View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Helsingin yliopiston digitaalinen arkisto

Modern web-programming language concurrency

Veli-Pekka Kestilä

Helsinki May 5, 2020

M. Sc. Thesis

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author

Veli-Pekka Kestilä

Työn nimi — Arbetets titel — Title

Modern web-programming language concurrency

Oppiaine — Läroämne — Subject

Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
M. Sc. Thesis	May 5, 2020	66 pages

Tiivistelmä — Referat — Abstract

This Masters Thesis compares Elixir, Go and JavaScript (Node.js) as programming language candidates for writing concurrent RESTful webservice backends. First we describe each of the languages. Next we compare the functional concurrency characteristics of the languages to each other. Finally we do scalability testing for each of the languages. Scalability testing is done using the Locust.io framework. For testing purposes we introduce for simple REST-api implementations for each of the languages. Result from the tests was that JavaScript performed the worst of the languages and Go was the most verbose language to program with.

ACM Computing Classification System (CCS):

Computing methodologies → Concurrent computing methodologies → Concurrent programming languages

Software and its engineering → Software notations and tools → General programming languages → Language features → Concurrent programming structures

Software and its engineering → Software notations and tools → General programming languages → Language features → Concurrent programming structures → Coroutines

Computer systems organization → Architectures → Distributed architectures → Cloud computing

Avainsanat — Nyckelord — Keywords

distributed systems

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction **1**

1.1 Background 1

1.2 Overview 3

2 Concurrency **3**

3 Languages **5**

3.1 JavaScript 5

3.1.1 Frameworks 15

3.1.2 Threading 15

3.2 Elixir 16

3.2.1 Language features 16

3.2.2 Erlang and OTP 23

3.3 Go 25

3.3.1 Simple Go program 26

3.3.2 Language features 27

3.3.3 Concurrency and channels 31

3.4 Conclusion 35

4 Comparison methodology **36**

4.1 Feature comparison 36

4.2 Empirical performance testing 38

5 Comparison **39**

5.1 Go Implementation 40

5.2 Node.js Implementation 45

5.3 Elixir Implementation 50

5.4 Feature comparison 54

5.5 Empirical comparison 56

5.6 Findings 60

6 Conclusion **61**

References **63**

1 Introduction

In this thesis I will describe and compare three web-service programming languages. Namely Go, Elixir and JavaScript (in Node.js run-time). My plan was to find out how each of the languages and their run-times deal with concurrency. Concurrency is important for web-services as it enables serving multiple clients from a single server.

In the tests I found that all of the languages perform reasonably well and most of the tests results were similar. Go and Elixir both were best in some of the tests and JavaScript was a bit slower than the other two with failure to perform in some of the CPU intensive tests. From this we can conclude that the lightweight threading approach used in Go and Elixir performs better than Event based concurrency used in JavaScript.

1.1 Background

In the last 10 years there has been emergence of new languages and frameworks for implementing web-services. Of course there are still a lot of projects done using PHP or Java. But for the new projects programmers are looking to more modern technologies as the basis for their project. One reason programmers are looking for these new languages is stricter memory requirements when running multiple Docker images. These stricter requirements are especially true in cloud services where price is often dictated on physical resources consumed.

One aspect of the new languages is how they handle concurrency. Concurrency and how it is handled is important when trying to serve as many clients as possible with minimal resources. Languages selected for closer examination have differing strategies on how concurrency is handled.

There are different ways to implement concurrency for Internet service[1]. In the lowest level there is threading, which uses either system threads or some kind of light weight threading scheme. Next level is using operating system processes. When a single computer does not have enough capacity, then the concurrency can be implemented using multiple computers. In practice modern Internet services use a combination of these strategies to implement concurrency. There are usually multiple computers and/or virtual machines serving the clients. These computers can have multiple processes running, and finally a single process can use multiple

threads internally.

What are then the benefits of using the different concurrency implementations? For multiple servers benefits come from added redundancy and the possibility to bring the servers closer to the clients. Both of these benefits can be expanded further by using the servers located in different data-centers around the world. Of course this decentralization of servers comes with the cost of more complicated implementation and problems, such as consistency and timely access of data.

Inside the servers we usually have multiple processors, and one way to gain benefit from those processors is to run multiple copies of the server program. This is easily done with using the operating system resources. A problem with process level concurrency is that sharing of data between processes is not necessarily very efficient. Another issue is that switching between the processes is an expensive operating system level operation. Also a simple process can only serve one client at the time.

To make it convenient to serve multiple clients we can utilize threads. All modern operating systems provide threading implementation. Threads share the same memory space, therefore communication between threads is easy. Scheduling of the threads happens at the operating system level so switching from one thread to another takes similar time as switching from one process to another.

To solve the problem with expensive context switches, programs can use lightweight threading. This lightweight threading does not suffer context switching penalty. Another way to avoid the context switches is to use event based concurrency.

Lightweight threads and event based concurrency both have the same weakness where badly behaving code can lead thread starvation and one event prevents others from running. These problems can be mitigated. In case of the thread this is done by forcing the switching of the running thread. And with even processing it is done by interrupting event processing code during its execution to divide it in smaller chunks and interleaving execution of multiple events.

Then the next question with all of these concurrency options is, who needs this increased concurrency? The simple answer is; all internet services. To a certain degree this is probably true. But one segment which will benefit this increased concurrency is large scale internet services like Google or Amazon. According to the definition in [2] a large scale internet service is one with millions hits per day and hundreds of servers.

Modern web service architectures usually employ single page JavaScript applications

in the front-end, connected with REST-API to stateless back-end. These back-ends usually need to handle multiple concurrent clients in a timely manner with ability to scale up and down depending on the load.

Functional and imperative programming style, as well as variable mutability are also important things to consider when doing concurrent programming. Functional style and immutable variables force programmers to write code which avoids certain kinds of hard to debug errors.

1.2 Overview

In this chapter I have shown background about how concurrency affects the internet services and have given background on reading the following chapters.

Next chapter is an overview of what the concurrency means in Computer Science. I will especially talk about the types of concurrency used in the languages in question.

In chapter 3. I will describe the three languages selected for the comparison and their background. I will also talk on the concurrency methods the languages use.

After describing the languages I will introduce the selected comparison methodology in chapter 4. Chapter 5. first describes the language implementations done for the empirical performance testing. After the implementation descriptions it will first have the feature comparison and then empirical comparison results.

Finally in Chapter 6. I will have conclusions and opinions on the test results. I will also talk about how to refine these tests and improve the test results.

2 Concurrency

What does the concurrency mean in Computer Science? Leslie Lamport explores this question in his ACM article[3]. According to him the beginning of research in concurrent systems is Edsger Dijkstra's 1965 paper describing the mutual exclusion problem. Most of the early concurrent system research concentrated on algorithms and correctness. Especially correctness is still an issue on poorly implemented concurrent algorithms and programs. Modern concurrency research is a wide field inside of Computer Science.

So to answer the question in context of this thesis; In Computer Science concurrency means that events, programs, processes and so on seemingly execute at the same

time. What this same time means is then implementation specific. For example in single processor this means that processor executes two or more items in smaller interleaved chunks. This definition of executing smaller interleaved chunks is important as it is the basis on which all of the languages explored reach seemingly concurrent processing of multiple requests.

On operating system level program execution happens in processes which are scheduled on the operating system level. Processes can then be divided in multiple threads[4]. Threads can be either scheduled by operating system or by virtual machine running inside the process. Threads can also be scheduled co-operative manner. In this thesis I will call these threads run inside of the process without operating system lightweight thread. Some implementations or sources call them green threads, user threads, coroutines or fibres.

The main difference between the operating system threads and lightweight threads is the way they are scheduled. Operating system thread scheduling is closer to scheduling processes and needs multiple operating system and CPU level operations. Lightweight threads can have more lightweight scheduling operations which take less time and resources.

One way of implementing lightweight threads is cooperative multitasking. In cooperative multitasking thread or process executes code for a certain number of steps before yielding the execution turn to the next thread or process. Classic well known examples of co-operative multitasking are Windows versions before windows NT and 95.

Event based concurrency has been an important part of computing for a long time as evident in a 1975 paper by Leon Presser [5]. Early concurrency related events were often related to the peripherals connected to the computers. Most common event based task even in most modern operating systems is handling of clock interrupt which is used for scheduling different processes to run. User interfaces often have this kind of event based model where the user presses a button or inputs text which is then processed by the event processor.

In this chapter I described what concurrency means and also introduces the concurrency concepts most prevalent to the following chapters. Each of the languages described in this thesis use either even based co-operative multitasking or lightweight threads based multitasking.

3 Languages

In this section I will look into three programming languages used in modern web-service back-ends. These languages were selected for their different characteristics which will be described in this chapter. One selection criteria was that they are modern or in case of JavaScript modern in web-service use, but still used widely enough. Another criteria was that each of the languages used new paradigms to implement concurrency instead of the traditional way of implementing concurrency with threads and processes.

These languages present different programming paradigms from functional to procedural and from dynamic typing to static typing. Each of the selected languages also has its own strengths and drawbacks. I will discuss these in detail when talking about the specific languages.

For each language I will briefly talk about their history. I will also introduce the language syntax. Then I will give the overview of the run-time system and how the concurrency is implemented in each of the languages. In conclusion I will look into the non concurrency related differences of these languages.

Next chapter after this will introduce the methodology used for comparing the concurrency features of these languages. After the methodology chapter follows the chapter describing the comparison process and its results.

3.1 JavaScript

JavaScript[6][7] is a language aspiring from humble beginnings. The language was designed by Brendan Eich during 10 days in 1995 for Netscape, and has come far from the beginnings. Most important steps for its development to the current state was the implementation of Ecma standardization process, Microsoft's XMLHttpRequest and Google's V8 interpreter. V8 virtual machine gave push for implementation of Node.js framework and upgraded JavaScript as a language which can also be used for server side implementations.

JavaScript syntax is heavily influenced by C. One of the reasons was that the language was created to be a lightweight version of the Java language. As a result JavaScript does not have Java like class and interface syntax. However this does not mean that JavaScript is not an object oriented language. It uses a Prototype-based object model. The language is also imperative, structured and functional.

JavaScript functions also have different characteristics depending on where they are used. This means JavaScript functions can work as object definitions when used as constructor. If a function is defined as an object property it will work as a member method. Finally it can work as a regular function when defined elsewhere in the code.

One of the main reasons of JavaScript's popularity is its use of event based concurrency. This stems from the language's original target to be used in user interfaces where a lot of functionality is related to user triggered events. In addition time consuming operations like network, database or disk access can be easily handled asynchronously using events. This makes the language perform well despite it being single threaded.

Version	Published	Description
1	Jun 1997	Initial standard.
2	Jun 1998	Edited to conform ISO/IEC 16262.
3	Dec 1999	Improved language by adding regular expression, exception handling and other improvements.
4	Abandoned	
5	Dec 2009	JSON support, "strict mode" and other enhancements.
5.1	Jun 2011	Conformation to ISO/IEC 16262:2011
6	Jun 2015	Multiple changes to language including but not limited to: classes, modules, for/of loops, generators, arrow functions, collections and promises.
7	Jun 2016	Language refactoring, exponential operator and Array.prototype.includes
8	Jun 2017	concurrency, atomics and async/await
9	Jun 2018	asychronous iteration and generators, enhancements to regular expressions and rest/spread parameters

Table 1: ECMAScript versions

The functionality of JavaScript language is defined through the ECMAScript standardization process. Currently there are nine versions of standard as seen in Table 1. The fourth version of the standard was abandoned. It is also clear that the pace of change to the language has picked up in recent four years.

```
1 function callingFunction(callback, errorCallback) {
2   firstAsyncTaskWithCallback((result1) => {
```

```
3     secondAsyncTaskWithCallback(result1, (result2) => {
4       thirdAsyncTaskWithCallback(
5             result1, result2, (result3) => {
6         callback(result3);
7       }, (error3) => {
8         console.log('Error', error3);
9         errorCallback(error3);
10      });
11    }, (error2) => {
12        console.log('Error', error2);
13        errorCallback(error2);
14    });
15  }, (error1) => {
16    console.log('Error', error1);
17    errorCallback(error1);
18  });
19 }
```

Listing 1: Nested code using callback functions.

```
1 function longRunningTask(param1, param2) {
2   return new Promise((resolve, reject) => {
3     // Do some asynchronous processing
4     if (result == 'successful') {
5       resolve('Success');
6     } else {
7       reject('Failure');
8     }
9   });
10 }
```

Listing 2: Example of promise definition.

```
1 function callingFunction() {
2   return firstAsyncTaskWithPromise()
3   .then((result1) => {
4     return secondAsyncTaskWithPromise(result1)
5     .then((result2) => {
6       return thirdAsyncTaskWithPromise(result1, result2)
```

```
7        .then((result3) => {
8          return result3;
9        });
10     });
11   })
12   .catch((error) => {
13     console.log('Error', error);
14     throw error;
15   };
16 }
```

Listing 3: Code from Listing 1 implemented with promises

Version six of the standard named ECMAScript 2015 is important for concurrency features in the language. Most important feature was to introduce a standard way to implement promises. The Promises were first introduced by Daniel Friedman and David Wise in 1976 and are well described in Wikipedia[8]. An example of a promise is given in Listing 2 that shows declaration of the promise and how resolve and reject functions are used to signal the result of the promise.

