KTH Computer Science
and Communication

Parallellprogrammering i Go och Erlang

YUUKI JONSSON, ANDREAS STARRSJÖ

Examensrapport vid CSC
Handledare: Alexander Baltatzis
Examinator: Mads Dam

Referat

Projektet ämnar att jämföra Go och Erlang, genom järmföreleser med Java för att lära oss mera om dessa språk samt deras användningsområden. Detta uppnåddes genom att programmera om en Java-implementation som vi programmerat i en annan kurs[1], och på så sätt lärde vi oss språkens olika styrkor, svagheter och när man ska använda dem. Slutsatserna är att både Erlang och Go är bättre när det gäller programmering av parallella system, än vad Java är. Däremot lämpar sig språken till olika ändamål, Erlang är mer lämpat för back-end kod för serverapplikationer, medan Go kan användas för mer allmän programmering, men är framförallt bättre vid kodning av klient-applikationer.

Abstract

Parallel programming in Go and Erlang

This projects goal was to compare Go and Erlang, with comparisons to Java, and learn more about these languages. This goal was achieved by reprogramming a Java program from an earlier course in the two other languages. By doing this we learned more about the strengths and weaknesses of the language as well as when to use them. Both of these languages are better at programming parallel systems then Java. On the other hand, it is obvious that the languages are not very alike at all and were designed for use in totally different situations. The results showed that Erlang is more suited for back-end code in server applications, while Go can be used in a more general sense, but is especially useful in client-side applications.

Innehåll

Kapitel 1

Inledning

Denna uppsats fokuserar på att ge läsaren en uppfattning av Erlang och Go samt visa vilka egenskaper och brister språken har och hur de kan användas i dagens IT-samhälle. En viktig aspekt av detta är processer. Båda språken har utmärkt stöd för parallellprogrammering och för att skapa processer, detta har utnyttjats i implementationen vars syfte är att förenkla analysen av de olika språken. Implementationen är ett banksystem som ska kunna hantera flera klienter på samma gång, detta måste ske genom parallellprogrammering. Uppsattsen går också igenom andra egenskaper (ex. data hantering, nätverkskommunikation) som språken har då de är nödvändiga för att få implementationen att fungera, dessa egenskaper kommer däremot inte att analyseras i detalj. Själva implementationen är en laboration från kursen DD1361 på kth [13].

1.1 Bakgrund

Med dagens teknikutveckling går processorer mot att ha ett ökat antal kärnor. Inom en snar framtid kommer processorer ha fler kärnor än antalet processer ett genomsnittligt program i dagsläget har. Vad som då händer är att datorn inte kommer kunna använda sig av hårdvaran fullt ut. Det kan till och med gå långsammare vid användning av en processor med många kärnor jämfört med en processor med endast en kärna, om inte programmet är optimerat för en flerkärnig processor [2]. Eftersom operativsystemet inte kan veta om delar av programmet går att köra parallellt eller inte så är det upp till programutvecklaren att se till att programmet är optimerat för att köras som flera processer.

1.2 Syfte

Det övergripande syftet för denna rapport var att analysera och jämföra två programmeringsspråk vi inte kunde, Go och Erlang, med ett språk vi hade tidigare erfarenhet av: Java. För att uppnå detta mål införskaffade vi teknisk information om dem. Vår primära fokus låg på processer, det vill säga vad språken tillhanda-

håller för verktyg för att skapa och hantera processer. Vi koncentrerade syftet till att analysera användningsområdet för respektive språk genom att väga in de styrkor och svagheter språken har. Fokus låg framförallt på processer men också andra egenskaper så som paradigmskillnader.

1.3 Redovisning av samarbete

Det är mycket svårt att definera vem som har gjort exakt vad i detta projekt. Yuuki var den som var ansvarig för Go programmet, Andreas var ansvarig för Erlang programmet. Uppsatsen var uppdelat på liknande sätt. Dock har vi båda gått igenom varandras texter och kod åtskilliga gånger och gjort förändringar.

Kapitel 2

Teknisk Bakgrund

2.1 Lista med förkortningar, definitioner och förklaringar

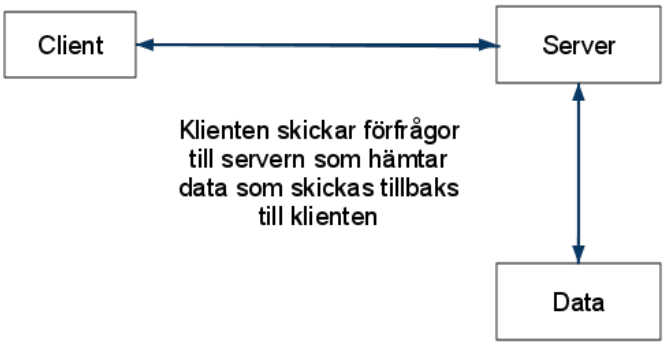
Processor: Hårdvara i en dator som utför uträkningar, kan refereras som datorns "hjärna" .
Processorkärna: Delkomponent till processorer som gör själva uträkningarna, en processor kan innehålla flera av dessa för att göra uträkningar samtidigt.
Process: Operativsystemet delar upp processorns kapacitet för att simulera parallell körning. Varje process får en viss körtid i processorn. Vad som egentligen sker är att operativsystemet byter mellan processerna i en hastighet som en människa inte kan uppfatta, processerna ser alltså ut som om de kör parallellt. En process har också eget minne som den får tilldelat av operativsystemet[11].
PID: Förkortning för Process ID, varje process i ett linux- eller unixsystem får ett ID av operativ systemet.
Tråd: Trådar är som processer ett verktyg för att dela upp processorn för att göra uträkningar samtidigt. Däremot så skiljer sig trådar genom att de delar minne om de skapas av samma process. Trådar skapas alltid under processer d.v.s en process kan innehålla flera trådar. [11, s.93]
Mutex: Mutex är ett lås som används vid till exempel parallellprogrammering. Mutex kan låsa en process så att den inte ska köra innan en annan process har kört klart och låser upp mutexet [11, s.128] .
Hot swaping: Hot swaping är när kod byts ut under körning.
TCP/IP: Transmission Control Protocol och Internet Protocol är två olika protokoll för dataöverföring. IP skickar data i småbitar till rätt adress utan att verifiera om det kom fram. TCP ser till att rätt data kom fram. För att skicka data så körs först TCP som genom IP skickar datan och sen verifierar TCP om rätt data kom fram eller inte. [11, s.585]
Typparametrisering: Datastrukturer som blir tilldelade en typ som parameter vid instansiering, för att endast stödja den typen.
GUI: Graphic User Interface, grafiskt gränssnitt.
USB: Universal Serial Bus, en standardiserad port till datorer. Syftar ibland till

USB-minne.
parallell körning: I denna rapport kommer parallell körning syfta på när användaren uppfattar det som parallellt, alltså kommer sekvensiell körning på samma processorkärna att räknas som parallellt fastän uträkningarna inte körs samtidigt.
parallellprogrammering: När programmet är skrivet så att uträkningar kan köras samtidigt.
Back-end: Den del av programmet som ligger längst bort från användaren, oftast kod som gör själva uträkningarna medan front-end är sånt som GUI.
Erlang-shell: En kommandotolk som kan köra Erlang moduler och program.
race condition: Om två processer använder sig av samma variabler kan leda till att godtyckligt beteende beroende på vilken process som får tag på variablerna först.

2.2 Implementation

Implementation baserades på en Javalaboration[13] från kursen programmeringsparadigm (DD1361). Den gick i stora drag ut på att "studera internet-orienterad programmering och då speciellt kommunikation via sockets."

Uppgiften innebar att man skulle laga och förbättra att mycket simpelt banksystem, där kunder kan logga in på bankens server med sitt kortnummer och en kod, få ett antal valmöjligheter(ta ut pengar, sätta in pengar osv.) via en meny och sedan logga ut. Vi valde denna labb för att det var en intressant uppgift som hade



Figur 2.1. En enkel Klient-Server model.

verklighetsförankring och kändes som ett lagom stort projekt att skriva om till Go och Erlang. Labben hade även potential för att kunna lösas med hjälp av processer, något som vi visste att både Go och Erlang är bra på.

På grund av tidsbrist valde vi att bara genomföra de mest vitala delarna av projektet och ignorera de triviala kraven som rent tekniskt var mycket enkla men tidsödande, till exempel: stöd för flera språk och många av de meddelanden som servern kunde skicka till klienten.

2.3 Go

Vad skaparna av Go ville var att skapa ett lättprogrammerat språk som är snabbt och effektivt vid både körning och kompilering. Samtidigt ville man att språket skulle gå snabbt att skriva så som python eller javascript. De tittade på brister bland de mest använda språken och skapade Go som någon form av sammanslagning av flera olika språk med deras respektive styrkor. Oftast handlade det om att det är för krångligt och ineffektivt att programmera i språk som C++ och Java, däremot vid val av ett lättare språk så blir följden ett långsammare program eller programmering med en mindre strikt kompilator. Konsekvenserna av en sådan kompilator kan vara oavsiktliga buggar av det krångligare slaget eller lång debuggingtid.

2.3.1 Syntax

Ett av Go's främsta syfte är att det ska gå snabbt och enkelt att programmera. Programmet som kodats ska också vara förhållandevist snabbt och effektivt. Det är tydligt att andelen kod som skrivs är överlag mindre än de etablerade kompilerade språken. Petar Maymounko säger detta om Go: "I have reimplemented a networking project from Scala to Go. Scala code is 6000 lines. Go is about 3000.(...)"[4]. Parenteser har plockats bort på majoriteten av platser där både Java och C++ har parenteser, t.ex. i if-satser och for loopar. Det går att likna det lite med Pythons syntax. Men till skillnad från Python används klammerparentes(måsvingar) för indentering.

Den främsta anledningen för att ta bort alla dessa konventioner är att programmeraren inte ska behöva skriva mer än det (enligt skaparna av Go) ytterst nödvändiga, men fortfarande ha läsbar kod. Det är helt enkelt en förändring bara för att slippa skriva långa rader med kod, något som Rob Pikes[3], en av Go's skapare, mer än gärna predikar om. Ett annat intressant designval som har införts är det som i allmänhet brukar förknippas med lokala(private) och globala(public) variabler. I Go är allt, både metoder och variabler(och annat dylikt) public om det börjar med en versal bokstav. De funktioner etc. som börjar med en gemen bokstav är således private. Det är extremt lätt att göra misstag med ett syntax som detta, därför är kompilatorn väldigt bra på att påpeka att ett byte från gemener till versaler (private till public) borde ske.

