Czech University of Life Sciences Prague

Faculty of Economics and Management

Department of Information Technologies

Bachelor's Thesis

Development of Cooperative Applications

Dominic Senarsky

© 2022 CZU Prague

CZECH UNIVERSITY OF LIFE SCIENCES PRAGUE

Faculty of Economics and Management

BACHELOR THESIS ASSIGNMENT

Dominic Senarsky

Informatics

Thesis title

Development of Cooperative Applications

Objectives of thesis

The main objective of the thesis is to develop several versions of backend applications using different approaches for handling multi-client access and evaluate and compare them based on the selected criteria.
Partial objectives are:
- define a multi-user access scenario and the specific needs and requirements it presents
- develop different versions of backend parts of applications using the chosen approaches and frameworks
- measure performance indicators of the created applications and evaluate their efficiency in a given scenario

Methodology

The methodology of the theoretical part is based on analysis of available scientific information sources. In the practical part, several versions of backend applications will be developed using chosen programming approaches. The performance of these applications in a given scenario will be measured and their overall efficiency will be compared using multiple criteria decision analysis. Based on the synthesis of knowledge obtained in the theoretical part and the results of the practical part, conclusions will be formulated.

The proposed extent of the thesis

40-50

Keywords

cooperative applications, backend, html 5, programming frameworks, performance testing

Recommended information sources

Balbaert, I., St. laurent, S. (2019). Programming Crystal. USA: The Pragmatic Programmers, LLC.
Elman, J., Lavin, M. (2014). Lightweight Django: Using REST, Web Sockets, and Backbone. California, USA: O'Reilly Media, Inc.
Harron, D. (2018). Node.js Web Development – Fourth Edition. Birmingham, UK: Packt Publishing Ltd.
Juric, S. (2018). Elixir in Action, Second Edition. New York, USA: Manning Publications Co.

Expected date of thesis defence

2021/22 SS – FEM

The Bachelor Thesis Supervisor

Ing. Jan Pavlík

Supervising department

Department of Information Technologies

Electronic approval: 17. 8. 2021

doc. Ing. Jiří Vaněk, Ph.D.

Head of department

Electronic approval: 5. 10. 2021

Ing. Martin Pelikán, Ph.D.

Dean

Prague on 10. 03. 2022

Declaration

I declare that I have worked on my bachelor thesis titled “Development of Cooperative Applications" by myself and I have used only the sources mentioned at the end of the thesis. As the author of the bachelor thesis, I declare that the thesis does not break any copyrights.

In Prague on 15.03.2022 ____________________________

Acknowledgement

I would like to thank my supervisor Ing. Jan Pavlík for his invaluable advice and feedback throughout the duration of the project. His unwavering support and consistent proactivity were crucial in overcoming any difficulties experienced during my research.

I would like to thank Tamás Kemény for his feedback and discussions regarding the research material and results.

I would like to give a special thanks to Nicholas Rakita for his support and feedback throughout the duration of the project. Nicholas was a key motivator and supporter of the research that I was aiming to understand and quantify in this thesis. he provided invaluable advice on how to conduct formal research and be eloquent with my thoughts in a programmatic environment.

Finally, I would like to thank my family back home and my loving partner for giving me the drive to publish my best work in an academic setting – and for the unending emotional support during difficult times.

Development of Cooperative Applications

Abstract

The purpose of this thesis is to identify and analyze various popular languages and frameworks which utilize the WebSocket API. It is necessary to measure both quantitative and qualitative sets of data and then compare them both subjectively and – when applicable – objectively. Python, Elixir, Crystal, and JavaScript are the programming languages which are used to test performance and gather analytical data for conducting research. By utilizing these distinct languages, valuable information can arise regarding WebSocket client-server communication.

The gathered data from each language's performance are analyzed using multi-criteria decision analysis – quantitative and qualitative performance matrices compact the data and provide a clear ranking of the optimal criteria based on weighted results. All the languages and frameworks utilized in this thesis provide advantages and disadvantages depending on the application criteria, and the decision for which to use in specific scenarios is emphasized accordingly.

Keywords: cooperative applications, backend, frontend, programming frameworks, WebSocket, performance testing

Vývoj Kooperativních Aplikací

Abstrakt

Cílem této bakalářské práce je identifikovat a analyzovat programovací jazyky a frameworky, které využívají rozhraní WebSocket API. Je nutné zohlednit jak kvantitativní, tak kvalitativní ukazele a následně je co nejvíce objektivně porovnat. Pro účely sběru analytických dat je v této práci využito programovacích jazyků Python, Elixir, Crystal a JavaScript. Využitím těchto odlišných jazyků lze získat cenné informace týkající se klient-server komunikace přes WebSocket.

Shromážděná data o výkonu každého zvoleného jazyka jsou analyzována pomocí multikriteriální rozhodovací analýzy – kvantitativní a kvalitativní výkonnostní matice koncentrují data a poskytují jasné pořadí variant. Zvolené jazyky a frameworky použité v této práci poskytují různé výhody či nevýhody a v závislosti na volbě vah kritériích, která zohledňuje zamýšlené použití aplikace, je poté možné učinit rozhodnutí, které bude pro daný scénář optimální.

Klíčová slova: kooperativní aplikace, backend, frontend, programovací frameworky, WebSocket, testování výkonu

Table of Contents

1 Introduction **10**

2 Objectives and Methodology **11**

2.1 Objectives 11

2.2 Methodology 11

3 Literature Review **12**

3.1 Web Applications Overview 12

3.2 WebSocket 13

3.3 WebSocket Use-Case 14

3.4 Programming Languages 15

3.4.1 JavaScript 16

3.4.2 Elixir 18

3.4.3 Python 21

3.4.4 Crystal 24

3.5 Multi-Criteria Decision Analysis 28

3.5.1 Selection of Criteria 29

3.5.2 Measured Criteria Options 30

3.5.3 Calculating Results and Sensitivity Analysis 30

4 Practical Section **32**

4.1 Practical Overview 32

4.2 The Development/Testing Environment 32

4.3 Web Application Benchmarking Software 33

4.4 The WebSocket Backend Servers 34

4.4.1 JavaScript 35

4.4.2 Elixir 36

4.4.3 Python 38

4.4.4 Crystal 39

4.5 Server/Client Execution Master and Subscripts 39

4.6 Qualitative MCDA 40

4.7 Qualitative Decisions for Weights 43

4.8 Quantitative MCDA 44

4.9 Quantitative Decisions for Weights 47

4.10 MCDA Sensitivity Analysis 49

5 Results and Discussion **51**

6 Conclusion **54**

7 References **55**

List of images

Image 1 - WebSocket browser compatibility 13
Image 2 - WebSocket connection client 14
Image 3 - Example of a single threaded program 22
Image 4 - Example of a multithreaded program 22
Image 5 - Implementation of a simple interpreter 26
Image 6 - MCDA detailed steps 29
Image 7 - k6 WebSocket client program 34
Image 8 - JavaScript WebSocket server 35
Image 9 - handler.ex program file – part of the WebSocket server 36
Image 10 - application.ex program file – part of the WebSocket server 37
Image 11 - Python WebSocket server 38
Image 12 - Crystal WebSocket server 39
Image 13 - Subscript for server init/kill (Crystal 8-core section) 40
Image 14 - Min-max normalization program file 47

List of tables

Table 1 - Qualitative matrix, approximate rankings 42
Table 2 - Qualitative numeric matrix 42
Table 3 - Quantitative performance matrix 46
Table 4 - Sensitivity analysis matrix 50

1 Introduction

The purpose of this thesis is to identify and analyze various popular languages and frameworks which utilize the WebSocket API. It is necessary to measure both quantitative and qualitative sets of data and then compare them both subjectively and – when applicable – objectively. Efficient communication on the Web is as common as ever, and with such a wide variety of options to choose from in a software development situation, there is much to consider when choosing a software stack. As this technology booms and WebSocket development becomes more relevant for global platforms, developers are left questioning what exactly to use in their various working environments regarding this technology. The demand for web applications is at an all-time high, whether it be accessed on laptop/desktop machines or on mobile devices.

There is a need for these Web applications to handle large amounts of concurrent users and manage the various events on each instance of the client's application in conjunction with the Web servers. Depending on the number of concurrent users and computations that must be performed for each user, the developer must consider that any chosen language and frameworks used for the development of a Web application adhere well to the necessary performance criteria. Different computational requirements are present depending on the potential userbase of the application as well as the number of processes that can occur for each user. Different languages and frameworks will all perform better or worse depending on the need for intense computation and user concurrency.

There is a lack of significant data regarding the performance of different programming languages compared to each other for the varying aspects of WebSocket functionality. The research conducted in this thesis should provide clarification of the criteria for a Web application that can be optimally handled by each of the implemented languages. These results are important as they provide a significant comparison for WebSocket client-server communication performance among four unique languages.

2 Objectives and Methodology

2.1 Objectives

The main objective of the thesis is to develop several versions of backend applications using different approaches for handling multi-client access and evaluate and compare them based on the selected criteria. Partial objectives are:

- define a multi-user access scenario and the specific needs and requirements it presents
- develop different versions of backend parts of applications using the chosen approaches and frameworks
- measure performance indicators of the created applications and evaluate their efficiency in a given scenario

2.2 Methodology

The methodology of the theoretical part is based on analysis of available scientific information sources. In the practical part, several versions of backend applications will be developed using chosen programming approaches. The performance of these applications in a given scenario will be measured, and their overall efficiency will be compared using multiple criteria decision analysis. Based on the synthesis of knowledge obtained in the theoretical part and the results of the practical part, conclusions will be formulated.

3 Literature Review

3.1 Web Applications Overview

The internet as we know it is composed of a seemingly endless amount of web applications developed for an incomprehensibly large amount of use cases. People use applications that communicate over web protocols on their phones, computers, tablets, TVs, and any type of smart technology devices. The industry of web application server and client development is constantly growing and shaping how people live their daily lives. Web applications allow us to simplify how we communicate, travel, make material purchases, learn, enjoy recreational activities, and much more. By updating interfaces of web applications centrally, users can always have up-to-date content on their personal or enterprise devices. Depending on the type of web communication protocol used for the developed applications, different features can be optimized for the benefit of the user base. The web communications protocol that stands out in the modern era of web application development is WebSocket. As WebSocket offers persistent bidirectional communication between a server and client, it can manage and process requests and data extremely quickly. This makes it an attractive technology to work with in web application development environments, as it is the optimal choice for client applications that require constant updates about state from a server. WebSocket is a ubiquitous contemporary solution for implementing duplex connections in modern web applications. Therefore, this thesis attempts to analyze several languages/frameworks allowing the use of WebSocket connections regarding their usability and performance.

3.2 WebSocket

The WebSocket computer communications protocol is the foundation of the technology that various programming languages and frameworks utilize, modify, and optimize for faster client-to-server communication on the Web. It is important to note that the WebSocket protocol is an IETF (Internet Engineering Task Force) standard, and the vast majority of modern browsers support the use of the WebSocket API initialization with some slight versioning and platform-dependent exclusions, as shown in the image below.

