HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

Institute for
Internet Technologies
and Applications

Object Caching

Studienarbeit

Abteilung Informatik
Hochschule für Technik Rapperswil

Frühjahrssemester 2012

Autoren:	**Lukas Hofmaier** (lhofmaie@hsr.ch)
	Raphael Kohler (rkohler@hsr.ch)
	Timon Brüllmann (tbruellm@hsr.ch)
Betreuer:	**Prof Dr. Josef M. Joller** (jjoller@hsr.ch)
Projektpartner:	**Georg Troxler**, staila GmbH, http://www.staila.com
Experte:	**Prof Dr. Josef M. Joller** (jjoller@hsr.ch)
Datum:	**29. Mai 2012**

Erklärung

Ich erkläre hiermit,

- dass ich die vorliegende Arbeit selber und ohne fremde Hilfe durchgeführt habe, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt ist oder mit dem Betreuer schriftlich vereinbart wurde,
- dass ich sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben habe.

Rapperswil, 30. Mai 2012

_____

Lukas Hofmaier

_____

Raphael Kohler

_____

Timon Brüllmann

Abstract

Bei der Firma staila technologies wird das Datenverarbeitungssystem Mercury entwickelt, welches Clients ermöglicht, Methoden von Objekten, die auf dem Server sind, über das Netzwerk aufzurufen. Um „Lost Update“-Fehler zu verhindern und die Performance durch Objektreplikation zu steigern, wurde ein Konzept entwickelt.

Anhand des Konzeptes wurden zwei Prototypen erstellt. Beim ersten Prototypen handelt es sich um ein vereinfachtes RMI-System, welches eine „Concurrency control“ beinhaltet. Um „Lost Updates“ zu verhindern, wird das Verfahren „Optimistic Concurrency“ eingesetzt.

Beim zweiten Prototypen wurde zusätzlich zur „Concurrency Control“ ein lokaler Cache implementiert.

Um die Funktionalität und die Performance der Prototypen zu vergleichen, wurde ein Test-Framework implementiert.

Mit Hilfe des Frameworks wurden mehrere, praxisbezogene Szenarien durchgespielt. Gemessen wurden dabei die Zeiten, welche zur Ausführung der Lese – und Schreiboperationen gebraucht wurden, sowie die Anzahl der aufgetretenen Konflikte.
Grundsätzlich kann gesagt werden, dass sich das Cache-System bei allen Szenarien als sehr gut bewährt hat, in welchen viele Lesezugriffe getätigt werden mussten. Der Cache bietet bei Lesezugriffen eine Leistungssteigerung um den Faktor 400. Dies war zu erahnen, da der Zugriff auf den lokalen Speicher natürlich viel schneller ist, als der Zugriff über das Netzwerk.

Bei permanent nur schreibenden Clients, skaliert das System ohne Cache hingegen besser. Bei acht Clients erreicht es bei den schreibenden Zugriffen eine Leistungssteigerung gegenüber dem System mit Cache um den Faktor 170. Die Zeit, um einen schreibenden Zugriff abzuwickeln, ist bei diesem Sy-

stem immer gleich gross. Beim System mit Cache, steigt die benötigte Zeit für schreibende Zugriffe bei steigender Anzahl Clients, sehr stark an.

Management Summary

Ausgangslage

Die Firma staila technologies entwickelt ein neuartigs Datenverarbeitungssystem mit dem Namen Mercury. Die höchste Priorität bei der Entwicklung kommt der Performance des Systems zu. Um dies zu erreichen, wurden neue Ideen und Konzepte entworfen, welche die heutigen, üblichen Vorgehensweisen verlassen und einen neuen Weg gehen.

Diese Arbeit fügt ein weiteres Puzzleteil hinzu. Um eine möglichst performante Kommunikation zwischen Client und Server in Mercury herstellen zu können, gibt es verschiedenste Algorithmen, welche für verteilte Systeme geeignet sind. Das Ziel dieser Arbeit ist für viel versprechende Ansätze einen Prototypen zu entwickeln und diesen dann zu testen. Aus den verschiedensten Tests soll dann ersichtlich werden, welcher Algorithmus sich für Mercury am besten eignet.

Vorgehen / Technologien

Anfangs wurde ein System mit integrierter Concurrency control gebaut. In einem nächsten Schritt wurde ein Prototyp gebaut, welcher zusätzlich einen lokalen Cache besitzt. Mit beiden Prototypen wurden Zeitmessungen durchgeführt und die Werte wurden miteinander verglichen.

Parallel zur Entwicklung der oben genannten Prototypen wurde ein Framework entwickelt, welches die Systeme testet. Es wurde dabei darauf geachtet, dass die Prototypen ohne Anpassung von Programmcode an das Framework angehängt werden können. Das Framework ist für die Zeitmessungen und die Ausgabe der Resultate, sowie für viele weitere Aufgaben, verantwortlich.

Alle Prototypen und das Framework wurden in Java entwickelt. Da keine andere Programmiersprache gewünscht wurde, und wir am meisten Erfahrung mit dieser Sprache gesammelt haben, war dies eine logische Wahl.

Ergebnisse

Das Ziel dieser Arbeit, zwei Prototypen und ein Framework zu entwickeln, wurde erreicht. Die Probleme, welche sich während der Entwicklungsarbeit stellten, werden in den folgenden Seiten dokumentiert. Weiter wurden die Entscheidungen und die Gründe, warum genau dieser Weg eingeschlagen wurde, genauestens dokumentiert.

Durch diese Arbeit wurde bewiesen, dass es sich lohnt, einen Cache einzubauen, wenn es zu vielen lesenden Zugriffen kommt. Falls im Einsatzszenario alle Clients permanent nur schreiben müssen, lohnt sich ein Cache nicht. Für diese Art der Anwendung skaliert ein System mit Cache einfach zu schlecht.

Auf der anderen Seite wurde ein System getestet, welches ähnlich wie Java RMI funktioniert, aber zusätzlich mit einer Concurrency Control versehen wurde. Dieses System sendet alle Anfragen, egal ob lesend oder schreibend, über das Netzwerk. Diese Vorgehensweise beweist seine Stärke in einem Umfeld, in welchem dauernd geschrieben werden muss. Die lesenden Zugriffe sind im Vergleich zu einem System mit Cache extrem langsamer, bedingt durch die Übertragung über das Netzwerk.

Diese Arbeit ist als Grundlage für zukünftige Entscheidungen bezüglich des Einsatzes eines Caches zu sehen. Es können Antworten zu den gängigsten Problemen, welche sich bei der Entwicklung eines solchen Systems stellen, gefunden werden. Weiter kann aufgrund der aufgeführten Messdaten abgeschätzt werden, ob sich die Einbindung und der damit verbundene Aufwand für die Entwicklung eines Caches, lohnt.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Idee ein RMI-System mit integriertem Client-Cache zu entwickeln, ist bei der Arbeit an Mercury entstanden. Mercury ist ein Datenverarbeitungssystem. Clients können sich an einen Server anmelden und Methoden auf Objekten ausführen, die auf dem Server vorhanden sind.

Führen mehrere Clients Methoden der gemeinsam genutzten Objekte aus, kann es vorkommen, dass Daten unabsichtlich überschrieben werden. Dieser Fehler kann auftreten, weil Clients Daten über Methoden auslesen und diese Daten später in Methoden mit Schreibzugriff weiterverwenden, obwohl die Daten inzwischen veraltet sind.

Ein Ziel dieser Arbeit ist es, ein Konzept zu entwickeln, um diese Fehler zu verhindern. Anhand dieses Konzeptes wird ein Prototyp diese Systems erstellt. Dieses System ist eine vereinfachte Implementation von RMI und enthält zusätzlich Mechanismen, die Lost Update Fehler verhindern. Der Prototyp soll unter anderem zeigen, ob das Konzept funktioniert.

Ein weiteres Ziel der Arbeit ist die Verbesserung der Performance in einer RMI Umgebung durch Replikation der Objekte beim Client. Objekte sollen näher beim Prozess, der darauf zugreift, platziert werden, um die Zugriffszeit zu verkürzen. Deshalb wurde in einem zweiten Schritt ein RMI-System entwickelt, welches Objekte bei den Clients lokal speichert. Lesezugriffe können vom Cache verarbeitet werden.

Mit der Replikation von Objekten entstehen neue Probleme. Gibt es mehrere Kopien von Objekten, müssen Änderungen an einer Kopie an die anderen Kopien weitergegeben werden. Dieser Aktualisierungsprozess kann die Performance wiederum mindern, da ein Server oft damit beschäftigt ist, Kopien zu aktualisieren. Es wurde ein Konzept erstellt, wie ein Caching von Objekten in einem RMI-System realisiert werden kann. Um die Funktionalität zu testen und die Performance zu messen, wurde das Konzept implementiert.

Ein weiteres Problem im Zusammenhang mit Objektcaching ist, dass das

System mehr Netzwerkbandbreite benötigt, da es mehr Nachrichten versendet, um die Kopien aktuell zu halten.

Um die Vor -und Nachteile der beschriebenen Konzepte zu finden, muss ein Test-Framework geschrieben werden. Dieses Framework soll Zeitmessungen der einzelnen Methodenaufrufe ermöglichen, was Fehler in den Konzepten schonungslos aufdecken wird. Das Framework soll also zeigen, in welchen Anwendungsfällen sich ein System mit einem Cache lohnt und in welchem Anwendungsfall dies purer Ballast wäre.

Weiter soll das Framework so entwickelt werden, dass neue, zu testende Systeme, einfach angehängt werden können, ohne dass Programmcode angepasst werden muss. Das Framework soll für den gesamten Testablauf eines der oben beschriebenen Systemen verantwortlich sein; vom Start des Servers und der Clients bis hin zum Herunterfahren dieser Komponenten.

Kapitel 2

RMI mit Concurrency Control

Führen mehrere Clients Methoden auf diesen Objekten auf dem Server aus, kann es zu sogenannten “Lost Update“ Fehlern kommen. Dieser Fehler und eine Lösung wird im Abschnitt 2.1 genauer beschrieben.

Um die Funktionalität der Lösung zu überprüfen, wurde ein vereinfachtes RMI-System implementiert. Zusätzlich zur RMI-Funktionalität enthält dieses System einen Mechanismus, um das Lost Update Problem zu verhindern. Die Implementation des RMI-Systems und die der Concurrency Control werden in den Abschnitten 2.2 und 2.3 erklärt.

Die Dauer, die Methodenaufrufe in diesem RMI-System benötigen, werden später mit einem zweiten RMI-System, welches Object-Caching einsetzt, verglichen. Auf diese Weise wird ersichtlich, welchen Performancegewinn Object-Caching bringt.

2.1 Concurrency control

Werden die Objekte auf dem Server von mehreren Clients verändert, kann es zum sogenannten “Lost Update“ Problem kommen. Angenommen zwei Clients, Client T und Client U, wollen den Kontostand von Konto A um 200 erhöhen: Wenn der Kontostand vor der Änderung 1000 beträgt, muss der Kontostand nach Abschluss der beiden Transaktionen 1400 betragen. Konto A wird im System mit einem Account-Objekt repräsentiert. Account bietet folgendes Interface:

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

Möchte man den Kontostand um 200 erhöhen, liest man mit `getBalance()` zuerst den akutellen Kontostand, addiert zu diesem 200 dazu und setzt den neuen Kontostand mit `setBalance()`. Beide Operationen kann man zu einer Transaktion zusammenfassen. Solange die Transaktionen sequenziel ablaufen, bleibt die Konsistenz erhalten. Laufen die Transaktionen auf dem Server gleichzeitig ab, können Inkonsistenzen auftreten. Lesen Client T und Client U den Kontostand nacheinander und addieren zum gelesenen Wert 200, um das Resultat mit `setBalance()` zu speichern, beträgt der Kontostand nach Abschluss beider Transaktionen 1200 statt 1400. Der Ablauf, der zum Lost Update Fehler führt, wird in Abbildung 2.1 illustriert.

Client T	**Client** U
balance = b.getBalance();	
	balance=b.getBalance();
b.setBalance(balance + 200)	
	b.setBalance(balance + 200);

Abbildung 2.1: Das Lost Update Problem

2.1.1 Optimistic Concurrency

Das System soll Lost Updates verhindern, indem es das Verfahren Optimistic Concurrency [5] einsetzt. Clients können jederzeit `setBalance()` aufrufen. Hat der Datensatz seit dem letzen Lesezugriff dieses Clients geändert, wird die Methode abgebrochen und der Client wird benachrichtigt. Er muss die aktuellen Daten selbständig mit `getBalance()`beim Server holen und kann es dann nochmals probieren. Hat zwischen dem letzen `getBalance()` und dem `setBalance()` kein anderer Client das Objekt verändert, läuft die Schreiboperation fehlerfrei durch.

2.2 Implementierung des RMI Systems

Das RMI-System wurde als Middleware implementiert. Clients fordern `Account`-Objekte von der Klasse `AccountService` über die Methode `Collection<Account> getAllAccounts()`. Mit diesen Account-Objekten - arbeitet der Client, wie mit lokalen Objekten, obwohl die Objekte auf einem entfernten Server in einer anderen “Virtual Machine“ leben. Die Middleware sorgt für Lokalitätstransparenz. Sie ermöglicht es, dass Objekte in unterschiedlichen Prozessen miteinander kommunizieren können. Methodenaufru-

fe werden in Nachrichten umgesetzt. Für den Nachrichtenaustausch wird ein Request-Reply Protocol eingesetzt [1].

2.2.1 Clientimplementation

Die Clients programmieren gegen das Interface `Account`. Dieses Interface wird von der Klasse `AccountStub` implementiert. Der Stub setzt alle Methodenaufrufe in Nachrichten um, die über einen TCP-Stream an den Server gesendet werden.

Nachrichten

Bei den Nachrichten handelt es sich um serialisierte Objekte des Typs `MethodCall`. `MethodCall` ist ein Data-Transfer-Object. Es enthält Informationen darüber, welche Methode aufgerufen werden soll, auf welchem Objekt die Methoden aufgerufen werden sollen und mit welchen Argumenten die Methode aufgerufen wird. Nachdem der Methodenauruf dem Netzwerkstream übergeben wurde, wartet der Client, bis er eine Antwort auf demselben Stream erhält. Die Antwort wird mit einem `ReturnValue`-Objekt übertragen.

Serialisierung

Objekte der Klasse `MethodCall` und `ReturnValue` müssen serialisiert werden, damit sie über einen TCP Stream zwischen Client und Server übertragen werden können. Die Java Plattform bietet dafür die Java Serialization API[2] an. Serialisierte Objekte werden mit dem Serializable Interface markiert. Möchte man die Objekte in einen Stream schreiben, übergibt man sie der Methode `writeObject(Object object)` eines `ObjectOutputStream`-Objekts.

Objekt Identifikation

Ruft der Client eine Methode auf einem `Account`-Objekt auf, muss er dem Server mitteilen, auf welchem Objekt die Methode aufgerufen werden soll. Dafür teilt er dem Server eine Objekt-ID im `MethodCall`-Objekt mit. Diese Objekt-IDs wurden künstlich eingeführt. Der Server ordnet jedem Objekt eine Objekt-ID zu. Diese Objekt-ID müssen in den Stubs auf Clientseite bei Methodenaufruf bekannt sein, und sie müssen mit den Objekt-IDs auf dem Server übereinstimmen. Der Client kann `Account`-Objekte beim Server anfordern. Das Interface `AccountService` bietet dafür die Methode `getAllAccounts()` an. Das `ReturnValue`-Objekt, das auf
`getAllAccounts()` vom Server an den Client gesendet wird, enthält eine

ArrayList von Objekt-IDs. Aus diesen IDs werden die AccountStubs auf dem Client generiert.

2.2.2 Serverimplementation

Für jede Klasse, die RMI-Methoden anbietet, hat der Server ein zugehöriges Skeletonobjekt. Für die Methodenaufrufe der Klasse `Account` ist ein Objekt vom Typ `AccountSkeleton` zuständig. Das Skeletonobjekt enthält alle `Account`-Objekte des Systems in einer HashMap. Die Objekt-IDs werden als Keys der HashMap eingesetzt. Der `AccountSkeleton` ist damit in der Lage ein `MethodCall`-Objekt, welches ein Client versendet hat, an das richtige Objekt weiter zu leiten.

Die Skeletonklasse setzt Java Reflection ein, um Methoden auf den Objekten aufzurufen. Das `MethodCall`-Objekt enthält ein Method-Objekt. Method stellt eine Methode `Object invoke(Object obj, Object[] args)` zur Verfügung. Das Objekt auf dem die Methode ausgeführt werden soll, beschafft sich der Skeleton aus der HashMap. Die Argumente werden im `MethodCall`-Objekt übermittelt. Der Rückgabewert wird in ein `ReturnValue`-Objekt gepackt und an den Client zurückgesendet.

Der Server akzeptiert eingehende TCP-Verbindungen und liest `MethodCall-Objekte` aus dem Stream, sobald welche vom Client gesendet werden. Der TCP-Stream wird erst geschlossen, wenn der Client ein EOF sendet. Für jede TCP-Verbindung, das heisst für jeden Client, wird ein `ClientHandler`-Objekt erstellt. Dieses enthält die Input- und OutputStream, um mit dem Client zu kommunizieren. Der `ClientHandler` liest die `MethodCall`-Objekte aus dem Stream und leitet sie an das `AccountSkeleton`-Objekt weiter. Die `MethodCall`-Objekte von verschiedenen Clients werden an das selbe `AccountSkeleton`-Objekt weitergeleitet (siehe Abbildung 2.2).

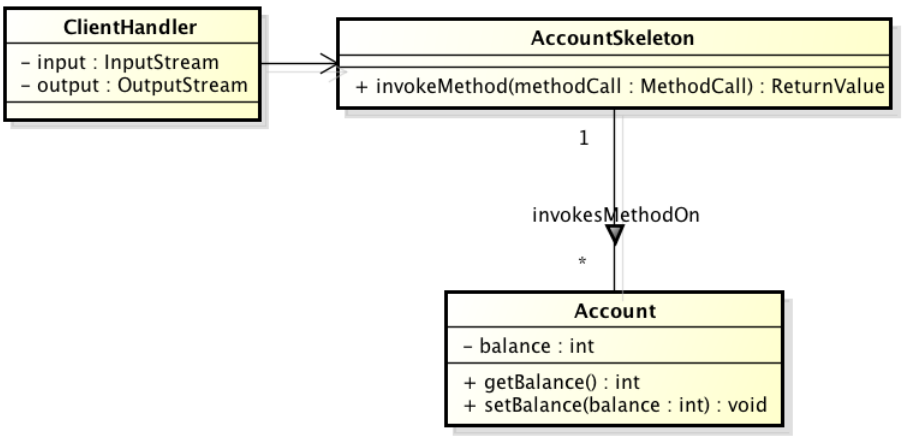

Abbildung 2.2: RMI-Serverimplementation Domain Class Diagramm

Schnittstelle zum Framework

Damit diese Clienthandler-Objekte mit den richtigen Skeleton-Objekten initialisiert werden, bietet das System eine zentrale ServerSystem-Klasse. Das Framework erzeugt eine Instanz der ServerSystem-Klasse. Wenn sich Client mit dem Server verbindet, fordert das Framework von dieser Serversysteminstanz einen Clienthandler an. Dieser wird mit den richtigen Skeleton--Objekten initialisiert. Das Framework übergibt dem Clienthandler die Input- und OutputStreamobjekte.

2.3 Concurrency Control Implementierung

Das System soll Lost Updates verhindern, indem Optimistic Concurrency eingesetzt wird. Das heisst, bei Schreibzugriffen wird überprüft, ob die Daten zwischenzeitlich verändert wurden. Dazu versioniert der Server die Objekte, deren Methoden auf entfernten Clients aufgerufen werden können. Die Versionen werden in der HashMap W gespeichert. Die HashMap W verwendet als Key die ID der Objekte und als Value die Version. Bei jedem Schreibzugriff, das heisst bei einem `setBalance()`, der ohne Konflikt ausgeführt wird, wird die Version des Objektes inkrementiert. Zusätzlich merkt sich der Server für jeden Client und für jedes Objekt, auf welcher Version des Objektes zuletzt ein Lesezugriff (`getBalance()`) stattgefunden hat. Dafür wird eine zweite HashMap R verwendet. In der HashMap R wird als Key ein String verwendet, der sich aus der IP-Adresse und der Objekt-ID des Methodenaufrufs zusammensetzt. Als Value speichert die HashMap R die Version ab. Ruft ein Client `getBalance()` auf, holt der Server die aktuelle Version aus

der HashMap W und speichert sie mit der entsprechenden IP und Objekt-ID in der HashMap R ab.

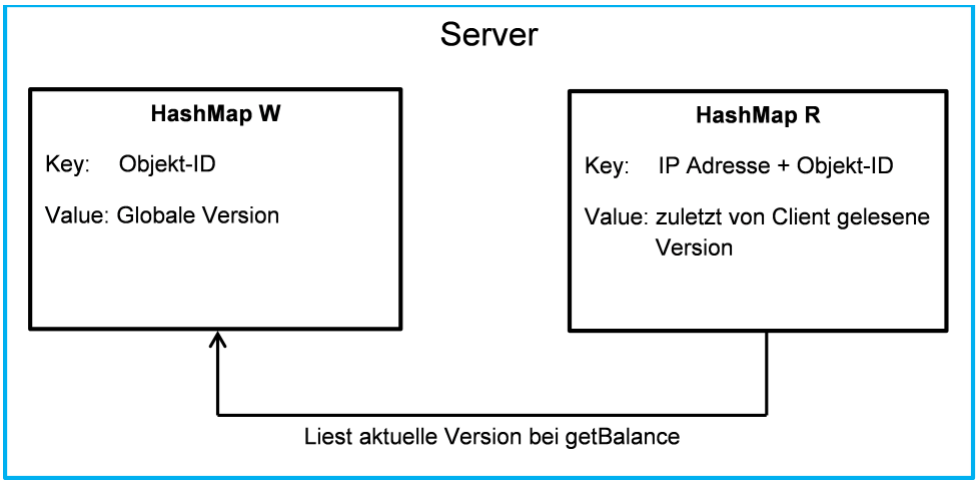

Abbildung 2.3: HashMaps für Concurrency Control

Ruft ein Client `setBalance()` auf, prüft der Server, ob die Version des Clients aus der HashMap R tiefer ist, als die aktuelle Version aus der HashMap W. Ist dies der Fall, wird der Methodenaufruf abgebrochen und der Client erhält eine Nachricht, dass die Methode nicht ausgeführt werden konnte.

