

## Rückrufe

### Evaluation von Systemen



## ■ Beispielszenario [Vergleiche Übungsaufgabe 1.]

### ■ Server-Seite

```
public interface VSBoard extends Remote {
    public void post(VSBoardMessage message);
    public VSBoardMessage[] get(int n);
    public void listen(VSBoardListener listener);
}
```

### ■ Client-Seite

```
public interface VSBoardListener extends Remote {
    public void newMessage(VSBoardMessage message);
}
```

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

→ Dem Server muss eine Referenz auf den Client vorliegen



## Probleme mit gewöhnlichen Referenzen bei Rückrufen

### ■ Lokaler Methodenaufruf

- Identischer Adressraum
- Referenz auch in aufgerufener Methode gültig

→ Rückruf 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

### „Rückruf“ 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



### Rückruf 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 Unterstützung durch das Betriebssystem möglich



### Call-by-Reference in objektorientierten Programmiersprachen

- Funktionsweise
    - Objekt kapselt Daten
    - Idealfall: Zugriff nur über Methodenaufrufe
    - Übertragung einer Remote-Referenz führt auf Server-Seite automatisch zur Erzeugung eines Objekt-Stub
    - Server kann transparent auf das Originalobjekt zugreifen
  - Einschränkung
    - Kein direkter Zugriff auf Objektzustand
    - z. B. keine „public“-Variablen
- Problem ohne spezielle Betriebssystem-Unterstützung lösbar



## Verwaltung von Rückruf-**{Stubs,Skeletons}**

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

→ Unnötig, falls die selbe Objektreferenz mehrfach übertragen wird
- Übliches Verfahren in Fernaufrufsystemen: Beidseitiger Einsatz von Hash-Tabellen
  - Client-Seite: Zuordnung lokaler Objektreferenzen auf Remote-Referenzen
  - Server-Seite: Abbildung von Remote-Referenzen auf Stubs

Wie lange sollen diese Informationen verfügbar gehalten werden?



## Freigabe von Stubs und Skeletons

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



- Jeder Server unterhält jeweils einen Remote-Referenzen-Zähler auf von ihm bereitgestellte Remote-Objekte
  - `dirty()`-Methode
    - Inkrementiert den Zähler
    - Aufgerufen vom Client bei Stub-Erzeugung (per Fernaufruf)
  - `clean()`-Methode
    - Dekrementiert den 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



- **Leases** im Kontext von Fernaufrufen
    - Garantie des Servers an den Client, dass ein bestimmtes Remote-Objekt für eine gewisse Zeit verfügbar ist
  - Leases in Java RMI
    - Standarddauer pro Lease: 10 Minuten
    - Rückgabewert von `dirty()`-Aufrufen
    - Verlängerung durch erneuten Aufruf von `dirty()`  
[Erfolgt üblicherweise nach Ablauf der Hälfte der Lease-Dauer.]
    - Ablauf eines Lease
      - Dekrementieren des entsprechenden Remote-Referenzen-Zählers
      - Bei Bedarf: Garbage-Collection des Stubs
- Leases stellen eine Absicherung des Servers gegen Verbindungsausfälle und Client-Abstürze dar



- Analyse des eigenen Systems
  - Leistungsfähigkeit
  - Antwortzeit
  - Durchsatz
  - Ressourcenverbrauch
  - Dienstgüte-Garantien
  - ...
- Vergleich mit anderen Systemen
  - Wie verhalten sich die unterschiedlichen Systeme in bestimmten Situationen?
  - Wo liegen die jeweiligen Stärken und Schwächen?
  - Ab welchen Punkten ist das eine bzw. das andere System besser?
  - ...



- Simulation
    - Messungen an einem Simulator, der das gewünschte Verhalten so gut wie möglich imitiert
    - Oftmals einfach zu realisieren
    - Ergebnisse spiegeln eventuell 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



## 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
    - Methoden brauchen selbst Zeit zur Ausführung
- Die versprochene Granularität wird (eventuell) nicht erreicht!

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

[Auszug aus der Java-Dokumentation zu `System.nanoTime()`]

