Verteilte Systeme

Vorlesung 5
Sebastian Iwanowski
FH Wedel

Vorsorgemaßnahmen gegen diese Risiken

Korrektur zum letzten Mal !

gegen Verklemmung (deadlock):

- Zyklenerkennung und -behebung in Wartegraphen
- Timeout nach hinreichend langer Wartezeit

gegen Verhungern (livelock):

- Timeout nach hinreichend langer Wartezeit
- Regelwerk (durch zentrale Kontrollmaßnahmen)

gegen Ungleichbehandlung (unfairness):

- Zufallsgenerator bei Verteilung von Aufgaben oder Ressourcen

gegen Wettlaufeffekte (race conditions):

- Sperren von bestimmten Prozessen bzw. Voraussetzen von bestimmten Bedingungen
- Monitore / Semaphore zur Prozess- und Ressourcenkontrolle

Details in Vorlesung 5 (2004) (Dining Philosophers etc.)

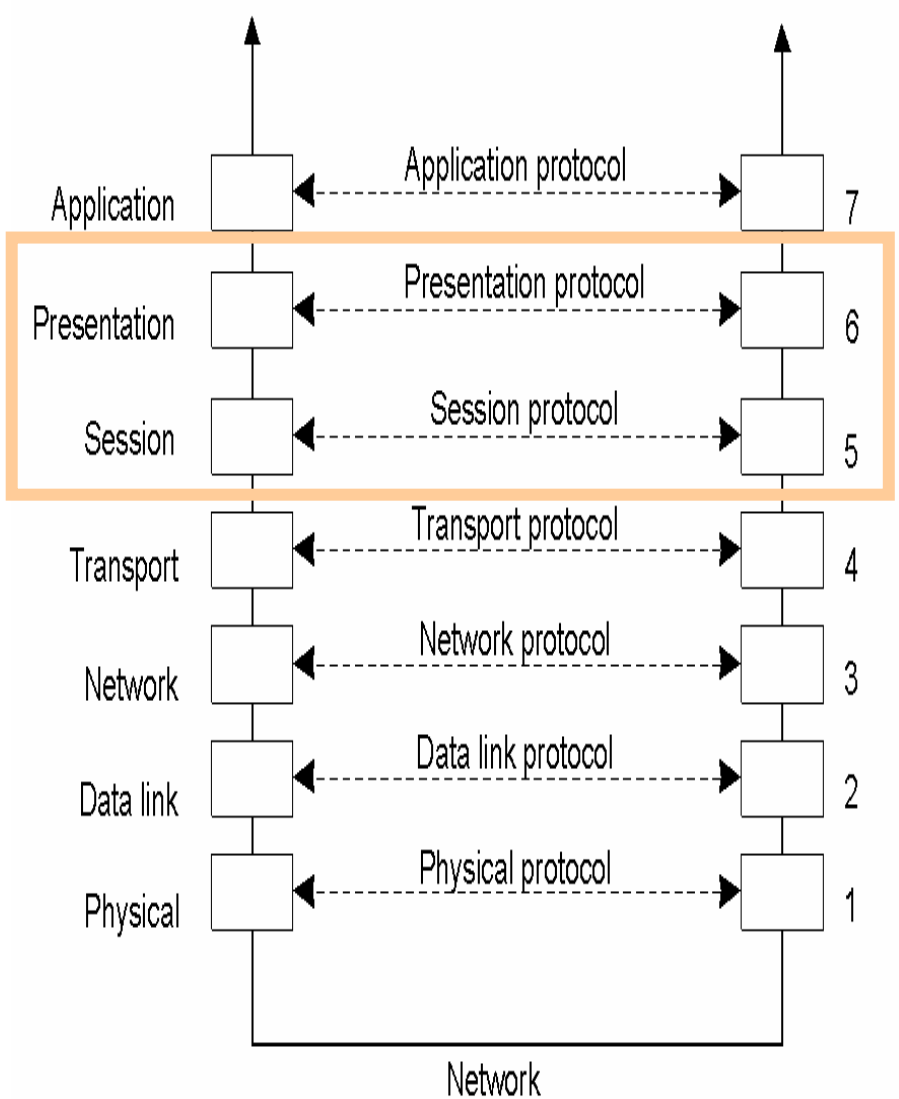
Verteilte Systeme

1. Innovative Beispiele aus der Praxis
2. Allgemeine Anforderungen und Techniken verteilter Systeme
3. Die Client-Server-Beziehung und daraus entstehende Fragestellungen
4. Nebenläufigkeitstechniken in Java

