

# ***Verteilte Systeme***

Vorlesung 4  
Sebastian Iwanowski  
FH Wedel

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

# Was ist Nebenläufigkeit ?

Ausführung mehrerer Programme in verschiedenen Prozessen im Betriebssystem

- Speicherbereiche vollkommen getrennt
- Kommunikation zwischen den Prozessen schwierig
- Prozesszustandswechsel zeitaufwändig

Ein Thread ist ein leichtgewichtiger Prozess:

- Mehrere Threads in einem Prozess möglich
- Zugriff auf gemeinsamen Speicher bzw. Referenzierung gemeinsamer Objekte
- Kommunikation zwischen Threads leicht möglich
- Threadzustandswechsel schnell

# Zustände und Operationen eines Threads

Verwaltung von Threads durch das „Laufzeitsystem“ (i.a. betriebssystemabhängig !)

Zustände eines Threads:

- laufend
- lauffähig (aber ein anderer ist gerade „dran“)
- vorübergehend blockiert, reaktivierbar nach Zeitlimit
- blockiert, auf Ereignis von außen wartend (z.B. Ein-/Ausgabe)
- unbestimmt blockiert, reaktivierbar auf expliziten Befehl von außen

Operationen mit einem Thread innerhalb eines Programms:

- erzeugen
- starten
- anhalten
- unterbrechen
- wiederaufnehmen

# Threadkonzept in Java

Verwaltung von Threads durch die Java Virtual Machine  
(betriebssystemunabhängig)

Die Ressourcen werden von der Java Virtual Machine auf die  
lauffähigen Threads „gerecht“ verteilt

Threads sind Objekte

Thread-Operationen können von anderen Objekten innerhalb  
und außerhalb des Threads ausgeführt werden

# Threadkonzept in Java

## Methoden der Java-Klassen Object und Thread:

### Class Object

*public void wait();*

*public void wait(long millis);*

*public void wait(long millis, int nanos);*

*public void notify();*

*public void notifyAll();*

### Class Thread

*public void run ();*

*public void start();*

*public void interrupt();*

*public sleep (long millis);*

*viele weitere Methoden*

# Threaderzeugung in Java (1. Variante)

```
class ExampleThread extends Thread { ...  
    ExampleThread(int param) { ... }  
    public void run() { ... }  
  
    public static void main(String[] args) {  
        ExampleThread t = new ExampleThread(42);  
        t.start();  
    }  
}
```

Spezialisierung von Thread  
Konstruktor (beliebig)

Ausführungsmethode  
(muss genau so heißen)

Erzeugen des Threads  
Starten des Threads

# Threaderzeugung in Java

## (2. Variante, für Mehrfachvererbung)

```
class ExampleRunnable extends SomeClass  
    implements Runnable {
```

Zuordnung zum  
Interface Runnable

```
    ExampleRunnable(int param) { ... }
```

Konstruktor (beliebig)

```
    public void run() { ... }
```

Ausführungsmethode  
(muss genau so heißen)

```
    public static void main(String[] args) {  
        ExampleRunnable r =  
            new ExampleRunnable (42);  
  
        Thread t = new Thread (r);  
        t.start(); } }
```

Erzeugen des „Threads“

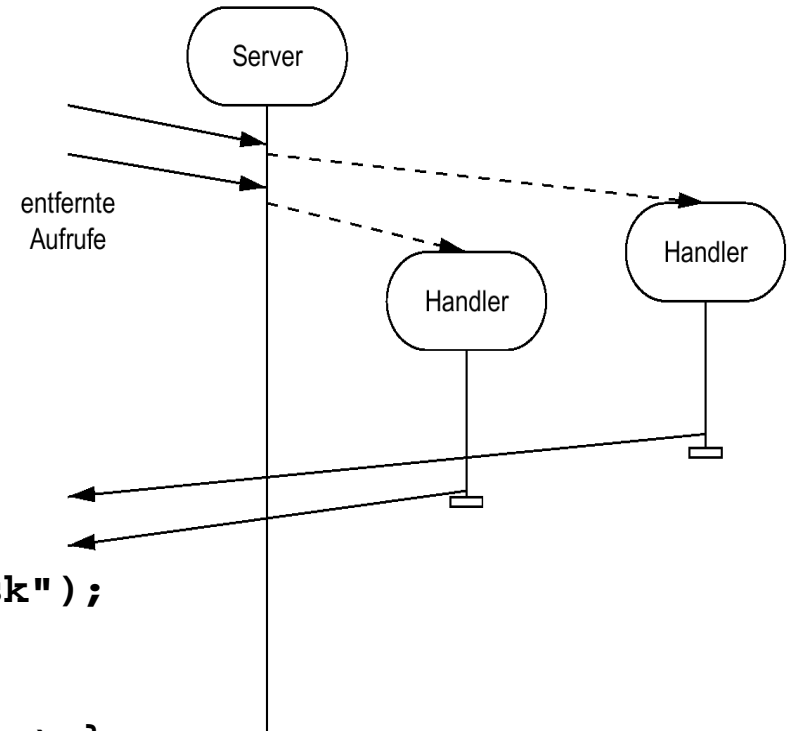
Zuordnung zu einem Thread  
Starten des Threads