Att vara tvungen att programmera i en utvecklingsmiljö som hjälper programmeraren att fylla i kod, är något som skaparna av Go helst av allt vill slippa[23]. Större delen av dagens Java-utvecklare sitter i en utvecklingsmiljö som näst intill skriver koden åt dem[25]. I Go kan en sådan miljö utvecklas men den kommer aldrig generera lika mycket kod som en bra utvecklingsmiljö för Java gör. Framför allt behövs det inte (all kod som skrevs till detta projekt skrevs med Notepad++ och Gedit).

Ibland känns det som om språket är gjort för att kunna programmera på så få rader som möjligt. Att semikolon för avslutad rad är valfri, eftersom i kompilatorn lägger till dem automatiskt, är också något som gör språket lättare att skriva. I

sin tur får denna förändring som konsekvens att semikolon används för att skriva flera kodrader på samma rad, med semikolon som radbrytning, vilket får koden att bli ännu mer kompakt. Användningsområdet för semikolon har ändrats genom att implementera om den som ett valfritt syntax för ny rad.

2.3.2 Språkets Struktur

I Go finns det inget som heter klasser, det finns Interface och Struct. Det finns heller ingen typ-hierarki eller något arv. Språket är statiskt typat och har strukturerad typning. Detta är ett designval som gjorts för att göra språket mer typsäkert.

Struct

Struct fungerar ungefär som Struct brukar fungera i de flesta språk, en sorts mindre klass vars huvudsyfte är att hålla data. Sättet att använda structar är väldigt likt, samt har ett liknande syntax, som i till exempel C++. Däremot finns det en avgörande skillnad: det går att skapa funktioner till dem. Som van Java-programmerare ser man snabbt att det går att, med hjälp av funktioner och structar, skapa Go-kod som har liknande struktur som klasser i Java. Structen blir någon form av plats för deklaration av variabler. En speciell funktion som enbart initierar variablerna kan fungera som konstruktor. Däremot är ett "klass-syntax" inte nödvändigt då det inte finns något arv i Go.

Interface

Interface är något som finns i både Java och Go, däremot så används de oftast på olika sätt. Det huvudsakliga syftet är att man som programmerare ska programmera mot ett interface istället för att skapa ett objekt av en klass. Vad interfaceskaparen kan göra är att ändra i koden som implementerar metoderna som interfaces initierar, utan att användaren av interfacet kommer att behöva göra några skillnader förutsatt att programmet fortfarande gör samma sak. Det går att likna det med header-filer i C++. Typer av interface är alltid dynamiska tills de tilldelas till en statisk typ av samma sort.

Arv

I Go finns inga klasser och inget arv, därför finns det heller ingen klasshierarki. Detta får konsekvenser som till exempel att standardbiblioteket inte har någon trädstruktur, men i grund och botten handlar det om att man har valt ett annat sätt att designa språket. Go skulle kunna ha arv även om bristen på klasser gör det bökigt men grundarna har medvetet tagit bort allting som har med arv att göra. Vad som i allmänhet har tagit arvets plats är inkapsling, det vill säga att lägga inre structar och funktioner. Detta är något som används flitigt av vissa Go-programmerare med mer vana.

Utan arv så finns det ingen hierarki mellan typer. Om detta är bra eller dåligt är en väldigt religiös fråga men vad som har implementerats är extremt många olika versioner av "samma" typ. Till exempel har man implementerat 8 olika sorters typer av integer, som ska täcka behovet av större siffror så som Long i Java.

En av anledningarna till att arv är borttaget är att kompilering tar längre tid med arv. Vid ett tillägg av ett paket av standard-biblioteket i Java eller C++ måste kompilatorn gå igenom inte bara koden för det specifika paketet som används, utan också paket som ligger högre upp i hierarkin. Som ett exempel så måste `#include<iostream>` i C++ läsa 25,326 rader från 131 olika filer medan import "fmt" som är motsvarigheten i Go läser 195 rader kod från en fil [4]. Detta är ett sätt för Go att minimera tiden det tar att kompilera program, därför tillåter heller inte kompilatorn inläsning av paket som inte används.

2.3.3 Go-routines

Den största skillnaden mellan Go och andra etablerade språk som Java, Python och C++ är att språket är skapat för att programmera parallellt. Det är med andra ord mycket lättare att skapa processer samt använda sig av dessa. Processerna kallas för Go-routines, som kodas med ett lätt syntax som gör vanliga funktioner till processer. Det främsta skälet till att Google implementerade stöd för parallell bearbetning är den ökade efterfrågan på sådan funktionalitet då processortillverkare nästan bara utvecklar fler-kärniga processorer. Tillverkarna kommer med mycket stor sannolikhet att fortsätta öka antalet processorkärnor i våra datorer i framtiden.

Varje Go-routine är en lättviktsprocess, att skapa en process kostar lite mer än att allokera mer stack-utrymme. Det mesta av komplexiteten vid skapande av processer gömmer språket undan, då det oftast inte är nödvändigt för programmeraren att handskas med variabler som process ID (PID) etc.

För att underlätta processhantering skapar Go en egen minimal stack för varje ny Go-routine som skapas. Detta underlättar för kompilatorn att hantera saker som läggs på stacken, då olika processer inte behöver hämta saker från samma stack. Som programmerare så märks det främst av genom att programmet nästan aldrig hamnar i "stack overflow" under körning om programmet använder sig av Go-routines på ett bra och effektivt sätt.

Go uppmuntrar programmeraren att använda sig så mycket som möjligt av processer. Genom att implementera verktyg för att till exempel kommunicera mellan processer ska man på ett lättare sätt kunna använda sig av dessa för att göra simplare saker mer effektivt.

Här är main funktionen som kör funktionen Write på två olika sätt först i en process och den andra direkt i main-processen.

```
func main(){
    go Write("something")
    Write("something else")
}
func Write(str string){
```

```
    print("write ")
    print(str)
    print("\n")
}
```

Det intressanta här är att programmet skriver ut "something else" innan "something" eftersom det tar längre tid att skapa processen och sen skriva ut istället för att bara göra ett funktionsanrop direkt i main-processen.

2.3.4 Channels

Channels är det verktyg som har implementerats i Go för att processer ska kunna kommunicera med varandra. I till exempel Java kan man med hjälp av globala variabler eller specifika biblioteksklasser, få trådar att kommunicera på ett synkroniserat sätt. Det som är unikt med channels är att det inte är en variabel som delas av olika processer utan en plats i minnet där processer kan läsa och skriva utan att få ett race condition. Det finns många likheter med unix pipes.

På lågnivå rekommenderas [5] det att använda channels som mutex, vilket är väldigt smidigt vid loopar som innehåller kod som skapar nya processer. Däremot så finns det stor potential att använda channels för att kontrollera åtkomsten av data, till exempel om programmet ska kunna köra två olika processer som ändrar på samma variabel så går det att reglera på ett enkelt sätt genom att stoppa in värdet i en channel som endast kan kommas åt av en process i taget.

För ett programmeringsspråk där parallellprogrammering är en så stor del behövs verktyg som channels. Vad programmerare absolut vill undvika vid parallellprogrammering är olika processer som läser och skriver på samma position i minnet, framförallt inte samtidigt. För att undervika detta har föregångare till Go, språk som Newsqueak [6], implementerat channels. Go har influerats av detta sätt att hantera problemet med flera processer som delar på minnet (Notera att Rob Pike också var med och skapade Newsqueak).

2.3.5 Funktioner och metoder

I Go har man lagt in stöd för funktionell programmering, dock klassas inte språket som funktionellt. Det beror framförallt på att Go semantiskt sett är ett imperativt språk, vilket går emot funktionella språkens ideal; att vara fri från tillstånd. Funktioner i Go fungerar som man förväntar sig, kod som gör operationer på parametrarna utan att skapa någon form av klass och utan att ändra på några globala variabler.

Det går att skriva funktioner som är bundna till någon typ. De kallas då för metoder istället för funktioner då de kan få tillstånd som lagras i typen. För att detta ska fungera så måste funktionen ta till exempel en pekare till typen som metoden är bunden till. Eftersom det inte finns några klasser i Go så kan man tänka sig att metoder egentligen är onödigt, men det underlättar vid programmering av större

program som ofta måste hålla reda på fler variabler än vad vanliga funktionella program behöver bry sig om.

Här är koden för metoden SetString(), func är syntaxen för att det är en funktion eller metod. “s * MyString” är för att binda metoden till typen MyString, sedan kommer metodnamn med parametrar som i det här fallet är av typen string. Slutligen kommer returvärdet som är av typen MyString.

```
package main
type MyString string
func (s * MyString) SetString(str string) MyString{
    *s = MyString(str)
    return *s
}

func main(){
    mStr := new(MyString)
    s := mStr.SetString("Hello World!")
    print(s)
}
```

I Go är `main()` alltid en funktion utan returvärde.

2.3.6 Tilldelningsoperatorer

I Go finns det två olika tilldelningsoperatorer. Ena med syntaxet “:=” och andra som bara är ett “=” tecken. Syntaxen “:=” är deklaration och tilldelning på samma gång medan “=” är endast tilldelning. Som insnöad Java-programmerare kan det kännas konstigt att inte behöva definiera variabler innan tilldelning. Eftersom det går att definiera innan tilldelning så går det att använda ett mer Java-likt sätt att skapa variabler. I allmänhet använder sig en van Go programmerare mer av “:=” medan en Java-programmerare skulle ha fler “=” i sin kod.
Följande kod gör samma sak:

```
var i int
i = 9
```

och

```
i := 9
```

2.3.7 Typparametrisering

För tillfället(MAR, 2011) har Go inte något stöd för typparametrisering(generics). På deras hemsida säger de att typparametrisering är praktiska men att de ökar komplexiteten och kan skapa förvirring [7]. Utveklarna av Go har ännu inte hittat en design som förbättrar språket tillräcklig mycket jämfört med komplexiteten

typparametrisering tillför. Det kan implementeras i en senare version av Go om de tycker att det går att genomföra på ett smidigt sätt.

2.3.8 Skräphantering

I språk där det inte finns någon skräphantering går det åt många rader kod för att hantera det manuellt. I till exempel Java klagar många på att dess skräphanterare är långsam, medan många andra tycker att det är värt priset för att slippa ta hand om minnet själv [29].