Kapitel 3

Object-Caching

Um die Dauer von Lesezugriffen im RMI-System zu verkürzen, wurde eine erweiterte Version des RMI-Systems mit integriertem Objectcaching erstellt. Lesezugriffe werden auf Objekten, die in den Cache kopiert wurden, ausgeführt. Schreibzugriffe werden an den Server gesendet.

Bei der Erstellung des Konzeptes mussten Designentscheide in den Bereichen Replica Placement, Update Distribution und Konsistenzprotokoll getroffen werden. Im ersten Teil dieses Kapitels werden diese Entscheidungen beschrieben. Der zweite Teil beschreibt, wie diese Entscheide implementiert wurden.

Mit der Replikation von Objekten treten neue Probleme im Zusammenhang mit der Concurrency Control auf. Im Abschnitt 3.4.4 werden verschieden Ablaufszenarien untersucht, die zu Lost Updates führen.

3.1 Replica Placement

Ziel des Object Caching ist es, die Zugriffszeit auf Objekte für Clients zu verringern. Deshalb macht es Sinn, dass Repliken vom Client initiiert werden. Die Alternative wäre, dass der Server die Repliken initiiert. Das macht Sinn, wenn der Server von Anfragen, die von den Repliken beantwortet werden können, entlastet werden soll. Werden die Replikas vom Client initiiert, spricht man von Client Caches. Client Caches verbessern die Zugriffszeit der Clients auf die Daten.

In unserem RMIwithObjectCaching-System besitzt jeder Client einen lokalen Cache auf seiner Maschine. Der Cache ist in der Grösse nicht limitiert, da es keine Testcases mit Szenarien, die mehrere Account-Objekte beinhalten, gibt.

Durch das Caching sollen Lesezugriffe schneller ausgeführt werden, da der

Aufruf nicht über ein Socket geschickt werden muss, sondern im RAM bleibt. Dies ist natürlich nur der Fall, wenn das Objekt im Cache vorhanden ist.

3.2 Update Distribution

Werden Kopien von Objekten in einem lokalen Cache angelegt, müssen diese Kopien aktualisiert werden. Um dies zu realisieren, gibt es mehrere Möglichkeiten.

3.2.1 Invalidation versus Data Transfer

Invalidation-Protocol Bei einem Invalidation protocol werden nur Meldungen an die lokalen Caches gesendet, die dem Cache mitteilen, dass ein Objekt nicht mehr aktuell ist. Der Vorteil dieser Möglichkeit ist, dass eine Invalidierungsmeldung nur beim ersten Write versendet werden muss. Ausserdem müssen keine Objektdaten übertragen werden. Das Verfahren spart also Bandbreite. Ein Invalidation-Protocol macht Sinn, wenn das "read-to-write-Verhältnis"klein ist.

Transfer Data Der zweite Ansatz ist, bei jedem Update die kompletten Daten eines Objektes an die lokalen Caches zu versenden. In diesem Fall kann der Client immer aus dem lokalen Cache lesen. Dieser Ansatz eignet sich bei einem hohen "read-to-write Verhältnis".

Unser System verwendet den Transfer Data Ansatz, da in unseren Testszenarien sehr viele Lesezugriffe vorhanden sind. Das heisst, dass die Informationen in einem Update mit grosser Wahrscheinlichkeit im Cache gelesen werden. In diesem Fall müssen weniger Nachrichten versendet werden, wenn die aktualisierten Objekte bereits im lokalen Cache liegen. Bei einem Invalidation-Protocol müsste man jedes Mal eine aktualisierte Version anfordern.

3.2.2 Pull versus Push

Die Verantwortung der Cache-Aktualisierung kann entweder beim Server oder beim Client liegen. Man unterscheidet zwischen "push-based" und "pull-based" Protokollen.

push-based Der Server sendet dem Client Updates, ohne dass der Client diese anfordert. Der Server muss Buch darüber führen, in welchen Caches Kopien aller Objekte vorhanden sind.

pull-based Clients überprüfen bei jedem Zugriff, ob die Daten im Cache aktuell sind. Wenn nicht, müssen die Daten neu angefordert werden. Das macht den Zugriff auf Daten langsamer. Der Vorteil dabei ist, dass sich der Server nicht darum kümmern muss.

In unsererem Cache werden Aktualisierungen durch "push-based" Updateds realisiert, da wir die Zugriffszeit für die Clients erhöhen wollen und sich die Anzahl der Clients in Grenzen hält. Der Server kann sich merken, welcher Cache welche Objekte enthält.

3.3 Konsistenzprotokoll

3.3.1 Konsistenzmodell

In unserem System greifen Clients auf gemeinsame Objekte zu, indem sie Schreib- und Leseoperationen darauf ausführen. Jeder Client kann eine lokale Kopie eines Objektes bei sich im Cache haben. Führt ein Client eine Schreiboperation auf einem Objekt aus, muss diese durch das ganze verteilte System propagiert werden. Ein Konsistenzmodel beschreibt unter anderem, in welcher Reihenfolge die Clients die Schreiboperationen sehen. Das Konsistenzprotokoll implementiert das Konsistenzmodel.

Sequenzielle Konsistenz ist ein Konsistenzmodel. Es definiert, dass alle Prozesse in einem verteilten System die selbe Reihenfolge der Operationen auf einem Objekt sehen [4]. Das Konsistenzprotokoll unseres Object-Caching-Systems implementiert sequenzielle Konsistenz.

3.3.2 Primary-Based Protokol

In unserem System sind alle Objekte zentral auf einem Server abgespeichert. Das Objekt auf dem Server ist das Referenzobjekt. Alle Schreiboperationen werden von den Clients an den Server gesendet. Die Reihenfolge der Schreiboperationen auf einem Objekt ist die Reihenfolge, wie sie beim Server chronologisch eintreffen. Nachdem der Server eine Schreiboperation verarbeitet hat, sendet er ein Update an alle Clients, die das aktualisierte Objekt bei sich im Cache haben. Damit sehen alle Clients die selbe Updatereihenfolge und die Anforderung für die sequenzielle Konsistenz ist damit erfüllt.

Ein Problem im Zusammenhang mit dem primary-based Protokoll ist, dass das System mehr Netzwerkbandbreite benötigt, da es mehr Nachrichten versendet, um die Kopien aktuell zu halten.

Ist jedem Objekt eine Maschine für die Koordination zugeordnet, nennt man das Protokoll "primary-based"[4]. Die Maschine, die das Objekt verwaltet, nennt man primary. Ist der primary eine fixer Server, werden alle Schreiboperationen an diesen Server gesendet. In diesem Fall spricht man von Remote-Write-Protocol. Unser System realisiert ein Remote-Write-Protocol.

3.4 Implementierung des Object-Caching

3.4.1 Implementierung des Replica Placement

Ruft ein Client eine Methode auf einem Objekt auf, überprüft das System zuerst, ob das Objekt bereits im lokalen Cache vorhanden ist. Der Cache hat eine HashMap, die als Keys Objekt-IDs und als Values Objekte enthält. Ist ein Objekt nicht in dieser HashMap vorhanden, wird eine `ObjectRequest`-Message an den Server gesendet. Der Server antwortet, indem er das gewünschte Objekt serialisiert an den Client zurücksendet. Der Client legt es in der HashMap C ab.

3.4.2 Implementierung Update Distribution

Da wir die Update Distribution push-based implementieren, informiert der Server alle Clients bei jedem Schreibzugriff. Damit der Server die Updates an die richtigen Clients sendet, merkt er sich bei einem `ObjectRequest`, an welche Clients er das Objekt gesendet hat. Er führt für jede Objekt-ID eine `ArrayList`, die alle Clients enthält, die das Objekt bereits angefordert haben. Wird auf einem Objekt eine Methode mit Seiteneffekt ausgeführt, sendet der Server allen Clients in der Liste eine `ObjectUpdate`-Message.

Implementierung des Nachrichtenaustauschs

Aufgrund der `ObjectUpdate`-Nachrichten muss der Client in der Lage sein, eine Nachricht zu empfangen, ohne dass er explizit darauf wartet. Im RMI-System ohne Cache schreibt ein Client die Nachricht für einen Methodenaufruf in den Stream zum Server und wartet anschliessend auf die Antwort. Er weiss, dass das nächste Objekt im Stream vom Typ `ReturnValue` und die Antwort auf den vorangegangenen Methodenaufruf sein wird. Das ist möglich, weil das nächste Objekt, das der Client aus dem Stream vom Server aus liest, immer die Antwort auf den Methodenaufruf ist.

Im Object-Caching System muss der Client in der Lage sein, `ObjectUpdate`-Messages zu empfangen und zu verarbeiten, ohne dass das System

weiss, wann ein `ObjectUpdate` über den Stream gesendet wird. Der Client kann sich nicht mehr darauf verlassen, dass Objekte aus dem TCP-Stream zum Server die Antwort auf einen Methodaufruf sind. Es könnte auch ein `ObjectUpdate` sein. Damit die Verarbeitung von eintreffenden Nachrichten im Client nicht von der ausgehenden Nachricht abhängig ist, wird der Nachrichtenaustausch im System in der Klasse `MessageManager` gekapselt. `MessageManager` bietet folgende Methoden an:

sendMessageCall Diese Methode nimmt alle Nachrichten entgegen, die an den Server gesendet werden sollen. Die Methode platziert alle Nachrichten in einer `BlockingQueue`

receiveReturnValue Hat ein Client eine Nachricht für einen Methodenaufruf an den Server gesendet, liefert diese Methode die darauffolgende ReturnValue-Nachricht. Da der `MessageManager` nur von einem Client benutzt wird, kann sich ein Client darauf verlassen, dass die ReturnValue-Nachricht zur vorhergehenden MethodCall-Nachricht passt. Die Methode blockiert bis ein ReturnValue empfangen wird.

receiveObject Antwortnachrichten für einen ObjectRequest können über diese Methode entgegengenommen werden.

receiveUpdate Die Methode blockiert, bis eine ObjectUpdate-Nachricht vom Server eintrifft. Wird eine ObjectUpdate-Nachricht empfangen, liefert die Methode ein ObjectUpdate-Objekt als Rückgabewert.

Die Verarbeitung der Nachrichten wird von mehreren Threads durchgeführt (siehe Abbildung 3.1. Für ausgehende Nachrichten enthält der `MessageManager` eine BlockingQueue. Bei Aufruf von sendMessageCall wird die zu sendende Nachricht in der Queue platziert. Ein Thread ist nur dafür zuständig, Nachrichten aus dieser Queue zu nehmen und in den Stream zum Server zu schreiben. Für eintreffende Nachrichten hat der Server mehrere BlockingQueues. Für jede der Nachrichtentypen ObjectRequestResponse, ReturnValue und ObjectUpdate gibt es eine BlockingQueue. Trifft eine Nachricht beim Client ein, muss sie einfach in die richtige Queue geschoben werden. Ein Client holt sie mit der entsprechenden receive-Methode wieder raus. Ein Thread ist dafür zuständig, eintreffende Nachrichten aus dem Stream zu lesen und der richtigen Queue hinzuzufügen.

Der Cache enthält ein Thread, der in einer Endlosschleife receiveUpdate aufruft. Diese Methode blockiert die meiste Zeit, da nicht andauernd Updates empfangen werden. Kommt eine ObjectUpdate-Nachricht rein, wird sie im MessageManager in der BlockingQueue für ObjectUpdates hinzugefügt. Der receiveUpdate-Methodenaufruf kommt zurück und aktualisiert den Cache.

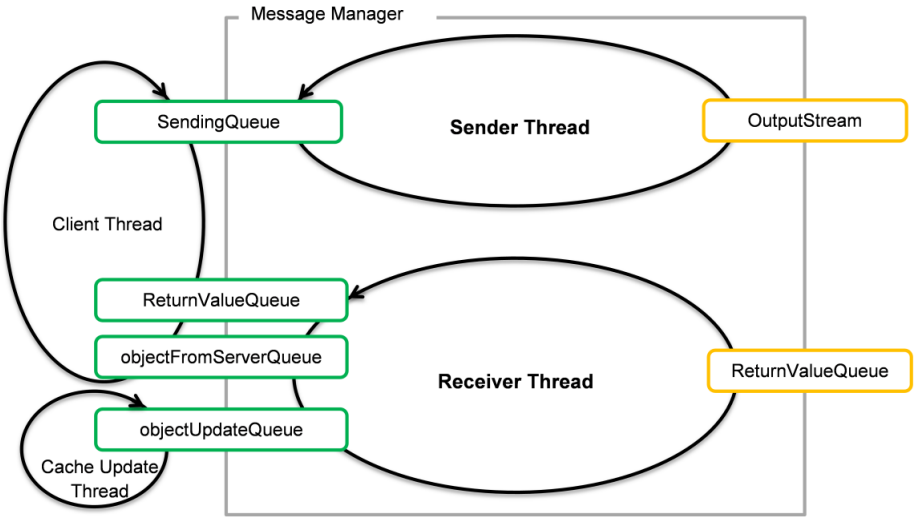

Abbildung 3.1: MessageManager Threads

3.4.3 Beispielablaufszenario

Die Funktionsweise des Protokolls und des Systems soll anhand eines Beispiels erklärt werden. Ein Client holt sich den aktuellen Wert eines Kontos mit `getBalance()` und setzt danach den Kontostand mit `setBalance()`. Das betreffende `Account`-Objekt ist zuerst noch nicht im Cache des Clients. Der Prozess läuft wie folgt ab.

1. Client $C1$ ruft `getBalance()` auf.
2. RMI-Stub führt einen Lookup für das Objekt im Cache aus.
3. Ist das Objekt nicht im Cache vorhanden, wird es mit einem `Object-Request`-Nachricht angefordert.
4. Ist das `Account`-Objekt beim Client eingetroffen, wird `getBalance()` auf dem Objekt ausgeführt. Der Wert wird an den Stub zurückgegeben und der Stub leitet ihn an den Client weiter.
5. `setBalance()` wird an den lokalen Cache weitergeleitet. Dieser enthält eine eigene Concurrency Control. Diese überprüft, ob das Objekt seit dem letzen `getBalance()` nicht durch ein Update vom Server verändert wurde. Hat es sich verändert, wird eine Exception ausgelöst. Wenn in der Zwischenzeit kein Update eingetroffen ist, wird ein `MethodCall` an den Server gesendet. Nachdem der Methodenaufruf an den Server gesendet wurde, wartet der Client auf die `ReturnValue`-Nachricht. Diese

trifft erst ein, wenn der Server den Methodenaufruf bereits verarbeitet hat und ein Update verschickt hat. Die Methode ändert den lokalen Cache nicht, weil zu diesem Zeitpunkt noch nicht klar ist, ob die Methode ausgeführt werden kann. Ein anderer Client könnte das Objekt inzwischen verändert haben. Nur der Server kann entscheiden, ob die Methode ausgeführt werden kann. Das Objekt im Cache ändert erst, wenn die `ObjectUpdate`-Nachricht vom Server eintrifft.

3.4.4 Implementierung der Concurrency Control

Im Unterschied zum RMI-System ohne Cache werden die `getBalance()` Methodenaufrufe nicht vom Server verarbeitet. Wie in Kapitel 2.3 beschrieben, werden Lost Updates im System ohne Cache verhindert, indem der Server Buch darüber führt, welche Version eines Account-Wertes ein Client mit `getBalance()` zuletzt geholt hat. Eine erster Lösungsansatz ist die HashMap mit den Clientversionen zu aktualisieren, wenn der Server dem Client ein neues Objekt in Form eines `ObjectRequestResponse` oder eines `ObjectUpdate` sendet. Wenn die Updatenachricht im Client ankommt und korrekt verarbeitet wird, kann der Server davon ausgehen, dass der Client mit der aktualisierten Version arbeitet.

Zusätzlich wird die Concurrency Control im lokalen Cache implementiert. Ruft ein Client `setBalance()` auf, wird bereits beim Client überprüft, ob seit dem letzten `getBalance()` eine `ObjectUpdate`-Nachricht für das betreffende Objekt eingetroffen ist.

Um mögliche Probleme zu finden, sollen verschieden Aufrufszenarien untersucht werden. Die Szenarien unterscheiden sich in der Aufrufreihenfolge der Operationen.

Szenario mit einzelnem Client

Im ersten Szenario treten keine Konsistenzprobleme auf. Nur ein Client ruft nacheinander `getBalance()` und `setBalance()` auf (siehe Abbildung 3.2). Laufen die Methodenaufrufe in dieser Reihenfolge ab, benötigt das System keine Concurrency Control.

1. Beim ersten `getBalance()` fordert der Cache ein neues Objekt an und speichert es.

2. Beim zweiten `getBalance()` liegt das Objekt bereits im Cache und muss nicht nochmals vom Server angefordert werden.

3. Ruft der Client ein `setBalance()` auf, wird dieser Aufruf direkt an den Server weitergeleitet. Der Aufruf blockiert, bis der Server eine Antwort sendet. Der Server sendet zuerst allen Clients das Update. Danach sendet er den Rückgabewert des Methodenaufrufs an den Client, der das `MethodCall`-Objekt verschickt hat. Das Update wird also vor dem ReturnValue versendet. Im ClientCache wird das Update und das ReturnValue verarbeitet. Es kann nicht gesagt werden, ob der Cache bereits ein Update erfahren hat, wenn `setBalance()` zurückkehrt.

4. Wenn der Client nochmals `getBalance()` aufruft, wird die Methode auf dem aktualisierten Objekt im Cache ausgeführt.

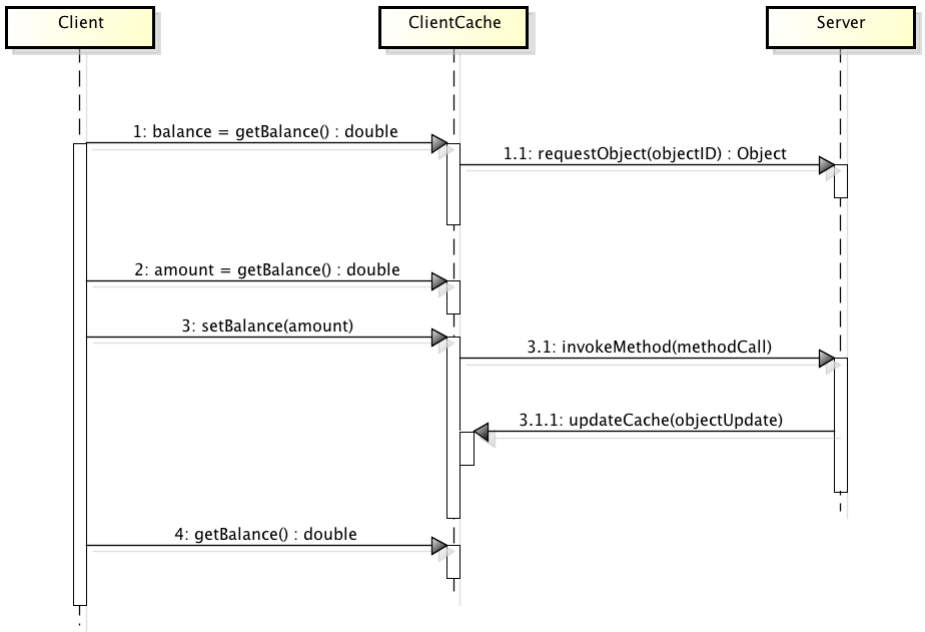

Abbildung 3.2: Szenario mit einem Client

Konflikt wird lokal festgestellt

In diesem Szenario versucht Client 1 ein `setBalance()` und das Objekt im lokalen Cache von Client 1 wurde seit dem letzen `getBalance()` im Cache durch ein Update aktualisiert. Der Konflikt kann schon in der Virtual Machine von Client 1 abgefangen werden (siehe 3.3).

1. Client 1 fordert mit `getBalance` ein Objekt an.

2. Client 2 fordert das selbe Objekt auch an.

3. Client 2 ruft `setBalance` auf und verändert das Objekt auf dem Server. Der Server sendet Client 1 ein Update.

4. Client 1 ruft ebenfalls `setBalance` auf. Da sich das Objekt in der Zwischenzeit geändert hat und Client 1 kein `getBalance()`ausgeführt hat, arbeitet der Client mit veralteten Daten. Der Clientcache vergleicht die aktuelle Versionnummer mit der Versionnummer, die das Objekt beim letzen `getBalance()`-Aufruf hatte. Das Objekt hat inzwischen ein Update erfahren und hat somit eine höhere Versionsnummer. Der Cache wirft eine Exception.

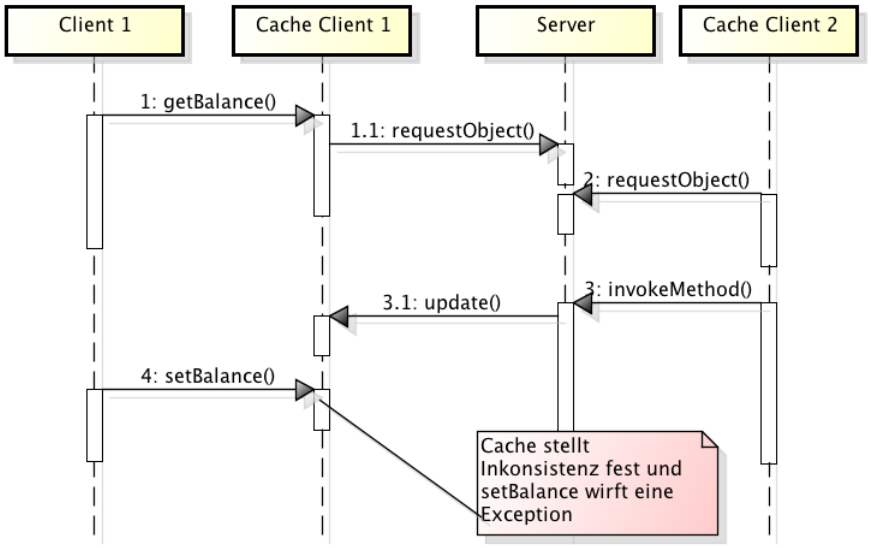

Abbildung 3.3: Konflikt wird in lokal behoben

Konflikt wird auf Server festgestellt

Eine weitere Möglichkeit, in welcher Reihenfolge Methodenaufrufe verarbeitet werden, beinhaltet, dass der Server ein `setBalance()`von einem Client erhält, bevor er das Update von einem vorangegangenen `setBalance()` an alle Clients versendet hat. In diesem Fall detektiert der Server die Inkonsistenz. Beide Clients haben sich die selbe Version des selben Objektes geholt. Diese Methodenaufrufe werden im Sequenzdiagramm 3.4 nicht mehr aufgeführt.

