Studienarbeit

Object Caching

Anforderungsspezifikation

Projektleiter	20.02.2012	Lukas Hofmaier / Raphael Kohler / Timon Brüllmann
Betreuer	20.02.2012	Prof. Dr. Josef M. Joller / HSR

Inhaltsverzeichnis

Testframework-Anforderungen

Funktionalität

Der Anwender kann ein definiertes Testszenario von einen Gerät aus starten. Alle nötigen Prozesse auf anderen Hosts werden automatisch gestartet. Das System sammelt die Reports an zentraler Stelle und macht sie dem Anwender zugänglich.

Der Server kann den Testablauf auf den Clients synchronisieren. z.B. mit Semaphoren.

Konfigurationsmöglichkeiten

- Die Konfiguration des Tests erfolgt über eine externe Datei (z.B. xml)
- Anzahl Reads und Writes soll konfiguriert werden können
- Ausserdem kann man die Zeit zwischen Object-Fetch und Method-Invocation einstellen.
    - Das sollte zu mehr Konflikten führen und eine realistischere Systemumgebung simulieren.
    - Zeit zwischen Get/Invoke (es sollen auch konfliktfreie Szenarien simuliert werden können)
- Es soll möglich sein eine Netzwerklatenz einzustellen
- N-Clients instanziieren
- Variable Anzahl von Objekten auf dem Server installieren
- Das Framework kann dem ClientSystemUnderTest direkt eine Collection von Account-Objekten übergeben.
- Client sollten via Parameter konfigurierbar sein
    - Cache Size

Testcases

Der Server stellt den Clients Testcases zur Verfügung. Er kann unterschiedlichen Clients unterschiedliche Testcases mitgeben. Dies soll es ermöglichen einfach Konfliktsituationen zu provozieren.

Messenwerte

Das Testframework misst beim Client. Das Testframework solle die Zeit messen können für folgende Operationen.

- Read von Object.
- Write von Object.
- Konfliktbehandlung

Anzahl aufgetretene Konflikte werden angezeigt.
Client speichert Messresultate und sendet sie an den Server.

Simulation

Das Testframework beschafft sich einen Stub eines Objektes. Clientcode kann auf Variablen zugreifen. RMI greift über getfn auf Variablen zu. Getter sind auch RMI Calls.

Server

Schnittstelle Server RMI-System

Der Server erstellt einen Dispatcher, der auf einen definierten Port hört. Der Dispatcher installiert pro TCP Verbindung einen Handler. Der Handler implementiert Runnable. Der Dispatcher. Der Typ des Handlers wird mittels Class.forName geladen. Dem Handler werden ein InputStream und ein OutputStream übergeben.

```
socket = server.accept();
Class[] paramters = new Class[2];
paramters[0] = InputStream.class;
paramters[1] = OutputStream.class;
Constructor ctor = clazz.getConstructor(paramters);
Object[] ctorArgs = new Object[1];
ctorArgs[0] = socket;
Runnable runnable = (Runnable) ctor.newInstance(ctorArgs);
new Thread(runnable).start();
```

Offene Punkte

Idee: Externes Tool, welches Netzwerktraffic generiert und diesen auf einem fixen Level hält. Dadurch wird gemessen, wie sich der Netzwerkverkehr auf die Performance des Projekts auswirkt.

Testframework-Lösungsansätze

Kopieren der jar-Files

Die Jar Files werden via Linux auf zuvor in einer Liste definierte Rechner kopiert. Dazu wird scp gebraucht, zum Beispiel: scp HelloWorld.jar student@152.96.193.10:/home/student/HelloWorld.jar

Starten der Clients

Die Clients auf den verschiedenen Rechnern werden via SSH gestartet. Ein SSH-Login funktioniert zum Beispiel so: ssh student@152.96.193.10. Danach muss man in den Ordner, in welchen das jar-File abgelegt wurde und startet dort das jar-File

Ablauf des TestFrameworks

Der Ablauf des Frameworks sieht wie folgt aus:

- Start des BashScripts
- Kopieren der jar Files auf die Clients, durch das Script
- Script startet den Server
- Script startet die Clients via SSH(übergibt IP des Servers)
- Clients erhalten ein Szenario vom Server
- Clients instanziieren das benötige ClientSystemUnderTest
- Clients warten auf „go“ des Servers
- Tests
- Clients schicken Messwerte an den Server

RMI-only Anforderungen

Einschränkungen

Der User des Systems muss dem System mitteilen, welches read-Operationen und welches write-Operationen sind. Ausserdem muss er dem System mitteilen die kausalen Abhängigkeiten zwischen den Methoden mitteilen.

