Überblick

RPC-Semantiken
- Fehler bei Fernaufrufen
- Fehlertolerante Fernaufrufe
- Übungsaufgabe 3

Fehlerquellen

- Anwendung
  - Fehler sind nicht im Fernaufruf begründet
  - Fehlersituation tritt bei lokalem Methodenaufruf ebenfalls auf
  - Beispiele
    - Falsche Eingaben
      [Vergleiche: `IllegalArgumentException` von `VSBoard.get()`]
    - Programmierfehler in der Anwendung
    - ...

- Umgang mit in der Anwendung begründeten Fehlern
  - Aus Sicht des Fernaufrufsystems: reguläres Verhalten
  - Keine Fehlerbehandlung im Fernaufrufsystem
  - Transparente Signalisierung

Fehlerquellen

- Rechner (Client und/oder Server)
  - Prozess-, Programm-, Rechnerabsturz
  - Verzögerungen (z. B. aufgrund von Überlast)
- Kommunikation
  - Nachrichten (Anfrage und/oder Antwort)
    - Reihenfolgeänderung
    - Korrumpierung
    - Vervielfachung
    - Verlust
  - Verbindung
    - Verlangsamung
    - Abbruch
- Umgang mit im Fernaufruf begründeten Fehlern
  - Fehlerbehandlung im Fernaufrufsystem
  - Nur bei Scheitern der Fehlerbehandlung: Signalisierung

Einordnung

- Rechnerfehler
  - 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 eventuell nur einer betroffen
- Kommunikationsfehler
  - 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 [Siehe Übungsaufgabe 4.]
    - ...
  - Beliebig komplex
    - Geringer Aufwand → Tolerierung weniger bzw. nur bestimmter Fehlern
    - Hoher Aufwand → Tolerierung vieler 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

Signalisierung von Fernaufruffehlern in Java RMI

- Exception-Klasse `java.rmi.RemoteException`
  - 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 von `RemoteException`
  - `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
  - ...

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!

Überblick

RPC-Semantiken

- Fehler bei Fernaufrufen
- Fehlertolerante Fernaufrufe
- Übungsaufgabe 3

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 ($\rightarrow$ Transaktionen)
  - ...

Kombinierter Ansatz

- Fehlertolerantes Kommunikationsprotokoll
  - Beispiel: TCP statt UDP
  - Tolerierung von
    - Reihenfolgeänderung
    - Korrumpierung
    - Vervielfachung
    - Verlust

    von Nachrichten
- 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

Fernaufrufsemantiken

- Bemerkung
  - Im Folgenden wird die Tolerierung von **Kommunikationsfehlern** betrachtet, Rechnerausfälle werden ausgeklammert
  - Die Tolerierung von Rechnerausfällen erfordert Mechanismen zum Wiederanlaufen (z. B. Zustandswiederherstellung)
- Semantiken
  - Maybe
  - At-Least-Once
  - At-Most-Once
  - Last-Of-Many
- Unterschiede
  - Mehrmaliges Senden von Anfragen
  - Aktualität der Antworten
  - Anzahl der Ausführungen $\rightarrow$ *Idempotente Operationen*?
  - Antwortspeicherung $\rightarrow$ 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
    - Lösungsansatz: Verwendung absoluter Beträge
    - Achtung: nicht-triviale Implementierung, da zusätzliche Synchronisation und/oder Ausnahmebehandlung nötig

Antwortspeicherung

- Problem
  - Server stellt eigene Ressourcen ($\rightarrow$ 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-{Least,Most}-Once

