Lappeenranta University of Technology
School of Business and Management
Degree Program in Computer Science

Timo Ruokolainen

Development of a distributed web server utilizing Elixir

Examiners:	Professor Jari Porras
	D.Sc. (Tech.) Antti Knutas

Supervisors:	Professor Jari Porras

ABSTRACT

Lappeenranta University of Technology
School of Business and Management
Degree Program in Computer Science

Timo Ruokolainen

Development of a distributed web server utilizing Elixir

Master's Thesis

2017

65 pages, 19 figures, 5 tables, 1 appendix

Examiners: Professor Jari Porras
D.Sc. (Tech.) Antti Knutas

Keywords: functional programming, Erlang, Elixir, distributed systems, concurrent programming, web server

Elixir is a relatively new functional programming language, which is based on the Erlang programming language. This master's thesis takes a look at the tools provided by Erlang and Elixir, and examines what a functional and concurrency oriented programming language can offer for the development of distributed systems. Elixir runs on Erlang's virtual machine that simplifies the development of concurrent applications. A distributed web server is developed using Elixir, which illustrates the potential benefits gained from utilizing Elixir. The web server employs many of the tools and techniques described in this thesis to induce concurrency, distribution and fault tolerance to the system. It is concluded that Elixir is a strong option for this type of project.

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto
School of Business and Management
Tietotekniikan koulutusohjelma

Timo Ruokolainen

Hajautetun web-palvelimen kehittäminen Elixirin avulla

Diplomityö

2017

65 sivua, 19 kuvaa, 5 taulukkoa, 1 liite

Työn tarkastajat:	Professori Jari Porras
	Tutkijatohtori Antti Knutas

Hakusanat: Funktionaalinen ohjelmointi, Erlang, Elixir, hajautetut järjestelmät, rinnakkainen ohjelmointi, web-palvelin

Elixir on suhteellisen uusi funktionaalinen ohjelmointikieli, joka perustuu Erlang ohjelmointikieleen. Tämä diplomityö perehtyy Erlangin ja Elixirin tarjoamiin työkaluihin, ja tarkastelee mitä funktionaalinen ja rinnakkaiseen ohjelmointiin perustuva ohjelmointikieli voi tarjota hajautettujen järjestelmien kehittämiseen. Elixir toimii Erlangin virtuaalikoneen päällä, joka yksinkertaistaa rinnakkaisten ohjelmien kehittämistä. Hajautettu web-palvelin kehitetään Elixirillä, joka havainnollistaa mitä etuja Elixir voi potentiaalisesti tuoda. Web-serveri käyttää monia tässä työssä kuvattuja työkaluja ja tekniikoita, joiden avulla järjestelmästä saadaan rinnakkainen, hajautettu ja vikasietoinen. Lopuksi päätellään, että Elixir on vahva vaihtoehto tämän tyyppiseen projektiin.

ACKNOWLEDGEMENTS

I would like to thank my supervisor Jari Porras for giving valuable advice. I also wish to thank my parents for all the years of support during my studies.

At Lappeenranta 23.5.2017

Timo Ruokolainen

TABLE OF CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

AST	Abstract Syntax Tree
BEAM	Björn's Erlang Abstract Machine
COP	Concurrency Oriented Programming
CPU	Central Processing Unit
DBMS	Database Management System
DETS	Disk-based Term Storage
EPMD	Erlang Port Mapper Daemon
ETS	Erlang Term Storage
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
IPC	Inter-process Communication
IPL	Information Processing Language
LAN	Local Area Network
LUT	Lappeenranta University of Technology
OS	Operating System
OTP	Open Telecom Platform
PID	Process Identifier
TCP/IP	Transmission Control Protocol / Internet Protocol
URL	Uniform Resource Locator

1 INTRODUCTION

This thesis is about developing a distributed web server using a functional programming language called Elixir. Elixir itself is based on the programming language Erlang. According to Armstrong (2003) developing concurrent systems may seem like a challenging task; however, the task is made considerably more manageable by using a programming language that was designed for the purpose of concurrent programming. Erlang makes it possible to developing systems that are distributed, fault tolerant and scalable. Erlang is a functional programming language that can be labeled as concurrency oriented (Armstrong, 2003, p. 1, 3-5).

Elixir builds on top of Erlang adding some new features. These new features include, but are not limited to, metaprogramming, protocols and mix tool for project management. The added functionality provides better structured source code: reducing duplicate code, boilerplate and noise (Jurić, 2014). Erlang provides Elixir useful facilities for creating concurrent systems, such as the Björn's Erlang Abstract Machine (BEAM) virtual machine and open telecom platform (OTP) library (Jurić, 2015, p. 6, 9-10). Both Elixir and Erlang application run on the BEAM virtual machine, trivializing the process of developing concurrent and distributed applications.

The end result of this thesis is a concurrent and distributed web server, which takes advantage of the concurrency oriented programming (COP) in Elixir, and employs many of the tools and techniques discussed in this paper. Presenting an alternative way to develop a typical concurrent system. The project source code is available in Github at https://github.com/truoko/sdws. This thesis also touches on the more advanced topic of metaprogramming.

1.1 Goals and delimitations

The goal of this thesis is to develop a distributed web server utilizing a relatively new functional programming language called Elixir. In addition to studying the potential benefits of using a functional and concurrency oriented programming language for development of a concurrent and distributed system. This thesis looks at distributed systems mostly in the context of Elixir and Erlang, concentrating on the programming languages, rather than the general problems revolving around distributed programming. This paper provides a description of the Elixir programming language and technologies related to it. A decent level of knowledge of programming in general is recommended, as this paper will not explain the basics of programming.

Research questions for this thesis are defined as follows:

1. What tools do the concurrency oriented Erlang and Elixir provide for development of distributed systems?
2. How to build a distributed web server utilizing Elixir?
3. How suitable is Elixir for developing a typical concurrent, distributed system, such as a web server?

1.2 Structure of the thesis

This thesis is divided into eight sections. The remainder of the thesis is structured as follows. Section 2 provides a summary of the literature review. Section 3 introduces the functional programming paradigm. Furthermore, the advantages and disadvantages of functional programming are explained, in addition to how functional programming can be utilized in practice. Section 4 is about Erlang programming language. It comprises of description of the language, libraries, BEAM virtual machine, hot code upgrade, using Erlang in practice and provides a client server example written in Erlang. Section 5 introduces the Elixir programming language. It provides description of Elixir and its concurrency and distribution orientation. The section continues with the basic building blocks of Elixir including data, processes, macros and an Elixir version of the client server example. This concludes the theory part of this thesis. Section 6 is the practical part of the thesis and presents the implemented distributed web server. Section 7 consists of discussion on the topic and results gained from the implementation. Section 8 gives a summary of the thesis.

2 LITERATURE REVIEW

Three topics were important in particular for the search of literature: functional programming, Erlang and Elixir programming languages. This thesis is based on three books about the Elixir programming language. Namely, Elixir in Action by Jurić, S., Metaprogramming Elixir by McCord, C. and Programming Elixir by Thomas, D. Additional literature was searched using a combination of several different keywords, which included: functional programming, distributed programming, Erlang, Elixir, use case and programming language. Various methods were used to search literature:

1. Web of Science website (webofknowledge.com) by Thompson Reuters, which is a citation index where from one can search and access to research content such as academic papers and books. The search resulted in 510 different records in various topics from the website.
2. Databases which are available to the Lappeenranta University of Technology (LUT). These databases included ProQuest XML, EBSCO databases, Emerald Journals, ScienceDirect, SpringerLink databases and Wiley Blackwell Online Library. These were used for gaining access to material licensed to LUT. The search resulted in 38 records.
3. Searching additional literature based on important authors. These included Joe Armstrong, one of the designers of Erlang and José Valim, the creator of Elixir. For this purpose, Web of Science website and Google Scholar (scholar.google.com) were used. In addition to the previous literature found the search resulted in two records by Armstrong, J.: Concurrency Oriented Programming (2003) and Programming Erlang (2013).

2.1 Analysis of Literature

HAMMER web server (hammer.nailsproject.net) was used to analyze the large set of 510 records from Web of Science. The service provides a network analysis interface for utilizing literature studies scripts. The analysis tool provides a general outline and gives statistical information based on the set of records, which in this case are the 510 records. This set of records would otherwise take researcher a long time to examine manually with huge quantities of literature available (Knutas et al., 2015, p. 1). The analysis resulted in myriad of statistical information.

The interesting part of the results was the 25 most important papers indicated by the analysis. These were the highest scoring records among the analysis. The analysis also included list of papers, which were not part of the original data set. In this list, there was two relevant and useful papers, namely Why Functional Programming Matters (1990) by Hughes, J. and Erlang (2010) by Armstrong, J. The analysis resulted to 27 possible references in total.

Finally, the found literature is filtered and unnecessary papers are excluded. Excluded literature includes papers, journals or books about medical science, neural networks, mathematics, code refactoring and programming languages or tools that are not relevant for this thesis. Furthermore, most of the literature found was on the topic of Erlang. Some of these papers were excluded on the basis of unnecessary repetition of information, which could be found on multiple sources. Other reason for excluding papers was that the literature was not relevant for a distributed system development with Elixir. Table 1 illustrates the final results of literature review.

Table 1. Summary of literature review.

Reference source	Total	Included references
HAMMER analysis	27	6
LUT databases	38	10
Other	5	5
Total	70	21

The analysis of single papers is covered in appendix 1, which includes the analyzed literature in alphabetical order. Most of the literature was based on Erlang, and unfortunately so far Elixir has very few scientific papers and journals available. Although, this was expected as Elixir is still a relatively new functional programming language. Overall, a good amount of references is included on all three topics of functional programming, Erlang and Elixir. This was helped by having a base of programming books about Elixir. Many of the references also included programming and case examples, of which both are well suited for this thesis.

3 FUNCTIONAL PROGRAMMING

Erlang and Elixir are both part of the functional programming paradigm, and it will be discussed in this section. According to Pickering (2007) of the three dominant programming paradigms functional programming is the oldest, with the first functional language information processing language (IPL) appearing in 1955, followed by Lisp in 1958. Despite of being the oldest paradigm it has not been the most prominent one. Procedural languages were particularly lucrative financially and were the most used since their appearance. Functional languages were mostly used in the academic world. Although, these days functional languages are beginning to be utilized more in other fields, and are solving increasingly complicated problems, naturally including also more trivial problems (Pickering, 2007, p. 1).

Functional programming is influenced by mathematics. It makes use of functions that do not modify the state of the application and are free of side effects. Functional programming in its purest form is a group composed of multiple functions, and these functions work by receiving arguments and giving back results. It has immutable data, meaning that once created the data cannot be changed. More specifically, the data is permanently in the memory, until it is garbage collected later. The result of immutable data is that functions always produce new data and do not change the arguments. Thus, new data is produced by using copies of original data and transformed with functions (Pickering, 2007, p. 1).

3.1 Advantages of functional programming

Hughes (1990) explains, ”The functional programmer sounds rather like a medieval monk, denying himself the pleasures of life in the hope that it will make him virtuous” (p. 2).

Functional programming can be described merely as an immutable, side effect free and mathematically inspired language, where the sequence of execution will not matter. However, Hughes (1990) further argues that this simple description is lacking. The main assets to the functional paradigm are the benefits to modular style of programming. It is widely known that modular design is required for successful software development. The objective is to develop software with small, simple and generic modules. Though, this is still not enough to realize the power of modules; the functional language must also facilitate ways to combine these modules (Hughes, 1990, 1-3).

According to Hughes (1990) there are three important advantages to consider with modularity to increase productivity (Hughes, 1990, 3):

1. Small sized modules are fast to develop.
2. The use of generic modules can be repeated in consecutive software. Resulting in quick development.
3. To cut down time used to debug programs, it is possible to test modules separately.

Using the functional paradigm will reduce the amount of code in a program, as opposed to imperative programming, which usually would result in a longer solution. Developers can get more work done with fewer work hours and resulting in more sustainable code that is simple to update (Hunt, 2014, p. 35). Furthermore, reduced code size and complexity will result in fewer bugs in the system (Haenisch, 2016, p. 30). In an experiment by Gat (2000) using Lisp resulted less development time and smaller code size compared to Java and C or C++. In addition to superior ability to perform during runtime compared to Java and similar runtime with C or C++. Despite the fact that Lisp is not very popular today (Gat, 2000, 21-23).

