DESIGN AND IMPLEMENTATION OF A WASM-BASED PROCESS EXECUTION SERVICE FOR A DISTRIBUTED-SYSTEMS MIDDLEWARE

MIKEL SOLABARRIETA ROMAN

Thesis supervisor: JORDI GUITART FERNANDEZ (Department of Computer Architecture)

Degree: Master Degree in Innovation and Research in Informatics (High Performance Computing)

Master's thesis

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

18/10/2023

Acknowledgements

To my family, friends, and Júlia.

Special thanks to Jordi, Louis, and Aratz.

In memory of Ian Denhardt, a.k.a zenhack.

Abstract

Distributed applications present a different set of challenges from their centralized counterparts, such as consensus, clustering, or message propagation. Wetware is a distributed systems middleware that provides a unified solution to the most common challenges of building distributed systems, simplifying the development and deployment of distributed applications. Wetware can be used as a standalone program, setting up nodes as independent processes and accessing them via object capabilities; or it can be embedded into applications that make selective use of its functionalities.

This thesis introduces a process executor for distributed applications as a novel component of Wetware, providing the means to run, manage, and communicate processes across a distributed cluster in a safe and performant manner through object capabilities. Process execution is based on WebAssembly, a binary instruction format that natively provides process isolation, portability, and performance; crucial characteristics of distributed processes in heterogeneous distributed systems. WebAssembly also introduces some challenges this thesis aims to solve, namely a lack of real parallelism and limited communication with elements outside its sandbox. Furthermore, this thesis develops tools to manage said processes and provide the means for inter-process communication, and demonstrates their validity and effectiveness through the development of a real-world distributed application that runs on a Wetware cluster through the process executor.

Next, the thesis analyzes the performance, resource usage, concurrency control mechanisms and scalability potential of the process executor for different workloads. As a result, we demonstrate that the executor is capable of running multiple processes across nodes on a Wetware cluster, with work-bound performance akin to high-level programming languages and capable of managing IO-intensive workloads.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Context

A distributed system is a group of autonomous interconnected computers that, while distinct and separated, operate together to achieve the same goal. They provide advantages over centralized applications: increased fault tolerance, reduced tendency towards some forms of bottlenecks, excellent horizontal scalability by adding more nodes, and benefit from concurrent application design. They also present a unique set of challenges that their centralized counterparts might not face: consensus [3], coordination between nodes is vital but not always simple [4], resource sharing [5], events may need to be ordered [6], faults can be dealt with but may require additional consideration [7], data must be kept consistent [8] and it needs to be stored and processed in separate nodes [9]. These issues have been studied in depth and all have open implementations of the state of the art solutions, that developers of distributed applications then have to individually integrate into their projects.

There has long been a global trend towards cloud computing. The industry is moving from self-hosted servers towards reliance on cloud provider services, usually from the Big Five [10]. This trend involves not only companies but individuals and organizations as well [11]. It is a natural move that provides convenience, simplicity, and economic savings but takes away control and power.

Cloud providers might deny or limit servicing companies or individuals for a wide array of reasons, or have a sudden change in billing policies. Even distributed applications are usually run in different nodes of the same cloud provider, as they provide computing nodes all across the world and automate tasks such as load balancing. The transition to web3 promised a step in the opposite direction by consolidating decentralization as a pillar for online platforms and services [12] and base it on peer-to-peer connections, but has had relatively low adoption and even its core definition and goals change depending on the source.

Wetware [13] is an open-source tool created with the purpose of making building and deploying peer-to-peer distributed applications easier, and allowing them to run (almost) anywhere. It groups solutions to common distributed system problems in a single package and streamlines the deployment of nodes, developing some of them in-house and selectively using existing ones when appropriate. One of its core components is the result of this thesis: a process executor that enables running isolated processes, inter-processes communication, and process management across a whole cluster. The security and flexibility of the middleware are complemented by the process isolation and portability provided by WebAssembly, the language chosen for the processes of the executor.

1.2 Motivation

The author personally uses tools such as Syncthing [14] or Anytype [15] in a daily basis, both distributed P2P applications centered around synchronization that have most likely faced many of the issues Wetware attempts to solve. They provide developers and users autonomy on how their software run, and where it runs. They provide independence from centralized resources, which is one of the main pushing factors behind this project.

In order for Wetware to work as intended, it needs a way of running and managing processes. It must be done respecting shared resources in a way that allows the rest of the middleware to keep functioning, with as little overhead as possible. Those processes need to have access to the middleware too, and have the means to communicate with each other. Meeting this requirement will complete a fundamental part of the middleware and take it closer to being ready for adoption.

Finally, while Go is not a language designed for High Performance Computing and lacks some of the low-level tools to make the most out of the hardware resources available; it does present a great opportunity to study and analyze how the language, as well as other technologies involved in this project, behave and can be used to maximize performance in HPC.

1.3 Goals and Contributions

This work consists of the **design and development of a process executor capable of running and managing programs written in WebAssembly and expose its functionality through object capabilities**. The executor is to be integrated into the wider Wetware middleware, as well as validated and benchmarked thorough the development the implementation of a real application. This thesis takes advantage of the state of the art of its two main programming languages to enable the use of Cap'n Proto capabilities from WebAssembly processes. Lastly, it demonstrates some of the benefits of this Wetware executor by integrating a consensus algorithm into a real application.

To the best of our knowledge, Wetware is unique on its field and is marginally different from other distributed system middleware in the variety of services it provides as well as its flexibility. The executor is a novel contribution into such a middleware, parting from an isolated process execution environment, running on a runtime particularly well fit for concurrent applications and exposed through an object capability.

1.4 Outline

To start off, chapter 2 will provide the background of this project, which lay the technological foundation for the development of the thesis by shedding a light over the different concepts and tools applied on this project. Chapter 3 contains the specification of the technical objectives and the requirements of this project, as well as laying out the work plan that was followed. Chapter 4 explains the various methodologies used during the thesis development, explaining communication frequency and channels with the thesis director, Wetware team and others; as well as contingency plans, development and testing. The latter two are explored in depth in chapter 5, which contains detailed information on the development, testing and verification of each of the software components that were contributed by this thesis. Chapter 6 evaluates the validity and performance of the development, analyzing the results and identifying potential improvement routes. Chapter 7 sums up the rest of the document and provides closing thoughts. Finally, appendix A contains component diagrams that reflect parts of the final codebase and are useful as complementary resources when studying segments of code.

Chapter 2

Background

This chapter provides the background on concepts and technologies of special relevance for this thesis, laying the grounds for the project, its development, and evaluation.

2.1 IaaS and PaaS

Infrastructure as a Service and Platform as a Service are two core concepts of modern cloud computing. IaaS refers to the offering of computing, storage, and networking resources as a service, usually based on charging the client proportionally to the amount of resources they use. IaaS is usually provided through some level of virtualization or containerization, increasing deployment flexibility, allowing fine-grain control and accountability of resource usage, as well as providing additional security and stability over bare-metal applications or applications running directly on the OS. IaaS conveniently provides high-performant infrastructure, transparent vertical and horizontal scalability, built-in security, data locality with servers spread across multiple locations, and potential cost reductions compared to setting up one's own infrastructure. These factors have made it appealing for companies, organizations, and individuals to migrate or deploy their production environments to cloud computing: cloud provider's hardware through IaaS. This appeal has marked a global trend for over a decade, which resulted in IaaS public cloud services reaching a market cap of $120.3 billion in 2023 [16]. The biggest IaaS providers currently include Amazon Web Services, Microsoft Azure, Google Cloud, Alibaba Elastic Compute Service, Digital Ocean, and Linode.

Platform as a Service is a level of abstraction over IaaS, where a software service is offered with abstraction from the hardware it is running on. Web page hosting is arguably one of the clearer examples. Another example more relevant to this thesis is providing middleware as a service such as the messaging service provided by AWS. Middleware, in this case, is defined as a software layer between the operating system and the application, abstracting the application from interacting with the operating system and the hardware.

While undoubtedly useful, both IaaS and PaaS do have drawbacks. Service providers can discriminate who to provide their services and for what price. They can stop providing service at any point, potentially leaving the most dependent users with no infrastructure whatsoever. Overuse of the resources can result in an unexpectedly inflated bill. Users have to trust their provider to run their software unaltered. Lastly, providers know what software is being executed. All these can be comprised of two terms: control and privacy.

Many modern applications require some level of distribution and process management, often made easier through the use of IaaS and PaaS. The next section will delve into Wetware, a project with the objective of facilitating those two features to developers through a middleware that can be easily deployed anywhere.

2.2 Wetware

Wetware is a modular distributed systems middleware intended for writing secure, scalable, and performant distributed applications [13]. Developing distributed applications often requires an investment of resources into integrating some distribution features, which often revolve around communication, clustering, and orchestration. Wetware provides features out of the box so by just importing them into an application they are ready to be used. Specifically, Wetware provides the following explicit features: peer discovery, clustering, inter-process communication, storage, message propagation, a service registry, and, as of now, process execution and management. Under the hood, it also takes care of tasks such as content routing or self-optimizing overlay networks.

These tools are exposed through object capabilities: transferable rights to perform operations on an object. Objects, in this context, refer to every accessible Wetware component: ranging from an entire node to a communication channel. Processes interact with Wetware through Cap'n Proto, an object capability protocol, and remote-procedure calls. Internally, Wetware is designed with separation of concerns in mind. Different feature groups are provided by different components, implemented as Go structs. As an example, dialing is performed by a *Dialer* component, and peer-to-peer connections are managed by a *Host* component. This thesis introduces the *Executor* component: a structure capable of receiving, running, and managing WASM processes.

Generally, processes will interact with Wetware through an object capability representing a node. A node is a unitary process with all the necessary services and components to provide every aforementioned feature. It exposes these features through so-called "node syscalls", RPC calls that perform one action and return the result in the form of a capability. Let's go over some examples. The *View* call will return the local view of a node in the form of a list of object capabilities, each representing a different node. The *Exec* call will create a new process and give the caller access to it through a *Process* capability, which can then be used to manage the newly created process. *Pub* will publish a message, and *Sub* will wait until it receives a message. Figure 2.1 illustrates how processes can interact with a Wetware node, whether they are running on the same machine or a different one. Figure 2.2 shows an example of process A sending process B a message through Wetware's publish/subscribe service. The node can then allow them to interact with the whole cluster. It is also possible to embed Wetware into applications, by importing the library into a project and initializing features from within.

Having all of Wetware's features exposed through capabilities has two main advantages:

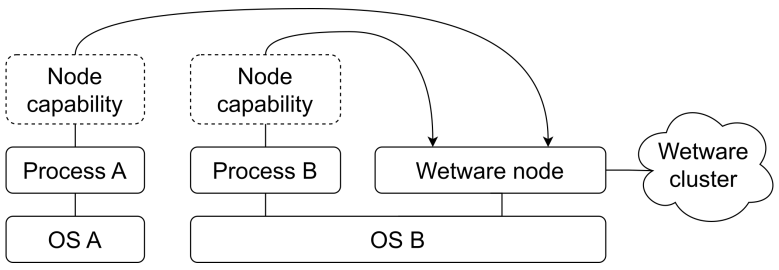


Figure 2.1: How processes use Wetware

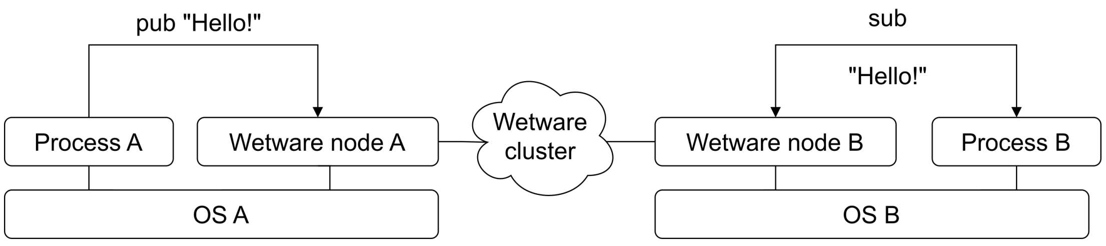


Figure 2.2: Publish/subscribe example with Wetware

1. They can be used from anywhere. Any holder of the capabilities can make use of them.
2. Any programming language with a Cap'n Proto implementation can use them.

Combining Wetware's portability and accessibility makes a more flexible tool that will hopefully make the lives of distributed application developers easier. This accessibility is an excellent fit for one of Wetware's main objectives: portability. Cluster may contain heterogeneous members, running on different operating systems and architectures. Wetware is designed to run on POSIX-compliant operating systems; such as Linux, BSD, macOS, or Plan9.

Applications built with Wetware have, by default, unstructured decentralization [17] due to how Wetware clusters are formed. Nodes have no guaranteed permanence in the network, fixed connection topologies, or neighborship relations. It is still possible, however, to build dependable applications by handling resiliency at an application layer, which will be explored in section 5.7.6. Nodes communicate through peer-to-peer connections established and managed through

LibP2P libraries. They may use different transport protocols but usually default to TCP and QUIC.

Wetware is an open-source project currently in its early stages, developed by a few volunteers, and licensed under both the Apache and MIT licenses. Wetware is written in Go, favors portability, and is designed in a modular manner so that users can choose which of its components to use.

In summary, Wetware is a portable and accessible distributed systems middleware, which natively provides core components of distributed applications. This thesis develops process execution features for the middleware, and tests those features through the implementation of real, distributed applications.

2.3 Distributed Systems

As mentioned in the introductory chapter, a distributed system is a group of autonomous interconnected computers that, while distinct and separated, operate together to achieve the same goal. Distributed systems may have different degrees of decentralization depending on how nodes connect to each other, with more centralized (figure 2.3a) or decentralized (figure 2.3b) network layouts. Distributed applications also have varying degrees of centralization depending on their design. Some may distribute work and centralize control, while others might have fully decentralized logic. On the same topic, distributed systems can be categorized as at least the following types: client/server [18], multi-tier [19] and peer-to-peer [20]. While Wetware provides the tools to develop the latter, it is possible to have internal P2P systems exposed through, for example, a client/server model.

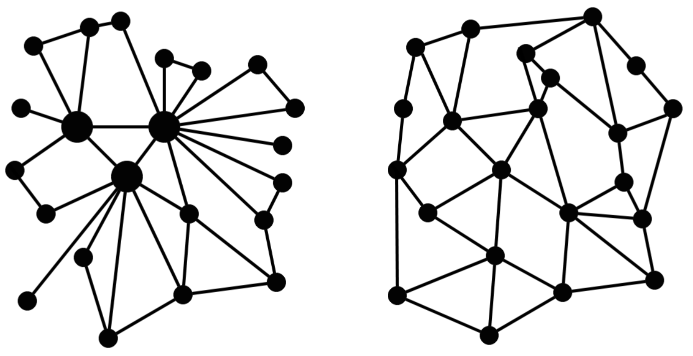

(a) Lower decentralization (b) Higher decentralization

Figure 2.3: Decentralization levels

Distributed systems come with their own set of benefits and disadvantages. Benefits include, among others:

- *High horizontal scalability*: additional workload can be accommodated by simply adding more nodes.
- *High availability*: data and resources can be found in multiple nodes.
- *Concurrency*: distributed applications are concurrent in nature.
- *Fault tolerance*: there is no central point of failure.
- *Load balancing*: nodes might distribute load among them to avoid overworking any one of them.
- *Flexibility*: nodes can be heterogeneous and run in different environments.
- *Locality*: data can be in the node closest to where it is needed.

While disadvantages include:

- *Complexity*: developing distributed applications is generally a more complex task than developing centralized applications.
- *Consistency*: data may need to be kept consistent across nodes.
- *Networking issues*: overhead, latency and network failures can have a very negative impact.

- *Consensus*: agreeing on values can be hard, and consensus algorithms can be complex.
- *Failures*: node fault tolerance bring complex failure scenarios.
- *Communication bottlenecks*: message passing through a distributed system is generally slower than sharing memory.

Distributed Systems are a very relevant field of computer science and they are present in most of the services we use day to day, from chatting applications, to banking systems and supercomputer systems. Peer-to-peer distributed systems are arguably less common in commercial applications but are nonetheless invaluable and present in many other real-world applications: file sharing, IoT devices, Virtual Private Networks... The latter are even one of the inspirations behind the creation of Wetware [13]. Peer-to-peer has its pros and cons, but is generally a fit choice for systems with a high degree of decentralization. Wetware handles P2P connections through the LibP2P networking stack.

2.3.1 LibP2P

LibP2P is a modular network stack. The modularity is due to the project being a combination of specifications, protocols, and libraries that form LibP2P as a whole. One of its main objectives is to provide a set of P2P functionalities that can be cleanly separated so that developers use only what they require. It is fully or partially implemented in many languages, including but not limited to Go, JavaScript, Rust, C++, Haskell, Java, and Python. It was originally a part of IPFS [21] but eventually became a standalone project.

LibP2P nodes use multiaddresses, which contain information about the network addresses and protocols. An example of a multiaddress is `/ipv6/::1/tcp/4242`. A node listening in that multiaddress is expecting connections to come from the `::1` loopback address and TCP connections to go to the 4242 port.

Wetware makes use of *go-libp2p*, the de facto LibP2P implementation in Go. It has ample documentation for each of its many components: transports, peer identity, content routing, security... It handles multiple communication protocols in a single established connection through stream multiplexing and supports a variety of transport protocols. Additionally, it natively provides techniques such as NAT hole punching, which will benefit Wetware's portability goals.

2.3.2 Consensus

Consensus comes out often when discussing potential requirements of Wetware users writing distributed applications: be it a distributed database, process coordination, fault tolerance, result propagation, etc. Integrating consensus algorithms into a program might take more low-level tweaking than other Wetware features, for that reason it is implemented at an application level and needs to be fully compatible with Wetware processes. While designing a consensus algorithm is outside the scope of this thesis, proving that consensus algorithms can be implemented and run on Wetware processes is not. Such demonstration must meet the following requirements:

- *WASM compatibility*: the consensus algorithm implementation needs to have WASM as a compilation target.
- *Cap'n Proto as a transport*: peers must communicate by sending messages through capnp RPCs.
- *Asynchrony*: the consensus algorithm might run in the foreground or background of processes, blocking execution only when explicitly indicated by application developers.
- *Performance*: The less time is spent maintaining a replicated log and managing the cluster, the better. Asynchrony is but one of the factors that will help achieve this.
- *Stability*: Wetware environments are not necessarily stable or fault tolerant. Peers need methods of reconnecting with one another, which need to be implemented for the Wetware middleware.

2.3.2.1 Raft

Raft is arguably the standard for distributed consensus algorithms since its publication on the notable article "In Search of an Understandable Consensus Algorithm" [3], when it took the spotlight from the Paxos [22] algorithm family that held the standard for years before that and are still widely used [23]. Its main advantage over Paxos is its simplicity, proven correctness, and completeness, e.g. by defining how a re-joining peer might catch up with the rest of peers. It was designed with tolerance to network and peer failure in mind.

Consensus is split into two main phases: leader election and log replication. The cluster has one leader at a time, chosen among peers of the cluster. Elections are held periodically, leadership periods are split into terms that end when the leader loses consensus or disconnects from the network. This leader is the only node authorized to operate on the log, and only by means of appending. Although it may cause bottlenecks for write-heavy applications with distributed writers or read-heavy applications that always require the latest value, having a single write-point ensures log integrity.

This is where log replication comes in. The append-only log is then replicated between peers, who can propose new values that are then voted, accepted or rejected, and appended to the log by the leader if accepted. The replication phase goes on until the leadership term ends due to any of the aforementioned reasons. The full Raft specification [3] defines the behavior for all considered scenarios.

After examining multiple Raft implementations, etcd's Raft was selected [24] as the main building block for the Raft object capability developed during the project and described in section 5.7.5. Etcd is a distributed key-value storage written in Go and used in projects such as Kubernetes. The three main characteristics that made it a good fit for this master's thesis are its modularity, transport agnosticism, and storage agnosticism. All three of them are essential for WASM compatibility.

Modularity refers to how the source code is structured in the project. If a Go module includes any code that can't be compiled to WASM, e.g. code that performs system calls unavailable in the WASM runtime, the whole module becomes unusable from WASM. As an example, some of the libraries tested during the development of this project, and even Wetware modules, bundled WASM-compatible features and WASM-incompatible code, e.g. setting up an HTTP server listening on a TCP socket, in the same package. That package could not be compiled to WASM. Etcd's Raft implementation has very granular packages, and the core Raft functionality fully supports WASM as a compilation target. Transport and storage agnosticism is also crucial due to WASM process isolation, which is further developed in chapter 2.5.2 and section 5.3.

The modular design of etcd's Raft combined with the clearly defined interfaces for the unimplemented parts, such as the transport, makes it relatively straightforward to integrate it with the different technologies used in this thesis. It does, however, come with some drawbacks.

In the first place, assembling is required. The modules are there, but tying them all together is up to the user. The main process loop needs to be implemented and requires some attention to do it properly. As a counter-argument, etcd provides ample documentation and its extended usage has created a multitude of examples. In the second place, etcd Raft's performance can degrade on certain networking environments [25], even if the main cause is Raft's strong leadership design. This might mean some very specific Wetware use cases require the integration of other consensus algorithms, which will benefit from the knowledge gained from the Raft integration.

In summary, the benefits of both Raft and etcd's implementation far out-weight the drawbacks, mainly simplicity, flexibility, etcd Raft's transport-agnosticism, having been heavily tested in real-world applications and the ample documentation about them.

2.4 Remote Procedure Calls

Remote Procedure Calls, or RPCs, are a form of inter-process communication widely used in distributed computing to cause a procedure to run on a remote space, such as a different node in a cluster. RPC providers usually follow a client/server approach where the caller (client) holds a stub representing the callee (server) and calls its methods as if it were an object. To illustrate, a node A might cause a node B to perform the $doThing$ procedure by holding a stub b of B and calling it as follows: $b.doThing()$. The most widely extended RPC framework is arguably gRPC [26], focused on high performance and based on the popular Protocol Buffers [27]. Wetware uses a successor of Protocol Buffers: Cap'n Proto.

2.4.1 Cap'n Proto

Cap'n Proto is an open-source Remote Procedure Call framework, as well as a serialization format. Its main focus is performance, as it was designed to be used for a wide range of applications; from communication between members of distributed systems to inter-process communication. This high performance focus shows in many features, including but not limited to:

- *Zero-copy deserialization*: It allows serialized data to be used directly from its serialized representation without the need for unmarshalling it into a data structure.
- *Incremental reads*: It is not necessary to have received a full Cap'n Proto message to be able to start reading it.
- *Random access*: Any of the fields of a message can be read without going through the previous messages.
- *Inter-process communication*: capabilities shared by two processes running on the same machine may use it to communicate by sharing memory.
- *Time travel promise pipelining*: results of a RPC can be used before the request has been sent, through result promises. Illustrated in figure 2.4. If A wants to get C from B, it can start a B RPC to retrieve C and begin it before the call even reaches B. Results might also be received in any order, although they will be perceived by the receiver as having arrived in

order [28]. Both these factors are beneficial not only to performance but also to potentially deal with network bandwidth or latency issues.

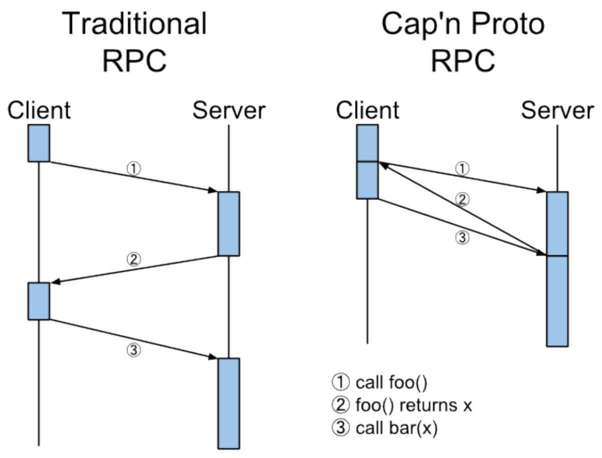


Figure 2.4: Promise pipelining (source: [1])

Cap'n Proto is based on object capabilities: transferable rights to perform operations on an object. The object capabilities security model is based on references to objects through which actions can be performed. As an example, an actor A exposes two methods through a capability: *sendMessage* to receive messages, and *start* to start some arbitrary functionality. Any actor with a capability holding a reference to A can perform any of these actions: send a message to A, or make A start some action. In the object capabilities security model, objects can only be interacted with through messages sent on references. These references can be obtained in one of the following four ways:

1. Initialization: Actor A already holds a reference to capability B.
2. Parenthood: Whenever actor A creates an object B, it gets a reference to it.
3. Endowment: Whenever actor A creates an object B, it can endow it any of the references it holds, similar to initialization.
4. Introduction: If actor A holds a reference to objects B and C, it can send the reference to either of them to the other one through a message.

The Cap'n Proto communication and serialization specification is language agnostic, which is in great interest of Wetware as it favors portability. Its schema language is designed with extensibility in mind, allowing schemes to evolve over time while maintaining backward compatibility. It was created by Kenton Varda, who was the main author of protobuff v2. Cap'n Proto is a CPU-bound protocol, making it a great fit for WebAssembly which at the moment has no official support for hardware acceleration. Used mainly in Distributed Computing use cases by companies like Cloudflare and Sandstorm. [29]. Its most widely used implementation, in C++, is developed and maintained mostly by Cloudfare [30]. It supports both packed and binary encoding, packed being the best performing one in most cases [31]. Internally, Cap'n Proto uses what it calls "The Four Tables" to manage references and inter-VAT connections [32], having four separate tables (*Questions*, *Answers*, *Imports* and *Exports*) per remote VAT.

