DEGREE PROJECT, IN INFORMATION TECHNOLOGY , FIRST LEVEL
STOCKHOLM, *SWEDEN 2015*

Evaluation of Golang for High Performance Scalable Radio Access Systems

FILIP FORSBY, MARTIN PERSSON

KTH ROYAL INSTITUTE OF TECHNOLOGY

ICT - SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Sammanfattning

Den ökande mobildataanvändningen sätter större press på de nuvarande teknologier som används i dagens radioaccessystem och nya lösningar behövs för att tillfredsställa de nya kraven. Det är därför viktigt att utvärdera uppkommande teknologier, vilket även inkluderar programmeringsspråk, för att betsämma dess lämplighet för den här typen av användingsområde.

Golang är ett nytt programmeringsspråk som ännu inte har blivit utvärderat. Den här rapporten har som syfte att genomföra en utvärdering av Golang för användning i högpresterande skalbara radioaccessystem. För att göra detta utvecklades en applikation från en redan existerande model skriven i Erlang, och de två implementationerna testades och jämfördes med specifika nyckelvärden i åtanke.

Resultaten visar att Golang presterar bra och har potential att vara en god kandidat för framtida system. Däremot visar sig språket att inte vara helt moget och saknar viktig funktionalitet och behöver vidareutvecklas för att bli väl lämpat för denna specifika applikation.

Abstract

Increasing mobile data traffic puts pressure on the current technologies used in today's radio access units and new solutions are needed in order to cope with the greater demands. It's therefore important to evaluate emerging technologies, including programming languages, to determine their suitability for this field of application.

Golang is one of these new programming languages that have not yet been evaluated. This thesis has the purpose to perform an evaluation of Golang used in a high performance scalable radio access system. To do this, an application is developed from an already existing model written in Erlang and the two implementations are compared and benchmarked with specific key aspects in mind.

The results show that Golang performs well and has the potential to be a good candidate for future systems. However, the language is found to not yet be fully mature and lacks important functionality required and needs to be further developed in order to be fully suitable for this specific application.

Table of contents

List of figures

1 Introduction

The mobile phone usage is greatly increasing in today's society, which goes hand in hand with the increased spread of smartphones [1]. The increased usage leads to a greater data flow between phones and base stations, which puts higher demands on the systems providing and supporting this traffic.

1.1 Background

The modern mobile communication network, LTE, is built up upon an infrastructure of several layers and protocols. Radio Resource Control (RRC) layer is part of the third layer in the LTE-stack. The RRC layer is responsible for setting up a context for each User Equipment (UE) that requests to connect to the network. The connection set up is handled by a base station (eNodeB) communicating with both the UEs and the network backbone. During this phase, several messages are sent containing information about the UE and the network configuration and are used for making decisions about the configuration of the connection that is being set up.

1.2 Problem

When new programming languages emerge, it is important to evaluate them to get an idea of its capabilities and performance qualities from different aspects that are of interest for both new and existing products and implementations.

Golang is one of these new languages that haven’t yet been evaluated when it comes to concurrent execution in high performance radio access systems. It is therefore interesting for tele-communication companies to have this evaluation performed in order to make future decisions in whether to consider using Golang, or just to get knowledge of the existing languages and its capabilities.

The application that this evaluation is performed upon is the Radio Resource Control (RRC) layer of the Long Term Evaluation (LTE) Radio Access Network (RAN). A simplified model of the application already exists and is written in Erlang. The task of this thesis is to develop a model in Golang corresponding to the one written in Erlang and to do tests upon the both and evaluate the results.

1.3 Purpose

The purpose of this thesis is to produce material that will be considered in future development as a basis for making decisions when it comes to programming languages and performance evaluation. The material produced should include certain measurements of performance with different Key Performance Indicators (KPIs). These KPI's should be evaluated and compared to the KPI's of the current implementation and result in a comparison between the implementations and a reflection about the results.

The purpose of this report is to describe the work that has been done in this thesis. The report is designed to give the reader a good idea of what the problem is, why the problem is of interest and what have been accomplished in terms of solving the problem.

1.4 Goal

This thesis has the goal to produce a similar model in the Golang programming language of the existing application handling the RRC protocol written in Erlang. The model will have to be precise enough to be able to make accurate assumptions and a fair comparison between the existing implementation and the implementation written in Golang.

Another goal is to evaluate the Golang language by itself and do a performance analysis which will lead to accurate KPI's that describe the characteristics of the language and a picture of how well the language is performing in general, without respect to the implemented RRC layer application. The goal is to have these KPI's to be as accurate as possible in order to both have a baseline to compare with in the model evaluation but also to make the measured performance correspond to the actual performance of the language.

Hopefully, this thesis will bring clarity about the suitability for Golang in a concurrent execution in high performance radio access systems and that the documentation will make clear what are the strengths and weaknesses.

1.5 Methodology

The methodology used in this work is an agile version of an iterative waterfall model. Since the work is done only by two people, it's agile by default because there is no need for a strict plan and changing a previous made decision is easy and doesn't directly involve or affect other people.

The time plan that was written in the beginning of the work took into consideration that certain parts needs to be done before other, but most likely would need to be revisited later on, making it an iterative waterfall model.

Chapter 3 describes more in detail how this project model were realized and taken into action.

1.6 Scope

The scope of this thesis has been narrowed down to only evaluate Golang on one specific application, the RRC application, which should be simplified in order to fit in the given timeframe but still provide valuable outcome. The application will be simplified in the way that only an attach and detach sequence is implemented and only the performance of the eNodeB is evaluated.

The study that is done in this thesis is a strict quantitative study and is only evaluating the aspects of Golang that can be measured, such as time consumption and memory usage. No attention has been made in order to evaluate the language in a qualitative point of view, with respect to ease of programming, community support or productivity.

1.7 Outline

To understand the methods and technologies this thesis builds on, the theoretical background in chapter 2 explains both Golang and Erlang as independent programming languages, followed by a description of the LTE network. Chapter 3 describes how this thesis is performed and the reason behind the choices that were made in order to fulfil the goals. The following chapter presents the solutions and implementations that were made, how the model of the RRC application is built and also the benchmarks that were performed on each languages individual core features. The results of the benchmarks and measurements are provided in chapter 5 with discussion and conclusions made from the results are displayed in chapter 6.

2 Theoretical background

2.1 Long Term Evolution

Long Term Evolution (LTE), more commonly referred to as 4G, is a standard for high-speed wireless communication for mobiles [2]. The LTE network architecture consists of the Evolved UMTS Terrestrial Radio Access Network (E-UTRAN) and the core network, Evolved Packet Core (EPC) [3]. The LTE Protocol is divided into two categories, the control plane which is used for configuring connections, which is utilized by the traffic plane to transfer data.

The E-UTRAN handles communications between the UEs and the EPC, and consists of the base stations (eNodeBs). An eNodeB contains one or more "cells" which each provides coverage to a geographic area and an UE communicates with one cell at a time. The base station provides the LTE network with two main functionalities, transmitting data to and from UEs, and sending and Radio Resource Control (RRC) signals used by the LTE control plane.

