HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

ifa Institute for Internet Technologies and Applications

Object Caching
Project Proposal

Lukas Hofmaier, Raphael Kohler, Timon Brüllmann

24. Mai 2012

1 Betreuer

J.M.Joller Abteilung für Informatik HSR/FHO

2 Motivation

Mercury ist ein Datenverarbeitungssystem, welches es Clients ermöglicht, Methoden von Objekten über das Netzwerk aufzurufen. Dabei entstehen unter anderem folgende zwei Problembereiche, auf welche das Augenmerk dieser Studienarbeit gelegt wird:

- Clients können Werte von Instanzvariablen von Objekten auf dem Server lesen. Die Clients können diese Werte im Hauptspeicher abspeichern und zu einem späteren Zeitpunkt in einer kausal abhängigen Methode als Argument einsetzen. Dabei kann ein Lost Update auftreten.
- Alle Methodenaufrufe werden über das Netzwerk an den Server gesendet. Die Methodenaufrufe auf den Clients sollen schneller werden.

3 Ziele der Arbeit

Die Arbeit wurde durch Mercury motiviert, unserer Ziele sind jedoch losgelöst von diesem System, da die genannten Probleme bei allen verteilten Software-Systemen auftreten.

3.1 Prioritäten

1. Das "Lost Update"-Problem soll vermieden werden:

- Instanzvariablen eines Objekts können durch mehrere Clients modifiziert werden. Liest ein Client den Wert eines Datenfeldes und verwendet er diesen Wert später als Argument in einer kausal abhängigen Schreibmethode, so muss sichergestellt werden, dass der Wert zwischen der Lese- und der Schreiboperation nicht von einem anderen Client verändert wurde. Wird dies nicht sichergestellt, kann es dazu führen, dass Änderungen, welche ein anderer Client zwischenzeitlich gemacht hat, verloren gehen.

2. Die Abhandlung der Schreib -und Lesezugriffe auf das Objekt, welches auf dem Server liegt, soll für den Client möglichst schnell passieren.

3.2 Hauptziele

3.2.1 RMI mit Concurrency Control

In dieser Semesterarbeit soll ein Konzept erarbeitet werden, welches Lost Updates bei kausal abhängigen Methoden verhindert. Zudem soll ein vereinfachtes RMI System implementiert werden, welches zeigen soll, ob sich dieses Konzept einfach realisieren lässt.

3.2.2 Testframework

Um das System zu testen, wird parallel dazu ein Testframework entwickelt. Dieses Framework soll in der Lage sein, unterschiedliche RMI Systeme als Server und Clients zu starten. Das Testframework ist in der Lage, eine Szenariobeschreibung einzulesen und dieses Szenario mit dem zu testenden System durchzuspielen. Ein Szenario lässt sich leicht spezifizieren und besteht aus einer Abfolge von Methoden-Aufrufen, welche der jeweilige Client tätigen muss. Das Framework sammelt dabei Messdaten über die Zeitdauer der einzelnen Methodenaufrufe. Die Messdaten sollen es ermöglichen abzuwägen, welche Systeme mit welchen Parameter sich für welchen Anwendungszweck eignen. Das Testframework registriert das Auftreten von Schreib-Konfikten beim Aufruf von Schreibmethoden. Alle Messdaten werden in Dateien gespeichert, was eine spätere Interpretation der Resultate ermöglicht.

3.3 Erweiterte Ziele

3.3.1 Object Caching

Um die Dauer von Methodenaufrufen zu verkürzen, soll das RMI System um einen Object Cache erweitert werden. Methoden ohne Seiteneffekte werden auf den Objekten im Cache ausgeführt. Für das System wird ein passendes Konsistenzmodell ausgewählt. Das Konsistenzmodell Einschränkungen für Leseoperationen. Es definiert Regeln, die definieren, wie aktuell Werte sind, die bei einer Leseoperation zurückgegeben werden. Dieses Modell wird durch ein geeignetes Konsistenzprotokoll durchgesetzt. Der User muss die Methoden eines Objekt

markieren, damit das System zwischen Methoden mit Seiteneffekt und solchen ohne Seiteneffekt unterscheiden kann. Methoden die Seiteneffekte aufweisen, werden auf dem Server ausgeführt, die Restlichen können direkt auf dem lokal zwischengespeicherten Objekt ausgeführt werden.

3.3.2 Entwicklung neuer Szenarien

Um die Lösungen mit und ohne Cache miteinander vergleichen zu können, müssen neue Testszenarien geschrieben werden können. Neue Testszenarien können über eine XML-Beschreibung spezifiziert werden.

