Kungliga Tekniska Högskolan
DD143X Examensarbete inom datalogi, grundnivå
Handledare: Alexander Baltatzis

En utvärdering av Go, Erlang och F# ur ett concurrencyperspektiv

Linus Eriksson
Frihetsvägen 29
177 53 Järfälla
072 - 253 74 76
liner@kth.se

Niclas Nilsson
Vintergatan 26
172 69 Sundbyberg
070 - 770 84 85
ninilss@kth.se

Abstract

This paper is an evaluation of the programming languages Go, Erlang and F#, three languages with a built in support for concurrency. The source code for nine sample programs written by the authors of this report will be presented. Each language has implemented one benchmark program that measures the amount of time it takes to terminate relative to how many threads are running, one simple chat server and one simple chat client. The introduction part of this paper will give a reason as to why one should care about languages with a built in support for concurrency. The background part of this paper will reason as to why these three languages were chosen for evaluation, as well as give a brief introduction to the languages' histories and traits. The main focus of this paper is the discussion part, where it will be brought forth what a programmer can expect of the languages and in which situations they are best used.

Innehållsförteckning

1. Introduktion
    1.1 Målgrupp
    1.2 Syfte
    1.3 Omfång
    1.4 Samarbetsutlåtande
2. Bakgrund
    2.1 Språken
        2.1.1 Go
        2.1.2 Erlang
        2.1.3 F#
    2.2 Utvecklingsmiljö
3. Utförande
    3.1 Programmen
        3.1.1 Chat-Server
        3.1.2 Chat-Client
        3.1.3 Benchmark
    3.2 Mätmetoder
        3.2.1 Prestanda
        3.2.2 Produktivitet
        3.2.3 Inbyggt stöd för concurrency
        3.2.4 Styrkor och svagheter i språken
        3.2.5 Enkelhet att anamma
4. Resultat
    4.1 Prestanda
    4.2 Produktivitet
5. Analys
    5.1 Prestanda
    5.2 Produktivitet
        5.2.1 Go
        5.2.2 Erlang
        5.2.3 F#
    5.3 Inbyggt stöd för concurrency
        5.3.1 Go
        5.3.2 Erlang
        5.3.3 F#
    5.4 Styrkor och svagheter
        5.4.1 Go
        5.4.2 Erlang
        5.4.3 F#
    5.5 Enkelhet att anamma
        5.5.1 Go
        5.5.2 Erlang
        5.5.3 F#
6. Diskussion
    6.1 Varför språk med inbyggt stöd för concurrency
    6.2 Vad man kan förvänta sig av språken
        6.2.1 Go

6.2.2 Erlang
6.2.3 F#
6.3 När bör man använda språken
6.3.1 Go
6.3.2 Erlang
6.3.3 F#
7. Källförteckning
Appendix A - Definitioner
Appendix B - Källkod
Go - Benchmark
Go - Chat Server
Go - Chat Client
Erlang - Benchmark
Erlang - Chat Server
Erlang - Chat Client
F# - Benchmark
F# - Chatserver
F# - Chat Client

1. Introduktion

I dagens IT-industri kan man se en trend där allt fler applikationer körs på stora servrar istället för att köras lokalt på användarnas egna datorer. Detta innebär att servern måste klara av att köra många olika processer samtidigt och behovet för concurrency ökar. De mest använda programmeringsspråken som idag används ute i industrin är Java, C och C++[1], men inget av dessa språk har ett starkt inbyggt stöd för concurrency. Därför kan det - om man vill följa trenden med att allt fler applikationer körs på servrar - vara nödvändigt att leta efter alternativa språk som är mer lämpade för denna typ av programmering. Tre lovande språk som är anpassade för detta är Go, Erlang och F#.

1.1 Målgrupp

Denna rapport riktar sig främst till personer med erfarenhet av att programmera i Java och som har viss kunskap om fundamentala teorier och grundläggande terminologi inom datavetenskap. Förklaringar av termer som läsaren inte förväntas känna till erbjuds i Appendix A - Definitioner.

Läsaren är intresserad av att lära sig nya programmeringsspråk, speciellt hur dessa hanterar concurrency och till vilka uppgifter dessa språk är bäst lämpade.

1.2 Syfte

Denna rapport har som syfte att utvärdera programmeringsspråken Go, Erlang och F# ur ett concurrencyperspektiv samt vilka problem de är mest lämpade att lösa. Detta ska vi undersöka genom att försöka svara på dessa frågor:

- Hur mäter sig deras prestanda mot varandra?
- Hur enkelt är språket att anamma för en programmerare med en bakgrund inom Java-programmering?
- Hur lång tid tar det att färdigställa en viss mängd kod, alltså hur produktiv kan man vara när man programmerar i språken?
- Hur stor feltolerans har de färdiga programmen och hur stor del av felen upptäcks vid kompilering?
- Hur starkt är det inbyggda stödet för concurrency?
- Vilka styrkor och svagheter har språken?

I slutet i denna rapport hoppas vi kunna ge den som inte har någon erfarenhet av Go, Erlang eller F# en klarare bild av hur det är att programmera i dessa språk, när man bör och inte bör använda dem, samt vad man kan förvänta sig av dem.

1.3 Omfång

Projektets omfång begränsas av tid samt våra resurser och tidigare erfarenheter av programmering, vilket inte är mer än den utbildning vi hittills fått på KTH. Den tid som kommer läggas på projektet är 320 timmar, uppdelade på oss två gruppmedlemmar. Under denna tid förväntas vi ha lärt oss att programmera i Go, Erlang och F#, fördjupat oss i språkens egenskaper, skrivit några exempelprogram, utfört enklare prestandatester samt skrivit

denna rapport. Vi kommer inte ta upp problem som vi stötte på när vi installerade språkens utvecklingsmiljö eller detaljerat återge hur det var att programmera dessa program specifikt.

För att kunna ge en mer detaljerad bild av hur det är att utveckla större projekt i de olika språken hade vi behövt utveckla större projekt. Då detta inte är rimligt att utföra under vår begränsade tid så måste vi hålla oss till att skriva mindre program och dra våra slutsatser därefter.

De prestandatester som vi kommer utföra kommer p.g.a. våra begränsade resurser inte kunna utföras parallellt på olika antal kärnor och olika antal datorer. Istället så kommer de att utföras på en persondator, vilket ger en viss uppskattning om prestanda men inte med den graden av precision som kan behövas ute i industrin.

Ett annat språk som vi hade velat utvärdera men som inte kommer tas upp i denna rapport är Scala - ett multiparadigmspråk med ett starkt inbyggt stöd för concurrency som körs på JVM, vilket gör det kompatibelt med program skrivna i Java.

1.4 Samarbetsutlåtande

Båda författarna var tidigt i projektet överrens om vem som skulle göra vad och hur det skulle utföras. Båda ser projektet som personligt givande och det har alltid stått klart vilken ambition som skulle eftersträvas. Alla beslut har accepterats av båda författarna och det har aldrig funnits någon projektledare.

Niclas Nilsson
Niclas är ansvarig för att programmera alla program i Go och Erlang, samt att utvärdera dessa språk och sammanfatta detta i rapporten.

