Universitetet
i Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

BACHELOR'S THESIS

Study programme/specialisation:	Spring semester, 2021
Bachelor in Computer Science	Open / Restricted access
Author(s): Andrea Esposito, John Marvin Cadacio	
Programme coordinator: Tom Ryen Supervisor(s): Leander Jehl	
Title of bachelor's thesis: Evaluation of WebAssembly as compile target and runtime for distributed or networked systems	
Credits: 20	
Keywords: WebAssembly Distributed systems Wasmtime gRPC	Number of pages: 79 Stavanger 15.05.2021

Title page for bachelor's thesis
Faculty of Science and Technology

WebAssembly for system programming

Evaluation of WebAssembly as compile target and runtime for distributed or networked systems

Andrea Esposito
John Marvin Cadacio

15 May 2021

Department of Electrical Engineering and Computer Science
Faculty of Science and Technology

University of Stavanger

Abstract

With the release of a new standard, WebAssembly has been a growing trend amongst developers. Primarily, desktop and mobile browsers have full support for WebAssembly as of today. Running it inside the browser gives many benefits, but how about outside the browser? This thesis presents a Wasm+gRPC application template that can be utilized to increase distributed applications' diversity. We also carefully analyze WebAssembly's efficiency and ease of use outside of the browser. Through the development of two Wasm-based distributed applications, we learned about Wasm's functionality, drawbacks, and, most notably, about how performant Wasm is in such applications. This process has been done for multiple programming languages, giving us a better understanding of how simple it is to embed Wasm in many environments.

Executing a set of benchmarks for each implementation, we show that WebAssembly runs remarkably well when embedded in programming languages such as Go and C#. Unfortunately, our findings suggest that Wasm has some flaws in terms of performance and stability, that due to the continuous Wasm updates, might get ironed out sooner than later.

Acknowledgements

We would like to express our gratitude to our supervisor, Leander Jehl, for his invaluable guidance throughout this project. We would also like to thank Theodor Ivesdal and Rodrigo Saramago for their valuable assistance with some of the complications we have encountered during our use of the Pitter machines and BBChain-cluster. Thanks, are also due to the developers of Wasmtime for kindly answering our questions regarding their runtime and libraries.

Contents

Chapter 1

Introduction

WebAssembly, also known as Wasm, is a new technology that first appeared in March 2017 and was later recognized as an open standard in December 2019 for creating language-agnostic computer programs designed for running mainly in modern web browsers but also on other platforms.

The primary purpose of Wasm is to be an efficient compilation target for programs written in other languages [19]. By utilizing Wasm, developers can write code in various programming languages, compile it into WebAssembly, and then deliver it to a supported platform, whether it be a laptop, a server, or even a "smart refrigerator." Furthermore, Wasm also offers other compelling advantages such as portability, security, size- and load-time-efficient binary format, and a near-native execution speed.

As the idea of Wasm continues to be a trend in the web development community, due to its design and number of advantages it provides [17], we have been starting to see more active use of this technology in non-Web environments as well (e.g., the way Shopify utilizes it [23]). Wasm is especially appealing outside the browser because it is a fast, scalable, and safe way to run the same code across various computer platforms. However, another often overlooked benefit that can be gained by utilizing Wasm is *diversity*, which is one of our main focuses in this thesis.

Finally, it is fundamental to mention that we have focused ourselves specifically on distributed systems, where Wasm, in conjunction with gRPC, a high-performance, open-source universal RPC framework, can be used to enhance software diversity.

Diversity and Wasm

One of the biggest motivations to utilize a distributed system is its "fault tolerance" nature or the ability to handle any fault. In the field of Distributed

Systems, this is one of the most widely studied topics, which has remained a hot topic for a multitude of reasons (e.g., [9], [3]). When dealing with a distributed ecosystem containing dozens or even thousands of machines, some will inevitably fail. Thankfully, due to distributed systems' characteristics, faults like a system crash on one server will not affect other servers. However, while simply utilizing a well-thought distributed system will most likely increase reliability regarding potential system halts due to hardware faults, it does not grant inherent protection against software faults of both accidental and malicious nature. That is where Wasm's use can be significant, thanks to the diversity that it can bring to these types of applications.

Realistically speaking, the probability of simultaneous attacks on various components of a distributed system cannot be discounted. If multiple components have the same vulnerabilities, they can all be compromised by a single attack. That is where diversity and Wasm come to play. With Wasm, the idea of having multiple components that perform the same functions, but that use different software, can be made a reality through embedding. Diversification can be used to reduce the likelihood of common vulnerabilities (for a more in-depth read about software and OS-diversity, refer to [21] [20]). Hence, through utilization of diverse software, we can improve security, which can be done more cost-efficiently than developing software variants.

Our contribution

This thesis presents a distributed application template that utilizes Wasm+gRPC to increase software diversity. Moreover, we also ran various benchmarks to evaluate the application template's usability and performance, and generally, the compatibility of these two technologies. Furthermore, all code is available on Github [1].

Ideally, projects following this template would offer an environment where applications written in various languages and powered by WebAssembly will run intuitively, performantly, and cost-efficiently.

The efficiency aspect comes from the fact that this type of application takes full advantage of gRPC and Wasm. While the gRPC framework gives us the ability to freely choose what programming languages to use for our clients and servers, the use of Wasm allows us to have the same freedom of choice regarding the programming language that would be used for the application logic.

We have developed various function templates that aim to either increase the usability of this type of application or offer a standardized way to interact

[1] Repository's link: https://github.com/AndreaEsposit/bachelors-thesis.

with WebAssembly modules.

This template's performance and usability analysis was done by evaluating two Wasm-based distributed applications of our creation. Through this analysis, we learned about Wasm's drawbacks, interesting functionalities, and most distinctly, about how performant Wasm is in such applications. The analysis was done for multiple programming languages, giving us a better understanding of how complex it is to utilize Wasm in various environments.

By utilizing a set of numerous benchmarks for each implementation, we have shown that WebAssembly runs particularly well when embedded in programming languages such as Golang and C#, while not performing well in languages such as Rust and Python. Unfortunately, our performance analysis suggests that Wasm possesses, as of early 2021, some flaws in terms of stability and, more specifically, in performance. Our findings imply that Wasm will, in some cases, specifically when an enormous amount of gRPC requests need to be processed, perform very poorly, suggesting that fewer requests with a bigger size might be preferable when possible. Fortunately, these performance problems will most likely be ironed out in the future, as Wasm keeps getting consistent features and performance updates.

Finally, based on our results, we can conclude that Wasm is already a good option for projects seeking to increase diversity and, therefore, security at the expense of some performance.

1.1 Outline

The thesis is outlined as follows:

Chapter 2 introduces the reader to important terminologies, processes and technologies used throughout the thesis.

Chapter 3 goes over a template of how Wasm can be combined with gRPC to increase diversity in distributed applications.

Chapter 4 consists of the implementation of a simple echo server and presents the results and evaluates the test data.

Chapter 5 consists of the implementations and presentations of the benchmark results of a storage application that follows our Wasm+gRPC template.

Chapter 6 concludes the thesis, discusses about shortcomings of the topic and further work.

Chapter 2

Background

This chapter introduces important terminologies, concepts, general processes, Wasm's limitations, and tools used throughout the thesis. First, we discuss Wasm beyond the browser, talking specifically about WASI, Wasm runtimes, and the instantiation process of a WebAssembly module. Subsequentially, we present gRPC, protocol buffers, and the general gRPC startup process. Finally, we briefly introduce a ready-done benchmarking tool used in this thesis.

2.1 Wasm beyond browser

Although most WebAssembly applications today are browser centric, WebAssembly itself has a lot of potential in other environments as well [1], such as on servers, on IoT devices, mobile/desktop apps or even embedded within another larger program.

The main advantages of Wasm remain essentially the same outside of the web:

- **Portability**: the ability of running the same code across a multitude of machines, as long as there is a supported Wasm runtime for those system.

- **Security**: Wasm executes within a sandboxed stack-environment, this means that the code cannot talk directly to the OS, and relays upon explicit imports to allow communication to with the host.

Nevertheless, Wasm by itself has some significant limitations, such as the inability to converse with the host system, or in other words, the lack of a built-in system interface. This limitation was solved with the introduction of **WASI** to the WebAssembly platform in 2019.

However, WASI was not designed to solve all of Wasm's problems, which means that a number of issues, aside from those that will be resolved once WASI's development is complete (e.g., network connectivity), will need to be addressed in other ways.

Fortunately, numerous developers are actively working on expanding Wasm's capabilities and addressing some of its other faults via Wasm's proposals(here is a handy rundown of all the active proposals: [8]). Thanks to these proposals, features like garbage collection, multi-threading, and multi-memory should soon be available.

2.1.1 WASI

WASI [5], the WebAssembly System Interface is an API designed to standardize the sandboxed execution of WebAssembly modules in non-web environments. Specifically, WASI provides a standard way for Wasm modules to interface with host runtimes and get access to several operating-system-like features, including files and filesystems, clocks, and random numbers.

The way WASI works is straightforward. The first step is to write an application in any preferred language. The written application is then built and compiled into WebAssembly targeting the WASI environment. It will generate a binary that requires a particular runtime to execute. The runtime of choice (e.g., Wasmtime, Wasmer) provides the necessary interfaces to the system calls.

Finally, we believe it is worth discussing the state of WASI as a compilation target, which has gone through numerous iterations due to the continuous WASI development.

The WASI compilation target also referred to as the "wasm32-wasi" target, is a new and still experimental target. Due to WASI's unfinished development state, the compile target is still considered (as of Febrary 2021) to be in its preview phase, which is a state that is unlikely to change until WASI's end of development. Such modules can be run directly in CLI runtimes (e.g., Wasmtime) or embedded in other languages utilizing Wasm-runtime libraries.

Due to the number of Wasm proposals, different highly experimental WASI targets are available, giving a possible glimpse at WASI's future functionalities. Unfortunately, using experimental WASI targets and experimental "flags" does not always result in full functionality, resulting in possible unsatisfactory performance and/or broken functionalities.

2.1.2 Wasm's role in gRPC servers

As of January 2021, WASI has some severe limitations regarding network applications. Despite the availability of a few methods for working with web sockets, such as *sock_recv*, *sock_send* and *sock_shutdown*, the current version of WASI is nowhere near able to offer complete and satisfactory network support. Due to this limitation, and our desire to take advantage of improved security (due to diversity) that comes when embedding Wasm, we have determined that the best possible solution for utilizing Wasm+gRPC is for Wasm to handle only the server application logic. This choice means that the hosting environment (e.g., Go) serves as the dedicated gRPC server, which will forward each request to the Wasm module. Figure 2.1 illustrates how a generic Wasm-gRPC server is going to work.

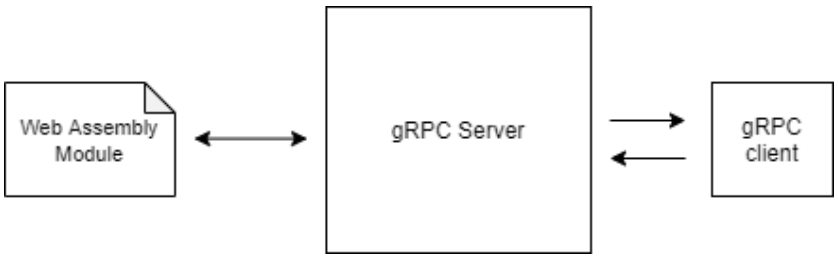

Figure 2.1: Generic Wasm-gRPC server

2.1.3 Wasm Restrictions

Rust will be used as our programming language of choice for our applications' logic. It is worth noting that this choice has not been left to chance but has been influenced by a necessity that we will explain here in detail.

As mentioned in Section 2.1, Wasm is executed in a sandboxed environment, which means that a module can declare and use its variables but cannot access anything outside its environment.

This sandboxed nature is one of Wasm's most significant selling points but is also why we are so limited when interacting with a Wasm module. Nonetheless, there are two main methods to communicate with an instantiated WebAssembly module:

- The first method, which is also the most straightforward one, is to use the arguments and return values of the module's imported functions. Although this seems to be a good approach, our options are heavily

limited since we can only exchange fundamental WebAssembly data types, such as i32, i64, f32, f64 (integers and floats), and we cannot return multiple values. Of course, these are significant limitations that will be addressed in the future, but for now, workarounds are required to use Wasm effectively in non-web environments.

- The second method is more of a workaround than anything else. While we wait for the interface-types proposal [7] to be implemented, the best solution is to use the raw WebAssembly memory, which means that we would be directly copying objects to and retrieving objects from Wasm-memory. Unfortunately, this method is inherently unsafe to use since allocation and deallocation of memory must be done manually. However, by being careful, we can use this workaround to pass non-fundamental data types like strings, arrays, and serialized data like JSON and protocol buffers between runtimes and Wasm-instances.

Realistically, we will be using a combination of these two methods. We will pass pointers as arguments and return values, using Wasm's memory to copy the data to and from those pointers.

However, because of the need to write data to specific pointers, memory must be managed manually within the module. This requirement, implies that while utilizing this method, the programming language to be compiled to Wasm needs to have manual memory management, which means that programming languages like Go cannot be used. In contrast, languages like Rust and C are ideal for such tasks since they offer manual memory management.

2.1.4 Wasm Runtimes

Each runtime does the same task, usually containing the same basic features. However, they all differ in performance, and each of them has some specific extra characteristic that makes it different from the others. For this project, we will utilize Wasmtime as our main runtime, mainly due to its considerably bigger online community, which led it to be one of the most utilized runtimes as of early 2021.

Wasmtime

Wasmtime [2] is an efficient, compact solution for working with the latest WASI/WebAssembly innovations. It can be embedded in a selection of programming languages, such as Python, Golang and Rust. Below is a list of key Wasmtime features:

- **Compactness:** Non-demanding standalone that can be scaled up as needs grow. Can work with small chips or be utilized with massive servers. Embeddable on almost any app.

- **Tweakablity:** Tweak wasmtime in advance for pre-compilation, generate light-speed code using Lightbeam, or interpret at runtime. Configurable for whatever you need Wasm to accomplish.

- **Speed:** Built based on Cranelift; perform high-quality runtime machine coding.

- **WASI-compatibility:** Supports a set of APIs, allowing you to implement alongside the host via the WASI interface.

- **Support:** Compliant with WebAssembly test suite standards with a large base of developer and contributor community support.

2.1.5 WebAssembly's module instantiation

Understanding a language-specific Wasm instantiation process (utilizing Wasmtime) can be somewhat daunting in the beginning, mainly due to some structural differences and dissimilarities regarding the names of classes and functions utilized in this process. However, the underlying instantiation process is principally the same in every programming language.

Fortunately for us, some fundamental classes retain their names from one Wasmtime version (e.g., GO, Python) to the other, making it much simpler to understand how to set up the instantiation process in a new programming language.

These classes are the `Store`, `Engine`, `Module`, `WasiConfig`, and `Instance` classes. Here is a short rundown of what these classes are supposed to do for a better understanding of the process (refer to the official Wasmtime libraries docs for an in-depth read, e.g. : [13] [12]):

`Store`: This is a general group of WebAssembly instances and host-defined items. Many WeAssembly objects, such as instances, functions, and globals, will be attached to and refer to a `Store`. Furthermore, instances are created by instantiating a `Module` within a `Store`.

`Engine`: This is an instance of a Wasmtime engine that is utilized to create a `Store`. This object is at its most stable when utilizing default configurations. However, we can also modify its behavior by utilizing a `Config` object at its creation, allowing us to enable experimental

Wasm proposals if we please (these are usually unstable and require the utilization of special Wasm modules).

`Module`: A `Module` is an in-memory representation of a WebAssembly binary's input. It is used in the instantiation process to construct an `Instance`. Nevertheless, the `Module` object is just a code representation of a Wasm module, which means that it cannot be used to interact with the Wasm module directly.

`WasiConfig`: It is an object that keeps track of all custom WASI configurations that we want to specify.

`Instance`: It is an instantiated module instance that can invoke exported functions and interact with the Wasm module's memory.

`Memory`: The runtime representation of a linear memory is called a `Memory` instance. This object stores a vector of bytes(linear representation of memory) and maximum data size if such maximum value has been previously defined [16].

Thanks to these short definitions, we can now use these terms to go over the general steps that need to be taken to embed and instantiate a WebAssemly module with WASI functionalities.