Promises help to deal problems with multiple nested callback functions. These problems are evident in Listing 1 where there are multiple callback functions and errors have to be handled in multiple different places. Reading and reasoning this kind of code becomes quickly cumbersome and can lead to programmer mistakes [9]. In Listing 2 we see the code with callbacks transformed in more manageable form with use of promises. There is only one place to deal with errors and results come as `return` value for the function so we don't need to give callback functions as parameters.

```
1 async function callingFunction() {
2   try {
3     const result1 = await firstAsyncTaskWithPromise();
4     const result2 = await secondAsyncTaskWithPromise(
5         result1
6     );
7     const result3 = await thirdAsyncTaskWithPromise(
8         result1, result2
9     );
10  } catch (error) {
```

```
11       console.log('Error', error);
12       throw error;
13     }
14     return result3;
15   }
```

Listing 4: Code from Listing 3 implemented using async/await.

The version 8 of the standard introduced async/await functionality to the language. With this improvement to the language programmers can declare function to be asynchronous using an async keyword. Inside an async function it is possible to use await keyword to wait for results from the functions returning promises. This async/await functionality is demonstrated in Listing 4 where it is easy to see how the program is cleaner and more concise when compared to the previous examples.

With the previously mentioned fast pace of innovation in the language, it has become usual for the JavaScript community and virtual machine implementations to adopt more popular features of the language standard even before actual standardization. This sometimes can lead to features to be implemented which does not end up in the final standard. An example of this is the `WeakMap.prototype.clear()`[10] method which was planned to ECMAScript version 6, but dropped before the final standard. On the other hand some of the standard mandated features can be implemented years after the standard has been published.

To counter this problem of unimplemented features, it is common to use libraries called polyfill[11] to patch the missing functionality on the JavaScript execution environment. This environment missing the standard defined functionality could be for example an old web-browser like Microsoft's Internet Explorer, which isn't updated anymore, but is widely used.

Another way to deal with the problems of unsupported language features is using a technique called source, to source compilation or transpiling[12]. Transpiling is a process of transforming source code of one programming language to another one. In the context of ECMAScript, transpiling is usually used to convert from newer versions to older and more widely supported versions. Usually the target for conversion has been the well supported version five of the language standard.

The most common transpiler for conversion between ECMAScript versions is called Babel[13]. Babel started in 2014 as a tool called 6to5. It was created by Sebastian McKenzie as a project to better understand programming languages. It was renamed

as Babel in 2015. Babel can be also used for other transformations like transforming JavaScript XML (JSX) to pure JavaScript code. JSX is a way to embed HTML code into JavaScript files and eliminating separate needs for visual template files.

Sometimes it is desirable to transpile from a different language to ECMAScript. This is done so that programmers can implement the program with language they are familiar with and then transpile it to JavaScript for execution in web-browsers, where you can only run some version of ECMAScript. Another reason for transpiling may be to add features not present in ECMAScript like static typing.

Microsoft has implemented a super set of ECMAScript with static typing called TypeScript[14]. It adds the possibility of giving type information to variables and function definitions, and checking that there are not unintentional type conversions in compile time. The weakness of TypeScript is that it does not protect against assignment and use of items (objects or functions) with wrong types during program execution. This problem is present in most of the statically typed languages in run-time. Mixing of plain JavaScript code and TypeScript code makes this problem worse in TypeScript than most of the other statically typed languages.

```
1
2 function ExampleClass(name, value, privateName) {
3     var privateValue = 5;
4     this.memberName = name;
5     this.memberValue = value;
6
7     this.getPrivateName = function() {
8         return privateName;
9     }
10
11     this.getPrivateValue = function() {
12         return privateValue;
13     }
14
15     this.setPrivateValue = function(value) {
16         privateFunction(value)
17     }
18
19     function privateFunction(value) {
```

```
20          privateValue = value;
21      }
22  }
23
24  ExampleClass.prototype.getName = function() {
25      return this.memberName;
26  }
27
28  ExampleClass.prototype.getValue = function() {
29      return this.memberValue;
30  }
31
32  ExmapleClass.prototype.setValue = function(value) {
33      this.memberValue = value;
34  }
35
36  var instance = new ExampleClass(
37      'public_name', 42, 'private_name'
38  );
39  instance.getPrivateName(); // Returns 'private_name'
40  instance.getName(); // Returns 'public_name'
41  instance.getValue(); // Returns 42
42  instance.setValue(84);
43  instance.getValue(); // Returns 84
44  instance.setPrivateValue(10);
45  instance.getPrivateValue(); // Returns 10
46  instance.memberName; // Direct access to instance variable
47  instance.memberValue; // Direct access to instance variable
48
49  instance = null; // Releases instance to be
50                   // garbage collected.
```

Listing 5: ECMAScript 5 class

```
1
2  class ExampleClass {
3
4      constructor(name, value) {
```

```
        this.memberName = name;
        this.memberValue = value;
    }

    get name() {
        return this.memberName;
    }

    get value() {
        return this.memberValue;
    }

    set value(value) {
        this.memberValue = value;
    }
}

let instance = new ExampleClass('public_name', 42);
instance.name(); // Returns 'public_name'
instance.value(); // Returns 42
instance.value(84);
instance.value(); // Returns 84
instance.memberName; // Direct access to instance variable
instance.memberValue; // Direct access to instance variable

instance = null; // Releases instance to be
                 // garbage collected.
```

Listing 6: ECMAScript 6 class

```

class ExampleClass {

    private privateName: string;
    private privateValue: number;
    public memberName: string;
    public memberValue: number;

```

```
9       constructor(
10          name: string, value: number, privateName: string
11      ) {
12          this.privateName = privateName;
13          this.memberName = name;
14          this.memberValue = value;
15      }
16
17      public getName(): string {
18          return this.memberName;
19      }
20
21      public getValue(): number {
22          return this.memberValue;
23      }
24
25      public setValue(value: number) {
26          this.memberValue = value;
27      }
28
29      public getPrivateName(): string {
30          return this.privateName;
31      }
32
33      public getPrivateValue(): number {
34          return this.privateValue;
35      }
36
37      public setPrivateValue(value: number) {
38          this.privateFunction(value);
39      }
40
41      private privateFunction(value: number) {
42          this.privateValue = value;
43      }
44  }
```

```
45
46 let instance = new ExampleClass(
47     'public_name', 42, 'private_name'
48 );
49 instance.getPrivateName(); // Returns 'private_name'
50 instance.getName(); // Returns 'public_name'
51 instance.getValue(); // Returns 42
52 instance.setValue(84);
53 instance.getValue(); // Returns 84
54 instance.setPrivateValue(10);
55 instance.getPrivateValue(); // Returns 10
56 instance.memberName; // Direct access to instance variable
57 instance.memberValue; // Direct access to instance variable
58
59 instance = null; // Releases instance to be
60                  // garbage collected.
```

Listing 7: TypeScript class

As mentioned before JavaScript is an object oriented language. In Listing 5 we can see how ECMAScript 5 class is initialized. This looks quite foreign for people who are used to how classes are defined for example in Java. Class is defined as a constructor function which can then be instantiated. Its parameters are by default private variables inside of the class. One can also define more private variables by using var keyword. It is also possible to define private functions which are only visible inside of the class. Functions defined inside the constructor can access both public and private instance variables. Functions defined using a prototype can only access the public instance variables.

ECMAScript 6 introduced a new way of defining classes. It is shown in Listing 6. The main differences are the use of a more Java like way of defining classes with class keyword and separate constructor. All methods are defined inside the class body. The listing also demonstrates new ways to define the methods by get and set to define getter and setter methods. The big difference to the previous example is that there are no private methods or instance variables. There is a proposal for private instance variables which is not standardized or widely implemented.

Many people are more familiar with the way classes are defined in TypeScript, as shown in Listing 7. In TypeScript there is a way to define variables and methods to

be private or public. Another important aspect shown here is the definition of static types for variables and parameters. Some of these definitions might be omitted as their types can be inferred from the usage and still enforced by the compiler.

3.1.1 Frameworks

Node.js[15][16] is the engine making server side JavaScript to happen. It is implemented on top of Google's V8[17], which is the JavaScript run-time build for Google's Chrome Internet browser. The project was started by Ryan Dahl in 2009. In addition to the simple event loop on top of V8 engine Node.js includes a low level I/O API and package manager in the form of npm (Node Package Manager).

As it happens sometimes with Open Source projects there was disagreement on how Node.js should be developed and for a while there was a competing fork of the runtime called IO.js. This disagreement ended in 2015 when Node.js Foundation was founded and both projects were merged to form Node.js version 4.0.

Technically Node.js runs a single main thread with an event loop to process messages. These messages consist of data and a function reference to process it. This single threaded design eliminates most of the thread switching overhead. Thread switching overhead is caused by maintenance work that needs to be done by the operating system when changing from one thread to another. In addition to this main thread it has helper threads to implement the non blocking I/O calls. This non blocking I/O is implemented using observer pattern[18]. Observer pattern is defined as one to many relation where many observers subscribe to information updates from a single producer.

There are also other projects implementing JavaScript on the server side, but none of them have grown as big as Node.js. This is because of the rapid growth of the Node.js ecosystem, especially in the form of npm package manager and related repositories consisting of over 800 000[19] packages. Because of this other platforms will be excluded from the evaluation.

3.1.2 Threading

Browser implementations of JavaScript virtual machines have WebWorker API [20] which defines how to use background threads to execute long-running CPU intensive code in browsers. Node.js introduced worker threads in release 10.5.0[21], which will allow running longer background tasks outside of the thread for the main event loop.

3.2 Elixir

Elixir[22][23] is a concurrent and functional programming language created by José Valim. Elixir is implemented on top of the Erlang[24] virtual machine BEAM (Bogdan's Erlang Abstract Machine) named after the Bogumil "Bogdan" Hausman who was the virtual machine's original creator. Elixir and Erlang are compiled to bytecode. This bytecode is then executed by the BEAM. Because both languages compile to the same bytecode, it is possible to invoke functions written in either of the languages from both of them. This is very similar to cross function call ability of languages targeting Java Virtual Machine (JVM) or Microsoft's Common Language Runtime (CLR) virtual machine.

The path[25] that led to creation of Elixir started when José Valim wanted to find a good solution for running code concurrently. Because of this need he started to explore different languages and ideas to use multi-core capabilities of the computers. This exploration led him to Erlang-language and BEAM which in his opinion were the answer. He also felt that Erlang was missing important features from other languages. Elixir was created to bring these features to the BEAM platform.

3.2.1 Language features

Next I will explore some of the features of Elixir[23]. In Elixir everything is an expression. Another choice is that the code is divided into the statements and expressions. The expressions are language constructs composed of variables and operations, which when evaluated return a result. An example of expression is a + b, which calculates together variables a and b. Statements on the other hand affect the flow of the program without returning results. While loop is a common example of a statement. While loop executes itself until the given condition is false without returning a result.

Erlang interoperability is an important part of the Elixir. This interoperability means that Erlang functions can be called from Elixir without run-time cost and enables usage of libraries written for Erlang in Elixir. This has had a positive effect on the speed of adaptation of the language, as there are already libraries for accessing outside resources like databases.

```
1 defmodule Unless do
2   defmacro macro_unless(clause, do: expression) do
3     quote do
```

```
4        if(!unquote(clause), do: unquote(expression))
5      end
6    end
7  end
```

Listing 8: Elixir macro definition [26].

