RMI, Zusammenfassung und Lehrevaluation

PROG 2: Einführung in die Programmierung für Wirtschaftsinformatiker

---

Steffen Helke
Technische Universität Berlin
Fachgebiet Softwaretechnik

8. Juli 2013

Übersicht

- Rückblick: RMI
- Zusammenfassung von PROG 2 & Klausurthemen
- Lehrevaluierung

Teil V der Vorlesung PROG 2

Netzwerkprogrammierung

RMI – Remote Method Invocation

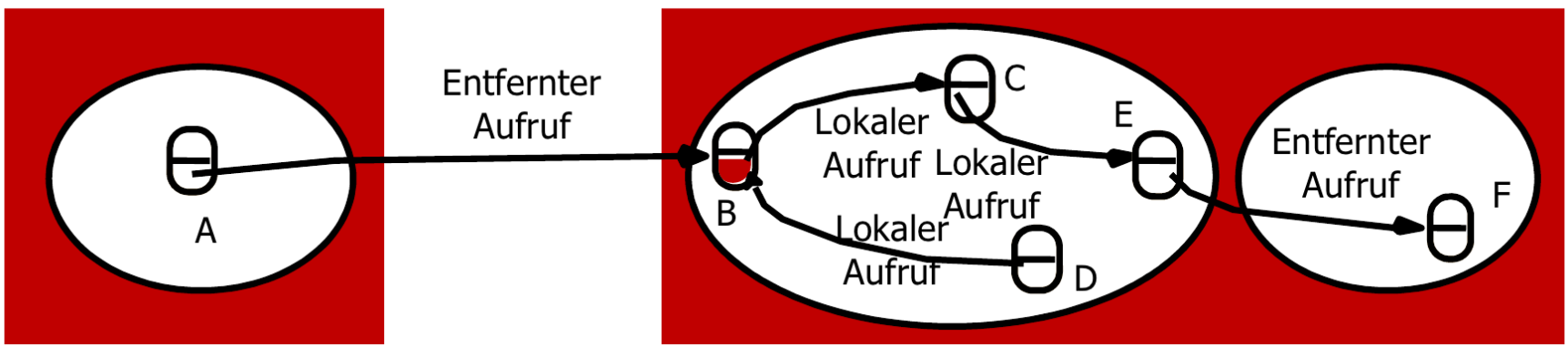
Rückblick mit neuem Beispiel

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

Verteiltes Objektmodell

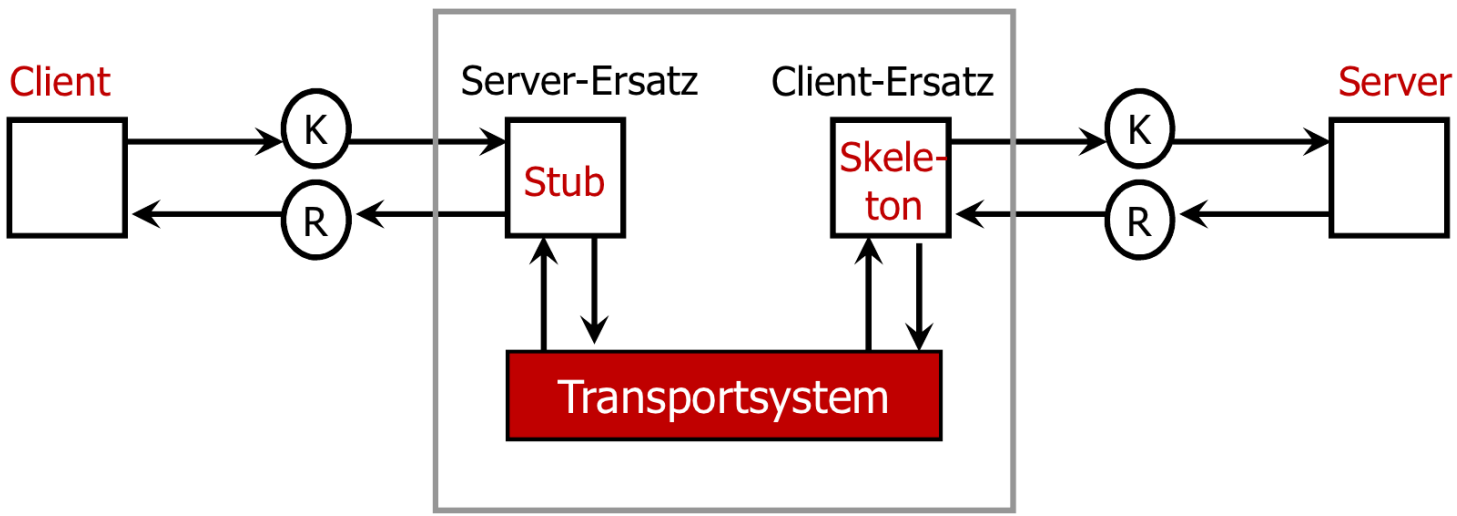
Grundidee von RMI