1. Client 2 sendet ein `setBalance()` an den Server. Bei der Verabeitung dieses Methodenaufrufs macht der Server drei Dinge in dieser Reihenfolge:

(a) Methode auf Object ausführen

(b) Update an alle Clients senden

(c) Versionnummern der ConcurrencyControl für jeden Client updaten

In diesem Szenario wird der `setBalance()`-Aufruf unterbrochen, bevor er die Versionsnummern updated.

2. Client 1 sendet ebenfalls ein `getBalance()`. Der Cache in Client 1 hat für den Methodenaufruf von Client 2 noch kein Update erhalten. Er wird an den Server weitergeleitet. Der Methodenaufruf trifft beim Server vor dem Versionsnummernupdate des Methodenaufrufs von Client 2 ein. Der Server stellt fest, dass Client 1 noch mit einer alten Version arbeitet und returniert Client 1 eine Exception.

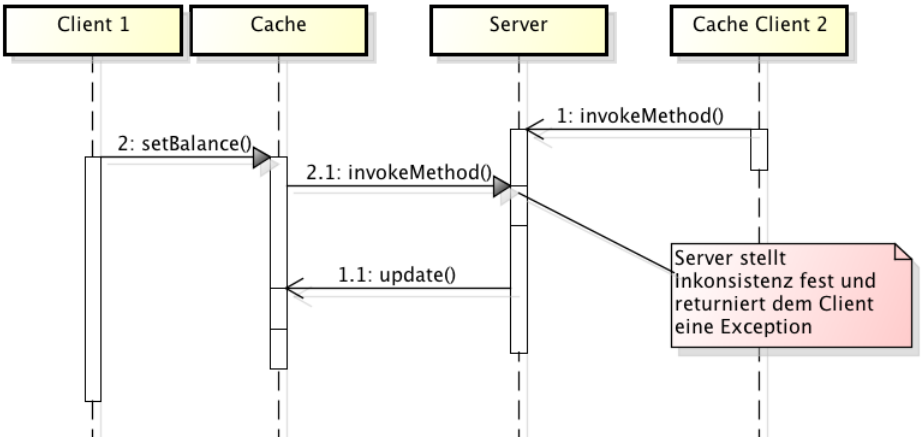

Abbildung 3.4: Konflikt wird auf Server festgestellt

Update und setBalance überkreuzen sich

Bei allen Fällen kann Inkonsistenz festgestellt werden, indem sich der Server und der Clientcache merkt, welche Version an den Client vor dem `setBalance()` herausgegeben wurde und überprüft, ob die Version des Objektes mittlerweile höher ist. Dabei müssen die Versionnummern im Cache und auf dem Server nicht synchron sein. Es gibt jedoch eine Aufrufabfolgemöglichkeit, bei der Inkonsistenz entstehen kann (siehe 3.5).

1. Client 2 sendet ein `setBalance()`. Der Server versendet das Update und aktualisiert die Versionsnummern der Clients.

2. Client 1 sendet ebenfalls ein `setBalance()`. Er sendet den Methodenaufruf dem Server, bevor er den Update erhalten hat. Zum Zeitpunkt der Ankunft des Methodenaufrufs beim Server hat der Server aber bereits das Update versendet und die Versionsnummer aktualisiert. Der Methodenaufruf und der Update haben sich also gekreuzt. Da der Server meint, dass Client 1 mit der aktuellen Version arbeitet, wird er den `setBalance()` Aufruf auf dem Objekt ausführen. Es kommt zu einem Lost Update.

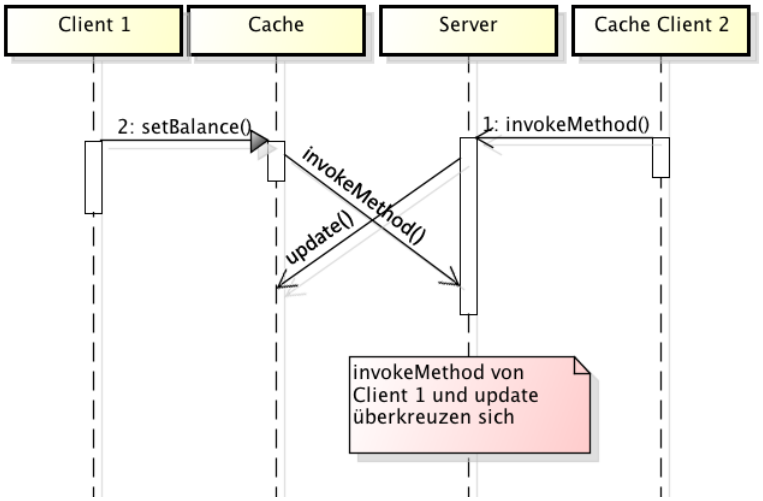

Abbildung 3.5: Update und setBalance überkreuzen sich

Um Inkonsistenz bei diesem Szenario zu erkennen und zu verhindern, muss die Nachricht, die den `setBalance()` Methodenaufruf enthält, die Versionnummer des Clients, der die Methode aufruft, mitsenden. Ist diese Versionnummer tiefer, als die aktuelle Versionnummer des Objektes, wird eine Exception geworfen. Damit die Versionnummern auf Server und auf Client immer synchron sind, muss bei einem ObjectRequest die aktuelle Versionnummer an den Client gesendet werden. Bei einem Update wird die Nummer einfach inkrementiert.

Alternativ zu der Lösung mit der Versionsnummer, die bei einem Methodenaufruf mitgesendet wird, könnte der Server auch eine Bestätigung des Client, dass er das Update erhalten hat, abwarten und erst nach dieser Bestätigung die Version des entsprechenden Clients aktualisieren.

Kapitel 4

Testing Framework

Im folgenden Kapitel wird auf die einzelnen Komponenten des Test Frameworks eingegangen und deren Funktionsweise erläutert. Die Aufgabe des Test Frameworks liegt darin, die verschiedenen Prototypen, welche im Laufe dieser Studienarbeit entwickelt wurden, mit diesem Framework einheitlich testen zu lassen. Das Test Framework wurde parallel zum RMI-System entwickelt.

4.1 Konzept

Das Framework lässt sich über eine Konfigurationsdatei konfigurieren und kann Testfälle, welche in einer XML Datenstruktur vorliegen, interpretieren und daraus die einzelnen Testszenarien generieren. Die Testszenarien werden vom Server auf die verfügbaren Testframework Clients verteilt und dort gestartet. Das Framework startet das zu testende System auf den verschiedenen Rechnern. Das zu testende System führt unmittelbar danach die Aktionen durch, welche im übergebenen Szenario definiert wurden.

Die mit den Messwerten ausgefüllten Szenarien werden an den Server zurückgeschickt und dort ausgewertet. Dies ermöglicht einen Vergleich der verschiedenen eingesetzten Algorithmen. Folgende Eckpunkte muss das Framework erfüllen:

- Die enwickelten Systeme müssen sich ohne Programmcode-Anpassungen an das Framework anbinden lassen.
- Das Framework muss die genauen Zeiten, welche für die Ausführungen der Operationen nötig waren, messen können.
- Nach einem Testlauf muss ein Bericht erstellt werden können, welcher die nötigen Informationen darstellt.

4.2 Allgemeine Informationen

Um möglichst genaue Messdaten zu erlangen, wird das ganze System nach einem Testlauf beendet. Eine Aneinanderreihung von verschiedenen Testläufen würde die Messdaten aufgrund der Java Virtual Machine beeinträchtigen. Es besteht die Möglichkeit, dass für einen zweiten Testlauf weniger Speicher zur Verfügung steht, weil der Speicher immer noch mit Objekten des ersten Testlaufs besetzt ist. Da dies nicht mit Sicherheit überprüft werden kann, dass die Grundvoraussetzungen für beide Testläufe gleich sind, wurde auf eine Aneinanderreihung von Testläufen verzichtet.

4.3 Der Frameworkserver

4.3.1 Das Startup-Script

Das Startup-Script ist ein in der Bash-Sprache geschriebenes Shell-Script. Das Script führt mehrere Methoden nacheinander aus:

1. Ein ant- Script wird angestossen, welches die Projekte kompiliert. Durch diesen Schritt entstehen die .jar- Dateien, welche im nächsten Schritt benötigt werden.

2. Das Script prüft die in der Config-Datei eingetragenen Zielrechner auf bereits vorhandene “client.jar“-Dateien und löscht diese Dateien, falls vorhanden.

3. Durch die Applikation “Secure Copy”wird die zuvor im Buildprozess erstellte Datei “client.jar“ auf die Zielrechner kopiert.

4. Via SSH wird der Frameworkclient auf den Zielrechnern gestartet.

5. Der Frameworkserver wird gestartet.

Sind diese Schritte abgeschlossen, beendet das Startupscript und die weitere Ausführung des Testlaufs wird durch den Frameworkserver orchestriert. Beim Starten des Scripts wird der Name der Datei mitgegeben, in welcher der genaue Testlauf definiert ist. Wird kein Argument mitgegeben, läuft der Standardtestlauf ab, welcher in der Datei “testCases.xml”beschrieben ist.
Probleme beim Schreiben des Scripts waren selten. Ein Problem, welches gelöst werden musste, war der Umstand, dass das Script nach dem Starten eines Clients nicht mehr weiterlief, sondern einfach wartete. Dies war bei folgender Zeile der Fall:

```
ssh student@${i} "java -jar ${remotePath}/${clientJar}"
```

Offensichtlich wartet das Script beim Ausführen eines Befehls auf einem fremden Rechner auf einen Rückgabewert in irgendeiner Form; also auf einen Errorcode oder aber auf einen Exitstatus. Das Absetzen dieses Befehls musste in einem seperaten Kind-Prozess stattfinden, damit der Eltern-Prozess die weiteren Aufgaben des Scripts abarbeiten konnte und nicht auf einen Rückgabewert wartete. Dies kann in der Bash-Shell mit einem "&" erreicht werden. Die simple Lösung des Problems sieht schlussendlich fast gleich aus:

```
ssh student@${i} "java -jar ${remotePath}/${clientJar}" &
```

4.3.2 Startprozess des Frameworkservers

In diesem Kapitel werden die Schritte beschrieben, welche vor dem Starten des Testlaufs durchgeführt werden. Der Frameworkserver, einmal gestartet, administriert den ganzen weiteren Ablauf des Testlaufs.

Die Kommunikation des Frameworkservers mit den Frameworkclients ist in folgendem Sequenzdiagramm grob dargestellt:

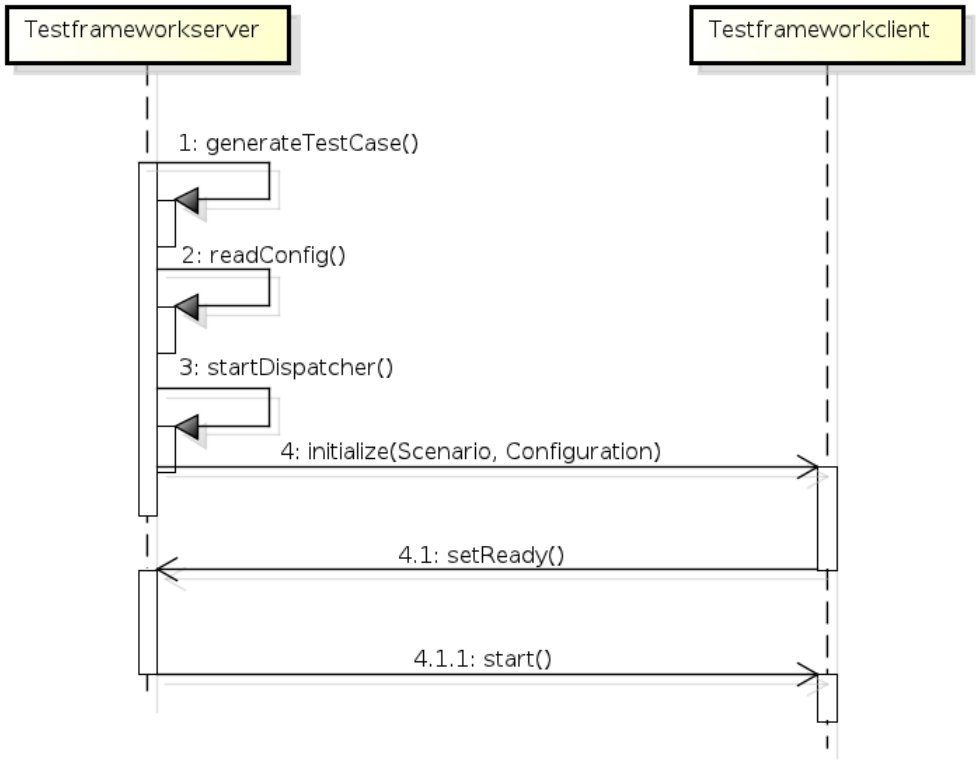

Abbildung 4.1: Initialisierung des Testframework

Die im Diagramm dargestellten Operationen lassen sich wie folgt erklären:

- Die Methode `generateTestCase()` ruft eine Factory auf, welche aus einer XML-Datei einen Testcase generiert. Diese Factory wird unter dem Kapitel 4.3.2 “Testcase-Factory“ genauer beschrieben.

- Alle Konfigurationsparameter, welche der Server braucht, sind in einer Konfigurationsdatei abgelegt. Durch eine Configurationfactory wird mit Hilfe der Konfigurationsdatei ein Configuration- Objekt erzeugt. Unter dem Kapitel 4.3.2 “Configuration Factory“ wird dies näher beschrieben.

- Nachdem die Konfigurationsparameter ausgelesen wurden, kann der Dispatcher in einem seperaten Thread gestartet werden. Weitere Ausführungen sind unter dem Kapitel 4.3.2 “Der Dispatcher“ zu finden.

- Bei der Initialisierung der Clients, wird diesen je ein Szenario und ein Configuration-Objekt gesendet. Das Configuration-Objekt umfasst alle für den Client wichtigen Konfigurationsparameter, wie etwa die Ports, unter welchen der Server seine Dienste zur Verfügung stellt. Was ein Szenario genau ist, wird unter dem Kapitel 4.3.2 “Testcase-Factory”genauer beschrieben.

- Pro Client hat der Server beim Auslesen der Konfigurationsdatei ein Client-Objekt instanziert. Dies wurde vorallem zur Kontrolle der Clients durch den Server so implementiert. Darauf wird im Kapitel 4.3.2 “ClientObjekt“ eingegangen.

- Nachdem alle Clients `setReady()` aufgerufen haben, ruft der Server `start()` auf allen Clients auf. Dieser Vorgang wird unter dem Kapitel 4.3.2 “Start-Methode“ genauer erläutert.

Testcase-Factory

Die Testcase-Factory liest ein XML-File, in welchem ein sogenannter Test-Case“ definiert ist. Wird beim Starten des Frameworkservers kein Parameter mitgegeben, wird der Standard-Testcase gewählt, welcher in der Datei “test-Cases.xml“ abgelegt ist. Wird aber ein Parameter mitgegeben, beschreibt dieser den Namen der auszuführenden TestCase-Datei.

Durch die eingelesenen Informationen erstellt die Factory die Szenarien, welche dann weiter an die verschiedenen Clients gesendet werden. Folgender Block ist ein Auszug aus einer TestCase-Datei:

```
<?xml version='1.0' encoding='UTF-8'?>
<TestRun>
  <TestCase SystemUnderTest="ch.hsr.objectCaching.
     rmiOnlyClient.RMIonlyClientSystem">
    <Account balance="1"></Account>
      <Scenario id="1">
        <ActionSequence>
          <Increment count ="100" delay="0" factor="1.1
             "></Increment>
        </ActionSequence>
      </Scenario>
      <Scenario id="2">
        <ActionSequence>
          <Increment count ="100" delay="0" factor="1.1
             "></Increment>
        </ActionSequence>
      </Scenario>
  </TestCase>
</TestRun>
```

Das Attribut "SystemUnderTest"definiert das zu testende System. Der Vorteil dieser Methode ist, dass das zu testende System einfach in XML angegeben wird und dann getestet werden kann. Es müssen so keine Anpassungen am Code vorgenommen werden, egal welches System getestet werden soll. Somit spielt es für das Framework keine Rolle, welches System getestet werden soll.

Eine TestCase-Datei beschreibt genau einen TestCase. Ein TestCase kann verschiedene Szenarien beinhalten oder aber auch nur ein Szenario beschreiben. Ist nur ein Szenario definiert, werden alle beteiligten Clients mit demselben Szenario arbeiten. Falls mehrere Szenarien beschrieben sind, werden die Clients unterschiedliche Szenarien durchführen.

Aus obigem XML-Code generiert die Factory nun ein Objekt vom Typ TestCase. Dieses Objekt definiert nun zwei Szenarien, welche wiederum aus mehreren Actions bestehen. Was genau eine Action ist, ist unter dem Kapitel 4.5 "Actions"beschrieben. Die Factory wird nun bei diesem Beispielcode eine Abfolge von 100 "Increment-Action" erstellen und diese in das Scenario-Objekt ablegen.

Configuration-Factory

Die Configuration-Factory liest alle Daten aus der Konfigurationsdatei aus und erstellt mit diesen Informationen ein Configuration-Objekt. Die Konfi-

gurationsdatei sieht wie folgt aus:

```
Client0=152.96.193.18
Client1=152.96.193.19
Clientport=36927
ServerRmiPort=36925
ServerSocketPort=36926
ServerRegistryName=Server
ClientRegistryName=Client
```

In der Konfigurationsdatei ist definiert, welche Clients auf welchem Port kontaktiert werden können. Weiter ist der Registry-Name definiert, damit der Server weiss, über welche Registry er die Methoden auf dem Client aufrufen muss.
Aus diesen Informationen erstellt die Factory ein Configuration- Objekt, welches sie dem Client als erstes sendet. Damit ist gewährleistet, dass der Frameworkserver und die Frameworkclients miteinander über RMI kommunizieren können.
Pro Client, welcher in der Konfigurationsdatei definiert ist, wird ein Client-Objekt instanziert, was im folgenden Kapitel beschrieben wird.

ClientObjekt

Das Clientobjekt besitzt unter anderem ein Datenfeld, welches genau zwei Zustände annehmen kann: Ready und NotReady. Führt der Frameworkclient die Methode `setReady()` auf dem Server aus, wird der Status auf dem jeweiligen Objekt auf Ready gesetzt. Folgend die Implementation der setReady-Methode:

```
public void setReady(String ip)
{
  logger.info("Setted ready with: " + ip);
  Client temp;
  if((temp = clientList.getClientByIp(ip)) != null)
  {
    temp.setStartingState(StartingState.READY);
  }
  if(checkAllReady())
  {
    start();
  }
}
```

Interessant an dieser Methode ist die Funktion der `checkAllReady()`-Methode. Dieser Mechanismus ist nicht nur dazu da, um alle Clients möglichst gleichzeitig zu starten, sondern auch, um die Threads zu steuern. Die Methode `setReady()` wird durch mehrere verschiedene Threads aufgerufen, von jedem Client genau ein Mal. Durch die Implementation von `checkAllReady()` wird aber nur der letzte Thread, welcher `setReady()` aufruft, weiterleben und `start()` ausführen können. Somit ist garantiert, dass auch danach nur ein Thread aktiv ist und keine Seiteneffekte entstehen können.

Die Implementation dieser Logik war nicht schwer, doch das Bewusstsein für die Problematik mit mehreren Threads muss ständig vorhanden sein. Die Einflüsse der parallelen Programmierung stellten ohnehin einen spannenden Aspekt dieser Arbeit dar.

Die Start-Methode

Die Methode `start()` gibt dem Client an, mit der Abarbeitung des Szenarios zu beginnen. Die erste Implementation dieser Methode sah wie folgt aus:

```
private void start()
{
  logger.info("Method start() invoked");
  for(int i = 0; i < clientList.size(); i++)
  {
    clientList.getClient(i).getClientStub().startTest();
  }
}
```

Nach einigen Testdurchläufen konnte festgestellt werden, dass die Clients ihre Szenarien nur sequentiell abarbeiteten und die Operationen auf dem Server nicht gleichzeitig geschehen. Schuld an diesem Umstand war obige Implementation der `start`-Methode. Der Server-Thread, welcher `startTest()` auf dem Client aufrief, wartete solange, bis die Methode zurückkehrte. Da die Methode natürlich erst nach der Beendigung des Szenarios zurückkehrt, konnte nur immer genau ein Client zu einem gewissen Zeitpunkt aktiv sein. Die jetzige Implementation und somit die Lösung des Problems sieht wie folgt aus:

```
private void start()
{
  logger.info("Method start() invoked");
  for(int i = 0; i < clientList.size(); i++)
  {
```

```
    ClientStart clientStart = new ClientStart(clientList.
    getClient(i));
    new Thread(clientStart).start();
  }
}
```

Um das Problem mit der Erstellung eines neuen Threads für jeden zu startenden Client zu umgehen, musste eine weitere Klasse Namens "ClientStart" geschrieben werden. Die Klasse implementiert natürlich das Interface "Runnable" und implementiert die "run()"-Methode wie folgt:

```
public void run()
{
  try{
      client.getClientStub().startTest();
  } catch (RemoteException e){
      logger.log(Level.SEVERE, "Uncaught exception", e);
  }
}
```

Mit dieser Implementation wird der Methodenaufruf `start()` nun von einem seperaten Thread ausgeführt. Dies führt dazu, dass die Clients nun zeitlich parallel gestartet werden und nicht nacheinander. Da die einzelnen Threads nach Beendigung der `run()`-Methode sterben, ist die Art der Implementation unbedenklich bezüglich Seiteneffekte oder "Zombie-Threads".

Der Dispatcher

Der Dispatcher stellt die Verbindung zwischen dem Frameworkserver und dem Server des Systems, welches getestet werden soll, dar. Dem Dispatcher wird die Portnummer angegeben, unter welcher er ein Socket öffnen soll. Weiter wird ihm ein String übergeben, in welchem der Pfad des zu testenden Systems drinsteht. Der Dispatcher instanziert daraufhin das zu testende System mit Hilfe des Strings und wartet danach in einem blockierten Zustand auf sich verbindende Clients. Nachdem ein Client auf dem geöffneten Socket eine Verbindung aufgebaut hat, übergibt der Dispatcher die Verbindung in Form eines Input und eines Outputstreams dem Server des zu testenden Systems.

