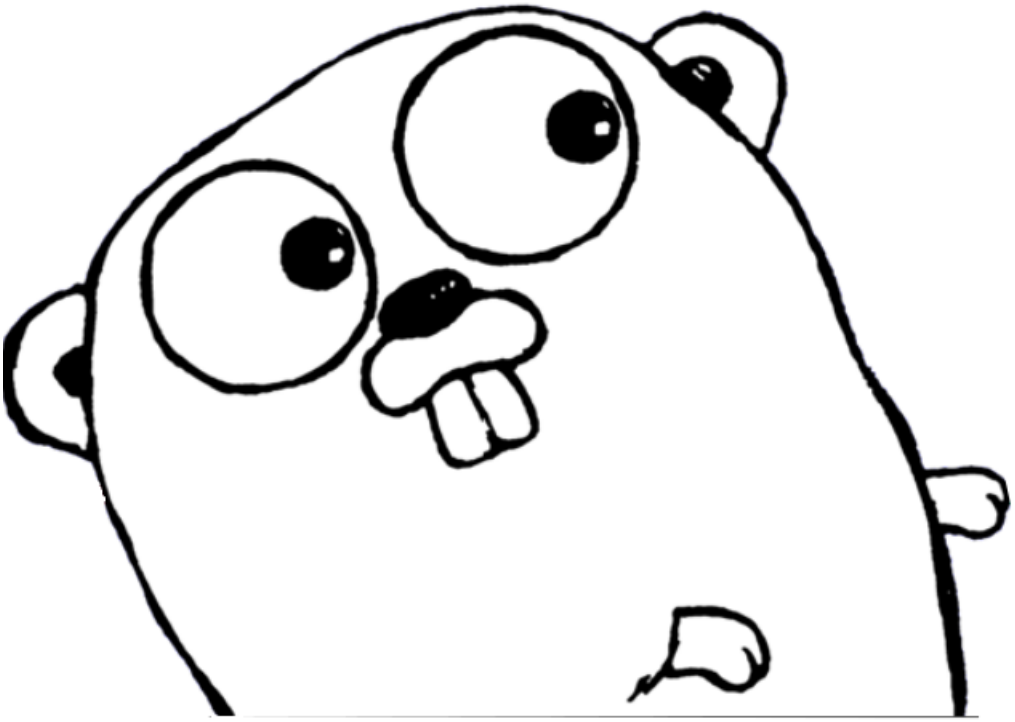
Kungliga Tekniska Högskolan

Go wiki

Niklas Peiper
Ringvägen 80
118 60 Stockholm
073-504 91 16
peiper@kth.se

Dara Reyahi
St. Göransgatan 152
112 17 Stockholm
070-799 49 56
darar@kth.se



Kandidatuppsats i DD143X dkand11
Handledare
Henrik Eriksson
Examinator
Mads Dam

Sammanfattning

Denna rapport ämnar att undersöka Googles nya programmeringsspråk Go. Språket är intressant framförallt för att det på ett unikt sätt implementerar parallellism inom programmering. Men också för dess omfattande bibliotek för webserver programmering.

Målet med detta arbete är att bilda en egen uppfattning av Go, dess för- och nackdelar och även om hur språkets framtid kan komma att se ut.

Språket jämförs med Java och C på olika punkter med hjälp av utförande av experiment. Dessa experiment visar prestanda skillnader i språken, stödet från de olika biblioteken och Gos övertag vid körning på en dator med en flerkärning processor.

Abstract

This raport aims to evaluate the new programing language from Google, Go. The most interesting thing about the language is foremost the unique way they have implemented a way for concurrent programing. But also thier extensive library for webserver programing.

The goal is to form an opinion of Go, its pros and cons aswell as the possible future of the language.

Experiments have been conducted to compare Go to Java and C on different aspects of thier language design. These experiments shows the difference in performence, library support and the advantage of Go on a multicored hardware.

Innehåll

1 Introduktion

1.1 Inledning

Den här rapporten skrivs som en del av kandidatexamensarbetet vid Datateknik på KTH. Deltagande i arbetet är Dara Reyahi och Niklas Peiper.

Programmeringsspråket Go släpptes av Google i november 2009 och hade då utvecklats internt i över två år av Robert Greisemer, Rob Pike och Ken Thompson[3]. Go ansågs då fortfarande vara i ett prototypstadie, men redan i Maj 2010 tillkännagav Rob Pike att det användes av Google "for real stuff"[2].

Go skapades för att försöka underlätta för utvecklare och "göra programmering roligt igen". Språket har en garbage collector och är designat för att kompileras snabbt, vara minimalistiskt och köra utan problem på mindre kraftfull hårdvara. Men det unika med Go är *goroutines*, goroutines fungerar som trådar som exekveras parallellt med andra goroutines, inklusive tråden de skapades i. Idén bakom detta är att Go ska ha hög *concurrency*, alltså att flera operationer kan köras samtidigt och interagera med varandra. Detta blir särskilt effektivt när man arbetar på processorer med flera kärnor eller flera separata processorer, något som inte var aktuellt när äldre språk(t ex C) skapades[1].

Go har sedan lansering dragit till sig mycket kritik, både positiv och negativ. Allt ifrån att Go kommer bli nästa stora lågnivåspråk till att det kommer att sopas under mattan. Det känns som att det enda sättet att få en opartisk åsikt om Go är att undersöka språket själv, och det är precis det vi ska göra.

Googles påståenden om Gos överlägsenhet är enligt Rob Pike: effektiv garbage collection, snabb och smart kompilering, hög concurrency och låg utvecklingstid[2]. Vi ska i vårt projekt evaluera och undersöka dessa påståenden.

1.2 Tillvägagångssätt

Denna uppsats kommer att evaluera och undersöka Go som en utmanare till de stora systemspråken. Vi ska undersöka om Go håller allt som Google lovar, vilka restriktioner språket har och inom vilket programmeringsområde språket kan utmärka sig i.

För att försöka få en så bred och objektiv bild av Go som möjligt så kommer vi att samla information via tre olika kanaler:

Vi kommer att ta del av information och kritik från artiklar, rapporter, Googles egna dokumentation och deras föreläsningar om språket. Språket har varit väldigt kontroversiellt och omtalat sen lanseringen så finns det redan gott om välargumenterade åsikter som vi kan ta hänsyn till.

Vi kommer också att skriva vår egna wiki i Go för att på så sätt få egen

erfarenhet om hur det är att använda språket i praktiken och hur det fungerar. Anledningen till att vi valde att göra en wiki är för att Google anser att Go ska utmärka sig särskilt vid serverside programmering[1].

Slutligen kommer vi att genomföra intervjuer med erfarna Go-programmerare. Vi kommer framförallt att fråga efter vad de tycker är bra och dåligt med Go, samt vad de tror om Go i framtiden.

Arbetet i projektet har delats upp på följande sätt: Dara Reyahi är huvudansvarig för programmeringsdelarna av arbetet och Niklas Peiper för studering av teoretiskt material och intervjuer. Övriga delar av arbetet har gjorts gemensamt.

2 Bakgrund

2.1 Googles motiv bakom Go

Under de senaste tio åren har datorer och sättet de används på förändrats enormt. Datorer är idag mycket snabbare och flerkärniga processorer har blivit standard. Däremot tar mjukvarukonstruktion fortfarande väldigt lång tid, kompilering av massiva program i C++ kan ta timmar, även om kompileringen sker på hundratals datorer samtidigt[4]. De gamla stora systemspråken (t ex C++, Java) har även svårt att implementera den parallella aspekten som krävs för att fullt utnyttja flerkärniga processorer[6].

Google anser att det krävs ett helt nytt systemspråk och tänk bakom detta språk för att förändra detta och de har därför utvecklat Go.

> *"There was a need for a fresh start. It wasn't enough to just add features to existing programming languages, because sometimes you can get more in the long run by taking things away."* - Rob Pike, 2009[5]

För att försöka lösa problemen med de gamla språken har Go utvecklats med åtanke med att det ska kompileras snabbt, kunna skrivas snabbt och fullt kunna utnyttja kraften av flera processorer[4].

2.2 Vad gör Go intressant

Go började som ett "20 percent project" (tid som ingenjörer på Google får för att jobba med sina egna hobbyprojekt och experiment) och plockades sedan upp som ett av Googles heltids-projekt[5]. Efter att ha utvecklats internt i över två år släppte Google språket under BSD-licensen med hopp om att locka till sig personer som kunde hjälpa till att utveckla det.

Språket kompileras direkt till maskinkod vilket gör att program skrivna i Go är väldigt snabba (Go ska enligt utvecklarna bara vara 10-20% långsammare än gcc C)[2]. Men trots detta så implementerar språket en Garbage Collector, säker typning och säkert minne (vilket i teorin borde eliminera risken för buffer overflows).