	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	WebView Android	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet
`WebSocket()` constructor	Yes	12	7	10	Yes	Yes	Yes	Yes	7	?	Yes	Yes

Image 1 - WebSocket browser compatibility, source: [1]

A WebSocket functions as an online event handler that manages a bi-directional persistent and interactive TCP connection between a server and the connected clients (most commonly accessed via a user's web browser) [2, p. 1]. This critical function of WebSocket is called a full duplex system, which allows for any nodes on end devices and software to receive and send data simultaneously [3]. Responses to data sent between the server and clients are managed through event-driven interactions, which, unlike HTTP, do not require constant polling of the server to instantiate and close connections when data is retrieved. This allows for the management of data in real-time and promotes the usage of various event-handling functions to sort data, send it where it needs to go, and in the format they need to be in – whilst also simultaneously receiving said data and managing it to be sent again [3]. Mozilla's MDN Web Docs provide a simple overview of a WebSocket connection client, as demonstrated below.

```
//Create WebSocket connection.

const socket = new WebSocket('ws://localhost:8080')


//Connection opened
socket.addEventListener('open', function (event) {
socket.send('Hello Server!');
});


//Listen for messages
socket.addEventListener('message', function (event) {
console.log('Message from server', event.data);
});
```

Image 2 - WebSocket connection client, source: [4]

The code snippet above covers what is fundamentally needed for creating/opening a WebSocket connection and listening for messages from a WebSocket server.

3.3 WebSocket Use-Case

The WebSocket communications API is compatible with common platforms such as iOS, Android, Web applications on-line, and locally executed desktop applications. Unlike http where the connections are prefaced with http:// or https://, WebSocket is written as ws:// or wss:// [2, p. 14]. This technology is commonly used in the following example scenarios:

- Real-time social media feed applications which constantly update
- Multiplayer video games that require fast visual feedback and exceptional performance
- Chat applications that require real-time message updates without loss of data or time
- Cryptocurrency price tracking and visual value updates in real time
- Geolocation live tracking
- Document and workspace live multi-client editing

There are many different multi-client situations and variables to consider when developing software whilst integrating the WebSocket API. Situations arise where an application will have a natural or strictly bounded maximum of *x* users using an application

concurrently, when in others there can be a theoretically unbounded number of users accessing an application concurrently.

The frameworks and libraries for languages analyzed in this paper are all optimized/designed for WebSocket communication, but likely vary greatly regarding their individual performance. The Node.js implementation for JavaScript is used along with the ws library, RiverSide for Elixir, the 'websockets' library for Python, and Kemal for Crystal. The natural burdens or complete lack of WebSocket implementations in the vanilla languages are generally ironed out by the listed frameworks. Every programming language is designed with core concepts in mind that the languages should cater to. For example, Elixir (a web-oriented language utilizing the Erlang VM) is designed with the capacity for reliable and scalable multi-client communication. However, a language such as JavaScript was designed for interactive website development, generally lacking in comparison when it comes to multi-client server development. For this reason, Node.js was created and now is used very commonly for developing multi-client applications in which vanilla JavaScript suffers so heavily.

3.4 Programming Languages

The following sections focus on the general function and application of selected programming languages – with a slight inclination towards features and examples of benefits for the development of WebSocket multiclient web applications. Key positive and negative aspects of the languages will be highlighted to clarify how a multi-client web application would benefit from the technology. In addition, external frameworks that are optimized for WebSocket implementation within native languages will be discussed and analyzed for their all-round effectiveness. Each language is considerably different and contain differences in their foundational programming paradigm, internal data structures, syntax, type system, and implementation (system for executing programs on a computer [interpretation vs compilation]).

It is important to mention that the concurrency and capacity for parallel processing capabilities of all languages play a large role in their performance for web socket communication, as having more processes executing in parallel allows for faster input connection filtering and return responses from the server.

In the practical part of this assignment, the literary review of each language will play a role in the analysis of language syntax, semantics, and the value provided by their

respective communities. Although the previously listed points cover the majority of quantifiable and qualitatively measured aspects of a language, it should be noted that they do not cover absolutely *everything*. Instead, the focus of the thesis work is placed on concepts that can be measured at an intermediate-advanced level in an MCDA evaluation.

3.4.1 JavaScript

JavaScript is a widely used scripting language that is included in every modern web browser and powers the interactive aspect of almost all web pages and web applications. Initially released in 1995, it was created with the intent to enable complex frontend scripting in the Netscape Navigator Browser [5, p. 5]. Applications written in this language span from desktop computers to phones and tablets, in recreational and professional environments alike. It is also commonly used for programming web servers, interactive controllers, video games, and a wide range of other applications. JavaScript works together with HTML and CSS in a three-layer system for developing websites and interfaces [6]. Nowadays with Node.js, JavaScript is used beyond frontend work as it supports both frontend and backend development. Code can be reused between the frontend and backend and being able to use JavaScript effectively for the full stack is a huge benefit [7, p. 2]. With the ability to implement applications with the same language on the client and server side, developers can work on both ends and migrate code between both structures very easily [7, p. 12]. In the practical section of this thesis, the ability for an entirely JS backend/frontend system is demonstrated by the fact that there is a Node.js WebSocket server which is communicating with virtual clients generated by a JavaScript client implementation.

The runtime environment that will be used and analyzed for JavaScript is Node.JS. Node.js was initially developed to simplify asynchronous input/output communication (allowing for input/output processes) communication within the JavaScript language [5, p. 354]. This is highly necessary for JavaScript, as the ECMAScript standard does not include I/O handling functionality [8]. As it is designed as a scripting language, the host environment should tell the language what to do and should have full control of specifications and the processes to run. Just as a web browser is a host environment for JavaScript, so is Node.js – as it is a server-side host environment designed for primarily network I/O operations [8]. It is necessary to evaluate the value that Node.js brings/adds to JavaScript WebSocket programming interfaces.

JavaScript has a different execution method as opposed to the other languages tested in the practical part of this thesis – the Just-In-Time (JIT) compiler of the run-time interpreter. The JavaScript JIT compiler does not compile a program in one go and then execute it, but instead will compile and recompile code as expressed and desired from the written code. This JIT compiler within JavaScript contains a profiler, which watches code that is being executed, and stores what object types are used and how many times various parts of the code are run. The profiler will consider parts of code 'warm' or 'hot code', if it is executed multiple times, then it is sent to be compiled and stored for further repeated use [5, p. 392]. The compiler is optimizing code snippets to increase performance as the program is executed.

JavaScript is dynamically typed, which means it assigns types to variables at runtime – only values have types, but bindings can hold values of various types [5, p. 403]. The compiler does not know anything about the specific property type that the source code is attempting to access and needs to be ready to handle code that will be able to manage any types. This process should be assisted by the user offering consistent types to run the code faster [5, p. 404]. JavaScript is also weakly typed, meaning that the interpreter or compiler can operate on data that are not given an explicit type, and will decide whether to do implicit type conversions [5, p. 19]. This is a big annoyance for many developers as it can be difficult to predict how the compiler will handle data. Accordingly, the ecosystem has sprouted many languages that transpile into JavaScript and can elegantly handle such worrying features.

Regarding the functionality and desire of JavaScript as a language, some developers complain about its design. Since the years of its release, the language has been updated in a sort of bolted-on fashion – with changes and fixes being amended to match the quickly growing needs of developers of web browsers and faster systems. There are various solutions offered by languages that transpile into JavaScript. TypeScript is used to solve the issues brought by dynamic typing [9]. It allows developers to specify types and avoid errors that present themselves during runtime for vanilla JavaScript [10]. CoffeeScript is a lightweight language that provides a wide range of syntactic sugars for JavaScript, improving the ease of use of JavaScript from the top-down syntax point of view for developers [11].

The Node.js implementation of JavaScript is used by many large corporations for building their web applications. When Netflix switched from its Java backend to Node.js in 2015, the business saw a 70% lower startup time for the web interface. In addition, since the backend is in JavaScript for Node.js, the transition for developers to create the frontend was

greatly simplified [12]. Another corporation that is using Node.js at a grand scale is NASA. The business saw a 300% improvement in the access time to a database, granting users the prompted data magnitudes faster than before. The architecture of Node.js also allowed for NASA to migrate old databases to the cloud and allow users to access them via APIs [12]. PayPal also moved from Java to Node.js as did Netflix and was handling twice the number of requests in comparison to the old infrastructure [12].

JavaScript ranks as one of the industry leaders in community support and has plenty of resources from various sources on topics one might need to search online. Although the solutions for certain issues or questions can be vastly different depending on the developers' responses, almost every solution to a query can be found quickly by searching online. The 'ws' library is used for JavaScript with Node.js to efficiently implement a WebSocket interface. It is one of the most popular WebSocket libraries for Node.js and provides a seamless implementation for developers to work with [13].

The capabilities of the Node.js provide immense potential for developers to make attractive and fast web applications. The vanilla implementation does not offer as much use for web application development in terms of efficient communication between the server and client but is still a decent choice for different levels of developers due to its community and available resources.

3.4.2 Elixir

Elixir is a fascinating language in many technical and aesthetic aspects; it was created by José Valim in 2011 with the goal of modernizing and improving the Erlang programming language [14]. Erlang is a highly scalable functional language designed in the 1980's primarily by Joe Armstrong along with his colleagues Robert Virding, and Mike Williams [15]. The initial design was built entirely around the concept of handling large-scale phone switching [14]. This led to the language being principally used for high-uptime and bandwidth systems such as those found in banking institutions, telecom industries, or instant messaging applications [16]. It excels at being used to build fault-tolerant applications and, due to its origins in the telecom industry, managed to support hundreds of thousands of users in a massive telephone exchange at the founding company, Ericsson [17]. Erlang, despite its unique and highly advantageous design principles, remains a rather niche language within the mainstream web development community. However, the Erlang community is still growing alongside Elixir's.

The Erlang VM (known as BEAM) is used for executing bytecode (.beam) – which is generated by the Erlang Run-Time System (ERTS) – and scheduling Erlang processes to be executed on the CPU. It creates a process scheduler on each core of a CPU, allowing processes to achieve high levels of concurrency and run at the same time, in parallel [18]. These processes are fully isolated from each other and do not share memory, in this case allowing a process to crash and not having it affect the rest of the system. The system then initiates a new process to replace the one that failed previously [19, p. 214]. The language is highly scalable due to the lack of any process locks such as the GIL covered in the Python section below. This hindrance in the Python language is unknown to Erlang, which allows processes to communicate asynchronously as separate concurrent groups of processes [19, p. 4]. As Erlang was not designed for multi-core computers and could not properly perform true symmetric multiprocessing until version R13b in 2019 – it is common to hear about concurrency in the same technical scope as parallelism [18]. All these innate benefits allow Erlang to excel in server-side systems. Erlang-built systems are scalable, responsive, fault tolerant, concurrent, and distributed [19, pp. 5-6].

