Netzwerkprogrammierung mit RMI

PROG 2: Einführung in die Programmierung für Wirtschaftsinformatiker

Steffen Helke

Technische Universität Berlin
Fachgebiet Softwaretechnik

1. Juli 2013

Übersicht

- SMTP
- RMI

Teil V der Vorlesung PROG 2

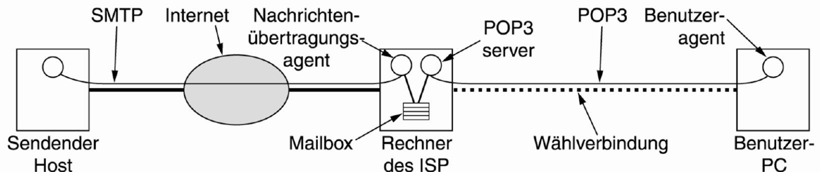
Netzwerkprogrammierung

Maildienste mit SMTP

Quelle: *Inhalt & Gestaltung nach Vorlesungsfolien von Peter Pepper und Odej Kao, TU Berlin*
Methodische- und Praktische Grundlagen der Informatik 4 (MPGI 4), WS 2010/11 bzw. WS 2011/12

Simple Mail Transfer Protocol - SMTP

- E-Mail-Übertragung basiert auf TCP-Verbindung
- speziell SMTP ist einfaches ASCII-Protokoll
- Mail-Server hört auf Port 25
- Client baut Verbindung auf und wartet, bis Server mit Kommunikation beginnt
- SMTP erfordert permanente Internetverbindung, anders als bei IMAP (Internet Message Access Protocol)



Ablauf einer Kommunikation via SMTP

1. Client spricht Mail-Server auf Port 25 an

```
telnet xy.tu-berlin.de 25
```

2. Server gibt Antwort, falls SMTP-Dienst verfügbar

```
220 xy.tu-berlin.de ESMTP Sendmail 8.11.6/8.9.3;
Mon, 25 Jun 2012 7:55:20 + 0200
```

3. Fortsetzung der Verbindung durch Client mit Befehl *HELO*

```
HELO z.tu-berlin.de
```

4. Falls Server akzeptiert, gibt es eine Antwort

```
250 xy.tu-berlin.de Hello z.tu-berlin.de[139.235.66.186]
```

5. Client erfragt verfügbare Befehle mit *HELP*
6. Antwort des Servers

```
214-Commands supported:
214 AUTH STARTTLS HELO EHLO MAIL RCPT DATA ...
```

Versenden von E-Mails via SMTP

7. Versenden einer E-Mail mit den Befehlen *MAIL FROM*, *RCPT TO*, *DATA* und *QUIT*

```
MAIL FROM: helke@tu-berlin.de
250 OK
```

```
RCPT TO: steffen.helke@tu-berlin.de
250 Accepted
```

```
DATA
354 Enter message, ending with "." on line by itself
```

```
Dies ist eine Testmail fuer die Vorlesung PROG 2
.
250 OK id=1Sj2Rw-0003WI-Aj
```

```
QUIT
221 xy.tu-berlin.de closing connection
Connection closed by foreign host.
```

Wie wird ein SMTP-Client in Java umgesetzt?

Varianten

1. Stream-Socket-Verbindung mit der Klasse *Stream* und *selbst zusammengebauten SMTP-Anfrage-String*

```
"HELO " + InetAddress.getLocalHost().getHostName();
```

```
"MAIL from: " + sender
```

```
"RCPT to: " + receiver
```

```
"DATA " + Textnachricht + "."
```

2. Benutzung von *komfortablen Methoden* der *JavaMail*-API

   ➜ schrittweises Erstellen von Nachrichten

   ➜ Versenden per SMTP, POP, IMAP oder NNTP

Variante 2: Mail mit JavaMail verschicken

➜ zur Benutzung von JavaMail wird *mail.jar* benötigt

Vorgehen

1. Definition einer Session mit wichtigen Eigenschaften

```
Properties prop = new Properties();
props.put("mail.host","mail.tu-berlin.de");
// null ist der fehlende Authentikator
Session mailConnect = Session.getInstance(props, null);
Message message = new MimeMessage(mailConnect);
```

2. Nachricht (MIME) erstellen

```
message.setContent("Hello","text/plain");
// wenn nur Text einzufuegen ist
message.setText("Hello");
message.setSubject("First");
```