- 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



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



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

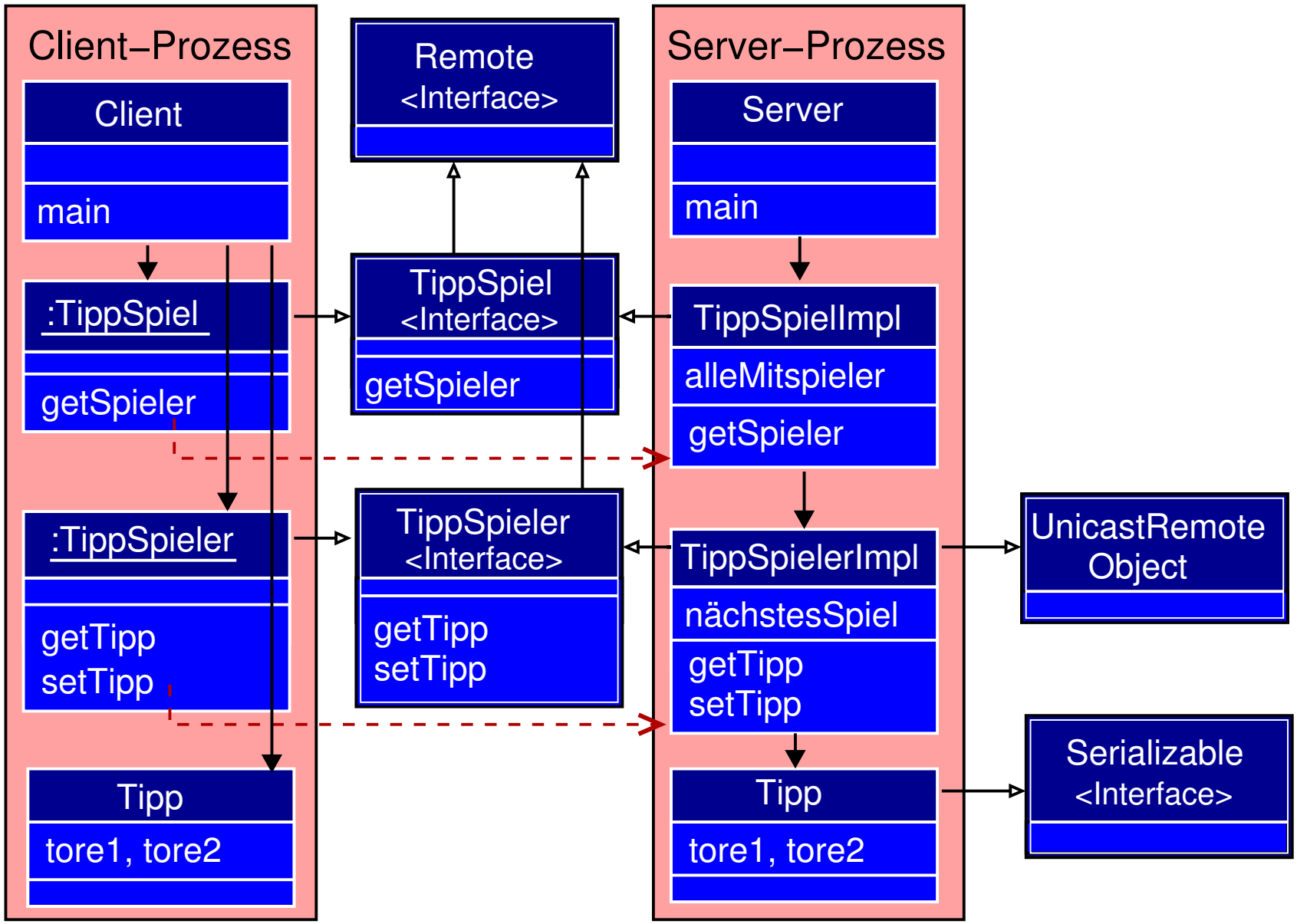
2. *Interface Implementation:* Definition einer Server-Applikation, die die entfernte Schnittstelle implementiert

3. *Generierung einer Stub-Klasse* aus der Interface Implementation mit Hilfe des RMI-Compilers *rmic*

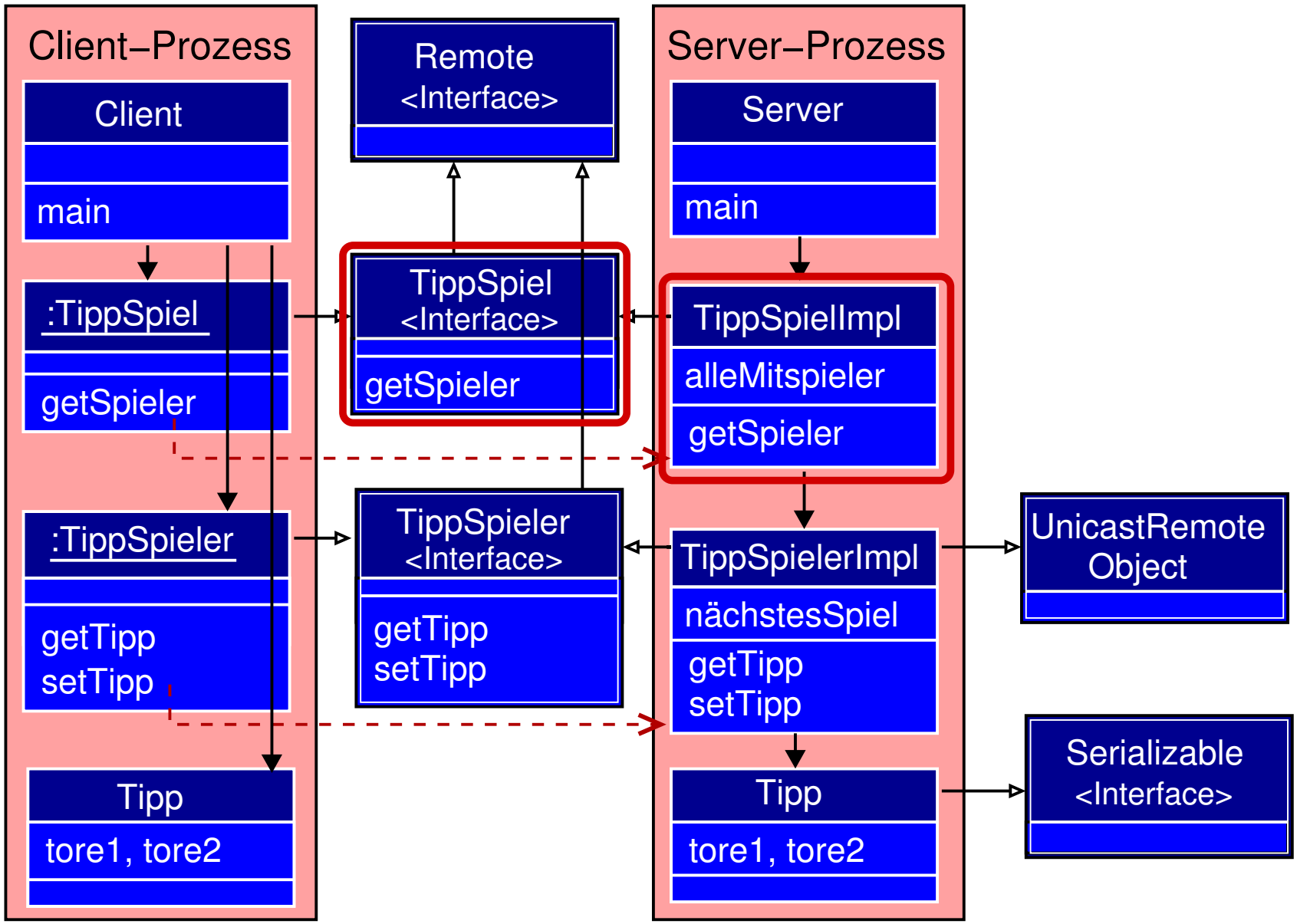
4. *Starten der Registry und des Servers*: Registry kann auch aus dem Server heraus gestartet werden