The following section will provide information on Elixir itself and how it builds off Erlang. In the previous paragraph, Elixir was mentioned as the wonderchild of Erlang. It provides all the amazing features of Erlang in a much more user-friendly and modern programmatic style. It is open source and has more than 700 active contributors to the project [19, p. 8]. Elixir borrows syntax from Ruby and is built to improve semantics and general readability – a notable annoyance for many developers using Erlang. Saša Jurić states in his book: “Personally, I find it much more pleasant to code in Elixir. The resulting code seems simpler, more readable, and less burdened with boilerplate, noise, and duplication. At the same time, you retain the complete runtime characteristics of pure Erlang code” [19, p. 13].

The creator of Elixir was a Ruby developer himself, and thus structured the Elixir language to be aesthetically pleasing to programmers, sharing similar syntactical constructs to Ruby. This borrowing of syntax from Ruby is seen commonly as well in the Crystal language, providing insight on how both creators of these languages placed syntactic sugar high on their priority list [20]. This is a big step forward in comparison to the dated and non-standard syntax of Erlang, which was originally started as a modified Prolog, with the it being highly syntactically resemblant [21]. Despite the friendlier syntax, Elixir has the same powerful features that functional languages have, as it still utilizes the Erlang VM.

Functional programming languages offer advantages such as those listed below by TutorialsPoint:

- State Free Programming − Functional programming does not support state, so there are no side-effect results, and we can write error-free code.
- Efficient Parallel Programming − Functional programming languages have NO Mutable state, so there are no state-change issues. One can program "Functions" to work parallel as "instructions". Such codes support easy reusability and testability.
- Efficiency − Functional programs consist of independent units that can run concurrently, and restart or trigger processes when other ones fail. As a result, such programs are more efficient.
- [Supports Higher-Order Functions − Functional programming supports higher-order functions.] [22]

The efficient parallel processing capabilities make Elixir stand out in comparison to the other languages in this thesis, as none provide such seamless support for parallel processing.

The community for Elixir is not as large as JavaScript and Python but does have a slightly more active community in comparison to the Crystal language. The Elixir language has official documentation that is well explained and detailed for any necessary usage of the core language, and like Crystal, will offer user support when inquired. Package/dependency management in Elixir is an improvement in terms of simplicity to that of Erlang. Most of the dependency management is integrated through the Mix dependency manager. Mix handles not only dependencies for an application, but also the creation, compilation, and testing process of the said application [23]. Hex is the package manager for Elixir that can be used to utilize and build Hex packages. Hex integrates with Mix's dependency handling to work side by side for package and dependency management [24]. By having all the Elixir application management covered by these tools, the process in the practical part for working with Elixir libraries was very straightforward. The framework used for Elixir in this thesis is RiverSide, which is a WebSocket server framework for Elixir. It consists of many libraries available for Elixir which work in conjunction to allow for elegant WebSocket server functionality [25]. Other frameworks are far too bloated or non-trivial to use for this thesis, in comparison to RiverSide.

Elixir is used in large corporations for handling web transactions and client-server communication. As an example of usage in a large corporation: "Bet365 switched from Java to Erlang and Elixir, the betting site handles over 6 million HTTP requests and 500,000 database transactions per second" [26]. In addition, Discord has been using Elixir since it was created as stated in the Discord blog: "From the beginning, Discord has been an early adopter of Elixir. The Erlang VM was the perfect candidate for the highly concurrent, real-time system we were aiming to build. We developed the original prototype of Discord in Elixir; that became the foundation of our infrastructure today. Elixir's promise was simple: access the power of the Erlang VM through a much more modern and user-friendly language and toolset" [27]. Due to its prominent features as a modern functional programming language, Elixir can be used comfortably and reliably for web application server-side development.

3.4.3 Python

Python has been one of the most popular programming languages since its release in 1991 by Guido Van Rossum [28]. This open-source language is growing steadily in popularity, especially among beginner developers. Due to its general syntactic simplicity and shallow learning curve, it is used globally by beginners and professionals alike, also adding to its growing adoption rates. It's applicability spans across many different types of systems and applications from web development all the way to machine learning applications, or even intensive data science research.

There is a large range of methods and modules that are built into the Python language standard library, allowing for many basic functionalities to be available for developers. This is a key feature of the language that generates its strong appeal, offering consistency and ease-of-use to programmers of all levels. The documentation for the language is abundant, and its community is one of the largest in the development world. The PEP guidelines implemented as a form of standardizing the Python language make finding solutions to issues much less confusing and scattered in comparison to a language such as JavaScript [29].

Compared to the other languages researched and discussed in this paper, Python (CPython) contains a Global Interpreter Lock (GIL) that allows only one thread to process execution instructions during runtime. This GIL significantly improves single-thread performance by reducing the number of thread locks necessary during program execution.

A thread lock manages and limits the assignment of resources to a single CPU thread so that other threads cannot pull from the same pool of operating system resources. The GIL introduced in Python (only in the CPython interpreter/compiler, the default implementation of the Python programming language) prevents programs from running on threads in true parallel, but only concurrently. If a user targets two different threads with python code, the GIL will prevent these two threads from running in parallel and will execute using threads concurrently [30]. Typically, asyncio is used to run Python coroutines concurrently and decide when they start and end execution [31]. It can be understood by recalling that multithreading involves concurrency, while multiprocessing is true parallelism. Below are simple graphic displays of a program being executed using single threading vs. multithreading:

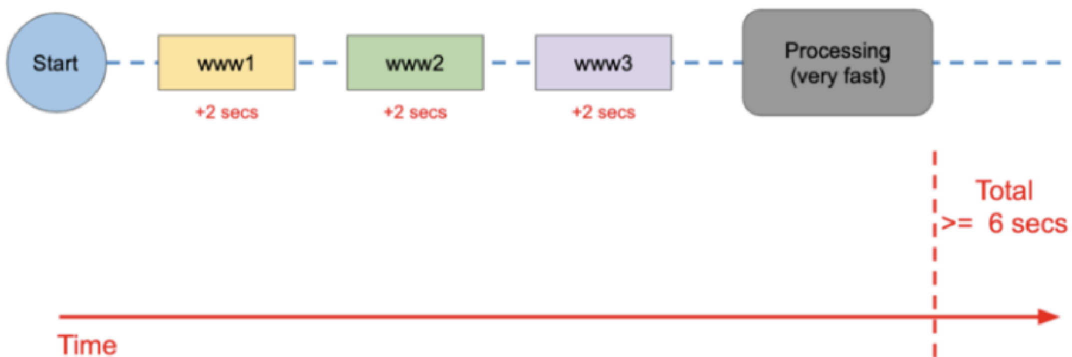

Image 3 - Example of a single threaded program, source: [30]

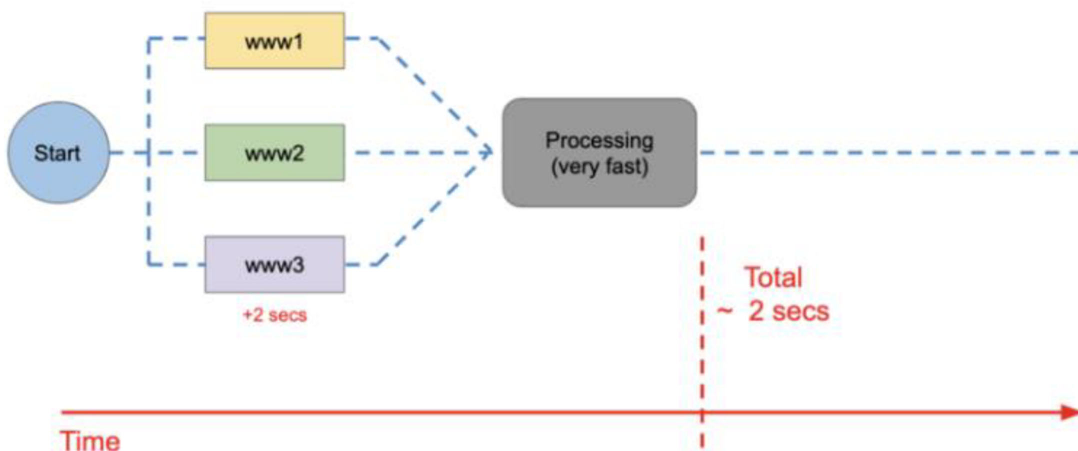

Image 4 - Example of a multithreaded program, source: [30]

It is apparent why the GIL in Python and languages that are strictly limited to single-thread execution have a strong effect on large and scalable applications implemented through WebSocket. If there is heavy real-time resource usage and requests sent to be processed by the backend, then single threaded execution won't be fast enough to manage all the requests.

The CPython interpreter includes its own compiler and virtual machine, which itself interprets the bytecode of the source code once compiled. The first step in the execution of a Python program takes part with the Python compiler, which compiles each statement in the source code into .pyc byte code groups after parsing through and checking for syntax errors (this helps with the speed at which files are loaded, once compiled). The Python virtual machine (PVM) takes the byte code as input, as well as library modules, and converts the byte code groups into machine code so that the processor can execute it through the PVM [32]. By using an implementation such as PyPy, developers can program in Python and execute using a JIT compiler. PyPy generally runs faster than CPython due to the JIT compiler and is preferred for longer run time programs with many types. PyPy has limited support for C extensions and runs them at slower speeds [33].

Python uses strong and dynamic typing. An object type cannot change without explicit conversion, and object types can be changed freely during runtime [34, p. 344]. The dynamic nature of Python allows for better reuse of code and easier integrations of interfaces without facing issues early on in implementations [34, p. 344]. Python's syntax is very comfortable to use for many developers, as it looks like pseudocode [34, p. 441]. The library used for the development of WebSocket servers in Python is called 'websockets'. It is simply installed with pip and allows for WebSocket server setup in cooperation with asyncio. The websockets library is very simply to use and required minimal troubleshooting [35].

Python is not primarily used by companies to directly manage the handshakes web application servers make, but to handle automation, scripts, machine learning, numerical operations on data, and a wide range of additional tasks on the server side. For example, Google has been using Python since its early days. The director of research at Google, Peter Norvig, stated, "Python has been an important part of Google since the beginning and remains so as the system grows and evolves. Today numerous Google engineers utilize Python, and we're searching for more people having the skills of this language." The

company uses it due to its easy maintenance, simplicity, AI/ML capabilities, and capacity for functionality in robotics projects [36]. Netflix also implements Python in their large streaming service application ecosystem. The company has dynamic software written in Python to strengthen the security of their infrastructure, examine and analyze data reports, and listen for various alerts [36]. In addition to this, Netflix also uses Python for machine learning to algorithmically sort and distribute films and shows to users, enhance and evolve the streaming service, and pull images from videos to create the thumbnails seen for all video content on the site [36]. Just as with Google, this goes to show that Python is highly utilized for company security maintenance and operation, number crunching, data analysis, and for creating and maintaining powerful machine learning models.