Variante 2: Mail mit JavaMail verschicken

3 Festlegen der Adresse

```
Address toAddr =
    new InternetAddress(steffen.helke@tu-berlin.de,
                        "Steffen Helke");
Address ccAddr =
    new InternetAddress(helke@cs.tu-berlin.de");
message.addRecipient(Message.RecipientType.TO, toAddr);
message.addRecipient(Message.RecipientType.CC, ccAddr);
```

4 Versenden der Nachricht

```
Transport.send(message);
```

Teil V der Vorlesung PROG 2

Netzwerkprogrammierung

RMI – Remote Method Invocation

Quelle: *Inhalt & Gestaltung nach Vorlesungsfolien von Peter Pepper und Odej Kao, TU Berlin*
Methodische- und Praktische Grundlagen der Informatik 4 (MPGI 4), WS 2010/11 bzw. WS 2011/12

RMI – Remote Method Invocation

Nachteile einfacher Socket-Programmierung

- Objektkommunikation wird unterschiedlich realisiert
  - ➜ *Kommunikation* zwischen Objekten *auf einem Prozess* ist *einfach*, z.B. durch Methodenaufrufe
  - ➜ *Kommunikation* zwischen Objekten *unterschiedlicher Prozesse* ist *aufwendig*, z.B. durch Socket-basierte Client-Server-Kommunikation

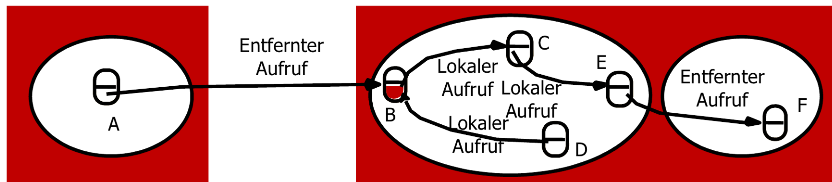
Lösungsidee von RMI

- Realisierung eines verteilten Objektmodells
  - ➜ Zugriffe auf entfernte Objekte werden genauso einfach wie Zugriffe auf lokale Objekte realisiert

Verteiltes Objektmodell

Grundidee

- lokaler und entfernter Zugriff auf Objekte ist möglich
- Objekte, die entfernte Aufrufe enthalten können, werden als *externe* oder *entfernte Objekte* bezeichnet
  ➜ im Beispiel die Objekte *B* und *F*
- Objekte können die Methoden anderer Objekte nur aufrufen, wenn sie deren Referenz kennen



Referenzen und Schnittstellen

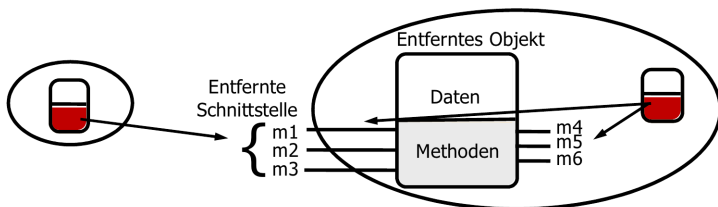
Entfernte Objektreferenz

- nur bei Kenntnis der entfernten Objektreferenz kann ein Objekt die Methode eines entfernten Objekts aufrufen

➜ Umsetzung in Java mit RMI-Protokoll

Entfernte Schnittstelle

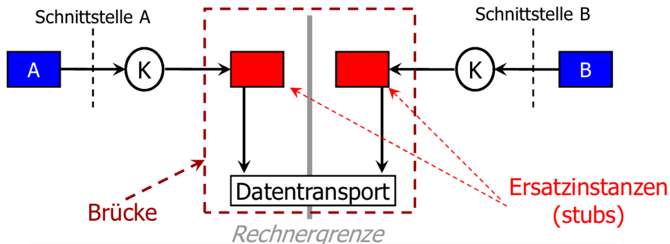
- Festlegung von Methoden, die von externen Objekten aufgerufen werden dürfen



Ersatzinstanzen (Stubs)

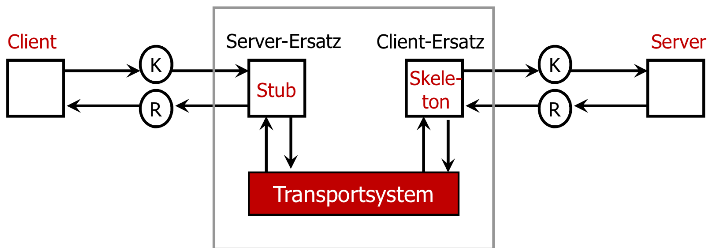
Idee

- Erzeugung der Illusion, dass die entfernten Objekte auch lokal existieren würden ➜ Umsetzung mit *Stubs*
- Stubs unterstützen die gleichen Schnittstellen, wie bei lokaler Kommunikation und einen Transportmechanismus