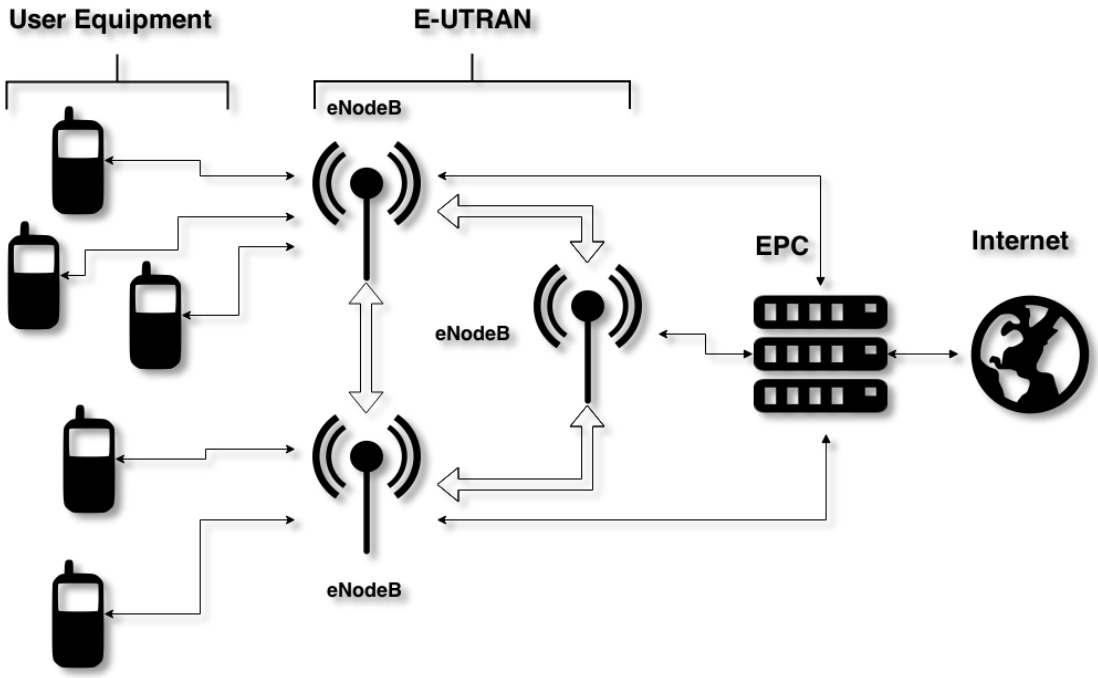

Figure 2:1 – LTE topology

2.2 Radio Resource Control

The Radio Resource Control (RRC) protocol is part of the LTE air interface control plane and is implemented in the eNodeB's as well as in the UEs [4]. The purpose of the protocol is to handle the control plane signaling between the User Equipment (UE) and the E-UTRAN. These signals consists of connection establishment requests, disconnect messages, handovers between eNodeB's as well as setting up radio bearers for data transfer.

2.3 Golang

Golang is a programming language developed by Google with the idea [5] that all the present major languages were both too difficult to program in or too inefficient when it comes to compilation and performance and a new programming language was needed. Golang was therefore designed "with an eye on felicity of programming, speed of compilation, orthogonality of concepts, and the need to support features such as concurrency and garbage collection." [6] The aim was to become as powerful as C but as easy as Python or JavaScript.

Golang started as a Google internal project in late 2007 [7] and is now an open source project that became public 2009 with the first stable release in 2012. The language has a big support community and has since its release had several big updates.

Golang programs run with a built in runtime system that has garbage collection, process scheduling and supports the build in primitives such as buffers and locks which are the base of the channels that are discussed later in this section.

2.3.1 Goroutines

A goroutine is the Golang version of a lightweight process. A goroutine is not visible to the operating system, it's only visible inside the virtual machine and to the own Golang scheduler. The goroutines are very lightweight because of their small memory footprint with a 2 KB stack [8], no separate heap or data area and no direct map to a thread in the OS level. The fact that it's independent from OS threads makes context switches between two goroutines fast and leaves more control to the built in scheduler to make more beneficial scheduling decisions.

2.3.2 Channels

As a part of making concurrent programming easier and a native part of the language, Golang has a feature called channels. A channel is a message passing interface for inter-process communication within the local program instance, where a goroutine can write or read to one or several channels to synchronize with other goroutines.

Sending data through a channel doesn't necessarily mean that the whole data is sent. For example, sending a string through a channel only sends a header with pointer to the actual data.

2.3.3 Scheduler

The Golang scheduler is a work stealing scheduler [9] that schedules one or more goroutines to one or more OS thread. The scheduler has three types of major structures called G, P and M.

G stands for goroutine and is the most fine grained of them all. P is for process and has one single G at the time that is currently executing and holds several other Gs in a queue that is ready for execution. M is short for machine and represents a thread in the OS and thus not scheduled by Golang. Each P is assigned to an M to get execution time in the processor from the OS. The reason for having both M and P [10] is that sometimes the M gets blocked by the OS for different reasons, such as when a system call is made. When the M is blocked by the OS, the scheduler assigns that current P to another M so that another G in the queue that is not blocking could get executed.

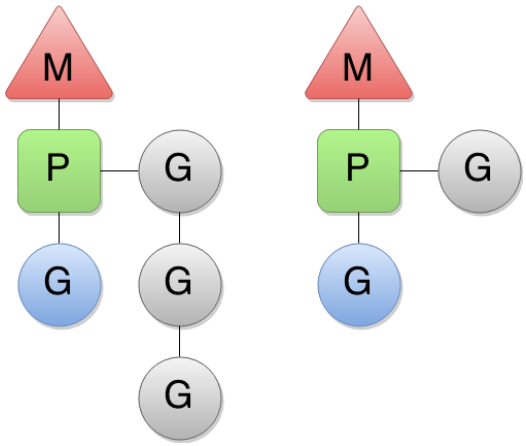

Figure 2:2 – The Golang scheduler data structures

In the example above, there is two Ps mapped to one M each. Both of the Ps have one active G and the left P has 3 Gs in the queue while the right P has one G in the queue.

The work stealing is performed when there is a P that has ran out of available Gs in the ready queue and is by that out of work. In order to have all the Ps active, the P without any Gs steals Gs from a P that has many G in its ready queue.

The Golang scheduler is non preemptive, which means that a goroutine runs until it yields and puts itself in the wait queue. This can lead to problems if a goroutine never suspends itself, such as starvation or memory problems.

2.4 Erlang

Erlang is developed by Ericsson and is a functional programming language. The first prototype of Erlang was released in 1987 [11] and was first available for Ericsson internal use only but is since 1998 available to the public as open source [12].

The main focus of Erlang is to be fault tolerant and highly concurrent for running systems that never go down and handles many contexts at the same time. Erlang is therefore built upon a virtual machine (VM) with its own garbage collector and lightweight processes that run independent from each other with no shared memory.

2.4.1 Lightweight process

Much like Golang, Erlang has its own lightweight processes that are scheduled by the runtime environment in the VM and not by the OS. To be able to have many processes running at the same time, the size of each process has to be small. A newly spawned process in Erlang uses 309 words of memory [13], which equals about 2,4 KB on a 64-bit system.

2.4.2 Message passing

The message passing functionality is a built in feature of Erlang, where every process has its own input buffer. Processes can then send messages to any process with the built in operator '!'. The message passing is asynchronous and the whole message is copied so the receiver has its own copy. All this is to

ensure stability so that no process should be relying on another, so a crash in one process will not affect the other.

2.4.3 Scheduler

The scheduler that runs in the Erlang VM is a preemptive scheduler that forces processes to stop executing if it doesn't suspend itself in time. Each process has a counter that is reset each time its scheduled and decremented for each recursion. A process is preempted (forced to stop) when the counter reaches zero.

2.5 Abstract Syntax Notation One

Abstract Syntax Notation One (ASN.1) is a standardized way to describe messages to be transferred between systems [14]. The standard is divided into two parts:

1. The rules of syntax for describing the contents of a message in terms of data type and content sequence or structure.
2. How the encoding of each data item in a message should be done.