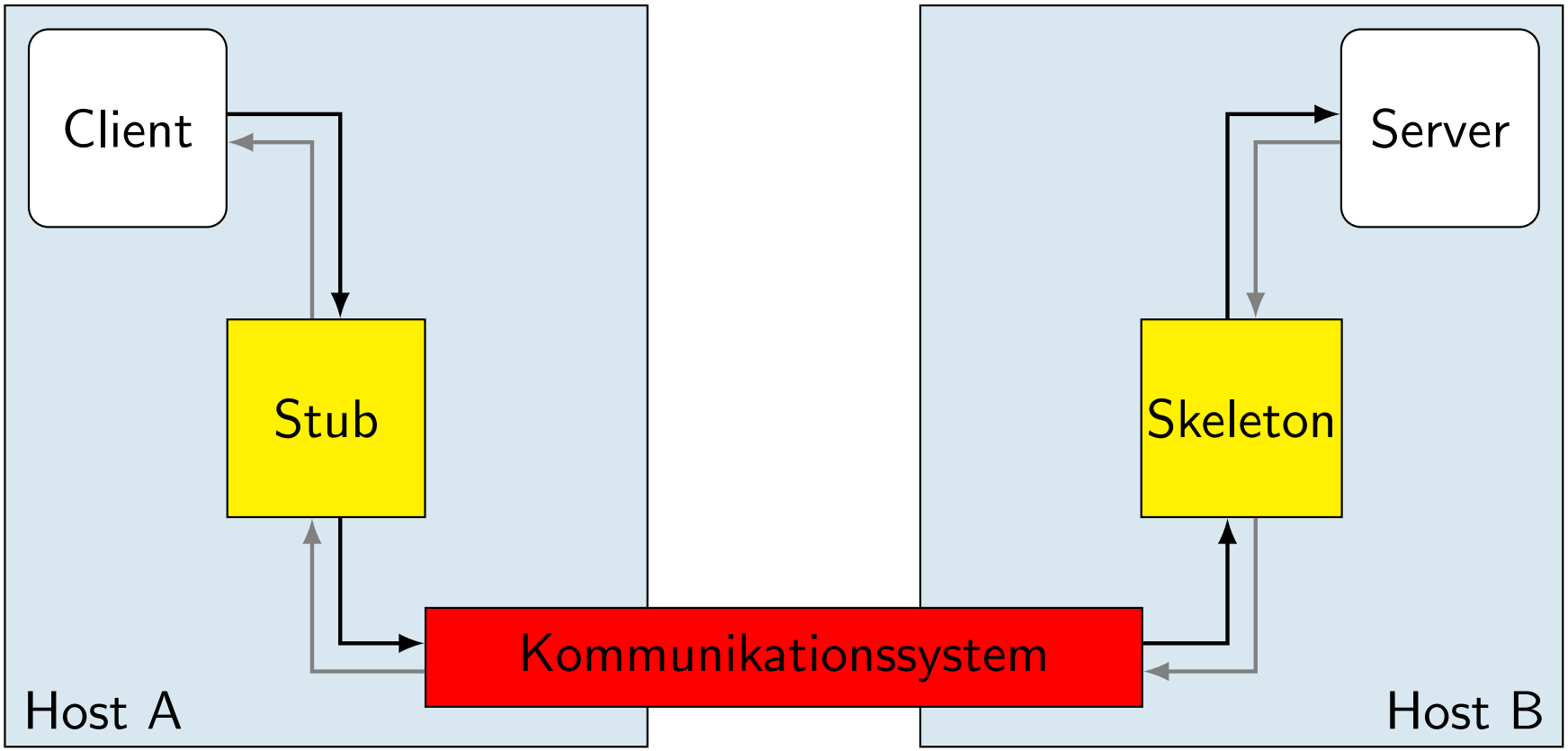
- At-Least-Once (z. B. Sun RPC)
  - Funktionsweise
    - Client wiederholt Anfrage, falls Antwort ausbleibt
    - Client akzeptiert die erste Antwort, die ihn erreicht
  - Eigenschaften
    - Anfragen werden eventuell mehrfach ausgeführt
    - Client verwendet eventuell veraltete Antwort
- At-Most-Once (z. B. Java RMI)
  - Funktionsweise
    - Client wiederholt Anfrage, falls Antwort ausbleibt
    - Server speichert Antwort
    - Server sendet bei Anfragewiederholungen gespeicherte Antwort
  - 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 eventuell mehrfach ausgeführt

Übungsaufgabe 3

- Simulation von Kommunikationsfehlern
- Bereitstellung von Fehlertoleranzmechanismen



Sabotage des Kommunikationssystems

- Simulation von Kommunikationsfehlern
  - Verlust von Nachrichten
  - Verzögerung einzelner Nachrichten
  - Vervielfachung von Nachrichten
  - Nicht betrachtet: Fehlerhafte Teilnachrichten
- Tests
  - Variation der Fehlerintensität
  - Kombination verschiedener Fehlerarten
- Mögliche Implementierung
  - Fehlerhafte `VSObjectConnection` → `VSBuggyObjectConnection`
  - Überschreiben von
    - `sendObject()` oder
    - `receiveObject()`

Implementierung der Fernaufrufsemantiken

- *Last-Of-Many*
  - Fernaufruf-IDs
  - Sequenznummern
  - Timeouts
- *At-Most-Once*
  - Einmalige Ausführung
  - Speicherung der Ergebnisse
  - Garbage Collection für Ergebnisse
- Hinweis

  Die zum Einsatz kommende Fernaufrufsemantik
  - *Maybe*
  - *Last-Of-Many*
  - *At-Most-Once*

  soll konfigurierbar sein!

Mögliche Timeout-Behandlung in Java

- Timer-Klasse `java.util.Timer`
  - Einfache Scheduler-Funktionalität
    - Unterstützung ein- sowie mehrmaliger Task-Ausführung
    - Task-Objekte: Instanzen von `TimerTask`-Unterklassen
  - Wichtige Methoden

```
void schedule(TimerTask task, long delay);
void scheduleAtFixedRate(TimerTask t, long dy, long period);
void cancel();
```

    - `schedule()` Einmalig auszuführenden Task aufsetzen
    - `scheduleAtFixedRate` Periodischen Task aufsetzen
    - `cancel()` Timer beenden
- Timeout-Handler-Klasse `java.util.TimerTask`
  - Timeout-Handler
    - Basisklasse für von `Timer` eingeplante Tasks
    - Auszuführenden Task-Code in Unterklassen implementieren
  - Wichtige Methoden

```
abstract void run();
boolean cancel();
```

    - `run()` Task ausführen → Timeout behandeln
    - `cancel()` Task bzw. Timeout abbrechen

Timer/TimerTask-Beispiel

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

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

        // Zu ueberwachenden Code ausfuehren
        [...]

        // Timeout deaktivieren und Timer aufraeumen
        handler.cancel();
        timer.cancel();
    }
}

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