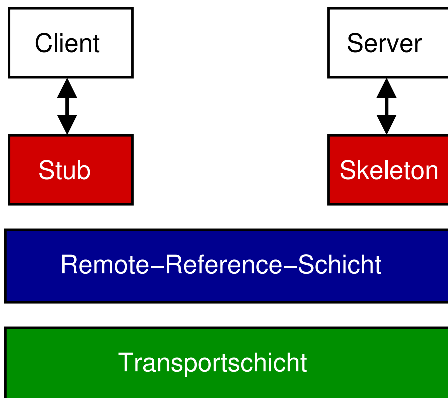
Stubs und Skeletons in Java

- nicht lokal vorhandene Kommunikationspartner werden über Ersatzinstanzen repräsentiert
  - ➜ auf Client-Seite *Stub*
  - ➜ auf Server-Seite *Skeleton*
- Ersatzinstanzen müssen zu übertragene Nachrichten über Transportsystem austauschen



RMI-Architektur

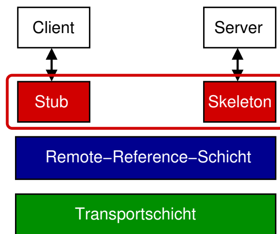
- ➜ möglichst viele Details der Client-Server-Kommunikation sollen für den Programmierer verborgen bleiben



Aufgaben von Stub und Skeleton

Stub auf Client-Seite

1. Marshalling der Methodenparameter
2. Unmarshalling der Rückgabewerte
3. Rückgabe von Werten an den Client



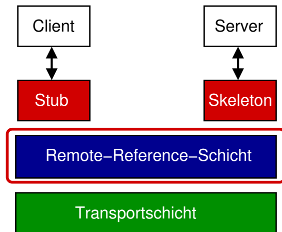
Skeleton auf Server-Seite

1. Unmarshalling der empfangenen Methodenparameter
2. Aufruf der Methode des Server-Objekts
3. Marshalling der Rückgabewerte oder von Exceptions

➜ Stub und Skeleton können mit RMI-Compiler *rmic* automatisch generiert werden

➜ ab Java 2 ist der Einsatz von Skeletons nicht mehr nötig

Remote-Reference-Schicht



Aufrufsemantik

- Ein Aufruf vom Client wird garantiert nur einmal beim Server ausgeführt

  ➜ *at-most-once*

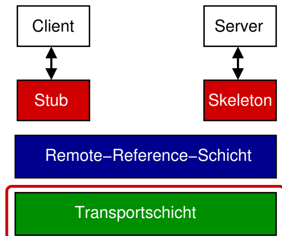
Java 1.1

- *nur Unicast* erlaubt
- Client kann nur bereits aktivierte Server-Objekte aufrufen

ab Java 2

- *Multicast*, d.h. Client-Anfrage an mehrere Server möglich
- mit Hilfe des Programms *rmid* (*RMI Activation System Daemon*) können nach einer Client-Anfrage die Server-Objekte noch aktiviert werden

Transportschicht



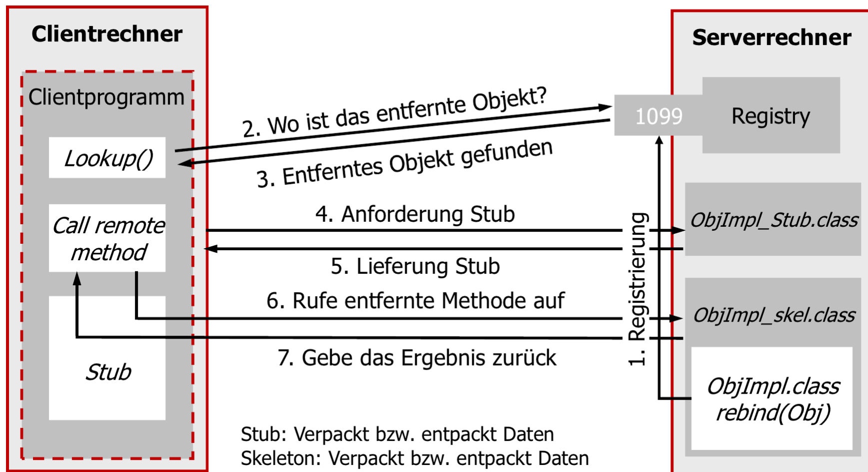
Funktion

- Verbindungsaufbau zwischen Client und Server
- Stream-basierte TCP/IP-Verbindungen