Logischerweise läuft der Dispatcher wiederum in einem seperaten Thread.

4.3.3 Herunterfahren des Frameworkservers

Nach Beendigung eines Testlaufs, muss der Server mehrere Arbeiten erledigen. Nachfolgend werden die wichtigsten Aufgaben aufgeführt:

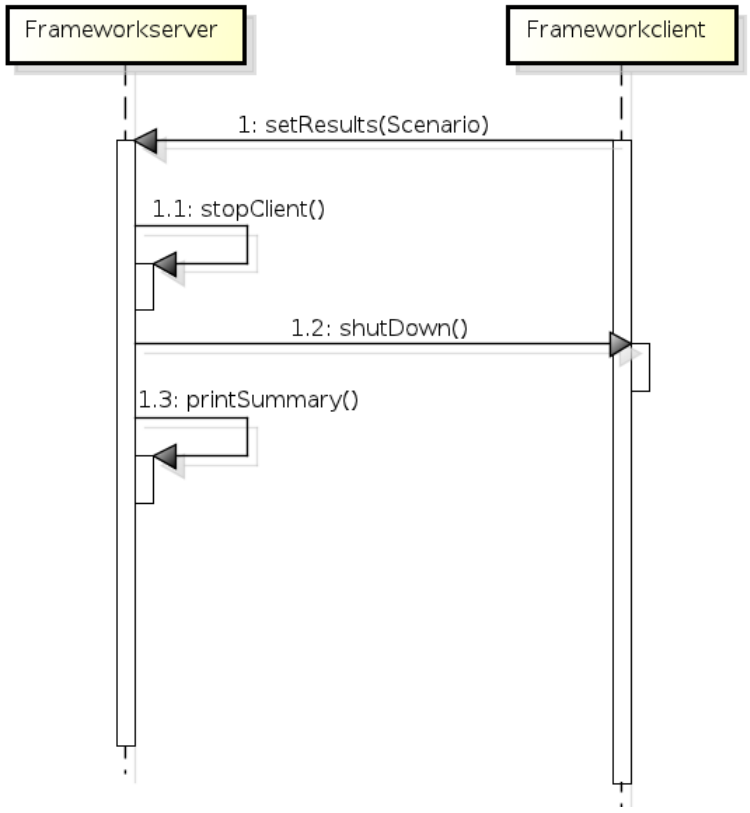

Abbildung 4.2: Testfamework Exit

Die im Diagramm ersichtlichen Operationen können folgendermassen erläutert werden:

- Die Methode `setResults(Scenario)` wird vom Client auf dem Server aufgerufen. Durch den Aufruf dieser Methode weiss der Server, dass der aufrufende Client sein Szenario durchgespielt hat. Weiter wird mit dem Aufruf der Methode wiederum ein Thread durch den RMI-Daemon gestartet, welcher die nachfolgenden Methoden anstösst.

- Aus der `setResults(Scenario)`- Methode wird direkt `stopClient()` aufgerufen. Was diese Methode genau macht, wird im Kapitel 4.3.3 “Stoppen eines Clients”genau beschrieben.

- Die Methode `shutDown()` wird auf dem Client aufgerufen. Der Client trennt daraufhin die RMI-Verbindung zum Server und schaltet sich selber ab. Nähere Informationen zum Abschaltungsprozess eines Clients kann unter dem Kapitel 4.4 “Testframework Client“ nachgelesen werden.
- Die Methode `printSummary()` verwertet die Ergebnisse, welche die Clients in ihrem Szenario gespeichert und wieder übertragen haben. Mehr Informationen dazu im Kapitel 4.3.3 “Report Generator“. Weiter wird auf der Konsole ausgegeben, ob es “Lost-Update“-Probleme gegeben hat oder nicht. Mehr zu diesem Thema unter dem Kapitel 4.3.3“Result Generator”.

Stoppen eines Clients

Ruft ein Client die Methode `setResults(Scenario)` auf dem Server auf, wird Serverintern der Status des Clients auf ”Down”gesetzt. Da die Methode `stopClient()` wiederum pro Client genau ein Mal aufgerufen wird, entsteht das Problem, dass mehrere Threads am Leben sind. Um dies ab einem gewissen Punkt zu verhindern, wurde ähnlich zur `setReady()`-Methode, folgende Implementation gewählt:

```
private void stopClient(String clientIp)
{
  Client temp;
  try {
if((temp = clientList.getClientByIp(clientIp)) != null)
        {
   logger.info("stop client with ip: " + clientIp);
   temp.setClientRunning(ShutedDown.DOWN);
   temp.getClientStub().shutdown();
}
} catch (RemoteException e) {
    logger.log(Level.SEVERE, "Uncaught exception", e);
}
if(checkAllShutedDown())
{
  printSummary();
}
}
```

Durch die Prüfung `checkAllShutedDown()`, wird nur der letzte aller Threads weiterleben. Somit wird auch nur ein Thread fähig sein, die Me-

thode `printSummary()` auszuführen, während die anderen Threads die Methode zu Ende abarbeiten und am Ende der Methode sterben. Durch diese Implementation wird sichergestellt, dass nur ein Thread weiterlebt und keine "Zombi-Prozesse" mehr vorhanden sind.

Weiter wird beim Abarbeiten dieser Methode die Methode `shutdown()` auf dem Client aufgerufen. Nach diesem Aufruf fährt sich der Client herunter und kappt alle akiven Verbindungen zum Server. Weitere Informationen zum Shutdown-Prozess des Clients sind unter der Kapitel des Frameworkclients ersichtlich.

Die oben genannte Implementation der `stopClient(String clientIP)`-Methode hat lange funktioniert. Während des Entwicklungsprozesses der RMI-Only Lösung lief das Framework mit der erwähnten Methode ohne Probleme. Bei den ersten Testläufen der RMI-Cache Lösung wurden beim Herunterfahren der Umgebung allerdings Exceptions generiert. Da die Fehlersuche bei Software mit einer Vielzahl von Threads sehr langwierig ist, nahm die Analyse des Fehlers viel Zeit in Anspruch.

Schlussendlich wurde bemerkt, dass die Methode `printSummary()` trotz aller Vorsichtsmassnahmen von jedem Client ein Mal ausgeführt wurde, was dazu führte, dass mehrere Zusammenfassungen generiert wurden und probiert wurde, den FrameworkServer mehrere Male herunterzufahren. Der Versuch den FrameworkServer mehrmals herunterzufahren, war natürlich der Grund für die Exceptions, da dem zweiten Client einfach der Verbindungspartner weggerissen wurde.

Der Grund des Fehlverhaltens lag in der nicht-funktionierenden Barriere `checkAllShutedDown()`. Nachdem der Methodenkopf durch "synchronized" erweitert wurde, funktionierte das Framework problemlos:

```
synchronized private void stopClient(String clientIp)
```

Anscheinend sind die Clients bei der Cache- Implementation immer zum fast gleichen Zeitpunkt fertig, was dazu führte, dass der Status "ShutedDown-.DOWN" nur unzuverlässig gesetzt wurde.

Fehleranalyse im Detail:

- Thread1 betritt die `stopClient(String clientIp)`-Methode und setzt seinen Status auf DOWN.
- Kurz nachdem Thread1 seinen Status gesetzt hat, wird ihm die Ressource entzogen und Thread2 kommt zum Zug.
- Thread2 betritt nun seinerseits die Methode und setzt seinen Status ebenfalls auf DOWN.

- Da nun alle Threads auf DOWN sind, wird die Barriere nicht mehr funktionieren und mehrere Threads führen `printSummary()` aus.

Dadurch, dass die Methode nun synchronized ist, ist gewährleistet, dass sich nur ein Thread in der Methode aufhalten kann. Das wiederum bedeutet, dass die Barriere `checkAllShutedDown()` nun wie erwartet funktioniert, da die Stati der Threads wieder zuverlässig gesetzt werden.

Report Generator

Beim Aufruf der Methode `printSummary()`, wird ein ReportGenerator instanziert, welche die Daten der Clients ausliest und in eine Textdatei schreibt. Folgende Werte wurden durch die Clients erfasst:

- Zeit, welche benötigt wurde um die entsprechende Action durchzuführen.
- Protokollierung, ob bei der Durchführung ein Konflikt generiert wurde oder nicht.
- Zeit, welche in einem Konfliktfall gebraucht wurde, um den Konflikt zu behandeln und die Operation noch einmal auszuführen.

Pro Client wird eine Datei erstellt, in welcher die Daten dargestellt werden. Folgend einen kurzen Auszug eines solchen Reports:

```
************************************************
Result for Client: 152.96.193.18 with ScenarioID: 1
OS: Linux / 3.0.0-17-generic-pae
************************************************
ActionNr;#ofTries;Time[ms];ACTION
0;0;42.674487;INCREMENT(READ) WITHOUT DELAY
0;1;77.969046;INCREMENT(WRITE) WITHOUT DELAY
1;0;79.684003;INCREMENT(READ) WITHOUT DELAY
1;1;81.984427;INCREMENT(WRITE) WITHOUT DELAY

------------------------------------------------
100% of all Action executed are successful
Total actions executed: 2, number of unsuccessful action 0
Total getBalance calls: 2, avg. execution time 61.179245 ms
Total setBalance calls: 2, avg. execution time 79.9767365 ms

Total Conflict: 0 / Gesamt Dauer: 282.311963 ms /
durch. Dauer pro Operation: 141.1559815 ms
```

Die genauen Messerwerte, mit Durchschnittswerten und Vergleichen zwischen den verschiedenen Systemen sind im Kapitel A dieser Arbeit zu finden.

Während der Entwicklung des Frameworks und der verschiedenen Systeme, schien die Ausgabe der Testergebnisse in eine Textdatei eine gute Idee zu sein. Während den Messungen des Systems wurde aber bemerkt, dass dies eine äusserst zeitraubende und mühsame Arbeit nach sich zieht: Für einen Testcase mit acht Clients, bei welchem der Messgenauigkeit zu liebe drei Durchläufe getätigt werden, fallen insgesamt 24 Textdateien an. Um einen Mittelwert der Ergebnisse zu bekommen, müssen alle Textdateien geöffnet werden, die Mittelwerte müssen berechnet und in einer weiteren Datei abgelegt werden.

Dieser Umstand wurde erst am Ende der Arbeit bemerkt, weshalb auf eine aufwändige Lösung verzichtet wurde. Mögliche Lösungsansätze für dieses Problem wären aber:

- Die Ergebnisse werden in eine Datenbank geschrieben.
- Die Ausgabe erfolgt in ein XML-File.

Die Ausgabe der Ergebnisse in eine Datenbank wäre die sauberste Lösung; sie ist jedoch auch mit sehr viel Aufwand verbunden. Ein XML-File wäre sicher einfacher zu handhaben als die Textdateien und hätte schlussendlich weniger Aufwand für die Messungen bedeutet.

Result Generator

Der ResultGenerator ist ein Generator, welcher für die Berechnung des zu erwartenden Endbestandes des Account-Objektes verantwortlich ist. Wenn alle Clients ihre Operationen abgeschlossen haben, wird das aktuelle Ergebnis des Account-Objektes und das erwartete Ergebnis auf der Konsole ausgegeben. Zusätzlich wird kontrolliert, ob die Ergebnisse übereinstimmen und eine entsprechende Meldung auf der Konsole ausgegeben:

```
19.04.2012 18:33:38 ch.hsr.objectCaching.testFrameworkServer.
Server printSummary
INFO: AccountBalance is: 1.4641001269340557
19.04.2012 18:33:38 ch.hsr.objectCaching.testFrameworkServer.
Server printSummary
INFO: AccountBalance should be: 1.4641001269340557
19.04.2012 18:33:38 ch.hsr.objectCaching.testFrameworkServer.
Server printSummary
INFO: No Lost-Updates!
```

Logger

Wie in dem Konsolenauszug unter 4.3.3 “Result Generator“ gesehen werden kann, wurde ein Logger eingebaut, welcher alle Methodenaufrufe protokoliert. Der Logger schreibt die Ereignisse erstens auf die Konsole und zweitens schreibt er alle Informationen zu einem Methodenaufruf in eine Datei. Diese Logdatei ist sehr informationsreich:

```
<record>
  <date>2012-04-05T17:58:43</date>
  <millis>1333641523151</millis>
  <sequence>9</sequence>
  <logger>TestFrameWorkServer</logger>
  <level>INFO</level>
  <class>ch.hsr.objectCaching.testFrameworkServer.
      MethodCallLogger</class>
  <method>methodCalled</method>
  <thread>11</thread>
  <message>setBalance got invoked by /152.96.193.9</
      message>
</record>
<record>
  <date>2012-04-05T17:58:43</date>
  <millis>1333641523198</millis>
  <sequence>10</sequence>
  <logger>TestFrameWorkServer</logger>
  <level>INFO</level>
  <class>ch.hsr.objectCaching.testFrameworkServer.
      Server</class>
  <method>setResults</method>
  <thread>12</thread>
  <message>Results from scenario 1 setted by
      152.96.193.9</message>
</record>
```

Die Notwendigkeit zur Implementierung eines Loggers wurde erst während des Projektes erkannt. Durch die vielen verschiedenen aktiven Threads war eine Überprüfung des korrekten Programmablaufs extrem schwierig. Mit der Hilfe des Loggers konnte dieses Problem behoben werden.

4.4 Testframework Client

Dieses Kapitel beschreibt die Aufgaben des Testframework Client sowie die Umsetzung der einzelne Komponenten. Der Testframework Client besitzt zwei Hauptaufgaben:

- Erzeugen und konfigurieren des benötigten ClientSystemUnderTest anhand von Parametern die vom Server kommen.
- Abarbeitung des durch den Framework Server zur Verfügung gestellten Szenarios auf dem zu testenden System sowie die Zeitmessungen der einzelnen Aktionen.

Folgende Abbildung zeigt die wichtigsten Interaktionen zwischen dem Testframework Server und dem Client.

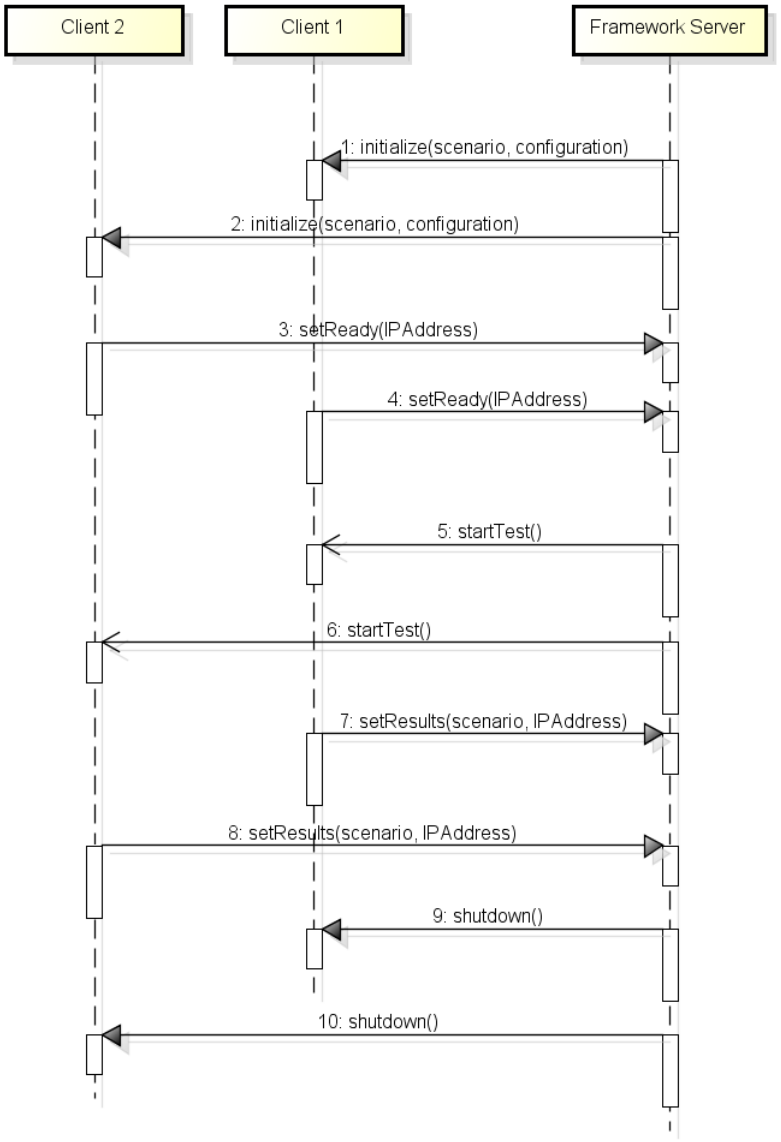

Abbildung 4.3: Kommunikation zwischen Server und Client

Die Umsetzung des Testframeworks wurde über das bewährte Client / Server Konzept realisiert, was zu einer schlanken Lösung auf Seiten des Clients führt. Der FrameworkClient und das ClientSystemUnderTest lassen sich dank diesem Konzept komplett vom Framework Server aus konfigurieren und steuern. Ein weiterer Vorteil dieses Designs liegt darin, dass sich neue Anforderungen jederzeit leicht umsetzen lassen.

4.4.1 ClientController

Beim Start der client.jar Datei kann das Startup-Script zwei Argumente mitgeben:

1. Pflicht Argument: Binding Namen
2. Optionales Argument: spezifischer RMI-Port

Die Kommunikation zwischen dem Testframework Client und dem Testframework Server ist über das in Java bereits vorhandene RMI [3] realisiert. Der Startvorgang eines Clients sieht wie folgt aus:

1. Es wird ein neues ClientController-Objekt instanziiert. Der ClientController implementiert das Client Interface, welches mit Remote markiert ist.
2. Aus dem instanziierten ClientController-Objekte wird ein Remoteobjekt generiert.
3. Der lokale Name Service wird auf dem spezifizierten Port gestartet. Ist kein Port als Argument mitgegeben worden, wird der Defaultport 1099 verwendet.
4. Das ClientController Remoteobjekt lässt sich nun in dem Java Name Service unter dem angegeben Binding Name veröffentlichen. Der Server ist jetzt in der Lage ein ClientController Stub über die IP Adresse und den Binding Name zu lokalisieren.

Über folgendes Client Interface wird der ClientController konfiguriert und gesteuert:

```
public void initialize(Scenario scenario,
Configuration configuration) throws RemoteException;
public void startTest() throws RemoteException;
public void shutdown() throws RemoteException;
```

Die Grundidee zu Beginn des Projekts war, dass alle Konfigurationswerte einzeln als Parameter an den ClientController übergeben werden. Es stellte sich jedoch schon nach kurzer Zeit heraus, dass eine Kapselung der benötigen Parameter in einem seperaten Objekt eine wesentlich bessere Lösung darstellt. Dieses neue Datatransfer Objekt beinhaltet sowohl alle nötigen Informationen für die Konfiguration des ClientSystemUnderTest als auch die des ClientControllers.

Initialize des ClientControllers

In der konkreten Implementation der Methode `initialize(Scenario scenario, Configuration configuration)` werden mehrere Aufgaben erledigt:

1. Aus dem Configuration Objekt wird die Server IP, der RMI Port und der Name, unter dem der FrameworkServer registriert ist, ausgelesen. Aus diesen Informationen wird eine neue RMI Verbindung zum Framework Server erzeugt.

2. Die Instanziierung des konkreten ClientSystemUnderTest übernimmt eine einzelne Klasse, welche als Factory implementiert ist. Der Factory wird nur der `fully qualified name`, welcher im Configuration Objekt gespeichert ist, übergeben. Die Factory selbst instanziiert den gewünschten ClientSystemUnderTest via Reflection und returniert das erzeugte Objekt.

3. Auf dem ClientSystemUnderTest Objekt wird nun ein neues Socket, welches zum ServerSystemUnderTest zeigt, gesetzt.

4. Ein neues Objekt vom Typ TestClient wird erzeugt und das aktuelle ClientSystemUnderTest Objekt wird im Konstruktor mitgegeben. Der TestClient ist die Schnittstelle zwischen dem ClientController und dem ClientSystemUnderTest.

5. Nun wird das Szenario, welches als Parameter übergeben wurde, auf dem TestClient Objekt gesetzt.

6. Daraufhin wird auf dem TestClient Objekt die `init()` Methode aufgerufen, in welcher sich der TestClient alle verfügbaren Account Objekte vom Server lädt.

7. Auf dem Testframework Server wird die Methode `setReady()` aufgerufen, dieser Methodenaufruf teilt dem Server mit, dass die Konfiguration des ClientControllers und des ClientSystemUnderTest erfolgreich war.