Go siktar på att få det bästa av båda världar och med hjälp av den senaste forskningen på området, implementera en skräphanterare som är tillräckligt snabb för alla [8]. Tyvärr så är Go fortfarande under utveckling och denna skräphanterare finns inte än. För tillfället använder Go sig av en simplare version. Vi märkte inte av några uppenbara brister i denna enklare version, men vi gjorde inte heller några undersökningar på dess prestanda.

En annan anledning för att ha en skräphanterare är att när man jobbar med flera processer är det svårt att veta exakt var och när man ska ta hand om minnet. Men om det görs med automatik så blir det mycket enklare att skriva bra och tydligare kod.

2.3.9 Nätverksprogrammering

Go använder sig av paketet `net` [20] för att kommunicera över TCP/IP. Det kan dock vara så att det finns bättre metoder än `net`, för Go var det enda språket där vi inte fick kommunikation mellan olika datorer att fungera, endast kommunikation på samma dator fungerade. Men detta kan även bero på operativsystemsskillnader(vi använde en Ubuntu och en Windows-maskin).

De första raderna av exempelkoden nedan är deklarationer av en `reader` som läser stdin och en `net.Conn` som senare håller i IP:n och porten. Metoden `con.Write()` skickar en sträng som skrivs in med `reader.ReadString()`, fast konverterad till bytes. Således läser `con.Read()` på samma port och tar emot data i form av bytes som skrivs till variablen `buff`. Variabeln `nr` som returneras av `con.Read()` är storleken på det som togs emot.

```
reader := bufio.NewReader(os.Stdin)
con net.Conn

conn, err := net.Dial("tcp4", "", "localhost:9999")

command, e := reader.ReadString('\n')
c.con.Write([]byte(command))

nr, err := con.Read(buff)
```

Det märks att även Go skickar bytelistor. Men det är mycket enklare att konvertera mellan de olika typerna i Go än i Erlang. Fel kastas också smidigt genom att de fångas hos mottagaren som kan välja om han vill ta hand om dem eller inte. Jämfört med Erlangs var Go lättare att få att fungera och arbeta med.

2.4 Erlang

Erlang är ett funktionellt programspråk som utvecklades på forskningsavdelningen hos Ericsson år 1987 för implementation av styrsystemen i telefonväxlar. Under de första åren användes det enbart internt av Ericsson.

1998 släppte Ericsson AXD301 växeln som innehöll över en miljon rader Erlang och hade en pålitlighet på 99.9999999% vilket motsvarar en tid offline per år på 0.03 sekunder [9]. Erlang har alltså potential för otrolig tillförlitlighet och stabilitet.

Språket är ett funktionellt programmeringsspråk, utan typdeklarationer, med dynamisk typning, som tillåter massiv parallellisering. 1998 släpptes Erlang som öppen källkod och det används av flera internationella företag samt undervisas på flera högskolor [10].

Erlang är bra på flera saker, men på grund av dess design fungerar Erlang bäst på en viss typ av system [21]. Dessa system inkluderar: Telekommunkationssystem, servrar till internetapplikationer och databashanteringssystem. Erlang passar bra för dessa system på grund av att språket från början designades med dessa typer av problem i åtanke, speciellt eftersom Erlang i början exlusivt användes i telefonväxlar.

2.4.1 Erlangs datatyper

Erlang har de vanliga typerna så som hel och flyt-tal, men det har även lite mer unika typer. Här kommer några exempel på andra typer som Erlang använder sig av.

Atomer är konstanter som alltid börjar med liten bokstav, annars tolkas de som variabler, om de inte innesluts av ‘ ’. Det finns inga boolska värden i Erlang, istället använder man atomerna ‘true’ och ‘false’.

Listor och **Tupler** liknar varandra och kan innehålla många olika typer. Listor innesluts av [ ], medans tupler innesluts av {}. Ex: [‘Atom’, 1, 2, 3, “String”, [‘lista’], Osv], {person, “Kalle Anka”, 22 }.

Strängar fanns inte med Erlang från början eftersom det inte fanns något behov av det i telefonvärlden, där Erlang föddes. Men stöd för strängar har lagts på efterhand, även om det inte är lika användarvänligt eller smidigt som i tex Java eller Python. Strängar är i Erlang listor med chars. [97, 98, 99] skrivs ut som “abc”. Men Erlang kommer skriva ut alla listor med siffror som strängar, om alla siffror i listan kan tolkas som bokstäver. Så fort en siffra finns som inte är en bokstav skrivs listan ut som en lista istället. [97,98,99,126] skrivs ut som: “abc~” medans [97,98,99,127] skrivs ut som: [97,98,99,127] eftersom ‘127’ är ett tecken som enligt Erlang inte ska vara med i en sträng.

2.4.2 Hot Swapping

Hot Swapping är en egenskap som gör att man kan byta ut eller lägga till delar av programmet under körning utan avbrott. Ett annat känt exempel på detta är USB, där du kan lägga till möss, tangentbort och skrivare eller annat utan att behöva starta om datorn. Det är inte så många språk som stödjer detta från början, vilket gör det till en intressant aspekt att undersöka.

När en modul byts ut till en nyare version med hjälp av Hot Swapping så läggs två versioner av programmet in i noden(mer om noder kommer längre ner). Den äldre versionen av modulen kommer att köras färdig och sen bytas ut mot den nya. Det går att tänka sig att referensen till modulen skrivs om till den nya. När den gamla modulen har kört klart kommer skräpsamlaren(garbage-collector) att se den som en process som har kört färdigt och ta hand om den.

2.4.3 Moduler

Moduler är kort sagt en samling av funktioner som ligger i samma fil. Man ropar på funktionerna i en specifik modul så här:
`Module:Function(Arguments).`
I moduler skriver du även metadata som till exempel:
`-module(calculator).`
`-export([add/2, subtract/2, multiply/2, divide/2]).`
`-module()` anger självklart modulens namn och är den enda metadatan som är ett krav. `-export()` gör de funktioner du anger tillgängliga för anrop(liknande public). Man kan även skriva `-import(modul, [funktion/1,...])` för att importera funktioner och moduler.

2.4.4 Lättare syntax

Pilsyntaxet "`->`" används på samma sätt som "`:=`" i prolog. På vänster sida skrivs funktionsnamnet med parametrar, och på höger sida läggs operationerna in.

```
-module(sample).
-export([add/2,add/3]).

add([],D) ->
    D;
add([X|X2],D) ->
    D2 = X + D,
    add(X2,D2).
add(X,Y,Z)-> Z+Y+X.
```

Punkt avslutar yttersta funktionen, om man har nästlade funktioner så avslutas inte den inre funktionen med punkt. Kommatecknet är som semikolon i Java, förutom när andra operationer används i slutet av en rad. Semikolon separerar funktioner som ska bete sig på olika sätt beroende på indata. Som i exemplet så kommer den

första `add` funktionen anropas om den första parametern är en tom lista, om det däremot ligger värden i listan samt funktionen avslutas med semikolon så kommer programmet att gå vidare till nästa `add` funktion och testar om parametrarna passar bättre. Om denna funktion avslutas med punkt så betyder det att inga fler av denna funktion finns. Funktioner med samma namn men med olika många parametrar tolkas som två olika funktioner. Alltså skulle kompilatorn klaga om exempelkoden ovan skulle ha ett semikolon istället för punkt på näst sista raden.

2.4.5 Processer

Erlang var som sagt från början tänkt att mest användas inom telefoni där varje samtal skulle vara en process som inte skulle krascha eller avbrytas om systemet fick fel och undergick underhåll. Det var därför fördelaktigt om processer kunde startas och stängas av mycket snabbt och inte ta upp så mycket minne. Processer delar inte heller minne, utan skickar data mellan sig, detta är långsammare men säkrare.

För att kunna skala upp program så mycket det går kan man antingen använda sig av bättre hårdvara, eller använda sig av mer hårdvara. Erlang valde mer hårdvara, och man kan enkelt sprida ut system på flera maskiner.

När man skapar nya processer så skapas dem i Erlangs virtuella maskin(EVM) som kan skapa en ny process på ett fåtal mikrosekunder. När processer skapas, skapas även några unika processer som kallas schemaläggare. Det skapas en sådan per kärna i datorn den virtuella maskinen körs på. Schemaläggaren har en lista med processer som den ska se till får tid att köra på kärnan, och när denna lista blir för lång så kan den flytta över processer till någon av de andra schemaläggarna(om de finns). Detta gör det mycket enkelt att programmera parallellt i Erlang, mycket av det svåra tas hand om språket själv.

För att kunna kommunicera mellan processer använder man tecknet '!'. Varje process har en inbyggt "brevlåda" där alla meddelanden de får hamnar i den ordning de togs emot. Om ett meddelande ska skickas till en process med PID 4711 till exempel skriver jag `4711 ! hello.` Där PID står för Process ID, ett unikt nummer för varje process som för tillfället körs.

Här kommer ett lite mer avancerat exempel från boken *Learn You Some Erlang for Great Good!* [28] som finns att läsa på internet.

```
-module(dolphins).
-compile(export_all).

dolphin() ->
  receive
    {From, do_a_flip} ->
      From ! "How about no?";
    {From, fish} ->
      From ! "So long and thanks for all the fish!";
    _ ->
      io:format("Heh, we're smarter than you humans.~n")
```

end .

Detta program tittar i sin "brevlåda" efter en tuppel med en PID och ett antal olika atomer(`do_a_flip` och `fish`). Om det tar emot `do_a_flip` skickar den tillbaka "How about no?" till det PID som skickade förfrågan. Om fish skrivs "So long and thanks for all the fish!". Annars skrivs "Heh [...]" ut lokalt för processen. För att starta programmet skrivs detta i Erlangs kommandotolk:

```
1> Dolphin = spawn(dolphins, dolphin).
2> Dolphin ! {self(), do_a_flip}.
3> flush().
```

`Spawn` kommandot säger vilken modul (dolphins) och vilken metod (dolphin) som ska köras som en process och sparar den nya processens PID som Dolphin. På nästa rad skickas föräldrarprocessens PID, som fås med funktionen `self()`, och `do_a_flip` till Dolpin. Sen skrivs processens "brevlåda" ut med kommandot `flush()`, och den innehåller "How about no?".

