Verteilte Systeme – Übung

Tobias Distler, Michael Gernoth

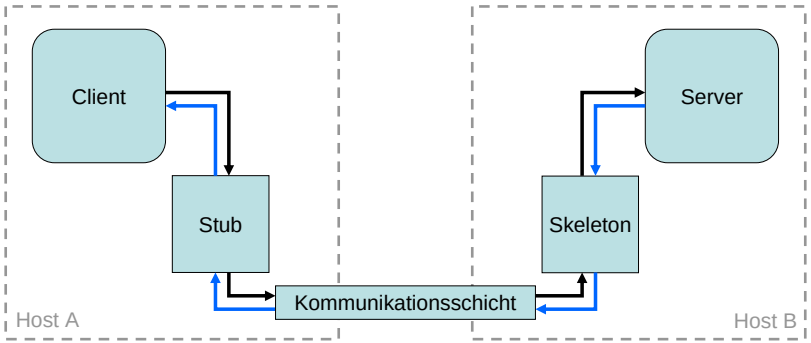
Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
www4.informatik.uni-erlangen.de

Sommersemester 2010

Überblick

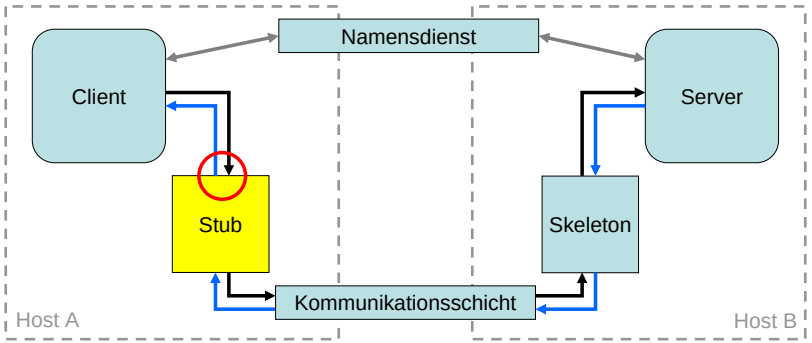
VS-Übung

- Entwicklung eines eigenen Fernaufrufsystems
- Orientierung an Java-RMI



Übungsaufgabe 6

- Integration von asynchronen Fernaufrufen
- Evaluation des eigenen Systems



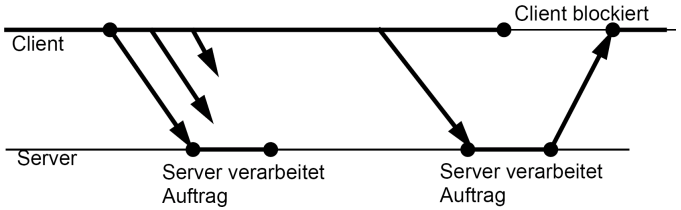
Überblick

- Fernaufrufvarianten
  - Synchron: Aufruf blockiert bis die Operation beendet ist
  - Asynchron: Aufruf blockiert nicht und kommt sofort zurück
- Asynchrone Fernaufrufe
  - Ohne Rückgabewert

    Notwendige Unterscheidung:
    - Ende der Operation ist für den Aufrufer irrelevant
    - Für den Aufrufer ist relevant, ob/wann die Operation beendet ist
  - Mit Rückgabewert
    - Ergebnis muss dem Aufrufer ,,nachgereicht" werden

Asynchrone Fernaufrufe

- Bezug zu lokalen asynchronen Methodenaufrufen
  - Verwandte Problemstellungen
  - Ziel ist (erneut) eine möglichst transparente Umsetzung im verteilten Fall
- Vorteile
  - Parallelverarbeitung von Client und Server möglich
    → effizientere Implementierung
  - Geeignet für hohen Durchsatz, besonders bei Operationen ohne Rückgabewert



Asynchrone Fernaufrufe mit Rückgabewert

Problemstellungen

- Wie erfährt der Initiator eines asynchronen Methodenaufrufs, dass die korrespondierende Operation
  - erfolgreich beendet
  - mit einem Fehler abgebrochen

  wurde?
- Wie wird ihm das Resultat mitgeteilt?