8. Im Anschluss wartet der ClientController, bis der TestFramework Server den Testlauf startet.

Start eines Testlaufs

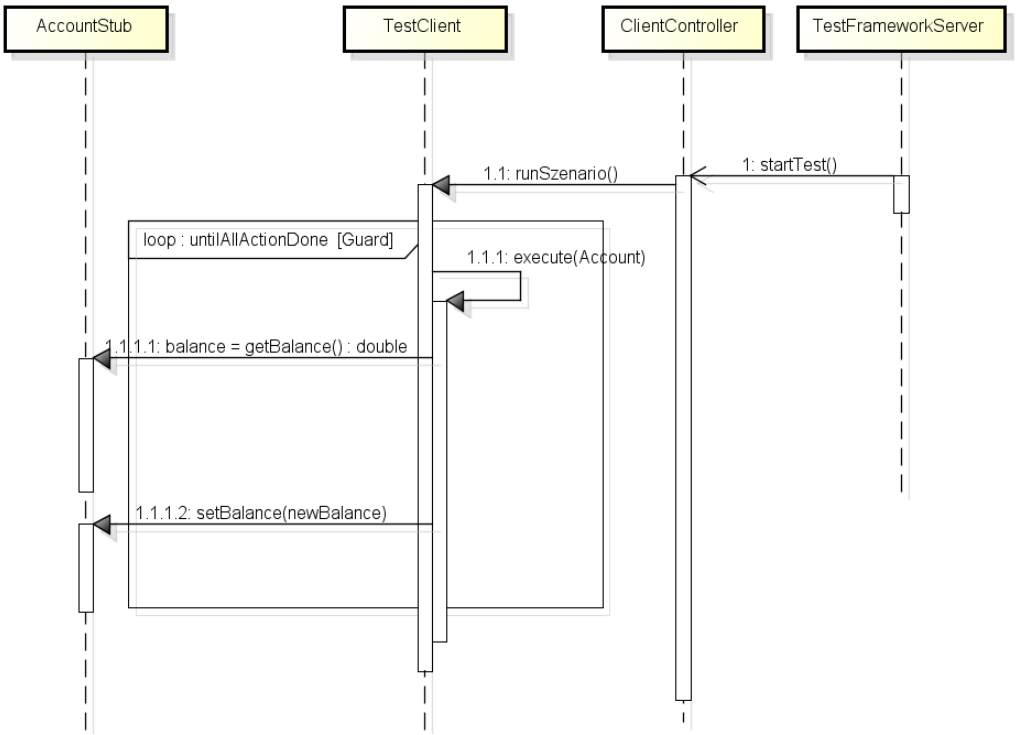

Abbildung 4.4: Start eines Testlauf durch den Testframework-Server

Das Sequenzdiagramm ist eine vereinfachte Darstellung des Testlaufs. Der Server startet den Testdurchlauf parallel auf allen ClientController über die Methode `startTest()`. Diese ruft auf dem TestClient `runScenario()` auf, eine detailierte Erklärung, was in dieser Methode passiert, ist im Kapitel "Test Client" 4.4.2 zu finden. Sind alle Aktionen des gegeben Szenarios abgearbeitet, wird das gesamte Szenario mit den gesammelten Messresultaten über die Methode `setResults(Scenario scenario, String IP)` zurück an den Server geschickt.

Herunterfahren des ClientController

Die Methode `shutdown()` beendet den TestframeworkClient und das ClientSystemUnderTest. Die `shutdown()` Methode besteht aus mehreren Teilschritten:

1. Das ClientSystemUnderTest wird heruntergefahren.

2. Die Verbindung zwischen dem spezifizierten Binding Namen und dem damit verbundenen ClientController Remoteobjekt wird im Name-Service von Java gelöscht.

3. Das ClientController Objekt wird von der RMI Runtime entfernt, dadurch sind keine RMI Aufrufe mehr möglich.

4. Der ClientController beendet sich selbst.

Beim Testen der shutdown()-Funktionalität konnte festgestellt werden, dass bei der Umsetzung wie oben beschrieben, der Server immer eine Exception bekommt. Es stellte sich heraus, dass das sofortige Herunterfahren des ClientControllers nach dem Unbinding und dem Unexport des ClientController Objekts dem Server keine Zeit lässt, seine RMI Verbindung zum ClientController korrekt zu schliessen. Die Lösung des Problems lag in einer Pause zwischen dem Unexport und dem Beenden des ClientControllers. Die Dauer der Verzögerung von zwei Sekunden stellte sich als ausreichend heraus. Folgender Codeausschnitt zeigt die Lösung des Problems:

```
private void shutdownClientController() throws RemoteException
{
  try {
Naming.unbind("rmi://localhost:" + ClientRmiPort + "/Client");
      } catch (MalformedURLException e1) {
  throw new RemoteException("Malformed URL has occurred
  in ClientController", e1);
} catch (NotBoundException e1) {
    throw new RemoteException("Unbinding the ClientController
    failed", e1);
}
  UnicastRemoteObject.unexportObject(this, true);
  closeClientController(2000);
}

private void closeClientController(final long delay) {
  new Thread() {
  @Override
  public void run() {
    logger.info("ClientController is shutting down");
    try {
  sleep(delay);
} catch (InterruptedException e) {}
```

```
  System.exit(0);
}
}.start();
}
}
```

Die Beendigung des ClientControllers wird in einem seperaten Thread ausgeführt, sodass der Main- Thread noch genügend Zeit hat, alle offenen Verbindungen ordnungsgemäss zu schliessen. Der Shutdown Thread beendet den ClientController nach der definierten Dauer mit `System.exit()`.

4.4.2 Test Client

Die TestClient Klasse ist die Schnittstelle zwischen dem ClientController und dem ClientSystemUnderTest. Alle Clientvarianten, die getestet werden sollen, müssen das Interface ClientSystemUnderTest implementieren, so lassen sich alle Varianten von ClientSystemUnderTest einheitlich mit einer TestClient Klasse testen. Desweiteren kann auf dem TestClient das Szenario gesetzt werden, welches der TestClient beim Aufruf von `runScenario()` durch den ClientController abarbeitet.

Das ClientSystemUnderTest Interface ist wie folgt definiert:

```
public interface ClientSystemUnderTest {
  public AccountService getAccountService();
  public void setServerSocketAdress(InetSocketAddress
  socketAdress);
  public void shutdown();
}
```

Ablauf

1. Der ClientController setzt die Socket-Adresse des Server in das ClientSystemUnderTest. Die nötigen Informationen sind im Configuration Objekt gespeichert.

2. Im Konstruktor des TestClients wird der AccountService, des übergebenen ClientSystemUnderTest über die Methode `getAccountService()` geladen.

3. Während der Initialisierungsphase des ClientController ruft dieser ebenfalls die `init()` Methode des TestClients auf, in dieser Methode wird eine Liste aller verfügbaren Accounts Objekte aus dem AccountService in den TestClient geladen.

4. Startet der Server nun den Testlauf wird endlos durch die Liste der Accounts interiert bis alle Aktionen des gesetzten Szenarios abgearbeitet sind.

5. Bei einem Shutdown durch den Testframework Server ist der TestClient ebenfalls für die ordnungsgemässe Beendigung des ClientSystemUnder-Test zuständig.

Concurrency im TestClient

Während den ersten Testmessungen mit den Cache-System zeigte sich folgendes Problem: Der letzte TestClient, der sich beim Server als bereit meldet, benötigte für die ersten Aktionen im Durchschnitt etwa 40 Milisekunden länger als die restlichen Clients. Durch die Analyse des Codes konnte das Problem lokalisiert werden. Die folgenden Schritte zeigen die Ursache des Problems auf:

1. Der letzte Client meldet sich über die Methode `setReady()` beim Server als bereit.

2. Beim Aufruf der `setReady()`-Methode auf dem Server wird nun geprüft, ob alle Clients bereit sind. Haben sich zuvor alle Clients als bereit für den Testlauf gemeldet, startet der Server auf jedem Client den Testlauf über die Methode `startTest()`.

3. Zu diesem Zeitpunkt wartet der setReady Thread des letzten Clients immer noch auf die Rückkehr des Aufrufs.

4. Gleichzeit wird auf dem letzten Client die `startTest()` Methode aufgerufen. Dadurch entstehen auf dem letzten Client, der sich beim Server gemeldet hat, für kurze Zeit zwei Threads.

5. Ist die Methode `setReady()` erfolgreich durchlaufen, kehrt der Aufruf zum letzten Client zurück. Daraufhin beendet der Thread auf dem Client, welcher die `setReady()`-Methode aufgerufen hat.

6. Gleichzeitig hat der `startTest()` Thread mit dem abarbeiten des Szenarios bereits begonnen. Da nun der Thread vom `setReady()` beendet wird, wird dem Szenario Thread die CPU-Ressource weggenommen. Dies führt nun zu einer Verlängerung der Ausführungsdauer der ersten Aktionen.

Die Lösung dieses Problem lag in einer Wartephase vor dem eigentlichen Abarbeiten des Szenarios. Diese Pause wurde über `Thread.sleep()` umgesetzt und wird vor der eingentlichen Messung gemacht. Dadurch haben alle ClientController Zeit, den `setReady()` Thread zu beenden, ohne den `startTest()` Thread zu beeinträchtigen.

4.5 Action

Ein Szenario beinhaltet eine Menge von Aktionen, welche auf ein Account Objekt ausgeführt werden können. Für eine einfache Erweiterbarkeit von neuen Aktionen, ist hierfür das Command-Pattern benutzt worden. Alle konkreten Implementierungen einer Aktion müssen von der abstrakten Klasse Action ableiten. Im Konstruktor dieser abstrakten Klasse wird ein neues Result Objekt instanziert. Im Kapitel "Result"4.6 wird dieser Typ genauer beschrieben. Dieses Objekt dient als Datenkontainer für alle Zeitmessungen die während dieser Aktion gesammelt werden sollen. Desweiteren deklariert Action drei abstrakte Methoden die von Subklassen implementiert werden müssen.

```
public abstract ActionTyp getActionTyp();
public abstract int getMinimalNumberOfTimeRecords();
public abstract void execute(Account account);
```

- Für eine detailierte Auswertung muss der Typ der Aktion zugänglich gemacht werden, dies ist über die Methode `getActionTyp()` möglich.
- Eine Aktion besteht aus einer Abfolge von `getBalance()` und `setBalance()` Aufrufen. Je nach Aktion können mehrere Methodenaufrufe auf das Account Objekt nötig sein, um die Aktion erfolgreich auszuführen. Um im Reporting festzustellen, ob die Aktion beim ersten Versuch erfolgreich war, muss die minimale Anzahl von TimeRecords in der Aktion gespeichert werden. Über die Methode `getMinimalNumberOfTimeRecords()` lässt sich diese Zahl auslesen. So lassen sich Konflikte, die bei der Ausführung dieser Aktion auftreten, im Reporting feststellen.
- Die Methode `execute(Account account)` beinhaltet die eigentlichen Methodenaufrufe, die auf dem Account-Objekt ausgeführt werden sollen. Das Account Objekt wird von TestClient an die Methode übergeben.

4.5.1 ActionTyp

Der ActionTyp wird zur Indentifizierung der Aktion benutzt und ist als Enum realisiert. Folgende Aktionstypen sind bereits vorhanden:

- `ReadAction`
- `WriteAction`
- `IncrementAction`

Die Werte dieses Enums werden zur Erkennung des Aktiontyps genutzt und werden bei der Report Generierung verwendet, um die einzelnen Aktionen zu unterscheiden.

4.5.2 IncrementAction

Die Aktion `IncrementAction` ist die Kernaktion für alle Szenarien. Sie besteht aus drei Teilschritten:

- Über die Methode `getBalance()` den aktuellen Kontostand des Accounts abfragen und zwischenspeichern.
- Den temporären Kontostand mit dem gegebenen Faktor multiplizieren.
- Versuchen den neuen Wert via `setBalance(balance)` Methode zurück in das Konto, welches auf dem Server liegt, zu schreiben.

```
@Override
public void execute(Account account) {
  boolean successful = false;
  double balance = 0;
  int numberOfTry = 0;
  while (!successful) {
    result.startTimeMeasurement(BasicAction.READ);
    balance = account.getBalance();
    result.stopTimeMeasurement();
    sleep(numberOfTry);
    try {
  result.startTimeMeasurement(BasicAction.WRITE);
  account.setBalance(balance * factor);
  result.stopTimeMeasurement();
  successful = true;
} catch (RuntimeException e) {
```

```
        successful = false;
     result.stopTimeMeasurement(ActionResult.FAILED);
}
  numberOfTry++;
}
}
```

Im Laufe der Realisierung stellte sich heraus, dass zwischen dem `getBalance()` und dem `setBalance()`-Aufruf eine minimal definierbare Zeitdauer gewartet werden können muss. Diese Warteperiode ist nötig, damit sich leichter ein Konflikt erzeugen lässt und so gezeigt werden kann, dass der Fehler serverseitig zu keinem Lost-Updates führt. Tritt serverseitig ein Concurrency Fehler auf, wird eine RuntimeExeption zurück an den Client geworfen. Dieser Fehler veranlasst, dass die Aktion nochmals neu gestartet wird. Die Aktion wird solange wiederholt bis diese erfolgreich aus der While-Schleife kommt. Die Verzögerung zwischen dem Lesen und Schreiben, welche über die Methode `sleep()` realisiert ist, führt nur beim ersten Druchlauf der IncrementAction zu einer zeitlichen Verzögerung.

4.5.3 WriteAction

Die WriteAction wurde nur zu Beginn der Realisierung benutzt und schreibt via `setBalance()` einen gegeben Wert auf einen Account.

4.5.4 ReadAction

Diese Aktion liest den aktuellen Kontostand eine Objekts und speichert diesen ab. Über `getBalance()` kann auf den Kontostand zugegriffen werden. Diese Aktion wurde für die Messungen benutzt.

4.6 Result

Für die möglichst genaue Zeitmessung einer Aktion wird die von Java bereitgestellte Methode `System.nanoTime()` genutzt. Die Klasse "Result"bietet eine `startTimeMeasurement(BasicAction type)` Methode, die ein neues TimeRecord Objekt erzeugt und die momentane Zeit in dieses TimeRecord Objekt schreibt. Die Information, ob es sich bei dieser Messung um eine lesende oder schreibende Methode handelt, wird über denn BasicAction Typ an den TimeRecord weiter gegeben. Diese ist nötig, damit bei der Auswertung genau analysiert werden kann, für welchen Methodeaufruf

(`getBalance()` / `setBalance()`) dieser TimeRecord erzeugt wurde. Über die Methode `stopTimeMeasurement(ActionResult result)` lässt sich die Zeitmessung beenden und der aktuelle TimeRecord wird in einer Liste gespeichert. Das Argument, welches in der `stopTimeMeasurment()` Methode übergeben wird, beinhaltet die Information über den Erfolg der Aktion. So lassen sich auch Aktionen erzeugen, die aus mehreren `getBalance()` oder `setBalance()` Aufrufe bestehen. Dies ist zum Beispiel in der `Increment Action` der Fall.

4.6.1 TimeRecord

Der TimeRecordtyp ist als reines Data Container implementiert worden. Es beinhaltet die Start- und die Stopzeit von genau einem Methodenaufruf. Für eine genaue Auswertung einer Aktion sind noch weitere Daten nötig. Daher besitzt dieser Typ zwei weitere Felder. Der BasicActiontyp beinhaltet Informationen für welche Account Methode, also `getBalance()` oder `setBalance()`, dieser Record gilt und in dem ActionResult kann die Information über den Erfolg der BasicAction gespeichert werden. Dadurch lassen sich zusammengesetzte Aktionen, welche aus `getBalance()` und `setBalance()` zusammengesetzt sind, einfach unterteilen. Ein weiterer Vorteil liegt in der Möglichkeit, ein Szenario bis in die einzelnen Methodenaufrufe einer Aktion zu analysieren.

4.6.2 BasicAction

Damit eine genaue Analyse der Aktionen gemacht werden kann, ist es nötig, dass zwischen `getBalance()` und `setBalance()` unterschieden werden kann. Eine Aktion kann aus einer Abfolge von Methodenaufrufen bestehen, daher muss bei jeder Zeitmessung angegeben werden können, ob es sich um eine “setter“ oder “getter“ Methode handelt; diese Information lässt sich daher ebenso im TimeRecord speichern.

4.6.3 ActionResult

Der Erfolg eines `getBalance()` oder `setBalance()` Aufrufs lässt sich ebenfalls im TimeRecord speichern. Dies hilft die Fehlerrate eines Methodenaufrufes zu ermitteln. Daher ist es nötig, dass beim Stoppen einer Messung der Erfolgstatus gespeichert wird.

Kapitel 5

Messergebnisse

Dieses Kapitel behandelt die Messergebnisse, welche mit dem zuvor beschriebenen System erzielt wurden. Dabei wird von verschiedenen Szenarien ausgegangen, welche die jeweiligen Vor- und Nachteile der zwei Systeme zum Ausdruck bringen sollen.

5.1 Laborumgebung

Alle Messergebnisse stammen aus einer Laborumgebung, mit folgenden Eigenschaften:

- Alle Testrechner, Server wie Client, besitzen folgende Hardware:
  - Prozessor: Intel Xeon CPU X3450, 8 x 2.67Ghz
  - Speicher: RAM 8GB
  - Netzwerkkarte: Intel 82578DM Gigabit Network Card
- Folgende Betriebssysteme wurden während den Messungen verwendet:
  - Server: Ubuntu, Version 11.10
  - Auf beiden Clients: Fedora Linux, Release 16
- Die Netzwerkinfrastruktur im Laborraum ist kaum belastet, nur durch den gebräuchlichen Leerlaufverkehr(STP, ARP usw.)
- Die Hardware-Ressourcen der Rechner, auf welchen die Applikation getestet wird, sind ausschliesslich nur durch die Applikation belegt und durch einzelne, übliche Prozesse vom Betriebsystem.

5.2 Testvorgaben

Die in dieser Arbeit ausgewiesenen Zahlen wurden alle in der Laborumgebung gemessen. Dabei wurde für jeden TestCase die Anzahl der Clients schrittweise von zwei bis acht Clients erhöht. Jeder Testlauf mit der gleichen Anzahl Clients, wurde drei Mal durchgeführt, um einen exakten Mittelwert der Ergebnisse ausweisen zu können. Drei Testdurchläufe pro Szenario und Clientanzahl wurden aufgrund der sehr dicht beieinander liegenden Ergebnisse der Testdurchläufe als genügend eingestuft. Wären die einzelnen Ergebnisse der Durchläufe extrem unterschiedlich ausgefallen, hätten fünf oder mehr Durchläufe stattfinden müssen.

5.3 Testergebnisse

In den folgenden Kapiteln werden die getesteten Szenarios und deren Ergebnisse gezeigt.
Die in der Legende der Tabellen beschriebenen Funktionen lassen sich wie folgt erklären:

setBalance() Die durchschnittliche Zeit, welche benötigt wird, um einen schreibenden Zugriff zu tätigen.

getBalance() Die durchschnittliche Dauer, um einen lesenden Zugriff zu tätigen.

Number of Conflict Anzahl Konflikte, welche während des Testlaufs aufgetreten sind.

Total Time(with Delays) Beschreibt die Zeit, welche gebraucht wurde, um das gesamte Szenario, inklusive Konfliktbewältigung und konfigurierte Verzögerung, durchzuspielen.

Pure Operation Time Die für das Durchspielen des Szenarios benötigte Zeit, ohne die konfigurierten Verzögerungen.

5.3.1 Nur lesende Clients

Szenario Code

Dieser TestCase sieht folgendermassen aus:

```
<?xml version='1.0' encoding='UTF-8'?>
<TestRun>
```

```
  <TestCase
    ClientSystemUnderTest="ch.hsr.objectCaching.
        rmiWithCacheClient.RMIwithCacheClientSystem"
    ServerSystemUnderTest="ch.hsr.objectCaching.
        rmiWithCacheServer.RMIWithCacheServerSystem">
    <Account balance="1"></Account>
      <Scenario id="1">
        <ActionSequence>
          <Read count="250"></Read>
        </ActionSequence>
      </Scenario>
  </TestCase>
</TestRun>
```

Szenariobeschreibung

Bei diesem Testcase führen alle Clients die gleichen Operationen aus. Alle Clients lesen das Objekt auf dem Server genau 250 Mal. Das Account-Objekt wird mit dem Wert "1" initialisiert, welcher bis zum Schluss unverändert bleiben wird.

Dieses Testszenario soll zeigen, welches der beiden System bei nur Lesezugriffen besser skaliert.

Ergebnisse System ohne Cache

Die folgende Tabelle stellt die Messergebnisse des Systems ohne Cache dar:

Legende	2 Clients	3 Clients	4 Clients	5 Clients
setBalance[ms]	0	0	0	0
getBalance[ms]	86.182	89.502273	91.313	94.490998
Number of Conflict	0	0	0	0
TotalTime(with Delays)[ms]	21546.54	22376.494	22829.21	23623.678
PureOperationTime[ms]	21545.686	22375.568	22828.264	23622.749

Legende	6 Clients	7 Clients	8 Clients
setBalance[ms]	0	0	0
getBalance[ms]	98.381525	105.14788	120.2947
Number of Conflict	0	0	0
TotalTime(with Delays)[ms]	24596.331	26287.93	30074.669
Pure Operation Time[ms]	24595.381	26286.969	30073.691

Diese Zahlen sind Durchschnittswerte von drei Messungen. Die genauen Messdaten können im Anhang gefunden werden. Die folgende Grafik zeigt

den Verlauf des Zeitaufwandes mit einer steigenden Anzahl Clients von zwei bis acht Clients:

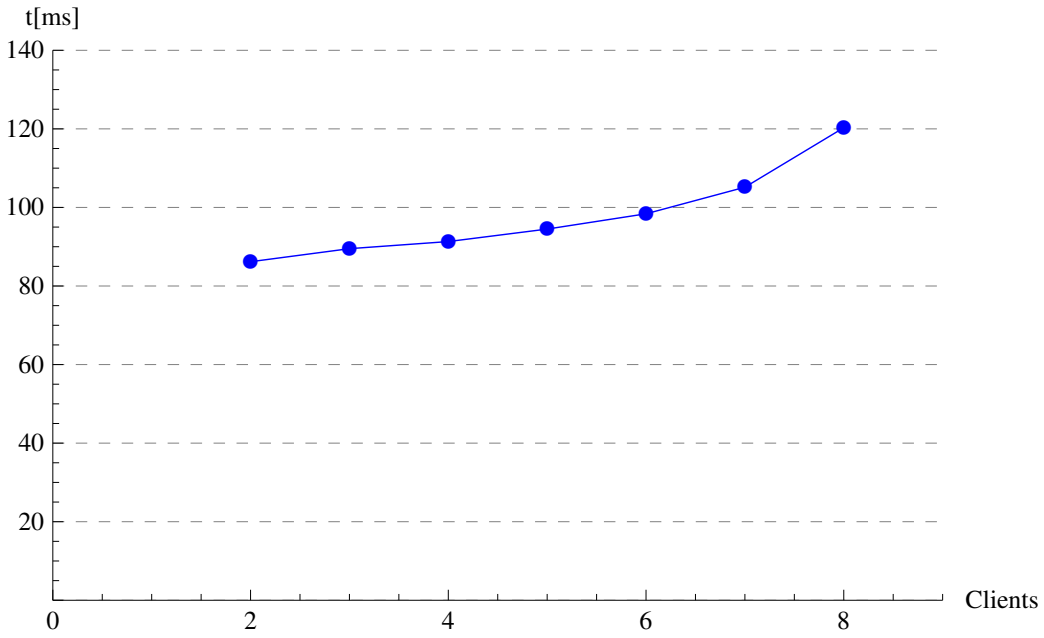

Abbildung 5.1: `getBalance()`-Zeitaufwand, System ohne Cache(nur lesende Zugriffe)

Es wird schnell ersichtlich, dass das System mit einer steigenden Anzahl Clients langsamer wird. Dies erscheint durchaus logisch, da jeder `getBalance()`-Aufruf an den Server gesendet wird. Wird der Server nun von mehreren Clients angefragt, verschlechtern sich auch dessen Antwortzeiten, was eine Verlangsamung des Aufrufs zur Folge hat.