2.4.6 Tillstånd

Som de flesta funktionella språk så är tillstånd inte implementerat i Erlang. Däremot finns det stöd för tillstånd genom en modul i standardbiblioteket [24]. Modulen förutsätter att programmeraren implementerar vissa specifika funktioner vid användning av tillstånd, med andra ord är det ungefär som ett interface som måste följas. Den viktigaste funktionen är den som initierar starttillståndet för programmet. Tillstånden representeras som olika funktioner, biblioteksmodulen tar en godtycklig funktion som körs vid anrop. Eftersom funktionen som ligger i biblioteksmodulen kan bytas så kommer programmet att bete sig olika med samma anrop, alltså har programmet olika tillstånd. För att byta funktion som ligger i biblioteksmodulen så måste funktionen som representerar det nuvarande tillståndet returnera nästa tillstånd, det vill säga nästa funktion.

2.4.7 Nodes

Erlang är skapat för att programmera massiv parallellprogrammering. I varje körning kan man skapa ett stort antal processer(ofta 10 000 eller mer) som kan kommunicera med varandra [9]. Däremot så kan inte processer som körs av olika system kommunicera med varandra utan att använda sig av bibliotek för kommunikation så som TCP/IP. Vid tillfällen där man har många körningar på samma dator så kan det vara onödigt att använda sig av detta.

Varje körning av ett Erlang-shell är i själva verket en node (för enkelheten skull kallar vi det nod) som är anonym. För att kunna använda sig av flera noder som kommunicerar med varandra så måste de namnges. Noder kan jämföras med någon form av process som ligger hierarkiskt högre upp, och som kan skapa processer inom processen. Det främsta syftet med att ha både noder och processer är för att kunna

distribuera körningar över flera system. Till detta används noder. Om noden behöver använda sig av flera processer så skapar den helt enkelt nya processer istället för att köra en ny nod parallellt. För att skydda noderna från att bli kontaktade från utomstående program så finns det ett inbyggt system för att skydda kommunikationen mellan dem. Noderna måste jämföra en atom som vid detta sammanhang kallas för "magic cookie". Om nodernas "magic cookie" överensstämmer med varandra så får de kommunicera.

2.4.8 Ports

Ports är Erlangs verktyg för att kommunicera med externa komponenter, så som andra program skrivna i till exempel C. En Port skickar data som bytes, med andra ord så kan vad som helst skickas in så länge Erlangprogrammet kan tolka vad som skickas in. Programmet måste skapa en ny process som kör Porten. Eftersom Erlangprogram oftast ligger som *back-end* i systemet så borde programmets processer, vid nästan alla tillfällen, stängas av först när alla andra processer som ligger närmare användaren har slutat sin körning. Det är Erlangs roll som *back-end* språk som troligtvis är anledningen till att detta har implementerats i språket.

2.4.9 Anonyma funktioner

Anonyma funktioner är funktioner som man inte behöver namnge, det går dock att tilldela funktionen till en variabel för användning på andra ställen i koden. Framförallt används det för att skapa mindre funktioner i anrop till biblioteksmoduler som tar en funktion som parameter [22]. Vissa funktioner i biblioteket tar endast anonyma funktioner. Med andra ord är anonyma funktioner som lambdauttryck.

Syntaxet är således fun(inparameter) -> operation end.

```
add_one(L) -> map(fun(X) -> 1 + X end, L).
```

där `fun(X) -> 1 + X end` är den anonyma funktionen som är nästlad med en map-funktion som finns implementerat i språket.

2.4.10 Mnesia / databas

Erlang har stöd för implementation av en disk-baserad databas som går att köra i Erlang-shell. Databasen fungerar som en vanlig databas, dock har inte Erlangs databas stöd för någon form av SQL utan använder sig av en modul i biblioteket som heter Mnesia [12]. Databasen är i huvudsak gjord för att lagra mindre mängder med data, då det är väldigt lätt att sätta upp. För att kunna använda sig av databasen vid kodning måste en header-fil inkluderas, header-filen är nämligen en förenklad version av den databasstruktur som databasen använder sig av.

För att komma åt data från databasen måste koden använda sig av funktioner som ligger i Mnesia. Funktionen som gör transaktionerna tar anonyma funktioner `fun()`:s, som är skrivna med ett speciellt syntax för att passa till databasen. Med

andra ord laddar man in olika funktioner och anrop i en anonym funktion som vid transaktionen körs för att göra förändringar i databasen.

2.4.11 Nätverksprogrammering

Erlang har många olika sätt att kommunicera mellan alla sina delar, till exempel mellan program, processer och noder, som inte behöver vara bundna till en fysisk maskin. Den modul som används vid kommunikation över nätverk heter gen_tcp [19] och innehåller funktioner för program att kommunicera med varandra med hjälp av TCP/IP-protokollet.

Följande kod skapar en socket, skickar ut värdet «2» till localhost på port 9999, tar emot en handskakning från servern och till sist stänger socket.

```
{ok, Socket}=gen_tcp:connect("localhost",9999,
                             [{active,false},{packet,0}]),
gen_tcp:send(Socket,<<2>>),
{ok,_} = gen_tcp:recv(Socket,0),
gen_tcp:close(Socket).
```

I programmet pågår en del konvertering mellan strängar, heltal och bytes, och dessa funktioner är inte så lätta att hitta i Erlang, vilket ökar chansen för buggar. Speciellt när listor av bytes automatiskt skrivs ut som strängar. Paradoxallt nog krashar Erlang när man försöker skicka in en sträng till en funktion som tar byte listor och vice versa.

Kapitel 3

Metod, utförande

3.1 Metod

Eftersom vi inte kunde något av språken vid projektets början valde vi att lära oss språken genom att helt enkelt programmera i dem. På grund av att vi valde att programmera samma program, baserat på samma laboration, flera gånger om så blev det en naturlig följd att själva jämförelsen sker med fokus på programmen vi skrev.

Med det likartade programmen, som fokuserar på parallellprogrammering, kan vi på ett lättare sätt jämföra Go och Erlang på de aspekter av programmering de är bra på, samt på ett enkelt sätt jämföra med det redan färdiga Javaprogrammet.

3.2 Utförande

Vi började läsa på om Go för att så fort som möjligt kunna programmera i språket. Erlang lämnades till senare för att inte blanda ihop språkens syntax under programmering. Eftersom syftet var att skriva kod som använder sig av språkens styrkor samt unika egenskaper (parallellism, processer, funktioner, etc) så valde vi snabbt att modifiera kraven på programmet beroende på hurvida språket stödjer specifika funktioner eller inte.

3.2.1 Go

Som Java-kunnig programmerare var det inte överdrivet svårt att sätta sig in i Go. Det största problemet var troligtvis att språkets unga ålder får som konsekvens att det inte finns så mycket dokumentation samt exempel förutom de som Go's skapare själva bidragit med. Eftersom standard-biblioteket inte är uppbyggt med hjälp av arv, var det ibland svårt att hitta rätt paket för rätt ändamål. Därför som vid många andra liknande fall kom Google (sök) till stor nytta. När man väl börjar lära sig syntaxen var det inte så svårt att få lättare program att göra som man ville.

Programmeringen började med att få en enkel kommunikation mellan server och klient att fungera. Liknande program fanns båda som exempel på den officiella hemsidan samt andra sidor på internet [14]. Att få klientdatan att läsas in från fil, samt att spara nya klienter eller ändrad information tillbaka till filen var nästa steg i programutvecklingen. Parallellt jobbade vi på kod för servern så att den kunde godkänna eller avböja inloggningsförsök.

Vi sprang däremot ganska snart in i "problematiken" med att klasser inte existerar. Eftersom vårt program är så pass litet blir det inte oöverskådligt hur man än programmerar det. Däremot var inte vårt sätt att betrakta problemet optimalt för att programmera i Go. Det avspeglade sig i att vi slutligen hade ett program som var Java-laborationen, fast programmerat i Go.

Vid detta skede i programmeringen skulle vi kunnat göra en omstrukturering av de program vi redan hade, alltså använda oss av en struktur där vi skulle utnyttja språket på ett bättre sätt. Till exempel ha en separat process som läser och skriver till filen istället för att läsa in en gång och sen skriva när servern stängs ner, eller kanske använda oss mer av funktioner än av endast metoder som vårt program gör. En annan tanke var att dela upp nätverkskommunikationen helt och använda oss av olika processer som gjorde endast ena delen av kommunikationen, det vill säga en process för att ta emot data och en för att skicka.

Vad som istället gjordes var att implementera stöd för insättning och uttag från samma konto samtidigt. Med andra ord implementerades en form av anropskö. Det var extremt lätt att göra utifrån den kod vi hade och dessutom fick vi större inblick i ett av verktygen som Go tillhandahåller, men som inte Java gör, nämligen Channels.

3.2.2 Erlang

På grund av att Erlang är ett funktionellt språk som drar inspiration från språk som Prolog, var det en stor omställning att gå från Go till Erlang. Syntaxen och tankesättet är väldigt olikt för språken. Men efter att ha läst och kodat Erlang i ett antal dagar gick det bättre, och vi lärde oss hur man skulle tänka för att inte fastna i koden. Det fanns gott om biblioteks-moduler som var oss till nytta.

Som vid programmeringen i Go så började vi med att skriva kod för server/klient kommunikation. Det gick snabbt att få sådan kod att fungera med hjälp av exempel från Erlangs officiella hemsida [19]. Istället för fil-hantering bestämmde vi oss för att använda en implementation av en databas som Erlang har. Denna databas var biblioteksmodulen mnesia som fungerade utmärkt för våra ändamål.

Det första stora problemet vi stötte på var att det inte finns några tillstånd i Erlang, med andra ord så blir det problem med modellen att först logga in och sen göra transaktioner till banken. Efter en del efterforskningar om det specifika problemet kom vi fram till att Erlang-biblioteket har moduler för hantering av tillstånd. Däremot så märktes det snabbt att det fanns en anledning till att språket inte har stöd för tillstånd. Eftersom det inte är meningen att använda sig av tillstånd i ett funktionellt språk så är det riktigt bökigt. Vi tyckte att det inte var tidsmässigt värt att implementera det.

Det slutade med att vi inte implementerade tillstånd i vår kod och arbetade som om klienten redan hade autentiserat sig på något sätt utanför vårt Erlang program, vilket gjorde det mycket lättare att få programmet att fungera. Kravändrigen för programmet bidrog med andra ord till att vi kunde använda oss av Erlang på ett mer naturligt sätt.

Kapitel 4

Resultat

Programmet skrivet i Go stödjer inloggning av flera klienter på samma konto, en server som skickar de mesta av menytexterna istället för att skicka allting en gång, alltså en lättare variant av Java-programmet som vi utgick ifrån fast med en ny funktionalitet.

