E Überblick über die 4. Übung

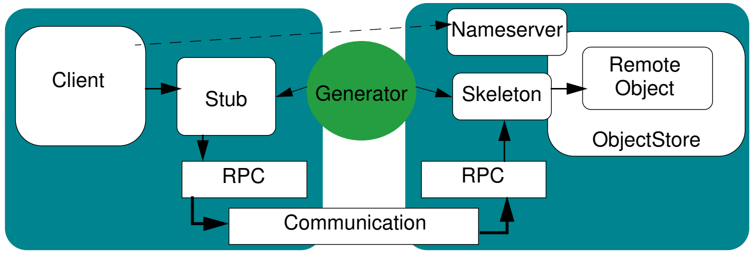
- ORB
- ANT
- ClassLoader

E.1 Object Request Brokers

- ermöglichen Methodenaufrufe an entfernten Objekten (Objekte in anderen JVM)
- Beispiel-ORBs: RMI, JavaIDL

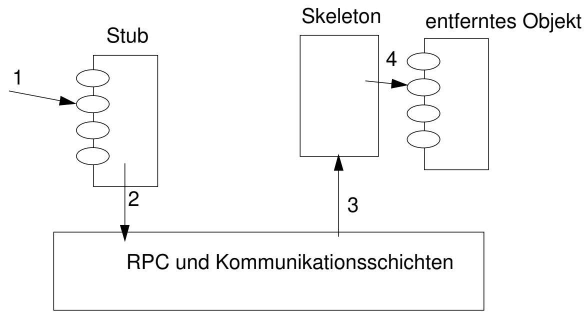
1 Komponenten eines ORBs

- *Kommunikationsschicht*: tauscht Daten zwischen zwei Rechnern aus
- *RPC Schicht*: definiert die Aufrufsemantik und das Marshalling
- *Object Store*: verwaltet den Lebenszyklus der Objekte
- *Stub / Skeleton Generator*: erzeugt Code für die Stubs und die Skeletons
- *Nameserver*: findet Objekte anhand deren Namen



2 Stubs und Skeletons

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



2 Stub

- implementiert den gleichen Typen wie das entfernte Objekt (gleiches Interface)
- verpackt einen Methodenaufruf in ein Request-Objekt:
  - Objekt ID, Methoden ID, Parameter
- verwendet die RPC-Schicht um eine Anforderung zu versenden
- transformiert das Rückgabeobjekt in den entsprechenden Typ
- Beispiel: erzeugte Methode (ohne Ausnahmebehandlung)

```
public int deposit(int param0) {
   Object[] parameters = new Object[1];
   parameters[0] = new Integer(param0);
   Request req = new Request(ctx, oid, 9, parameters);
   Object ret = req.send();
   return ((Integer)ret).intValue();
 }
```


2 Stub (2)

- Beispiel: erzeugte Stub-Klasse

```
package test;
import orb.*;
public class AccountImpl_Stub implements test.Account {
  int oid;
  ClientContext ctx;
  public AccountImpl_Stub(int oid, ClientContext ctx) {
    this.oid = oid;
    this.ctx = ctx;
  }
  // erzeugte Methoden
  // ...
}
```


2 Skeleton

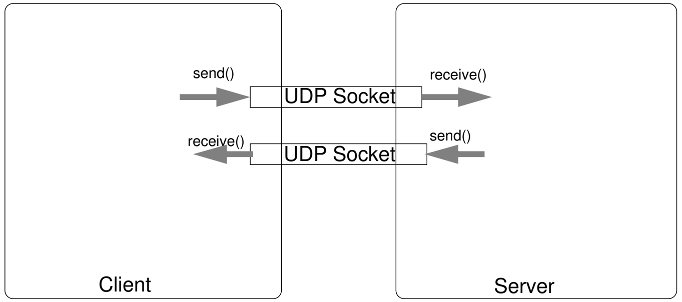
- ruft Methoden am "echten" Objekt auf
- notwendige Informationen:
  - Objektreferenz
  - Methoden ID
  - Parameter