2.5 Programming languages

This thesis is built upon two programming languages: Go and WebAssembly. This section will go over the two, underlining their most relevant characteristics and how they relate to each other.

2.5.1 Go

Go is a multi-paradigm high-level programming language, developed by Google and originally designed by Rob Pike, Robert Griesemer, and Ken Thompson. It is statically typed, imperative, garbage-collected, and has memory-safety features. Go has built-in dependency management and a suite of tools for testing and profiling. Go has manual memory allocation, and provides some level of memory management isolated from the garbage collector through arenas. It has a syntax somewhat similar to C, and natively supports a wide array of operating systems and architectures, listed in table 2.1. Go also supports cross-compilation, making it easy to generate binaries for multiple platforms from a single source. In addition to the de facto native compiler, *gccgo*, *gollvm* and *TinyGo* are also widely extended. The latter, TinyGo, was initially used in this thesis to generate WASM

	Architecture						
	386	AMD64	ARM		RISC-V	WASM	others
Operating System	32b	64b	32b	64b	64b	64b	64b
AIX	no	no	no	no	no	no	$*^1$
Android	yes	yes	yes	yes	no	no	no
Darwin	no	yes	no	yes	no	no	no
Dragonfly	no	yes	no	no	no	no	no
FreeBSD	yes	yes	yes	yes	yes	no	no
illumos	no	yes	no	no	no	no	no
iOS	no	yes	no	yes	no	no	no
JavaScript$^\diamond$	no	no	no	no	no	yes	no
Linux	yes	yes	yes	yes	yes	no	$*^2$
NetBSD	yes	yes	yes	yes	no	no	no
OpenBSD	yes	yes	yes	yes	no	no	no
Plan9	yes	yes	yes	no	no	no	no
Solaris	no	yes	no	no	no	no	no
WASIP1$^\diamond$	no	no	no	no	no	yes	no
Windows	yes	yes	yes	yes	no	no	no

$*^1$	PPC(64b)
$*^2$	Loong(64b),MIPS(32b,64b,LE&BE),PPC(32b,64b),S390X

Table 2.1: Operating systems and processor architectures supported by Go

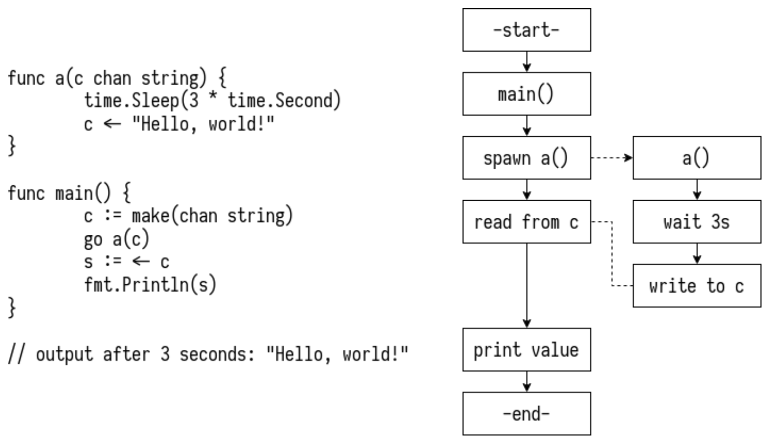
bytecode from Go sources due to its focus on embedded devices and WASM. It was dropped somewhere along the development process in favor of Go's native compiler for technical reasons, explained in section 5.3.1.

Note that the "architectures" listed in the official Go documentation also refer to instruction sets, and "Operating Systems" refer to runtimes on top of operating systems, marked with $\diamond$. Go is often used for distributed application development for both its performance and its built-in concurrency design due to the concurrent nature of distributed applications.

Go's concurrency model is a natural match for Wetware's heavily asynchronous code base, has low code complexity without compromising performance, natively supports WASM as a compilation target, and generates a single, statically linked, binary file. The last point, combined with the wide range of architectures Go supports, makes Wetware node deployment simple.

This thesis was started using the latest available Go version: 1.20. It then transitioned to 1.21 while the version was still in development before its stable release on August 2023.

2.5.1.1 Concurrency in Go



```
func a(c chan string) {
        time.Sleep(3 * time.Second)
        c <- "Hello, world!"
}

func main() {
        c := make(chan string)
        go a(c)
        s := <- c
        fmt.Println(s)
}

// output after 3 seconds: "Hello, world!"
```

Figure 2.5: Go concurrency example code and flow diagram

Go supports concurrency through goroutines, channels, selects, and other more traditional tools such as locks. Section 5.1.3 goes over the Go scheduler and the potential performance impact of each of these tools, while this section focuses on going over the basic features and syntax. Goroutines are effectively lightweight threads managed by the Go runtime. They are relatively cheap to create and run on a CSP-like model. Goroutines communicate through channels, which are message-passing interfaces.

> Don't communicate by sharing memory; share memory by communicating.
>
> Rob Pike

Figure 2.5 contains an example of a short Go program of the main goroutine spawning a new goroutine to run the *a* function, and receiving a message from it through a channel. Channels can be either buffered or non-buffered. Buffered channels won't block the execution of the reader or writer while the channel is not empty or full, respectively. Unbuffered channels will block until the reader and writer are both ready. The example provided in figure 2.5 has an unbuffered channel. Buffered channels are created with `make(chan <type>, <buffer size>)`.

Concurrent Go applications usually employ the `select` keyword, an asynchronous analog to `switch`. The example shown in listing 2.1 waits for the first of the cases to happen: either a message will be received through the channel, the timeout will expire or the context will be canceled. The WebAssembly runtime used in this thesis binds processes to contexts this way, stopping WASM processes when the context is canceled.

```go
someChannel := make(chan struct{})
...
select {
    case <- someChannel:
        // message arrived
    case <- time.After(30 * time.Second):
    case <- ctx.Done():
        // message didn't arrive
}
```

Listing 2.1: Select usage example

Lastly, Go functions often receive a context as their first argument, of type `context.Context`. The context is not only useful for passing information to newly created goroutines, but is an elegant way of orchestrating the stop of multiple goroutines in different execution stages. The example of listing 2.1 shows a goroutine waiting, among other things, for its context to be canceled. The context cancellation will have originated on another goroutine, for example in the main goroutine due to a keyboard interrupt. Contexts will be present in many of the listings throughout this document. As the Go saying says: "context should flow through your application".

2.5.1.2 From Go to WebAssembly

Go allows choosing WebAssembly as its compilation target. The source code is embedded with the garbage collector and the runtime scheduler to form a single WASM bytecode file interpretable by any WebAssembly runtime. TinyGo allows the option of configuring the scheduler that will be embedded into the WebAssembly, but standard Go supports no such feature. This somewhat limits our performance tuning options but is worth it overall. Go sources can be compiled into WASM through the following command: `env GOOS=wasip1 GOARCH=wasm go build -o main.wasm main.go`.

2.5.2 WebAssembly

WebAssembly, or WASM, is a programming language created in 2015 as a means to provide fast, portable low-level code on the Web [33]. More accurately, it is a portable intermediate code format that is then executed by a WebAssembly runtime. While it was originally intended to run only in web browsers, it was quickly adopted outside that use case for its portability and its capacity to provide near-native performance [34], even if it has higher prediction misses and cache pressure than native languages [35].

WASM is commonly a compilation target for other programming languages such as C, Rust, or Go. Languages with lightweight runtimes tend to provide the best performance when compiled to WASM [36]. WASM programs are compiled and loaded into the runtime as modules, where they can import functions from other WebAssembly modules; even if they were originally written in a different programming language. The WebAssembly runtime can also expose so-called "host functions": functions that when called by the WASM guest, are performed by the host. There are multiple WebAssembly runtimes, implemented in different languages and with different design considerations. Any one of them should be able to run any WASM program as long as both meet the standard specification. Wetware uses Wazero, a zero-dependency runtime, as its WASM runtime. Figure 2.6 illustrates an example of programs written in multiple program languages that, once compiled to WASM, can be run by any WebAssembly runtime. These runtimes can also run on the operating systems of their choice, potentially making WebAssembly an extremely portable language.

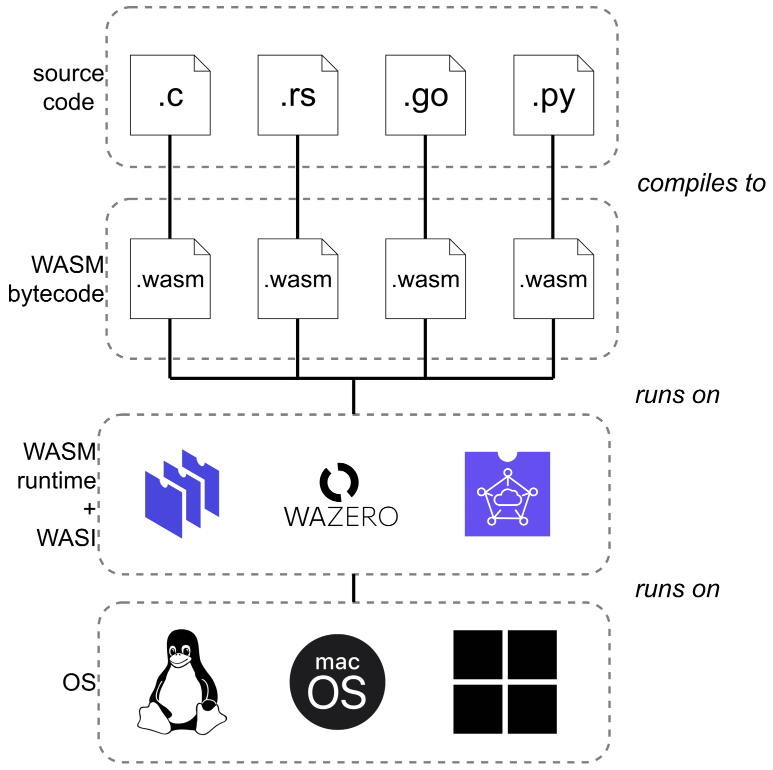


Figure 2.6: WASM component hierarchy

WebAssembly processes are single-threaded and support no real parallelism, although this might change as the standard of one of its components, WASI, evolves. Wetware brings parallelism into its WASM runtime by allowing running processes to create other processes that can run in parallel to it. Scheduling is usually embedded into the WASM bytecode by the compiler, leaving concurrency support up to the source language. Some compilers such as TinyGo allow specifying what scheduling strategy to use for compilation. Garbage collection is also embedded into the WASM target. WASM doesn't support hardware acceleration, but there are examples of runtimes purposely breaking isolation to use GPU acceleration in a WASM application [37].

The memory of a WASM process is represented as a resizeable byte array. The memory is split into pages, each one of 64KiB. Processes can grow their memory by growing their number of pages, but can never access memory outside those pages. This isolates the process and prevents unauthorized access or resource utilization. This memory is disjoint from user space, the runtime engine, or the execution stack, keeping the process from jumping to inaccessible locations. The memory array is initialized by the WASM runtime before the process starts, and freed after the process ends. From the perspective of the OS, the memory of the WASM process is a subset of the memory of the program containing the runtime.

The effectiveness of the process isolation depends on the correct implementation of specific runtimes. If done incorrectly, not a remote possibility for such a complex specification, processes can escape the sandboxing and perform attacks such as stack overflow attacks, as demonstrated in [38].

WebAssembly is neither just for the web nor assembly code, but provides built-in isolation, close to native performance, and a very high degree of both portability and flexibility; making it a great fit as the programming language for distributed systems built with Wetware.

To further illustrate the specific use case of Go and WASM in this project: Go source code has to go through several steps until it can be run as a WASM process. Figure 2.7 enumerates the states the program goes through, as well as the action that transitions it from one state into the next. Different WASM runtimes and Go compilers might produce slight variations, the figure represents the process using Go's native compiler and the Wazero WASM runtime.

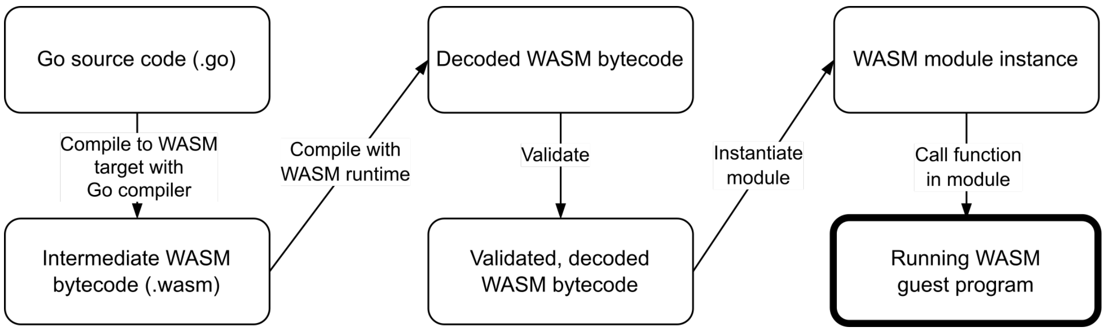

Go source code (.go)
Compile to WASM target with Go compiler
Intermediate WASM bytecode (.wasm)
Compile with WASM runtime
Decoded WASM bytecode
Validate
Validated, decoded WASM bytecode
Instantiate module
WASM module instance
Call function in module
Running WASM guest program

Figure 2.7: Steps to go from Go source to running WASM guest process

As a closing note, one of the objectives of this project is to allow WASM processes to interact with object capabilities the same way OS-level processes do (figure 2.1). This will imply breaking the sandboxing and considering its possible effects.

WASM provides built-in process isolation, with very granular control over the resources granted to the process, potentially high performance, and extreme flexibility for Wetware users. Combined with Object Capabilities, Wetware applications can be written in any programming language that has WASM as a compilation target and has Cap'n Proto support.

2.5.2.1 WASI

The WebAssembly System Interface, or WASI, is a specification with the goal of enabling WASM guest processes to interact with elements outside their sandbox, e.g. sockets, file systems...

Without WASI, WASM processes have no defined way of performing asynchronous operations, which could potentially lead to WASM runtimes designing and implementing their own without standardization, breaking one of WebAssembly's main advantages: portability. WASI is still under development and might be subject to change at any time, with the current version often being referred to as WASI Preview 1 or WASIP1. WASI exposes the aforementioned features through a POSIX-like API, often abstracting them from the application and providing an experience akin to native execution.

2.5.2.2 Wazero

Wazero [39] is a zero-dependency WASM compiler and runtime written in Go. No external dependencies means no usage of Cgo, a Foreign Function Interface (FFI) that allows calling C functions from Go code. Cgo is usually used to implement performance-demanding functions that benefit from manual memory management and low-level control. Cgo can provide better or worse performance depending on its specific use, but it does limit portability and cross-compilation [40].

By not relying on Cgo, Wazero can be statically linked and produce a clean binary, something that has proven to be a challenge with FFI. Statically linked programs are generally easier to deploy, which is one of the main requirements of the wider Wetware project.

The Wazero compilation engine implements its own assembler, which allows parallelization of WebAssembly compilation, something the WebAssembly binary is optimized for [41]. Assembled code is then executed as native Go code, and is protected against asynchronous preemption of the WASM program by another goroutine. It does so by grouping instructions as Go Assembler functions, which are considered unsafe to preempt by the Go runtime as of Go 1.20 [41].

Many of Wazero's features are exposed as a Go interface. As an example, WASM guests can be granted access to a file system. Said file system is in truth a Go interface with file system access syscalls mapped to its methods in the guest. When creating a WASM guest process in Wazero, users have the option of giving it a file system through the `WithFS(fs.FS)` option. `fs.FS` is an interface with methods such as `OpenFile(path string, flag Oflag, perm fs.FileMode) (File, Errno)` mapped to their respective syscalls, in this case `sys_open`.

The Wazero project has a very active community and open communication with the development team through Slack channels. It's open-sourced, hosted on GitHub, and allows for both open discussion and submission of pull requests. The team has proven to act fast upon found issues, which ended up being vital for this thesis as explained in section 5.3. It is also vital for fixing potential security vulnerabilities that may allow processes to break their isolation or exploit the runtime in any way. Many of the WASI features for Wazero are being developed in a separate repository [42] as an extension to Wazero, while some vital features are integrated into the main repository.

Ironically for the context of this thesis, Wazero generally performs worse in performance benchmarks regarding execution time when compared to the rest of the state-of-the-art [43]. The tests were performed in x86 architectures, and the source speculates on the potential for better results on ARM architectures. It's worth noting that the Wazero team has recently shifted their focus towards performance, and that behind our suspected reasons for the lack of performance there are relevant advantages for Wetware. The two main potential factors are:

- *No dependencies.* There is a choice to be made about performance: on one hand Go is generally slower than C. On the other hand, using C from Go implies careful usage of FFI and it will introduce some overhead [44]. Wazero opts for tackling the former and aiming for the most performant Go code while abstaining from using Cgo for external function usage [45].
- *Maturity.* Wazero is still in its early stages, thus there is potential for significant performance improvements. In fact, its first official release was on September 1, 2022: slightly more than a year before this thesis' finalization. The release notes of the latest versions as of September 12, 2023, show performance as one of the focal points.

2.6 State of the Art

There are some programs and middlewares that can be compared to Wetware and the executor developed in this thesis, if only in feature subsets.

CORBA [46] is a standard with the purpose of facilitating communication and collaboration between processes distributed across networks and running on heterogeneous hardware and operative systems, through an object-oriented approach [47]. It shares its purpose with some of Wetware's components. It also shares the purpose of running on heterogeneous language, as well as the independence from programming languages which it achieves through specification, while Wetware does it through object capabilities.

Boinc [48] allows individuals to donate the computing power of their devices to research projects, creating a centralized but distributed application with a central control point but distributed worker nodes. It is a middleware for volunteer computing, arguably one of the most used distributed system-related middleware. The Ethereum Virtual Machine is a set of distributed clients acting as a virtual machine to execute programs defined on its own programming language [49]. Boinc offers a subset of the capabilities Wetware intends to offer, and the Ethereum implementations also pack some of the features in some cases. They are, nevertheless much more specific, lacking accessibility and requiring a harder integration process for a fraction of the benefits.

Sledge [50] is an edge computing middleware that provides a WASM process executor, and is perhaps the one that comes closest to what this thesis is trying to achieve. Yet, the two are fundamentally different and only coincide in supervising process execution as well as host-guest communication. Sledge is optimized for bursty client rates and short-lived computations. Contrary to Wetware, which delegates scheduling to Go and Wazero, Sledge implements its own runtime and has fine-grain control over the scheduling.

Suborbital's Sat [51] is a WASM process executor based on Wasmtime, WasmEdge, and Wasmer WebAssembly runtimes. Sat is designed for edge computing and gives the user the option of using either of the three aforementioned runtimes. Their E2 Core [52] also contains WASM execution capabilities, for processes focused on ETL and application plugins. WaPC [53] is a protocol for communicating in and out of WebAssembly, which proved to be one of the main challenges for the development of this thesis.

Lastly, Erlang's BEAM provides many of the process management features Wetware aims to provide [54], as well as being able to create distributed Erlang systems through various communicating runtimes. These distributed systems support passing messages between processes on different nodes, one of the key Wetware features. Erlang provides process linking and monitoring across distributed runtimes [55] [56], a feature also implemented in this thesis. Implementations are radically different, as the features in this project were designed around object capabilities and other Wetware functionalities. Furthermore, we provide the means to use these features from any programming language that supports WebAssembly is a compilation target and has a Cap'n Proto implementation.

The idea of using Cap'n Proto in WebAssembly has been proposed by capnp's creator [57] and seems to have been demonstrated earlier [58], but to the best of our knowledge this thesis is the first application to integrate both into a practical use case. The novelty of the involved technologies is made more apparent as some of the features of both Go and Wazero were being developed parallel to this project and were integrated in its intermediate stages.

Chapter 3

Objectives and Planning

The project was planned around a set of objectives, which are listed in this chapter followed by an overview of the work plan.

3.1 Objectives

This section lists the objectives of the project. Objectives (O) may have functional (R) and non-functional (NR) requirements.

O1 Create a WASM process executor (E) based on the Wazero runtime.

O1.R1 E must run processes that comply with the WASM and WASI specifications.

O1.R2 E must receive WASM bytecode and output a *Process* object capability.

O1.R3 E must have a minimal CPU footprint when idle.

O1.R4 E must keep functioning when WASM processes fail.

O1.R5 E must keep track of process hierarchy.

O1.R6 *E* must implement the following process management features and expose them through the *Process* object capability:

O1.R6.1 Stop a process.

O1.R6.2 Link two processes: if either ends, so will the other.

O1.R6.3 Monitor a process: the monitor will be notified when a process ends.

O1.R6.4 List all running processes.

O1.NR1 The code style must be in line with the wider Wetware project.

O1.NR2 Methods and structures should be private unless there is a reason not to.

O1.NR3 CLI output must be clean.

O1.NR4 The code must be documented.

O2 Expose the executor through an object capability.

O2.R1 An *Executor* capability must contain a method that receives WASM bytecode and outputs.

O2.R2 A *Process* capability must give access to process management features, with the exception of **O1.R6.4** which will be provided by the executor.

O3 Integrate the executor into Wetware.

O3.R1 Wetware nodes must initialize an executor when they are created.

O3.R2 Wetware nodes must expose their executors through an *Executor* method in the object capabilities of the nodes.

O3.R3 A new CLI command must be created to allow executing a program from a WASM file or byte stream, in an executor.

O3.NR1 There must be wrapper methods to abstract users from direct capability usage.

O3.NR2 Log output format must be consistent with the output of other Wetware components.

O4 Validate the executor through a real application.

O4.R1 The application must be a distributed web crawler C.

O4.R2 C must have a distributed log to agree on visited pages.

O4.R3 C must crawl real pages.

O4.R4 C must store its output in a database.

O5 Analyze the executor performance and scalability.

O5.R1 A basic work-bound application must be created to test work-bound scalability.

O5.R2 A strong scalability analysis must be performed.

O5.R3 Performance and concurrency must be characterized.

O5.NR1 Executor profiling, along with **O5.R2** and **O5.R3** should be considered to improve executor performance.

O6 Create a Raft library (RL) for the processes that run in the executor. Besides an objective, it is also a dependency of **O4**.R2.

O6.R1 RL must be a valid compilation target for WASM.

O6.R2 RL must use be based on Cap'n Proto.

O6.R3 RL must allow configuring hooks for when a new value is received.

O6.R3.1 RL must use object capabilities as its high-level transport.

O6.R3.2 *RL* must expose Raft functionality through a object capabilities.

O6.NR1 It should be based on an existing implementation.

3.2 Work Plan

The thesis has been developed alongside a full-time job, with most of the work being carried out as daily part-time sessions complemented by full-time sessions on the weekends.

The work plan, laid out in figure 3.1, is as follows. The initial steps involve researching the state of the art, which tools would be best suited for the project, and how they might be used. Next is validating the selected tools by creating small test programs for both individual and integration tests. After that comes the core of the project: developing the executor and using it for experiment applications. On one hand, the development is split into different "core" problems:

1. Design and implement a basic WASM process executor that runs in a Wetware cluster node and can be invoked via Cap'n Proto RPCs.
2. Allow WASM processes run in the executor to use capabilities passed on by the executor. WASM processes are natively isolated, thus this step involves finding a way of breaking that isolation, dealing with asynchronous calls in a synchronous environment, and ensuring it works across environments and devices.
3. Process management tools:
   (a) Keep track of process hierarchies, via local process process trees.
   (b) Create and delete a process, developed at step 2.
   (c) Link two processes so if one ends, the other does too.
   (d) Pause/resume a process.
   (e) Monitor processes so the monitor gets notified when the process status changes.

   (f) List processes on an executor. List processes on a cluster.

4. Integrate the executor with the rest of the Wetware codebase, which is constantly evolving.

5. Improve the different components as seen fit while developing the rest of the steps, both for performance and correctness.

On the other hand, experimentation involves creating a basic application, integrating the Raft consensus algorithm with Cap'n Proto and Wazero, and creating a distributed application based on the basic application using this Raft library.

1. Create a basic application that runs multiple processes, possibly in multiple executors, and takes advantage of the functionality offered by Wetware. The application could have distributed workers but would have a central coordinator process. Communication among processes would be performed through capabilities, either the ones provided by Wetware or some created for this specific application. The selected application is a web crawler with main and worker processes, in which the workers are distributed but the coordinator is not.

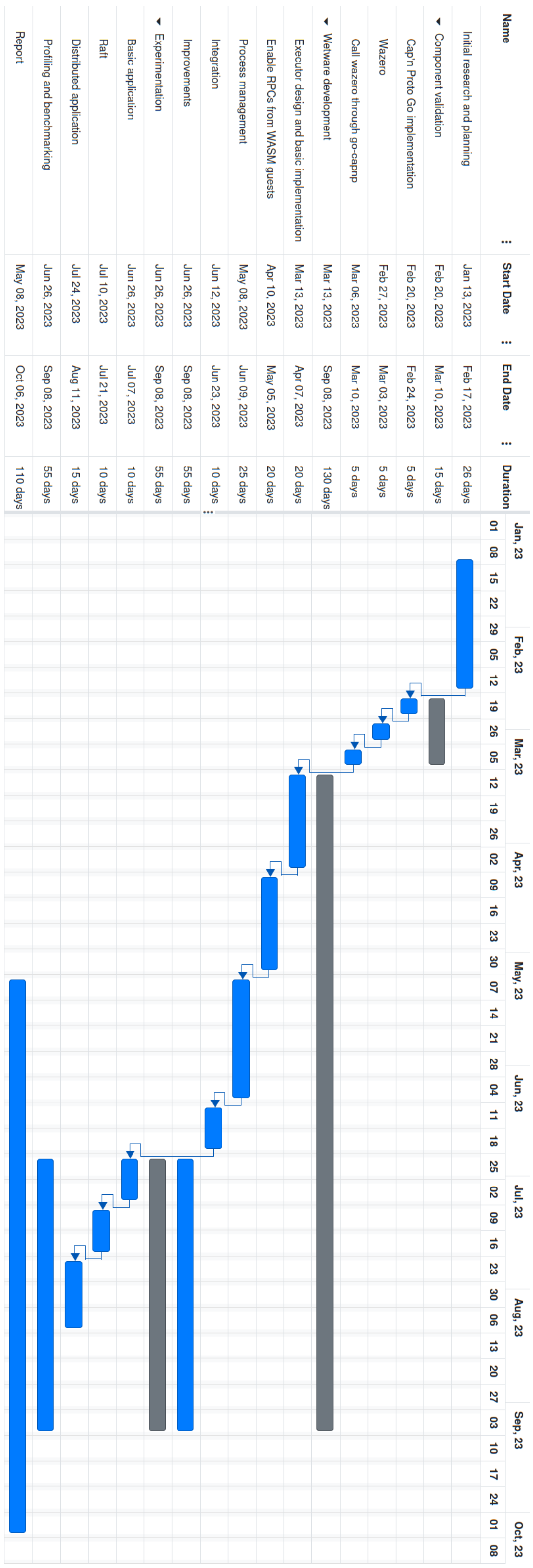
2. Create a Raft library that uses Cap'n Proto as a transport and can run in WASM.

