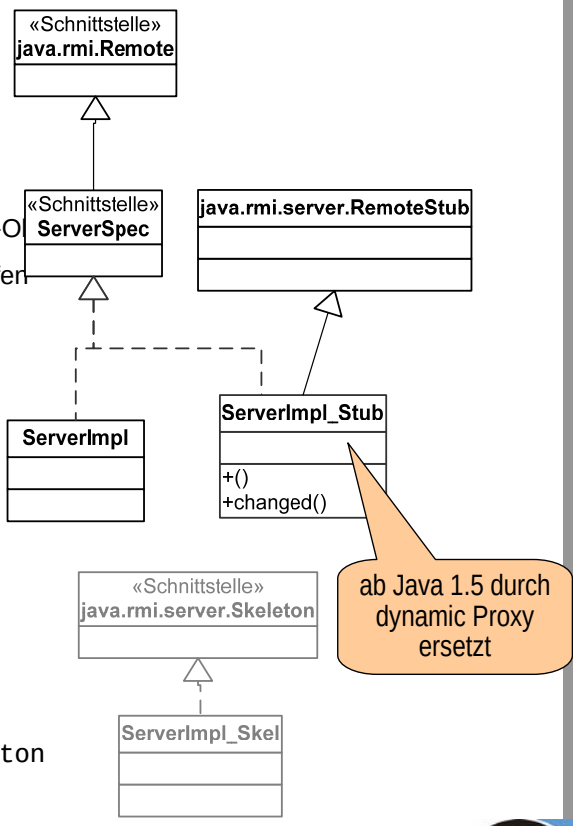
Distributed Garbage Collection

Verteilung und Nachladen von Code

Realisierung eines Remote-Objekts

Interfaces und Klassen

- Interface Spezifikation für Remote-Objekt
  - muss **`java.rmi.Remote`** erweitern
  - definiert die die sichtbaren Methoden des Remote-O
  - jede Methode muss eine `RemoteException` werfen
- Server-Objekt
  - muss Interface implementieren
  - muss an das RMI System exportiert werden
- Stub-Objekt
  (ab Java 1.5 durch dynamic Proxy ersetzt)
  - wird mit **`rmic`** erzeugt
  - implementiert Interface
- Skeleton-Objekt (nur bei Java 1.1 verwendet)
  - wird mit **`rmic`** erzeugt
  - erweitert Basisklasse `java.rmi.server.Skeleton`



Beispiel Calculator

Im folgenden Beispiel wird anhand eines einfachen Calculator-Servers die Realisierung eines Client-Server-Programms veranschaulicht

Folgende Schritte sind dabei durchzuführen :

- Definition eines Interfaces für das Remote-Objekt
- Implementierung des Interfaces für den Server
- Generierung von Stub- und Skeleton-Klassen mit `rmic`
- Implementierung des Server-Programms, welches das Server-Objekt anlegt und bei der RMIRegistry registriert
- Realisierung eines Client-Programms, welches auf das Server-Objekt zugreift und dieses verwendet
- Starten des `rmiregistry`
- Starten des Server-Programms
- Starten des Client-Programms

Beispiel Calculator: Interface *Calculator*

- Im folgenden Beispiel wird eines Calculator-Servers über RMI realisiert:
- Interface `Caluclator` definiert eine Reihe von Methoden für die Grundrechnungsarten

```
public interface Calculator extends java.rmi.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;

}
```

Beispiel Calculator: Implementierung `CalculatorImpl`

- Klasse `CalculatorImpl` stellt eine entsprechende Implementierung von `Caluclator bereit`

```
public class CalculatorImpl implements Calculator {

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

    public long sub(long a, long b) throws java.rmi.RemoteException {
        return a - b;
    }

    public long mul(long a, long b) throws java.rmi.RemoteException {
        return a * b;
    }

    public long div(long a, long b) throws java.rmi.RemoteException {
        return a / b;
    }
}
```

Beispiel Calculator: Generierung von Stub und Skeleton

- Stub- und Skeleton-Klassen müssen nicht geschrieben werden
- Sie können durch das Programm