Under hela utvecklingstiden har ett av de största målen med språket varit att det ska hålla hög concurrency, vilket har lett till att språket idag har ett stort stöd för programmering på flerkärniga och nätverkade datorer.

Ett annat stort mål med språket är att det ska vara väldigt lätt för erfarna programmerare att sätta sig in i. Utvecklarna anser att C/C++/Java programmerare skall kunna utnyttja språket till fullo efter tre föreläsningar som Google har gjort om det som skiljer Go.

Men det som framförallt gör språket intressant är att det har en möjlighet att bli ett av de stora systemspråken och därmed konkurrera med bl a C/C++.

2.3 Vad utmärker Go

Trots att Go lånar många idéer från redan existerande språk så är Go ett helt nytt språk med unika egenskaper vilket gör att program skrivna i Go skiljer sig mycket från dem. Även om det i teorin skulle fungera att rakt översätta t ex ett Java eller C++ program till Go skulle det inte producera ett effektivt program. För detta krävs det först en förståelse för vad som utmärker Go.

2.3.1 Gos syntax

Gos syntax är C-lik men har ändrats enligt två principer. För det första ska syntaxen kännas enkel, utan för många obligatoriska nyckelord och med så lite repetition som möjligt.

Syntaxen ska även vara lätt att analysera, överblicka och kunna *parsas* utan en symboltabell. Detta gör det enkelt att bygga verktyg så som debuggers, IDEs etc för Go. Något som är onödigt svårt för C och andra språk som bygger på C[3].

Variabeldeklaration Gos syntax skiljer sig från C vid bl a vid variabeldeklaration. I Go sker deklarationen 'baklänges', detta underlättar t ex vid deklareringen av flera pekare. Skriver du exempelvis

```
int* a, b;
```

Listing 1: variabeldeklaration C

i C så kommer a att vara en pekare men inte b. I Go skriver man istället

```
var a,b *int
```

Listing 2: variabeldeklaration Go

vilket gör både a och b till pekare. Detta underlättar vid deklareringen av flera variabler samtidigt och gör det mer tydligt. Ändringar i variabeldeklarering har också gjorts för att minska onödig upprepning. Vid instansering av nya objekt blir detta tydligt, genom att skriva ':=' så deklareras variabeln automatiskt till den typ av objekt som skapats. Ex:

```
myT := new(T)
```

Listing 3: objekt-instansiering Go

Funktioner En av Gos unika egenskaper är att funktioner och metoder kan returnera flera värden. Detta har implementerats för att eliminera användandet av returvariabeln som både värde och fel-koll (Ex att returvärdet -1 innebär att det blev fel under funktionens körning)[3].

Som ett enkelt exempel har vi funktionen för att skriva till en fil. I Go ser signaturen för *File.Write ut på följande sätt:

```
func (file *File) Write(b []byte) (n int, err Error)
```

Listing 4: *File.Write signatur i Go

Funktionen returnerar både n antal bytes som skrevs och ett icke-nil Error *err* om n inte är lika med antal bytes b. I och med detta syns det enkelt hur många bytes man lyckades skriva och varför resten inte kunde skrivas, till skillnad mot write i C där ett negativt returvärde innebär att *någonting* gick fel och programmeraren får leta upp varför själv.

Ett annat intressant tillägg i Gos funktioner är nyckelordet *defer*. Med defer kan programmeraren köa-upp funktionsanrop till att köras precis innan funktionen returnerar. Detta är speciellt användbart i situationer där resurser ska frias oavsett vilken väg funktionen går och var den returnerar.

Detta demonstreras enklast genom ett enkelt kodexempel:

```
func Foo(filename string) (string, os.error) {
        f, err := os.Open(filename, os.O_RDONLY, 0)
        if err != nil {
                return "", err
        }
        defer f.Close() //f.Close kommer nu att köras
                        //när funktionen Foo är klar.

        //gör annat...

        if fel {
                return "", err //här körs f.Close
        }
        return string(result), nil //här körs f.close

}
```

Listing 5: Defer kodexempel i Go

Att använda defer på detta sätt har två stora fördelar:

För det första så garanterar det att programmeraren aldrig glömmer att stänga filen, ett misstag som lätt kan hända när funktionen uppdateras och fler returvägar läggs till.

För det andra så sätts anropet till close nära open vilket gör det mycket enklare att se att man verkligen stänger filen istället för att sätta anropen till close i slutet av funktionen, vid varje return[3].

2.3.2 Concurrency i Go

När man talar om concurreny inom programmering finns det vissa problem som man ofta stöter på, *deadlock* (två trådar som väntar på att respektive ska bli klar) och *starvation* (tråden får aldrig tillgång till nödvändig data och programmet kan inte avslutas)[7]. Det som är svårt är alltså att ge trådar rätt tillträde till delade variabler för att förhindra ovannämda problem. Gos lösning till detta är att följa deras motto:

> *"Do not communicate by sharing memory; instead, share memory by communicating."* - Go Team[6]

Detta görs genom att delade variabler skickas runt via *channels*, kanaler, och på så sätt delas aldrig variablerna av separata trådar. Endast en goroutine har tillträde till ett värde vid en viss tidpunkt i exekveringen, en design som hindrar både deadlock och starvation.

Denna model kan ses som en typningssäker generalisering av Unix pipes, och härstammar från *Hoare's Communicating Sequential Processes* (CSP)[6].

Goroutines Gos nyckel till hög concurrency är goroutines. De döptes till goroutines då de existerande termerna (*trådar*, *processer* etc) inte är applicerbar för att förklara hur de fungerar. En goroutine är billig (den kostar bara lite mer än allokering av stackminne) och exekveras parallellt med andra goroutines i samma minnesrymd[3].

Goroutines delar på olika trådar i operativsystemet så att om en skulle blockera i väntan på I/O så kommer andra att fortsätta köras.

För att skapa en goroutine och använda den för att köra en funktion används prefixet "go". När sedan funktionsanropet är klart kommer goroutinen att automatiskt avslutas. Ex:

```
go vari.LongCalculation()
```

Listing 6: Goroutines exempel

Här kommer vari.LongCalculation() att köras parallellt och tråden som utförde instruktionen kommer inte att vänta på den. Men för att den skapade goroutinen ska kunna kommunicera (för att t ex signalera om att den är klar) så behövs *channels*.

Channels Channels används för att kommunicera och synkronisera goroutines. De allokeras med make där man väljer vilken typ (strängar, integers etc) som kanalen kommunicerar med. Efter att en kanal skapats kan man skicka och ta emot signaler via den och en kanal som ska ta emot ett värde väntar alltid på att den ska få det. Kombinationen av channels och goroutines kan då lätt användas för att t ex skapa en server som tar förfrågningar från flera håll och exekverar dessa parallelt på olika kanaler. Exempel på hur man skapar och använder en kanal:

```
c := make(chan int)

go foo(){
        vari.LongCalculation()
        //Skicka värde till kanalen
        //(signalera att search är klar)
        c <- 1
        }

//Väntar på att c ska få ett värde
<-c
```

Listing 7: Channels exempel

Då en kanal är en typ kan även dessa användas som vilken annan typ som helst. Det gör att man kan skicka kanaler via kanaler för att t ex implementera säkert och parallell demultiplexing[3].

Den parallela aspekten med kanaler kan även utnyttjas för att utföra operationer över flera CPU kärnor. Om man t ex vill utföra en operation på värden i ett lista skapar man en goroutine som kör den önskade funktionen och som skickar en signal till en gemensam kanal när den är klar. Kanalen kan sedan vänta på att få värden från alla goroutines (alla operationer är klara). De individuella operationerna körs parallelt och behöver inte avslutas i någon specifik ordning.

3 Utförande

3.1 Att komma igång med Go

Kompilatorer Det finns för närvarande två olika kompilatorer för Go. Den första kompilatorn, *Gc*, skrevs i C och använder *yacc/bison* för parsern. Google ville från början skriva den i Go men valde att inte göra så då man (ifall man själv vill kompilera kompilatorn) skulle behöva en annan Go kompilator[4].

Den andra kompilatorn *gccgo* skrevs av Ian Lance Taylor som en frontend till gcc. Den enda riktiga anledningen till att Ian skrev gccgo var för att han tyckte att språket verkade intressant och han ville bidra till projektet[8].

Gc kompilerar märkbart snabbare men producerar mindre effektiv kod än gccgo. Gccgo använder för närvarande dock en OS-tråd för varje goroutine något som kommer ändras i framtiden[9]. Fördelen med att ha två kompilatorer tidigt i utvecklingstadiet av ett språk är att det blir lätt att upptäcka luddigheter i språket.

Dokumentation Google har varit noga med att hålla dokumentationen av språket uppdaterat under utvecklingen och har trots att språket är så ungt en väldigt omfattande dokumentation. De har lagt upp flera föreläsningar och guider om språket, främst riktade till erfarna programerare för att de lätt ska kunna föra över sina kunskaper till Go. Förutom detta finns flertal olika programexempel speciellt framtagna för att demonstrera unika egenskaperna med Go. All dokumentation finns på *Golang.org* där man även själv kan bidra till utvecklingen av språket.