The syntax is abstract and designed to be able to handle any datatype and isn't based on a specific programming language or software. The standard enables transferring data structures between any systems on any hardware that implements the standard.

3 Methodology

3.1 Micro benchmarks

In order to be able to determine the suitability of using Golang in the RRC application, micro benchmarks were developed to examine the differences in critical properties between the languages in a system perspective. For the RRC-application these properties consisted mostly of the concurrent properties of the languages but also application specific properties such as encoding and decoding outgoing and incoming messages. For each of the developed benchmarks, runtime data was collected in order to compare the languages for these properties.

The result of these benchmarks is a Key Performance Indicator (KPI) where a value is assigned to each of these critical properties. These KPIs are used as a baseline when it comes to estimating the differences between the languages in critical functions of the RRC application. In addition to estimating the performance of specific functionality this baseline also grants the ability to validate more complex self-developed code by having numbers of certain properties.

The KPIs of interest based on the present model of the RRC application are the following:

- Signaling cost between processes
- Message passing cost between processes
- Creation cost of processes
- The cost of encoding/decoding ASN.1 messages

The reasoning behind the choices of KPIs origins from the process diagram of the present implementation, Figure 3:1. Every user equipment (UE) connected to an eNodeB is represented by a set of processes, which explains the importance of a low process creation cost. All external communications to and from the eNodeB are done using ASN.1 encoded messages and latencies are highly dependent on performance in this area. Internal communications are done using the message passing primitive available in the programming language.

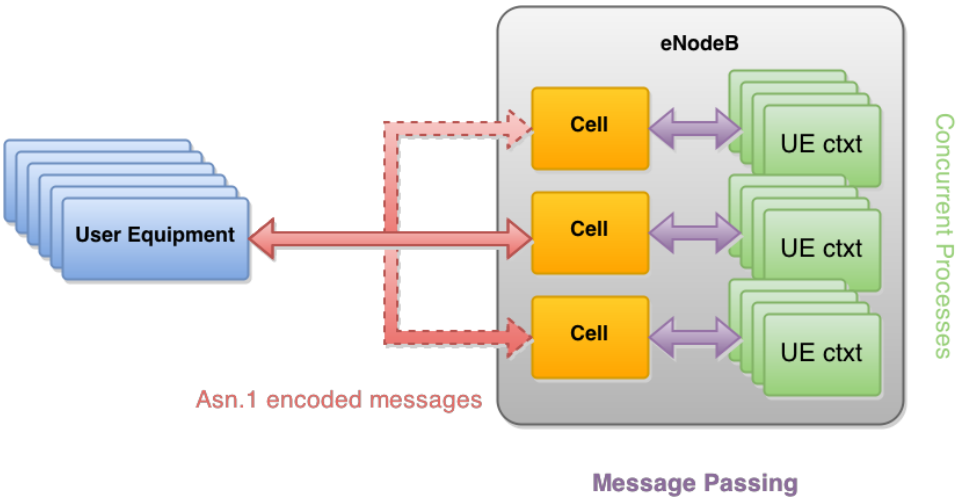

Figure 3:1 – Process diagram of the RRC application

3.2 The RRC Model

To be able to see the impact of the results from the micro benchmarks, a simple model of the RRC application was developed in each programming language. The key aspect of the model is to capture the concurrency properties of the real application, see Figure 3:1.

To be able to see the concurrency properties with a minimal set of functions implemented in the model, only the functions needed to attach and detach a UE were chosen for development. By simulating an attach-detach sequence of an arbitrary numbers of mobiles, the differences in the languages can be visualized. By changing the number of UEs as well as with the number of cores that the application is allowed to run on, an estimation of how well the performance of the application scales with the different parameters can be done.

In order to create a stripped version of an application without changing the behavior of the system too much, a synthetic workload were used. The synthetic workloads were inserted in strategic positions within both the attach and detach functions to prevent the simulated UEs to complete their sequence alone, without interference from the additional UEs.

By inserting this workload in the model and still be able to produce reliable statistics, the need for a benchmark of the synthetic workload was crucial.

3.3 Test automation

The developed benchmarks depend on several parameters and in order to be able to run a high number of tests without manual input of parameters, test automation was needed.

The automation would need to be able to handle multiple benchmarks with varying number of parameters, and for each and every parameter know how to vary that parameter and between which values. With the ability to run a great number of tests without the need of human supervision, the effort it takes to produce reliable output is decreased.

4 Implementation

During the thesis two different “modules” were implemented, the micro benchmarks and the model of the application. How these were implemented is described in this chapter.

4.1 Micro Benchmarks

In order to get hold of the requested KPIs, a series of applications were developed in both Erlang and Golang which utilize their concurrent primitives respectively. During each of these tests, data about memory and elapsed time is collected.

4.1.1 Signaling cost

The benchmark consists of sending M signals between N lightweight processes using the languages message-passing primitives, in order to estimate the time of a lightweight context switch. The elapsed time between the time of sending the first signal and receiving the last signal is measured as well as the maximum memory usage.

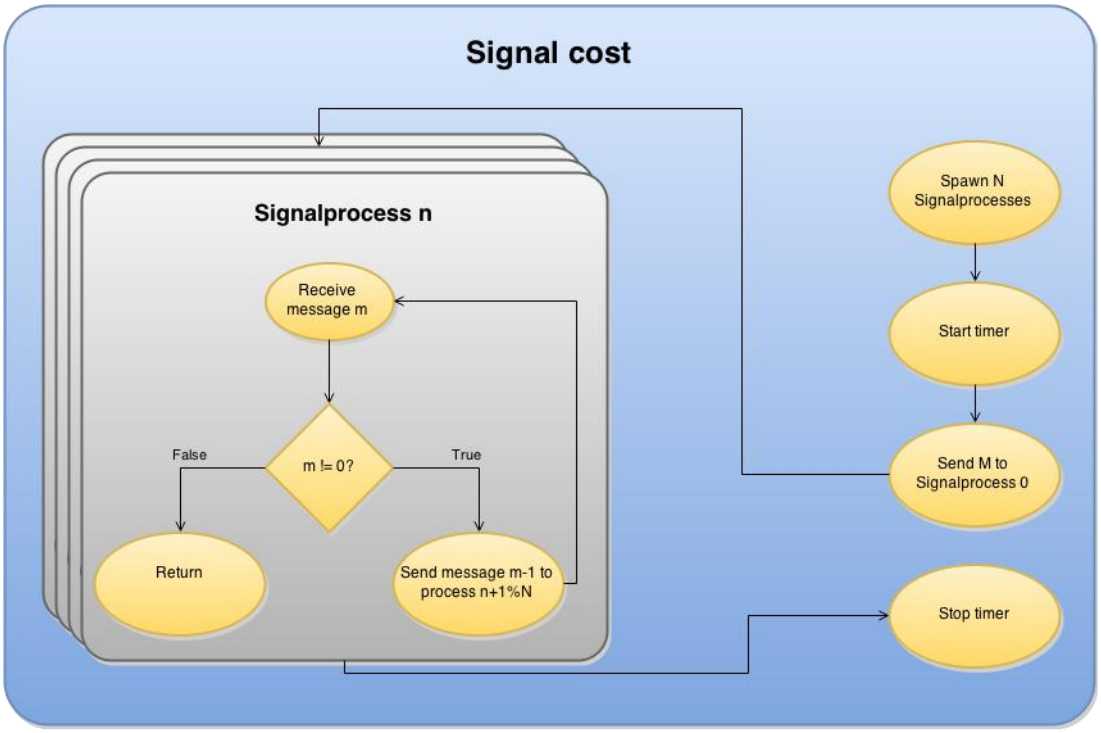

Figure 4:1 – Signal cost

By doing this micro benchmark the signaling time from one process to another is approximated as well as the overhead memory consumption when spawning lightweight processes.