2 Skeleton Beispiel

```
package test;
import orb.*;
public class AccountImpl_Skel  implements Skeleton {
  AccountImpl  object;
  public AccountImpl_Skel() { /* ... */ }
  public void init(int oid, Object object) {
    this.oid = oid;
    this.object = (AccountImpl) object;
  }
  public Object invoke(int mid, Object[] parameters)
                                    throws Exception {
    switch(mid) {
      ...
      case 9: {
         return new Integer(
    object.deposit( ((Integer)parameters[0]).intValue() ));
      }
    }
  }
}
```

3 Kommunikationsschicht

- zuständig für den Datenaustausch zwischen zwei Rechnern
- verwendet **`DatagramSocket`** (UDP)

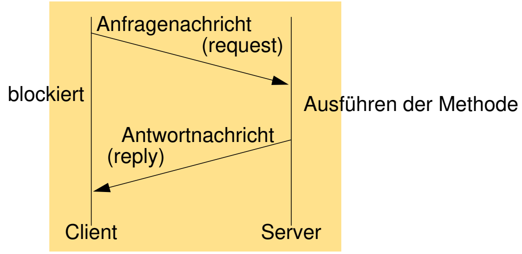


4 RPC-Schicht

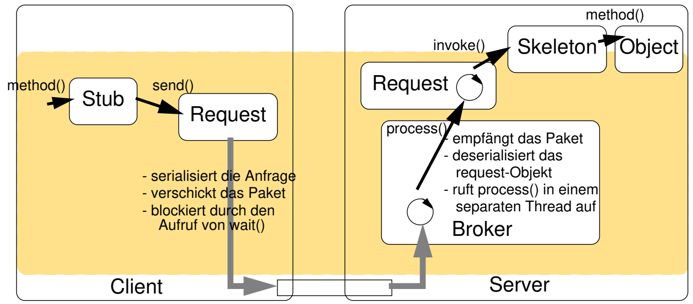
- erledigt die Weiterleitung von Methodenaufrufen
- wird von den Stubs und den Skeletons verwendet
- verwendet die Kommunikationsschicht um Bytes zu versenden

4 RPC

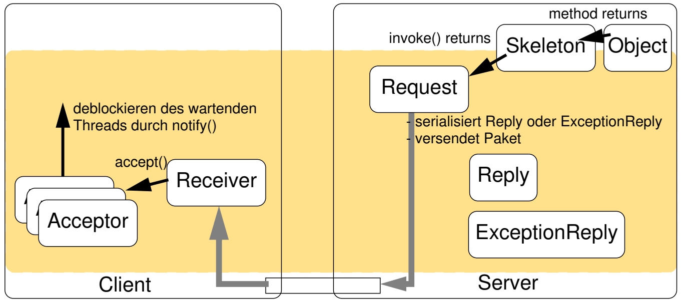
- einfachster RPC ist ein primitives request/reply-Protokoll:



4 Request



4 Reply



4 Marshalling

- RPC-Schicht verpackt die Parameter in ein Anfragepaket und den Rückgabewert in ein Antwortpaket == Marshalling
- kopiere alle Parameter zum Server (besser wäre es, eine Referenz zu verschicken wenn ein Parameter ein *remote Interface* implementiert)
- verwende ObjectStreams/ByteArrayStreams/Datagrams um Objekte zu verschicken:

```
ByteArrayOutputStream stream = new ByteArrayOutputStream();
ObjectOutputStream out = new ObjectOutputStream(stream);
out.writeObject(...);
byte[] buf = stream.toByteArray();
DatagramPacket packet = new DatagramPacket(buf, ... );
```

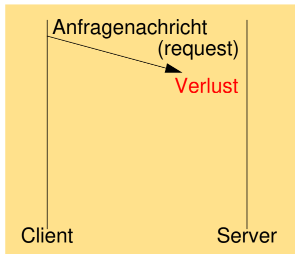
- nimm an, dass alle Parameter in ein `DatagramPacket` passen