→ Mechanismus notwendig, um den Aufrufer einer Methode über das Ausführungsende zu informieren bzw. ihm den Rückgabewert zu übergeben

Asynchrone Fernaufrufe mit Rückgabewert

Naiver Implementierungsansatz (,,*fork*"-Variante)

- Vorgehensweise
  - Jeder Fernaufruf wird in einem eigenen Thread ausgeführt (→ `fork()`)
  - Dieser wartet auf das Eintreffen des Ergebnis
  - Sobald das Ergebnis verfügbar ist, wird es an einer dafür vorgesehenen Stelle gesichert und der Original-Thread benachrichtigt

  → Original-Thread kann nach dem ursprünglichen Aufruf sofort weitermachen
- Nachteil: Zusätzliche Kosten für
  - Thread-Erzeugung
  - Kontextwechsel
  - Thread-Beseitigung

Asynchrone Fernaufrufe mit Rückgabewert

Implementierungskonzepte

- Ergebnisannahme über einen speziellen Anweisungsblock
  - Vergleiche: Interrupt-Routine
- Bereitstellung eines speziellen Stellvertreters
  - „*Future*", „*Promise*", ...
  - → siehe nächste Folien
- Spracheinbettung durch *Future-Variablen*
  - Erweiterung des Future-Konzepts
  - Verbergung der Ergebnisannahme vor dem Programmierer
  - Beispiel (Pseudo-Code):

```
FUTURE int future;
int num = 47;
future = multiply(4, 7);
[...]
print(num + future);
```


Futures

- Oftmals synonym verwendet: *Promise*
- Schnittstelle

```
boolean poll();
<beliebiger Datentyp> get();
```

- Funktionsweise
  1. Beim asynchronen Aufruf wird (statt dem eigentlichen Ergebnis) sofort ein Future-Objekt zurückgegeben
  2. Das Future-Objekt lässt sich befragen, ob der tatsächliche Rückgabewert der Operation bereits vorliegt bzw. ob die Operation beendet ist → `poll()`
  3. Ein Aufruf von `get()`
     - liefert das Ergebnis der Operation sofort zurück, sofern es zu diesem Zeitpunkt bereits vorliegt **oder**
     - blockiert solange, bis das Ergebnis eingetroffen ist

Futures in Java — `java.util.concurrent.Future`

Schnittstelle `Future`

- Umfang
  - Methoden der allgemeinen Future-Schnittstelle
  - Zusätzliche Methoden zum Abbrechen von Tasks
- Schnittstelle

```
public interface Future<V> {
    public V get();
    public V get(long timeout, TimeUnit unit);

    public boolean isDone();   // --> poll()

    public boolean cancel(boolean mayInterruptIfRunning);
    public boolean isCancelled();
}
```


Futures in Java — `java.util.concurrent.ExecutorService`

Anwendungsbeispiel Executor-Service

- Interface `ExecutorService`
  - Erlaubt asynchrone Ausführung von Tasks
  - Task bei Executor-Service „abgeben", Ergebnis per Future
  - Zentrale Methode

```
<T> Future<T> submit(Callable<T> task)
```

- Interface `Callable`
  - Schnittstelle

```
public interface Callable<V> {
    V call() throws Exception;
}
```

  - Unterschiede zu `Runnable`
    - Rückgabewert
    - Exception

Futures in Java – Anwendungsbeispiel `ExecutorService`

Klasse `java.util.concurrent.Executors`

- Überblick
  - Hilfsmethoden zur Erzeugung von `Callable`-Objekten
  - Bereitstellung von `ExecutorService`-Implementierungen
- Wichtige Factory-Methoden für `ExecutorServices`
  - Ausführung in einem einzigen Thread

```
public static ExecutorService newSingleThreadExecutor();
```

  - Wiederverwendung von Threads

```
public static ExecutorService newCachedThreadPool();
```

  - Konstante Thread-Anzahl

```
public static ExecutorService newFixedThreadPool(int nThreads);
```

  - ...