➔ 5. Entfernte Aufrufe
   - 5.1 Remote Procedure Call
   - 5.2 Remote Method Invocation
   - 5.3 Java-RMI
   - 5.4 Praxisbeispiel

6. Objektmigration
7. Agententechnologie
8. Dienstevermittlung
9. Synchronisation von Daten
10. Konzepte zur Erzielung von Fehlertoleranz
11. Web Services

Middleware-Schichten bei den Kommunikationsprotokollen



Applications

RMI, RPC and events

Request reply protocol

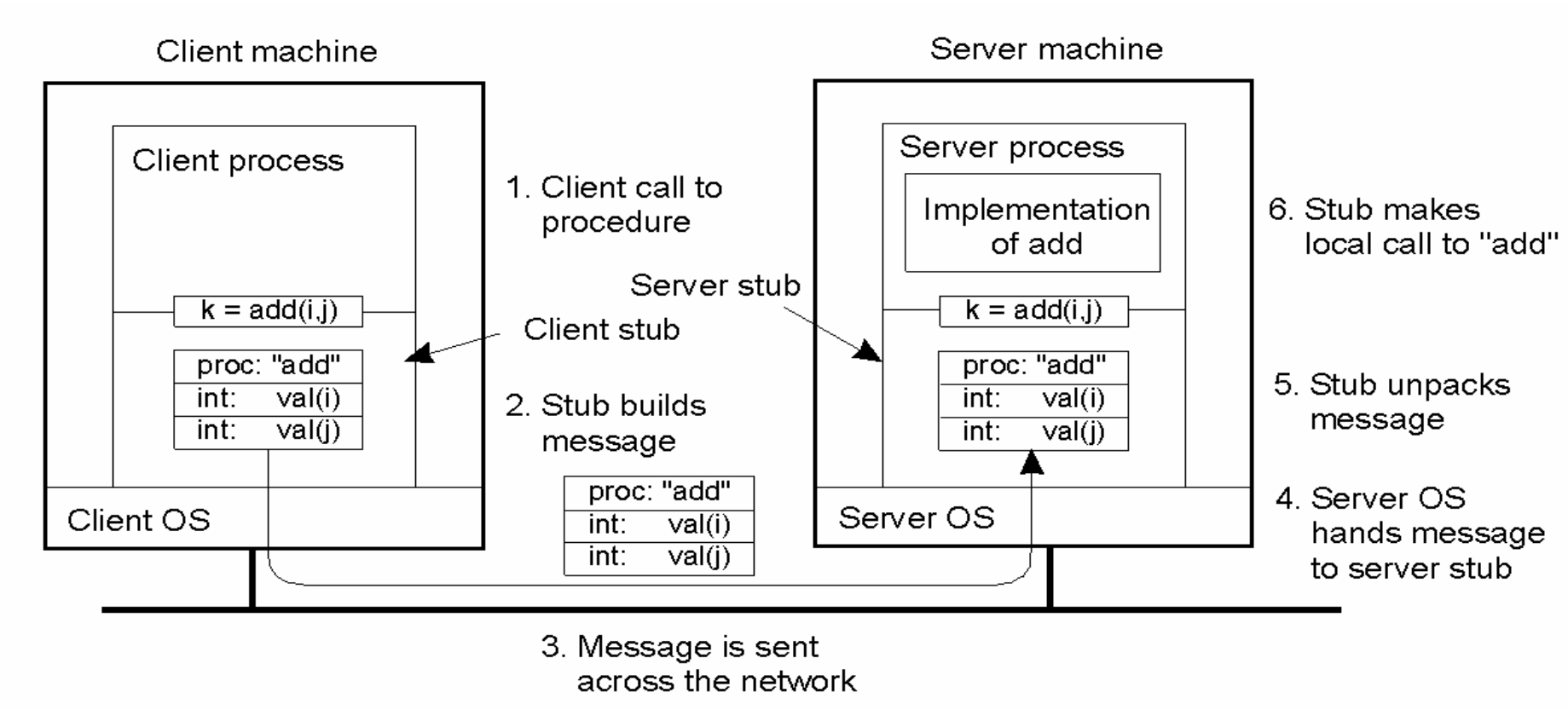
External data representation

Operating System

Middleware layers

5.1 Remote Procedure Call (RPC)

Architektur:



5.1 Remote Procedure Call (RPC)

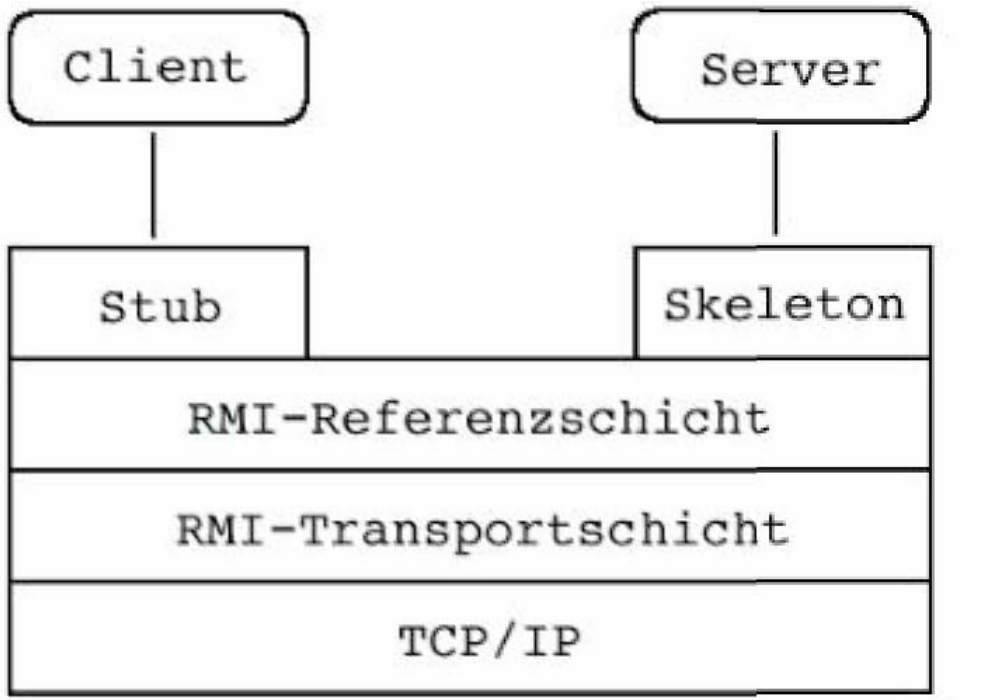
Funktionsabfolge:

1. Aufrufende Prozedur im Client ruft den Stub wie eine normale Prozedur auf.
2. Der Client-Stub erzeugt eine Nachricht und ruft sein lokales Betriebssystem auf.
3. Das lokale Betriebssystem sendet die Nachricht an das entfernte Betriebssystem.
4. Das entfernte Betriebssystem gibt die Nachricht an den Server-Stub weiter.
5. Der Server-Stub extrahiert die Parameter und ruft die entsprechende Server-Prozedur auf.
6. Die Server-Prozedur erledigt die Arbeit und sendet das Ergebnis dem Server-Stub.
7. Der Server-Stub erzeugt eine Nachricht und ruft sein lokales Betriebssystem auf.
8. Das Betriebssystem des Servers sendet die Nachricht an das Betriebssystem des Clients.
9. Das Betriebssystem des Clients gibt die Nachricht an den Client-Stub weiter.
10. Der Client-Stub packt das Ergebnis aus und teilt es der aufrufenden Prozedur mit.