3.4.4 Crystal

Crystal is a relatively new language, with version 0.1.0 officially released in June 2014 [37]. The initial start of the project began in 2011 with Ary Borenszweig [38, p. 1]. The core team consists of 8 developers and more than 450 contributors to the language [39]. Crystal was designed to function as a high-performance version of Ruby. "Crystal brings much greater performance in places where Ruby is in need of it" [38, p. 5]. This is in part due to Crystal being statically typed and compiled, not interpreted like Ruby.

Crystal does support concurrency, but not in the standard format through threads and thread locks and schedulers [40]. A fiber starts with a stack size of 4kb and can grow to a maximum of 8MB, which is the standard memory usage for a thread. On a 64-bit machine, millions of fibers can be spawned [38, p. 160]. As standard threads are preemptive (OS can interrupt a thread and the scheduler will switch to another as needed without direct access from the active thread), fibers are cooperative and can manually yield control and allow various other threads to begin processing before these finish [41]. This is helpful as it reduces the thread switching overhead. The runtime schedule within Crystal already has a group of fibers ready for execution, an event loop fiber that checks for async communications to be executed, and then fibers that are waiting to execute with the yield command. This provides room for other fibers to execute if necessary while the aforementioned fibers yield [41]. As of September 2019, Crystal implemented parallelism and fibers are able to be instantiated on each core's threads and share memory, only requiring developers to synchronize state among the fibers. This feature is still heavily in development and is not yet fully stable [42].

The capacity for multiprocessing is of great use for the practical part of this thesis, especially for WebSocket communication I/O handling.

Crystal is a compiled language; the Crystal compiler is self-hosted, meaning that it is written natively in the Crystal language itself. This allows for much easier understanding of how it works behind the scenes, providing for quick debugging when necessary [38, pg. 4]. Crystal is statically typed and has type inference. This means that the compiler can infer the type of a variable if not explicitly given, variables must always have inferred or explicitly given types before run-time [43]. Crystal programs are compiled into native executable binary code and executed immediately after this process is finished [38, p. 6]. Code is compiled using the LLVM toolchain and is not interpreted or compiled through an abstract machine such as a VM, compared to other languages [38 p. 7]. The smart compiler has many built in functions to catch errors before the testing or production phase of a program. This includes static type-checking, preventing nil pointer exceptions, and other predefined input tests. For example, empty arrays will cause the compiler to return an error as it cannot assume the type, and this needs to be strictly defined beforehand, unlike in Ruby. However, there are many situations in which types are inferred for objects and don't need to be explicitly set before runtime. The compiler prevents objects that would result in user error when developing and not providing types, allowing for more efficient and robust code [38, p. 24]. Crystal union types are used very commonly and allow instances of values to hold a set number of types. This is important in Crystal for simplified exception handling and debugging. Union types are powerful as they allow for expressions to hold multiple possible types at runtime, the Crystal compiler checks that any method calls using the defined expressions are valid within the range of union types in said expressions [38, p. 25].

Performance times and memory usage in Crystal are quite admirable when compared to other languages; in the chart below are performance and lines of code statistics derived from a study comparing the efficiency of several compiled languages adjacent to Crystal.

Language	Time(s)	Memory (Mb)	Lines of code (LOC)
C++	1.94	1.0	101
Rust	2.42	4.8	86
Crystal	2.91	1.2	77
Nim	3.14	0.8	98
Go	4.2	10.0	124
Java	4.03	513.8	136

Image 5 - Implementation of a simple interpreter, source: [38, p. 8]

In combination with Crystal, The Kemal Framework will be used to speed up and optimize our code for WebSocket programming within Crystal. Kemal is an intuitive and modular web framework architecture developed by Serdar Dogruyol [38, p. 175]. The baseline requirements for Kemal to run have CPU and memory requirements listed at +/- 1MB. Considering how fast Crystal processes execute, these requirements are certainly impressive. This framework provides a RESTful interface, which means that it was built to adhere to the REST API specifications [38, p. 177]. Compared to the Ruby Sinatra framework, Kemal can process approximately 28 times more requests per second. This is very impressive considering the speed at which the Sinatra web framework can function, and the large number of applications built with it in modern development.

The communities for both Crystal and Kemal are quite sparse in comparison to those of the other languages and frameworks mentioned in this paper. As discovered in the practical section, it has a community size similar to that of the Elixir language. This lack of support is due both to the youngness of the language and to the rate at which it has been adopted among developers worldwide. Both the Framework and the language do uphold an actively updated documentation page and offer for users to send in questions if they need assistance regarding the technical details of each system. Installing additional libraries or frameworks in Crystal is very simple and only requires for the names of dependencies to be

added into a shards.yml file, and then be executed with the command ‘shards install’ [44]. The simplicity of this dependency manager proved to be very comfortable to work with in the practical section.

A web server written with the Crystal standard library performs better than web servers written in Node.js, Nim, Rust, and Scala [38, p. 9]. When paired with Kemal, we get a performance output that exceeds that of Sinatra/Rails (Ruby), Pheonix (Elixir), and even Martini/Gin (Go). All this information makes Crystal stand out in instances of web server application programming. As visualized in the practical part of this thesis, Crystal ranked extremely well in high-performance computations. As an example of the impressive performance of this language, listed are examples of companies using Crystal at the enterprise level even though it is such a young language, as stated by Ivo Balbaert and Simon St. Laurent: “Some 15 companies, such as ProTel, Bulutfon, DuoDesign, Appmonit, RainForest QA and Manas itself, already use it for production projects. Some of them, such as ProTel and Bulutfon, experienced scaling problems with their Ruby server infrastructure. For that reason, they rewrote their web service using the Kemal framework in Crystal. In one instance at ProTel, 100 Unicorn workers could be replaced by a single Kemal process to do the same amount of work” [38, p. 16].

Another example of Crystal being used in a company is provided in a discussion by the founder and CEO of Red Panthers P. S. Hartsankar, a Ruby on Rails development studio which builds web/mobile applications: “Our local dashboard for the previous POS system was too slow: delay of seconds during a sale is not acceptable. The client required a better response time at the local readers, so we rewrote it in Crystal and are now able to provide a 10 to 15 micro-second response, a 200,000x improvement! We are a Ruby on Rails firm, and if Crystal hadn’t existed, our client was leaning toward C++ or Go since we had also worked in Go before. But using Crystal felt more natural for us as we already have parts of the code written in Ruby. It helped us to easily port these to Crystal” [38, p. 16]. This shows the potency of Crystal as a web application development language and how it can perform at blazingly fast speeds, essentially providing a fast and comfortable alternative to C++.

3.5 Multi-Criteria Decision Analysis

The MCDA method is a globally used technique for weighting and scoring various parts of some system or research to assist with higher-level decision making [45, pp. 9-10]. MCDA provides an ordering of options after the final sum of weighted totals are calculated. This technique allows for a complex mixture of variable criteria to be analyzed together, but in a modular way as that each criteria's results can provide valuable information about itself and the other criteria it is being ordered against.

These separated parts of the studied and analyzed material can then be combined with all the other parts and generalized to assist other readers in their decisions and opinions of the referenced material. This thesis should not force others into a decision, but rather guide them along gently to make their own informed decisions based off the context provided. Weights and scores for the analyses need to be consistent across all variables, to appear as objective as possible, and when the decisions are purely subjective, the evidence for the scoring is backed up by evidence from practical experience [45, p. 46].

The image below provides a breakdown of the steps that will be generally followed in the following practical section of this thesis, and a portion are described in further detail further on as well.

1. **Establish the decision context.**
   1.1 Establish aims of the MCDA, and identify decision makers and other key players.
   1.2 Design the socio-technical system for conducting the MCDA.
   1.3 Consider the context of the appraisal.
2. **Identify the options to be appraised.**
3. **Identify objectives and criteria.**
   3.1 Identify criteria for assessing the consequences of each option.
   3.2 Organise the criteria by clustering them under high-level and lower-level objectives in a hierarchy.
4. **'Scoring'. Assess the expected performance of each option against the criteria. Then assess the value associated with the consequences of each option for each criterion.**
   4.1 Describe the consequences of the options.
   4.2 Score the options on the criteria.
   4.3 Check the consistency of the scores on each criterion.
5. **'Weighting'. Assign weights for each of the criterion to reflect their relative importance to the decision.**
6. **Combine the weights and scores for each option to derive an overall value.**
   6.1 Calculate overall weighted scores at each level in the hierarchy.
   6.2 Calculate overall weighted scores.
7. **Examine the results.**
8. **Sensitivity analysis.**
   8.1 Conduct a sensitivity analysis: do other preferences or weights affect the overall ordering of the options?
   8.2 Look at the advantage and disadvantages of selected options, and compare pairs of options.
   8.3 Create possible new options that might be better than those originally considered.
   8.4 Repeat the above steps until a 'requisite' model is obtained.

Image 6 - MCDA detailed steps, source: [45, p. 50]

3.5.1 Selection of Criteria

Criteria that will be aggregated and analyzed during the practical MCDA matrices are decided upon after the completion of the backend servers and frontend client and are able to change throughout the analyses. The selected criteria must be independent and measurable so that they are able to be assessed in some qualitative setting when necessary. All criteria must be measured against an objective that will lead to an ordered result of criteria performance at the end of the analysis [45, p. 11].

Criteria weights are subjectively assigned based on a predefined valuation of the categories that are scored in the MCDA matrices. These weights are used to prioritize the value of different categories of performance scores and are used by multiplying them with the relevant scores to provide a final weighted score. By assigning weights that are backed up with evidence of specified priority, analysis of the scoring can be easily visualized to

assist with identifying the more/less important scores in analyses – providing a more concrete and relevant final scoring for the analysis matrices.

Feedback about choices and performance of the selected criteria is crucial for continuing the analysis and capturing more accurate data as it progresses by learning from the existing results. Data from this thesis can be shared with colleagues and then modified/improved based on the expressed subjective opinions of any involved party [45, p. 13].

3.5.2 Measured Criteria Options

There is a quantitative and qualitative matrix populated with the necessary results based on variables that are most relevant for these analyses. The justifications for the weighting and the measured variables are clarified in the relevant sections below. The qualitative analysis is included as a sort of complementary addition to the primary analysis – the quantitative MCDA. There are key differences in how the qualitative and quantitative matrices are populated and measured as listed below:

- Quantitative analysis consists of data measured *purely* objectively as a result of output from the used system. The results are based on raw performance outputs and are aggregated as such, with no subjective influence besides the final weighting of scores.
- Qualitative analysis consists of *primarily* subjectively allocated numeric values measured based on the experience and studies of the researcher conducting the analysis. Decisions on scores can be derived from certain objective data gathered through the research process but consist of a majority of subjective opinion. Weights are subjectively decided upon, as with the quantitative analysis.

