Verteilte Systeme

Vorlesung 5 vom 06.05.2004

Dr. Sebastian Iwanowski
FH Wedel

Inhaltsverzeichnis für die Vorlesung

Zur Motivation: 4 Beispiele aus der Praxis

Allgemeine Anforderungen an Verteilte Systeme

Konzepte verteilter Hardware

Die Client-Server-Beziehung und daraus entstehende Fragestellungen

Grundlagen der Kommunikation in verteilten Systemen

➔ Nebenläufigkeitstechniken

Entfernte Aufrufe / Objektmigration

Namensverwaltung / Namenssuche

Dienstevermittlung

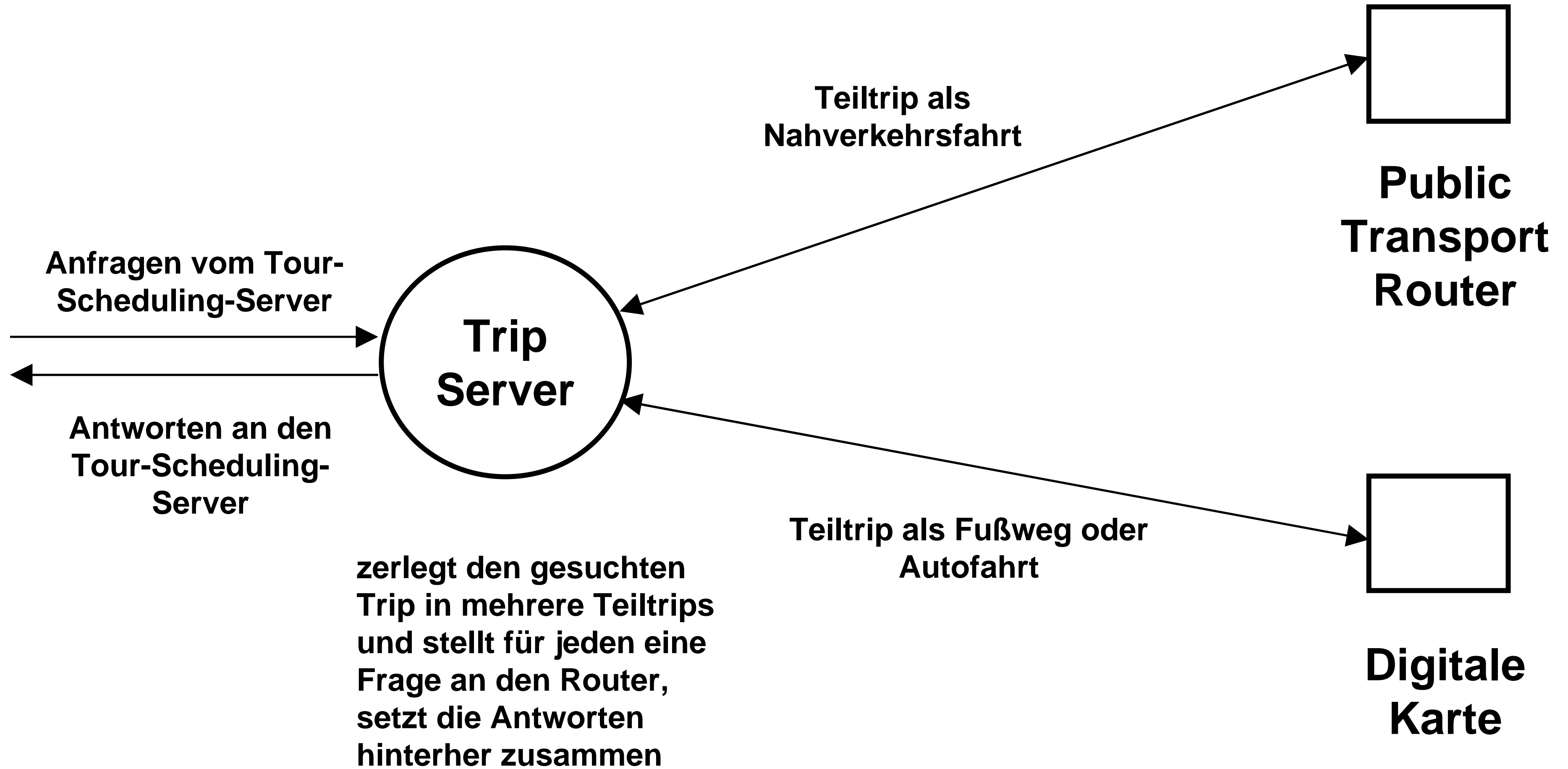
Synchronisation von Daten