3.2 Jämförelser med andra språk

För att kunna bilda en uppfattning om ett språks styrkor och svagheter är en nödvändig del att jämföra dess skillander med andra språk. Enligt Google kommer Go bli ett stort systemspråk och vi väljer därför att jämföra det med två av de idag populäraste, nämligen C och Java.

Vi kommer att undersöka körtid, kompileringstid, syntax, rader kod, implementeringstid, felhantering och utvecklingsmiljöer. För att kunna undersöka detta så kommer vi att köra olika experimentprogram i språken där skillnaderna blir tydliga och relevant experimentdata kan utvinnas.

I första experimentet jämförs Quicksort i de tre olika språken. Genom att implementera denna algoritm kommer skillnader i framförallt körtid att noga kunna testas.

I det andra experimentet kommer en enkel webserver att implementeras i språken. Detta experiment kommer främst att undersöka hur enkel en sådan

implementering är att göra, vilket inbyggt stöd (i form av bibliotek) det finns och prestandan hos denna webserver. För att undersöka prestandan kommer programmet *ApacheBench* att köras mot de olika servrarna.

3.2.1 Quicksort

Quicksort är en sorteringsalgoritm som utvecklades år 1960 av C. A. R. Hoare under ett besök i Moskva. Han jobbade då på ett översättningsprojekt och algoritmen utvecklades för att sortera orden som skulle översättas så att de enklare kunde jämföras med en existerande databas[10].

Algoritmen använder en så kallad ”divide and conquer” strategi där varje lista delas in i två sub-listor. Sorteringen sker enligt följande steg:

1. Välj ut ett element, *pivot*, ur listan.

2. Ordna om listan så att element med lägre värde än pivot ligger innan och värden med högre värde än pivot ligger efter. Efter denna partionering befinner sig pivot-elementet i sin slutgiltiga position.

3. Sortera rekursivt sub-listorna framför och efter pivot. Detta kommer att ske tills det att listorna är 1 eller 0 element långa varvid sorteringen är klar.

```
function Quicksort(array)
        var list, less, greater
        if length(array) <= 1
                return array    //en lista med 0
                                //eller 1 element
                                //är redan sorterad
        //välj och ta bort pivot element från array
        for each x in array
                if x <= pivot then append x to less
                else append x to greater
        return  conca(quicksort(less), pivot,
                quicksort(greater))
```

Listing 8: Pseudokod för Quicksort

Implementeringen av Quicksort i C gjordes enligt pseudokoden ovan och finns i appendix [A.1.1 C]. För implementeringen i Java används Arrays.Sort som är en Quicksort med några optimeringar[11]. I Go skrevs två quicksorts, en utan användning av goroutines och en annan där första partioneringen delades upp på två processorkärnor.

```
func quicksort(a []int, lo int, hi int) {
        i,j := lo,hi
        var x int
        x = a[(lo+hi)/2] //x är pivot

        //ordna om listan med pivot

        if lo < j { quicksort(a, lo, j)}
        if i < hi { quicksort(a, i, hi)}
}
```

Listing 9: Quicksort i Go

Quicksort experimentet kommer att ge en rättvis bedömning över de olika språkens körtid. Då testmiljön, testdatan och algoritmen är lika för de tre språken så kommer skillnader i körtid endast att bero på språkets effektivitet.

Testerna kommer att utföras med tre listor innehållande en, två och fyra miljoner slumpmässigt genererade element. Dessa kommer att sorteras 50 gånger av varje språk för att få fram ett genomsnittsvärde.

Hypotes Go har en stor fördel i detta test då den kan utnyttja testdatorns båda processorkärnor. Detta leder oss till hypotesen att implementeringen i Go med stor sannolikhet kommer att vara den snabbaste av de tre.

Vidare tror vi att implementeringen i Java kommer att vara betydligt långsammare än de två andra då denna kommer att köras i Javas virtuella maskin, JVM.

3.2.2 Webserver

En webserver visar websidor på begäran av en klient, oftast en webläsare. Websidorna innehåller HTML dokument och andra element som tillhör sidan till exempel bilder och JavaScript. Även om den grundläggande funktionen hos en webserver är att skicka data så behöver en fullständig webserver även kunna ta emot data, via t ex webformulär och uppladning av filer.

För detta experiment implementerades en enkel webserver i de tre språken som lyssnar efter inkommande anslutningar och då visar en websida. Utöver detta saknar servern någon annan funktionalitet.

Implementeringen av webservern i Go sker enkelt efter importeringen av ”http”-paketet.

```
package main
```

```
import (
        "fmt"
        "http"
)

func handler(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintf(w, "hello world!")
}

func main() {
        http.HandleFunc("/", handler)
        http.ListenAndServe(":8080", nil)
```

Listing 10: Enkel webserver i Go

Programmets main-funktion börjar med att anropa http.HandleFunc, som deklarerar att alla anrop till webserverns root ("/") ska hanteras av funktionen handler.

Den kallar sedan på http.ListenAndServe som lyssnar på porten den fått som inparameter och hanterar inkomna anslutningar.

Webservern i Java skrevs på liknande sätt och finns dokumenterad i appendix [A.2.1 Java]. Då en webserver i C måste skrivas från grunden utan något biblioteksstöd valde vi att testa våra Java och Go implementationer mot *nginx*, en resurssnål och effektiv webserver skriven i C.

I detta experiment kommer flera olika aspekter av språken att jämföras.

För det första kommer en jämförelse i själva implementationen att ske, dvs vilket stöd det finns för webservrar i språket och hur enkel den är att skriva.

Servrarnas prestanda kommer också att analyseras med hjälp av ApacheBench, ett program från Apache som skapades specifikt för att testa prestandan hos en webserver. Analysen kommer att ske genom att servrarna anropas en miljon gånger för att samla in data om hur många anrop servern kan svara i sekunden och vilken responstid dessa anrop har. Detta test kommer att köra i flera omgångar på varje server där antal samtidiga anrop ökas.

Hypotes Då Java och Go har stora väldokumenterade bibliotek för implementering av webservrar tror vi att servrarna i dessa språk kommer vara enkla att skriva.

Prestandamässigt tror vi att Go definitivt kommer att hålla högre effektivitet än Java, framförallt då vi tror att JVM är en stor nackdel för Java.

Däremot är vi osäkra på om Go eller C kommer att visa sig vara mest

effektiv, då vi inte explicit programmerat för flera processorer (goroutines) så tappar Go sin stora fördel.

3.3 Go Wiki

Trots att Go fortfarande är ett relativt nytt språk så har det ett stort och väldokumenterat bibliotek för server och webapplikations programmering. Detta är ett område där Google vill att Go ska vara effektivt och där de tror att Go har sin framtid, värt att notera är att språkets officiella sida *golang.org* har körts från en server skriven i Go sen det släpptes. Det ter sig då självklart att vidare undersöka Gos funktionalitet inom detta område då denna rapport ämnar att undersöka Gos framtida potential. För att göra detta ska Go-webservern som skrevs för experiment [3.2.2 Webserver] byggas på med fler funktioner så att den fungerar som en enkel wiki.

3.3.1 Wiki

En wiki är en typ av webserver som låter användare skapa websidor och editera innehållet i dem med hjälp av en webläsare. Detta är användbart i arbeten där ett sammarbete sker mellan många personer då alla har tillgång att se och ändra sidor. Den mest kända wikin är *wikipedia*, en encyclopedia som under flera år utvecklats till en av de mest besökta sidorna på nätet.

För att säkra en wiki mot vandalisering kan man implementera användarroller, logga ändringar och spara historik för artiklar. Detta kan också användas för att öka trovärdigheten hos wikin då användare kan se vem som bidrigit med vad, samt att krav på referencer kan ställas vid ändringar.

Ytterligare funktionallitet som kan implementeras är en sökfunktion för wikins artiklar, översättning till andra språk, direktlänkning mellan artiklar och diskussionssidor.

3.3.2 Skapandet av wikin

För att skapa wikin ska Go-webservern från [3.2.2 Webserver] byggas på med flera funktioner:

- Varje artikel i wikin ska ha en bestämd struktur som definierar hur varje sida sparas på servern.
- En metod måste skrivas till wikin som tar en sådan struktur som inparameter och sparar den som en fil på servern. Denna metod kommer att användas vid ändringar och skapandet av nya artiklar.
- På samma sätt måste en metod skrivas som läser innehållet i en sådan fil och returnerar artikeln för att visas.

- I kodexemplet för Go-webservern [Listing 10] deklareras i mainfunktionen att alla anrop till roten av webservern ska hanteras av funktionen handle. Wikin kommer att behöva hantera anrop olika beroende på om artikeln ska visas eller ändras.