3.5.3 Calculating Results and Sensitivity Analysis

The results of both qualitative and quantitative matrices are normalized (if applicable) and summed for a final performance score that provides a distinct ordering of the analyzed criteria. Based on the resulting ordering of results, a discussion surrounding the results will be formulated and evaluated to summarize the research performed and concluded. In the case of min-max normalized performance scores, the lower final scores will be ranked as

the better potential outcomes whilst the higher will be ranked as the worse potential outcomes. In the case of no normalization, the highest scores will be ranked as the best potential outcomes, and the lower scores will be ranked as the worse potential outcomes.

A sensitivity analysis will be performed to expose the differences in results when the weights for certain categories are modified differently from the optimal weighting proposed for the criteria. This will highlight the differences in performance score ordered results when priority is placed on different criterion. After the analysis is complete, the results will be discussed and measured against each other in order to describe the situations in which the different weighted categories would be advantageous or disadvantageous regarding the resulting performance scores [45, pp. 99-101].

4 Practical Section

4.1 Practical Overview

The purpose of the following practical section is to benchmark each language and ascertain which situations they perform best in. Backend WebSocket servers were developed and then set up to interact with a frontend client which instantiates large amounts of users connecting to the server. Load tests with varying numbers of users and processes were triggered for each server to test how well each programming language's server application could perform under different levels of stress. The development environment was set up to mimic common intermediate developer environments, and the frameworks/libraries chosen for WebSocket development in each language were those that provide the most intuitive and streamlined functionality. MCDA matrices were modeled to explore the various scores of selected criteria in both qualitative and quantitative settings for the four languages. By having performed multiple rounds of testing at different levels that imitate real-world web applications, a conclusion was formulated that derives the advantages and disadvantages of each language in similar real-world situations.

4.2 The Development/Testing Environment

All server applications were written using the Atom text editor on a Windows 10 system. The Windows Subsystem for Linux was installed and utilized for scripting and load testing purposes. The goal was to emulate an environment in which beginner-intermediate developers would be most commonly desiring to study and develop their applications.

The specifications of hardware/software that were actively utilized during development and load testing are as follows:

- CPU: Ryzen 7 3700x
- RAM: 16GB 2400 MHz
- GPU: NVIDIA RTX 3070
- OS: Windows 10 Build 19044 with Ubuntu 20.04 and WSL version 1

Languages and Frameworks:

- Python 3.8.10 using the 'websockets' library
- Crystal 1.2.2 using the Kemal framework
- JavaScript using the Node.js 17.3.0 runtime environment with the ‘ws’ library
- Elixir 1.13.0 using the RiverSide framework

An interesting side note is that using bash in the Ubuntu console client vs. bash in CMD results in a significant difference in web server performance. The Ubuntu console is approximately 1.5 to 2 times slower, which makes it seem like the implementation is slowing down the processes. This is not relevant for performance scores as the results gathered were consistent across one system.

4.3 Web Application Benchmarking Software

Artillery was used as the first load testing client, and most of the existing tests were performed with this software [46]. However, Artillery did not include the ability to benchmark response times using actual data from the WebSocket connection. It only allowed for measurements of the opening/closing connection; hence why an alternate tool was found.

The next-best tool was k6. The load testing tool k6 was used for the WebSocket client-side virtual user (VU) instantiation and messages sent from said VUs. For the script that was used, there were a set number of *virtual users* to create and how many *iterations* of the exec function to process per VU [47]. When a WebSocket connection was initiated, it would execute the code block of event handlers based on data sent form the server and wait for a close request from any source. The JavaScript file with the configuration and socket event handlers is seen in image 7.

```
import ws from 'k6/ws';
import { check } from 'k6';

export const options = {
  discardResponseBodies: true,
  scenarios: {
    contacts: {
      executor: 'per-vu-iterations',
      vus: 1000,
      iterations: 100,
      maxDuration: '1h30m',
    },
  },
};

export default function () {
  const url = 'ws://localhost:3000';
  const params = { tags: { my_tag: 'hello' } };

  const res = ws.connect(url, params, function (socket) {
    socket.on('open', () => console.log('connected'));
    socket.on('message', (data) => {console.log('msg received');
socket.close()});
    socket.on('close', () => console.log('disconnected'));
  });


  check(res, { 'status is 101': (r) => r && r.status === 101 });
}
```

Image 7 - k6 WebSocket client program, source: [47]

The script was run within the sub and master scripts, and once a server was started, this connected to the servers over port 3000 and initiated the send and receive functions. Meta-analysis was performed on all benchmark files and averaged then normalized to obtain the final results, which are presented with the weighting applied.

4.4 The WebSocket Backend Servers

In the following section, there are code snippet examples of web servers that were used for the necessary computational load testing. These are not the true variants used during load testing but demonstrate how methods and events were written in each language.

4.4.1 JavaScript

A version of the JavaScript WebSocket server can be seen in image 8 below.

```
const WebSocket = require('ws');

// Creating a websocket instance, binding to port
const wss = new WebSocket.Server({port: 3000});
console.log("JavaScript server listening on port 3000");


function fib (num) {
    if(num <= 1) return 1;
    return fibonacci(num-1) + fibonacci(num - 2);
}

// Handling when a WebSocket client connection requests the initial handshake
wss.on('connection', (ws) => {
    fib (35);

    // Sends a message to the connected client
    ws.send("");

    // Event handling when client sends data to the server instance
    ws.on('message', function message(data) {
        console.log("");
    });

    // On WebSocket client connection close write disconnect message
    ws.on('close', function close() {
        console.log('client disconnected');
    });
});
```

Image 8 - JavaScript WebSocket server, source: Own work

Creating the JavaScript backend was quite straightforward, and it functions as expected for a WebSocket communication server.

4.4.2 Elixir

An example of two of the four necessary program files for the Elixir WebSocket server can be seen in images 9 and 10 below.

```
# Define event handler module
defmodule WebSocket.Handler do
  use Riverside, otp_app: :websocket_server

  def fib(n) do
    if n<=1 do
      1
    else
      fib(n-1) + fib(n-2)
    end
  end

  # Decorator to add RiverSide framework functionality to below functions
  # Alternatively, use @decorate_all <decorator> to cover all functions in a
module
  @impl Riverside

  # Create WebSocket Instance callback function, handler
  def init(session, state) do
    fib(35)

    # Send message to client
    deliver_me("")
    {:ok, session, state}
  end

  @impl Riverside

  # Print message on receival from client and return to sender callback
function
  def handle_message(msg, session, state) do
    IO.puts msg
    deliver_me(msg)
    {:ok, session, state}
  end

 @impl Riverside
  def terminate(reason, session, state) do
   # Perform socket close cleanup functionality if necessary
    :ok
  end
end
```

Image 9 - handler.ex program file - part of the WebSocket server, source: Own work

```
# Define Application module
defmodule WebSocket.Application do

# Application entry point
# Specifies supervisor behavior for RiverSide handler
use Application
  @impl true
  def start(_type, _args) do
    IO.puts "Elixir server listening on port 3000"

    # Set child spec
    [
      {Riverside, [handler: WebSocket.Handler]}
    ]


    # Starts the supervisor with one-for-one strategy
    # with the custom RiverSide handler
    |> Supervisor.start_link(
      # If a child dies, only one will be restarted, in this case only this one
      strategy: :one_for_one,
      name: WebSocket.Supervisor
    )

  end
```

Image 10 - application.ex program file - part of the WebSocket server, source: Own work

The Elixir application was the most difficult to put together and understand functionally due to the separation of program files and the necessity for a specific implementation as required through the Riverside framework.

4.4.3 Python

A version of the Python WebSocket server can be seen in image 11 below.

```
import asyncio
import websockets
PORT = 3000

# Used to slow down speed of responses
def fib(n):
    if n <= 1:
        return 1
    return fib(n-1) + fib(n-2)

# Handler takes instance of websocket client
async def handler(websocket):
    fib(35)
    await websocket.send("")

    # Try used to catch when client connection is closed
    try:
        async for message in websocket:
            print("")
            await websocket.send(message + " <- message sent from client")

    # Print WebSocket client disconnect message on close, triggered by disconn
err
    except websockets.exceptions.ConnectionClosed as e:
            print("Client disconnected")
            print(e)

# Start the WebSocket server and take handler coroutine, host, and port
ss = websockets.serve(handler, "localhost", PORT)
print ("Python server listening on port: " + str(PORT))

# Run start server once
asyncio.get_event_loop().run_until_complete(ss)

# Continue event loop so that server stays alive
asyncio.get_event_loop().run_forever()
```

Image 11 - Python WebSocket server, source: Own work

The Python server was not particularly difficult to create and understand functionally.

4.4.4 Crystal

A version of the Crystal WebSocket server can be seen in image 12 below.

```
require "kemal"

def fib(n)
  if n <= 1
    1
  else
    fib(n-1) + fib(n-2)
  end
end


# Create websocket handler, matches the port on localhost
ws "/" do |socket|
  fib(35)

  # Send message to the connected client
  socket.send ""

  # on close returns client disconnect message and client socket PID
  socket.on_close do |_|
    puts "Client disconnected: #{socket}"
  end
end

Kemal.run
```

Image 12 - Crystal WebSocket server, source: Own work

The Kemal structure implemented within Crystal was very straightforward and concise. All project files including program files, master and sub-scripts, load testing results, and required framework files for the thesis work were kept up to date on a private GitHub repository for organization and versioning purposes.

4.5 Server/Client Execution Master and Subscripts

A set of scripts were written to ease the process of testing the servers with various load sizes and programs. One master script executes the subscripts in a serial order that can be modified for different testing approaches. A subscript exists for every individual serial/concurrency load test and is executed in the order specified in the master script.

Below is an example (with the crystal server utilizing 8 cores) of how the separate scripts initiate one server, run the load test k6 script, and then consequently shut down the server and send data results to a file.

```
############# CRYSTAL #############
pushd Crystal/crystal_server_compute/
CRYSTAL_WORKERS=8 ./crystal_server_compute36 & # server process sent to background to allow for other cmds
cr_pid=$!
popd
pushd JavaScript/javascript_client/
sleep 2 # allow the server to start, takes longer with kemal in crystal
k6 run $test > test_results/crystal_report_8.txt
sleep .5
kill $cr_pid # kill last server running in background
pkill -f 'crystal' #kill all cached crystal run processes
echo -e "\n^^^^^^^^^^^^^^^^^^^^^^^^^^^^CRYSTAL_8 SUMMARY^^^^^^^^^^^^^^^^^^^^^^^^^^^^\n\n\n" >> test_results/crystal_report_8.txt
popd
```

Image 13 - Subscript for server init/kill (Crystal 8-core section), source: Own work

Each script sends the output of each individual client/server process concatenated into one output file at the end of all processes. The resulting file is then used to succinctly analyze the time performance results of each language's server. These outputted results were then used to fill in the criteria results in the quantitative MCDA matrix.

