Technische Hochschule Nürnberg

Projektarbeit

Robuste Software für parallele Computerarchitekturen

Dozent: Prof. Dr. Axel Hein

Banking Server

vorgelegt von	Christopher Althaus
	Markus Endres
	Alexander Baumgärtner
	Christoph Bohner
	Ulrich Hecht
Erstgutachter	Prof. Dr. Axel Hein
Abgabe:	27. Mai 2013

Erklärung

Hiermit versichern wir, dass wir die Arbeit selbständig verfasst, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet zu haben.

_____

Markus Endres

_____

Alexander Baumgärtner

_____

Christoph Bohner

_____

Ulrich Hecht

_____

Christopher Althaus

Erlangen, den 3. Juli 2013

Inhaltsverzeichnis

1 Vorwort

In diesem Dokument wird die Projektarbeit mit dem Thema Banking-Server dokumentiert. Ziel war es, einen Banking-Server mit allen gängigen Funktionen in der Programmiersprache Erlang zu implementieren. Grundlage war das *Behaviour-Modul* **gen_server**, welches hauptsächlich der Kommunikation diente. Die Bearbeitung der Aktionen und Transaktionen sollte in sogenannte *Workerprozesse* ausgelagert werden. Zudem sollte es möglich sein, mit mehreren Clients gleichzeitig auf den Server zu agieren. Eine persistente Datenspeicherung, sowie eine Fehlerbehandlung bei Ausfall eines *Workerprozesses* waren ebenso Voraussetzung, wie das Testen der Software mittels der Erlang-Module **fprof** und **cprof**. Das Endprodukt sollte die üblichen Vorkommnisse im bargeldlosen Zahlungsverkehr darstellen können, der Sicherheitsaspekt wurde bei der Bearbeitung des Projekts jedoch nicht berücksichtigt.

2 Teamorganisation

Die Projektgruppe A bestand aus Markus Endres, Christoph Bohner, Ulrich Hecht, Christopher Althaus sowie Alexander Baumgärtner. Innerhalb der Gruppe wurde festgelegt, dass eine hierarchiefreie Organisation herrschen soll. Dies bedeutete auch, dass alle Teammitglieder gleichermaßen Aufgaben zu bearbeiten hatten.

Für die erste Präsentation wurden die Themen verteilt, sodass jeder einen Aufgabenteil zu bearbeiten hatte. Da sich diese Art der Organisation bewährte, behielten wir diesen Weg auch zur Erstellung dieser Arbeit bei. Ulrich Hecht und Christopher Althaus wurde die Ausarbeitung des Clients, sowie die Erstellung des Servers übertragen. Markus Endres und Alexander Baumgärtner befassten sich mit der Programmierung des Workers. Die Störungsfunktion und die Erstellung der Abschlusspräsentation fielen in den Aufgabenbereich von Christoph Bohner und Markus Endres. Die Live-Demo wurde von Christopher Althaus und Ulrich Hecht vorbereitet, wobei Letzterer sich zudem mit dem Profiling auseinandersetzte. Aufgrund einer unbekannten Programmierumgebung waren das finale Design und die Schnittstellen nicht von Anfang an abzusehen. Somit waren über den gesamten Zeitraum der Bearbeitung eine enge Zusammenarbeit und gute Kommunikation Grundlage für ein erfolgreiches Umsetzen des Projekts.

3 Konzept

Im Folgendem wird das Gesamtkonzept des Bankingservers beschrieben. Dies geschieht verteilt anhand von mehreren Sequenzdiagrammen.

3.1 Grundkonzept

Den Anfang macht hier das Diagramm zum Grundkonzept des Bankingservers.

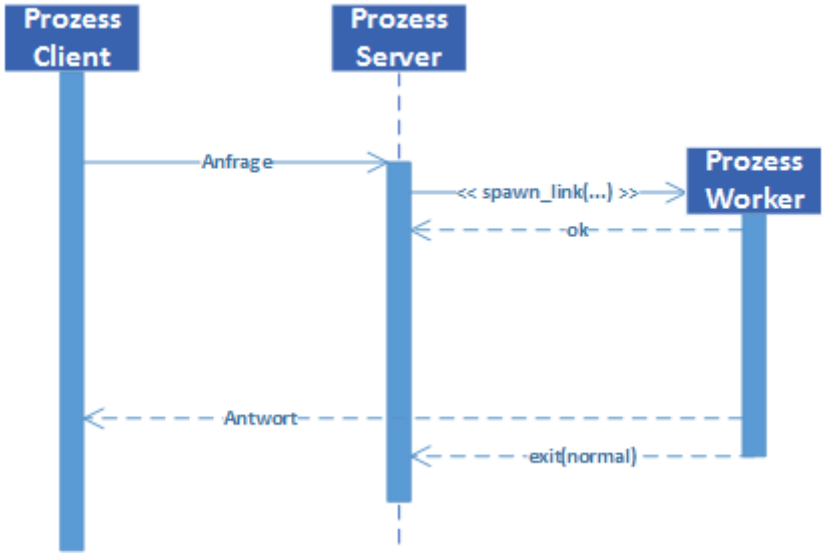

Abbildung 1: Das Grundkonzept des kompletten Bankingservers

Der Startzustand besteht darin, dass Client so wie Server gestartet sind. Der Server ist kurz darauf in seinem Wartezustand, in dem er auf eintreffende Nachrichten wartet.
Sobald der Client eine Anfrage an den Server richtet, wie z.B. "Konto erstellen", startet dieser einen Arbeitsprozess. Dieser neue Arbeitsprozess, oder auch *Worker*, erhält während seines Aufrufs alle notwendigen Daten, die er zur Erfüllung seiner Aufgabe benötigt. Dazu zählt auch die Prozess-ID des Clients.
Der Server erhält nach dem erfolgreichen Start des Workers ein "ok" zurück.
Der arbeitende Prozess ist nun damit beschäftigt die Aufgabe des Clients zu bearbeiten. So lange dies geschieht, bleibt der Server weiterhin erreichbar für neue Anfragen (dazu später mehr).
Sobald der Worker fertig mit seiner Arbeit ist, sendet er eine Antwort, welche eine Erfolgsmeldung und mögliche Kontodaten enthält, an den Client. Dieser kann die Nachrichten wie gewohnt über den `receive`-Befehl auslesen.
Der Server hingegen erhält bei der Beendigung des Workers ein normales "exit", welches signalisiert, dass der arbeitende Prozess sich ordnungsmäßig beendet und somit seine Aufgabe erfüllt hat.

3.2 Multiclient-Konzept

Die Anfragen, welche an dem Server gestellt werden, sollen möglichst schnell abgearbeitet werden. Dazu ist es erforderlich das Abarbeiten dieser Anfragen zu parallelisieren.