Linus Eriksson
Linus är ansvarig för att utföra prestandatesterna och analysera resultaten. Han är även ansvarig för att programmera programmen i F#, samt att utvärdera språket och sammanfatta detta i rapporten.

2. Bakgrund

2.1 Språken

När vi skulle välja vilja språk som vi skulle utvärdera så ställde vi tre kriterier som ett språk var tvunget att uppfylla:

- Språket ska vara relevant i dagens industri.
- Språket ska ha ett inbyggt stöd för concurrency.
- Det ska vara möjligt att utveckla program i språket på våra persondatorer.

Då Go, Erlang och F# uppfyller dessa kriterier så bestämde vi att det var kring dessa språk som rapporten ska vara uppbyggd.

2.1.1 Go

Go är ett imperativt språk med en syntax inspirerad av bl.a. Java och C. Det utvecklas med öppen källkod av Google och är fortfarande väldigt ungt; version 1 släpptes den 28:e mars 2012. På Google kände många utvecklare att de var tvungna att välja mellan effektiv kompilering, effektiv exekvering eller snabb och enkel programmering[2]. Med Go ville man skapa ett språk som kombinerade alla dessa drag och som skulle vara anpassat till stora projekt som ofta ligger på serversidan. Resultatet blev ett kompilerat, starkt typat och imperativt språk, med skräpsamling och inbyggt stöd för trådar. Kommunikationen mellan trådar sker via så kallade “kanaler”, vilket kan ses som en variabel genom vilken meddelanden och signaler kan skickas.

2.1.2 Erlang

Erlang (**Er**icsson **Lan**guage) utvecklades under 80-talet av Ericsson för att kunna användas på deras telefonväxlar. Allt eftersom språket utvecklades började det användas mer och mer för att skapa prototyper och år 1995, efter det att ett stort projekt skrivet i Erlang släpptes[3], ansågs språket vara tillräckligt moget för att användas i större projekt. Det var dock patentskyddat fram till 1998 då det släpptes som open source. Erlangs struktur är hårt knuten till dess tänkta användning för telefonväxlar; det har en enkel och kraftfull concurrencymodell och har en delegerande roll i ett system, där mer resursintensiva beräkningar utförs av program skrivna i C som enkelt kan anropas via meddelanden i Erlang.

Program skrivna i Erlang skalar väl relativt antalet processer och processer kommunicerar genom att skicka meddelanden till varandra. Syntaxen är väldigt lik Prologs syntax och har lånat många koncept därifrån, såsom atomer, tupler, mönstermatchning och konstanta variabler. Något som dock inte lånats från Prolog är backtracking.

Erlang är ett tolkat språk och ett programs kod kan spridas ut och exekveras på flera olika datorer samtidigt. Koden kan hot-swappas och det faktum att koden körs genom en interpreter gör Erlang till ett relativt operativsystemsoberoende språk.

2.1.3 F#

F# (uttalas F Sharp) är ett nytt språk som började utvecklas 2005 och är en del av Microsofts .NET-språk såsom C#, Visual Basic, JScript och många andra språk. F# och alla andra programmeringsspråk i .NET-familjen kompileras ner till något som kallas Common Language Infrastructure eller Microsoft Intermediate Languange och körs sedan på Common Language Runtime, som är en virtuell maskin som .NET-ramverket körs på. Detta medför att alla språken inom .NET-familjen lätt kan samarbeta med varandra. Det var detta som Microsoft var ute efter när de började utveckla F#, ett starkt typat, typ-härledande och funktionellt språk som samtidigt hade tillgång till hela .NET-biblioteket. 2010 meddelade Microsoft att källkoden till F# skulle släppas fri [4].

2.2 Utvecklingsmiljö

Programmen som är skrivna i Go respektive Erlang är utvecklade på en MacBook Pro i programmet TextMate. TextMate är en simpel texteditor med syntax-highlighting för både Go och Erlang, men som inte har några andra, för projektet, användbara funktioner. Kompilering och exekvering av programmen sker manuellt via en terminal.

Programmen skrivna i F# är däremot skrivna i Microsoft Visual Studio 2010 på en HP Pavillion körandes Windows 7. Visual Studio är Microsofts egna utvecklingsmiljö för en mängd språk, däribland F#. Vid kompilering får man en exekverbar fil som man sedan kan köra via filhanteraren.

3. Utförande

För att kunna skapa en bild av språken och göra en rättvis jämförelse mellan dem så måste vi skriva några mindre program på ett femtiotal rader kod vardera, som utför samma uppgift och som har en viss grund i concurrency. Eftersom vi inte kommer kunna utveckla några större projekt så måste vi även läsa och sammanfatta artiklar, böcker, bloggar och forumposter som rör de syften som vi har ställt upp.

3.1 Programmen

3.1.1 Chat-Server

En chat-server som tar emot och skickar textmeddelanden över sockets. När servern tar emot en förfrågan från klienten om anslutning så skapar servern en ny tråd som ligger och lyssnar efter meddelanden från klienten. När ett meddelande tagits emot så skickas detta ut till alla anslutna klienter. Genom att på detta sätt skapa trådar och kommunicera över sockets kan vi undersöka hur grundläggande trådhantering sker i språket, samt hur meddelanden kan skickas mellan olika program.

3.1.2 Chat-Client

Chat-klienten kopplar upp sig till chat-servern via sockets. En tråd väntar på meddelanden från socketen och skriver ut dessa till terminalen när de tas emot, medan en annan tråd tar emot input från användaren via kommandoraden, som den sen skickar till servern.

3.1.3 Benchmark

Ett enkelt benchmarktest som skapar ett visst antal trådar, där varje tråd i sin tur har en räknare som räknar upp till en miljon - en uppgift som inte fyller något annat syfte än att sysselsätta processorn. Genom att variera antalet trådar kan vi skaffa oss en uppskattning över hur språken skalar beroende på antalet trådar som körs parallellt.

3.2 Mätmetoder

3.2.1 Prestanda

Med programmet Benchmark kan vi i varje språk skapa ett varierande antal trådar som utför samma uppgift. Genom att variera antalet trådar och mäta hur lång tid det tar att terminera programmet så kan vi uppskatta och jämföra hur de olika språken skalar beroende på antalet trådar som körs parallellt.

Mätningarna görs med körningar på 1000, 5000, 10000 och 30000 trådar. Genom att köra programmen i varje språk 5 gånger för varje olika antal trådar så kan vi räkna ut ett medelvärde för hur lång tid det tar för varje program att terminera, relativt antal trådar. Totalt ska alltså 20 mätningar göras för varje språk.

3.2.2 Produktivitet

Genom att mäta hur lång tid det tog att skriva varje program så kan vi skapa en uppfattning om hur effektivt man kan programmera i de olika språken, relativt varandra. Samtidigt kan vi även mäta hur stor del av den tiden som spenderas med att felsöka, vilket tillsammans med våra egna sammanställda erfarenheter ger en bild över hur produktiv man som programmerare kan vara när man programmerar i språken.