Med Erlang blev resultatet ett program som genomförde bank-transaktionerna i tre steg. Klienten som skickar kommandon över tcp/ip och tar emot data som servern skickar ut. Servern som tar emot data och anropar nästa modul som har alla funktioner med olika anrop till databasen. Däremot tillhandahåller inte programmet inloggningsfunktioner just för att Erlang är funktionellt och saknar bra stöd för tillstånd.

Koden till våra program går att hitta i Appendix-delen av uppsatsen eller på internet[26] [27].

Kapitel 5

Diskussion och Slutsatser

5.1 Allmänna förhinder

Vi fick inte så många förhinder som satte ordentliga stop i utvecklingen av de olika programmen. Ett av hindrena var tillstånd i Erlang, men det kunde vi komma runt genom att designa om programmets struktur och på så sätt slippa störas av det. Det andra som vi uppfattade som störande var bristen på exempelkod och bristfällig dokumentation av Go, men det är bara naturligt för ett så ungt språk.

Ett mindre problem värt att notera var att när vi läste in rader från skrivbordet i Go, fick vi med radavslutningstecken('\n'), och eftersom vi arbetade på olika operativsystem (Windows Vista och Ubuntu 10.04) fick vi olika många tecken beroende på vem som körde programmet. Detta beror på att i ett Windows-operativsystem är ny rad definerad som två bytes men i ett Linuxbaserat operativsystem är ny rad endast en byte. När vi i koden sedan skulle utföra operationer på strängen vi läst in fick vi en del buggar på grund av att strängarna inte var så långa som vi räknade med att de skulle vara.

5.2 Diskussion

5.2.1 Effektivt syntax

Ett av argumenten för att skapa Go var att språk som till exempel Java har ett komplicerat syntax i den bemärkelsen att programmeraren måste skriva mycket kod. Samtidigt är språket extremt litet medan standard-biblioteket bara växter. Framförallt bidrar det till att programmerare inte klarar sig utan vissa hjälpmedel. Det är extremt ansträngande att skriva Java-kod utan en ordentlig utvecklingsmiljö, som hjälper en med vilka metoder som finns var samt vad de tar som parametrar etc. Denna åsikt får medhåll av Pike [23].

Som tidigare nämnt blir Java-kod ofta väldigt komplex. Ofta beror det på att programmerare måste skriva stora kodbitar för liten funktionalitet, samt delar upp koden i många bitar. Det är inget fel på att dela upp kod men det finns en god-

tycklig gräns där koden blir mindre läslig bara för att koden ligger på många olika ställen. Eftersom det varierar från person till person kommer det alltid finnas programmerare som delar upp i så små delar som möjligt medan andra kommer lägga så mycket som möjligt på samma ställe.

I ett funktionellt språk går det oftast inte att göra saker på olika sätt så koden delas upp i funktioner som skapar en naturlig uppdelning oberoende på hur många filer som tillhandahålls. Eftersom Go's syntax är väldigt likt syntax för funktionella språk bildas en viss uppdelning. Dessutom är syntaxen i Erlang och Go sådan att koden blir mindre, detta bidrar med hjälp av Go's struktur att det oftast är lättare att läsa kod som är skriven i Go eller Erlang än i Java när det börjar bli stora volymer av kod. Eftersom en stor del av dagens programmerare använder mycket tid till att läsa andras kod är det viktigt att det ska vara lätt att läsa kod. Det finns visserligen en personlig skillnad på hur lätt det är att läsa vissa paradigmer men efter en längre tid med samma språk kommer skillnaden ligga i hur mycket kod, samt hur lätt det är att genom koden.

Fördelen med Erlang och Go är just att det är mindre kod, oftast endast ett bra sätt att programmera saker på, för de flesta fallen endast ett verktyg per funktionalitet, samt en struktur som inte är lika godtycklig som Java.

5.2.2 Parallellprogrammering

Denna aspekt är troligtvis den vi har spenderat mest tid på att analysera. I vårt bankprojekt används processer för att skicka data från flera klienter samtidigt som är oberoende. För just detta ändamål är stödet för processer eller trådar tillräcklig i alla språk. Om man däremot går in på hur man som programmerare måste utveckla koden så är skillnaden stor. I Go och Erlang är syntaxen och tankesättet i stort sätt likadant. Enda skillnaden är att Erlangs spawn måste ta en anonym funktion, medan Go kan anropa vilken funktion eller metod som helst.

Däremot i Java är det betydligt krångligare. Den klass som ska köras parallellt måste använda sig av ett interface som har en speciell metod vars syfte är att köras som en tråd. Med andra ord går det endast att skapa en sorts tråd per klass, vilket inte är ett hinder för Erlang eller Go då funktionen inte bryr sig det minsta om den kommer köras i en egen process eller inte. Det är mer smidigt att skapa processer i Go och Erlang då funktionerna som ska köras parallellt nödvändigtvis inte behöver vara menade för det i första hand. Då det inte är funktionerna som administrerar om de ska köras i en separat process eller inte, utan koden som anropar dem, behöver inte funktionerna explicit vara designade för att köras parallellt.

I Java måste man designa hela klassen för att kunna köra den som en egen tråd. Det kan finnas tillfällen när detta är bättre, då man i Go och Erlang kanske inte har anpassat alla funktioner för att köras som processer. Bara för att det går att köra vilken funktion som helst som en ny process så behöver det inte alltid vara bättre. Det som kan hända om en funktion, som inte designades för det, försöker köras som en process är att koden måste skrivas om, vilket kan bidra till att koden får flera lager av onödig komplexitet. I de flesta fallen bidrar den lättare syntaxen,

samt implementeringen av processer direkt i språket, till att Erlang och Go är de bättre språken på detta område. Den enda fördelen Java ger mot Go och Erlang är att man måste strukturera sitt program innan kodningen börjar, med andra ord något som borde göras oavsett språk.

Parallell körning av processer i Go och Erlang skiljer sig också från varandra. Troligtvis beror skillnaden på syftet med språken. I Go är det lätt att programmera parallellt, samtidigt som det går att koda snabbt. Processerna kan visserligen kommunicera med varandra men det är till exempel svårt att döda en speciell process utifrån utan att döda hela programmet. Erlang som har en annan struktur på hur processer fungerar då det finns implementation för flera noder att kommunicera med varandra, där de i sin tur tillhandahåller processer inom varje nod. Med andra ord är Erlangs processer till exempel mer användbart för större kluster av olika system, som kan kommunicera med varandra på ett säkert sätt, medan Go inte har stöd direkt i språket för sådana saker. Det mynnar helt enkelt ut i att Go utvecklades för att skriva enklare program snabbt medan Erlang med sina telefonväxlar är utvecklat för stabilitet. Parallell programmeringen är således mer komplex i Erlang men medför större funktionalitet.

5.2.3 Kommunikation mellan processer

Det går i alla språk att få processer att kommunicera med varandra. I våra program har vi dock endast använt oss av det i programmet som är skrivet i Go. Vi hade kunnat använt oss av det i Java men det var inget krav, i Erlang skulle det endast komplicera koden mer än nödvändigt. Vi har därmed inget konkret att jämföra med. För att få ett konkret exempel kollar vi på hur man skulle kunna implementera Go-programmet i de andra språken. I Erlang skulle vi få implementera en process för databas-transaktionerna som skulle få kommunicera med ett godtyckligt antal sockets. Med “receive” kan programmet köra ett anrop i taget. Java-programmet skulle troligtvis fungera på samma sätt. Däremot finns det ingen implementation för kommunikation mellan trådar direkt i språket. Med andra ord måste det implementeras egen kod som tar hand om kommunikationen. Som trådning i Java måste klassen använda sig av ett speciellt interface, eller det nyare biblioteket “Java.util.concurrent” som medför verktyg för trådar, alternativt använda sig av nyckelordet `Synchronized`.

Den största skillnaden mellan de olika språken gällande verktyg för kommunikation mellan processer är att i både Go och Erlang finns det stöd för kommunikation direkt i språken, medan Java måste använda sig av standard-biblioteket på ett eller annat sätt. Go har troligtvis det absolut simplaste sättet att kommunicera. Med channels går det lätt att lägga in och ta ut data från olika processer utan att få läsning av samma värde på flera ställen. Det går också att använda sig av detta för mer avancerade operationer.

I Erlang kan processer skicka saker till varandra, dock måste en process aktivt skicka till en annan process som aktivt måste ta emot. Det är helt enkelt ett annat sätt att tänka än Go’s channels. På samma sätt fungerar det för noder, däremot

sker kommunikationen med en viss säkerhet genom nodernas magic cookie.

Med Javas lösning går det visserligen att välja sätt att genomföra kommunikationen. Däremot måste programmeraren själv implementera de olika funktionaliteterna som krävs för att kommunikationen ska fungera. Klasser som varje programmerare själv måste implementera kan ge problem vid större projekt då olika programmerare har implementerat sin egen version av till exempel en channel klass, vilket leder till förvirring. Med andra ord är det mer jobb och troligtvis långsammare, dock går det att implementera egna funktionaliteter i klassen.

Sätten att kommunicera skiljer sig åt i alla språken där det egentligen inte finns något sätt som är bättre än ett annat. Största skillnaden är troligtvis att man som programmerare är mer van vid ett sätt att kommunicera och därför har lättare för att arbeta med det sättet.

5.2.4 Kompilator

I detta projekt använde vi oss av kompilatorer som tillhandahålls bland annat av de officiella hemsidorna. Go har en kompilator som är extremt sträng. Sådant som klassas som varningar i till exempel C++ ger kompileringsfel i Go. Hela konceptet varningar existerar inte, istället får man kompileringsfel för till exempel oanvända variabler samt oanvända paket som importerats. Varför kompilatorn är utvecklad på detta sätt har sin förklaring i att ett av Go's grundkoncept är att det ska gå snabbt att kompilera. Med andra ord är allt som inte används onödigt och tar längre tid att kompilera samt tar onödig plats i minnet vid körning. Samtidigt menar personer som Rob Pike att om kompilator hittar nåt att klaga på så är det värt att fixa felet: "We made a deliberate decision: no warnings. If it's worth complaining about, it's worth fixing. -rob" [30]. Eftersom Go till skillnad från Erlang inte stödjer kodbyte under körning samt ihoplänkning av program under körning måste programmet vara så perfekt som möjligt vid kompilering samt distribution.