  `rmic`

  automatisch generiert werden
- **`rmic`** wird mit Server-Implementierungsklasse aufgerufen

```
Microsoft Windows XP

> rmic –keep CalculatorImpl
```

```
rmic <options> <class names>
where <options> includes:
-keep Do not delete intermediate generated source files
-g Generate debugging info
-depend Recompile out-of-date files recursively
-nowarn Generate no warnings
-verbose Output messages about what the compiler is doing
-classpath <path> Specify where to find input source and
        class files
-d <directory> Specify where to place generated class
        files
-J<runtime flag> Pass argument to the java interpreter
-v1.1 Create stubs/skeletons for JDK 1.1 stub protocol
        version
-vcompat (default) Create stubs/skeletons compatible with
        both JDK 1.1 and Java 2 stub protocol versions
-v1.2 Create stubs for Java 2 stub protocol version only
```

```
public final class CalculatorImpl_Stub
    extends java.rmi.server.RemoteStub
    implements Calculator, java.rmi.Remote
{
     ...

}
```

```
public final class CalculatorImpl_Skel
    implements java.rmi.server.Skeleton
{
     ...


}
```

Beispiel Calculator: Server-Programm

- Server-Programm hat folgende Aufgaben:
  - erzeugen des Server-Objekts
  - exportieren des Server-Objekts an das RMI
  - Registrieren des Server-Objekts unter einem RMI-URL mittels Klasse `Naming`

```
import java.rmi.Naming;

public class CalculatorServer {

   public CalculatorServer() {
          try {
              Calculator c = new CalculatorImpl();
              Naming.rebind("rmi://localhost:1099/CalculatorService",
                        UnicastRemoteObject.exportObject(c));
          } catch (Exception e) {
                    System.out.println("Trouble: " + e);
          }
     }

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

Exportieren von Server-Objekten

- Server-Objekte müssen bei ihrer Erzeugung an das RMI exportiert (bekannt gemacht) werden
- Dies kann auf zwei Arten passieren
  1. mit static-Methode `exportObject` von Klasse `UnicastRemoteObject` (wie im letzten Beispiel)
  2. durch Ableiten von Klasse `UnicastRemoteObject`,
     Implementierung eines Konstruktors der `RemoteException` wirft
     und Aufruf des Konstruktors dieser Basisklasse `UnicastRemoteObject`

```
public class CalculatorImpl
     extends java.rmi.server.UnicastRemoteObject  implements Calculator {

   // Implementations must have an explicit constructor in order to declare the
   //RemoteException exception
   public CalculatorImpl()
         throws java.rmi.RemoteException {
         super();
   }
   ...
```

Beispiel Calculator: Client-Programm

- Client-Programm wird:
  - sich über den Naming-Service mit dem URL das Remote-Objekt holen
  - das Remote-Objekt verwenden (dabei direkt den Stub aufrufen)
  - eine Reihe von Exceptions abfangen

```
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);
    }
  }
}
```

Beispiel Calculator: Starten der Programme

- Starten des RMIRegistry-Programms

```
Microsoft Windows XP

> rmiregistry
```

- Starten des Server-Programms

```
> java CalculatorServer
```

- Starten des Client

```
> java CalculatorClient
1
9
18
3
```

Remote-Objekte

Exkurs: Dynamic Proxy

Exkurs: Dynamic Proxy

- package `java.lang.reflect`
- dynamisch erzeugter Proxy implementiert Liste von Interfaces
  - mit statischer Methode Proxy.newProxyInstance erzeugt
- Proxy-Objekt hat gleiches Interface wie ursprüngliches Objekt, ruft aber InvocationHandler auf

```
public interface InvocationHandler {
        Object invoke(Object proxy, Method method, Object[] args)
                                                throws Throwable
}
```

```
public calls Proxy {
        public static Object newProxyInstance(
                                                ClassLoader loader,
                                                Class<?>[] interfaces,
                                                InvocationHandler h )
                        throws IllegalArgumentException;
        public static Class<?> getProxyClass(ClassLoader loader,
                                                Class<?>... interfaces)
                        throws IllegalArgumentException
        …
}
```

Anwendung:

```
Foo f= (Foo)Proxy.newProxyInstance(null, new Class[] {Foo.class}, handler);
```

Beispiel Dynamic Proxy: TraceHandler (1)

- Beispiel: TraceHandler
  - TraceHandler realisiert InvocationHandler
  - zuerst Trace-Ausgabe
  - dann Aufruf der eigentlichen Methode mit den Argumenten