- Metoden som ska hantera visningar behöver endast använda sig av läsmetoden och sedan skriva ut artikeln till klienten. Däremot kommer metoden som ska hantera ändringar att först kunna ta emot information från klienten för att sedan spara den.

- Genom att använda Gos template-bibliotek kan HTML-kod sparas i separata filer utanför källkoden för wikin. Denna separation gör det enkelt att ändra artiklarnas generella utseende utan behöva ändra i serverns källkod.

- Annan funktionalitet som kan läggas till är en enkel sökfunktion för att leta upp artiklar, ett system för att lagra artikelhistorik och direktlänkning mellan artiklar.

3.4 Intervju

En erfararen programmmerare kan ge en unik inblick av språket som våra expriment och undersökningar inte kan ge. Genom att hålla en intervju hoppas vi ta del av programmerarens perspektiv på språket.

Personen vi har valt att intervjua är Stefan Nilsson, universitetslektor på Kungliga Tekniska Högskolan i Stockholm, erfaren programmerare.

Frågor vi hoppas få svar på är vad Stefan tror om Gos framtid, programerings område där Go kan bli starkt och hans personliga åsikt om språket som helhet.

4 Resultat

4.1 Quicksort

Nedan finns resultaten från Quicksort-experimentet listade i tabellform och i ett jämförande diagram. Dessa tester kördes på samma hårdvara och genomsnittsvärdet är ett genomsnitt av 50 körningar.

Go-implementeringen kördes både på en kärna och parallellt på två kärnor, där uppdelningen skedde i första partitioneringen så att de båda listorna sorterades samtidigt.

C

Antal element	Min (ms)	Max (ms)	Genomsnitt (ms)
$1 * 10^6$	313,364	315,944	314,636
$2 * 10^6$	662,359	664,013	662,747
$4 * 10^6$	1384,988	1390,131	1385,726

Java

Antal element	Min (ms)	Max (ms)	Genomsnitt (ms)
$1 * 10^6$	968,103	980,096	973,843
$2 * 10^6$	2138,477	2168,953	2148,617
$4 * 10^6$	4579,850	4771,823	4645,412

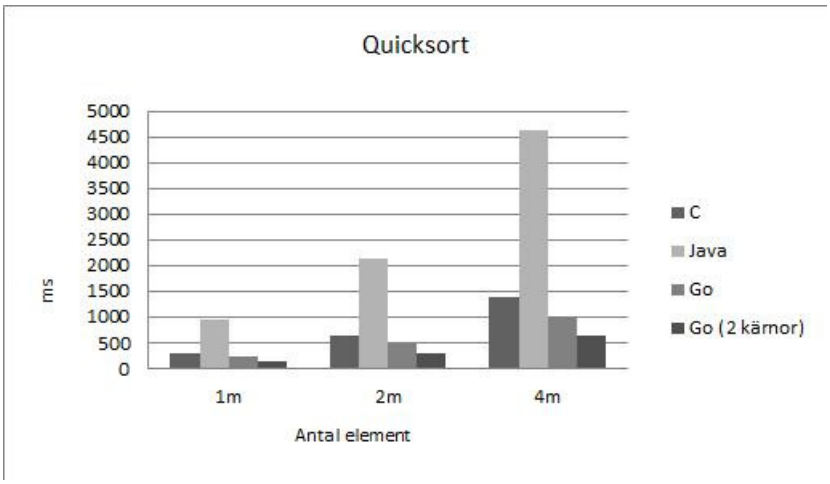
Go

Antal element	Min (ms)	Max (ms)	Genomsnitt (ms)
$1 * 10^6$	245,502	251,597	248,107
$2 * 10^6$	494,908	530,544	502,910
$4 * 10^6$	1009,153	1049,753	1024,938

Go (2 kärnor)

Antal element	Min (ms)	Max (ms)	Genomsnitt (ms)
$1 * 10^6$	188,674	195,825	192,283
$2 * 10^6$	352,193	413,622	388,937
$4 * 10^6$	760,237	853,971	797,472

Sammanställning I detta diagram presenteras genomsnittsvärderna för alla körningar.



Figur 1: Quicksort

4.2 Webserver

Nedan finns resultaten från webserver-experimentet, de har tagits fram med hjälp av ApacheBench vid lokal körning. Vid varje test har en miljon anrop skett till servern med olika mängder samtidiga trådar. Resultaten som samlats in är antal svar per sekund, överföringshastighet och responstid.

Vid 75 respektive 150 samtidiga trådar slutade Java- och Go-servern att svara.

C

Trådar	Svar/sek	Överföringshastighet (Kbytes/sek)	99% av anrop besvarade inom (ms)
25	19 757	6154	2
50	19 364	6032	4
75	19 173	5973	5
100	18 780	5850	7
125	18 560	5782	9
150	18 289	5697	11
500	16 284	5073	22

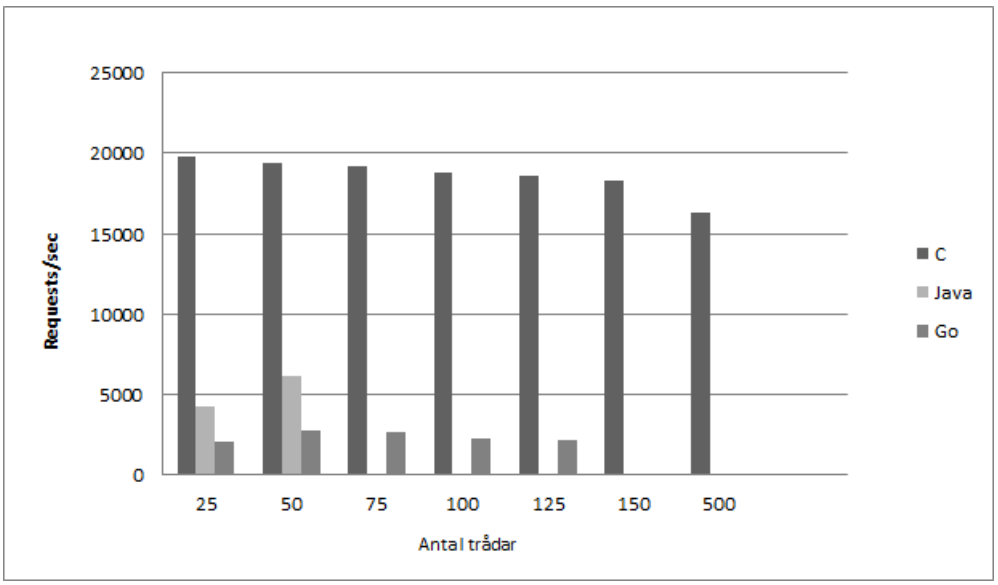
Java

Trådar	Svar/sek	Överföringshastighet (Kbytes/sek)	99% av anrop besvarade inom (ms)
25	4216	551	12
50	6104	828	17
75	N/A	N/A	N/A
100	N/A	N/A	N/A
125	N/A	N/A	N/A
150	N/A	N/A	N/A
500	N/A	N/A	N/A

Go

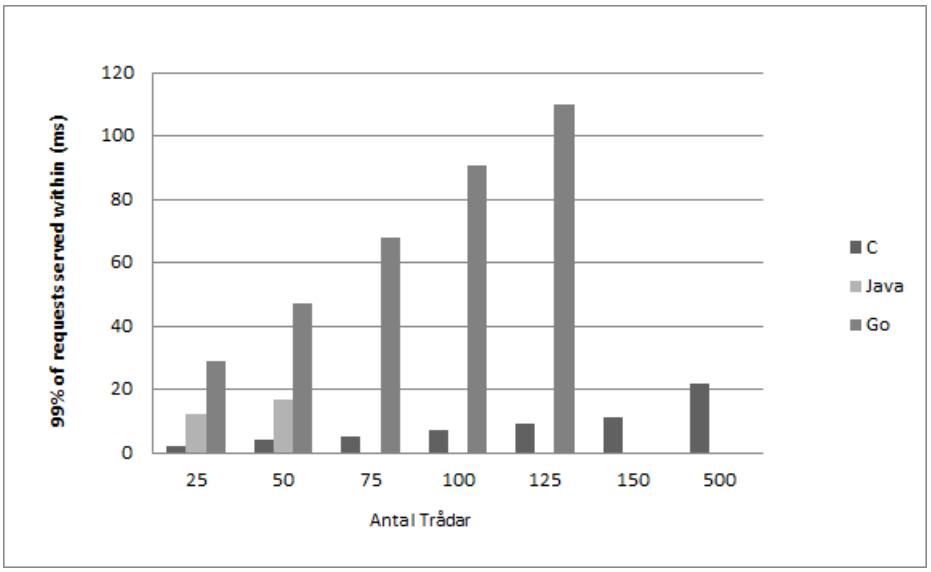
Trådar	Svar/sek	Överföringshastighet (Kbytes/sek)	99% av anrop besvarade inom (ms)
25	2013	241	29
50	2752	330	47
75	2633	316	68
100	2302	276	98
125	2123	255	110
150	N/A	N/A	N/A
500	N/A	N/A	N/A

Sammanställning I detta diagram visas en jämförelse av antal svar/sekund som de olika servrarna skickar ut beroende på antal samtidga anrop.