Mehr zum Thema in Tanenbaum, Kap. 2.2

5.2 Remote Method Invocation (RMI)

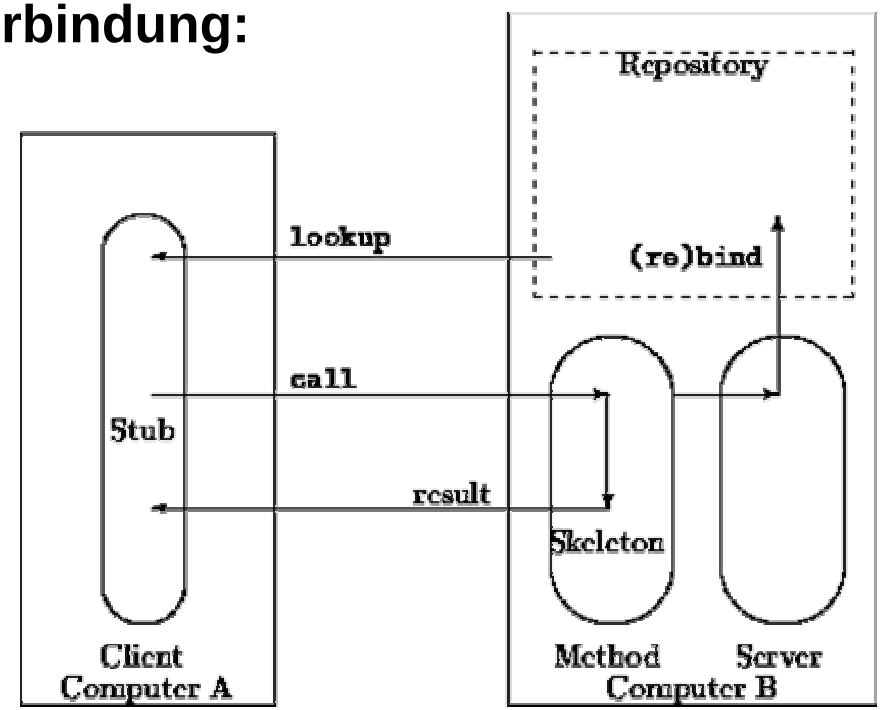
Prinzip:



- **Server veröffentlicht Schnittstelle**
- **Client muss Proxyklasse für Schnittstelle einrichten**
- **Namensverwaltung notwendig**

5.2 Remote Method Invocation (RMI)

Stub-Skeleton-Verbindung:



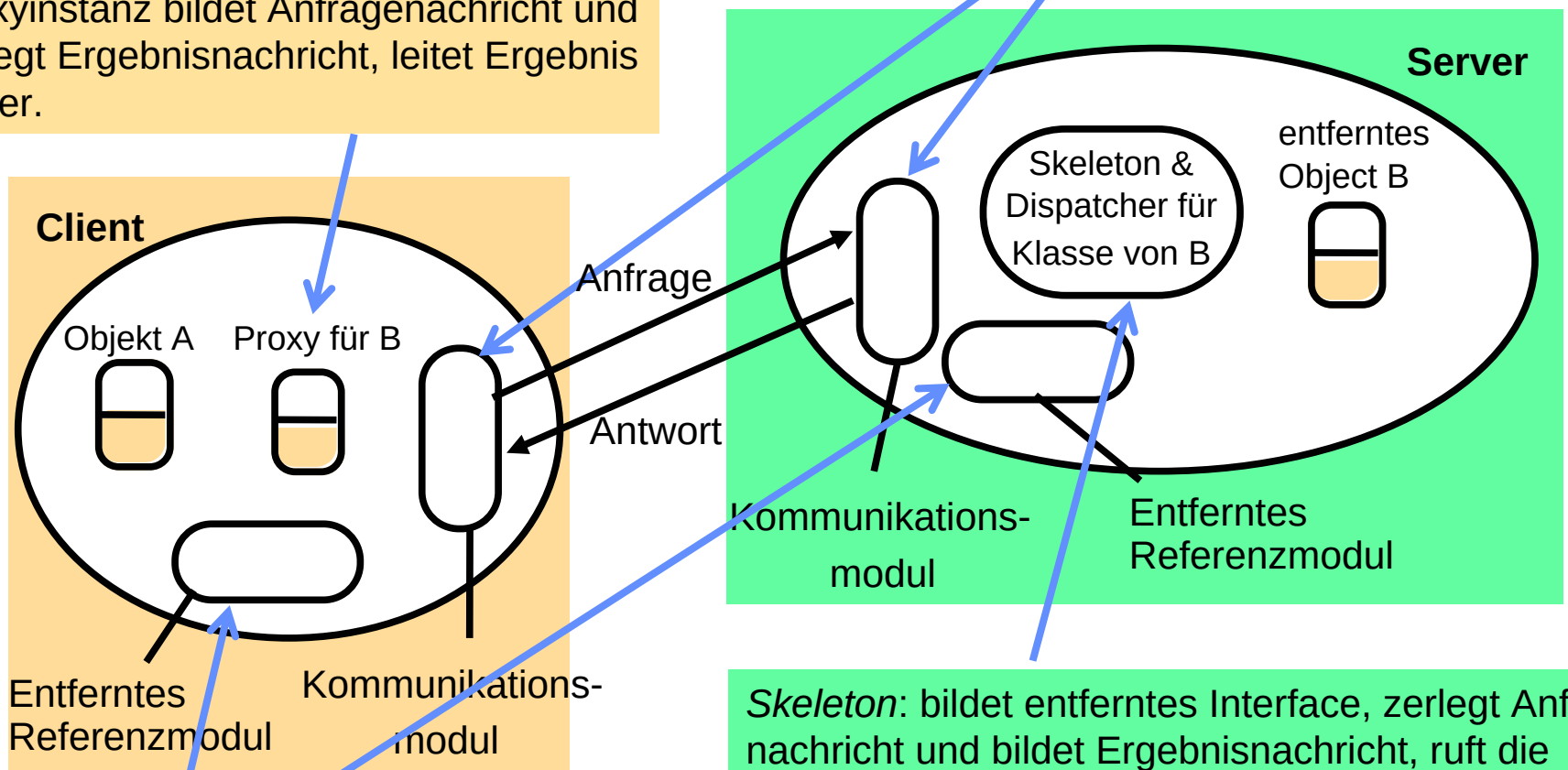
- **Server registriert seine Methoden im Repository**
- **Client sieht dort nach und erzeugt seine eigenen Interfaces im Stub.**
- **Erst danach ist eine Auftragsbearbeitung über Stub und Skeleton möglich.**

5.2 Remote Method Invocation (RMI): Funktionsablauf

Proxyklasse implementiert das entfernte Interface.

Proxyinstanz bildet Anfragenachricht und zerlegt Ergebnisnachricht, leitet Ergebnis weiter.

führt Kommunikations-protokoll aus



Übersetzt zwischen lokalem und entfernten Objekt. Erzeugt entfernte Objektreferenzen.

Skeleton: bildet entferntes Interface, zerlegt Anfrage-nachricht und bildet Ergebnisnachricht, ruft die implementierte Methode im entfernten Objekt auf