A case study by Haenisch (2016) studied the benefits of utilizing functional programming in an Internet of Things (IoT) software. The software was used to lower power usage in paper machines by utilizing a sensor network. Figure 1 depicts code size comparison by building same system with three different programming languages. The languages seen on the bottom are C, Ruby and Elixir. The code size is measured in lines, and the result was that by using Elixir the amount of code shrunk down by a factor of 5 (Haenisch, 2016, p. 33-34).

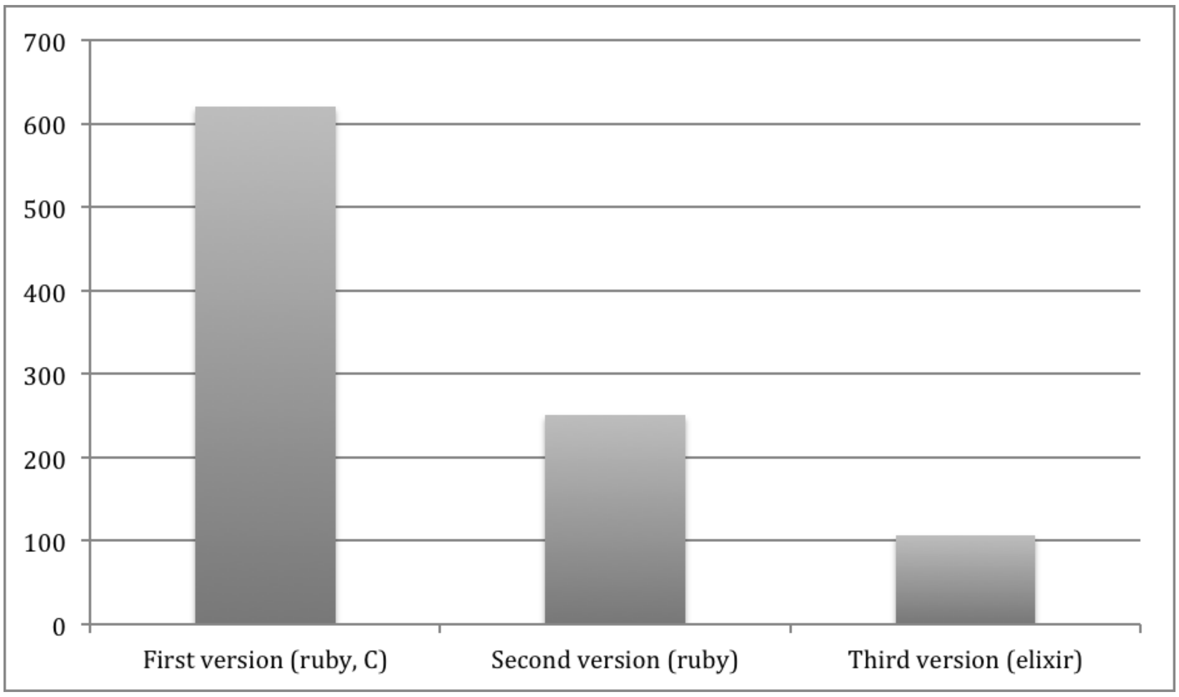


Figure 1. Code size measured in lines (Haenisch, 2016, p. 35).

Hunt (2014) explains that in functional programming functions are isolated to avert possible side effects. A black box view at a function should have a defined input and output with no quantitative side effects. Without side effects, generic functions can be used repeatedly and have no unpredictable behavior. This makes it easy to use functions that match the specifications exactly. The outcome is code that is simple to produce, test, manage and secure to reuse (Hunt, 2014, p. 33-35).

As mentioned, functional programming advocates the use of immutable data. Immutable data is beneficial when creating concurrent systems. Threads and processes can work as a group without negative effects on one another. In other words, immutable data is more thread-safe than mutable data, since there will be no conflicting data. Furthermore, recursion is the inherent approach to flow control. Common constructs concerning iteration and looping, such as for or while loops can be achieved via recursive functions (Hunt, 2014, p. 35-36).

3.2 Disadvantages of functional programming

Functional programming also has several disadvantages keeping it from becoming the mainstream programming paradigm. According to Hunt (2014) disadvantages to functional programming include (p. 36-37):

- The functional programming paradigm is less known to developers and may find it difficult to get started.
- Functional programming idioms are not as intuitive to developers who are used to conventional programming. This may result in difficulties regarding debugging and maintenance.
- Input-output is more difficult. Data in, result out style of functional programming is not as suitable for input-output situations, where stream style processing is superior.
- Functional paradigm is less suited for creating interactive systems where actions are governed by user requests.
- Infinite looping or otherwise perpetual applications can be challenging to develop.
- Current hardware is not optimized for functional programming.
- Functional languages are not data oriented. Systems that rely on database reads and writes and manage that data may benefit more from using object oriented languages.
- Functional programming has been generally though as a programming paradigm that belongs to academic use. This style of thinking is slowly changing though.

3.3 Functional programming in practice

After defining the functional programming paradigm, its practical implications will be discussed mostly relating to parallel and distributed programming. According to Burton (1986) functional programming has been utilized in the academic field for years for artificial intelligence research using programming languages such as LISP.

Furthermore, rising prices of software compared to hardware pushes for development utilizing functional languages (Burton, 1986, p. 1).

Functional programming is one of the approaches, aside from procedural and relational solutions, to be utilized in parallel computing. Since functional programming offers some advantages for developing applications that are simple to read, write, transform and verify (Burton, 1986, p. 1). According to Hammond (1994) in functional programming, it is fairly trivial to divide the problem into individual modules or functions and process them in parallel. This parallelization is possible since, as previously mentioned, in pure functional programming functions do not have side effects. Functional programming offers better automation of parallel processes, typically with more options compared to imperative programming languages. Moreover, a pure functional program which has predetermined input may be run in sequence or in parallel, and nevertheless have the same output. In addition to the parallel program ending in the same state, unless this is prevented by lack of available resources. This type of determinism in functional programs offers some benefits which include: not requiring a parallel machine for working out bugs, tasks can run in any sequence and deadlocks are not possible in most situations (Hammond, 1994, p. 1, 7).

As functional programming works well with parallel computing, it can also be utilized in distributed systems. According to Bal et al. (1989) one of the problems that distributed programming has to consider is parallel execution. The general consensus is that functional programming languages are suitable for distributed programming; however, the language needs to be effective in allocating tasks among the free central processing units (CPUs). Functional programming is mathematically based, and functions can be run in any sequence, for instance, in a function h(f(3,4), g(8)) either f or g function can be assessed first, in addition to being able to run in parallel to one another. The programming language does not necessarily have to be a pure functional language, but an impure language will need a method to decide which expressions are and are not run in parallel in a distributed system. Some potential challenges also exist by using functional languages. Namely, one might want to avoid running very simple functions in parallel to avoid unnecessary messaging to pass results, in a case where the parallel execution is too costly. Another potential challenge is to decide whether argument evaluation is performed by using a local CPU or a remote one. In an optimal situation, a compiler will determine which CPU to utilize for a specific function (Bal et al., 1989, p. 270, 273, 305).

4 ERLANG PROGRAMMING LANGUAGE

Armstrong (2003) explains, “The real world, the world in which we live and breathe and are born and die is *concurrent*. Paradoxically, the programming languages which we use to write programs which interact with the real world are predominately sequential” (p. 1).

Programming applications with sequential programming languages may give the perception that developing a concurrent application has to be challenging. The task can be made more straightforward by using a programming language that was designed for the purpose of writing distributed, fault tolerant and scalable systems (Armstrong, 2003, p. 1, 3). Consequently, these are some of the main concepts that drove the development of Erlang.

Erlang is open source and was created in 1986 by a Swedish telecom company called Ericsson. Erlang can be a good choice for anyone wanting to write fault tolerant and concurrent systems, providing the essential concurrency primitives with extensive, tested libraries (Larson, 2009, p. 48). Armstrong (1997) explains that the team behind Erlang wanted to create a programming language that resembles Prolog, but would have better support for concurrency and error handling (Armstrong, 1997, p. 196).

4.1 Open Telecom Platform

Erlang ships with a library called open telecom platform (OTP). Although it was produced having telecom systems in mind in the first place, it has many other possible applications in the realm of developing reliable distributed systems (Armstrong, 2003, p. 5). OTP contains extensive collection of tools, and as of 2010, 49 subsystems are incorporated. Every system has their own use and are very capable. For example, one of these subsystems is Mnesia, which is a relational and real-time database. With huge development teams, it is useful to have standardized solutions to frequent problems. There is no need to reinvent the design for common problems. Libraries can be utilized instead, which provide a set of standard solutions for creating fault tolerant systems (Armstrong, 2010, p. 73).

OTP also offers behaviors in addition to useful tools. Behaviors abstract away common problems by offering solutions to a nonfunctional part of a problem. A developer’s job is to make the functional part of the problem, usually called a callback module. For instance, OTP offers the means to do live updates to a running system. The mechanism for switching an old module to a new one is always the same; OTP knows how to phase out the old module. The developer provides only the functional parts that will change depending on the situation (Armstrong, 2013, p. 361). One of the most important behaviors is the GenServer module, which deals with processes and communication. It will be discussed in section 5 in the context of Elixir.

4.2 BEAM virtual machine

Björn's Erlang Abstract Machine (BEAM) is a virtual machine used to compile Erlang to C, and afterwards the code can be compiled using a regular C compiler. The performance is similar to C in many cases (Armstrong, 1997, p. 197). According to Hausman (1994) compiling Erlang code to C gives three advantages (p. 1, 17):

1. Since C compilers are abundant and available to nearly every processor good portability can be achieved. The only requirement is that the processor includes a gcc compiler.
2. Great optimization of hardware in low-level.
3. Erlang applications can be connected to applications written in other languages by utilizing interfaces, which work with C.

Furthermore, the BEAM virtual machine is convenient, since operating systems (OS) cannot be expected to support quick messages between processes and a massive amount of concurrent processes. For true portability across differing systems, the virtual machine manages scheduling, memory management and sending of messages (Larson, 2009, p. 55). Figure 2 illustrates how the BEAM virtual machine works. Starting from the top there are the CPU cores. The cores can be as numerous as one has available in the system. Rest of the figure consists of the BEAM virtual machine running on a single OS process. The BEAM itself consisting of several schedulers that allocate work, handle multiple processes and make parallelization achievable. Each scheduler works as an OS thread and normally there exists same number of schedulers as CPU cores. Lastly, on the bottom of the figure there are the processes that do the allocated work (Jurić, 2015, p. 6, 135).

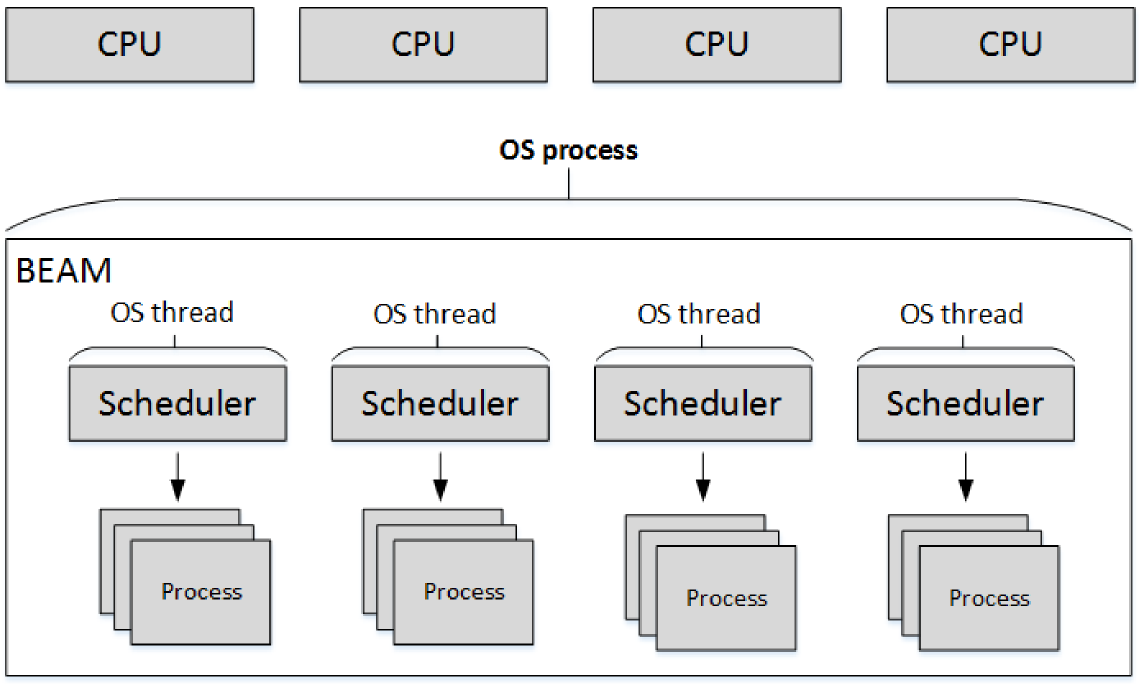


Figure 2. BEAM virtual machine (Jurić, 2015, p. 135).