Figur 2: Svar/sek

Här presenteras en jämförelse av servrarnas olika responstid beroende på antal samtidiga anrop.



Figur 3: Responstid

4.3 Go Wiki

I [3.3.2 Skapandet av wikin] beskrevs de funktioner som en enkel wiki behöver. En fullständig redovisning av vår wiki finns i appendix [A.3 Wiki]. Nedan förklaras hur dessa funktioner implementerats.

Varje artikel på wikin är definierad i servern som en enkel *struct* med en sträng som titel och en byte-array för dens innehåll. Till denna struct skapades en sparfunktion som sparar artikeln på servern. Denna funktion sparar också historiken för artiklarna.

```
func (p *Page) save() os.Error {
        oldVer, err := loadPage(p.Title)

        filename := p.Title + ".txt"
        tid := time.LocalTime().String()

        if err != nil {
        }else {
                filenameHis := "history/"+p.Title+tid+".txt"
                ioutil.WriteFile(filenameHis, oldVer.Body, 0600)
        }
        return ioutil.WriteFile(filename, p.Body, 0600)
}
```

Listing 11: *Page.Save()

För att läsa in dessa sidor från hårddisken skrevs en loadfunktion som tar en titeln för artikeln som inparameter och returnera en page-struct med artikelns data och ett Error som är nil om läsningen gick bra.

```
func loadPage(title string) (*Page, os.Error) {
        filename := title + ".txt"
        body, err := ioutil.ReadFile(filename)
        if err != nil {
                return nil, err
        }
        return &Page{Title: title, Body: body}, nil
    }
```

Listing 12: loadPage

Anropen till servern hanteras olika beroende på om servern ska visa eller ändra en artikel. Detta implementeras lätt med Gos ”http”-bibliotek genom att skapa *handlers* med olika funktioner beroende på vilken address klienten anropar. Dessa deklareras i main-metoden innan http.ListenAndServe anropas, som lyssnar på inkommande anslutningar.

```
func main() {
        //HandleFuncs första parameter är en relativ
        //address från server-rooten
        http.HandleFunc("/view/", makeHandler(viewHandler))
        http.HandleFunc("/edit/", makeHandler(editHandler))
        http.HandleFunc("/save/", makeHandler(saveHandler))
        http.ListenAndServe(":8080", nil)
}
```

Listing 13: Handlers

ViewHandler och editHandler använder sig av templates för att generera sidorna i wikin för att skilja HTML-koden från källkoden till servern. Dessa finns lagrade på hårddisken och kan ändras utan att påverka servern.

För att användare av wikin enkelt ska kunna direktlänka mellan olika artiklar översätts ”[PageName]” till HTML-länkar till det angivna namnet. Detta sker i sparningen av en sidan med hjälp av regex.ReplaceAllStringFunc.

```
func saveHandler(w http.ResponseWriter,
                 r *http.Request, title string) {
        body := r.FormValue("body")
        re,_ := regexp.Compile("\\[[a-zA-Z]+\\]")
```

```
        //ReplaceAll kommer returnera en kopia av body där
        //alla matchningar av regexet re byts ut mot
        //return värdet av repl (som kommer anropas
        //med den matchade strängen)
        bodyre := re.ReplaceAllStringFunc(body,repl)
        p := &Page{Title: title, Body: []byte(bodyre)}
        err := p.save()
        if err != nil {
                http.Error(w, err.String(),
                        http.StatusInternalServerError)
                return
        }
        http.Redirect(w, r, "/view/"+title, http.StatusFound)
}

func repl(match string) string {
        matchnew := strings.Trim(match, "[]")
        s:= "<a href=\"/view/"+matchnew+"\">"+matchnew+"</a>"
        return s
}
```

Listing 14: Direktlänking

5 Diskussion

5.1 Jämförelse med andra språk

Det bästa sättet att få en överblick över ett programmeringsspråks styrkor och svagheter är att jämföra det med andra språk. Vi har jämfört Go med språken C och Java genom att implementera en enkel webserver och en quicksort i de olika språken. Inför dessa två experiment utformades en hypotes om vad för resultat som förväntades och i webserverns fall om hur enkel en implementering skulle bli att göra.

Förutom testdata från experimenten jämfördes även allmäna skillnader såsom typning, implementeringstid, felhantering och utveckling- smiljöer.

Den största skillnaden som märktes när vi jämförde typningen i Java/C och Go var att Go, trots att det är starkt typat, känns väldigt dynamiskt. Gos unika sätt att deklarera variabler och instantiera dessa kändes från början väldigt annorlunda för oss som är vana Java/C programmerare. Men bara efter några timmars programerande i Go kändes syntaxen både naturlig och enkel att använda.

Go-funktioners förmåga att returnera flera värden är något som vi eftertrankar i andra språk efter att ha programmerat i Go. Framförallt då detta underlättar felhantering och naturligt möjliggör någonting som måste ske via omvägar i andra språk.

5.1.1 Quicksort

Det första experimenten som genomfördes var en quicksort. Vi valde att göra en quicksort då det är en känd sorteringsalgoritm som fungerar bra för att demonstrera och testa en av Gos funktioner, nämligen goroutines. En annan stor fördel med Quicksort är att experimentets data är lätt att kontrollera, både hur indata ser ut och att den verkligen blir sorterad.

Det Go har till fördel i detta experiment är att den kan välja att dela upp sorteringen av listan över de två processorkärnorna som fanns i hårdvaran och på så sätt effektivisera exekveringen av algoritmen.

I vår hypotes inför detta experiment ansåg vi att Go (på två kärnor) skulle vara snabbast av de tre och Java långsammast. Resultatet av detta experiment [4.1 Quicksort] verifierade denna hypotes.

Det resultat som förvånade oss var att Go på en kärna körde snabbare än C. Detta kan bero på allmänna felkällor i experimentet då C teoretiskt borde vara snabbare.

Jämför man resultaten av Go på en och två kärnor ser man en tydlig förbättring i körtid då sorteringen utförs parallellt. Även om en naiv hypotes vore att körtiden halveras så var detta inte fallet då detta skulle kräva en perfekt partionering av första listan.

5.1.2 Webserver

Det andra experimentet som genomfördes var ett webservertest i de tre olika språken. Detta experiment utfördes för att testa prestanda för webservrar hos de olika språken och hur enkelt en webserver var att implementera. Vi valde detta experiment efter vår första intervju med Stefan Nilsson då det framgick att webserverprogrammering är en av Gos starka sidor, framförallt då det har ett väldigt bra bibliotek för webservrar.

Vår hypotes inför detta experiment var att implementering skulle ske enklast i Go och Java då båda dessa har väldokumenterade bibliotek för att skapa webservrar. Det var dock betydligt enklare att skapa en webserver i Go än vi anat och gjorde det på bara några rader [Listing 10] till skillnaded från Java appendix [A.2.1 Java]

För en webserver i C användes programmet nginx, en färdig och optimerad högprestanda webserver. Detta gör självklart att resultaten blev skeva till fördel för C, men visar på att välskriven C-kod är svårslagen.

Resultaten för detta experiment finns presenterade i [4.2 Webserver]. I [Fig. 2] visas antal svar per sekund beroende på antal samtidiga trådar för de olika servrarna. Här syns tydligt att nginx är överlägsen de servrar som vi implementerat, vilket är ett resultat som inte är oväntat.

Däremot ser vi även att Java-servern kan svara på nästan dubbelt så många anrop per sekund i jämförelse med Go-servern. Det intressanta är dock att Java-servern inte klarar av fler än 50 sammtidiga anrop innan den ger ”Time-out”, till skillnad från Go-servern som klarar att hantera upp till 125.

I den andra grafen, [Fig. 3] jämförs responstiden på servrarna beroende på antal sammtidiga anrop. Responstiden i denna graf är tiden som 99% av anropen har svarats inom. Återigen ser vi att nginx är betydligt bättre än både Go och Java. Vi ser även att Gos responstid är ungefär dubbelt så hög som Javas och den växer även kraftigt i förhållande till antal sammtidiga anrop.

Vi är en aning förvånade över Go-serverns prestanda resultat då webserverprogrammering som sagt ska vara en av Gos starka sidor. Implementeringen skedde väldigt enkelt men efter Quicksort experimentet trodde vi även att Go kunde mäta sig med Java och C prestandamässigt, vilket inte är fallet.

5.1.3 Felkällor

I dessa två experiment har vi identifierat några felkällor som kan ha påverkat resultaten:

- Tidtagningen för Quicksort-testerna skedde med hjälp av språkens egna

bibliotek, där fel kan uppkomma[8].