4.1.2 Message passing cost

To determine the cost of sending larger messages between processes a development of the previous program, Signal cost, was done with the functionality to send messages of different sizes between processes. With an additional argument to the program, S, messages of size $32 * 2^S$ bytes is passed between the processes. Other than this change the program works like Signal cost.

4.1.3 Process creation cost

This benchmark was developed to approximate the time it takes to spawn a lightweight process in respective language. In order to do so, N processes is spawned recursively and when the last process has terminated the elapsed time is measured.

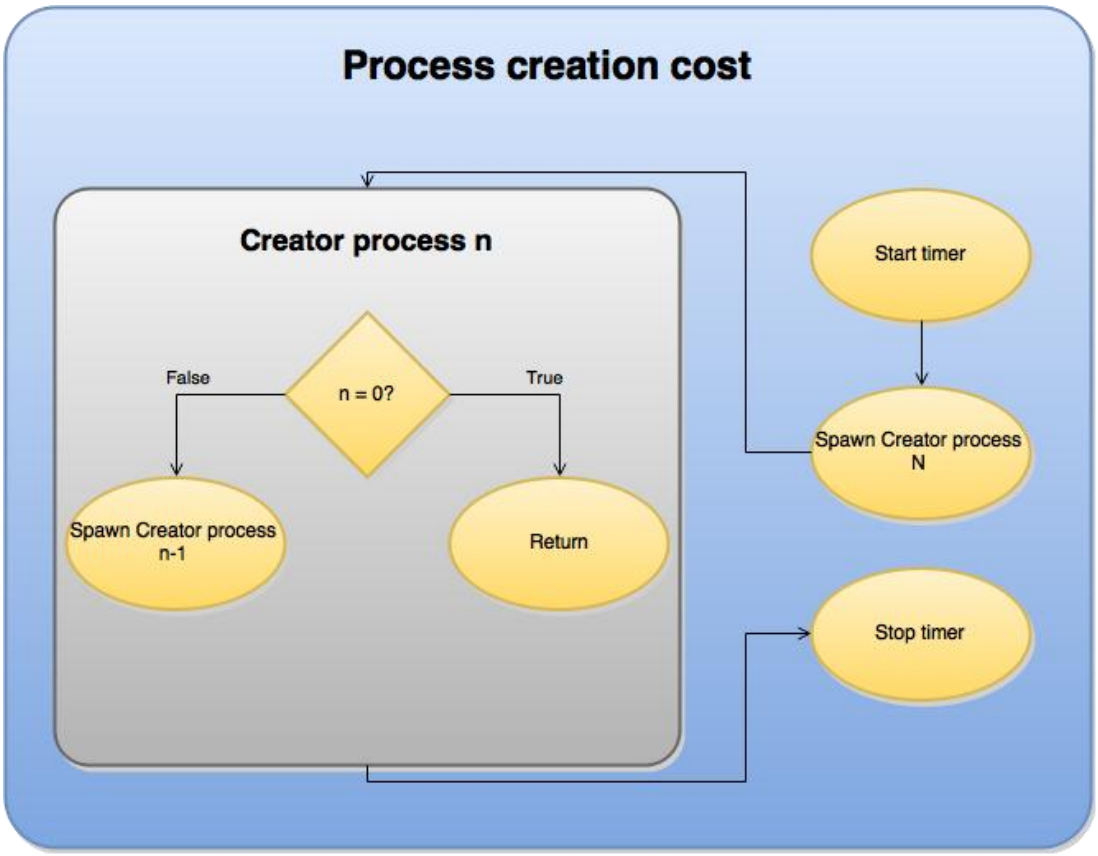

Figure 4:2 – Process creation cost

Keep in mind that the result from this test also contains the context switch time between the spawned lightweight processes. A more straight forward approach to do this benchmark would be to simply measure the time to spawn N processes, but due to differences in the scheduling algorithm between the two languages those measurements is not comparable.

4.1.4 Encoding ASN.1 messages

In order to measure the languages ability of encoding ASN.1 messages, this benchmark was developed to time the encoding of a message with a **encoded** size of S, N times. The message is an example message used in the RRC application with arbitrary length. In Erlang, which comes with an ASN.1 compiler, the native functions are used for encoding messages. In Golang however, the native implementation for ASN.1 handling is not sufficient (e.g. doesn't support the ASN.1 Packed-Encoding-Rules which is used in the application or the generation of data types based on ASN.1 definitions) and a

3rd party ASN.1 to C compiler was used (asn1c). The generated C code is linked into the Golang code which utilizes the functions for encoding ASN.1 messages.

4.1.5 Decoding ASN.1 messages

This benchmark is implemented like the previous benchmark, but instead of encoding messages this benchmark uses the generated C functions to decode ASN.1 messages. The benchmark consists of decoding an encoded sample message of size S, N times and returns the elapsed time for the operation.

4.1.6 Synthetic workload

The benchmark of the synthetic workload differs from the other benchmarks in the sense that it doesn't measure a specific property of interest of the language. The reason behind the benchmark is solely to see how the two languages handle the workload, which is critical when it comes to understanding the output from the benchmarking of the implemented model.

The micro benchmark works like the implemented workload. It consists of two processes that work by the same pattern. It does a series of calculations, signals the other process then waits to be signaled, see Figure 4:3. Only one process does calculations at a time.

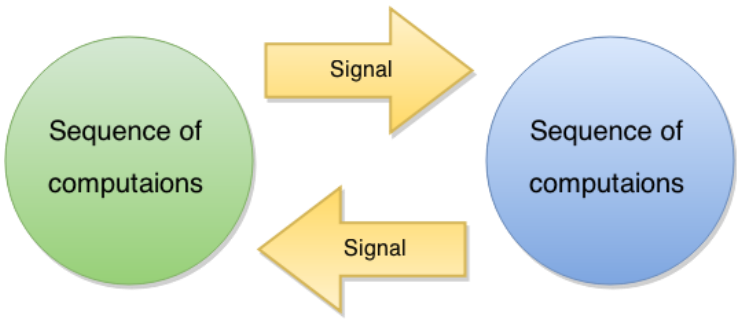

Figure 4:3 – Workload

4.2 RRC Model

This section explains how the model of the RRC application is implemented. The model consists of two different applications, the UE-simulator and eNodeB. Both of these applications run independent from each other and can even run on different machines with a network connection between them. To communicate with each other, they use the Remote Procedure Call (RPC).

Since this test case only focus on the performance of the eNodeB functionalities, the UE –simulator only contains the needed functions to get the eNodeB to work.

4.2.1 UE-simulator

The role of the UE-simulator is to simulate a number of UEs, which all runs the same attach-detach sequence. The simulator runs each UE as a lightweight process which then competes with the other UEs to finish the sequence. The sequence consists of a number of ASN.1 encoded messages passed between the UE and the eNodeB, where each message received progress the UE through the sequence. A description of how the UEs act during the sequences can be seen in Figure 4:4.

