Verteilte Systeme - 6. Übung

Tobias Distler, Michael Gernoth, Reinhard Tartler

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
www4.informatik.uni-erlangen.de

Sommersemester 2008

Übungsaufgabe 5

- RPC-Aufrufsemantiken
  - Fehlermodell
- Implementierung einer *Last-of-Many* Semantik
- Lösungshinweise
  - Timeout

Transparenz beim Fernaufruf

Rückblick

- Alles bisher betrachteten Probleme drehten sich um die Parameterübergabe (lokaler vs. verteilter Fall)
  - Call-by-Value
  - Call-by-Value/Result
  - Call-by-Reference
  - Probleme mit Gültigkeitsbereichen, lokalen Repräsentationen, Kommunikationskosten, ...
- Bisher vorausgesetzt:
  - Alles läuft fehlerfrei ab!

Fehler & RPC (lokaler Fall)

- Ich rufe eine Methode auf, diese wird genau einmal ausgeführt, und kehrt dann zurück

→ **Exactly-Once-Semantik**

- Idealfall.
- Im Falle eines Ausfalls sind sowohl Aufrufer und Aufgerufener abgestürtzt.

Fehler & RPC (entfernter Fall)

- Viel mehr Fehlermöglichkeiten:
  - Im Kommunikationssystem: Nachrichten gehen verloren, kommen in geänderter Reihenfolge an, werden dupliziert, werden verändert
  - Server-Rechner fällt aus
  - Client-Rechner fällt aus
- Server und Client können Unabhängig voneinander ausfallen.

→ Im Falle eines Teilausfalls muss der Überlebende diesen Fall *sinnvoll* behandeln.

→ Siehe auch die Unterklassen von `java.rmi.RemoteException`

⇒ **Exactly-Once-Semantik** *nicht erreichbar!*

Transaktionale Systeme

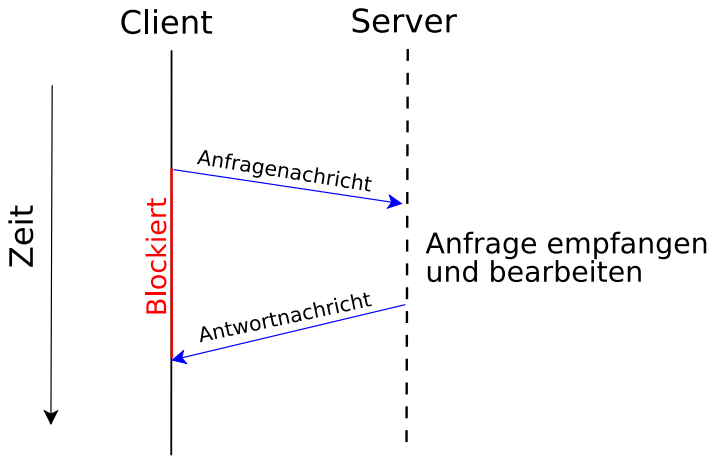
- Ganz dicker Hammer
- Fehlertolerante Systeme, die Ausfälle durch Zustandssicherungen verkraften können
- Jede Operation wird in *Transaktionen* eingepackt:
  - Muss explizit oder implizit gestartet (*start*) werden.
  - Muss explizit abgeschlossen (*commit*) werden.

→ *All-Or-Nothing* Semantik.

⇒ Letzendlich nur innerhalb der Anwendung richtig implementierbar:
  - Ein Kommunikations- bzw. Fernaufrufsystem kann im allgemeinen nicht das Verhalten der Anwendung vorraussagen.
  - Im besten Fall können Heuristiken angewendet werden.

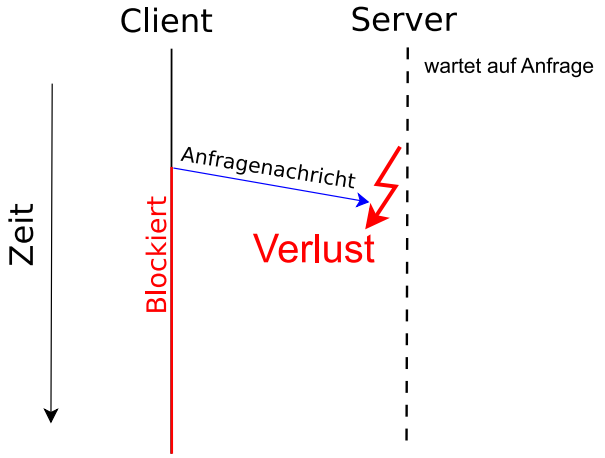
Beispiel: Trivialer RPC & Fehler

einfachster RPC ist ein primitives request/reply Protokoll:



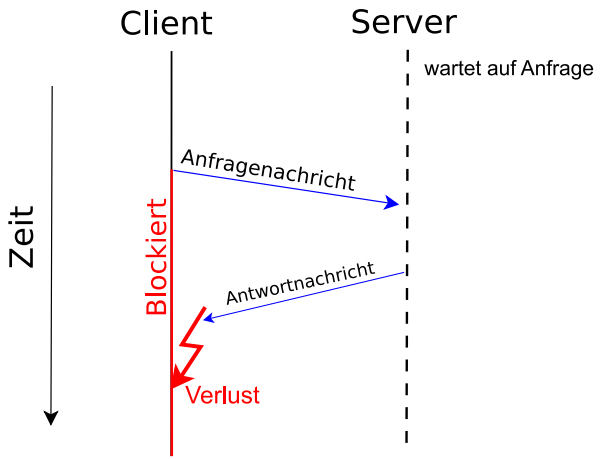
Beispiel: Trivialer RPC & Fehler

Verlust einer Anfragenachricht



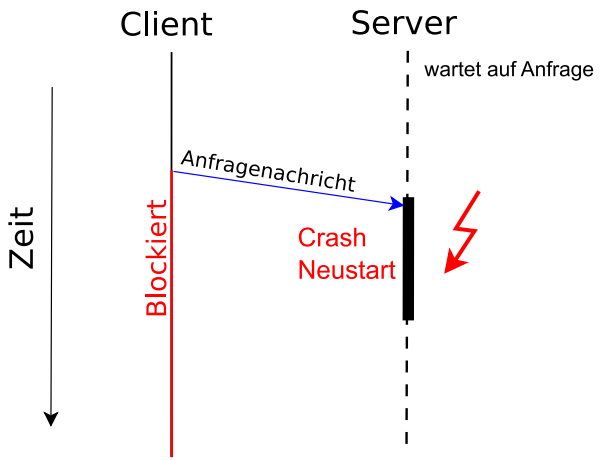
Beispiel: Trivialer RPC & Fehler

Verlust einer Antwortnachricht



Beispiel: Trivialer RPC & Fehler

Fehler während des Ausführens der Methode



Was kann man tun?

- Erster Versuch: Zuverlässiges Kommunikationsprotokoll verwenden (z.B. TCP statt UDP)
- Vorteil: Problem der Nachrichtenverluste, -verdoppelungen, -veränderungen, Reihenfolge-Änderung gelöst...
- Wirklich?

Nachteile/Probleme?

- Overhead!
  - Verbindungsaufbau
  - Request und Reply mit ACK
  - Verbindungsabbau

  $\rightarrow$ führt zu Performanceverlust bei Fehlerfreiheit
- löst nicht das Problem von verzögerten Nachrichten
- Was tun z.B. bei *Connection Closed*, oder wenn Server vorübergehend ausgefallen ist?

  $\Rightarrow$ Zusätzliche Fehlerbehandlung trotzdem notwendig!
- Falls TCP nicht automatisch vorhanden (z.B. kleine eingebettete Systeme): Implementierung von TCP ziemlich aufwendig!

Was kann man tun?

- Zweiter Versuch: Eigene Fehlerbehandlung bei Verlust von Nachrichten
- Anfrage nach einem Timeout erneut schicken
  - bei Verlust der Anfrage: ok
  - bei Verlust der Antwort wird die Methode doppelt ausgeführt
    - Was ist mit evtl. Seiteneffekten?
    - Idempotente Methode?
  - Vielleicht war auch kein Fehler aufgetreten, die Antwort wurde einfach noch nicht geschickt (längere Ausführungsdauer der Methode)?

RPC-Aufruf genauer betrachtet

- Abhängig von der gewünschten Aufrufsemantik
- Abkehr vom Idealbild einer *Exactly-Once-Semantik*
  - Mögliche Varianten sollten aus Vorlesung bereits bestens bekannt sein!
    - Best-Effort
    - At-Least-Once
    - At-Most-Once
    - Last-Of-Many
    - Last-One (Last-Of-Many mit Orphan-Behandlung, siehe Vorlesung)
- Wie implementiert man diese?

RPC-Aufruf genauer betrachtet

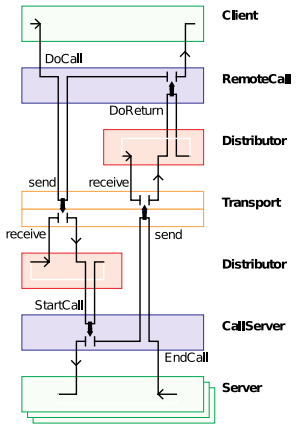
Implementierung von...

- best effort ...
- at-least-once
  - Timeout
  - Wiederhohlung
  - ....
- at-most-once
  - Aufrufkennung
  - Ergebnisspeicherung
  - ...
- last-of-many
  - Eindeutige IDs (Wiederhohlungskennung)
  - ...
  - siehe folgende Seiten

Last-of-Many-Semantik (Nelson-RPC)

(1) Request senden, mit [ID, SeqNo]; Timeout starten

(2) Request empfangen, [ID, SeqNo] merken Methode ausführen



(3) Methode ist fertig [ID, Seqno löschen und mit Ergebnis senden

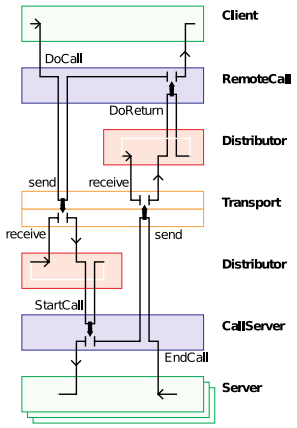
(4) Antwort Empfangen [ID, Seqno] stimmt Timeout löschen Ergebnis zu Client

Last-of-Many-Semantik (Nelson-RPC)

(1) Timeout läuft ab:
Request nochmal senden. gleiche ID, neue Seqno
Timer neu starten

(2) Request kommt an
Fall a)
Methode noch in Bearbeitung
[ID, neue Seqno merken]

Fall b)
ansonsten:
Methode (u.U erneut ausführen



(3) Antwort kommt an

Falls alte Seqno:
Antwort verwefen!

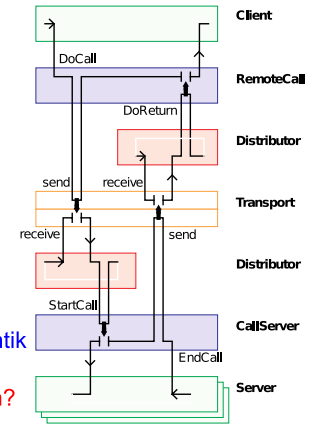
Last-of-Many-Semantik (Nelson-RPC)

Weitere Optimierung:

Server merkt sich bereits gesendete Antwort

Bei erneutem Request wird ohne Methodenausführung die Antwort erneut gesendet

"fast" exactly-once-Semantik
Problem:
Wie lange Antwort merken?



Wie kann man die *Nelson-RPC* optimieren?

Verhinderung der Mehrfach-Ausführung

- eindeutige IDs pro Anfrage; Wiederholung mit gleicher ID. So kann der Server kann feststellen, dass diese Anfrage schon bearbeitet wurde.
- Was soll der Server tun, wenn er eine Anfrage erneut bekommt?
  - Falls Antwort noch nicht gesendet wurde (alte Anfrage noch in Bearbeitung), kann er die Anfrage verwerfen.
  - Ansonsten: Antwortnachrichten speichern und bei wiederholter Anfrage erneut versenden.
- Wie lange soll der Server eine Antwortnachricht speichern?

Wie lange soll der Server eine Antwortnachricht speichern?

- bis zur nächsten Anfrage
  - Client hat die Antwort sicher, weil er weiter gemacht hat.
  - $\rightarrow$ Was wenn der Client abstürtzt?
  - $\rightarrow$ Was wenn der Client nur sehr langsam weiter arbeitet?
  - $\rightarrow$ Wass wenn der Client *normal* weiterarbeitet, aber keine weitere Anfragen an den server macht?

$\Rightarrow$ zusätzlich: Timeouts im Server
  - Idealerweise *unendlich* lang.
  - Konkrete Zeit vom Anwendungsfall abhängig.

Timeouts in Java

- Zur Implementierung des Timeouts im Client gibt es grundsätzlich zwei Ansätze:
  - Timeouts in Sockets
  - `ReentrantLocks` und `Conditions`.
  - TimerThreads
- Timeouts in Sockets
  - `java.net.Socket` bietet die Methode `setSoTimeout` an.
  - Im Falle eine timeouts wirft der Socket eine `java.net.SocketTimeoutException`.
  - *Für unsere Aufgabe NICHT geeignet!*
- `ReentrantLock` und `Condition`
  - Wurde bereits in der 2. Tafelübung vorgestellt
  - Im Prinzip gangbare Lösung
  - Benötigt trotzdem eine Konstruktion mit Hilfsthreads

Die Klasse `java.util.Timer` (und co.)

- Die Klasse `Timer` kann `TimerTasks` zeitversetzt ausführen:
  - periodisch
  - zu einem festem, späterem Zeitpunkt
- `TimerTask` ist eine Abstrakte Basisklasse und implementiert Interface `Runnable`
- `Timer.schedule(TimerTask t, long ms)` initiiert den Aufruf.
- mit `TimerTask.cancel()` kann der Task abgebrochen werden.

Minimales Beispiel

```
import java.util.TimerTask;
import java.util.Task;

class Alarm extends TimerTask {
  public void run() {
    System.out.println("RING RING");
  }
}

public class TimerExample {
  public static void main() {
    Timer t = new Timer();
    TimerTask alarm = new Alarm();

    t.schedule(alarm, 5000);
    if (some_busy_checks()) {
      alarm.cancel();
    }
  }
}
```