# Aufteilung des Servers in Annahme- und Bearbeitungsteil

## Auftragsannahme:

```
public class Server extends Thread {  
    public Server() { this.start() }  
    public static void main(String[] args) {  
        new Server(); }  
    public void run()  
    {while (true) {  
        System.out.println("waiting for new task");  
        try { System.in.read(...);  
            Handler handler = new Handler (... ) }  
        catch (...) { ... } } } }
```



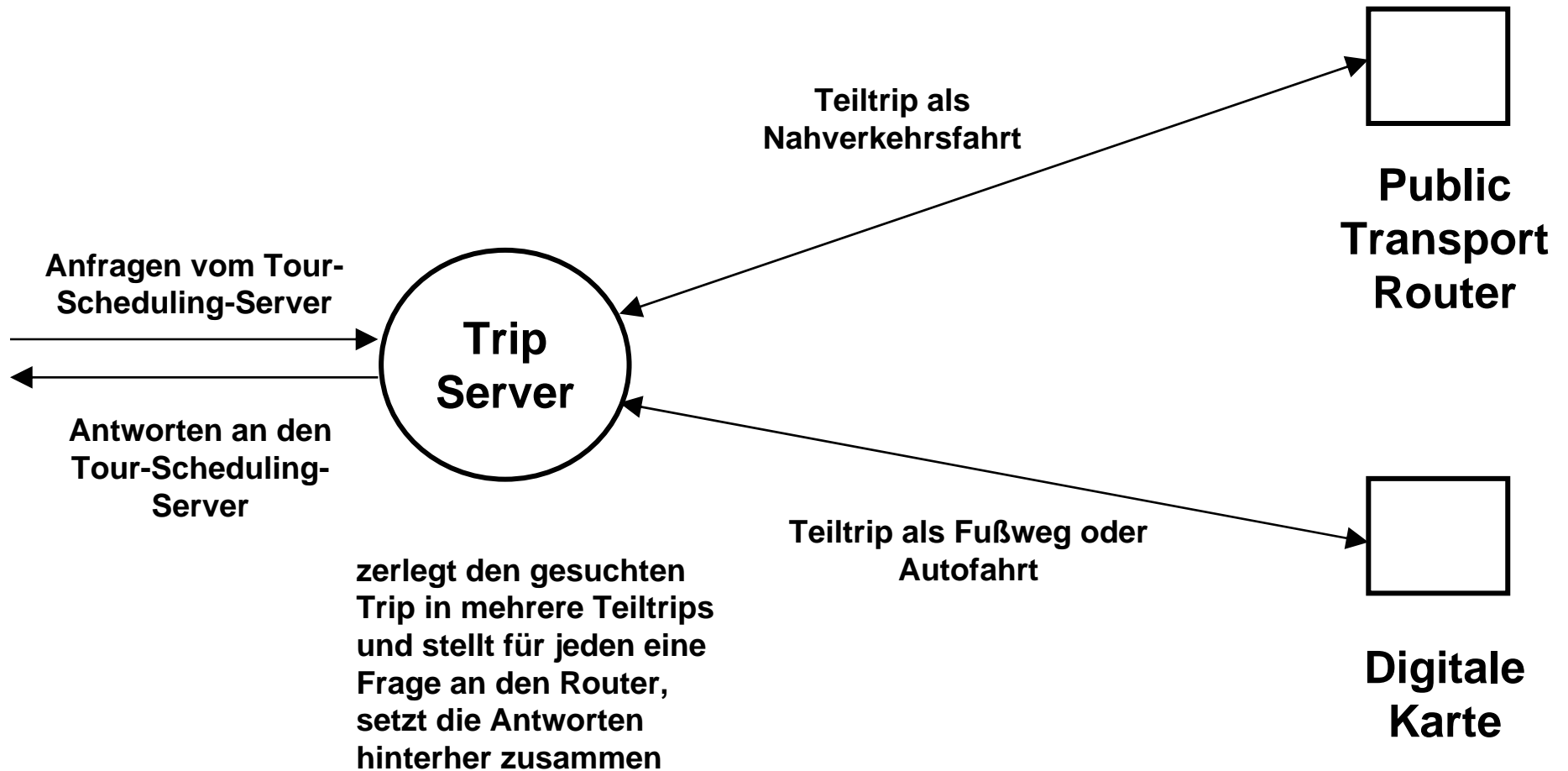
## Auftragsbearbeitung:

```
public class Handler extends Thread {  
    public Handler() { this.start() }  
    public void run() {  
        //führt die Bearbeitung durch } }
```