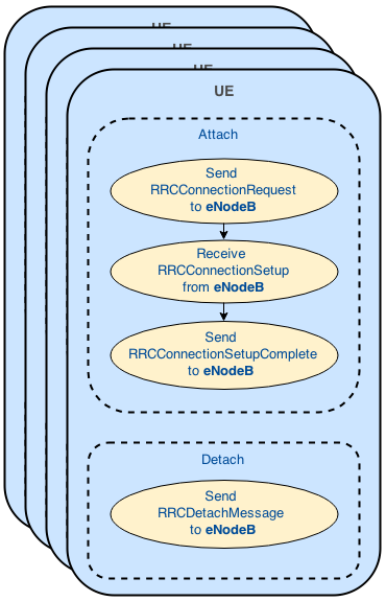

Figure 4:4 – UE

4.2.2 eNodeB

The eNodeB receives the messages sent from the UEs and handles them according to a predefined pattern. The eNodeB consist of two parts; the common part (also known as cell) and the individual part (also known as UE context). The messages from the UEs are received in the cell and from there redirected to the proper UE context if there is any, otherwise, a context is created and the message redirected to it. This means that each UE connected to the eNodeB has its own UE context that is running in its own process.

For a UE to be able to make an RPC call, the eNodeB also has to be an RPC server, with a handler that is being called each time a call is made.

The UE context is designed as a state machine, that after each sent and received message puts itself into another state. For each message that's received by the UE context, a synthetic workload is applied which is supposed to simulate the workload of a real eNodeB implementation.

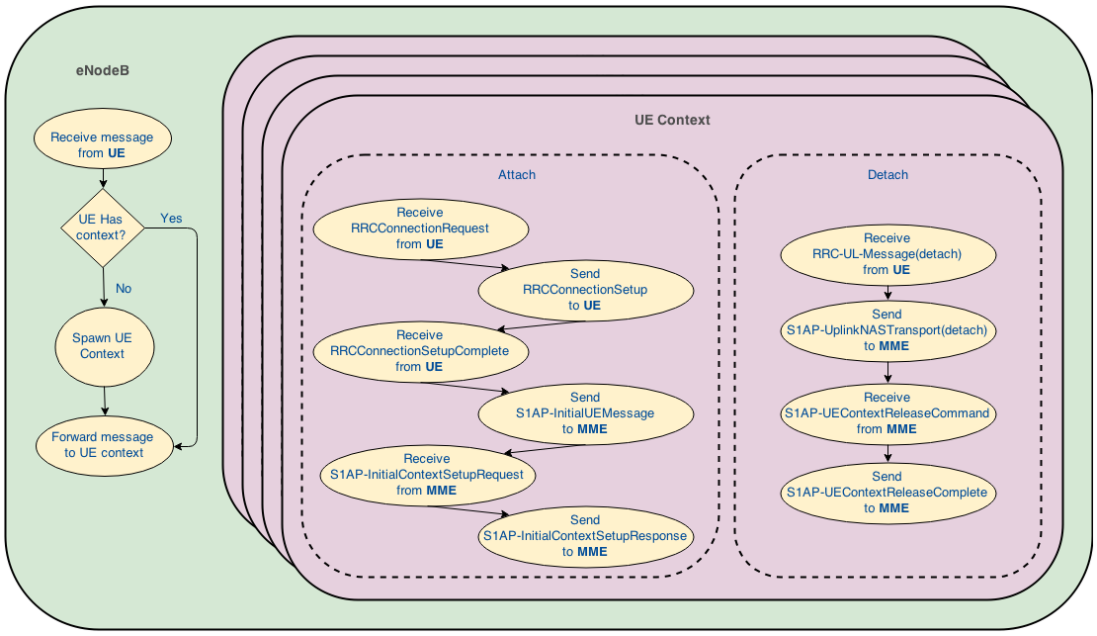

Figure 4:5 – eNodeB

4.3 Test automation

The test automation tool was developed as a bash-script. The script reads input from a file where the test cases are defined. The syntax for a test case is the following:

<name>; <erlang name>; <golang name>; <parameter set #1>, <parameter set #2>, ..., <parameter set #N>

- Name is used as a test case identifier to map the output from the test run to.
- Erlang name is the name of the function to run in Erlang to execute appropriate benchmark.
- Golang name is the name of the compiled binary of the implementation of the benchmark in Golang.
- Each parameter set is a set of 4 values; <start value>, <end value>, <operator>, <modifier>.
- For each parameter set the script will modify the arguments, one at a time, between the runs.

E.g. a test case described with the 2 parameter sets *"10,100,*,10,1,5,+,2"* would result in 6 runs with the following parameters:

"10 1" "10 3" "10 5" "100 1" "100 3" "100 5"

5 Results

In this chapter, the results of the tests that have been done are presented. The results from the micro benchmarks are presented in the first section and the results from model with all the parts put together is presented in the second section.

All the tests were performed on a computer running Linux Ubuntu 14.04.02 LTS as operating system which has two Intel Xeon X5690 processors with 6 cores and 12 threads each, running at 3,47 GHz. The RAM consists of 12 sticks with 4 GB each, making it a total of 48 GB.

5.1 Micro Benchmark

This section displays the data that was collected when performing micro benchmarks on the two different languages. The data is presented as graphs and explained in words. There was a large amount of data collected from the tests but only a small part of it is presented as the rest is considered as redundant.

5.1.1 Memory usage

In Figure 5:1, the memory usage for an entire program in both Golang and Erlang is displayed. This program consists only of the virtual machine for each language and a certain amount of spawned and alive processes, which can be seen in the x-axis. This means that the memory usage on the y-axis is the memory corresponding to the virtual machine and the spawned processes. The virtual machine will not use more memory with a growing amount of processes, and all the processes are equally big in terms of memory usage. This makes a linear relationship and can be described as $y = kx + m$ where y is the total memory usage, m is the memory used by VM and k is the memory used by a single process.

As seen in the graph, the difference between Golang and Erlang is constant and stays the same. This means that the k-values for both languages are the same, and in other words, the size of a process is the same for both languages. This conforms to the definitions of the both languages [8] [13]. The difference between the two languages lies in the m-value, or the overhead for the VM. Erlang has a notably bigger overhead than Golang which is a result of the bigger VM that Erlang has.

Below are the formulas for each language populated with real values (in megabytes) from the test. The memory used by a single process is 0,0026 MB (2,7 KB) for both languages and memory used for the VM is 1,2 MB for Golang and 16,9 MB for Erlang.

$$y_{Golang} = 0{,}0026x + 1{,}2$$

$$y_{Erlang} = 0{,}0026x + 16{,}9$$

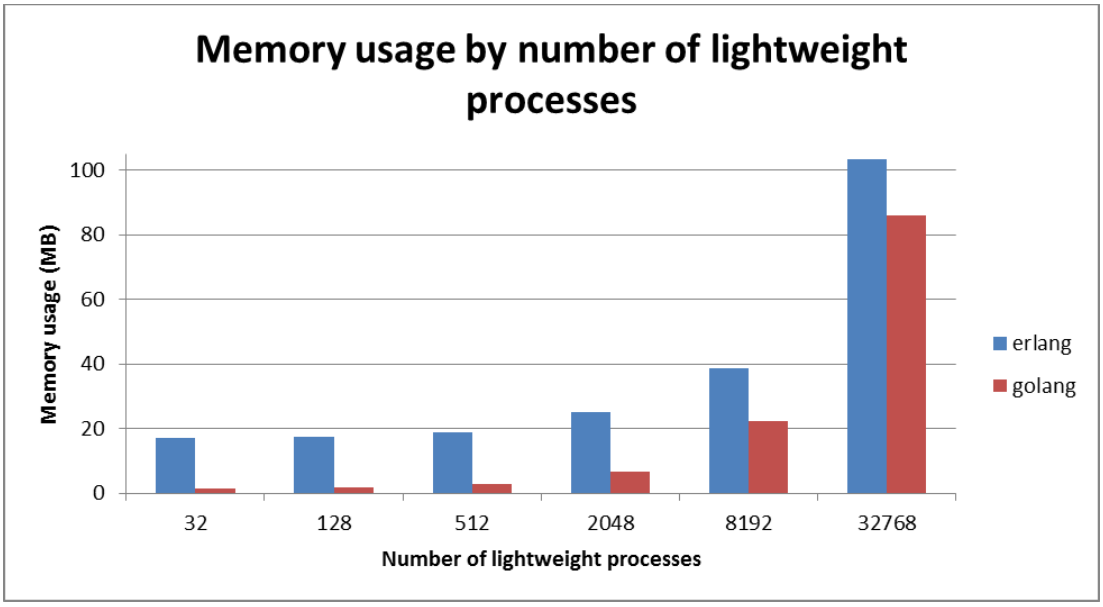

Figure 5:1 – Memory usage by number of lightweight processes.