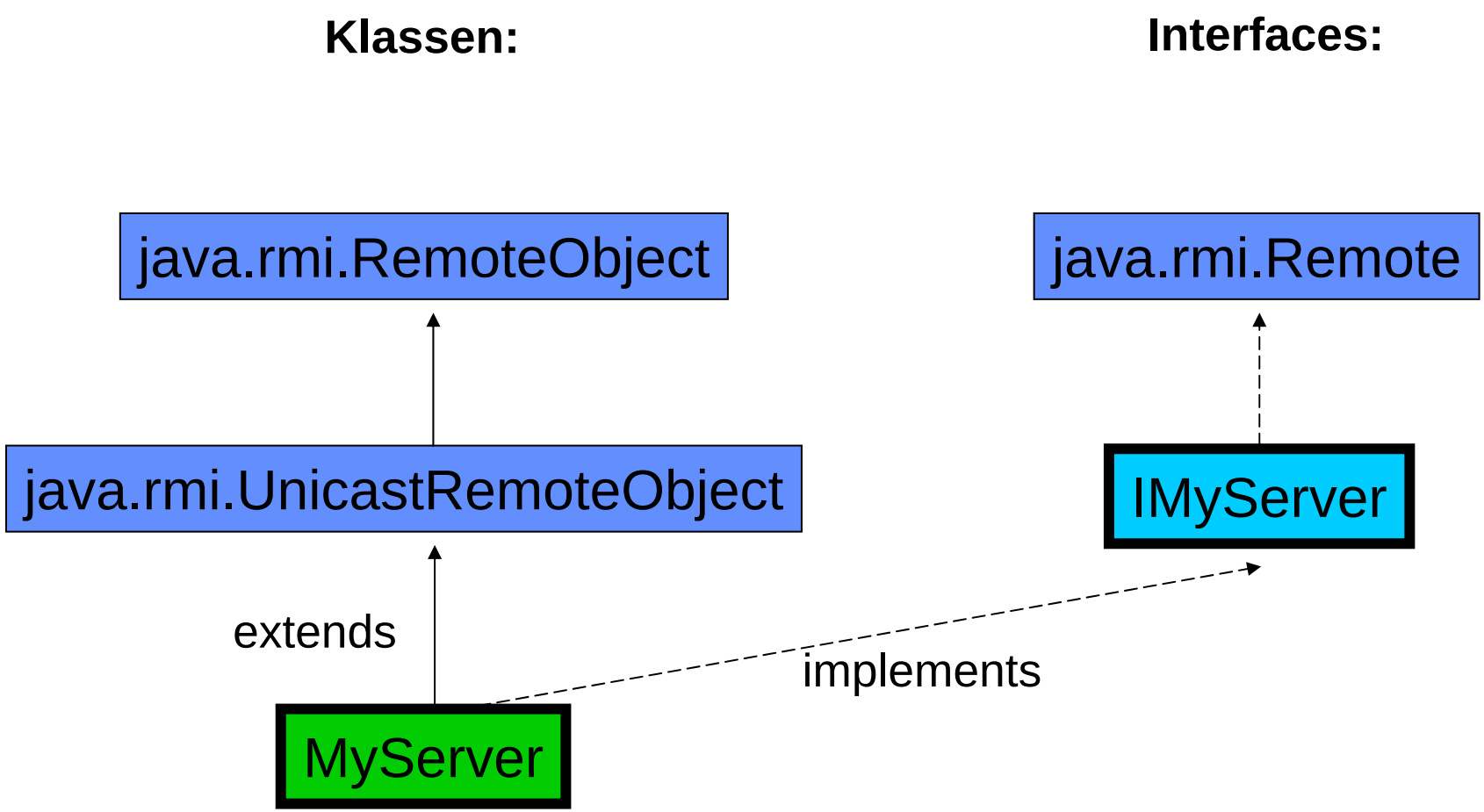
Dispatcher: bekommt die Nachricht vom Kommunikationsmodul, ruft die Methode des Skeletons auf (benutzt *methodID* der Nachricht).

5.3 Java-RMI

Herausforderung und Ziele:

- **Vereinfachung der Entwicklung zuverlässiger verteilter Anwendungen**
- **Automatisches Erzeugen von Threads**
- **Unterstützung von Unicast and Multicast**
- **Speicherbereinigung (Garbage Collection)**

