# 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 2010



## Überblick

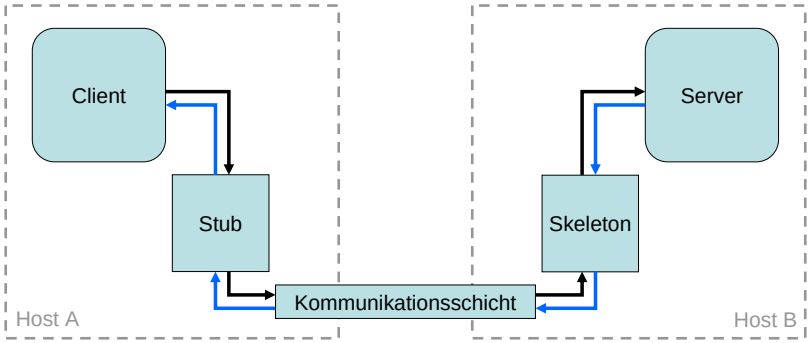






## VS-Übung

- Entwicklung eines eigenen Fernaufrufsystems
- Orientierung an Java-RMI

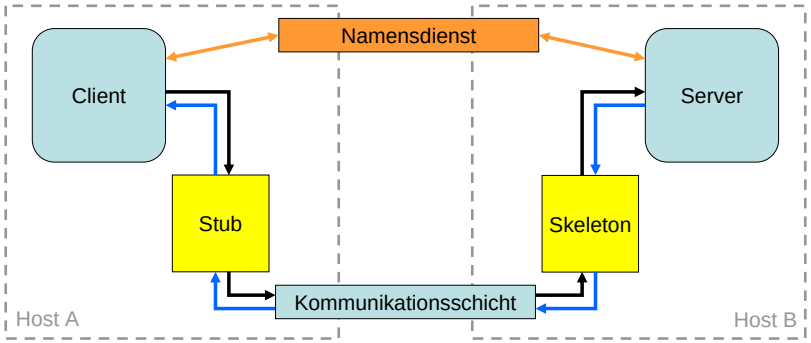







## Übungsaufgabe 4

- Integration von Rückrufen
- Bereitstellung eines Namensdienstes
- Ausführliche Tests



## Designentscheidung

Wo soll die {S,Des}erialisierung der Parameter stattfinden?

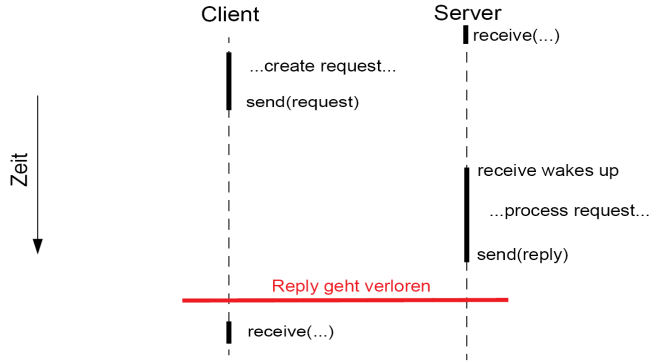
- Im Kommunikationssystem
  - Nachrichtenformat: beliebige Objekte
  - Stub & Skeleton einfach
  - Abhängigkeit von jeweiliger Kommunikationssystem-Implementierung → Probleme bei heterogenen Kommunikationssystemen
- In Stub & Skeleton
  - Nachrichtenformat: Byte-Array
  - Stub & Skeleton komplexer
  - Unabhängigkeit von Kommunikationssystem-Implementierungen





## Problemszenario

- Ein Fernaufruf aus Sicht von Stub & Skeleton (Skizze)
- Annahme: zuverlässiger Kommunikationskanal