5.1.2 Creating a lightweight process

The time needed for creating a lightweight process for each language is displayed in Figure 5:2. The graph shows how many nanoseconds it takes to spawn a process. The time is a mean value of 1 000 000 created processes, which is done 100 times and the final value is a mean value of these 100 runs.

The graph shows that creating a process in Erlang is more expensive when it comes to time consumed than creating a process in Golang. The time needed when there are several threads mapped to the OS instead of one is increased for both languages. Erlang is just adding 1/8 of the time compared to Golang adding more than twice the time making the ratio go from 1:5,7 at one thread

to 1:2 at two threads. This means that the Erlang VM is better at handling multiple threads compared to one, but the Golang VM is so much faster at handling one core that this advantage is of less importance

The test were made with up to 16 threads mapped to the OS, but the nature of this test with only two processes running at the same time made it irrelevant to show data for more threads than 2 as it had the same outcome.

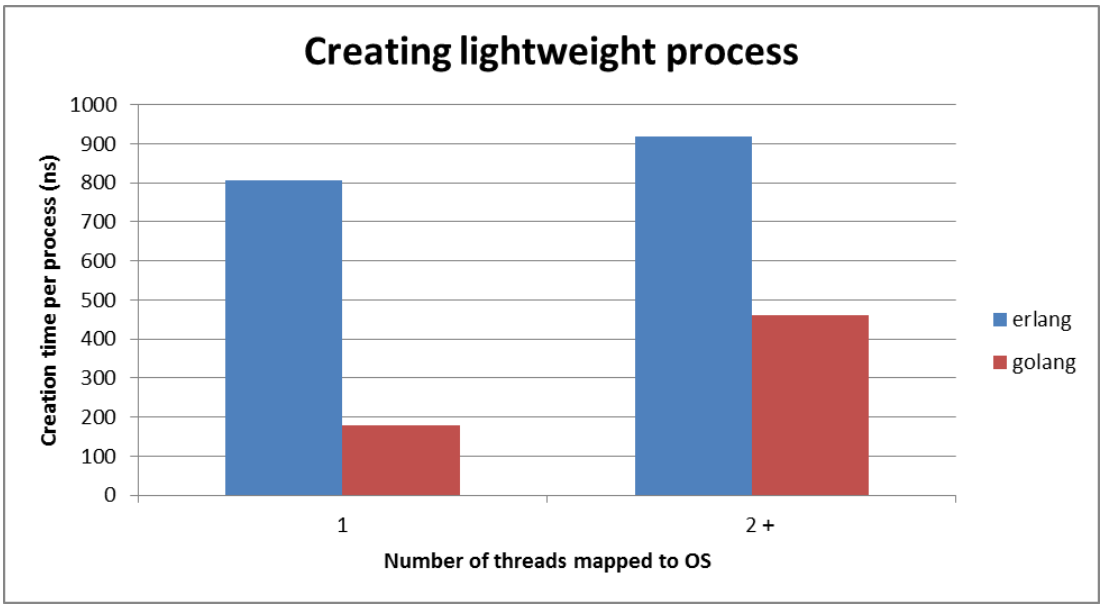

Figure 5:2 – Creating a lightweight process.

5.1.3 Encode ASN.1

Figure 5:3 shows the time consumed for encoding different message structures with different sizes with the ASN.1 notation. The size is linearly increased and the values on the x-axis are the final size of the encoded message. The y-axis show how long time it takes in microseconds to encode one single message, and the value is a mean of 100 runs of a mean of 10 000 messages being encoded.

The graph shows that both Erlang and Golang have a linear increase in execution-time, with Golang slightly faster than Erlang.

Worth noting is that the implementation in Golang is made using the open source library asn1c which is written in C and is then called from Golang, so this result does not show the performance of Golang, but of the asn1c-library.

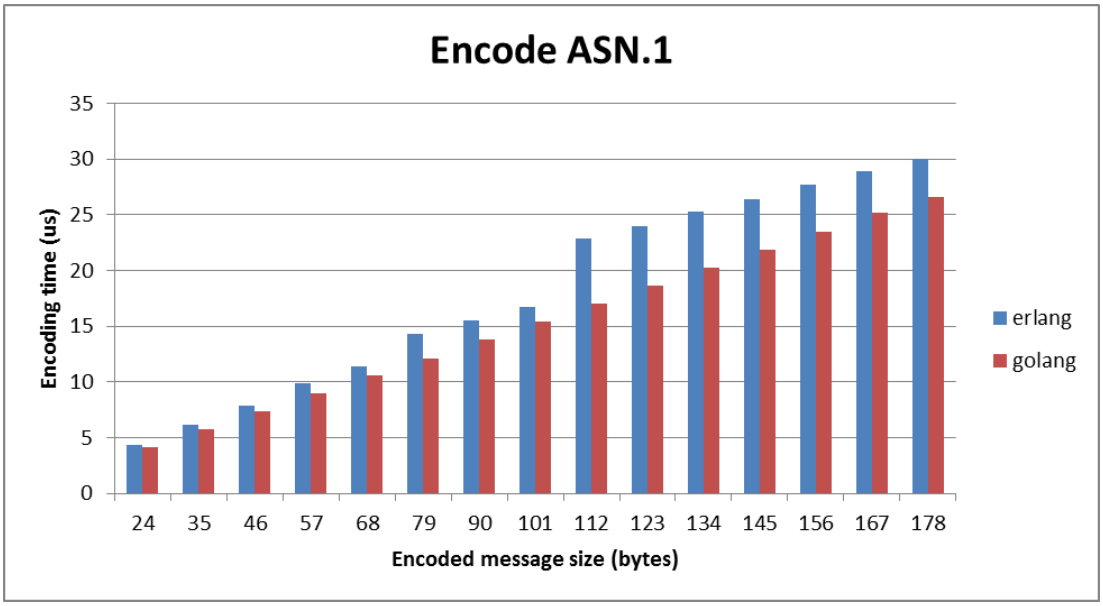

Figure 5:3 – Encode ASN.1 messages

5.1.4 Decode ASN.1

Figure 5:4 shows the time consumed for decoding messages that are in the ASN.1 format, in the same way as described in encoding the message. The messages that are being decoded are the same messages that just previously got encoded.

As a comparison to the times for encoding, where Golang was slightly faster, Erlang is now more than twice as fast compared to Golang. It's not the fact that Erlang has a faster decode than encode, but it's Golang that has a much slower decode than encode.

Again, this is not a result that represents the performance of Golang as a language, because of the implementation using the external asn1c C library.