4 Fehlerbehandlung

- Implementierung einer bestimmten Aufrufsemantik
  - exactly once
  - at least once
  - at most once
  - last of many
  - ...
- abhängig von der Servicequalität der Kommunikationsschicht
- muss mit Kommunikationsfehlern umgehen können:
  - verlorene Pakete
  - veränderte Reihenfolge (non-FIFO)
  - duplizierte Pakete
  - veränderte Pakete

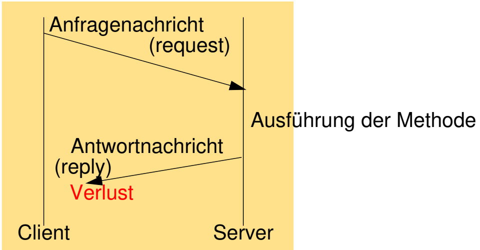
4 Fehlerbehandlung (2)

- Kommunikationsfehler können zu folgenden Problemen führen:
  - Verlust einer Anfragenachricht



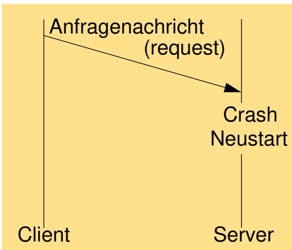
4 Fehlerbehandlung (3)

- Kommunikationsfehler können zu folgenden Problemen führen:
  - ◆ Verlust einer Antwortnachricht



4 Fehlerbehandlung (4)

- Kommunikationsfehler können zu folgenden Problemen führen:
  - ◆ Fehler während des Ausführens der Methode



4 Fehlerbehandlung (5)

- Annäherung an *exactly once*:
  - ◆ Um verlorene Anfragenachrichten entgegenzuwirken werden die Pakete nach einem Timeout erneut gesendet.
  - ◆ Zum vermeiden von mehrfach Ausführungen bei doppelt gesendeten Paketen werden IDs verwendet, welche bei einem erneuten versenden der gleichen Anfrage nicht verändert werden.
  - ◆ Der Server soll abgesendete Antworten puffern, für den Fall, dass eine Antwort verloren geht und um eine erneute Ausführung der Methode zu vermeiden.
    Der Puffer kann mithilfe einer **`Hashtable`** implementiert werden, die Anfrage IDs auf Antworten abbildet.
  - ◆ Den Absturz eines Servers zu verkraften ist etwas schwieriger ..... daher ignorieren wir das Problem.

5 Object Store

- Interface:
  - ◆ **`int registerObject(Object obj)`**
    gibt eine eindeutige Objekt-ID zurück.
  - ◆ **`Object lookupByID(int oid)`**
    liefert das Objekt zur gegebenen Objekt-ID
- Implementationstechnik:
  - ◆ **`java.util.Hashtable`** kann verwendet werden um die Abbildung zwischen Objekt-IDs und Objekten vorzunehmen (Hinweis: die Hashtabelle kann nur Objekte speichern, die *OID* sollte daher in ein **`Integer`**-Objekt gekapselt werden.)

6 Nameserver

- ist aufgeteilt in einen ID-Finder und einen Proxy-Erzeuger
- ID Finder
  - bildet Namen auf OIDs ab:
    - `void bind(String name, int oid)`
    - `int lookupID(String name)`
  - verwendet **`Hashtable`**
  - ist selbst als entferntes Objekt mit der OID 1 implementiert
- ein realistischer Nameserver würde Referenzen auf Stub-Objekte zurückliefern anstatt OIDs
- das erfordert, dass zusammen mit der OID der Klassenname des Stubs zurückgegeben wird.