- Javakoden kommer inte att vara optimerad de första gångerna den körs, vilket kan påverka programmets körtid[8].
- Webservrarna kördes lokalt vilket skulle kunna påverka resultatet för prestandatesten
- Val av Go-kompilator påverkar hur effektivt programmet blir då gccgo ger mer effektiv kod än 8g

5.2 Wiki

Vi valde att utveckla vår simpla Go-webserver till en enkel wiki för att få mer erfarenhet av språket, anledningen till att vi valde en wiki var för att ytterligare utforska Gos bibliotek för webservrar (framförallt paketen http och templates).

För att utveckla vår webbserver till en wiki behövde vi implementera flera olika nya funktioner. Bland annat för att servern ska kunna spara och skriva sidor till hårddisken, men också för att ta emot ändringar från användarel. En ingående beskrivning om hur dessa problem löstes finns i [4.3 Go Wiki].

Det finns inte så mycket att diskutera om resultatet av vår wiki, förutom det vi lärt oss av att skriva den. Funktionerna wikin behövde blev med hjälp av Gos bibliotek enkla att implementera och efter att ha jobbat med det kan vi inte annat än att hålla med om att webapplikations programmering har gjorts enkelt av Go.

5.3 Slutsats

Go anses idag fortfarande vara ett experiment. Språket är fortfarande ungt och uppdaterades varje vecka fram till mars, att uppdateringar idag endast sker en gång i månaden visar att det börjar bli stabilt[12].

Det unika sätt Go implementerar parallellprogrammering och BSD-lisensen har lockat till sig ett stort intresse, många är intresserade av språket och många jobbar på det. Ett bevis på detta är t ex Gccgo.

Detta höga intresse leder också till att åsikterna om språket är många och polariserade. Målet med denna rapport har varit att undersöka språket och på så sätt bilda vår egna uppfattning och åsikt om det.

Vi har under detta arbete kommit fram till vad vi anser är bra och mindre bra med Go. Trots att vissa delar av språket skiljer sig väldigt mycket från Java och C, så fann vi att övergången till Go från dessa språk hade gjorts enkel, någonting som utvecklarna har haft i åtanke.

De ändringar som gjorts från Cs syntax till Go anser vi ger språket en mer modern känsla där onödiga nyckelord och upprepningar har tagits bort. Syntaxen är ren och känns dynamisk, trots den strikta typningen.

Quicksort-experimentet visade oss att Gos möjlighet att utnyttja flerkärniga och nätverkade datorer ger det ett övertag över de nu stora systemspråken. Detta är någonting som vi tycker är väldigt intressant och talar för Gos framtid, då flerkärniga proccessorer har blivit standard.

Genom att skapa en webserver för prestanda-experimentet och sedan utveckla det till en wiki fick vi möjligheten att utforska en större del av Gos bibliotek. Framförallt paketen för webserver-och webapplikationsprogrammering, områden som Go utvecklats för att ha stort stöd i. Att göra detta övertygade oss om att Go inom en snar framtid kommer bli en stark spelare inom de områderna. Vi hittade i biblioteket mycket nytänk och smarta lösningar som gjorde serverprogrammeringen enkel, bl a paketet Templates som används för att separera html som ska generaras från serverns källkod. Prestandan på servern var dock inte den bästa och det återstår att se om optimeringar kommer att göras.

En nackdel vi funnit med Go är att det fortfarande är ostabilt. Med uppdateringar varje månad kan program skrivna för några månader sen helt plötsligt sluta kompilera, någonting som vi anser avskräcker mer seriösa och kommersiella projekt i Go. Detta är dock någonting som förhoppningsvis kommer att lösas då språket blir äldre.

Det är ännu för tidigt för att säga om Go kommer att bli ett av de stora systemspråken eller inte. I intervjun med Stefan Nilsson kom det fram att den främsta anledningen till att Java lyckades bli stort var för att språket fyllde ett behov av portabel kod som flera stora företag hade.

Go fyller idag inte ett sådant stort behov och kommer därför troligtvis inte att genomgå den popularitetsexplosion som Java hade. Dock behöver språket inte heller fylla ett stort hål för att vara relevant, och hurvida en gradvis ökning av Gos användande slutligen kommer att göra det till ett av de stora språken återstår att se.

Referenser

[1] R. Pike, *Go Course Day 1*, Oktober 2009
http://www.golang.org/doc/GoCourseDay1.pdf

[2] R. Pike, *The Go Programming Language*, Oktober 2009
http://www.youtube.com/watch?v=rKnDgT73v8s

[3] Go Team, *Effective Go*
http://golang.org/doc/effective_go.html

[4] R. Pike, *Another Go at Language Design*, Augusti 2010
http://www.youtube.com/watch?v=7VcArS4Wpqk

[5] P. Ryan, *Go: new open source programming language from Google*, November 2009
http://arstechnica.com/open-source/news/2009/11/go-new-open-source-programming-language-from-google.ars

[6] Go Team, *Frequently Asked Questions*
http://golang.org/doc/go_faq.html

[7] C. Rance, S. Smolka, *Strategic directions in concurrency research*, December 1996, Publicerad i AMC Computing Surveys

[8] S. Nilsson, *Intervju om Go*, Mars 2011

[9] I. L. Taylor, *Gccgo in GCC*, December 2010
http://www.airs.com/blog/archives/448

[10] C. A. R. Hoare, *En interjvu med C.A.R. Hoare*
http://cacm.acm.org/magazines/2009/3/21782-an-interview-with-car-hoare/fulltext

[11] Java Platform Standard Ed. 6, *Class Arrays*
http://download.oracle.com/javase/6/docs/api/

[12] The Go Programming Language Blog, *Go becomes more stable*
http://blog.golang.org/2011/03/go-becomes-more-stable.html

A Källkod

A.1 Quicksort

A.1.1 C

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

//metod för att byta plats på två element i listan
void swap(int *x,int *y)
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}

//quicksort
void quicksort(int list[],int m,int n)
{
    int key,i,j,k;
    if( m < n)
    {
        k = (m+n) /2
        swap(&list[m],&list[k]);
        key = list[m];
        i = m+1;
        j = n;
        while(i <= j)
        {
            while((i <= n) && (list[i] <= key))
                    i++;
            while((j >= m) && (list[j] > key))
                    j--;
            if( i < j)
                    swap(&list[i],&list[j]);
        }

        swap(&list[m],&list[j]);

          //rekursivt anrop
        quicksort(list,m,j-1);
```

```
        quicksort(list,j+1,n);
    }
}



void main()
{
    const int MAX_ELEMENTS = 4000000;

    int *list=(int*)malloc(sizeof(int)*MAX_ELEMENTS);
    int i = 0;

    for(i = 0; i < MAX_ELEMENTS; i++ ){

        list[i] = rand();

    }

    struct timeval tic, toc;
         gettimeofday(&tic,NULL);

    quicksort(list,0,MAX_ELEMENTS-1);
    gettimeofday(&toc,NULL);

    time_t diff = (toc.tv_sec-tic.tv_sec)*1000000 + (toc.tv_usec-tic.tv_us
         printf("%d\n",diff);

 free(list);
}
```

A.1.2 Go

```
package main

import (
        "flag"
        "fmt"
        "rand"
        "time"
        "runtime"
)
```

```
var first = 0

func main() {
        runtime.GOMAXPROCS(2);
        var count, mag int
        flag.IntVar(&count, "s", 1000000)
        flag.IntVar(&mag, "m", 1000000)
        flag.Parse()

        //skapa slice
        var sl = make([]int, 0, count)  // initial length 0


        //seeda random
        rand.Seed(time.Nanoseconds())

        for i := 0; i < count; i++ {
                n := len(sl)
                sl = sl[0:n+1]
                sl[n] = rand.Intn(mag)
        }

        var a = time.Nanoseconds()
        quicksort(sl, 0, len(sl)-1)
        var b = time.Nanoseconds() - a
        fmt.Println(b)
}

func quicksort(a []int, lo int, hi int) {
        i,j := lo,hi
        var x int
        x = a[(lo+hi)/2]

        for i<=j {
                for a[i] < x {i++}
                for a[j] > x {j--}
                if i <= j {
                        a[i], a[j] = a[j], a[i]
                        i++
                        j--
                }
        }
        if lo < j {
                if(first == 0) {
```

```
                        first = 1
                        go quicksort(a, lo, j)
                        }else {
                        quicksort(a, lo, j)
                }
                }
        if i < hi { quicksort(a, i, hi)}
}
```

A.2 Webserver

A.2.1 Java