3.2.3 Inbyggt stöd för concurrency

Att mäta det inbyggda stödet för concurrency hos ett språk är dels svårt och dels subjektivt. Dock så råder det en allmän konsensus om hur väl lämpade de tre språken är för concurrency, en konsensus som vi ska undersöka genom att analysera artiklar och jämföra med våra egna erfarenheter efter att ha programmerat i språken.

3.2.4 Styrkor och svagheter i språken

Efter att ha programmerat i språken så kommer vi ha bildat oss en personlig uppfattning om språkens styrkor och svagheter. Genom att läsa och analysera artiklar om språken så kommer vi blanda våra egna och andras åsikter för att försöka skapa en så objektiv bild som möjligt av språkens styrkor och svagheter.

3.2.5 Enkelhet att anamma

Det är viktigt för ett programmeringspråks spridning att det är lätt att anamma för nya användare. Eftersom ingen persons bakgrund är den andres lik så är det svårt att ge en generell bild över hur lätt språket är att anamma. Därför kommer vi utgå från våra egna bakgrunder, vilka stämmer överrens med många andra högskolestuderandes bakgrund. Därigenom hoppas vi skapa en så objektiv analys som möjligt.

4. Resultat

4.1 Prestanda

Vid prestandatesterna användes en HP Pavillion dv6-1320eo Entertainment Notebook. Datorn har en dubbelkärnig processor på 2,1 GHz och 4096 MB RAM-minne[5]. Alla åtgärder åtogs för att se till så att så få utomstående processer kördes under testet, för att minska störningar i resultaten.

Resultaten av våra mätningar av programmet Benchmark visas bäst med ett diagram som vi ser i figur 1 nedan. Notera är att diagrammet visas i logaritmisk skala.

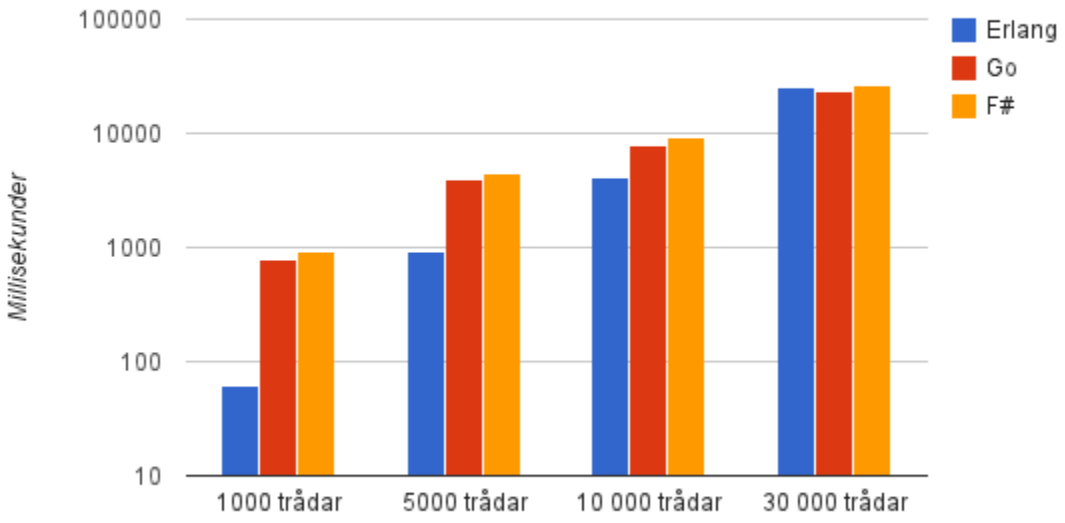

Fig 1. Diagrammet visar, i millisekunder, hur snabbt de olika programspråken körde klart beroende på hur många trådar som användes.

Vid användning av 1000 trådar så var Erlang mer än 10 gånger så snabbt jämfört med både Go och F# men avståndet till de andra två språken minskades avsevärt när antalet trådar ökades. När 10 000 trådar skapades så var Erlang fortfarande snabbast, ungefär dubbelt så snabbt som både Go och F#. Vid 30 000 trådar så vart tiderna mellan de tre språken väldigt jämna, F# och Erlang var nästan exakt lika snabba medan Go var lite snabbare än de andra två språken. 30 000 trådar var även Erlangs gräns för vad som kunde skapas på denna dator innan programmet kraschade.

Go och F# var relativt jämna under hela testet, men Go var hela tiden snäppet snabbare. Men även Go hade en övre gräns för hur många trådar som kunde köras. När 1 000 000 trådar skapades så kraschade Go medan F# klarade av att köra klart.

4.2 Produktivitet

I figur 2 nedan finns en tabell över hur lång tid det tog oss att skriva koden för varje program, samt hur mycket av den tiden som spenderades med felsökning.

Språk	**Program**	**Total tid**	**Varav felsökning**
Erlang	Chat Server	3h 20min	1h
Go	Chat Server	2h	30min
F#	Chat Server	4h 30min	1h
Erlang	Chat Client	2h	30min
Go	Chat Client	50min	10min
F#	Chat Client	1h 40min	40min
Erlang	Benchmark	1h 30min	20min
Go	Benchmark	2h 20min	1h 10min
F#	Benchmark	1h 30min	1h 10min

Fig 2. Tabell över hur mycket tid som spenderades med att skriva varje program.

5. Analys

5.1 Prestanda

Vid prestandatesterna så kördes programmet Benchmark. Det är ett simpelt program som uppfyller sin funktion, nämligen att sysselsätta processorn. Det kan dock vara lite missvisande; kollar man på diagrammet i figur 1 så visas till exempel hur fort de olika programmen klarade av att köra 10 000 trådar samtidigt. Att starta den 10 000:e tråden innan den första har hunnit köra klart bör inte vara något problem, men när antalet trådar ökar så ökar även risken för att den första tråden hinner göra klart sin uppgift innan den sista tråden har skapats. Detta kommer av att trådarna startas en åt gången och det tar såklart tid om trådarna är många. Vi anser dock att vi undviker detta problem då vi bara mäter upp till 30 000 trådar samtidigt, vilket i sammanhanget inte är extremt mycket.

Valet av att köra upp till just 30 000 trådar kommer av att Erlang inte klarade av att köra fler trådar än så samtidigt utan att krascha, så för att få en rättvis jämförelse så stannade vi där. Vi fick även göra en avvägning av hur länge trådarna skulle köras och vi valde att en tråd räknar till 1 000 000. Vi ansåg att detta var en bra kompromiss, tillräckligt långt för att trådarna inte skulle avsluta för snabbt, men samtidigt inte heller så långt att testerna skulle ta flera timmar.

Utifrån våra tester så kan Go ses som det programmeringsspråket som är mest balanserat av de tre. Det var inte snabbast i våra tester samtidigt som det inte heller klarade av att köra flest trådar samtidigt. Det klarade dock av mer än tre gånger så många trådar som det språk som klarade av minst antal trådar, Erlang, och körde klart programmet snabbare än det språk som var långsammast, F#.