# Beispiel TripServer aus dem Touristeninformationssystem

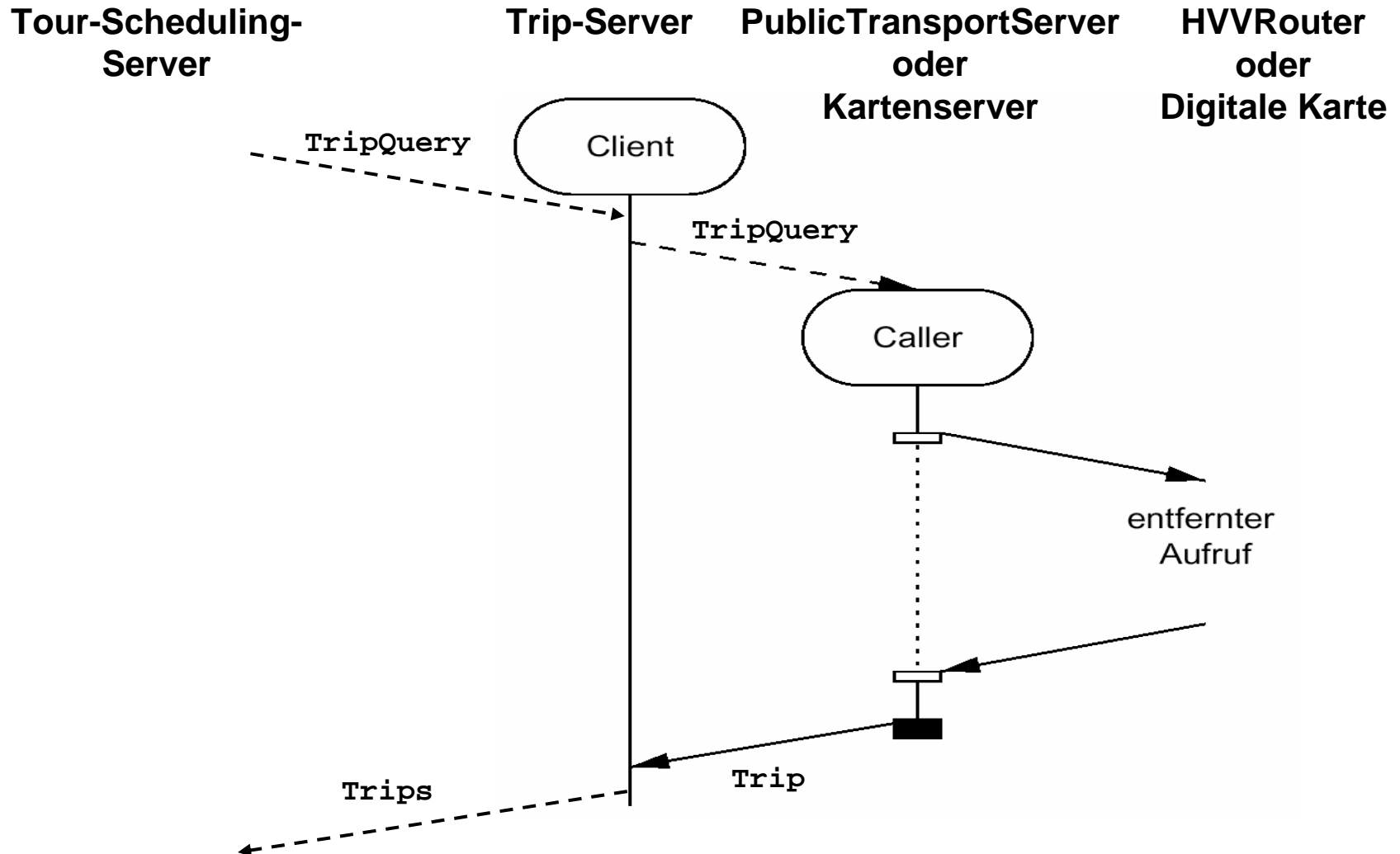
*Die im Folgenden vorgestellte Fallstudie entstand in Zusammenarbeit mit **Sven Bischoff**, Dipl.-Inform. (FH), ehemals Werkstudent und Diplomand bei der DaimlerChrysler AG, Bereich Telematics Research*