1. We start by creating a new Wasmtime `Engine`, either using the default configurations or using custom-made ones.
2. We create a `Store`-object from the configuration provided by the `Engine`.
3. We continue by creating a new `WasiConfig` object and then customizing it (e.g., specifying which device directory our module has access to).
4. We instantiate a WASI-object, usually referred to as a `WasiInstance` object or a wrapper class, such as the `Host` class in .NET. This class represents the WASI configurations used when creating an `Instance` object. It is crucial to keep in mind that the WASI version that needs to be specified when creating this object needs to match the version used as compiling target for the WebAssembly module that we will use. If we do not specify the correct version, we might encounter compatibility issues and/or loss in performance.

5. Subsequently we create a `Module` object, which is an in-memory representation of the WebAssembly module that we want to embed. Since creating a `Module` currently involves compiling code, it is crucial to notice that this step can be an expensive operation.

6. Finally, we can create the `Instance` object by utilizing the WASI-object and the `Module`-object. Furthermore, utilizing this Instance object, we can utilize the exported module's functions and access a linear representation of the module's memory.

The process, however, becomes much more straightforward if there is no need to utilize WASI, permitting us to skip steps 3 and 4 entirely.

2.2 gRPC and protocol buffers

2.2.1 gRPC

gRPC [14] stands for *Remote Procedure Call*. It is a modern RPC framework developed by Google and is based on the client-server architecture. Client is able to call methods from a server as if they were local methods. In other words, it allows programs to execute a procedure of another program from a different platform. The thing about gRPC is that developers has no need to explicitly code the details for the remote interaction. This is automatically handled by the underlying framework transparently. gRPC uses HTTP/2 as it's underlying transport protocol, which results in high-performance.

Furthermore, gRPC supports several programming languages, both officially and unofficially, this means of course that our clients and servers can be written in many different languages and will still continue to work.

2.2.2 Protocol Buffers

Protocol buffers [11] are a method of serializing data, which can be transmitted or be stored in files. In comparison to other formats such as JSON or XML, protocol buffers are known for being smaller, faster and simpler. They are made to be optimized for transmitting data between multiple micro-services in a language- and platform-neutral way.

2.2.3 Setting up a gRPC server

The general setup method for a gRPC server is similar in most programming languages and usually follows these basic steps:

1. Define a service in a `.proto` file.
2. Generate server and client code in the programming language of choice, which is done utilizing a protocol buffer compiler.
3. Use the X(programming language of choice) gRPC API to write a client and server for the service.

2.3 Utilities

2.3.1 ghz - gRPC benchmarking and load testing tool

ghz is a command line utility and Go package used for load testing and for benchmarking gRPC services [10]. This tool has many useful features such as the ability of using proto files, pre-built protoset bundles or even server reflection. Moreover, it allows you to test unary, streaming and duplex calls using JSON or binary data, as well as giving you the choice regarding how many clients and messages will be used for the benchmarking process.

Chapter 3

Our Wasm+gRPC template

This chapter will go over the general steps needed to create a Wasm+gRPC template that can quickly and efficiently be utilized to create an application for developers aiming to increase diversity thus, security in distributed applications. Ideally speaking, by following our template, a developer should be able to write highly reusable code that could be used in a variety of distributed applications. Utilizing this methodology would additionally also speed up development time.

It is worth noting that we are not focusing on any specific programming language, which means that we are going explain in a language-agnostic manner. Furthermore, Wasmtime is the runtime of choice for Wasm embedding.

3.1 Preamble

When interacting with a Wasm module from a host environment, there are not many types of formats that can be utilized for communication since there cannot be any assumptions about the host environment, which means that regular language-specific objects cannot be utilized. As a result of this constraint, the best solution is to utilize well-known and largely supported serialization formats, such as JSON, XML, TOML, and Protocol Buffers. However, since we are already utilizing protocol buffers for client-server communication, we think it would be logical and convenient to do the same for host-to-Wasm communication. This decision implies that we will need to utilize the same proto file to generate code for the server, the clients, and the Wasm-module. Using this methodology, we can quickly produce code for inter-language communication that is performantly efficient and, most importantly, easy to generate. Furthermore, through the Protobuf WellKnownTypes, we have access to a growing amount of useful predefined types of message fields,

which can be very helpful in several situations, such as the Timestamp-field, often utilized for logging.

3.2 Wasm integration in a gRPC server

Once we are done with the Wasm instantiation process (explained in-depth in subsection 2.1.5), we have a few options for working with the instantiated module's memory and its exported functions. We recommend defining and storing these exported functions in advance using Wasmtime-function objects, primarily to avoid having to re-define them any time we want to use them, thus minimizing unnecessary overhead. Furthermore, for the same performance reasons, the module's memory object should also be preemptively created and stored for future reuse.

This definition of exported Wasm objects should be done directly after the Wasm Instantiation process, and we advise using a `"[string]WasmFunction"` map to store the exported functions efficiently.

Finally, these exported objects should be stored either in a custom WasmContainer-object or directly in the base Server/Service class, which we obtain when generating code for the gRPC server. Utilizing the latter storage system, we will have direct access to these exported objects inside the methods that our gRPC server implements.

3.3 The WasmInstantiate function

On accounting for the significant library-specific differences that we encountered while working with Wasm in various programming languages, we have concluded that creating a helper instantiation function would be highly beneficial for others that aim to diversify distributed applications.

The `WasmInstantiate` function template aims to offer a highly reusable function capable of adjusting to many different applications. Furthermore, this template will also serve as a founding stone for future more complex projects that will utilize new Wasm features. However, it is essential to note that the structure of this function may need changes and adjustments as the framework utilized by Wasmtime and other runtimes like Wasmer might change in the future.

Utilizing this template would allow inexperienced developers to immediately dive into the Wasm embedding world without needing to find their way around language-specific differences in the Wasm instantiation processes. The arguments of this function should be the following:

- `.wasm` file location, utilized when creating the Module object
- A string array containing all the functions that we intend to export.
- A bunch of Wasi specific arguments, such as:
  - A list of directories (and their aliases) to which the Wasm module should have access.
  - The stdout/stdin/stderr files of choice for the Wasm module.

Moreover, this function should return a map containing all the exported Wasm functions and the linear representation of the module's memory, which can then be, as we have discussed in the previous section, stored somewhere for future use. By having an array containing all the functions we intend to export, we can easily define and store the exported functions we need in a loop. Here is a code snippet of how that part of the code would look like:

```
1 for functionName in functionsToExport{
2     instanceExports[functionName] = instance.exports[functionName]
3 }
```

Listing 3.1: Storing exported functions

Finally, it is worth mentioning that we intend to go over some of our implementations of the `WasmInstantiate` function during the following chapters. Furthermore, the entire code of said functions is available in Appendix C.

3.4 The callWasm function

Due to the need to utilize serialized data for communication between a host environment and Wasm, we have developed a function template that should always be utilized to safely and reliably interact with a Wasm module. The following list is a general rundown of the steps that must be taken when interacting with a Wasm module from the host environment's perspective:

1. *Set aside a portion of Wasm-memory for the message we intend to pass to the Wasm Module (Memory allocation).* This step is crucial because skipping it could lead to various errors, including nonvoluntary overwriting of previously stored data or even out-of-bounds errors in the case where the message size exceeds the size of the Wasm-memory.

   - This step should be done directly in the Wasm module by defining an allocation function and invoke it from the host environment.

2. *Copy the serialized message into the linear representation of the Wasm module.* This data should be explicitly copied into the portion of Wasm-memory that we have previously allocated.

3. *Invoke whatever Wasm exported function we want to invoke.* Subsequentially, in the Wasm function itself, make sure to store the function's response, in the form of a serialized message, directly into the Wasm-memory. As previously mentioned in Section 2.1.3, this needs to be done since there is no way to return the response message to the host environment directly (as of early 2021).

4. We can now *retrieve the response serialized message* from the host environment's linear Wasm-memory representation.

The steps above would typically work with every type of request. However, it is essential to keep in mind that we are not taking care of memory deallocation, which will lead to a memory leak in the long run. To avoid such a problem, we advise deallocating the portions of Wasm-memory utilized for requests and responses. This memory deallocation should be done after retrieving the answer message from Wasm-memory. Doing it sooner would lead to deleting the response message before retrieving it, resulting in an error. Finally, as we have explicitly stated in Section 3.1, we think that proto buffers should be utilized as a serialization format, but this is in no way mandatory.

Chapter 4

Simple Echo server

This chapter introduces a practical use of Wasm in a gRPC Echo server application, which is usually used to test if a connection between a client and a server is successful. It consists of a server that sends the client back whatever data the client had previously sent to the server.

Furthermore, the aim of this chapter is to gain an understanding of the challenges of using Wasm and gRPC together, as well as how a gRPC server performance degrades when utilizing Wasm.

We will start by briefly going through the proto-definition of the message we used for this application before going in-depth over the steps we took to implement the application itself.

4.1 Proto Definitions

Each gRPC service is generated from one protocol buffer service definition. In the case of an Echo server, we only must define one service which contains an RPC method that sends an echo message in the protocol buffer file. Messages are also defined alongside the service definitions. Compiling the `.proto` file in our chosen language will generate service interface code and stubs (for some languages, the preferred term is client) specific to the language used. After generating the API, each service and method must be implemented in the server-side.

Furthermore it is essential to be consistent when defining services and messages definitions to avoid possible errors when implementing the gRPC server and client.

In Listing 4.1, we have chosen to use `echo_message` as both our request and response message-type, as there is no difference between a response and a request message in this type of application.

```
message echo_message {
    string content = 1;
}

service Echo{
    rpc Send(echo_message) returns(echo_message);
}
```

Listing 4.1: Echo server's service and message definition

Additionally, as we intend to utilize proto-messages to communicate between the host environment and the Wasm module, we need to utilize this same `.proto` file to generate code that will allow us to work with the `echo_message` on the application-logic side of the server (Wasm module). We generate the needed code by utilizing the `protoc-rust` crate (packages in Rust), which will generate Rust code with structs and methods derived from the `.proto` file.

4.2 Application logic

As previously mentioned in Section 2.1.3, we have decided to utilize Rust as our programming language of choice to write the application's logic of our projects. Practically speaking, this is an excellent choice for many reasons that we will briefly list here below:

- Rust provides explicit control over where and how memory is allocated and deallocated.
- Thanks to Cargo (Rust package manager), compiling to Wasm/WASI is pretty straightforward.

We will now go through the practical steps we took to write our server logic, in addition to the process taken to compile to WASI.

We start by specifying that we are building a `cdylib` in `Cargo.toml`, which will enable us to share our Echo-Wasm "library" of functions with other languages.

Subsequently, we can begin writing our code. The first things that need to be defined are the memory allocation and deallocation functions, which are essential to pass non-fundamental data types, such as protobuf messages, to our WebAssembly module. These functions can be easily defined using the globally configured allocator from the standard Rust library or any other allocator. Listing 4.2 presents these functions:

```rust
use std::alloc::{alloc, dealloc, Layout};

#[no_mangle]
pub unsafe fn new_alloc(length: usize) -> *mut u8 {
    let align = std::mem::align_of::<usize>();
    let layout = Layout::from_size_align_unchecked(length, align);
    alloc(layout)
}

#[no_mangle]
pub unsafe fn new_dealloc(ptr: *mut u8, length: usize) {
    let align = std::mem::align_of::<usize>();
    let layout = Layout::from_size_align_unchecked(length, align);
    dealloc(ptr, layout);
}
```

Listing 4.2: new_alloc and new_dealloc functions in Rust

As we can see, the allocation function needs to know (its argument) the size of the message we intend to copy on the Wasm-memory, and it returns the pointer of the portion of memory allocated. The deallocation function simply deallocates a defined amount of memory based on a pointer (an address) and a length value. The latter represents how much memory needs to be deallocated, while the former indicates its location.

From here on out, following what we said in Section 3.4 about memory allocation and deallocation, we will be assuming that memory management is correctly being taken care of by the host environment each time it invokes the echo function.

The `echo` function is the primary function of this application, and it is an excellent example of how a gRPC-Wasm function should behave. This function has the length of a serialized protobuf message and its pointer as arguments and has a pointer of a serialized response protobuf message as its response value. Unfortunately, as we have already mentioned multiple times in the previous sections, Wasm modules are (as of early 2021) unable to return multiple values, which means that we cannot return both the pointer and the size of the serialized response message to the host. To circumvent this problem, we have decided only to return the pointer of the response message while saving its length into memory, which means that by utilizing a simple `get_len()` function (visible in Listing 4.4) , we can retrieve the response's length. Moreover, due to this function's simplistic nature, there is no need for memory allocation or deallocation since we will only return an integer to the host environment. Here is a code snippet for the `echo` function:

```rust
#[no_mangle]
pub extern "C" fn echo(ptr: *mut u8, length: usize) -> *mut u8 {
```

```
3       let slice = unsafe { std::slice::from_raw_parts(ptr, length) };
4
5       let recived_message: echo_message = protobuf::Message::parse_from_bytes(
        slice).unwrap();
6
7       let mut new_message: echo_message = protobuf::Message::new();
8       new_message.set_content(recived_message.get_content().into());
9
10      println!(
11          "Wasm has recived this message: {:?}, sending it back!",
12          recived_message.get_content()
13      );
14
15      let mut new_bytes = protobuf::Message::write_to_bytes(&new_message).
        unwrap();
16      unsafe {
17          MESSAGE_LEN = new_bytes.len() as i32;
18      }
19      let new_ptr = new_bytes.as_mut_ptr();
20
21      std::mem::forget(new_bytes);
22      new_ptr
23  }
```

Listing 4.3: Echo function in Rust

It should be noticed that the use of static mutable variables, like the one that we use to store the response's length, is inherently unsafe. However, since Wasmtime does not yet implement the Wasm thread proposal, this solution can be utilized as our Wasm module will only be accessed by one thread at a time. This workaround, needless to say, will not be viable anymore once the wasm multi-threaded proposal is implemented as utilizing it would mean incurring into race conditions.

In Listing 4.4, we can see the `get_len()` function and the static mutable variable `MESSAGE_LEN` utilized to store the lengths of responses' sizes.

```
1  static mut MESSAGE_LEN: i32 = 0;
2
3  #[no_mangle]
4  pub extern "C" fn get_message_len() -> i32 {
5      unsafe { MESSAGE_LEN }
6  }
```

Listing 4.4: get_len function

Finally, compiling to WASI is a simple task in Rust. By utilizing `cargo-wasi` [15], a Cargo subcommand to build code for the `wasm32wasi` target, we can execute the command `cargo wasi build` to generate our `.wasm` file.

The WebAssembly module is now ready to be embedded within the gRPC server using Wasmtime.

4.3 Client

The gRPC client for an echo server is as simple as it can be. We created a client which contains a method that takes in a user input. Clients work as gRPC stubs and use the gRPC-API auto-generated *Send* method. In practice, the only thing that the client needs to do is call the `Send` function and forward the user input to the server.

In order for the client to connect to the server, a channel needs to be established. This channel will then be subsequently used to instantiate a stub with it. Moreover, when establishing a connection, the dial options for the communication must be specified. We used the dial-option `grpc.insecure_channel` for simplicity purposes and make it possible to connect without any transport security protocols such as *Transport Security Layer* or the deprecate *Secure Sockets Layer*. After a stub has been created, the last thing to do is to call the method on the stub.

4.4 Server

The server side implements the generated interfaces and handles the calls of a client. Multiple servers can be initialized on the same machine. However, a unique port must be assigned for each server to run simultaneously.

4.4.1 Embedding the WASI module in Golang

For the server-side of this echo application, we have decided to utilize Golang as our host runtime environment. This decision was dictated by our belief that Golang would play well in a gRPC server setting, considering that it is well known to be a programming language that tackles concurrency performantly.

The embedding process is just a practical application of what we have already gone over in Section 2.1.5. The only particularity with this specific embedding process is that we have specified the `StdoutFile` in the WASI configurations, which was done to see print statements from the `echo` function.

Below in Listing 4.5, we can notice that a `Linker` object is needed to link the WASI object and the `Module` object. Moreover, this object is not something present in all Wasmtime libraries but is specifically required in this version and few others. What `linker.DefineWasi(wasi)` essentially does is ensuring that all exported functions are available for linking. The linking itself does not happen, however, until we invoke the `linkerİnstatiate` function, which returns an `Instance` object (refer to Section 2.1.5).

```
1  func main() {
2    dir, err := ioutil.TempDir("", "out")
3    check(err)
4    defer os.RemoveAll(dir)
5    stdoutPath := filepath.Join(dir, "stdout")
6
7    engine := wasmtime.NewEngine()
8    store := wasmtime.NewStore(engine)
9    linker := wasmtime.NewLinker(store)
10
11   wasiConfig := wasmtime.NewWasiConfig()
12   wasiConfig.SetStdoutFile(stdoutPath)
13
14   wasi, err := wasmtime.NewWasiInstance(store, wasiConfig, "
       wasi_snapshot_preview1")
15   check(err)
16
17   err = linker.DefineWasi(wasi)
18   check(err)
19
20   module, err := wasmtime.NewModuleFromFile(store.Engine, "../wasm/
       echo_server.wasm")
21   check(err)
22   instance, err := linker.Instantiate(module)
23   check(err)
24   }
```

Listing 4.5: Wasmtime Embedding for Go

After the instantiation process, we store a map of Wasm exported functions (in this case: `echo`, `new_alloc`, `new_dealloc`, and `get_len`) and the WasmMemory inside the autogenerated `EchoServer` class. By utilizing this method, we can take hold of those objects when we receive gRPC requests.