Ergebnisse System mit Cache

In der folgenden Tabelle werden die Messdaten des Cache-Systems dargestellt:

Legende	2 Clients	3 Clients	4 Clients	5 Clients
setBalance[ms]	0	0	0	0
getBalance[ms]	0.288	0.287	0.285	0.298
Number of Conflict	0	0	0	0
TotalTime(with Delays)[ms]	73.087	72.581	72.039	75.422
PureOperationTime[ms]	72.249	71.855	71.304	74.603

Legende	6 Clients	7 Clients	8 Clients
setBalance[ms]	0	0	0
getBalance[ms]	0.3011	0.3201	0.281
Number of Conflict	0	0	0
TotalTime(with Delays)[ms]	76.014	80.838	71.001
Pure Operation Time[ms]	75.289	80.041	70.262

Diese Zahlen sind wiederum Durchschnittswerte. In der folgenden Grafik werden die durchschnittlichen Zeiten der `getBalance()`-Methode in Abhängigkeit der Anzahl Clients ausgegeben:

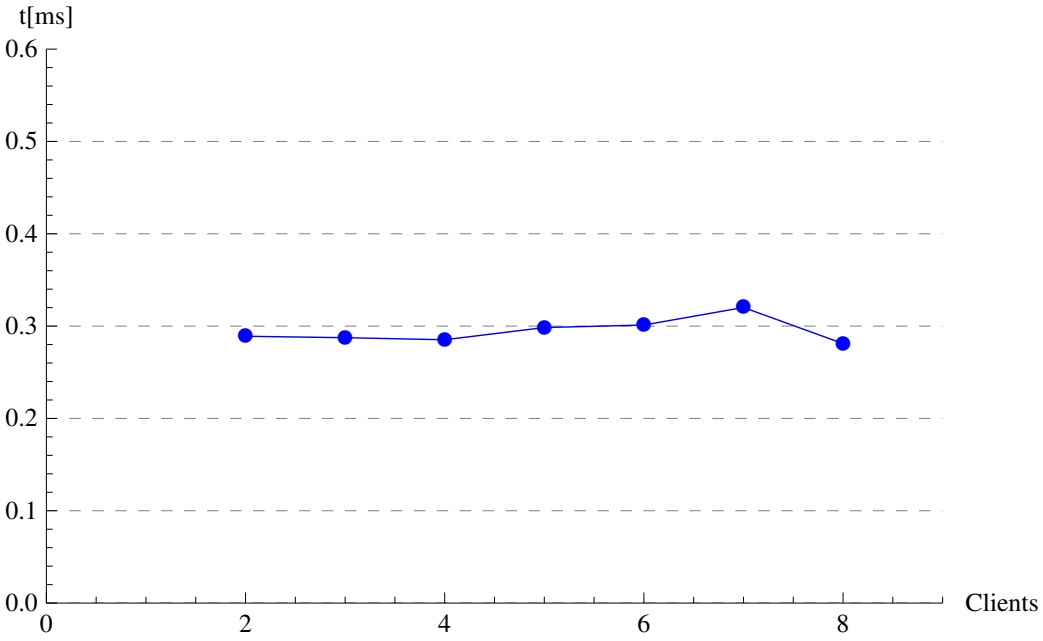

Abbildung 5.2: `getBalance()`-Zeitaufwand, System mit Cache(nur lesende Zugriffe)

Damit die Grafik einen Sinn ergibt, mussten die Zeiten auf der Y-Achse angepasst werden. Hätte man die Skala aus dem RMI-Only-System übernommen, wäre auf der Grafik nur eine Linie ersichtlich. Es muss dennoch bemerkt werden, dass bei dieser Grafik die Skala auf der Y-Achse extrem klein ist und Messunterschiede, wenn auch nur minimale, in dieser Grafik stark zum Tragen kommen. Ein Vergleich mit der vorhergehenden Grafik des Systems ohne Cache ist also mit Vorsicht zu geniessen.

Aus der Grafik wird ersichtlich, dass bei nur lesenden Zugriffen das System perfekt skaliert. Die Werte betragen circa alle, egal wieviele Clients beteiligt sind, 0.3ms.

Interpretation

Vergleicht man die Messergebnisse der zwei Systeme miteinander, fallen zwei Merkmale auf:

- Jeder einzelne getBalance-Aufruf dauert beim System ohne Cache extrem viel länger
- Das Cache-System skaliert viel besser als das System ohne Cache

Da jeder getBalance-Aufruf zum Server, das heisst über das Netzwerk, übertragen werden muss, ist es logisch, dass das System im Durchschnitt viel langsamer ist. Beim Cache-System wird nur der erste Aufruf zum Server übertragen, alle folgenden Aufrufe werden lokal aus dem Cache beantwortet. Dies erklärt den enormen Geschwindigkeitsvorteil des Cache-Systems bei jedem einzelnen Aufruf.

Weiter skaliert das Cache-System bei diesem Testcase um einiges besser. Dadurch, dass bei steigender Anzahl Clients auch immer mehr Clients ein Update erhalten müssen, hätte man auch eine leichte Steigung der Messwerte beim Cache-System erwarten können. Da die Werte aber so klein sind und die Messmethode mit der in Java implementierten Funktion `nanoTime()` gemacht wurde, können kleine Ungenauigkeiten in diesem Bereich nicht ganz ausgeschlossen werden. Es bleibt auf jeden Fall zu beachten, dass das Cache-System wunderbar skaliert. Vermutlich hätte das Szenario mit einer Vielzahl der Clients durchgeführt werden können, ohne grosse Geschwindigkeitseinbussen in Kauf nehmen zu müssen.

Auf der anderen Seite skaliert das System ohne Cache weitaus schlechter. Bei der Erhöhung der Clients von vier bis acht, also rund eine Verdopplung der Clients, wächst die durchschnittliche Zeit zum Ausführen der getBalance-Methode um etwa 30ms. Dies entspricht einem Drittel der Zeit, welche ein `getBalance()`-Aufruf bei vier Clients benötigt. Würde man die Anzahl Clients noch verdoppeln oder sogar verdreifachen, würde die benötigte Zeit zum Ausführen der `getBalance()`-Methode vermutlich bald eine Schmerzgrenze der Wartezeiten durchbrechen.

5.3.2 Ein schreibender, mehrere lesende Clients

Szenario Code

Das Szenario ist wie folgt aufgebaut:

```
<?xml version='1.0' encoding='UTF-8'?>
<TestRun>
```

```
<TestCase
   ClientSystemUnderTest="ch.hsr.objectCaching.
      rmiWithCacheClient.RMIwithCacheClientSystem"
   ServerSystemUnderTest="ch.hsr.objectCaching.
      rmiWithCacheServer.RMIWithCacheServerSystem">
     <Account balance="1"></Account>
       <Scenario id="1">
         <ActionSequence>
           <Increment count="900" delay="0" factor="
              1.1"></Increment>
         </ActionSequence>
       </Scenario>
       <Scenario id="2">
         <ActionSequence>
           <Read count="3000" delay="100"></Read>
         </ActionSequence>
       </Scenario>
       <Scenario id="3">
         <ActionSequence>
           <Read count="3000" delay="100"></Read>
         </ActionSequence>
       </Scenario>
       ...
       ...
```

Das Szenario ist nicht abschliessend dargestellt, da sich nun bis zum Szenario mit der Id acht, die Szenariodefinitionen wiederholen. Der erste in der Clientliste eingetragene Client, wird immer den schreibenden Auftrag bekommen. Die anderen eins bis sieben Clients werden nur den Auftrag bekommen zu lesen.
Für diesen Testcase wurde bei der Leseaktion extra noch ein Delay eingebaut. Die Clients müssen so lange lesen, wie der schreibende Client am Arbeiten ist. Wären die lesenden Clients schon vor dem schreibenden Client fertig, würden sie sich herunterfahren und keine Updates mehr erhalten. Dies würde den Testcase verfälschen und somit nutzlos machen. Damit nun keine Million Reads angegeben werden mussten, wurde ein Delay eingebaut, damit die Clients lange genug aktiv bleiben.

Szenariobeschreibung

In diesem Szenario geht es darum, dass nur ein Client den Balance-Wert erhöht und alle anderen Clients diesen Wert lesen. Es soll daher gezeigt wer-

den, wie stark sich die Updates, welche beim System mit Cache versendet werden, auf die Performance des Systems auswirken. Weiter ist von Interesse, wie stark die Performance bei einer steigenden Anzahl Clients, daher einer steigenden Anzahl Updates, tangiert wird.

Die folgende Auswertung der Messresultate beschränkt sich auf den schreibenden Client. Es soll untersucht werden, ob und wie stark sich eine steigende Anzahl lesender Clients auf die Schreibdauer eines Clients auswirkt.

Ergebnisse System ohne Cache

Die Messwerte des schreibenden Clients sehen wie folgt aus:

Legende	2 Clients	3 Clients	4 Clients	5 Clients
setBalance[ms]	79.870	79.951	79.821	79.858
getBalance[ms]	80.124	80.140	80.114	80.142
Number of Conflict	0	0	0	0
TotalTime(with Delays)[ms]	80003.557	80051.669	79973.797	80006.288
PureOperationTime[ms]	79997.736	80045.911	79968.060	80000.453

Legende	6 Clients	7 Clients	8 Clients
setBalance[ms]	79.914	79.876	80.081
getBalance[ms]	80.133	80.124	80.201
Number of Conflict	0	0	0
TotalTime(with Delays)[ms]	80029.782	80006.137	80105.944
Pure Operation Time[ms]	80024.009	80000.355	80100.144

Die Ergebnisse der `setBalance()`-Aurufe werden in dieser Abbildung grafisch dargestellt:

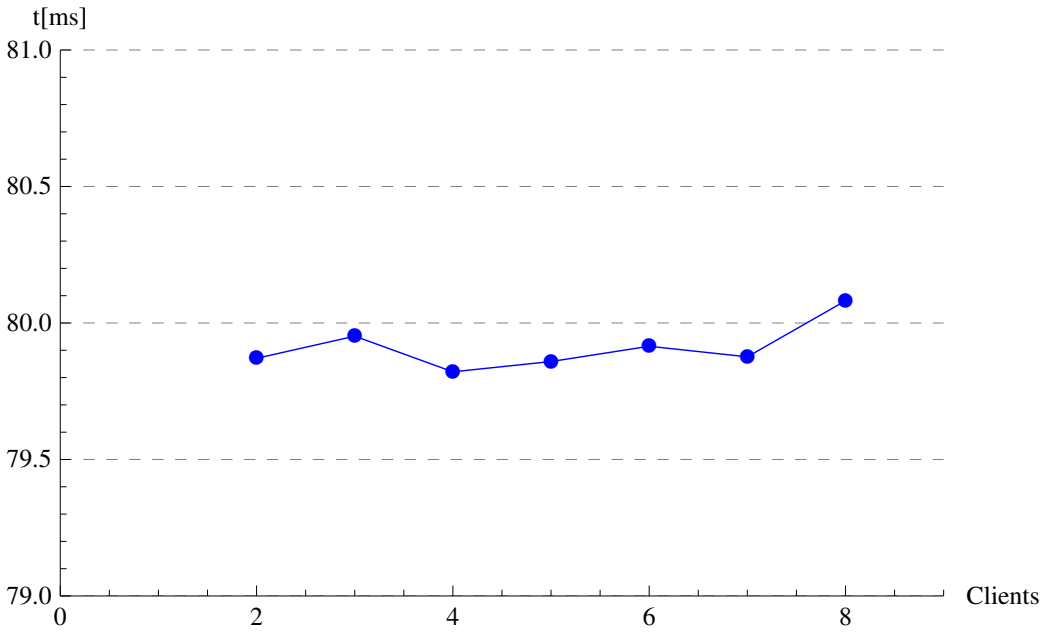

Abbildung 5.3: `setBalance()`-Zeitaufwand, System ohne Cache(lesende/schreibende Zugriffe)

Das System ohne Cache skaliert sehr gut. Die Werte für einen schreibenden Zugriff bleiben stabil.

Ergebnisse System mit Cache

Folgende Tabelle zeigt die Messergebnisse des schreibenden Clients mit dem Cache-System:

Legende	2 Clients	3 Clients	4 Clients	5 Clients
setBalance[ms]	88.7521	88.789	88.603	88.620
getBalance[ms]	0.1837	0.184	0.185	0.184
Number of Conflict	0	0	0	0
TotalTime(with Delays)[ms]	80144.642	80149.519	79982.694	80025.973
PureOperationTime[ms]	80131.176	80136.188	79969.438	80012.822

Legende	6 Clients	7 Clients	8 Clients
setBalance[ms]	88.655	88.766	88.920
getBalance[ms]	0.189	0.180	0.192
Number of Conflict	0	0	0
TotalTime(with Delays)[ms]	80003.756	80093.180	80244.857
Pure Operation Time[ms]	79990.542	80081.593	80231.319

In der folgenden Grafik ist der Verlauf der Messwerte der `setBalance()`-Aufrufe des schreibenden Clients zu sehen:

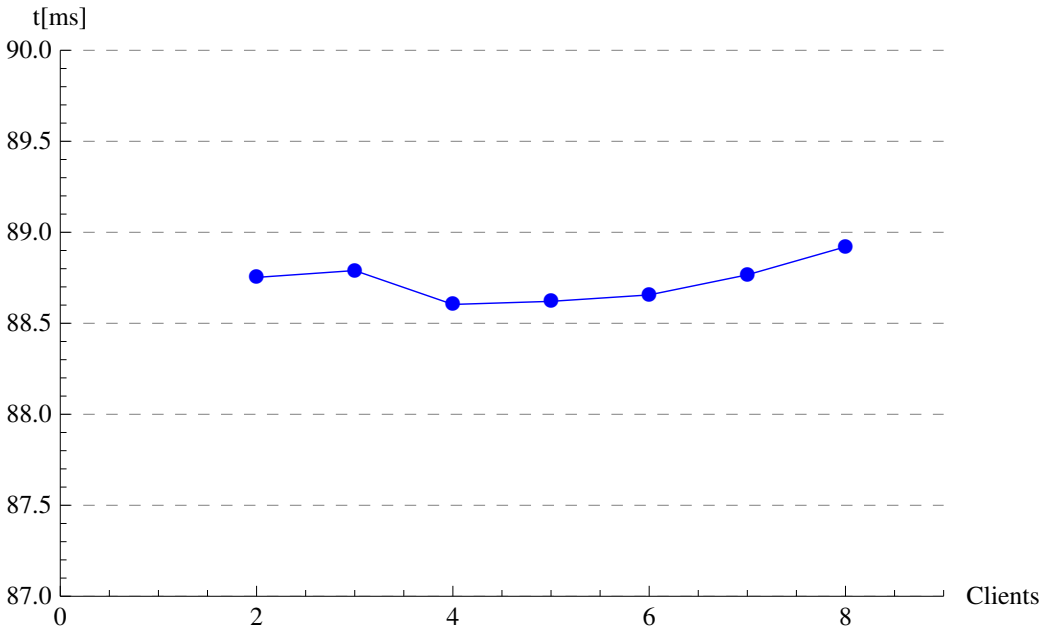

Abbildung 5.4: `setBalance()`-Zeitaufwand, System mit Cache(lesende/schreibende Zugriffe)

Das System mit Cache skaliert sehr gut. Die gemessenen Zeiten der `setBalance()`-Aufrufe steigen auch mit zunehmenden Clients nicht an.

Interpretation

Aufgrund der Messergebnisse kann geschlossen werden, dass die Updates des Servers keinen Einfluss auf die Performance der Systeme hat. Da die Ergebnisse stabil bleiben, egal wie viele Clients beteiligt sind, kann eine starke Beeinträchtigung des Servers und der Clients ausgeschlossen werden. Es ist daher anzunehmen, dass die Performancewerte weiter stabil bleiben würden, auch wenn die Anzahl Clients um ein Vielfaches steigen würde.

Auch das System ohne Cache skaliert sehr gut. Die Werte der `setBalance()`-Aufrufe sind stabil und liegen alle nahe beieinander. Aufgrund der Zahlen ist ersichtlich, dass der Server die Anfragen schnell und zuverlässig abfertigen kann. Trotz der Steigerung der Clients, konnten keine Performanceverluste ausgemacht werden. Dies lässt darauf schliessen, dass der Server

durch die Anfragen nicht stark belastet wurde.

Abschliessend lässt sich zu diesem Testcase sagen, dass weder das eine, noch das andere System abfällt. Beide arbeiten stabil und fehlerlos und wenn nur die `setBalance()`-Werte betrachtet werden, auch fast gleich schnell. Die langsamen `getBalance()`-Aufrufe bleiben ein grosser Nachteil des Systems ohne Cache.

5.3.3 Nur schreibende Clients

Szenario Code

Der XML-Code, welcher dieses Szenario beschreibt, ist hier zu sehen:

```
<?xml version='1.0' encoding='UTF-8'?>
<TestRun>
  <TestCase
    ClientSystemUnderTest="ch.hsr.objectCaching.
      rmiWithCacheClient.RMIwithCacheClientSystem"
    ServerSystemUnderTest="ch.hsr.objectCaching.
      rmiWithCacheServer.RMIWithCacheServerSystem">
    <Account balance="1"></Account>
      <Scenario id="1">
        <ActionSequence>
          <Increment count="1000" delay="0" factor="1.1
            "></Increment>
        </ActionSequence>
      </Scenario>
  </TestCase>
</TestRun>
```

Szenariobeschreibung

In diesem Testcase führt jeder Client 1'000 Mal ein Increment auf den Balance-Wert aus. Ein Increment bedeutet den Wert zu lesen und ihn dann mit einem Faktor multipliziert wieder zu schreiben.

Ziel dieses Szenarios ist zu zeigen, wie sich die zwei Systeme bei permanentem Schreibzugriff verhalten. Dieses Szenario stellt eine enorme Belastung für den Server dar, da er mit Anfragen geradezu bombadiert wird. Durch die vielen Anfragen wird das System auch eine längere Zeit arbeiten müssen und so zum Vorschein bringen, ob es stabil und über längere Zeit sicher und fehlerlos läuft.

Bei steigender Anzahl Clients, wird es auch mehr Schreibzugriffe auf den Server geben. Dies führt zu mehr Netzwerkverkehr und vor allem zu mehr Last auf dem Server, welcher alle Anfragen beantworten muss. Es soll ersichtlich werden, welches Serversystem mit dieser erhöhten Last besser umgehen kann.

Ein weiteres Ziel dieses Szenarios ist zu testen, ob und wie stark sich die unterschiedliche Implementierung des Concurrency-Control Mechanismusses auf die Performance auswirkt. Das System mit Cache verfügt über zwei Mechanismen, einer im Cache und der zweite auf dem Server. Beim System ohne Cache ist nur ein Concurrency-Control auf dem Server implementiert.

Ergebnisse System ohne Cache

Die Messerwerte des RMI-Only-Systems:

Legende	2 Clients	3 Clients	4 Clients	5 Clients
setBalance[ms]	80.119789	80.245101	80.301833	80.341942
getBalance[ms]	79.877294	79.774314	79.754846	79.772539
Number of Conflict	500	1208.44	1661.33	2367.0667
TotalTime(with Delays)[ms]	240019.37	353423	415639.29	539152.12
PureOperationTime[ms]	240003.25	353400.43	425985.248	539121.21

Legende	6 Clients	7 Clients	8 Clients
setBalance[ms]	80.308807	80.276285	80.30729
getBalance[ms]	79.700759	79.73294	79.710938
Number of Conflict	2902.2778	3242.90	4371.66
TotalTime(with Delays)[ms]	624456.67	678747.88	859636.25
Pure Operation Time[ms]	624422.52	678712.13	859593.5

Die nächste Grafik zeigt die Messdaten der `setBalance()`-Methode:

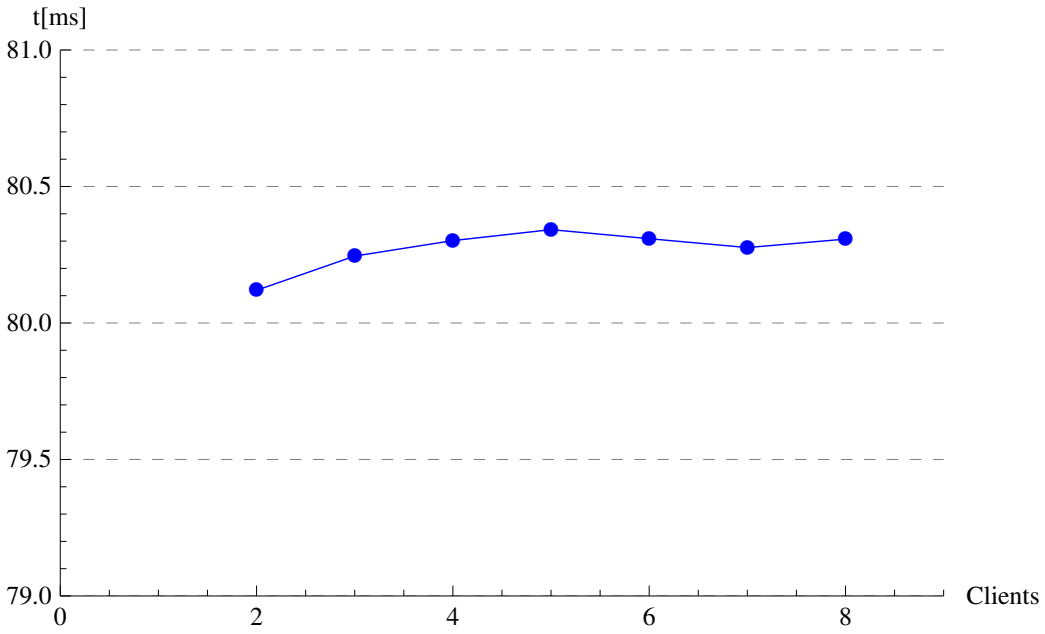

Abbildung 5.5: `setBalance()`-Zeitaufwand, System ohne Cache(nur schreibende Zugriffe)

Die Werte der `setBalance()`-Aufrufe bleiben stabil bei 80ms. Die Anzahl der Konflikte steigt mit der steigenden Anzahl Clients.