3.3.3 Feingranulares Locking

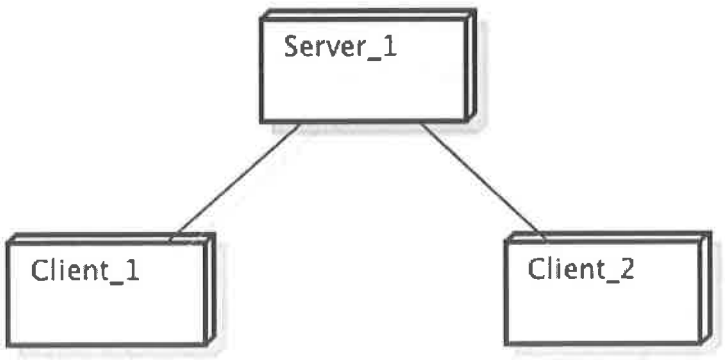
Der Concurrency-Mechanismus soll es ermöglichen, dass Anfragen parallel ausgeführt werden können.

3.4 Szenario

Als Ausgangslage für das gesamte Projekt wird folgendes Szenario eingesetzt:

3.4.1 Deployment

Es wird ein verteiltes System mit einem Server und zwei Clients verwendet.



3.4.2 Kausaler Zusammenhang

Die Clients möchten den Kontostand um 10 % erhöhen. Dazu führen die Clients folgende Schritte aus:

```
1. b = getBalance();
2. setBalance( b * 1.1 );
```

Diese Schritte werden zum Prozess "Kontostand Erhöhen" zusammengefasst.

3.4.3 Account Typ

Auf einem Server wird ein Account-Objekt instanziert Die Zugriffssynchronisation wird nicht über den in Java eingebauten Mechanismus synchronized direkt auf den Methoden realisiert. Das RMI System ist für die Zugriffssynchronisation verantwortlich. Das Interface von Account sieht wie folgt aus:

```
public interface Account {
    public int getBalance();
    public void setBalance(int balance);
}
```

Zwischen getBalance und setBalance besteht ein kausaler Zusammenhang: setBalance darf nur ausgeführt werden, wenn das Argument balance aktuell ist. Ist der Wert nicht mehr aktuell, wird der "Kontostand-Erhöhen"-Prozess abgebrochen. Der Client wiederholt in diesem Fall den gesamten Prozess (aktuelle Daten holen, Daten schreiben) automatisch.

Dem Client ist das Interface Account bekannt. Das Interface wird vor Prozessstart deployed. Zwei Clients beschaffen sich eine Referenz, in Form eines Proxy-Objektes, auf das Account-Objekt. Das Proxy-Objekt leitet Methodenaufrufe des Clients an den Server weiter und nimmt Rückgabewerte vom Server entgegen. Der entfernte Methodenaufruf ist für den Client transparent.

3.4.4 TestFramework

Bei jedem Szenario werden folgende Messdaten festgehalten.

- durchschnittliche Dauer des Methodenaufrufs getBalance()
- durchschnittliche Dauer des Methodenaufrufs setBalance()
- Anzahl aufgetretene Konflikte
- Dauer eines setBalance(); Methodenaufrufs im Falle einer Konfliktsituation
  - Zeitdifferenz zwischen erstem setBalance bis erfolgreichem setBalance
  - Zeitdifferenz zwischen setBalance und Empfang der zugehörigen Exception

3.4.5 Test-Szenarien

3.4.5.1 Konfliktloser Zugriff

Das Szenario soll zeigen, wie lange Methodenaufrufe dauern, wenn keine Konflikte auftreten. Ein Client führt den Prozess Kontostand-Erhöhen 10000-mal aus.

3.4.5.2 Race Condition

Das Szenario soll zeigen wie viele Konflikte auftreten, wenn beide Clients permanent Kontoerhöhungen ausführen wollen. Beide Clients führen den Prozess Kontostand-Erhöhen 10000 mal aus. Der Server startet die Clients, mit einer möglichst kleinen Verzögerung, hintereinander.

3.4.5.3 Erzwungene Konflikte

In diesem Szenario soll jeder setBalance-Aufruf von Client2 zu einer Konfliktsituation führen. Damit sind Konfliktsituation nicht dem Zufall überlassen, sondern treten zuverlässig auf und können besser analysiert werden.
Dies wird folgendermassen erreicht:

- Client1 führt ununterbrochen Kontoerhöhungen aus
- Client2 führt 100 Kontoerhöhungen aus. Nach dem getBalance()-Aufruf wartet Client2 10ms
- Währenddessen führt Client1 eine Schreiboperation aus
- Dadurch werden die Daten im Speicher von Client2 veraltet
- Client2 führt nun eine Schreiboperation aus und erhält eine Konfliktmeldung

4 Sitzungen

Mit dem Betreuer dieser Arbeit findet eine wöchentliche Sitzung statt. Dabei werden die Fortschritte besprochen und falls vorhanden, offene Fragen diskutiert. Die Länge der Sitzung ist daher variabel, je nachdem ob und wieviele offene Fragen und Probleme bestehen. Die Entscheidungen, sowie die diskutierten Punkte, werden in einem Sitzungsprotokoll schriftlich festgehalten.

Rapperswil, den 24. Mai 2012