Erlang var till en början det överlägset snabbaste språket i våra tester av programmet Benchmark. Dock så klarade Erlang inte av att köra klart när antalet trådar översteg 30 000 och närmade sig de andra programmeringsspråken, tidsmässigt, varje gång som antalet trådar höjdes. Att tiderna för just Erlang ökade så dramatiskt kan bero på att det är det enda språket som använder en så kallad interpreter. Detta kan medföra att programmet som körs inte får tillgång till lika mycket processorkraft eller minne som programmen som inte körs genom en interpreter. Självklart kan det även bero på att Erlang helt enkelt inte klarar av att köra så många trådar samtidigt.

F# var det långsammaste språket av de tre, även om det inte var mycket långsammare än Go och att det vid 30 000 trådar var lika snabbt som Erlang. Men även om F# kanske inte är det snabbaste alternativet på marknaden så kan det vara ett bra alternativ om man ska köra applikationer med ett stort antal trådar, då det enligt våran undersökning är det språk som klarar av att köra flest antal trådar parallellt.

5.2 Produktivitet

Som vi såg i Figur 2 i avsnitt 4.2 så var chat- och serverklienten de program som tog längst tid att skriva, något som kan vara missvisande. Dessa program var de som vi skrev först, vilket innebar att vi var tvungna att uppdatera mycket utav våra kunskaper om socket-programmering. Dessutom så spenderades mycket av den tiden med att lära sig språket, då ingen av oss använt varken Erlang, Go eller F# innan. Därför skulle vi vilja påstå att om vi hade skrivit dessa program sist så bör tiden ha minskat markant.

5.2.1 Go

Att programmera i Go känns väldigt likt att programmera i Java. Go har dock ingen typhierarki och överlag så finns det ingen nästlad arvshierarki, vilket får Go att kännas lättviktigt i jämförelse med Java. Dessutom implementerar Go pekare på ett sätt som mer liknar referenser i Java än pekare i C, men som fortfarande uppvisar samma betéende som pekare i C. Det faktum att språket är starkt typat, samtidigt som det är typhärledande gör att man kan skriva renare kod med mindre risk för typkonverteringsfel.

Av alla benchmark-program så tog det längst tid att skriva det i Go. Detta var för att det inte finns någon inbyggd mekanism för att mäta hur lång tid det tar att terminera en process och dess underprocesser. Därmed var vi tvungna att komma på en egen lösning, vilket tog extra tid.

Att kompilera koden görs enkelt med kommandot “go build” och vill man dessutom kompilera och sedan köra koden så görs det med kommandot “go run”. Detta anser vi underlättar utvecklinsprocessen rejält, precis som i Java. Därmed vill vi påstå att produktiviteten under Go är jämställd med den hos Java.

5.2.2 Erlang

Då språket är långt ifrån maskinnära har användaren liten kontroll över minnesaddressering och när en variabel har blivit tilldelad ett värde så går detta värde inte att ändra. Dessa två egenskaper gör att man som programmerare inte kan skriva över variablers värden av misstag, vilket i sin tur leder till en dramatisk minskning av tid som spenderas med felsökning. När ett fel dock uppstår så är felmeddelandena oftast kryptiska och ger inte särskit mycket information.

Något som vi aldrig stötte på i vårt projekt, men som vi tror kan ha en stor betydelse i större projekt är det faktum att man under körning av ett program kan byta ut delar av den kod som körs. Detta anser vi ökar produktiviteten rejält då man inte behöver avbryta körningen av ett stort program endast för att byta ut en liten del av dess kod, något som det inte finns stöd för hos Go eller F#.

5.2.3 F#

F# skrivs i Microsofts egna välutvecklade utvecklingsmiljö Visual Studio. Som nybörjare i språket så är det både en fördel och nackdel att Visual Studio är så avancerat. Det är lätt hänt att man blir avskräckt av alla funktioner som finns och att man inte förstår hur allting fungerar. Fördelen däremot är att man får så mycket hjälp under själva programmeringen. Visual Studio kompilerar kod medan den skrivs så syntaktiska fel är oftast lätta att upptäcka eftersom Visual Studio säger till när det blivit fel.

5.3 Inbyggt stöd för concurrency

5.3.1 Go

I Go skapas processer genom att man anger nyckelordet “go” följt av den funktion som ska köras som en ny asynkron process. Det finns speciella variabler som kallas för “kanaler”, till vilka processer kan skicka värden. När man ska läsa ut ett värde från en kanal så väntar den läsande processen tills det att en annan process skickar ett värde till kanalen, varefter värdet kan läsas av. Dessa kanaler kan vara både buffrade och obuffrade.

Eftersom en kanal kan användas av flera processer så kan man skapa flera identiska processer som lyssnar på samma kanal. På så sätt så kan man utföra identiska beräkningar parallellt genom en och samma datakälla, utan att behöva göra omfattande omstruktureringar av sin kod.

5.3.2 Erlang

Processer i Erlang skapas med ett enkelt funktionsanrop, som startar en asynkron process och returnerar dess PID. För att kommunicera med en process så anger man dess PID samt meddelandet som ska skickas till processen. För att ta emot ett meddelande så anger man ett kommando som låser processen fram tills dess att ett meddelande tas emot, varefter man kan analysera meddelandet. Det går att sätta en högsta tid som en process ska vänta på ett meddelande.

Flera processer kan inte dela på resurser. Det enda sättet för en process att dela data med en annan är att den ena processen skickar datan till den andra, vilket resulterar i att den mottagna datan är en kopia av den ursprungliga och alltså inte är densamma som den ursprungliga datan. Vill man då att flera processer ska kunna läsa från samma resurs så måste programmeraren själv definiera en kontrollstruktur som fördelar in-datan mellan dem.

Erlangs virtuella maskin har hand om köerna av processer som väntar på att få köras. Den virtuella maskinen anpassar automatiskt köerna så att varje processorkärna får en separat kö, utan att programmeraren behöver bestämma vilken process som körs på vilken processor.

5.3.3 F#

F# har något som kallas för “Async-objekt” som representerar de beräkningar som ska utföras asynkront. Dessa beräkningar startas genom att använda olika startfunktioner som väljer om beräkningarna ska göras på till exempel tråden som körs nu eller en bakgrundstråd. Ett Async-objekt skapas lätt genom att helt enkelt omge det som ska köras asynkront med *async{}*.

Man måste sedan specificera vad som inom Async-objektet som ska köras asynkront. I F# definierar man variabler och metoder med nyckelordet “let” och vill man säga att en metod ska köras asynkront lägger man bara till ett utropstecken, nyckelordet blir alltså “let!”.