Protokolle

- RMI benutzt JRMP (*Java Remote Method Protocol*)
  ➜ *Java abhängiges Protokoll*
- RMI-IIOP (*RMI over Internet Inter-Orb Protocol*)
  ➜ *Java unabhängiger*, Anbindung an CORBA möglich (*Common Object Request Broker Architecture*)
- RMI-Transportschicht leicht auch auf andere Protokolle, wie z.B. UDP oder SSL anpassbar

Aufruf einer entfernten Methode



Java-Komponenten für RMI

Java.rmi Definition von Klassen, Interfaces und Exceptions, die von aufrufenden Methoden gesehen werden

Java.rmi.registry Definition von Klassen, Interfaces und Exceptions zur Benennung von entfernten Objekten

Java.rmi.server Definition von Klassen, Interfaces und Exceptions, die nur auf Server-Seite sichtbar sind

Java.rmi.dgc Verteilte Speicherbereinigung (*Distributed Garbage Collection*)

*Java.rmi.** Inhalt sind Funktionalitäten, wie z.B. das Interface *Remote* zur Definition eines entfernten Interfaces, *RMISecurityManager* zur Behandlung von Sicherheitsaspekten und *Naming* zur Umsetzung einer Registry

Schritte zur Erstellung eines RMI-Programms

1. *Interface Definition:* Festlegung der entfernten Schnittstelle
   ➜ Client nutzt die in der Schnittstelle aufgeführten Methoden zur Interaktion mit dem Server
2. *Interface Implementation:* Definition einer Server-Applikation, die die entfernte Schnittstelle implementiert
3. *Generierung einer Stub-Klasse* aus der Interface-Definition mit Hilfe des RMI-Compilers *rmic*
4. *Starten der Registry und des Servers*: Registry kann auch aus dem Server heraus gestartet werden
5. *Definition und Start des Clients*, der die entfernten Methoden des Servers verwendet

Schritt 1: Definition entferntes Interface

- Erweitern der Standardschnittstelle *java.rmi.Remote*
- Festlegung der durch einen Client zugereifbaren entfernten Methoden in diesem aus *Remote* abgeleitetem Interface
- Kennzeichnen mit *public* bei allen Methodensignaturen
- Deklaration der Ausnahme *RemoteException* in allen Methodensignaturen

  ➜ nötig, um Netwerkprobleme oder andere Verbindungsstörungen zwischen Client und Server sichtbar zu machen

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote {
    String sayHello() throws RemoteException;
}
```

Schritt 2: Implementation entferntes Interface

➜ Verwendung von Klassen aus *java.rmi.server*

UnicastRemoteObject Wird in der Regel genutzt, um entfernte Unicast-Objekte zu implementieren

ObjID Eindeutige numerische Identifikation des entfernten Objektes

RemoteObject Implementierung des entfernten Objektes

RemoteServer Framework zur Unterstützung von Referenzierung und Aufrufsemantik

RMIClassLoader Laden von Klassen übers Netzwerk inkl. Ermittlung der Position

RMISocketFactory Erstellung von Sockets zur Durchführung von entfernten RMI-Aufrufen

Schritt 2: Erstellung eines Server-Objektes

Aufbau

- Server-Klasse enthält die Implementierung aller entfernten Methoden und der *main*-Methode
- die *main*-Methode enthält den Server-Code
  - ➜ Erzeugung eines Objektes der Klasse *UnicastRemoteObject*
  - ➜ Bindung dieses Objektes an die *RMIRegistry* für externen Zugriff

Verwendung von UnicastRemoteObject

```
// Objekterzeugung, wird erst spaeter gebunden
protected UnicastRemoteObject()

// Objekterzeugung, wird automatisch gebunden
protected UnicastRemoteObject(int port)

// Remote-Object ueber statische Methode zurueckgeben
public static Remote exportObject(Remote obj)
```

Schritt 2: Registrieren entfernter Objekte

Anmeldung

- alle entfernten Objekte bei RMI-Registry anmelden
  - ➜ *host*: Position der Registry, bei keiner Angabe *localhost*
  - ➜ *port*: Angabe erforderlich, falls abweichend vom Port *1099*
  - ➜ *name*: Bezeichner für das Objekt

Funktionalität in java.rmi.Naming