→ Timing-Problem: Antwort trifft vor Aufruf von `receive()` ein





## Timing-Problem

Lösungsmöglichkeiten

- Problemlösung auslagern
  - Pufferung ankommender Daten in unteren Schichten
  - z. B. im Empfangs-Socket
- Optimistische Lösung bei echter Verteilung
  - Ausnutzung der Netzwerklatenz
  - Mit hoher Wahrscheinlichkeit tritt das Problem nicht auf
- Sichere Lösung
  - Mehrere Threads auf Client-Seite
  - `receive()` wird vor `send()` aufgerufen

## Callback

Beispielszenario (vgl. Übungsaufgabe 1)

- Server-Seite

```
public interface Messageboard extends Remote {
    public void post(Message msg);
    public Message[] get(int n);
    public void addListener(MessageboardListener listener);
}
```

- Client-Seite

```
public interface MessageboardListener extends Remote {
    public void newMessagePosted(Message msg);
}
```

→ Der Server muss den Client (per Fernaufruf) zurückrufen können

→ Dem Server muss eine Referenz auf den Client vorliegen





## Probleme mit Referenzen bei Callbacks

Primitive Referenzen

- Lokaler Methodenaufruf
  - Identischer Adressraum
  - Referenz auch in aufgerufener Methode gültig
  - → Callback erfordert keine spezielle Betrachtung
- Fernaufruf
  - Unterschiedliche Adressräume
  - Referenz normalerweise nicht in aufgerufener Methode gültig
    (Ausnahme: z. B. ,,Distributed Shared Memory (DSM)"-Systeme)
  - → Einfache Übertragung einer Referenz (meist) nicht sinnvoll

→ Spezielle Semantiken für Fernaufrufe notwendig





## Umsetzungsmöglichkeiten in verteilten Systemen

,,Callback" per **Call-by-Value-Result**

- Funktionsweise
  - Auf Server-Seite wird eine Kopie des Originalobjekts erzeugt
  - Aufgerufene Methode kann Kopie modifizieren
  - Kopie wird an Client zurückgesendet
  - Originalobjekt wird durch Kopie ersetzt
- Nachteile
  - Gültigkeit der Referenz ist beschränkt auf Methodenausführung
    - Bei Beendigung der Methode wird Kopie für Server wertlos
    - Referenz kann nicht dauerhaft gespeichert werden
  - Komplettes Objekt wird doppelt übertragen
    - Betrifft auch Daten, die auf Server-Seite nicht benötigt werden
    - Problematisch für große Objekte

## Umsetzungsmöglichkeiten in verteilten Systemen

Callback per **Call-by-Reference**

- Funktionsweise
  - Objekt wird auf Client-Seite für Fernaufrufe verfügbar gemacht
  - Dem Server wird als Parameter eine **Remote-Referenz** übergeben
  - Jeder Server-seitige Zugriff auf das Objekt erfolgt per Fernaufruf
  - Aufgerufene Prozedur kann Daten des Aufrufers direkt verändern
- Vorteile gegenüber Call-by-Value-Result
  - Speicherung der Referenz für spätere Verwendung möglich
  - Geringere zu übertragende Datenmenge
- Nachteil

  Im allgemeinen Fall meist nur mit spezieller Betriebssystem-Unterstützung möglich





## Umsetzungsmöglichkeiten in verteilten Systemen

Call-by-Reference in Objekt-orientierten Programmiersprachen

- Funktionsweise
  - Objekt kapselt Daten
  - Zugriff (Idealfall) nur über Methodenaufrufe
  - Übertragung einer Remote-Referenz führt auf Server-Seite automatisch zur Erzeugung eines Objekt-Stubs
  - Server kann transparent auf das Original-Objekt zugreifen
- Einschränkung
  - Kein direkter Zugriff auf Objektzustand (keine ,,public"-Variablen)

→ Problem ohne spezielle Betriebssystem-Unterstützung lösbar





## Verwaltung von Callback-{Stubs,Skeletons}

- Naiver Ansatz
  - Bei jeder Weitergabe einer Objekt-Referenz wird ein neuer Stub sowie ein neuer Skeleton erzeugt

  → Unnötig, falls die selbe Objektreferenz mehrfach übertragen wird!
- Übliches Verfahren in vielen Fernaufrufsystemen:

  Verwaltung mittels Hashtabellen
  - Client-Seite: Zuordnung lokaler Objektreferenzen zu Skeletons
  - Server-Seite: Abbildung von Remote-Referenzen auf Stubs

Wie lange sollen diese Informationen verfügbar gehalten werden?