The language also includes a way to change and extend itself with the help of macros. These macros can be used to manipulate Abstract Syntax Tree (AST). AST is the internal presentation of Elixir code and it is made of tuples. In the above Listing 8, I define macro called *macro_unless*. One should note the `defmacro`, `quote`, and `unquote` operators. Defmacro starts the macro definition. Quote returns given expressions in AST format. All the variables are evaluated on quote. Because of this there is unquote which can be used to inject variables which have not been yet evaluated. In the above example clause and expression are not defined before the macro is called, and because of that they are unquoted.

Macros are quite a powerful language feature and many of the Elixirs features are implemented as macros. Example of this kind of feature is `unless`-expression which is implemented as a macro using `if`-expression.

Polymorphism was introduced by Strachey in 1967 and reprinted in 2000[27]. Polymorphism is a way for the function to accept multiple inputs. For example in Java this means that different objects implementing interfaces either via superclass or interface definition can be processed with the same method. In Elixir this polymorphism is done with protocols. Protocols work with providing a dynamic dispatch mechanism. This is similar to Clojure[28] which is a language designed for JVM environments. This feature of Elixir should not be confused with the multiple dispatch feature. Multiple dispatch means that the method selections are based on the run-time type or other dynamic attribute. In contrast to multiple dispatch, Elixir protocols dispatch only on a single type.

Because documentation is often an important part of the program, Elixir provides a way to document code using docstrings[29]. Docstrings were introduced in the original TECO implementation of popular text editor Emacs. There are multiple other languages which support docstrings. For example Lisp, Clojure, Gherkin, Julia and Python. Python is probably the most popular one of these languages.

Contrary to many popular programming languages like Java, Elixir doesn't have variables or shared data structures. Function calls in Elixir are implemented by

message passing. In Elixir programs mutable data is stored in processes implementing the actor model. This makes it easy to implement concurrent programs in Elixir, because data-access concurrency issues disappear when using this kind of shared-nothing architecture. Pure shared-nothing architecture also makes scaling up to be as simple as adding one more computing node.

Functional design can be seen in Elixir on use of recursion instead of loops and in use of higher-order functions. It is also usual that functional languages prefer immutable data structures over mutable ones. Functional features are becoming more popular and are adopted by procedural and object oriented languages like Java. For example Java 8 has higher order functions, recursion capabilities and immutable data structures.

```
1  defmodule Loops do
2    def recursive([head|tail]) do
3      IO.puts "Head #{head}"
4      recursive(tail)
5    end
6
7    def recursive([]) do
8    end
9
10   def for_loop(list) do
11     for item <- list do
12       IO.puts "Item: #{item}"
13     end
14   end
15 end
16
17 Loops.recursive([1, 2, 3, 4])
18 Loops.for_loop([1, 2, 3, 4])
```

Listing 9: Elixir iteration examples.

```
1  import java.util.List;
2  import java.util.Arrays;
3
4  class Loops {
5    public static void recursive(List<Integer> list) {
```

```
6       if (!list.isEmpty()) {
7         System.out.println("Head: " + list.get(0));
8         recursive(list.subList(1, list.size()));
9       }
10    }
11
12    public static void loop(List<Integer> list) {
13      for (int item: list) {
14        System.out.println("Item: " + item);
15      }
16    }
17
18    public static void main(String[] args) {
19      List<Integer> list = Arrays.asList(1, 2, 3, 4);
20      recursive(list);
21      loop(list);
22    }
23  }
```

Listing 10: Java iteration examples.

Listing 9 demonstrates two examples on how to loop through a list in Elixir. First the recursive one is calling itself, processing the head of the list and passing the end of the list to the next recursion. When the list is empty, it will match to the function expecting an empty list, and the recursion will end. This is also an example showing how pattern matching in Elixir works when calling functions. The second function shows how to do the same thing using an iterator.

Listing 10 display an example in modern Java for comparison. As there is no pattern matching capability, a recursive method needs to use the if-operator to determine if recursion should end. Also extracting items from the list is more cumbersome without the pattern matching operations. The iterator version of the function is quite similar to the Elixir version.

The common belief is that threads and processes are better handled as operating system features. The problem is that it is relatively expensive to create either threads or processes in programs. Because of this cost, it is preferable to create threads or processes and use them multiple times. Even recycling threads and processes has its cost as it needs operating system intervention to schedule them for execution,

which is an expensive context switch.

The context switch happens when the processor changes the execution from one thread to another. In context switch the execution of the current thread is stopped and the state of it is saved. This state saving includes all of the registers, other necessary data and program counter pointing to the next instruction to be executed when a thread is rescheduled. After this the operating system decides which thread to execute next and that thread's state is loaded in the processor to be executed. After loading data the processor starts to execute the new thread.

Because of this Elixir via Erlang has lightweight processes, which do not suffer aforementioned penalties. This is because the creation and scheduling of processes are implemented on language and virtual machine level. This is helped by the fact that the language does not have mutable variables which might make it imperative to have safeguards against accidental modification on different threads or processes data.

To make it cleaner to handle error cases in the chain of functions, one can use a technique called Railway oriented programming[30]. In this technique we have two tracks(paths) through the program. One of the tracks is the success track and another one is the error track. Any function in the chain of functions can go from the success track to the error track, but not vice versa.

The main idea is that the chained functions after the first one accept success and error inputs and emit both of them according to the rules laid out in the previous paragraph. It is also possible to adapt functions which do not support Railway oriented programming to this paradigm by using higher-order functions.

```
1   defmodule Railway do
2     def process(data) do
3       with {:ok, true} <- verify_data(data),
4            {:ok, value} <- process_data(data)
5       do
6         value
7       else
8         {:error, error} -> error_handler(error)
9      end
10   end
11 end
```

Listing 11: Elixir railway oriented programming.

Railway oriented programming in Elixir benefits from the languages `with-construct`. With-construct makes it easy to test things and go to the error case when necessary. Also it is easy to make functions wrapping other functions. And finally it is common for elixir functions to return `{:ok, data}` and `{:error, error}` constructs as a result. In Listing 11 we can see a function called process which is using with-construct to do railway oriented programming. As long as the functions return patterns starting with `:ok` and matching the second parameter it will return extracted value in the end. In the case of the first error it will go to the else section where the error can be handled.

Usually programming projects consist of multiple files and use third party libraries in addition to standard libraries shipping with the language. With Elixir there are two tools shipped with the language which are helpful. These tools are called Mix and Hex. Mix is the build tool similar to Make, Maven or Gradle. Hex is the package manager similar to popular NPM for JavaScript or Bundler for Ruby. Maven and Gradle also include package management for Java projects. It is also possible to format source code and run tests with mix. Unified formatting for the code and automated testing help working in teams and maintaining code quality.

Like many modern languages Elixir ships with an interactive shell for running source code interactively. In Elixir this tool is called iex. With iex it is possible to test small pieces of code or load your program and call its functions to check that they work as intended. Debugging is also done with the help of iex. User first starts an iex session with the program code loaded, and then starts the graphical Erlang debugger called Debugger. This method can be also used to do remote debugging by connecting iex and the debugger to a program that is already executing in a remote or local machine.

Elixir also contains lazy and asynchronous collections which can be used to minimize the need of computations when transforming data from one form to another. These collections are implemented with stream-module. The operations of stream-module can be chained together and combined so that there is reduced need for intermediate data structures. This way of processing data is especially beneficial when we need to handle very large or infinite data-sets.

Fail-Fast principle states that a system or software should catch potential problems

near to the place where they are introduced. In programming fail-fast can be done by using assertions which leads this technique to be also called assertive programming. Assertions are used to declare how the program state should be and cause failure if the current state does not match the expected state. A common example of an unexpected state is when a method receives a parameter which is Null instead of the expected object.

```
defmodule Example do
  def example({:ok, x}) do
    "Got :ok with value #{x}."
  end

  def example({:error, x}) do
    "Got :error with value #{x}."
  end

  def example(_) do
    "Error. Wrong input."
  end
end
```

Listing 12: Elixir function pattern matching examples.

In Elixir there are two mechanisms to enforce validity of the program state. Both of the ways check the parameters function receives. First of them is pattern matching in function parameters as shown in Listing 12. In the listing there are three functions. Each of the functions take one parameter. Functions will be matched in the order they are introduced. This means that the first function will match with tuple containing two values, atom :ok and any. The second function will match with tuple containing two values, atom :error and any. And the last function will match everything else.

```
defmodule Example do
  def example({:ok, x}) when is_integer(x) do
    "Got :ok with value #{x}."
  end

  def example({:error, x}) when is_integer(x) do
    "Got :error with value #{x}."
```

```
8      end
9
10     def example(_) do
11       "Error. Wrong input."
12     end
13   end
```

Listing 13: Elixir function with guard.

The Second mechanism is called guards. This feature comes from Erlang and it can be used to assert the type of the function parameter. In Listing 13 is the same code from Listing 12 with amended guard clauses. This causes that variable x needs to be of type integer for the functions to be called. With pattern matching and guards programmers can write more assertive code and also fail fast when functions are called with wrong parameters. As the Elixir code is based on supervisors and processes. In Elixir it is actually preferable for processes to fail and then program to return to a known state from before the process was called.

In most modern programming languages there is usually some support of extended character sets. The most common of these is UTF-8. Problem we often encounter is that programming languages do not support this Unicode character set properly[31]. In Elixir all of the strings are stored as UTF-8 and these strings work according to the standard.

It is a common problem with programming languages that the Unicode support does not behave according to the standard. Common operations with problems are, reversing multibyte strings and converting the case of the letter.

3.2.2 Erlang and OTP

When talking about Elixir it is impossible not to talk about Erlang at the same time. This is because a lot of the libraries in the language come from Erlang. Elixir also benefits from all the features provided by the Erlang runtime system (ERTS) and Open Telecom Platform (OTP). OTP and Erlang are actually so intertwined that when the name Erlang is used it means the combination of both language and OTP runtime system and libraries.

The name of the language Erlang, is reference to "Ericsson language" and Danish mathematician Agner Krarup Erlang. Originally the language was developed as a

replacement to Programming Language for Exchanges (PLEX) used in AXE line of telecommunication network switches. The first version of the language was implemented in Prolog and influenced by the aforementioned PLEX language. The language was released as open source in 1998.

Erlang is a functional, dynamically typed and concurrent general purpose programming language. It is known especially for its good support for concurrency. The concurrency support comes from the use of lightweight co-operative processes which are scheduled at runtime level. In some other concurrent languages these processes are called Agents[32]

```
-module(fact).     % Module and file name 'fact.erl'
-export([fac/1]). % Function exported by module

fac(0) -> 1; % If 0, then return 1, otherwise
fac(N) when N > 0, is_integer(N) -> N * fac(N-1).
```

Listing 14: Erlang example, adapted from Wikipedia.

Listing 14 shows an example of a simple Erlang module and a function calculating factorials iteratively. Comments are denoted with %. Erlang code is organized in modules, and here this module is called `fact`. Module also exports one function called `fac` which takes one parameter for other modules to call. This is done with export definition `-export([fac/1])`. If the function would take two parameters it would be exported with `-export([fac/2])` definition. The function `fac` has two implementations. One for parameter zero which returns 1 and another for when parameter is `N > 0`.

Calling a function with a parameter that is smaller than zero or not an integer will cause the function to crash. This crash will be then propagated to the supervisor of the current process. It is common to organize programs in Erlang so that the processes are allowed to crash when there is error. This works because processes are started by the supervisor process which then handles the error by rerunning the process.

As you can see in Figure 1 running application in Erlang is composed of multiple layers. The bottom layer is actual hardware (or virtualized hardware in case of most cloud services. On top of that is the operating system layer, providing file access and other operating system services. ERTS runs on top of the operating system layer. Moving up, the next layer is a BEAM-virtual machine running on top of ERTS.