3. Create a distributed application based on the basic one, removing the central coordinator and making use of the Raft implementation.

4. While the previous steps are being done, profile their execution and benchmark different cases.

5. Write and run a benchmarking application.

Finally, once the development has reached enough momentum, start writing the final document and keep on it for the remainder of the project.



Name	Start Date	End Date	Duration
Initial research and planning	Jan 13, 2023	Feb 17, 2023	26 days
▾ Component validation	Feb 20, 2023	Mar 10, 2023	15 days
Cap'n Proto Go implementation	Feb 20, 2023	Feb 24, 2023	5 days
Wazero	Feb 27, 2023	Mar 03, 2023	5 days
Call wazero through go-capnp	Mar 06, 2023	Mar 10, 2023	5 days
▾ Wetware development	Mar 13, 2023	Sep 08, 2023	130 days
Executor design and basic implementation	Mar 13, 2023	Apr 07, 2023	20 days
Enable RPCs from WASM guests	Apr 10, 2023	May 05, 2023	20 days
Process management	May 08, 2023	Jun 09, 2023	25 days
Integration	Jun 12, 2023	Jun 23, 2023	10 days
Improvements	Jun 26, 2023	Sep 08, 2023	55 days
▾ Experimentation	Jun 26, 2023	Sep 08, 2023	55 days
Basic application	Jun 26, 2023	Jul 07, 2023	10 days
Raft	Jul 10, 2023	Jul 21, 2023	10 days
Distributed application	Jul 24, 2023	Aug 11, 2023	15 days
Profiling and benchmarking	Jun 26, 2023	Sep 08, 2023	55 days
Report	May 08, 2023	Oct 06, 2023	110 days

Figure 3.1: Work plan

Chapter 4

Methodology and Resources

4.1 Development

Contributions to Wetware were made following an agile methodology, based on short-term goals building towards a longer-term objective. There was constant communication with the rest of the team, as well as weekly meetings to discuss news, progress, roadblocks and comment on any Wetware-related ideas. The incremental, short-term goal-oriented process allowed us to adapt to the quickly changing environment and the even more quickly changing code-base. The submitted code was often reviewed by other peers, and the code of other peers was reviewed by me.

4.1.1 Collaboration and Communication

Louis Thibault, the creator of Wetware maintained close contact with other open-source communities such as Cap'n Proto or Wazero was one of the main pushing forces behind the project. He provided invaluable insight into how Wetware worked as a whole, and its internal components, and helped me navigate and learn about lower-level workings of both Cap'n Proto and Wazero through regular discussions on the topic and collaborative experimentation.

The Wazero [59] and Cap'n Proto [60] open communication channels offered a lot of information and quick responses to our questions.

4.1.2 Communication

Regular meetings were held weekly with the thesis director, as well as the Wetware team. Meetings with the director took place either in person or through Google Meet, while Wetware meetings were always carried out remotely with Google Meet and Zoom. Besides the weekly meetings, there was constant communication with the Wetware team through Matrix to comment on development-related topics. Communication with the Go Cap'n Proto community was carried out through their matrix channel [60], and communication with Wazero happened on their Slack channel [59].

4.1.3 Version Control

Most of the work on the Wetware project took place in the main repository [61] as well as a personal fork [62]. Experimentation was often carried out on now abandoned branches, where a feature was iterated on until a working version was found. That version was then refined in a new branch, which was later merged into the principal branch. There are two main reasons for experimenting in isolated branches. The first one is that the Wetware codebase changed very rapidly and would constantly cause issues with the experimentation code, which changed even more rapidly. The second is that code merged into the principal branch must be reviewed and might impact potential Wetware users, which is not appropriate for the volatile nature of the experimentation. Once features were ready for review, a pull request was created, reviewed, iterated on, and submitted. The web crawler [63] and raft implementation [64] [65] have their own repositories.

4.1.4 Roadblock Resolution

The high degree of experimentation and usage of cutting-edge versions of languages and libraries meant that even if the plan to achieve an objective seemed feasible, there were bound to be unexpected hindrances. For this reason, possible alternative approaches were often discussed to circumvent possible roadblocks. An issue with asynchronous communication, discussed in chapter 5, is a perfect example of an approach that seemed feasible, ended up not being possible, and required diverging into alternative solutions. In this example, we were conscious that achieving asynchronous communication could present problems dependent on the WebAssembly runtime and studied migrating to another runtime if the problems became project blockers.

4.1.5 Iterative Design

New things were constantly being learned as development progressed, making some previously implemented features worth revisiting to apply newly learned lessons, optimize performance, and improve code maintainability. The agile approach set short-term goals that left a lot of leeway to change what objectives would be prioritized next, and how they would be approached. Lastly, characterizing performance gave some insight that proved useful on some occasions to optimize parts of the code, which was then improved and re-characterized iteratively.

4.2 Characterization

Both computational performance and correct concurrency control have been characterized. Characterizing performance through profiling and benchmarking led to a better understanding of our resources, potential improvements, and limitations. Concurrency characterization showed whether synchronization primitives were being adequately utilized.

4.2.1 Profiling

Profiling was carried out with two main tools: *pprof* and Intel VTune. Intel VTune is a performance analysis tool for x86 architectures with advanced profiling features, more so on Intel processors like one used for the development of this thesis. While VTune provides valuable insight, our analysis relies more in *pprof*, as it can be configured from within the Go program to pinpoint specific parts of the code. Pprof was originally developed to profile C++ programs, however, Go has built-in support for it. Conveniently, there is a tool for profiling WebAssembly modules built and run on Wazero [66], allowing the profiling of both Wetware host and guest code with the same tool. The WebAssembly profiler acts as an additional layer on top of the Wazero runtime and allows profiling CPU and memory usage. Go's profiler has more features, enabling the profiling of CPU usage, memory usage, block and goroutine usage, mutex contention, and low-level execution tracing. Some of these features allow configuring the sample rate, useful for balancing performance and accuracy. Listing 4.1 shows how a profile of a section of the code (`doWork()`) can be generated and stored in a file. Such file can then be processed by the *pprof* command-line tool to provide results in a variety of formats; from *top*-like commands to a full-featured web user interface.

```go
runtime.SetCPUProfileRate(100000)
f, _ := os.Create("cpu-prof.pprof")
defer f.Close()
if err := pprof.StartCPUProfile(f); err != nil {
    panic(err)
}
doWork()
pprof.StopCPUProfile()
```

Listing 4.1: CPU profiling of a Go code segment

4.2.2 Benchmarking

Benchmarking usually involves comparing something against a standard. For the development of Wetware, the baseline was the result obtained by measuring the performance of a feature on its previous iteration. Comparing current performance with past performance kept code from regressing, as well as providing useful insight into what worked and what didn't. Benchmarks could either be performed manually, as explored later in chapter 5, or automated with Go's built-in benchmark tools.

4.3 Testing

Unit and integration testing are vital parts of the development process. Unit tests verified that standalone components worked, while integration tests verified components interacted correctly with one another to provide the expected overall functionality.

Unit tests are straightforward and usually follow the same formula: an initial setup where components are initialized, a section that performs the action being tested, and a final section verifying the action had the desired effect. Listing 4.2 contains an example of a unit test used for the process tree developed earlier in this chapter. For reference, unit tests are located in the same directory as the code they are testing, in the main Wetware repository.

```go
func TestProcTree_Insert(t *testing.T) {
    // child, parent, branchof, 0=left 1=right
    matches := [4][4]uint32{
        {12, 5, 5, 0},
        ...
    }
    pt := testProcTree()  // generates a pre-populated process tree
    for _, match := range matches {
        pid, ppid, expectedId, side := match[0], match[1], match[2], match[3]
        pt.Insert(pid, ppid)
        ...
    }
    ept := ProcTree{/* expected shape */}
    if *pt != ept {
        t.Failf("final tree did not match expected tree: %v - %v", *pt, ept)
    }
}
```

Listing 4.2: Simplified unit tests for ProcTree.Insert

Go provides the means to benchmark applications as a complement to unit testing. While only informative in this thesis, benchmarking results could potentially be used to reject changes that entail some level of performance loss. The sample in listing 5.9 is the source code of the benchmark used to measure the performance of inserting sibling processes on a process tree.

```go
func BenchmarkTree_InsertSibling(b *testing.B) {
    t := csp.ProcTree{...}
    var offset, total uint32 = 1, 10000
    b.ResetTimer()
    for i := offset; i < total+offset; i++ {
        t.Insert(i, 0)
    }
}
```

Listing 4.3: Simplified benchmark for ProcTree.Insert

Another fundamental consideration when verifying the correctness of the code is verifying that certain packages can be compiled to, and therefore used from, WebAssembly. Most of the repositories developed during the thesis contain a Makefile with a `make test wasm` command. This command builds a list of predetermined packages with the WASM compilation target, allowing us to see if any proposed changes would break WASM compatibility.

Lastly, knowing that a component works in an isolated test does not guarantee it will work when it is used in a Wetware cluster. Integration tests in this case consist of instantiating Wetware nodes and testing whether the components behaved as expected when invoked through RPCs in a real environment; e.g. starting a Wetware node and running a simple Wetware process through its executor, used to verify whether capability bootstrapping works in a real environment. There have been many times were unit tests of the executor or other components passed, but using those components through a client that connects to a Wetware cluster failed. Initially, these integration tests intended to be performed by an automated CI tool whenever a commit was uploaded to any branch of the git repository, however this methodology was discarded during the development of the thesis for complexity and task priority reasons, as the rapidly changing library required continuous adjustments and generated conflicts between developers of different components. After the disabling of the CI tool, integration tests were performed manually each day to ensure no new errors had been introduced.

4.4 Experimentation

Analyzing the validity of the executor and its process management capabilities, as well as its performance impact required two distinct methods of experimentation. The first one focused on validating the solution and verifying it worked not only on a single machine but as a distributed system with multiple hardware nodes. The second one is creating work-bound and IO-bound applications and observing how the executor and the applications behave.

4.4.1 Validation

Validation is done through continuous unit and integration testing, as well as an implementation of a functional crawler that runs parallel Wetware processes on one or multiple nodes and accesses external resources through object capabilities. While initially simple, the web crawler application evolved to contain a distributed log for pages to have consensus on visited pages.

4.4.2 Performance

Evaluation of performance was done with some experiments involving both work-bound and IO-bound workloads. The work-bound program consisted of looped arithmetic operations spread over multiple processes and repeated an arbitrary number of times. IO-bound behavior was analyzed by giving the web crawler access to varying amounts of resources. The graphs shown in chapter 6 were created using Matpltolib [67] and SciencePlots [68].

4.5 Testbed

This thesis had tests performed utilizing three devices, whose specifications are available in table 4.1. The validation experiment with a multi-node web crawler used all three devices. All other tests and measurements took place on the desktop PC. All three devices are connected to the same local network through Ethernet of a maximum of 1Gbps for the desktop and laptop, and 100Mbps for the Raspberry Pi.

Device	**Component**	**Specification**
Desktop PC	CPU	Intel Core i7-6700K, 4.00GHz, 4C8T
	Memory	4x8GB dual channel DDR4, 2133MHz
	L1 Data	4x32kB, 8-way associative, 64B lines
	L2	4 x 256kB, 4-way associative, 64B lines
	L3	8 MB, 16-way associative, 64B lines
	OS	Linux 6.5.5-arch1-1
Laptop	CPU	Intel Core i7-7500U, 2.70GHz, 2C4T
	Memory	1x16GB DDR4, 2133MHz
	L1 Data	2x32kB, 8-way associative, 64B lines
	L2	2x256kB, 4-way associative, 64B lines
	L3	4MB, 16-way associative, 64B lines
	OS	Linux 6.5.5-arch1-1
Raspberry Pi	Model	3B
	CPU	4× ARM Cortex-A53, 1.2GHz
	Memory	1x1GB LPDDR2, 900 MHz
	L1 Data	4x16KB
	L2	512KB
	OS	Linux 5.10.103-v7+

Table 4.1: Testbed specifications

Chapter 5

Development

5.1 Development Considerations

This section will go over the most important factors that were considered when developing the components described in this chapter.

5.1.1 Terminology

The execution of WASM code in the chosen WASM runtime will be referred to with different terms: *guest process*, *WASM process*, and *WASM module instance*. In contrast to this, the concept of a process running in a Wetware executor will be referred to as *Wetware process* or simply *process*. The latter encapsulates an object capability that wraps a guest process. On a similar note, channels might refer to either the native communication tool of the Go programming language, or the general-purpose communication tool provided by Wetware as an object capability, as illustrated by figure 5.1. The context in which it is used will make it clear which one of them a segment is referencing.

5.1.2 Thread Safety

Thread safety is a fundamental requirement for the executor and practically every software component developed for this thesis. Executors, processes, and other

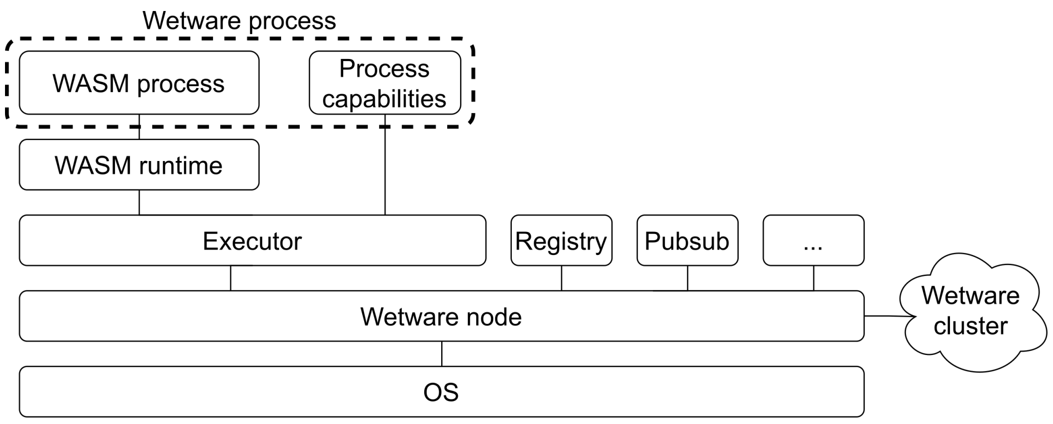


Figure 5.1: Wetware processes vs. WASM process

components will be attending multiple calls concurrently, often parallelly. Correct usage of concurrency control and communication mechanism is paramount, for both optimal performance and avoiding issues such as deadlocks. The tools and mechanisms used to ensure thread safety are mainly: channels, system calls, and mutexes. They all have a variety of types and uses, which will be explained in more detail throughout chapter 5.

5.1.3 Scheduling

Due to Wetware's overall design, the technologies used for the development of the process executor, and the high amount of communication distributed systems usually require, concurrency is a must for this project. Understanding Go's approach to concurrency can lead to better performant application design. For that reason, this section will go into some level of detail on how concurrency is handled in Go and how it is utilized in this project.

Work in Go programs is performed by goroutines: lightweight threads managed by the Go runtime [2]. How these lightweight threads are matched to OS threads is usually explained through the Gs, Ms, and Ps. G refers to a goroutine, of type g. M refers to an OS thread, of type m. An M can be running Go code, runtime code, a system call, or idle. P refers to a logical processor of type p, such as a CPU core. For two Gs to run in parallel, they need to be executed on different Ps. The Go scheduler's primary function is to match a G, an M, and a P. Every

g, m, and p object is allocated on the heap and is never freed in order to avoid write barriers on the execution of the scheduler.

Because goroutines are Go runtime objects and any number of Gs can share a M OS thread, switching context between goroutines is much cheaper than switching context between OS threads; switching Gs on an M has a significantly lower cost than switching Ms on a P. Three registers need to be stored and loaded when performing a goroutine context switch: the program counter, the stack pointer, and the general-purpose data register DX. Go's concurrency model is partially preemptive and will context-switch goroutines after a fixed amount of time to keep certain code patterns, such as tight loops, although cooperative context switching is often preferred.

The scheduler attempts to exploit data locality by keeping related Gs on the same P [69]. Each P has a local run queue (LRQ) of Gs to run, as well as access to a global run queue (GRQ) shared with the rest of Ps. LRQs have a maximum capacity of 256 Gs. Figure 5.2 shows an example of a scheduler state where a logical processor P is running a goroutine G_1 through an OS thread M, while having G_2 and G_3 on its LRQ. The GRQ contains G_4 and G_5. When a new G' is added to a P with a full LRQ, Gs are moved out of the queue in a batch for this same reason. If a P attached to a M runs out of tasks in the local queue, it will take runnable Gs either from the global queue, the LRQ of other processes, or the network poller, in order to keep the OS thread M to be preempted from P.

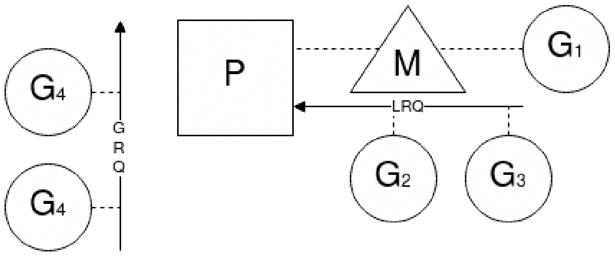


Figure 5.2: Scheduler state example representation

IO-induced context switches also introduce significantly less overhead than preemptive context switches. In the context of this thesis, we will differentiate between two types of IO-bound concurrency: local IO and global IO. Local IO occurs when a goroutine performs IO calls towards other goroutines through Go channels, all of which happens inside the Go runtime. Global IO occurs when IO operations take place outside the Go runtime, e.g. writing to a TCP socket. Starting with the former, when a goroutine needs to write to or read from a non-buffered channel, the goroutine will park itself and allow the execution of another goroutine on its M. The software developed in this project takes heavy advantage of that by performing a lot of local IO-bound work where goroutines perform some work and either read from or write to a channel. When a goroutine needs to perform globally IO-bound work, it will make an asynchronous system call. Modern operating systems natively support asynchronous system calls through tools such as epoll [70] on Linux. When a G performs an asynchronous system call it will be placed in the queue of a new entity distinct from P: the network poller. Once the asynchronous syscall ends, the network poller will place G back in the LRQ of the P that had it. Figure 5.3 shows the state of figure 5.2 in which G_1 is performing an asynchronous system call and has thus been moved to the network poller. Once it is finished, G_1 will be put back in P's LRQ. The network poller is a key pillar for distributed applications such as Wetware, where RPCs are performed all throughout the code base and therefore need to handle network operations efficiently without blocking execution.

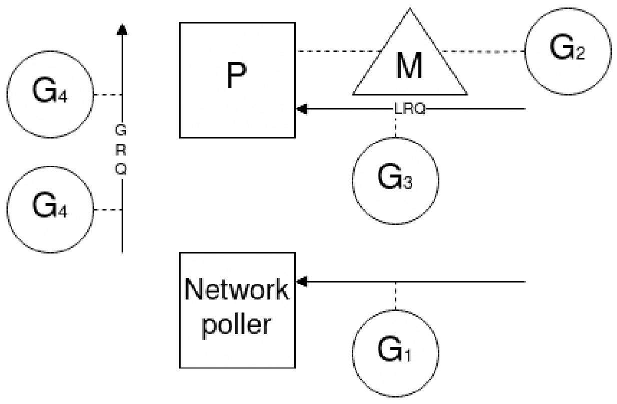


Figure 5.3: Scheduler state example representation with a network poller

Go provides multiple native execution-yielding mechanisms as detailed in table

Table 5.1: Block-levels of native yielding interfaces [2]

	Blocks		
Interface	**G**	**M**	**P**
mutex	Y	Y	Y
note	Y	Y	Y/N*
park	Y	N	N

5.1. Goroutines can park themselves, yielding execution to other Gs on the same M and P. Goroutines can be parked for a multitude of reasons, the most common ones being:

- *Use of the* `go` *keyword.* The keyword implies the creation of a new goroutine, after which the Go runtime is given the opportunity to change what goroutine is executed.
- *System calls.* The M will preempt execution from G to run the system call.
- *Synchronization.* Use of channels, atomic objects, and other concurrency tools.
- *Garbage collection.* Garbage collection runs on a set of goroutines, which also need an M.

Notes, in the context of table 5.1, refer to OS features such as epoll, signals, or OS concurrency tools. Whether or not these features block logical processors depends on the specific feature.

Lastly, mutexes block the M running the G that uses it directly without going through the Go scheduler. Despite being the most penalizing mechanism, they are sometimes required and are best used to perform quick operations, minimizing the time between the lock and unlock operations.

Go provides native tools to minimize performance losses when using such tools. An example amply utilized in multiple occasions throughout the development of this thesis is `sync.Map`, which provides built-in thread safety and becomes more performant than manually implemented maps with read/write mutexes as application parallelism grows [71].

As a closing thought, Go manages to make IO-bound workloads seem like CPU-bound workloads to the OS it's running on by managing goroutines with the Go runtime, running on user space as much as possible, and keeping OS implication to a minimum. The concepts mentioned throughout this section will be present all throughout this chapter, and will give the reader a better understanding of what is happening under the hood when using channels, attending or performing RPCs, or dealing with thread safety.

5.2 Base Executor Design

The highest-level concept for the executor is defined as: A software component capable of running and managing WASM processes, exposed through an object capability. This basic definition implies the integration of Cap'n Proto and Wazero, the two core tools we've researched earlier. Figure 5.4 shows the main components of the executor in its final form, although some elements can be ignored for now as they will be developed later in this chapter. The components relevant to this section are: the executor, the processes, the Wazero runtime, and the WASM module instances. Components marked with an asterisk (*) are exposed through capabilities.

Let's now go over the basics of how a Wazero runtime can be configured, shown in listing 5.1. The configuration is built utilizing the Functional Options Pattern, which consists of adding methods to a type T that take a t, apply some configuration, and return the changed t, e.g. `func (t *T) withChange(...) *T`. When the Wazero runtime executes WASM module instances, it binds the execution to an arbitrary context. The option `WithCloseOnContextDone(true)` specifies that when the context bound to a running guest process finishes, so does the guest process. This is a requirement for the correct shutdown of executors, as well as for developing the *Kill* command as will be explored later on. After the runtime is created, it is configured to use Wazero's partial WASI implementation by the function shown in line 5 of listing 5.1.

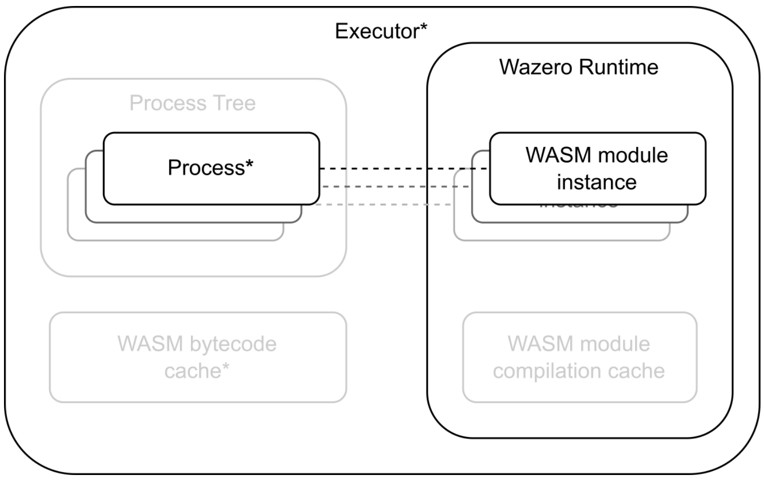


Figure 5.4: Main components of the executor

```go
cfg := wazero.
    NewRuntimeConfigCompiler().
    WithCloseOnContextDone(true)
runtime := wazero.NewRuntimeWithConfig(ctx, cfg)
wasi_snapshot_preview1.Instantiate(ctx, runtime)
```

Listing 5.1: Wazero runtime creation and configuration

Next let's understand how the runtime can be used to configure and compile a WASM module, followed by instantiating the module and starting a guest process by calling the main function. The first step is to decode and validate the intermediate bytecode, done in the first line of listing 5.2. Instantiating the module requires some configuration, which in this case includes:

- Don't run any start functions when instantiating the module. If unset, guest execution would start as soon as instantiation is done, instead of when explicitly requested.
- Set the clock precision and initialize the clocks.
- Set a source for random generation.
- Set the module name, used to tell modules with equal bytecode apart.