Erlang kan däremot byta ut kod under körning samt tvingar inte koden att importera extern kod som den använder. Kompilatorn litar helt enkelt på att modulen som behövs är importerad när koden väl ska köras. Det leder till att kompilatorn inte fångar upp fel förutom de mest elementära felen så som syntaxfel. Varför kompilatorn är mindre strikt är troligtvis för att ihoplänkningen under körning bidrar till att kompilatorn inte kan tvinga en att definiera vilka externa moduler som ska köras. Egenskapen att kunna byta kod under körning bidrar också till lättare felsökning av kod samt lättare uppdatering av kod då det endast är nödvändigt att byta den del av koden som inte fungerar.

5.2.5 Spekulation om språkens framtid

Erlang

Erlang kommer nog fortfarande att användas som back-end i kombination med andra språk. Dessa språk borde vara bättre på att göra GUI:n och annat som

Erlang inte är bra på. Dessutom har Erlang stöd för kommunikation med andra språk som underlättar denna struktur.

När antalet kärnor i våra datorer stiger kommer behovet av ett språk som kan använda dessa kärnor att öka, och eftersom Erlang redan har stöd för multipla processer kommer dess popularitet att öka drastiskt [15].

En annan faktor som ligger till Erlangs fördel är att datorer kommer att bli vanligare och vanligare i våra liv. Erlang, som är bra på att arbeta med kluster av datorer, kommer att kunna utnyttja detta mycket bättre än många andra språk och kanske integreras med allt fler maskiner, inte nödvändigtvist datorer, i hemmet.

Go

Det är svårt att säga vad som kommer att hända med Go. Det finns en chans att det kommer att bli ett mycket använt språk som kanske inte tar samma plats som C++ eller någon av de större språken har nu men som ändå kan bli ett realistiskt alternativ.

Eftersom Go är relativt litet, snabbt och enkelt att lära sig, kan det komma att bli ett språk för utbildning om hur man använder processer och på ett bra sätt utnyttjar parallellprogrammering. De flesta utbildningar inom datalogi använder sig av språk som Java som första språk. Detta kan få till följd att programmerare kommer att välja bort Go på grund av sitt typsystem samt sina designval, till förmån för objektorienterade språk. Dock är det många som har stora förhoppningar för att Go ska bli nästa stora språk [16], för sitt alternativa sätt att strukturera koden [31].

Java

Java har i ganska många år varit ett poulärt språk att lära sig programmera i, om detta kommer att ändras till förmån för andra språk som Go eller kanske C# är svårt att säga. Java är inte öpenkällkod, till skillnad från Erlang och Go, utan ägs av Oracle, kan komma att hindra Javas utveckling. Som Javautvecklare är man i stor utsträckning bunden till hur Oracle tycker att Java ska vara, vilket många tycker är alldeles för restriktivt.

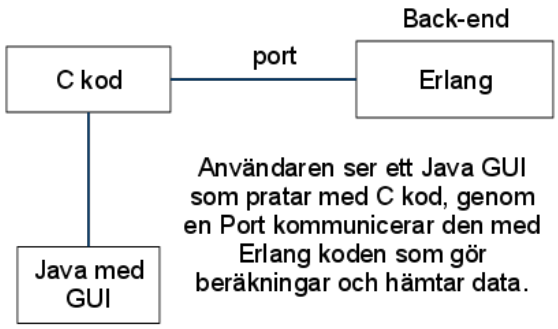
Men det finns en del som tror att Oracle kommer att fungera bra som ansvariga för Java och lyssna på utvecklarnas åsikter och förslag samtidigt som de fortsätter att utveckla och förbättra språket. Programmeringsspråket Scala [17] som bygger på Javas Virtuella Maskin(JVM), som också har bättre inbyggt stöd för multipla processer, kan kanske komma att ta över efter Java.

5.3 Slutsatser

Medan vi har programmerat i dessa språk har vi kommit fram till att deras egenskaper gör dem, helt naturligt, anpassade för olika ändamål.

Erlang, med sin massiva potential för parallellprogrammering och processer, passar mycket bra för projekt med många små uträkningar som kan spridas ut på flera

maskiner och processer.Vårt program med ett banksystem där varje klient loggar in som en egen process faller in i denna kategori. Däremot lämpar sig inte språket till programmering av användarnära program som till exempel grafiska gränssnitt, då stöd för till exempel tillstånd saknas i språket. Därför skulle vårt program bli bättre om klientsidan var helt eller delvis programmerad i ett annat språk. Eftersom Erlang har stöd för kommunikation med andra program skrivna i andra språk, skulle i alla fall delar av logiken kunna skrivas i Erlang. En mindre strikt kompilator och möjligheten att byta kod under körning gör Erlang till ett bra språk för kod som ligger på en server, då det möjligtvis är svårare att få koden rätt från början men är lättare att underhålla samt reparera.



Figur 5.1. En Model för hur Erlang kan användas i ett system.

Go, med sin spartanska och enkla syntax och snabbhet, passar bra på mindre projekt som man vill skriva snabbt utan att behöva offra kvalitet. Men på grund av Go's ålder och dess experimentella natur, kan det komma att ändra sig relativt snabbt. Go skapades av programmerare som ansåg att det tog för lång tid att programmera C++, men att det var för osäkert att skriva Python. Det går visserligen att skriva kodmässigt stora program men språket har en tendens att bidra med dålig strukturering av kod då strukturen skiljer sig från de mer etablerade språken. Om man är ovan kommer det att leda till att strukturskillnaden blir ett hinder för att programmera större program i Go. Det bästa tillfället skulle vara ett parallellt program som inte är alldeles för stort. Program som annars skulle programeras i Python går utan problem att skriva i Go, samt att skriva i Go bidrar med enkla hjälpmedel för att skapa processer som Python saknar. Den snabba kompileringen gör att det ännu mer konkurrens-kraftigt mot Python eftersom programmet kan kompileras istället för att köras som ett script och ändå skrivas snabbt och säkert. Det är som sagt inte omöjligt att skriva stora program men troligtvis är det bättre att använda till exempel C++ om programmet inte kräver processer samt processhantering. Med sin strängare kompilator är Go mer gynnsamt vid kodning av kod på klientsidan, som måste fungera när det lanseras.

Självklart går det att använda både Erlang och Go som allmäna programmeringsspråk när man jobbar med ett projekt som har nytta av processer, vilket det

är stor chans för idag, men ännu större chans för i framtiden. Detta kanske är extra sant för Go eftersom det är designat för att kunna hantera stor funktionalitet med en liten och enkel syntax.

5.4 Felkällor

En av de största felkällorna är att majoriteten av vår information och kodexempel kommer från olika sprida källor på internet som vi har dålig information på hur tillförlitliga de är. Till exempel programmerares personliga bloggar/hemsidor och guider(tutorials) från olika hemsidor.

Chansen att någon av dem skriver totalt motstridande eller felaktig information är kanske inte så stor, men det är något vi inte kan ignorera. Däremot är det större chans att vi inte har använt de bästa metoderna/funktionerna som finns i språket för att vi läste en artikel eller ett program från en programmerare som inte visste vad han gjorde, och vi antog att hur han/hon löste problemet var det enda (och det rätta) sättet att lösa det på.

Däremot kommer mycket av information från en offentlig och säker källa, nämligen de officiella hemsidorna för språken. Där står de mesta av informationen vi läst från andra sidor och även all dokumentation om språken.

Litteraturförteckning

[1] KTH CSC progp10. Lars Arvestad(2010). Officiell kurshemsida på KTH för programmeringsparadigm år 2010, http://csc.kth.se/utbildning/kth/kurser/DD1361/progp10, Besökt: 10 Feb 2011.

[2] The Multi-Core Dilemma. Patrick Leonard (2007). Ett blogginlägg om processorer med flera kärnor och hur man måste anpassa programmen efter detta: http://software.intel.com/en-us/blogs/2007/03/14/the-multi-core-dilemma-by-patrick-leonard/ , Besökt: 18 Feb 2011.

[3] Google Research Scientists and Engineers. sida om Rob Pike, grundaren av Go. http://research.google.com/people/r/index.html, Besökt: 9 April 2011.

[4] Another Go at Language Design. Rob Pike (2010). En presentation av Go som hölls på OSCON (Open Source CONvention) 2010, http://assets.en.oreilly.com/1/event/45/Another%20Go%20at%20Language%20Design%20Presentation.pdf, Besökt: 10 April 2011.

[5] Effective Go. Information om Channels på Go's officiella hemsida. http://golang.org/doc/effective_go.html#channels, Besökt: 28 Mars 2011.

[6] Newsqueak: A Language for Communicating with Mice. Rob Pike (1994). En uppsats om Newsqueak, ett annat språk som Pike inspererades av när han jobbade på Go, http://swtch.com/řsc/thread/newsqueak.pdf, Besökt: 2 April 2011.

[7] The Golang FAQ. Information om Generics på Go's officiella hemsida: http://golang.org/doc/go_faq.html?#generics, Besökt: 1 April 2011.

[8] The Golang FAQ. Information om Garbage Collection på Go's officiella hemsida: http://golang.org/doc/go_faq.html?#garbage_collection, Besökt: 1 April 2011.

[9] Concurrency Oriented Programming in Erlang. Joe Armstrong (2002). En vetenskaplig artikel som beskriver Erlangs styrkor i parallellprogrammering: http://ll2.ai.mit.edu/talks/armstrong.pdf, Besökt: 15 Mars 2011.

[10] The Erlang FAQ. En lista på företag som aktivt använder Erlang från deras officiella hemsida: http://www.erlang.org/faq/introduction.html#id49899 Besökt: 2 April 2011.

[11] Modern Operating Systems, 3/E. Andrew S. Tanenbaum(2007). ISBN: 0136006639

[12] The Erlang Mnesia Reference Manual. Officiell dokumentation om mnesia: http://www.erlang.org/doc/man/mnesia.html Besökt: 24 Mars 2011.

[13] Lab: Internet/sockets. Laborationen vi skulle göra i programmeringsparadigm (2010): http://www.csc.kth.se/utbildning/kth/kurser/DD1361/progp10/labbar/inet/sockets/ Besökt: 15 Feb 2011.

[14] Simpler chat server and client in golang. Ett blogginlägg som beskriver ett enkelt chat program i Go (2009): http://raycompstuff.blogspot.com/2009/12/simpler-chat-server-and-client-in.html Besökt: 20 Feb 2011.

[15] What's all this fuss about Erlang? Joe Armstrong (2007). En artikel om Erlang och hur indruktionen av många kärnor kommer att föra in Erlang i rampljuset: http://pragprog.com/articles/erlang Besökt: 28 Mars 2011.