# 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

## Methoden des Trip-Servers:

```
void processInitialRequest (TripQuery tripQuery);
```

```
/* wird vom Tour-Scheduling-Server über eine Socket-Verbindung aufgerufen,  
bricht dessen Socket-Verbindung wieder ab, zerlegt den Trip in Teiltrips  
und ruft für jeden Teiltrip ask auf */
```

```
void ask (TripQuery tripQuery);
```

```
/* baut Socket-Verbindung zum Caller auf, stellt die Frage an den Caller,  
der die Socket-Verbindung wieder abbricht */
```

```
void processAnswer (TripQuery tripQuery, Trip tripAnswer);
```

```
/* wird vom Caller aufgerufen, der eine Socket-Verbindung zum Trip-Server aufbaut,  
puffert Antwort und löst unter Umständen weitere Aktionen aus zur Beantwortung des  
initialRequests des Tour-Scheduling-Servers */
```

```
void giveFinalAnswer (TripQuery tripQuery, Trip tripAnswer);
```

```
/* baut eine Socket-Verbindung zum Tour-Scheduling-Server auf,  
ruft eine entsprechende Methode des Tour-Scheduling-Servers auf,  
wenn alle Teilergebnisse vorliegen und zusammengesetzt sind */
```

# Beispiel TripServer aus dem Touristeninformationssystem

**Callback-Verfahren, Java-spezifische Variante mit Threads,**  
erfordert nur 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 (jeweils in eigenem AskingThread unter Mitteilung des  
InitialThreads),  
setzt dann InitialThread in den unbestimmt blockierten Zustand mit wait(),  
prüft nach Reaktivierung, welche Antworten vorliegen, ruft dann entweder  
giveFinalAnswer auf oder setzt InitialThread wieder in den blockierten Zustand */
```

```
Trip ask (TripQuery tripQuery, Thread initialThread);
```

```
/* baut Socket-Verbindung zum Caller auf, stellt die Frage an den Caller und setzt  
seinen eigenen Thread in den Wartezustand, der durch das Ereignis „Antworteingang“  
automatisch beendet wird, weckt dann den InitialThread mit notify() und gibt das  
Ergebnis an processInitialRequest zurück, bricht dann die Socket-Verbindung zum  
Caller wieder ab */
```