```
// Registrierung unter einem Namen mit Object-Referenz
static void bind(String name,Remote obj) throws
  AlreadyBoundException,RemoteException, ...
// alte Object-Bindung ueberschreiben
static void rebind(String name,Remote obj) throws ...
// Object wieder aus Registry entfernen
static void unbind(String name) throws ...
// Dump aller Registry-Eintraege mit einer URL
static String[] list(String url) throws ...
```

Beispiel: Impl. entferntes Interface & Server

```java
import java.rmi.registry.*;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class Server implements Hello {
 public Server() {}
 public String sayHello() {return "Hello, world!";}

 public static void main(String args[]) {
  try {
   Server obj = new Server();
   Hello stub =
       (Hello) UnicastRemoteObject.exportObject(obj);
   // Objekt binden
   Registry registry = LocateRegistry.getRegistry();
   registry.bind("Hello", stub);
   // Server erfolgreich gestartet
   System.err.println("Server ready");
  } catch (Exception e) { ... }
 }
}
```

Beispiel: Implementierung Client

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {
 public static void main(String[] args) {

 String host = (args.length < 1) ? null : args[0];
  try {
   Registry registry = LocateRegistry.getRegistry(host);
   Hello stub = (Hello) registry.lookup("Hello");
  // entfernter Methodenaufruf
   String response = stub.sayHello();
  // Ausgabe zur Kontrolle
   System.out.println("Reply des Servers: " + response);
  } catch (Exception e) { ... }
 }
}
```

Schritte 3, 4 & 5: Kompilieren und Starten

Vorgehen

- *Generierung von Stubs & Skeletons* mit RMI-Compiler *rmi*

```
javac *.java
rmic Server
```

- *Starten der Registry* falls nicht per Default gestartet

```
rmiregistry &
```

- *Starten des Servers* mit Security-Policy

```
java -Djava.security.policy=java.policy Server 1099 &
```

- *Starten des Clients* mit Security-Policy

```
java -Djava.security.policy=java.policy Client host 1099
```

Sicherheit im Sinne von Security

Problem

- Download von Klassen bei Default-Einstellung nicht erlaubt
  ➜ Setzen entsprechender Rechte nötig

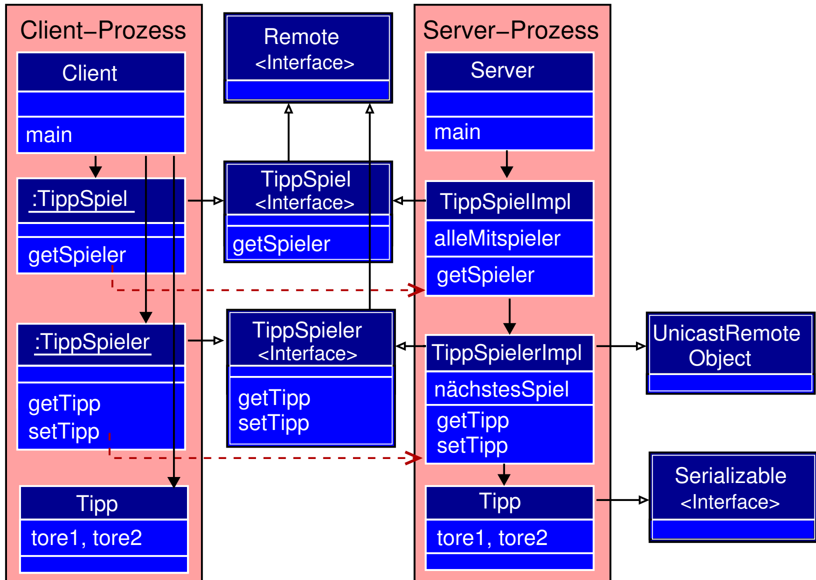
Umsetzung

- Verwendung des *RMISecuritymanager* mit diversen Methoden
- alternativ Anpassen von vordefinierter *Java-Policy*