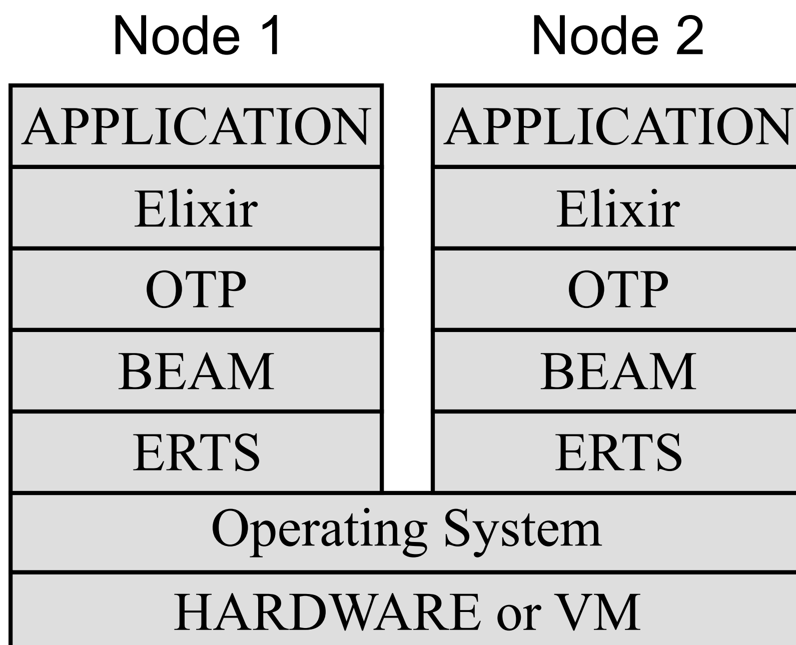

Figure 1: Erlang runtime stack.

OTP runs on top of the BEAM-virtual machine as it is written in Erlang and one can think of it as the standard library for Erlang language. Last, but not least is the application layer or in case of Elixir programs, Elixir layer and Elixir applications.

3.3 Go

Go[33][34] is a programming language designed by Robert Griesemer, Rob Pike and Ken Thompson at Google. The motivation for creation of the language was to address the perceived shortcomings of other programming languages used at Google while also importing the perceived good qualities of those languages. Some of these perceived features are static typing, run time efficiency, reliability, usability, high performance networking and multiprocessing. The languages from where these features were adapted from are C++, Python and JavaScript.

When designing the language three authors had agreement that they would only include features which all three of them agreed on. This meant that some of the features found in other modern programming languages were not implemented in the new language. Some of these are inheritance, generic programming, assertions, pointer arithmetic, implicit type conversions and unions (tagged or untagged). Designers are especially against pointer arithmetic and assertions.

Go was designed 2007 and the first public version was published by Google in 2009. Version 1.0 of the language was announced three years later in 2012. There exist different compilers for the language. There are two compilers which compile Go to native executable. These two compilers are the Google compiler and GCC Go compiler. Google compiler is faster and only supports x86, amd64 and arm platforms. GCC Go compiler is slower, but supports more code optimizations shared with all languages supported by GCC. GCC Go also supports architectures which are not supported by Google compiler. There is also a GopherJS compiler whose responsibility is to compile Go code into JavaScript, which can then be used either in Node.js-runtime or virtual machines in web-browsers.

3.3.1 Simple Go program

Next I will define a simple Go program and go through some basic features of the language. These features include how code is packaged and how the program is compiled and run.

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     fmt.Println("Simple example")
7 }
```

Listing 15: Simple Go example.

In Listing 15 we can see a minimal Go program. The first line defines the package where the code belongs to. Package `main` is special as it is used as the program entry point with `main` function. An `import` function is used for importing other packages. In this example I import one package called `fmt` which is used for formatted I/O operations. If there are multiple packages to be imported, then it is recommended to use the version of import with braces to enclose the package names.

Inside of main function there is only one statement. In this statement we can see how functions exported from the package have names starting with capital letter. Functions which have names starting with a lowercase letter are not exported and are considered private to that package. Strings are denoted with double quotes and

bool	string			
int	int8	int16	int32	int 64
uint	unit8	uint16	uint32	uint64
byte	rune			
float32	float64	complex64	complex128	

Table 2: Basic types provided by Go

statements are by default terminated at the end of the line. It is also possible to use semicolon to terminate the statement.

Because Go is a compiled language, to run this example one needs to use the command **go build** to create an executable. Building is done with information provided by the source files, and separate Makefile or similar is not needed. After the build finishes a single executable file is produced. Users can then run this file in any system which is compatible with the build environment to produce the result.

3.3.2 Language features

Go has a limited set of basic types which are listed in Table 2. Bool is normal boolean type. String type is composed of bytes. The encoding used in strings is UTF-8. Size of the default int and uint is architecture dependent, and is 32bits in code compiled for 32bit systems and 64bits for 64bit systems. Rune is a special type which is 32bits in size and contains one Unicode code-point. It can be thought to have the same functionality as char type in C when it is used in storing characters.

Memory in Go is allocated either automatically in the case of declaring new variables of a certain type or by using keywords `new` and `make`. New keyword allocates a zeroed out block of memory and returns a pointer to it. Type of the allocated pointer is decided by the parameter given to the new-function. Keyword `make` is used when initializing `slices`, `maps` and `channels`. The `make` does not return a pointer. It instead returns a data structure which can be used to access the actual data. Because of this it can be passed as value instead of pointer to functions which mutate the data structure.

```
1 package main
2
3 import "fmt"
4
```

```go
5  type ExampleType struct {
6    intvalue int
7    stringvalue string
8  }
9
10 var pkgVariable int = 0
11 var pkgArray [4]int
12
13 func main() {
14   var a = 1
15   b := 2
16   c := new(ExampleType)
17   c.intvalue = 3; c.stringvalue = "Go"
18   d := make([]int, 5)
19   fmt.Println(a, b, pkgVariable, c, d)
20 }
```

Listing 16: Go memory reservation examples.

In Listing 16 we can see different ways to allocate memory. On line 10 and 11 are the examples on how to declare package level variables. The first one is a declaration of a simple basic type variable, and the second one is declaration of a static array. Inside the main function lines 14 and 15 show the two different ways to declare function level variables. Variable allocation on line 15 is the recommended and usually used form. On line 16 memory is allocated for `ExampleType` struct and its contents are initialized to zeros. Finally on line 18 I allocate `slice` of 5 int-values. Slices are the way to make array manipulation simpler and more dynamic in Go.

After allocating all the memory, the next question is how to release it. As Go is garbage collected language, it does this memory release automatically when memory is no longer needed. To decide when the memory block is no longer needed Go keeps track on the references to it. This usually means that the memory allocated in a function is released when the function exits. Exception to this of course is if the function returns a pointer to the memory it allocated. On the other hand package level variables are allocated when the program starts and will remain in the memory until the program exits.

Another aspect in addition to garbage collection is that Go is a memory safe language. This means it is not possible to access memory which is outside of the

allocated size of an object. This prevents many of the common attacks which try to access outside of the memory area to execute arbitrary code. These types of attacks are very common against programs written in C and C++.

One thing to take note about memory access in Go is that it is not thread safe. So it is important to take care not to use the same memory areas in different Go routines at the same time. These Go routines and how to safely communicate with them will be introduced in the next section.

```
1 package main
2
3 import "fmt"
4
5 type WrittenWork interface {
6     Author() string
7     Title() string
8 }
9
10 type Thesis struct {
11     writer string
12     topic string
13 }
14
15 func (t Thesis) Author() string {
16     return t.writer;
17 }
18
19 func (t Thesis) Title() string {
20     return t.topic
21 }
22
23 func main() {
24     var w WrittenWork
25     t := Thesis{"Jane Doe", "Really exciting thing."}
26     w = t
27     fmt.Printf("type of w = %T, value of w = %v\n", w ,w)
28     fmt.Printf("type of t = %T, value of t = %t\n", t ,t)
29 }
```

```
30 // type of w = main.Thesis, value of w = {Jane Doe Really
     exciting thing.}
31 // type of t = main.Thesis, value of t = {%!t(string=Jane
     Doe) %!t(string=Really exciting thing.)}
```

Listing 17: Example of Go interface.

Go uses a structural type system[35] for its interfaces. In structural type systems the equivalence between two things are determined according to their properties. So two interfaces which have the same properties are considered equal despite their names. In Listing 17 we can see how interfaces can be declared and then a combination of struct and functions created to conform the declared interface. In the printouts we can see that the line 30 struct is accessed as interface and in line 31 it is accessed as plain struct when dumping the contents.

```
1  package main
2
3  type A uint32
4  type B uint32
5
6  func main() {
7      var a A = 32
8      var b B = 64
9      c := a + b
10 }
11 // Results: prog.go:9:12: invalid operation: a + b (
     mismatched types A and B)
```

Listing 18: Example of declaring two nominal types in Go

Suppose that you declare two types A and B like in Listing 18, and A and B share the same basic type. If you try to add them together you will get an error from the compiler complaining that the types A and B are not compatible. This happens because Go is a strongly typed language. Because of the strictness of this type system it is also often called strictly typed language.

As previously mentioned Go programs start executing from a package called main. Go programs are usually divided into packages. One package includes variables, types, functions and sub-packages. Package in Go is defined by creating a folder with a package name. This folder then contains files belonging to the package. Package

can be divided to multiple files to make the code cleaner and more manageable. Sub-package is created just by adding a folder inside of the parent package. Import string for the package would be "package" and for sub-package "package/sub-package".

Go distribution includes a build tool which can be invoked through go command. Build tool will automatically find the project files and compile them to the executable binary.

3.3.3 Concurrency and channels

One of the main features of Go language is easy concurrency by using Go routines. Go routines are lightweight threads scheduled by the Go run-time environment. This scheduling happens in predetermined points. Some of these scheduling points are channel sending and receiving, thread blocking on syscall, garbage collection and starting new a thread using the go keyword.

When a new Go routine is created with the `go` keyword, it goes to the queue of waiting to be scheduled. There is no guarantee that it will be scheduled next. Calling of the go keyword forces the program to run the Go routine scheduler. Go routine scheduler will start executing the Go routine waiting at the top of the execution queue. Also the newly created Go routine will be allocated two kilobytes of stack space which can be grown if needed. Compared to the operating system threads this process of creating Go routine is very lightweight, because it does not involve context switches and saving of CPU registers like an operating system thread would need.

This same lightness is applied to scheduling of Go routines. The Go run-time creates a necessary amount of operating system threads when it starts up, and then it can schedule any of the waiting Go routines to one of these threads. When run-time schedules a Go routine to the currently running thread, from the operating system point of view nothing changes. As Go routines are always in known state when rescheduling, Go run-time just needs to save three registers, namely program counter (PC), stack pointer (SP) and data register (DX). This is very little compared to 48 registers which needs to be saved when switching between operating system threads or contexts.

```
1 package main
2 
3 import "fmt"
```

```
4  import "time"
5
6  func msgSender(sendMsg chan<- string) {
7      sendMsg <- "Hello"
8      sendMsg <- "World!"
9  }
10
11 func msgReceiver(receiveMsg <-chan string) {
12     msg := <- receiveMsg
13     fmt.Println(msg)
14     msg2 := <- receiveMsg
15     fmt.Println(msg2)
16 }
17
18 func main() {
19     msgChannel := make(chan string)
20     go msgReceiver(msgChannel)
21     go msgSender(msgChannel)
22     time.Sleep(2 * time.Second)
23 }
24
25 // Result:
26 // Hello
27 // World
```

Listing 19: Basic example of channels in Go

Languages with multiple threads need a simple and efficient way to communicate between these threads. In Go this is handled by channels. Channels in Go are typed communication streams which can be read from and written to. By default channels are blocking and are created with the `make` keyword.

Listing 19 is an example on how the channels can be used to communicate between Go routines. I first allocate a channel on line 19 which can be used for sending and receiving messages. This channel is then given as a parameter to `msgReceiver` function which is started as a Go-routine. This function will receive a message two times from the channel before it exits. As channels are blocking it will stop on line 12 if it is run before `msgSender` Go-routine is run.

On line 21 I start `msgSender` Go-routine. This is responsible for sending two messages. The first message when the routine is run is "Hello". This is sent to the channel and then the channel blocks and the next Go-routine is scheduled. When `msgReceiver` is scheduled it will read this message, print it out and block on the receiving next message on line 14.

Then at some point `msgSender` is scheduled again to send the second message, and this will then result in the `msgReceiver` receiving the message. The main Go routine will wait for two seconds which should be enough time for both of the other Go routines to execute.

Another thing to note is that programmers can limit the use of the channel. On the `msgSender` function `sendMsg` channel is limited only for sending messages. And on the `msgReceiver` function `receiveMsg` is limited only to receive messages.

In Listing 20 there is a more advanced channel example. It has a buffered channel with buffer size of 5 and an example of dual direction channel as a function parameter. There is also an example of using boolean channel to wait when it is time for the program to end. From the example it can be seen that the channel buffers are fixed and will need to be decided during creation. When the channel buffer is full it will block. Buffered channel will also block if someone tries to read from it when it is empty.

Usually using the same channel to communicate between Go routines is not very practical. In the example of dual direction channel in Listing 20, it is possible that the same process sends and reads messages from the channel. Because of this it is recommended to pass one channel for reading and another channel for writing to the Go routines if two way communication is needed.