4.6 Qualitative MCDA

This section is meant to be a short analysis on the languages used and their pros and cons. It does not factor into the final decision of optimal criteria for cooperative application development using WebSocket. It simply provides a succinct overview of the subjective experience of programming in and learning about each language during the practical part.

Based on the experience and practical work performed with each language, a qualitative analysis was performed to provide insight into the development process throughout the duration of this thesis. The outcome of the qualitative analysis of the languages does not have a high priority in fulfilling the objectives; however, it is important for developers to work with a language that is comfortable and manageable for them. In the results and discussion, the criteria results were analyzed and compounded along with the quantitative results to provide a summary of which languages are optimal in different situations. The selected criteria used to subjectively analyze all four languages are as follows:

- Flexibility/Extendability: This criteria measures the capabilities of the language in question to extend existing features or provide entirely new ones by writing code in the language itself. Specifically, this criteria focuses on the ability to do this in a way that makes the new/changed features look similar to existing language

constructs. Related concepts include: Macros, Reflection, Self-Modifying Code, Monkey Patching, & DSLs

- Syntactic Sugar: Syntactic Sugar refers to the inclusion of superfluous language constructs and expression forms that provide a more terse or expressive representation for a given language feature. Some examples of Syntactic Sugar include: List Comprehensions, Multiple Assignment, Augmented Assignment, Block Arguments, Keyword Arguments, String Interpolation
- High Levelness: This criteria expresses the overall abstractness of the given language, that is, how distant code written in the language is from its eventual representation in executable machine code. High level languages provide many facilities which abstract over the mechanics of their execution which help to simplify programming, mitigate many types of bugs, and make code written in these languages easier to reason about. Examples of High-Level features include: Garbage Collection, Advanced Type Systems, Concurrency Models, Error Handling Facilities, and Abstraction Paradigms (i.e. OOP or Function Level programming)
- Language Documentation: This criteria measures the existence of basic documentation resources for a given language including information on the usage of built-in language features and constructs, along with indexes of Standard Library Classes/Functions and examples/instructions on their use.
- Libraries/Modules: This metric represents the availability and extensiveness of 3rd party codebases for a given language. This includes libraries and frameworks which extend the functionality of the language or provide facilities for more elegant expressions of frequently used code patterns.
- Communities/User Base: This criteria factors in all the existing communities and size of the developer user base for the language. Search result hits and stack overflow communities were analyzed to come up with a generalized result.

The table below factors in the above criteria and displays the results of each language with low, average, or high rankings of the languages. Plus and minus signs are used to add or diminish priority on the values. Table 2 provides a detailed analysis of table 1 and provides numeric values which were weighted and totaled.

Criteria	JavaScript	Elixir	Python	Crystal
Flexibility/Extendability	Average	Low+	Average-	High
Syntactic Sugar	Low	Average+	Average	High-
High levelness	High	High	High	High-
Language Documentation	High	High-	High	High-
Libraries/Modules	High	Average	High	Low+
Community/User Base	High-	Low+	High	Low

Table 1 - Qualitative matrix, approximate rankings, source: Own work

Criteria	JavaScript	Elixir	Python	Crystal	Criteria Weights
Flexibility/Extendability	8.0	6.5	7.5	10	.3
Syntactic Sugar	5.0	7.5	6.0	8.0	.05
High levelness	10	10	10	9.0	.05
Language Documentation	10	9.0	10	9.0	.2
Libraries/Modules	10	7.5	10	6.5	.2
Community/User Base	9.0	6.5	10	6.0	.2
Total Unweighted	**52**	**47**	**53.5**	**48.5**	
Total Weighted	**8.95**	**7.425**	**9.05**	**8.15**	

Table 2 - Qualitative numeric matrix, source: Own work

4.7 Qualitative Decisions for Weights

The weights shown in the previous MCDA matrix were constructed based on the results and evidence listed below:

- Flexibility/Extendability: Highest weight due to the criticality of this metric in regards to the subjective usage of a language. Extendability is the essence of a good language, in that it enables the extension/addition of features not covered by the core language. Because 3rd party frameworks/libraries were being used for Websocket functionality, this metric is an important measure of usability for this use-case.
- Syntactic Sugar: Lowest weight tied with high-levelness. Not essential in any way, simply a measure of the amount of bells-and-whistles a language provides that enhance usability.
- High-levelness: Important metric for language usability overall, however all the languages included in this thesis were fairly high-level, and thus this this was an insignificant criteria on which to compare them.
- Documentation: Important metric for language usability, in that without proper documentation of the core language it is impossible to understand how to use the language/STDLIB in the first place.
- Libraries: Important metric because without existing libraries/frameworks much functionality would have to be re-invented for a language. This is especially relevant regarding the topic of this thesis: WebSocket implementations.
- Community: Important metric that runs parallel to Documentation/Libraries. Without a substantial community of users, a language will not have enough 3rd party code or discussions of errors/features to be usable.

4.8 Quantitative MCDA

The quantitative analysis focused on testing criteria that can accurately generalize real world scenarios of web application computational intensity. The Fibonacci algorithm (starting from 1) was implemented naively to emulate a computation involving many nested function calls and arithmetic. In terms of what the selected criteria are measuring – 10,100/100,1000 is serial testing and 100,10/1000,100 is concurrency testing. For the former, testing was done with fewer concurrent VUs and more operations per VU. With the latter, testing was done by instantiating more concurrent VUs but performing less operations per each VU. It was necessary to focus on both concurrent and serial testing, as with real applications these parameters vary greatly depending on the purpose and scope of the application. The tests performed were sufficient to gather the necessary amount of relevant data to fulfill the objectives of this thesis. The criteria chosen for testing client virtual user handling and processes with the servers in each language are the following (the first number denotes the number of virtual users and second how many serial processes to execute for each virtual user. Comma separated values after the dash are separate load test variants executed with the prefixed VU/processes parameters):

- 10, 100 - No Fibonacci, Fibonacci 20, and Fibonacci 36
- 100, 10 - No Fibonacci, Fibonacci 20, and Fibonacci 36
- 100, 1000 - No Fibonacci, Fibonacci 20
- 1000, 100 - No Fibonacci, Fibonacci 20

For the load testing, a total of 8 separate tests were run for the four languages which make up the column criteria, and the speed at which each language performs the previously described load tests is measured in seconds. For Crystal and Elixir, variants of the lowest and highest number of cores possible to use will be declared for program executions. As this is not possible for Python and JavaScript, they only have one possible program execution configuration. Below are the criteria used for each of the languages:

- Crystal using 1 core
- Crystal using 8 cores
- Crystal using 1 core and compiled with --release

- Crystal using 8 cores and compiled with --release
- Elixir using 1 core
- Elixir using 8 cores
- Python
- JavaScript

Each available load testing script was run three times for each language and the mean is derived in order to work with more accurate data. The scripts individually output data results, and those results were then all concatenated into one file. These concatenated results were originally outputted from the k6 load testing framework that returns the time of process completion in seconds which is used for the results in the MCDA quantitative matrix below (table 3). The memory test was performed by screen recording the resource monitor Private (KB) memory counter and using the highest value reached during the trials. Table 3 displays the quantitative performance matrix of load tests which were averaged over three runs each. The Fibonacci sections measured results in seconds, and the RAM sections measured results in the maximum number of megabytes reached for the language during load testing.

The superscript * denotes the original values prior to outlier modification

The superscript ** denotes that the performance results of JavaScript must be observed skeptically

Criteria	JS	Ex 1 core	Ex 8 cores	Py	Cry 1 core	Cry 1 core Release	Cry 8 cores	Cry 8 cores Release	Criteria Weights
Red lettering = Min/Max Normalized, Blue cells = Worst score, Green cells = Best score									
NoFib									
10, 100	1.00 | **.087**	1.60 | **.348**	1.50 | **.304**	0.80 | **0**	2.80 | **.870**	1.50 | **.304**	3.10 | **1**	1.40 | **.261**	0.092
100, 10	0.70 | **.111**	1.20 | **.667**	1.40 | **.889**	0.60 | **0**	1.20 | **.667**	1.20 | **.667**	1.50 | **1**	1.50 | **1**	0.092
100, 1000	117.1 | **0**	123.7 | **.117**	117.3 | **.004**	173.7 | **1**	129.1 | **.212**	119.7 | **.046**	153.7 | **.647**	151.2 | **.602**	0.092
1000, 100	80.8 | **.388**	110.5 | **.719**	135.7 | **1**	127.4 | **.907**	46.0 | **0**	47.1 | **.012**	122.3 | **.851**	134.9 | **.991**	0.092
Fib20									
10, 100	1.0 | **0**	1.8 | **.533**	1.5 | **.333**	2.5 | **1**	1.2 | **.133**	1.2 | **.133**	1.4 | **.267**	1.4 | **.267**	0.092
100, 10	0.9 | **0**	1.3 | **.286**	1.4 | **.357**	2.3 | **1**	1.3 | **.286**	1.2 | **.214**	1.5 | **.429**	1.5 | **.429**	0.092
100, 1000	117.4 | **0**	137.0 | **.171**	127.6 | **.089**	232.2 | **1**	137.8 | **.178**	136.3 | **.165**	158.6 | **.359**	155.3 | **.330**	0.092
1000, 100	83.2 | **.352**	98.8 | **.511**	146.6 | **1**	79.2 | **.311**	50.2 | **.014**	48.8 | **0**	132.2 | **.853**	130.4 | **.834**	0.092
Fib36									
10, 100	102.0 | **.742**	134.7 | **1**	20.8 | **.100**	134.7 | **1** (2464.4)*	89.2 | **.640**	52.0 | **.346**	14.1 | **.047**	8.2 | **0**	0.092
100, 10	101.9 | **.748**	133.6 | **1**	20.2 | **.100**	133.6 | **1** (799.8)*	88.5 | **.642**	52.3 | **.355**	13.4 | **.046**	7.6 | **0**	0.092
RAM max val., Fib20									
100, 1000	56.7 | **.144**	45.1 | **.093**	47.1 | **.102**	24.0 | **0**	251.2 | **1**	250.3 | **.996**	54.5 | **.134**	50.2 | **.115**	0.02
1000, 100	69.1 | **.162**	70.8 | **.171**	56.2 | **.089**	40.3 | **0**	189.1 | **.836**	218.3 | **1**	164.1 | **.696**	116.4 | **.428**	0.02
RAM max val., NoFib									
100, 1000	56.2 | **.069**	44.2 | **.043**	48.5 | **.052**	24.8 | **0**	255.0 | **.507**	478.8 | **1**	35.5 | **.024**	48.1 | **.051**	0.02
1000, 100	65.8 | **.127**	66.9 | **.134**	54.1 | **.056**	45.0 | **0**	190.1 | **.888**	208.4 | **1**	150.4 | **.645**	98.4 | **.327**	0.02
Normalized performance scores, totaled									
Unweighted	**2.93**	**5.793**	**4.475**	**7.218**	**6.873**	**6.238**	**6.998**	**5.635**	
Weighted	**0.233****	**0.501**	**0.390**	**0.664**	**0.400**	**0.286**	**0.536**	**0.452**	

Table 3 - Quantitative performance matrix, source: Own work

A simple program was used with the NumPy library to quickly input load testing data and return normalized values from 0 to 1 using min-max normalization.

```
import numpy as np
def norm(x):
    min = np.min(x)
    max = np.max(x)
    range = max - min
    return [(a - min) / range for a in x]


x = [102.0, 134.7, 20.8, 89.2, 52, 14.1, 8.2]
normX = norm(x)
roundNormX = [round(e, 3) for e in normX]

print(roundNormX)
```

Image 14 - Min-max normalization program file, source: Own work

4.9 Quantitative Decisions for Weights

There are many varieties of web applications that fall into all three categories of computations which were tested. Due to this, the full-scale quantitative MCDA performance matrix is used when the weightings are equal for all computation categories.