Die Latenz von etwa 80ms kommt auf Grund des Netzwerks zustande. Wie in den ausgewiesenen Zahlen der vorhergehenden Testcases gesehen werden konnte, benötigt der durchschnittliche Aufruf über das Netzwerk etwa 80ms.

Folgende Grafik stellt die durchschnittliche Anzahl Konflikte dar:

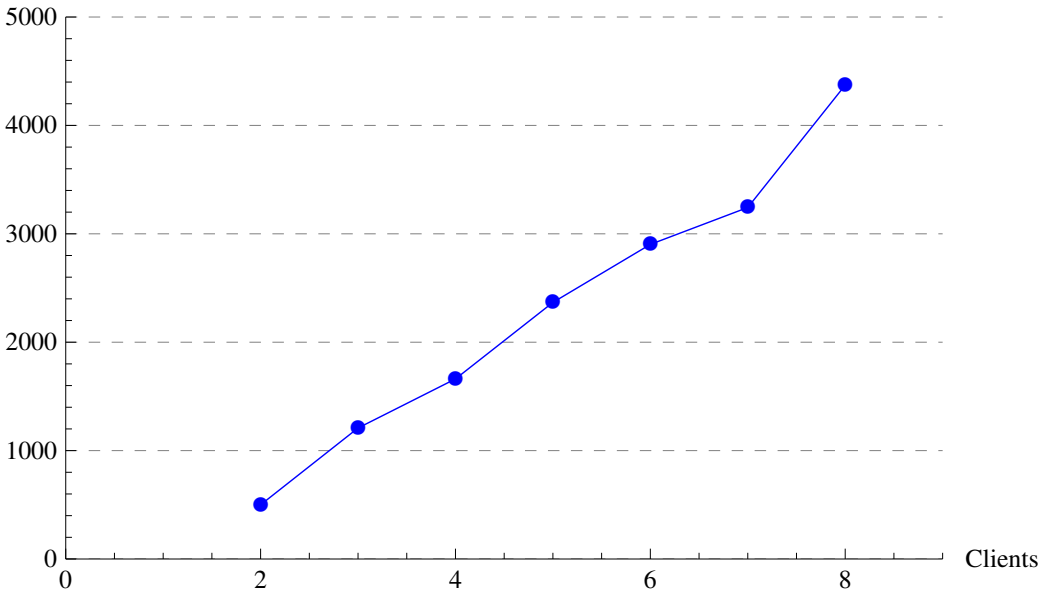

Abbildung 5.6: Anzahl Konflikte, System ohne Cache(nur schreibende Zugriffe)

Die Grafik zeigt eine lineare steigende Gerade. Jeder dazukommende Client verursacht im ganzen System circa 1000 Konflikte mehr, bei einer Operationsanzahl von 1000 Operationen.

Die “Pure Operation Time“ ist die Summe der Zeiten aller ausgeführten Operationen:

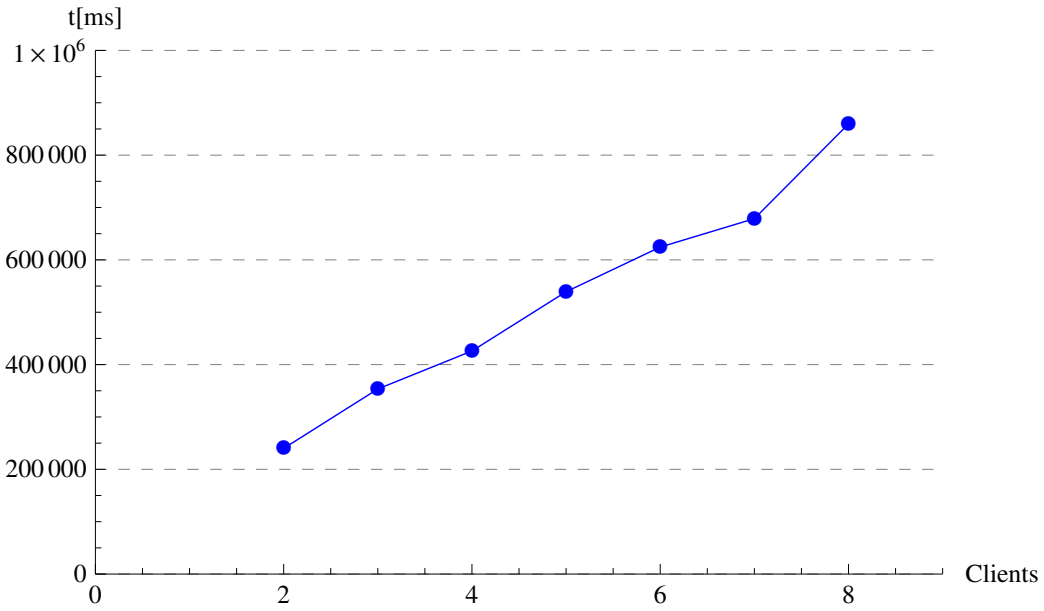

Abbildung 5.7: Pure Operation Time System ohne Cache(nur schreibende Zugriffe)

Obwohl die `setBalance()`-Aufrufe immer konstant gleich viel Zeit benötigen, steigt durch die Konfliktanzahl die "Pure Operation Time" an. Da die Gerade der `setBalance()`-Methode keine Steigung aufweist und die Konfliktanzahl linear steigt, ist bei der "Pure Operation Time" auch eine linear steigende Gerade zu sehen. Der Zusammenhang wird hier noch einmal dargelegt:

- Bei steigender Clientanzahl, steigt auch die Anzahl der Konflikte.
- Für jeden Konflikt muss noch einmal eine Operation getätigt werden. Das heisst, die Anzahl Operationen steigt mit der Anzahl Konflikte.
- Da die Zeit, welche für eine Operation benötigt wird, immer gleich bleibt, steigt die Gerade der "Pure Operation Time" analog zu der Geraden der Konfliktanzahl.

Es ist anzunehmen, dass die Werte des Systems ohne Cache linear weiter steigen würden, bei einer steigenden Anzahl Clients.

Ergebnisse System mit Cache

In der folgenden Tabelle sind die Werte des Cache-Systems dargestellt:

Legende	2 Clients	3 Clients	4 Clients	5 Clients
setBalance[ms]	104.2852	130.9624	247.2939	427.0246
getBalance[ms]	0.1346	0.1007	0.1014	0.0993
Number of Conflict	681.5	1532.22	1916.5	2243.8
TotalTime(with Delays)[ms]	17607.64	331890.26	721508.42	1385805
PureOperationTime[ms]	176052.29	331868.2	721482.83	1385777

Legende	6 Clients	7 Clients	8 Clients
setBalance[ms]	648.4426	981.2542	1353.4886
getBalance[ms]	0.1008	0.0972	0.0907
Number of Conflict	2534.5	2871.33	3254.70
TotalTime(with Delays)[ms]	2303134	3782179	5761716
Pure Operation Time[ms]	2303103	3800575	5761681

Die folgende Grafik zeigt die Entwicklung der Messwerte der `setBalance()`-Aufrufe:

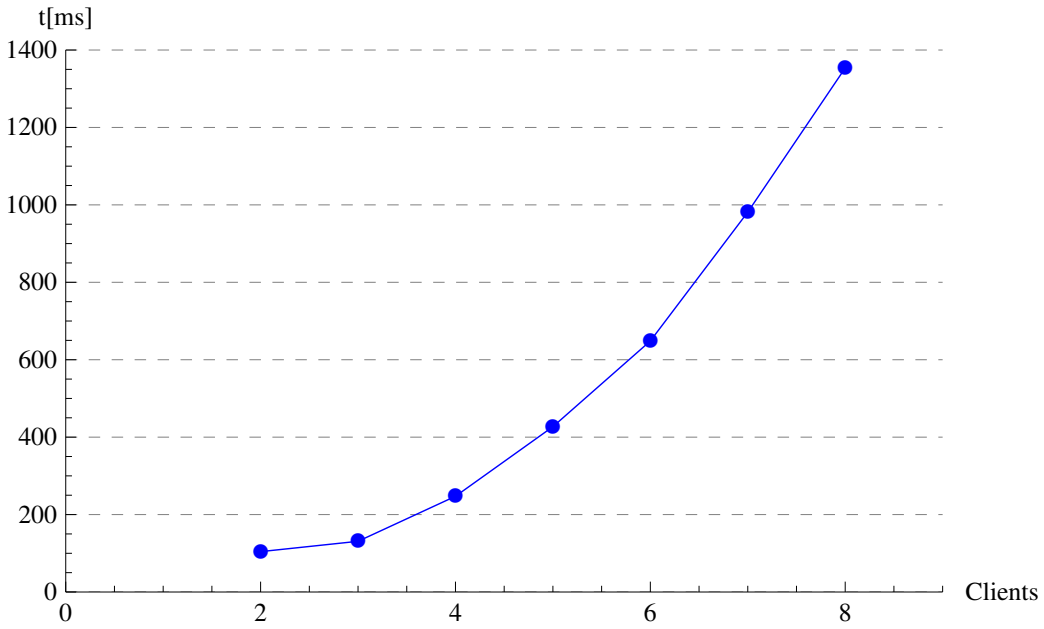

Abbildung 5.8: `setBalance()`-Zeitaufwand, System mit Cache(nur schreibende Zugriffe)

Wenn alle beteiligten Clients schreibende Zugriffe tätigen, skaliert das System mit Cache sehr schlecht.

In der nächsten Grafik wird die durchschnittliche Anzahl Konflikte pro Client gezeigt:

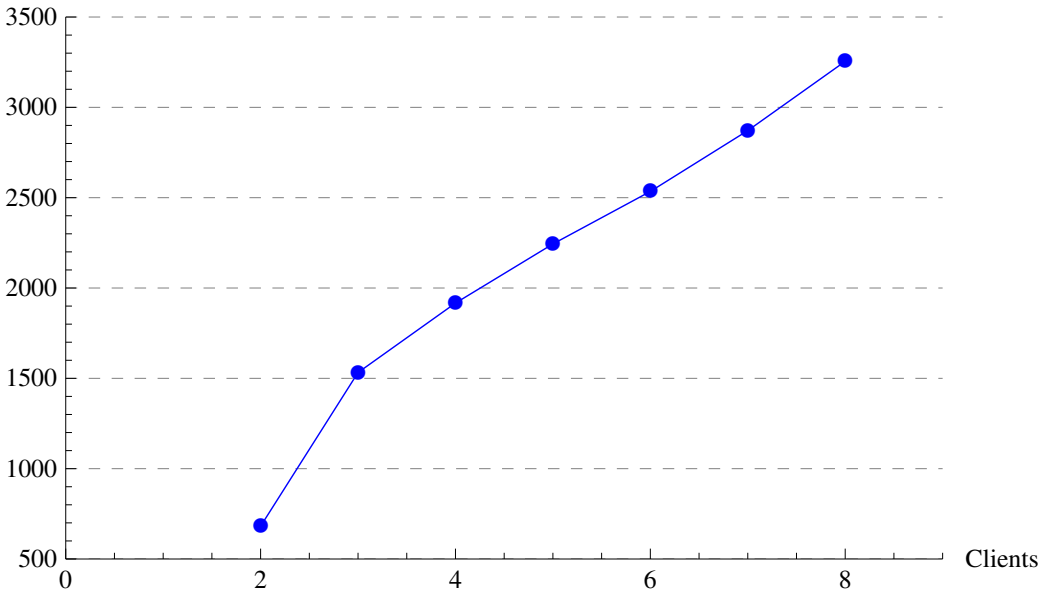

Abbildung 5.9: Anzahl Konflikte, System mit Cache(nur schreibende Zugriffe)

Die Anzahl der Konflikte nimmt linear mit der Anzahl Clients zu. Dies zeugt von einer schwachen Skalierbarkeit, da davon ausgegangen werden kann, dass diese Funktion linear weitersteigen würde, würden noch mehr Clients in den Test einbezogen.

Pro zusätzlicher Client ergibt sich eine Zunahme der Konflikte um 250. Würde die Funktion analog weitersteigen, und davon ist auszugehen, würde die Anzahl Konflikte pro Client vermutlich auf über 6000 ansteigen, bei 20 Clients. Dennoch sollte bedacht werden, dass dieser Testcase ein Extrem darstellt, da jeder Client mit vollem Tempo zu schreiben versucht.

Die Zeiten der `setBalance()`-Aufrufe steigen stark an, während die Anzahl der Konflikte nur linear steigt. Die Behandlung von 3000 Konflikten verursacht also eine ungemein höhere Zeitverzögerung, als dies 2000 Konflikte tun.

Die nächste Grafik beschäftigt sich mit der reinen Durchführungszeit:

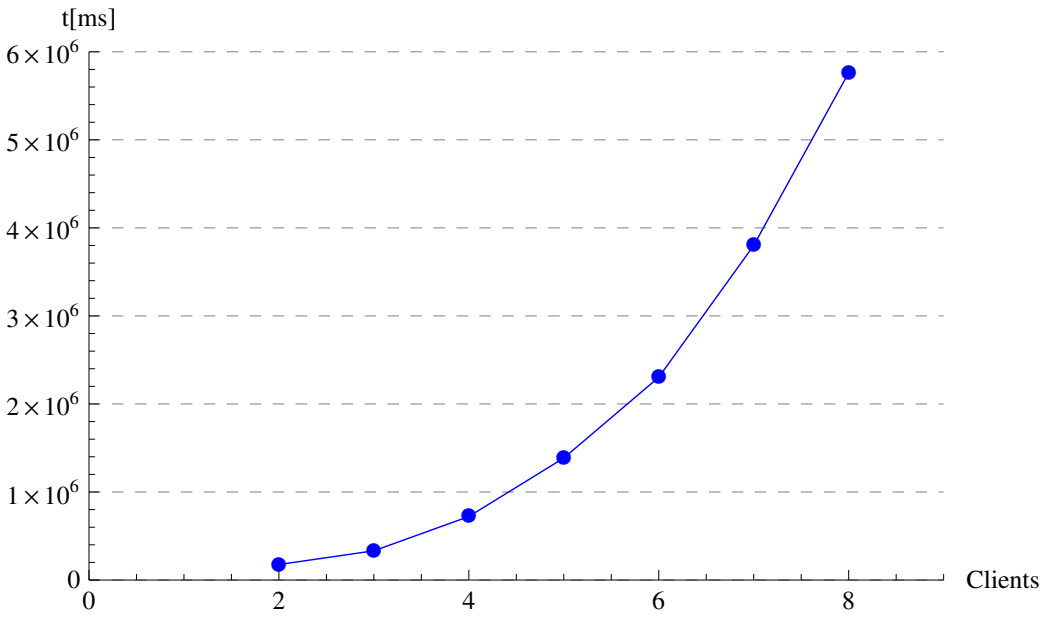

Abbildung 5.10: Pure Operation Time, System ohne Cache(nur schreibende Zugriffe)

Die starke Steigung dieser Funktion war durch die starke Steigung der `setBalance()`-Funktion abzusehen. Zwischen diesen zwei Grafiken besteht ein direkter Zusammenhang. Eine stark ansteigende `setBalance()`-Funktion führt unweigerlich zu einer stark ansteigenden Funktion der Durchführungszeit. Da die `setBalance()`-Aufrufe mehr Zeit benötigen, wird der Testcase auch mehr Zeit in Anspruch nehmen.

Interpretation

Die `setBalance()`-Werte des System ohne Cache sind sehr stabil. Obwohl die `getBalance()`-Aufrufe gewohnt um einiges langsamer waren, als beim System mit Cache, konnte das System ohne Cache schlussendlich durch die stabilen Werte der `setBalance()`-Aufrufe die besseren Ergebnisse liefern.

Die `setBalance()`-Werte des Systems mit Cache skalieren schlecht. Der Grund dafür liegt in der Tatsache, dass das System mit Cache bei jedem `setBalance()`-Aufruf, Updates an die anderen Clients senden muss. Je mehr Clients involviert sind, umso mehr Updates müssen versendet werden, was die starke Steigung der Kurve erklärt.

Das Anzahl der Konflikte beim System mit Cache weisen an sich einen besseren Verlauf, als die Konfliktanzahl des Systems ohne Cache, aus. Da aber die `setBalance()`-Aufrufe des Systems mit Cache viel schlechter skalierten,

als die Werte des Systems ohne Cache, blieb der Performancevorteil aus. Insgesamt skaliert bei diesem Szenario das System ohne Cache um einiges besser.

Bei beiden Systemen treten sehr viele Konflikte auf, vor allem natürlich bei acht Clients. Bei einem solchen Einsatzszenario scheinen beide Systeme nicht ideal zu sein, da die Konfliktbewältigung viel Performance benötigt. Bei sehr vielen Schreibzugriffen, macht der Cache aber bei einer steigenden Anzahl Clients immer weniger Sinn, da die Geschwindigkeit der `setBalance()`-Methode stetig abnehmen wird.

5.4 Messerfahrungen

Die Auswertung der Messdaten und deren Interpretation nahm sehr viel Zeit in Anspruch. Wie bereits im Kapitel 4.3.3 "Report Generator" erläutert, hätte das Speichern der Resultate in einer anderen Art als Textdateien sehr viel weniger Zeit gekostet. Da die Resultate pro Szenario in einer einzelnen Textdatei vorhanden waren, mussten zusätzlich ein kleines Script geschrieben werden, welches aus jedem Szenario die Zusammenfassungen liest und diese in einer gemeinsamen Datei abspeichert. Jedoch war der Aufwand für das Zusammenführen der Daten trotzdem extrem hoch, da die Daten manuell aus der Zusammenfassung nach Excel exportiert werden mussten.

Start Script Zu Beginn der Messphase nutzte man das bereits vorhandene Start-Script für die einzelnen TestCases. Schon nach kurzer Zeit musste das Script jedoch so angepasst werden, dass es selbstständig alle Tests für einen Testcase ausführt. Das heisst, es skaliert den Test von zwei bis acht Clients selbstständig und führt dabei jeweils dreimal denselben Test durch, um einen verlässlichen Mittelwert pro Anzahl Clients ermitteln zu können.

Read Aktion Während des ersten Testlaufs des Szenarios, bei welchem ein Client schreibt und die restlichen nur lesen, stellte sich ein Problem mit dem Szenario Aufbau heraus. Die Idee dieses Szenarios liegt darin, dass die lesenden Clients solange aktiv gehalten werden, wie der schreibende Client Zeit benötigt für die Abarbeitung seines Szenarios. Das Problem war nun, dass die Clients bei der Cache Variante viel früher fertig waren und terminierten, dadurch konnte die gewünschte Interaktion zwischen den Clients nicht aufgezeigt werden. So blieb nur die Möglichkeit den Testcase anzupassen.

1. In einer ersten Variante wurde die Anzahl der Reads von 1'000 auf 1'000'000 vergrössert. Das Problem mit dieser Lösung war, dass das

Szenario Objekt, welches auf dem Server für die einzelnen Clients erzeugt wurde, zu gross wurde und zu einem Heap Space Fehler führte. Eine Erhöhung des JVM Heap Space hätte das Problem ebenfalls nicht gelöst. Die Auswertung eines Clients hätte eine Datei mit über 1'000'000 Zeilen erzeugt. Diese Datei wäre enorm gross geworden und hätte die manuelle Auswertung extrem erschwert.

2. Die Erkenntnis aus dem ersten Versuch führt dazu, dass die Read Aktion modifiziert werden musste. Wie bereits in der Increment Aktion wurde auch in der Read Aktion ein Delay eingebaut, der dabei helfen soll, die Dauer eines Szenarios vorhersagbar zu machen und das Szenario Objekt nicht zu gross werden lässt.

Logger Bereits zu Beginn der Messungen kamen erste Zweifel hoch, über den Speicherplatzverbrauch der einzelne Testcases. Wie sich später herausstellte, lag der Grund für den grossen Speicherplatzverbrauch in der extrem grossen Logdatei, die vom Testframework Server generiert wurde. Die Datei wurde teilweise so gross, dass das Betriebssystem keinen Platz mehr für sich selber besass und den Testcase einfach beendet hat. Die Lösung lag in der Deaktivierung des Loggers in der Methode `methodCalled()`. Was jedoch zur Folge hatte, das alle Messungen nochmals gemacht werden mussten, um eine Verfälschung der Daten auszuschliessen.

Kapitel 6

Ausblick

Die Concurrency control und das Object-Caching könnten noch optimiert und erweitert werden.

6.1 Feingranulares Locking

Bei den realisierten Prototypen werden Lese- und Schreibzugriffe auf dem Server seriel ausgeführt. Der Ablauf des Methodenaufrufs inklusive der Aktualisierung der Concurrency control wird ohne die Unterbrechung durch einen anderen Remote-Methodenaufruf ausgeführt. Damit sollen Inkonsistenzen vermieden werden. Die betreffenden Methoden werden zurzeit mit `synchronized` markiert. Dies kann sich negativ auf die Performance auswirken.

Der Concurrency control Mechanismus könnte so erweitert werden, dass Methodenaufrufe auf dem Server wenn möglich parallel ablaufen. Der Concurrency-Mechanismus soll es ermöglichen, dass Anfragen parallel ausgeführt werden können.

6.2 Local-Write Protokoll

Unser System realisiert ein Remote-Write-Protocol. Alle Schreiboperationen werden an einen einzigen fixen Server weitergeleitet. Eine Variante des primary-based Protokolls wäre, dass das Referenzobjekt (primary) zwischen unterschiedlichen Prozessen (oder Clients und Server) hin und her wandern kann. Dies nennt man ein Local-Write Protokoll [4]. Wenn ein Client ein Objekt updaten möchte, lokalisiert er es und holt das Objekt in seinen Cache. Schreibzugriffe könnten somit lokal ausgeführt werden und es wäre kein zentraler Server mehr nötig. Nachdem ein Objekt beim Client lokal verändert wurde, werden Updatenachrichten an alle anderen Kopien gesendet.

6.3 Generierung von nativen Java Objekten

Die Stub- und die Skeletonklassen wurden in beiden implementierten Prototypen von Hand programmiert. Das heisst, für jeden zusätzlichen Typ zu `Account` muss man die zugehörigen Stub und Skeletonklassen schreiben. Diesen Vorgang könnte man automatisieren, indem man einen Compiler schreibt der Stub- und Skeletonclass-Files generiert (ähnlich wie der Java RMI Compiler).