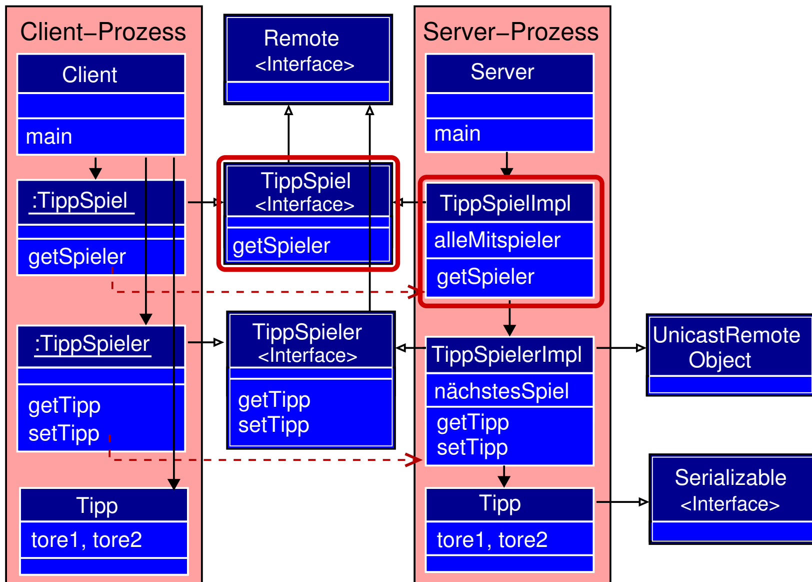
```
// umfangreiche Freigabe
grant codeBase "file:/home/src/" {
    permission java.security.AllPermission; };

// selektive Freigabe
grant {
    permission java.net.SocketPermission "hostname:1099",
      "connect","resolve"; };
```

Beispiel: EM-Fußball-Tippspiel



Tippspiel: Interface & Implementierung

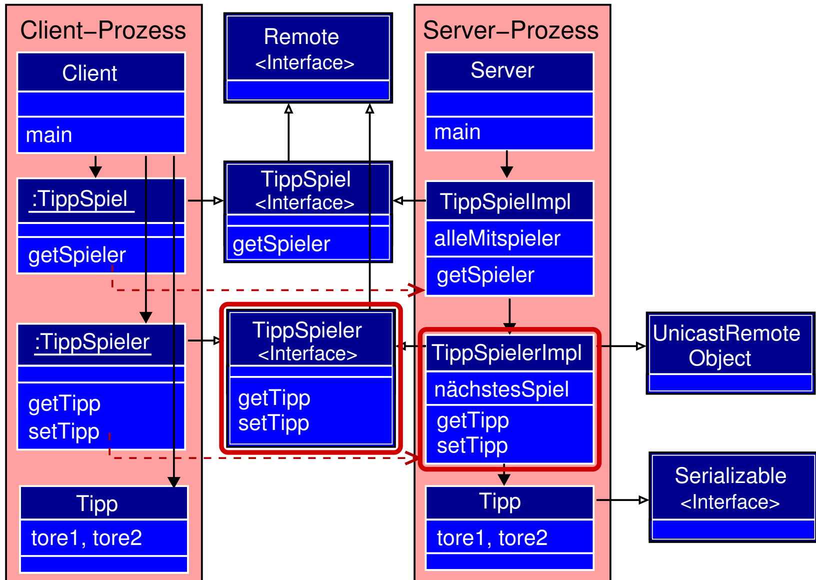


TippSpiel: Interface & Implementierung

```java
public interface TippSpiel extends Remote {
 public TippSpieler getSpieler(int spielerId)
                            throws RemoteException;}
```

```java
1  public class TippSpielImpl implements TippSpiel {
2   public List<TippSpielerImpl> alleMitspieler =
3                           new ArrayList<TippSpielerImpl>();
4   // Konstruktor zum Aufbau der Daten, z.B. aus XML-Datei
5   public TippSpielImpl() throws RemoteException {
6     Tipp t = new Tipp(0,0);
7     TippSpielerImpl s = new TippSpielerImpl(42,"Steffen",t);
8     alleMitspieler.add(s); ...
9   }
10  // entfernte Methode zur Rueckgabe eines TippSpielers
11  public TippSpieler getSpieler(int spielerId)
12                            throws RemoteException {
13    TippSpieler spieler = null;
14    for (TippSpielerImpl s: alleMitspieler) {
15      if (s.spielerId == spielerId) { spieler = s;}
16    }
17    return spieler; }}
```