5.4 Styrkor och svagheter

5.4.1 Go

Styrkor:

- Syntaktiskt likt Java och C
- Concurrencystöd inbyggt i språket
- Snabb kompileringstid[2]
- Skräpsamling
- Strängar och mappar

Svagheter:

- Ungt språk med oklar framtid
- Få bibliotek
- Relativt dålig prestanda[6]

5.4.2 Erlang

Styrkor:

- Skapat med stark concurrencymodell som mål
- Stöd för anrop av program skrivna i C
- Ett program som exekverar kan ha sin kod utspridd på flera datorer
- Kod kan hot-swappas

Svagheter:

- Ej lämpat för beräkningsintensiva program

5.4.3 F#

Styrkor:

- Tillgång till hela .NET-biblioteket
- Stark, välutvecklad utvecklingsmiljö - Visual Studio
- Multiparadigm - Funktionellt, imperativt och objektorienterat
- Öppen källkod - frivilliga kan hjälpa att utveckla språket

Svagheter:

- Relativt nytt språk med stundtals bristande dokumentation

5.5 Enkelhet att anamma

5.5.1 Go

Go's syntax är en vidareutveckling av C/Java's syntax och det följer samma imperativa modell, vilket gör det lätt att anamma för en Java-utvecklare. Variabel- och funktionsdeklarationer görs i omvänd ordning jämfört med Java, men detta tog det inte lång tid innan vi anammade och vi tror inte heller att det ska ta lång tid att anamma för en erfaren Java-utvecklare. Att funktioner kan returnera tupler är helt främmande från vad som är möjligt i Java och kan till en början vara förvirrande. Att förstå hur man använder kanaler i Go kan liknas med att förstå hur pekare fungerar i C - det är förvirrande till en början men när man har förstått exakt hur det fungerar så är det otroligt kraftfullt. Just kanaler var det som vi hade svårast att anamma när vi började programmera i Go.

Överlag så var Go lätt att anamma och vi tror att folk med en viss kunskap om Java kommer ha det lika lätt som oss. En genomgående känsla vi hade när vi programmerade i Go var att det var som att det var som att programmera i en uppgraderad och förenklad version av Java.

5.5.2 Erlang

För en programmerare utan bakgrund i något annat språk än Java kommer Erlang verka väldigt främmande. Erlang är svagt typat, alla variabler är konstanta, tupler och atomer står för en stor del av koden, det finns ingen objektorientering och syntaxen är helt annorlunda. Programmeringsstilen är också annorlunda så den oerfarne Erlangutvecklaren kommer antagligen att känna sig vilse, till en början.

Har man dock en erfarenhet av programmeringsspråket Prolog så lär man känna sig relativt hemma med Erlang. Mycket av syntaxen är lånad från just Prolog och koncept såsom atomer, tupler, typmatchning och enbart länkade listor skapar den programmeringsstil som Prologutvecklare är vana vid. Något som dock är väldigt annorlunda är att Erlang saknar backtracking, vilket är grundpelaren i Prolog. Rent syntaktiskt så är de båda språken dock väldigt lika.

5.5.3 F#

F# är ett så kallat multiparadigmspråk, det är både funktionellt och imperativt. Funktioner och variabler kan deklareras på ett sätt som gör att de beter sig som de hade gjort i C#, men huvudtanken bakom språket är att använda det på ett funktionellt sätt. För en person som bara har bakgrund i Java så kan F#, precis som Erlang, verka främmande. Att börja skriva kod funktionellt kan vara en stor omställning, med bland annat variabler som inte kan byta värde och funktioner som evalueras snarare än körs rad för rad.

Att F# är med i .NET-familjen och kan använda alla dess bibliotek är något som kan underlätta vid lärandet av F# om man har tidigare erfarenhet av att programmera i .NET-språk. Om man har erfarenhet av andra .NET-språk så har man antagligen också lite erfarenhet av Visual Studio, Microsofts utvecklingsmiljö för .NET-språk.

F# har hämtat mycket inspriration från programmeringsspråket Haskell, så en bakgrund med Haskell eller liknande funktionella språk underlättar vid lärandet av F#.

6. Diskussion

6.1 Varför språk med inbyggt stöd för concurrency

Java's implementation av trådar är omständlig, relativt den hos Go, Erlang och F#.
De funktioner som ska köras som trådar måste wrappas av en egen trådklass och kommunikationen mellan dem är komplicerad då utvecklaren själv måste hålla koll på vilka resurser de delar och se till så att de inte använder dessa samtidigt.

Vad vi upptäckte under projektets gång var att dessa problem som finns i Java försvinner när man programmerar i Go, Erlang eller F#. Att utveckla multitrådade program blir mycket mer intuitivt och mindre krävande av programmeraren, vilket i sin tur medför att man kan skriva multitrådade program mycket mer effektivt. Detta anser vi vara aktuellt då man kan se en trend av att allt fler program körs på servar med många användare samtidigt, vilket medför att man måste programmera multitrådat och ha en god grund för concurrency.

6.2 Vad man kan förvänta sig av språken

6.2.1 Go

Go är ett väldigt flexibelt språk som kan användas i många olika situationer. Det har en blandning av effektiv exekvering och enkel programmering, vilket gör att det inte alltid är bäst för en specifik uppgift men att det kan användas för att utveckla alla delar i ett stort projekt.

Har man en bakgrund inom Java/C-programmering så är Go lätt att lära sig och använda. Det färdiga programmet är kompilerat, vilket medför att programmeraren själv ansvarar för att programmet går att köra på en viss plattform.

Det finns än så länge ingen utvecklingsmiljö som är anpassad för Go, så man får som programmerare nöja sig med en kombination av texteditor/terminal.

Go är väldigt nytt och utvecklas löpande av Google och ett hundratal frivilliga bidragare. För tillfället så har språket få bibliotek[7], men efter att ha samlat in åsikter från forum, bloggar och artiklar så har vi upplevt att det allmänna intresset för Go är starkt och förutspår att antalet bibliotek kommer växa inom de närmsta åren, främst på grund av att Google har en historia av att utveckla icke-vinstdrivande projekt, att det i nuläget finns ca. 200 frivilliga bidragare och att det finns en allmän nyfikenhet för språket.

Även om det är ett generellt språk så är det inte lämpligt att skriva grafiska applikationer på klientsidan i Go. Det finns en handfull grafiska bibliotek utvecklade av tredje parter, men de bibliotek som är direkt anpassade för Go är fortfarande primitiva.

Under projektets gång upplevde vi att den officiella dokumentationen för Go[8] är välskriven och lättförstådd. Dessutom hade vi lätt för att hitta information på forum och bloggar. Något man bör vara medveten om är att "Go" är ett sökord som genererar många sökträffar som inte är relaterade till just Go (vi hade mer tur med att använda sökordet "Golang").

6.2.2 Erlang

Erlang är inte ett mångsidigt språk. Det är utvecklat för att skapa snabba och lättöverskådliga multitrådade program som delegerar ut beräkningsintensiva uppgifter till program skrivna i C. Dessa program måste i sin tur vara speciellt anpassade för att interagera med Erlang. Vill man göra något annat än att transportera och behandla meddelanden så bör man antagligen se till andra programmeringsspråk.

Erlang är väldigt annorlunda från imperativa språk som Java och C. Det lånar dock många drag från Prolog och funktionella språk, så en bakgrund inom dessa eller liknande språk kan vara till stor fördel när man försöker lära sig Erlang.

Språket har funnits länge och är väletablerat i industrin. Erlangs officiella dokumentation[9] är formell och kortfattad, vilket kan göra den svårförstådd. Det kan överlag vara svårt att hitta dokumentation från tredje parter på forum och bloggar.

Program körs genom en interpreter och är relativt operativsystemsoberoende. Det går dessutom att sprida ut ett programs kod över flera datorer. Denna kod kan sedan hot-swappas under körning.