```
class TraceHandler implements InvocationHandler{
   public TraceHandler(Object t) {
      target = t;
 }

 public Object invoke(Object proxy, Method m, Object[] args)
             throws Throwable {
       System.out.print(target + "." + m.getName() + "(");
       if (args != null) {
          for (int i = 0; i < args.length; i++) {
            System.out.print(args[i]);
            if (i < args.length - 1)
               System.out.print(", ");
          }
       }
       System.out.println(")");
       return m.invoke(target, args);
   }
   private Object target;
}
```

Ausgabe der Traceinfo

Aufruf der eigentlichen Methode

Beispiel Dynamic Proxy: TraceHandler (2)

- Test mit Proxy für `Integer`-Werte
- mit Trace der `compareTo`-Methode des `Comparable`-Interfaces

```
public class ProxyTest {
   public static void main(String[] args) {
      Object[] elements = new Object[1000];

      // fill elements with proxies for the integers 1 ... 1000
      for (int i = 0; i < elements.length; i++) {
         Integer value = i + 1;
         Class[] interfaces = value.getClass().getInterfaces();
         InvocationHandler handler = new TraceHandler(value);
         Object proxy = Proxy.newProxyInstance(null, interfaces, handler);
         elements[i] = proxy;
      }

       Integer key = …;
      // search for the key
      int result = Arrays.binarySearch(elements, key);

      // print match if found
      if (result >= 0)
          System.out.println(elements[result]);
   }
}
```

{ Comparable.class }