Tippspieler: Interface & Implementierung

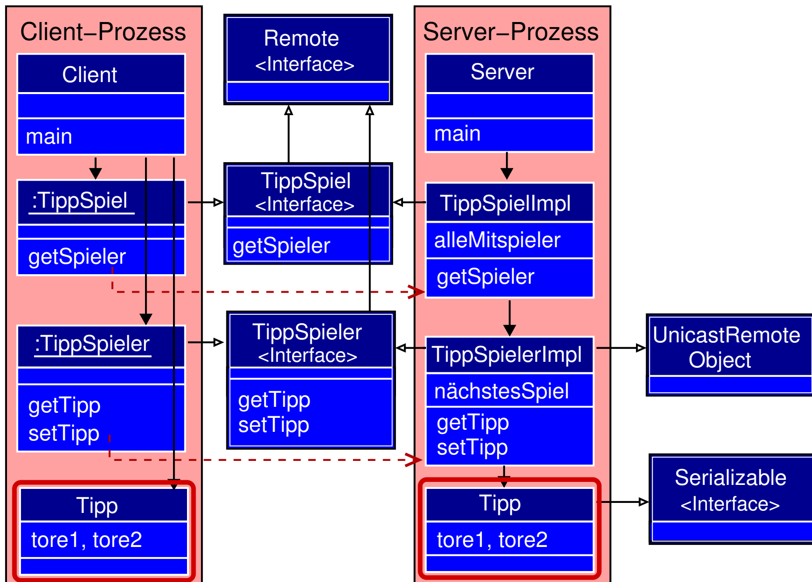


TippSpieler: Interface & Implementierung

```java
public interface TippSpieler extends Remote {
 Tipp getTipp() throws RemoteException;
 void setTipp(Tipp spielTipp) throws RemoteException; }
```

```java
1  public class TippSpielerImpl extends UnicastRemoteObject
2                                 implements TippSpieler {
3   private String spieler;
4   public int spielerId;
5   private Tipp naechstesSpiel;
6   // Konstruktoren
7   public TippSpielerImpl() throws RemoteException{};
8   public TippSpielerImpl(int id,String spieler,Tipp t)
9                                         throws RemoteException {
10    this.spielerId = id; this.spieler = spieler;
11    this.naechstesSpiel = t;
12  }
13  // entfernte Methoden: R\"uckgabe & Setzen eines Tipps
14  public Tipp getTipp() throws RemoteException {
15    return naechstesSpiel; }
16  public void setTipp(Tipp t) throws RemoteException {
17    this.naechstesSpiel = t; }
```

Tipp: Objekt zum Speichern eines Tipps



Tipp: Objekt zum Speichern eines Tipps

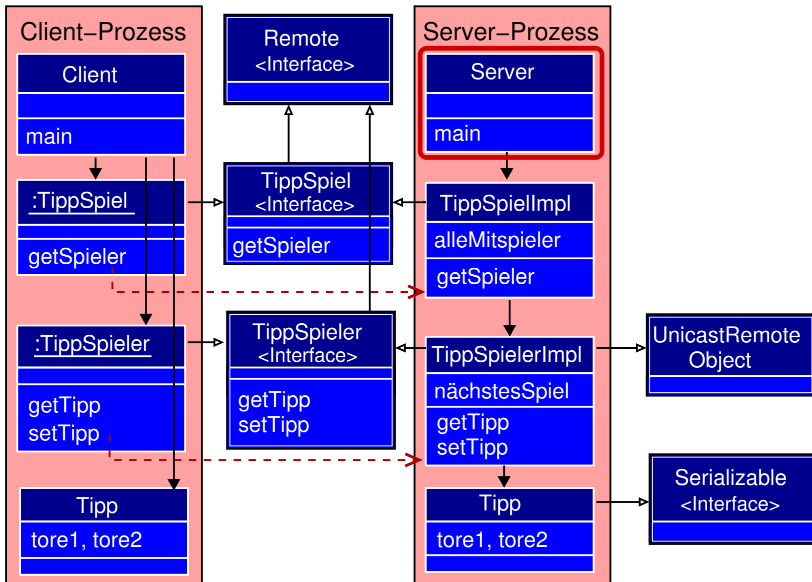
```java
import java.io.Serializable;

public class Tipp implements Serializable {
  private int tore1;
  private int tore2;

  public Tipp(int toreMannschaft1, int toreMannschaft2) {
      this.tore1 = toreMannschaft1;
      this.tore2 = toreMannschaft2;
  }

  public String toString() {
       return tore1 + ":" + tore2;
  }
}
```