Objekttypen könnten zum Client mittels eines speziellen Class-Loaders übertragen werden. Damit wäre es nicht mehr nötig, dass der Client bereits mit den Objekttypen ausgeliefert wird. Man könnte neue Objekttypen zur Laufzeit hinzufügen und an die Clients senden.

6.4 Speichern der Messdaten in eine Datenbank

Anstelle der momentanen Ausgabe der Messergebnisse in Textdateien könnte man eine Datenbankanbindung bauen. Das würde bedeuten, dass nach einem erfolgreichen Testlauf der Frameworkserver die zurückgegebenen Daten in einer Datenbank ablegt. Dies würde die Auswertung der Messdaten extrem vereinfachen und somit effizienter machen.

Anhang A

Anhang

A.1 Messresultate

Die folgenden sechs Messtabellen beinhalten die Mittelwerte aus den Messungen und sind in Millisekunden angegeben.

Tabelle A.1: Reads with RMIOnly

	Client 1	Client 2	Client 3	Client 4	Client 5	Client 6	Client 7	Client 8
setBalance								
getBalance	86.1303998	86.23508755						
Number of Conflict	0	0						
Total Time(with Delays)	21533.43944	21559.64155						
Pure Operation Time	21532.59994	21558.77189						
setBalance								
getBalance	89.20024015	89.91072374	89.39585569					
Number of Conflict	0	0	0					
Total Time(with Delays)	22300.94111	22478.61909	22349.92188					
Pure Operation Time	22300.06004	22477.68094	22348.96392					
setBalance								
getBalance	91.09778361	91.56437592	90.91916236	91.67090159				
Number of Conflict	0	0	0	0				
Total Time(with Delays)	22775.42651	22892.15533	22730.66879	22918.59007				
Pure Operation Time	22774.4459	22891.09398	22729.79059	22917.7254				
setBalance								
getBalance	94.17678573	94.44776867	94.61011508	94.36004445	94.86027485			
Number of Conflict	0	0	0	0	0			
Total Time(with Delays)	23545.16958	23612.87787	23653.47607	23590.85772	23716.00718			
Pure Operation Time	23544.19643	23611.94217	23652.52877	23590.01111	23715.06871			
setBalance								
getBalance	97.85615092	99.45811367	98.57316939	97.85162729	98.60172415	97.9483634		
Number of Conflict	0	0	0	0	0	0		
Total Time(with Delays)	24464.90431	24865.56375	24644.25684	24463.88173	24651.39571	24487.98169		
Pure Operation Time	24464.03773	24864.52842	24643.29235	24462.90683	24650.43104	24487.09085		
setBalance								
getBalance	105.9184333	106.3717696	105.0343079	106.1758691	104.5564049	104.2853052	103.6930454	
Number of Conflict	0	0	0	0	0	0	0	
Total Time(with Delays)	26480.56111	26593.91744	26259.64685	26544.88835	26140.02238	26072.33423	25924.14245	
Pure Operation Time	26479.60833	26592.9424	26258.57698	26543.96727	26139.10123	26071.3263	25923.26134	
setBalance								
getBalance	119.6124322	119.1203406	121.1468855	119.674115	119.9025924	120.8797197	121.1135701	120.9084593
Number of Conflict	0	0	0	0	0	0	0	0
Total Time(with Delays)	29903.97718	29781.14986	30287.77163	29919.55143	29976.71233	30220.764	30279.35228	30228.07249
Pure Operation Time	29903.10804	29780.08516	30286.72138	29918.52874	29975.64811	30219.92991	30278.39254	30227.11482

Tabelle A.2: Reads with Cache

	Client 1	Client 2	Client 3	Client 4	Client 5	Client 6	Client 7	Client 8
setBalance								
getBalance	0.297583161	0.280414924						
Number of Conflict	0	0						
Total Time(with Delays)	75.21408133	70.96021966						
Pure Operation Time	74.39579033	70.103731						
setBalance								
getBalance	0.283629707	0.295661851	0.282970525					
Number of Conflict	0	0	0					
Total Time(with Delays)	71.61007233	74.71670733	71.418955					
Pure Operation Time	70.90742667	73.91546467	70.74263133					
setBalance								
getBalance	0.296050801	0.280570846	0.281436288	0.282813488				
Number of Conflict	0	0	0	0				
Total Time(with Delays)	74.85035867	70.80903467	71.09234233	71.40615333				
Pure Operation Time	74.01270033	70.14271167	70.359072	70.703372				
setBalance								
getBalance	0.285663225	0.312828462	0.288400196	0.295459635	0.309716468			
Number of Conflict	0	0	0	0	0			
Total Time(with Delays)	72.19344	79.04818833	72.886582	74.65476567	78.331988			
Pure Operation Time	71.41580633	78.20711533	72.100049	73.86490866	77.42911697			
setBalance								
getBalance	0.285606248	0.310732133	0.290229814	0.318903688	0.293176263	0.308295527		
Number of Conflict	0	0	0	0	0	0		
Total Time(with Delays)	72.10436927	78.42981466	73.35994733	80.36126267	74.03508633	77.79525333		
Pure Operation Time	71.40156097	77.68303333	72.557451	79.72592199	73.29406567	77.07388167		
setBalance								
getBalance	0.324125513	0.298296187	0.338366839	0.325962512	0.339631495	0.300458849	0.314308424	
Number of Conflict	0	0	0	0	0	0	0	
Total Time(with Delays)	81.80961433	75.30799067	85.383507	82.32748533	85.690358	75.92357366	79.43009767	
Pure Operation Time	81.031379	74.57404667	84.59170967	81.49062799	84.90787367	75.11470767	78.577106	
setBalance								
getBalance	0.28349158	0.294649181	0.302092575	0.245091289	0.280779916	0.290295597	0.239235395	0.312778815
Number of Conflict	0	0	0	0	0	0	0	0
Total Time(with Delays)	71.564313	74.42772367	76.25655467	61.95237633	70.895478	73.39102067	60.53241633	78.99331833
Pure Operation Time	70.872895	73.66229533	75.52314367	61.2728223	70.194979	72.57389933	59.80884867	78.19470363

Tabelle A.3: Increments and Reads with RMIOnly

	Client 1	Client 2	Client 3	Client 4	Client 5	Client 6	Client 7	Client 8
setBalance	88.75211012							
getBalance	0.18371334	0.152781074						
Number of Conflict	3	0						
Total Time(with Delays)	80144.64256	200693.218						
Pure Operation Time	80131.17694	305.5621483						
setBalance	88.78985339							
getBalance	0.184562508	0.156996487	0.134534757					
Number of Conflict	2	0	0					
Total Time(with Delays)	80149.51928	200674.4439	301065.2484					
Pure Operation Time	80136.18861	313.992973	403.604272					
setBalance	88.60361699							
getBalance	0.185507381	0.158436158	0.142924763	0.130708076				
Number of Conflict	2	0	0	0				
Total Time(with Delays)	79982.69419	200676.1169	300963.2363	401400.282				
Pure Operation Time	79969.438	316.872315	428.7742883	522.8323047				
setBalance	88.62054979							
getBalance	0.1842018	0.159435489	0.139849256	0.132829466	0.11501215			
Number of Conflict	3	0	0	0	0			
Total Time(with Delays)	80025.97315	200690.0179	300966.4152	401324.8629	601792.8136			
Pure Operation Time	80012.82228	318.871014	419.547768	531.3178637	690.0728973			
setBalance	88.65589752							
getBalance	0.189578621	0.158217769	0.140795985	0.135737402	0.115720343	0.114206524		
Number of Conflict	1	0	0	0	0	0		
Total Time(with Delays)	80003.75672	200685.4107	300972.399	401341.5356	601793.7972	601951.1304		
Pure Operation Time	79990.54276	316.4355387	422.3879557	542.9496067	694.322057	685.2391453		
setBalance	88.76628302							
getBalance	0.180374119	0.16237285	0.139947829	0.133043553	0.119098594	0.118055838	0.111005254	
Number of Conflict	1	0	0	0	0	0	0	
Total Time(with Delays)	80093.18019	200695.6241	300969.906	401338.9308	601805.658	601939.478	601835.6149	
Pure Operation Time	80081.59392	324.745699	419.843492	532.1742117	714.591563	708.3350293	666.031523	
setBalance	88.92028329							
getBalance	0.19257422	0.159252102	0.143602412	0.133032453	0.116684273	0.118941429	0.11796638	0.115468577
Number of Conflict	1	0	0	0	0	0	0	0
Total Time(with Delays)	80244.85721	200686.1422	300966.3739	401326.0257	601777.6492	601903.4693	601908.981	601923.4471
Pure Operation Time	80231.3192	318.5042047	430.8072363	532.1298147	700.1056383	713.648576	707.7982773	692.8114623

Tabelle A.4: Increments and Reads with Cache

	Client 1	Client 2	Client 3	Client 4	Client 5	Client 6	Client 7	Client 8
setBalance	88.75211012							
getBalance	0.18371334	0.152781074						
Number of Conflict	3	0						
Total Time(with Delays)	80144.64256	200693.218						
Pure Operation Time	80131.17694	305.5621483						
setBalance	88.78985339							
getBalance	0.184562508	0.156996487	0.134534757					
Number of Conflict	2	0	0					
Total Time(with Delays)	80149.51928	200674.4439	301065.2484					
Pure Operation Time	80136.18861	313.992973	403.604272					
setBalance	88.60361699							
getBalance	0.185507381	0.158436158	0.142924763	0.130708076				
Number of Conflict	2	0	0	0				
Total Time(with Delays)	79982.69419	200676.1169	300963.2363	401400.282				
Pure Operation Time	79969.438	316.872315	428.7742883	522.8323047				
setBalance	88.62054979							
getBalance	0.1842018	0.159435489	0.139849256	0.132829466	0.11501215			
Number of Conflict	3	0	0	0	0			
Total Time(with Delays)	80025.97315	200690.0179	300966.4152	401324.8629	601792.8136			
Pure Operation Time	80012.82228	318.871014	419.547768	531.3178637	690.0728973			
setBalance	88.65589752							
getBalance	0.189578621	0.158217769	0.140795985	0.135737402	0.115720343	0.114206524		
Number of Conflict	1	0	0	0	0	0		
Total Time(with Delays)	80003.75672	200685.4107	300972.399	401341.5356	601793.7972	601951.1304		
Pure Operation Time	79990.54276	316.4355387	422.3879557	542.9496067	694.322057	685.2391453		
setBalance	88.76628302							
getBalance	0.180374119	0.16237285	0.139947829	0.133043553	0.119098594	0.118055838	0.111005254	
Number of Conflict	1	0	0	0	0	0	0	
Total Time(with Delays)	80093.18019	200695.6241	300969.906	401338.9308	601805.658	601939.478	601835.6149	
Pure Operation Time	80081.59392	324.745699	419.843492	532.1742117	714.591563	708.3350293	666.031523	
setBalance	88.92028329							
getBalance	0.19257422	0.159252102	0.143602412	0.133032453	0.116684273	0.118941429	0.11796638	0.115468577
Number of Conflict	1	0	0	0	0	0	0	0
Total Time(with Delays)	80244.85721	200686.1422	300966.3739	401326.0257	601777.6492	601903.4693	601908.981	601923.4471
Pure Operation Time	80231.3192	318.5042047	430.8072363	532.1298147	700.1056383	713.648576	707.7982773	692.8114623

Tabelle A.5: Increment with RMI

	Client 1	Client 1	Client 3	Client 4	Client 5	Client 6	Client 7	Client 8
setBalance	79.88494732	80.35463039						
getBalance	80.09688983	79.65769827						
Number of Conflict	0	1000						
Total Time(with Delays)	159993.6124	320045.1244						
Pure Operation Time	159981.8372	320024.6573						
setBalance	80.2496546	80.2415055	80.24414323					
getBalance	79.7797009	79.76839771	79.77484438					
Number of Conflict	1262.666667	1216.333333	1146.333333					
Total Time(with Delays)	362112.424	354683.2319	343473.3778					
Pure Operation Time	362089.3418	354660.138	343451.8226					
setBalance	80.15141703	80.46003281	80.27991848	80.31596343				
getBalance	79.87746297	79.61325694	79.7735209	79.75514354				
Number of Conflict	618	2620.666667	1155	2251.666667				
Total Time(with Delays)	217479.5386	579595.5407	344942.1162	520539.9548				
Pure Operation Time	258947.8157	579563.07	344920.0688	520510.012				
setBalance	80.35449155	80.43233567	80.18630604	80.35847868	80.37809671			
getBalance	79.74390019	79.7056632	79.92048743	79.75720459	79.73543926			
Number of Conflict	2925	2320.666667	966.6666667	2627.333333	2995.666667			
Total Time(with Delays)	628433.7078	531754.3528	315022.7632	580796.0703	639753.6832			
Pure Operation Time	628398.2989	531724.7546	315002.8107	580762.8581	639717.3371			
setBalance	80.2308993	80.49502893	80.33487974	80.15514838	80.3146161	80.32226677		
getBalance	79.77223766	79.52432042	79.68407984	79.84056625	79.69028765	79.69306084		
Number of Conflict	2322	4615.333333	3579.333333	1473	1828	3596		
Total Time(with Delays)	531599.4731	898606.1884	732819.1373	395716.6914	452525.3781	735473.1515		
Pure Operation Time	531569.8547	898561.049	732779.4173	395692.8489	452497.8599	735434.0697		
setBalance	80.1676569	80.47851429	80.16980256	80.16244265	80.32417436	80.31336876	80.31803659	
getBalance	79.82961223	79.54110694	79.82498685	79.83344453	79.69143681	79.70517789	79.70481282	
Number of Conflict	2542.666667	4158	2771.333333	2218	2804	2483	5723.333333	
Total Time(with Delays)	566899.6924	825427.3086	603497.6909	514952.1536	608746.3636	557376.0229	1074335.939	
Pure Operation Time	566868.9213	825383.8342	603464.6678	514924.2385	608712.6705	557344.7251	1074285.839	
setBalance	80.27192817	80.28294552	80.20315236	80.32812347	80.36069126	80.3363715	80.32392308	80.35118848
getBalance	79.73543396	79.72272658	79.80845011	79.69464028	79.66410629	79.68747732	79.70162231	79.67304743
Number of Conflict	2389	4528.333333	2897.333333	4891.666667	4422.666667	5119.333333	6308.333333	4416.666667
Total Time(with Delays)	542332.0046	884686.5892	623696.8012	942851.4246	867804.1285	979293.5347	1169574.58	866850.9584
Pure Operation Time	542301.1116	884644.2508	623663.0258	942805.5733	867760.5418	979246.9224	1169519.958	866806.6294

Tabelle A.6: Increments with Cache

	Client 1	Client 2	Client 3	Client 4	Client 5	Client 6	Client 7	Client 8
setBalance	102.7806239	105.7899225						
getBalance	0.154795163	0.114516045						
Number of Conflict	364	999						
Total Time(with Delays)	140419.2351	211722.047						
Pure Operation Time	140401.9624	211702.6138						
setBalance	131.9304088	130.2387696	130.7180326					
getBalance	0.102254868	0.094740155	0.105231737					
Number of Conflict	1490.333333	1698.666667	1407.666667					
Total Time(with Delays)	328837.4258	351735.1225	315098.2257					
Pure Operation Time	328815.4064	351712.9528	315076.2522					
setBalance	245.4813981	246.002005	249.4989884	248.1935531				
getBalance	0.104287108	0.099689005	0.09674781	0.105058824				
Number of Conflict	1901.666667	1983.666667	1908.333333	1872.333333				
Total Time(with Delays)	712757.9397	734225.5366	725860.466	713189.7386				
Pure Operation Time	712731.402	734199.7522	725836.5127	713163.6379				
setBalance	422.6858988	431.1307779	424.3506417	422.8843742	434.0714585			
getBalance	0.100397232	0.102601448	0.099876919	0.099314214	0.094604726			
Number of Conflict	2233.666667	2282	2210.666667	2243.666667	2249			
Total Time(with Delays)	1368430.447	1415436.114	1362420.349	1372154.218	1410584.019			
Pure Operation Time	1368402.355	1415405.648	1362391.825	1372125.349	1410557.726			
setBalance	633.6899419	632.850428	651.8117506	661.8941898	650.8977784	659.5116546		
getBalance	0.096611878	0.103654481	0.099793234	0.110117902	0.094176604	0.100688455		
Number of Conflict	2439.666667	2528.333333	2570.666667	2539.333333	2539.333333	2589.666667		
Total Time(with Delays)	2180750.589	2240554.915	2328286.39	2343802.974	2357282.21	2368125.679		
Pure Operation Time	2180722.804	2240523.318	2328255.055	2343768.19	2357252.558	2368094.014		
setBalance	996.0017939	979.1018702	984.1872579	970.6266778	986.8964467	972.8950982	979.0706172	
getBalance	0.08713013	0.102047511	0.097543514	0.097680819	0.101237145	0.094529867	0.100549815	
Number of Conflict	2887.333333	2864	2912	2854	2912.666667	2748.666667	2920.666667	
Total Time(with Delays)	3745292.521	3784832.939	3852360.971	3741589.448	3862598.573	3647216.497	3841362.983	
Pure Operation Time	3874277.613	3784797.237	3852325.359	3741554.513	3862561.57	3647183.84	3841326.585	
setBalance	1371.737057	1336.24019	1341.227634	1390.588066	1315.429351	1358.513198	1375.528872	1338.64465
getBalance	0.096206615	0.088073456	0.086994473	0.091539497	0.098812224	0.083976054	0.090198492	0.090338685
Number of Conflict	3264.333333	3201	3246.666667	3277.333333	3137.333333	3381	3312.666667	3217.333333
Total Time(with Delays)	5849894.483	5620727.755	5699669.104	5947747.687	5443141.355	5952080.468	5932280.277	5648183.283
Pure Operation Time	5849856.677	5620695.143	5699636.544	5947711.926	5443103.127	5952048.18	5932244.797	5648148.292

A.2 Inhalt CD

Ordner doc/projektmanagement Enthält folgende Dokumente

- Anforderungsspezifikation.pdf
- Projektplan.pdf
- Sitzungsprotokolle.pdf
- PersönlicheBerichte.pdf
- Aufgabenstellung.pdf

doc/bericht Enthält den Bericht der Arbeit sowie das Poster und den Kurzbericht dazu.

- BerichtObjectCaching.pdf
- PosterObjectCaching.pdf
- AbstractObectCaching.pdf

src Enthält den Quellcode der Prototypen und des Frameworks sowie den Quellcode der Dokumentation in Latex.

Glossary

Account Die in diesem Dokument vorgestellten Konzepte werden alle anhand eines simplen Bankkonto-Objektes beschrieben. Das Bankkonto verfügt über einen Saldo, eine Methode um darauf zu schreiben und eine zweite um den Saldo auszulesen. . 3

ARP Das Address Resolution Protocol (ARP) ist ein Netzwerkprotokoll, das zu einer Netzwerkadresse der Internetschicht die physikalische Adresse (Hardwareadresse) der Netzzugangsschicht ermittelt und diese Zuordnung gegebenenfalls in den so genannten ARP-Tabellen der beteiligten Rechner hinterlegt.. 44

Concurrency Control Behandelt die korrekte Planung und Durchführung von miteinander in Konflikt stehenden Operationen.. 3

Hashmap In der Informatik bezeichnet man eine spezielle Indexstruktur als Hashtabelle bzw. Streuwerttabelle. Als Indexstruktur werden Hashtabellen verwendet um Datenelemente in einer großen Datenmenge aufzufinden.. 6

Interface Eine Schnittstelle (englisch interface) dient in der objektorientierten Programmierung der Vereinbarung gemeinsamer Signaturen von Methoden, die in unterschiedlichen Klassen implementiert werden. Die Schnittstelle gibt dabei an, welche Methoden vorhanden sind oder vorhanden sein müssen.. 3

Konsistenz Als Konsistenz bezeichnet man die Korrektheit der gespeicherten Daten nach einem schreibenden Zugriff.. 4

Lost Update Verlorenes Update bezeichnet in der Informatik einen Fehler, der bei mehreren parallelen Schreibzugriffen auf eine gemeinsam genutzte Information auftreten kann.. 3

Middleware Middleware oder Vermittlungssoftware bezeichnet in der Informatik anwendungsneutrale Programme, die so zwischen Anwendungen vermitteln, dass die Komplexität dieser Applikationen und ihre Infrastruktur verborgen werden.. 4

Object-Caching Beschreibt das Ablegen von Objekten im lokalen Speicher.. 3

Optimistic Concurrency Bei diesem Verfahren "hofft" man, dass keine Konflikte auftreten. Man ist optimistisch.. 4

Request-Reply Protocol Ein Protokoll, welches meist nur ein Datenpacket sendet und dann mit dem Senden weiterer Packete wartet, bis es eine Antwort erhalten hat.. 4

RMI RMI heisst "Remote Method Invocationünd ermöglicht das Aufrufen von Methoden auf anderen Rechnern.. 3

Serialisierte Serialisierung wird hauptsächlich für die Persistierung von Objekten in Dateien und für die Übertragung von Objekten über das Netzwerk bei Verteilten Softwaresystemen verwendet.. 5

STP Das Spanning Tree Protocol (STP) baut einen Spannbaum zur Vermeidung von Schleifen in redundanten Netzpfaden im LAN, speziell in geswitchten Umgebungen, auf.. 44

TCP Das Transmission Control Protocol (TCP) ist eine Vereinbarung darüber, auf welche Art und Weise Daten zwischen Computern ausgetauscht werden sollen.. 5

Virtual Machine Die Java Virtual Machine ist der Teil der Java-Laufzeitumgebung (JRE) für Java-Programme, der für die Ausführung des Java-Bytecodes verantwortlich ist.. 4

Abbildungsverzeichnis

Literaturverzeichnis

[1] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, 1988.

[2] Todd Greanier. Discover the secrets of the java serialization api. `http://java.sun.com/developer/technicalArticles/Programming/serialization/`, 2000. [Online; accessed 17-May-2012].

[3] Oracle. An overview of rmi applications. `http://docs.oracle.com/javase/tutorial/rmi/overview.html`, 2012. [Online; accessed 03-April-2012].

[4] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms*. Pearson Prentice Hall, 2007.

[5] Wikipedia. Optmistic concurrency — Wikipedia, the free encyclopedia. `http://de.wikipedia.org/wiki/Optimistic_Concurrency`, 2012. [Online; accessed 10-May-2012].