```
500.compareTo(547)
750.compareTo(547)
625.compareTo(547)
562.compareTo(547)
531.compareTo(547)
546.compareTo(547)
554.compareTo(547)
550.compareTo(547)
548.compareTo(547)
547.compareTo(547)
547.toString()
547
```

Remote-Objekte

Motivation

Grundarchitektur

Implementierung von Remote-Objekten

Parameterübergabe

Callbacks

Distributed Garbage Collection

Verteilung und Nachladen von Code

Parameterübergabe und Rückgabewerte

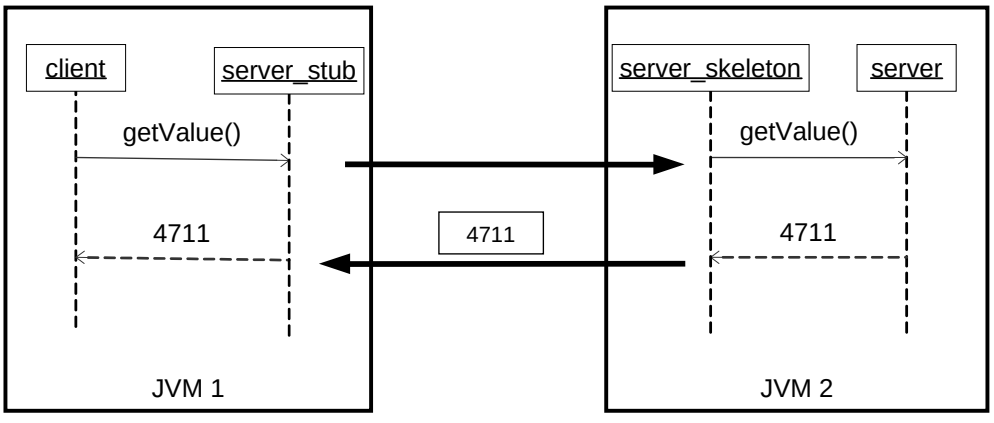
Bei RMI sind folgende Arten der Parameterübergabe und Rückgabewerte zu unterscheiden:

- **Basisdatentypen** (int, double, boolean, etc.)
  - ➔ **Werte** werden über das Netzwerk übertragen
- **Serialisierbare Objekte** (implementieren Interface `Serializable`)
  - ➔ Objekte werden serialisiert über das Netzwerk übertragen

    und auf der anderen Seite eine **Kopie** aufgebaut
- **Remote-Objekte** (implementieren Interface `Remote`)
  - ➔ Es wird auf der empfangenden Seite ein **Stub** für das Objekt aufgebaut
- **alle anderen** (nicht serialisierbar, nicht Remote)
  - ➔ diese **können nicht als Parameter** übergeben werden

Parameterübergabe bei Basisdatentypen

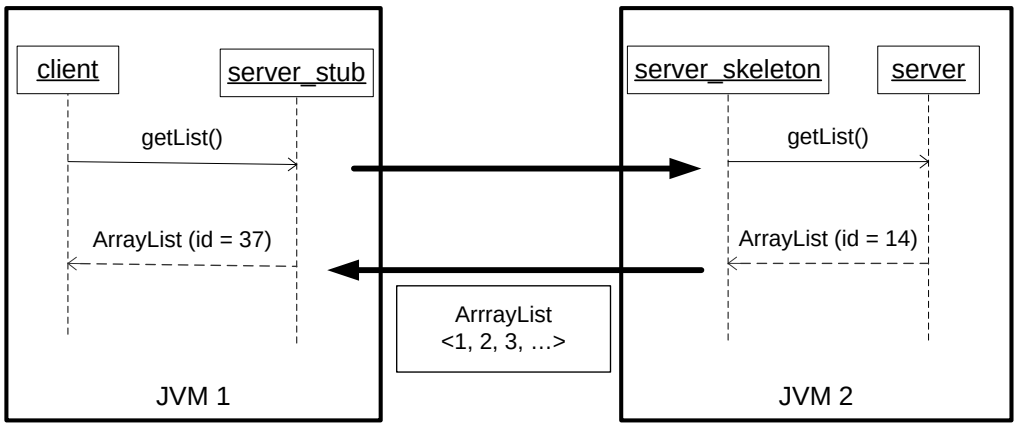
- Auf Senderseite werden Werte „gemarshaled“ und über das Netzwerk gesendet
- Auf der Empfängerseite werden die Werte wieder entpackt und an Client geliefert



Parameterübergabe bei serialisierbaren Objekten

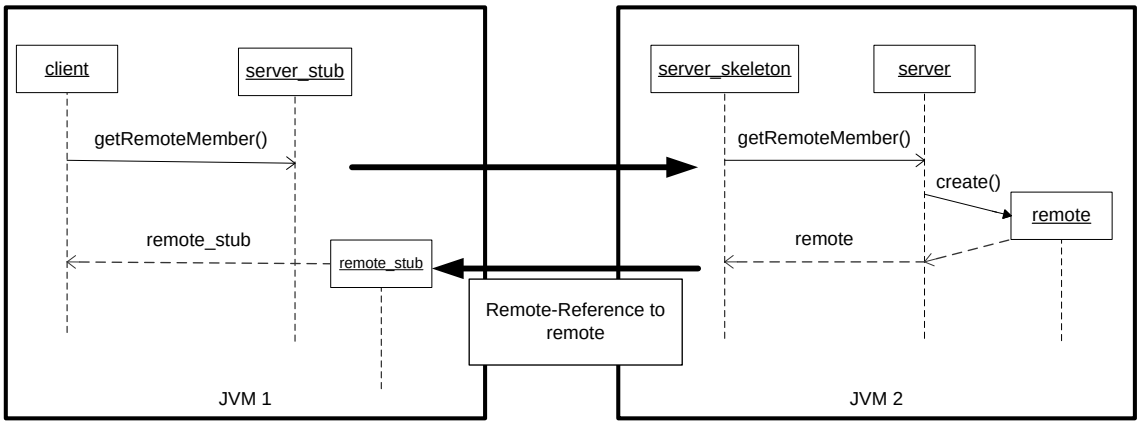
- Inhalt des Objekts wird über den Java-Serialisierungsmechanismus serialisiert und über das Netzwerk übertragen
- auf der Empfängerseite wird ein neues Objekt mit gleichem Inhalt aufgebaut
- Problem, wenn Klasse des serialisierten Objekts auf Empfängerseite nicht bekannt ist!

  ➔ siehe Nachladen von Code



Parameterübergabe bei Remote-Objekten

- Es wird kein Wert übergeben sondern
- über das RMI-System wird eine Remote-Referenz auf das Server-Objekt übermittelt
- auf der Clientseite wird dafür ein Stub erzeugt und eine Referenz auf den Stub dem Empfänger übergeben



Beispiel Banksystem

- `BankManager` ist ein Remote-Objekt, das Zugriff auf Kunden (`Customer`) und Konten (`Accounts`) bietet
- Sowohl `Account` als auch `Customer` sind Remote-Objekte
- `BankManager` bietet den Einstiegspunkte und wird als einziges Objekt registiert
- Vom `BankManager` holt sich das Client-Programm die weiteren Remote-Objekte

Beispiel Banksystem: Interface für Remote-Objekte