5. *Definition und Start des Clients*, der die entfernten Methoden des Servers verwendet

Beispiel: EM-Fußball-Tippspiel



Tippspiel: Interface & Implementierung

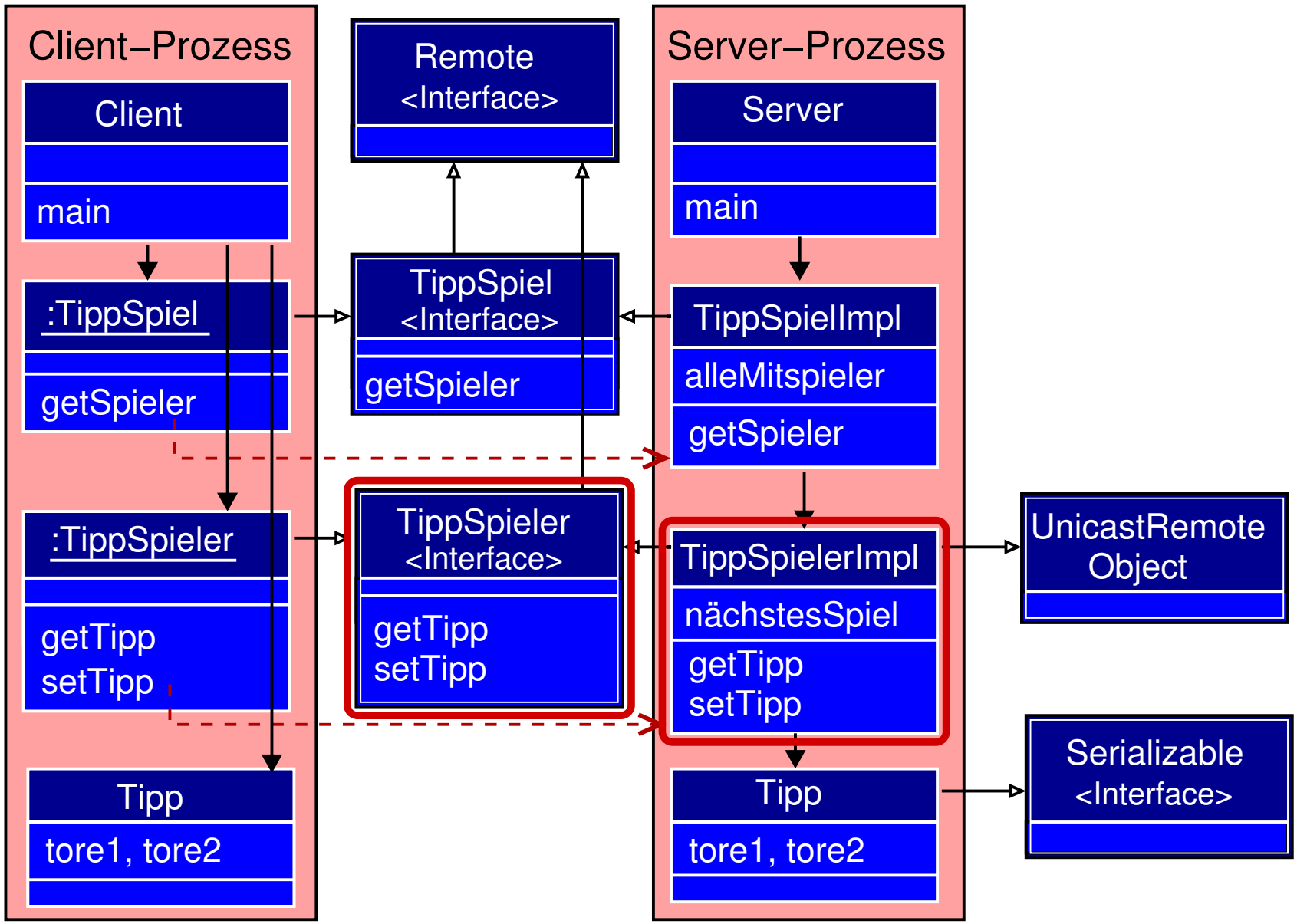


TippSpiel: Interface & Implementierung

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

```
1  public class TippSpielImpl implements TippSpiel {
2   public List<TippSpielerImpl> alleMitspieler =
3                          new ArrayList<TippSpielerImpl>();
4
5   // Konstruktor zum Aufbau der Daten, z.B. aus XML-Datei
6   public TippSpielImpl() throws RemoteException {
7    Tipp t = new Tipp(0,0);
8    alleMitspieler.add(new TippSpielerImpl(42,"Steffen",t));
9   }