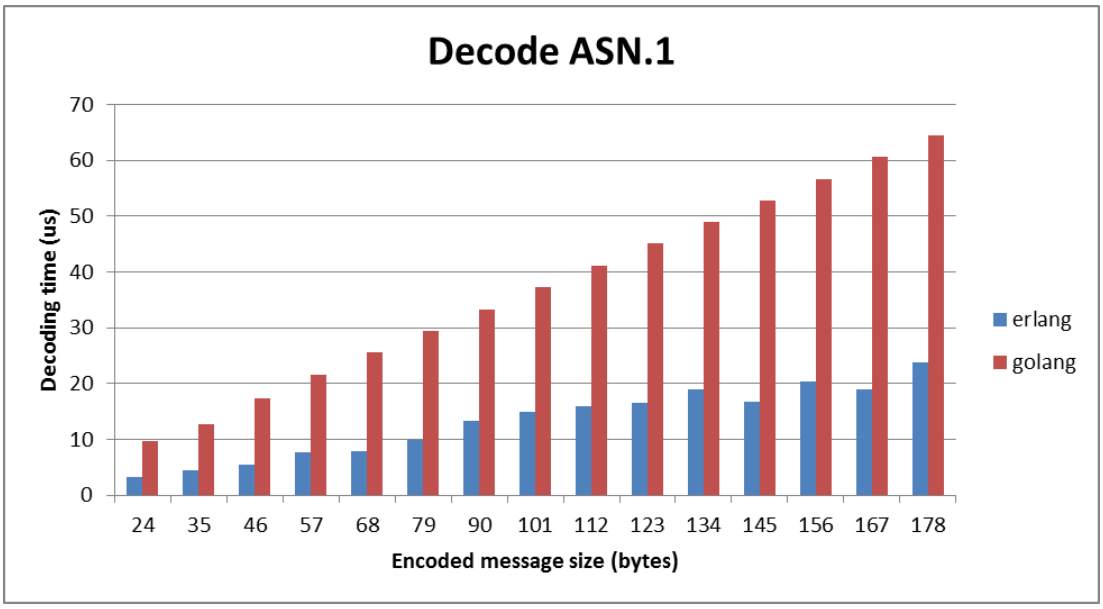

Figure 5:4 – Decode ASN.1 messages

5.1.5 Message passing

In Figure 5:5, the time needed for passing messages of varying size between two processes is shown. The result has been narrowed down to only show message passing when only having two processes active at the same time, the sender and receiver. Having more processes active makes the test measure the wrong things with overhead that only appears when having a larger amount of simultaneous processes active.

As the graph shows, the time for passing messages in Erlang is linearly increasing with the size of the message, while in Golang the time is constant no matter how big the message is. This has its explanation in the implementation in each language, as described in the background of this thesis. When passing messages in Erlang, the whole message is copied and sent so both the sender and receiver has its own copy of the message, while in Golang only the header of the message is sent a pointer to the message in the shared memory so the actual message itself is not copied and sent.

Because of the behavior described above, the comparison is not as fair as it could be, but this is the way they have designed message passing in the two languages.

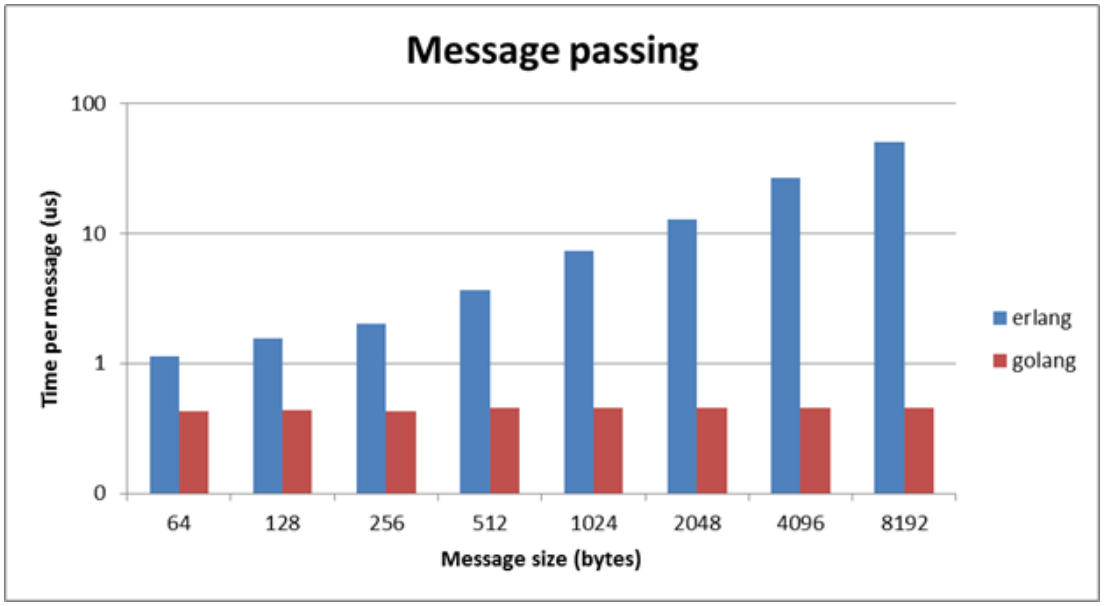

Figure 5:5 – Message passing

5.2 RRC Model

In this section, the data collected from the developed model in Golang as well as the already existing model in Erlang will be displayed. The results from the model do not have the same purpose as the results from the micro benchmarks. These results show how well the scheduler performs, instead of the other language specific primitives. This is because of the workload that has been applied, which stands for almost all the computations, making the other functionalities irrelevant.

5.2.1 Ratio of elapsed time

Figure 5:6 shows a table of the ratio between Golang and Erlang in the elapsed time per UE. The leftmost column specifies the amount of CPU cores used for each amount of concurrent UEs. Ratio is calculated as the average time for Erlang divided by the average time for Golang and shows how many times slower Erlang is than Golang.

In this table, latency is specified as the time a specific UE context is alive, that is from when the first attach message has been received until the last detach message has been sent. This is not the exact definition of latency, but from a UE perspective this will be the latency for attach and detach combined.

Average time is specified as the times for all the concurrent UEs to complete its whole attach and detach sequences divided by the number of UEs.

Number of UEs		
Cores	Ratio Latency (Erl/Go)	Ratio Avg. Time (Erl/Go)
1k		
1	7,32	6,12
2	8,11	5,98
4	10,00	5,79
8	10,49	6,35
5k		
1	7,35	5,93
2	7,73	5,96
4	8,73	6,24
8	10,86	5,65
10k		
1	8,40	5,90
2	8,08	5,86
4	10,23	5,71
8	10,83	5,81

Figure 5:6 – Ratio of elapsed time

As visualized in the table, the ratio for the average time is around six for all the configurations. The explanation for this can be found in the result from the workload micro benchmark, that showed a ratio of **approximately 6,2.** The latency ratio, on the other hand, is increasing with the number of CPU core. This is a result of the differences in the two languages' runtime scheduler, where Erlang has a preemptive scheduler while Golang has not. This means that all Erlang processes get an equal share of the available processors, while the Golang processes are let to run until they're done.

5.2.2 Distribution of latency

In this graph (Figure 5:7) the differences in the schedulers get even clearer. What's interesting in this graph is maybe not the gap between Golang and Erlang, again because of the workload time difference, but in the distribution curve. The curve for Golang is somewhat even, while the curve for Erlang has exponential characteristics with an abrupt end. This means that there are many Erlang processes that finish in the same time, while the Golang processes finish more sporadically. This is the exact characteristics of the schedulers, with Erlang being fair and makes sure all processes gets the same amount of execution time and Golang lets a process run, making the other wait for an indefinite time.

The x-axis is left without grading as it's only the characteristics that are of interest.