```
public interface BankManager extends Remote {

  public Account getAccount(String accountNumber) throws RemoteException;

  public Customer getCustomer(String clientName) throws RemoteException;
}
```

```
public interface Account extends Remote {

  public BankManager getBankManager() throws RemoteException;

  public Customer getCustomer() throws RemoteException;

  public long getBalance() throws RemoteException;

  public long getCash (long amount) throws NoCashAvailableException, RemoteException;
}
```

```
public interface Customer extends Remote {

  public BankManager getBankManager() throws RemoteException;

  public String getName() throws RemoteException;
}
```

Beispiel Banksystem: BankManager Implementierung

- BankManager verwaltet Mengen von Kunden und Konten
- BankManager erlaubt Zugriff auf Konto und Kunde

```
public class BankManagerImpl extends UnicastRemoteObject implements BankManager {
  private Hashtable accounts;
  private Hashtable customers;

  // public No-argument constructor
  public BankManagerImpl() throws java.rmi.RemoteException {
    initialize();
  }

  public Account getAccount(String accountNumber)
      throws RemoteException {
    return (Account) accounts.get(accountNumber);
  }

  public Customer getCustomer(String customerName)
      throws RemoteException {
    return (Customer) clients.get(customerName);
  }
```

Remote-Objekt

Remote-Objekt

Beispiel Banksystem: Server-Programm

- Server-Programm legt `BankManager` an und registriert diesen als einziges Objekt

```
public static void main(String args[]) {
   new BankSystemServer();
   UnicastRemoteObject bmr = null;

   try {
     BankManagerImpl bm = new BankManagerImpl();
   } catch (RemoteException remoteException) {
     System.err.println("Failure during object export to RMI: " + remoteException);
   }

   // Register the BankManager object as a server with an external name
   try {
     Naming.rebind("//localhost/BankSystem", bmr);
   } catch (RemoteException remoteException) {
     System.err.println("Failure during Name registration: " + remoteException);
   } catch (MalformedURLException malformedException) {
     System.err.println("Failure during Name registration: " + malformedException);
   }

}
```

Beipiel Banksystem: Client-Programm

- Client-Programm holt sich über RMIRegistry den `BankManager`
- greift damit auf `Account`-Objekte und `Customer`-Objekte zu

```
try {

     bm = (BankManager) Naming.lookup("rmi://localhost:1099/BankSystem");

   } catch (MalformedURLException malformedException) {
     System.err.println("Bad URL: " + malformedException);
   } catch (NotBoundException notBoundException) {
     System.err.println("Not Bound: " + notBoundException);
   } catch (RemoteException remoteException) {
     System.err.println("Remote Exception: " + remoteException);
   }

   try {
     Account account = bm.getAccount("4461");
     Customer customer = account.getCustomer();
     String name = customer.getName();
     long cash = account.getBalance();

     NumberFormat currencyFormat = NumberFormat.getCurrencyInstance(Locale.US);
     String balanceString = currencyFormat.format(cash);
     System.out.println(name + "'s account has " + balanceString);

   } catch (RemoteException remoteException) {
     System.err.println(remoteException);
   }
```

Remote-Objekte

Motivation

Grundarchitektur

Implementierung von Remote-Objekten

Parameterübergabe

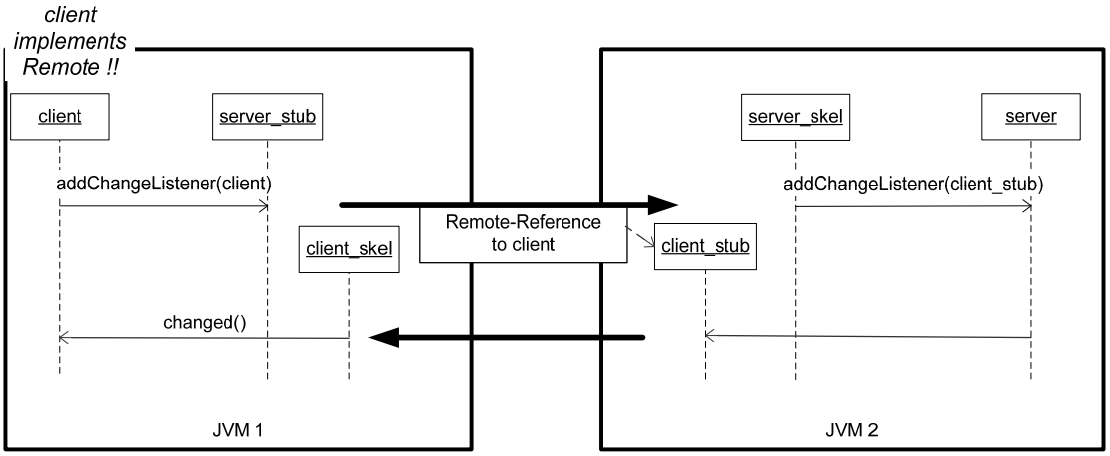
Callbacks

Distributed Garbage Collection

Verteilung und Nachladen von Code

Callbacks

- In machen Fällen ist es notwendig, dass der Server dem Client eine Meldung schickt.
  - z.B. Benachrichtung über Änderungen
- In diesem Fall sind die Rolle von Client und Server zu vertauschen
  - am Client muss ein Remote-Objekt existieren
  - eine Remote-Referenz wird dem Server übergeben
  - am Server wird ein Stub für das Remote-Client-Objekt angelegt



Beispiel Banksystem: Remote Change Notification

- Remote-Listener-Interface für Client

