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 24. Juni 2013

Inhaltsverzeichnis

1 Implementierung

1.1 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, das innerhalb einer Shell mehrere Clients gleichzeitig benutzt werden können.
Die Funktion des Clients soll folgender Aufruf aus der Shell verdeutlichen:

```
(bc@localhost)1> bc:start(client1).
Banking Client wurde gestartet
ok
(bc@localhost)2> client1 ! konto_anlegen.
konto_anlegen
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, das 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 1: 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 abgehangenen Rückgabewerten des Worker-Prozesses zurück. Im negativ 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 existierendem Konto. Fehlerhaftes Verhalten eines Worker-Prozesses, welches dazu führt, das 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.

```
loop() ->
 receive
      konto_anlegen ->
          gen_server:cast({bs, 'bs@localhost'}, {konto_anlegen, self()}),
```

```
            loop();
        {kontostand_abfragen, KontoNr} ->
            gen_server:cast({bs, 'bs@localhost'}, {kontostand_abfragen, self(), KontoNr}),
            loop();
          ...
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 anderem 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 und die dazu nötigen Argumente auch die eigene Prozess Id übergeben, damit der Worker Prozess später an den Client eine Antwort schicken kann.

1.2 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.

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

Der Aufruf lässt erkennen, das 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 eigenem 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, das vom Server aufgespannt Prozesse an ihn eine Nachricht senden, wenn sie beendet werden und er im Falle eines "gekillten" 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 beendet wird, innerhalb der "*terminate*" Methode.
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 stets die Client PID, mittels derer der Worker-Prozess später eine Antwort an den Client zurückzuschicken kann. Danach folgt falls nötig die Kontonummer und anschließend weitere Argumente, die für die Ausführung der Aktion nötig sind. Innerhalb der Funktion wird stets 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 zurückerhaltende 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, das 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 das Atom *'Exit'*, die Prozess Id des beendeten Prozesses und die Art der Beendigung als Atom.
Ist die Beendigungsart als "*normal*" angegeben, bedeutet dies, das 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, ist 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, welche zwei Konten gleichzeitig bearbeiten. Darunter fällt eine Überweisung. Hierbei 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, das 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.