5.3 Benötigte Klasseneinbindung des Servers



5.3 Beispiel: Schläger (Bat) trifft Ball

```
public class Bat {
    public void play (Ball ball) {   ball.hit();   }
    public static void main (String args[]) {
          Ball ball = new Ball();
          Bat bat = new Bat();
          bat.play(ball)  } }
```

soll auf Client laufen

```
public class Ball {
    public void hit() {
           System.out.println("Ball has been hit.") } }
```

soll auf Server laufen

5.3 Beispiel: Schläger (Bat) trifft Ball
Implementierung des Servers

Interface IBall:

```
import java.rmi.*;

public interface IBall extends Remote {
    public void hit() throws RemoteException;
}
```

5.3 Beispiel: Schläger (Bat) trifft Ball
Implementierung des Servers

Class Ball:

```
import java.rmi.*
import java.rmi.server.*;

public class Ball extends UnicastRemoteObject implements IBall {
    public Ball() throws RemoteException {
            super(); }
    public void hit() {
            System.out.println("Ball has been hit."); }

    public static void main(Sting args[]) {
            try {
               Ball myBall = new Ball();
               Naming.rebind("Ball1", myBall);
            } catch (Exception e) { e.printStackTrace(); }}}
```

5.3 Beispiel: Schläger (Bat) trifft Ball
Implementierung des Clients

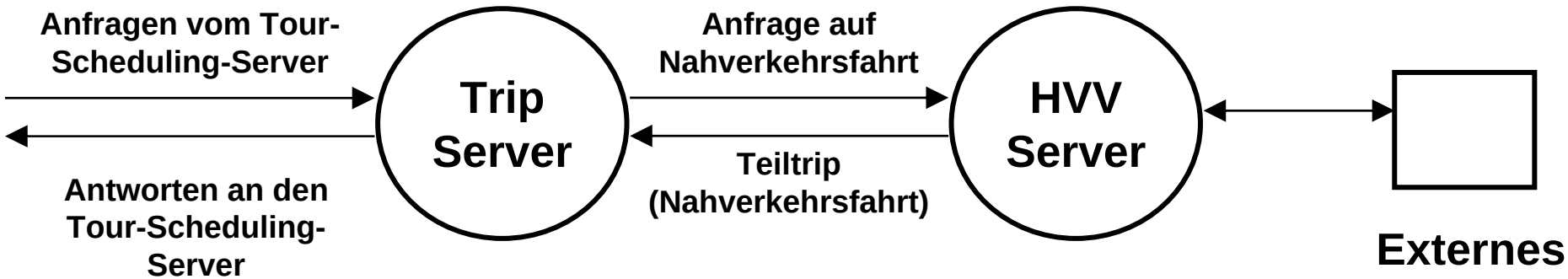
Class Bat:

```
import java.rmi.*;
public class Bat {
   public Bat () {
        super();
        System.setSecurityManager(new RMISecurityManager());
   }
   public void play(Ball ball) {
      try { ball.hit(); }
      catch (RemoteException e) { ... }
   }
   public static void main(String args[]) {
      Bat bat = new Bat();
      try {Ball ball = (Ball) Naming.lookup("hostname"+"Ball1");
           bat.play(ball); }
      catch (RemoteException e) { ... }
   }
}
```

5.4: Praxisbeispiel:

Java-RMI im Touristeninformationssystem

5.4 Java-RMI im Touristeninformationssystem



zerlegt den gesuchten Trip in mehrere Teiltrips und stellt für jeden eine Frage an den entsprechenden Routing-Server, setzt die Antworten hinterher zusammen

ist Teil des Touristeninformationssystems, stellt Verbindung zur externen Informationsquelle dar.

5.4 Java-RMI im Touristeninformationssystem

Interface des TripServers (für Serverfunktion):

```
public interface ITripServer extends Remote
{
  public Trip getTrip(Integer queryId, TripQuery tripQuery)
        throws RemoteException;
        // für Anfrage des TourSchedulingServers
}
```