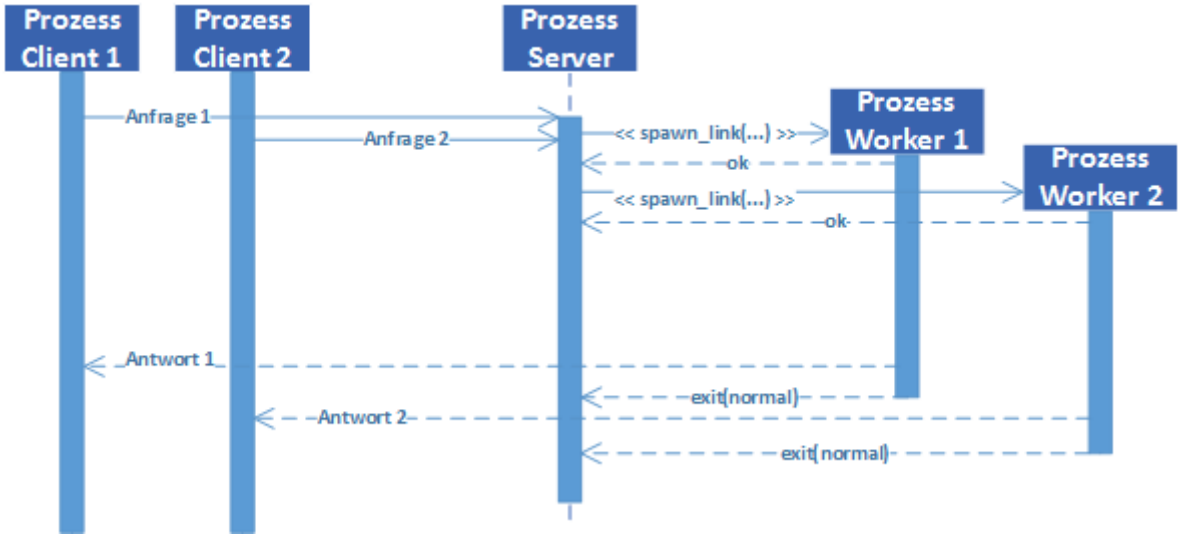

Abbildung 2: Bankingserver mit mehreren parallelen Clients

In diesem Sequenzdiagramm befinden sich zwei Clients, welche fast zeitgleich jeweils eine Anfrage an den Server richten.
Der Server bekommt zuerst die "Anfrage 1" von "Client 1", welche somit auch als erstes abzuarbeiten ist. Dazu wird, wie schon im Grundkonzept beschrieben, ein Worker (hier "Worker 1") gestartet.
So lange der Server mit der Ausführung von `spawn_link` für "Worker 1" benötigt, wird "Anfrage 2" nicht bearbeitet. Diese Verzögerung ist jedoch minimal, muss allerdings trotzdem in Kauf genommen werden.
Darauf folgend wird "Anfrage 2" so schnell wie möglich bearbeitet. Dies geschieht hier in "Worker 2".
Die Clients erhalten nach der Abarbeitung ihrer entsprechenden Anfragen nach schon bekanntem Verfahren die jeweiligen Antworten von den Workern.

3.3 Störungskonzept

Eine der Aufgaben des Projekts war es, eine Störquelle einzubauen, welche den normalen Betrieb der Worker beeinträchtigen soll. Dieses wird dadurch realisiert, dass ein Störprozess vereinzelt Worker beendet, bevor sie mit ihren Aufgaben fertig sind.

3.3.1 Das Wirken der Störung

Das folgende Diagramm zeigt wie besagter Prozess den ersten laufenden Worker beendet, den zweiten allerdings ohne einzugreifen arbeiten lässt.

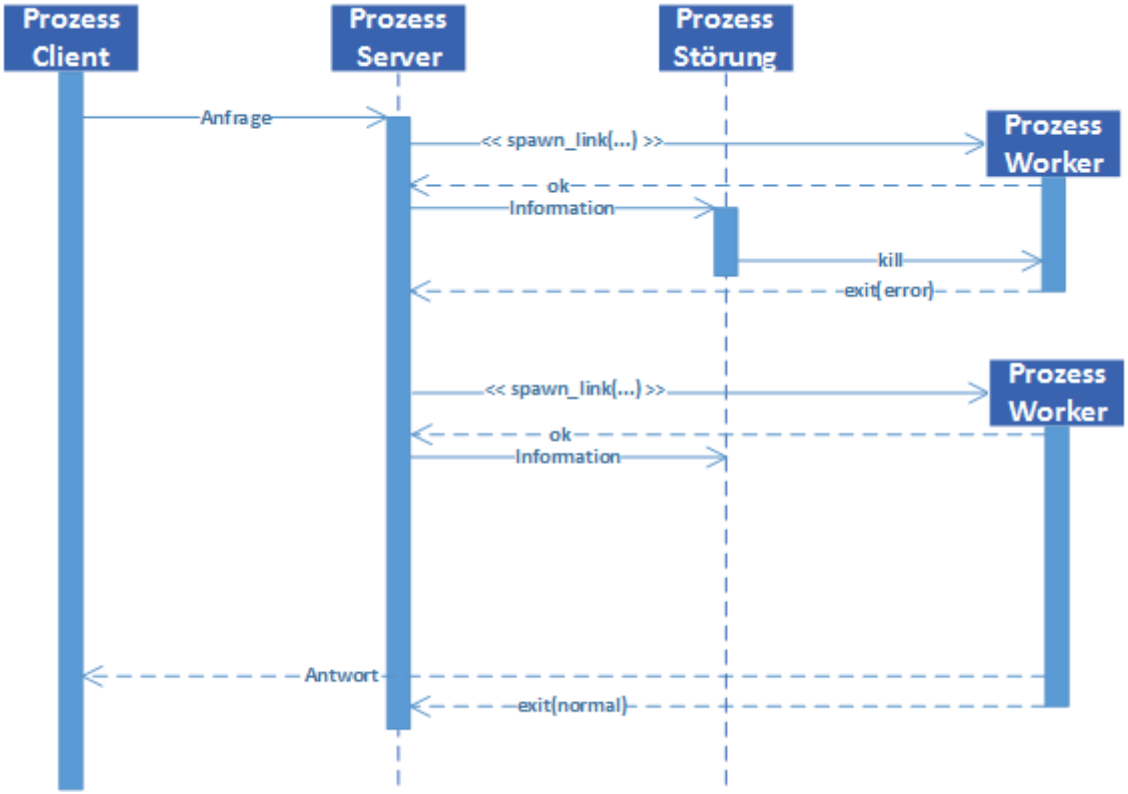

Abbildung 3: Bankingserver mit eingebauter, aktiver Stoerungsquelle

Client und Server verfahren vorerst nach bekanntem Vorgehen. Nachdem allerdings der neue Worker sein "ok" an den Server zurückschickt, informiert der Server die Störquelle über den neuen Arbeitsprozess.
Dieses Verhalten des Servers ist normalerweise nicht in der endgültigen Software enthalten und ist hier nur erforderlich, um Störungen und Fehlverhalten zu simulieren.
Der Störprozess entscheidet nun, dass dieser Worker von ihm beendet wird ("kill"). In Folge dessen erhält der Server, wegen der Verlinkung mit dem Worker, ein "exit(error)". Dies signalisiert dem Bankingserver, dass der Arbeitsprozess sich mit einem Fehler beendete. Wenn der Worker seine Aufgabe noch nicht erledigen konnte, bevor er vom Störprozess beendet wurde, ist die Anfrage des Clients immer noch ausstehend. In diesem Beispiel wird von diesem Fall ausgegangen.
Es ist also erforderlich, dass der gleiche Arbeitsprozess mit den gleichen Parametern erneut gestartet wird.