[16] Interfaces vs Inheritance (or, watch out for Go!). Michele Simionato (2009). Ett blogginlägg om gränssnitt, arv och Go i allmänhet: http://www.artima.com/weblogs/viewpost.jsp?thread=274019 Besökt: 26 Feb 2011.

[17] Scala. Scalas officiella hemsida: http://www.scala-lang.org/ Besökt: 6 April 2011.

[18] Internet socket. Wikipedias sida om internet sockets, som används vid TCP/IP kommunikation: http://en.wikipedia.org/wiki/Internet_socket Besökt: 3 April 2011.

[19] The Erlang gen_tcp Reference Manual. Information om gen_tcp från Erlangs officiella hemsida: http://www.erlang.org/doc/man/gen_tcp.html, Besökt: 1 Mars 2011.

[20] Golang - Package net. Information om paketet net från Go's officiella dokumentation: http://golang.org/pkg/net/ Besökt: 4 April 2011.

[21] The Erlang FAQ. En kort beskrivning av vilka sorters program är bäst anpassade för att programmeras i Erlang: http://www.erlang.org/faq/introduction.html#id50265 Besökt: 8 April 2011.

[22] Erlang System Documentation. Officiella exempel från Erlangs dokumentation om anonyma funktioner: http://www.erlang.org/doc/programming_examples/funs.html Besökt: 11 April 2011.

[23] OSCON 2010: Rob Pike, Public Static Void". Rob Pike (2010). Ett övergripande tal om Go och vilken nisch Go siktar på att fylla, framfört på OSCON 2010: http://www.youtube.com/watch?v=5kj5ApnhPAE Besökt: 11 April 2011.

[24] The Erlang gen_fsm Reference Manual. Dokumentation från Erlangs officiella hemsida om gen_fsm, modulen som används för att skapa tillstånd: http://www.erlang.org/doc/man/gen_fsm.html Besökt: 12 April 2011.

[25] Eclipse. Eclipse, en utvecklingsmiljö för bland annat Java, officiella hemsida: http://www.eclipse.org/ Besökt: 12 April 2011.

[26] Dkand-Erlang. Andreas Starrsjö & Yuuki Jonsson(2011). Hela Erlang koden på Andreas GitHub sida: https://github.com/LURKEN/Dkand-Erlang Besökt: 13 April 2011.

[27] Dkand-Go. Andreas Starrsjö & Yuuki Jonsson(2011). Hela Go koden på Andreas GitHub sida: https://github.com/LURKEN/Dkand-Go Besökt: 13 April 2011.

[28] Learn You Some Erlang for Great Good! Frederic Trottier-Hebert (2009 - idag). En internetbok som lär ut grunderna i Erlang, mycket intressant och lärorik: http://learnyousomeerlang.com/ Besökt: 14 Feb 2011.

[29] Why Java Will Always Be Slower than C++. Dejan Jelovic(okänt datum, men senare än år 2000). En artikel om varför Java är långsammare än till exempel C++: http://www.jelovic.com/articles/why_java_is_slow.htm Besökt: 13 April 2011.

[30] For loop - underscore placeholder. En disskusion på Go's officiella forum om varningar och error i Go(2009): http://groups.google.com/group/golang-nuts/browse_thread/thread/a1b6e1f4093e2f34/9c447a46a3a45d76 Besökt: 13 April 2011.

[31] Effective Go - Interfaces. Go's officella dokumentation om Interfaces: http://golang.org/doc/effective_go.html#interfaces Besökt: 13 Mars 2011.

Kapitel 6

Appendix

.1 Appendix A - Go koden

Go koden består av fyra filer: main.go(som bara binder samman atm.go och filehandler.go), ATMClient.go, atm.go och fileHandler.go. Koden finns även att ladda ner på GitHub[27].

.1.1 main.go

```
package main
import(
   "./atm"
   "fmt"
   "./fileHandler"
   )
func main(){
   fmt.Println("Starting server")
   clients := fileHandler.DoStuff();
   i := clients[1].GetCardnr()
   print(i)
   t:= new(atm.Thread)
   t.ReadClients(clients)
   t.ListenForConnections()
}
```

.1.2 ATMClient.go

```
package main
import(
   "net"
```

```
  "fmt"
  "bufio"
  "os"
)
type Client struct{
  con net.Conn
  buff []byte
  reader *bufio.Reader
}
func main(){
  c := new(Client)
  c.run()
}

func (c *Client) run(){
  buff := make([]byte, 1000)
  c.buff = buff
  c.con = c.connect()
  reader := bufio.NewReader(os.Stdin)
  c.reader = reader
  c.login()
}

func (c *Client) connect() (conn net.Conn){
  conn, err := net.Dial("tcp4", "", "localhost:9999")
  if err != nil {fmt.Println(err)}
  return conn
}

func (c *Client) login() {
  for i:=0;i<3;i++{
    c.recieve()//ta emot card-text
    c.sendCommand()
    str := c.recieve2()//ta emot passwd-text
    if str == "error"{fmt.Println("FEL KORTNR!")
    }else{
      fmt.Println(str)
      c.sendCommand()
      //check if accepted
      msg := c.recieve2()
      c.con.Write([]byte("handskakning"))
      if msg == "loggedin" {//important!
        c.loggedIn()
      }else{fmt.Println("FEL KOD!")}
```

```
    }
  }
  fmt.Println("exit")
}

func (c *Client) loggedIn(){
  for{
    c.recieve()
    c.sendCommand()
  }
}

func (c *Client) sendCommand(){
  command, e := c.reader.ReadString('\n')
  if e != nil {print(e)}
  c.con.Write([]byte(command))
}

func (c *Client) recieve(){
  nr, err := c.con.Read(c.buff)
  if err != nil {print(err)}
  if nr > 0 {fmt.Println(string(c.buff[0:nr]))}
}

func (c *Client) recieve2() (string){
  nr, err := c.con.Read(c.buff)
  if err != nil {print(err)}
  if nr > 0 {return string(c.buff[0:nr])}
  return "error";
}
```

.1.3 atm.go

```
package atm
import(
  "net"
  "fmt"
  "os"
  "bufio"
  "strconv"
  "./fileHandler"
)
var clients []fileHandler.Client
```

```
type Thread struct{
  listener *net.TCPListener
  login chan int
}
type Cl struct{
  con net.Conn
  buf []byte
  client *fileHandler.Client
}
func (t *Thread) run(){}
func (t *Thread) ReadClients(cs []fileHandler.Client){
  clients = cs
}
func (t *Thread) ListenForConnections(){
  ip := net.ParseIP("127.0.0.1")
  addr := &net.TCPAddr{ip, 9999};
  listener, err := net.ListenTCP("tcp", addr);
  if err != nil {fmt.Println(err)}
  go t.Command()
  login := make(chan int, 1)
  t.login = login
  t.listener = listener
  t.login<-1
  for{
    <-t.login
    go t.AcceptClient()
  }
}
func (t *Thread) AcceptClient(){
  fmt.Println("Listening for clients")
  conn, _ := t.listener.AcceptTCP()
  t.login<-1
  c := new(Cl)
  c.con = conn
  buf := make([]byte,1024)
  c.buf = buf
  length := len(clients)
  for i := 0; i < 3 ; i++ { //tre forsok att logga in
    c.con.Write([]byte("Cardnr:"))
    size, _ := c.con.Read(c.buf)//waiting for cardnr
    size = size-1; // ta bort \n i slutet
    size = 4;
    if size > 0 {
      for j:=0;j<length;j++{
```

```
        tmp := clients[j].GetCardnr()
        tmp2, _ := strconv.Atoi(string(c.buf[0:size]))
        if tmp == tmp2 {
          fmt.Println("hittat kund, vantar pa passwd")
          c.con.Write([]byte("Code:"))
          nr2, err := c.con.Read(c.buf)//waiting for code
          nr2 = 1;
          if err != nil {print(" error")}
          if(nr2 > 0){
            code := clients[j].GetCode()
            code2, _ := strconv.Atoi(string(c.buf[0:(nr2)]))
            if code == code2 {c.loggedIn(j)}
          }
        }
      }
      fmt.Println("WRONG INPUT!")
    }
    c.con.Write([]byte("error"))//didnt find cardnr
  }
}

func (c *Cl) loggedIn(kundID int){
  fmt.Println("KUNDEN AR NU INLOGGAD!")
  c.con.Write([]byte("loggedin"))
  tmp,_ := c.con.Read(c.buf) //handskakning
  if tmp > 0 {}
  for {
    c.PrintMeny(kundID);
    size, _:= c.con.Read(c.buf)//waiting for input
    input, _ := strconv.Atoi(string(c.buf[0:(size-2)]))
    if input == 1 {
        c.con.Write([]byte("remove how much?"))//write
        nr, _ := c.con.Read(c.buf)//read
        antal, _ := strconv.Atoi(string(c.buf[0:nr-2]))
        c.removeMoney(kundID,antal) //write
    }
    if input == 2 {
        c.con.Write([]byte("insert how much?"))
        nr2, _ := c.con.Read(c.buf)
        antal, _ := strconv.Atoi(string(c.buf[0:(nr2-2)]))
        c.insertMoney(kundID,antal)
    }
    if input == 3  {
        fmt.Println("logout")
```

```
            c.con.Write([]byte("logout\n"))
            logout()
            break;
        }
        if input == 4 {
          fmt.Println("EXIT")
          c.con.Write([]byte("EXIT\n"))
          exit()
        }
    }
}
func (c *Cl) PrintMeny(kundID int){
    saldo:=<-clients[kundID].Saldo
    CardNr := strconv.Itoa(clients[kundID].GetCardnr())
    str:=strconv.Itoa(saldo)
    c.con.Write([]byte("CardNr: "+CardNr+" Saldo: "+str +
    " \n1. removeMoney\n2. insertMoney\n3. logout\n4. exit\n"))
    clients[kundID].Saldo<-saldo
}
func (c *Cl) printBalance(kundID int){
    saldo:=<-clients[kundID].Saldo
    str:=strconv.Itoa(saldo)
    c.con.Write([]byte(str))
    clients[kundID].Saldo<-saldo
}
func (c *Cl) removeMoney(kundId int, antal int){
    saldo := <- clients[kundId].Saldo
    clients[kundId].Saldo <- saldo - antal
}
func (c *Cl) insertMoney(kundId int, antal int){
    saldo :=<-clients[kundId].Saldo
    clients[kundId].Saldo<- saldo + antal
}

func exit(){os.Exit(0)}
func logout(){}
func (t *Thread) Command(){
    reader := bufio.NewReader(os.Stdin)
    command, _ := reader.ReadString('\n')
    fmt.Println(command)
    e := t.listener.Close()
    fileHandler.WriteToFile(clients);
    print(e)
    os.Exit(0)
```

```
}
```

