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 28. Juni 2013

Inhaltsverzeichnis

1 Teamorganisation

Die Projekt-Gruppe 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 auseinander setzte. Auf Grund einer unbekannten Programmierumgebung waren das letztendliche Design und die Schnittstellen nicht von Anfang an abzusehen. So war über den gesamten Zeitraum der Bearbeitung eine enge Zusammenarbeit und gute Kommunikation Grundlage für ein erfolgreiches Umsetzen des Projekts.

2 Implementierung

2.1 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 compilieren.

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.

Jedoch 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. 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:

```
(bc@localhost)1> Client1 ! konto_anlegen.
konto_anlegen
OK: 2
(bc@localhost)1> Client1 ! {konto_abfragen, 2}.
konto_abfragen
OK: 0
```

2.2 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:

```
(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, 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 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 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.

```
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 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 Antwork vom Worker Prozess aus schicken zu können.

2.3 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, 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 das Atom *'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.

2.4 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.

2.4.1 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 plus Übergabeparameter auf interne Funktionen mappt. Dies ist anhand von folgendem Codebeispiel nachvollziehbar.

```
init() ->
    receive
        [konto_anlegen, ClientPId] -> konto_anlegen(ClientPId);
        [konto_loeschen, ClientPId, Kontonr] -> konto_loeschen(ClientPId, Kontonr);
        [kontostand_abfragen, ClientPId, Kontonr] -> kontostand_abfragen(ClientPId,
            Kontonr);
        ...
```

```
    end.
```

2.4.2 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 kann es editiert werden. So kann ein Sperrvermerk auf das Konto gelegt werden, als auch natürlich Geld eingezahlt und abgehoben werden. Das Konto enthält Informationen über das Vermögen des Besitzers, wie hoch der Dispokredit ist, wie stark dieser verzinst wird, ein Sperrvermerk und weiterhin noch eine Transaktionsliste. In dem Workermodul ist dies folgendermaßen realisiert worden:

```
{KONTONUMMER,
    {sperrvermerk, false/true},
    {vermoegen, VERMöGEN},
    {maxDispo, DISPOHöHE},
    {dispoZins, DISPOZINS},
    {transaktionsliste, TRANSAKTIONSLISTE}
}
```

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.

2.4.3 Die Transaktionsliste

Die *Transaktionsliste* dokumentiert alle Zugriffe, welche mit dem Konto stattfinden. Sei dies nun Vermögen transferieren, oder dass eine Überweißung getätigt wird. Sie ist dem in der realen Welt

vorhanden Kontoauszug nachempfunden und hat deswegen auch einige 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 *Transaktions-ID* 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.

2.4.4 Worker-Funktionen

Im Client wird zwischen 2 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.

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