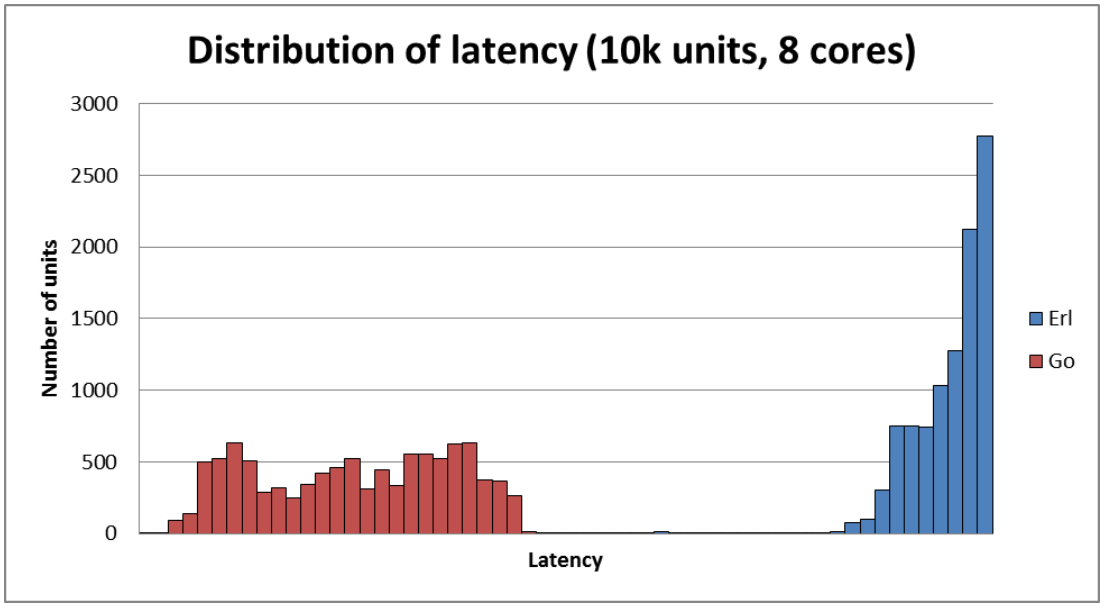

Figure 5:7 – Distribution of latency

6 Conclusion

During this thesis, many tests have been performed and a lot of data has been collected. We therefore think that we have enough information in order to draw a few conclusions about how well Golang has performed in this application in comparison to Erlang.

From the results of the implementation of the RRC model that was made, there was one overwhelming difference that could be observed; the scheduler. The Erlang scheduler is fair and ensures that all processes gets its equal share of the processing power, while the Golang scheduler is less fair but more capable. In this particular field of application, having low latency with the price of losing fairness is more valuable than having it the other way. We therefore think that the scheduler that Golang possess is suitable for this kind of applications.

Following the track with low latency, it showed that spawning a process in Golang is fast compared to Erlang. Taking only half of the time when spawning a process is important when trying to hold the latency low in an environment where processes are frequently spawned.

A part where the two languages differed a great deal was the implementation of the message passing primitive. Golang has an implementation that's built upon channels that any process can read and write to, making it a shared memory implementation where all the processes have the same copy of the message. Erlang, on the other hand, has implemented it in the way that each process has its own connection that's not built upon shared memory. This distinction makes Golang faster as it does not need to send whole messages, but only a pointer to it. However, this feature comes with a downside; robustness. The reason why Erlang sends whole messages as copies is just because of robustness. It ensures that no process is dependent of another and in case of a crash, no other process will be affected, and that can't be achieved with shared memory.

The ASN.1 notation for serialization of messages is widely used within this field of operation, making it important to have good support for it in the programming language that is used. Golang has, as mentioned before, a native support for ASN.1, but with some limitations. These limitations turned

out to be more extensive than first thought. The support was so limited that it was not possible to use the native functionality in this application so a third party C-library was used instead. It was very time consuming to get the external library to work fluently with Golang code and it's not reproducible if another library would be used instead. It's therefore of great importance that the native support is extended in order to have a decent functionality of ASN.1 in Golang.

Our final conclusion is that Golang has good potential to become a very suitable programming language for the given field of application. The concurrency model is good and the performance is high. With that being said, we still think that the language is not mature enough yet to be considered as an alternative. There are still a few things that need to be further developed in order to become fully suitable, especially native support for the ASN.1 standard.

During the work, we have had the opportunity to program a decent amount of code in Golang and by that have some user experience. But since this thesis does not cover the qualitative perspectives of Golang, there is no conclusion on how well Golang performed from that point of view. Future work would therefore preferably include an evaluation of Golang with the qualitative properties as main focus.

7 References

[1] O. Findahl, ”Den mobila boomen fortsätter,” .SE (Stiftelsen för internetinfrastruktur), [Online]. Available: http://www.soi2014.se/den-mobila-boomen-fortsatter. [Använd 28 05 2015].

[2] Wikipedia, ”Wikipedia,” 29 May 2015. [Online]. Available: http://en.wikipedia.org/wiki/LTE_(telecommunication). [Använd 29 May 2015].

[3] Tutorialspoint, ”Tutorialspoint,” [Online]. Available: http://www.tutorialspoint.com/lte/lte_quick_guide.htm. [Använd 29 May 2015].

[4] LteWorld, ”Radio Resource Control (RRC),” [Online]. Available: http://lteworld.org/specification/radio-resource-control-rrc. [Använd 01 June 2015].

[5] K. Lemons, ”Go: A New Language for a New Year,” 06 01 2012. [Online]. Available: http://kylelemons.net/blog/2012/01/06-go-new-language-new-year.article#TOC_1.1.. [Använd 01 06 2015].

[6] Golang, ”Frequently Asked Questions (FAQ),” Google, [Online]. Available: https://golang.org/doc/faq. [Använd 28 05 2015].

[7] Golang, ”Release History,” Google, [Online]. Available: https://golang.org/doc/devel/release.html. [Använd 28 05 2015].

[8] Golang, ”Go 1.4 Release Notes,” Google, [Online]. Available: http://golang.org/doc/go1.4#runtime. [Använd 28 05 2015].

[9] D. Vyukov, ”Scalable Go Scheduler Design Doc,” 02 05 2012. [Online]. Available: https://docs.google.com/document/d/1TTj4T2JO42uD5ID9e89oa0sLKhJYD0Y_kqxDv3I3XMw. [Använd 12 05 2015].

[10] ”The Go scheduler,” Morsing's Blog, 30 06 2013. [Online]. Available: https://morsmachine.dk/go-scheduler. [Använd 12 05 2015].

[11] Erlang, ”History of Erlang,” [Online]. Available: http://www.erlang.org/course/history.html. [Använd 28 05 2015].

[12] Erlang, ”About,” [Online]. Available: http://www.erlang.org/about.html. [Använd 28 05 2015].

[13] Erlang, ”Processes,” [Online]. Available: http://erlang.org/doc/efficiency_guide/processes.html. [Använd 28 05 2015].

[14] J. Flanigan, ”Techtarget,” November 2010. [Online]. Available: http://whatis.techtarget.com/definition/ASN1-Abstract-Syntax-Notation-One. [Använd 02 June 2015].

8 Appendix

8.1 Acronyms

ASN.1 – Abstract Syntax Notation One

EPC – Evolved Packet Core

KPI – Key performance indicator

LTE – Long Term Evolution (also known as 4G)

MME – Mobility Management Entity

OS – Operating System

RAN – Radio Access Network

RPC – Remote Procedure Call

RRC – Radio Resource Control

VM – Virtual Machine

TRITA TRITA-ICT-EX-2015:119