4.4.2 Server-side: Send Function

We decided that since the Echo server only provides the echo service, there was no need to build a generic `callWasm` function like the one we mentioned in Section 3.4. Instead, we wrote code directly within the autogenerated `Send` function. The `Send` function, depicted in Listing 4.6 receives every gRPC request for this specific server. The function itself is mainly based on the utilization of pointers to pass and retrieve data from and to the instantiated Wasm Module (`Instance` object). Moreover, we can see that since WebAssembly only accepts integers and floating points when interacting with exported functions, we need to convert our integers to supported data types. In this case `int32`. Additionally, the pointers returned by the exported functions also need to be cast to `int32`, as those value are by default interpreted as `interface{}` objects.

```
1  func (server *EchoServer) Send(ctx context.Context, message *pb.EchoMessage)
       (*pb.EchoMessage, error) {
2      fmt.Printf("Server recived: '%v'\n", message.Content)
3    receivedBytes, err := proto.Marshal(message)
4    check(err)
5
6      server.mu.Lock()
7    defer server.mu.Unlock()
8
9    ptr := server.copyToMemory(receivedBytes)
10
11   newPtr, err := server.funcs["echo"].Call(ptr, int32(len(receivedBytes)))
12   check(err)
13   newPtr32 := newPtr.(int32)
14
15   nml, err := server.funcs["get_len"].Call()
16   check(err)
17   newMessageLen := nml.(int32)
18
19     buf := server.memory.UnsafeData()
20
21
22   returnMessage := &pb.EchoMessage{}
23   if err := proto.Unmarshal(buf[newPtr32:newPtr32+newMessageLen],
       returnMessage); err != nil {
24     log.Fatalln("Failed to parse message: ", err)
25   }
26
27   _, err = server.funcs["dealloc"].Call(ptr, int32(len(recivedBytes)))
28   _, err = server.funcs["dealloc"].Call(newPtr32, newMessageLen)
29
30   return returnMessage, nil
31 }
```

Listing 4.6: Echo Server's Send RPC handler

At the end of the function, the exported *dealloc* method is called upon to release the memory currently allocated in the Wasm module.

Notably, as we can see in this function, we utilize locks to assure that only one client at the time has access to the Wasm functions, as not doing so would result in errors.

4.5 Benchmarking

The goal for this benchmark is to evaluate the application's performance, and with this we mean: *latency* and *throughput*. Both of these are important to test as they have an effect on how well the gRPC application is working. We have performed several tests to gauge the application's overall performance, both with and without Wasm. This was done by utilizing two of the Pitter machines located in the Linux Lab at the University of Stavanger. These Pitter machines are a group of GNU/Linux machines connected together over a LAN, and represent normal desktop computers. The specifications of these machines are specified in Table 4.1.

CPU	Intel® Core™2 Duo Processor E6300
RAM	2 GB (DDR3)
Network	Ethernet, 1 Gb/s
OS	Scientific Linux 7.9 (Nitrogen)
Go ver.	1.16 linux/amd64

Table 4.1: Specifications of the Pitter machines

In the benchmark we send large amount of requests, while gradually increasing the number of clients. The large numbers of requests and the increasing number of clients is beneficial as we assume that the benchmarks will show significant results when stressed.

Additionally, we created a simple automation script to work alongside ghz for a more efficient benchmarking process. The script uses the following CLI command x-number of times:

```
1  ghz --insecure --proto <proto_file_location> --call proto.
    Echo.Send -c <n-clients> -n <n-messages> -d "\{"content":
    <data_string>\}" -O 'pretty' <server_IP_address>:50051.
```

This will generate a JSON file that will be then analyzed by the script to give us the average latency and throughput. Once all the x-number of benchmarks have been run, the script will generate another JSON file with the overview for each of benchmark and a final average between them all. This data will be utilized to create detailed graphs that show the differences in performance between the Wasm and Wasmless implementation of the application.

4.5.1 Performance comparison

Considering the purpose of the application, we opted to test for data sizes of 10 bytes, 1 KB and 10 KB to examine how well both implementations will perform with these sizes. Each implementation is executed 10 times, sending 100000 requests per data size.

Figures 4.1, 4.2 and 4.3 show each implementation's average latency and throughput measured for different data sizes. Already from the results, it is apparent that Wasmless performs exceptionally better than Wasm with smaller payloads. The figures show an apparent reduction in throughput for both implementations as the number of clients increases.

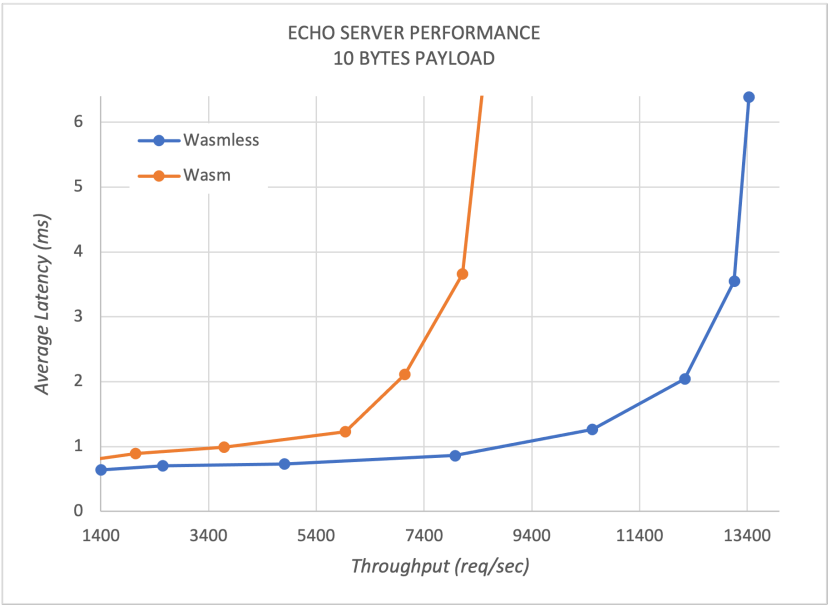

Figure 4.1: 100 000 requests. Payload 10 Bytes

The additional tasks that Wasm has to execute during an operation may explain the poor performance. By additional tasks, we mean allocation and deallocation of memory, alongside marshalling and unmarshalling of messages. It is clear that Wasm is substantially slower than the latter.

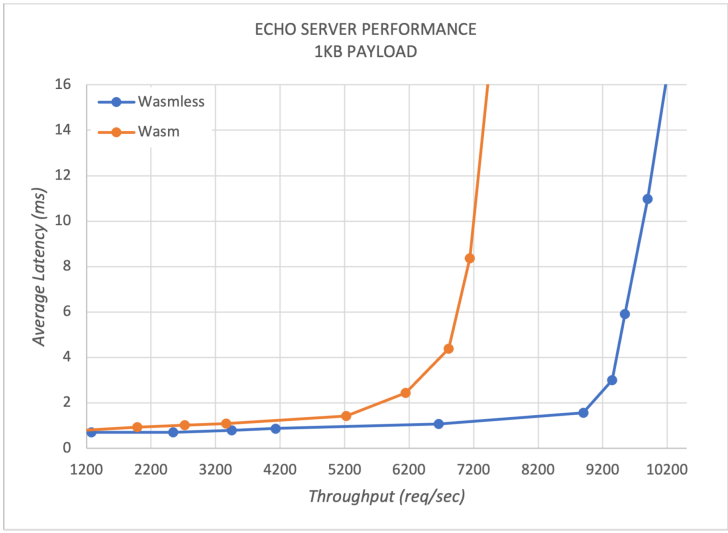

Figure 4.2: 100 000 requests. Payload 1 KB

Both Wasm and Wasmless displays a clear performance trend; throughput is reduced as the payload size becomes larger. The throughput is computed by taking the total number of requests, both successful and failed, and dividing by the total duration of the test, $T = count/total$. As latency increases, we can see that the application's throughput begins to suffer, resulting in a near improvement halt after a certain point and evident performance degradation.

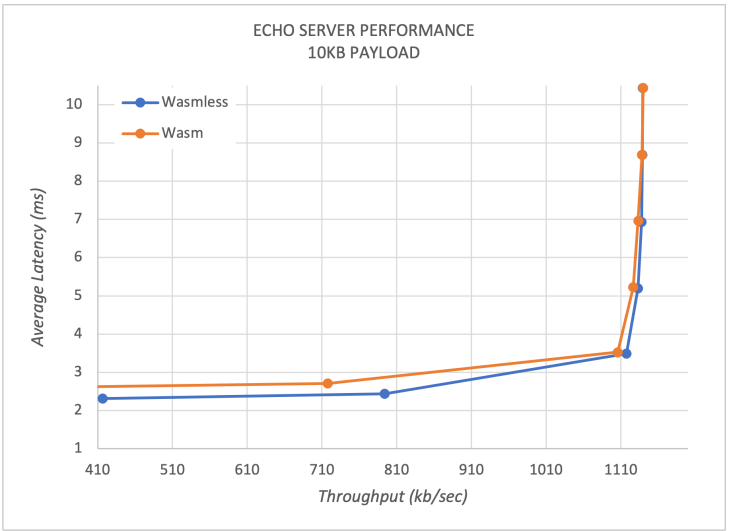

Figure 4.3: 100 000 requests. Payload 10 KB

The graphs for both implementations, depicted in Figure 4.3, are nearly identical in terms of latency and throughput measurements. For payload sizes of 10 KB or larger, it is evident that the overall performance is much worse than a smaller-sized payload. This poor performance might be due to a network bottleneck caused by the network we have utilized for these benchmarks, which was not particularly good (environment specifications in Table 4.1). Moreover, the share of network allotted to us was not always consistent, considering that we could not control the utilization of the network by other students.

Ultimately, an interesting thing that we can assume by these results is that the overhead we get when utilizing Wasm can be amortized with size. It means that instead of using a higher number of minor calls (size-wise), it would be wise to use a more prominent call (size-wise) to group the several small calls or straight up use only big calls. Doing so should get us less Wasm overhead on average.

Chapter 5

Storage Server

This chapter presents our Wasm-Storage application, which can be essentially thought of as an accurate and proper utilization of our Wasm+gRPC template to create a distributed system application that utilizes Wasm diversification to increase its security.

The main idea behind this application is to use multiple servers, each implemented in a different language, with the same WebAssembly module at the core of the storage service that they will provide.

By employing multiple servers, we can prevent having a single point of failure in the application. Moreover, using Wasm, the storage application can be conveniently embedded within various programming languages, thereby enhancing software diversification and, ideally, reducing the likelihood of common vulnerabilities among the various servers. Additionally, by utilizing multiple servers, the probability of simultaneous, not malicious, software failure and software bugs is decreased.

5.1 Proto Definitions

The application logic will operate on three servers written in different languages, which will be discussed further later in subsection 5.2.2. As this is a storage server, the functionalities will mainly be writing and reading files. The proto file will be used to generate the structural code necessary to build the Storage servers.

```
1 service Storage {
2   rpc Read(ReadRequest) returns (ReadResponse);
3   rpc Write(WriteRequest) returns (WriteResponse);
4 }
5
6 message WriteRequest {
```

```
7    string FileName = 1;
8    string Value = 2;
9    google.protobuf.Timestamp Timestamp = 3;
10 }
11
12 message ReadRequest { string FileName = 1;}
13
14 message WriteResponse { int32 Ok = 1;}
15
16 message ReadResponse{
17   string Value = 1;
18   int32 Ok = 2;
19   google.protobuf.Timestamp Timestamp = 3;
20 }
```

Listing 5.1: Storage server's service and message definitions

As we can see above in Listing 5.1, the `Storage` service definition implements `Read()` and `Write()` RPC methods. The `Write` method takes a `WriteRequest` as input, where each write request has a filename string, value string, and a timestamp for keeping records of when a file was created, updated, or deleted. The `Read` method on the other hand, takes a `ReadRequest` as input. This method returns a response of a value string, a timestamp, and also a boolean integer for stating if the reading of file(s) was successful or not.

5.1.1 Compiling process to WASI

The storage application logic is simple, it has a `store_data()` method that writes to a `.json` file and a `read_data()` method that reads from a file and returns its content, and of course the communication with the host runtime still happens through protobuf messages. It is worth noting that this application utilizes WASI to a greater extent than the previous one, because in the host environment we will be giving the Wasm instance access to a specified directory in our system. This is because the key idea in WASI is "capability-based security" meaning that access to system resources must be explicitly declared.

However, as of early 2021, there are some problems when you try to give resource access to WebAssembly modules compiled with the library method that we have previously used, and some workarounds need to be taken to make it work. We start by not making a Rust library, and writing our application logic directly in the *main.rs* file. Unfortunately, this new method results in either having to use a dummy "main function" or having to explicitly tell the compiler that the file does not have a main function, which can be done by writing `#![no_main]` on the top of the file. After that we can write our code exactly in the same style as we did with the echo server, treating it like a library. When it comes to the compiling process of our code to WASI, we will

use another more experimental compiling command, which will allow our new type of Wasm Module to work as expected. Here is the following command:

```
rustc --target wasm32-wasi -- -Z wasi-exec-model=reactor
```

With this command, we are targeting WASI with an unstable (Z flag) WASI execution model: the reactor model. The main concept of a reactor model in WASI is a module that does not have a regular start function that runs and exists, but that instead has a simpler `_initialize()` entry point. This entry point just does whatever low-level initialization the binary and WASI implementation needs (e.g. setting up a preloaded directory), and then exits, leaving everything set up by the `_initialize()` function in place. Using this model we are then able to just call whatever function the module exports just like a library. This is of course still experimental and unstable as of early 2021 and needs the latest nightly version of `rustc` (the rust compiler) to work.

5.2 Implementation

In this section, we first present the client. Afterwards, we take an in-depth look at how the servers are implemented. Our implementation of a storage server is largely derived from the echo server application that we've created.

5.2.1 Client-side

We implemented a CLI-based client which can interact with multiple storage servers. Implementing a GUI-based client was also an alternative; however, a CLI is a better option considering the advantages it provides, such as being lightweight and efficient. This client is intended to be a simple command-line interface, so users will not find it hard to interact with the servers. Our implementation lets us freely choose how many servers to connect to, which we can decide by simply adding and removing IP addresses from a string array. The program will connect to each server separately, forwarding all requests to every server. The program will wait for all servers' responses before allowing the user to send another request. In other words, this set of requests sent to different servers is treated as a single long request.

From a user's point of view, the application is pretty straightforward. It starts by greeting the user with a message that asks whether the user wants to write to a file (store data) or read an already existing file stored in the servers. According to the inputted command, the client will be required to

either provide the file name and value (content to store) or just the file name. Moreover, we implemented simple quality of life features, such as showing the user only the newest version of the storage file, by comparing every gRPC response's timestamps. Following a write-request, the user is notified if any gRPC connections were lost during the request process.

5.2.2 Server-side

Since the Storage servers are going to provide multiple services (Read/Write), unlike the Echo server, which provided only one service, we have decided that it would be beneficial to implement a generic `CallWasm` function, like the one we have mentioned in Section 3.4.

This `CallWasm` function would essentially be acting as our primary way of interaction with an instantiated Wasm module. Moreover, since we are going to implement such a function in every server, we can expect every server to resemble each other greatly, with, of course, some language-specific alterations.

When communicating with a Wasm module that has been instantiated, we must use locks, as we briefly discussed in subsection 4.4.2 . Locks are needed because of how we communicate with the Wasm instance, such as how we obtain the length of returned protobuf messages from exported functions, which is inherently not thread-safe.

Furthermore, not utilizing locks to access the Wasm module will result in errors when multiple threads try to do so. This problem happens because, as we previously stated in Section 4.2, Wasmtime does not yet implement multi-threading.

It is crucial to mention that the positioning of the locks will be the same in every implementation. This decision was taken to avoid situations where an implementation's performance would be greater than another purely because the former implementation has more code that can be executed in a multi-threaded way.

Additionally, utilizing locks, we assure that multiple writes and reads to/from a file(storage) are not concurrent, as it could result in inconsistent data.