- Set environment variables. The variable *ns* is required for the capability bootstrapping explained later in this chapter.
- Bind the *stdin*, *stdout* and *stderr* of the instance.

The only steps left are to export the main function, identified by the *_start* name, and call it with a context, as mentioned when listing 5.1 was explained. `fn.Call(ctx)` will block the execution of the current goroutine and run the guest process in a separate goroutine. The whole life cycle of the WASM guest will be bound inside that goroutine and thus WASM guests process have no real parallelism.

```go
compiled, _ := runtime.CompileModule(ctx, bytecode)
modCfg := wazero.NewModuleConfig().
    WithStartFunctions().         // ommit start func.
    WithSysNanosleep().           // sleep precision.
    WithSysNanotime().            // clock precision.
    WithSysWalltime().
    WithRandSource(rand.Reader). // rng.
    WithName(name).               // module name.
    WithEnv("ns", name).          // env variables.
    WithArgs(args...).            // args
    WithStdin(os.Stdin).
    WithStdout(os.Stdout).
    WithStderr(os.Stderr)
mod, _ := runtime.InstantiateModule(ctx, compiled, modCfg)
fn := mod.ExportedFunction("_start")
if fn == nil {
    return fmt.Errorf("ww: missing export: _start", mainFunc)
}
fn.Call(ctx)
```

Listing 5.2: Create a WASM module and call a function

As a side note, the initial versions of the executor capability allowed passing Wetware channel capabilities as arguments, to use as the process' *stdin*, *stdout* and *stderr* by binding them to byte streams created by the executor and passed to the Wazero module instantiation through `modCfg.WithStd(in|out|err)` configuration options. We discovered a bug in *go-capnp* where the arguments of two methods of a capability could be switched if they had the same signature, which has been since solved. While functional, the concept was discarded due to the additional overhead and points of failure it introduces. It is still possible to achieve the same functionality by manually passing channels to a process and replacing the guest program's *std* outputs.

These steps allow the execution of WASM code, but that still needs to be exposed through an object capability. The *Executor* capability serves that purpose, through the `exec` method.

```
interface Executor {
    exec @0 (bytecode :Data) -> (exitCode :UInt32);
}
```

Listing 5.3: Executor capability definition

The sequence diagram displayed in figure 5.5 sums everything up and shows the basic steps of running a WASM process through a Wetware RPC.

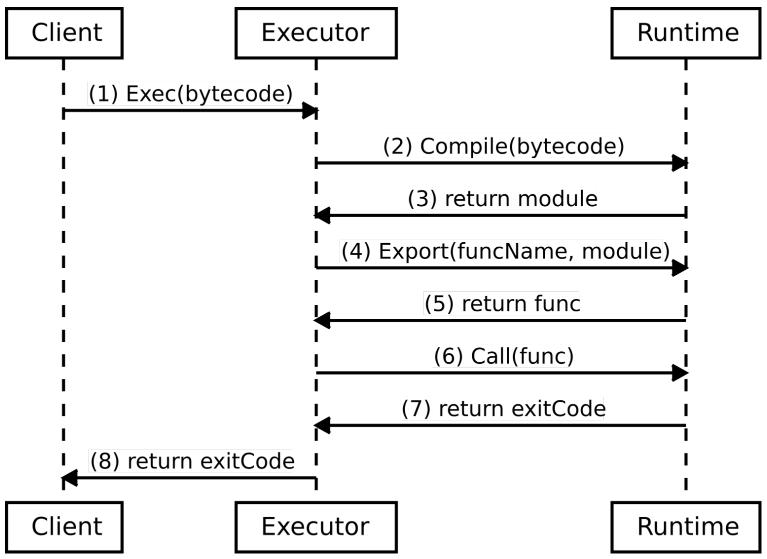


Figure 5.5: Steps of running a WASM function through a Capnp RPC

For the sake of simplicity, this section has omitted some segments of the code that are not relevant as they don't contribute to the understanding of the process and deal with specific problems, e.g. ensuring resources are closed after they are used or dealing with errors. An exception to this are command line arguments.

Wetware processes can receive command line arguments, by including an `args` parameter of type `List(Text)` in the `exec` method signature. This text list is then parsed and passed to the process with the `WithArgs(s ...string)` option in the module configuration.

5.2.1 Exposing Process Control

In order to maintain consistency with the Object Capability oriented design of Wetware, managing a process should be done with a capability presenting said process. Users are given access to processes through a capability created by the executor to encapsulate the aforementioned `fn.Call(ctx)`. This way `exec` will no longer block until the WASM process is done, the `Process.Wait` function will serve this purpose, see listing 5.4. Other process functionalities, such as *Kill* or *Link* will be exposed through this very same capability later on.

```
interface Executor {
    exec @0 (bytecode :Data) -> (process :Process);
}

interface Process {
    wait   @0 () -> (exitCode :UInt32);
}
```

Listing 5.4: Process capability definition

The sequence diagram representing inter-component interactions that happen during the creation and execution of a process, from executing WASM bytecode to waiting for it to finish is expanded in figure 5.6.

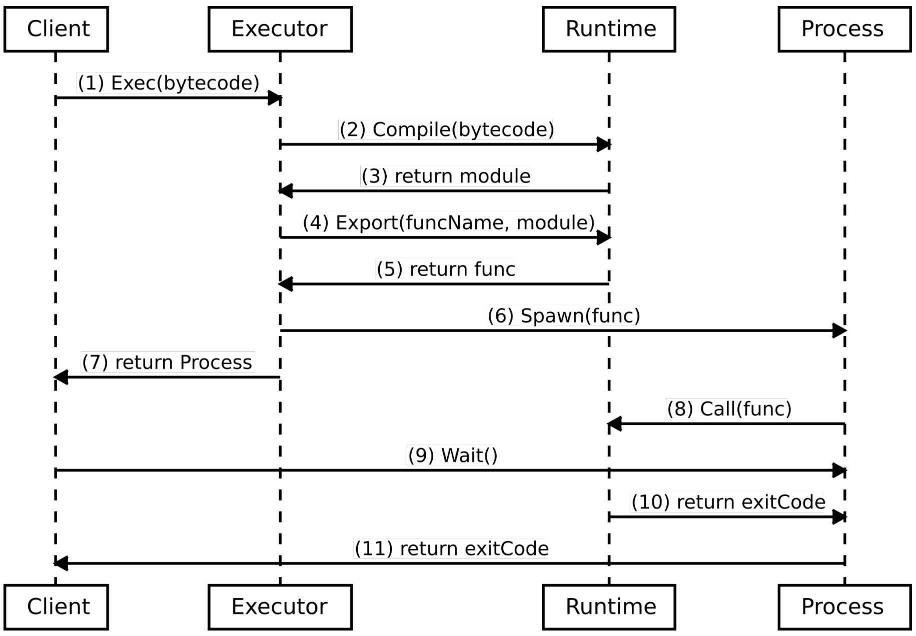


Figure 5.6: Addition of a Process capability to figure 5.5

The full definition of the process implementation is included in appendix A, however listing 5.5 shows the partial definition of a process for illustration purposes. Two of the attributes are of special relevance: *cancel* and *killFunc*. When called, *cancel* will stop the context that is being used to both run the WASM program, and serve the capabilities of *process*. Calling *cancel* will therefore stop the attached guest code as well as release its object capability. On the other hand, *killFunc* is a private function that is always called by the public *Kill* method. The *killFunc* function is passed onto a process by its executor upon creation. Section 5.4.2 provides more insight into the stopping of a process.

```
type killFunc func(uint32)
type process struct {
    args      []string
    time       int64
    done      <-chan execResult
    killFunc // killFunc must call cancel()
    cancel    context.CancelFunc
    result    execResult
}
```

Listing 5.5: Process capability definition

Listing 5.6 contains a segment of the executor's *Exec* method, where it creates a process, inserts it into the tree and its map, followed by spawning a new goroutine that is responsible for starting the WASM process. That goroutine will block in the line that contains `fn.Call` until the WASM process finishes.

```go
...
done := make(chan execResult, 1)
proc := &process{
    Args:     args,
    time:     time.Now().UnixMilli(),
    killFunc: r.Tree.Kill,
    done:     done,
    cancel:   cancel,
}

r.Tree.Insert(c.args.Pid, c.args.Ppid)
r.Tree.AddToMap(c.args.Pid, proc)

go func() {
    defer close(done) // end the context
    defer c.cancel()  // stop the rpc provider
    defer proc.kill() // terminate the process
    vs, err := fn.Call(c.ctx) // start the WASM process
    done <- execResult{
        Values: vs,
        Err:    err,
    }
}()
...
```

Listing 5.6: Segment of Executor.Exec implementation

5.3 Host-Guest Communication

Expanding the executor definition stated at the beginning of section 5.2: the executor is a software component capable of running and managing *Wetware*

WASM processes, exposed through an object capability. Wetware WASM processes refer to WASM processes capable of using Wetware tools, which needs to be done through object capabilities by design. The executor designed in the previous section can run WASM programs, but those programs are isolated and lack the means to access external resources, such as object capability connections. This section deals with possible approaches and expands on the one that was followed through for our implementation.

The most important characteristic of the communication between the guest and the host is asynchrony. If a WASM process blocks when writing to or reading from the communication medium, execution of the other goroutines in the WASM process will stop. Programs intended for distributed systems, which are the main focus of Wetware, will most likely import a lot of networking. Without asynchrony, processes would become intolerably slow and, perhaps even more importantly, easily deadlock. For these reasons, the transport used by the guest to access capabilities must support asynchronous communication.

Arguably, the most intuitive approach to this problem is to implement host functions, as they are intended to do what the guest can't. While perhaps possible through some form of communication multiplexing or in-guest implementation of *epoll*, the efficacy of this method is not guaranteed to the best of our knowledge. The following quote, extracted from the official Go documentation on WASM, states how host calls will block goroutines.

> WASM is a single threaded architecture with no parallelism. The scheduler can still schedule goroutines to run concurrently, and standard in/out/error is non-blocking, so a goroutine can execute while another reads or writes, but any host function calls [...] will cause all goroutines to block until the host function call has returned [72].

We tested host functions by implementing read/write functions which were used as a transport for bootstrapping capabilities. We implemented read and write calls to Go's native *ReadWriteCloser* type, commonly implemented by transports, buffers, files, and other sensible types. These functions were exported to the WASM runtime so that guest processes could use them. We then implemented

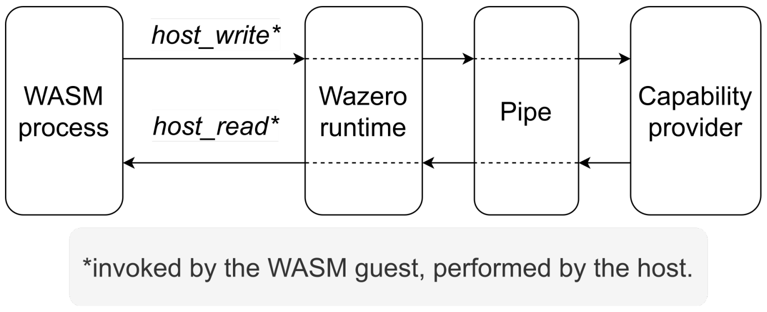


Figure 5.7: Asynchronous host-guest communication through host functions and a pipe

a transport for the WASM guest that would read and write using these system calls. When creating a Wetware process, one of the ends of the pipe was used to provide a capability by a goroutine running on the host. The other end of the pipe was bound to the WASM guest and read from/written to by the guest process, by allowing capabilities to interact with this custom transport. This design is illustrated in figure 5.7. The host function approach worked for Go processes, but consistently deadlocked Wetware processes and was deemed unfeasible.

Conveniently, one of the latest WASI proposals advocates for support of asynchrony in system sockets [73], which depended on the WASM guest implementing *netpoll*. Wazero implemented this functionality while this thesis was ongoing, and we even took a very small part in the conversation around its implementation through Wazero's Slack channels. At the same time, and also during the development of this thesis, Google introduced a new *wasip*1 compilation target separate from the previously existing *JS* in Go 1.21 (see section 5.3.1). This compilation target introduced the *netpoll* instruction for WASM guests [74], making it possible to combine the latest Go and Wazero versions to use TCP sockets as the transport for capability bootstrapping. Relying on cutting-edge versions of these two technologies was vital for the project, but did bring some development headaches which will be later discussed in section 5.3.1.

The transport used for bootstrapping is then multiplexed by Cap'n Proto to serve as a transport for multiple capability connections at once, meaning implementing this correctly will break WASM guest isolation in a controlled manner, and pro-

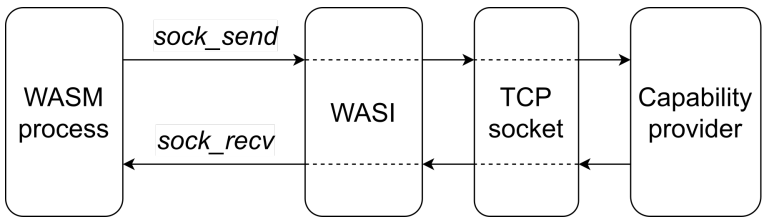


Figure 5.8: Asynchronous host-guest communication through a pre-opened TCP socket

vide a clear way for Wetware processes to interact with the outside world through capabilities.

Figure 5.9 contains the flowchart for attending process execution requests. As shown in the diagram, the host must pre-open a TCP port before passing. The Wazero runtime will then bind this port to the file descriptor 3 of the guest. This file descriptor belongs to the pre-opened TCP port, as it is the first one after stdin (0), stdout (1), and stderr (2). The guest will use the socket like it would any other file. Any number N of ports can be pre-opened for the WASM guest, which can then be accessed by the guest starting with file descriptor 3 up to FD $2+N$. Only the first file descriptor is used, nonetheless. The flow of diagram 5.9 splits into three separate goroutines:

1. Receive the RPC, attend the process, and return its capability as the RPC result.
2. Provide the server-end of the Wetware session capability the WASM process will receive.
3. Start the WASM process by calling a WASM module function.

Goroutines 2 and 3 share a context, independent from the context of the RPC call, so if the context is ended or either of them fails, the other can be easily stopped.

The guest steps, shown in figure 5.10, involve creating a connection listener from a file with FD 3, accepting a connection through the listener, and using the connection as the capability transport. These steps are abstracted from the final user, which will be further explained in section 5.5.1.1.

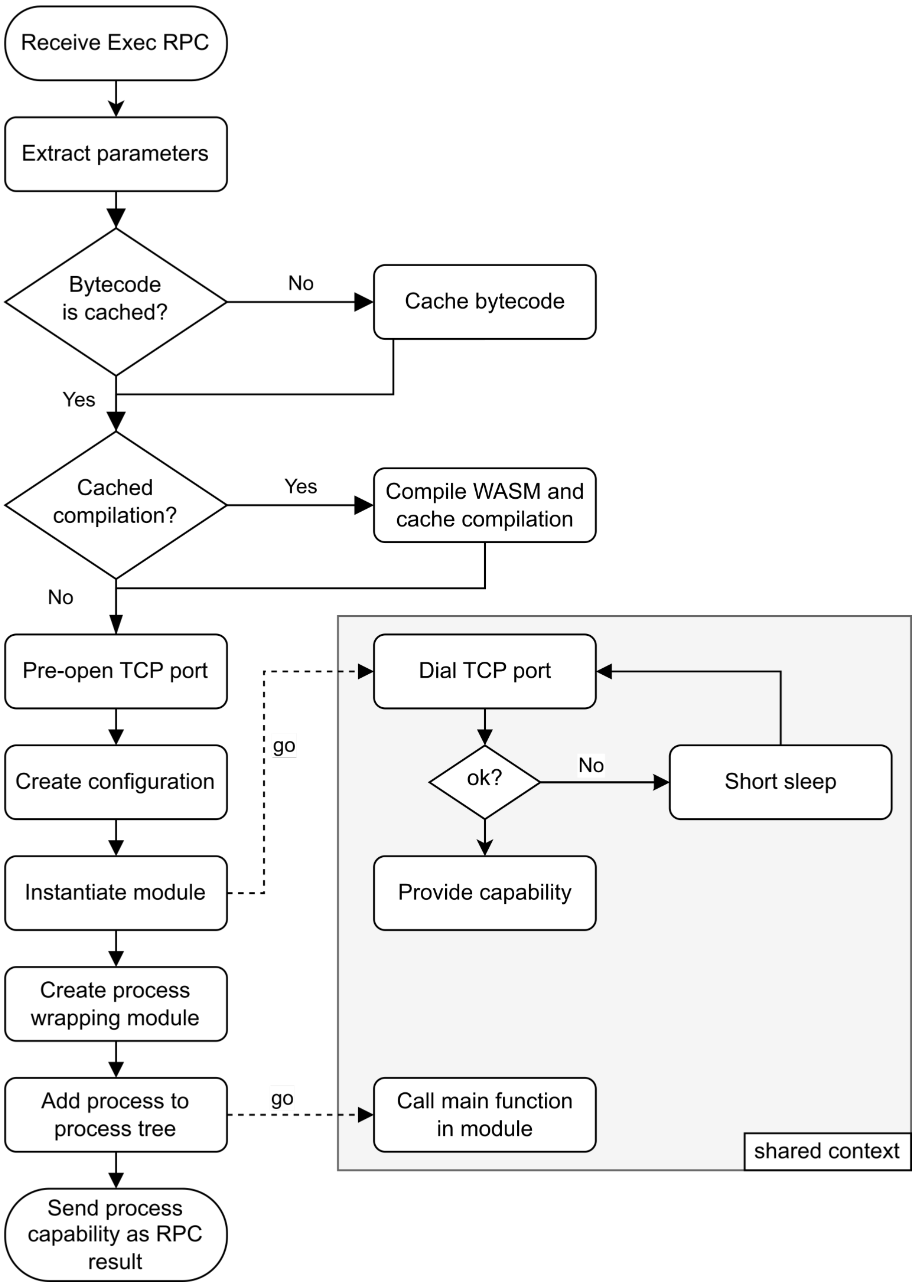


Figure 5.9: Exec RPC server flowchart

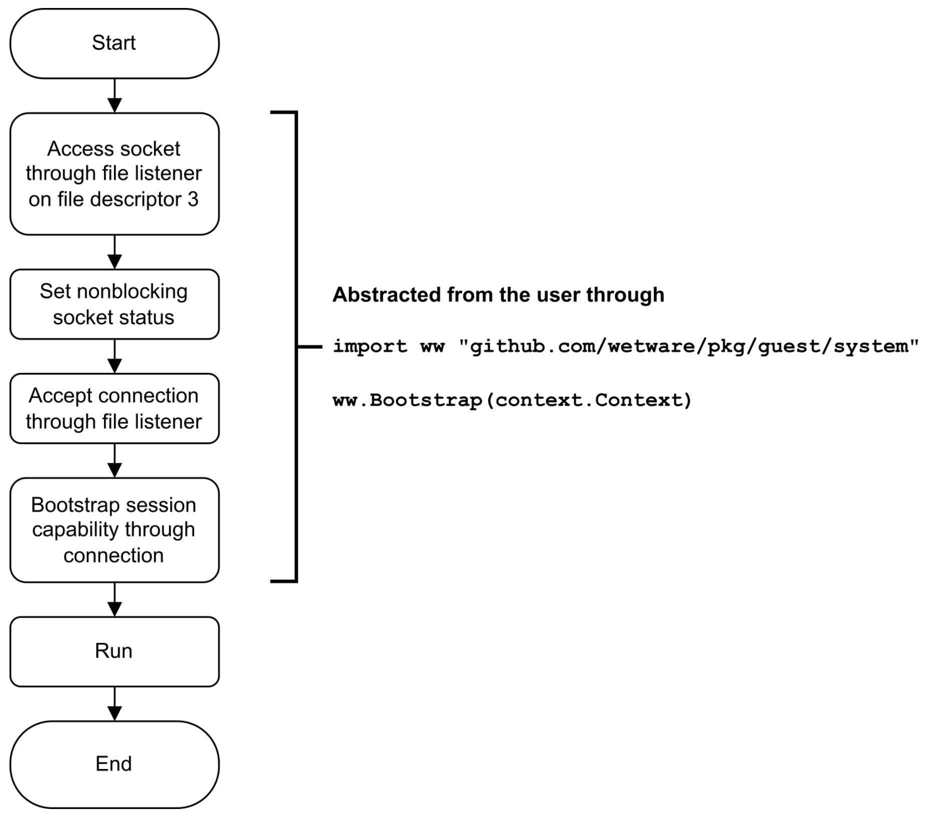

Start
Access socket through file listener on file descriptor 3
Set nonblocking socket status
Accept connection through file listener
Bootstrap session capability through connection
Run
End
Abstracted from the user through
import ww "github.com/wetware/pkg/guest/system"
ww.Bootstrap(context.Context)

Figure 5.10: Wetware WASM program initialization flowchart

This method of communication introduces overhead in both module instantiation and execution in the form of TCP dials and unnecessary data copying. While not computationally expensive, it entails switching context in a goroutine level several times which is not ideal. The fast advancements in the Go WASM compilation target, Wazero, and WASI make the earlier approaches worth revisiting once WASI matures some more; to avoid the aforementioned limitations in order to find a way to allocate a no-copy transport for RPC calls. This allocation is ideally invoked on-demand by the guest via file-system or host functions so that capabilities held by the guest code can each have their own transport and achieve higher throughput. Regarding security, communicating over sockets exposes Wetware to a different set of vulnerabilities than using host functions [75], which applications developers might need to consider.

5.3.1 Migration to Go 1.21

Section 5.3 required upgrading Go from version 1.20 to 1.21, which had no official release at the time. Due to the open-source nature of Go, the 1.21 version could be manually built by building the latest commit, often referred to as *gotip*, of the language's git repository [76].

Go 1.21, however, was not compatible with some of Wetware's dependencies. Tracking down the issue showed that one of libp2p's dependencies, *quic-go*, dropped support for QUIC draft-29 and adopted Go's native *crypto/tls* library for TLS instead of implementing its own as it did with the previous Go versions. This caused errors where libp2p components required code components from *quic-go* that were no longer present, preventing Wetware from compiling with Go 1.21.

Solving this issue required forking many of the dependencies shown below and temporarily replacing them as Wetware's dependencies until the issue was resolved in the second half of August 2023 with the official release of Go 1.21 and *go-libp2p* v0.30.0. Walking down the dependency tree resulted in 13 separate repositories needing to be forked and modified, as listed in table 5.2. All repositories were hosted on GitHub and the forks can be accessed at `https://github.com/mikelsr/<Name>`.

Once the dependencies were identified, forked, and modified, Wetware was modified to depend on the forks instead of the originals: from that point until the official Go 1.21 released in the latter half of August 2023 and the dependencies made the required changes to be compatible. During that period Wetware also needed to be compiled with `gotip`.

As an additional note, the *SetNonblock* system call was not correctly implemented for some time. After many trials, debugging, and asking the Wazero development team, we narrowed down the issue and moved on to other parts of the thesis while it was fixed in a pull request [77] which was active for the same issue on an HTTP server built in a WASM guest.

The issue that pushed us to Go 1.21 and the newly introduced features were

Owner	**Name**	**Changelog**
ipfs	go-peertaskqueue	Modify dependencies.
ipfs	boxo	Modify dependencies.
libp2p	go-libp2p	Modify dependencies. Disable CI. Modify mocks.
libp2p	go-libp2p-kad-dht	Modify dependencies.
libp2p	go-libp2p-kbucket	Modify dependencies.
libp2p	go-libp2p-pubsub	Modify dependencies.
libp2p	go-libp2p-record	Modify dependencies.
libp2p	go-libp2p-routing-helpers	Modify dependencies.
libp2p	go-libp2p-testing	Modify dependencies.
libp2p	go-libp2p-xor	Modify dependencies.
lthibault	go-libp2p-inproc-transport	Modify dependencies.
quic-go	quic-go	Modify dependencies. Generate 1.21 library. Drop 1.19 library. Merge pending PR from origin to adopt native *crypto/tls*.
quic-go	webtransport-go	Modify dependencies.

Table 5.2: Wetware dependency repositories modified for Go 1.21 compatibility

not adopted as quickly in the compiler we were using up to that point to compile.

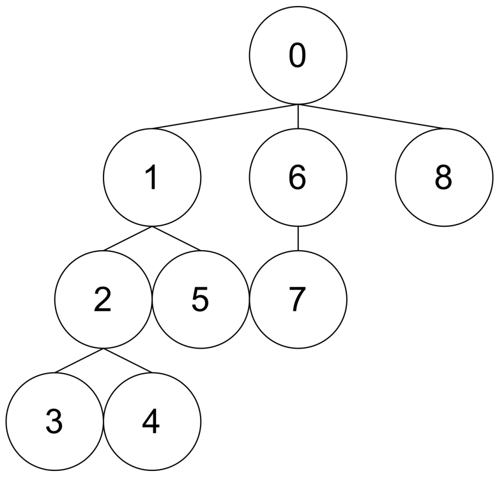
Lastly, it should be mentioned that there was an active conversation about migrating to other WebAssembly runtimes that supported asynchronous communication as opposed to the standard, but this contingency plan ended up not needing to be carried out.

5.4 Process Management

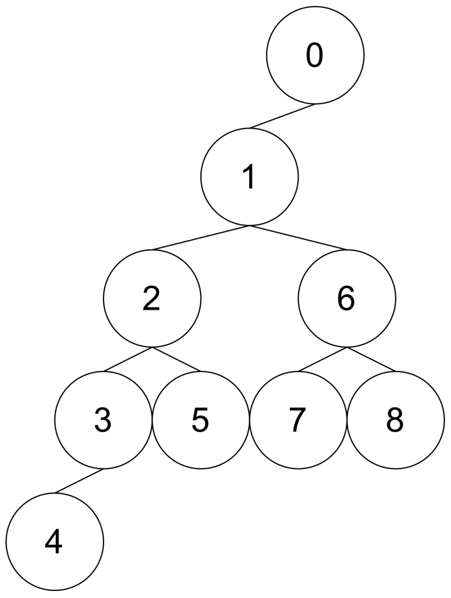
Process management is a fundamental part of the executor. This section will go over the different process management features designed and implemented during the development of the thesis, diving into each of them at a low level.

