ulm university universität ulm

Ansätze für ein deterministisches Java Laufzeitsystem

Fachgruppe Betriebssysteme | 15.10.2010

Franz J. Hauck | Institut für Verteilte Systeme

© 2010, Franz J. Hauck | Institut für Verteilte Systeme | http://www-vs.informatik.uni-ulm.de/~hauck/

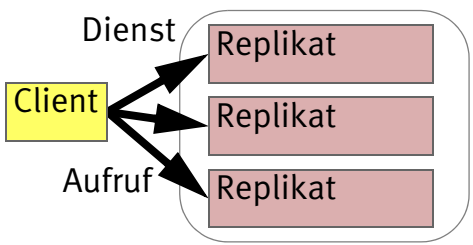
Kontext

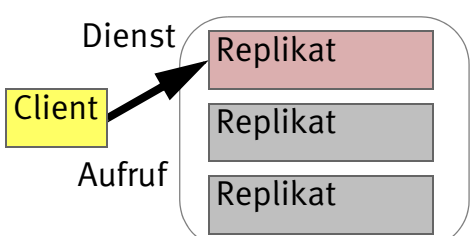
- fehlertolerante bzw. hochverfügbare Dienste
  - Anfrage-Antwort-Schema (RPC-ähnliche Interaktion)
- aktive oder passive Replikation
  - redundante Dienstinstallationen (Replikate)

© 2010, Franz J. Hauck | Institut für Verteilte Systeme | http://www-vs.informatik.uni-ulm.de/~hauck/

Kontext

- fehlertolerante bzw. hochverfügbare Dienste
  - Anfrage-Antwort-Schema (RPC-ähnliche Interaktion)
- aktive oder passive Replikation
  - redundante Dienstinstallationen (Replikate)
  - alle Replikate führen alle Anfragen aus (aktive Replikation)
  - ein ausgezeichnetes Replikat führt alle Anfragen aus und aktualisiert andere Replikate (passive Replikation)





© 2010, Franz J. Hauck | Institut für Verteilte Systeme | http://www-vs.informatik.uni-ulm.de/~hauck/

Kontext (2)

- zustandsbehaftete Dienste
  - im Gegensatz zu klassischer Organisation:
    - zustandslose Logik + (fehlertolerante) Datenbank
  - „günstige“ Randbedingungen:
    - Datenmenge passt in Hauptspeicher
    - keine komplexen Queries auf Daten
    - häufige nur-lesende Operationen
  - erhoffte Vorteile
    - geringere Kosten und größere Effizienz

© 2010, Franz J. Hauck | Institut für Verteilte Systeme | http://www-vs.informatik.uni-ulm.de/~hauck/

Kontext (3)

- aktive und passive Replikation fordern deterministische Ausführung
  - Sicherung der Konsistenz zwischen Replikaten
  - typisch: sequentielle, geordnete Ausführung von Anfragen

▲ Multicore-/Multiprozessorsysteme können nicht ausgenutzt werden

- geringe Performanz

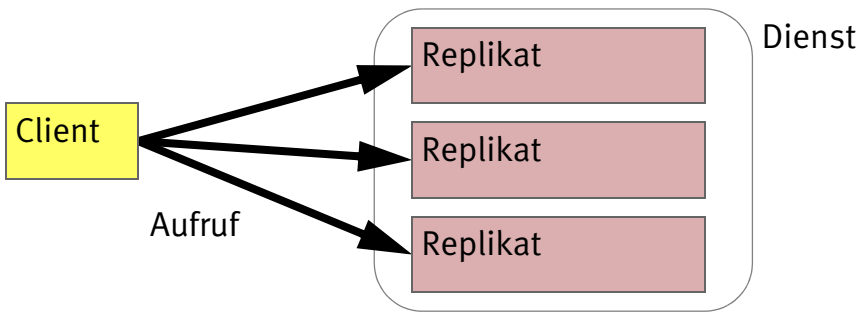
✱ **Angestrebt:** deterministische nebenläufige Ausführung

- Blick über diesen Kontext hinaus
  - deterministische nebenläufige Ausführung auch für Debuggingzwecke interessant

© 2010, Franz J. Hauck | Institut für Verteilte Systeme | http://www-vs.informatik.uni-ulm.de/~hauck/

Quellen für Indeterminismen

Szenario: aktiv replizierter Dienst mit mehreren Replikaten



- unterschiedliche Ankunftsreihenfolge von Anfragen
- unterschiedliche Ankunftsreihenfolge von Antworten verschachtelter Aufrufe
- unterschiedliches Scheduling der Anfrageausführung
- lokale Aufrufe indeterministischer Operationen