Go implementation

The Go storage-server was the founding stone of all other storage implementations, mainly because of our more profound understanding, at this point, of how Wasmtime and gRPC worked in this programming language.

Firstly, we can start by saying that there are not many differences between the embedding process of the WebAssembly module here compared to the one done in the Echo server. The first difference is that instead of instantiating the Wasm module directly in the `main` function, we create this language version of the WasmInstantiate function, which can be seen in its entirety in Listing C.1. Secondly, considering that this application will utilize files as storage, we need to pass the directory access to the Wasm module before instantiating it, which can be done by modifying the WASI configurations before the instantiation occurs.

Subsequently, another difference is that we have to export and invoke the `_initialize` function, as Wasmtime will not be calling this function by default. This interaction with the `_initialize` function should be done in the Wasm instantiation phase.

Moreover, failing to call the `_initialize` function, or forgetting to do so, will make our instance incapable of loading the custom WASI configurations, which would result in it being unable to access the pre-opened directory of choice. Listing 5.2 shows how we specified the pre-opened directory (the "data" directory) and of how to export and call the `_initialize` function.

```
1 err = wasiConfig.PreopenDir("./data", ".")
2 check(err)
3
4 ....
5
6 intializeFunc := instance.GetExport("_initialize").Func()
7 _, err = intializeFunc.Call()
8 check(err)
```

Listing 5.2: Specification of the pre-opened directory and _initialize function

Now that we have gone over the new Wasm instantiation process, we can describe how we have implemented the `callWasm` function in Go.

`callWasm` is a function that should be utilized in every gRPC method, no matter what type of gRPC messages or gRPC service is being used. Furthermore, using the `protoMessage` interface, which is an interface that is implemented automatically by all types of generated proto messages, we can deliver a high degree of genericness.

This function handles all repetitive procedures that must be performed when communicating with an instantiated Wasm module, such as memory allocation and deallocation, and marshaling and unmarshaling of gRPC messages.

The first parameter in `callWasm` is the name of the Wasm function that we want to invoke in the form of a string, which will be used to obtain the

desired Wasm function from a map containing all exported Wasm functions of the Storage Server.

The second and third parameters are generic proto, but while the former is utilized to pass the gRPC request to the Wasm module, the latter acts as a shell of the future gRPC response.

This third argument is an empty gRPC response proto message either of type WriteResponse or ReadResponse, which will be used as the unmarshal-target for the response data we retrieve from the Wasm Memory. Moreover, this third argument will also be the returning value of the function once the unmarshalling process has been completed.

The reason why we need to have this third parameter is mainly because of Golang's lack of generics. The lack of generics results in the inability to:

- Specify the type of proto message to which the data we get from the WebAssembly linear memory representation should be unmarshalled.
- Specify what type of proto message we want the function to return.

Fortunately for us, the latter problem gets resolved by having the return value be a generic proto message; however, there is no simple way to solve the unmarshalling problem, which is why we need that third argument. In Listing 5.3 below, we can see the `callWasm` function in code format:

```
1  func (server *StorageServer) callWasm(fn string, requestMessage proto.
       Message, responseMessage proto.Message) proto.Message {
2    recivedBytes, err := proto.Marshal(requestMessage)
3    check(err)
4
5    server.mu.Lock()
6    defer server.mu.Unlock()
7
8    ptr := server.copyToMemory(recivedBytes)
9    len := int32(len(recivedBytes))
10
11   resPtr, err := server.funcs[fn].Call(ptr, len)
12   check(err)
13   resPtr32 := resPtr.(int32)
14
15   _, err = server.funcs["dealloc"].Call(ptr, len)
16   check(err)
17
18   resultLen, err := server.funcs["get_len"].Call()
19   check(err)
20   intResLen := resultLen.(int32)
21
22   buf := server.memory.UnsafeData()
23   if err := proto.Unmarshal(buf[resPtr32:resPtr32+intResLen],
       responseMessage); err != nil {
24     log.Fatalln("Failed to parse message: ", err)
25   }
26
```

```
27    _, err = server.funcs["dealloc"].Call(resPtr32, intResLen)
28    check(err)
29
30    return responseMessage
31  }
```

Listing 5.3: callWasm function in Golang

Inside of `callWasm`, we utilize the `copyToMemory` function, which in reality is simply a partition of the callWasm function that we have decided to create to make the function easier to read. The `copyToMemory` function handles the copying of serialized data to the Wasm's memory. The listing below shows the code for the `copyToMemory` function:

```
1  func (server *StorageServer) copyToMemory(data []byte) int32 {
2
3    ptr, err := server.funcs["alloc"].Call(int32(len(data)))
4    check(err)
5
6    ptr32 := ptr.(int32)
7
8    buf := server.memory.UnsafeData()
9    for i, v := range data {
10     buf[ptr32+int32(i)] = v
11   }
12   return ptr32
13 }
```

Listing 5.4: copyToMemory function in Golang

Finally, it is worth mentioning that we need to cast the returning `callWasm`'s value to the specific gRPC function response type before utilizing it to reply to the client.

Python implementation

We used Python for creating the second server, since it has great support on both gRPC and Wasmtime, just like in Go. This means that this implementation closely resembles the Go implementation, but with a few subtle differences.

The Wasm embedding process and the implementation of the `WasmIntantiate` function were straightforward, closely resembling what we did in Golang. One minor difference with Python's `WasmInstantiate` function is that the exported Wasm objects (functions and memory) are stored like global variables, instead of storing them inside the `Server` class.

Additionally, there are some more subtle diversities here and there that come in the form of some language-specific differences with the functions we

used. For example, in our `copy_memory` function, we do not have to specify what type of array it has to take in, unlike in Go where we have to specify that the method should take a byte array as an argument. What is great about Python is that it is able to automatically recognize what type of argument the function takes in. Subsequently, this makes the code smaller and cleaner, which can be seen in Listing 5.5.

```
def copy_to_memory(sdata: bytearray):
    ptr = instanceExports["alloc"](len(sdata))

    for i, v in enumerate(sdata):
        instanceExports["memory"].data_ptr[ptr + i] = v

    return ptr
```

Listing 5.5: copy_to_memory function in Python

As previously mentioned, WASM only works with data types of integers and floats. The function above returns a pointer that was of type i32 originally. However, unlike Go, where we have to be specific about which numerical type we want our pointer to be converted into, Python automatically converts any numerical int type (i32, i64..) to Python int. This shows how simple this interpreted language is compared to other programming languages.

Pointers in Python are not commonly used to store and manage allocated memory addresses, as they are in other languages such as Go, C, or Rust. Python is heavily focused on usability rather than on efficiency. This is one of the differences when working on the server implementation.

```
class StorageServicer(storage_pb2_grpc.StorageServicer):

    def Read(self, request, context):
        return_message = call_wasm(
            read, request, storage_pb2.ReadResponse())
        return return_message

    def Write(self, request, context):
        return_message = call_wasm(
            write, request, storage_pb2.WriteResponse())
        return return_message
```

Listing 5.6: StorageServicer

The `StorageServicer` class that we can see in Listing 5.6 extends the autogenerated StorageServicer class and implements the two functions needed by our service, Read and Write. Each function contains two parameters: request and context. The context parameter is not used here. However, it

is worth noting that the context variable contains various useful contextual data, such as timeout limits.

Finally, we can take a look at the `Server` class and at its `run` method utilized to run the gRPC server itself. Below is the code for the run function:

```
class Server:
    @staticmethod
    def run():
        server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
        storage_pb2_grpc.add_StorageServicer_to_server(
            StorageServicer(), server)
        server.add_insecure_port(grpc_address)
        server.start()
        print("Server is running at: " + grpc_address)

        try:
            while True:
                time.sleep(86400)
        except KeyboardInterrupt:
            server.stop(0)
```

Listing 5.7: Python Server Class

The `grpc.server` function creates a server. We call it with the only required argument, a `futures.ThreadPoolExecutor` with the maximum number of workers set to 10.
Subsequently, we call the `add_StorageServicer_to_server` function to connect the `StorageServicer` to the service that we want to serve.

Using `add_insecure_port`, we set up the listening IP address and port, and subsequently start the server utilizing the start() method.
The `add_insecure_port` function is used since we are not setting transport protocols for our clientserver communication and authentication.

In Python gRPC, the server can be stopped from running by implementing `wait_server_termination()` [4] at the end of the method, which is a built-in API for Python gRPC. This API will block the current thread until the server stops. The wait will not consume computational resources during blocking, and it will block until one of the following conditions are met. Either the server is terminated, or a timeout occurs. Stopping the server by typing crtl + c in the terminal prints out a stack trace whenever the API throws an exception, hence making the terminal cluttered. However, it works as expected. We suspect that this may be a work in progress issue due to the API still in experimental stage. Hence, we opted for a different method using a sleep-loop as listed in Listing 5.7.

.NET implementation

The .NET implementation differs slightly from the other implementations due to how gRPC works in .NET and some specific C# capabilities.

Creating a .NET gRPC server is done by setting up a .NET Core gRPC server project, which we can do by utilizing the Visual Studio IDE and selecting gRPC when creating a new project. Setting up a server using this method will provide us with a simple project template for creating gRPC ASP.NET Core service using .NET Core. Furthermore, since we would be utilizing ASP.NET, an open-source, server-side web application framework, we would have easy access to valuable features such as logging, dependency injections, authentication, and authorization.

It is also important to mention that the template we generate using this method uses TLS (Transport layer security) by default, which is a great security feature. Unfortunately, it can cause some compatibility problems. These problems come from Kestrel's utilization in every ASP.NET Core project template that by default does not support HTTP/2 with TLS on macOS and older versions of Windows, such as Windows 7. Fortunately, this compatibility problem is resolved by configuring a Kestrel HTTP/2 endpoint without TLS in the project-generated *Program.cs* file.

Another thing worth keeping in mind when working with gRPC in .NET (version .5 in our case) is that the constructor of the class representing the service that we are going to use, in this case, the `Storage` service, will be called every time the server receives a gRPC request. As a result of this behavior, a new instance of the service is created for each request, making it impossible to share state between requests directly. Since each request will be utilizing a different instance of the Storage service, we cannot create an Instance of a Wasm module directly inside the gRPC service class. Doing so would result in horrible performance caused by repeatedly creating new Wasm Instances, which we have established in Section 2.1.5, are often created by expensive instantiation processes.

Our solution to this problem was to create an additional Wasm service/-class that would act as a store for the Wasm instance and implement the .NET's `CallWasm` function.

Nevertheless, as we mentioned above, this new service cannot be directly added to the `Storage` class, as it will be created anew alongside the class at every gRPC call, rendering this helping class ultimately futile. We can, however, register this Wasm service in our server as a singleton-service and then pass it as a reference to every gRPC request. This registration process can be seen in the listing below:

```
1 public void ConfigureServices(IServiceCollection services)
2 {
3     string[] functions = {"store_data", "read_data"};
4     var wasmLocation = "../wasm_module/storage_application.wasm";
5     Dictionary<string, string> preOpenedDirs = new Dictionary<string, string
      >();
6     preOpenedDirs.Add("./data", ".");
7     services.AddGrpc();
8     services.AddSingleton(new WasmSingleton(functions, wasmLocation,
      preOpenedDirs));
9 }
```

Listing 5.8: Registering the WasmSingleton service

Using this approach, we ensure that only one instance of the `WasmSingleton`, which is our Wasm service/class, would exist at any given time. This means that even though the Storage service class constructor will be invoked several times, the same Wasm instance will be utilized, thus avoiding unnecessary overhead.

Moreover, as we can see in Listing 5.8 line 6, the constructor of the `WasmSingleton` class is essentially this implementation's version of the `WasmInstantiate` function. This constructor will handle the Wasm instantiation process and the subsequential storing of the specified exported functions.

Finally, we can go over our version of the `callWasm` function in this implementation, which due to generics and dynamic objects, can be written in a short and straightforward manner. Furthermore, we do not necessarily need to define all exported functions in advance because, using dynamic types, we can directly invoke the exported functions without defining them first. This method comes at a performance cost, which is, fortunately, practically neglectable. Figure 5.1 illustrates an example of how we would invoke an exported function without first defining it.

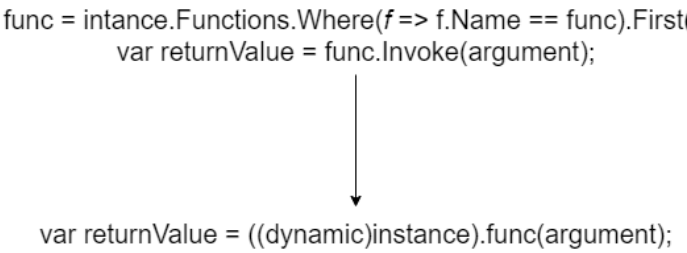

Figure 5.1: Dynamic exported functions

Additionally, here is a little snippet of how we have use generics in the `callWasm` function:

```
1  public T callWasm<T>(String fn, IMessage message) where T : IMessage<T>, new
       ()
2  {
3      try{
4          ....
5      }
6      finally{
7          mu.ReleaseMutex();
8      }
9      message = parse<T>(result);
10     return message is T value ? value : default(T);
11 }
```

Listing 5.9: callWasm function in C#

The `T` in the function listed in Listing 5.9 represents the type of gRPC message that we intend to unmarshal into and that we plan to use as return value for this function. The `IMessage` interface in the arguments is an interface that every gRPC message has to implement, making it the C# counterpart of the Go's `proto.Message` interface.

Rust implementation

The Rust implementation was the one that needed the most amount of work, partly due to the need to utilize an unofficial Rust gRPC implementation, as there are no official ones, and because of some incompatibilities between Rust's Wasmtime and the gRPC unofficial implementations.

As of early 2021, there are many different Rust implementations for gRPC, notably the Tonic and gRPC libraries (crates), which provide a full implementation of the gRPC protocols. However, these two libraries are not equal at the moment, as the gRPC library is, in its developers' works, not production-ready and lacks in stability and performance. These shortcomings drove us to choose the more performant, stable, and documented Tonic library. The Tonic library is especially great because of its focus on the support and utilization of `async/wait` in Rust, which theoretically should deliver excellent performance.

The first big problem that we encounter when trying to implement the server, following the type of structure that we have utilized so far for the other implementations, is the inability to share our Wasm instance or our exported Wasm functions between different threads. This limitation comes from the initialization process of a Wasm module in Rust, which starts by initializing a Wasmtime store that acts as an anchor for all the other Wasmtime objects. As of version 0.23.0 of Rust' Wasmtime library, this store object is not thread-safe, which means that it and any other object connected to it, such as exported functions, are pinned to a single thread. This single

thread behavior happens automatically through the lack of implementation of the Send and the Sync trait (propriety), which means that we cannot utilize Wasm objects in asynchronous functions. Even though we can set up Tonic to work with just one thread at the cost of losing performance, there is no way to circumvent the asynchronous function problem. This means that we cannot store Wasmtime objects directly in the store struct, as we cannot use them in the gRPC functions.

Because of Tonic's inherent reliance on `sync/wait`, we need to utilize a slightly more complicated solution to use Wasmtime in this case. Our solution comes in the form of an "actor," but before we can talk about how we implemented it, we have to explain what an actor is and why it resolves our problem. The fundamental intention behind an actor is to spawn a self-contained task that performs a specific job autonomously, that disregards what the other parts of the program are doing. An actor will then be able to communicate with the rest of the program by utilizing message-passing channels. The actor's autonomy means that we will essentially be running it in parallel to the Tonic server. In our case, the actor will have full exclusivity on the Wasm instance ownership, which will then be available to the rest of the program through indirect access by talking to the actor. Assigning the task of Wasmtime handling uniquely to the actor means that every Wasmtime object will remain on one single thread, thus taking care of the single-threaded nature of these types of objects.

This `WasmActor` can be divided into two parts: the task and the handle. The task is the autonomous spawned thread that performs the actor's duties, which means that it takes care of the Wasm instance's interactions. The handle is a struct that the Tonic server will utilize to communicate with the task. Figure 5.2 illustrates how this type of server would work.