10  // entfernte Methode zur Rückgabe 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

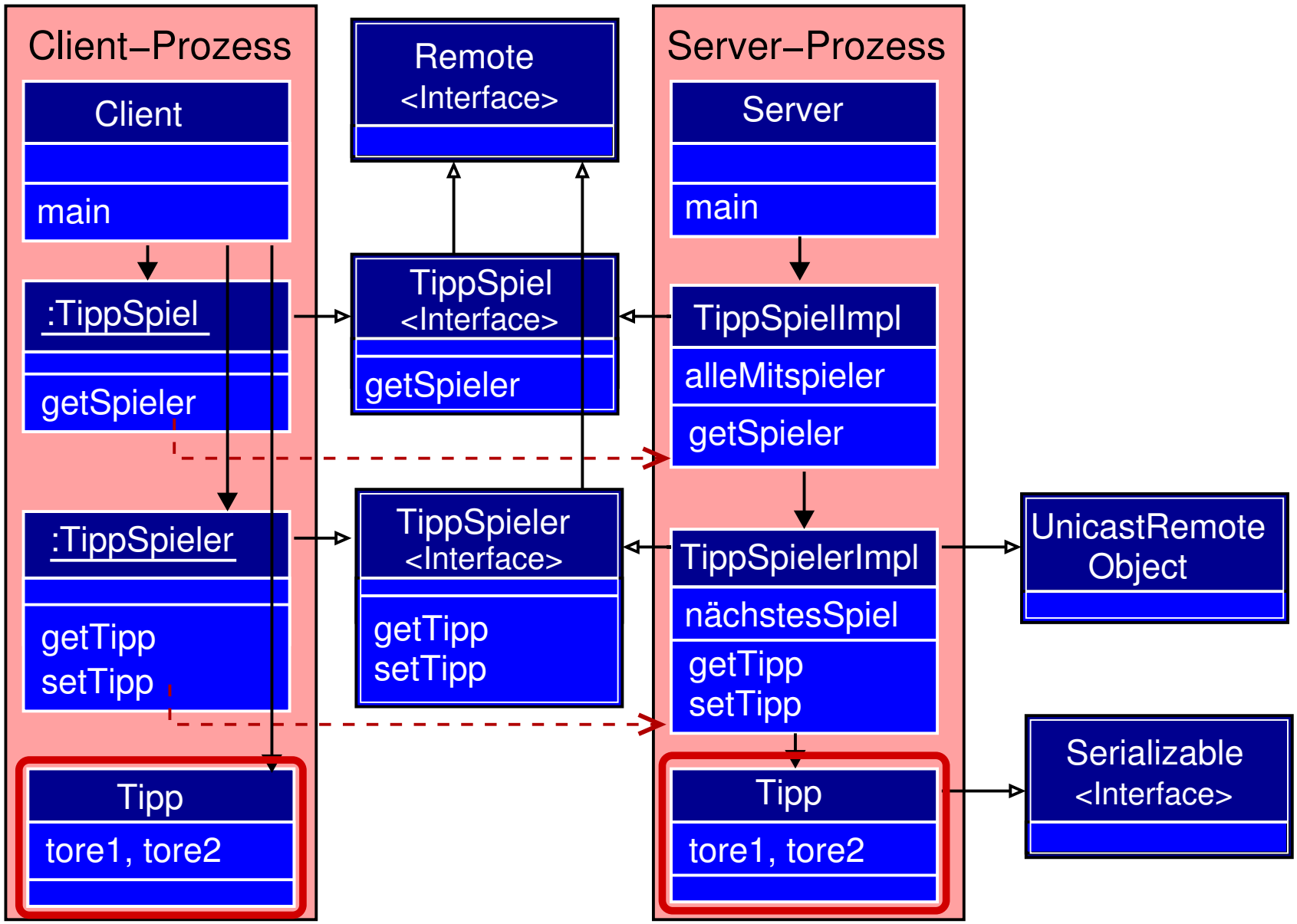


TippSpieler: Interface & Implementierung

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

```
1  public class TippSpielerImpl extends UnicastRemoteObject
2                                implements TippSpieler {
3   public int spielerId;
4   private String spieler; private Tipp naechstesSpiel;
5
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: Rückgabe & 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

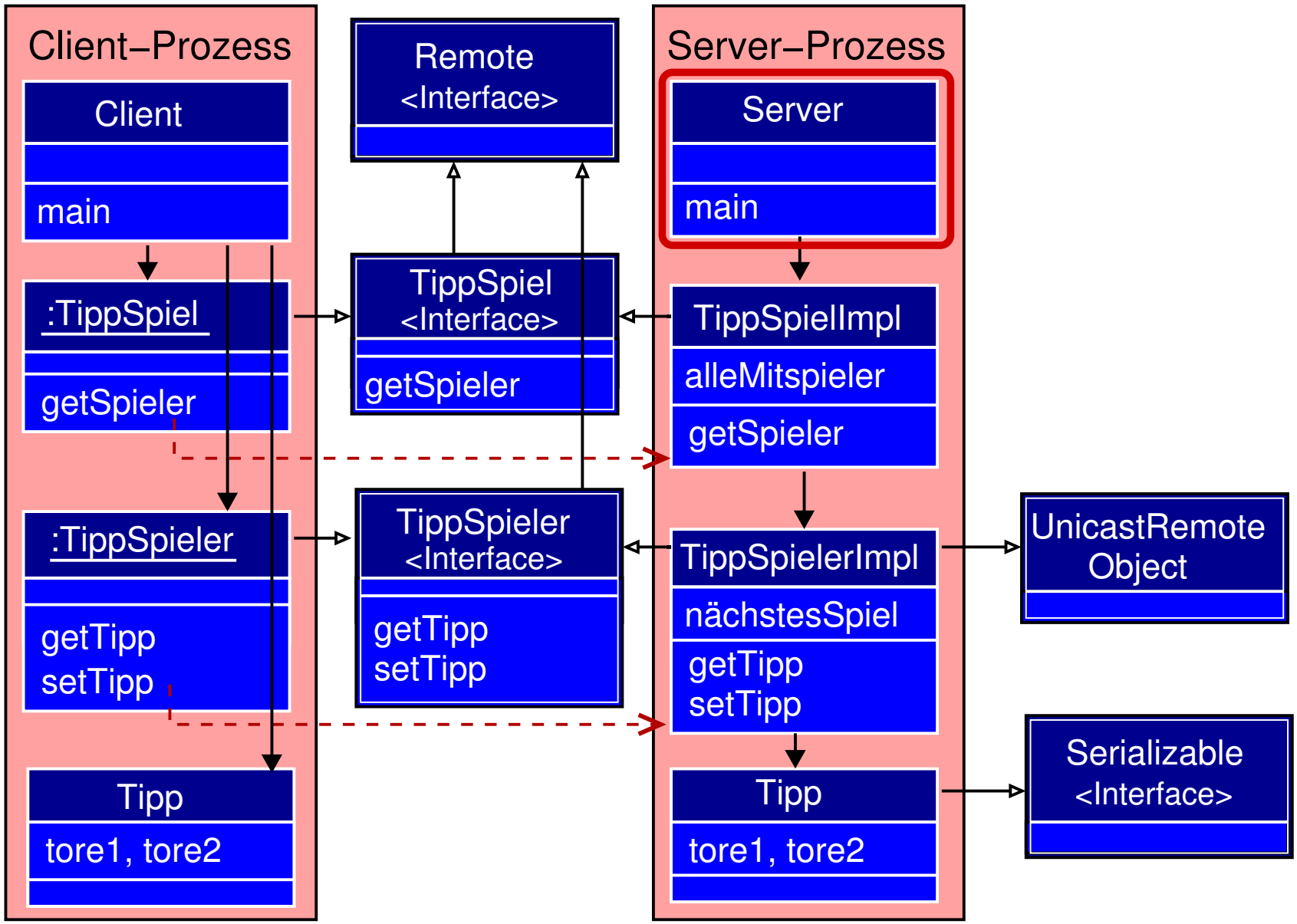
```
import java.io.Serializable;

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;
  }
}
```