5.4.1 Process Hierarchy

Process trees are conceptually multi-branch trees where each process is a node identified by its PID. To keep algorithmic complexity low, it was decided to represent the multi-branch tree as a binary tree where the left child of a node represents a child process and the right child represents a sibling brother: a process that shares the same parent. As an example, the process tree shown in figure 5.11a can be represented as the binary tree of figure 5.11b. *PID*s have the *uint32* bytes, as executors do not support 2^{32} concurrent active processes. Each process tree has an atomic counter increased every time a process is created, which is used to generate *PID*s.



(a) Natural tree



(b) Binary representation

Figure 5.11: Representations of the process tree

The executor may receive simultaneous calls that may be handled in parallel. For this reason, thread safety is a must. A read/write mutex is employed to achieve this, allowing any number of reads to happen simultaneously but ensuring no read is being performed while the tree is being altered. The two write operations that can be performed on the tree are adding a node and deleting a node. More

Table 5.3: Read/write requirements of process tree operations

	Operation						
Access	**find**	**findParent**	**insert**	**pop**	**delete**	**trim**	**kill**
read	yes	yes	yes	yes	yes	yes	yes
write	no	no	yes	yes	yes	yes	yes

complex write operations which are composed of any combination of additions and deletions, may need to acquire the write lock to ensure the integrity of the tree. For this reason, the tree has been designed in such a way the private methods don't acquire or release locks. Instead, public methods wrap private methods and are responsible for acquiring and releasing locks. As an example, the private `find` method does not acquire a lock, but the public method `Find` does. This ensures every user of the module uses the thread-safe methods, but allows for careful and deliberate omission inside the package so that functions such as `Delete`, which has an implicit call to `find` inside, can avoid deadlocks. Table 5.3 shows the read/write requirements of each function implemented for the process tree.

Listing 5.7 contains the definition of the process tree. *PIDC* is a counter that increases each time a process is created, giving them unique PIDs. *TPC* is used to keep track of the total number of processes. *Root* is the start of the tree. *Map* maps PIDs to the process objects, so that tree nodes are kept lightweight and actual capabilities are accessible in $O(1)$ time.

```
type ProcTree struct {
    PIDC AtomicCounter // generates PIDs
    TPC  AtomicCounter // total process count
    Root *ProcNode     // root node
    Map  *sync.Map     // process map
    Mut  *sync.RWMutex // read-write mutex
}
```

Listing 5.7: Process tree struct definition

5.4.1.1 Traversal

While there is no optimal search algorithm for these binary process trees, some will perform better in certain conditions. Pre-order DFS may perform better when looking for older processes as they are more likely to be in the upper branches. In-order and post-order DFS may perform better when looking for newer processes.

The Wetware process tree currently performs in-order DFS operations, but other implementations may be added in the future in order the select the best-fitted one as the tree evolves. To improve search efficiency, executors contain a thread-safe map of $PID \rightarrow Process$ to retrieve processes in *O(1)* constant time instead of the linear time it'd take to transverse the tree. Operations on the tree still require manual search to ensure the process is still in the tree, but other calls such as getting process information are significantly sped up this way.

`Find(pid uint32)` implements a rudimentary DFS algorithm that traverses the tree until it finds a node with the specified *PID*. `FindParent(pid uint32)` is used to find the parent of a process. It does so by first finding the process and then following a linear path upwards until the path either ends or turns right, signifying it has arrived at the parent process.

5.4.1.2 Addition

Adding a new node to the tree is done through an *Insert* method. *Insert*, defined in listing 5.8, works as follows:

1. Find the parent node of the new process by traversing the tree looking for a node with a PID that matches the new process' *PPID*.
2. Check if the parent node has a left branch, which represents a child process.
    (a) Condition 2 is true: starting with the left branch of the parent, iterate over the sibling processes and insert the new process as the right branch of the last sibling.
    (b) Condition 2 is false: insert the new process as the left branch of the parent.

```go
func (pt ProcTree) insert(p process) error {
    parent := find(p.ppid)
    if parent == nil {
        return errors.New("parent not found")
    }
    if parent.Left == nil {
        parent.Left = n
        return nil
    }
    next := parent.Left
    for next.Right != nil {
        next = next.Right
    }
    next.Right = &ProcNode{
        Pid: p.pid,
    }
}
```

Listing 5.8: ProcTree.Insert simplification

5.4.1.3 Deletion

There are three deletion-related operations implemented for the process tree. Firstly, `Pop(pid uint32)` removes a process and its children from the process tree, and fills the gap the process may have left with its next sibling (its right branch). Next, there is a `Kill(pid uint32)` method that calls the kill function of a process, as well as calling the `Kill` method of each process, explained in figure 5.12 of section 5.4.2. Lastly, there is a `Trim` function. Trim finalizes the processes that are not part of the tree. The current implementation should not allow this scenario to happen, but this method exists as a contingency.

5.4.1.4 Process Tree Benchmarking

Benchmarking the publicly exposed methods produces the results of listing 5.9. Each of the tests performs 10000 addition or deletion operations, and tests are repeated 100000 times in order to average the results. The first column contains the name of the test, the second column specifies how many times the addition or deletion operation was performed, and the third column shows the average time it took to perform each operation. The major performance bottleneck is the tree transversal. *InsertSibling* traverses the right branch until reaching the end, as opposed to *InsertChild* and *DeleteDesc* which also need to explore left branches. *DeleteAsc*, on the other hand, finds the node it is expecting at the very root of the tree every single time and is therefore much faster than the other methods. Repeating the tests using the thread-unsafe methods provided extremely similar results, making the performance impact of concurrency control methods negligible in strictly sequential use of the tree.

```
$ go test -bench=.
goos: linux
goarch: amd64
pkg: github.com/wetware/pkg/cap/csp/server
cpu: Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz
BenchmarkTree_InsertSibling-8   1000000000      0.1050 ns/op
BenchmarkTree_InsertChild-8     1000000000      0.2080 ns/op
BenchmarkTree_DeleteDesc-8      1000000000      0.2541 ns/op
BenchmarkTree_DeleteAsc-8       1000000000      0.0003308 ns/op
```

Listing 5.9: Process tree insert and delete benchmarks

5.4.2 Process Ending

As mentioned in section 5.2, when building a Wazero runtime for the Wetware executor the `WithCloseOnContextDone` must be set to true. This option allows calling the `cancel` function of the context passed to the WASM process instantiation, which will result in Wazero ending the process (section 5.2.1). Figure 5.12 shows a detailed layout of what happens when the *Kill* method of a process is invoked. It first causes the tree to stop its child processes, followed by removing the branch from the tree, stopping linked processes and lastly notifying monitors.

5.4.3 Process Execution Pause and Resume

To our understanding the Wazero runtime does not natively provide means of pausing and resuming the execution of processes. When asked about this topic through the Wazero Slack Channel the Wazero team replied:

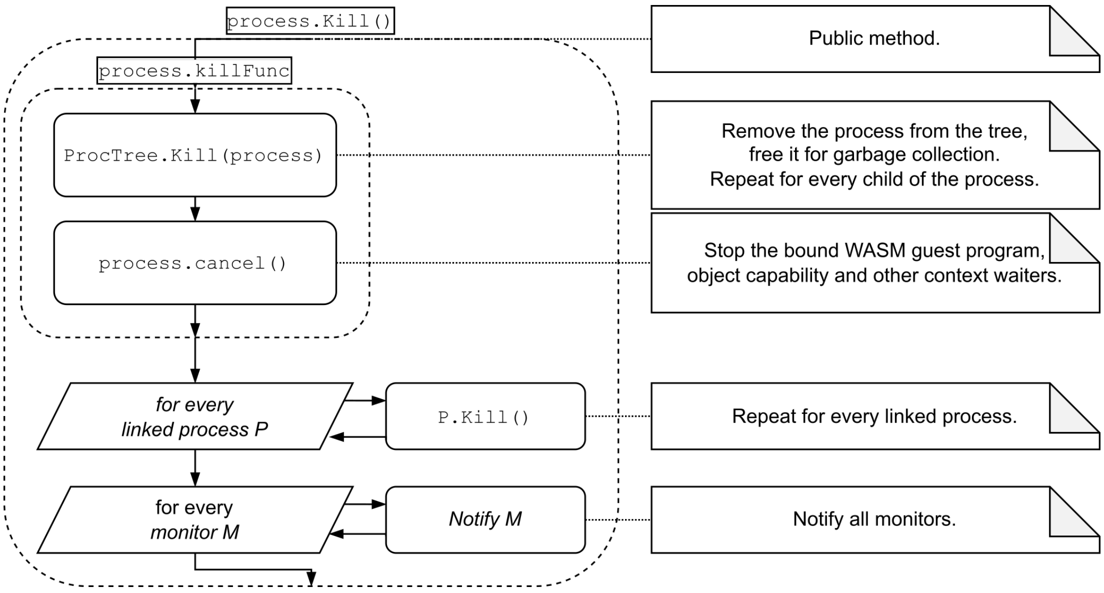

process.Kill()
process.killFunc
ProcTree.Kill(process)
process.cancel()
for every linked process P
P.Kill()
for every monitor M
Notify M
Public method.
Remove the process from the tree, free it for garbage collection. Repeat for every child of the process.
Stop the bound WASM guest program, object capability and other context waiters.
Repeat for every linked process.
Notify all monitors.

Figure 5.12: Step-by-step flow of process.Kill

> The runtime does not schedule processes, it's the application's responsibility to drive compilation, instantiation, and execution of the WASM modules; Wazero gives you the building blocks, then the application has full responsibility of deciding how scheduling is done. [59]

While it might possible to edit the Wazero source code to implement this functionality, it is a very demanding task that is not viable due to project scope and deadlines. Instead we opted to provide the means of sending pause and resume signals to the running process by means of an `EventHandler` capability.

```
struct Process {
    ...
    pause  @2 () -> ();
    resume @3 () -> ();
}
```

Listing 5.10: Process.Pause and Process.Resume RPC definition

The executor passes a `EventHandlerSetup` capability to the process on creation and waits for the process to start, creates an event handler, and provides its capability through a call to `EventHandlerSetups` only method. The executor then links said capability to the process capability, which is now able to provide `pause` and `resume` methods shown in listing 5.10.

The capability, implemented in listing 5.11, contains an event handler with two channels: one to signal a pause request and another for resume requests. Channels will only be written if they are empty to avoid buffering calls and unreliable behavior. The channels are made available to the process that acts upon the events via `OnPause` and `OnResume` methods that return read-only ends.

```go
type EventHandler struct {
    pause  chan struct{}
    resume chan struct{}
}

func (e EventHandler) Pause(ctx context.Context, call api.
   Events_pause) error {
    if len(e.pause) > 1 {
        return errors.New("already paused")
    }
    select {
    case <-ctx.Done():
        return ctx.Err()
    case e.pause <- struct{}{}:
    }
    return nil
}

func (e EventHandler) Resume(ctx context.Context, call api.
   Events_resume) error {
    if len(e.pause) > 1 {
        return errors.New("not paused")
    }
    select {
    case <-ctx.Done():
        return ctx.Err()
    case e.resume <- struct{}{}:
    }
    return nil
}

```

```
func (e EventHandler) OnPause() <-chan struct{} {
    return e.pause
}

func (e EventHandler) OnResume() <-chan struct{} {
    return e.resume
}
```

Listing 5.11: EventHandler capability implementation

Once the process starts, it may create an event loop to handle these signals. Listing 5.12 shows a basic implementation of an application that can pause and resume its execution. It is a simple web crawler that sends an HTTP request to a URL (not shown), scans it for more URLs (not shown), stores the newly found URLs in a queue, and repeats the loop by crawling the next URL in that queue.

Every time a URL is crawled, the code will arrive at the `select` statement. If during the crawl the pause method of the process' event loop was invoked, execution will be blocked until the resume method is called. A call to `runtime.Gosched` was added as a minor optimization, as this method will immediately yield the processor to other processes without needing to check the resume channel first. The executor can further define yielding behavior by means of the Wazero `WithOsyield` module configuration.

```
func main() {
    ...
    urls = make(chan string)
    for {
        select {
        case <-eventHandler.OnPause():
            runtime.Gosched()
            <-eventHandler.OnResume()
        case <-crawl(ctx, urls, <-urls):
        }
    }
 }
```

Listing 5.12: In-process event loop

The code of listing 5.13 contains a partial implementation of the crawl function, skipping the HTTP requesting and web-page parsing, that performs a task, notifies its completion via the returned channel and writes the results in a new goroutine. The write-end of the queue will block until the read-end is also accessed, thus it will also be paused when the main loop pauses.

```go
func crawl(ctx context.Context, urls chan string, url string) <-
    chan struct{} {
    // page := httpGet(url)...
    // parse page and create a url list in @results...
    var results []string

    done := make(chan struct{})
    go func() {
        done <- struct{}{}
        for _, r := range results {
            urls <- r
        }
    }()
    return done
}
```

Listing 5.13: Example of pausable crawl function

5.4.4 Process Linkage

The process linking feature developed in this section is defined in the following way: linking process A to process B establishes that ending either process A or B will also end process B or A, respectively. It is a variation of Erlang's linking [55], where process B will send an exit signal with the reason behind the exit upon B's exit. When approaching the design of this functionality it is worth considering that linking two processes that belong to the same executor is potentially more performant than linking processes across executors. For this reason, it was decided to provide two methods: a general one for processes that may or may not share an executor, and a local one for processes that wish to optimize linkage performance. For the rest of this sub-section, 'process' will refer to the Go structure wrapping the WASM process, and not the WASM process itself.

Let's start with the general approach. Each process now has a unique `id` method that returns a unique, randomly generated number assigned to it on its creation. Each process also has a thread-safe map M containing any number of capabilities referencing other processes, mapped by their `id`. As explained in section 5.4.2, every process P has deferred a call to `P.killFunc` which will be executed once the underlying WASM process ends or is canceled for any reason. When process A links to process B, it stores B in M_A it passes a reference to itself which B stores in M_B. B, or any other process, will call `B.killFunc` followed by a call to the RPC `P.Kill` for every process in M, which in this case will include `A.Kill()`. Termination will propagate along links so that in a scenario $A \leftrightarrow B \leftrightarrow C$ where $A \leftrightarrow B$ represents A's linking to B, terminating C will also result in A's termination. We'll refer to this $A \Leftrightarrow C$ relation as an *indirect* link caused by the propagation of B's termination, as opposed to *direct* links created by the RPC. Calls to unlink indirectly linked processes will have no effect.

```
interface Process {
    ...
    id     @4 () -> (id :UInt64);
    link   @5 (other :Process, roundtrip :Bool) -> ();
    unlink @6 (other :Process, roundtrip :Bool) -> ();
}
```

Listing 5.14: Process.Link and Process.Unlink RPC definition

The capability defined in listing 5.14 and implemented on listing 5.15 allows linking processes, as long as the connection between them doesn't fail. As in Erlang, if a link or unlink is called on already linked or un-linked processes, no operation will be performed and no error will be returned. Any invocation of `A.Link(B)` implies an implicit invocation from A to `B.Id()`, as well as a round trip call `B.Link(A, true)`.

```
type process struct {
    ...
    linked *sync.Map
}

func (p *process) Link(ctx context.Context, call api.Process_link
   ) error {
```

```go
    other := call.Args().Other() // skip error management
    f, _ := other.Id(ctx, nil)   // get future of Id RPC
    otherId := f.Id()
    p.links.Store(otherId, other)
    if !call.Args().Roundtrip() {
        f, _ := other.Link(ctx, func(args api.Process_link_Params
) error {
            args.SetRoundtrip(true)
            return args.SetOther(api.Process_ServerToClient(p))
        })
        <-f.Done()
    }
    return nil
}

func (p *process) Unlink(ctx context.Context, call api.
    Process_unlink) error {
    other, _ := call.Args().Other() // skip error management
    f, _ := other.Id(ctx, nil) // get future of Id RPC
    otherId := f.Id()
    p.links.Delete(otherId)
    if !call.Args().Roundtrip() {
        f, _ := other.Unlink(ctx, func(args api.
    Process_unlink_Params) error {
            args.SetRoundtrip(true)
            return args.SetOther(api.Process_ServerToClient(p))
        })
        <-f.Done()
    }
    return nil
}

func (p *process) Kill(ctx context.Context, call api.Process_kill
) error {
    p.kill()
    return nil
}

func (p *process) kill() {
    // defer iterating over linked processes am killing them
    defer p.linked.Range(func(key, value any) bool {
        value.(*process).Kill(ctx, nil)
        return true
    })
    // but kill p first
    p.killFunc(p.pid)
}
```

Listing 5.15: Changes made to process for (un)link implementation

Processes that share an executor, however do not need to pay this performance penalty. Two alternative calls, defined in listing 5.16, allow linking process *A* to *B* by providing *A*'s PID. Upon termination, *B* will call the private function `A.kill()` directly instead of going through the `A.Kill()` RPC. Termination will be propagated all the same, but linkages set up this way will avoid RPCs. Processes now have two separate maps to track links and local links, as well as perform two iterations on termination. Go allows for using the same map for different key and value types, nevertheless, type casting and checking would introduce a small penalty, as well as allowing potential collisions between link IDs and local link PIDs.

```
interface Process {
    ...
    linkLocal   @7 (other :UInt32) -> ();
    unlinkLocal @8 (other :UInt32) -> ();
}
```

Listing 5.16: Process.LinkLocal and Process.UnlinkLocal RPC definition

Ideally, *B* would be able to check if *A* is a local process implicitly without needing to discern two distinct methods. The current implementation of `go-capnp` does not, to the best of my knowledge, provide the tools necessary for it.

5.4.5 Process Monitoring

Monitoring a process will trigger an event when the monitored process stops. In Wetware's case, processes are monitored through a *Monitor* RPC that is held by the monitored process, blocking the call at the monitoring client side. When the monitor ends, its capability is released and the monitor is unblocked. It is a concept similar to Erlang's monitors, which according to the documentation [78] are defined as follows: "*Monitors are fired when the monitored process terminates, does not exist at the moment of creation, or if the connection to it is lost.*"

The monitor implementation is perhaps the most straightforward. Upon receiving a monitor call, the goroutine processing the call is blocked writing to a queue (a channel) of monitors. The queue is only consumed at process termination, unblocking the call and thus notifying the monitor. Monitors can stop monitoring a process P by releasing their call to $P.Monitor$.

5.4.6 Process Listing

Process listing takes place at two levels: executor level and cluster level. The cluster level is explained in section 5.5. Regarding the executor level, executors have a *Ps* call that provides read-only information about its running processes. This information, which is encoded as a Cap'n Proto message of type *Info*, contains the following fields: *pid*, *ppid*, *cid*, *args*, and *creation time*. The executor assigns the values of each of these fields to processes upon creation, and the process implementation has a `info` method that returns an *Info* message populated with the corresponding values. As shown in the simplified listing 5.17, upon receiving a *Ps* call, executors:

1. Create a snapshot of the process tree.
2. Allocate a new list for Cap'n Proto messages of our custom *Info* type.
3. Iterate over the snapshot and populate the list.
4. Return the list as the result of the call (not shown).

```go
func (r Runtime) Ps(ctx context.Context, call core_api.
    Executor_ps) error {
    ...
    snap := r.Tree.MapSnapshot()
    _, seg := capnp.NewSingleSegmentMessage(nil)
    pl, _ := proc_api.NewInfo_List(seg, int32(len(snap)))

    i := 0
    for _, v := range snap {
            info, _ := v.(*process).info()
            pl.Set(i, info)
            i++
    }
    ...
}
```

Listing 5.17: Iteration over processes on an executor for process listing

5.5 Integration into Wetware

Integrating the executor into Wetware requires ensuring its compatibility with the rest of the components and configuring the life-cycle of the executor throughout the lifespan of a node. It mostly boils down to two steps: initialization and destruction. Destruction is arguably the simplest of the steps. By ensuring the context running the node is propagated to the executor, Wazero runtime, and the spawned processes. It ensures that once the node stops, so do the executor, runtime, and processes. This is of course, as long as these requirements are met:

- The Wazero runtime is configured with `WithCloseOnContextDone(true)`, for propagating stops to processes.
- Any *select* statements that might prevent a correct shutdown must contemplate the case of the context ending.
- Every spawned goroutine is tied to a context bound to the “original” context.

Initialization, on the other hand, refers to a wider set of considerations. First and foremost is giving nodes access to an executor. Wetware implements each node as a *Server* structure, thus giving it access to an executor is as simple as giving it an attribute of type *Runtime*, which is the name given to the executor implementation. Next comes initializing the executor when the *Server* is initialized by configuring and instantiating it in the server constructor. The most important consideration is correctly configuring the Wazero runtime depending on the architecture, as it only supports its “compilation mode” in *amd64* and *arm64* architecture. This mode is considerably faster than its “interpreter mode” [79] as it natively executes code as opposed to interpreting it. Listing 5.18 shows a simplification of the construction of an executor as part of the *Server* constructor.

```go
...
if runtime.GOARCH == "amd64" || runtime.GOARCH == "arm64" {
    conf.RuntimeConfig = wazero.
        NewRuntimeConfigCompiler().
        WithCompilationCache(wazero.NewCompilationCache()).
        WithCloseOnContextDone(true)
} else {
    conf.RuntimeConfig = wazero.
        NewRuntimeConfigInterpreter().
        WithCompilationCache(wazero.NewCompilationCache()).
        WithCloseOnContextDone(true)
}
r := wazero.NewRuntimeWithConfig(ctx, conf.RuntimeConfig)
wasi_snapshot_preview1.Instantiate(ctx, r)
executor := csp_server.Runtime{
    Runtime: r,
    Cache:   make(csp_server.BytecodeCache),
    Tree:    csp_server.NewProcTree(ctx),
}
...
server := Server{
    ...
    ExecutorProvider: executor,
}
```

Listing 5.18: Integrating the executor into the node constructor

5.5.1 Command-Line Interface

The ability to execute a WASM process in a Wetware cluster is also provided as part of the Wetware CLI. There is already a suite of CLI features that simplify connection and discovery, making the command implementation straightforward. Before proceeding, however, we must understand how to connect to a cluster. A user entry point to a Wetware cluster is a *Terminal*. Terminals are capabilities with a single method: *Login*. Login receives a *Signer* that acts as a user identifier, and grants a session to the user based on its internal policies. The session can be nil, or provide access to some or all of its capabilities. Listing 5.19 contains the schemes of both Terminal and Session. Once a user has access to a session, and that session has been granted access to the executor, the capability holder is free to run WASM code.

```
interface Terminal {
    login @0 (account :Cluster.Signer) -> (session :Session);
}

struct Session {
    local          :group{
        peer   @0 :Text;     # peer.ID
        server @1 :UInt64;   # routing.ID
        host   @2 :Text;     # hostname
    }

    # Access-controlled capabilities.  These will be set to null
    # unless permission has been granted to use the object.
    view       @3 :Cluster.View;
    executor   @4 :Executor;
    capStore   @5 :CapStore.CapStore;
    extra      @6 :List(Extra);

    struct Extra {
        name   @0 :Text;
        client @1 :Capability;
    }
}
```

Listing 5.19: Definiton of Terminal and Session

Listing 5.20 contains a simplification of the full `ww cluster run` command, where only 2 lines are dedicated to networking and logging it, thanks to the streamlining of the process by Wetware. The rest of the code focuses on opening a WebAssembly file, parsing command-line arguments, and executing the WebAssembly content. It is worth noting that the CLI passes the arguments it received, excluding the name of the WASM file, to the process. The command uses 0 as the *PPID* of the new process, which the executor interprets as a new top-level process. If the WASM process is to create more sub-processes, it should pass its own *PID* as their *PPID*. When the command exits it releases the process capability and closes the connection, canceling the context of the real process held by the executor to end, and the WASM process to be stopped. This is then propagated to its child processes.

```go
func run() cli.ActionFunc {
    return func(c *cli.Context) error {
        wasm, _ := os.ReadFile(c.Args().First())
        // Prepare argv for the process.
        args := []string{}
        if c.Args().Len() > 1 {
            args = append(args, c.Args().Slice()[1:]...)
        }
        // Get a session.
        h, _ := vat.DialP2P()
        sess, close, _ := BootstrapSession(c, h)
        defer close()
        // Execute the process
        p, release := sess.Executor().
            Exec(c.Context, api.Session(sess), wasm, 0, args...)
        defer release()
        return p.Wait(c.Context)
    }
}
```

Listing 5.20: Cluster-run command

5.5.1.1 Guest Bootstrapping and QoL

Wetware processes have to log into clusters through terminals, as any Wetware user would. It is something that should be done by every process and has the potential to involve arguably hard to understand instructions, potentially presenting an initial roadblock for developers that choose to use Wetware. For this reason, login into a terminal and other common tasks are implemented as one-line, single invocation functions that users can import from the `guest/system` package. The web crawler application discussed later on this chapter imports `guest/system` as `ww`; and uses these features by getting a Wetware session simply by calling `ww.Bootstrap(ctx)` as well as retrieving the command-line arguments with `ww.Args()`. It could also get its own attributes through the following functions: `ww.PID()`, `ww.PPID()`, `ww.CID()`.