E.2 Bauen von Projekten mit ANT

- Was ist ANT?
- ANT im Detail
  - Projekte
  - Properties
  - Targets
  - Core Tasks
  - Eigene Tasks
- Zusätzliche Informationen

6 Nameserver

- Stub-Erzeuger
  - sucht die ID (und den Klassennamen); erzeugt daraufhin ein Stub-Objekt
    - `Object lookup(String name)`
  - der Nameserver lädt die Klasse und erzeugt eine Instanz der Klasse

1 Was ist ANT?

- ANT ist ein Werkzeug zum Übersetzen von größeren Java Projekten.
- Arbeitsschritte werden nur angestossen, wenn die Abhängigkeiten sich verändert habe.
- Wie bei “make” können Ziele (targets) und Abhängigkeiten (dependences) angegeben werden. Die Beschreibung erfolgt in einer XML-Datei.
- ANT ist eine Java-Anwendung und kann durch Klassen flexibel erweitert werden.

2 Warum nicht das gute alte “make”?

- Make berücksichtigt die Eigenheiten von Java nicht
  - Verzeichnisstruktur von Paketen müssen explizit angegeben werden
  - Die teilweise automatische Übersetzung der Klassen durch den Compiler wird nicht berücksichtigt
- Beim Aufruf von javac wird jedesmal eine neue JVM gestartet (ist langsam)
- Make ist nicht durch Java-Klassen flexible erweiterbar

3 Grundlagen

- Jedes Projekt besitzt ein *Buildfile* (build.xml)
- Ein Buildfile beschreibt mehrere *Targets*
  - Targets definieren typische Aufgaben des Übersetzungsprozesses
  - Targets können Abhängigkeiten zu anderen Targets besitzen
- Jedes Taget kann sich aus verschiedenen *Tasks* zusammensetzen
  - Tasks werden sequentiell hintereinander abgearbeitet
- Konfiguration des Übersetzungsprozesses durch *Properties*

4 Beispiel: Buildfile <project>

- Das Buildfile beschreibt den Generierungsprozess in XML
- Das <project>-Tag beschreibt das Projekt und enthält beliebig viele Properties und Targets.

```
<?xml version="1.0" encoding="UTF-8"?>
<project basedir="." default=”compile" name=”example">
<!--
        [ property definitions ]
        [ path and patternset ]
        [ targets ]
-->
</project>
```


4 Beispiel: Buildfile <property>

- Properties dienen im Buildfile zur einfachen Konfiguration
- Properties können mit Hilfe des <property>-Tags definiert werden
- Zusätzlich gibt es Properties für bestimmte Aufgaben zum Beispiel für Verzeichnispfade (<path>-Tag)

4 Beispiel: Buildfile <property>

```
<?xml version="1.0" encoding="UTF-8"?>
<project basedir="." default="all" name="example">
  <property name="srcDir" value="src"/>
  <property name="buildDir" value="classes"/>

  <path id="classpath">
     <pathelement location="lib/"/>
     <pathelement path="${java.class.path}/"/>
     <pathelement path="${junit}"/>
  </path>

  <!--
         [ targets ]
   -->
</project>
```


5 Beispiel: Buildfile <target>

- Die Targets dienen zum Beschreiben einzelner Abschnitte des Übersetzungsprozesses
- Sie besitzen immer einen Name
- Können weitere Targets als Abhängigkeit benennen
- Sollten eine Beschreibung besitzen, welche während der Übersetzung angezeigt wird
- Umfasst mehrere Arbeitsschritte (Tasks)
- Ausgehend vom im <project>-Tag benannten default-Target werden alle Abhängigkeiten überprüft. Ist eine Abhängigkeit neuer als das Ziel, so werden die Tasks abgearbeitet.

5 Beispiel: Buildfile <target>

```
<target name="all" depends="test, doc, dist"
     description="runs test, javadoc and dist"/>

<target name="clean" depends="cleanTestLog"
   description="deletes all generated files">
    <delete dir="${buildDir}"/>
    <delete dir="${docDir}"/>
    <delete file="${distFile}"/>
</target>

<target name="cleanTestLog"
   description="deletes generated JUnit log files">
    <delete>
      <fileset dir=".">
        <include name="TEST-*.txt"/>
      </fileset>
    </delete>
</target>
```


5 Beispiel: Buildfile <target>

```
<target name="prepare"
  description="creates output directories">
    <mkdir dir="${buildDir}"/>
    <mkdir dir="${docDir}"/>
</target>

<target name="compile" depends="prepare"
  description="compile source files">
    <javac srcdir="${srcDir}" destdir="${buildDir}"
       classpathref="classpath"/>
</target>
```