According to Jurić (2015) the maximum limit of processes in a BEAM instance in theory is around 268 million. An Erlang process can be called a “basic concurrency primitive”, a unit of concurrency for building concurrent systems. Normally a system is running a lot of processes, up to millions of them. Processes are used excessively in Erlang systems, since the processes are cheap, requiring minuscule amounts of memory (Jurić, 2015, p. 6, 135).

4.3 Hot code loading

Erlang and by definition also Elixir has the ability to update modules without system down time. This feature was brought by the need to have telecom applications constantly running without disturbance even when updating the system. The Erlang runtime system which consists of the BEAM virtual machine, kernel and standard library has a global table that includes module addresses. When performing a code swap, the old module address is changed to contain the address of the new module. The new module will be referred as the “current” and the old one as “old”, as these are the two module types that can exist in Erlang systems. It is possible to run different versions of the module concurrently and new calls to the module will be directed to the new version (Zhang, 2011, p. 11; Ericsson AB, n.d.).

4.4 Erlang systems

Erlang was originally used for creating telecom switches that handle even hundreds of thousands of concurrent connections. Other important considerations were the need for complicated data structures, data storage, fault tolerance and the ability to scale the system when needed. Concurrent message passing utilizing complicated data structures can be seen in many Erlang based systems. The language is efficient for the purpose of creating systems with similar requirements to high performing telecom switches. However, it can also be used for general-purpose development (Armstrong, 2010, p. 75).

One might ask that what kind of system has similar requirements to telecom switches, or otherwise benefit the most from Erlangs characteristics and libraries. Table 2 illustrates which systems gain and which systems lose from utilizing Erlang according to Larson (2009).

Jurić (2015) explains from personal experience that it is possible to build two comparable web servers systems, where the other system employs only Erlang as a sole technology. The example in question includes two concrete web server projects that have existed. The web servers had several different technical demands, including: concurrent client connections, taking care of lengthy requests, controlling global in-memory state, persistent data that endures restarts and executing tasks in the background. The non-Erlang web server utilized different technologies for all of these demands, which were all needed for solving problems, where the other technologies proved lacking. The second server only made use of Erlang for similar prerequisites. Though, it is noted that an early version of a system may use Erlang as the sole technology, but should employ other technologies if required (Jurić, 2015, p. 8-9).

Table 2. Erlangs suitability for software projects (Larson, 2009, p. 56).

Suitable for	Unsuitable for
Irregular concurrent programs, which require deriving parallelism from different concurrent tasks.	Concurrency more appropriate to synchronized parallel execution.
Network servers.	Programs utilizing abundance of floating-point numbers.
Distributed systems.	Code that needs nonportable instructions.
Parallel databases.	Code that needs an aggressive compiler (Except when benchmarking process spawns and message passing).
Graphical user interfaces (GUI), interactive systems.	Programs that utilize libraries, which need to run in different execution environments, for example, in Java virtual machine.
Monitoring, control and testing tools.	Programs which need to utilize large libraries written in other languages.

Finally, it is not mandatory to program an application with only using Erlang. According to Larson (2009) Erlang can also be used to program a part of a system cooperating with parts built in different programming languages, only using Erlang for the portion of the system, which benefits the most from utilizing Erlang. Erlang includes a C library that can be used for linking to different parts of the system. With the library, non-Erlang parts of the system can look as if they are Erlang processes, with a possibility to use Erlang’s process messaging. Furthermore, protocols that utilize text or binary while using basic inter-process communication (IPC) structures can also be employed (Larson, 2009, p. 56).

4.5 Client server example in Erlang

The client server architecture is commonly used, and it is also utilized in section 6 in the implemented web server. Figure 3 illustrates a client server module written in Erlang. The server has a simple “Hello world!” type of function, where the client sends a string to the server and the server replies with a new string, adding “Hello” in front and an exclamation mark after. The module uses basic process communication and utilizes pattern matching to examine the process messages. The module consists of four functions, which are explained below:

1. Start/0 function starts the server process by using spawn/3. The zero in start/0 is the arity of the function; referring to the amount of arguments it takes. This way of identifying unique functions is used throughout this thesis. According to Virding et al. (1996) the process created using spawn/3 is concurrent. The function returns a process identifier (PID), which is utilized for exchanging information among processes. Spawn/3 will return instantly, returning the PID and without pausing to assess the spawned function (Virding et al., 1996, p. 67).
2. Client/2 function sends a message and waits for a response from the server. Virding et al. (1996) explains that the PID is used to determine the process one wants to communicate with. Once the PID is known, communication between processes is possible using the following pattern: “Pid ! Message” to send messages and “receive … end” block to receive the messages. Between the receive-end block, one can pattern match against messages to pick messages from a process mailbox, which collects the received messages. Client/2 function adds its own PID to the message using a tuple, this way the server process will know which process to send the reply to (Virding et al., 1996, p. 68-71).
3. Recursive server/0 function waits for client messages to get requests, and afterwards sends the result back. Similar to the client, correct message is recognized by utilizing pattern matching. The server also uses a guard to check that the message contains a list, since in Erlang strings are implemented as lists. Virding

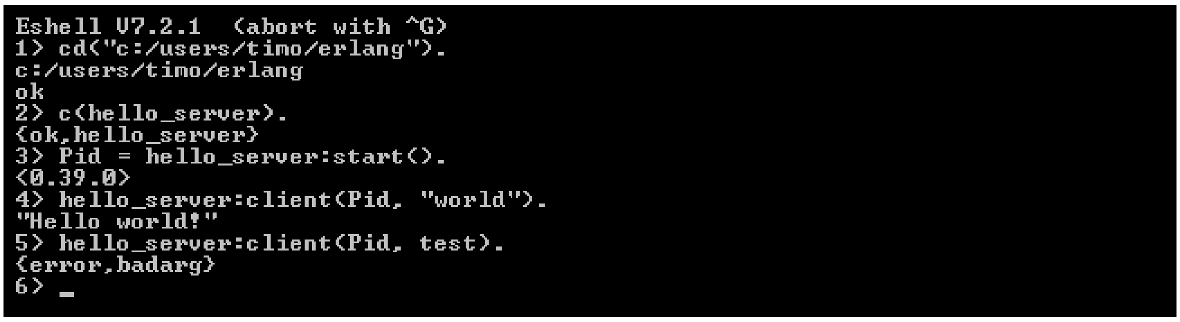
et al. (1996) defines guards as conditions that have to be met, otherwise the clause right after the guard is not picked. In other words, guards can be though to enhance the pattern match it is used in conjunction (Virding et al., 1996, p. 28).

4. Finally, the concat/1 function does string concatenation for given string, adding "Hello" in front and an exclamation mark after.

```erlang
-module(hello_server).
-export([start/0, server/0, client/2]).

% Spawn a new process running the server function
start() ->
    spawn(hello_server, server, []).

% Client function for sending requests to the server and waiting for
response
client(Pid, String) ->
    % Sends message to the process identified by Pid
    Pid ! {self(), String},

    % Receives messages sent back to this process from the server
    receive
        {Pid, Result} ->
            Result
    end.

% Recursive server function, started with spawn()
server() ->
    % Receives the string sent from client and sends new string back
    receive
        {From, String} when is_list(String) ->
            From ! {self(), concat(String)},
            server();
        {From, _} ->
            From ! {self(), {error, badarg}},
            server()
    end.

% String concatenation function
concat(String) ->
    "Hello " ++ String ++ "!".
```

Figure 3. Simple client server example in Erlang.

In figure 4 the usage of the module can be seen in Erlang’s shell. The second command compiles the code. Start/0 function returns the PID for the server, client/2 function sends a message with the PID as the first argument and the server loop returns the answer.



```
Eshell V7.2.1  (abort with ^G)
1> cd("c:/users/timo/erlang").
c:/users/timo/erlang
ok
2> c(hello_server).
{ok,hello_server}
3> Pid = hello_server:start().
<0.39.0>
4> hello_server:client(Pid, "world").
"Hello world!"
5> hello_server:client(Pid, test).
{error,badarg}
6> _
```

Figure 4. Using the client server module in Erlang’s shell.

5 ELIXIR PROGRAMMING LANGUAGE

According to the creator of Elixir, José Valim, Erlang is superb for developing distributed systems; however, there were some essential features that could be added. For instance, these included metaprogramming, polymorphism and superior tools. Because of these demands Elixir was created (Thomas, 2016, foreword).

Thomas (2016) explains, “Programming should be about transforming data” (p. 1). Elixir can match the description of pure functional programming very well. According to Pickering (2007) pure functional programming was about using groups of functions, taking inputs to create new data, exhibiting no side effects and not altering the state of the application. In addition to utilizing recursion for iteration (Pickering, 2007, p. 1). According to Thomas (2016) in Elixir functions can be combined together, passing the result along a chain of functions, transforming the inputs into desired outputs. This works in a similar manner as Unix, just replacing the shell commands with functions. Furthermore, Elixir has immutable data and makes running parallel functions simple by utilizing numerous cheap processes on top of the Erlang virtual machine (Thomas, 2016, p. 1-3).

5.1 New features in Elixir

Elixir and Erlang are remarkably similar languages. They both create OTP applications, work on top of BEAM virtual machine and Elixir can use Erlang functions, and vice versa. Aside from differences in syntax, there are several new features in Elixir. According to Jurić (2014) Elixir brings new features for structuring the code better. This means new ways to reduce duplicate code, boilerplate and noise within the code (Jurić, 2014).

Metaprogramming is one of the tools for structuring code. McCord (2005) explains that with metaprogramming it is possible to expand the language via macros. Defining new, reusable features that do not exist yet in Elixir and its libraries. For instance, Elixir does not include the while loop, but with metaprogramming it is possible to recreate it in Elixir as a macro. Likewise, other custom keywords can be added to the language, if one finds that something essential is missing (McCord, 2015, p. 1-2, 5).

Another very useful feature is the pipeline operator “|>”. With the operator one can transform data with no extra in-between variables. It is possible to start with data, pass that data through several functions and end up with desired result. The transformation of data can be read with ease from top to bottom (Jurić, 2014). Figure 5 illustrates the power of the pipeline operator. An arbitrary list of 1 to 5 is created and three iteration functions are used to transform the data. With the pipeline operator, the result is passed to the next function.

```
Enum.to_list 1..5                  # Create a list of integers from 1 to 5
|> Enum.map(&(&1 * &1))            # Calculate square of every element
|> Enum.reject(&Integer.is_even/1) # Drop even numbers
|> Enum.shuffle()                  # Put the list elements in random order

[9, 25, 1]                         # End result after all transformations
```

Figure 5. Pipeline operator example.

Protocols are also implemented in Elixir. A protocol is fairly comparable to what an interface is in object oriented languages. The usage of any data structure is possible with protocols, as long as there exists an implementation for said data structure. The appropriate function implementations are created for each data type and the correct function is chosen depending on the input at runtime. Thus, protocols make polymorphism possible (Jurić, 2014; Jurić, 2015, p. 125).

The last addition to Elixir mentioned here is the mix tool. According to Jurić (2014) the tool simplifies the process of creating new projects in Elixir, hiding redundant details from the user. It is to help the user to construct fully operational concurrent systems. Merely seven files are automatically generated in the process. Furthermore, developers can expand the mix tool by adding new tasks to the tool (Jurić, 2014).

5.2 Concurrency and distribution orientation

Steen & Tanenbaum (2016) define distributed systems loosely as, “A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system” (p. 968). Furthermore, according to Shankar (2013) the entities that run applications can be called threads or processes. A distributed application is run simultaneously by using numerous processes, which are interacting collectively and might be located in multiple computers or in a single computer. These days, distributed systems are utilized everywhere, as before they merely existed in operating systems. Nearly all programming languages have supporting structures for distributes systems. These include spawning processes for running functions and tools for synchronization. With multiple processes, an application can affect other parts of the system more than just in the beginning and at the end of execution (Shankar, 2013, p. 1).