Konzepte zur Erzielung von Fehlertoleranz

Sicherheit

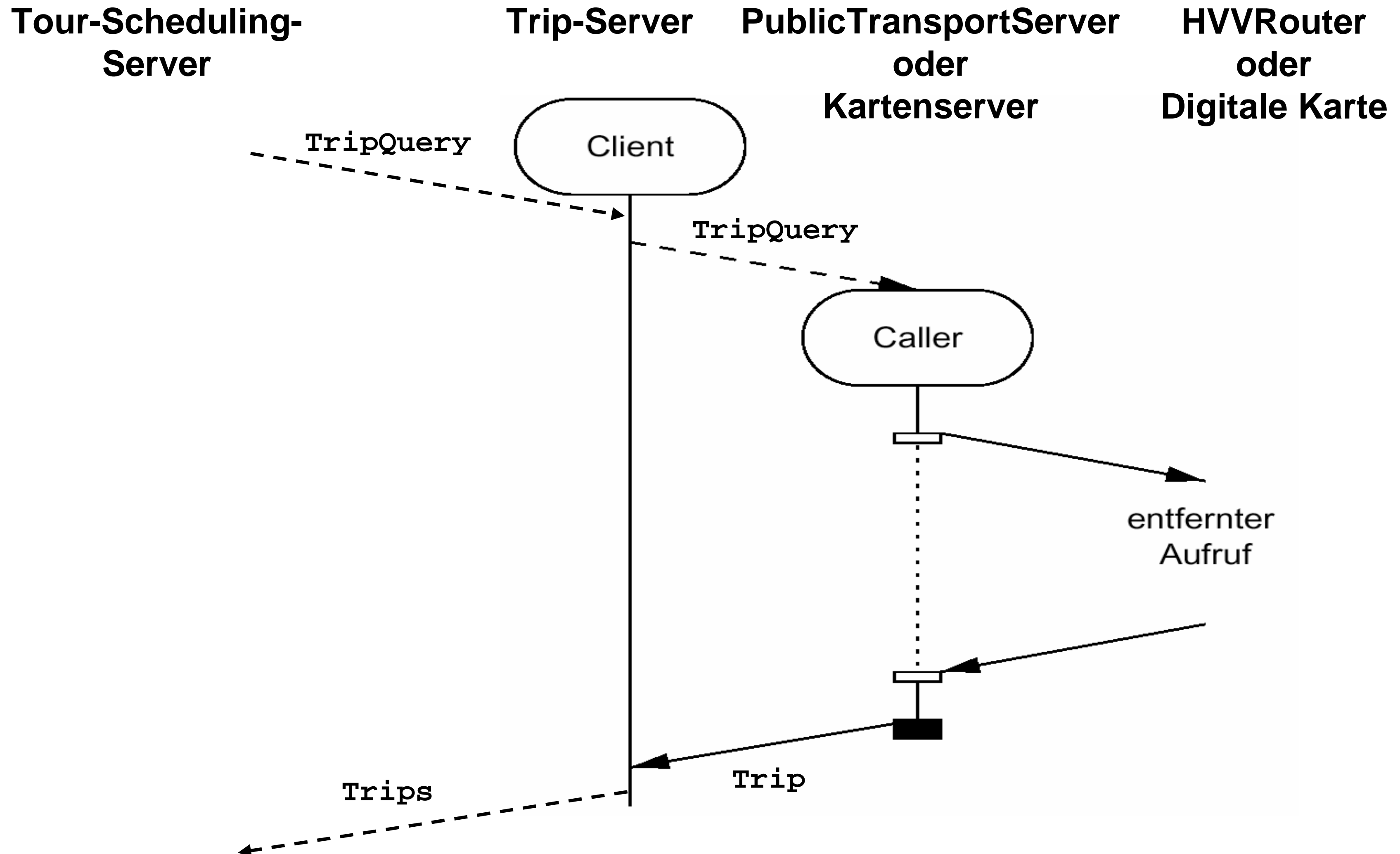
Ausblick auf konkrete Software: J2EE, SOAP,...

Beispiel TripServer aus dem Touristeninformationssystem



Beispiel TripServer aus dem Touristeninformationssystem

Callback-Verfahren:



Beispiel TripServer aus dem Touristeninformationssystem

Callback-Verfahren, autonome Variante ohne Threads

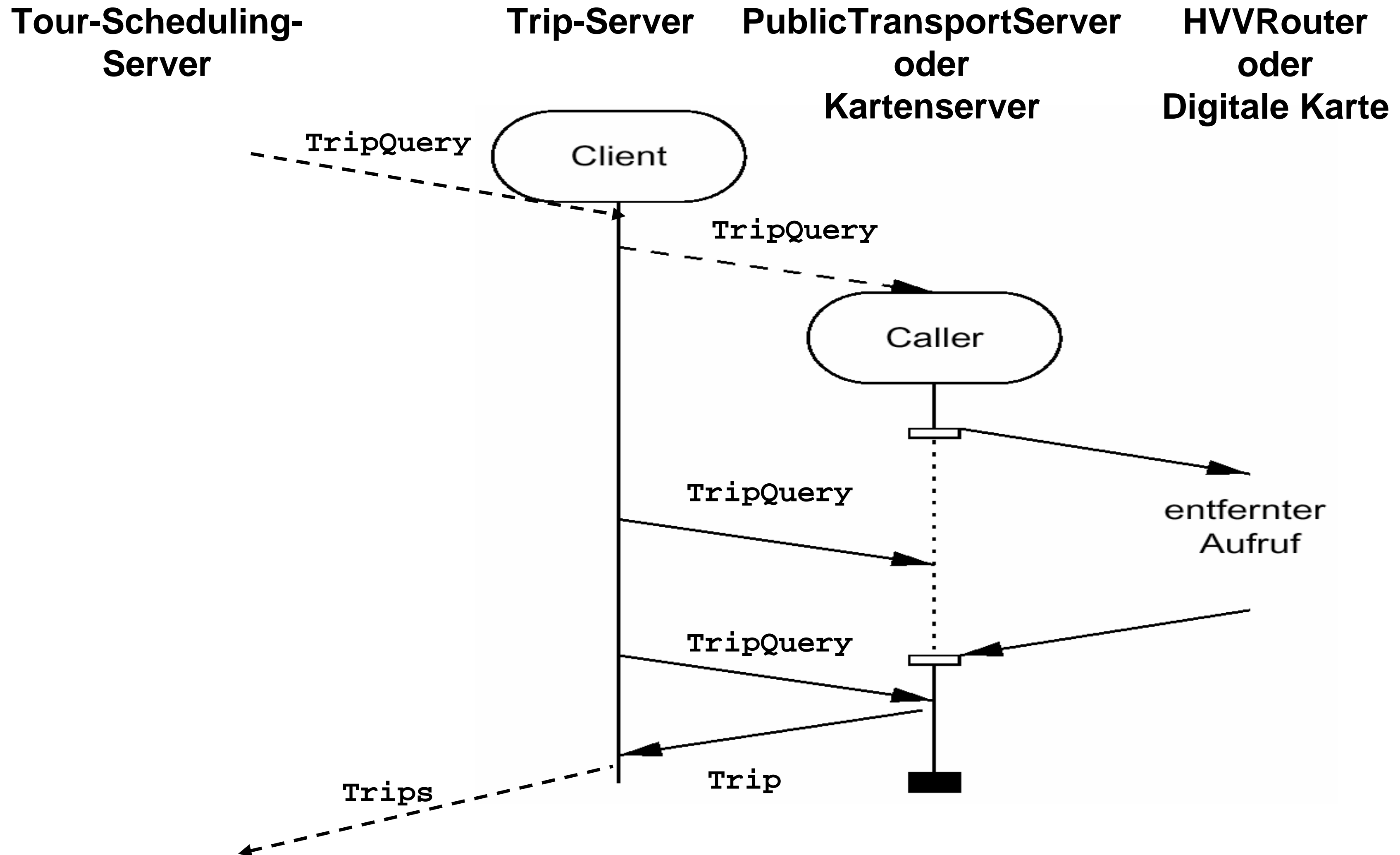
- erfordert 2 getrennte Socket-Verbindungen pro Teiltrip zwischen Trip-Server und Caller
- erfordert mindestens 4 getrennte Methoden

