

Fernaufwurfsemantiken

Fehler bei Fernaufrufen

Fehlertolerante Fernaufrufe

Übungsaufgabe 3



# Reaktion des Fernaufrufsystems auf Fehler

- In der Anwendung begründete Fehler
  - Fehlersituationen treten bei lokalem Methodenaufruf ebenfalls auf
  - Beispiele
    - Falsche Eingaben [Vergleiche: `VSAuctionException` bei `VSAuctionService.registerAuction()`]
    - Programmierfehler in der Anwendung
  - Reaktion des Fernaufrufsystems
    - Aus Sicht des Fernaufrufsystems: Reguläres Verhalten
    - Keine Fehlerbehandlung im Fernaufrufsystem → Transparente Signalisierung
- Im Fernaufruf begründete Fehler
  - Fehlersituationen sind bei lokalem Methodenaufruf nicht relevant
  - Beispiele
    - Rechner: Prozess-, Programm-, Rechnerabsturz, Verzögerungen (Überlast)
    - Nachrichten: Reihenfolgeänderung, Korrumpierung, Verlust
    - Verbindung: Verlangsamung, Abbruch
  - Reaktion des Fernaufrufsystems
    - Fehlerbehandlung im Fernaufrufsystem
    - Signalisierung nur bei Scheitern der Fehlerbehandlung



- Rechnerfehler
  - Lokaler Methodenaufruf
    - Aufrufer und Aufgerufener in gleichem Maße betroffen
    - Im Fehlerfall sind beide abgestürzt 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!**



- Fehlertolerierung
  - Einsatz von Fernaufrufsemantiken
  - Problem: Nicht alle Fehler lassen sich tolerieren
- Fehlersignalisierung
  - **Verletzung der Transparenzeigenschaften**
    - Benachrichtigung an den Benutzer des Fernaufrufsystems
    - Benutzer des Fernaufrufsystems muss darauf vorbereitet sein
  - Umsetzung in Java RMI mittels `java.rmi.RemoteException`
    - Muss von jeder Methode einer Remote-Schnittstelle geworfen werden
      - **Verletzung der Zugriffstransparenz**
    - Unterklassen von `RemoteException` (Beispiele)

Exception	Beschreibung
<code>ConnectException</code>	Verbindungsaufbau fehlgeschlagen
<code>NoSuchObjectException</code>	Remote-Objekt nicht (mehr) verfügbar
<code>ServerError</code>	Auspacken der Anfrage, Ausführung der Methode oder Einpacken der Antwort fehlgeschlagen
<code>UnknownHostException</code>	Remote-Host nicht bekannt



- Probleme
  - Keine definitive Fehlererkennung (Liegt überhaupt ein Fehler vor?)
  - Keine exakte Fehlerlokalisierung (Wo liegt der Fehler?)
- Beispielszenario: Ein 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
- Erkenntnis

**Eine präzise Fehlererkennung ist in verteilten Systemen im Allgemeinen nicht möglich!**



- **Ansatzpunkt**
  - Tolerierung von Kommunikationsfehlern
  - Wiederanlaufen nach Rechnerausfällen erfordert zusätzliche Mechanismen
- **Semantiken**
  - Maybe
  - At-Least-Once
  - At-Most-Once
  - Last-of-Many
- **Unterschiede**
  - Mehrmaliges Senden von Anfragen
  - Aktualität der Antworten
  - Anzahl der Ausführungen
    - Idempotente Operationen?
    - Duplikaterkennung?
  - Antwortspeicherung → Wie lange wird eine Antwort aufgehoben?