According to Bal et al. (1989) there are three requirements for distributed programming languages that differ from utilizing sequential programming. The requirements are explained below (Bal et al., 1989, p. 266-267):

1. Using multiple processors. The first requirement for distributed applications is the ability to execute code on more than one processor. Meaning that varying parts of the application are allocated to multiple processors. This increases the performance of the system by using available CPUs optimally. In addition to performance, parallel execution can enhance reliability and availability for creating more fault tolerant systems. Processors also may have differences in structure or data that are essential for executing a specific function.
2. Cooperation between processors. Communication and synchronization between processes has to be achievable. Processes encaging in parallel processing can trade temporary results and synchronize their operations if required to do so. Services in distributed systems need the support of other processes, for instance, in fetching a certain file using a file service.
3. Possibility of a partial failure in the system. With a system merely utilizing one

processor, all work stops when there is a problem with the CPU. In distributed systems, though, this is not a problem due to the availability of multiple CPUs. Other CPUs can take on the work left by the failing CPU, losing some of the computing power, but nevertheless the system sustains its full functionality. Thus, fault tolerant applications can be developed that identify hardware failures and recover from them.

The requirements for distributed programming can be achieved by the operating system or the programming language itself. Advantages, such as better readability, portability and static type checking are possible, when utilizing a programming language designed for distributed programming. In addition to an increasingly abstract and high level message transmitting, compared to the model offered by the operating system (Bal et al., 1989, p. 267). Armstrong (2003) further argued that concurrency should be achieved through the programming language, rather than the operating system. This makes it possible to not depend on operating system architecture in the matters of scheduling and synchronization, when creating concurrent systems. If this is not the case, a potential problem arises when the operating system does not support cheap processes, or it is not possible to use concurrent processes in huge quantities. Moreover, concurrent behavior might not be the same between different operating systems (Armstrong, 2003, p. 1).

Next, the basic requirements for a distributed programming language will be discussed in the context of Elixir and Erlang. Starting with the first requirement that was the usage of multiple processors. According to Jurić (2015) the BEAM virtual machine is able to make use of all accessible CPU resources. A process in BEAM instance executes concurrently to other processes, and thus can execute in parallel. Although, concurrency alone does not guarantee parallel execution and make things faster. Processes executing concurrently have their own execution context, and the processes can run in parallel only with more than one usable CPU. The BEAM virtual machine handles the process task scheduling and also their concurrency. BEAM schedulers give all processes chance to run tasks, until their time window is over. Afterwards, the scheduler replaces the executing process with one of the other processes that are waiting for execution (Jurić, 2015, p. 134-136).

The second requirement was cooperation between processors or processes. One of the concepts of Elixir is to execute smaller pieces of code that are self-sufficient and execute these pieces concurrent to one another. Processes in BEAM are indeed self-sufficient and isolated from one another, sharing no data. The cooperation between these processes can be done by sending messages. Aside from error handling and monitoring this is the only medium of cooperation that may be utilized. Since the communication is done with messages and without sharing data, many of the familiar methods for synchronizing processes are not required: these include locks, mutexes and semaphores. Standard messages are asynchronous in essence, meaning that after sending a message, the process proceeds to execute new tasks. If one wants to synchronize this process there is no preexisting architecture for it. Instead, the sender process can add its PID to the message and pause until it gets a reply. The process that receives the message can recognize the sender process from the received PID and respond. Furthermore, the cooperation of CPUs is handled by the BEAM virtual machine. The BEAM schedulers use all accessible CPUs in the system and the schedulers are able to allocate tasks from processes to CPUs (Thomas, 2016, p. 171; Jurić, 2015, p. 6, 141).

To address the third and final requirement of partial failures in the system, according to Armstrong (2010) Erlang has a “let it crash” mentality: if a failure occurs in a process it can just crash and afterwards different processes can fix the problem by, for example, restarting the process. It is not a big deal if one process would crash, since Erlang based systems usually have abundance of light processes running. Isolated processes that crash cannot negatively influence different processes, unless they are specifically linked to each other by the developer. Furthermore, Erlang has dynamic type checking along with data that is devoid of corruption, leading to data that does not require considerable user checking. Systems utilizing the OTP library can also use supervisor trees. These trees are built from nodes that consist of multitude of processes. Nodes that are lower in the three hierarchy are the worker processes that do tasks that are allocated to them, and nodes residing higher up the tree are the supervisors that can observe other processes, including the workers, while detecting and fixing problems (Armstrong, 2010, p. 70-71).

5.3 Clusters in Elixir and Erlang

According to Thomas (2016) in Elixir nodes are actually instances of the BEAM virtual machine. Simply by running an application it can already be considered a node. Clusters can be formed by connecting BEAM instances together, and it is possible to use same machine, local area network (LAN) or the Internet. A lot of the utility a node supplies to processes in its own BEAM instance can be used in other nodes. This kind of inter-node communication adds scalability and reliability to systems (Thomas, 2016, p. 193, 202).

Trottier-Hebert (n.d.) explains that after creating an Erlang virtual machine node, the node establishes a connection to Erlang port mapper daemon (EPMD). EPMD allows nodes to register, communicate with different nodes and give alerts in case of name conflicts, basically working as a name server. EPMD uses the port 4369 by default. In addition, to communicate between nodes, a random port is appointed for every node. These ports can also be changed to have a specific range. This also means that having a very large cluster will need a great deal of connections among the nodes (Trottier-Hebert, n.d.).

5.4 Data in Elixir

In Elixir data is dynamic. For instance, a variable is not an integer unless one puts an integer in it. Meaning that the kind of data the variable accommodates currently, decides the variable type (Jurić, 2015, p. 21).

5.4.1 Data immutability

As Elixir is a functional programming language, it has immutable data. This means that once data has been created it cannot be changed, the data will reside in the memory until garbage collected. Although, it is possible to bind values again to refence another value. All data structures are immutable in Elixir, ranging from integers to more complicated structures like nested lists. The key point is that new copies of data will be returned, when dealing with functions transforming data (Thomas, 2016, p. 19-22).

Immutable data is practical since one can trust that code does not modify existing data in unexpected ways. This is useful when dealing with systems with abundance of processes and more than one thread of execution. A piece of code may or may not be executing in parallel with other code, but data immutability gives some protection to avoid side effects, which can result in an unknown state of the program (Thomas, 2016, p. 19-20).

5.4.2 Pattern matching

Pattern matching is, to a certain extent, same as assigning values to variables in imperative programming languages. The difference is that in Elixir assignments can be called assertions and the equal sign is called a match operator. When doing a pattern match, Elixir makes an attempt to produce a left side that is equal to the right side. The creator of Erlang, Joe Armstrong, relates the equal sign to the one utilized in mathematics. One needs to relearn the way mathematical functions work, in order to grasp the idea behind pattern matching (Thomas, 2016, p. 13-14, 18).

A value on the right side of the match operator can be bound to a variable on the left side. Furthermore, binding multiple values with variables using one pattern match is possible, for example, by utilizing data structures like lists. To have a successful pattern match two restrictions apply: structure on both sides has to be similar and all elements in the pattern on the left side need to have a value to match with on the right side. Elixir and Erlang assertions are almost equivalent. The difference between them is that Elixir permits matching and binding variables multiple times, and in Erlang assignments can be done one time for a single variable (Thomas, 2016, p. 14-15, 18).

5.4.3 Data storage

There are a few ways to store data in Elixir, but two options will be discussed here, namely Erlang Term Storage (ETS) and the Mnesia database. Both can be found from the standard

OTP library.

Erlang term storage (ETS) is comparable to a rudimentary database, which utilizes memory for data storage. Even though designed for Erlang, it works with Elixir. Further worth mentioning is that there exists a disk-based term storage (DETS), with the difference that the data goes to the hard disk. In ETS data is saved as tuples, which is a data type that includes several pieces of data in one. Furthermore, tuples do not pose many restrictions neither on the type nor the amount of data. A single item in the tuple is always appointed as a key. There are a few ways to define the relationship between data and the key, the default setting being a “set”, where every key corresponds to single entry and no more. Other processes can discover tables by utilizing a return value, returned after table creation. In addition, there is an option to let processes find tables by name (Laurent & Eisenberg, 2014, p. 33, 136-138).

Mneasia is an entirely distributed database management system (DBMS) which is powered by Erlang, and utilizes ETS and DETS. Similar to other OTP systems also Mnesia works with Elixir. The systems name comes from a Greek word “mnesia” meaning memory (Laurent & Eisenberg, 2014, p. 146). Mnesia was created to be utilized in telecommunication systems, similar to the design philosophy behind Erlang. Mnesia is very closely tied to Erlang, as it works in the same address space with the overlying program. This makes Mnesia unique among other DBMS, offering some interesting advantages. Mnesia provides quick database lookups, fault tolerance by having copies of the database in different nodes and permitting database configurations even when the program is still running. In addition, the complexity of the data put into the database is not restricted, even items such as trees or functions are fine. Normally, if the overlying program crashes the DBMS is affected, since Mnesia runs in the same address space as the main program. However, this is alleviated by using a functional and concurrency oriented language, where functions do not have side effects and problems can even be contained to a single process letting the main program to keep running. Most parts of the DBMS were developed as Erlang applications with the exception of the query language, which is included as a component of the Erlang syntax (Mattsson et al., 1999, p. 3-4, 7-8).

5.5 Processes

Elixir systems are generally built from groups of processes, and these processes work as self-sufficient components with the ability to use messages for inter-process communication. Elixir's process orientation makes it convenient to run distributed tasks over many CPUs and machines. Even the Elixir terminal is executed in a process. Processes are identified by using PIDs, messages sent with send/2 and spawn/3 is utilized to run a function in a process. Similar to Erlang, processes possess a mailbox, where the messages can be inspected utilizing the "receive … end" block (Laurent & Eisenberg, 2014, p. 97-99).

Process PID consists of three integers that are unique to the current running instance of BEAM. Messages can be sent when the receiving processes PID is known. Messages are asynchronous by default. Thus, if one wants to know if the message reached its destination or if the destination process even exists, a communication pattern is required. Figure 6 illustrates the basic communication pattern between processes. The sender process sends a message and begins to wait for a reply. In addition, it adds its PID to the message, so that the receiver process can identify where the message came from, and lastly the receiver process sends a response (Laurent & Eisenberg, 2014, p. 97-98; Jurić, 2015, p. 141).

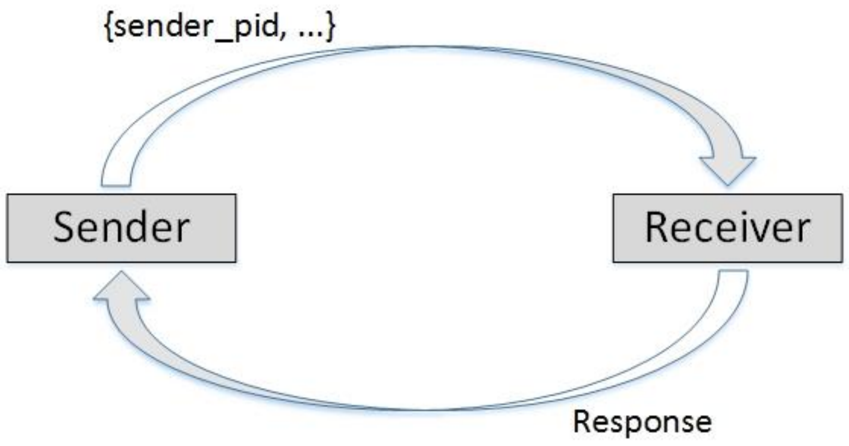


Figure 6. Synchronous send and receive pattern (Jurić, 2015, p. 141).

Furthermore, according to Trottier-Hebert (n.d.) there is no difference in using data structures like, for instance, PIDs in the current BEAM instance or a remote one. Once the PID of a process is known, communication is possible and messages can be sent to remote processes (Trottier-Hebert, n.d.).

5.5.1 Generic server

Generic server or GenServer is likely the most important of the behaviors in Elixir. To some extent, behaviors can be compared to abstract functions in other programming languages. According to Laurent & Eisenberg (2014) behaviors are ready made solutions for process initiation and interaction. The developer needs to provide callback modules to respond to certain events related to the utilized behaviors (Laurent & Eisenberg, 2014, p. 154).

The GenServer behavior supplies a collection of methods to initiate a process, answer to incoming requests and exit the process in an elegant manner (Laurent & Eisenberg, 2014, p. 154). Writing functions for managing messages from scratch is more troublesome in general than using the methods ready in GenServer. GenServer provides solutions for low-level process tasks, which can be used by most servers, since they do not differ much in their basic requirements. GenServer callback functions are set up into one module and depending on the circumstance the correct function is invoked (Thomas, 2016, p. 204).