```
public interface RemoteAccountListener extends Remote {
   public void accountChanged(RemoteAccountEvent e) throws RemoteException;
}
```

- BankManager wird um Remote-Methode addAccountListener erweitert

```
public interface BankManager extends Remote {
   public void addRemoteAccountListener(RemoteAccountListener l) throws RemoteException;
}
```

```
public class BankManagerImpl implements BankManager {
  public void addRemoteAccountListener(RemoteAccountListener l) throws RemoteException {
    listeners.add(l);
  }
```

- Der Client muss RemoteAccountListener implementieren und sich als Listener bei der Bank anmelden;
  Er kann dann von allen Kontobewegungen benachrichtigt werden

```
public class BankUser implements RemoteAccountListener {
   public void accountChanged(RemoteAccountEvent e) throws RemoteException { … }
}
```

```
   ...
   // add client as account listener at the bank manager
   bm.addRemoteAccountListener(this);
   ...
```

Remote-Objekte

Motivation

Grundarchitektur

Implementierung von Remote-Objekten

Parameterübergabe

Callbacks

Distributed Garbage Collection

Verteilung und Nachladen von Code

Distributed Garbage Collection

- Garbage Collector gibt Objekte frei, wenn sie nicht mehr referenziert werden
- Bei Remote-Objekten ist es dem Garbage Collector aber nicht möglich zu entscheiden, ob Remote-Clients das Objekt noch referenzieren

➔ Distributed Garbage Collector arbeitet mit:

- *Reference Counting*: Es werden die Zugriffe von Remote-Clients auf die Remote-Objekte gezählt
- *Lease Time*: Greift ein Client eine bestimmte Zeit – der Lease Time – nicht auf das Remote-Objekt zu, wird angenommen, dass der Client das Objekt nicht mehr benötigt.

Konsequenz: Beim Client ist es möglich, dass Remote-Objekte verschwinden und er mit ungültigen Remote-Referenzen umgehen können muss.

Lease Time:

- System Property `java.rmi.dgc.leaseValue`
- Standardwert 10 min
- Einstellung als Option bei java: `java -Djava.rmi.dgc.leaseValue=20000`

Remote-Objekte

Motivation

Grundarchitektur

Implementierung von Remote-Objekten

Parameterübergabe

Callbacks

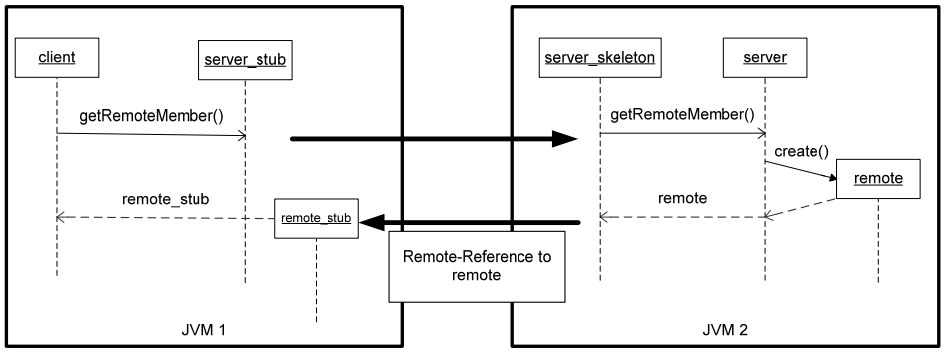
Distributed Garbage Collection

Verteilung und Nachladen von Code

Verteilung und Nachladen von Klassen

- Bei Zugriff auf Remote-Objekte ist es nötig, dass Code vom Server nachgeladen wird
  - für Stubs
  - für Parameter und Rückgabewerte

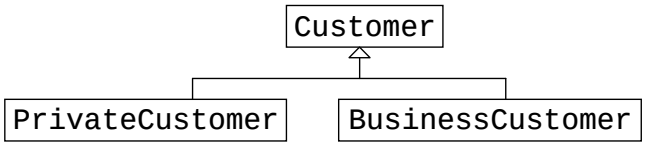
Beispiel: Stub



Beispiel Nachladen von Klassen