© 2010, Franz J. Hauck | Institut für Verteilte Systeme | http://www-vs.informatik.uni-ulm.de/~hauck/

Ankunftsreihenfolge

Problem: unterschiedliche Ankunftsreihenfolge von Anfragen

- Einsatz total geordneter Multicast-Systeme
  - viele verschiedene verfügbare Algorithmen und Systeme
  - Fehlertoleranz muss gewahrt bleiben
    - Ausfall eines oder mehrerer Replikate darf Rest nicht stören
  - z.B. Einsatz von Einigungsprotokollen wie Paxos

Ankunftsreihenfolge

Problem: unterschiedliche Ankunftsreihenfolge von Anfragen

- Einsatz total geordneter Multicast-Systeme
  - viele verschiedene verfügbare Algorithmen und Systeme
  - Fehlertoleranz muss gewahrt bleiben
    - Ausfall eines oder mehrerer Replikate darf Rest nicht stören
  - z.B. Einsatz von Einigungsprotokollen wie Paxos

Problem: unterschiedliche Ankunftsreihenfolge von Antworten verschachtelter Aufrufe

- Einsatz total geordneter Multicast-Systeme
  - Antworten verschachtelter Aufrufe werden mit den Anfragen in globale Ordnung gebracht

© 2010, Franz J. Hauck | Institut für Verteilte Systeme | http://www-vs.informatik.uni-ulm.de/~hauck/

Scheduling

Problem: unterschiedliches Scheduling der Bearbeitung

- sequentielle Bearbeitung
  - einfach aber wenig effizient
- nebenläufige Bearbeitung
  - erfordert deterministisches Scheduling
- typischer Ansatz:
  - sperrenbasierte Koordinierung nebenläufiger Aktivitäten
  - Scheduling garantiert gleiche Lockreihenfolge in allen Replikaten
    - zum Schutz vor Inkonsistenzen muss Programmierer in jedem Fall Sperren nutzen
  - Leader-Follower-Ansatz: LSA
  - deterministische Schedulingentscheidung ohne Kommunikation: PDS, MAT

© 2010, Franz J. Hauck | Institut für Verteilte Systeme | http://www-vs.informatik.uni-ulm.de/~hauck/

Indeterministische Operationen

Problem: Aufruf lokaler indeterministischer Operationen

- Problem liegt im Wesentlichen in den Bibliotheken des verwendeten Systems!
- Verwendung von Java: JDK

* Ziel: deterministisches Java-System

Bisherige Arbeiten

- EU-Projekt XtreemOS
  - Virtual Nodes Framework für hochverfügbare Dienste

- Details
  - Dienstzugang: RMI/SOAP/DIXI
  - Gruppenkommunikation: JGroups, AGC
  - Scheduling: alle bekannten Algorithmen implementiert, integriert mit Standard-Java-Koordination
  - Tool für das Patchen der Anwendung
    - Eclipse-Plugin
    - Umwandlung der Java-Koordinierung in Aufrufe an unseren Scheduler

Bisherige Arbeiten (2)

- Beispiel für Code-Umwandlung:

```
synchronized( f(a) ) {
   g();
}
```

wird zu

```
{ tmp= f(a);
  try {
     ourScheduler.lock( tmp );
     g();
  }
  finally {
     ourScheduler.unlock( tmp );
  }
}
```

Deterministisches Java

- Eliminierung von `synchronized`-Anweisungen im JDK
  - Patch-Vorgang erzeugt neue umbenannte Version der JDK-Klassen
  - gepatchte und ungepatchte Klassen koexistieren
  - in der Anwendung werden JDK-Klassen durch umbenannte Klassen ersetzt

▲ Problem

- Klassen mit besonderer Bedeutung verlieren ihre Bedeutung nach Umbenennung

▲ Problem

- manche Klassen sind inhärent indeterministisch

Verlust von Java-Semantik

- Beispiel `java.lang.Object`
  - implizite Basisklasse für alle Klassen
  - Lösung: alle Klassen ohne explizite Basisklasse erben von `vnode.java.lang.Object`
  - Implementierung von `hashcode()` ist indeterministisch
    - gepatchte Version enthält deterministische Implementierung
  - `hashcode()` an Array-Objekten muss beim Aufruf ersetzt werden
- Beispiel `java.lang.String, java.lang.Class`
  - werden automatisch durch Literale erzeugt: `"hallo", java.lang.Object.class`
  - sind unveränderlich und nutzen keine Koordinierung
  - bleiben ungepatcht

© 2010, Franz J. Hauck | Institut für Verteilte Systeme | http://www-vs.informatik.uni-ulm.de/~hauck/

Verlust von Java-Semantik (2)