3.3.2 Das Ausgleichen der Störung

Nachdem die Unterbrechung durch den Störungsprozess verursacht wurde, muss dies erkannt und aufgelöst werden.
Wie anhand von dem Diagramm **??** erklärt wurde, gibt ein von dem Störungsprozess beendeter Worker ein *exit(error)* an den Server zurück. Dieser erkennt diese Meldung und hat nun die Aufgabe, potenziellen Schaden zu erkennen.
Um die Suche nach nicht ausgeführten Anfragen durchzuführen, werden die zugehörigen Transaktionslisten durchsucht.

Transaktionsliste des Servers Die Transaktionsliste des Servers enthält sämtliche Transaktionen, welche derzeit durchgeführt werden. Wurde eine Anfrage erfolgreich bearbeitet, wird der dazugehörige Eintrag dieser Liste gelöscht.
Der Aufbau eines Tupels der Liste ist folgender:

```
%
%{ TransaktionsID, { Aktion, [ ClientPID | Arguments ] ] } }
%
```

- "TransaktionsID": Ist eine eindeutige ID, welche eine Transaktion markiert.

Das darauf folgende Tupel dient dafür, die restlichen Informationen zu speichern, welche unter Umständen dafür benötigt werden, um Worker erneut zu starten.

- "Aktion": Speichert die Aufgabe, welche durchgeführt werden soll.
- "ClientPID": Steht für die Prozess ID des Clients, welcher die Anfrage gestellt hat.
- "Arguments": Der Rest der Liste besteht aus den Argumenten für den Worker, welche die Aufgabe ausführen muss.

Transaktionen in der allgemeine Kontenspeicherung Im *dets*-Modul für die allgemeine Kontenspeicherung wird jede durchgeführte Transaktion bei den einzelnen Konten gespeichert. Die angehängten Transaktionslisten sind folgendermaßen aufgebaut:

```
%
% {TransaktionsID,
% Aktion,
% {zeit, Datum, Uhrzeit},
% {notizen, Notizen},
% {wer, Kontonummer},
% {wert, Betrag}
```

```
8 % }
9 %
```

- "TransaktionsID": Diese ID eindeutig markiert einzelne Transaktionen und stimmt mit der Transaktions-ID der Transaktionsliste des Servers überein.
- "Aktion": Speichert die durchgeführte Aktion.
- Tupel "Zeit": Speichert den Zeitpunkt in dem die Aktion ausgeführt wurde.
- Tupel "Notizen": Speichert zusätzliche Notizen bei einzelnen Transaktionen.
- Tupel "Wer": Speichert die Information, von welchem Konto die Transaktion ausgelöst wurde. Dies ist vor allem bei Überweisungen wichtig.
- Tupel "Betrag": Speichert den Betrag, welcher bei der Transaktion bewegt wurde.

Sobald der Server die Nachricht eines sich fehlerhaft beendeten Workers erhält, überprüft er, ob die Transaktion, welche der Worker auszuführen hatte, erledigt wurde. Dies geschieht dadurch, dass der Server die Transaktions-ID des Workers in der Transaktionsliste des betreffenden Kontos sucht. Kann er diese finden, wurde die Anfrage vor dem Wirken der Störung durchgeführt. Dies bedeutet, dass die Aufgabe des Workers nicht wiederholt werden muss. Wird die entsprechende Transaktions-ID nicht gefunden, wurde die Anfrage noch nicht bearbeitet. Ein gleicher Worker muss gestartet und die Aufgabe wiederholt werden.

4 Bedienungsanleitung

Im folgenden Abschnitt wird das Starten und die Nutzung des Banking-Servers grundsätzlich beschrieben. Dabei ist es wichtig zu wissen, dass der Server auf einem eigenen Knoten läuft. Dadurch wird ermöglicht, dass die Software auch auf einem verteilten System eingesetzt werden kann. Bevor man das Programm ausführen kann, sind die Module **bs**, **bc**, **bw** und **virus** zu kompilieren.

4.1 Das Starten

Der einfachste Weg eine Erlang-Shell einem gewissen Knoten zuzuweisen besteht darin, das Ziel einer Verknüpfung zu modifizieren. Dafür klickt man mit der rechten Maustaste auf die Verknüpfung und wählt den Unterpunkt "Eigenschaften". In dem Tab "Verknüpfungen" ergänzt man unter "Ziel:" die bisherige Eingabe um den Zusatz `erl -sname bs@localhost`. Gesamt sollte das Textfeld in etwa so gefüllt sein: `"C:\Program Files\erl5.10.1\bin\werl.exe" erl -sname bs@localhost`. Die Benennung mit "bs" ist für den Server ausschlaggebend.

Dahingegen kann der Knoten für den Client beliebig genannt werden. Es ist notwendig, eine neue Shell zu starten. Diesmal sollte das Ziel der Verknüpfung um `erl -sname bc@localhost` erweitert werden. Im weiteren wird der Name des Knotens "bc" fest definiert sein. (Gesamteingabe: `"C:\Program Files\erl5.10.1\bin\werl.exe" erl -sname bc@localhost`.) Nun sollten parallel zwei verschiedene Shells laufen. Eine für den Server (Servershell) und eine für den Client (Clientshell). Um den Server zu starten, führt man in der Servershell das Kommando `bs:start().` aus und ein Client kann in der Clientshell durch den Befehl `bc:start(Client1).` angelegt werden. "Client1" ist in diesem Fall ein frei wählbarer Name. Zusätzlich könnten noch weitere Clients angefügt werden.

Ab dem jetzigen Zeitpunkt können in der Clientshell Anfragen an einen ausgewählten Client gesendet werden. Beispielsweise:

```
1 (bc@localhost)1> client1 ! konto_anlegen.
2 konto_anlegen
3 OK: 2
4 (bc@localhost)1> client1 ! {konto_abfragen, 2}.
5 konto_abfragen
6 OK: 0
```

4.2 Das Beenden

Um den Server ordnungsgemäß zu beenden, tätigt man die Eingabe `bs:stop()` in der Servershell. An den Client muss man hingegen die Anfrage `stop` senden und bekommt die Nachricht "Banking Client wurde beendet", sobald der Prozess abgeschlossen ist.