Client

Das RMI-only System stellt dem Client folgende Schnittstelle bereit.

```
interface Account{
   int getBalance();
}
```

Das System stellt dem Client eine Schnittstelle zur Verfügung um sich Referenzen auf Account Objekte zu beschaffen.

```
class AccountService{
   Account getAccount(int accountID);
}
```

Ruft ein Client eine Methode auf einem Account Objekt auf, werden diese auf dem Server ausgeführt. Netzwerkadresse des Servers ist fix. Der Client greift immer auf dieselbe IP zu.

Server

Der Server nimmt Nachrichten die Methodenaufrufe beinhalten entgegen, packt sie aus und leitet die Remote Method Invocation an das richtige Objekt weiter. Der Rückgabewert der Methode wird wieder in eine Nachricht verpackt und an den Client zurückgesendet.

Das System stellt einen ClientHandler mit einem Konstruktor der einen Socket als Parameter nimmt zur Verfügung. Dieser Clienthandler implementiert Runnable. Das komplette System muss mit dem Clienthandler instanziiert werden.

Concurrency Control

Der Server enthält Logik, das "Lost Update" - Problem verhindert.
Das "Lost Update"-Problem kann wie folgt beschrieben werden.

Transaction T	**Transaction U**
balance = b.getBalance(); $200	
	balance = b.getBalance(); $200
	b.setBalance(balance * 1.1); $220
b.setBalance(balance * 1.1); $220	

Wollen zwei Client auf dem Server dem Account b Geld hinzufügen, muss beim setBalance Aufruf dafür gesorgt werden, dass das Argument balance dem aktuellen Wert auf dem Server entspricht.

Beim Ausführen von setBalance muss sichergestellt werden, dass sich seit dem letzten getBalance-Aufruf des Clients nichts mehr verändert hat.

RMI-only-Lösungsansätze

Nachrichtenübermittlung

Nachrichten sind serialisierte Java Objekte. Ein Methodenaufruf ist beispielsweise ein Objekt der Klasse MethodCall.

Konflikterkennung

Auf dem Server werden Hash-Maps angelegt, welche als History dienen. Dabei wird eine Logik implementiert, welche mit Hilfe der Hash-Maps feststellen kann, ob bei einem Schreibzugriff ein Konflikt entstanden ist oder nicht.

Es gibt zwei mögliche Lösungsansätze:

- Es werden insgesamt zwei Hash-Maps implementiert:
  - In der einten Map führt jeder Schreibzugriff auf ein Objekt zur Änderung des zugehörigen Eintrages in der Hash-Map. In diesem Eintrag wird gespeichert, welches Objekt(Als Key in der Map, Name?) verändert wurde und zusätzlich eine inkrementierte Ganzzahl, welche sich fortlaufend erhöht und eine Art "Schreib-Version" darstellt. Das heisst zum Beispiel, es liegt ein Eintrag in der Map vor, welcher beschreibt, dass Objekt o1 bereits 5 Mal verändert wurde. Nach einem erneuten Schreib-Zugriff auf das Objekt, wird der bestehende Eintrag geändert, sprich die Ganzzahl in der Map lautet nun 6.
  - In der zweiten Hash-Map wird jeder Lese-Zugriff festgehalten. Wird auf ein Objekt ein Lese-Zugriff ausgeführt, wird in der Map der entsprechende Eintrag geändert. Bei einem Lese-Zugriff wird gespeichert, von wem, welches Objekt gelesen wurde. Zusätzlich wird die "Schreib-Version" des gelesenen Objektes aus der ersten Hash-Map gelesen und hier abgelegt. Somit ist bekannt, welche Version eines Objektes der Client gelesen hat, wenn er einen Schreibzugriff ausführen möchte.

Das Problem bei dieser Variante ist, dass es zum Phänomen "Lost Update" kommen kann. Wenn man das Fibonacci-Beispiel heranzieht, könnte folgendes passieren:

1. Client1 liest die Zahl x1 aus dem Stub, welcher er vom Server erhalten hat.
2. Client2 führt einen Schreibzugriff auf das Fibonacci-Objekt aus.
3. Client1 liest nun den Wert x2 aus dem Stub
4. Client1 möchte nun einen Schreibzugriff ausführen und der Server wird dies erlauben. Da die Lese-Operation von x2 den Eintrag in der Has-Map dahingehend geändert hat, dass Client1 nun die neuste Version besitzt, sieht der Server den Konflikt nicht, obwohl Client1 eine alte(!) Version von x1 besitzt. Client1 kann nun einen Schreibzugriff ausführen, obwohl dies eigentlich abgebrochen werden müsste.