## Verwaltung von Callback-{Stubs,Skeletons}

Freigabe von Stubs und Skeletons

- Allgemein
  - Wo?
    - Auf Applikationsebene
    - Im Skeleton des Original-Fernaufrufs (auf Server-Seite)
  - Wie?
    - Explizit: z. B. konkrete Anweisung
    - Implizit: z. B. Methodenende
    - Automatisiert: z. B. Garbage-Collection
- Java-RMI
  - Stub wird gelöscht, sobald keine Referenz mehr auf ihn verweist (reguläre Garbage Collection)
  - Zusätzlich: Distributed Garbage Collection

## Verwaltung von Stubs & Skeletons in Java-RMI

Distributed Garbage Collection

- Jeder Server unterhält jeweils einen Remote-Referenzen-Zähler auf von ihm bereitgestellte Remote-Objekte
  - `dirty`-Methode
    - Inkrementiert Zähler
    - Aufgerufen vom Client bei Stub-Erzeugung (per Fernaufruf)
  - `clean`-Methode
    - Dekrementiert Zähler
    - Aufgerufen vom Client bei Stub-Freigabe (per Fernaufruf)
- Lokal bereitgestelltes Remote-Objekt wird vom Server der Garbage Collection überlassen, sobald
  - keine lokalen Referenzen mehr auf das Objekt existieren **und**
  - der Remote-Referenzen-Zähler auf Null steht.





## Verwaltung von Stubs & Skeletons in Java-RMI

Distributed Garbage Collection: **Leases**

- Garantie des Servers an den Client, dass ein bestimmtes Remote-Objekt für eine gewisse Zeit (Default-Wert: 10 Minuten) verfügbar ist
- Wird beim Aufruf von `dirty()` zurückgegeben
- Kann durch erneuten Aufruf von `dirty()` verlängert werden (erfolgt üblicherweise nach Ablauf der Hälfte der Lease-Dauer)
- Wird ein Lease vom Client nicht verlängert, dekrementiert der Server den entsprechenden Remote-Referenzen-Zähler

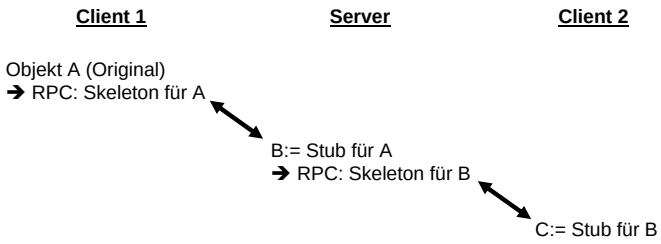
→ Leases stellen eine Absicherung des Servers gegen Verbindungs- bzw. Client-Ausfälle dar





## Aufrufketten

- Szenario: Weiterleitung von Fernaufrufen
- Naive Umsetzung




- Optimierung
  - Kein zusätzlicher Skeleton für B beim Server
  - Weitergabe der Remote-Referenz auf A an Client 2

## Unit-Testing

- „Unit"
  - Kleinste testbare Einheit einer Applikation
  - Beispiele
    - Funktion
    - Programm
- Motivation
  - Inkrementelle Entwicklung
  - Weniger Debugging am kompletten System
  - Kleinerer Programmcode
  - Reduzierung des Wartungsaufwands





## Übersicht

- Allgemeine Testklasse (`junit.framework.TestCase`)
  - Dient als Oberklasse für alle JUnit-Tests
  - Verwaltet Einzeltests
  - Erzeugt Testumgebung
- Einzeltest
  - Implementierung in Methode einer Unterklasse von `TestCase`
  - Methodenname des Einzeltests beginnt mit „`test`"
- Skizze

```
public class MyTest extends TestCase {
    public void testMyObjectFirstTest() { [...] }
    public void testMyObjectSecondTest() { [...] }
    [...]
}
```