```
import java.io.*;
import java.net.*;
import java.util.StringTokenizer;

public class HttpServer {

public static void main(String[] args) throws IOException{
        ServerSocket ss = new ServerSocket(1234);
        while(true){
                Socket s = ss.accept();
                BufferedReader request =
                        new BufferedReader(
new InputStreamReader(s.getInputStream()));
                String str = request.readLine();
                StringTokenizer tokens =
                        new StringTokenizer(str," ?");
                tokens.nextToken();
                String requestedDocument = tokens.nextToken();

                s.shutdownInput();
                PrintStream response =
                        new PrintStream(s.getOutputStream());
                response.println("HTTP/1.1 200 OK");
                response.println("Server : Java webserver experiment");
                if(requestedDocument.indexOf(".html") != -1)
                        response.println("Content-Type: text/html");
                if(requestedDocument.indexOf(".gif") != -1)
                        response.println("Content-Type: image/gif");

                response.println();
```

```
                File f = new File("."+requestedDocument);
                FileInputStream infil = new FileInputStream(f);
                byte[] b = new byte[1024];

                while(infil.available() > 0){
                        response.write(b,0,infil.read(b));
                }

                s.shutdownOutput();
                s.close();
        }
}
}
```

A.3 Wiki

```
package main
        import (
                "http"
                "io/ioutil"
                "os"
                "regexp"
                "template"
                "strings"
                "time"
                "fmt"
        )

        type Page struct {
                Title string
                Body  []byte
        }

        func (p *Page) save() os.Error {
                oldVer,err := loadPage(p.Title)

                filename := p.Title + ".txt"
                tid := time.LocalTime().String()

                if err != nil {
                }else {
                        filenameHis := "history/"+p.Title+tid+".txt"
                        ioutil.WriteFile(filenameHis, oldVer.Body, 0600)
```

```
        }
        return ioutil.WriteFile(filename, p.Body, 0600)
}

func loadPage(title string) (*Page, os.Error) {
        filename := title + ".txt"
        body, err := ioutil.ReadFile(filename)
        if err != nil {
                return nil, err
        }
        return &Page{Title: title, Body: body}, nil
}

func viewHandler(w http.ResponseWriter, r *http.Request, title
 string) {
        p, err := loadPage(title)
        if err != nil {
        http.Redirect(w, r, "/edit/"+title, http.StatusFound)
                return
        }
        renderTemplate(w, "view", p)
}

func editHandler(w http.ResponseWriter, r *http.Request,
title string) {
        p, err := loadPage(title)
        if err != nil {
                p = &Page{Title: title}
        }
        renderTemplate(w, "edit", p)
}

func repl(match string) string {
        matchnew := strings.Trim(match, "[]")
        s:= "<a href=\"/view/"+matchnew+"\">"+matchnew+"</a>"
        return s
}

func saveHandler(w http.ResponseWriter, r *http.Request,
title string) {
        body := r.FormValue("body")
        re,_ := regexp.Compile("\\[[a-zA-Z]+\\]")
        bodyre := re.ReplaceAllStringFunc(body,repl)
        p := &Page{Title: title, Body: []byte(bodyre)}
```

```
                err := p.save()
                if err != nil {
                        http.Error(w, err.String(),
 http.StatusInternalServerError)
                        return
                }
                http.Redirect(w, r, "/view/"+title, http.StatusFound)
        }

        var templates = make(map[string]*template.Template)

        func init() {
                for _, tmpl := range []string{"edit", "view"} {

templates[tmpl] = template.MustParseFile(tmpl+".html", nil)
                }
        }

        func renderTemplate(w http.ResponseWriter, tmpl string, p *Page) {
                err := templates[tmpl].Execute(w, p)
                if err != nil {
                        http.Error(w, err.String(),
http.StatusInternalServerError)
                }
        }

        const lenPath = len("/view/")

        var titleValidator = regexp.MustCompile("^[a-zA-Z0-9]+$")

        func makeHandler(fn func(http.ResponseWriter, *http.Request,
 string)) http.HandlerFunc {
                return func(w http.ResponseWriter, r *http.Request) {
                        title := r.URL.Path[lenPath:]
                        if !titleValidator.MatchString(title) {
                                http.NotFound(w, r)
                                return
                        }
                        fn(w, r, title)
                }
        }

        func main() {
                http.HandleFunc("/view/", makeHandler(viewHandler))
```

```
        http.HandleFunc("/edit/", makeHandler(editHandler))
        http.HandleFunc("/save/", makeHandler(saveHandler))
        http.ListenAndServe(":8080", nil)
}
```

B Testdata

B.1 Webserver

B.1.1 Java

25

```
Server Software:
Server Hostname:        localhost
Server Port:            1234

Document Path:          /index.html
Document Length:        4 bytes

Concurrency Level:      25
Time taken for tests:   237.152 seconds
Complete requests:      1000000
Failed requests:        0
Write errors:           0
Total transferred:      134000000 bytes
HTML transferred:       4000000 bytes
Requests per second:    4216.70 [#/sec] (mean)
Time per request:       5.929 [ms] (mean)
Time per request:       0.237 [ms] (mean, across all concurrent requests)
Transfer rate:          551.79 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.3      0      13
Processing:     0    6   2.2      5      57
Waiting:        0    5   2.1      5      57
Total:          1    6   2.1      5      57

Percentage of the requests served within a certain time (ms)
  50%      5
  66%      6
  75%      7
  80%      8
  90%      9
```

```
  95%      9
  98%     11
  99%     12
 100%     57 (longest request)
```

50

```
Server Software:
Server Hostname:        localhost
Server Port:            1234

Document Path:          /index.html
Document Length:        4 bytes

Concurrency Level:      50
Time taken for tests:   163.807 seconds
Complete requests:      1000000
Failed requests:        0
Write errors:           0
Total transferred:      138888902 bytes
HTML transferred:       4000000 bytes
Requests per second:    6104.73 [#/sec] (mean)
Time per request:       8.190 [ms] (mean)
Time per request:       0.164 [ms] (mean, across all concurrent requests)
Transfer rate:          828.01 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.5      0      16
Processing:     1    8   2.9      8      40
Waiting:        0    8   2.8      7      40
Total:          2    8   2.7      8      41

Percentage of the requests served within a certain time (ms)
  50%      8
  66%      8
  75%      9
  80%     10
  90%     12
  95%     13
  98%     16
  99%     17
 100%     41 (longest request)
```

B.1.2 C

25

```
Server Software:        nginx/0.7.67
Server Hostname:        localhost
Server Port:            80

Document Path:          /
Document Length:        169 bytes

Concurrency Level:      25
Time taken for tests:   50.615 seconds
Complete requests:      1000000
Failed requests:        0
Write errors:           0
Non-2xx responses:      1000000
Total transferred:      319000000 bytes
HTML transferred:       169000000 bytes
Requests per second:    19757.01 [#/sec] (mean)
Time per request:       1.265 [ms] (mean)
Time per request:       0.051 [ms] (mean, across all concurrent requests)
Transfer rate:          6154.77 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.1      0       5
Processing:     0    1   0.5      1      84
Waiting:        0    1   0.5      1      82
Total:          1    1   0.5      1      84

Percentage of the requests served within a certain time (ms)
  50%      1
  66%      1
  75%      1
  80%      1
  90%      1
  95%      2
  98%      2
  99%      2
 100%     84 (longest request)
```

50

```
Server Software:        nginx/0.7.67
Server Hostname:        localhost
Server Port:            80
```

```
Document Path:          /
Document Length:        169 bytes

Concurrency Level:      50
Time taken for tests:   51.642 seconds
Complete requests:      1000000
Failed requests:        0
Write errors:           0
Non-2xx responses:      1000000
Total transferred:      319000000 bytes
HTML transferred:       169000000 bytes
Requests per second:    19364.03 [#/sec] (mean)
Time per request:       2.582 [ms] (mean)
Time per request:       0.052 [ms] (mean, across all concurrent requests)
Transfer rate:          6032.35 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    1   0.3      1       8
Processing:     1    2   1.0      2     135
Waiting:        0    1   1.0      1     132
Total:          2    3   1.0      3     136

Percentage of the requests served within a certain time (ms)
  50%      3
  66%      3
  75%      3
  80%      3
  90%      3
  95%      3
  98%      3
  99%      4
 100%    136 (longest request)
```

75

```
Server Software:        nginx/0.7.67
Server Hostname:        localhost
Server Port:            80

Document Path:          /
Document Length:        169 bytes

Concurrency Level:      75
```

```
Time taken for tests:   52.155 seconds
Complete requests:      1000000
Failed requests:        0
Write errors:           0
Non-2xx responses:      1000000
Total transferred:      319000000 bytes
HTML transferred:       169000000 bytes
Requests per second:    19173.76 [#/sec] (mean)
Time per request:       3.912 [ms] (mean)
Time per request:       0.052 [ms] (mean, across all concurrent requests)
Transfer rate:          5973.08 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    1   0.3      2       9
Processing:     1    2   0.6      2      32
Waiting:        1    2   0.6      2      32
Total:          2    4   0.5      4      32
ERROR: The median and mean for the initial connection time are more than t
       deviation apart. These results are NOT reliable.