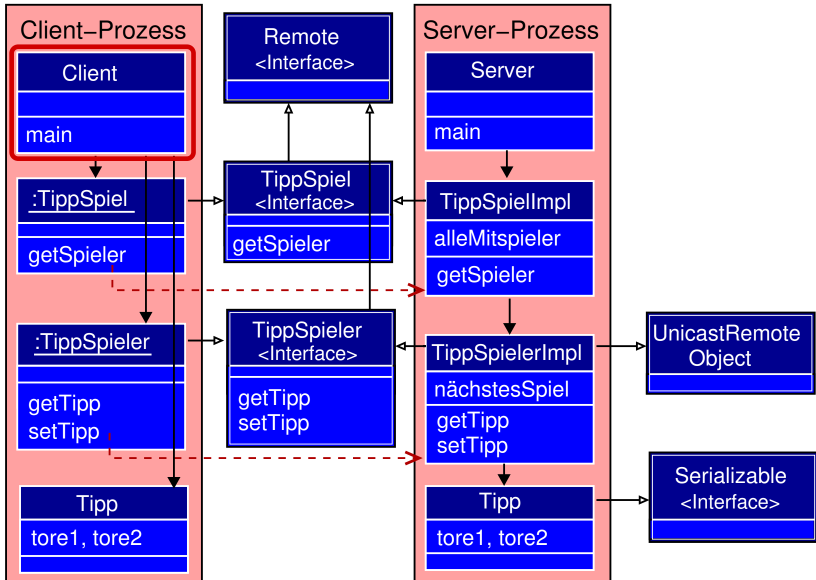
Server: Implementierung



Server: Implementierung

```
1  import java.rmi.*;
2  import java.rmi.server.*;
3  import java.rmi.registry.*;
4
5  public class Server {
6   public static void main(String args[]) {
7    try {
8     TippSpielImpl spielImpl = new TippSpielImpl();
9  // Erzeugen eines Stub-Objektes
10    TippSpiel tippSpiel = (TippSpiel)
11                UnicastRemoteObject.exportObject(spielImpl);
12 // Erzeugen einer RMI-Registry
13    Registry registry =
14                LocateRegistry.createRegistry(10009);
15 // Stub-Objekt binden
16    registry.bind("EMTippSpiel",tippSpiel);
17 // Server erfolgreich gestartet
18    System.out.println("Server bereit");
19   } catch (Exception e) { ... }
20 }
```

Client: Implementierung



Client: Implementierung

```java
import java.rmi.registry.*;

public class Client {
 public static void main(String args[]) {
  String host = "bolero.cs.tu-berlin.de";
  try {
 // RMI-Registry bestimmen
   Registry registry =
               LocateRegistry.getRegistry(host,10009);
 // Stub-Objekte suchen
   TippSpiel tippSpiel = (TippSpiel)
                  registry.lookup("EMTippSpiel");
 // R\"uckgabe einer entfernten Objektreferenz
   TippSpieler spieler = tippSpiel.getTippSpieler(42);
 // Setzen eines neuen Tipp
 // Voraussetzung: Tipp-Objekte sind serialisierbar
   spieler.setTipp(new Tipp(1,4));
   Tipp spielTipp = spieler.getTipp();
   System.out.println("Neuer Spieltipp ist: " + spielTipp);
  } catch (Exception e) { ... }
}
```

Aufteilung auf Server & Client

Server-Klassen

1. *TippSpiel.class*
2. *TippSpielImpl.class*
3. *TippSpielImpl_Stub.class*
4. *TippSpieler.class*
5. *TippSpielerImpl.class*
6. *TippSpielerImpl_Stub.class*
7. *Tipp.class*
8. *Server.class*

Client-Klassen

1. *TippSpiel.class*
2. *TippSpielImpl_Stub.class*
3. *TippSpieler.class*
4. *TippSpielerImpl_Stub.class*
5. *Tipp.class*
6. *Client.class*

➜ *RMIClassLoader* erlaubt auch entfernte Klassen nachzuladen
z.B. könnte der Client *TippSpielImpl_Stub.class* und *TippSpielerImpl_Stub.class* vom Server laden

Parameter von Remote-Methoden

Möglichkeiten der Parameterübergabe

1. als Kopie des Objekts *deep-copy*
2. als Objektreferenz[1]

Parameterübergaben bei Remote-Methoden

- Werte von einfache Datentypen (z.B. *int*)
  ➜ werden als *Kopie des Werts* übergeben
- Objekte, die das Interface *Serializable* implementieren (z.B. *Tipp*)
  ➜ werden als *Kopie des Objekts* übergeben
- Objekte, die das Interface *Remote* implementieren (z.B. *TippSpieler*)
  ➜ werden als *Objektreferenz* übergeben

[1] Im RMI-Kontext findet man auch die Unterscheidung *call-by-reference* und *call-by-value*. Diese Begrifflichkeit ist aber irreführend, da es sich hier nicht um die klassische Semantik von *call-by-reference*, wie z.B. in der Programmiersprache C++ handelt.