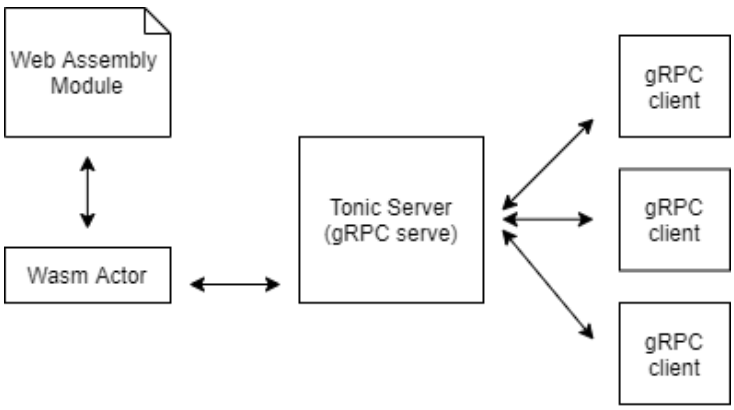

Figure 5.2: Tonic Server structure

The Wasm Instance initialization process and Wasi configuration can occur in the `WasmActor` (the task) constructor. It is worth noting that this process is slightly different from the other implementations, as the developers of Wasmtime in Rust have decided to divide the Wasmtime library into a bunch of smaller size libraries. Thanks to this decision, we can select exactly what type of features we want to import, rendering the file sizes smaller in the process. We need these three libraries for this type of WASI server: the base `wasmtime` library, `wasmtimewasi`, and `wasi_cap_std_sync`. The last two libraries are specific for working with WASI. They give us the tools necessary to instantiate WASI and modify the WASI configuration, such as a preopen directory.

Once we are done with the Wasm module's instantiation process, we can store the Wasm memory and all the Wasm functions needed in the actor's struct, similar to how we did it in the server struct of both the Go and Python implementations. Since Tonic utilizes *prost* as its unofficial protobuf library, and prost's generic protobufmessage is not object-safe, we cannot create a generic `call_wasm` function like the ones in the other implementations. As a result of this missing feature, we will not be able to use a generic protobuf-message either as a parameter or a return value, which means that we need to create a specific `WasmActor`'s method for each gRPC function that our service provides. Our solution, however, was to create a method called `handle_request` that utilizes an enum as the parameter to specify what gRPC message is being called. Nevertheless, this still means that there cannot be a generic way to interact with the Wasmtime instance, but we can keep the code slightly cleaner using this method.

```
enum ActorRequest {
    Read {
        respond_to: oneshot::Sender<ReadResponse>,
        request: ReadRequest,
    },
    Write {...},
}

struct WasmActor {
    receiver: mpsc::Receiver<ActorRequest>,
    funcs: HashMap<String, wasmtime::Func>,
    memory: wasmtime::Memory,
}

impl WasmActor {
    fn new(receiver: mpsc::Receiver<ActorRequest>, dir: Dir) -> Self {...}

    fn handle_request(&mut self, msg: ActorRequest) {
        match msg {
            ActorRequest::Read {respond_to, request,} =>
            {
                let mut buf = BytesMut::with_capacity(500);
                request.encode(&mut buf).unwrap();

                let bytes_vec: Vec<u8> = buf.to_vec();
                let result = self.call_wasm("read", bytes_vec);

                let buf = &result[..];

                let response: proto::ReadResponse;
                response = prost::Message::decode(buf).unwrap();

                let _ = respond_to.send(response);
            }
            ActorRequest::Write {respond_to, request,} => {...}
        }
    }
    fn copy_to_memory(&mut self, data: Vec<u8>) -> (i32, i32) {...}
    fn call_wasm(&mut self, f_name: &str, data: Vec<u8>) -> Vec<u8> {...}
}
```

Listing 5.10: WasmActor

`ActorRequest` shown in Listing 5.10, is the enum that defines the kind of message that we can send to the actor through the handle. Utilizing an enum, we can have many distinct types of messages, one for each gRPC kind of request. Once we are done interacting with the Wasm instance and its exported methods, we need to return a message to the handle by using a oneshot channel, a message-passing channel that allows us to send exactly one message.

In our solution, we match the enum inside a handle_request method on the `WasmActor` struct, but that is not the only way we can structure this. We could also directly match the enum in the `run_my_actor` function. However, each matching branch would have to call a specific method on the actor object,

such as a store_data method, using this structure type.

In the `run_my_actor` function, shown in Listing 5.11, we can detect when the actor should shut down by looking at failures to receive messages. When all senders to the receiver have been dropped, we can be sure that we will never receive another message, which means that the actor can be shutdown. When this happens, the call to `.recv()` returns None, and since it does not match the pattern `Some(msg)`, the while loop exits, and the function returns.

```
fn run_my_actor(mut actor: WasmActor) {
    while let Some(msg) = actor.receiver.blocking_recv() {
        actor.handle_request(msg);
    }
}
```

Listing 5.11: run_my_actor function

Once the actor has been defined, a solution to communicate with it is needed. It comes in the form of a handle to the actor, which will be an object that the server will use to talk to the actor and what actually keeps the actor alive. Unfortunately, because of the previously mentioned prost library limitation, much code will need to be repeated for each type of request sent to the handle.

```
#[derive(Clone)]
pub struct WasmHandle {
    sender: mpsc::Sender<ActorRequest>,
}

impl WasmHandle {
    pub fn new(dir: Dir) -> Self {
        let (sender, receiver) = mpsc::channel(8);
        thread::spawn(move || {
            let actor = WasmActor::new(receiver, dir);
            run_my_actor(actor);
        });
        Self { sender }
    }
    pub async fn get_write_response(&self, request: WriteRequest) ->
    WriteResponse {
        let (send, recv) = oneshot::channel();
        let msg = ActorRequest::Write {
            respond_to: send,
            request,
        };
        let _ = self.sender.send(msg).await;
        recv.await.expect("Actor task has been killed")
    }
    pub async fn get_read_response(&self, request: ReadRequest) ->
    ReadResponse {...}
}
```

Listing 5.12: WasmHandle.

In the end, once we are done with both the actor and the handle, we can add the handle to the Tonic server struct, giving the server a way to communicate with the Wasm instance. Using this type of structure for the gRPC server resolves all the problems that we encountered by trying to use Wasmtime and Tonic together. However, the amount of work needed to make this code reusable for another gRPC server is not indifferent.

In this implementation, we can see that the `WasmActor` constructor is essentially this version's `WasmInstantiate` function, much like the `Wasmsingleton` constructor in C#.

However, due to this implementation's heavy reliance on server/application-specific methods and enums, we retain that creating a proper `WasmInstantiate` function does in practice little to no difference to the overall complexity of this type of server, rendering this function pointless.

We could say that the lack of a gRPC generic message makes this code almost unreusable compared to all the other server implementations. Hopefully, this is a problem that will be fixed in future `prost`'s updates since the object-safe version of the generic protobuf message trait is likely to be released.

It is safe to say that the code will become more generic and reusable when such a change is made, which will result in the abandonment of the server's specific code and a shorter code overall.

5.3 Storage algorithm

In a storage application with a structure like this one, it is crucial to keep in mind different things such as:

- Consistency
- Resilience (number and types of failures tolerated)
- Complexity

Taking a more in-depth look at our storage application logic, we can see that we have been following what a certain L. Lamport (for a more in depth read [18]) defines as universally accepted consistency guarantees for a read/write storage abstraction. By utilizing locks when interacting with storage files, we guarantee that read and write do not happen concurrently, thus ensuring the safety, regularity, and atomicity of every storage server. This is not ideal for performance since concurrent reading could technically be allowed as long as no writes are being processed simultaneously. However,

because of Wasm's non-multithreaded nature, this cannot be done, as only one client can ever have access to the Web-Assembly module at the time.

Merely using locks in each storage server does not ensure consistency between separate storage. Furthermore, there is no fault-tolerant mechanism in operation, as whenever a server encounters an error or fails, it will be automatically dropped by the clients connected to it. Many of these issues could be addressed using a basic storage algorithm, such as the ABD algorithm [24]. This algorithm archives a state of consistency between each storage by splitting read operations into two rounds: a straightforward read request to every server and a round-back write to each server that does not have the most recent data version. This mechanism should be adopted, but it is essential to remember that write-back requests "steal" time from other read or write requests since these cannot happen concurrently when utilizing Wasm. Unfortunately, this means that the overall application performance is going to be affected. For simplicity and to benchmark write and read requests singularly, we decided not to implement this feature. However, we strongly advocate for such a mechanism to be utilized in this type of application.

On the other hand, ABD's use will improve the application's apparent performance to a user because of how crash failures are handled. ABD allows an acceptable t crash failures out of $n = 2t + 1$ storage servers, which means that if you are using three servers, ideally speaking, you can afford the crash of one single server before the storage application stops function the way it should. This type of behavior means that a client's read or write request is considered completed if two out of three servers have acknowledged it.

Given that different Wasm server implementations might have vastly different performances, we may infer that using such an approach to tackle crash errors would be advantageous since customer operations' performance would improve. A client would not wait for slow or faulty connections but would deem the read/write request terminated even though it was still being handled by one of the servers (increase apparent performant). To actually boost performance, a system for dropping pending requests might be set up, with the caveat that not all requests to slow storage should be dropped. Doing so would make a slow server, more or less completely inconsistent with all the other storages.

5.4 Benchmarking setup

In this section, we will go through the setup needed for benchmarking this application. Since our University permitted us to utilize a set of powerful machines located on campus, called BBchain machines, we decided to create

a custom-made benchmarking tool capable of fully utilize such devices. The table below are the specifications of the BBchain machines.

CPU	Intel® Xeon® E-2136 Processor
RAM	32GB RAM
Storage	1.6Tb SDD
Network	Intel I210 Gigabit NIC speed 1Gb/s (available)
OS	Ubuntu 18.04.4 LTS
Go ver.	1.16 linux/amd64
.Net ver.	5.0.200
Python ver.	3.6.9

Table 5.1: Specifications of BBchain machines.

Unfortunately, the ghz tool does not have an option to connect to multiple gRPC servers at once and thus had to be put on the side as it was no longer relevant. Moreover, to make the benchmarks even more realistic, we decided to run the clients on different BBchain machines. As a result of this , the odds of a client-side bottleneck is significantly diminished, and the benchmark will depend more on the server's efficiency alone. The approach that we took to create our custom benchmarking tool can be divided into different stages. We started by creating a command-line program that creates a gRPC client, establishes connections to m specified servers, and sends n gRPC requests sequentially to each server. This way, every request will be forwarded to m servers, forming an m-set of requests. Using a pre-allocated array, the program will store how long each m-set of requests takes (latency in micro-seconds) and at what second it returns. Once the client is done sending all its requests, it will wait some seconds and then write the benchmark data to a CSV file.

Using this type of program, we can easily start several custom clients on the same computer or different computers using a shell script. Furthermore, once the benchmark has ended, we can utilize another shell script to collect logs from all machines that have worked as clients. The data will then be processed using Python code to obtain more valuable information, such as average latency and the average throughput. The throughput comes by counting how many m-set of requests returned simultaneously (with second precision) throughout the benchmark.

It's also worth noting that the first and last seconds of the benchmarking process will be discarded, as it is highly unlikely that the benchmark would begin or finish its execution exactly at the begging or at the end of a second.

Furthermore, the type of connection we use with gRPC (TCP connections) is known to have a warm-up time. This means that the first few requests would almost certainly have a higher latency than the rest of the benchmark and would not be very representative compared to the rest of the data.

5.5 Benchmarking

The benchmarking procedure for this application is going to be more involved than previous benchmarks. The assessment of the performance of Wasm with gRPC remains our primary concern. However, unlike in the past, we can compare the results of various Wasm implementations rather than simply comparing Wasm to non-Wasm implementations. Since this program was designed to use various servers simultaneously, we will benchmark each server independently to collect data representing the particular implementation. Subsequently, we will be benchmarking the program as it was intended to be used. In other words, we will compare the performance of different sets of servers. Each test was run 10 times, and the values that are gonna be used in the graphs will be an average of those benchmarks.

Finally, we want to address a library version problem encountered in the experimental phases before the actual performance evaluation process. Some of the libraries we used in our implementations received updates during our testing period, resulting in significant efficiency and consistency improvements. Furthermore, there was a case where performance did not increase after an update but decreased by 15%. Consequentially, we decided not to update any library used by our implementations during the benchmarking phase to maintain consistency. Not doing so would result in dramatically different and inconsistent results. Table 5.5 lists the names and versions of the libraries that we have been using throughout the project.

wasmtime-go	v0.22.0
wasmtime (rust)	v0.23.0
wasmtime-wasi (rust)	v0.23.0
wasi-cap-std-sync (rust)	v0.23.0
wamtime-py	v0.22.0
wasmtime-.NET	v0.22.0

Table 5.2: Libraries utilized

We have tried to use the same version for all the libraries. However,

version 0.22 for the wasmtime-Rust libraries seemed to have some bugs when interacting with WebAssembly's memory, so we decided to use the yet-to-be-released version 0.23, which at the time was more stable than the official version.

5.5.1 Benchmark 1

In this section of the benchmarking process, we will look at the results of each implementation individually. We begin by comparing Wasm-based storage implementations to the corresponding non-Wasm-base storage. Finishing with a comparison of all Wasm implementations to see just how different such implementations and Wasmtime libraries are in terms of performance. The reading and storing aspects of the application will be benchmarked separately, as not doing so would give us an unclear image of each storage version's actual performance.

Moreover, we want to know if a Wasm storage system is capable of handling both small and large files well. Thus, we opted to test with two different data sizes: 10 bytes and 1 MB to test both the reading and the writing (storing) of data. For simplicity purposes, the same file is used during writingbenchmarking and ensured that such a file already existed before each benchmark. The file creation process would add unnecessary latency to the final result, misrepresenting the program's actual writing performance. It is important to note that TLS is omitted for any of the tests so we can get an idea of the raw performance with each.

Figure 5.3: Wasm performance comparison for reading files of 10 bytes each with different number of clients

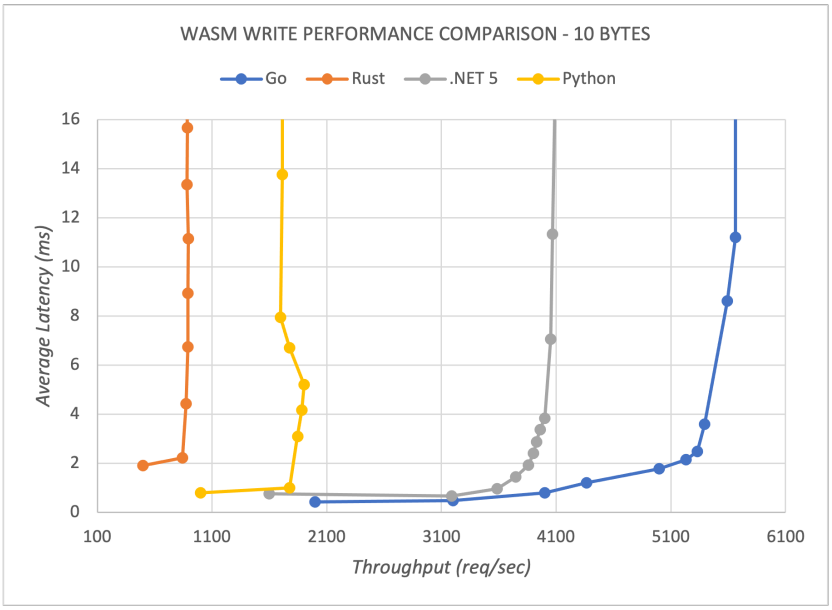

Figure 5.4: Wasm performance comparison for writing files of 10 bytes each with different number of clients

Figure 5.3 and Figure 5.4 demonstrate the performance of Wasm when writing and reading 10 byte files. The graphs display the average time delay in milliseconds and the number of requests per second for each write/read operation.

From Figure 5.3 and Figure 5.4, we can get some interesting results:

1. The average latency for each file operation increases with the number of clients used. By latency, we refer to the time delay between initial input and output.

2. The performance disparity between Go and .NET 5 may be explained by the fact that, despite the small amount of code that runs concurrently in these gRPC implementations, Go typically handles concurrency better than .NET, as gRPC-Go utilizes by default one goroutine (lightweight threads managed by the Go runtime) per method call, which are known to be performant and fast.

3. Looking at the performance differences for reading and writing for each implementation, we can see that .NET 5 and Go have a significant boost in performance when comparing write-operations to read-operations. It is apparent that Go is better overall.

4. On the other hand, looking at Python and Rust (yellow and red lines) reads and writes performances, we can see little to no performance improvement, as the maximal throughput is about the same in both graphs. Python, illustrated by the yellow lines, is an interpreted programming language, making it an unsuitable candidate for a server where speed is a priority. Furthermore, Python is known for its "poor" multithreading, due to the presence of a Global Interpreter Lock (GIL) [22]. Therefore, Python was expected to have worse performance compared to the other implementations. On the other hand, we have Rust, represented by red lines, that does support real concurrency. However, due to our implementation that utilizes an "actor" to handle the Wasm instance (subsection 5.2.2), the server is practically a single-threaded application.

5. We can see that the throughput decreases after a certain number of clients. However, this does not occur simultaneously for all implementations, as they all begin to degenerate at different client number thresholds.