To build a service utilizing GenServer, there are a few methods that are required with some of them being optional to use. These methods include: init/1, handle_call/3, handle_cast/2, handle_info/2, terminate/2 and code_change/3. For building a fairly elementary service, init/1 is needed to start the process and then either the synchronous handle_call/3 or asynchronous handle_cast/2 method to handle requests coming to the service. Handle_info/2 handles all other messages not related to the GenServer calls or casts. Terminate/2 is for stopping the process and commencing clean up in case of failure or after receiving a shutdown signal. Code_change/3 is used for hot code loading while maintaining the service state (Laurent & Eisenberg, 2014, p. 154).

5.5.2 Supervisors

Supervisors make it possible to develop resilient, fault tolerant systems. Including systems utilizing the Erlangs “let it crash” mentality that was previously mentioned. According to Thomas (2016) supervisor’s responsibility is to monitor and restart worker processes. Supervisors can be made using the OTP supervisor behavior. The behavior takes a list of processes to be supervised, instruction how to act in case of a dying process and information about countering situations where a process can keep restarting in a loop. Supervisors can prevent the entire system coming to a halt from a single failure or error. It is better to merely restart a very small part of the system, for instance, only one process. The risk in this case is losing the progress of the singular process, a relatively small problem compared to system wide downtime. In other words, in Elixir it is essential to keep the system functional as a whole than to pay attention to failing code (Thomas, 2016, p. 219).

Furthermore, more intricate systems may utilize supervisor trees. In supervisor trees, supervisors are constructed in a deep hierarchy, where supervisors oversee other supervisors, and the tree continues like this with again more consecutive follower supervisors or worker processes. All single worker processes completing individual tasks can be supported by a supervisor and restarted independently (Jurić, 2015, p. 221-223; Laurent & Eisenberg, 2014, p. 161).

5.6 Metaprogramming

Metaprogramming was one of Elixirs defining new features. According to McCord (2015) metaprogramming gives the power to extend the language via macros. Macros are for creating features that one might need, but are missing from the language, generating new keywords that are now part of the language. For example, macros make possible to create increasingly productive libraries, domain-specific languages and systems with improved performance. This is possible since macros enable code to generate more code itself (McCord, 2015, p. 1, 5).

Nearly all programming languages use an abstract syntax tree (AST), which is a lower level presentation of the code typically hidden from the user. Both arguments and return values are ASTs in macros. During compilation, the code is put into a tree hierarchy and afterwards transformed to, for example, machine code. Even though AST is typically only used by compilers and people creating programming languages, Elixir allows developers to access the AST form and utilize it to generate powerful code. Elixir developers are in possession of the same tools that were utilized to build the standard library (McCord, 2015, p. 2-5).

The AST form can be better understood by comparing it to a programming language where the source code is composed as an AST. Table 3 illustrates how Lisp code is very similar to Elixir ASTs. The main difference being that Elixir uses brackets instead of parentheses. In Elixir, the quote keyword exposes the AST form of the expression, which in this case is 2 * 3 + 1. An AST of an expression is presented as a tuple with three elements. For example, a slightly more simple expression like 6 - 1 is represented as: {:-, [context: Elixir, import: kernel], [6, 1]}. The First element contains an atom expressing a function call, the second element has some metadata and the third element includes an argument list. Essentially, the advantage in Elixir is the capability to access low level ASTs directly, while also retaining a higher-level syntax (McCord, 2015, p. 3, 8-10).

Table 3. Lisp code compared to Elixir AST representation (McCord, 2015, p. 10).

Lisp:	Elixir (metadata truncated):
(+ (* 2 3) 1)	quote do: 2 * 3 + 1 {:+, _, [{:*, _, [2, 3]}, 1]}

Figure 7 illustrates the syntax used with macros. The macro has the same function as the Erlang server seen previously in figure 3. It takes a string and uses the string concatenation operator “<>”, adding “Hello” in front of the string and an exclamation mark after. The first macro uses the AST form directly and the second macro uses the quote keyword, both returning the same result. For example, passing “world” to either macro will return “Hello world!”.

```elixir
defmodule Concat do
    # Using AST form
    defmacro hello(string) do
        {:<>, [context: Elixir, import: Kernel],
        ["Hello ", {:<>, [context: Elixir, import: Kernel], [string, "!"]}]}
    end

    # With quote
    defmacro hello_q(string) do
        quote do
            unquote("Hello ") <> unquote(string) <> unquote("!")
        end
    end
end
```

Figure 7. String concatenation macro.

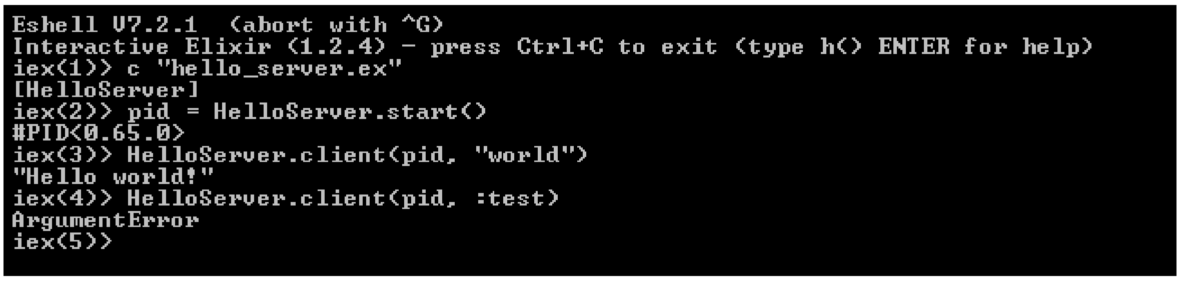
5.7 Client server example in Elixir

Figure 8 illustrates the Elixir version of the client server architecture example from section 4. The structure of the module is the same as in the Erlang version, but the differences in syntax are obvious. The server has the same simple “Hello world!” function using string concatenation and utilizes pattern matching. Server/0 and concat/1 functions are set private, meaning they cannot be invoked outside of the module. One difference to Erlang is that strings in Elixir are implemented as binary, instead of lists. Furthermore, to make the server/0 function recursive, it only needs to call itself once at the end of the function, instead of adding the call on all of the possible pattern matches in the function.

```elixir
defmodule HelloServer do
    # Spawn a new process running the server function
    def start() do
        spawn(fn -> server() end)
    end

    # Client function that sends requests and waits for response
    def client(pid, string) do
        send(pid, {self, string})

        receive do
            {_pid, result} ->
                result
        end
    end

    # The recursive server function. Waits for messages and sends back the
resulting string
    defp server() do
        receive do
            {from, string} when is_binary(string) ->
                send(from, {self, concat(string)})
            {from, _} ->
                send(from, {:error, ArgumentError})
        end

        server()
    end

    # String concatenation function
    defp concat(string) do
        "Hello " <> string <> "!"
    end
end
```

Figure 8. Simple client server example in Elixir.

Figure 9 depicts the usage of the module. The module works the same as the Erlang version. The first command compiles the code and afterwards functions start/0 and client/2 are called to use the module.



```
Eshell V7.2.1  (abort with ^G)
Interactive Elixir (1.2.4) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> c "hello_server.ex"
[HelloServer]
iex(2)> pid = HelloServer.start()
#PID<0.65.0>
iex(3)> HelloServer.client(pid, "world")
"Hello world!"
iex(4)> HelloServer.client(pid, :test)
ArgumentError
iex(5)>
```

Figure 9. Using the client server module in Elixir’s interactive shell.

6 IMPLEMENTATION

The implemented project is a distributed web server. This section gives a description of the implemented system. The web server utilizes most of the tools and techniques discussed thus far. The online Cambridge University Press (n.d.) dictionary defines web server as, “A computer system or program that sends websites and information to internet users”.

A web server can be described as conventional system that is able to utilize concurrency effectively. It uses numerous hypertext transfer protocol (HTTP) sessions that need to be sustained and run concurrently. As web traffic increases, web servers have to manage multitude of requests per second. A part of the requests may be lost or experience delay in the case that the web server is unable to answer to the increasing work load from requests. In particular, request regarding dynamic web pages can cause the CPU becoming a bottleneck. Having capacity for managing numerous concurrent HTTP connections is an asset (Armstrong, 2003, p. 4; Iyengar et al., 1997, p. 1943).

6.1 Requirements

The function of the web server is a rudimentary collection of notes. Notes can be viewed and added using a web browser or client module. Table 4 describes the requirements for the web server. The functional requirements are made from the attributes that the server at minimum is a web server, which is concurrent, distributed and fault tolerant. In addition, distribution transparency is added as a non-functional requirement. Steen & Tanenbaum (2016) explain that distribution transparency is one of objectives in distributed systems, which should be taken accord in the systems design. The system seems to the user as a singular entity where, for example, the user does not know where a task is executing and data’s exact location is a matter of little importance (Steen & Tanenbaum, 2016, p. 970).

Table 4. Requirements for the web server.

Functional requirements		
ID	Requirement	Priority
FR1	Clients must be able to store notes on the server.	High
FR2	Clients must be able to view the saved notes.	High
FR3	The implemented web server needs to be able to answer client HTTP requests.	High
FR4	The implemented web server needs to be able to return a HTTP response.	High
FR5	The web server must be concurrent and be able to manage multiple HTTP requests at any given time.	High
FR6	Forming a cluster needs to be possible; multiple web server nodes can be started and added to the cluster.	High
FR7	The web server nodes need to have data replication. Connecting to any node will result in having access to the same data.	High
FR8	The web server must be able to recover from crashing processes.	Medium
Non-functional requirements		
ID	Requirement	Priority
NFR1	Distribution transparency: a user connecting to the system and using the service should not be able to tell whether the web server is distributed or not.	Medium

6.2 Used technologies

The web server is developed in the Microsoft Windows environment using Visual Studio Code with installed support for Elixir syntax. The web server is built with added distribution and fault tolerance by using the modules and behaviors found in the Elixir standard library and Erlang’s OTP library. This includes the distributed Mnesia used as the server database. Utilized behaviors include GenServer and Supervisor. GenServer is used to initiate processes and for process messaging. Supervisor is used for monitoring other supervisors or processes, while attempting to restart them in case of a failure. External libraries are not utilized, as the modules and behaviors provided by the standard libraries should suffice for the purposes of this project.

6.3 System topology

The web server connections are made using gen_tcp module from the standard Erlang library. Gen_tcp offers the Transmission Control Protocol / Internet Protocol (TCP/IP) socket interface (Ericsson AB, n.d.). The web server client requests and server responses follow the HTTP/1.0. protocol. The default port is 80 according to the protocol, but the port can also be changed (RFC1945, 1996).

Figure 10 illustrates the server topology for one node. The TCP supervisor holds the listen socket and listens for incoming TCP connections. The supervisor passes the socket to a pool of TCP acceptor processes, which accept connections from clients. Messages received from the TCP socket are transformed to normal Elixir process messages, and afterwards passed on to worker processes. Finally, the worker processes form an appropriate HTTP response based on the received request. The TCP acceptor processes work as a middleman accepting data from clients, and from there on it is possible to send the data to any node using regular process messaging.

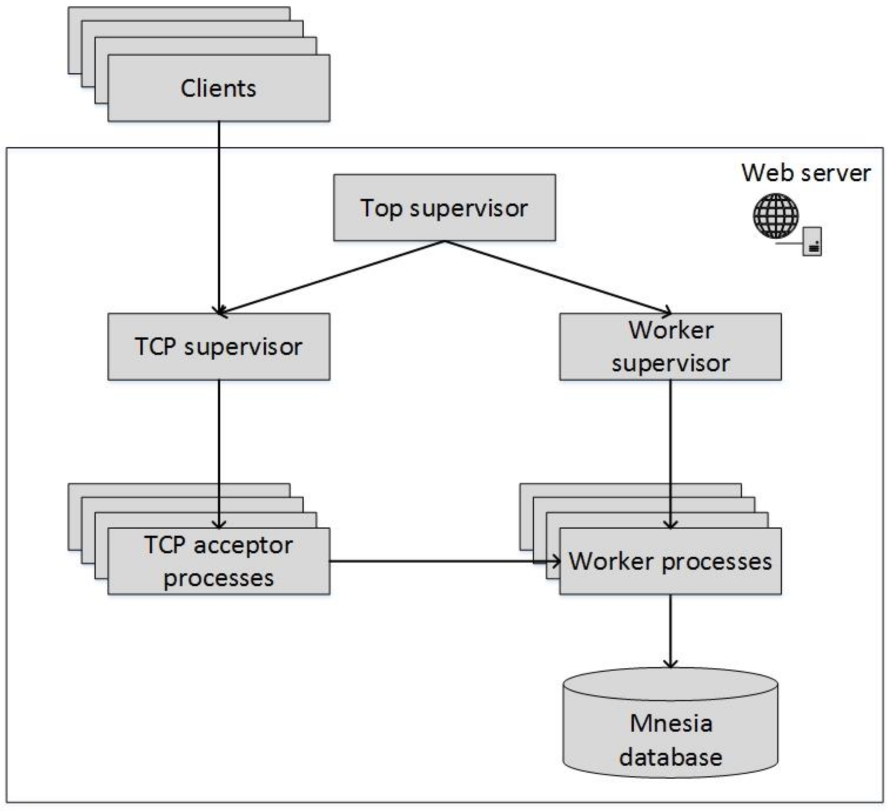


Figure 10. Topology of a single node.