Futures in Java – Anwendungsbeispiel `ExecutorService`

- Beispielklasse

```
public class FutureExample implements Callable<Integer> {
    private int a, b;

    public FutureExample(int a, int b) {
        this.a = a;
        this.b = b;
    }

    public Integer call() throws Exception {
        return a * b;
    }
}
```

- Aufruf

```
ExecutorService es = Executors.newSingleThreadExecutor();
FutureExample task = new FutureExample(4, 7);
Future<Integer> f = es.submit(task);
[...]
System.out.println("result: " + f.get());
```


Motivation

- Analyse des eigenen Systems
  - Leistungsfähigkeit
  - Antwortzeit
  - Durchsatz
  - Ressourcenverbrauch
  - ...
- Vergleich mit anderen Systemen
  - Wie verhalten sich die unterschiedlichen Varianten in bestimmten Situationen?
  - Wo liegen die jeweiligen Stärken und Schwächen?
  - ...

Unterscheidung

- Simulation
  - Messungen an einem Simulator, der das gewünschte Verhalten so gut wie möglich imitiert
  - Oftmals einfach zu realisieren
  - Ergebnisse spiegeln evtl. nicht exakt die Realität wider
- Evaluation
  - Messungen an einem konkreten System (bzw. Prototyp)
  - Im Allgemeinen aufwändiger zu realisieren
  - Ergebnisse entstammen einem realistischen Szenario

→ Evaluationen besitzen mehr Aussagekraft als Simulationen

Mögliche Probleme

- Nicht bzw. schwer zu evaluierende Merkmale
  - Eingeschränkte Quantifizierungsmöglichkeiten
  - Merkmal ist nicht isoliert messbar
  - ...
- Fehlende Vergleichsmöglichkeiten
  - Eigene Variante ist konkurrenzlos (eher selten der Fall)
  - Andere Varianten besitzen abweichenden Fokus
  - ...
- Beispiel: ,,Effizienz vs. Fehlertoleranz"
  - Aussagen über das Ausmaß von Fehlertoleranz können oft nicht durch Messergebnisse gestützt werden, stattdessen: oberflächliche Beschreibung (z. B. Anzahl und Art tolerierbarer Fehler)
  - Fehlertoleranz ist (fast) immer mit Effizienzeinbußen verbunden

  → Der durch den Einsatz fehlertoleranter Systeme erreichbare Gewinn lässt sich schlechter evaluieren als die damit verbundenen Verluste

Vorgehensweise

- Vorbereitung
  - Konzipierung der Evaluationsszenarien
  - Dokumentation der Evaluationsszenarien, -umgebung
  - Formulierung einer Erwartungshaltung
- Durchführung
  - Abarbeitung der vorbereiteten Szenarien
  - Sammlung der Messergebnisse
- Nachbereitung
  - Aufbereitung der Ergebnisse (z. B. in Diagrammen)
  - Beschreibung der Ergebnisse (textuell)
  - Interpretation der Resultate
  - Abgleich der Resultate mit der Erwartungshaltung

Messungen

- Mögliche Fehlerquellen
  - Existenz einer Aufwärmphase mit atypischen Systemeigenschaften
  - Verfälschung von Messungen durch unbeabsichtigtes Caching
  - Erhöhte Netzwerklatenzen aufgrund außergewöhnlicher Lastsituationen
  - Verzögerungen durch Log- bzw. Debug-Ausgaben
  - Beeinflussung des Systems durch die Messung selbst
  - ...
- Maßnahmen zur Kompensation
  - Messungen später beginnen (nicht bereits ab dem Zeitpunkt 0)
  - Messungen mehrfach durchführen
  - Verwendung von externen Messgeräten/-programmen
  - Geschickte Wahl der Messgrößen, z. B. CPU-Zyklen statt Zeit
  - Passende Wahl der Analysegrößen bei der Nachbereitung, z. B. Median vs. arithmetisches Mittel

Integration von asynchronen Fernaufrufen

- Future-Schnittstelle

```
public interface VSFuture {
    public Object get() throws InterruptedException,
                               ExecutionException;
    public boolean isDone();
}
```