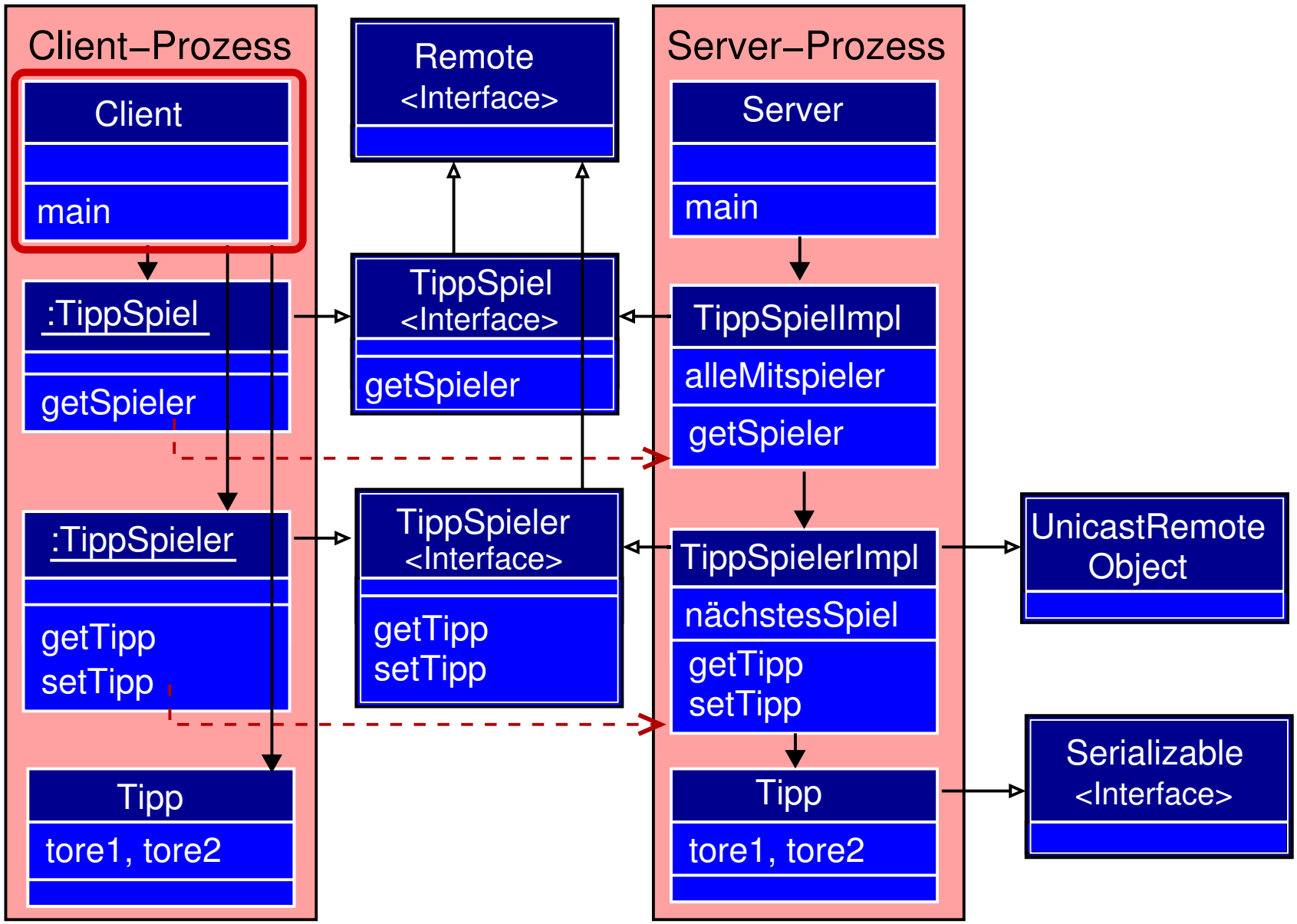
Server: Implementierung



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
10    // Erzeugen eines Stub-Objektes
11    TippSpiel tippSpiel = (TippSpiel)
12              UnicastRemoteObject.exportObject(spielImpl);
13
14    // Erzeugen einer RMI-Registry
15    Registry registry =
16              LocateRegistry.createRegistry(10009);
17
18    // Stub-Objekt binden
19    registry.bind("EMTippSpiel",tippSpiel);
20
21   } catch (Exception e) { ... }
22 }
```

Client: Implementierung



Client: Implementierung