Det finns flera olika utvecklingsmiljöer att välja mellan när man ska programmera i Erlang, bland annat ett officiellt mode till Emacs och ett plugin till Eclipse[10]. Dessvärre så undersökte vi dem inte inför denna rapport.

6.2.3 F#

F# tillhör .NET-familjen och på grund av att .NET är ett så välutvecklat ramverk med ett stort klassbibliotek[11] så blir F# ett programmeringsspråk som kan användas i en mängd olika situationer. Tillgången till .NET medför också att F# lätt kan integreras med andra språk som också ingår i .NET-familjen.

F# finansieras av Microsoft och finns sedan 2010 implementerat i Visual Studio som ett av huvudspråken. Språket utvecklas konstant, inte bara av Microsoft utan även av frivilliga bidragare då källkoden till F# är öppen.

Att språket finns med i Visual Studio och att Microsoft släppte källkoden visar på deras engagemang och tilltro till språket. Lägg därtill närheten till de andra språken i .NET-familjen, där de flesta är starkt etablerade på marknaden, så kan nog F# själv bli ett stort språk.

6.3 När bör man använda språken

6.3.1 Go

Så gott som alla delar av ett system kan skrivas i Go. Det finns stöd för beräkningar nere på bit-nivå samtidigt som många delar av språket håller en hög abstraktionsnivå. Meningen med utveklingen av Go har beskrivits som ett försök att skapa ett generellt språk som utnyttjar styrkan hos multicore-processorer samtidigt som det ska göra det lättare för programmerare att utveckla sina produkter[2]. Efter att ha studerat Go närmare så är vi beredda att hålla med om att det i alla fall gör det lättare att utveckla i.

Något som man bör vara väldigt medveten om är att Go fortfarande är väldigt ungt och inte än når upp till den prestanda som kan fås av program skrivna i C och Java[12]. Vi tror och hoppas att Go's prestanda kommer bli allt bättre ju mer tid utvecklarna får att förbättra kompilatorerna, men i nuläget så är prestandan inte en anledning till att vilja utveckla system i Go.

Eftersom Go är ett språk med målet att förbättra produktiviteten hos Java - och enligt oss har lyckats - så är det i nuläget främst produktivitet som är den enda anledningen till att använda Go. Dess framtid vilar helt på ifall det kommer att användas inom industrin. Om intresset för Go faktiskt finns och många projekt skrivs i språket så tror vi att det är en stark utmanare till Java. Om inte så kommer det antagligen att glömmas bort och hamna i skymundan, som så många andra språk har gjort. Därför vill vi inte uppmana läsaren till att använda Go till stora projekt som kommer att behöva underhållas under många år framöver, då dess framtid är så pass oklar. Däremot uppmanar vi läsaren till att själv utforska det eftersom det tillför många nya koncept, samtidigt som det är väldigt likt Java och C.

6.3.2 Erlang

Om man ska utveckla ett stort system där många noder ska vara sammankopplade samtidigt så är Erlang ett bra alternativ. Det lämpar sig väldigt väl för system som inte utför särskilt beräkningsintensiva uppgifter, men som kräver att man ska ha en lättöverskådlig och stabil kontroll över alla uppkopplingar och kommunikationen mellan dem. Med detta så är det inte sagt att systemet inte ska ha möjlighet att utföra mer beräkningsintensiva uppgifter, detta måste dock skötas av program skrivna i C.

Erlang bör främst ses som ett styrsystem som kan skapa och övervaka många processer, inte som ett generellt programmeringsspråk som lämpar sig för många uppgifter. Några exempel på system skrivna i Erlang är Facebook-chattens backend[13], Yahoo's tjänst Delicious[14] samt många telecom-tjänster, vilket kan ge en klarare bild av Erlang's användningsområde.

Användningsområdet för Erlang kan lättast beskrivas med ett citat från Tim Bray, ledaren för webbtekniker på Sun Microsystems: "*If somebody came to me and wanted to pay me a lot of money to build a large scale message handling system that really had to be up all the time, could never afford to go down for years at a time, I would unhesitatingly choose Erlang to build it in.*" [15]

6.3.3 F#

Brian, en utvecklare på F# Team hos Microsoft skriver på sin blogg: "*In a given blog post, or talk, or video on F#, it's easy (and indeed often appropriate) to "pigeonhole" the language, focusing on one particular useful aspect. But on occasion it's useful to step back and see the myriad ways that people perceive and use F#.*"[16] Vad han menar är att F# har inte en speciell funktion som språket är bäst på utan det är bra på väldigt mycket. Han ger en lista på vad han anser vara anledningar till att använda F# och i listan hittar vi bland annat vad som kallas för "unit of measure".

Unit of measure innebär att programmeraren har möjlighet att binda en enhet till en variabel, till exempel svenska kronor, meter eller kilogram. Detta visar sig vara bra att kunna göra när programmeraren måste hålla reda på många olika enheter som ska interagera med varandra på ett speciellt sätt. Någon som räknar på att bygga broar får inte blanda ihop meter med millimeter

och i finansvärden får man inte blanda ihop svenska kronor med amerikanska dollar. F# hjälper programmeraren genom att ge felmeddelanden om man blandar enheter på ett sätt som man själv inte specificerat.

Detta var dock bara ett exempel på en av styrkorna som F# har som många andra programmeringsspråk inte har. Eftersom F# både är funktionellt och imperativt så blir F# ett väldigt brett språk och skulle man märka att F# inte räcker till så är inte till exempel C# långt borta och inte svårt att nästla in i sitt F#-program, vilket också Brian tar upp i sin lista. Man får tillgång till alla fördelar som funktionell programmering innebär medan man samtidigt lätt till exempel kan ansluta till databaser skrivna i C#.

