

Verteilte Systeme - Übung

Tobias Distler, Michael Gernoth

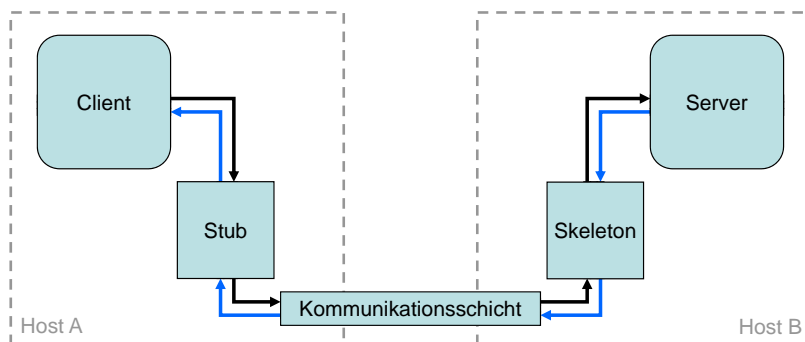
Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)

www4.informatik.uni-erlangen.de

Sommersemester 2009

VS-Übung

- ▶ Entwicklung eines eigenen Fernaufrufsystems
- ▶ Orientierung an Java-RMI



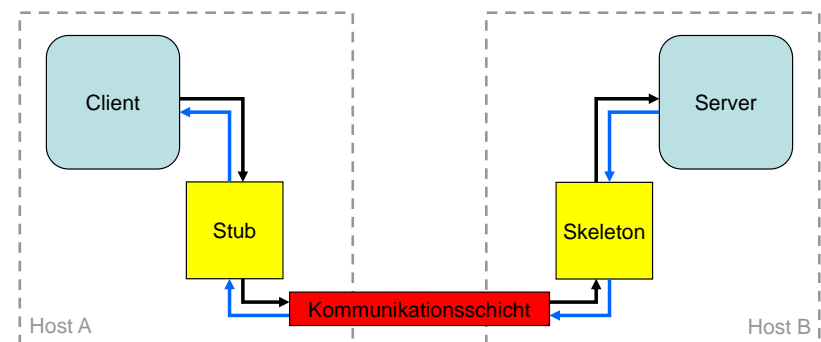
Überblick

RPC-Semantiken

Fehler bei Fernaufrufen
Fehlertolerante Fernaufrufe
Übungsaufgabe 5

Übungsaufgabe 5

- ▶ Bereitstellung von Fehlertoleranz
- ▶ Simulation von Kommunikationsfehlern



RPC-Semantiken

Fehler bei Fernaufrufen

Fehlertolerante Fernaufrufe
Übungsaufgabe 5

Fehlerquellen

Was kann da schon schief gehen?

- ▶ Rechner (Client und/oder Server)
 - ▶ Prozess-, Programm-, Rechnerabsturz
 - ▶ Verzögerungen (z.B. aufgrund von Überlast)
- ▶ Kommunikation
 - ▶ Nachrichten (Anfrage und/oder Antwort)
 - ▶ -reihenfolgeänderung
 - ▶ -korrumpierung
 - ▶ -vervielfachung
 - ▶ -verlust
 - ▶ Verbindungs
 - ▶ -verlangsamung
 - ▶ -abbruch

→ Im Fernaufruf begründete Fehler bzw. Ausnahmesituationen

Fehlerquellen

Was kann da schon schief gehen?

- ▶ Anwendung (siehe Tafelübung zu Aufgabe 4)
 - ▶ Beispiele
 - ▶ Falsche Eingaben
 - ▶ Programmierfehler in der Anwendung
 - ▶ ...
 - ▶ Keine im Fernaufruf begründeten Fehler
 - ▶ Transparente Signalisierung

→ Aus Sicht des Fernaufrufsystems: „reguläres Verhalten“

→ Keine Fehlerbehandlung im Fernaufrufsystem

Einordnung

- ▶ Rechner
 - ▶ Lokaler Methodenaufruf
 - ▶ Aufrufer und Aufgerufener in gleichem Maße betroffen
 - ▶ Im Fehlerfall sind beide weg bzw. langsam
 - ▶ Fernaufruf
 - ▶ Aufrufer und Aufgerufener können unabhängig ausfallen
 - ▶ Im Fehlerfall ist evtl. nur einer betroffen
- ▶ Kommunikation
 - ▶ Lokaler Methodenaufruf
 - ▶ Keine Netzwerkkommunikation
 - ▶ Fehlerart nicht relevant
 - ▶ Fernaufruf
 - ▶ Temporäre oder sogar dauerhafte Fehler möglich
 - ▶ Nicht alle Fehler lassen sich im Fernaufrufsystem tolerieren

Konsequenz

**Das komplexere Fehlermodell macht es unmöglich
Fernaufrufe vollständig transparent zu realisieren!**

Fehlerbehandlung auf Fernaufrufsystemebene

- ▶ Fehlertolerierung
 - ▶ Transparente Mechanismen
 - ▶ RPC-Semantiken (später mehr)
 - ▶ Replikation
 - ▶ ...
 - ▶ Beliebig komplex
 - ▶ Geringer Aufwand → Tolerierung von wenigen (oder nur bestimmten) Fehlern
 - ▶ Hoher Aufwand → Tolerierung von vielen Fehlern
 - ▶ **Trotzdem: Nicht alle Fehler lassen sich tolerieren**
- ▶ Fehlersignalisierung
 - ▶ Benachrichtigung an den Benutzer des Fernaufrufsystems
 - ▶ Notwendig wenn Fehler nicht toleriert werden konnten
 - ▶ Benutzer des Fernaufrufsystems muss darauf vorbereitet sein
 - ▶ **Verletzung der Transparenzeigenschaften**

Fehlererkennung bei Fernaufrufen

- ▶ Probleme
 - ▶ Keine definitive Fehlererkennung (Liegt überhaupt ein Fehler vor?)
 - ▶ Keine exakte Fehlerlokalisierung (Wo liegt der Fehler?)
 - ▶ Beispiel
 - ▶ Szenario: Client erhält keine Antwort auf seine Anfrage
 - ▶ Mögliche Gründe
 - ▶ Anfrage ging verloren
 - ▶ Antwort ging verloren
 - ▶ Server ausgefallen
 - ▶ Server überlastet
 - ▶ Netzwerk überlastet
 - ▶ ...
 - ▶ Konsequenz: Mindestens einer der beiden Fernaufruf-Teilnehmer kann nicht erkennen, ob (und wenn ja wo) ein Fehler vorliegt
- Eine präzise Fehlererkennung ist in verteilten Systemen (im Allgemeinen) nicht möglich!

Signalisierung von Fernaufruffehlern in Java-RMI

Exception-Klasse `java.rmi.RemoteException`

- ▶ Allgemein
 - ▶ Oberklasse für alle Fernaufruf-Ausnahmesituationen
 - ▶ Muss von jeder Methode einer `Remote`-Schnittstelle geworfen werden → **Verletzung der Zugriffstransparenz**
 - ▶ Beispiel

```
public interface RemoteInterface extends Remote {
    public void foo() throws RemoteException;
}
```

- ▶ Unterklassen
 - ▶ `ConnectException`: Verbindungsaufbau fehlgeschlagen
 - ▶ `NoSuchObjectException`: Remote-Objekt nicht (mehr) verfügbar
 - ▶ `ServerError`: Auspacken der Anfrage, Ausführung der Methode oder Einpacken der Antwort fehlgeschlagen
 - ▶ `UnknownHostException`: Remote-Host nicht bekannt
 - ▶ ...

RPC-Semantiken

Fehler bei Fernaufrufen
Fehlertolerante Fernaufrufe
Übungsaufgabe 5

Was beinhaltet Fehlertoleranz für Fernaufrufe?

- ▶ Probleme
 - ▶ Unpräzise Fehlererkennung kann zu inkonsistenten Sichtweisen führen, z.B. bei einer verlorenen Antwortnachricht:
 - ▶ Client geht von einem Fehler aus, da er keine Antwort erhalten hat
 - ▶ Server befindet sich im Normalzustand, da er die Antwort gesendet hat
 - ▶ Wiederherstellung einer konsistenten Sichtweise erfordert weitere Kommunikation zwischen Client und Server
 - ▶ Diese Nachrichten können ebenfalls Fehlern unterliegen
 - ▶ Erneutes Senden und Ausführen einer Anfrage kann zu unerwünschten Zuständen führen
- ▶ Allgemeine Hilfsmittel
 - ▶ Duplikaterkennung
 - ▶ Idempotente Operationen
 - ▶ Rollbacks (→ Transaktionen)
 - ▶ ...

Kombinierter Ansatz

- ▶ Fehlertolerantes Kommunikationsprotokoll
 - ▶ Beispiel: TCP (statt UDP)
 - ▶ Tolerierung von Nachrichten
 - ▶ -reihenfolgeänderung
 - ▶ -korrumpierung
 - ▶ -vervielfachung
 - ▶ -verlust
- ▶ Aufrufsemantiken
 - ▶ Wiederherstellung konsistenter Sichtweisen von Client und Server
 - ▶ Tolerierung von Verzögerungen (Rechner und/oder Netzwerk)
- ▶ Hinweise
 - ▶ Alle durch das Kommunikationsprotokoll tolerierte Fehler sind ersatzweise durch eine entsprechende Aufrufsemantik tolerierbar
 - ▶ Die optimale Kombination von Kommunikationsprotokoll und Aufrufsemantik hängt vom Einzelfall ab

RPC-Semantiken

- ▶ Bemerkung
 - ▶ Im Folgenden wird die Tolerierung von **Kommunikationsfehlern** betrachtet, Rechnerausfälle werden ausgeklammert
 - ▶ Die Tolerierung von Rechnerausfällen erfordert Mechanismen zum Wiederanlaufen (z.B. Zustandwiederherstellung)
- ▶ Aus der Vorlesung bekannt
 - ▶ Maybe
 - ▶ At-Least-Once
 - ▶ At-Most-Once
 - ▶ Last-Of-Many
- ▶ Unterschiede
 - ▶ Mehrmaliges Senden von Anfragen
 - ▶ Aktualität der Antworten
 - ▶ Anzahl der Ausführungen → *Idempotente Operationen?*
 - ▶ Antwortspeicherung → *Wie lange wird eine Antwort aufgehoben?*

Idempotenz

- ▶ Idempotente Funktionen (Mathematik)
 - ▶ Definition

$$f(x) = f(f(x))$$
 - ▶ Beispiele
 - ▶ $f(x) = c$ (konstante Funktion)
 - ▶ $f(x) = x \cdot 1$ (Multiplikation mit 1)
 - ▶ $f(x) = \frac{x}{1}$ (Division durch 1)
- ▶ Idempotente Operationen (Informatik)
 - ▶ Eigenschaften
 - ▶ Mehrfache Ausführung erzeugt stets den selben Rückgabewert
 - ▶ Mehrfache Ausführung hinterlässt die Anwendung auf dem Server in stets dem selben Zustand
 - ▶ Bankautomat-Beispiel
 - ▶ Verwendung absoluter Beträge
 - ▶ **Achtung: nicht-triviale Implementierung, da zusätzliche Synchronisation und/oder Ausnahmebehandlung nötig**

Antwortspeicherung

- ▶ (Wiederkehrendes) Problem
 - ▶ Server stellt eigene Ressourcen (Antwort-Cache) für (fehlertolerante) Fernaufrufe von Clients bereit
 - ▶ Mit jedem neuen Fernaufruf werden zusätzliche Ressourcen belegt
 - Wann können diese Ressourcen freigegeben, d.h. gespeicherte Antworten verworfen werden?
- ▶ Lösungsansätze (Kombinationen möglich bzw. nötig)
 - ▶ Explizit
 - ▶ Benachrichtigung durch Client oder Nachfrage vom Server
 - ▶ **Problem: Nicht alle Clients können oder wollen sich daran halten**
 - ▶ Implizit
 - ▶ Bei neuem Fernaufruf eines Client wird die alte Antwort gelöscht
 - ▶ **Problem: Letzter Fernaufruf eines Client**
 - ▶ Timeout
 - ▶ Antwortlöschung nach Ablauf eines fernaufrufspezifischen Timeout
 - ▶ **Als Rückfallposition immer nötig**

At- $\{\text{Least, Most}\}$ -Once

- ▶ At-Least-Once (z.B. SUN-RPC)
 - ▶ Funktionsweise
 - ▶ Client wiederholt Anfrage, falls Antwort ausbleibt
 - ▶ Client akzeptiert die erste Antwort, die ihn erreicht
 - ▶ Eigenschaften
 - ▶ Anfragen werden evtl. mehrfach ausgeführt
 - ▶ Client verwendet evtl. veraltete Antwort
- ▶ At-Most-Once (z.B. Java-RMI)
 - ▶ Funktionsweise
 - ▶ Client wiederholt Anfrage, falls Antwort ausbleibt
 - ▶ Server speichert Antworten
 - ▶ Server sendet bei Anfragewiederholungen gespeicherte Antworten
 - ▶ Eigenschaften
 - ▶ Anfragen werden höchstens einmal ausgeführt
 - ▶ Speichern von Antworten erforderlich

Last-Of-Many

- ▶ Funktionsweise
 - ▶ Client wiederholt Anfrage, falls Antwort ausbleibt
 - ▶ Client akzeptiert nur Antwort auf seine aktuellste Anfrage
- ▶ Implementierung
 - ▶ Fernaufruf muss eindeutig identifizierbar sein
 - ▶ Client
 - ▶ Remote-Objekt
 - ▶ Remote-Methode
 - ▶ Aufrufzähler
 - ▶ Jede Fernaufrufnachricht muss eindeutig identifizierbar sein
 - ▶ Anfragezähler
 - ▶ Zuordnung: Antwort zu Anfrage
- ▶ Eigenschaften
 - ▶ Keine Antwortspeicherung nötig
 - ▶ Anfragen werden evtl. mehrfach ausgeführt

Ein paar Worte zu *Exactly-Once* im lokalen Fall...

- ▶ Ideale Semantik: Beschreibt den lokalen, fehlerfreien Fall
 - ▶ Aufrufer tätigt *genau einen* Aufruf
 - ▶ *Genau eine* Methodenausführung
 - ▶ Auferufener liefert *genau eine* Antwort
- ▶ Entscheidende Fragestellung bei Rechnerausfall
Wurde die Methode vor dem Rechnerausfall noch ausgeführt?
- ▶ Idee: Ausführungs-Log (nur 1. & 2. \leadsto At-Most-Once, nur 2. & 3. \leadsto At-Least-Once)
 1. Vor Ausführung: Anfrage in Log schreiben
 2. Methodenausführung
 3. Nach Ausführung: Ausführungsbestätigung in Log schreiben

→ **Reicht nicht aus, da Rechnerausfall zwischen 1. und 2., während 2. oder zwischen 2. und 3. stattgefunden haben könnte**
- Die Realisierung von *Exactly-Once* ist (bereits) im lokalen (Fehler-)Fall nicht trivial!

Ein paar Worte zu *Exactly-Once* im lokalen Fall...

Abhilfe: Transaktionale Systeme

- ▶ Funktionsweise
 - ▶ Jede Operation ist eine Transaktion
 - ▶ Transaktion muss explizit oder implizit gestartet (*start*) werden
 - ▶ Transaktion muss explizit abgeschlossen (*commit*) werden
 - ▶ Falls Transaktion abbricht: *rollback*
 - ▶ Semantik
 - ▶ Annäherung an *Exactly-Once* → *Last-One*-Semantik
 - ▶ Achtung: Für seiteneffektfreie Operationen manchmal auch als *Exactly-Once* bezeichnet
- Je nach Betrachtungsweise erlauben Transaktionen die Implementierung von *Exactly-Once* (oder auch nicht)

RPC-Semantiken

Fehler bei Fernaufrufen
Fehlertolerante Fernaufrufe
Übungsaufgabe 5

Ein paar Worte zu *Exactly-Once* im verteilten Fall...

- ▶ Zusätzliches Problem
 - ▶ Annahme: *Exactly-Once/Last-One* am Server implementiert
 - ▶ Trotzdem: Unschärfe am Client
 - ▶ Solange der Client-Stub keine Bestätigung vom Server hat, dass die Operation erfolgreich ausgeführt wurde, kann er seinem Aufrufer kein OK geben
 - ▶ Diese Bestätigung kommt vielleicht nie... (→ Netzwerkpartition)
 - ▶ Konsequenz
 - ▶ Permanente Kommunikationsfehler → Weder *Exactly-Once* noch *Last-One* lassen sich in verteilten Systemen realisieren
 - ▶ Temporäre Kommunikationsfehler → Kommunikationsfehler lassen sich durch entsprechende RPC-Semantiken tolerieren, *Exactly-Once/Last-One* analog zum lokalen Fall realisierbar
- ⇒ Ob die Implementierung von *Exactly-Once* möglich ist, hängt sowohl im lokalen als auch im verteilten Fall von der (leider nicht einheitlichen) Definition ab!

Übersicht

Teilaufgaben

- ▶ Implementierung von *Last-of-Many*
- ▶ Implementierung von *At-Most-Once*
- ▶ Sabotage des Kommunikationssystems

Implementierung der RPC-Semantiken

- ▶ *Last-of-Many*
 - ▶ Fernaufruf-IDs
 - ▶ Sequenznummern
 - ▶ Timeouts
- ▶ *At-Most-Once*
 - ▶ Antwort-Cache
 - ▶ Erweiterte Synchronisierung
- ▶ Hinweis
 - Die zum Einsatz kommende RPC-Semantik
 - ▶ *Maybe*
 - ▶ *Last-of-Many*
 - ▶ *At-Most-Once*
 - soll konfigurierbar sein!

Mögliche Timeout-Behandlung in Java

java.util.Timer

Klasse java.util.Timer

- ▶ Allgemein
 - ▶ Einfacher Scheduler
 - ▶ Unterstützung ein- sowie mehrmaliger Task-Ausführung
 - ▶ Task-Objekte: Instanzen von TimerTask-Unterklassen
- ▶ Methoden
 - ▶ Einmalig auszuführenden Task (→ Timeout-Handler) aufsetzen


```
void schedule(TimerTask task, long delay)
```
 - ▶ Timer beenden


```
void cancel()
```
 - ▶ ...

Mögliche Timeout-Behandlung in Java

java.util.TimerTask

Klasse java.util.TimerTask

- ▶ Allgemein
 - ▶ Basisklasse für von Timer eingeplante Tasks
 - ▶ Auszuführenden Task-Code in Unterklassen implementieren
- ▶ Methoden
 - ▶ Task ausführen (→ Timeout behandeln)


```
abstract void run()
```
 - ▶ Task abbrechen (→ Timeout deaktivieren)


```
boolean cancel()
```
 - ▶ ...

Mögliche Timeout-Behandlung in Java

Beispiel

```
public class TimerExample {
    public static void main(String[] args) {
        Timer timer = new Timer();
        TimerTask handler = new TimeoutHandler();

        // Timeout auf 5 Sekunden setzen
        timer.schedule(handler, 5000);

        // Zu ueberwachenden Code ausfuehren
        [...]

        // Timeout deaktivieren
        handler.cancel();
    }
}

class TimeoutHandler extends TimerTask {
    public void run() {
        System.err.println("ALARM!");
    }
}
```

Sabotage des Kommunikationssystems

- ▶ Simulation von Kommunikationsfehlern
 - ▶ Verlust von Nachrichten
 - ▶ Verzögerung einzelner Nachrichten
 - ▶ Vervielfachung von Nachrichten
- ▶ Tests
 - ▶ Variation der Fehlerintensität
 - ▶ Kombination verschiedener Fehlerarten
- ▶ Mögliche Implementierung
 - ▶ Fehlerhafte `VSConnection` → `VSBuggyConnection`
 - ▶ Überschreiben von
 - ▶ `sendMessage(VSMessage msg)` oder
 - ▶ `receiveMessage()`