Figure 11 illustrates a cluster with four BEAM virtual machine nodes. All of the nodes are identical in structure and interconnected. Trottier-Hebert (n.d.) explains that the inter-node connections are made using TCP, and when a new node connects to any of the other nodes in a cluster, it will be connected to all of them (Trottier-Hebert, n.d.).

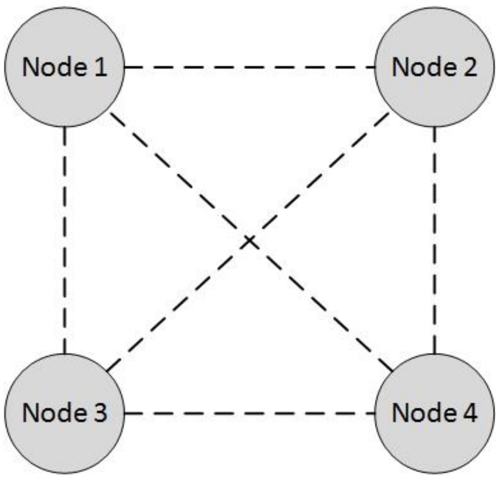


Figure 11. Topology of a cluster (Trottier-Hebert, n.d.).

6.4 Node features

The web server has GET and POST methods implemented. The server parses the received HTTP request to get the following information: method, requested resource and message body, if any. The HTTP response from the server is assembled based on this information. If the requested method is GET, the worker processes read the notes data from the database and the data is sent to the client. If successful, the client gets a “200 OK” reply and the notes data as an HTTP table. In a case where the requested method is POST, text is parsed from the request and stored into the database. The client receives a “204 No Content” reply, indicating that the request was successful, but no new resource was created in the process.

Aside from the client module in the web server, the preferred way to use the server is naturally with a web browser. Table 5 depicts the web server resources and the used uniform resource locator (URL).

Table 5. Web server resources.

Resource description	URL
View the notes data	http://hostname:port/
Add a note to the database	http://hostname:port/add_note?text=”…”

6.5 Concurrency

Concurrency is essentially provided by the BEAM virtual machine, as long as tasks are run in separate processes. The limiting factor is the amount of CPU cores available. The web server pre-spawns a pool of processes ready to accept incoming TCP connections. After a process accepts a new TCP connection, a new process is spawned to take its place in the process pool. Furthermore, a pool of worker processes is spawned to generate the HTTP replies.

To confirm that the server is concurrent, JMeter (http://jmeter.apache.org/) is used for load testing. Figure 12 depicts the BEAM virtual machine scheduler usage during high load with a quad core CPU. 1000 concurrent users per second are simulated for a duration of 30 seconds. All four schedulers are utilized, while running processes in parallel. The web server throughput was around 692 requests per second using pools of 100 processes.

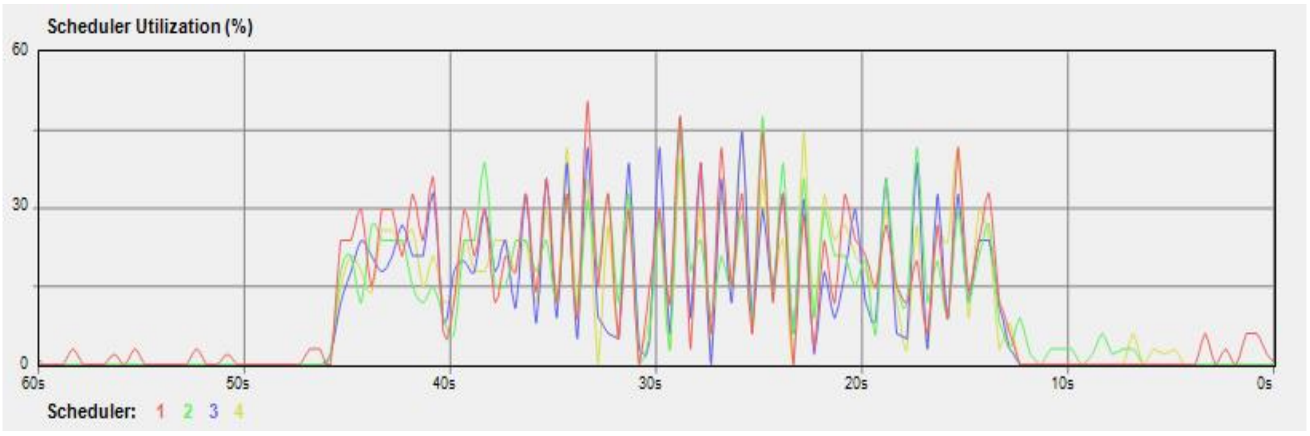


Figure 12. Scheduler utilization.

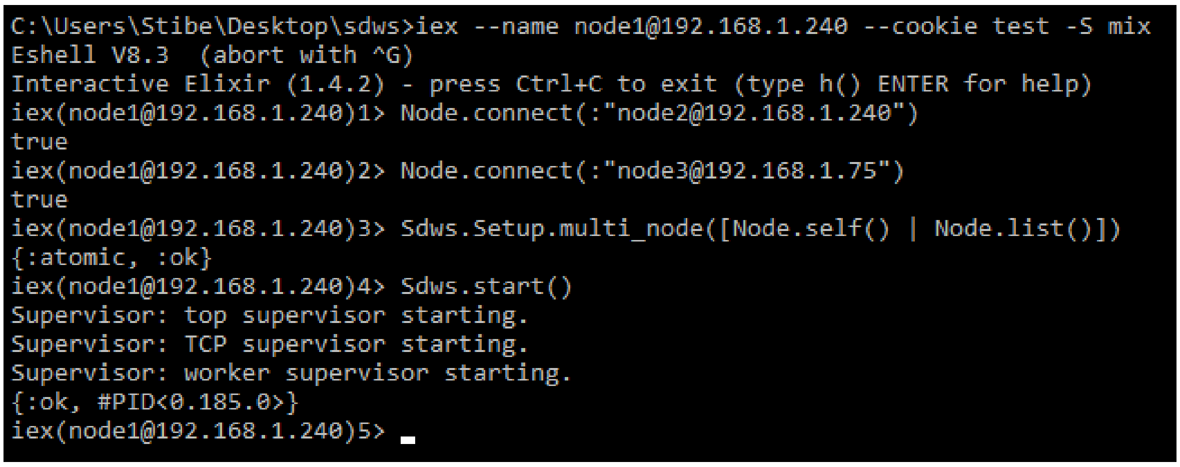
6.6 Distribution

Distribution is achieved by using the BEAM virtual machine instances as nodes and the distributed Mnesia database to replicate data to all nodes. Furthermore, worker processes on all nodes belong to a global, named process group formed by using pg2 module. After a user connects to a node, it is not apparent that which node generates the HTTP reply in the cluster.

6.6.1 Setting up a cluster

Figure 13 depicts how to create a small cluster of three nodes, where one of the nodes is a remote node connected though LAN. On the first line of figure 13, the Elixir interactive shell is started with a name and a security cookie. The name is used to identify this instance of BEAM virtual machine and the cookie is a security agreement, meaning only nodes that have the same cookie may join the cluster. The last option indicates that in this case the program is compiled and shell started using the mix tool. All of the nodes are started in a similar manner, and after starting the virtual machine instances, the nodes are connected together using the connect/1 function.

At this point the cluster is already formed, what remains is to set up the system and run it. Multi_node/1 function initiates the database for multiple nodes, taking a list of the nodes as an argument. After this step, the web server can be started with start/0 or start/1, which takes a port number as an argument. Otherwise the server port is set to 80 by default.

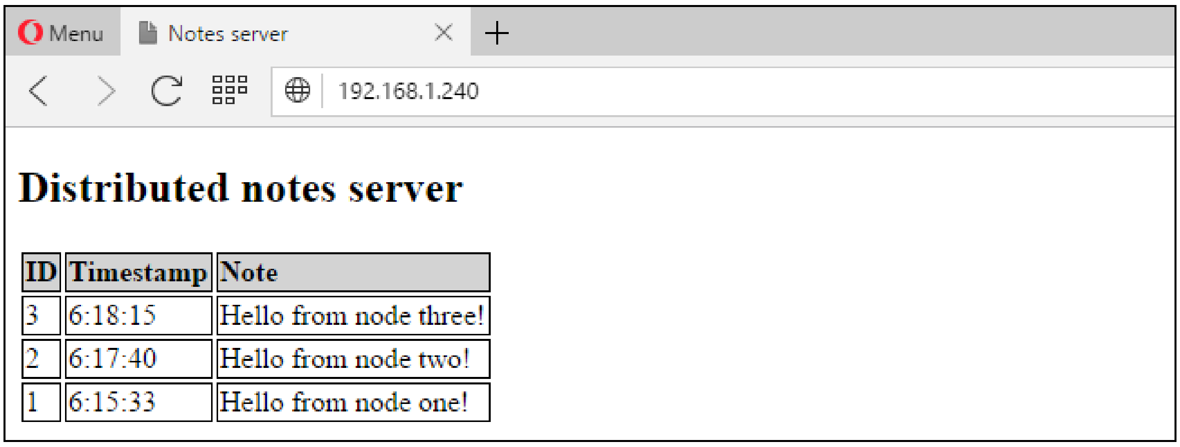


```
C:\Users\Stibe\Desktop\sdws>iex --name node1@192.168.1.240 --cookie test -S mix
Eshell V8.3  (abort with ^G)
Interactive Elixir (1.4.2) - press Ctrl+C to exit (type h() ENTER for help)
iex(node1@192.168.1.240)1> Node.connect(:"node2@192.168.1.240")
true
iex(node1@192.168.1.240)2> Node.connect(:"node3@192.168.1.75")
true
iex(node1@192.168.1.240)3> Sdws.Setup.multi_node([Node.self() | Node.list()])
{:atomic, :ok}
iex(node1@192.168.1.240)4> Sdws.start()
Supervisor: top supervisor starting.
Supervisor: TCP supervisor starting.
Supervisor: worker supervisor starting.
{:ok, #PID<0.185.0>}
iex(node1@192.168.1.240)5> _
```

Figure 13. Setting up a cluster.

6.6.2 Data replication

As mentioned, data replication is achieved with the distributed Mnesia database. The database consists of one table containing the notes data. Any changes made to this table are visible in all of the nodes in a cluster. Both writes and reads to the database are done using atomic transactions. Figure 14 is an example when some data has been added from each node. The same data can be obtained by connecting to any node.



Distributed notes server

ID	Timestamp	Note
3	6:18:15	Hello from node three!
2	6:17:40	Hello from node two!
1	6:15:33	Hello from node one!

Figure 14. Notes data viewed in a web browser.

6.6.3 Process groups version 2

Pg2 module provides a process group, a collection of processes that can be found using a shared name. These processes can exist in multiple nodes and removal of lost members is automatic (Ericsson AB, n.d.).

The pg2 module is utilized to discover worker processes from multiple web server nodes. There are other methods for process discovery, but in this case pg2 is convenient for grouping all the worker processes under a single name. Figure 15 illustrates how the web server uses the process group. Again, the cluster consists of three nodes. Get_members/1 function from pg2 module is used to get a list of the worker processes, and afterwards one worker process can be picked from the list.