5 Fallstudie

In diesem Abschnitt wird gezeigt, wie ein Einsatz des Banking-Servers in der Realität aussehen könnte. Alle Charaktere sind frei erfunden. Ähnlichkeiten zu lebenden oder verstorbenen Personen wären zufällig und nicht beabsichtigt. Die Fallstudie betrachtet die Nutzung eines Bankkontos über einen Monat hinweg. Dabei werden verschiedene Clients und Konten benutzt. Die Konten wurden bereits erstellt und die Clients wurden gemäß der Bedienungsanleitung gestartet. Hier eine Aufstellung über Personen und zugehörige Kontonummern:

Tabelle 1: Verwendete Personen und Kontonummern

Kontonummer	Person	Rolle
13	Martin Schmidt	Student
14	Alfred Schmidt	Vater
15	OnlineKauf GmbH	Online Shopping Portal
16	Informatik AG	Arbeitgeber

Zu Beginn der Studie ist auf dem Bankkonto von Martin kein Geld, wobei der Kontostand der anderen Personen immer ausreichend groß ist. Am 01.07.13 bekommt Martin seinen Lohn von der Buchhaltungsabteilung seines Arbeitgebers "Informatik AG" überwiesen.

```
(bc@localhost)1> informatikAG_Buchhaltung ! {geld_ueberweisen, 13, 16, 560}.
{geld_ueberweisen,13,16,560}
OK: 'Geld wurde ueberwiesen'
```

Zusätzlich erhält der Student zwei Tage später sein Taschengeld in Form einer Überweisung von seinem Vater.

```
(bc@localhost)2> alfred_onlineBanking ! {geld_ueberweisen, 13, 14, 60}.
{geld_ueberweisen,13,14,60}
OK: 'Geld wurde ueberwiesen'
```

Martin freut sich über das Geld und kauft sich am 05.07. neue Fußballschuhe im Wert von 200 Euro bei dem Internet-Shopping-Portal "OnlineKauf GmbH". Die Zahlung erledigt er noch am gleichen Tag via Online-Banking.

```
(bc@localhost)3> martin_onlineBanking ! {geld_ueberweisen, 15, 13, 200}.
{geld_ueberweisen,15,13,200}
OK: 'Geld wurde ueberwiesen'
```

Nach dem Ende der zweiten Juliwoche ruft Martin seinen Kontostand ab, um den Überblick über seine Finanzen zu bewahren.

```
(bc@localhost)4> martin_onlineBanking ! {kontostand_abfragen, 13}.
{kontostand_abfragen,13}
OK: 420
```

Am 19. des Monats besucht Martin seine Oma, die ihm einen 50-Euroschein zusteckt. Diesen will er gleich auf sein Konto einzahlen und benutzt dazu die örtliche Bankfiliale.

```
(bc@localhost)5> filialeOberndorf ! {geld_einzahlen, 13, omaGeschenk, 50}.
{geld_einzahlen,13,omaGeschenk,50}
OK: 470
```

Durch einen unglücklichen Vorfall verliert der Student nur drei Tage später seine Bankkarte. Natürlich ruft er sofort die Bank an und lässt sein Konto sperren.

```
(bc@localhost)6> filialeOberndorf ! {konto_sperren, 13}.
{konto_sperren,13}
OK: 'Konto wurde gesperrt'
```

Das Konto ist ab diesem Zeitpunkt gesperrt und es können keine Zugriffe darauf stattfinden. Zum Glück findet Martin seine Bankkarte schon bald wieder. Am 28.07. hebt er 20 Euro am Geldautomaten ab, um ins Kino zu gehen.

```
(bc@localhost)7> geldautomat_Kinokomplex ! {geld_auszahlen, 13, 20}.
{geld_auszahlen,13,20}
OK: 450
```

Am Monatsende wird noch einmal die Historie über den gesamten Monat ausgegeben.

```
(bc@localhost)8> martin_onlineBanking ! {historie, 13}.
{historie,13}
OK: [{transaktionsliste,[{<6747.299.0>,auszahlung,
                         {zeit,{2013,7,28},{13,16,46}},
                         {notizen,[]},
                         {wer,13},
                         {wert,20}},
                        {<6747.285.0>,entsperrung,
                         {zeit,{2013,7,25},{13,16,43}},
                         {notizen,"Konto entsperrt"},
                         {wer,13},
                         {wert,0}},
                        {<6747.255.0>,sperrung,
                         {zeit,{2013,7,22},{13,12,27}},
                         {notizen,"Konto gesperrt"},
```

```
16                          {wer,13},
17                          {wert,0}},
18                         {<6747.236.0>,einzahlung,
19                          {zeit,{2013,7,19},{13,10,5}},
20                          {notizen,omaGeschenk},
21                          {wer,13},
22                          {wert,50}},
23                         {<6747.228.0>,erfragung,
24                          {zeit,{2013,7,13},{13,6,3}},
25                          {notizen,"Kontostand wurde abgefragt"},
26                          {wer,13},
27                          {wert,0}},
28                         {<6747.187.0>,auszahlung,
29                          {zeit,{2013,7,05},{13,2,4}},
30                          {notizen,[]},
31                          {wer,13},
32                          {wert,200}},
33                         {<6747.173.0>,einzahlung,
34                          {zeit,{2013,7,03},{12,57,34}},
35                          {notizen,14},
36                          {wer,13},
37                          {wert,60}},
38                         {<6747.125.0>,einzahlung,
39                          {zeit,{2013,7,01},{12,53,8}},
40                          {notizen,16},
41                          {wer,13},
42                          {wert,560}}]},
43     {vermoegen,450}]
```

Fazit Die Fallstudie zeigt, dass die grundlegenden Anforderungen an einen Banking Server erfüllt werden. Die Funktionalitäten werden restlos angeboten und durch den Umstand, dass der Server auf einem Knoten läuft, können Clients von beliebigen Orten mit ihm verbunden werden. Es ist einleuchtend, dass ein Banking Server ohne ein Authentifikationsverfahren in der realen Welt nicht einsetzbar ist. Die Sicherheitsziele wurden bei der Projektarbeit aber außen vor gelassen. Nach der Fallstudie könnte man den Server als funktionsfähig und einen guten Ansatz betrachten, andererseits bietet dieser Verbesserungspotential und große Lücken im Punkt Sicherheit.