```
1  package main
2
3  package main
4
5  import "fmt"
6  import "time"
7
8  func msgSendReceiver(sendReceiveMsg chan string) {
9      msg := <- sendReceiveMsg
10     fmt.Println("received: ", msg)
11     sendReceiveMsg <- "pong"
```

```go
}

func bufferedReceiver(bufferedChannel <-chan string, exit
   chan<- bool) {
    fmt.Println("received: ", <- bufferedChannel)
    fmt.Println("received: ", <- bufferedChannel)
    fmt.Println("received: ", <- bufferedChannel)
    time.Sleep(5 * time.Second)
    exit <- true
}

func main() {
    msgChannel := make(chan string)
    bufferedChannel := make(chan string, 5)
    exitChannel := make(chan bool)
    go msgSendReceiver(msgChannel)
    msgChannel <- "ping"
    msg := <- msgChannel
    fmt.Println("received: ", msg)
    bufferedChannel <- "msg 1"
    fmt.Println("Send msg 1")
    bufferedChannel <- "msg 2"
    fmt.Println("Send msg 2")
    bufferedChannel <- "msg 3"
    fmt.Println("Send msg 3")
    go bufferedReceiver(bufferedChannel, exitChannel)
    <- exitChannel
    fmt.Println("Exit after ~5 seconds")
}

// received:  ping
// received:  pong
// Send msg 1
// Send msg 2
// Send msg 3
// received:  msg 1
```

```
47 // received:  msg 2
48 // received:  msg 3
49 // Exit after ~5 seconds
```

Listing 20: Advanced example of Channels in Go

When implementing multithreading applications in Go, it is important to remember that the memory access between Go routines (and in extension underlying operating system threads) is not exclusive, which means that the Go-routines using the same memory area need to agree between themselves which of them can access the given area of memory. This coordination is usually done using channels.

Another concurrency related issue is that two Go routines can end up in deadlock if they are waiting for messages from each other. This of course is a common synchronization related problem in most languages providing concurrency.

3.4 Conclusion

In this chapter I went through the three different languages used for implementing a web-service back-end. Of these three languages Elixir is functional and Go is procedural language. JavaScript is a mix of functional and procedural programming styles. JavaScript is also the only of these languages to support Object-oriented programming style through its prototype based inheritance.

Another difference between the languages is that JavaScript is a scripting language which is run in a virtual machine without any beforehand compilation. Elixir is compiled to byte-code which can be run in a BEAM virtual machine. Go on the other hand is compiled to machine code which can be then executed in the target processor. Also for Node.js there exists tools to package the node interpreter and all of the JavaScript files into one executable.

There are drawbacks in each of these choices. Go programs can only be run on the processor and operating system type they are compiled to. This is actually true of Elixir programs as well, because they have dependencies to OS level components like libraries, which need to be the same versions used for compiling the program. Difference here is that Go programs can be statically linked. This means the resulting binary can be run on any of the operating systems supporting it.

Each of the languages have an ecosystem comprising tooling, dependency management and libraries. JavaScript is ahead of the other languages with its NPM-package

manager and very active developer community. Elixir also has npm-style package management and can use Erlang libraries. This means that Elixir can use all of the time tested frameworks and libraries from Erlang. Go is the least complete one in this aspect, but it also has most commonly used libraries available, but no centralized way of accessing them.

4 Comparison methodology

In the previous chapter I described three programming languages and their execution environment. In this chapter I will describe methodology for comparing concurrency features of these languages. Traditional way for comparing performance related characteristics of languages is to create series of test programs as evident in [36] [37] [38] [39].

This way of comparing languages isn't without its pitfalls. As described in [39] there are measurable differences on code quality and use of beneficial language features between different programmers depending on their familiarity of the language and programming experience. Also choice of running hardware can affect the comparison results. For example Go and Elixir can benefit from multiple cores more than NodeJS can.

Another way to compare languages is to compare individual features. This can be accomplished by selecting a list of features and then marking up if a given language supports the feature or not. Here problem comes from biases in selecting the features to compare and different ways to implement things in languages which can lead ambiguities on deciding if a feature is implemented or not.

For the comparison I have selected to do feature comparison and empirical performance testing. Feature comparison is selected so that we can form a clear picture of what selected languages offer for the programmer in terms of tackling concurrency. Empirical performance testing on the other hand will give more objective comparison on the performance difference between languages.

4.1 Feature comparison

In feature comparison we will compare the following concurrency related features of the languages. Some of the languages might not have all of the features listed in the following list. I will take note of this when making the comparison and give examples

of ways one could circumvent the missing feature from the language. Things in this comparison mostly affect the implementation phase of the program's life-cycle. And they can make the implementation harder or make it necessary for the programmer to write more code in one of the languages than on the other.

- Multi-threading
- Lightweight user space threading
- Event based asychronous programming
- Thread/concurrent execution starting (implicit / explicit)
- State handling and Mutability
- Inter-process (thread) communication method
- Error handling

Multi-threading was selected as it is a common way to use multiple processors or processor-cores when running in the single node computer. Another choice for this would be using multiple processes with a technique called forking. Main difference with these is that in multi threading the threads can access memory of the other threads. Using multiple processes programmers need to use special memory areas or other inter-process communication methods. Both of these methods suffer from task switching overhead when used in single processor or processor-core.

Lightweight user space threading is used to minimise task switching overhead. Another benefit with this technique is that single thread can have mode dynamic stack handling. Usually implementations start with a small stack per thread and may enlarge it if necessary.

Event based reactive programming is a technique where usually blocking operations are delegated to happen in background asynchronously and will call event hooks when operation finishes or needs to interact with other parts of the program. Techniques like callbacks, promises, events and reactive programming can be used to implement this technique.

In some languages thread starting might need multiple lines of code. In other languages threads can be started with single instruction or even without any programmer interaction. These differences affect how likely and how easily programmers can add thread based concurrency in their programs.

State handling and mutability are important because most non-trivial programs have some state and mutating this state can lead to hard to find and fix errors. One also needs to take care of limiting the access to shared mutable state like variables or memory locations. In some languages there are non mutable variables and as such programmers can not make mutability related errors.

Inter-process or in the context of selected languages inter-thread communication is used for sending messages between different running tasks. There are different ways to archive this[40]. Common among these are either pipes or shared memory based methods. Also there might be different ways to initialize these communication channels.

Error handling[41] also usually comes harder in concurrent programs and it is important to compare the different ways this can be handled in different languages and how this affects the programs implemented in the language.

4.2 Empirical performance testing

I will compare performance and scalability with small REST Web-service. This Web-service will be implemented so that it will have all of the data locally available so that we do not end up measuring database connection speed instead of the implementations made with selected programming languages. Selected functionality should represent common tasks one can find in many of the Web-service back-ends for simple mobile- or web-applications.

The first task will be a login where I will implement password encryption and checking using bcrypt[42] adaptive hashing algorithm for which there are ready made available libraries for all of the selected languages. Time cost of the password generation depends on the computational cost parameter which will be configured to 20 iterations. This will test how languages handle high CPU load situations.

Next task will be querying and modifying the user information. There will be respectively implemented as GET and POST requests. Pattern used will be first storing the user information and then reading it from the server. This will test how the language will handle a modify and read access pattern.

The third task will be a GET task which will simulate long Database query or long request to some back-end service. This test will measure how the language will handle parallelising long running connections which keep connection to the client open for longer time, but are not CPU intensive. For the simulation I will use a five

second wait.

Testing will be implemented in Amazon AWS service using C5n.metal instance hosting the REST-service and four T2.medium and four T2.small instances to generate the load. All of the instances will be running Linux based on Fedora 31 server distribution. Fedora is selected because I was most familiar with it. For allowing the larger amount of connections in tests some of the Linux kernel parameters need to be changed. For server open files per process parameter need to be raised bigger than expected connection count.

Fedora will have the following versions of language run-times and libraries installed.

- Node.js 12.16.1
- Go 1.13.6
- "Erlang 10.6.4, OTP 22, Elixir 1.9.2

Also generating executables and compiling of the programs will be done in the same operating system using the same versions.

All of the servers will be allocated from the same AWS availability zone to minimise the effect of network infrastructure on the testing.

For load generation I will use the Locust load-testing framework. Locust load generating scripts are implemented in python and run on multiple machines coordinated by one master machine.

Load-testing will start with one CPU and then on each testing iteration I will double the number of CPU in use up to and including 64 CPU. For Node.js we use the Cluster API[43] to have multiple Node.js processes listening to the same TCP port. Cluster can be parameterized to tell how many of the OS processes it will create. For Go we will use GOMAXPROCS environment variable to tell how many threads to create for executing user level Go code. For Elixir we can use the Erlang run-time +S option to tell how many scheduler threads to create.

5 Comparison

In this chapter I will first describe the implementation of the three test programs and load generation methods. Second part of the chapter will concentrate on the test

results and analysis. I implemented the program outlined in the previous chapter in the three programming languages described earlier.

The first I implemented was a Go-version. Next I did TypeScript/Node.js-version and the last one I chose to implement was the Elixir-version. All three versions were relatively simple to implement, but each of them needed some language specific extra work, which I will describe more closely in following three chapters.

5.1 Go Implementation

The first language implementation I did was a Go-program. Implementation was quite straightforward using `net/http`-library and `golang.org/x/crypto/bcrypt`-library. For in-memory storage I used Map implementation from Go's sync package. This Map implementation is thread-safe so it can be accessed concurrently from multiple Go-routines. Go code was the most verbose of the three programming languages which is evident in following examples. Most of the verbosity comes from error handling.

```
1  func runServer() {
2    http.HandleFunc(
3        "/password/set/", bcrypt.HandleSetPassword
4    )
5    http.HandleFunc(
6        "/password/verify/", bcrypt.HandleVerifyPassword
7    )
8    http.HandleFunc("/longPoll", sleeper.LongPollHandler)
9    http.HandleFunc("/user/get/", userinfo.HandleGetUser)
10   http.HandleFunc("/user/set/", userinfo.HandleSetUser)
11   http.HandleFunc("/", handler)
12   log.Fatal(http.ListenAndServe(":8080", nil))
13 }
```

Listing 21: HTTP-Request routing code for Go implementation.

For processing HTTP-requests from load generating clients I implemented request routing using aforementioned net/http-library. The function implementing routes and starting server can be seen in Listing 21. Routing is defined by giving a route, for example "/password/set" and a package containing function to handle the request. After necessary request handlers have been defined, HTTP-server can be started

using http.listenAndServe function call. Only parameter needed in our case is a port for listening to the requests. Once started HTTP-server will keep running until terminated.

```
func LongPollHandler(writer http.ResponseWriter, _ *http.
   Request) {
    time.Sleep(5 * time.Second)
    _, err := fmt.Fprint(writer, "OK")
    if err != nil {
        log.Printf("Error writing response: %v", err)
        http.Error(
            writer,
            "Can't write response",
            http.StatusInternalServerError
        )
    }
}
```

Listing 22: Simple HTTP-Request handling function in Go.

In Listing 22 we can see a simple HTTP-Request handler. It does not use any parameters which is why `http.Request` input is assigned to the underscore character. Output is written through `http.ResponseWriter` which is assigned to the writer variable. Handler itself only sleeps five seconds to simulate long lasting backend operation and then returns either OK-text or error.