5 Beispiel: Buildfile <target>

```
<target name="test" depends="compile, cleanTestLog"
  description="runs all JUnit tests">
    <junit printsummary="yes">
      <classpath refid="classpath"/>
      <batchtest>
        <fileset dir="${srcDir}">
       <include name="**/*Test.java"/>
     </fileset>
      </batchtest>
    </junit>
</target>

<target name="doc" depends="compile"
  description="generates javadoc">
  <javadoc sourcepath="${srcDir}" destdir="${docDir}"
                packagenames="org.blub.*"
             classpathref="classpath" />
</target>
```


6 Beispiel: Ausgabe von ANT

- ANT für Aufgabe 3 könnte so aussehen:

```
faui40c: 11:39 felser/afg3 > ant
Buildfile: build.xml

prepare:
    [mkdir] Created dir: /proj/i4mw/felser/afg3/classes
    [mkdir] Created dir: /proj/i4mw/felser/afg3/docs

comm:
    [javac] Compiling 3 source files to /proj/.../afg3/classes

stubgen:
    [javac] Compiling 1 source file to /proj/.../afg3/classes

orb:
    [javac] Compiling 15 source files to /proj/.../afg3/classes

all:

BUILD SUCCESSFUL
Total time: 10 seconds
faui40c: 11:39 felser/afg3 >
```


7 Core Tasks

- Ant
  Ant, AntCall, AntStructure
- Basics
  Condition, Chmod, Copy, Delete, Echo, Exec, Fail, Java, MkDir, Move, PathConvert, Property, Sleep, Touch, Tstamp
- Dokumentation und Test
  Javadoc, Mail, Record,
- Package
  Jar, SignJar, Gzip, Zip, GUzip, War, Tar
- Versionskontrolle
  CVS, CVSPass, Patch

8 Optionale Tasks

- Tools
  JavaCC, Javah, JDepend, JJTree, JProbe
- Dokumentation und Test
  JUnit, JUnitReport, Mime Mail, Test
- Package
  Cab, RPM
- Versionskontrolle
  Clearcase, PVCS, SourceSave

9 Entwicklung eigener Tasks (1)

- Vorgehensweise bei der Entwicklung eigener Tasks
  - (1) Ableiten der Klasse **org.apache.tools.ant.Task**
  - (2) Implementieren einer **set**-Methode für jedes Attribut des Tasks

    ```
    public void setAttrName(type attrName);
    ```

  - (3) Implementieren einer **create** oder **add**-Methode für jedes untergeordnete Element

    ```
    public ChildTask creatChildTask();
    ```

    oder

    ```
    public void addChildTask(ChildTask childTask);
    ```

  - (4) Implementieren der eigentlichen Task-Methode

    ```
    public void execute();
    ```


9 Entwicklung eigener Tasks (2)

- Programmcode eines Beispieltasks:

```
public class ZakTask extends Task {
 private String msg;

 // The method executing the task
 public void execute() throws BuildException {
   System.out.println(msg);
 }

 // The setter for the "message" attribute
 public void setMessage(String msg) {
    this.msg = msg;
 }
}
```


9 Entwicklung eigener Tasks (3)

- Einbindung des Tasks in ein Buildfile:

```
<project name="TestZakTask" default="main" basedir=".">

  <taskdef name="mytask" classname="ZakTask"/>

  <target name="main">
    <mytask message="Hello World! ZakTask works!"/>
  </target>

</project>
```


10 Zusätzliche Informationen

- Mehr Informationen unter
  - ◆ http://jakarta.apache.org/ant/
  - ◆ Ant in Anger - http://jakarta.apache.org/resources.html
- Ant im CIP-Pool:

```
#java
setenv JAVA_HOME /local/java-1.4.2

# ant
setenv ANT_HOME /local/jakarta-ant
set path=( /local/jakarta-ant/bin $path)
```