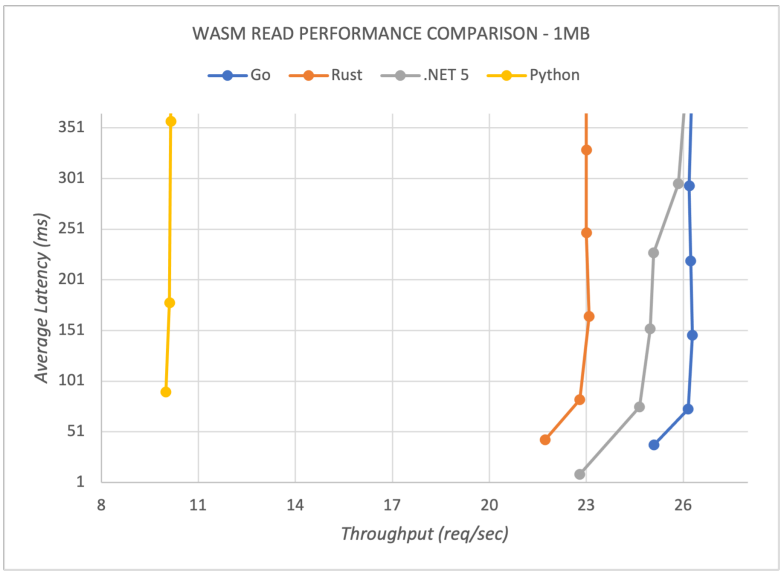

Figure 5.5: Wasm read performance for each server with 1MB file

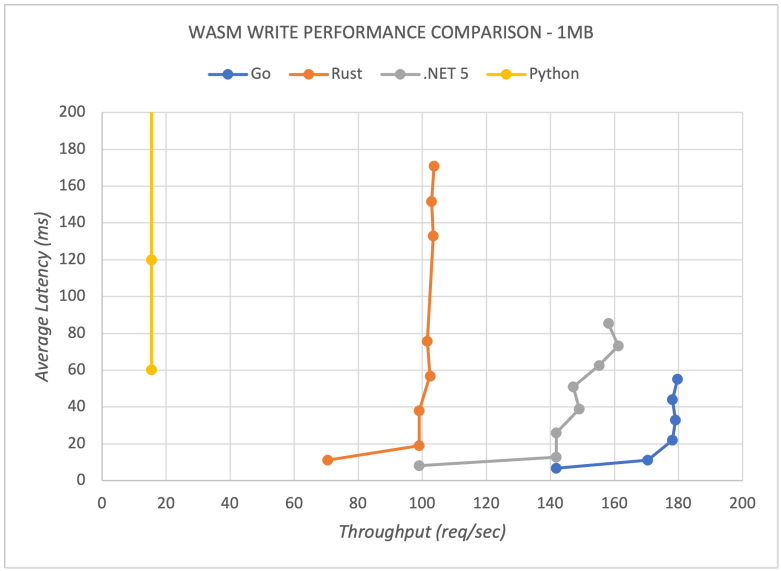

Figure 5.6: Wasm write performance for each server with 1MB file

Figure 5.5 and Figure 5.6 show the respective write and read results for a 1 MB file. We may draw similar conclusions regarding these 1MB figures as we did with the previous 10 bytes figures. We can see that the data is much more jittery in this case, which may be due to network congestion caused by the large amount of data being transmitted over the network.

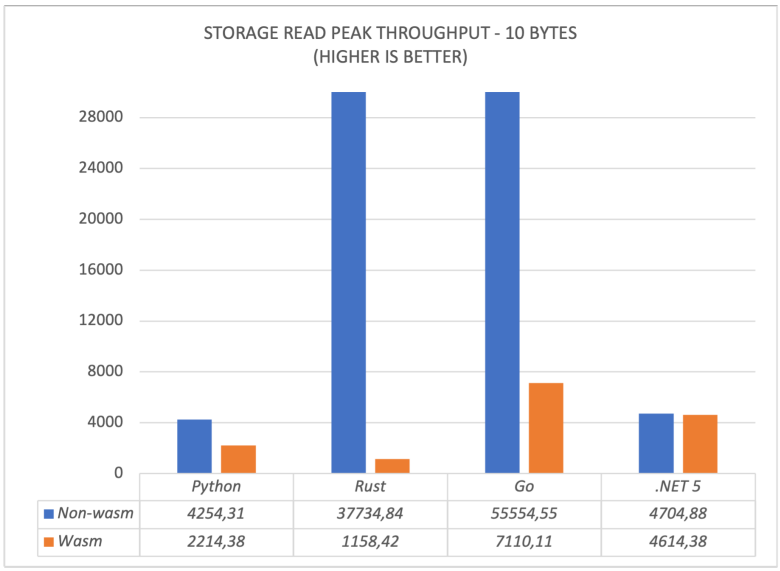

Figure 5.7: Peak throughput comparison between Wasm and non-wasm server implementations for read operation with file of 10 bytes in size

Figure 5.7 and Figure 5.8 compare the average maximum performance of Wasm-implemented servers to non-Wasm-implemented servers. Non-Wasm outperforms its Wasm counterpart by more than two. While Wasm is regarded as fast and efficient, these results are unsurprising, provided that Wasm performs additional tasks before reading and writing operations. By additional tasks, we again refer to memory allocation, deallocation and additional marshalling and unmarshalling of data.

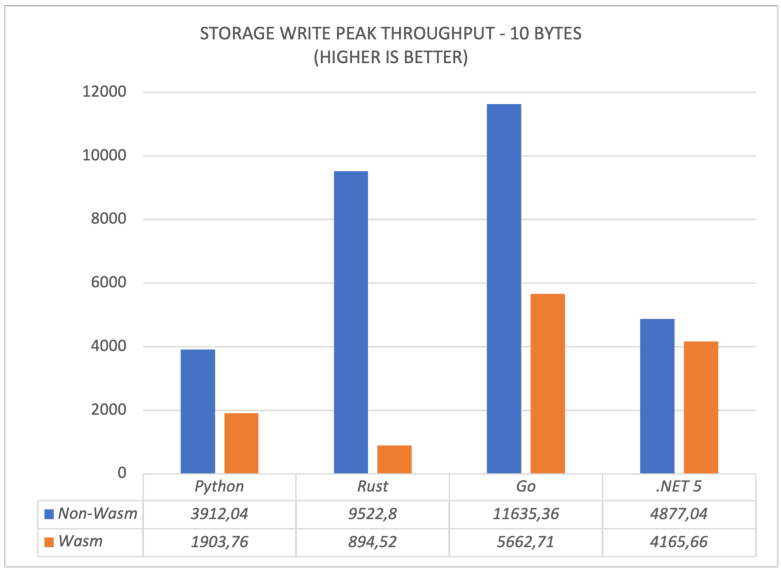

Figure 5.8: Peak throughput comparison between Wasm and non-wasm server implementations for write operation with file of 10 bytes in size

However, we see from these figures that the massive disparity in performance is not present between the .NET Wasm and non-Wasm servers. In the write benchmark, non-Wasm has a 17% advantage over Wasm and a mere 5% advantage in the reading benchmark. This minor performance disparity does not reflect the visible performance disparity between non-Wasm implementations and their Wasm parallels. This behavior can be explained either by the possibility of an impressively good optimized .NET Wasmtime library, making Wasm performance much more in line with native performance, or perhaps, by a structural mistake during the implementation of this specific Wasmless server.

Figures 5.9 and 5.10 show these servers' performance when reading and writing operations are done with a 1MB file. Here, the Wasmless implementations' performances are remarkably similar to each other, as most of the time is being used doing I/O, making the singular server performances almost neglectable in this case.

Figure 5.9 shows a similar execution behavior for all Wasm implementations, which are about a hundred times worse than their Wasm counterparts. Notably, we can see that even Python and the virtually single-threaded Rust implementation perform just as well as the Go and .NET implementations, showing that single-threaded performance does not matter when file-sizes become this big. When it comes to figure 5.10, we can see that the performance disparity is more accentuated, making Python Wasm the least performant

server by a wide margin.

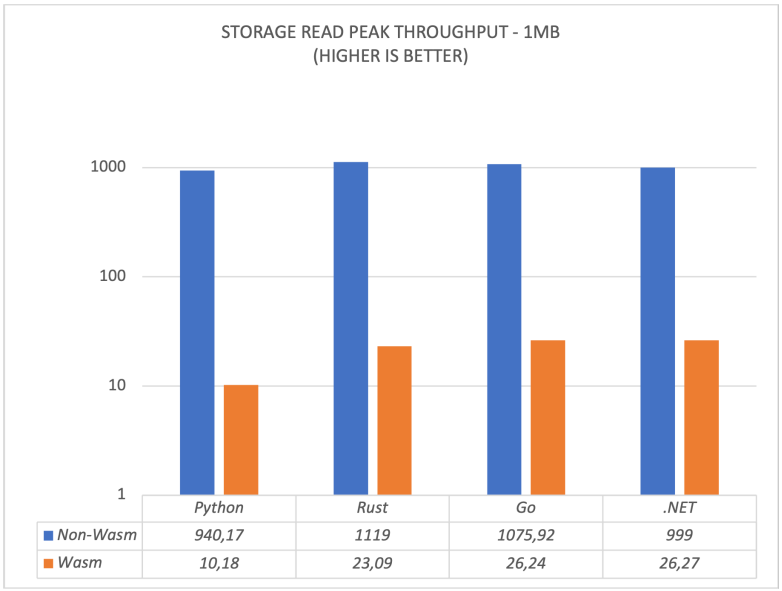

Figure 5.9: Peak throughput comparison between Wasm and non-Wasm server implementations for read operation with file of 1 MB in size

Figure 5.10: Peak throughput comparison between Wasm and non-Wasm server implementations for write operation with file of 1 MB in size

As previously stated, it is to be expected for Wasm to perform worse than its Wasmless counterparts. That is mainly because of all the additional actions executed by Wasm. When the file used is around a thousand times bigger, it is critical to note how much of a dive the Wasm servers have taken relative to Wasmless servers. For all implementations, the performance gap widened significantly, indicating that Wasm servers of this kind are not the best option for a storage application that handles medium-sized data.

Finally, looking at the gathered values, we can conclude that an increasing number of concurrent clients will substantially impact the application's latency, which can be better seen taking a look at the graphs B.1–B.4 in Appendix B. Overall, Wasm's latency worsens over time for each concurrent client, while throughput progressively increases at a steady pace.

5.5.2 Benchmark 2

This segment's benchmark is more focused on how a set of three different servers would do if we waited for two out of three responses. In this scenario, the benchmarks divide into three groups. In the first group, we will run .NET 5, go, and python servers with Wasm on three BBChain machines simultaneously to model real-world storage structures used in data centers. Doing such a test would theoretically mean that our storage application would run at the speed of .NET 5. We chose to leave out the Rust server because it is not suitable for this test due to its poor performance with Wasm and could result in an unnecessary performance hit. The second group consists of three Wasm servers that are all written in the same programming language, in this case Golang. The third group is the same as the second group, except the Wasm is absent. Both three groups work on files that are 10 bytes in size.

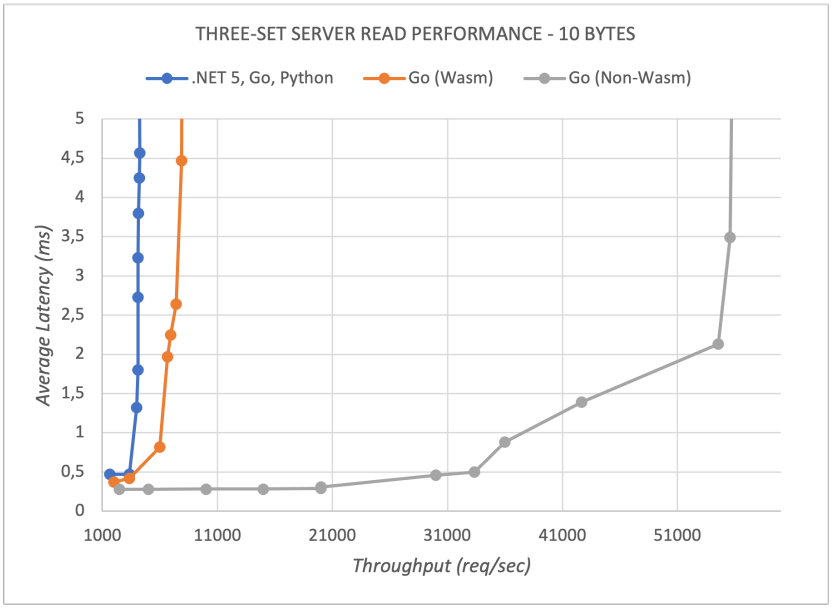

Figure 5.11: Read performance comparison of different sets of serves. Wasm(diversified) vs Wasm vs Wasmless

Figure 5.11 and Figure 5.12 plot the performance of reading and writing 10-byte files for Groups 1, 2, and 3. We may draw similar conclusions from the previous benchmark and the following conclusions after doing a rigorous review and analyzing the graphs for each group:

1. Group 1 (Golang + Python + .NET 5), represented as a blue line, runs close to the .NET 5 server's speed for both read and write operations. A better overview of the performance is documented in the Appendix A (Calculations and test data). Although it operates similarly to a single .NET 5 server's speed, the reading process only gains a tiny amount of throughput. The writing process has a slightly higher throughput advantage, but not by much.

2. Group 2 and Group 3, represented by an orange line and a light gray line, perform better than running a single server. This performance improvement might be due to the fact that we are receiving two out of three replies.

3. Groups 1 and 2 are identical in terms of how the output is plotted as the number of concurrent clients in the background increases.

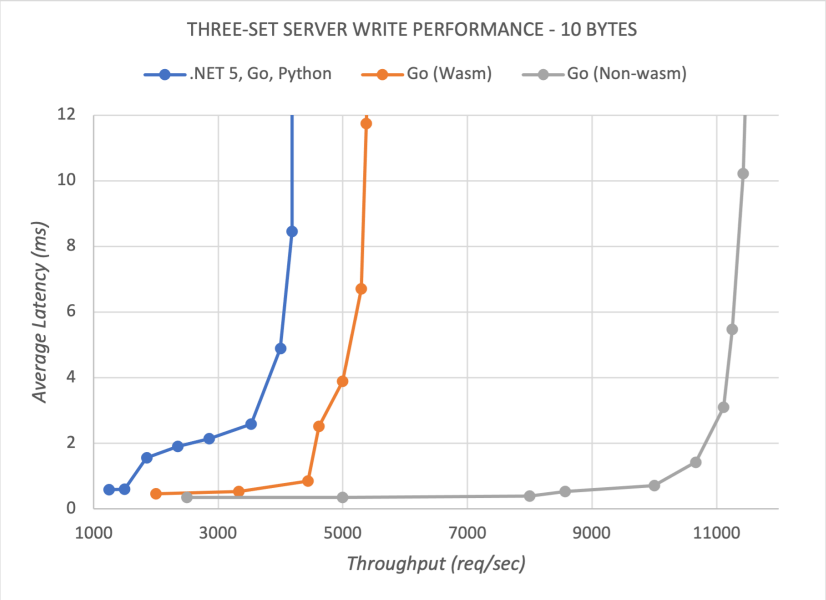

Figure 5.12: Write performance comparison of different sets of serves. Wasm(diversified) vs Wasm vs Wasmless

5.6 Runtime comparison

The Wasm runtime landscape is vast and varied, and many different runtime libraries can be utilized to embed WebAssembly modules in various languages. Although we have focused solely on the utilization of Wasmtime so far, which is nowadays the library that gets the most consistent updates, there are many other options to explore, such as SSVM and Wasmer.

The latter is considered by many to be the best Wasm library for embedding, mainly because of the impressive high number of supported languages, which is superior to any other runtime library, Wasmtime included. While the Wasmtime runtime can be used for embedding in five different languages, Wasmer can do the same in an impressive fifteen languages, keeping the library structure and functionalities pretty consistent among all implementations. Moreover, after some recent updates, the Wasmer and Wasmtime libraries have become fairly similar, rendering the migration process between the two libraries moderately seamless.

While Wasmtime was built on an optimized Cranelift code generator to produce high-quality machine code at runtime, Wasmer was created to be pluggable, making it possible to choose between diverse compilation frameworks, rendering Wasmer more customizable than Wasmtime.

Following this article [6] we can see that Wasmer ships with out-of-the-box

support for three different compilation frameworks: Singlepass, Cranelift (like Wasmtime), and LLVM.