```

var users sync.Map

func HandleGetUser(
    writer http.ResponseWriter,
    request *http.Request
) {
  username := request.URL.Path[len("/user/get/"):]
  user, ok := users.Load(username)
  if !ok {
    log.Printf("Error retrieving user: %v", username)
    http.Error(
        writer, "User not found.", http.StatusBadRequest
```

```
14       )
15       return
16     }
17     bytes, err := json.Marshal(user)
18     if err != nil {
19       log.Printf("Error marshalling JSON reponse: %v", err)
20       http.Error(
21           writer, "Can't create response",
22           http.StatusInternalServerError
23       )
24       return
25     }
26     _, err = writer.Write(bytes)
27     if err != nil {
28       log.Printf("Error writing response: %v", err)
29       http.Error(
30           writer, "Can't write response",
31           http.StatusInternalServerError
32       )
33     }
34   }
```

Listing 23: Get user request handler in Go

More complex Go-function can be seen in Listing 23. Here we can see how the username is extracted from the incoming URL and how it is used for retrieving user information from sync.Map storing the user information. If a user is found, then this information is marshalled into JSON-object and returned to the requesting user.

```
1
2  const passwordCost = 20
3  var passwords sync.Map
4
5  type Password struct {
6    Password  string  `json:"password"`
7  }
8
9  func HandleSetPassword(
```

```
10    writer http.ResponseWriter,
11    request *http.Request
12 ) {
13   username := request.URL.Path[len("/password/set/"):]
14   body, err := ioutil.ReadAll(request.Body)
15   if err != nil {
16     log.Printf("Error reading body: %v", err)
17     http.Error(
18         writer, "Can't read body", http.StatusBadRequest
19     )
20     return
21   }
22   var password Password
23   err = json.Unmarshal(body, &password)
24   if err != nil {
25     log.Printf("Error reading request body: %v", err)
26     http.Error(
27         writer, "Can't read request body",
28         http.StatusBadRequest
29     )
30     return
31   }
32   passwordHash, err := bcrypt.GenerateFromPassword(
33     []byte(password.Password), passwordCost
34   )
35   passwords.Store(username, passwordHash)
36   _, err = fmt.Fprint(writer, "OK")
37   if err != nil {
38     log.Printf("Error writing response: %v", err)
39     http.Error(
40         writer, "Can't write response",
41         http.StatusInternalServerError
42     )
43   }
44 }
45
```

```
46 func HandleVerifyPassword(
47     writer http.ResponseWriter,
48     request *http.Request
49 ) {
50   username := request.URL.Path[len("/password/verify/"):]
51   body, err := ioutil.ReadAll(request.Body)
52   if err != nil {
53     log.Printf("Error reading body: %v", err)
54     http.Error(
55         writer, "Can't read body", http.StatusBadRequest
56     )
57     return
58   }
59   var password Password
60   err = json.Unmarshal(body, &password)
61   if err != nil {
62     log.Printf("Error reading request body: %v", err)
63     http.Error(
64         writer, "Can't read request body",
65         http.StatusBadRequest
66     )
67     return
68   }
69   passwordHash, ok := passwords.Load(username)
70   if !ok {
71     _, _ = bcrypt.GenerateFromPassword(
72         []byte(password.Password), passwordCost
73     )
74     log.Printf("Password not found for user: %v", username)
75     http.Error(
76         writer, "Password not found",
77         http.StatusBadRequest
78     )
79     return
80   }
81 
```

```
82    err = bcrypt.CompareHashAndPassword(
83      passwordHash.([]byte), []byte(password.Password)
84    )
85    if err != nil {
86      errorText := fmt.Sprintf("Passwords didn't match for
       user: %v", username)
87      log.Printf(errorText)
88      http.Error(writer, errorText, http.StatusForbidden)
89      return
90    }
91    _, err = fmt.Fprint(writer, "OK")
92    if err != nil {
93      log.Printf("Error writing response: %v", err)
94      http.Error(
95          writer, "Can't write response",
96          http.StatusInternalServerError
97      )
98    }
99  }
```

Listing 24: Go password encryption and verification

Password handling is the Most CPU-intensive and complex task tested. In Listing 24 we can see functions for setting password and verifying given password to the stored one. Password encryption is implemented using Bcrypt-library. Library takes password in byte array and returns hash containing unique salt, number or hashing rounds and hashed password.

This password is then stored in sync.Map like user information to be retrieved when using verification function. In case the user is not found, a given password is hashed. This extra step is not strictly necessary in our testing implementation, but in real world application this would slow down attacks trying to determine usernames by calling verify password multiple times with different candidates.

5.2 Node.js Implementation

Next language implementation was with TypeScript to be run in Node.js-environment. The program was implemented using Express framework to handle the HTTP-

Requests, Cluster library for starting multiple Node.js-processes, bcrypt.js for encrypting password and node-ipc for communication between Node.js processes forked to exploit multiple processors.

Complication with Node.js implementation was that it does not support a convenient way to exploit multithreading in user programs. Options which can be used are either web-workers or forking the Node.js processes. I chose forking with Cluster library as the way to use multiple processors. Another problem which came with forking was the communication between forked processes. As there were no good shared memory solutions I decided to use Inter Process Communication (IPC) for sending messages between forked processes. IPC is also the way Express exploits multiple processes to handle the web traffic.

For storing the user information and passwords I implemented a shared hash table distributed over the running Node.js processes. This hash table is used through the IPC. On the client side there is a library which will either send a store or a retrieve call to the node determined by hashing the access key. Hashing is done using the fnv1a hash function. On the listening side data is then stored to JavaScript Map using these keys. When Node.js implementation is run on single process mode, it will bypass the IPC communication. One optimization which could be done is bypassing IPC when storing information to the local process.

```
1  public start() {
2      this.app.use(express.json());
3      this.app.post(
4          '/password/set/:username',
5          this.setPassword.bind(this)
6      );
7      this.app.post(
8          '/password/verify/:username',
9          this.verifyPassword.bind(this)
10     );
11     this.app.get(
12         '/longPoll', this.longPollHandler.bind(this)
13     );
14     this.app.get(
15         '/user/get/:username', this.getUser.bind(this)
16     );
```

```
17      this.app.post(
18          '/user/set/:username', this.setUser.bind(this)
19      );
20      this.app.get('/', this.default.bind(this));
21      this.app.listen(8080);
22  }
```

Listing 25: HTTP-Request routing code for Node.js implementation.

As we can see in Listing 25 Node.js implementation has similar HTTP-request routing implemented with Express as there was in Go implementation shown in Listing 21. It also behaves same way as Go implementation that each URL is mapped to handling function and the server runs after started until terminated.

```
1 private async longPollHandler(_, response) {
2     await this.sleep(5000);
3     response.status(200).send('OK');
4 }
```

Listing 26: Simple HTTP-Request handling function in Node.js.

In Listing 26 we can see the simple HTTP-request handler. This one implements the same functionality as the Go version in Listing 22. Only main difference is that typing of the function parameters is done through type inference and thus it is not necessary to write them out explicitly. Another thing to note is the use of JavaScript async/await to make code more compact and readable. Without async/await programmers would have to explicitly write promise handling code which would make the implementation more verbose and error prone.

```
1
2  private readonly storeClient: StoreInterface;
3
4  private async getUser(request, response) {
5      if (request.params.username) {
6          const data = await this.storeClient.getUserData(
7              request.params.username
8          );
9          response.status(200).send(data);
10     } else {
11         response.status(400).send('Missing username!');
```

```
12     }
13 }
```

Listing 27: Get user request handler in Node.js.

In Listing 27 is the get user handler for Node.js. Here we can see that the TypeScript and Node.js code is more compact than the Go counterpart in Listing 23. Mainly there is less need for explicit error handling as Express will take care of the possible exceptions.

```
1
2 private async setPassword(request, response) {
3     if (request.params.username && request.body.password) {
4         await this.bcrypt.setPassword(
5             request.params.username, request.body.password
6         );
7         response.status(200).send('OK');
8     } else {
9         response.status(400).send(
10            'Missing username or password!
11        ');
12    }
13 }
14
15 private async verifyPassword(request, response) {
16     if (request.params.username && request.body.password) {
17         if (await this.bcrypt.verifyPassword(
18             request.params.username, request.body.password
19         )) {
20             response.status(200).send('OK');
21         } else {
22             response.status(403).send(
23                 'Passwords did not match for the user:' +
24                 `${request.params.username}`
25             );
26         }
27     } else {
28         response.status(400).send('Missing username!');
```

```
29     }
30 }
31 
32 // BCrypt class for handling the password operations.
33 
34 import {compare, genSalt, hash} from 'bcryptjs'
35 import {StoreInterface} from "./store_interface";
36 
37 const PASSWORD_COST = 20;
38 
39 export class BCrypt {
40 
41     private readonly storeClient: StoreInterface;
42 
43     constructor(storeClient: StoreInterface) {
44         this.storeClient = storeClient;
45     }
46 
47     private static async encryptPassword(
48         password: string
49     ): Promise<string> {
50         return await hash(
51             password, await genSalt(PASSWORD_COST)
52         )
53     }
54 
55     private static async testPassword(
56         password: string, passwordHash: string
57     ): Promise<boolean> {
58         return await compare(password, passwordHash);
59     }
60 
61     public async setPassword(
62         username: string, password: string
63     ): Promise<void> {
64         const passwordHash =
```

```
65              await BCrypt.encryptPassword(password);
66          await this.storeClient.storePassword(
67              username, passwordHash
68          );
69      }
70
71      public async verifyPassword(
72          username: string, password: string
73      ): Promise<boolean> {
74          const passwordHash =
75              await this.storeClient.getPassword(
76                  username
77              );
78          return await BCrypt.testPassword(
79              password, passwordHash
80          );
81      }
82  }
```

Listing 28: Node.js password encryption and verification.

Password handling implementation for the Node.js is shown in Listing 28. First there are two functions for setting and verifying the password. These two just take care of the HTTP-request and response handling. Implementation uses Node.js bcryptjs-library. The library does the password hashing in a JavaScript event-model friendly way by yielding the execution time also to other functions in the run-queue. As mentioned earlier, passwords are stored in the map distributed over the running Node.js processes.

5.3 Elixir Implementation

Last language implementation was the one written in Elixir. Elixir implementation uses Mix build tool to manage dependencies and Phoenix-framework to abstract the HTTP connection handling. As Elixir is a functional language which does not have mutable variables it needs some way to handle the mutable state needed by the service. As I did not want to introduce outside services which might influence the testing result I decided to use Erlang Term Storage (ETS) which is a in-memory

key value database included in Erlang.

```
defmodule ScalabilityTestWeb.Router do
  use ScalabilityTestWeb, :router

  pipeline :api do
    plug :accepts, ["json"]
  end

  scope "/", ScalabilityTestWeb do
    pipe_through :api

    post(
        "/password/set/:user_name",
        PasswordController,
        :set_password
    )
    post(
        "/password/verify/:user_name",
        PasswordController,
        :verify_password
    )
    get "/longPoll", LongPollController, :index
    get "/user/get/:user_name", UserController, :get_user
    post "/user/set/:user_name", UserController, :set_user
    get "/", DefaultController, :index

  end
end
```

Listing 29: HTTP-Request routing code for Elixir implementation.

In Listing 29 we can see the HTTP-request routing implementation for the Phoenix in Elixir. One thing to note is the use of Domain Specific Language (DSL) to define the HTTP-Verbs and accompanying paths. Usage of DSL in here strives to make the code more compact and readable.

Each of the HTTP-endpoint definitions have a path, controller module (for example PasswordController) and function to be called (:set_password, :verify_password).

Another thing to note is the pipe_through :api and corresponding pipeline :api definitions which tell Phoenix to treat incoming HTTP-body elements as JSON objects.

```
defmodule ScalabilityTestWeb.LongPollController do
  use ScalabilityTestWeb, :controller

  def index(conn, _params) do
    Process.sleep(5000)
    text(conn, "OK")
  end
end
```

Listing 30: Simple HTTP-Request handling function in Elixir.

The same long http-request emulating function as in Go and Node.js is implemented in Listing 30. use ScalabilityTestWeb, :controller line defines this module as Web-request controller. This controller only implements one function called index. Function itself is as compact as the TypeScript implementation.