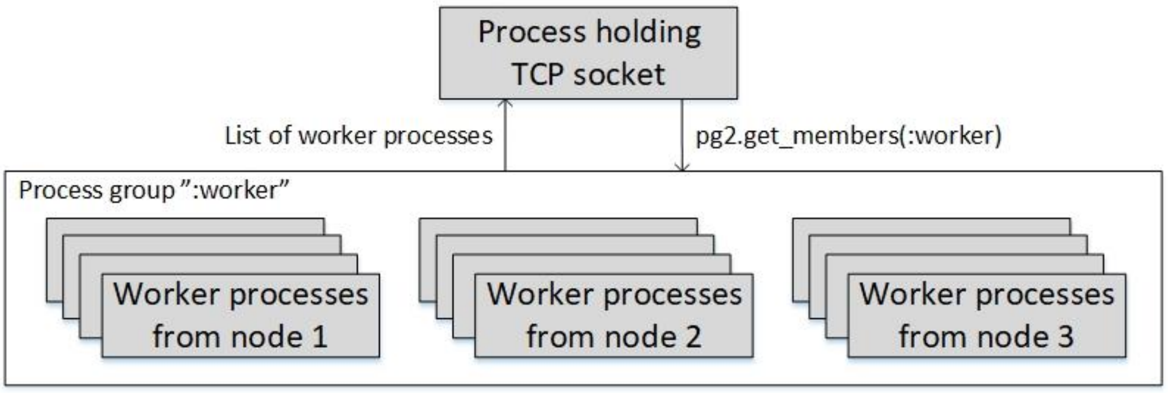
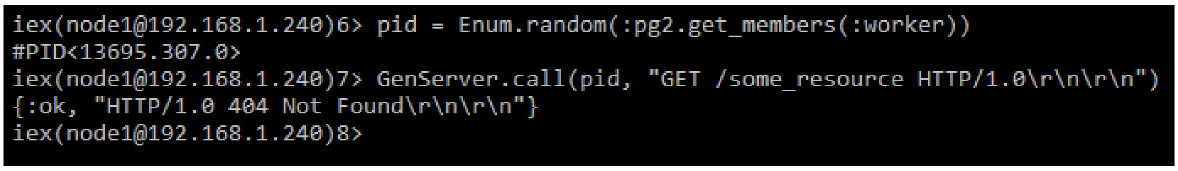

Process holding TCP socket
List of worker processes
pg2.get_members(:worker)
Process group ":worker"
Worker processes from node 1
Worker processes from node 2
Worker processes from node 3

Figure 15. Worker process group.

Figure 16 is an example how the worker process group can be used manually. A single PID is chosen at random from the worker process list. Now that the process PID is known, GenServer behavior can be used to send a message to the chosen process, requesting a fake resource and it gets a 404 reply. The web server works similarly and sends the client requests to random worker processes.

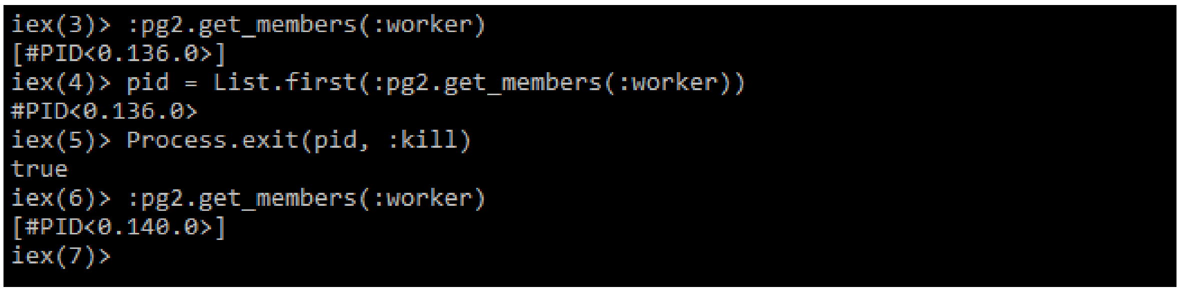


```
iex(node1@192.168.1.240)6> pid = Enum.random(:pg2.get_members(:worker))
#PID<13695.307.0>
iex(node1@192.168.1.240)7> GenServer.call(pid, "GET /some_resource HTTP/1.0\r\n\r\n")
{:ok, "HTTP/1.0 404 Not Found\r\n\r\n"}
iex(node1@192.168.1.240)8>
```

Figure 16. Manually using worker process group.

6.7 Fault tolerance

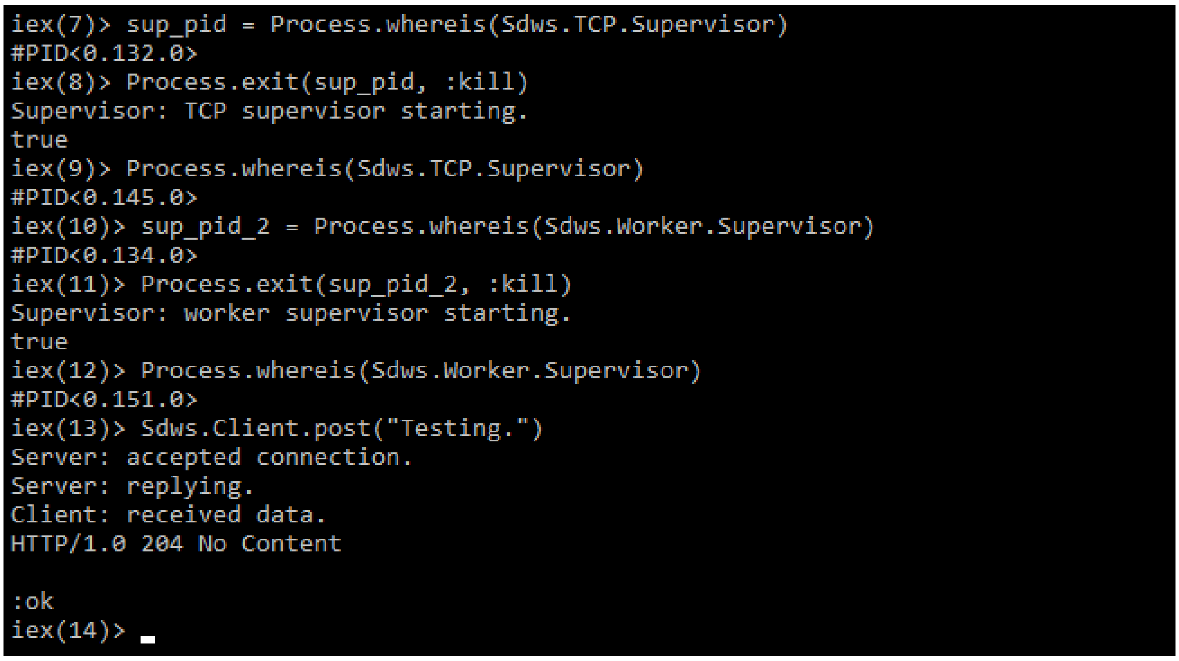
The web server processes are supervised by supervisors, which will attempt to restart processes in case of a failure. The supervisors form a supervisor tree, where a top supervisor supervises both the TCP and worker supervisors, as previously seen in figure 10. The web server is started with only one TCP and worker process to make fault tolerance more evident, since permanently losing even a single process will render the system useless. In Figure 17 get_members/1 function returns the PID of the only worker process in the pool. After terminating the process using an exit signal, a new process is returned by get_members/1. The process was restarted by a supervisor and has a different PID now.



```
iex(3)> :pg2.get_members(:worker)
[#PID<0.136.0>]
iex(4)> pid = List.first(:pg2.get_members(:worker))
#PID<0.136.0>
iex(5)> Process.exit(pid, :kill)
true
iex(6)> :pg2.get_members(:worker)
[#PID<0.140.0>]
iex(7)>
```

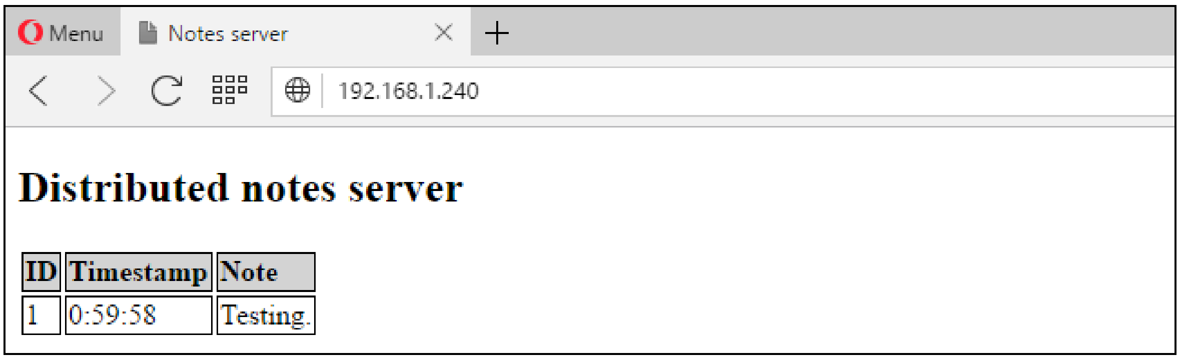
Figure 17. Terminating a worker process.

In figure 18 the supervisors are terminated similarly using an exit signal. The function whereis/1 returns the supervisor PID. After terminating either supervisor, the supervisors are restarted by the top supervisor. Afterwards, the supervisors can still be found using whereis/1, this time with new PIDs. Even after terminating the only worker process and both supervisors the system is still functional. A POST request is made using the client module and figure 19 depicts the result.



```
iex(7)> sup_pid = Process.whereis(Sdws.TCP.Supervisor)
#PID<0.132.0>
iex(8)> Process.exit(sup_pid, :kill)
Supervisor: TCP supervisor starting.
true
iex(9)> Process.whereis(Sdws.TCP.Supervisor)
#PID<0.145.0>
iex(10)> sup_pid_2 = Process.whereis(Sdws.Worker.Supervisor)
#PID<0.134.0>
iex(11)> Process.exit(sup_pid_2, :kill)
Supervisor: worker supervisor starting.
true
iex(12)> Process.whereis(Sdws.Worker.Supervisor)
#PID<0.151.0>
iex(13)> Sdws.Client.post("Testing.")
Server: accepted connection.
Server: replying.
Client: received data.
HTTP/1.0 204 No Content

:ok
iex(14)>
```

Figure 18. Terminating supervisors.



Distributed notes server

ID	Timestamp	Note
1	0:59:58	Testing.

Figure 19. Test request viewed in a web browser.

The system can be stopped by killing the top supervisor. Although, it would be possible to even have an additional supervisor supervising the top supervisor, and so forth.

6.8 Cluster fault tolerance

The implemented web server does not have any specific recovery measures against losing large parts of the cluster. That said, the global worker group made by utilizing pg2 module is able to remove lost worker processes from the group, thus requests will not be sent to processes that do not exist. Losing nodes will not directly affect other Mnesia databases, the data will still be accessible in other nodes. The data is also saved to the working directory, so it will not be completely lost.

7 DISCUSSION AND RESULTS

Erlang’s OTP library offers a vast collection of tools, and it is convenient that the libraries are cross compatible in both Erlang and Elixir. The web server utilized a few Erlang modules, such as the gen_tcp and pg2. To effectively use these modules, some knowledge in Erlang is very beneficial, since the documentation and most examples that can be found are written in Erlang. However, the syntax in Erlang is very different to Elixir and having to learn both is time consuming. Moreover, learning a seemingly unusual functional programming language can be a challenge.

The BEAM virtual machine and the tools provided by Elixir simplified the process of creating a concurrent and distributed web server. Since the web server serves all client requests using individual processes, BEAM is able to schedule the tasks over the available CPUs. The spawned processes did not take up much memory when idle, TCP acceptor processes took only around 3 kilobytes and worker processes 14 kilobytes. Furthermore, the BEAM virtual machine instances were used as nodes and Elixir provided the required tools to form a cluster. To achieve distribution transparency, the distributed Mnesia database and a pg2 module were utilized from the OTP library. The only problem faced during development was having firewalls block the TCP connections needed by the cluster.

Behaviors streamlined the process of adding basic process communication and fault tolerance to the web server. By making use of a supervisor tree, the individual web server nodes were very fault tolerant. The processes only use messages to exchange data. Thus, even a system running on a single computer can be fault tolerant. If distribution is necessary, at least creating small clusters is feasible, similar to what was done with the implemented web server. Furthermore, additional scaling to the system would be possible by adding more processes, CPU cores or nodes.

A fully developed web server would need a few extra considerations and support for newer versions of HTTP. The web server could be improved by having a better method for serving requests. Currently, the received requests are sent to a random node for processing by using a global process group. The web server process pools are also static, the amount of processes is already decided after starting the supervisors. A smarter load balancing strategy would be desirable. Serving requests randomly also affects performance, as some processes may get more work than others. Furthermore, the web server has no implemented protection against connection problems or losing parts of the cluster due to netsplits.