```
1  import java.rmi.registry.*;
2  public class Client {
3   public static void main(String args[]) {
4    String host = "bolero.cs.tu-berlin.de";
5    try {
6     // RMI-Registry bestimmen
7     Registry registry =
8                 LocateRegistry.getRegistry(host,10009);
9     // Stub-Objekte suchen
10    TippSpiel tippSpiel = (TippSpiel)
11                registry.lookup("EMTippSpiel");
12
13    // Rückgabe einer entfernten Objektreferenz
14    TippSpieler spieler = tippSpiel.getTippSpieler(42);
15
16    // Setzen eines neuen Tipp
17    // Voraussetzung: Tipp-Objekte sind serialisierbar
18    spieler.setTipp(new Tipp(1,4));
19
20
21   } catch (Exception e) { ... }
22 }
```

Client: Implementierung

```
1  import java.rmi.registry.*;
2  public class Client {
3   public static void main(String args[]) {
4    String host = "bolero.cs.tu-berlin.de";
5    try {
6     // RMI-Registry bestimmen
7     Registry registry =
8                 LocateRegistry.getRegistry(host,10009);
9     // Stub-Objekte suchen
10    TippSpiel tippSpiel = (TippSpiel)
11                registry.lookup("EMTippSpiel");
12
13    // Rückgabe einer entfernten Objektreferenz
14    TippSpieler spieler = tippSpiel.getTippSpieler(42);
15
16    // Setzen eines neuen Tipp
17    // Voraussetzung: Tipp-Objekte sind serialisierbar
18    spieler.setTipp(new Tipp(1,4));
19    Tipp spielTipp = spieler.getTipp();
20    System.out.println("Neuer Spieltipp ist: " + spielTipp);
21   } catch (Exception e) { ... }
22 }
```

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 einfachen 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.