```
  def get_user(conn, %{"user_name" => user_name}) do
    user_info = ScalabilityTest.UserStore
        .get_user_info(user_name)
    json(conn, user_info)
  end
```

Listing 31: Get user request handler in Elixir.

User request handler for Elixir is shown in Listing 31. Here we can see that the function gets two parameters. The first parameter is the connection object and the second is the HTTP-request parameters. This function uses a matching operation to match and assign the username passed from the HTTP-router. Because of this implicit parameter checking we do not need to do extra parameter validation which was done in the Node.js and Go implementations for the same function.

After the function gets parameters it fetches the user information from ETS. If ETS does not have the user information it will return just an empty list. json-function in the end of the function writes the resulting user information into the connection as a JSON-object.

```
defmodule ScalabilityTestWeb.PasswordController do
```

```
2    use ScalabilityTestWeb, :controller
3
4    def set_password(
5      conn,
6      %{"user_name" => user_name, "password" => password}
7    ) do
8      password_hash = Bcrypt.Base.hash_password(
9          password, Bcrypt.gen_salt(20, false)
10     )
11     ScalabilityTest.UserStore.set_user_password(
12         user_name, password_hash
13     )
14     json(conn, "OK")
15   end
16
17   def verify_password(
18     conn,
19     %{"user_name" => user_name, "password" => password}
20   ) do
21     password_hash = ScalabilityTest
22         .UserStore
23         .get_user_password(user_name)
24     case Bcrypt.verify_pass(password, password_hash) do
25       true -> json(conn, "OK")
26       false -> send_resp(
27         conn,
28         403,
29         "Passwords didn't match for the user: " + user_name
30       )
31     end
32   end
33 end
```

Listing 32: Elixir password encryption and verification.

Finally in Listing 32 we have a password controller which takes care of hashing the initial password with BCrypt and verification of the existing password. Pass-

Feature \ Language	Go	Node.js (TypeScript)	Elixir
Multi-threading	Yes	No	Yes
Lightweight threading	Yes	No	Yes
Event based asynchronous programming	No	Yes	No
Explicit concurrency	Yes	No	No
Mutable variables	Yes	Yes	No

Table 3: Language feature comparison.

words are also stored in ETS storage, like I did with user information. Here the implementation is more compact than with Go and Node.js.

5.4 Feature comparison

In this section I will compare the three selected languages. Selected comparison metrics are described in section 4.1. and 4.2. In some aspects the languages differ quite much from each other and may have different strategies to attain the same goal. I will note these differences in this comparison.

First is the feature comparison of the three languages. In Table 3 we can see simple **yes** and **no** comparison of the language implementations of common concurrency features.

Here Go and Elixir have run-time multithreading implementation to benefit from multi processor architectures. Node.js does not have a multithreading implementation, but as we could see in the section 5.2. it has a way to start multiple Node.js instances and use IPC to communicate between these instances.

Also Go and Elixir have lightweight threading implementations for organising concurrency inside the single processor more efficiently. In Node.js we archive similar single processor concurrency using events and callbacks based asynchronous programming. In load testing I will explore which of these concurrency strategies will perform better.

Explicit concurrency means that the programmer needs to explicitly make the program behave concurrently. In Go programmers need to explicitly start Go-routine.

Feature \ Language	Go	Node.js (TypeScript)	Elixir
Inter-process communication method	Channels	Pipes	Function calls
Error handling	Function return value	Exceptions Promise rejects	Return values, Exceptions

Table 4: Additional language features.

Another choice is doing the concurrency implicitly in the way Node.js and Elixir do this. In Elixir all functions are implicitly concurrent and in Node.js events and callbacks also naturally lead to concurrent code.

The last difference is mutability of variables. In Elixir all variables are immutable. This means that when a variable is set its value can not be changed. It is possible to rebind the variable name with a new value, but if the variable was passed as parameter to another function it will not change the value passed before rebinding.

This behaviour is different compared to Go and Node.js which have mutable variables. This mutability means that value of memory location referenced by variable can be changed and if another part of the program accesses the value it will now get the new value. Problem with mutable variables is that it can lead to hard to debug concurrency errors and in case of Go programmer also needs to take care that two Go-routines do not access the same variable at the same time.

In Table 4 we can see different ways languages handle the inter-process communication and error handling. Each of the three languages have different ways to accomplish this. The Go programmer needs to explicitly define the channel which is given as a parameter to the Go-routines wanting to use it for communication. In Node.js different processes usually use Operating System pipes to communicate with each other. And finally Elixir implicitly does message passing to do function calls which means that the function can be executed in any of the available threads and there is no need for separate functionality to pass information between the threads.

Error handling between the languages also uses different strategies. The Go function call usually returns a tuple of result and error values. If there was no error, then error part of the tuple is nil. Errors are then explicitly processed in part of the normal program flow.

JavaScript uses the exceptions where function normally returns only successful values and in case of error function throws an exception which is then processed in a separate path using try/catch blocks. Another way to propagate errors is to reject promises which will then be processed in catch function.

The Elixir function can either return error values or throw exceptions. Usually exceptions are not processed using try/catch blocks. Instead in Elixir there are special processes called supervisors which are used to restart processes if they have an error. It is said that in Elixir the default error handling method is: "Let it fail."

5.5 Empirical comparison

Empirical comparison was implemented using Locust-framework. The framework uses Python-programs to run the tests. In Listing 33 we can see how the test is composed. When the script is loaded it reads in pre-generated test users. This way each of the tests will use the same user information for the test. These unique pre generated users are distributed evenly on the load generation nodes.

```
1  from locust import HttpLocust, TaskSet, task, between
2  import csv
3
4  USER_INFO = []
5  with open('users.csv', 'r') as file:
6      reader = csv.reader(file, delimiter=';')
7      next(reader)
8      USER_INFO = list(reader)
9
10 class PasswordTestTasks(TaskSet):
11
12     acceptHeader = {'Content-Type': 'application/json'}
13
14     @task
15     def set_get_password(self):
16         if len(USER_INFO) > 0:
17             user_info = USER_INFO.pop()
18             username = user_info[0]
19             password = user_info[1]
20
```