5.5.2 Process Listing

Besides the *exec* method, Wetware can list processes through its command line tool. The command, `ww ps`, connects to a Wetware cluster and lists all running processes across all the nodes in the cluster by iterating over the view of the initial node and calling the *Ps* method of the executor of each of the nodes. Listing 5.21 shows a cropped sample of the output when running the web crawler application.

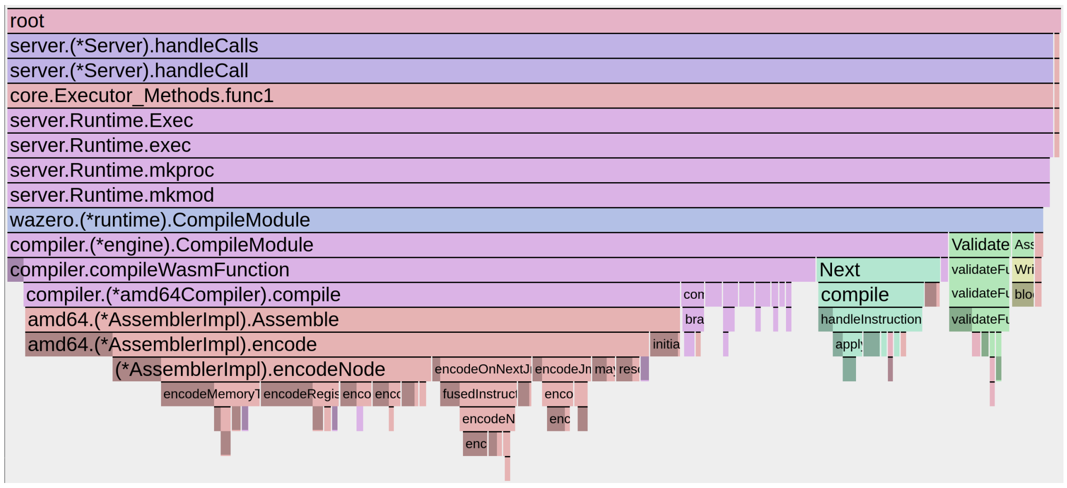
```
Executor         PID PPID Creation            CID          Args
d0ef56e03d7a10e8 3   2    Mon Oct 9 21:...    z26L4mZvv... [...]
d0ef56e03d7a10e8 4   2    Mon Oct 9 21:...    z26L4mZvv... [...]
d0ef56e03d7a10e8 5   2    Mon Oct 9 21:...    z26L4mZvv... [...]
d0ef56e03d7a10e8 6   2    Mon Oct 9 21:...    z26L4mZvv... [...]
d0ef56e03d7a10e8 7   2    Mon Oct 9 21:...    z26L4mZvv... [...]
d0ef56e03d7a10e8 8   2    Mon Oct 9 21:...    z26L4mZvv... [...]
d0ef56e03d7a10e8 9   2    Mon Oct 9 21:...    z26L4mZvv... [...]
d0ef56e03d7a10e8 2   1    Mon Oct 9 21:...    z26L4mZvv... [...]
```

Listing 5.21: Output of the “ww ps” command

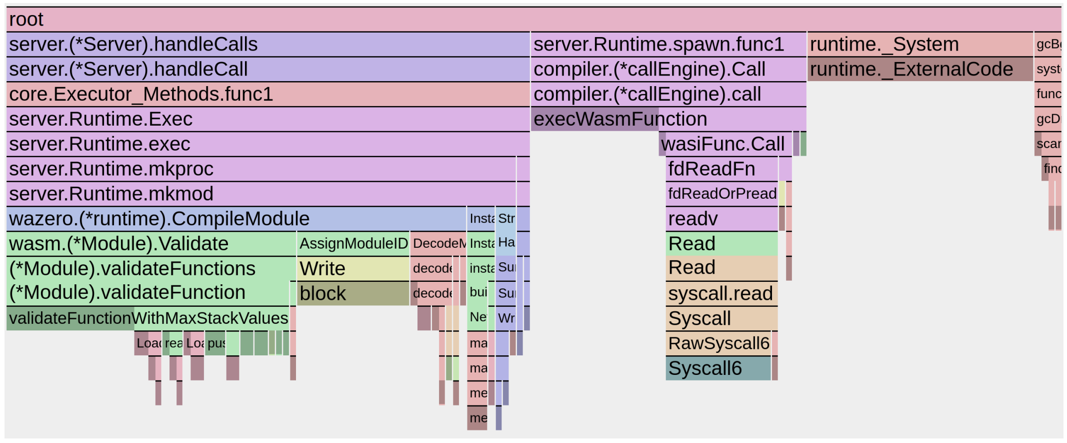
5.6 Caching

Profiling the basic *Executor.Exec* method showed that the vast majority of the time was spent compiling the WASM bytecode, as shown in figure 5.13a. The function performs computational work and has no internal asynchrony, thus any other goroutine cannot take advantage of that time. For this reason, Wazero provides the option of using a compilation cache which will significantly shorten the compilation process by only requiring some final decoding and validation steps. The compilation cache, provided natively by Wazero, is set through the

`WithCompilationCache(wazero.NewCompilationCache())` runtime option. Figure 5.13b contains a flamegraph of *Executor.Exec* when the provided bytecode has already been compiled, and the compilation cached. It is not only clearer, but an order of magnitude more performant, as seen later down this section in figure 5.14.



(a) No caches



(b) Compilation cache

Figure 5.13: Flamegraphs of Executor.Exec with and without caching compilation

Wetware clusters may have varying network latency and throughput. Currently, spawning a process through the *Exec* RPC requires sending the full bytecode of the process over the wire. To counter this drawback, we created a second cache to store un-compiled bytecodes. The cache has a simple interface akin to a map, as shown in listing 5.22, with `put`, `get`, and `has` methods. It associates bytecodes to their content ID, or CID, generated from hashing the raw bytes composing the bytecode.

Every process knows its own CID, and is guaranteed to have its own bytecode on the cache of the executor running it. Any process can thus replicate itself with just its CID on its local executor, or it can retrieve its bytecode from the local cache and propagate into new executors.

```
interface BytecodeCache {
    put @0 (bytecode :Data) -> (cid :Data);
    get @1 (cid :Data) -> (bytecode :Data);
    has @2 (cid :Data) -> (has :Bool);
}
```

Listing 5.22: Bytecode Cache definition

A bytecode can be reduced to a wrapper around a thread-safe map having CIDs as keys and bytecodes as values. By keeping track of when bytecodes were last stored or loaded, it can drop the ones that have gone unused the longest time when it reaches its maximum size; in order to accommodate new bytecodes. The implementation of this size management method will be dropped from the current design in favor of keeping track of how many ongoing processes belong to each bytecode, to be implemented as future work later down the road.

Spawning processes through CIDs is done through `Executor.ExecCached` instead of `Exec`. As a preventive optimization, the executor automatically stores bytecodes received through `Exec`. This way it guarantees that running processes have their bytecode stored in the cache. In the same way, it guarantees any process run through `ww cluster run <wasm>` can replicate itself locally without worrying about retrieving the bytecode. Processes can verify the bytecode retrieved from a cache by generating its CID and comparing it to their own. On the other hand, sensitive processes should only be run on trusted clusters as they have no way of verifying that an executor runs the bytecode unmodified [80].

```
interface Executor {
    ...
    execCached @1 (cid :Data, ppid :UInt32, bctx :BootContext) ->
    (process :Process);
}
```

Listing 5.23: ExecCached method definition

Wetware is balancing the idea of implementing a global cache on top of the bitswap protocol [81], allowing executors to share and exchange shards of bytecodes and potentially improving performance as the number of nodes in a cluster grows. The proposal would simplify the development by enabling the replication of processes in new executors by providing just a CID. For these reasons it might be implemented in the future.

Profiling the same function as in figure 5.13 to observe the performance improvements of the bytecode cache will provide no result, as the bytecode cache's purpose is to reduce the bandwidth used by calls to *Exec*, which will not reflect on the profile of the function itself. Instead, figure 5.14 shows the results of benchmarking remote calls to *Exec*. Bear in mind that the effectiveness of the bytecode cache will vary. In the case of figure 5.14 the bytecode was small and RPCs were performed on the same device. The performance gain will be inversely proportional to the bytecode size and the decrease in network throughput.

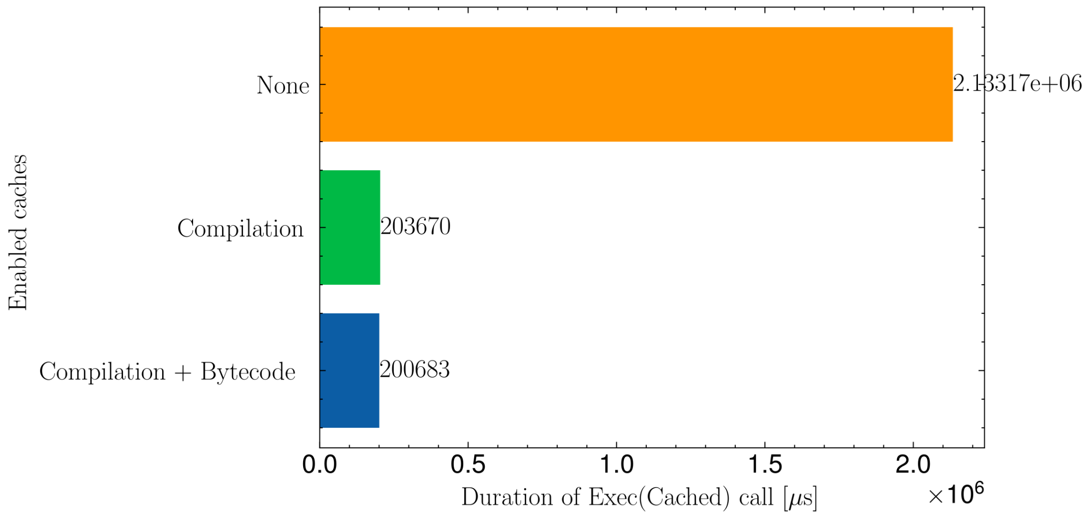


Figure 5.14: Exec(cached) execution time improvements with caches

The final diagram containing the main components of an executor is shown in figure 5.15, caching WASM programs on a Wetware and Wazero level.

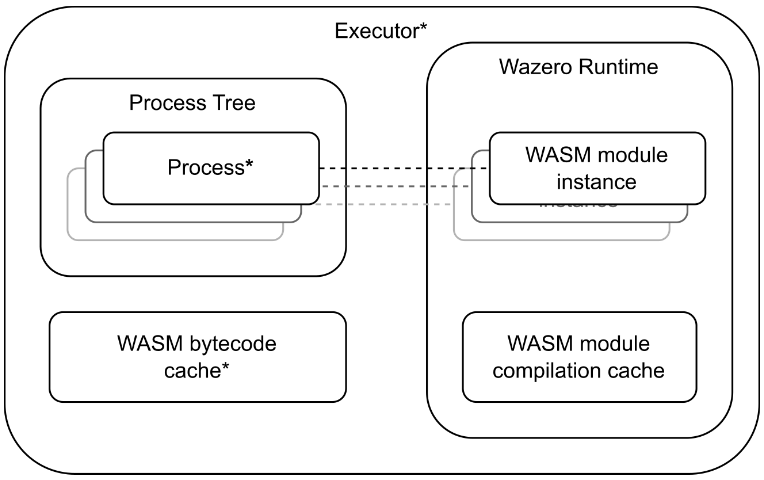


Figure 5.15: Final main component diagram of the executor

5.7 Proof-of-Concept

This section contains a proof of concept of the executor's capabilities when running on one or multiple Wetware nodes. The most basic demonstration is a simple application that spawns new processes to perform an action, ensuring processes can communicate and have access to the same object capabilities. After that, development takes a detour from validation to create a Cap'n Proto and WebAssembly compatible Raft library [64] based on Etcd's Raft implementation [24]. Lastly, the validation is expanded to make better use of its resources, use a distributed log, and run in a stable manner; simulating a more complex application that runs on and benefits from Wetware. The application is available in a public git repository [63].

5.7.1 Prerequisites

Before delving into the proofs of concepts, there are some functional prerequisites necessary to make everything work.

5.7.1.1 Channels

Channels are inter-process communication tools that were already part of Wetware when this thesis started. Creating them is straightforward, it's enough to import the `comm` package and instantiate them as `channel := new(comm.SyncChan)`. Once instantiated, their capability is generated with `comm.NewChan(channel)`.

5.7.1.2 Capability Storage

Capability stores were developed as part of this thesis. A capability storage is used to store and retrieve object capabilities. By holding object capabilities it keeps their connections alive and prevents their providers from being freed or garbage collected by ensuring they have at least one active reference. Each Wetware node has a *CapStore* capability, defined in listing 5.24. Both storing and retrieving capabilities require providing a unique string, used to identify the capability.

```
interface CapStore {
    set @0 (id :Text, cap :Capability) -> ();
    get @1 (id :Text) -> (cap :Capability);
}
```

Listing 5.24: Capability storage definition

5.7.2 HTTP Requests

WebAssembly modules cannot natively perform HTTP requests. It is a good opportunity to showcase the simplicity of adding functionality to the Wetware process. We implemented a simple HTTP Requester capability by wrapping some native Go HTTP functionality, namely GET and POST requests, in Cap'n Proto RPCs, shown in listing 5.25.

```
interface Requester {
    get  @0 (url :Text) -> (response :Response);
    post @1 (url :Text, headers :List(Header), Body :Data) -> (
    response :Response);

    struct Header {
        key   @0 :Text;
        value @1 :Text;
    }
    struct Response {
        status @0 :UInt32;
        body   @1 :Data;
        error  @2 :Text;
    }
}
```

Listing 5.25: Capability storage definition

5.7.3 Initialization

Running the web crawler applications developed later in this section requires two steps:

1. Initialize the HTTP provider as a standalone Go application, and save its capability in the capability storage. Both these actions are performed by a short Go program that uses Wetware as any Go process would.
2. Run the crawler as a Wetware program through the command line: `ww cluster run <crawler.wasm>`.

The repository [63] contains a shell script named `run.sh` that takes care of both steps. To run the web crawler, it creates a Wetware cluster through `ww start` and starts the crawler with `./run.sh <nprocs> <entrypoint>`, where `nprocs` is the number of processes the web crawler will use and `entrypoint` is the URL where the application will start its crawl. The repository also contains a `Makefile` that builds the required executable and WebAssembly files.

5.7.4 Basic Validation

Basic validation is performed with a rudimentary web crawler that starts with a coordinator node. The coordinator then spawns an arbitrary number of workers, as well as a channel for each one of the workers. It passes the identifiers of the HTTP requester capability and the capability of their corresponding channel as arguments, so worker processes can fetch both capabilities when they start. Page crawling is carried out with regular expressions, see *crawler/http.go* [63].

The coordinator will:

1. Create N channels. The queue will have one value at creation, the starting URL. Spawn N workers, assign one channel to each.
2. Create two queues: one for URLs and one for channels.
3. Put every channel in the channel queue.
4. Put the entry-point URL in the URL queue.
5. *Loop*:
    (a) Get a channel from the channel queue, get a URL from the URL queue, send the URL through the channel.
    (b) Create a new goroutine that will:
        i. Wait for a message from the channel.
        ii. Put the channel back in the queue.
        iii. *If* the message is empty, do nothing.
        iv. *If* the message contains URLs, put the URLs in the queue.
6. End sub-processes.

Workers will:

1. Retrieve the HTTP requester and channel capabilities.
2. Wait for a URL to arrive through the channel.
3. Crawl the URL, extract links, and send them through the channel.
4. GOTO 2.

Implementation of the Wetware application is relatively straightforward. While inefficient, it makes use of the following features developed during the thesis:

- Utilize the `ww cluster run` command to run a Wetware process on a Wetware node.
- Bootstrapping and utilizing object capabilities from WASM guests, via asynchronous host-guest communication.
- Spawn sub-processes from a Wetware process.
- Communicate between processes.
- Manage a processes by killing them.

Not all applications will match this centralized coordinator-worker model, however. Distributed applications will usually need some level of resiliency and consensus, so they can continue working successfully when a process fails. In this application, the crawler will stop if the coordinator stops. Furthermore, there is a clear bottleneck in which every worker needs to wait for an assignment from the coordinator, which may overload the coordinator as the number of workers increases, as well as cause additional latency when running the crawler in multiple nodes. The following section, detailing our Raft library, lays the foundation to overcome these shortcomings.

5.7.5 Raft Over Cap'n Proto

Running Raft with object capabilities as transport required complementing the *etcd-raft* library. Many of the implementation details are not relevant but can still be found in the repository hosting our implementation [64], which started off as an adaptation of a library that used Protocol Buffers as a transport [82] but quickly diverged and turned into a noticeably distinct project. The final result is a general-purpose library that can be used not only for Wetware processes, but any type of application.

Our implementation is based on *Node* structures, which represent Raft nodes. They are exposed through the object capability shown in listing 5.26. If Raft nodes are started from and embedded into another Go application, this application can interact directly through Go methods without requiring capability usage. Raft nodes do, however, interact with each other exclusively through capabilities. A connection between two Raft nodes occurs when they hold each other's capabilities.

```
interface Raft {
    add     @0 (node :Raft) -> (nodes :List(Raft), error :Text);
    # Propose adding $node to the Raft.
    remove  @1 (node :Raft) -> (error :Text);
    # Propose removing $node from the Raft.
    send    @2 (msg  :Data) -> (error :Text);
    # Send a message to the Raft node.
    put     @3 (item :Item) -> (error :Text);
    # Put an k/v pair on the cluster.
    items   @4 ()           -> (objects :List(Item));
    # List every k/v pair in the cluster.
    members @5 ()           -> (members :List(Raft));
    # List every memeber of the cluster.
    id      @6 ()           -> (id :UInt64);
    # Get the Raft ID of the node.
}
```

Listing 5.26: Raft node capability definition

While it is not feature-complete, we followed the implementation instructions in the *etcd-raft README* [24]. The most relevant missing feature is creating and recovering from snapshots, which is not required for our web crawler. *Nodes* are responsible for sending out heartbeats, finding peers, processing and storing values, as well as handling events. Every other Raft feature is provided by Etcd's library, which just needs importing and initializing.

Sending heartbeats is very straightforward: a loop will send out a heartbeat periodically, based on a period value configured when creating the node.

Finding peers, and processing and storing values are more challenging features. Because of Wetware's process isolation and quickly evolving nature, we opted for leaving implementation up to the user by requiring them to implement the types listed in listing 5.27. All of them are implemented by the final version of our web crawler, as described later in section 5.7.6. Nodes rely on *RaftStore* to store values, *RaftNodeRetrieval* for obtaining the capabilities of a node given that node's ID, and *OnNewValue* may optionally be used to run an arbitrary function each time a new value is received.

```
type RaftStore func(storage raft.Storage, hardState raftpb.
    HardState, entries []raftpb.Entry, snapshot raftpb.Snapshot)
    error
type RaftNodeRetrieval func(context.Context, uint64) (api.Raft,
    error)
type OnNewValue func(Item) error
```

Listing 5.27: Configurable raft function types

The heart of the library is the *Node.Start* method, simplified in listing 5.28. It initializes the listing, and loops indefinitely reacting to events. The *pause* and *stop* cases will temporarily pause or permanently stop the node, and canceling the context for whatever reason will cause *stop* to trigger. The most important method is `doReady`, whose flow diagram is represented in figure 5.16. It must be noted that the *addNode* method shown in the diagram will receive a Raft node ID and call *RaftNodeRetrieval* to obtain its capability.

```
func (n *Node) Start(ctx context.Context) {
    n.Init()
    var err error
    for {
        select {
        case <-n.ticker.C:
            n.Raft.Tick()
        case ready := <-n.Raft.Ready():
            err = n.doReady(ctx, ready)
        case pause := <-n.pauseChan:
            err = n.doPause(ctx, pause)
        case <-ctx.Done():
            err = ctx.Err()
        case err := <-n.stopChan:
            defer close(n.stopChan)
            defer n.doStop(ctx, err)
            return
        }
        if err != nil {
            go func() {
                n.stopChan <- err
            }()
        }
    }
}
```

Listing 5.28: Node.Start simplification

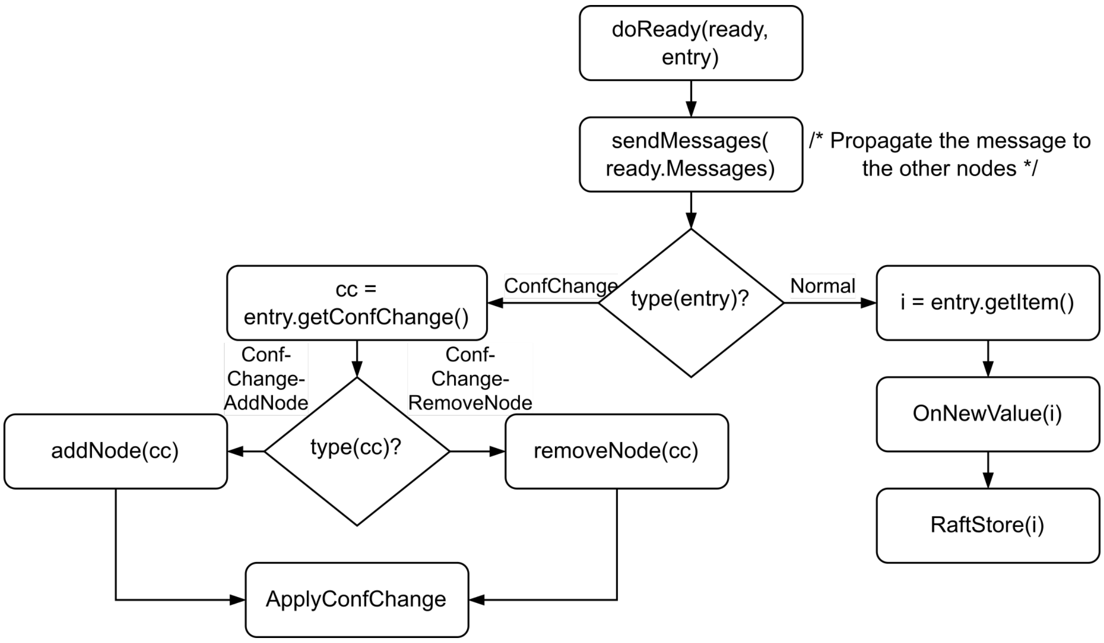


Figure 5.16: Node.doReady flowchart

By default, *etcd-raft* is designed to use fast, permanent storage for the log. It does, however, provide the means to use custom storage implementations as well as having an in-memory implementation. The solution developed for this project exposes the configuration of the storage via a `WithStorage(raft.Storage)` method, very similar to how the original implementation does. Because it was designed with Wetware processes in mind, nodes use in-memory storage by default as writing to disk would break process isolation. This is the default behavior but is not always enforced, as Wazero allows exposing parts of the file system to WASM modules and could therefore be supported by Wetware in the future. The `raft.Storage` might also be implemented with underlying uses of object capabilities to a resource that accesses permanent storage, though the performance cost when compared to direct access would be significant. It should lastly be noted that messages are encoded using JSON, and *etcd-raft* provides the means to encode them in the Protocol Buffer format.

Raft is now ready to be used in our application, as long as we provide *RaftNodeRetrieve*, *RaftStore* and optionally *OnNewValue* functions.

5.7.6 Feature-Rich Validation

The final version of the crawler, found in the git repository main branch [63], has no centralized coordinator. The initial process does have the responsibility of creating the rest of the processes, but once that is done all nodes crawl URLs and share their findings through the distributed log. They act as autonomous actors, and the web crawling application will keep functioning even if multiple processes fail or there is some kind of network partition.

Before proceeding into the inner workings of the application, let's first describe the *RaftStore*, *RaftNodeRetrieval*, *OnNewValue* implementations. *RaftStore* uses the default implementation provided by etcd to write the values to memory. Regarding *RaftNodeRetrieval*, every process stores the capability of its Raft node in the capability storage during initialization, using its Raft ID as the key. The actual implementation is shown in listing 5.29. *OnNewValue* handles URLs, as described in the next paragraph.

```go
func (c *Crawler) retrieveRaftNode(ctx context.Context, id uint64
  ) (raft_api.Raft, error) {
    r, err := c.CapStore.Get(ctx, c.idToKey(id))
    if err != nil {
        return raft_api.Raft{}, nil
    }
    return raft_api.Raft(r).AddRef(), nil
}
```

Listing 5.29: RaftNodeRetrieval implementation

Each crawler node keeps track of URLs in four separate structures. Firstly, a fixed-sized local queue *LQ* with the URLs this particular crawler will crawl next. Nodes inform others of their intention to crawl a URL by claiming it. Each crawler stores the claims from the rest of the crawlers in a *CM* claim map, which times each claimed to the time it was claimed. Claims have a maximum duration of 5 minutes, after which they are evicted to a global queue GQ. The global queue contains every unclaimed URL that's been discovered but hasn't been crawled yet. When nodes run out of URLs in their LQ, they claim a number of URLs from GQ. URLs visited by any node are stored in a set of visited URLs VS. When crawling a page, nodes extract the URLs it references and filter out the ones already in VS. They put $max(x, y)$ URLs in their LQ, where x is half of the discovered URLs and y are the free slots left in LQ.

Coming back to *OnNewValue*, nodes communicate their visits, findings and claims through the distributed log; removing the need for Wetware channels. Upon receiving a new value representing one of those three actions, they update their VS, GQ and CM respectively. This is implemented in $OnNewValue$.

When developing this iteration of the web crawler we came upon the issue: if processes are running on different executors, and each executor has one capability storage associated, should they all store every capability on every capability storage of the cluster, or share the same one? We opted for the former because it would be simpler to implement in this specific case. Upon creation, each process fetches and store a session from every peer in the cluster through their original session, storing them in their *Cluster.Sessions* attribute. Registering capabilities is done through a *storeCap* method that iterates over the sessions and stores the capabilities in all of them. We modified the HTTP requester provider as well, as that also needs to be accessible from every capability store. This issue may present a roadblock for new Wetware users, which has been duly noted and will be worked on in the near future.