Probleme im Bereich IT-Sicherheit (Security) beim Einsatz von RMI

Sicherheitsrisiken von RMI-Applikationen

Angriffspunkte

- verteilte Objekte sind grundsätzlich leicht angreifbar
- Hauptrisiko: *dynamisches Nachladen von Klassen* zur Laufzeit

Schadensmöglichkeiten

- heruntergeladener Programmcode kann beliebige für den Benutzer zugelassene Aktionen ausführen
- problematische Aktionen können sein

  ➜ Daten ausspionieren

  ➜ Daten verändern

  ➜ Daten schädigen

Dynamisches Herunterladen von Klassen

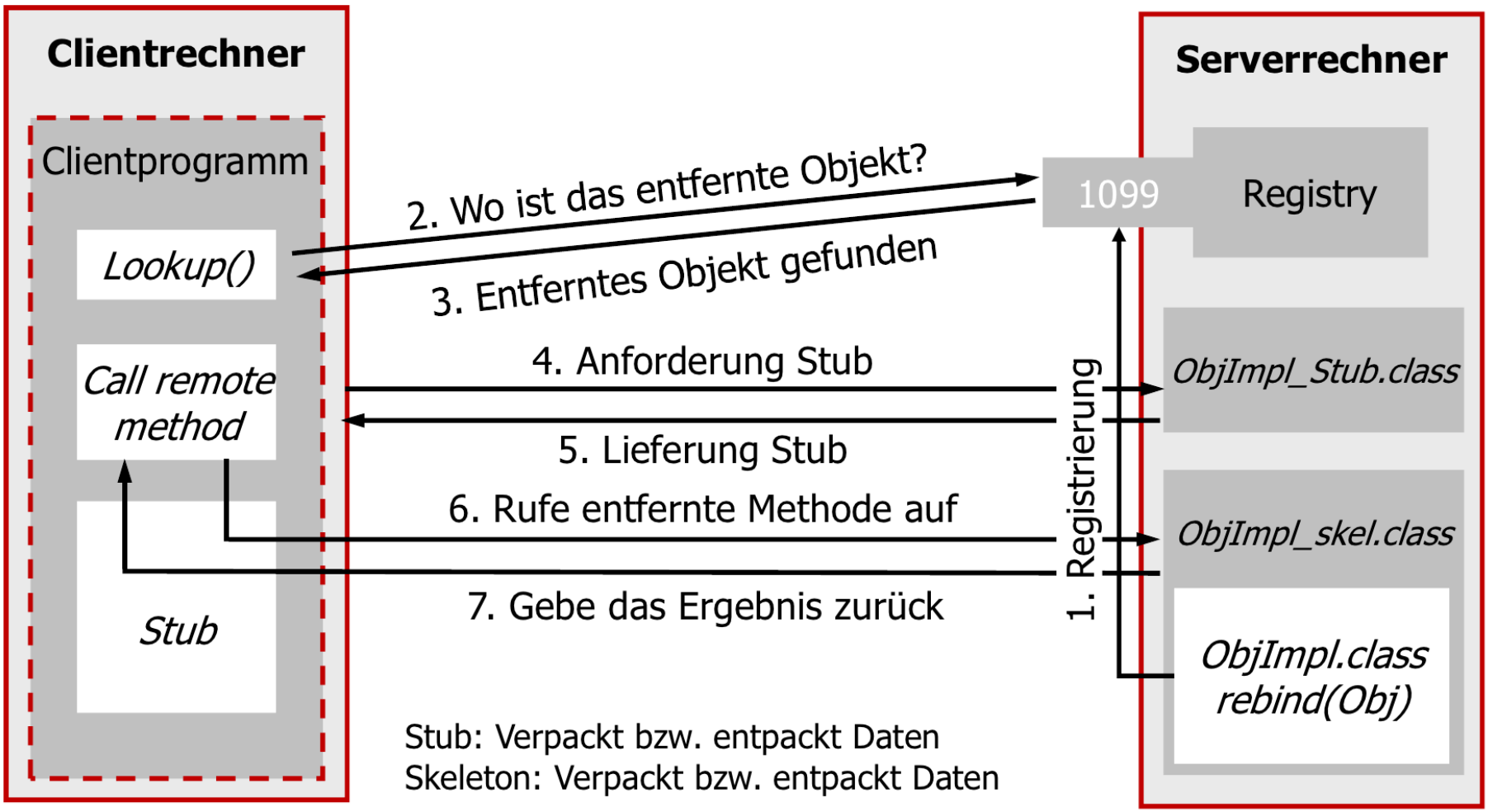
Welche Klassen können nachgeladen werden?

1. RMI-Client kann Stub-Klassen laden
2. RMI-Client kann Klassen der Objekte laden, die er serialisiert vom RMI-Server übertragen bekommt
3. RMI-Server kann Klassen der Objekte laden, die er serialisiert vom RMI-Client übertragen bekommt

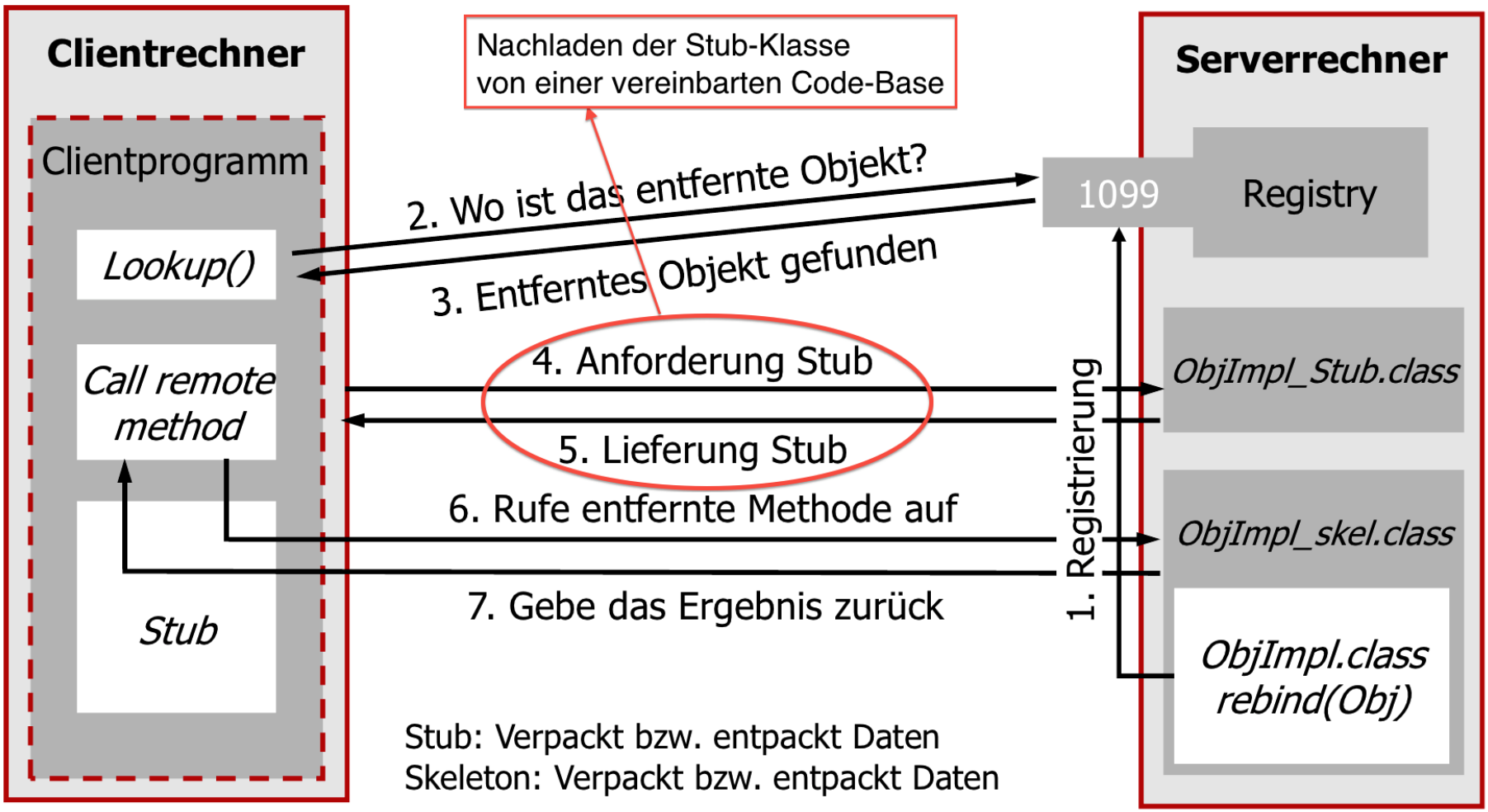
Konsequenzen

- ➜ RMI-Server und RMI-Client können durch geladene Klassen angegriffen werden
- ➜ ein RMI-Client kann über einen RMI-Server sogar andere RMI-Clients angreifen

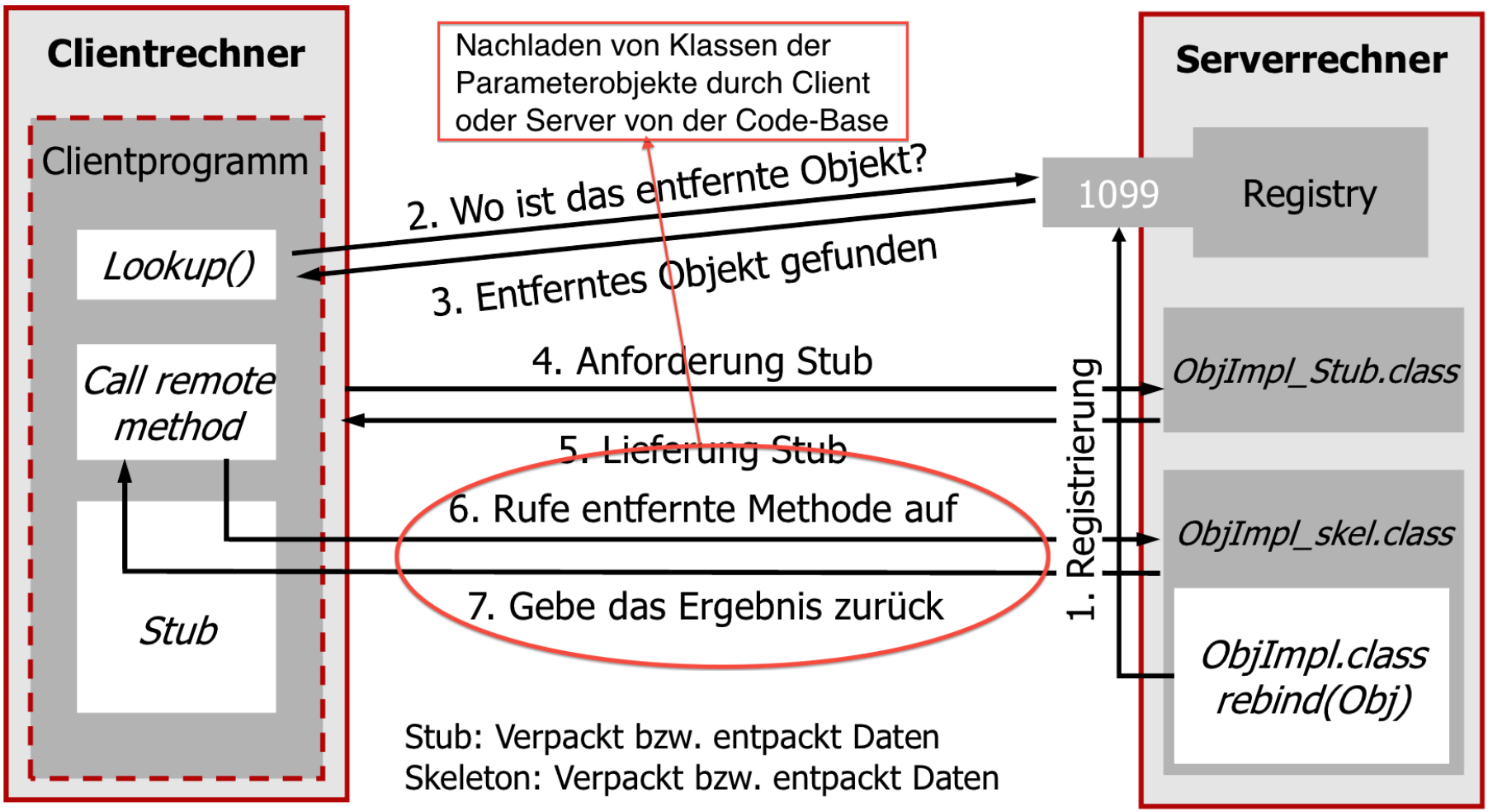
Dynamisches Herunterladen von Klassen



Dynamisches Herunterladen von Klassen



Dynamisches Herunterladen von Klassen



Nachladen von Code für den Client

Vorgehen

1. Installation eines RMISecurityManagers beim Client
   ➜ ohne kann kein Code geladen werden
2. Bereitstellen der zu ladenen Klassen auf einem Webserver
   ➜ z.B. *www.user.tu-berlin.de/helke/prog2/classes*
3. Socketverbindung zum Server und zum Webserver für den Client in einer Policy-Datei freigeben

Client-Policy-Datei