- **Singlepass** has impressively fast compilation times and is not prone to JIT-bombs, making it desirable for something like blockchains.
- **Cranelift** (default option) has relatively fast compilation times and execution times.
- **LLVM** has slow compilation times and speedy execution times, allegedly close to native times, making this the best option in applications where speed is essential.

However, while the Singlepass and the Cranelift frameworks can easily be enabled in most of the Wasmer libraries, the enabling process for LLVM is not as simple, mainly due to bugs and some cross-platform issues.

While the default libwasmer embedded inside a Wasmer library like Wasmer-go does not support LLVM, the entire API already supports it, which means that with a custom libwasmer that includes LLVM, we could be able to utilize LLVM. Unfortunately, after some testing on our own and a few interactions with the Wasmer developers, we could not utilize LLVM reliably. Thus, we decided not to include LLVM Wasmer in our benchmarks. However, we have been reassured that LLVM will be available for use once some bugs and problems have been ironed out. According to the Wasmer developers, LLVM should drastically boost the Wasmer performance in applications where speed is crucial.

In this segment, we will see how Wasmer stacks against Wasmtime. More precisely, we will be comparing Wasmer-Cranelift-go with Wasmtime-go. Naturally, while working with the Wasmer-go's implementation, we tried to keep the application's structure as similar to the one used for Wasmtime-go as possible, thus making it easier to concentrate exclusively on the performance discrepancies between the two runtimes.

The following Figure 5.13 and Figure 5.14 show the results of the benchmark tests run for Wasmer. Wasmer's tests were run as many times as with Wasmtime's. Despite the fact that both runtimes are based on Cranelift, we can see that Wasmtime is a more efficient and quicker overall runtime. With five concurrent clients, Wasmer performed very close to Wasmtime for both read and write operations but performance gradually declined. When executing write operations, the performance difference becomes more apparent after five clients. However, the performance gap also becomes apparent for read operations, but it narrows towards the end.

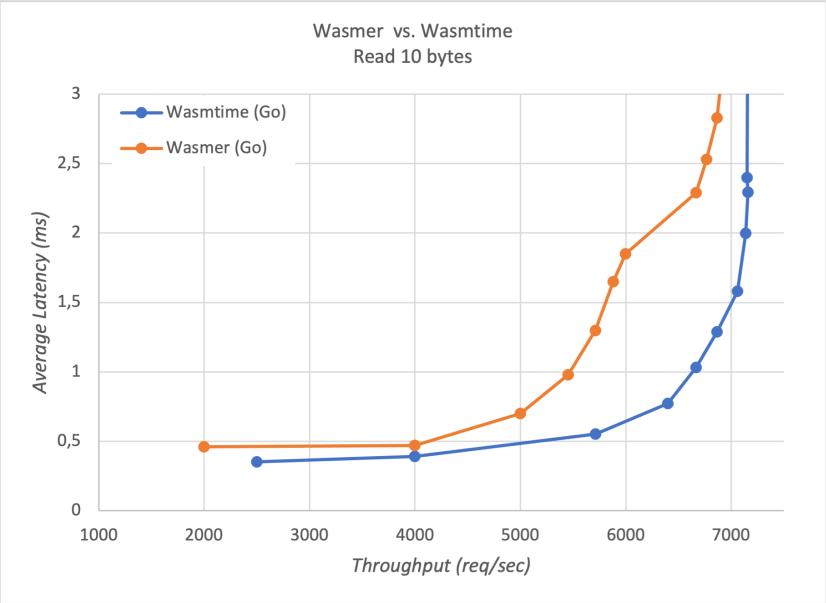

Figure 5.13: Performance between Wasmer and Wasmtime running read operations

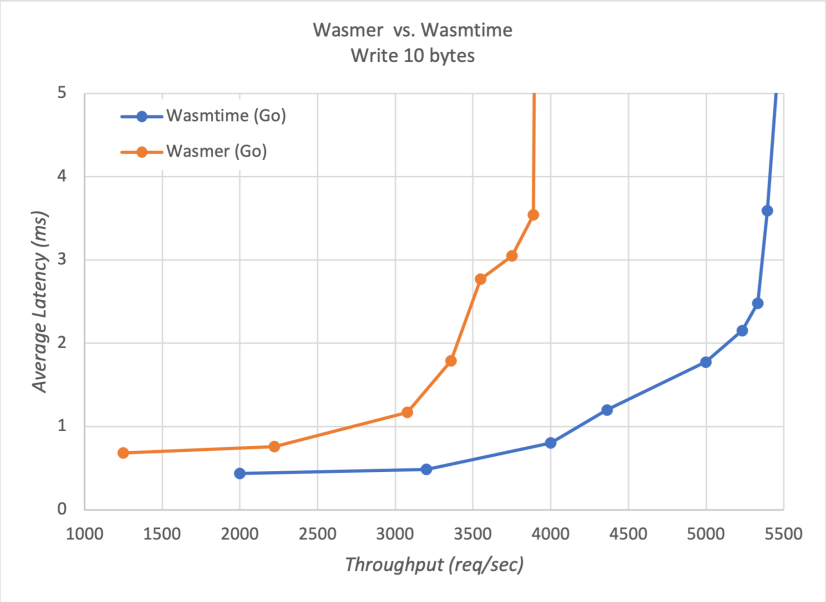

Figure 5.14: Performance between Wasmer and Wasmtime running write operations

Chapter 6

Discussion

6.1 Memory in WebAssembly

When dealing with Wasm embedded in other programming languages, the memory architecture of WebAssembly modules is most definitely one of the most important factors to keep in mind.

Wasm's sandboxed design offers a high degree of security, with features such as the ability to always know the size of a module's memory. The runtime will use this information to determine if a memory offset accessed by a module is still within the allocated memory boundaries. As a result of this feature, a module cannot access another module's memory or the memory of the runtime's underlying operating system unless explicitly given access.

However, even though this memory design has many advantages, it also creates considerable efficiency problems when utilizing Wasm in many different applications. Conceptually, there is a choice between copying the data from the host (hosting application) to the module or the module owning the data and managing its lifetime when working with Wasm. Depending on the use case, both of these approaches are valid and should be considered.

However, our experimentation with gRPC and Wasm revealed that the latter method, in which data is transferred from a host to the module, is inefficient since object-pointers cannot be passed to the modules. As a result of this constraint, all Wasm interactions that include objects as arguments must copy those objects from the host memory to the module memory, which adds overhead to the host-guest interaction process.

Furthermore, these objects have to be copied in byte format, adding serialization to the process needed when interacting with a module's memory. This also works the other way around when retrieving objects from the module's memory, as said objects must be in byte format, meaning that

deserialization is needed here.

Last but not least, while WebAssembly runtimes do an excellent job of isolating the memory instances of different modules within their linear memories, WebAssembly modules are not inherently safe from memory vulnerabilities. Since transferring objects to a module involves dealing directly with memory allocation and deallocation, vulnerabilities such as buffer overflow or use-after-free are possible, meaning that caution is required while handling memory.

Finally, we would suggest using Wasm more with CPU-intensive applications instead of using it with storing or data-heavy applications. Using Wasm in data-heavy applications would undoubtedly result in worse performances. However, the use of Wasm in such applications could be considered acceptable in not performance-focused applications and applications where security is considered especially important.

6.2 gRPC and Wasm complications and possible improvements

Working with gRPC and Wasm at the same time adds a whole new layer of complexity. The most elegant way to communicate with WebAssembly modules in gRPC applications, in our opinion, is to use proto-messages. We can keep much of the program logic in the WebAssembly module by using these messages for communication, essentially reducing the host environment to a medium for communicating with the Wasm module.

However, a regular gRPC server's normal behavior is to unmarshal every proto-message when received, which is desirable for standard gRPC services. Nevertheless, this behavior is not ideal when using Wasm, as it adds the need for additional marshaling and unmarshalling when interacting when a WebAssembly module.

As we have observed multiple times during our benchmarks, the amount of data passed between a guest (Wasm module) and the host directly correlates to how long a request will take.

The loss of throughput with growing payload sizes is a logical consequence of any gRPC server. What is fascinating here is to see how much the performance worsens.

In Wasm servers, during our numerous benchmarks, we have observed that the efficiency hit is much more noticeable than what is visible in standard non-Wasm gRPC servers. The evident deterioration in performance when utilizing Wasm is mainly due to the introduction of additional tasks required

when interacting with Wasm modules.

To get a clearer understanding of how gRPC requests operate when utilizing Wasm, refer to the list below, displaying all of the steps involved.

1. Receive gRPC request
2. Marshal grpc request
3. — Allocate memory for that specific gRPC request inside the module's memory
4. Copy marshaled gRPC request into module's memory
5. — Unmarshal proto message inside the module
6. — Do whatever this specific gRPC requests needs to do
7. — Save in the module's memory the resulting marshaled gRPC response and returns to host
8. — Deallocate memory for the gRPC request
9. Unmarshal the response from a linear byte-array representation of the module's memory
10. — Deallocate the Wasm response from the module's memory
11. Return the gRPC response to the client

Looking at the list above, we can see that, in addition to the time necessary for memory allocation and deallocation in a Wasm module, a considerable portion of the time is spent marshaling and unmarshalling proto-messages. In addition to the marshaling and unmarshalling that occurs naturally in gRPC servers, a Wasm-powered server performs two additional marshaling and unmarshalling sets.

One of these sets is needed when dealing with proto-messages coming in and out of the module's memory.

Another set is required within the module, where request messages are in byte form, and response messages must be saved in memory in the byte-form. As the size of these messages grows, the process of marshaling and unmarshalling becomes increasingly slower, which is an issue that cannot be solved until the WebAssembly interface-types proposal is implemented. Fortunately, while we wait for the aforementioned proposal to be implemented,

we have some ideas that might improve the overall efficiency of our gRPC servers:

– A custom serializer for all gRPC servers could return each gRPC request in byte-array format, bypassing the automatic unmarshalling present in every gRPC request (Wasm or non-Wasm). Using this method, we would also be avoiding the need to marshal the request before copying it to the Wasm module. This solution, however, would most likely eliminate certain valuable functionality of gRPC servers, such as the option to cancel an existing request or to return from a request if it takes too long (timer).

– Assign a portion of the marshaling and unmarshalling processes to clients using a wrapper proto-message with two fields, one reflecting the type of request and the other containing the actual proto-message in byte format. Using this sort of structure, we avoid marshaling before copying the proto-message into the module's memory and eliminating the need to unmarshal what is obtained from the module. However, using this system would imply that a gRPC server would only support one kind of generic gRPC request, with its actions changing based on the "requestType" field present in the wrapper proto-message.

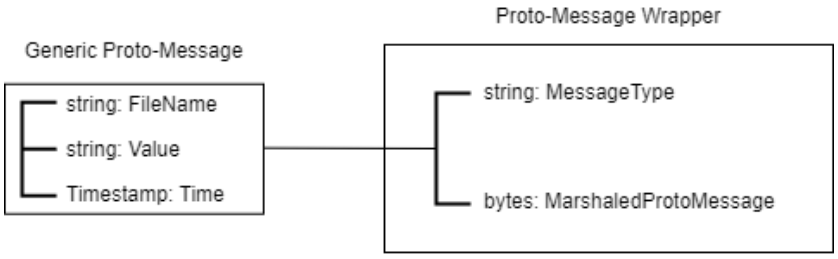

Figure 6.1: Proto-Message Wrapper

Using the wrapper method, we can drastically reduce the time wasted marshaling and unmarshalling data, and keep functionalities like interrupting ongoing requests. Unfortunately, using a wrapper without a custom serializer prevents us from preserving the traditional configuration of gRPC services, in which each service has a role on both the server and client sides. This might not sound like a problem, but it has some profound implications regarding the utilization of some gRPC features. Using this arrangement implies that

a server could only provide one service at a time per client, meaning that streams and unary messages could not coexist. One solution would be to have two types of wrappers: one for streams and one for standard unary messages. However, even with this system, we would be unable to have multiple streams open for the same client.

This issue can be addressed using a custom serializer that uses the wrapper proto-message as an envelope for any gRPC request/response, preserving the traditional gRPC structure and allowing us to separate each service into different gRPC functions. Of course, utilizing this structure would mean that the wrapper's "MessageType" field will no longer be used and can thus be removed.

We are confident that this improvement in the Wasm gRPC servers will result in a successful performance gain, as the overhead will be significantly reduced.

6.3 Current state of the Wasm Runtime libraries

Wasm's two primary runtimes are undeniably Wasmtime and Wasmer. Both provide a large amount of support for multiple programming languages, with Wasmer taking the lead on this front.

Wasmtime, making it simpler to understand how its libraries function and giving many examples of its utilization, even though some of them are outdated. Furthermore, since the Wasmtime community seems to be much more active, it is easier to get support when an issue occurs, making Wasmtime the superior alternative in our opinion.

Since these two runtime libraries have become more and more alike in performance and structure over time, we believe Wasmer is in a near second position, which could rise to first place in the future once LLVM is fully ready to be used.

These runtimes are updated at an astonishing rate. Wasmtime updates happen almost weekly, resulting in regular changes to how said libraries operate, possibly breaking old projects based on old versions. Keeping this in mind, when library consistency is required, we recommend Wasmer over Wasmtime, which, while being older, tends to keep the same library structure after updates.

6.4 Conclusion

Throughout this project, we worked on WebAssembly outside of the browser, with the primary focus on evaluating its performance and usability in distributed systems. We discussed what WebAssembly is, its advantages and disadvantages, and implemented two applications that benefit from the utilization of Wasm to increase security through diversification and Wasm's sandboxed nature. These applications were tested for throughput and latency in different scenarios. The performance in a LAN environment and the disparity in performance with and without WebAssembly when dealing with concurrent access and various file sizes were among our priorities while testing. During the evaluation process, interesting strengths and weaknesses for each Wasm implementation emerged.

In the latter project, the storage application, since we were dealing with I/O operations rather than what we did with the echo server where operations were executed exclusively in memory, the advantages and disadvantages of using Wasm were more pronounced.

The Go implementation was the most effective compared to the other implementations in terms of Wasm efficiency. It is an expected result as Golang is an efficient language and the Wasmtime package used is often updated containing potential bug fixes and optimizations. Furthermore, Golang supports concurrency from the language level and uses lightweight goroutines to explain its performance during the benchmark tests.

The Wasm .NET 5 implementation, on the other hand, was not far behind the Wasm go implementation. Surprisingly, the Wasm Python implementation did considerably well, especially considering how it handled small payloads (file size) compared to the Wasm Rust implementation.

Based on the results, we can safely conclude that utilizing our server structure, Wasm performs admirably when working with smaller payloads. It begins to struggle with larger payloads, such as 1 MB, but it also demonstrated promising results. With further optimizations, Wasm embedded applications' speed and reliability could improve to resemble the performance that can be seen in non-Wasm applications.

On a final note, even though our server structures are functional and could perform better after some improvements, such as a custom serializer, we believe that the fast evolution of Wasm and its runtimes will inherently make them obsolete. However, we are sure that the utilization of Wasm can, as of today, already bring considerable advantages to things like distributed systems and applications that value diversity and security. We consider Wasm to be a feasible option for applications where performance is not critical, such as video encoding or applications that heavily rely on digital signatures.

References

[1] Robert Aboukhalil. *Beyond The Browser: Getting Started With Serverless WebAssembly*. URL: https://www.smashingmagazine.com/2019/08/beyond-browser-serverless-webassembly/. (Visited: 18.01.2021).

[2] Bytecode Alliance. *Wasmtime*. URL: https://docs.wasmtime.dev. (Visited: 21.04.2021).

[3] Shantanu Alshi. *Fault Tolerance in Distributed Systems: Introduction*. URL: https://medium.com/@shantanualshi/fault-tolerance-in-distributed-systems-introduction-fcf618e8bac4. (Visited: 08.04.2021).

[4] The gRPC Authors. *Welcome to gRPC Python's documentation!* URL: https://grpc.github.io/grpc/python/grpc.html. (Visited: 20.04.2021).

[5] Lin Clark. *Standardizing WASI: A system interface to run WebAssembly outside the web*. URL: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/. (Visited: 18.01.2021).

[6] WASI collaborators. *Ivan Enderlin*. URL: https://wasmer.io/posts/wasmer-go-embedding-1.0. (Visited: 31.03.2021).

[7] Github contributors. *Interface Types Proposal*. URL: https://github.com/WebAssembly/interface-types/blob/master/proposals/interface-types/Explainer.md. (Visited: 23.02.2021).

[8] Repository contributors. *WebAssembly proposals*. URL: https://github.com/WebAssembly/proposals. (Visited: 10.03.2021).

[9] Flavin Cristian. "Understanding fault-tolerant distributed systems". In: *Communications of the ACM* (1991). DOI: 10.1145/102792.102801.