Percentage of the requests served within a certain time (ms)
  50%      4
  66%      4
  75%      4
  80%      4
  90%      4
  95%      4
  98%      5
  99%      5
 100%     32 (longest request)
```

100

```
Server Software:        nginx/0.7.67
Server Hostname:        localhost
Server Port:            80

Document Path:          /
Document Length:        169 bytes

Concurrency Level:      100
Time taken for tests:   53.248 seconds
Complete requests:      1000000
Failed requests:        0
```

```
Write errors:           0
Non-2xx responses:      1000000
Total transferred:      319000000 bytes
HTML transferred:       169000000 bytes
Requests per second:    18780.22 [#/sec] (mean)
Time per request:       5.325 [ms] (mean)
Time per request:       0.053 [ms] (mean, across all concurrent requests)
Transfer rate:          5850.48 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    2   0.4      2      12
Processing:     1    3   0.7      3      18
Waiting:        1    3   0.7      3      18
Total:          3    5   0.5      5      19

Percentage of the requests served within a certain time (ms)
  50%      5
  66%      5
  75%      5
  80%      6
  90%      6
  95%      6
  98%      6
  99%      7
 100%     19 (longest request)
```

125

```
Server Software:        nginx/0.7.67
Server Hostname:        localhost
Server Port:            80

Document Path:          /
Document Length:        169 bytes

Concurrency Level:      125
Time taken for tests:   53.877 seconds
Complete requests:      1000000
Failed requests:        0
Write errors:           0
Non-2xx responses:      1000000
Total transferred:      319000000 bytes
HTML transferred:       169000000 bytes
```

```
Requests per second:    18560.84 [#/sec] (mean)
Time per request:       6.735 [ms] (mean)
Time per request:       0.054 [ms] (mean, across all concurrent requests)
Transfer rate:          5782.14 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    3   0.5      3      11
Processing:     1    4   0.8      4      28
Waiting:        1    3   0.9      3      28
Total:          4    7   0.7      7      29

Percentage of the requests served within a certain time (ms)
  50%      7
  66%      7
  75%      7
  80%      7
  90%      7
  95%      7
  98%      8
  99%      9
 100%     29 (longest request)
```

150

```
Server Software:        nginx/0.7.67
Server Hostname:        localhost
Server Port:            80

Document Path:          /
Document Length:        169 bytes

Concurrency Level:      150
Time taken for tests:   54.675 seconds
Complete requests:      1000000
Failed requests:        0
Write errors:           0
Non-2xx responses:      1000000
Total transferred:      319000000 bytes
HTML transferred:       169000000 bytes
Requests per second:    18289.79 [#/sec] (mean)
Time per request:       8.201 [ms] (mean)
Time per request:       0.055 [ms] (mean, across all concurrent requests)
Transfer rate:          5697.70 [Kbytes/sec] received
```

```
Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    4  42.1      3    3015
Processing:     2    5   1.9      5     229
Waiting:        1    4   1.9      4     228
Total:          5    8  42.1      7    3020

Percentage of the requests served within a certain time (ms)
  50%      7
  66%      8
  75%      8
  80%      8
  90%      8
  95%      9
  98%      9
  99%     11
 100%   3020 (longest request)
```

500

```
Server Software:        nginx/0.7.67
Server Hostname:        localhost
Server Port:            80

Document Path:          /
Document Length:        169 bytes

Concurrency Level:      500
Time taken for tests:   61.408 seconds
Complete requests:      1000000
Failed requests:        0
Write errors:           0
Non-2xx responses:      1000000
Total transferred:      319000000 bytes
HTML transferred:       169000000 bytes
Requests per second:    16284.49 [#/sec] (mean)
Time per request:       30.704 [ms] (mean)
Time per request:       0.061 [ms] (mean, across all concurrent requests)
Transfer rate:          5073.00 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0   23 252.3      5    9028
Processing:     2    7   3.5      7     234
Waiting:        1    5   3.3      5     232
```

```
Total:          7   30 252.7     12    9039

Percentage of the requests served within a certain time (ms)
  50%     12
  66%     13
  75%     14
  80%     15
  90%     16
  95%     18
  98%     19
  99%     22
 100%   9039 (longest request)
```

B.1.3 Go

25

```
Server Software:
Server Hostname:        localhost
Server Port:            8080

Document Path:          /
Document Length:        19 bytes

Concurrency Level:      25
Time taken for tests:   496.606 seconds
Complete requests:      1000000
Failed requests:        0
Write errors:           0
Non-2xx responses:      1000000
Total transferred:      123000000 bytes
HTML transferred:       19000000 bytes
Requests per second:    2013.67 [#/sec] (mean)
Time per request:       12.415 [ms] (mean)
Time per request:       0.497 [ms] (mean, across all concurrent requests)
Transfer rate:          241.88 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.2      0       8
Processing:     0   12   6.9     11      48
Waiting:        0   12   6.9     11      48
Total:          0   12   6.9     11      48

Percentage of the requests served within a certain time (ms)
```

```
  50%      11
  66%      15
  75%      17
  80%      19
  90%      22
  95%      26
  98%      28
  99%      29
 100%      48 (longest request)
```

50

```
Server Software:
Server Hostname:        localhost
Server Port:            8080

Document Path:          /
Document Length:        19 bytes

Concurrency Level:      50
Time taken for tests:   363.287 seconds
Complete requests:      1000000
Failed requests:        0
Write errors:           0
Non-2xx responses:      1000000
Total transferred:      123000000 bytes
HTML transferred:       19000000 bytes
Requests per second:    2752.64 [#/sec] (mean)
Time per request:       18.164 [ms] (mean)
Time per request:       0.363 [ms] (mean, across all concurrent requests)
Transfer rate:          330.64 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.3      0      18
Processing:     0   18   9.3     15      70
Waiting:        0   18   9.3     15      69
Total:          0   18   9.3     15      70

Percentage of the requests served within a certain time (ms)
  50%     15
  66%     19
  75%     23
  80%     25
  90%     32
```

```
  95%     39
  98%     45
  99%     47
 100%     70 (longest request)
```

75

```
Server Software:
Server Hostname:        localhost
Server Port:            8080

Document Path:          /
Document Length:        19 bytes

Concurrency Level:      75
Time taken for tests:   379.656 seconds
Complete requests:      1000000
Failed requests:        0
Write errors:           0
Non-2xx responses:      1000000
Total transferred:      123000000 bytes
HTML transferred:       19000000 bytes
Requests per second:    2633.96 [#/sec] (mean)
Time per request:       28.474 [ms] (mean)
Time per request:       0.380 [ms] (mean, across all concurrent requests)
Transfer rate:          316.38 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.5      0      24
Processing:     0   28  12.5     26      90
Waiting:        0   28  12.5     26      88
Total:          1   28  12.4     26      90

Percentage of the requests served within a certain time (ms)
  50%     26
  66%     31
  75%     34
  80%     36
  90%     44
  95%     55
  98%     64
  99%     68
 100%     90 (longest request)
```

100

```
Server Software:
Server Hostname:        localhost
Server Port:            8080

Document Path:          /
Document Length:        19 bytes

Concurrency Level:      100
Time taken for tests:   434.348 seconds
Complete requests:      1000000
Failed requests:        0
Write errors:           0
Non-2xx responses:      1000000
Total transferred:      123000000 bytes
HTML transferred:       19000000 bytes
Requests per second:    2302.30 [#/sec] (mean)
Time per request:       43.435 [ms] (mean)
Time per request:       0.434 [ms] (mean, across all concurrent requests)
Transfer rate:          276.55 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.4      0      26
Processing:     0   43  19.2     39     115
Waiting:        0   43  19.2     38     115
Total:          2   43  19.2     39     115

Percentage of the requests served within a certain time (ms)
  50%     39
  66%     47
  75%     55
  80%     61
  90%     74
  95%     81
  98%     88
  99%     91
 100%    115 (longest request)
```

125

```
Server Software:
Server Hostname:        localhost
Server Port:            8080
```

```
Document Path:          /
Document Length:        19 bytes

Concurrency Level:      125
Time taken for tests:   470.858 seconds
Complete requests:      1000000
Failed requests:        0
Write errors:           0
Non-2xx responses:      1000000
Total transferred:      123000000 bytes
HTML transferred:       19000000 bytes
Requests per second:    2123.78 [#/sec] (mean)
Time per request:       58.857 [ms] (mean)
Time per request:       0.471 [ms] (mean, across all concurrent requests)
Transfer rate:          255.10 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.3      0      24
Processing:     2   59  24.7     52     123
Waiting:        1   59  24.6     52     123
Total:          5   59  24.6     52     123

Percentage of the requests served within a certain time (ms)
  50%     52
  66%     69
  75%     81
  80%     86
  90%     96
  95%    101
  98%    107
  99%    110
 100%    123 (longest request)
```