The fib36 Python computation results were made equivalent to the second worst performer, as the outlier minimizes the normalized result of all other criteria. This method was chosen as it doesn’t significantly affect the resulting performance scores for all tests.

The three various types of load testing completed include a simple message send and receive with no computation, the server computing naive Fibonacci 20 on message receival, and the server computing naïve Fibonacci 36 on message receival. These three cases were implemented to mimic scenarios in the real world of cooperative application development. The naive Fibonacci algorithm was chosen because it is a good synthetic test of numeric computation and many nested function calls. This being said, the results should still be taken with a grain of salt as this test is highly artificial compared to the diverse range of potential algorithms computed in real life instances.

Weights were initially set to be equal for all categories of computations in order to gather information on the best performers for all scales of computational difficulty. It is likely that the computation required for Fibonacci 20 would be the most representative of real-world use case web server computations. However, due to the vast number of applications that also do very little computation or very high levels of computation, it cannot be assumed that there is any case that is the most relevant for this level of research,

and thus all three computation categories were analyzed equally in the MCDA sensitivity analysis section.

As a reference, below are some examples of what types of web application might fall into each of the tested computation categories:

Situations where little to no computation might be performed (I/O performance) such as with no Fibonacci computation:

- Web applications which pull from and push data into a database
- Web applications which are formatting data such as rendering an HTML page or simple data organization on a website
- Simple web app games or mobile games which are synchronizing states between the server and connected clients

Situations where computations similar to Fibonacci 20 might be performed (standard algorithms):

- Document editors/manipulators which perform sorting/typing/simple mathematical algorithms on text
- Ranking algorithms on social media sites or forums for various criteria
- IOT applications which are monitoring real life sensory feedback and reacting as necessary
- Online or mobile games with pathfinding and additional AI calculations

Situations where computations similar to Fibonacci 36 might be performed (high performance computing [HPC], numeric crunching):

- Applications that perform image/video/audio processing; files are converted into different formats
- Map based applications monitoring traffic and conditions or finding optimal routing paths for calculating directions

By weighting all categories equally for initial performance scores and then conducting a sensitivity analysis, a generalized group of results were used to make informed decisions on the proficiency and use cases of each language for each computational category.

4.10 MCDA Sensitivity Analysis

In addition to presenting an equal weighting for all computational load test criteria, three variants were presented for the sensitivity analysis of the weighting for the quantitative performance matrix. By clearly prioritizing each of the relevant load testing categories, the stability and robustness of the matrix for the criteria weights were discovered. Variance in performance was different when different weightings were prioritized – but not by too grand of a scale. Additionally, the modifications of weights changed the behaviors of the optimal solutions of the model. The following weights were applied (Memory usage weights were left unchanged):

- Prioritizing NoFib
- Prioritizing Fib20
- Prioritizing Fib36

Table 4 presents all three of the additional weighting variants with the final results (in seconds) and colorized cells for identifying optimal and least optimal performers.

The superscript ** denotes that the performance results of JavaScript must be observed skeptically

Computation Type													
NoFib				Fib20				Fib36		RAM Fib20		Ram NoFib	
10, 100	100, 10	100, 1000	1000, 100	10, 100	100, 10	100, 1000	1000, 100	100, 1000	1000, 100	100, 1000	1000, 100	100, 1000	1000, 100
Weights – NoFib Priority													
0.115	**0.115**	**0.115**	**0.115**	**0.0575**	**0.0575**	**0.0575**	**0.0575**	**0.115**	**0.115**	**.02**	**.02**	**.02**	**.02**

Performance Scores with Weights Applied and Summed							
JS	**Ex 1 core**	**Ex 8 cores**	**Py**	**Cry 1 core**	**Cry 1 core Release**	**Cry 8 cores**	**Cry 8 cores Release**
0.269**	**0.538**	**0.384**	**0.64**	**0.448**	**0.308**	**0.553**	**0.454**

Weights – Fib20 Priority													
0.0575	**0.0575**	**0.0575**	**0.0575**	**0.115**	**0.115**	**0.115**	**0.115**	**0.115**	**0.115**	**.02**	**.02**	**.02**	**.02**

Performance Scores with Weights Applied and Summed							
JS	**Ex 1 core**	**Ex 8 cores**	**Py**	**Cry 1 core**	**Cry 1 core Release**	**Cry 8 cores**	**Cry 8 cores Release**
0.256**	**0.518**	**0.36**	**0.72**	**0.383**	**0.279**	**0.461**	**0.396**

Weights – Fib36 Priority													
0.0575	**0.0575**	**0.0575**	**0.0575**	**0.0575**	**0.0575**	**0.0575**	**0.0575**	**0.23**	**0.23**	**.02**	**.02**	**.02**	**.02**

Performance Scores with Weights Applied and Summed							
JS	**Ex 1 core**	**Ex 8 cores**	**Py**	**Cry 1 core**	**Cry 1 core Release**	**Cry 8 cores**	**Cry 8 cores Release**
0.407**	**0.662**	**0.281**	**0.76**	**0.495**	**0.33**	**0.362**	**0.289**

Table 4 - Sensitivity analysis matrix, source: Own work

5 Results and Discussion

Based on the literature review and the performance data from the criteria provided during the practical load tests, the objectives of the thesis were achieved. Practical performance results and optimal rankings of languages were gathered to fulfill the goals of the thesis. Qualitative features were documented to provide insight into the advantages and disadvantages of working with of each of the languages.

Python was the language that was easiest to learn and work with during the duration of this thesis, and JavaScript followed closely after. This was expected due to the community size and the available online documentation for troubleshooting and programmatic inquiries. Both have many frameworks and libraries online which create opportunities to work on a wide range of applications. Crystal was very pleasant syntactically, and overall comfortable to learn how to use from the official documentation. Elixir proved to be the most difficult language to learn and work with practically, due to it being functional and not primarily OOP like other languages. All in all, all of the programming languages were manageable to work with and had abundant community support and more than adequate official documentation. All the languages used are recommended for use at any level of application development. Python and JavaScript are a better choice for beginners, and Elixir and Crystal are ideal for more experienced developers – although none are too simple or difficult to be neglected for projects of any scope.

JavaScript performed the best as seen in the final performance scores of the quantitative MCDA matrix, however, there is an issue with the way JavaScript was performing the naive Fibonacci computations. From the extensive testing within the scope of the WebSocket application, any sequence calculated below Fibonacci 37 by the V8 compiler was seemingly automatically caching the previous Fibonacci function results – making it difficult to test JavaScript performance accurately against the other languages. When monitoring the runtimes of the Fibonacci numbers 37 and higher, the results seemed to be consistently sporadic in terms of performance timings, and thus no performance results would be better in comparison to Fibonacci 36. This makes it appear like the V8 compiler was memoizing results automatically even when prompted to explicitly perform the function in the naive exponential time complexity Fibonacci program. There were attempts to disable compiler optimization to fairly test the language against the others, but

all were unsuccessful. Due to the unknown optimizations that occur with the JavaScript JIT compiler, the quantitative results for the language were not acceptable. The optimizations JavaScript utilizes do still give it value in this research, as the algorithm was implemented identically to the others, but V8 was performing memoization from the JIT compiler, and not explicitly from the written code. This provides evidence that JavaScript may often output unpredictable results with many different types of algorithms and execute them contrary to how the developer would expect them to be executed.

After accounting for the results of the sensitivity analysis, the supplementary models provided an overview of results when pivoting the priority weights to different scales of load tests. A total of four quantitative matrices were created and scored, ranked by best/worst performance, and analyzed to provide a final overview of how each of the languages perform.

JavaScript using Node.js performed well based on how it was computing the algorithms but could not be reliably tested at lower computation levels. At the highest computation level, JavaScript struggled to keep up with Elixir or Crystal, thus leaving it as an optimal choice for developing applications without intensive number crunching. The ability to easily learn the language and find information on how to develop with it from the large community makes it a solid contender for developing WebSocket applications. However, it would not be recommended to be used on its own in larger projects with more intensive demands, due to the typing system, how the JIT compiler automatically optimizes functions, and the pace at which it performs for larger number crunching/nested function computations.

Factoring in the JavaScript memoization issue, the next-best performer during the sensitivity analysis for serial and concurrency testing was the Crystal programming language. Both the 1-core and 8-core executions of the Crystal program computations had high overall rankings. The results indicated that Crystal performs very well under light, midline, and heavy computational loads. The recent addition of parallelism to the Crystal language provided a significant amount of value to its performance for intense computations – whilst during lighter computations it performed tremendously well with its original single core implementation. The Crystal language alongside the Kemal framework is strongly recommended to be used for WebSocket application development.

Elixir performed similarly to Crystal, the variance in performance scores was dependent on the type of computation and intensity of numeric computation. The

difference being that when utilizing eight cores, elixir consistently performed better than when it was executing on one core. This is implied since Elixir was natively designed to be using all available cores on machine during its execution. Overall, it was the next best contender to Crystal after reviewing the outcomes calculated during the sensitivity analysis. It is strongly recommended to use Elixir for developing applications with rigorous computational and function management demands, particularly with large amounts of concurrent users.

Python ranked the lowest in all final outcomes of the quantitative performance score matrices. It was the best performer during the NoFib computation tests, but only at the lowest level of serial and concurrent computation. This was an expected outcome due to Python being a purely interpreted language and being thread-locked by the GIL. It is not recommended to use CPython for WebSocket application development under any scenario, besides very specific cases (small applications, learning to program with WebSockets, as a supplementary language for non-web-based communication functions). The better alternative would likely be to use an implementation of Python such as PyPy due to the JIT compiler, when it is applicable.

Random-access memory usage of each of the languages showed that all languages besides Crystal were allocating a similar amount of maximum Private memory to their run-time processes. During multiple rounds of testing, Crystal was the only language that suffered from a memory leak on one occasion, hitting a peak of 2GB at its peak. This was experienced on the 8-core execution and is likely a fault due to how young the implementation of parallel processing is in Crystal. Overall, memory readings did not significantly impact the review of each language.