7. Källförteckning

1. *Programming Language Popularity* [hemsida]. c2011 [uppdaterad 2011 apr 13; citerad 2012 apr 12]. Tillgänglig på: http://langpop.com/
2. *Go: FAQ* [hemsida]. Inget datum [citerad 2012 apr 12]. Tillgänglig på: http://golang.org/doc/go_faq.html
3. Cesarini F, Thompson S. *Erlang Programming.* Sebastopol: O'Reilly Media, 2009; p. 3
4. *Announcing the F# Compiler + Library Source Code Drop* [hemsida]. c2010 [uppdaterad 2010 nov 23; citerad 2012 apr 12]. Tillgänglig på: http://blogs.msdn.com/b/dsyme/archive/2010/11/04/announcing-the-f-compiler-library-source-code-drop.aspx?wa=wsignin1.0
5. *Hevlet Packard. Produktspecifikationer för HP Pavilion dv6-1320eo Entertainment notebook-datorer* [hemsida]. Inget datum [citerad 2012 apr 12]. Tillgänglig på: http://h10025.www1.hp.com/ewfrf/wc/document?docname=c01945575&tmp_task=prodinfoCategory&cc=se&dlc=sv&lang=sv&lc=sv&product=4064674/
6. *C, Erlang, Java and Go Web Server performance test* [hemsida]. c2009 [uppdaterad 2009 nov 11; citerad 2012 apr 12]. Tillgänglig på: http://timyang.net/programming/c-erlang-java-performance/
7. *Go Language Resources* [hemsida]. Inget datum [citerad 2012 apr 12]. Tillgänglig på: http://go-lang.cat-v.org/pure-go-libs
8. *Go: Documentation* [hemsida]. Inget datum [citerad 2012 apr 12]. Tillgänglig på: http://golang.org/doc/
9. *Erlang: Documentation* [hemsida]. Inget datum [citerad 2012 apr 12]. Tillgänglig på: http://www.erlang.org/doc/
10. *Erlang: FAQ: Tools* [hemsida]. Inget datum [citerad 2012 apr 12]. Tillgänglig på: http://www.erlang.org/faq/tools.html
11. *.NET Framework Class Library* [hemsida] Inget datum [citerad 2012 apr 12]. Tillgänglig på: http://msdn.microsoft.com/en-us/library/ms229335.aspx
12. Hundt R. *Loop Recognition in C++/Java/Go/Scala* [rapport online]. Inget datum [citerad 2012 apr 12]. Tillgänglig på: https://days2011.scala-lang.org/sites/days2011/files/ws3-1-Hundt.pdf
13. *Facebook Chat* [hemsida]. c2008 [uppdaterad 2008 maj 14; citerad 2012 apr 12]. Tillgänglig på: http://www.facebook.com/note.php?note_id=14218138919
14. *Delicious* [hemsida]. Inget datum [citerad 2012 apr 12]. Tillgänglig på: http://www.delicious.com/
15. Cesarini F, Thompson S. *Erlang Programming.* Sebastopol: O'Reilly Media, 2009; p. 2
16. *Nine reasons to use F#* [hemsida]. c2010 [uppdaterad 2010 apr 1; citerad 2012 apr 12]. Tillgänglig på: http://lorgonblog.wordpress.com/2010/04/01/nine-reasons-to-use-f/

Appendix A - Definitioner

Term	Förklaring
atom	Ord som är lånat från logiken - en variabel som är unik och global. Kan liknas med enum från Java.
backtracking	Förmågan hos ett programmeringsspråk att gå tillbaka i koden (“backtracka”) för att testa nya värden på variabler.
benchmark-test	Att köra ett program med syfte att utvärdera dess prestanda.
concurrency	Förmågan att utföra flera beräkningar samtidigt, samt förmågan att kommunicera mellan dem.
dynamiskt typat	Ett dynamiskt typat språk utför automatiskt typkonverteringar mellan variabler, så att t.ex. en sträng kan tolkas som ett heltal.
Eclipse	En utvecklingsmiljö för programmering, främst för Java.
Ericsson	Telecom-företag.
funktionellt språk	Programmeringsspråk där funktioner kan vara parametrar.
imperativt språk	Programmeringsspråk där koden ses som en sekvens av instruktioner.
JVM	Java Virtual Machine, en virtuell maskin som exekverar program som kompilerats till Java-bytekod.
PID	Förkortning för Process ID - det unika identifikationsnummer som ges till varje process.
skala väl	Ett program som fungerar väl för både små och stora indata.
starkt typat	I ett starkt typat språk är variablers typ väl definierade och går inte att blanda utan implicita typkonverteringar.
tråd	En process som kan dela resurser med andra trådar.
tupel	En mängd variabler som i sig kan betraktas som en variabel, inte helt olikt en lista med konstant längd.
typ-härledande	När ett språk är typ-härledande behöver man inte specificera variablers datatyp, utan kompilatorn listar automatiskt ut det vid kompilering.
mönstermatchning	Förmågan hos ett språk att själv kunna förstå vart programmet ska fortsätta genom att jämföra mönstret hos indatat med de möjliga alternativen.

Appendix B - Källkod

Go - Benchmark

```
package main

import (
	"fmt"
	"os"
	"strconv"
	"time"
)

func main() {
	n_threads, _ := strconv.Atoi(os.Args[1])
	n_loops, _ := strconv.Atoi(os.Args[2])
	c := make(chan int, n_threads)

	start_time := time.Now()

	// Spawn n_threads number of threads
	for i := n_threads; i > 0; i-- {
		go loop(n_loops, c)
	}

	// Wait for all threads to finish
	i := 0
	for i < n_threads{
		<- c
		i++
	}

	// Collect time taken to finish
	duration := time.Since(start_time).Nanoseconds()/1000000
	fmt.Println(duration)
}

// Loops n_loops number of times, then sends a signal
// to the channel c
func loop(n_loops int, c chan int){
	ret := 0
	for i := n_loops; i > 0; i-- {
		ret += 1
```

```
	}
	c <- 1
}
```

Go - Chat Server

```
package main

import (
	"fmt"
	"net"
	"os"
	"container/list"
)

// A list that holds all connected clients
var clientList = list.New()

func main() {
	fmt.Println("Server started.")

	host := "localhost"
	port := "1234"

	// Opens up a port
	lis, _ := net.Listen("tcp", host + ":" + port)

	// Listens for incoming connections and spawns a
	// client process for each accepted connection
	for {
		con, _ := lis.Accept()
		fmt.Println("Client connected.")

		clientList.PushFront(con)
		go clientListener(con)
	}
	lis.Close()
}

// Listens for incoming messages from the port
func clientListener(con net.Conn) {
	data := make([]byte, 1024)
	loop := true
```

```
	for loop {
		n, error := con.Read(data)
		switch error {
			// Client disconnects, ends the loop
			case os.EOF:
				fmt.Println("Client disconnected.")
				loop = false
			case nil:
				fmt.Print(string(data[0:n]))
				sendToAll(data[0:n])
		}
	}

	// Remove client from list
	remove(con)
	con.Close()
}

// Sends a message to all connected clients
func sendToAll(msg []uint8){
	this := clientList.Front()
	for {
		if this == nil {
			return
		}

		this.Value.(net.Conn).Write(msg)
		this = this.Next()
	}
}

// Removes a client from the list of clients
func remove(con net.Conn) {
	this := clientList.Front()
	for {
		// End of loop
		if this == nil{
			return
		}
		if this.Value == con{
			clientList.Remove(this)
		}
		this = this.Next()
	}
}
```

Go - Chat Client

```
package main

import (
	"fmt"
	"net"
	"os"
)

func main() {
	host := os.Args[1]
	port := os.Args[2]
	name := os.Args[3]

	// Connect to the server
	addr, _ := net.ResolveTCPAddr("tcp", host + ":" + port)
	con, _ := net.DialTCP("tcp", nil, addr)

	// Spawn listener thread
	go listener(con)

	// Forever listen for input from the terminal and
	// sends it to the socket
	var in = make([]byte, 1024)
	for {
		n, _ := os.Stdin.Read(in)
		msg := name + ": " + string(in[0:n])
		con.Write([]uint8(msg))
	}
	con.Close()
}

// Listens for incoming messages on the port and
// prints them to the terminal
func listener(con net.Conn) {
	var in = make([]byte, 1024)

	for {
		n, _ := con.Read(in)
		fmt.Print(string(in[0:n]))
	}
}
```