Callback-Verfahren, Java-spezifische Variante mit Threads

- erfordert nur eine getrennte Socket-Verbindung pro Teiltrip zwischen Trip-Server und Caller
- erfordert mindestens 3 getrennte Methoden

Beispiel TripServer aus dem Touristeninformationssystem

Polling-Verfahren:



Beispiel TripServer aus dem Touristeninformationssystem

Polling-Verfahren, Java-spezifische Variante mit Threads,

erfordert periodisch eine Socket-Verbindung pro Teiltrip zwischen Trip-Server und Caller

Methoden des Trip-Servers:

```
void processInitialRequest (TripQuery tripQuery);
/* wird vom Tour-Scheduling-Server über eine Socket-Verbindung in einem eigenen
InitialThread aufgerufen,
zerlegt den Trip in Teiltrips und ruft für jeden Teiltrip ask auf,
Schleife:
   setzt InitialThread periodisch in den befristeten Wartezustand mit wait(timeout).
   ruft für jeden Teiltrip getTrip auf, wenn timeout vorüber ist,
   wenn alle Antworten vorliegen,
      verarbeitet sie weiter und ruft giveFinalAnswer auf, verlässt Schleife
   sonst
      bleibt in Schleife */

void ask (TripQuery tripQuery);
/* baut Socket-Verbindung zum Caller auf, stellt den Auftrag an den Caller,
fragt nach dem Ergebnis und bricht die Socket-Verbindung gleich wieder ab. */

Trip getTrip (TripQuery tripQuery);
/* baut Socket-Verbindung zum Caller auf, fragt nach dem Trip zur gegebenen
tripQuery und bricht die Socket-Verbindung gleich wieder ab. */

void giveFinalAnswer (TripQuery tripQuery, Trip tripAnswer);
/* ruft eine entsprechende Methode des Tour-Scheduling-Servers auf,
wenn alle Teilergebnisse vorliegen und zusammengesetzt sind */
```

Nebenläufigkeitstechniken

letzter Teil: Risiken von Nebenläufigkeit und Vorsorgemaßnahmen

Risiken von Nebenläufigkeit

Verklemmung (deadlock)

- wechselseitiges Warten auf reservierte Betriebsmittel oder wechselseitiges Warten auf Prozesse.

Verhungern (livelock)

- Warten auf Ressourcen, die nie zur Verfügung stehen, oder auf Prozesse, die nie geschehen.

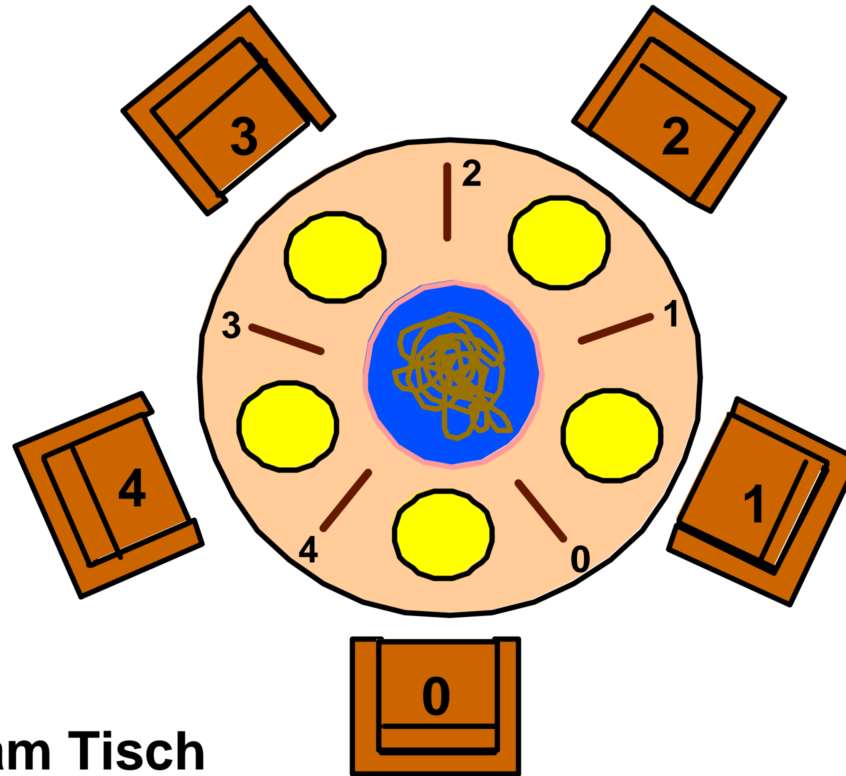
Ungleichbehandlung (unfairness)