```
grant {
  permission java.net.SocketPermission
             "bolero.cs.tu-berlin.de:1024-65535",
             "connect";
  permission java.net.SocketPermission
             "www.user.tu-berlin.de:80",
             "connect";
};
```

Nachladen von Code für den Client

Definition eines RMISecurityManagers

```
import java.rmi.RMISecurityManager;
public class Client {
 public static void main(String[] args) {
  String host = "bolero.cs.tu-berlin.de";

  if (System.getSecurityManager()==null)
    System.setSecurityManager(new RMISecurityManager());

  try {
    Registry registry =
         LocateRegistry.getRegistry(host,1099);
    ...
  } catch (Exception e) { ... }
}
```

Aufruf des Client-Prozesses

```
java -Djava.rmi.server.codebase=http://www.user.tu-berlin.de/helke/prog2/classes/
     -Djava.security.policy=my.policy Client 1099
```

Zusammenfassung

Lehrinhalte von PROG 2
Klausurthemen & Organisatorisches

Themenblöcke in Prog 2

I. GUI-Programmierung, Ereignisbehandlung (Listener-Pattern), Exception-Handling, Model-View-Control

II. Ein- und Ausgabekonzepte, Streams, Serialisierung

III. XML-Parser, SAX, DOM

IV. Multithreading, Producer-Consumer-Problem, Semaphoren, bedingte und unbedingte Synchronisation

V. Netzwerkprogrammierung, Sockets, TCP/IP, UDP, HTTP, SMTP, RMI, Security-Probleme

Themen in der Klausur

Was kommt dran?

- ➜ Fragen können aus allen fünf Themenblöcken kommen
- ➜ es gibt insgesamt fünf Aufgaben
- ➜ auch ein paar Fragen zu Konzepten, also erlerntes Wissen aus der Vorlesung
- ➜ überwiegend Programmieraufgaben, also erlerntes Wissen aus den Tutorien und den Hausaufgaben

Wie kann ich mich vorbereiten?

- ➜ Proberechnen von alten Klausuraufgaben
- ➜ PROG 2-Klausur orientiert sich an älteren MPGI 4-Klausuren
- ➜ alte MPGI 4-Klausuren z.B. bei der Freitagsrunde abrufbar

Zusammenfassung

Lehrevaluierung

➜ Da die Ergebnisse nur fakultätsöffentlich sind, haben wir diese hier wieder entfernt. Sie können aber auf den WEB-Seiten der TU[2] eingesehen werden.

[2] http://www.eecs.tu-berlin.de/menue/studium_und_lehre/qualitaetsmassnahmen/lv-befragung/