## ■ Idempotente Funktionen (Mathematik)

### ■ Definition

$$f(x) = f(f(x))$$

### ■ Beispiele

- Konstante Funktion  $f(x) = c$
- Multiplikation mit 1  $f(x) = x \cdot 1$
- Division durch 1  $f(x) = \frac{x}{1}$

## ■ Idempotente Operationen (Informatik)

### ■ Charakteristika mehrfacher Ausführungen

- Identische Rückgabewerte
- Identische Anwendungszustände

### ■ Beispiele

- Leseoperation auf unverändertem Zustand
- Zustandsmodifikation durch Setzen neuer Daten

### ■ Triviale Kombination idempotenter Operationen nicht immer idempotent



- Problem
  - Server stellt eigene Ressourcen für Fernaufrufe bereit (→ Antwort-Cache)
  - Mit jedem neuen Fernaufruf werden zusätzliche Ressourcen belegt
  - Wann können die gespeicherten 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 Clients wird die alte Antwort gelöscht
    - **Problem: Letzter Fernaufruf eines Clients**
  - Timeout
    - Antwortlöschung nach Ablauf eines fernaufrufspezifischen Timeout
    - **Als Rückfallposition immer nötig**
- Herausforderung: Aufrechterhaltung der Semantikgarantien





- At-Least-Once
  - 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
  - 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

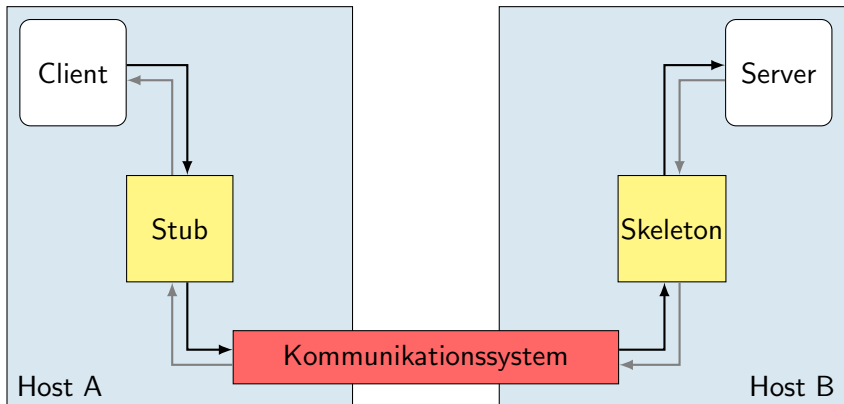


- Funktionsweise
  - Client wiederholt Anfrage, falls Antwort ausbleibt
  - Client akzeptiert nur Antwort auf seine aktuellste Anfrage
- Implementierung
  - Fernaufruf muss eindeutig identifizierbar sein
    - Client
    - Remote-Objekt
    - Remote-Methode
    - Aufrufzähler
  - Zusätzlich: Jede Fernaufrufnachricht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

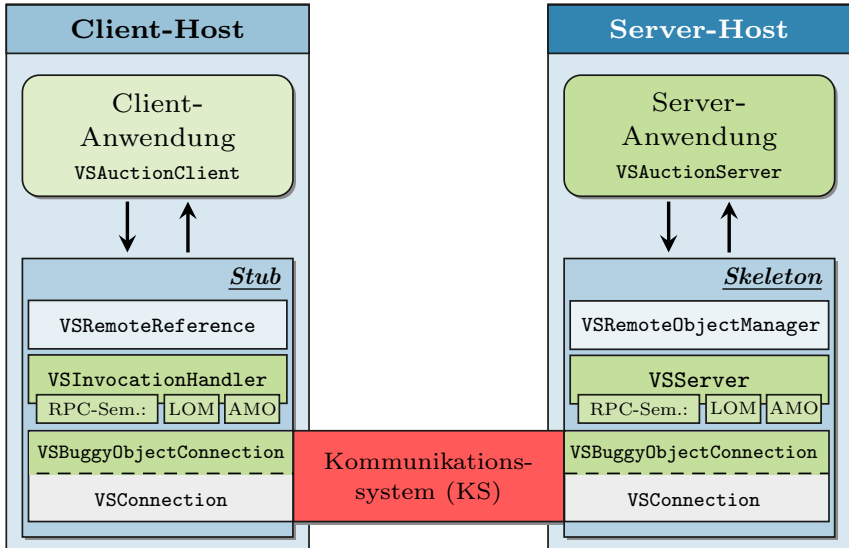


# Übungsaufgabe 3

- Simulation von Kommunikationsfehlern
- Bereitstellung von Fehlertoleranzmechanismen