6 Conclusion

The primary aim of this thesis was to determine which languages and relevant frameworks would perform the best when set up as WebSocket servers handling various levels of computational loads from multiple clients. Each level of load testing was implemented to mimic the computational use-cases of existing developed web applications. A proposed sample of virtual users that connect to the servers – as well as how many computations were performed for each virtual user – were defined and tested against.

In the theoretical part of the thesis, the four chosen programming languages and their respective frameworks/libraries were described in terms of functionality, characteristics and structure, history, benefits/setbacks, and general development performance. Information and details on programming languages were constructed from extensive literature review.

In the quantitative practical part of the thesis, comparisons were made using serial and concurrency load tests at different scales, and then the performance results were consequently analyzed using multi-criteria decision analysis. The qualitative analysis proved useful for identifying features and setbacks of languages and providing insight on the practical experience with each. It is important to understand the comfort and options of working with a programming language when beginning projects of any scale and complexity. The necessary program files and master/sub-scripts were developed in order to test the practical application of each language's function in the scope of WebSocket communication with a client. The optimal language for WebSocket performance at an average computational load was identified based on the performance of all selected criteria. After the review of the weighting priority sensitivity analysis, distinctions were made for performance advantages/disadvantages among languages depending on the intensity of arithmetic computation.

7 References

1. MDN CONTRIBUTORS. WebSocket() - web apis: MDN. *mdn web docs* [online]. 6 March 2022. [Accessed 10 August 2021]. Available from: https://developer.mozilla.org/en-US/docs/Web/API/WebSocket/WebSocket#browser_compatibility

2. FETTE, Ian and MELNIKOV, Alexey. RFC 6455 - the websocket protocol. *Document search and retrieval page* [online]. December 2011. [Accessed 2 August 2021]. Available from: https://datatracker.ietf.org/doc/html/rfc6455

3. Half duplex and full duplex. *Study CCNA* [online]. 30 July 2021. [Accessed 4 August 2021]. Available from: https://study-ccna.com/half-duplex-and-full-duplex/

4. MDN CONTRIBUTORS. WebSocket - web apis: MDN. *Web APIs | MDN* [online]. 3 July 2021. [Accessed 10 August 2021]. Available from: https://developer.mozilla.org/en-US/docs/Web/API/WebSocket

5. HAVERBEKE, Marijn. *Eloquent javascript: A modern introduction to programming*. San Francisco, CA : No Starch Press, 2019. ISBN 978-1-59327-950-9

6. JavaScript Tutorial. *JavaScript tutorial* [online]. 2022. [Accessed 20 November 2021]. Available from: https://www.w3schools.com/js/

7. HERRON, David. *Node.js web development server-side development with node 10 made easy*. Birmingham : Packt Publishing, 2018. ISBN 978-1-78862-685-9

8. MDN CONTRIBUTORS. A re-introduction to JavaScript (JS tutorial) - javascript: MDN. *JavaScript | MDN* [online]. 18 February 2022. [Accessed 21 December 2021]. Available from: https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript

9. SENGSTACKE, Peleke. JavaScript transpilers: What they are and why we need them. *DigitalOcean* [online]. 15 September 2020. [Accessed 18 December 2021]. Available from: https://www.digitalocean.com/community/tutorials/javascript-transpilers

10. What is typescript? *TypeScript Tutorial* [online]. 14 April 2021. [Accessed 12 December 2021]. Available from: https://www.typescripttutorial.net/typescript-tutorial/what-is-typescript/

11. CoffeeScript: Introduction. *GeeksforGeeks* [online]. 15 March 2021. [Accessed 13 December 2021]. Available from: https://www.geeksforgeeks.org/coffeescript-introduction/

12. KANERIYA, Tejas. 15+ popular companies using Node.js in 2022. *Insights on Latest Technologies - Simform Blog* [online]. 25 February 2022. [Accessed 3 March 2022]. Available from: https://www.simform.com/blog/companies-using-nodejs/

13. Websockets/WS: Simple to use, blazing fast and thoroughly tested websocket client and server for node.js. *GitHub* [online]. 7 February 2022. [Accessed 5 January 2022]. Available from: https://github.com/websockets/ws

14. DREIMANIS, Gints. What is elixir and why should you use it? *Serokell Software Development Company* [online]. 15 April 2020. [Accessed 25 November 2021]. Available from: https://serokell.io/blog/introduction-to-elixir

15. PETIT, Charles. A brief history of Erlang and elixir. *A brief history of Erlang and Elixir* [online]. 30 January 2019. [Accessed 10 November 2021]. Available from: https://www.welcometothejungle.com/en/articles/history-erlang-elixir

16. Index - erlang/OTP. *Erlang.org* [online]. n.d. [Accessed 6 November 2021]. Available from: https://www.erlang.org/

17. CASSEL, David. Why Erlang? Joe Armstrong's legacy of fault-tolerant computing. *The New Stack* [online]. 19 December 2021. [Accessed 23 November 2021]. Available from: https://thenewstack.io/why-erlang-joe-armstrongs-legacy-of-fault-tolerant-computing/

18. Learn you some Erlang. *The Hitchhiker's Guide to Concurrency | Learn You Some Erlang for Great Good!* [online]. n.d.. [Accessed 23 November 2021]. Available from: https://learnyousomeerlang.com/the-hitchhikers-guide-to-concurrency

19. JURIĆ Saša. *Elixir in action*. Shelter Island, NY : Manning, 2019. ISBN 9781617295027

20. SCHUINDT, Fernando. Elixir vs ruby: How switching to elixir made our team better. *Foxbox Digital* [online]. n.d. [Accessed 24 November 2021]. Available from: https://www.foxbox.com/blog/elixir-vs-ruby

21. 10 academic and historical questions. *Erlang* [online]. n.d. [Accessed 14 November 2021]. Available from: https://www.erlang.org/faq/academic.html

22. Functional Programming - Introduction. *tutorialspoint* [online]. n.d. [Accessed 7 February 2022]. Available from: https://www.tutorialspoint.com/functional_programming/functional_programming_introduction.htm

23. Introduction to mix. *elixir* [online]. n.d. [Accessed 7 February 2022]. Available from: https://elixir-lang.org/getting-started/mix-otp/introduction-to-mix.html

24. The package manager for the Erlang ecosystem. *Hex* [online]. n.d. [Accessed 8 February 2022]. Available from: https://hex.pm/

25. LYOKATO. Lyokato/Riverside: Elixir library: Simple websocket server framework. *GitHub* [online]. 11 December 2021. [Accessed 14 December 2021]. Available from: https://github.com/lyokato/riverside

26. ŁĘCKI, Bartosz. 10 companies that use elixir in production. *NetGuru* [online]. 17 June 2021. [Accessed 8 March 2022]. Available from: https://www.netguru.com/blog/10-companies-use-elixir

27. VISHNEVSKIY, Stanislav. How discord scaled elixir to 5,000,000 concurrent users. *Discord Blog* [online]. 17 September 2021. [Accessed 8 March 2022]. Available from: https://blog.discord.com/scaling-elixir-f9b8e1e7c29b

28. What is Python? *Python institute* [online]. n.d. [Accessed 10 July 2021]. Available from: https://pythoninstitute.org/what-is-python/

29. Python enhancement proposals. *The Python Programming Language* [online]. n.d. [Accessed 12 July 2021]. Available from: https://www.python.org/dev/peps/

30. Python concurrency tutorial. *INE* [online]. 28 October 2020. [Accessed 16 July 2021]. Available from: https://blog.ine.com/python-concurrency-tutorial

31. Asyncio - asynchronous I/O. *asyncio - Asynchronous I/O - Python 3.10.2 documentation* [online]. n.d. [Accessed 16 July 2021]. Available from: https://docs.python.org/3/library/asyncio.html

32. RAVI, Shalini. How does python code run: Cpython and python difference. *C# Corner* [online]. 14 February 2020. [Accessed 20 July 2021]. Available from: https://www.c-sharpcorner.com/article/why-learn-python-an-introduction-to-python/

33. Differences between PyPy and CPython. *Differences between PyPy and CPython - PyPy documentation* [online]. n.d. [Accessed 22 July 2021]. Available from: https://doc.pypy.org/en/latest/cpython_differences.html

34. RAMALHO, Luciano. *Fluent python: Clear, concise, and Effective Programming*. Sebastopol, CA : O'REILLY MEDIA, INC, USA, 2015. ISBN 978-1-491-9-46008

35. Getting started. *websockets 10.2 documentation* [online]. n.d. [Accessed 25 July 2021]. Available from: https://websockets.readthedocs.io/en/stable/intro/index.html

36. PATEL, Jeel. 7 popular tech companies that use python in 2022. *Monocubed* [online]. 9 March 2022. [Accessed 11 March 2022]. Available from: https://www.monocubed.com/blog/companies-that-use-python/

37. Blog - Release Notes. *The Crystal Programming Language* [online]. n.d. [Accessed 3 September 2021]. Available from: https://crystal-lang.org/blog/#release_notes

38. BALBAERT, Ivo and LAURENT, Simon St. *Programming Crystal: Create high-performance, safe, concurrent apps*. Raleigh, NC : The Pragmatic Bookshelf, 2019. ISBN 978-1-68050-286-2

39. Crystal team. *The Crystal Programming Language* [online]. n.d. [Accessed 10 September 2021]. Available from: https://crystal-lang.org/team/

40. GASKINS, Jamie. Enabling crystal's new Multicore Support. *DEV Community* [online]. 25 September 2019. [Accessed 16 September 2021]. Available from: https://dev.to/jgaskins/enabling-crystal-s-new-multicore-support-4l4g

41. Concurrency. *The Crystal Programming Language* [online]. n.d. [Accessed 25 September 2021]. Available from: https://crystal-lang.org/reference/guides/concurrency.html

42. WAJNERMAN, Juan and CARDIFF, Brian J. Parallelism in Crystal. *The Crystal Programming Language* [online]. 6 September 2019. [Accessed 25 September 2021]. Available from: https://crystal-lang.org/2019/09/06/parallelism-in-crystal.html

43. Type inference. *The Crystal Programming Language* [online]. n.d. [Accessed 27 September 2021]. Available from: https://crystal-lang.org/reference/1.3/syntax_and_semantics/type_inference.html

44. The shards command. *The Crystal Programming Language* [online]. [Accessed 2 March 2022]. Available from: https://crystal-lang.org/reference/1.3/the_shards_command/index.html

45. DEPARTMENT FOR COMMUNITIES AND LOCAL GOVERNMENT: LONDON. *Multi-criteria analysis: A Manual*. London : Crow, 2000. ISBN 978-1-4098-1023-0

46. Testing websockets: Artillery. *Artillery Docs* [online]. n.d. [Accessed 14 January 2022]. Available from: https://www.artillery.io/docs/guides/guides/ws-reference

47. WebSockets. *k6 Docs* [online]. n.d. [Accessed 15 January 2022]. Available from: https://k6.io/docs/using-k6/protocols/websockets/