## Suites `junit.framework.TestSuite`

- Aufbau
  - Zusammenfassung einzelner Testfälle
  - Schachtelung von Test-Suites möglich
- Umsetzung
  - Bereitstellung mittels statischer Methode `suite()`

```
public static Test suite()
```

- Skizze

```
public class TestSuiteExample extends TestCase {
    public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTest(new TestSuite(MyTest.class));
        suite.addTest(new TestSuite(AnotherTest.class));
        return suite;
    }
}
```





## Fixtures `junit.framework.TestCase`

- Vordefinierte Methoden
  - `protected void setUp()`
    - Einrichtung von Rahmenbedingungen
    - z. B. Erzeugung von Objekten, Aufbau von Verbindungen
  - `protected void tearDown()`
    - Aufräumen der Testumgebung
    - z. B. Objekte freigeben, Sockets schließen
- Ablauf für **jeden** Einzeltest
  1. `setUp()`
  2. `testMyObject()` (eigentlicher Test)
  3. `tearDown()`
- Verwendung optional

## Zusicherungen `junit.framework.Assert`

- `assert*`-Methoden (alle statisch)
  - `assertTrue()` stellt fest, ob eine Bedingung wahr ist
  - `assertEquals()` verifiziert, ob zwei Objekte gleich sind
  - `assertNull()` prüft, ob Ausdruck `null`
  - `assertSame()` verfiziert, ob zwei übergebene Referenzen auf das gleiche Objekt verweisen
  - ...
- `fail()`-Methode
  - Signalisiert, dass eine Stelle im Testfall erreicht wurde, die bei korrekter Ausführung nicht hätte erreicht werden dürfen
  - Beispiel

```
try {
    int[] array = new int[4];
    array[4] = 47;
    fail("ArrayIndexOutOfBoundsException expected");
} catch(ArrayIndexOutOfBoundsException e) {}
```





## Testlauf starten

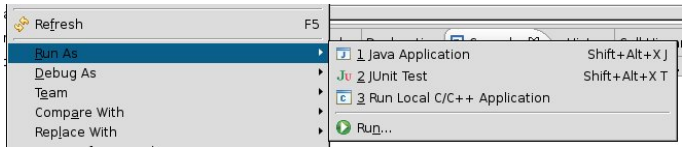
- Kommandozeile
  - Allgemein: per `junit`-Skript

```
junit <Klassenname>
```

  - Beispiel

```
junit vsue.tests.VSCallbackTest
```

- Eclipse








## Beispiel

Beispielklasse `Money`

```
public class Money {
    private double amount;

    public Money(double amount) {
        this.amount = amount;
    }

    public double getAmount() {
        return this.amount;
    }

    public double addAmount(double amount) {
        this.amount += amount;
        return getAmount();
    }
}
```





## Beispiel

Testklasse `MoneyTest`

```
import junit.framework.TestCase;

public class MoneyTest extends TestCase {
    public MoneyTest(String name) {
        super(name);
    }

    public void testGetAmount() {
        Money money = new Money(2.0);
        assertTrue(2.0 == money.getAmount());
    }

    public void testAddAmount() {
        Money money = new Money(1.0);
        assertTrue(2.0 == money.addAmount(1.0));
    }
}
```

## Integration von Rückrufen

- Kombination von Funktionalitäten
  - Client muss auch Remote-Objekte bereitstellen können
  - Server muss auch Fernaufruf starten können
  - → Verschmelzung von `VSClient` und `VSServer` zu `VSRemoteEntity`

```
public class VSRemoteEntity {
    public void init(int serverPort);
    public void exportObject(Object object);
    public Object lookup(String host, int port,
                         Class interfaceClass);
}
```