Erlang - Benchmark

```
-module(proc).
-export([time/2, start/2, loop/2]).

% Measures the amount of time taken for all the processes
% to terminate
time(Threads, Loops) ->
	{Ms, _} = timer:tc(proc, start, [Threads, Loops]),
	round(Ms/1000).

% Creates a number of threads that loops a number of times
start(Threads, Loops) ->
	case Threads >= 0 of
		true ->
			spawn(fun() -> loop(Loops, 0) end),
			start(Threads - 1, Loops);
		false ->
			ok
	end.

% A simple loop looping for a number of times
loop(Loops, Val) ->
	case Loops > 0 of
		true ->
			loop(Loops - 1, Val + 1);
		false ->
			ok
	end.
```

Erlang - Chat Server

```
-module(server).
-export([start/1]).

-define(TCP_OPTIONS, [binary, {packet, 0}, {active, false}, {reuseaddr, true}]).

% Starts the server
start(Port) ->
    {ok, LSocket} = gen_tcp:listen(Port, ?TCP_OPTIONS),
              Sender = spawn(fun() -> sender([]) end),
    accept(LSocket, Sender).

% An infinite loop that listens for connections on
% the port. A new client process is spawned for
% each incoming connection attempt
accept(LSocket, Sender) ->
    {ok, Socket} = gen_tcp:accept(LSocket),
    Sender ! {add, Socket},
              spawn(fun() -> loop(Socket, Sender) end),
    accept(LSocket, Sender).

% Sends a message to all sockets in the socket list
send_all(Sockets, Message) ->
       case Sockets of
              [] ->
                     ok;
              [Socket|T] ->
                     gen_tcp:send(Socket, Message),
                     send_all(T, Message)
       end.

% A process that manages the adding/removing of clients and
% the sending of messages
sender(Sockets) ->
              receive
                     {add, Socket} ->
                            io:fwrite("Client added.~n"),
                            sender(Sockets ++ [Socket]);
                     {remove, Socket} ->
                            io:fwrite("Client removed.~n"),
                            sender(Sockets -- [Socket]);
                     {send, Message} ->
                                   io:fwrite("Message recieved: ~s", [Message]),
                            send_all(Sockets, Message),
```

```
                            sender(Sockets)
            end.

% A client process that listens for incoming messages coming
% through the port
loop(Socket, Sender) ->
    case gen_tcp:recv(Socket, 0) of
        {ok, Data} ->
            Sender ! {send, Data},
                                loop(Socket, Sender);
        {error, closed} ->
            Sender ! {remove, Socket},
                                ok
    end.
```

Erlang - Chat Client

```
-module(client).
-export([start/3]).

-define(TCP_OPTIONS, [binary, {packet, 0}, {active, false}]).

% Starts the client
start(IP, Port, UserName) ->
	{ok, Socket} = gen_tcp:connect(IP, Port, ?TCP_OPTIONS),
	spawn(fun() -> listener(Socket) end),
	loop(Socket, UserName).

% An infinite loop that listens for input from the
% command line and sends the input to the socket
loop(Socket, UserName) ->
	Msg = io:get_line(standard_io, "> "),
	gen_tcp:send(Socket, UserName ++ ": " ++ Msg),
	loop(Socket, UserName).

% An infinite loop that listens for incoming messages
% from the port and prints them to the terminal
listener(Socket) ->
	case gen_tcp:recv(Socket, 0) of
		{ok, Msg} ->
			io:fwrite("~s", [Msg]),
			listener(Socket);
		{error, Reason} ->
			io:fwrite("Server went down. ~s~n", [Reason]),
			ok
	end.
```

F# - Benchmark

```
#light
open System.Threading
open System.Diagnostics

let nrThreads = 5000
let countTo = 1000000

let rec threads(a) =

    if a = countTo then
        ()
    else
        threads (a + 1)

let thread x =
    let a = 1
    let b = threads a
    b

let run() =
    let t = new Thread(new ThreadStart(thread))
    t.Start()

let stopWatch = Stopwatch.StartNew()

for i = 1 to nrThreads do
    run()

stopWatch.Stop()
printfn "%s: %f" "Total time" stopWatch.Elapsed.TotalMilliseconds

// Make sure that the window won’t close on termination
while(true) do
    ()
```

F# - Chatserver

```
#light
open System
open System.IO
open System.Net
open System.Net.Sockets
open System.Threading
open System.Collections.Generic

type ClientTable() = class
    let clients = new Dictionary<string, StreamWriter>()

    // Add a client and it's stream writer
    member t.Add(name, sw:StreamWriter) =
        lock clients (fun () ->
            if clients.ContainsKey(name) then
                sw.WriteLine("ERROR - Name already in use")
                sw.Close()
            else
                clients.Add(name, sw))

    /// Close and remove a client
    member t.Remove(name) =
        lock clients (fun () -> clients.Remove(name) |> ignore)

    /// Get a copy of the current client list
    member t.Current =
        lock clients (fun () -> clients.Values |> Seq.toArray)

    /// Check if client exists
    member t.ClientExists(name) =
        lock clients (fun () -> clients.ContainsKey(name))
end

type Server() = class
    let clients = new ClientTable()

    let sendMessage name message =
        let combinedMessage =
            Printf.sprintf "%s: %s" name message
        for sw in clients.Current do
            try
                lock sw (fun () ->
                    sw.WriteLine(combinedMessage)
                    sw.Flush())
            with
            | _ -> () // Some clients may fail
    let emptyString s = (s = null || s = "")

    // Colon explains the type
    let handleClient (connection : TcpClient) =
```

```
        let stream = connection.GetStream()
        let sr = new StreamReader(stream)
        let sw = new StreamWriter(stream)

        let rec requestAndReadName() =
            sw.WriteLine("What is your name? ");
            sw.Flush()

            let name = sr.ReadLine()

            if clients.ClientExists(name) then
                sw.WriteLine("ERROR - Name already in use")
                sw.Flush()
                requestAndReadName()
            else
                name
        let name = requestAndReadName()
        clients.Add(name, sw)

        let rec listen() =
            let text = try Some(sr.ReadLine()) with _ -> None
            match text with
            | Some text ->
                if not (emptyString(text)) then
                    sendMessage name text
                Thread.Sleep(1)
                listen()
            | None ->
                clients.Remove name
                sw.Close()
        listen()

    let server = new TcpListener(IPAddress.Loopback, 1234)

    let rec handleConnections() =
        server.Start()
        if (server.Pending()) then
            let connection = server.AcceptTcpClient()
            printf "New Connection"
            let t = new Thread(fun () -> handleClient connection)
            t.Start()
        Thread.Sleep(1)
        handleConnections()

    member server.Start() = handleConnections()
end
(new Server()).Start()
```

F# - Chat Client

```
#light
open System
open System.ComponentModel
open System.IO
open System.Net.Sockets
open System.Threading

let tc = new TcpClient()
tc.Connect("localhost", 1234)

let run() =
    let sr = new StreamReader(tc.GetStream())
    while(true) do
        let text = sr.ReadLine()
        if text <> null && text <> "" then
            printfn "%s" text
let t = new Thread(new ThreadStart(run))
t.Start()

let sw = new StreamWriter(tc.GetStream())

while true do
    sw.WriteLine(Console.ReadLine())
    sw.Flush()
```