- Bevorzugung von bestimmten Prozessen bei der Ressourcenverteilung

Wettlaufeffekte (race conditions)

- Ergebnis einer Berechnung hängt von der Ausführungsreihenfolge ab

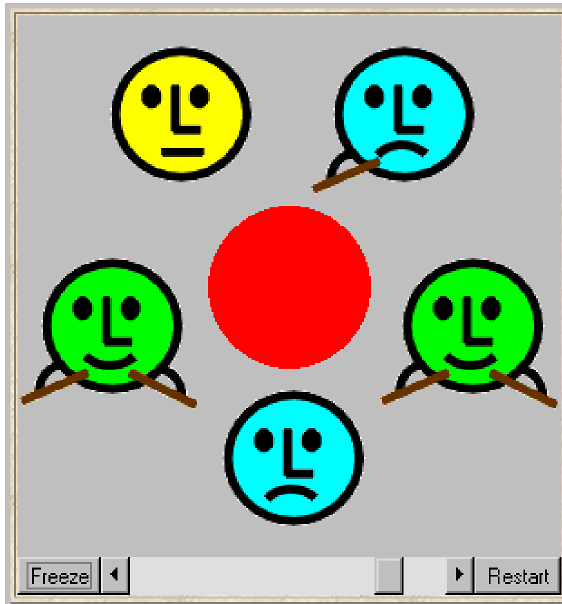
Beispiel für Deadlockgefahr: Dining Philosophers (Dijkstra)



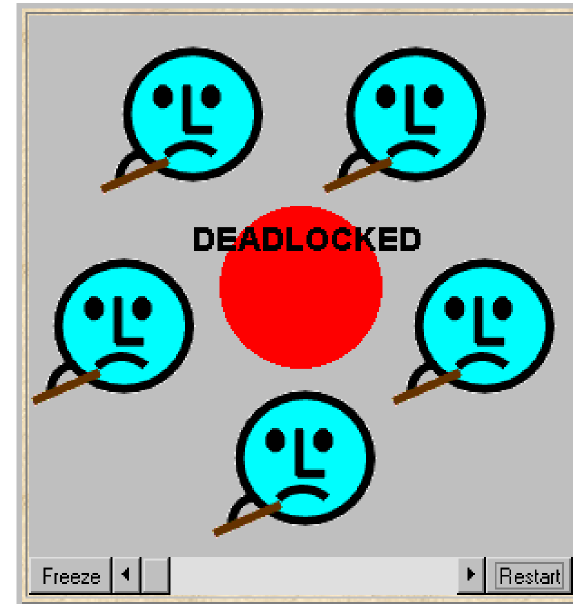
- 5 Philosophen sitzen am Tisch
- Zwischen je 2 Philosophen liegt eine Gabel
- Philosophen wechseln zwischen zwei Aktivitäten: Denken und Essen
- Zum Essen braucht ein Philosoph die beiden Gabeln zu seinen Seiten
- Zum Denken braucht ein Philosoph keine Ressourcen

Beispiel für Deadlockgefahr: Dining Philosophers (Dijkstra)

Optimaler Zustand:



Deadlock:



Implementierungsbedingungen:

Alle Philosophen wechseln zyklisch zwischen den Zuständen:

- **Denken (immer ohne Gabeln)**
- **Hungrig sein (sie haben aber noch nicht beide Gabeln)**
- **Essen (immer mit 2 Gabeln)**

Die Philosophen sollen <u>autonome</u> Softwareeinheiten sein

Mögliche Implementierung eines Philosophen:

```
class Philosopher extends Thread {
  ...
  public void run() {
    try {
      while (true) {
        view.setPhil(identity,view.THINKING);
        sleep(controller.sleepTime());          // Denkzeit
        view.setPhil(identity,view.HUNGRY);     // will essen
        right.get();                            // greift die rechte Gabel
        view.setPhil(identity,view.GOTRIGHT);
        left.get();                             // greift die linke Gabel
        view.setPhil(identity,view.EATING);
        sleep(controller.eatTime());           // Speisezeit
        right.put();                           // legt die rechte Gabel hin
        left.put();                            // legt die linke Gabel hin
      }
    } catch (java.lang.InterruptedException e){}
  }
}
```

Problem:

Es muss verhindert werden, dass zwei Philosophen gleichzeitig dieselbe Gabel greifen !
➔ Die Methode get darf nicht gleichzeitig von verschiedenen Threads aufgerufen werden