- Beispiel `java.lang.RuntimeException, java.lang.Exception, java.lang.Error`
  - Beispiel: `RuntimeException` – Exception, die nicht deklariert werden muss
  - umbenannter Ersatz verliert diese Eigenschaft
  - Lösung: gepatchter Ersatz erbt vom Original
    - in abgeleiteten Klassen muss Basisklasse gegen gepatchte Version getauscht werden
- Beispiel `java.lang.Thread`
  - Originalklasse startet Aktivitätsträger
  - gepatchte Version erbt vom Original
    - in abgeleiteten Klassen muss Basisklasse ersetzt werden
    - gepatchte Version interagiert mit deterministischen Scheduler

© 2010, Franz J. Hauck | Institut für Verteilte Systeme | http://www-vs.informatik.uni-ulm.de/~hauck/

Indeterministische Klassen

- Beispiel: Sockets, Dateioperationen, jegliche I/O
  - dürfen im Replikat nicht genutzt werden
    - ähnliche Einschränkung wie beispielsweise in Google App Engine
  - Replikat darf verschachtelten Aufruf an anderen Dienst durchführen
    - z.B. Dateidienst
  - alternative Idee: vollständig im Replikat gekapselter Dateisystembereich
    - kein anderer Zugriff als durch das Replikat
    - Problem: I/O-Fehler, Out-of-Memory-Fehler

© 2010, Franz J. Hauck | Institut für Verteilte Systeme | http://www-vs.informatik.uni-ulm.de/~hauck/

Indeterministische Klassen (2)

- Beispiel: `System, Runtime`
  - Austausch gegen deterministische Implementierung mit ähnlicher Funktion
    - z.B. Ausgabeströme übergeben Zeichen in asynchron an Virtual-Node Runtime, welche die Daten ausgibt
  - Einschränkung mancher Funktionen
    - z.B. Security-Manager, plattformabhängige Operationen
- Beispiel: `random()`
  - Übergabe eines zufälligen Seeds durch die total-geordnete Gruppenkommunikation an alle Replikate
  - ausgetausche deterministische `random()`-Implementierung

© 2010, Franz J. Hauck | Institut für Verteilte Systeme | http://www-vs.informatik.uni-ulm.de/~hauck/

Indeterministische Klassen (3)

- Beispiel: native Methoden
  - `native synchronized` Methoden werden ersetzt durch Wrapper
  - native Methoden, die intern synchronisieren, müssen noch aufgespürt werden
  - Idee: Wrapper-Code für die nativen Synchronisierungsaufrufe
    - Notifizieren des deterministischen Schedulers
  - zur Zeit noch unklar, wieviele native Methoden wirklich nötig sind
    - viele gehören zur I/O und werden daher sowieso nicht gebraucht

© 2010, Franz J. Hauck | Institut für Verteilte Systeme | http://www-vs.informatik.uni-ulm.de/~hauck/

Verbleibende Probleme

- Klassenhierarchie von `enum`-Objekten
- Java-Memory-Model

```
sharedA= 0; sharedB= 0;
```

Thread 1	Thread 2
`localL= sharedA;`	`localR= sharedB;`
`sharedB= 42;`	`sharedA= 21;`

  - welche Werten haben `localL` und `localR`?

Verbleibende Probleme

- Klassenhierarchie von `enum`-Objekten
- Java-Memory-Model

```
sharedA= 0; sharedB= 0;
```

Thread 1	Thread 2
`localL= sharedA;`	`localR= sharedB;`
`sharedB= 42;`	`sharedA= 21;`

  - welche Werten haben `localL` und `localR`?
  - **Problem:** fehlerhafte Koordinierung
    - in nicht-repliziertem Code kann dieser Fehler unentdeckt bleiben
    - in repliziertem Code führt er zu sehr wahrscheinlichen Inkonsistenzen
  - Lösungsidee: Eclipse-Plugin überprüft korrekte Koordinierung soweit möglich
    - Warnhinweise

© 2010, Franz J. Hauck | Institut für Verteilte Systeme | http://www-vs.informatik.uni-ulm.de/~hauck/

Zusammenfassung

- deterministische Java-Bibliotheken
  - schwierig, aber mit Einschränkungen möglich
- spezielle Sandbox-Umgebung für Replikate in hochverfügbaren Diensten
- Nebeneffekt: Überprüfung zur korrekten Programmierung gemäß dem Java-Memory-Model
- Anwendbarkeit der Ergebnisse z.B. für Debugerleichterungen müssen noch untersucht werden

© 2010, Franz J. Hauck | Institut für Verteilte Systeme | http://www-vs.informatik.uni-ulm.de/~hauck/