- Dieses Problem kann behoben werden, wenn alle Datenfelder eines Objektes nur gemeinsam herausgelesen werden können. Das heisst es ist nicht möglich, dass wie beim Fibonacci-Beispiel, die Datenfelder x1 und x2 separat ausgelesen werden können(d.h. die Getter-Methode gibt ein Array mit den beiden Zahlen zurück). Dies führt dazu, dass alle gelesenen Datenfelder das gleiche "Lese-Alter" aufweisen. Auf

diese Weise sind Fehler, wie oben beschrieben, nicht möglich. Problematisch bei diesem Lösungsansatz ist aber, dass mehr als nötig übertragen wird und somit der Netzwerk-Traffic steigt, sowie die Performance im allgemeinen leidet.

- Es werden mehrere Hash-Maps benötigt, je eine pro Datenfeld und zusätzlich eine pro Objekt:
    - Jeder Lesezugriff auf ein Datenfeld führt zur Aktualisierung des Eintrages in der Hash-Map des jeweiligen Datenfeldes. Im bekannten Fibonacci-Beispiel wird somit auf dem Server für die Datenfelder x1 und x2 je eine separate Hash-Map geführt. Somit kann ein Konflikt detektiert werden, auch wenn das gleiche Szenario wie oben beschrieben, auftritt. Zusätzlich wird pro Objekt eine weitere Hash-Map benötigt, welche die Schreibzugriffe auf das Objekt festhält.

Konfliktbehandlung

Wird ein Konflikt detektiert, gibt es verschiedene Möglichkeiten darauf zu reagieren:

- Man bricht den Schreibzugriff ab, das heisst, es werden keine Änderungen vorgenommen und der Client wird durch eine Message(?) informiert, dass seine Änderungen nicht wirksam waren.

Ablauf

Ein Stub eines Fibonacci-Objektes, welches auf dem Server liegt, wird per fetch() dem Client zur Verfügung gestellt. Dieser kann getter-Methoden, welche keine Seiteneffekte beinhalten, ausführen.

Szenario

Auf einem Server wird ein Account-Objekt instanziiert. 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) noch automatisch.

Dem Client ist das Interface Account bekannt. Das Interface wird vor Prozessstart deployed. Zwei Client beschaffen sich eine Referenz in Form eines Proxy-Objektes auf das Account-Objekt. Messdaten werden für mehrere Szenarios gesammelt.

Bei jedem Szenario werden folgende Messdaten festgehalten.

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

Konfliktloser Zugriff

Das Szenario soll zeigen wie lange Methodenaufrufe dauern, wenn keine Konflikte auftreten. Ein Client möchte den Kontostand um 10% erhöhen. Er führt dazu folgende Schritte aus:

```
b = getBalance();
setBalance( b * 1.1 );
```

Dieser Vorgang wird 10000-mal wiederholt.

Race Condition

Das Szenario soll zeigen wie viele Konflikte auftreten, wenn beide Clients permanent Kontoerhöhungen ausführen wollen.

Beide Clients erhöhen den Kontostand um 10%. Jeder für sich führt dieselben Schritte wie beim Szenario "Konfliktlos" aus. Der Server startet die Clients hintereinander mit einer möglichst kleinen Zeitdifferenz.

Dieser Vorgang wird 10000-mal wiederholt.

Erzwungene Konflikte

In diesem Szenario soll jeder setBalance-Aufruf von Client 2 zu einer Konfliktsituation führen. Damit sind Konfliktsituation nicht dem Zufall überlassen, sondern treten zuverlässig auf und können besser analysiert werden.

Client 1 führt ununterbrochen Kontoerhöhungen aus. Client 2 führt 100 Kontoerhöhungen aus. Client 2 wartet nach dem getBalance()-Aufruf 10ms. In dieser Zeit sollte Client1 eine Schreiboperation ausführen und die Daten im Speicher von Client 2 werden veraltet.

Szenarien für Object Caching

Für das Object Caching wäre es sicher auch interessant Szenarien mit unterschiedliche read / write Verhältnissen zu testen. Beispiel. Clients lesen nur Daten. Client 1 liest nur. Client 2 schreibt nur. Wenn dieses Szenario langsamere mit Object Caching ist, ab welchem Verhältnis lohnt sich Object Caching.