5.1 Profiling

Der zuvor beschriebene exemplarische Anwendungsfall soll nun als Grundlage dienen, die Implementierung mittels eines *Profilers* hinsichtlich der Performanz zu untersuchen. Hierzu stellt die Erlang-Distribution unter anderem die Werkzeuge *fprof* und *cprof* zur Verfügung, die zu diesem Zweck verwendet werden. Während *cprof* nur Funktionsaufrufe zählt, liefert *fprof* ausführlichere Details zum Ablauf, mit dem Nachteil, dass die Ausführungsgeschwindigkeit des Programms gleichzeitig negativ beeinflusst wird. Da der Profiler unvollständige und teilweise abgefälschte Daten liefert, wenn Worker-Prozesse von außen beendet werden, muss die Störungssimulation für die Analysen deaktiviert werden. Dies ist ohnehin notwendig um sinnvolle Werte zu erhalten, da Störungen in der Realität nur selten auftreten.
Die in den folgenden Tabellen angegebenen Zeitwerte ergeben sich aus der durchschnittlichen Ausführungsdauer inklusive aufgerufener Funktionen.

Tabelle 2: Aufrufstatistik der Worker-Funktionen

Modul	Funktion	Aufrufe	Zeit
bw	geld_auszahlen	4	31,536
bw	geld_einzahlen	4	19,529
bw	geld_ueberweisen	4	47,131
bw	historie	1	15,990
bw	konto_entsperren	1	31,093
bw	konto_sperren	1	15,990
bw	kontostand_abfragen	1	15,096

Im Folgenden werden Aufrufe an das `dets`-Modul betrachtet, da für die persistente Speicherung der größte Zeitanteil benötigt wird. Über der durchgezogenen Linie werden jeweils zusätzlich die aufrufenden Funktionen und die Dauer, die für die Aufrufe benötigt wurde, aufgelistet.

Tabelle 3: Aufrufstatistik für `dets:open_file`

Modul	Funktion	Aufrufe	Zeit
bw	daten_lesen	9	10,318
bw	daten_schreiben	8	4,06
dets	open_file	17	7,373

Tabelle 4: Aufrufstatistik für `dets:close`

Modul	Funktion	Aufrufe	Zeit
bw	daten_lesen	9	1,821
bw	daten_schreiben	8	3,992
dets	close	17	7,373

In Tabelle **??** wird deutlich, dass die Worker-Funktionen verhältnismäßig lange damit beschäftigt sind, die Datenbank per `dets:open_file` zu öffnen. Bei einer nachträglichen Optimierung sollte daher ein *Handle* zur Datenbank an den Worker übergeben werden, statt Diese jedes mal erneut zu öffnen. Weiterhin fällt auf, dass `open_file` 17 mal aufgerufen wird, obwohl nur neun Transaktionen in der Fallstudie durchgeführt werden. Der Grund dafür ist, dass sowohl `daten_lesen` als auch `daten_schreiben` die Datenbank jeweils öffnen und wieder schließen. Eine Optimierung erübrigt sich jedoch durch die vorhergehende. Dennoch fällt mit Kenntnis der Tatsache, dass `daten_schreiben` für jede Transaktion nach `daten_lesen` aufgerufen wird auf, dass `open_file` schneller arbeitet, wenn die Datenbank im selben Prozess bereits zuvor geöffnet und wieder geschlossen wurde.

Tabelle 5: Aufrufstatistik für `dets:insert`

Modul	Funktion	Aufrufe	Zeit
bs	erzeuge_transaktion	9	0,018
bw	daten_schreiben	21	6,624
dets	insert	30	4,642

Tabelle 6: Aufrufstatistik für `dets:lookup`

Modul	Funktion	Aufrufe	Zeit
bw	daten_lesen	9	0,047
dets	insert	9	0,047

Wie in Tabellen **??** und **??** ersichtlich, wird für das Schreiben in die Datenbank erwartungsgemäß mehr Zeit als für das Lesen benötigt.

6 Implementierung

6.1 Client

Das Client-Modul ist die eigentliche Schnittstelle der Applikation für den Benutzer. Mittels des Moduls kann ein Nutzer möglichst komfortabel Anfragen an den Banking-Server schicken, ohne sich dabei Gedanken machen zu müssen, wie der Server an sich aufgebaut ist und angesprochen werden müsste.

Um einen Client zu starten, wird die Funktion *start* aufgerufen, welche als einzigen Parameter den Namen, auf den der Client registriert werden soll, entgegennimmt. Alle weiteren Aktionen erfolgen ausschließlich über Nachrichten, die an den registrierten Client geschickt werden. Dies hat den Vorteil, dass innerhalb einer Shell mehrere Clients gleichzeitig benutzt werden können.

Die Funktion des Clients soll durch folgende Aufrufe aus der Shell verdeutlicht werden:

```
1 (bc@localhost)1> bc:start(client1).
2 Banking Client wurde gestartet
3 ok
4 (bc@localhost)2> client1 ! konto_anlegen.
5 konto_anlegen
6 OK: 6
```

Im ersten Schritt wird ein neuer Client mit dem Namen "*client1*" erzeugt. Danach ist es möglich, über den Client Aufträge an den Server zu schicken.

So wird in Schritt drei mittels des Befehls *konto_anlegen* ein neues Konto angelegt. Als Antwort für die Anfrage bekommt der Client "*OK: 6*" zurück. Dies bedeutet, dass die Anfrage korrekt ausgeführt wurde, und ein neues Konto mit der Kontonummer "*6*" angelegt wurde.

Die verschiedenen Befehle, die der Client ausführen kann und deren Syntax, sind in der nachfolgenden Tabelle aufgelistet.

Tabelle 7: Client Funktionen

Funktion	**Syntax**
Konto anlegen	client ! *konto_anlegen*
Konto löschen	client ! {*konto_loeschen*, Kontonummer}
Kontostand abfragen	client ! {*kontostand_abfragen*, Kontonummer}
Historie ausgeben	client ! {*historie*, Kontonummer}
Geld einzahlen	client ! {*geld_einzahlen*, Kontonummer, Betrag, Verwendungszweck}
Geld auszahlen	client ! {*geld_auszahlen*, Kontonummer, Betrag}
Geld überweisen	client ! {*geld_ueberweisen*, ZielKontonummer, UrsprungsKontonummer, Betrag}
Dispokredit beantragen	client ! {*dispokredit_beantragen*, Kontonummer}
Konto sperren	client ! {*konto_sperren*, Kontonummer}
Konto entsperren	client ! {*konto_entsperren*, Kontonummer}
Client beenden	client ! *stop*

Es ist zu beachten, dass die Nachrichten stets an den registrierten Client-Namen geschickt werden müssen. Der in der Tabelle verwendete Client-Name (*client*) dient lediglich der Veranschaulichung. Funktionen, die Übergabeparameter enthalten, müssen grundsätzlich als Tupel geschickt werden und zwingend alle Argumente enthalten. Wird dem Client eine Nachricht geschickt, deren Kommando er nicht kennt, oder deren Argumente unvollständig sind, quittiert er diese mit *"Unbekanntes Kommando"*.