Elixir matches the requirements for a distributed programming language adequately. Projects making best use of the Erlang’s virtual machine, OTP library, Elixir’s features and Elixir’s functional elements should benefit the most from utilizing Elixir. Elixir was well suited for the distributed web server developed in this thesis, and it can be considered as an option for a system with similar requirements. Elixir can be further extended with metaprogramming, if one should find that a feature is missing.

8 SUMMARY

This master’s thesis introduced a different way of programming concurrent systems, utilizing a relatively new functional programming language called Elixir. The goal was to implement a distributed web server using the tools provided by Elixir and Erlang.

The first step was conducting a literature review, getting information on the topics of functional programming, Erlang and Elixir. After the analysis, 21 items from the literature search were chosen as a literature base for this thesis. The functional programming paradigm was introduced. The paradigm has its own advantages and disadvantages, but in general it is suitable for parallel and distributed programming. The BEAM virtual machine and the tools provided by Erlang were discussed, including the open telecom platform (OTP), describing some of the tools and behaviors it can supply. Elixir extends Erlang, adding new features to the language and provides a new syntax.

A distributed web server was implemented utilizing Elixir. The BEAM virtual machine handles parallel execution, since the requests to the web server are executed in individual processes. Distribution is achieved by connecting the virtual machine instances and forming a global process group, thus the web server tasks can be run in any node. In addition, data is replicated to all nodes by using the distributed Mnesia database. The results gained from the implementation are subjective, but it demonstrates how the aforementioned tools can be utilized to create a concurrent and distributed system. Furthermore, the usage of supervisors proved to make nodes very fault tolerant. At the very least, Elixir can be considered as a strong option for similar projects. Learning the language can be challenging but nevertheless worthwhile.

REFERENCES

Armstrong, J. (1997). The development of Erlang. *Proceedings of the second ACM SIGPLAN International Conference on Functional Programming*, 196-203.

Armstrong, J. (2003). Concurrency oriented programming in Erlang. Sweden: Swedish Institute of Computer Science.

Armstrong, J. (2010). Erlang. *Communications of the ACM*, 53(9), 68-75.

Armstrong, J. (2013). Programming Erlang: Software for a concurrent world. USA: The Pragmatic Programmers.

Bal, H., Steiner, J., & Tanenbaum, H. (1989). Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3), 261-322.

Berners-Lee, T., Fielding, R., & Frystyk, H. (1996). Hypertext transfer protocol -- HTTP/1.0. Retrieved from http://www.rfc-editor.org/info/rfc1945

Burton, F. (1986). Functional programming for concurrent and distributed computing. USA, Utah: University of Utah, Department of Computer Science.

Cambridge University Press. (n.d.). Meaning of web server. Retrieved from http://dictionary.cambridge.org/dictionary/english/web-server

Cesarini, F., & Thompson, S. (2009). Erlang programming: A concurrent approach to software development. USA, Sebastopol: O’Reilly Media.

Ericsson AB. (n.d.). Compilation and code loading. Retrieved from http://erlang.org/doc/reference_manual/code_loading.html

Ericsson AB. (n.d.). Gen_tcp. Retrieved from http://erlang.org/doc/man/gen_tcp.html

Ericsson AB. (n.d.). Pg2. Retrieved from http://erlang.org/doc/man/pg2.html

Gat, E. (2000, December 4). Point of view: Lisp as an alternative to Java. *Intelligence,* 11(4), 21-24.

Haenisch, T. (2016). A case study on using functional programming for internet of things applications. *Athens Journal of Technology & Engineering,* 3(1).

Hammond, K. (1994). Parallel functional programming: An introduction. U.K, Glasgow: University of Glasgow, Department of Computer Science.

Hausman, B. (1994). Turbo Erlang: Approaching the speed of C. Sweden: Ellemtel Telecommunications Systems Laboratories.

Hughes, J. (1990). Why functional programming matters. *Research Topics in Functional Programming*, D. Turner, Addison-Wesley, 17-42.

Hunt, J. (2014). A beginner's guide to Scala, object orientation and functional programming. Switzerland: Springer International Publishing.

Iyengar, A., MacNair, E., & Nguyen, T. (1997). An analysis of web server performance. *Global Telecommunications Conference,* 3.

Jurić, S. (2014). Why Elixir. Retrieved from http://theerlangelist.com/article/why_elixir

Jurić, S. (2015). Elixir in action. New York, NY: Manning Publications.

Knutas, A., Hajikhani, A., Salminen, J., Ikonen, J., & Porras, J. (2015). Cloud-based bibliometric analysis service for systematic mapping studies. *Proceedings of the 16th International Conference on Computer Systems and Technologies*, 184-191.

Larson, J. (2009). Erlang for concurrent programming. *Communications of the ACM, 52*(3), 48-56.

Laurent, S., & Eisenberg, D. (2014). Introducing Elixir. USA: O'Reilly Media.

Mattsson, H., Nilsson, H., & Wikström, C. (1999). Mnesia a distributed robust DBMS for telecommunications applications. Sweden, Stockholm: Ericsson Telecom AB, Computer Science Laboratory.

McCord, C. (2015). Metaprogramming Elixir. USA: The Pragmatic Programmers.

Pickering, R. (2007). Foundations of F#. USA: Apress.

Shankar, U. (2013). Distributed programming theory and practice. New York, NY: Springer Science+Business Media.

Steen, M., & Tanenbaum, A. (2016). A brief introduction to distributed systems. *Computing*, 98(10), 967-1009.

Thomas, D. (2016). Programming Elixir. USA: The Pragmatic Programmers.

Trottier-Hebert, F. (n.d.). Distribunomicon. Retrieved from http://learnyousomeerlang.com/distribunomicon

Virding, R., Wikström, C., & Williams, M. (1996). Concurrent programming in Erlang 2nd Ed. J. Armstrong (Ed.). UK, Hertfordshire: Prentice Hall International.

Zhang, J. (2011). Characterizing the scalability of Erlang VM on many-core processor (Master’s thesis). Sweden, Stockholm: KTH Royal Institute of Technology.

APPENDIX 1. Analysis of included literature.

A Beginner’s Guide to Scala, Object Orientation and Functional Programming by Hunt, J. describes the programming language Scala, which is a multi-paradigm language utilizing both object orientation and functional programming. Scala and object oriented programming are not relevant for this thesis. However, the book provides information about the functional paradigm, and has insight on what are the advantages and disadvantages, when utilizing functional programming.

A Case Study on Using Functional Programming for Internet of Things Applications by Haenisch, T. discusses a case study, where the benefits of using functional programming to develop an IoT application was studied. In the study C, Ruby and Elixir were used to build the same system for saving power in paper machines. Code size and complexity are measured in the different systems. The author concludes that Elixir might be good fit for IoT applications.

Characterizing the Scalability of Erlang VM on Many-core Processors by Zhang, J. is a master’s thesis, which discusses the usage of multiple processor cores with Erlang development. The thesis investigates how parallel Erlang VM with 64 cores scales and concludes that Erlang is ready for systems utilizing multiple cores. Furthermore, the thesis provides information on Erlang runtime system and hot code loading.

Concurrency Oriented Programming in Erlang by Armstrong, J. gives insight on the concurrency orientation of Erlang. It has information on process creation, message passing and OTP libraries. In addition to having simple code examples written in Erlang such as sort, factorial and binary tree functions to name a few.

Concurrent Programming in Erlang by Virding, R. et al. describes how to program concurrent programs with Erlang. It includes an introduction to the language in general, and information on concurrent programming and distributed programming with Erlang.

(continues)

Distributed Programming in Erlang by Wiksröm, C. presents Erlang as a way to make the development of big concurrent and distributed systems simpler. Author argues that with very large scale applications it is important to have low level mechanics that have good definitions and are easy to grasp. The paper highlights Erlangs approach regarding process creation, asynchronous messages and process linking. The paper also has information on the performance of Erlang.

Distributed Programming Theory and Practice by Shankar, U. describes the practical and harsh implications of programming distributed systems. The author point out that writing distributed software in a correct manner is difficult since, for example, thread execution speeds differ and may cause race conditions. The book concentrates on writing distributed programs utilizing services with a practical programming notation, which can be applied in many programming languages.

Elixir in Action by Jurić, S. starts with giving general information on Erlang and Elixir. It depicts, how to build scalable, fault tolerant, distributed and available Elixir systems. The book gives advice on how to solve practical problems using Elixir, how to utilize the OTP libraries and managing own projects with the mix tool implemented in Elixir.

Erlang – An Experimental Telephony Programming Language by Armstrong, J and Virding, R. describes an experimental programming language Erlang that is utilized in telecommunication applications. The paper introduces the basic demands for such language including: robustness, real time, distribution, fine grain, functional notation, concurrent and modular. Sequential and concurrent problems are illustrated with examples.

Erlang by Armstrong, J. discusses effectiveness of Elang for fault tolerant, distributed and real-time systems. The author argues that Erlang distinguishes itself from other programming languages by including concurrency to the language itself instead of the used operating system. The papers also promoted the way of letting processes crash and employing other processes to repair the problem. In addition, the paper describes different systems created with Erlang.

(continues)

Erlang for Concurrent Programming by Larson, J. illustrates the concurrency oriented nature of Erlang by providing sequential and concurrent examples. Descriptions of standard behaviors and worker processes are presented. The author states that Erlang is good for developing distributed Internet server applications, graphical user interfaces and batch applications. A list the type of systems that are suitable for Erlang development and the ones which are not suitable is provided in the paper.

Foundations of F# by Pickering, R. describes the functional programming language F#. While F# is out of the scope for this thesis and is not relevant, the book does provide good information on the functional programming paradigm starting from the history of the paradigm. The author argues that the advantages of functional programming can be seen from what it makes possible, rather than concentrate on the language restrictions. Functional programming provides features that are inspired by mathematics, which are not found in imperative languages.

Metaprogramming Elixir by McCord, C. depicts the details on how to utilize metaprogramming in practice with Elixir. This includes creating new features by extending the language via macros and constructing libraries in Elixir.

Mnesia A Distributed Robust DBMS for Telecommunications Applications by Mattsson, H, Nilsson, H, and Wikström, C. discusses the multiuser distributed database management system (DBMS) called Mnesia. Mnesia ships with the Erlang runtime system and can be used also in Elixir. The paper illustrates the requirements that Erlang applications have for DBMS and how Mnesia can answer those requirements. The author states that Mnesia DBMS addresses all or most needs of data management in the field of telecommunication.

Programming Elixir by Thomas, D. The book describes the Elixir programming language and explains how to use it to program concurrent programs. It is for experienced programmers, but guides the reader to start thinking about programming problems in a functional way or more specifically in Elixir.

(continues)

Programming Erlang by Armstrong, J. is about programming with Erlang starting from the basics and including also more advanced topics with many programming examples. The author explains that these days, hardware is getting increasingly parallel and that new programming languages have to include support for concurrency or perish. Furthermore, the author argues that Erlang can save time for developing concurrent, multiuser or systems that change as time passes.

Purity in Elang by Pitidis, M. and Sagonas, K. discussed the purity of Erlang. The author states that purity is important part of referential transparency, which refers to the fact that when language expression is evaluated twice, the end result is the same value. The paper presents a static analysis for finding out purity levels, gives statistics on pure functions and discusses user-defined guards.

The Development of Erlang by Armstrong, J. describes the process of creating the Erlang language. The paper depicts the whole development process for Erlang. Beginning from the early goals and requirements, and moving to the early experiments with the language. The author continues with the creation of the virtual machine, building a user base and adding tools to the language. The author also discusses the future for the language and finishes with reflections of the experience. Furthermore, the paper adds many basic code examples in Erlang also including hot code replacement.

Turbo Erlang: Approaching the Speed of C by Hausman, B. discusses the BEAM virtual machine and how it compiles Erlang into C. The paper depicts the many benefits gained from compiling to C code. The author states that performance is close to very optimized C.

Why Functional Programming Matters by Hughes, J. discusses the importance of functional programming. The papers goal is to illustrate the merits of functional programming to non-functional developers, while assisting functional programmers to gain most from utilizing the paradigm. The paper highlights two important topics that are higher-order functions and lazy evaluation. The author argues that modularity is the solution to successful development.

(continues)

World-Class Product Certification Using Erlang by Wiger, U., Ask, G. and Boortz, K. illustrates how Erlang can be used for test suite development. For creating testing tools, Erlang provides declarative syntax and pattern matching. The author concludes that Erlang is excellent for automating test systems.