The initial process evenly spawns other processes across the cluster, as shown by listing 5.30. It shows a fragment of the method used by the initial process to spawn the rest of the processes.

```
func (c *Crawler) spawnCrawlers(ctx context.Context, n uint64)
    error {
    sessions := c.Cluster.Sessions
    sessions = append(sessions, c.Session)
    for i := uint64(1); i < uint64(n); i++ {
        s := sessions[int(i)%len(sessions)]
        e := s.Executor()
        _, release := e.ExecCached(
            ctx,
            core.Session(s),
            ww.Cid(),
            ww.Pid(),
            <args>...,
        )
        defer release()
        ...
    }
    ...
    return nil
}
```

Listing 5.30: Fragment of spawnCrawlers

Finally, we did some performance tuning to the application taking advantage of its heavily IO-bound workload. Crawlers now perform multiple "simultaneous" HTTP requests, that while not truly parallel, do increase the output of the nodes. Increasing the *LQ* size also seems to reliably improve performance up to a point by reducing the number of items pulled from *GQ*. The web crawler has, however, the purpose of validating the executor and its capabilities and not measuring its performance.

Chapter 6

Evaluation

The evaluation was performed utilizing the hardware specified in section 4.5. Unless explicitly stated otherwise, tests and measurements were carried out on the desktop PC. All the tool-chain and software libraries required to build and perform these tests are available through a custom Docker image based on Alpine Linux, available to use at a personal Wetware fork [62].

6.1 Validity

This section evaluates whether the executor and the rest of the software developed throughout this thesis fulfill their objectives. Unitary tasks, such as ending or linking processes, were simple to validate with very short-lived tests. Nonetheless, the executor is not meant to run in isolation. It will be used as part of a distributed applications middleware and therefore requires a distributed application to be validated: the web crawler.

The final web crawler version was run in a Wetware cluster formed by the three devices listed in section 4.5, which shared the same local network. The final demonstration ran 24 processes evenly distributed across devices and created the initial process on the desktop PC. It runs for 30 minutes. The target URL was provided by a custom HTTP server running on the Raspberry Pi and written by us [83], to effectively measure URL conflicts between nodes as well as overall performance.

Listing 6.1 shows a fragment of the output of `ww ps` when running the web crawler application in the three-node cluster. The full IDs of the executors running in the desktop PC, laptop, and Raspberry pi are *38908fe2b3d81d435*, *7158b49af891e7b25*, and *9214905b4efea7a19*, respectively. The initial process was spawned in the desktop PC.

```
Executor  PID  PPID  Creation      CID     Args
38908...  2    1     Wed Oct 11... z26...  [iPH...]
38908...  3    2     Wed Oct 11... z26...  [iPH...]
92149...  2    1     Wed Oct 11... z26...  [j92...]
92149...  3    1     Wed Oct 11... z26...  [j92...]
7158b...  2    1     Wed Oct 11... z26...  [mLw...]
7158b...  3    1     Wed Oct 11... z26...  [mLw...]
```

Listing 6.1: Output of “ww ps” when running the web crawler

As a fancy addition for debugging and visualizing the results, we’ve also added a *Neo4j* capability for nodes to store their results in a graph database that allows us to more easily evaluate the correctness of the crawler. Database credentials are provided as command-line arguments to `run.sh` and therefore *ww cluster run*, which the initial process then propagates to the rest of the processes. Write operations against the database are performed by sending *POST* requests reusing the HTTP requester capability, see *crawler/neo4j.go* [63].

These last two factors, combined with the logs produced by the crawler processes and the output of `ww ps`, allow us to verify that:

- The web crawler was performing its function correctly.
- The application was running in a distributed manner throughout a Wetware cluster.
- The application was running on different hardware architectures.
- Processes in the cluster could transparently communicate with each other, as well as make use of other capabilities provided outside their node.

- Processes reached consensus through Raft nodes.
- The executor is stable, allowing our crawler to run at a steady pace for 30 minutes.
- The executor is successfully integrated into Wetware.
- The executor and processes can be managed through their object capabilities.

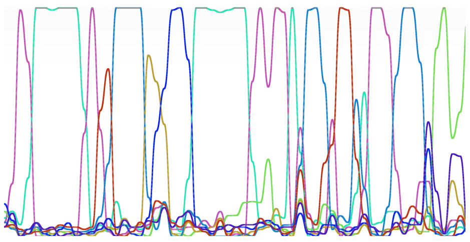
The initial rudimentary version of the executor allowed us to escalate the web crawler to hundreds of processes, but the final version has significantly more communication overhead and heavier process requirements, starting show signs of slowing down after a couple of dozen of processes; most likely due to the communication and synchronization overhead.

6.2 Performance Characterization

This section provides some insights into the performance results obtained by evaluating the executors under intensive workloads, in addition to explaining how the tests were carried out.

Measuring how the basic executor performs provides insight into how Go's scheduler affects the runtime and things to consider when developing programs that will run on the Wetware executor. The number of logical cores was limited either through manual CPU status configuration through the `/sys/devices/system/cpu/cpuX/online` file, or through setting the environment variable `GOMAXPROCS`, which limits the amount of Ps a Go process can use at any given time. Alternatively, the Docker image allows limiting the process' access to certain cores through the `cpuset` parameter.

Limiting the number of Ps on an OS level might have a heavier footprint on the tests that use fewer cores due to having to use the same cores for both running the test program and doing regular OS tasks. On the other hand, limiting Ps through `GOMAXPROCS` allows the processes to run in "freer" cores but might involve more context switches: while `GOMAXPROCS` ensures only a certain number of them are used at any given time without enforcing which ones are used. Figure 6.1a shows that while only one logical core is in use by the application at any given time, the core being used changes over the life of the application. Sub-figure 6.1a shows that the Wetware processes are mostly kept on the same core, avoiding context switches and benefiting from data locallity.


(a) GOMAXPROCS=1

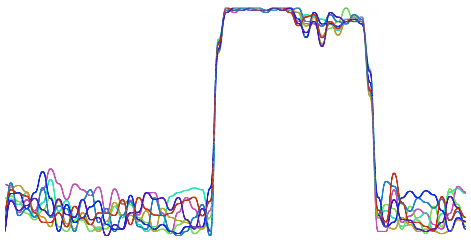

(b) GOMAXPROCS=8

Figure 6.1: CPU usage of an Executor running 8 Wetware processes

The executor will be benchmarked by running a CPU-intensive process (*Busy*), shown in listing 6.3, in several concurrent processes. It is a work-bound program that requires no synchronization, allowing us to test the executor with a minimal synchronization footprint from the WASM guests.

The experiment can be reproduced by executing `ww benchmark -procs M -iters N -size O -yield Q`, where M is the number of Wetware processes to spawn each iteration, N is the number of iterations, O is the total number of times a process will loop, and Q is the frequency with which each process will voluntarily yield execution.

A simplified version of the main benchmark loop is shown in listing 6.2, with some initialization, error handling, and time measuring and processing were left out for simplicity's sake. The first step is to set up a Wetware node and its executor on that very process. After that, the process will perform N iterations and on each of them it will spawn M processes that will run the *busy* program, simplified in listing 6.2.

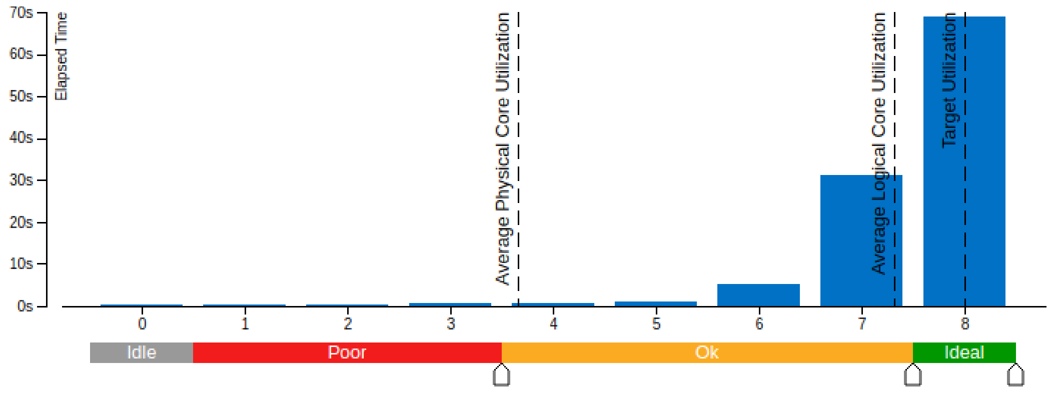
```
cid := executor.Cache.ExposedPut(busyWasm)
for i := int64(0); i < iters; i++ {
    var wg sync.WaitGroup
    for j := int64(0); j < procs; j++ {
        wg.Add(1)
        go func(k int64) {
            defer wg.Done()
            p, _ := executor.ExecCached(c.Context, cid, args)
            csp.Proc(p).Wait(c.Context)
        }(j)
    }
    wg.Wait()
}
```

Listing 6.2: Main host busy function

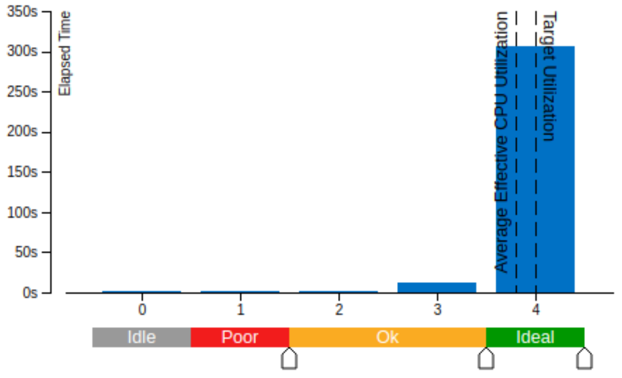
Busy will run its Wetware initialization, explained later down this chapter, and perform a loop with O iterations. If the current iteration is a multiple of Q, it will voluntarily yield. Yielding can be used to simulate a more realistic workload, occasionally breaking the tight loop Go's scheduler may struggle with.

```
func main() {
    ww.Bootstrap(context.Background())
    for i := int64(0); i < size; i++ {
        if yield != 0 && i%yield == 0 {
            runtime.Gosched()
        }
    }
}
```

Listing 6.3: Main guest busy function



(a) Scenario A



(b) Scenario B

Figure 6.2: Simultaneous utilization of logical CPU cores

Let's present two scenarios for the performance characterization of the application on a processor with 4 physical and 8 logical cores. Scenario A has all 8 virtual cores enabled. Scenario B has one logical core of each physical core enabled, and the other one disabled. Both scenarios run *busy* with the following parameters: $procs = 32$, $iters = 1$, $size = 10^9$, $yield = 0$. Figure 6.2 shows the CPU utilization results reported by Intel's VTune Profiler for both scenarios, with a 1kHz sampling rate. None of the scenarios achieve full CPU utilization, as there are other OS processes running and sharing the CPU. It can be observed, however, that scenario B is closer to ideal CPU utilization than scenario A, showing average physical core utilization values of 95.0% and 91.6% respectively. Furthermore, Scenario A spends significantly more time under-utilizing one of the cores compared to scenario B. This indicates effective utilization of every available core diminishes as the number of available cores increases.

On the other hand, table 6.1 shows metrics favoring scenario A. This scenario reported that 6.6% of inactive wait time had poor CPU utilization against the 21.1% of scenario B. This metric is the sum of two other metrics: Inactive Sync Wait Time and Preemption Wait Time. Inactive Sync Time refers to the time a thread has been inactive and excluded from execution by the operating system scheduler for synchronization, while Preemption Wait Time is caused by thread preemption. Both of the factors are reduced as the number of available cores increases, somewhat countering the poorer CPU utilization of figure 6.2. Both of them are also likely caused by having too many threads contending for the cores, also known as thread oversubscription [84]. The Go runtime utilizes 14 OS threads for scenario A and 10 threads for scenario B, as in these scenarios it uses *GOMAXPROCS* OS threads to run user code and 6 additional threads to run runtime code. Go doesn't provide any way of changing the number of runtime threads without delving into the language's source code, thus this number is a permanent factor the program will deal with.

	Scenario A		Scenario B	
Metric	Seconds	Normalized	Seconds	Normalized
Inactive Wait Time	2404.136	1.00	4426.744	1.00
IWT with Poor CPU Usage	158.673	0.0658	934.043	0.211
Inactive Sync Wait Time	86.310	0.0359	401.82	0.0908
Preemption Wait Time	72.363	0.0301	532.223	0.0907

Table 6.1: Benchmark thread wait time metrics

Table 6.2 shows a slightly more efficient usage of L1, L2 and L3 caches, as well as memory, in scenario A. "Memory Bound" refers to the fraction of the pipeline slots that might be stalled by load/store instructions throughout the execution of the program, while "Cache Bound" shows the percent of clock ticks spent retrieving data from the caches, including coherence penalties for shared data.

Pipeline Slots	Scenario A	Scenario B
Memory Bound	7.6%	8.0%
Cache Bound	16.2%	19.2%

Table 6.2: Memory and cache bound stalls

6.2.1 Scalability Analysis

A strong scalability analysis with the busy program and the following parameters shows interesting results: $procs = 32$, $iters = 1$, $size = 2 \cdot 10^9$, $yield = 0$. We performed the analysis with the aforementioned fixed-sized problem and two different core-count control methods:

- *GOMAXPROCS*: limit the number of cores through the Go runtime. Scheduling, networking, system calls and garbage collection do not respect this core limit. Shown as *gomaxprocs* in graph labels, and referred to as Go-level control in this section.
- *CPU hot-plugging*: disable logical cores through setting `/sys/devices/system/cpu/cpu<n>/online` to 0. Cores 1 to 4 are located in different physical cores, which they share with their respective counterpart from cores 5 to 8. It is labeled as *cpu hot-plug* and referred to as OS-level control throughout this section.

Both scenarios were run on an Intel Core i7-6700K with 4 physical and 8 logical cores running GNU/Linux with a minimal set of services: Bluetooth, window managers and other unnecessary background services were disabled. Configuring the application's garbage collector to run more or less frequently did not provide any apparent benefit after some testing and was therefore left as-is.

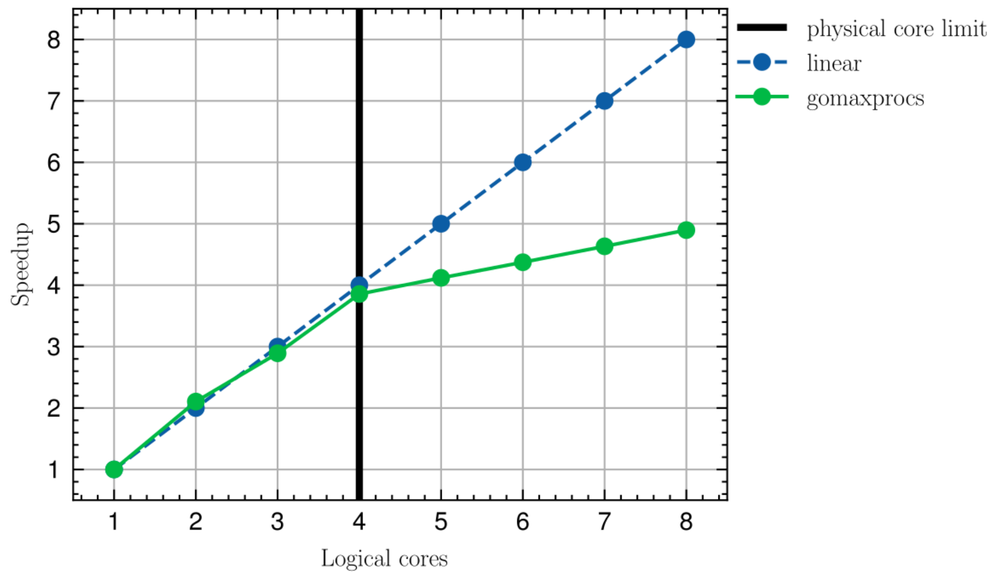


Figure 6.3: Strong scalability analysis through *GOMAXPROCS*

Figure 6.3 shows the speedup when controlling the number of cores through *GOMAXPROCS*. The speedup calculated as $speedup_n = \frac{t_1}{t_n}$ where t_1 is the execution time with one core and t_n is the execution time with n cores. The figure shows a linear speedup close to the ideal speedup up to the physical core limit, with a speedup degradation once that threshold is crossed. There are multiple potential causes:

- *Hyper-threading*: while hyper-threading provides performance improvements in this application, the improvements can be limited when compared to real parallelism.
- *Highly parallel workload*: the *busy* processes are work-bound and highly independent. They can be fully run in parallel and benefit more from additional physical cores than hyper-threading.
- *Runtime goroutines*: while user code is running exclusively in the number of specified cores, the runtime may run on other cores. The fewer cores the application uses though *GOMAXPROCS*, the less impact the runtime will have on its performance as it will be running on separate cores. The

more cores the application uses, the more likely user code is to conflict with runtime code.

- *OS footprint*: the operating system has a footprint that can minimally impact the application as more cores are taken by the application and need to be preempted to run OS tasks.

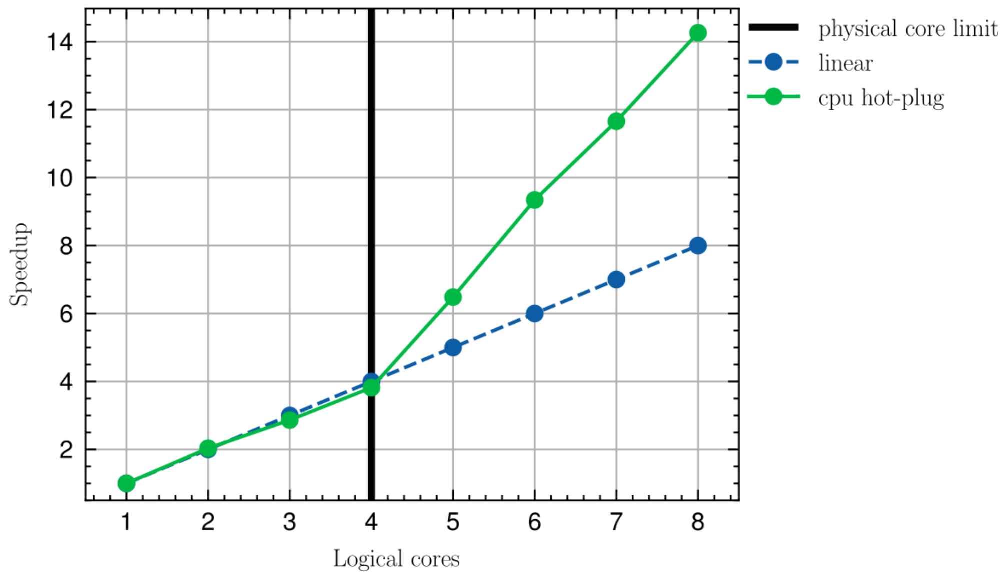


Figure 6.4: Strong scalability analysis through CPU hot-plugging

Figure 6.4 shows a reverse change in the speedup slope when compared to figure 6.3. Going over the physical core barrier actually results in super-linear speedup [85], as opposed to the previous speedup degradation. Figure 6.5 shows the total runtime for each one of the cases, a very helpful measure to understand why the super-linear speedup occurred. The single-core case execution takes almost thrice the runtime when the system is truly single-cored, when compared to the *GOMAXPROCS* method. This difference is most likely caused by:

- *Runtime and user code conflicts*: any network operation, asynchronous system call, scheduling decision, synchronization operation, or garbage collector run is executed on the same cores running the user code; which has the most impact with low core counts.

- *Concurrency penalty*: Wetware is a highly concurrent application that derives in higher performance penalties when there are not enough parallelization resources available.
- *OS footprint*: OS tasks conflict more often with our application core the lower the core count is.

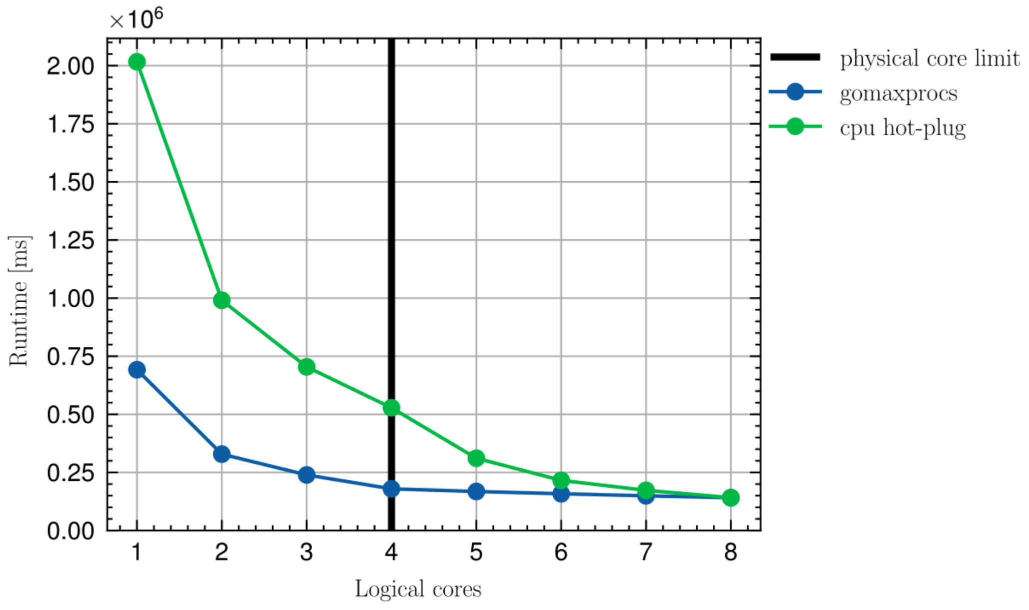


Figure 6.5: Runtime comparison of core control with OS-level CPU management vs GOMAXPROCS

Both core control methods have the same result when they allow the application to fully utilize all the available hardware.

6.2.2 Benchmark Against Native Programs

This section benchmarks Wetware processes against their Go (version go1.21.1 linux/amd64) and Python (version 3.11.5) counterparts. The go implementation is extremely similar to the original source of the Wetware busy implementation. It spawns *procs* goroutines, each of which loops for *size* iterations and performs the same busywork as in the Wetware application. The main goroutine waits for the rest to finish before ending the program, as seen in listing 6.4.

```go
package main

const size, procs = int64(2000000000), 32
var yield = int64(0)

func main() {
    wait := make(chan struct{}, procs)
    for i := 0; i < procs; i++ {
        go func() {
            for j := int64(0); j < size; j++ {
                if yield != 0 && j%yield == 0 {
                    j = j
                }
            }
            wait <- struct{}{}
        }()
    }
    for i := 0; i < procs; i++ {
        <-wait
    }
}
```

Listing 6.4: Benchmarked implementation of *busy* in Go

The Python implementation of listing 6.5, executed with the default Cpython interpreter, spawns processes using the *multiprocessing* library, and waits for them to finish their iterations. The variable *yield* has been changed by *y* due to *yield* being a reserved keyword in Python.

```python
import multiprocessing

def loop():
    j, y = 0, 0
    while j < 2_000_000_000:
        j += 1,
        if y != 0 and j % y == 0:
            j = j

procs = []
for i in range(32):
    t = multiprocessing.Process(target=loop)
    t.start()
    procs.append(t)
for t in procs:
    t.join()
```

Listing 6.5: Benchmarked implementation of *busy* in Python

Results show that the Wetware program performs somewhere in the middle between Python and Go. The single-core case places Wetware approximately an order of magnitude faster than Python, and an order of magnitude slower than Go. Nevertheless, the gap between Wetware and Go gets progressively smaller as more cores are added.

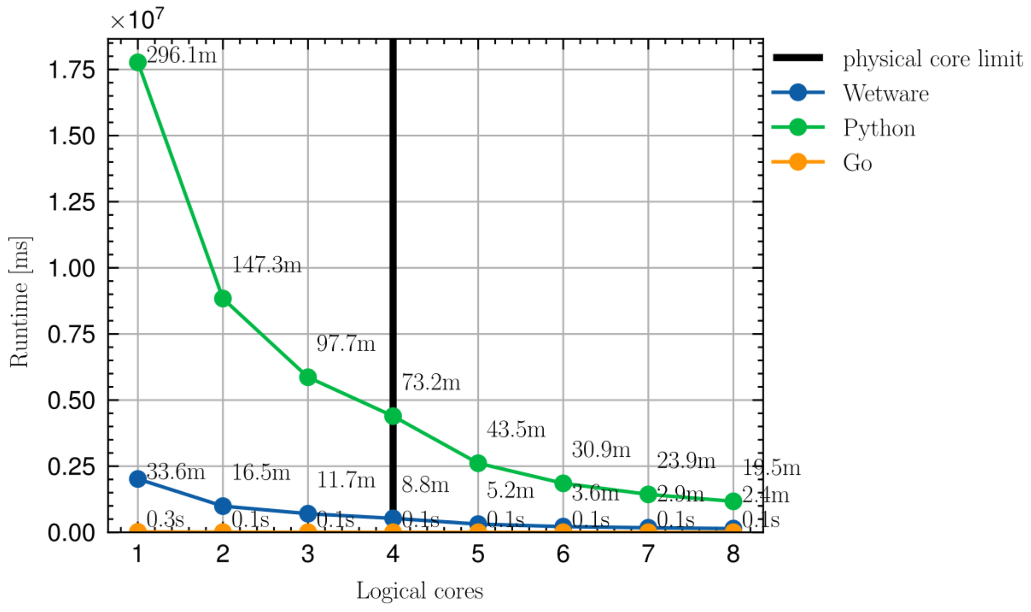


Figure 6.6: Runtime comparison for equivalent programs

6.2.3 Concurrency Characterization

Performing goroutine and block profiling provides helpful information about what use is being made of concurrency control mechanisms. Block profiles, unfortunately, do not include networking or asynchronous system call data. They do

Flat	**Flat%**	**Sum%**	**Cum**	**Cum%**	**Name**
11645	82.55%	82.55%	11645	82.55%	runtime.selectgo
1656	11.74%	94.29%	11645	11.74%	runtime.chanrecv1
515	3.65%	97.29%	515	3.65%	sync.(*Mutex).Lock
279	1.98%	99.92%	279	1.98%	runtime.chanrecv2
0	0.00%	99.92%	131	0.93%	net.(*Buffers).WriteTo
0	0.00%	99.92%	402	2.85%	io.ReadFull
0	0.00%	99.92%	402	2.85%	io.ReadAtLast
0	0.00%	99.92%	5287	38.19%	golang.org/x/sync/errgroup .(*Group).Go.func1
0	0.00%	99.92%	108	0.77%	github.com/wetware/pkg/ cluster.(*Router).advance
0	0.00%	99.92%	217	1.64%	github.com/wetware/ pkg/boot/ socket.(*Socket).tickloop

Table 6.3: Top concurrency contentions on Wetware server running a webcrawling application

nonetheless give a clear view of the difference of use between channels and mutexes. Tables 6.3 and 6.4 are taken directly from the results generated by *pprof*, and must be taken with a grain of salt. This is especially true when analyzing block profiles, as in this case, as it is by far the least documented profiling type of the tool [86].