Die Rückgabe des Clients ist entweder positiv oder negativ. Konnte die Anfrage erfolgreich bearbeitet werden, gibt der Client "*OK:* " mit den angehangenen Rückgabewerten des Worker-Prozesses zurück. Im negativen Fall gibt der Client "*Fehler:* " mit den angehangenen Fehlerinformation des Workers-Prozesses zurück.

Als nicht erfolgreich werden lediglich fehlerhafte Anfragen gewertet, beispielsweise die Anfrage des Kontoguthabens zu einem nicht existierenden Konto. Fehlerhaftes Verhalten eines Worker-Prozesses, welches dazu führt, dass für eine Anfrage ein Worker neu gestartet werden muss, bleibt für den Client verborgen.

Die eigentliche Anfrage an den Server erfolgt durch einen asynchronen Aufruf des Servers, wie folgender Codeausschnitt des Client-Moduls zeigt.

```
1 loop() ->
2   receive
3     konto_anlegen ->
```

```
4           gen_server:cast({bs, 'bs@localhost'}, {konto_anlegen, self()}),
5           loop();
6       {kontostand_abfragen, KontoNr} ->
7           gen_server:cast({bs, 'bs@localhost'}, {kontostand_abfragen, self(),
                KontoNr}),
8           loop();
9           ...
10  end.
```

Da es sich bei dem Server um einen generischen Server handelt, erfolgt der Aufruf mittels *gen_server(Name, Nachricht)*. Der Server läuft auf einem anderen Node als der Client. Deshalb ist es notwendig, als Namen nicht nur den registrierten Namen des Servers anzugeben, sondern auch den Namen des Nodes ('bs@localhost'), auf dem dieser ausgeführt wird. Als Nachricht werden neben der Aktion, die durchgeführt werden soll, die dazu nötigen Argumente und die eigene Prozess-ID übergeben. Die eigene Prozess-ID ist notwendig, um dem Client eine Antwort vom Worker-Prozess aus schicken zu können.

6.2 Server

Das Server-Modul dient der Interaktion von Client und Worker-Prozessen. Das Modul nutzt die "gen_server"-Funktionalität, welche eine möglichst einfache Client/Server-Kommunikation ermöglicht.
Der Server wird mittels `bs:start()` gestartet. Diese Funktion ruft intern `gen_server:start_link(ServerName, CallBackModule, Arguments, Options)` auf, wie in folgendem Modulausschnitt zu sehen ist.

```
1 start() -> gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).
```

Der Aufruf lässt erkennen, dass keine erweiterten Funktionalitäten des generischen Servers benutzt werden. Es wird lediglich der Name des Servers festgelegt, welcher gleich dem Modulnamen ist und der Name des Callback-Moduls, welches dem eigenen Modul entspricht. Da für die Funktionalität des Banking-Servers keine Initialisierungsdaten und Optionen benötigt werden, werden hierfür nur leere Listen übergeben.
Durch den Aufruf von `gen_server:start_link` wird die Init-Funktion des Banking-Servers ausgeführt. Innerhalb der Init-Funktion wird das Flag "*trap_exit*" auf *true* gesetzt. Dies führt dazu, dass vom Server aufgespannte Prozesse an ihn eine Nachricht senden, wenn sie beendet werden und er im Falle eines abgestürzten Prozesses nicht selber beendet wird. Zudem wird in der Init-Funktion die *Dets*-Datenbank geöffnet, welche die offenen Transaktionen verwaltet. Geschlossen wird die Datenbank erst wieder, wenn der Server innerhalb der `terminate`-Methode beendet wird.
Die Hauptaktivität des Banking-Servers liegt in den Methodenaufrufen von `handle_cast(Name, Message)`. Dies sind die Callback-Funktionen der vom Client aufgerufenen `gen_server:cast(...)`-Methoden. Für jeden Transaktionstyp existiert dabei eine `handle_call`-Callback-Methode. Wie

diese im Detail aufgebaut sind, soll folgender Ausschnitt verdeutlichen.

```
handle_cast({geld_einzahlen, ClientPId, Kontonr, Verwendungszweck, Betrag},
    LoopData) ->
  erzeuge_transaktion(geld_einzahlen, [ClientPId, Kontonr, Verwendungszweck,
      Betrag]),
  {noreply, LoopData};
```

Zu Erkennen ist, dass die Nachricht jeweils auf die auszuführende Transaktion gematched wird. Im dargestellten Fall auf "*geld_einzahlen*". Die Reihenfolge der Argumente ist im Wesentlichen stets die selbe. Nach der auszuführenden Aktion folgt die Client-PID, mittels derer der Worker-Prozess später eine Antwort an den Client zurückschicken kann. Danach folgt, falls nötig, die Kontonummer und anschließend weitere Argumente, die für die Ausführung der Aktion notwendig sind.
Innerhalb der Funktion wird die Methode `erzeuge_transaktion(Aktion, Args)` aufgerufen, welche im folgenden Ausschnitt gezeigt wird.

```
erzeuge_transaktion(Action, Arg) ->
  TId = spawn_link(bw, init, []),
  dets:insert(transaction, {TId, {Action, Arg}}),
  TId ! [Action|Arg].
```

Die Funktion spannt den Worker-Prozess auf und ruft die in ihm enthaltene Init-Funktion auf. Die zurückerhaltene Prozess-ID wird zusammen mit dem Transaktionstyp und den zusätzlichen Argumenten in der Transaktionsdatenbank gespeichert. Anschließend wird an den gestarteten Worker eine Nachricht mit der auszuführenden Transaktion und den Argumenten gesendet.
Wie bereits erwähnt, überwacht der Server die Worker-Prozesse, und bekommt eine Nachricht, falls sich ein solcher beendet. Hierbei wird zwischen einem normalen und einem fehlerhaften Beenden unterschieden. Wie dies getan wird zeigt folgender Ausschnitt aus dem Server-Modul.

```
handle_info({'EXIT', TId, error}, LoopData) ->
   io:format("Worker Exit: ~p (~p)~n", [error, TId]),
   ...
  {noreply, LoopData};

handle_info({'EXIT', PId, normal}, LoopData) ->
   io:format("Worker Exit (not handled): ~p (~p)~n", [normal, PId]),
   dets:open_file(transaction, [{file, "db_transaction"}, {type, set}]),
   dets:delete(transaction, PId),
   dets:close(transaction),
{noreply, LoopData}.
```