- Integration von Call-by-Reference
  - Neues Interface für Remote-Dienste: `VSRemote`
  - Parameter ist exportiertes Remote-Objekt → `by-Reference`
  - Sonst → `by-Value` (wie bisher)





## Bereitstellung eines Namensdienstes

- Funktionalität
  - Verwaltung von Remote-Referenzen
  - Gezielter Zugriff auf einzelne Remote-Objekte über „Namen"
- Schnittstelle (vgl. `java.rmi.Naming`)

```
public interface VSNameService {
    public void bind(String name, VSRemote object);
    public String[] list(String name);
    public VSRemote lookup(String name);
    public void rebind(String name, VSRemote object);
    public void unbind(String name);
}
```

- Umsetzung
  - Integrierte Lösung: Namensdienst als Remote-Objekt
  - Keine externe Registry





## Bereitstellung eines Namensdienstes

Namensdienst als Remote-Objekt

- Problemstellungen
  - Dienst kann auch lokal aufgerufen werden; entscheidendes Kriterium: Adresse im Parameter `name`
  - Die Registrierung eines Remote-Objekts (`bind()`) erfolgt nicht notwendigerweise beim Namensdienst des selben Rechners, auf dem es zuvor exportiert wurde
- Mögliche Umsetzung
  - Remote-Komponente
    - Verwaltung lokal registrierter Remote-Objekte
    - Bereitstellung von Objekt-Referenzen (im Zuge von `lookup()`)
  - Lokale Komponente
    - Erzeugung von Stubs aus Remote-Referenzen (`lookup()`)
  - → Jeder Host muss beide Komponenten aufweisen!

## Ausführliche Tests

Testen mit JUnit

- Ziele
  - Alle Fehler der aktuellen Implementierung des Fernaufrufsystems sollen gefunden werden!
  - Komplettierung der Systemfunktionalität
- Vorgehensweise
  - Überprüfung des Systemverhaltens im fehlerfreien Fall
  - Überprüfung des Systemverhaltens in absichtlich herbeigeführten Fehlersituationen
- Mögliche Szenarien
  - Portierung (und Erweiterung) der Messageboard-Implementierung aus Aufgabe 1
  - Implementierung eines eigenen Testfalls





## Ausführliche Tests

Mögliche Fehlerquellen

- Applikation
  - Identische Fehlersituation entsteht auch bei lokalem Aufruf
  - Beispiele
    - Unerlaubte Eingaben
    - Programmierfehler

  → Fernaufrufsystem muss Fehlermeldung nur propagieren
- Fernaufrufsystem
  - Fehler sind im Fernaufruf bedingt
  - Beispiele
    - Unerreichbarer Server
    - Verbindungsabbruch
    - ...
  - Näheres in der Tafelübung zu Aufgabe 5...

  → Fernaufrufsystem muss Fehler (soweit möglich) behandeln





## Ausführliche Tests

Exceptions bei Fernaufrufen in Java

- Applikation
  - Allgemeine Laufzeitfehler (`RuntimeException`)
    - `NullPointerEcxeption`
    - `BufferOverflowException`
    - `ClassCastException`
    - ...
  - Methodenspezifische Fehler
    - `IOException`
    - `InstantiationException`
    - `NoSuchMethodException`
    - ...

  → Exception muss zum Aufrufer durchgereicht werden
- Fernaufrufsystem
  - Kommunikation: `RemoteException`
  - (Allgemeine Laufzeitfehler: siehe oben)





## Ausführliche Tests

Variationsmöglichkeiten

- Methodensignaturen (Parameter & Rückgabewert)
  - Ausschließlich primitive Datentypen bzw. Objekte
  - Mischung aus beidem
  - Keine Parameter
- Parameterwerte
  - Sinnvolle Eingaben
  - Falsche Eingaben (z. B. `null`) → Ausnahmebehandlung!
- Remote-Objekte
  - Zustandsbehaftet
  - Zustandslos
- Clients
  - Sequenzieller Zugriff
  - Paralleler Zugriff
- ...