.1.4 fileHandler.go

```
package fileHandler
import(
  "os"
  "fmt"
  "strings"
  "strconv"
)
type Client struct{
  cardnr int;
  Saldo chan int
  code int;
  name string;
}

var debug bool  = true;
var clients []Client

func DoStuff() []Client{
  clients := make([]Client,100000);
  clients = ReadWholeFile();
  return clients;
}
//reads clients.txt and inserts values into clients
func ReadWholeFile() []Client{
  var myFile *os.File;
  var myFileInfo *os.FileInfo;
  var myError os.Error;

  myFile, myError = os.Open("clients.txt",os.O_RDONLY,0)
  if myError != nil {fmt.Println(myError);}
  myFileInfo,_ = myFile.Stat();
  size := myFileInfo.Size;
  WholeFile := make([]byte,size);

  if debug {fmt.Println("Laser filen!");}
  i,e := myFile.Read(WholeFile);
  if e != nil {
    fmt.Println(i);
    fmt.Println(e);
```

```
  }
  WholeFileString := string(WholeFile[0:i]);
  rader := strings.Split(WholeFileString, "\n", -1)
  l := len(rader);
  clients := make([]Client,l);

  if debug {fmt.Println("Hittade dessa Clienter:");}

  for i := 0; i < l; i++{
    ord := strings.Split(rader[i], " ", 4)

    cardnr, _ := strconv.Atoi(ord[0])
    saldo, _ := strconv.Atoi(ord[1])
    code, _ := strconv.Atoi(ord[2])

    if debug {
      fmt.Printf("%v %v %v %v %v\n","    ",
      cardnr,saldo,code, ord[3]);
      }
    clients[i].cardnr = cardnr;
    s := make(chan int, 1)
    clients[i].Saldo = s
    clients[i].Saldo <- saldo;
    clients[i].code = code;
    clients[i].name = ord[3];
  }
  myFile.Close();

  return clients;
}

func WriteToFile(clients []Client){
    f, err := os.Open("clients.txt",
    os.O_RDWR | os.O_CREATE, 0666)
    if err != nil {return}
  if debug {fmt.Println("oppnat filen");}
    defer f.Close()
  l := len(clients);

  for i := 0; i < l; i++ {
    cardnr := strconv.Itoa(clients[i].cardnr)
    saldo := strconv.Itoa(<-clients[i].Saldo)
    code := strconv.Itoa(clients[i].code)
```

```
    _, err = f.Write([]byte(cardnr+" "))
    _, err = f.Write([]byte(saldo+" "))
    _, err = f.Write([]byte(code+" "))
    _, err = f.Write([]byte(clients[i].name))
    if i != l-1 { _, err = f.Write([]byte("\n")) }
  }
  if debug {fmt.Println("Skrivit till filen");}
  f.Close();
  if debug {fmt.Println("Stangt filen");}
}
func (c* Client) GetCardnr() int{ return c.cardnr }
func (c* Client) GetSaldo() chan int{ return c.Saldo }
func (c* Client) GetCode() int{ return c.code }
func (c* Client) GetName() string{ return c.name }
func (c* Client) SetSaldo(in int){ c.Saldo <- in; }
```

.2 Appendix B - Erlang koden

Även Erlang koden består av fyra filer: client.erl, server.erl och db.erl, som även inkluderar även en header-fil som heter bank.hrl. Koden finns även att ladda ner på GitHub[26].

.2.1 client.erl

```
-module(client).
-export([send/1,balance/1,deposit/2,withdraw/2]).

send(Message) ->
    {ok,Socket} = gen_tcp:connect("localhost",9999,
                  [{active,false},{packet,0}]),
    gen_tcp:send(Socket,Message),
     recive(Socket),
    gen_tcp:close(Socket).

balance(Who) ->
     io:fwrite("Balance:\n"),
    {ok,Socket} = gen_tcp:connect("localhost",9999,
              [{active,false},{packet,0}]),

     gen_tcp:send(Socket,<<1>>),
     {ok,_} = gen_tcp:recv(Socket,0),%%Hej!

     gen_tcp:send(Socket,Who),
```

```
    recive(Socket),

   gen_tcp:close(Socket).

deposit(Who,Val) ->
    io:fwrite("deposit:\n"),
   {ok,Socket} = gen_tcp:connect("localhost",9999,
                 [{active,false},{packet,0}]),

    gen_tcp:send(Socket,<<2>>),
    {ok,_} = gen_tcp:recv(Socket,0),%%Hej!

    gen_tcp:send(Socket,Who),
    recive(Socket),

    Val2 = lists:flatten(io_lib:format("~p", [Val])),

    gen_tcp:send(Socket,Val2),
    recive(Socket),

   gen_tcp:close(Socket).

withdraw(Who,Val) ->
    io:fwrite("withdraw\n"),
   {ok,Socket} = gen_tcp:connect("localhost",9999,
                   [{active,false},{packet,0}]),

    gen_tcp:send(Socket,<<3>>),
    {ok,_} = gen_tcp:recv(Socket,0), %%Hej!

    gen_tcp:send(Socket,Who),
    {ok,_} = gen_tcp:recv(Socket,0),    %%Hej!

    Val2 = lists:flatten(io_lib:format("~p", [Val])),

    gen_tcp:send(Socket,Val2),
    recive(Socket),

   gen_tcp:close(Socket).

recive(Socket) ->
    {ok,A} = gen_tcp:recv(Socket,0),
   io:fwrite(A++"\n").
```

.2.2 server.erl

```
-module(server).
-export([start/0]).
-import(db).

start() ->
    io:fwrite("Starting Server\n"),
    db:start(),
    io:fwrite("Starting Database\n"),
    listen().

listen() ->
    {ok, LSocket} = gen_tcp:listen(9999,
        [binary,{packet,0},{active,false},{reuseaddr, true}]),
    accept(LSocket).

accept(LSocket) ->
   {ok, Socket} = gen_tcp:accept(LSocket),
   spawn(fun() -> loop(Socket) end),
   accept(LSocket).

loop(Socket) ->
   case gen_tcp:recv(Socket, 0) of     %%Ta_emot_data
       {ok, Data} ->
            io:fwrite("\n"),
            io:fwrite(Data),     %%skriver_ut_data
            E= test(Data,Socket),
            E,
           loop(Socket);
       {error, closed} ->
           ok
   end.

test(<<1>>,Socket)->     %%Balance
    io:fwrite("Balance:"),

    gen_tcp:send(Socket, "Hej!"),
    {ok,Who} = gen_tcp:recv(Socket,0),

    Bal = db:balance(binary_to_list(Who)),
    io:fwrite("Balance: "++Bal++"\n"),
    gen_tcp:send(Socket, Bal);
```

```
test(<<2>>,Socket)->    %%InsertMoney
    io:fwrite("InsertMoney\n"),

    gen_tcp:send(Socket, "Vilken User:"),
    {ok,Who} = gen_tcp:recv(Socket,0),

    gen_tcp:send(Socket, "Hur mycket:"),
    {ok,Val} = gen_tcp:recv(Socket,0),

    Bal = db:deposit(list_to_integer(binary_to_list(Val)),
                     binary_to_list(Who)),
    gen_tcp:send(Socket, Bal);

test(<<3>>,Socket)->      %%RemoveMoney
    io:fwrite("RemoveMoney\n"),

    gen_tcp:send(Socket, "Hej!"),
    {ok,Who} = gen_tcp:recv(Socket,0),

    gen_tcp:send(Socket, "Hej!"),
    {ok,Val} = gen_tcp:recv(Socket,0),

    Bal = db:withdraw(list_to_integer(binary_to_list(Val)),
                      binary_to_list(Who)),
    gen_tcp:send(Socket, Bal);
test(<<4>>,Socket)->     %%logout
    io:fwrite("logout"),
    gen_tcp:send(Socket, "4"),
    close(Socket);

test(<<5>>,Socket)->     %%exit
    io:fwrite("5:"),
    gen_tcp:send(Socket, "5");

test(_,Socket) ->
    io:fwrite("Fel i kommando!"),
    gen_tcp:send(Socket, "Fel i kommando!").
```

.2.3 db.erl

```
-module(db).
-export([read/1,withdraw/2, deposit/2,balance/1,listAll/0]).
-export([start/0,stop/0]).
```

```
-include("bank.hrl").
balance(Who) ->
    FUN = fun() ->
        case mnesia:read({account,Who}) of
            []->
                ok;
            [E] ->
                E1 = E#account.balance,
                E1
            end
        end,
    {atomic, E}  = mnesia:transaction(FUN),
    lists:flatten(io_lib:format("~p",[E])).

read(Who)->
    FUN = fun()->
        case mnesia:read({account,Who}) of
            []->
                ok;
            [E] ->
                E3 = E#account.code,
                E3
            end
        end,
    mnesia:transaction(FUN).

deposit(Amount,Who)->
    FUN = fun()->
        case mnesia:read({account,Who})of
            []->
                ok;
            [E] ->
                Bal = E#account.balance,
                Nbal  = Amount + Bal,
                E1 = E#account{balance=Nbal},
                mnesia:write(E1),
                Nbal
            end
        end,
        {atomic, E}  = mnesia:transaction(FUN),
        lists:flatten(io_lib:format("~p",[E])).

withdraw(Amount,Who)->
    FUN = fun()->
```

```
        case mnesia:read({account,Who}) of
            []->
                ok;
            [E] ->
                Bal = E#account.balance,
                if
                    Bal >= Amount ->
                        Nbal  = Bal - Amount,
                        E1 = E#account{balance=Nbal},
                        mnesia:write(E1),
                        Nbal;
                    Bal < Amount ->
                        ok
                    end
            end,
        {atomic, E}  = mnesia:transaction(FUN),
        lists:flatten(io_lib:format("~p",[E])).

listAll()->
    FUN = fun()->
        case mnesia:dirty_all_keys(account) of
            [] ->
                ok;
            [E|_] ->
                bal = E#account.balance
            end
        end,
    mnesia:transaction(FUN).


stop()->
    mnesia:stop().

start()->
    mnesia:start().
```

.2.4 bank.hrl

bank.hrl inkluderas av db.erl och innehåller databas strukturen.

```
-record(account, {name,balance,code}).
```