# Übungsaufgabe 3



- *Last-of-Many*
  - Fernaufruf-IDs
  - Sequenznummern
  - Timeouts
- *At-Most-Once*
  - Einmalige Ausführung
  - Speicherung der Ergebnisse
  - Garbage-Collection für Ergebnisse
- Auswahl der Fernaufrufsemantik
  - Methodenspezifische Festlegung
  - Annotierung der Anwendungsschnittstelle durch den Programmier
    - `@VSRPCSemantic(VSRPCSemanticType.LAST_OF_MANY)` bzw.
    - `@VSRPCSemantic(VSRPCSemanticType.AT_MOST_ONCE)`
  - Analyse der Annotation durch das Fernaufrufsystem zur Laufzeit



- Annotationen: Bereitstellung von Metadaten im Quelltext
- Beispiel: Kennzeichnung von schreibenden bzw. lesenden Methoden
  - Hilfs-enum zur Typunterscheidung

```
public enum VSMethodType {  
    READ_ACCESS,  
    WRITE_ACCESS  
}
```

- Definition der Annotation mittels @interface in VSAnnotation.java

```
@Retention(RetentionPolicy.RUNTIME)  
public @interface VSAnnotation {  
    VSMethodType value();  
}
```

- @Retention-Annotation: Sichtbarkeit von VSAnnotation zur Laufzeit
- Spezifizierung des Rückgabetyps der Standardmethode value()

- Einsatz der Annotation

```
@VSAnnotation(VSMethodType.WRITE_ACCESS)
```

[Hinweis: Sollte der Methodenname von „value()“ abweichen, muss beim Einsatz der Annotation der Methodenname explizit angegeben werden. Beispiel: foo() → @VSAnnotation(foo = VSMethodType.WRITE\_ACCESS)]



- Beispiel: Schnittstelle eines Speichers für Schlüssel-Wert-Paare

```
public interface VSKeyValueStore {
    @VSAAnnotation(VSMethodType.WRITE_ACCESS)
    public void put(String key, String value);

    @VSAAnnotation(VSMethodType.READ_ACCESS)
    public String get(String key);
}
```

- Analyse der Schnittstelle VSKeyValueStore

- Zugriff auf Annotation mittels Method.getAnnotation()

```
for(Method method: VSKeyValueStore.class.getMethods()) {
    VSAAnnotation annotation = method.getAnnotation(VSAAnnotation.class);
    VSMethodType type = annotation.value();
    System.out.println(method.getName() + ": " + type);
}
```

- Ausgabe

```
get: READ_ACCESS
put: WRITE_ACCESS
```



# Mögliche Timeout-Behandlung in Java

- Timer-Klasse `java.util.Timer`
  - Einfache Scheduler-Funktionalität für `TimerTask`-Objekte
  - Zentrale 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`
  - Basisklasse für von `Timer` eingeplante Tasks
  - Zentrale 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 aufräumen
        handler.cancel();
        timer.cancel();
    }
}

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



- Setzen von Socket-Timeouts mittels `setSoTimeout()`
  - Konfigurierung der Maximaldauer, die ein Leseaufruf am Socket blockiert
  - Leseaufruf kehrt bei Timeout-Ablauf mit `SocketTimeoutException` zurück
- Beispiel

```
// Socket-Timeout setzen
Socket socket = [...];
try {
    socket.setSoTimeout(5000);
} catch(IOException ioe) {
    System.err.println("Setting the socket timeout failed with " + ioe);
    // Fehlerbehandlung
}

// Leseaufruf starten
try {
    socket.getInputStream().read();
} catch(SocketTimeoutException ste) { // -> "Timeout: Read timed out"
    System.err.println("Timeout: " + ste.getMessage());
} catch(IOException ioe) {
    System.err.println("I/O error: " + ioe);
}
```



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