[10] Bojan D. *gRPC benchmarking and load testing tool*. URL: https://ghz.sh/docs/intro.html. (Visited: 26.01.2021).

[11] Google Developers. *Protocol Buffers*. URL: https://developers.google.com/protocol-buffers. (Visited: 26.01.2021).

[12] Wasmtime-Go Developers. *wasmtime-go*. URL: https://pkg.go.dev/github.com/bytecodealliance/wasmtime-go. (Visited: 13.02.2021).

[13] Wasmtime-Rust Developers. *Crate wasmtime*. URL: https://docs.rs/wasmtime/0.26.0/wasmtime/. (Visited: 25.02.2021).

[14] Google. *gRPC - documentation.* URL: https://www.grpc.io/docs/. (Visited: 26.01.2021).

[15] WebAssembly Community Group. *cargo wasi*. URL: https://github.com/bytecodealliance/cargo-wasi. (Visisted 03.03.2021).

[16] WebAssembly Community Group. *Runtime Structure*. URL: https://webassembly.github.io/spec/core/exec/runtime.html#memory-instances. (Visisted 13.02.2021).

[17] Vignesh Iyer. *WebAssembly: All you need to know about this Emerging Trend*. URL: https://medium.com/@vigneshiyer_44732/webassembly-all-you-need-to-know-about-this-emerging-trend-e002f8e18d04. (Visited: 26.01.2021).

[18] Leslie Lamport. "On Interprocess Communication". In: *Distributed Computing* 1.1 (1986), pp. 77–101. DOI: 10.1007/BF01786227.

[19] Mozilla and individual contributors. *WebAssembly*. URL: https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts. (Visited: 26.01.2021).

[20] Miguel Garcia; Alysson Bessani; Nuno Neves. "Lazarus: Automatic Management of Diversityin BFT Systems." In: *Middleware '19: Proceedings of the 20th International Middleware Conference* (2019), pp. 241–254. DOI: 10.1145/3361525.3361550.

[21] Miguel Garcia; Alysson Bessani; Ilir Gashi; Nuno Neves; Rafael Obelheiro. "OS diversity for intrusion tolerance: Myth or reality?" In: (2011). DOI: 10.1109/DSN.2011.5958251.

[22] *Standardizing WASI: A system interface to run WebAssembly outside the web*. URL: https://wiki.python.org/moin/GlobalInterpreterLock. (Visited: 26.03.2021).

[23] Duncan Uszkay. *How Shopify Uses WebAssembly Outside of the Browser*. URL: https://shopify.engineering/shopify-webassembly. (Visited: 24.01.2021).

[24] Gregory Chockler; Rachid Guerraoui; Idit Keidar; Marko Vukolic. "Reliable Distributed Storage". In: *Computer* 42 (4 2009), pp. 60–67. DOI: 10.1109/MC.2009.126.

List of Figures

List of Listings

List of Tables

Appendices

Appendix A

Calculations and test data

Benchmark 2 - Chapter 4 tables are included in this chapter. It is mostly used as a reference for improved comprehension.

.NET 5, Go, Python

	Read		Write	
Clients	Throughput	Latency	Throughput	Latency
1	1249.0	0.58	1665.66	0.47
2	3332.33	0.47	1499.0	0.59
4	3999.0	0.89	1850.85	1.55
6	3999.0	1.32	2351.94	1.90
8	4089.9	1.8	2856.14	2.13
10	3999.0	2.33	3528.41	2.58

3x Go (Wasm)

	Read		Write	
Clients	Throughput	Latency	Throughput	Latency
1	1999.0	0.37	1999.0	0.46
2	3332.33	0.42	3332.33	0.52
4	5999.0	0.82	4443.44	0.84
6	6665.66	1.97	4614.38	2.51
8	6955.52	2.25	4089.9	3.88
10	7406.40	2.64	5293.11	6.70

Table A.1: 2 out of 3 replies tests.(part 1)

3x Go (Non-Wasm)				
	Read		Write	
Clients	Throughput	Latency	Throughput	Latency
1	2499.0	0.28	2499.0	0.34
2	4999.0	0.28	4999.0	0.34
4	9999.0	0.28	7999.0	0.39
6	14999.0	0.28	8570.42	0.53
8	19999.0	0.28	9999.0	0.71
10	29999.0	0.46	10665.66	1.42
20	33332.33	0.5	11110.11	1.79
36	35999.0	0.88	11249.0	3.09
64	42665.66	1.39	11427.57	5.47
120	54544.45	2.13	11537.46	10.22
200	55554.55	3.49	11694.9	17.04
300	55713.28	5.44	11640.79	26.55

Table A.2: 2 out of 3 replies tests.(part 2)

Appendix B

Additional Graphs

This chapter contains supplementary diagrams that can be used to explain the variations in growth of the various Wasm implementations of chapter 4 (benchmark 1).

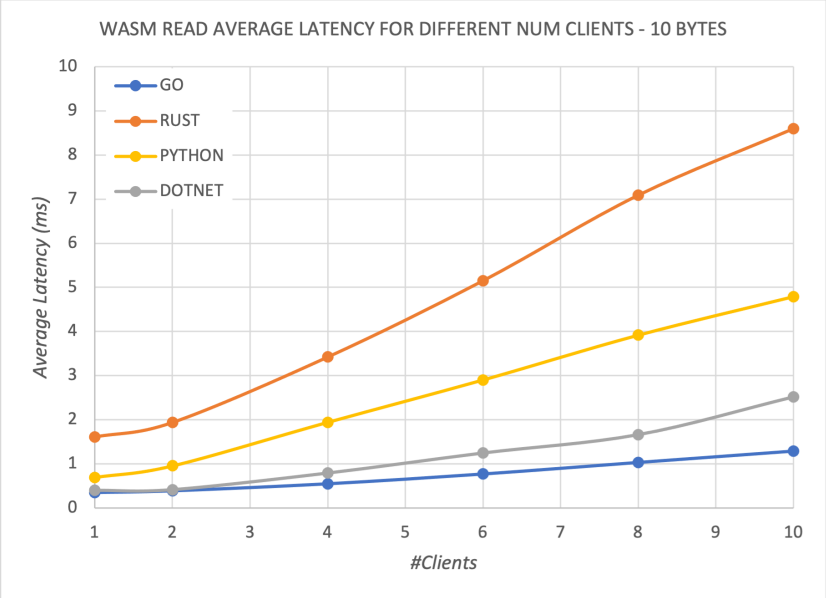

Figure B.1: Average latency experienced for different number of clients (Read-10Bytes)

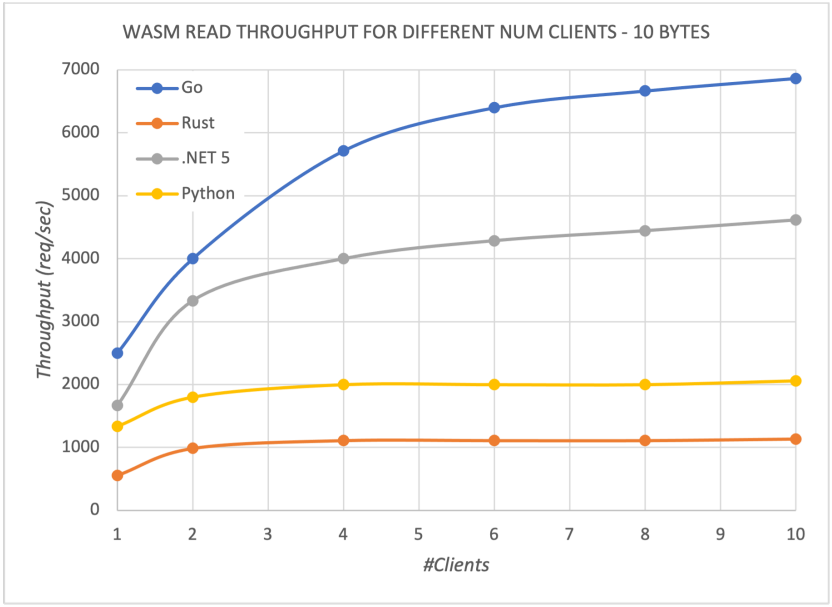

Figure B.2: Average throughput experienced for different number of clients (Read-10Bytes)

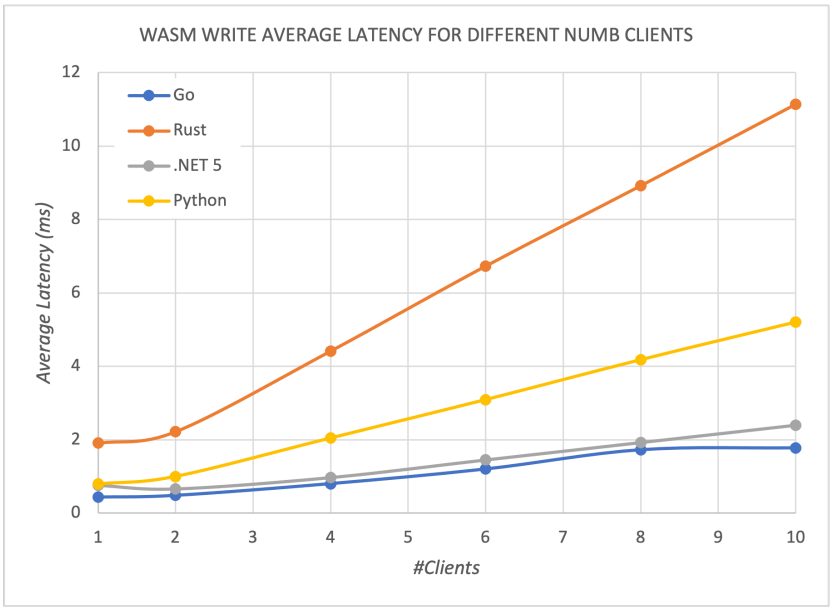

Figure B.3: Average latency experienced for different number of clients (Write-1MB)

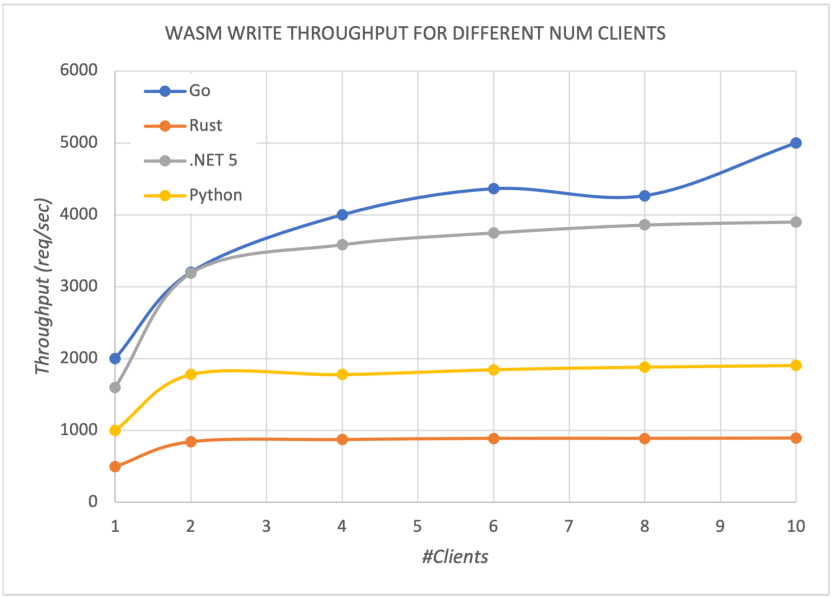

Figure B.4: Average throughput experienced for different number of clients (Write-1MB)

Appendix C

WasmInstantiate functions

```
1  func check(err error) {
2    if err != nil {
3      panic(err)
4    }
5  }
6  func WasmInstantiate(functions []string, wasmLocation string, preOpenedDirs
       map[string]string, stdoutPath string, stdinPath string, stderrPath
       string) (funcMap map[string]*wasmtime.Func, memory *wasmtime.Memory) {
7    engine := wasmtime.NewEngine()
8    store := wasmtime.NewStore(engine)
9    linker := wasmtime.NewLinker(store)
10
11   wasiConfig := wasmtime.NewWasiConfig()
12   if stdoutPath != "" {
13     err := wasiConfig.SetStdoutFile(stdoutPath)
14     check(err)
15   }
16   if stdinPath != "" {
17     err := wasiConfig.SetStdinFile(stdinPath)
18     check(err)
19   }
20   if stderrPath != "" {
21     err := wasiConfig.SetStderrFile(stderrPath)
22     check(err)
23   }
24
25   if len(preOpenedDirs) != 0 {
26     for dir, alias := range preOpenedDirs {
27       err := wasiConfig.PreopenDir(dir, alias)
28       check(err)
29     }
30
31   }
32
33   wasi, err := wasmtime.NewWasiInstance(store, wasiConfig, "
       wasi_snapshot_preview1")
34   check(err)
35
36   err = linker.DefineWasi(wasi)
37   check(err)
38
39   module, err := wasmtime.NewModuleFromFile(store.Engine, wasmLocation)
```

```
  check(err)
  instance, err := linker.Instantiate(module)
  check(err)

  in := instance.GetExport("_initialize").Func()
  _, err = in.Call()
  if err != nil {
    panic(err)
  }

  funcs := make(map[string]*wasmtime.Func)
  funcs["alloc"] = instance.GetExport("new_alloc").Func()
  funcs["dealloc"] = instance.GetExport("new_dealloc").Func()
  funcs["get_len"] = instance.GetExport("get_response_len").Func()

  for _, name := range functions {
    funcs[name] = instance.GetExport(name).Func()
  }
  mem := instance.GetExport("memory").Memory()

  return funcs, mem
}
```

Listing C.1: Golang's WasmInstantiate function

```
def WasmInstantiate(functions, wasmLocation, preOpenedDirs={}, stdoutPath=""
    , stdinPath="", stderrPat="")
{
    store = wasmtime.Store()
    linker = wasmtime.Linker(store)

    wasi_config = wasmtime.WasiConfig()

    if len(preOpenedDirs) != 0:
        for key, value in preOpenedDirs.items():
            path = Path(__file__).parent / key
            wasi_config.preopen_dir(str(path), value)
    if stdoutPath != "":
        wasi_config.stdout_file(stdoutPath)
    if stdinPath != "":
        wasi_config.stdin_file(stdinPath)
    if stderrPat != "":
        wasi_config.stderr_file(stderrPat)

    wasi = wasmtime.WasiInstance(store, "wasi_snapshot_preview1",
    wasi_config)
    linker.define_wasi(wasi)

    path = Path(__file__).parent / wasmLocation
    module_linking = wasmtime.Module.from_file(store.engine, path)

    instance_linking = linker.instantiate(module_linking)

    init = instance_linking.exports["_initialize"]
    init()

    instanceExports["alloc"] = instance_linking.exports["new_alloc"]
    instanceExports["dealloc"] = instance_linking.exports["new_dealloc"]
    instanceExports["get_len"] = instance_linking.exports["get_response_len"
    ]
```

```
    instanceExports["memory"] = instance_linking.exports["memory"]

    for name in   :
        instanceExports[name] = instance_linking.exports[name]

```

Listing C.2: Python's WasmInstantiate function

```
public WasmSingleton(string[] services, string wasmLocation, Dictionary<
    string, string> preOpenedDirs = null, string stdoutPath = null, string
    stdinPath = null, string stderrPath = null)
{
    using var engine = new Engine();
    using var store = new Store(engine);
    WasiConfiguration wasiConfiguration = new WasiConfiguration();

    if (stdoutPath != null)
    {
        wasiConfiguration.WithStandardOutput(stdoutPath);
    }
    if (stdinPath != null)
    {
        wasiConfiguration.WithStandardInput(stdinPath);
    }
    if (stderrPath != null)
    {
        wasiConfiguration.WithStandardError(stderrPath);
    }

    if (preOpenedDirs != null){
        foreach (KeyValuePair<string, string> entry in preOpenedDirs)
        {
            wasiConfiguration.WithPreopenedDirectory(entry.Key, entry.Value)
    ;
        }
    }

    using var module = Module.FromFile(engine, wasmLocation);

    using var host = new Host(store);
    host.DefineWasi("wasi_snapshot_preview1", wasiConfiguration);
    instance = host.Instantiate(module);

    ((dynamic)instance)._initialize();

    memory = instance.Memories.Where(m => m.Name == "memory").First();

    funcs = new Dictionary<String, Wasmtime.Externs.ExternFunction>();

    foreach (string func in services)
    {
        funcs[func] = instance.Functions.Where(f => f.Name == func).First();
    }
}
```

Listing C.3: WasmSingleton the .NET's WasmInstantiate function