Lösung: **Durch Monitormethoden**
(Methoden, die nur von einem Thread gleichzeitig benutzt werden dürfen)

Implementierung der Gabel:

```
class Fork {
  private boolean taken=false;
  private int identity;
  ...
  void put() {                                         ← keine Monitormethode
    taken=false;
    display.setFork(identity,taken);
    notify();
  }

  synchronized void get()                              ← Monitormethode
     throws java.lang.InterruptedException {
    while (taken) wait();
    taken=true;
    display.setFork(identity,taken);
  }
}
```

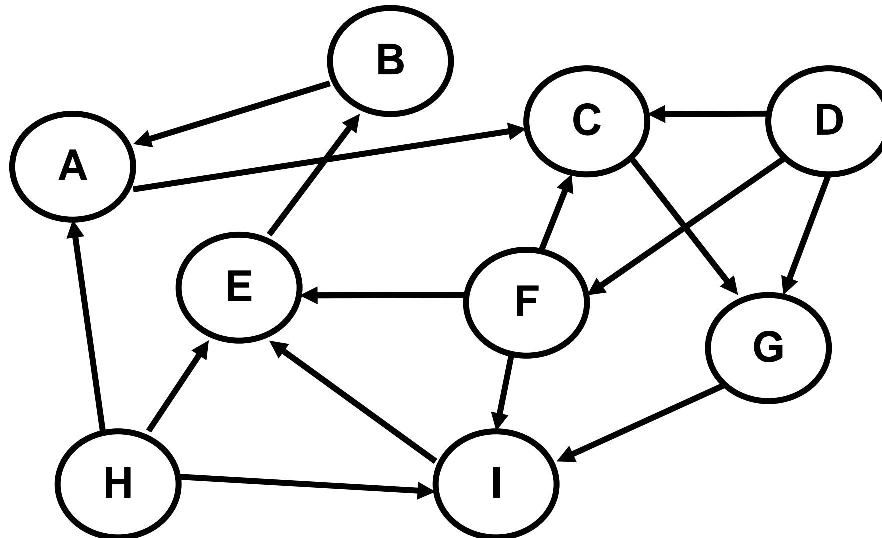
Verallgemeinerung von Monitoren: Semaphore

- **Es gibt endlich viele Marken**
- **Jeder Thread, der eine geschützte Methode ausführen will, muss eine Marke nehmen**
- **Am Ende der geschützten Methode legt der Thread die Marke wieder zurück**

Wie kann man einen Deadlock beheben ?

Antwort: Wer einen Deadlock erkennt, gibt die Ressourcen frei !

Deadlockerkennung: Analyse des Wartegraphen



- **Finde einen Kreis → Deadlock !**
- **Starte lokal mit der Kreissuche durch sukzessives Anstoßen des Nachbarn und füge dich selbst in die Nachbarliste ein**
- **Bei jedem Propagationsschritt wird der neue Nachbar der bisherigen Nachbarliste hinzugefügt**
- **Ein Kreis ist erkannt, wenn sich ein neu angestoßener Nachbar in der bisherigen Nachbarliste wiederfindet**

Wie kann man einen Deadlock verhindern ?

➔ durch Festlegung eines Regelwerks, an das sich alle halten müssen

Beispiel eines deadlockverhindernden Regelwerks für die Philosophen:

- **Es dürfen gleichzeitig nur 4 Philosophen am Tisch sitzen (d.h. mindestens einer darf keine Gabeln beanspruchen).**
- **Jeder Philosoph, der gerade gegessen hat, muss zum Denken den Tisch verlassen.**

Funktionierendes, aber schlechtes Beispiel:

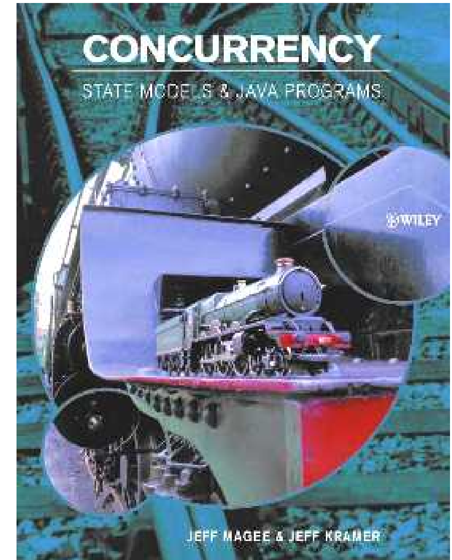
verletzt die Softwareautonomie der Philosophen!

- **Jeder Philosoph mit einer geraden Id ergreift zuerst die linke Gabel.**
- **Jeder Philosoph mit einer ungeraden Id ergreift zuerst die rechte Gabel.**

Beispiel für Deadlockgefahr: Dining Philosophers (Dijkstra)

Benutztes Material aus:

Jeff Magee / Jeff Kramer: Concurrency, State Models and Java Programs, Wiley 1999, ISBN 0-471-98710-7



Weiteres Material zum Buch in:

http://www.doc.ic.ac.uk/~jnm/book/

Weitere Infos zum Thema auf S. 188 ff. in:

Heinz Kredel / Akitoshi Yoshida: Thread- und Netzwerk-Programmierung mit Java, dPunkt-Verlag 1999, ISBN 3-932588-28-2

Weitere Vorsorgemaßnahmen

gegen Verhungern (livelock):

- Timeout nach hinreichend langer Wartezeit (hilft auch bei Deadlock)
- Regelwerk

gegen Ungleichbehandlung (unfairness):