```
public interface BankManager extends Remote {

  public Customer getCustomer(String clientName) throws RemoteException;
}
```



Es existieren Spezialisierungen von `Customer` ! Diese sind Client eventuell nicht bekannt!

```
public class BankManagerImpl extends UnicastRemoteObject implements BankManager {
  public Customer getCustomer(String customerName) throws RemoteException {
     …
     return businessClients.get(customerName);
  }
```

Client: Müsste alle Spezialisierungen von Customer kennen

Konretes Objekt vom Typ `PrivateCustomer` oder `BusinessCustomer`

```
  try {

     Customer customer = account.getCustomer();
  } catch (RemoteException remoteException) {
      System.err.println(remoteException);
  }
```

Nachladen von Code

- Um Code von einer externen Site nachladen zu können, ist folgendes zu tun
  - es muss beim Client ein `RMISecurityManager` installiert sein (sonst kann kein externer Code gelader werden)
  - Es müssen Permissions für den Client gesetzt werden:
    - Socketverbindung zum Server für RMI über Port 1099>
    - Socketverbindung zum Nachladen von Code hier über Webserver, d.h. Port 80 (oder 8080)

Beispiel:

Clientprogramm:

```
public class MyClient {
    public static void main(String[] args) {
        System.setSecurityManager(new RMISecurityManager());
        System.setProperty("java.security.policy", "client.policy");
        ...
```

Policy-Datei für Client:

```
grant {
    permission java.net.SocketPermission "server-url:1024-65535", "connect";
    permission java.net.SocketPermission "server-url:80", "connect";
    permission java.net.SocketPermission "server-url:8080", "connect";
}
```

Verteilung des Klassencodes

- Die Klassen sollen zur Verteilung auf 3 Teile aufgeteilt werden:

  server:
  - alle Klassendateien, die für die Ausführung des Servers erforderlich sind
  - alle Stub-Klassen

  download:
  - alle Klassendateien, die dynamisch vom Client nachgeladen werden sollen
  - inklusive alle abhängigen (z.B. Basisklassen und Interfaces)

  client:
  - alle Klassendateien, die unmittelbar vom Client verwendet werden
  - die Policy-Datei (z.B. client.policy)

- Die 3 Teile werden dann folgend verteilt:

  server:
  - auf dem Rechner des Servers

  download:
  - bei Verwendung eines Webservers ins download-Verzeichnis des Webserver

  client:
  - auf den Rechern des Client

Starten der verteilten Programme

- Am Server
  - Starten des RMIRegistry-Programms

```
Microsoft Windows XP

> rmiregistry
```

  - Starten des Server-Programms mit Angabe des Download-Verzeichnisses als Codebase

```
> java  -Djava.rmi.server.codebase=http://localhost/download/
    MyServerApp
```

- Am Client
  - Starten des Client-Programms, hier zusätzlich unter Angabe einer Policy-Datei

```
> java -Djava.security.policy=client.policy MyClientApp
```

Literatur

- Horstmann, Cornell, Core Java 2, Band 2 Expertenwissen, Markt und Technik, 2002: Kapitel 5
- Krüger, Handbuch der Java-Programmierung, 3. Auflage, Addison-Wesley, 2003, http://www.javabuch.de: Kapitel 46
- Fundamentals of RMI, Short Course, http://java.sun.com/developer/onlineTraining/rmi/