5.4 Java-RMI im Touristeninformationssystem

Konstruktor des TripServers:

```
public TripServer () throws RemoteException {
    ... // einige TripServer-spezifische Initialisierungen

    try {
      java.rmi.registry.LocateRegistry.createRegistry(1099);
       // macht Registraturtabelle auf (Serverfunktion)
    }
      catch(java.rmi.server.ExportException bindExc) { ...
       // für den Fall, dass Registraturtabelle schon existiert
    }

    System.setSecurityManager(new RMISecurityManager());
       // für Stubgenerierung des TripServers (Clientfunktion)

    try{
      Naming.rebind ("TripServer", this);
       // für Anfragen an den TripServer (Serverfunktion)
    }
    catch(Exception x){...}
}
```

5.4 Java-RMI im Touristeninformationssystem

Anfragemethode des TripServers an den HVVServer (für Clientfunktion):

```
private Trip ask // interne Methode des TripServers
  (StopPlace start, StopPlace dest, TimeStamp departure, TimeStamp arrival)
    ...
    TripQuery query = ...;
    try {
        IHVVServer hvvServer = (IHVVServer) Naming.lookup
                ("//" + System.getProperty ("HVVServerhost") +"/HVVServer");
        Integer publicQueryId = new Integer (maxQueryId++);
                // Zuweisung einer QueryId: Objekt wegen Serialisierbarkeit
        ...
        answer = hvvServer.processQuery (publicQueryId, query);
                // remote call (setzt automatisch neuen Thread)
        return answer;
    }
    catch (Exception e) {
        System.err.println
                ("TripServer: Exception while quering for the Trip: "
                        + e.getMessage());
        return null;
        }
```

5.4 Java-RMI im Touristeninformationssystem

Interface des HVVServers (für Serverfunktion):

```
public interface IHVVServer extends Remote{
  public SimTrip processQuery (Integer queryId, TripQuery query)
                                                    throws RemoteException;
        // für Anfrage des TripServers
}
```

5.4 Java-RMI im Touristeninformationssystem

Konstruktor des HVVServers:

```
public HVVServer () throws RemoteException {
    ... // einige HVVServer-spezifische Initialisierungen

    try {
      java.rmi.registry.LocateRegistry.createRegistry(1099);
       // macht Registraturtabelle auf (Serverfunktion)
    }
      catch(java.rmi.server.ExportException bindExc) { ...
       // für den Fall, dass Registraturtabelle schon existiert
    }

    //  System.setSecurityManager(new RMISecurityManager());
       // nicht nötig, wenn Router nicht in RMI-System

    try{
      Naming.rebind ("HVVServer", this);
       // für Anfragen an den HVVServer (Serverfunktion)
    }
    catch(Exception x){...}
}
```

Zusammenfassung: Java-RMI

- **maschinenunabhängige Namensverwaltung für Objektreferenzen**
- **Senden einer Nachricht (synchrone Kommunikation)**
- **RMI übernimmt Parameter- und Ergebnistransfer (Marshalling and Demarshalling):**
  - **Umrechnung in externe Datenrepräsentation**
  - **Wiederherstellung von Objektstrukturen**
  - **Nachrichtenobjekte müssen Interface "*serializable* " implementieren**
- **RMI übernimmt Threadsetzung und Koordination des Multithreading**
- **RMI läuft in heterogenen SW-Umgebungen, solange die Server die Virtual Machine von Java verwenden (betriebssystemunabhängig)**

Zusammenfassung: Java-RMI

einfache Sicherheitsfunktionen für RMI-Stubs

z.B. ist verboten:

- **Abhören von Ports**
- **Öffnen von Dateien**
- **Prozesse starten**
- **Manipulation fremder Threads**
- **Beenden einer Java Virtual Machine**

Probleme bei Java-RMI

- **standardmäßig keine Verschlüsselung von Daten**
- **standardmäßig keine Authentifizierung**
- **kein Zugriffsrechtesystem: jeder kann die Registratur abfragen**
- **Stubs / Skeletons können simuliert werden: Trojaner möglich !**
- **keine Versionskontrolle zwischen Stub und Skeleton: Inkonsistenzen möglich**
- **keine Transaktionskonzepte für Transaktionen aus mehreren Methoden oder Zugriff auf gemeinsame Daten**

Beim nächsten Mal:

Objektmigration
Agententechnologie