Verteilte Systeme

Vorlesung 4 vom 29.04.2004
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 und Prozessen

Konzepte zur Erzielung von Fehlertoleranz

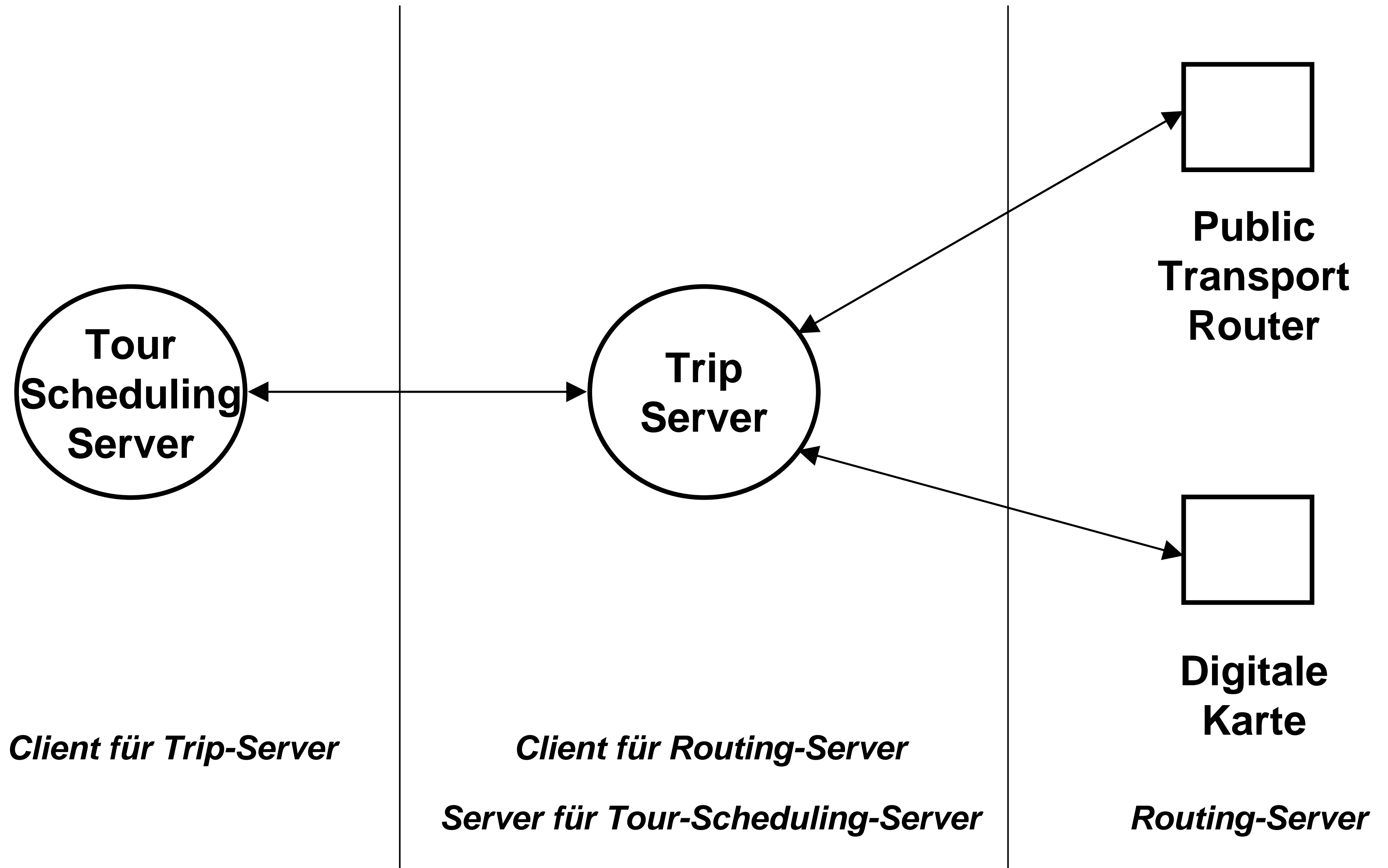
Sicherheit

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

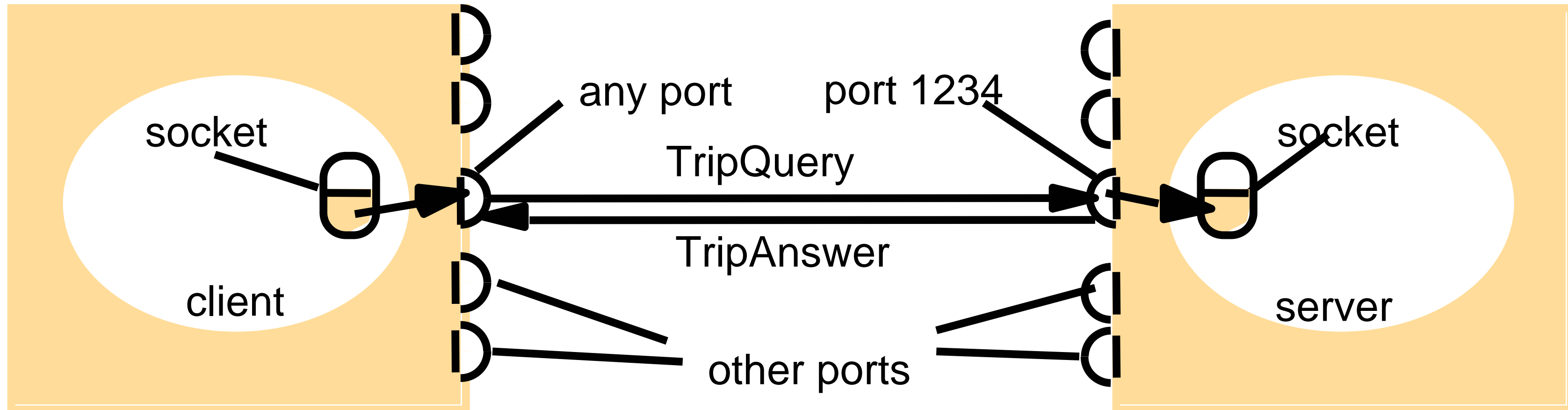
Grundlagen der Kommunikation in Verteilten Systemen

letzter Teil: Socketkommunikation in Java

Beispiel TripServer aus dem Touristeninformationssystem



Beispiel TripServer aus dem Touristeninformationssystem



Für Anfragen eines Clients an den Server:

Einmalige Anmeldung an definierten Port erforderlich

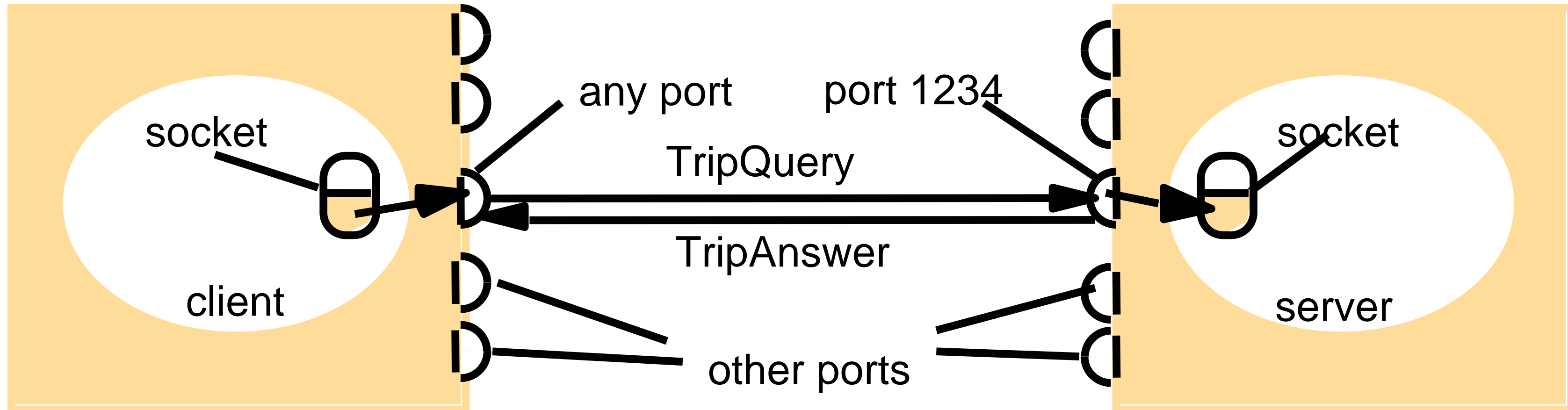
Wiederholte Datenübertragung in beiden Richtungen möglich

In Java:

Socketeinrichtung über Classes Socket und ServerSocket

Übertragung von Daten über Streams

Socketeinrichtung



IP-Adresse = 138.37.94.248
Netz-Name = „TripServer“

IP-Adresse = 138.37.88.249
Netz-Name = „HVVServer“

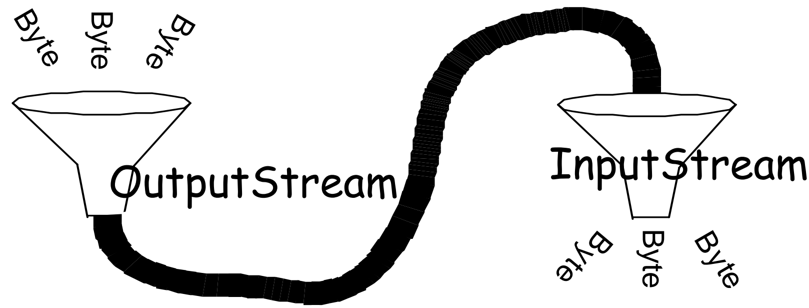
Im Client:

```
Socket server
   = new Socket(„HVVServer", 1234);
System.out.println
   ("Connected to " +
   server.getInetAddress());
```

Im Server:

```
int port = 1234;
ServerSocket server = new
   ServerSocket(port);
while (true) {
   Socket client = server.accept();
   System.out.println ("Client " +
        client.getInetAddress() +
                      "connected."); }
```

Übertragung von Daten über Bytes



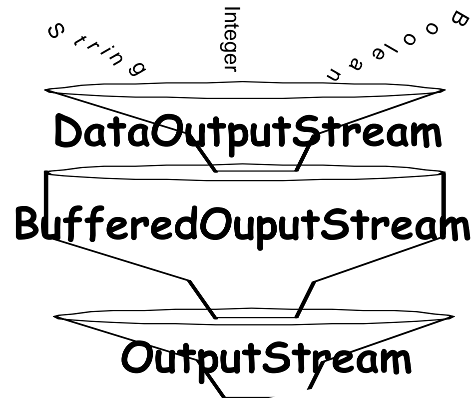
Senden:

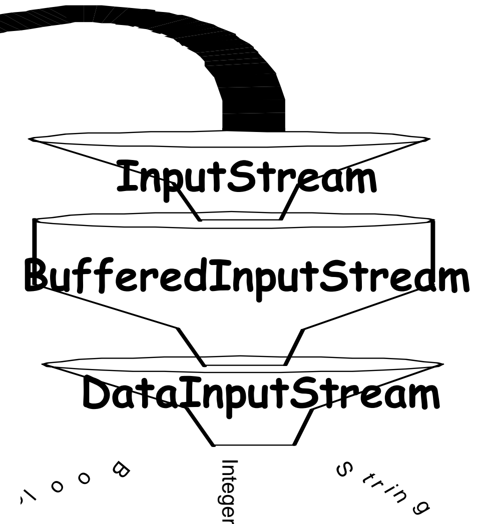
```
Socket socket;
   // muss inititialisiert werden
OutputStream out =
   socket.getOutputStream();
Object obj = new Object ();
byte b[] = obj.getBytes();
   /* getBytes() muss vom Objekt
      implementiert werden */
out.write(b);
```

Empfangen:

```
Socket socket;
   // muss inititialisiert werden
InputStream in =
   socket.getInputStream();
byte b[] = new byte[100];
   // 100 muss groß genug sein
int num = in.read(b);
Object obj = new Object(b);
   /* new Object(byte[]) muss vom
   Objekt implementiert werden */
```

Übertragung von Daten über String-Filter





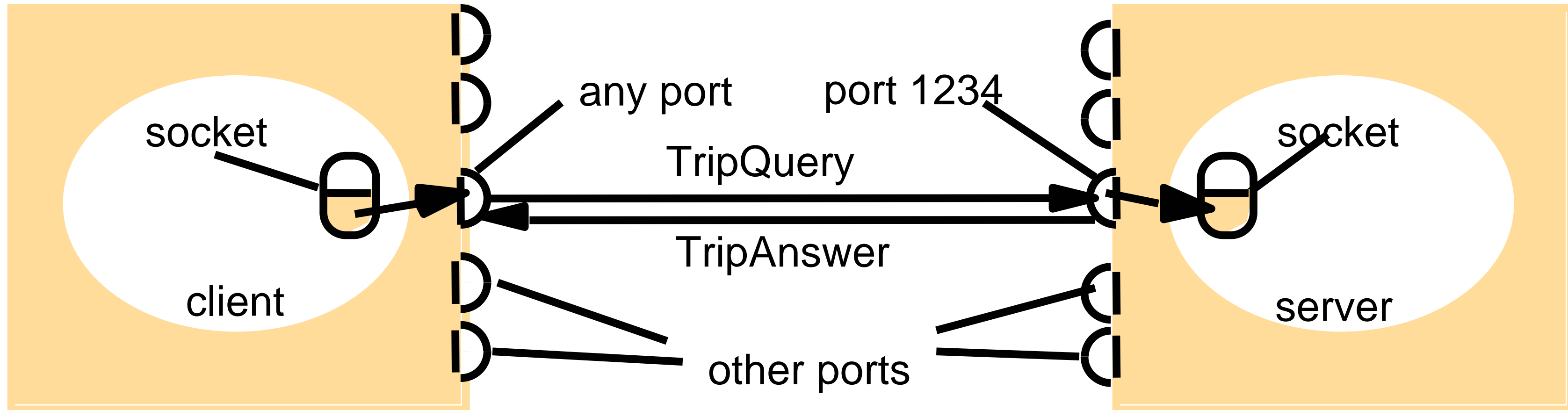
Senden:

```
Socket socket;
   // muss inititialisiert werden
DataOutputStream out = new
   DataOutputStream(new
   BufferedOutputStream(
   socket.getOutputStream()));
Object obj = new Object;
out.writeUTF(obj.toString());
out.flush();
```

Empfangen:

```
Socket socket;
   // muss inititialisiert werden
DataInputStream in = new
   DataInputStream(new
   BufferedInputStream(
   socket.getInputStream()));
String str = in.readUTF());
Object obj = new Object (str)
   /* new Object(String) muss vom
   Objekt implementiert werden */
```

Beispiel TripServer aus dem Touristeninformationssystem



Offene Fragen:

Wie verknüpft man in Client und Server Frage und Antwort ?

➔ durch Entfernte Aufrufe

Wie verhindert man die Blockade von Client und Server ?

➔ durch Nebenläufigkeitstechniken

Nebenläufigkeitstechniken

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(); }
    }
```

Erben von Thread

Konstruktor (beliebig)

Ausführungsmethode
(muss genau so heißen)

Erzeugen des Threads

Starten des Threads

Threaderzeugung in Java (2. Variante, für Mehrfachvererbung)

```java
class ExampleRunnable extends SomeClass
                    implements Runnable {

    ExampleRunnable(int param) { ... }

    public void run() { ... }

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

Zuordnung zum Interface Runnable

Konstruktor (beliebig)

Ausführungsmethode (muss genau so heißen)

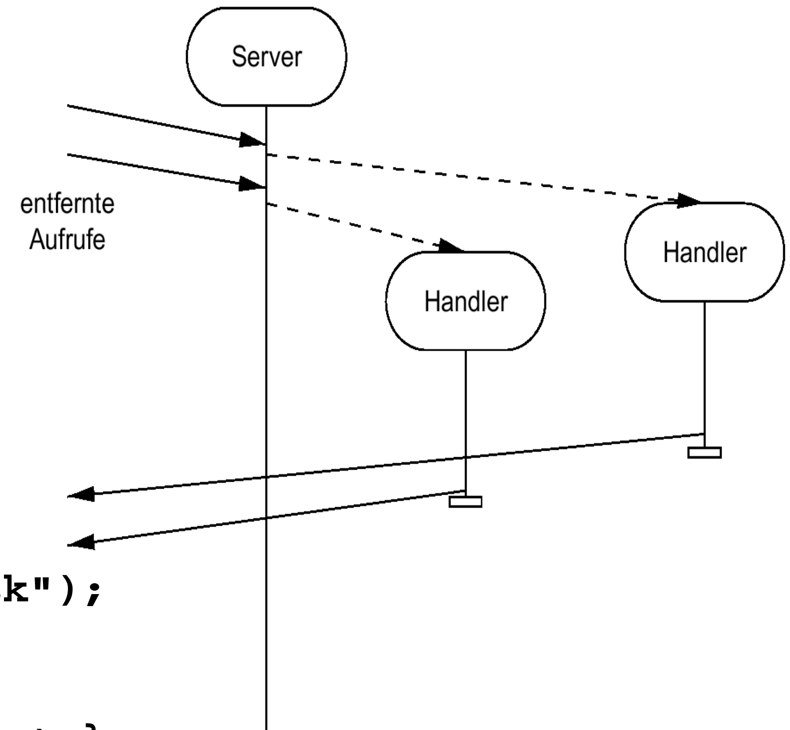
Erzeugen des „Threads“

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 (...) { ... } } } }
```

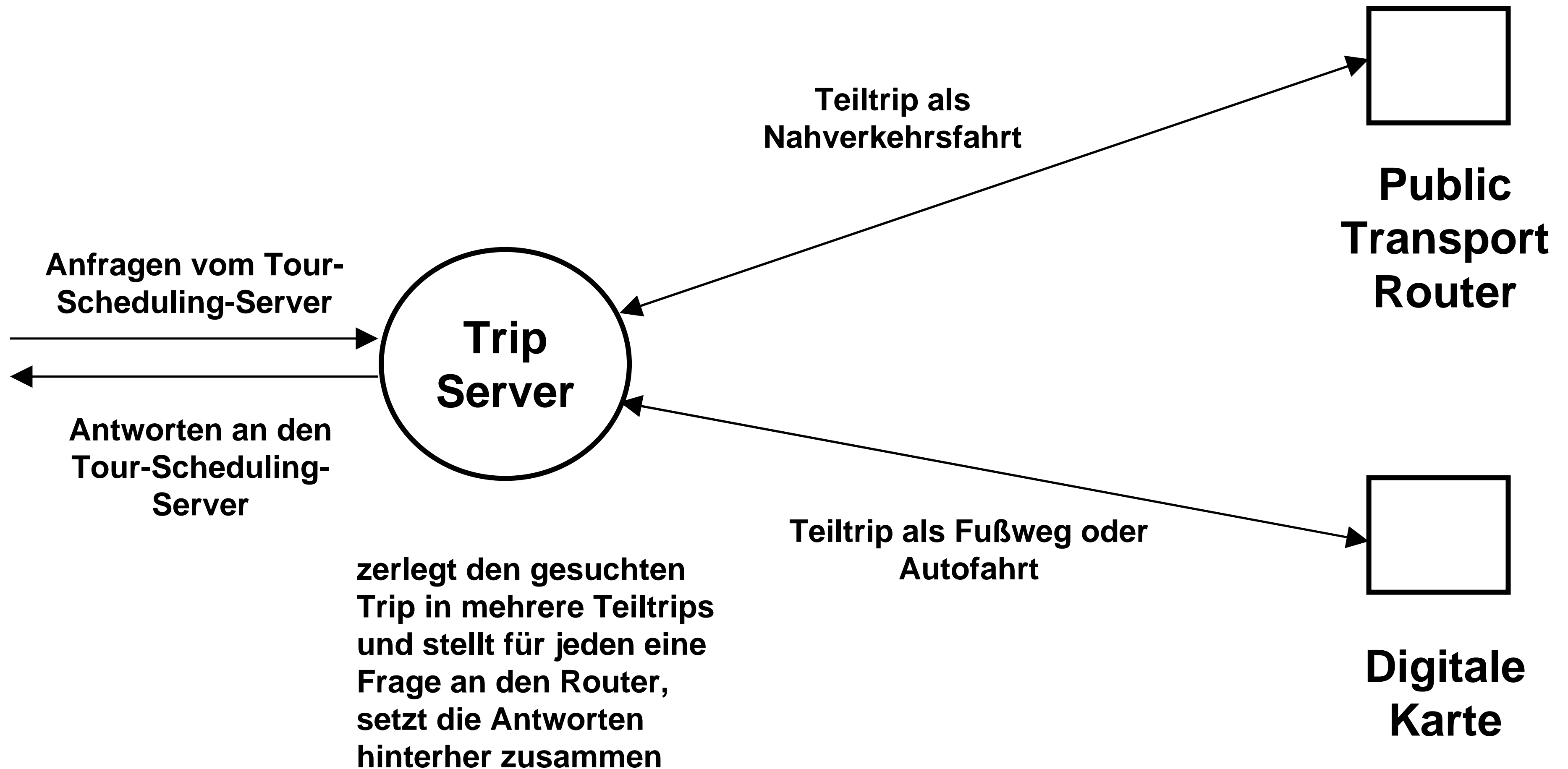
Auftragsbearbeitung:

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

Beispiel TripServer aus dem Touristeninformationssystem

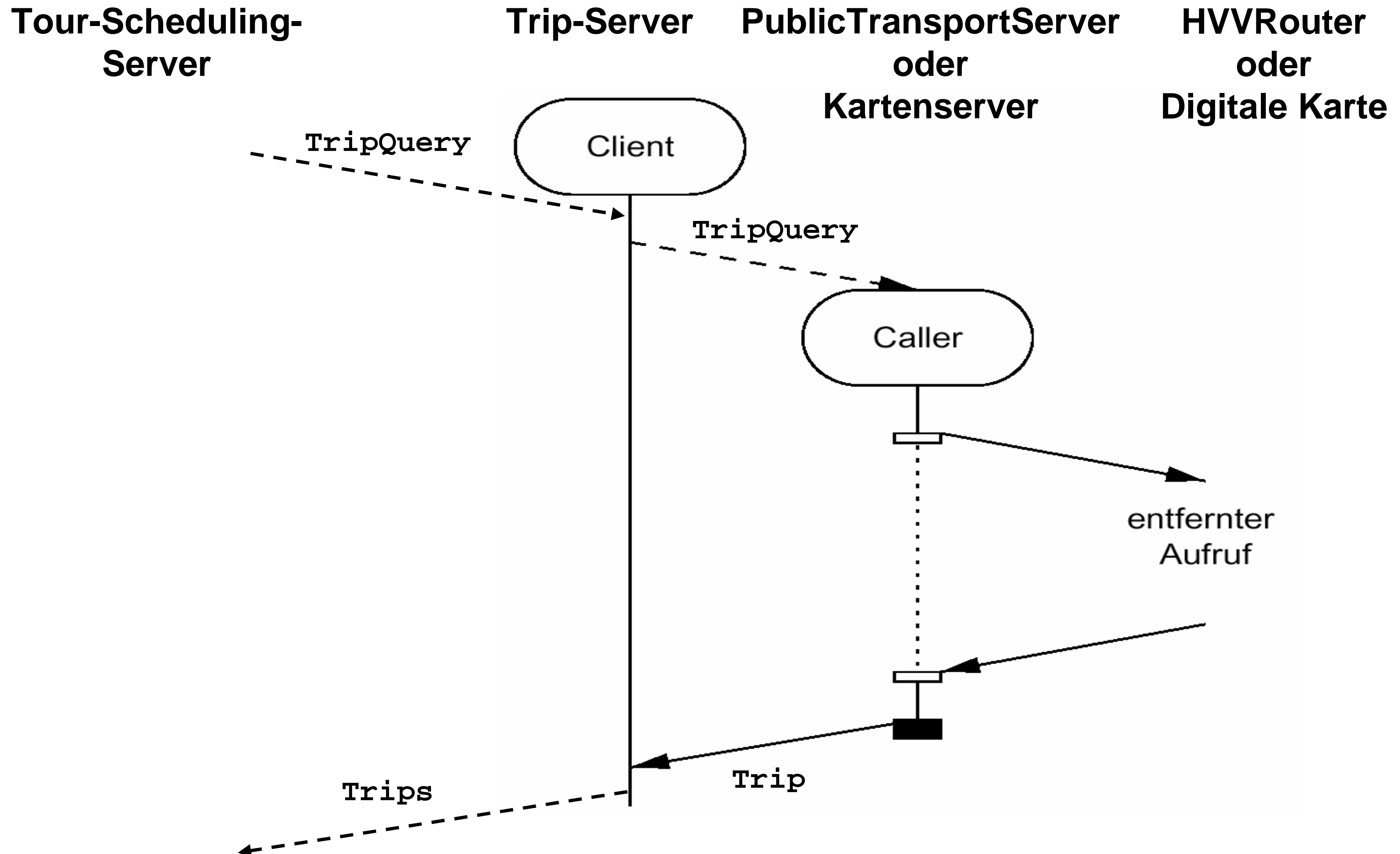
*Die im Folgenden vorgestellte Fallstudie entstand in Zusammenarbeit mit **Sven Bischoff**, Dipl.-Inform. (FH), ehemals Werkstudent und Diplomand bei der DaimlerChrysler AG, Bereich Telematics Research*

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 */
```

Beim nächsten Mal:
Polling-Verfahren in der Fallstudie
Risiken von Nebenläufigkeit und Vorsorgemaßnahmen
Entfernte Aufrufe