- Vereinfachungen gegenüber dem Java-Original
  - Kein `cancel()`
  - Keine Generics (→ einheitlicher Rückgabewert vom Typ `Object`)

Integration von asynchronen Fernaufrufen

Anforderungen an die Implementierung

- **Keine Änderungen auf Server-Seite des Fernaufrufsystems**
  - Server-Seite beantwortet Anfragen auf identische Weise, unabhängig davon, ob der ursprüngliche Methodenaufruf synchron oder asynchron erfolgte
  - Client-Seite kapselt komplette Funktionalität für asynchrone Fernaufrufe
- **Anwender legt alleinig fest, ob ein Aufruf synchron oder asynchron durchgeführt werden soll**
  - Pro Remote-Objekt: Zusätzliche Hilfsschnittstelle für asynchrone Methodenaufrufe, die nur auf Client-Seite bekannt sein muss
  - Client-Seite des Fernaufrufsystems lenkt jeden asynchronen Aufruf intern auf den korrespondierenden synchronen um

  → Server-Seite kann unverändert bleiben

Integration von asynchronen Fernaufrufen

Synchron oder asynchron?

- Syntax
  - Kennzeichnung asynchroner Schnittstellen und Methoden durch das Postfix „`Async`"
  - `<Asynchroner Name> = <Synchroner Name>Async`
- Beispiel
  - Reguläre (synchrone) Remote-Schnittstelle

```
public interface RemObj extends VSRemote {
    public void sayHello();
    public int multiply(int a, int b);
}
```

  - Asynchrone Hilfsschnittstelle

```
public interface RemObjAsync extends RemObj {
    public VSFuture sayHelloAsync();
    public VSFuture multiplyAsync(int a, int b);
}
```

Integration von asynchronen Fernaufrufen

Synchron oder asynchron?

```
VSRemoteEntity re = new VSRemoteEntity();
[...]
```

- Synchroner Aufruf

```
RemObj proxy = (RemObj) re.lookup(host, port,
                                  RemObj.class);
proxy.sayHello();
System.out.println(proxy.multiply(4, 7));
```

- Asynchroner Aufruf

```
RemObjAsync proxy = (RemObjAsync) re.lookup(host, port,
                                            RemObj.class);
VSFuture helloFuture = proxy.sayHelloAsync();
VSFuture multiplyFuture = proxy.multiplyAsync(4, 7);

while(!helloFuture.isDone()) Thread.sleep(1000);
System.out.println(multiplyFuture.get());
```


Evaluation des eigenen Systems

Vorbereitung

- Konzipierung der Evaluationsszenarien
  - Vergleich mit anderen Ansätzen
    - Lokale Methodenaufrufe
    - Fernaufrufe mit Java-RMI
  - Vergleichsgrößen
    - Dauer eines Lookups
    - Antwortzeit eines Methodenaufrufs
    - Erreichbarer Durchsatz von Methodenaufrufen
- Formulierung einer Erwartungshaltung
  - [Vorschläge?]

Durchführung & Nachbereitung → Übungsaufgabe 6

Zeitmessung in Java

- Verfügbare Methoden (`java.lang.System`)
  - Aktuelle Zeit in Millisekunden

    ```
    public static long currentTimeMillis();
    ```

  - Aktuelle Zeit in Nanosekunden

    ```
    public static long nanoTime();
    ```

- Hinweise
  - Beide Methoden verwenden die Zeitmessung des Betriebssystems
    - Betriebssysteme messen Zeit in *Ticks* des System-Timers („*Jiffy*“)
    - Linux (allgemein): 1ms $<$ 1 Jiffy $<$ 10ms
    - Linux 2.6 auf i386: 1 Jiffy := 4 ms
  - Methoden brauchen selbst Zeit zur Ausführung

→ Die versprochene Granularität wird (evtl.) nicht erreicht!

Zeitmessung in Java

Auszug aus der Java-API zu `java.lang.System.nanoTime()`:

> “Differences in successive calls that span greater than approximately 292 years (2^{63} nanoseconds) will not accurately compute elapsed time due to numerical overflow.”