Es ist ersichtlich, dass die Nachrichten, welche über die Beendigung eines Prozesses informieren, mit der Callback-Funktion `handle_info(Nachricht, LoopData)` abgehandelt werden. Übergeben wird dabei in jedem Fall die Zeichenkette *'EXIT'*, die Prozess-ID des beendeten Prozesses und die Art der Beendigung als Atom.
Ist die Beendigungsart als "*normal*" angegeben, bedeutet dies, dass der Prozess ordnungsgemäß durchgelaufen ist. Sollte dies der Fall sein, wird aus der Transaktionsdatenbank des Servers die Transaktion, für welche der beendete Prozess zuständig war, entfernt. So stehen innerhalb der Datenbank nur Transaktionen, die noch ausgeführt werden müssen, oder gerade ausgeführt werden.
Ist ein Prozess "gekillt" worden, oder abgestürzt, lautet der Beendigungsgrund "*error*". Da nicht klar ist, ob der Prozess zum Zeitpunkt seines Absturzes bereits die Transaktion ausgeführt hat oder nicht, muss zunächst in der Datenbank der Worker geprüft werden, ob die Transaktion auf dem entsprechenden Konto bereits gespeichert wurde. Ist dies der Fall, kann die Transaktion aus der Transaktionsliste des Servers gelöscht werden. Ansonsten muss diese neu gestartet werden.
Besondere Aufmerksamkeit muss man Transaktionen widmen, die zwei Konten gleichzeitig bearbeiten. Darunter fällt eine Überweisung. Hier wird von einem Konto zunächst Geld abgebucht und daraufhin auf das andere Konto eingezahlt. Ist der Worker-Prozess genau zwischen diesen zwei Schritten abgestürzt, kann die Überweisungstransaktion nicht einfach erneut ausgeführt werden, da dies dazu führen würde, dass zweimal Geld abgebucht wird. Um dies zu verhindern, kann eine Überweisung wahlweise komplett neu gestartet werden, oder im zweiten Schritt (Geld auf dem Empfängerkonto gutschreiben) fortgesetzt werden.

6.3 Worker

Im Folgenden wird genauer auf die Implementierung des Worker-Prozesses eingegangen. Es wird die Strukturierung des Kontos erklärt, sowie der Aufbau einer Transaktionsliste und welche Transaktionen existieren. Weiterhin wird beispielhaft auf externe und interne Funktionen des Workers eingegangen.

6.3.1 Allgemeines

Das Worker-Modul ist ein Prozess, in dem die Anfrage des Clients verarbeitet wird. Es wird vom Server für jede Benutzereingabe des Clients aufgerufen. Das Modul wird durch den Server mithilfe der `init()`-Funktion aufgerufen. Um dem Worker anschließend mitteilen zu können, welche Funktion er ausführen soll, bleibt dieser in einem *receive*-State, in welchem er den Befehl zusammen mit dem Übergabeparameter an interne Funktionen weiterleitet. Dies ist anhand von folgendem Codebeispiel nachvollziehbar.

```
1 init() ->
2   receive
3       [konto_anlegen, ClientPId] -> konto_anlegen(ClientPId);
```

```
4         [konto_loeschen, ClientPId, Kontonr] -> konto_loeschen(ClientPId,
              Kontonr);
5         [kontostand_abfragen, ClientPId, Kontonr] ->
              kontostand_abfragen(ClientPId, Kontonr);
6         ...
7     end.
```

6.3.2 Das Konto

Das Konto ist das Herzstück des Servers und stellt das Konto des Benutzers dar. Es kann mithilfe des Aufrufs `konto_anlegen` angelegt und mit diversen weiteren Funktionen editiert werden. So kann ein Sperrvermerk auf das Konto gelegt werden, als auch Geld eingezahlt und abgehoben werden. Das Konto enthält Informationen über das Vermögen des Besitzers, wie hoch der Dispokredit ist, wie stark dieser verzinst wird, einen Sperrvermerk und weiterhin noch eine Transaktionsliste. In dem Workermodul ist dies wie folgt realisiert worden:

```
1 {KONTONUMMER,
2    {sperrvermerk, false/true},
3    {vermoegen, VERMöGEN},
4    {maxDispo, DISPOHöHE},
5    {dispoZins, DISPOZINS},
6    {transaktionsliste, TRANSAKTIONSLISTE}
7 }
```

Folgende Tabelle zeigt alle Funktionen, mit denen das Konto manipuliert werden kann.

Item	Funktion
Kontonummer	konto_anlegen konto_loeschen
Sperrvermerk	konto_sperren konto_entsperren
Vermögen	geld_einzahlen geld_abheben geld_ueberweisen
DispoZins und MaxDispo	dispokredit_beantragen
Transaktionsliste	Auf die Transaktionsliste kann nicht von außen zugegriffen werden, da diese bei jeder Manipulation des Kontos automatisch aktualisiert wird .

Um das aktuelle Vermögen eines Kundenkontos zu bekommen, kann die Funktion *kontostand_abfragen* aufgerufen werden.

6.3.3 Die Transaktionsliste

Die *Transaktionsliste* dokumentiert alle Zugriffe, welche mit dem Konto stattfinden. Sei dies nun das Transferieren von Vermögen, oder dass eine Überweisung getätigt wird, es wird jedes mal ein neuer Eintrag in der Transaktionsliste erstellt. Sie ist dem in der realen Welt vorhanden Kontoauszug nachempfunden und hat deswegen zum beschreiben der einzelnen Transaktionen auch fest definierte Atome für die Typifizierung der einzelnen Einträge.

- *erstellung*
- *erfragung*
- *sperrung*
- *entsperrung*
- *einzahlung*
- *loeschung*
- *auszahlung*
- *ueberweisung*

Diese Atome stehen an zweiter Stelle, direkt nach der *Transaktions-ID*. Die *Transaktions-ID* ist in allen Konten einmalig und wird vom Server verwendet um zu überprüfen, ob die Transaktion nach einem Fehler erneut ausgeführt werden muss (siehe: Server). Weiterhin sind in der *Transaktionsliste* Felder vorgesehen, wie das Datum und die Uhrzeit zu der auf das Konto zugegriffen wurde, zusätzliche Notizen, die Person, welche auf das Konto zugegriffen hat und außerdem noch der Betrag um den sich ein Feld änderte.

6.3.4 Worker-Funktionen

Im Client wird zwischen zwei verschiedenen Arten von Funktionen unterschieden. Es gibt ëxterne Funktionenünd ïnterne Funktionen". Die externen Funktionen werden vom Server initialisiert und haben schlagenden Charakter. Interne Funktionen hingegen existieren um Übersichtlichkeit und Struktur in das Programm zu bringen, als auch damit Funktionen nicht mehrfach implementiert werden müssen, wie z. B.: Schreib-/ Lesezugriffe auf die Datenbank.

Interne Funktionen Die internen Funktionen sind Methoden, welche von den externen Funktionen benötigt werden. Sie werden von allen externen Funktionen benutzt und können in drei Bereiche unterteilt werden.

Error-Handling kann nur von der internen Funktion `daten_lesen` aufgerufen werden, da ein kritischer Fehler, welcher nicht durch den Server abgefangen werden kann, nur dann entsteht, wenn die Datenbank nicht geöffnet oder erstellt werden kann.