- Zufallsgenerator bei Verteilung von Aufgaben oder Ressourcen

gegen Wettlaufeffekte (race conditions):

- Sperren von bestimmten Prozessen bzw. Voraussetzen von bestimmten Bedingungen

Beispiel für Wettlaufeffekte

Umbuchung

```
read (A, a1)
a1 := a1 - 300
write (A, a1)
read (B, b1)
b1 := b1 + 300
write (B, b1)
```

Prozess 1: Umbuchung eines Betrages von Konto A nach Konto B

Prozess 2: Tagesgenaue Zinsgutschrift für Konto B

Zinsgutschrift

```
read (B, b2)
b2 := b2 * 1.0001
write (B, b2)
```

Anfangskontostand für B: 1000 €

Erst Prozess 1, dann Prozess 2:

Umbuchung	Zinsgutschrift
read (A, a1)	
a1 := a1 - 300	
write (A, a1)	
read (B, b1)	
b1 := b1 + 300	
write (B, b1)	
	read (B, b2)
	b2 := b2 * 1.0001
	write (B, b2)

Erst Prozess 2, dann Prozess 1:

Umbuchung	Zinsgutschrift
	read (B, b2)
	b2 := b2 * 1.0001
	write (B, b2)
read (A, a1)	
a1 := a1 - 300	
write (A, a1)	
read (B, b1)	
b1 := b1 + 300	
write (B, b1)	