E.3 ClassLoader

- Wie werden Klassen in die Java Virtual Machine (JVM) geladen?
  - vom lokalen Dateisystem (**`CLASSPATH`**)
  - von einer Instanz eines **`ClassLoader`**
- ... und wann?
  - Wenn sie zum ersten Mal gebraucht werden!

```
class Test {
    public String toString() {
        Hello hello = new Hello();
    }
}
```

```
ClassLoader cl = new ...;
Class c = cl.loadClass("Test");
Object t=c.newInstance();
t.toString();
```

(1) Die Klasse **`Test`** wird vom Klassenlader **`cl`** geladen

(2) **`toString`** wird ausgeführt, dabei wird die Klasse **`Hello`** benötigt

(3) der Klassenlader von **`Test`** (= **`cl`**) wird beauftragt **`Hello`** zu laden

E.3 ClassLoader (2)

- Ein ClassLoader erzeugt einen Namensraum.
  - Es können Klassen mit demselben Namen innerhalb einer JVM existieren, wenn sichergestellt ist, dass sie von verschiedenen Classloader-Instanzen geladen werden.
- **`java.lang.ClassLoader`**
  - kann eine Klasse aus einem Array von Bytes erzeugen, welches dem Format einer Klassendatei entspricht (**`defineClass()`**)
  - wenn diese Klasse andere Klassen verwendet, wird der ClassLoader angewiesen diese Klassen zu laden (**`loadClass(String name)`**)
  - nach zuvor geladenen Klassen kann mittels **`Class findLoadedClass(String name)`** gesucht werden
- Mehrere **`ClassLoader`** können in einer JVM aktiv sein.

1 ClassLoader - Beispiel

```
import java.io.*;

public class SimpleClassLoader extends ClassLoader {
    public synchronized Class loadClass(String name,
                                        boolean resolve) {

        Class c = findLoadedClass(name);
        if (c != null) return c;

        try {
            c = findSystemClass(name);
            if (c != null) return c;
        } catch(ClassNotFoundException e) {}

        try {
            RandomAccessFile file =
               new RandomAccessFile("test/" + name + ".class", "r");
            byte data[] = new byte[(int)file.length()];
            file.readFully(data);
            c = defineClass(name, data, 0, data.length);
        } catch(IOException e) {}

        if (resolve)
            resolveClass(c);
        return c;
    }
}
```


2 Beispiel Appletviewer

- Java Klassen, die vom Browser über das Netzwerk geladen wird
- Beispiel: applet.html:

```
<html>
  <body>
    <APPLET  CODE=SimpleApplet.class
             CODEBASE=http://www4/~felser/ WIDTH=100 HEIGHT=100 >
    </APPLET>
  </body>
</html>
```

- Beispiel: SimpleApplet.java

```
import java.applet.Applet;

public class SimpleApplet extends Applet {
    public void init() {
        // ...
    }
    //..
}
```

- Starten mittels Browser oder: **`appletviewer applet.html`**

2 Beispiel Appletviewer (2)

- **`appletviewer`** verfügt über einen speziellen ClassLoader
- Folgende Schritte laufen beim Start des **`appetviewer`** ab:
  1. Eine Zeichenkette wir in ein Objekt vom Typ **`URL`** umgewandelt.
  2. Der Zeichenstrom des **`URL`**-Stroms wir nach einem **`<applet>`**-Tag durchsucht.
  3. Ein **`AppletClassLoader`** wird mit Inhalt des Tags **`codebase`** erzeugt.
  4. Die mit dem Tag **`code`** festgelegte Klasse wird durch den **`AppletClassLoader`** geladen.
- Der **`AppletClassLoader`** wird mit einem Objekt vom Typ **`URL`** initialisiert:
  `AppletClassLoader(URL baseURL)`
- **`loadClass(String name)`** erzeugt eine URL aus **`baseURL`** und dem Namen der Klasse und ruft dann **`loadClass(URL url)`** auf
- **`loadClass(URL url)`** lädt die Klasse unter Verwendung von **`defineClass()`**