Datenbankinteraktionen wurden um sicher zu gehen mit dem Modul *dets* realisiert. Es garantiert, dass mehrere Prozesse gleichzeitig aus der Datenbank lesen und schreiben können. Alle externen Funktionen, welche ein Konto manipulieren, rufen in ihrem Quellcode, vielleicht auch über weitere interne Funktionen, die Methoden *daten_lesen* und `daten_schreiben` auf. In folgenden Beispielcode wird die Funktion `daten_lesen` beschrieben.

```
1 daten_schreiben(Konto) ->
2    dets:open_file(konten, [{file, "db_konten"}, {type, set}]),
3    dets:insert(konten, Konto),
4    dets:close(konten).
```

Kontoänderungen geschehen über `kontoinfo` oder `kontolog`. Ist eine bestimmte Information des Kontos gesucht, kann diese mithilfe der Funktion `kontoinfo` abgefragt werden.

Funktion	Übergabeparameter	Rückgabewert	Kurzbeschreibung
kontoinfo	Feld; Konto	Inhalt des Feldes	extrahiert den Wert des gesuchten Feldes und gibt diesen zurück.
kontolog	Operation; Notizen, Kontonummer, Betrag; Konto	Aktualisiertes Konto	schreibt Typifizierung (Siehe Transaktionsliste) zusammen mit den weiteren Parametern in das Konto.
kontochange	Feld; Wert; Konto	Aktualisiertes Konto	schreibt in das entsprechende Feld im Konto den übergebenen Wert.

Externe Funktionen Die externen Funktionen zeichnen sich dadurch aus, dass diese von dem Server aus aufgerufen werden können. Sie spiegeln die Interaktionsmöglichkeit zwischen Client und dem Server wieder. Folgende Tabelle zeigt übersichtlich, welche Funktionen bereits existieren. Zusätzlich werden noch Übergabeparameter, Rückgabewerte und eine Kurzbeschreibung angezeigt.

Funktion	Übergabeparameter	Rückgabewert	Kurzbeschreibung
konto_anlegen	PID des Clients	{ok, Kontonummer}	legt ein neues Konto in der Datenbank an. Existiert die Datenbank nicht, wird eine neue angelegt.
kontostand_abfragen	PID des Clients; Kontonummer	{ok, Kontostand}	liest das aktuelle Vermögen mithilfe der Kontonummer aus dem angegebenen Konto.
history	PID des Clients; Kontonummer	{ok, [Transaktionsliste, Kontostand}	gibt alle Zugriffe, welche mit dem Konto durchgeführt wurden, zusammen mit dem aktuellen Kontostand zurück.
konto_sperren	PID des Clients; Kontonummer	{ok, 'Konto wurde gesperrt'}	setzt den Sperrvermerk im Konto auf *true*
konto_entsperren	PID des Clients; Kontonummer	{ok, 'Konto wurde entsperrt'}	setzt den Sperrvermerk im Konto auf *false*
geld_einzahlen	PID des Clients; Kontonummer; Verwendungszweck; Betrag	{ok, Vermoegen}; {nok, 'Konto gesperrt'}	zahlt auf das in der Kontonummer angegebene Konto den Betrag ein, und fügt in der Transaktionsliste als Notiz den Verwendungszweck hinzu.
konto_loeschen	PID des Clients; Kontonummer	{ok, Kontonummer}; {nok, 'Konto gesperrt'}	Wenn das Konto nicht gesperrt ist, wird es gelöscht.
geld_abheben	PID des Clients; Kontonummer; Betrag	{ok, Vermoegen}; {nok, 'Konto gesperrt'}; {nok, 'Nicht genug Geld'}	Wenn genug Geld auf dem Konto ist kann eine Auszahlung stattfinden, dies schließt den Dispokredit mit ein.
geld_ueberweisen	PID des Clients; Ziel-Kontonummer; Kontonummer; Betrag	{ok, 'Geld wurde ueberwiesen'}; {nok, 'Eins der Konten ist gesperrt'}; {nok, 'Nicht genug Geld'}	Hier wird von Kontonummer zu Ziel-Kontonummer der Betrag überwiesen.
dispokredit_beantragen	PID des Clients; Kontonummer	{ok, 'Dispokredit'}; {nok, 'Konto gesperrt'}	Es wird ein Dispokredit von 10% des Vermögens gewährt und ein DispoZins auf 12% gesetzt.

7 Fazit

Zusammengenommen ist in dem Projekt das gewünschte Ergebnis erreicht worden und der *einfache* Banking-Server wurde voll funktionsfähig implementiert. Alle wesentlichen Anforderungen konnten umgesetzt werden. Besonders profitieren konnte die Projekt-Gruppe A dabei vom Einsatz des Behaviour-Moduls *gen_server*. Die Umsetzung gelang derart, dass der Server auch von anderen Rechnern über den Knoten *bs@localhost* angesprochen werden kann. Das Projekt erforderte eine tiefgehende Auseinandersetzung mit der Erlang-Programmierumgebung und gute Teamarbeit. Abschließend kann festgestellt werden, dass diese, oder eine ähnliche Serverstruktur, durchaus in der Programmiersprache Erlang implementiert werden kann und auch in der Wirtschaft eingesetzt werden könnte. Insbesondere die Parallelität und die Informationsvermittlung über das Senden von Nachrichten sind Aspekte, die Potential bieten und für einen Einsatz von Erlang-Serverstrukturen sprechen.

8 Glossar

Behavior-Modul Ein Behavior-Modul ist vergleichbar mit einer Schnittstelle. Sobald es eingesetzt wird, müssen durch das Modul spezifizierte Callback-Funktionen implementiert werden.

dets Erlang-Modul, dass eine Schnittstelle zur persistenten Datenspeicherung bietet und einige Methoden für einen einfachen Datenzugriff abstrahiert.

gen_server Behavior-Modul *generischer Server*, das Richtlinien zur Implementierung eines Servers festlegt

Profiler Werkzeug, um Zeitmessungen in einer Software durchzuführen

Worker(-Prozess) Prozess, der nur zur asynchronen Abarbeitung einer spezifischen Aufgabe von einem übergeordneten Prozess gestartet wird und sich nach Fertigstellung dieser selbst beendet.

9 Literaturverzeichnis

Für die Projektarbeit wurde auf die folgenden Quellen zurückgegriffen:

- http://www.erlang.org/doc/man/gen_server.html
- http://www.erlang.org/doc/design_principles/gen_server_concepts.html
- http://learnyousomeerlang.com/clients-and-servers
- http://www.erlang.org/doc/man/dets.html
- http://www.erlang.org/doc/efficiency_guide/profiling.html
- http://20bits.com/article/erlang-a-generic-server-tutorial
- http://www.erlang.org/doc/apps/tools/cprof_chapter.html
- http://www.erlang.org/doc/apps/tools/fprof_chapter.html
- http://www.erlang.org/doc/man/fprof.html
- Francesco Cesarini, Simon Thompson: Erlang Programming (2009)

Für alle Internetquellen gilt der 25.Juni 2013 als das letzte Datum des Zugriffs.