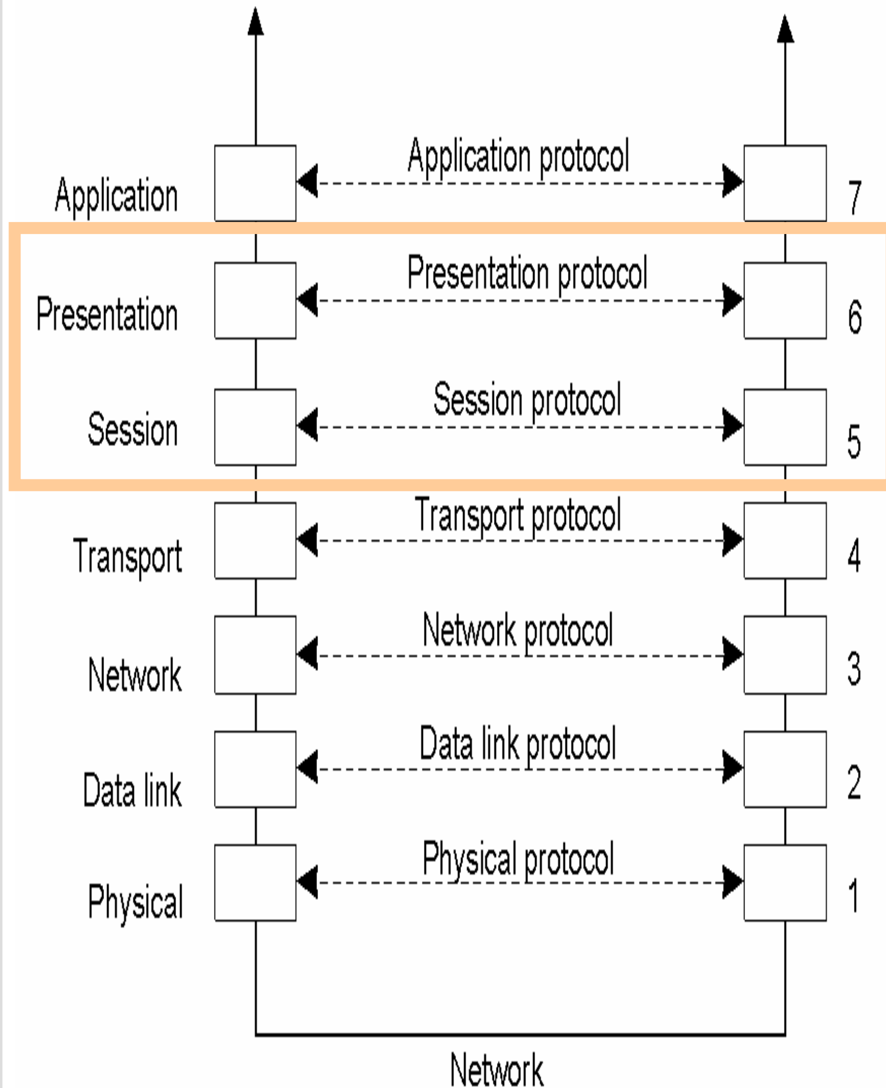
Endkontostand für B: 1300,13 €

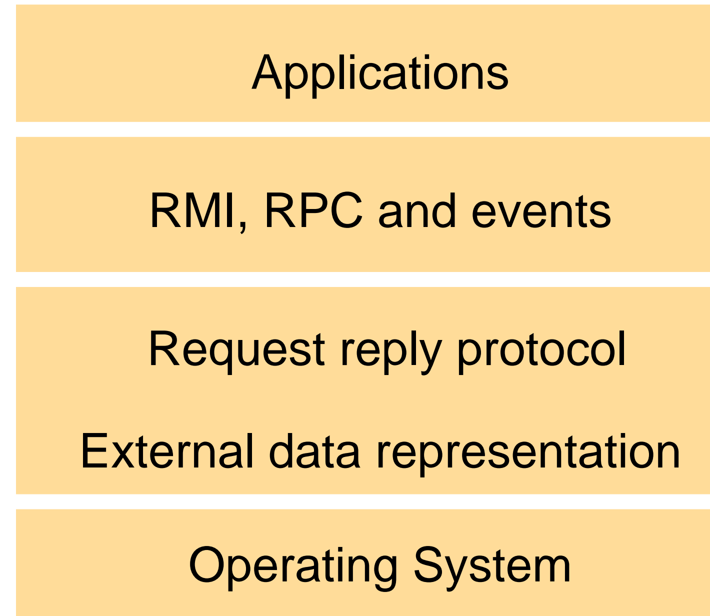
Endkontostand für B: 1300,10 €

Entfernte Aufrufe / Objektmigration

Teil 1: Referenzierung entfernter Objekte

Middleware-Schichten bei den Kommunikationsprotokollen





Middleware layers

Beispiel für Nachrichtenformat: CORBA CDR message

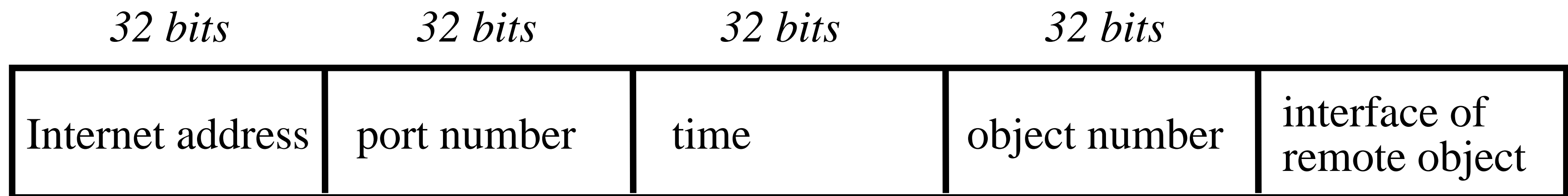
index in sequence of bytes	← 4 bytes →	*notes on representation*
0–3	5	*length of string*
4–7	"Smit"	*'Smith'*
8–11	"h "	
12–15	6	*length of string*
16–19	"Lond"	*'London'*
20-23	"on "	
24–27	1934	*unsigned long*

The flattened form represents a *Person* struct with value: {'Smith', 'London', 1934}

Repräsentation der Referenz auf ein entferntes Objekt

Problem:

Referenzen auf Objekte eines Adressraumes haben in einem anderen (entfernten) Adressraum keine Bedeutung



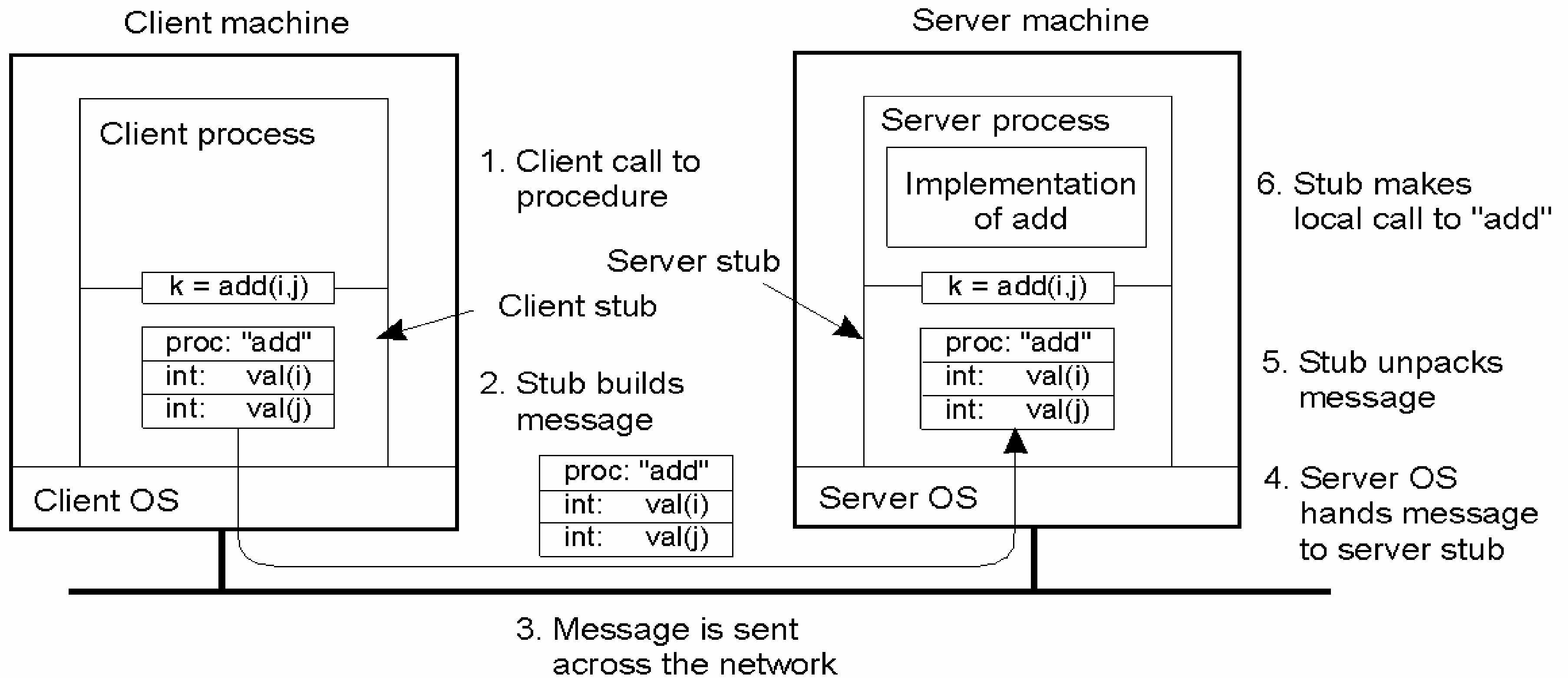
Mehr zum Thema in Coulouris, Kap. 4.3

Entfernte Aufrufe / Objektmigration

Teil 2: Remote Procedure Call

Remote Procedure Call

Architektur:



Remote Procedure Call

Funktionsabfolge:

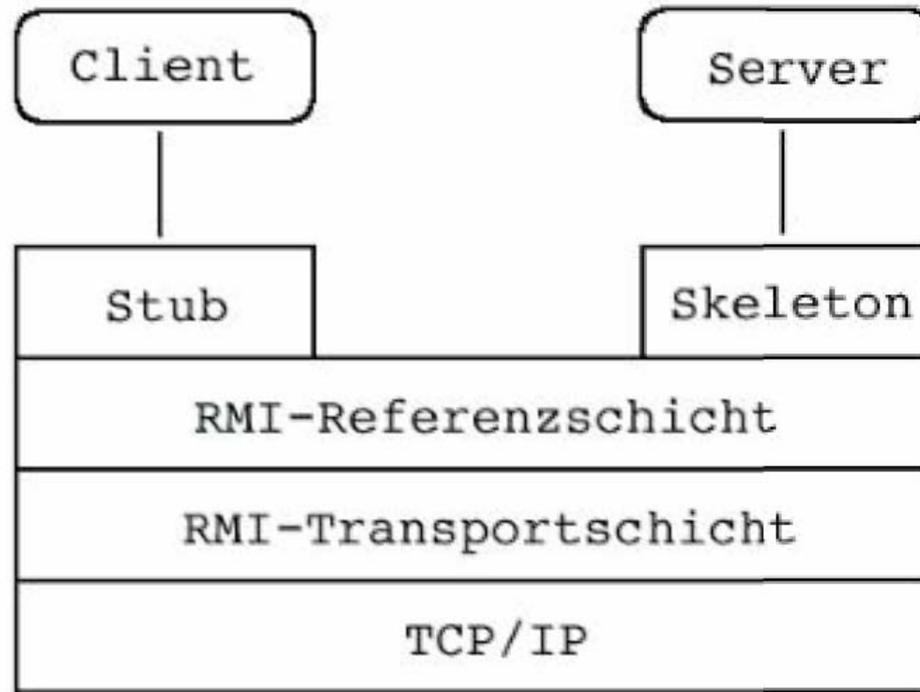
1. Aufrufende Prozedur im Client ruft den Stub wie eine normale Prozedur auf.
2. Der Client-Stub erzeugt eine Nachricht und ruft sein lokales Betriebssystem auf.
3. Das lokale Betriebssystem sendet die Nachricht an das entfernte Betriebssystem.
4. Das entfernte Betriebssystem gibt die Nachricht an den 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

Entfernte Aufrufe / Objektmigration

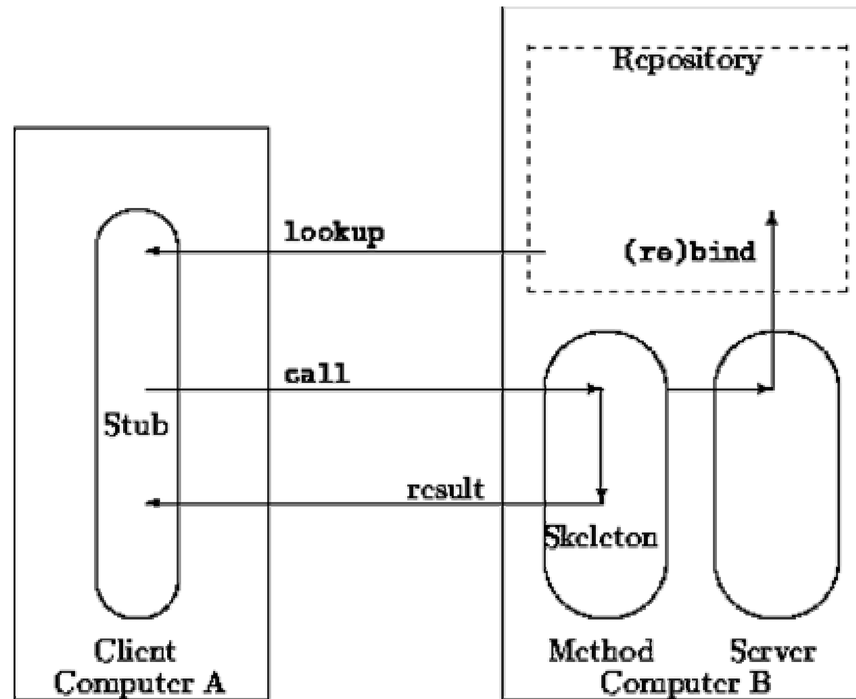
Teil 3: Remote Method Invocation

Remote Method Invocation: Prinzip



- **Server veröffentlicht Schnittstelle**
- **Client muss Proxyklasse für Schnittstelle einrichten**
- **Namensverwaltung notwendig (siehe Referenzierung entfernter Objekte)**

Remote Method Invocation: 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.

Java-RMI: Herausforderungen und Ziele

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

Beim nächsten Mal:
Java-RMI
Objektmigration