```
void giveFinalAnswer (TripQuery tripQuery, Trip tripAnswer);
```

```
/* ruft eine entsprechende Methode des Tour-Scheduling-Servers auf,  
wenn alle Teilergebnisse vorliegen und zusammengesetzt sind */
```

# Beispiel TripServer aus dem Touristeninformationssystem

## Callback-Verfahren, autonome Variante ohne Threads

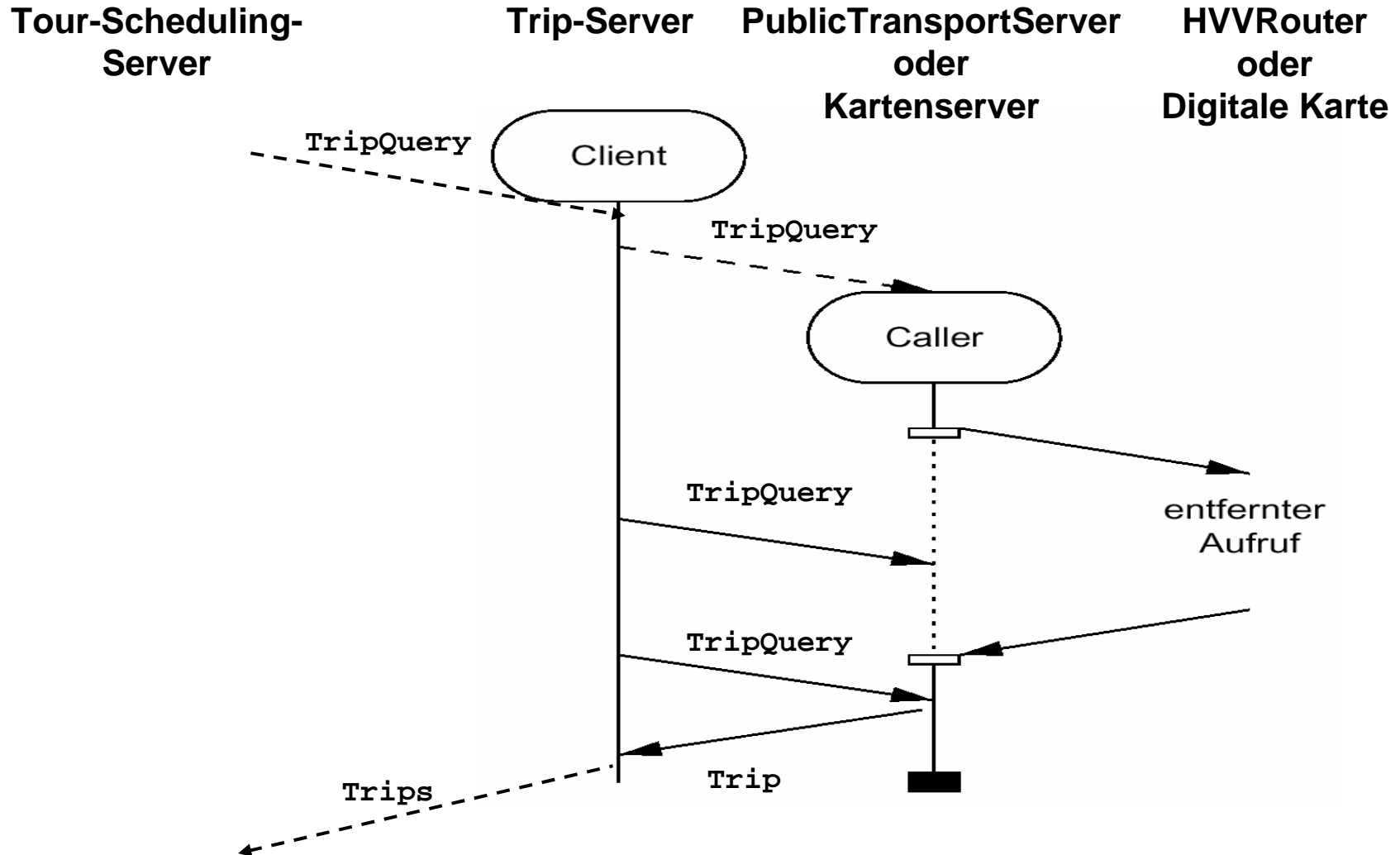
- erfordert 2 getrennte Socket-Verbindungen pro Teiltrip zwischen Trip-Server und Caller
- erfordert mindestens 4 getrennte Methoden

## 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 */
```

# 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

# Vorsorgemaßnahmen gegen diese Risiken

## gegen Verklemmung (deadlock):

- Monitore / Semaphore
- 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

**Details in Vorlesung 5 (2004) (Dining Philosopher etc.)**

**Beim nächsten Mal:  
Entfernte Aufrufe**