Table 6.3 shows the number of contentious when a goroutine was left waiting for a resource before resuming its execution. There is an overwhelming predominance of channel contentions, amounting to a cumulative 97.94% of the contentions.

Table 6.4 shows similar results regarding the time spent waiting for resources by goroutines. This time, the cumulative delay caused by channels is 94.83%, proportionally slightly less than that of the rest of synchronization mechanisms.

We conclude that there is an acceptable concurrent resource utilization, making

Flat	**Flat%**	**Sum%**	**Cum**	**Cum%**	**Name**
3942.91s	72.84%	72.84%	3942.81s	72.84%	runtime.selectgo
1089.36s	20.12%	92.96%	1086.36s	20.12%	runtime.chanrecv1
209.48s	3.87%	96.83%	209.48s	3.87%	runtime.chanrecv2
171.51s	3.17%	100.00%	171.51s	3.17%	sync.(*WaitGroup).Wait
0	0.00%	100.00%	92.23s	1.70%	io.ReadFull
0	0.00%	100.00%	92.23s	1.70%	io.ReadAtLast
0	0.00%	100.00%	92.23s	3.17%	golang.org/x/sync/ er-rgroup .(*Group).Wait
0	0.00%	100.00%	171.51s	6.33%	golang.org/x/ sync/errgroup .(*Group).Go.func1
0	0.00%	100.00%	90s	1.66%	go.opencensus.io/stats/ view .(*worker).start
0	0.00%	100.00%	100.99s	1.87%	github.com/wetware/ ...(*Socket).heartbeat

Table 6.4: Top concurrency delys on Wetware server running a webcrawling application

use of lightweight concurrency controls via channels over more costly mechanisms such as mutexes. As closing remarks, figure 6.7 was generated from a goroutine profile and shows the major causes of goroutine parking. It shows the majority of parks were caused by select statements and read or write operations over channels, matching the results of the block profiling. These operations were often caused by RPCs or SPSC queue usages.

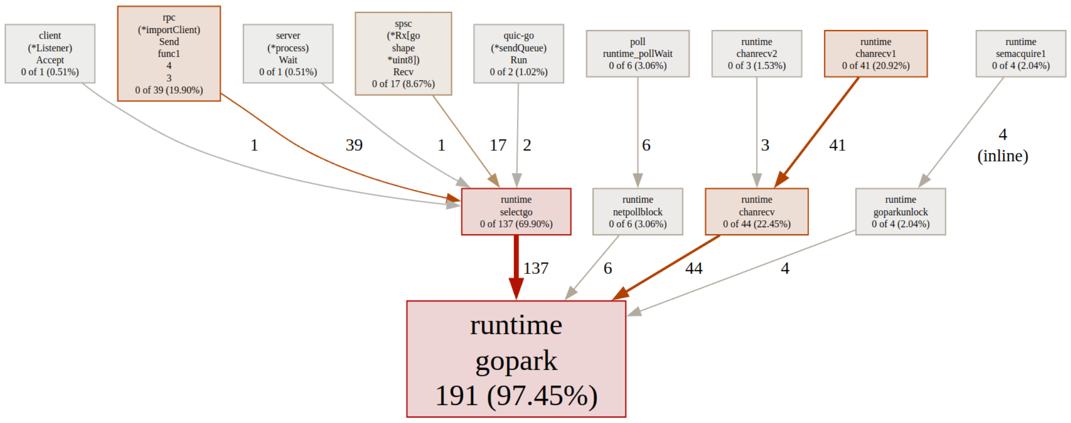

client (*Listener) Accept 0 of 1 (0.51%)
rpc (*importClient) Send func1 4 3 0 of 39 (19.90%)
server (*process) Wait 0 of 1 (0.51%)
spsc (*Rx[go shape *uint8]) Recv 0 of 17 (8.67%)
quic-go (*sendQueue) Run 0 of 2 (1.02%)
poll runtime_pollWait 0 of 6 (3.06%)
runtime chanrecv2 0 of 3 (1.53%)
runtime chanrecv1 0 of 41 (20.92%)
runtime semacquire1 0 of 4 (2.04%)
1
39
1
17
2
6
3
41
4 (inline)
runtime selectgo 0 of 137 (69.90%)
runtime netpollblock 0 of 6 (3.06%)
runtime chanrecv 0 of 44 (22.45%)
runtime goparkunlock 0 of 4 (2.04%)
137
6
44
4
runtime gopark 191 (97.45%)

Figure 6.7: Causes of goroutine parking

Chapter 7

Conclusions

In this thesis, we designed and developed a WebAssembly-based process execution and management tool for a P2P distributed systems middleware. We offered background into the technologies it is built upon, as well as some insight into how they fit together to form the executor. The background study proved vital for both design and implementation, allowing us to build a tool that takes advantage of the fortes of each of its technological pillars: process scheduling and concurrency are handled by Go, a language designed for concurrency. Isolation is built into WebAssembly. Inter-process communication is performed with Cap'n Proto, a high-performance RPC system. These components consolidate into a process execution technology stack with which to run, manage, communicate, and coordinate processes across a cluster.

We provided isolated WebAssembly processes the means of reaching outside their sandbox through object capabilities, and facilitated sharing capabilities among processes. Through benchmarking, profiling, and characterization we identified and overcame some performance bottlenecks as well as planned future performance-enhancing measures. We validated and tested the solution, along with demonstrating how to integrate new features into Wetware processes through Raft and HTTP capabilities.

We conclude that Wetware is a viable tool for building and deploying distributed, P2P applications with performance comparable to higher-level programming languages; all while providing, process management features, inter-process communication, and a suite of general-purpose distributed system tools. This project aims to provide insight, the first examples, and a path to follow for anyone who wishes to create P2P distributed applications that benefit from our WebAssembly and object capability-based approach.

This work contributed to the Wetware project [61], developed experimental features [62], produced a general-purpose Raft library with Cap'n Proto as a transport [64], as well as creating an application integrating them: the first public demonstration of Wetware running a distributed application [63]; All of the aforementioned contributions are publicly available in git repositories and are open to new contributions and experimentation.

7.1 Future Work

The Go Cap'n Proto library is working on implementing *third party hand-off* or 3PH. Once it is completed, a capability A that obtains a capability B from a third party C will attempt to establish a direct connection with B instead of proxying calls through C. This has substantial performance improvement potential, but will surely require some changes to how WASM host-guest asynchronous communication is handled. On the same note, the most relevant bottleneck for the executor when creating a process is managing the TCP port. The next short-term goals include this host-guest communication rework, which will likely require collaboration with Wazero and Go Cap'n Proto.

Developing the web crawler application showed that making capabilities accessible for processes in different executors spread throughout a cluster is more complex than it should be. Providing a new, improved approach by making the required changes, possibly synchronizing capability storages in a cluster, is paramount. As a side note, capability stores will also benefit from 3PH.

Lastly, there is a steep learning curve for some of the tools used in this thesis, perhaps even for Wetware. It is important to work on accessibility and user experience, to make using the executor as well as the middleware, simpler.

7.2 Personal Assessment

This project has not only been a great learning experience but also a chance to contribute to an open-source project being built with cutting-edge tools. Being part of a multi-disciplinary project that encourages collaboration, has introduced me to people and communities, as well as challenged me to build and study performant software without the low-level control provided by other programming languages more fit for HPC. All throughout the development of this thesis I've encountered subjects studied during the master's degree courses, from consensus algorithms for distributed systems to process scheduling and IPC; making the thesis both an overture and the finale of this journey.

Acronyms

3PH	Third Party Hand-Off
AIX	Advanced Interactive eXecutive
AMD	Advanced Micro Devices
API	Application Programming Interface
ARM	Advanced RISC Machine
BEAM	Bogdan's Erlang Abstract Machine
BE	Big Endian
BSD	Berkeley Software Distribution
CID	Content ID
CI	Continuous Integration
CLI	Command-Line Interface
CORBA	Common Object Request Broker Architecture
CSP	Communicating Sequential Processes
Capnp	Cap'n Proto
DFS	Depth First Search
ETL	Extract, Transform, Load
FD	File Descriptor

FFI	Foreign Function Interface
HPC	High Performance Computing
HTTP	Hypertext Transfer Protocol
IO	Input/Output
IPFS	Inter-Planetary File System
IWT	Inactive Wait Time
IaaS	Infrastructure as a Service
iOS	iPhone OS
IoT	Internet of Things
LE	Little Endian
LRQ	Local Run Queue
MIPS	Microprocessor without Interlocked Pipeline Stages
NAT	Network Address Translation
OS	Operating System
P2P	Peer-to-Peer
PC	Personal Computer
PID	Process ID
PPC	PowerPC
PaaS	Platform as a Service
PoC	Proof of Concept
QUIC	Quick UDP Internet Connections
QoL	Quality of Life
RISC	Reduced Instruction Set Computing
RPC	Remote Procedure Call

SPSC Single-Producer/Single-Consumer

TCP Transmission Control Protocol

TLS Transport Layer Security

UDP User Diagram Protocol

UML Unified Modeling Language

URL Uniform Resource Locator

VAT Virtual Address Table

WASIP1 WASI Preview 1

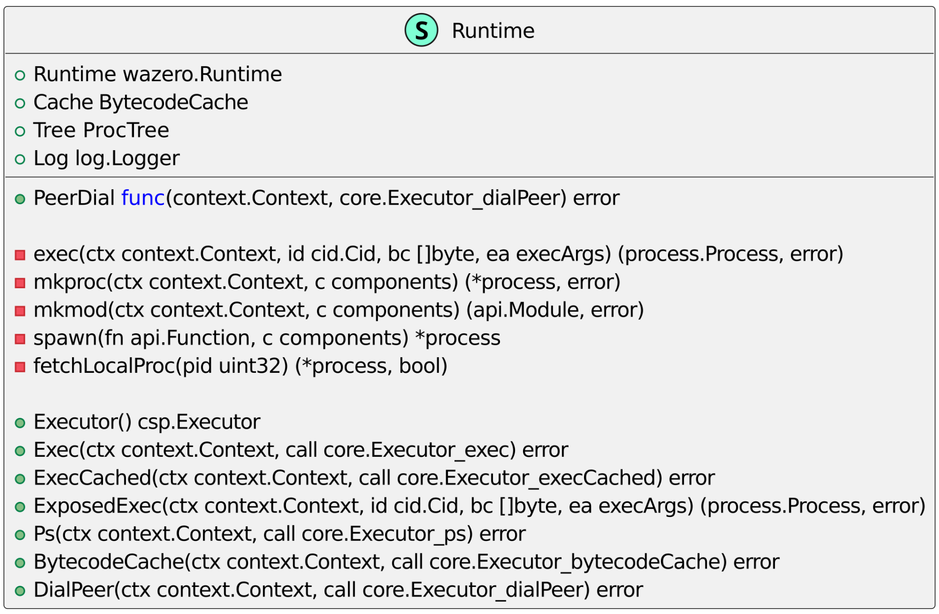
WASI WebAssembly System Interface

WASM WebAssembly

WW Wetware

Appendices

Appendix A: UML Component Diagrams

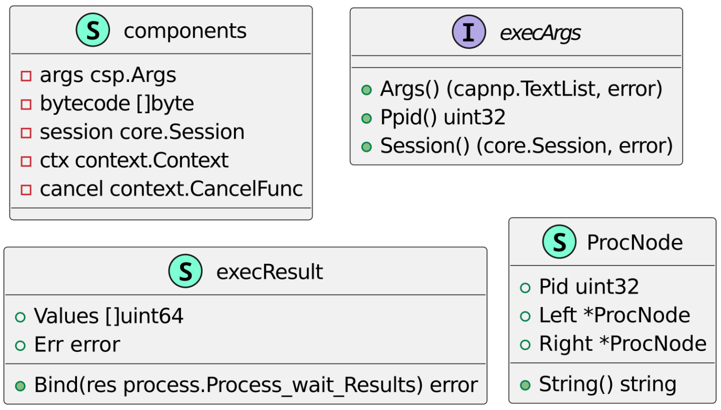


```
Runtime
○ Runtime wazero.Runtime
○ Cache BytecodeCache
○ Tree ProcTree
○ Log log.Logger
● PeerDial func(context.Context, core.Executor_dialPeer) error

■ exec(ctx context.Context, id cid.Cid, bc []byte, ea execArgs) (process.Process, error)
■ mkproc(ctx context.Context, c components) (*process, error)
■ mkmod(ctx context.Context, c components) (api.Module, error)
■ spawn(fn api.Function, c components) *process
■ fetchLocalProc(pid uint32) (*process, bool)

● Executor() csp.Executor
● Exec(ctx context.Context, call core.Executor_exec) error
● ExecCached(ctx context.Context, call core.Executor_execCached) error
● ExposedExec(ctx context.Context, id cid.Cid, bc []byte, ea execArgs) (process.Process, error)
● Ps(ctx context.Context, call core.Executor_ps) error
● BytecodeCache(ctx context.Context, call core.Executor_bytecodeCache) error
● DialPeer(ctx context.Context, call core.Executor_dialPeer) error
```

Figure A.1: UML diagram of the main executor package (Executor)



```
components
□ args csp.Args
□ bytecode []byte
□ session core.Session
□ ctx context.Context
□ cancel context.CancelFunc

execArgs
● Args() (capnp.TextList, error)
● Ppid() uint32
● Session() (core.Session, error)

execResult
○ Values []uint64
○ Err error
● Bind(res process.Process_wait_Results) error

ProcNode
○ Pid uint32
○ Left *ProcNode
○ Right *ProcNode
● String() string
```

Figure A.2: UML diagram of the main executor package (Miscellaneous)

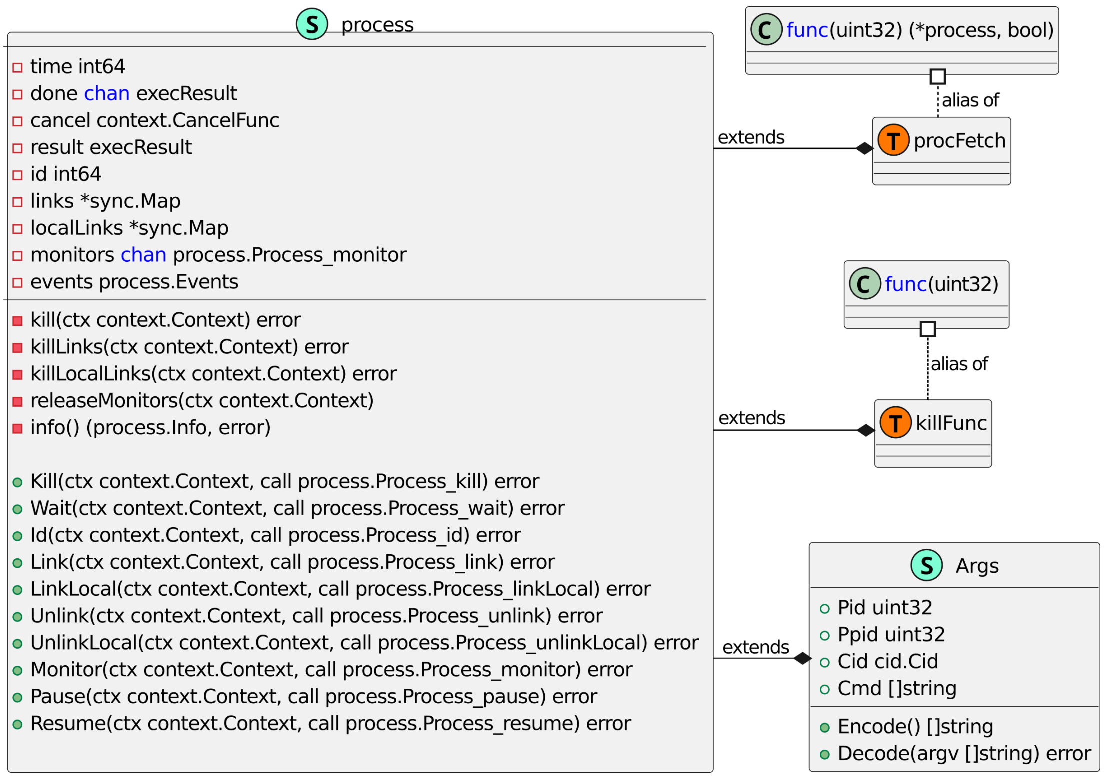

S process
time int64
done chan execResult
cancel context.CancelFunc
result execResult
id int64
links *sync.Map
localLinks *sync.Map
monitors chan process.Process_monitor
events process.Events
kill(ctx context.Context) error
killLinks(ctx context.Context) error
killLocalLinks(ctx context.Context) error
releaseMonitors(ctx context.Context)
info() (process.Info, error)
Kill(ctx context.Context, call process.Process_kill) error
Wait(ctx context.Context, call process.Process_wait) error
Id(ctx context.Context, call process.Process_id) error
Link(ctx context.Context, call process.Process_link) error
LinkLocal(ctx context.Context, call process.Process_linkLocal) error
Unlink(ctx context.Context, call process.Process_unlink) error
UnlinkLocal(ctx context.Context, call process.Process_unlinkLocal) error
Monitor(ctx context.Context, call process.Process_monitor) error
Pause(ctx context.Context, call process.Process_pause) error
Resume(ctx context.Context, call process.Process_resume) error
C func(uint32) (*process, bool)
alias of
T procFetch
extends
C func(uint32)
alias of
T killFunc
extends
S Args
Pid uint32
Ppid uint32
Cid cid.Cid
Cmd []string
Encode() []string
Decode(argv []string) error
extends

Figure A.3: UML diagram of the main executor package (Process)

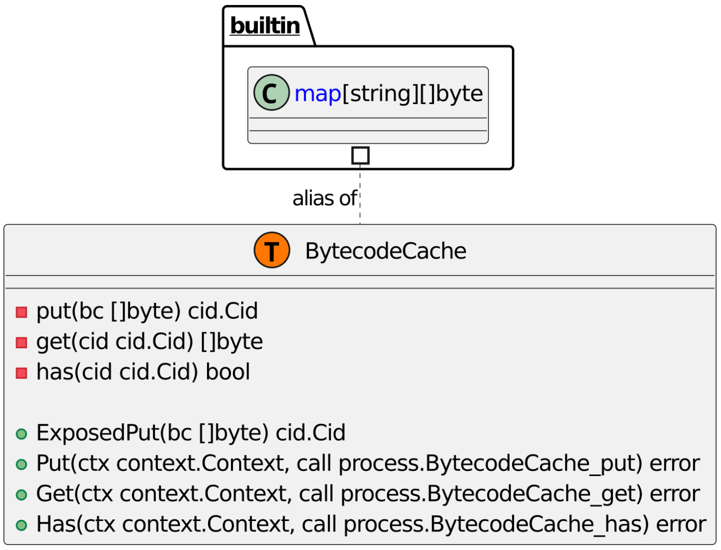

builtin
C map[string][]byte
alias of
T BytecodeCache
put(bc []byte) cid.Cid
get(cid cid.Cid) []byte
has(cid cid.Cid) bool
ExposedPut(bc []byte) cid.Cid
Put(ctx context.Context, call process.BytecodeCache_put) error
Get(ctx context.Context, call process.BytecodeCache_get) error
Has(ctx context.Context, call process.BytecodeCache_has) error

Figure A.4: UML diagram of the main executor package (Bytecode Cache)

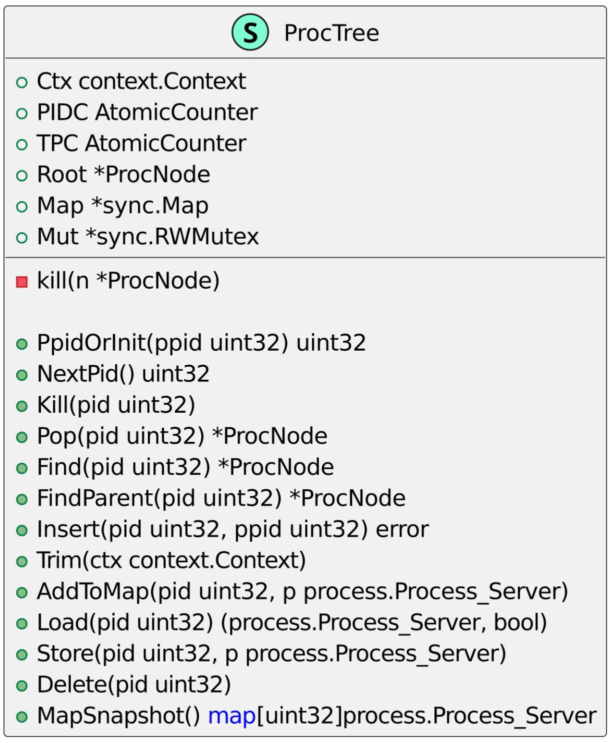

S ProcTree
Ctx context.Context
PIDC AtomicCounter
TPC AtomicCounter
Root *ProcNode
Map *sync.Map
Mut *sync.RWMutex
kill(n *ProcNode)
PpidOrInit(ppid uint32) uint32
NextPid() uint32
Kill(pid uint32)
Pop(pid uint32) *ProcNode
Find(pid uint32) *ProcNode
FindParent(pid uint32) *ProcNode
Insert(pid uint32, ppid uint32) error
Trim(ctx context.Context)
AddToMap(pid uint32, p process.Process_Server)
Load(pid uint32) (process.Process_Server, bool)
Store(pid uint32, p process.Process_Server)
Delete(pid uint32)
MapSnapshot() map[uint32]process.Process_Server

Figure A.5: UML diagram of the main executor package (Process Tree)

Bibliography

[1] Kenton Varda. “Cap’n Proto”. https://capnproto.org/. [Online; accessed 2023-10-10]. viii, 16

[2] Go. “Scheduling Structures”. https://go.dev/src/runtime/HACKING. [Online; accessed 2023-10-10]. x, 44, 47

[3] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC’14, page 305–320, USA, 2014. USENIX Association. 1, 13

[4] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and David Maier. Blazes: Coordination analysis for distributed programs, 2013. 1

[5] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for {Fine-Grained} resource sharing in the data center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011. 1

[6] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, pages 179–196. 2019. 1

[7] Abdeldjalil Ledmi, Hakim Bendjenna, and Sofiane Mounine Hemam. Fault tolerance in distributed systems: A survey. In *2018 3rd International Conference on Pattern Analysis and Intelligent Systems (PAIS)*, pages 1–5, 2018. 1

[8] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'cause i'm strong enough: Reasoning about consistency choices in distributed systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 371–384, 2016. 1

[9] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010. 1

[10] Wal Van Lierop. "Step Up Or Break Up: The Challenge For Big Tech". https://www.linkedin.com/pulse/step-up-break-challenge-big-tech-wal-van-lierop/?articleId=6722195716942503936. [Online; accessed 2023-10-10]. 1

[11] Cecilia Rikap and Bengt Åke Lundvall. Big tech, knowledge predation and the implications for development. *Innovation and Development*, 12(3):389–416, 2022. 1

[12] Alex Murray, Dennie Kim, and Jordan Combs. The promise of a decentralized internet: What is web3 and how can firms prepare? *Business Horizons*, 66(2):191–202, 2023. 2

[13] Louis Thibault. "Wetware". https://hackmd.io/@lthibault/SJzOIt9k3. [Online; accessed 2023-10-10]. 2, 6, 11

[14] "Syncthing" . https://syncthing.net/. [Online; accessed 2023-10-10]. 2

[15] "Anytype" . https://anytype.io/. [Online; accessed 2023-10-10]. 2

[16] Meghan Rimol DeLisi and Catherine Howley. Gartner says worldwide iaas public cloud services revenue grew 302022, exceeding $100 billion for the first time. *Gartner*. 6

[17] Aratz Manterola-Lasa, Diego Casado-Mansilla, and Diego López-de Ipiña. Unsserv: unstructured peer-to-peer library for deploying services in smart environments. In *2020 5th International Conference on Smart and Sustainable Technologies (SpliTech)*, pages 1–6, 2020. 8

[18] Richard M Adler. