I 8. Übung

I.1 Überblick

- RPC-Aufrufsemantiken
  - Fehlermodell
- Aufgabe 6
  - Implementierung einer Last-of-Many-Semantik
  - Lösungshinweise
    - Timeout

2 Fehler & RPC

- Lokaler Fall
  - Ich rufe eine Methode auf, diese wird genau einmal ausgeführt, und kehrt dann zurück: "Exactly-Once-Semantik"
    (Idealfall. Wenn mein Rechner dabei ausfällt, interessieren mich weitere Details nicht...)

I.2 Transparenz beim RPC

1 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!

3 Fehler & RPC

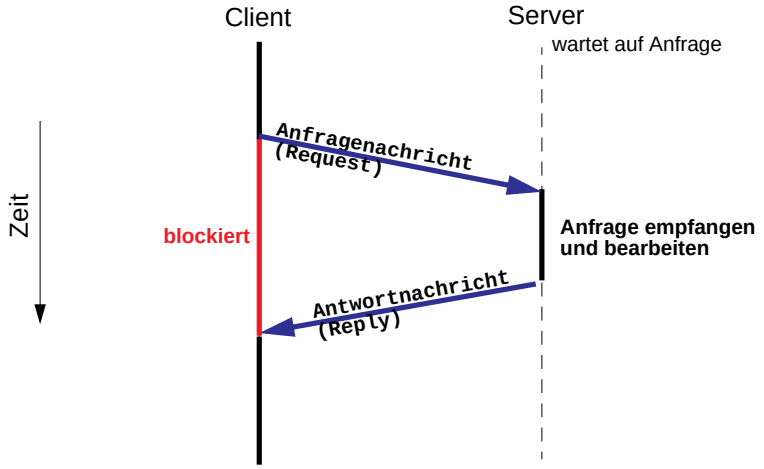
- Lokaler Fall
  - Ich rufe eine Methode auf, diese wird genau einmal ausgeführt, und kehrt dann zurück: "Exactly-Once-Semantik"
    (Idealfall. Wenn mein Rechner dabei ausfällt, interessieren mich weitere Details nicht...)
  - Teilweise auch von Interesse: Fehlertolerante Systeme, die Ausfälle durch Zustandssicherungen verkraften können:
    Transaktionales Verhalten ("All-Or-Nothing-Semantik") angestrebt.

4 Fehler & RPC

- Lokaler Fall
  - Ich rufe eine Methode auf, diese wird genau einmal ausgeführt, und kehrt dann zurück: "Exactly-Once-Semantik"
    (Idealfall. Wenn mein Rechner dabei ausfällt, interessieren mich weitere Details nicht...)
  - Teilweise auch von Interesse: Fehlertolerante Systeme, die Ausfälle durch Zustandssicherungen verkraften können:
    Transaktionales Verhalten ("All-Or-Nothing-Semantik") angestrebt.
- Verteilter 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

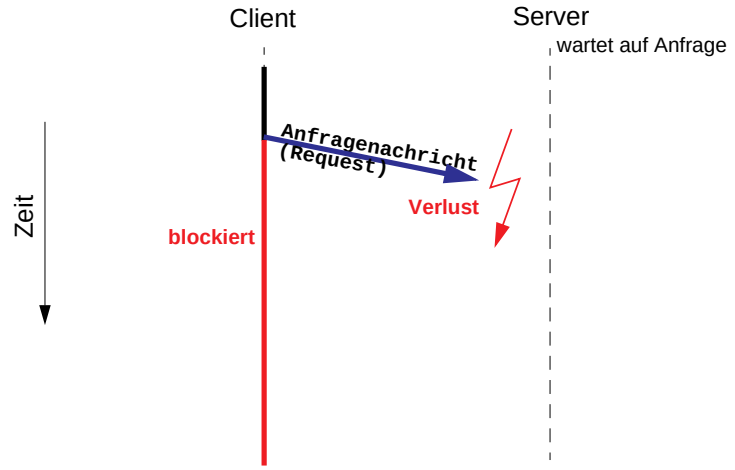
5 Beispiel: Trivialer RPC & Fehler

- einfachster RPC ist ein primitives request/reply-Protokoll:



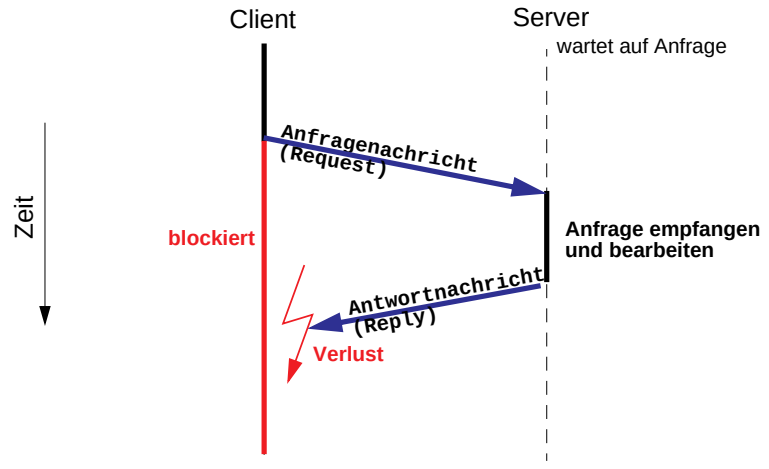
5 Beispiel: Trivialer RPC & Fehler (2)

- Verlust einer Anfragenachricht



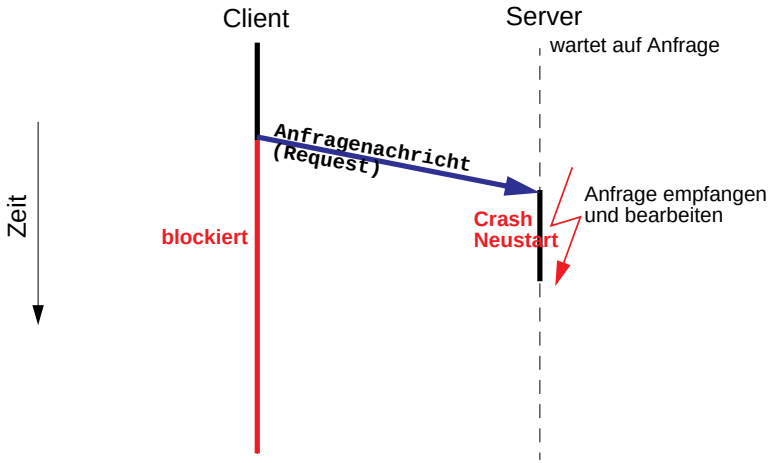
5 Beispiel: Trivialer RPC & Fehler (3)

- Verlust einer Antwortnachricht



5 Beispiel: Trivialer RPC & Fehler (4)

- Fehler während des Ausführens der Methode



6 Was kann man tun?

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

6 Was kann man tun?

- Erster Versuch: Zuverlässiges Kommunikationsprotokoll verwenden (z.B. TCP statt UDP)
- Vorteil:
  Problem der Nachrichtenverluste, -verdoppeltungen, -veränderungen, Reihenfolge-Änderung gelöst...
- Nachteile/Probleme:
  - Overhead (Verbindungsaufbau, Request und Reply mit ACK, Verbindungsabbau) führt zu Performanceverlust bei Fehlerfreiheit
  - Was tun z.B. bei "Connection Closed", oder wenn Server vorübergehend ausgefallen ist?
    Zusätzliche Fehlerbehandlung trotzdem notwendig!
  - Falls TCP nicht automatisch vorhanden (z.B. kleine eingebettet Systeme): Implementierung von TCP relativ aufwendig!

6 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
    - Idempotente Methode?
  - Vielleicht war auch kein Fehler aufgetreten, die Antwort wurde einfach noch nicht geschickt (längere Ausführungsdauer der Methode)?

7 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!

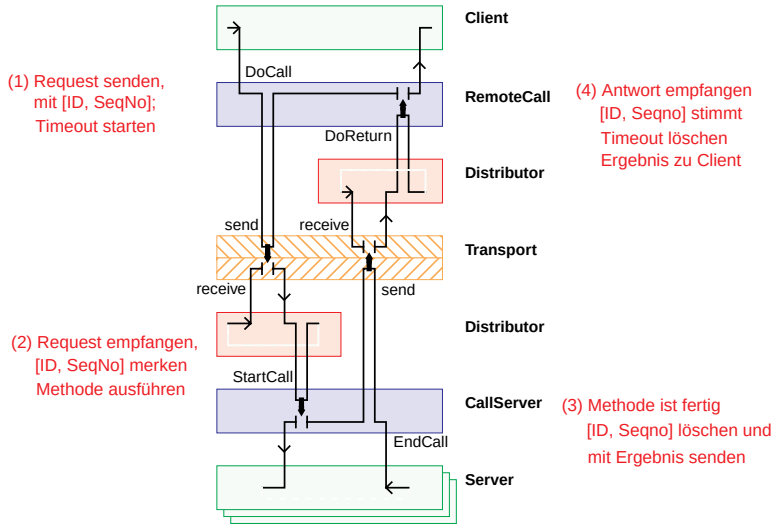
8 RPC-Aufruf genauer betrachtet

- Abhängig von der gewünschten Aufrufsemantik
  - ◆ Abkehr vom Idealbild einer "Exactly-Once-Semantik"
  - ◆ Mögliche Varianten sollten aus Vorlesung bereits besten 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?

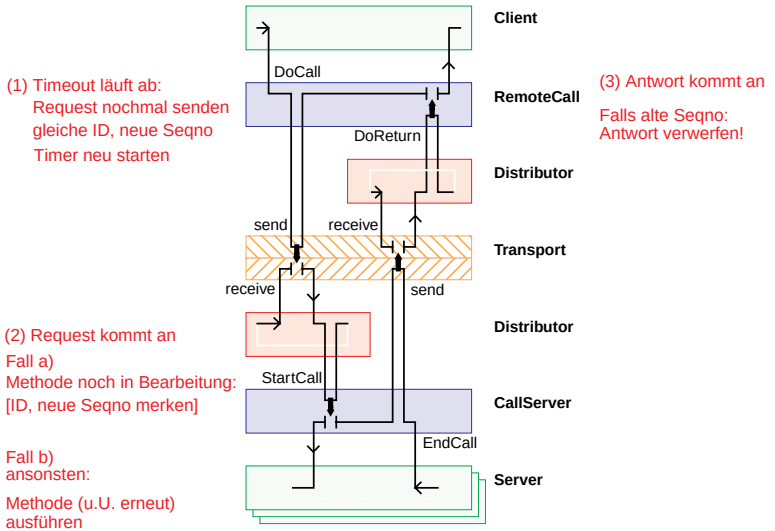
8 RPC-Aufruf genauer betrachtet

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

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



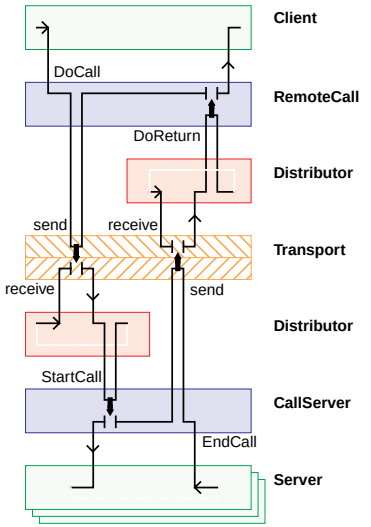
9 Last-of-Many-Semantik (Nelson-RPC)



9 Was kann man tun?

- Verhinderung der Mehrfach-Ausführung
  - ◆ eindeutige IDs pro Anfrage; Wiederholung mit gleicher ID
    - 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?

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



Weitere Optimierung:

Server merkt sich bereits gesendet Antwort

Bei erneutem Request wird ohne Methodenausführung die Antwort erneut gesendet

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

I.3 Aufgabe 6

- Timeout
  - ◆ Aufgabe: eine bestimmte Zeit lang warten
- mit Signalen (siehe nächste Folien) relativ kompliziert
- mittels **`select`** oder **`poll`**
  - ◆ ungeeignet wenn die Threads als User-Level Threads implementiert sind
  - ◆ Beispiel: 2 Sekunden warten mit **`select`**

```
struct timeval timeout;
timeout.tv_sec = 2;
timeout.tv_usec = 0;
select(1, NULL, NULL, NULL, &timeout);
```

  - ◆ ... mit **`poll`**

```
poll(NULL, 0, 2000);
```

  - ◆ (**`select`** und **`poll`** warten bis sich der Zustand eines Filedeskriptors ändert)
- RCX: **`void sleep(int ticks);`**

1 Zeitgeber

- Timer einstellen

```
#include <unistd.h>
unsigned int alarm(unsigned int sec);
```

  - nach **sec** Sekunden wird ein **SIGALRM** ausgelöst
  - **sec** == 0 löscht den Timer falls noch nicht abgelaufen

- Alternative (für periodische **SIGALRM** Signale):
  - Intervalltimer einstellen (**setitimer**)

2 Wiederholung: POSIX-Signale

- verzögerte Signale
  - während der Ausführung der Signalhandler-Prozedur wird das auslösende Signal blockiert
  - bei Verlassen der Signalbehandlungsroutine wird das Signal deblockiert
  - es wird maximal ein Signal zwischengespeichert
- mit **sa_mask** in der **struct sigaction** kann man zusätzliche Signale blockieren
- Modifikation der Signal-Maske vom Typ **sigset_t** mit fogenden Makros:
  - **sigaddset()**, **sigdelset()**, **sigemptyset()**, **sigfillset()**

2 Wiederholung: POSIX-Signale

- Signalhandler installieren:

```
#include <signal.h>

int sigaction(int sig,                          /* Signal */
              const struct sigaction *act,      /* Handler */
              struct sigaction *oact );  /* Alter Handler */
```

  - Handler bleibt installiert, bis neuer Handler mit **sigaction** installiert wird

- **sigaction** Struktur

```
struct sigaction {
    void (*sa_handler)(int);
    sigset_t sa_mask;
    int sa_flags;
}
```

  - **sa_handler**: Signal-Handler oder **SIG_DFL**, **SIG_IGN**
  - **sa_mask**: Signalmaske während der Ausführung des Signal-Handlers
  - **sa_flags**: Verhalten beim Signalempfang
    (z.B.: **SA_NODEFER**, **SA_RESTART**)

2 Wiederholung: POSIX-Signale

- Beispiel:

```
#include <signal.h>
void my_handler(int sig) { ... }
...
struct sigaction action;
sigemptyset(&action.sa_mask);
action.sa_flags = 0;
action.sa_handler = my_handler;
sigaction(SIGUSR1, &action, NULL); /* return abfragen ! */
```

2 Wiederholung: POSIX-Signale

- Ändern der prozessweiten Signal-Maske

```
int sigprocmask(int how,      /* Verknüpfung der Masken */
                const sigset_t *set, /* neue Maske */
                sigset_t *oset);     /* alte Maske */
```

  - how:
    - **SIG_BLOCK**: Vereinigungsmenge zwischen übergebener und alter Maske
    - **SIG_SETMASK**: Setzen der Maske ohne Beachtung der alten Maske
    - **SIG_UNBLOCK**: Schnittmenge zwischen inverser übergebener Maske und alter Maske

2 Wiederholung: POSIX-Signale

- Unterbrechen von Systemcalls
  - der Systemcall setzt dann **errno** auf **EINTR**
  - in einigen UNIXen (z.B. 4.2BSD) werden unterbrochene Systemcalls automatisch neu aufgesetzt
  - bei einigen UNIXen (SVR4, 4.3BSD), kann man für jedes Signal einstellen (**SA_RESTART**), ob ein Systemcall automatisch neu aufgesetzt werden soll
  - POSIX.1 läßt dies unspezifiziert

2 Wiederholung: POSIX-Signale

- Warten auf Signale

```
int sigsuspend(const sigset_t *mask);
```

  - wartet auf Signale, die in **mask** enthalten sind
  - (**mask** wird damit zur aktuellen Signal-Maske)
  - kehrt nach Bearbeitung des Signalhandlers zurück

- Abfrage blockierter Signale

```
int sigpending(sigset_t *set);
```

  - **sigpending** speichert alle Signale, die blockiert sind, aber empfangen wurden, in **set** ab

3 Signalsemantik bei PThreads

- Grundlegendes Signal-Behandlungskonzept unverändert:
  - Signal ignorieren / Default-Reaktion / Signal abfangen
- Einstellung gilt immer für alle Threads eines Prozesses
- Problem
  - welchem Thread wird ein eintreffendes Signal zur Bearbeitung zugestellt

3 Signalzustellung

- Lösung unterscheidet nach der Art der Signal-Entstehung
- Traps: synchron durch die Programmausführung erzeugt
  (Segmentation fault, Illegal instruction, ...)
  - ◆ Signal wird an den verursachenden Thread zugestellt
- Explizite Signalerzeugung durch das Programm
  (Funktion **`pthread_kill`**)
  - ◆ Signal wird an den angegebenen Ziel-Thread zugestellt
- Interrupts: asynchron von “außen” erzeugt
  (Interrupt, Quit, Hangup, SIGIO, ...)
  - ◆ Signal wird dem gesamten Prozess zugestellt

3 Signal-Masken

- Signale können in UNIX maskiert werden
- Signal-Masken werden an neu erzeugte Threads vererbt
- Threads können thread-spezifische Signal-Masken setzen

```
#include <pthread.h>
#include <signal.h>

int pthread_sigmask(int how, const sigset_t *set,
                                             sigset_t *oset);
```

- Bearbeitung von Signalen an den gesamten Prozess:
  - ◆ Zustellung erfolgt an einen der Threads, die das Signal nicht blockiert haben
  - ◆ Auswahl des Threads erfolgt zufällig