```
21             self.client.post(
22                 '/password/set/%s' % username,
23                 name='/password/set/[username]',
24                 json={'password': password},
25                 headers=self.acceptHeader
26             )
27             self.client.post(
28                 '/password/verify/%s' % username,
29                 name='/password/verify/[username]',
30                 json={'password': password},
31                 headers=self.acceptHeader
32             )
33
34 class PasswordTest(HttpLocust):
35     task_set = PasswordTestTasks
36     wait_time = between(1, 1)
37     sock = None
38
39     def __init__(self):
40         super(PasswordTest, self).__init__()
41         global USER_INFO
```

Listing 33: Python password set and verify test.

Actual test task is defined in the form of the method set_get_password and decorated with @task decorator. Method is called multiple times during the test run and it will one by one get users from the USER_INFO array. After getting the user information it will create a POST message to the server setting the users password. After this POST method returns it will then make a second request to verify the password.

This test was run on all three languages with different processor configurations. Load generation was done using four T2.medium instances. Each test was run for 10 minutes to see how many requests each language could process during the given time. In Figure 2 we can see the requests completed with each language. Node.js performed worst of all of the languages. It managed to complete the least requests in all of the CPU/client combinations and it also failed to perform without errors in case of 64 CPU and 128 clients combination. With multiple CPU Elixir performs

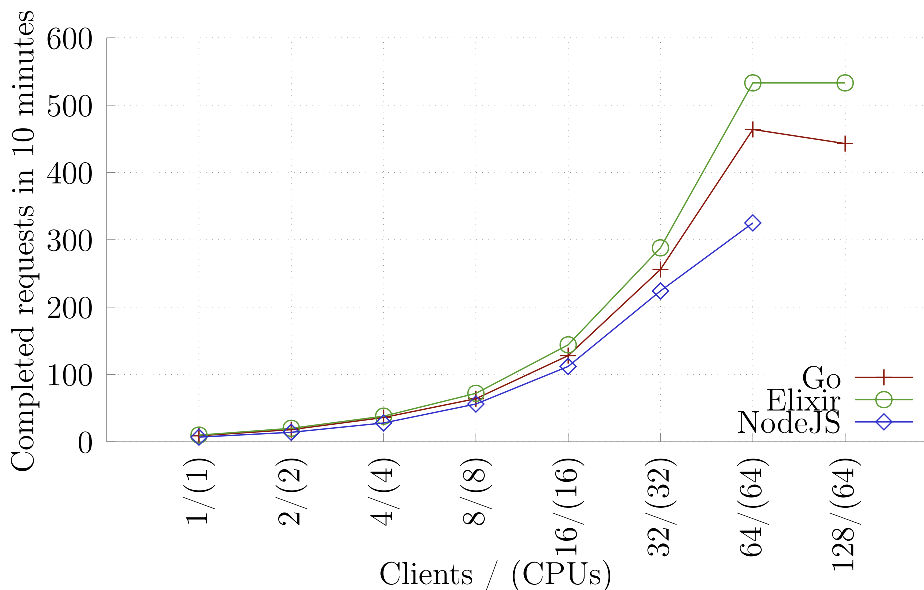

Figure 2: Password test, total successful requests in 10 minute period.

better than GO. It should be noted that with all languages having double the number of clients for CPU started to generate errors, with exception of 64 CPU for Go and Elixir. For this test I used 4 load generation computers.

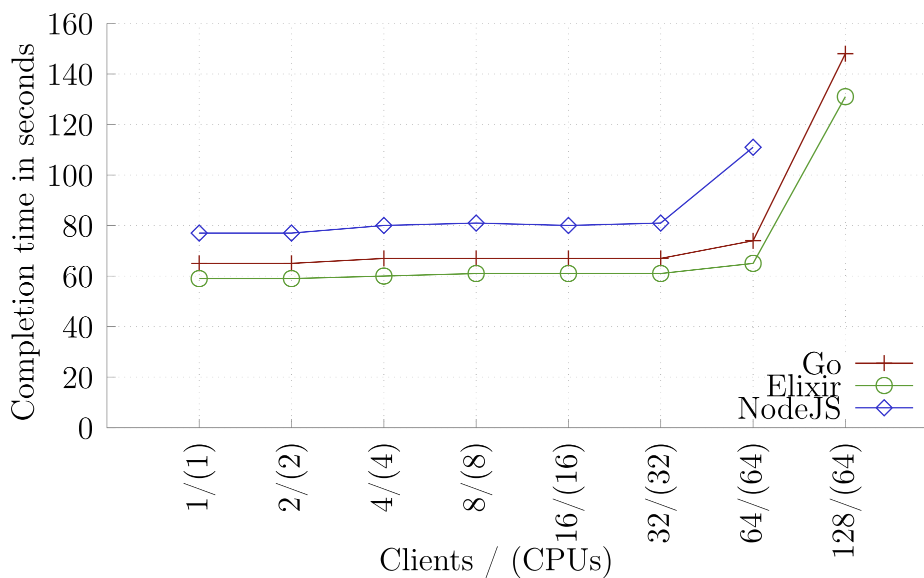

Figure 3: Password test, maximum completion time in seconds for 99th percentile.

As we can see in Figure 3 most languages perform quite nicely with the increase of processors and load. Here also the order remains unsurprisingly the same with Elixir being clearly best performing of the three languages. Another thing of note is that adding the processors has a negative impact to the performance. For this test

I used 4 load generation computers.

Next test is the saving and retrieving user information. In this test I ran two different variants. One with 4000 clients and 4 load generating computers and another with 8000 clients with 8 load generating computers.

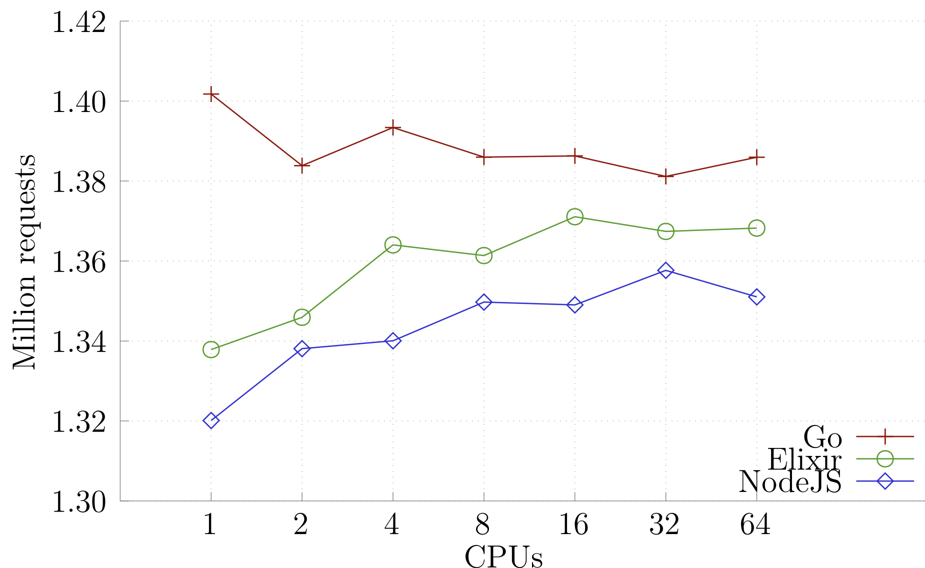

Figure 4: Userinfo test, total successful requests with 4000 clients in 10 minute period.

In the 4000 client test shown in Figure 4 we can see that Go performs best of the languages. Elixir being second and the Node.js implementation worst. For Elixir storage implementation might be affecting the results. All of the requests are close to each other. Only interesting result is that Go implementation performs better with one processor than with multiple ones.

CPU \ Language	Go	Elixir	Node.js
1	2410247	2293269	1971003
64	2451769	2416383	2410811

Table 5: Userinfo test, total successful requests with 8000 clients in 10 minute period.

But as we can clearly see from Table 5 the limiting factor in our test isn't the implementations, but the number of load generating clients. Here we can see that Node.js implementation with one processor performs significantly worse than the other implementations. Order of the implementations still stays the same. For getting better

	Go		Elixir		Node.js	
	50	99	50	99	50	99
1	460	1500	550	1600	550	1600
2	480	1500	490	1700	530	1600
4	470	1500	480	1600	510	1600
8	470	1500	480	1600	500	1500
16	480	1500	470	1500	490	1500
32	480	1500	480	1500	490	1500
64	470	1500	480	1500	500	1500

Table 6: Userinfo test, maximum completion time in milliseconds with 4000 clients on 50th and 99th percentiles.

	Go		Elixir		Node.js	
	50	99	50	99	50	99
1	120	990	180	1100	230	1700
64	110	840	120	880	130	870

Table 7: Userinfo test, maximum completion time in milliseconds with 8000 clients on 50th and 99th percentiles.

results one would need to significantly raise the amount of load generating computers. Here we also don't see a similar rise in request processing for Go with single processor implementation.

For the request completion times we can see in Table 6 that 4000 clients perform quite evenly. Go seems to be quite equal between processor counts and also has the lowest response time with one processor setup. When we look into Table 7 with 8000 clients we can see more differences between the languages. Here Node.js clearly starts to struggle when serving 8000 clients with a single CPU, but when running with 64 CPU the differences mostly vanish.

5.6 Findings

All of the languages performed relatively well in the testing. Mainly Node.js implementation had problems in multiprocessor setups as it has been designed for single threaded operation. All in all the concurrent runtime capabilities of Node.js were

the worst of the languages. In some of the tests Elixir performed a bit better than Go and in others this was reversed.

Implementing concurrent applications in Elixir is the simplest of the languages. This is because the programmer does not really need to think about concurrency at all. For most finetuning possibilities Go is the language of choice. Drawback of this is that it is easier for the programmer to make the common concurrency programming mistakes. Node.js event based concurrency paradigm may need some getting used to if a programmer comes from a more traditional procedural or object oriented style of programming. Go and JavaScript are closer to the more mainstream C influenced languages and Elixir may have a bit steeper learning curve for the people coming from procedural and object oriented languages.

6 Conclusion

In this thesis I have introduced and compared three programming languages, namely Go, JavaScript (with Node.js runtime) and Elixir. All of the languages are used in creating REST backends for web-services.

In the testing I found out that all of the languages had trouble when there were a lot of CPU intensive calculations paired with HTTP-client requests. Another finding was that more regular HTTP-client requests with 8 load generating servers and 1000 clients each and one CPU couldn't produce enough load to make notable differences in any of the language specific test implementations.

In regards to testing, the next steps would be to build a testing environment with more load generating servers, clients and more automation. This environment could be then used to produce more significant differences between the languages. Another thing to do would be optimization of the language implementations. This might lead to better results on the tests.

Of course the weakest result in Userinfo test still managed to handle over 2200 requests in a second. Of course real world REST-backends often have more complicated business logic and slow databases to deal with. One of the further paths would be to study how the behavior of the language implementations change when adding these to the tests.

From a programming point of view I would personally recommend either Elixir or TypeScript as the programming languages for implementing the services. Go has

also its place, but it is quite verbose compared to the other languages. If the uptime of the service to be implemented is very important then the possibility of updating Elixir applications without shutting them down might be useful.

References

[1] M. Welsh, D. Culler, and E. Brewer, "SEDA: An architecture for well-conditioned, scalable internet services", *Operating Systems Review (ACM)*, vol. 35, no. 5, pp. 230–243, Dec. 2001, ISSN: 01635980. DOI: 10.1145/502059.502057. [Online]. Available: https://dl.acm.org/doi/10.1145/502059.502057.

[2] D. Oppenheimer and D. A. Patterson, "Architecture and dependability of large-scale internet services", *IEEE Internet Computing*, vol. 6, no. 5, pp. 41–49, Sep. 2002, ISSN: 10897801. DOI: 10.1109/MIC.2002.1036037.

[3] L. Lamport, "The computer science of concurrency: The early years", *Communications of the ACM*, vol. 58, no. 6, pp. 71–76, Jun. 2015, ISSN: 15577317. DOI: 10.1145/2771951. [Online]. Available: https://dl.acm.org/doi/10.1145/2771951.

[4] *Thread (computing)*, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Thread_(computing).

[5] L. Presser, "Multiprogramming Coordination", *ACM Computing Surveys (CSUR)*, vol. 7, no. 1, pp. 21–44, Mar. 1975, ISSN: 15577341. DOI: 10.1145/356643.356646. [Online]. Available: http://dl.acm.org/doi/10.1145/356643.356646.

[6] C. Severance, "JavaScript: Designing a Language in 10 Days", *Computer*, vol. 45, no. 2, pp. 7–8, Feb. 2012, ISSN: 0018-9162. DOI: 10.1109/MC.2012.57. [Online]. Available: http://ieeexplore.ieee.org/document/6155645/.

[7] *JavaScript*, 2018. [Online]. Available: https://en.wikipedia.org/wiki/JavaScript.

[8] *Futures and promises*, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Futures_and_promises.

[9] K. Kambona, E. G. Boix, and W. De Meuter, "An evaluation of reactive programming and promises for structuring collaborative web applications", in *Proceedings of the 7th Workshop on Dynamic Languages and Applications - DYLA '13*, New York, New York, USA: ACM Press, 2013, pp. 1–9, ISBN: 9781450320412. DOI: 10.1145/2489798.2489802. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2489798.2489802.

[10] *WeakMap.prototype.clear()*, 2020. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WeakMap/clear.

[11] Remy Sharp, *What is a Polyfill?*, 2010. [Online]. Available: https://remysharp.com/2010/10/08/what-is-a-polyfill.

[12] *Source-to-source compiler*, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Source-to-source_compiler.

[13] *State of Babel 2016.12.07*, 2019. [Online]. Available: https://babeljs.io/blog/2016/12/07/the-state-of-babel.

[14] *TypeScript*, 2019. [Online]. Available: http://www.typescriptlang.org/.

[15] *nodejs.org*, 2018. [Online]. Available: https://nodejs.org/.

[16] *Node.js*, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Node.js.

[17] *V8 JavaScript Engine*, 2018. [Online]. Available: https://v8.dev/.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, 1995, p. 395, ISBN: 0201633612. [Online]. Available: https://dl.acm.org/citation.cfm?id=186897.

[19] *https://npmjs.com*, 2019. [Online]. Available: https://npmjs.com.

[20] *Web Workers*, 2020. [Online]. Available: http://www.whatwg.org/specs/web-workers/current-work/.

[21] *Node v10.5.0 (Current) | Node.js*, 2019. [Online]. Available: https://nodejs.org/en/blog/release/v10.5.0/.

[22] *https://elixir-lang.org/*, 2018. [Online]. Available: https://elixir-lang.org/.

[23] *Elixir (programming language)*, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Elixir_(programming_language).

[24] *Erlang (programming language)*, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Erlang_(programming_language).

[25] Erlang Solutions, *Erlang Factory SF 2015 - Panel Discussion*, 2015. [Online]. Available: https://www.youtube.com/watch?v=oZwfi8JZ3kU.

[26] *Macros - Elixir*, 2019. [Online]. Available: `https://elixir-lang.org/getting-started/meta/macros.html`.

[27] C. Strachey and Christopher, "Fundamental Concepts in Programming Languages", *Higher-Order and Symbolic Computation*, vol. 13, no. 1/2, pp. 11–49, 2000, ISSN: 13883690. DOI: `10.1023/A:1010000313106`. [Online]. Available: `http://link.springer.com/10.1023/A:1010000313106`.

[28] *Clojure*, 2019. [Online]. Available: `https://clojure.org/`.

[29] *Docstring*, 2019. [Online]. Available: `https://en.wikipedia.org/wiki/Docstring`.

[30] S. Wlaschin, *Railway oriented programming*, 2014. [Online]. Available: `https://fsharpforfunandprofit.com/rop/`.

[31] E. Mortoray, *The string type is broken*, 2013. [Online]. Available: `https://mortoray.com/2013/11/27/the-string-type-is-broken/`.

[32] Y. Shoham, "Agent-oriented programming", *Artificial Intelligence*, vol. 60, no. 1, pp. 51–92, Mar. 1993, ISSN: 00043702. DOI: `10.1016/0004-3702(93)90034-9`.

[33] *https://golang.org/*, 2018. [Online]. Available: `https://golang.org/`.

[34] A. A. A. Donovan and B. W. Kernighan, *The Go Programming Language*, 1st. Addison-Wesley Professional, 2015, ISBN: 0134190440.

[35] *Structural type system*, 2020. [Online]. Available: `https://en.wikipedia.org/wiki/Structural_type_system`.

[36] V. W. Freeh, "A Comparison of Implicit and Explicit Parallel Programming", *Journal of Parallel and Distributed Computing*, vol. 34, no. 1, pp. 50–65, Apr. 1996, ISSN: 0743-7315. DOI: `10.1006/JPDC.1996.0045`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0743731596900453`.

[37] M. Fourment and M. R. Gillings, "A comparison of common programming languages used in bioinformatics", *BMC Bioinformatics*, vol. 9, no. 1, p. 82, Dec. 2008, ISSN: 1471-2105. DOI: `10.1186/1471-2105-9-82`. [Online]. Available: `https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-9-82`.

[38] J. Fang, A. L. Varbanescu, and H. Sips, "A Comprehensive Performance Comparison of CUDA and OpenCL", in *2011 International Conference on Parallel Processing*, IEEE, Sep. 2011, pp. 216–225, ISBN: 978-1-4577-1336-1. DOI: `10.1109/ICPP.2011.45`. [Online]. Available: `http://ieeexplore.ieee.org/document/6047190/`.

[39] L. Prechelt, "An empirical comparison of seven programming languages", *Computer*, vol. 33, no. 10, pp. 23–29, 2000, ISSN: 00189162. DOI: `10.1109/2.876288`. [Online]. Available: `http://ieeexplore.ieee.org/document/876288/`.

[40] P. K. Immich, R. S. Bhagavatula, and R. Pendse, "Performance analysis of five interprocess communication mechanisms across UNIX operating systems", *Journal of Systems and Software*, vol. 68, no. 1, pp. 27–43, Oct. 2003, ISSN: 01641212. DOI: `10.1016/S0164-1212(02)00134-6`.

[41] J. Xu, A. Romanovsky, and B. Randell, "Concurrent exception handling and resolution in distributed object systems", *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 10, pp. 1019–1032, Oct. 2000, ISSN: 10459219. DOI: `10.1109/71.888642`.

[42] N. Provos and D. Mazières, "A Future-Adaptive Password Scheme", in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '99, USA: USENIX Association, 1999, p. 32.

[43] *Node.js Cluster API*, 2020. [Online]. Available: `https://nodejs.org/api/cluster.html`.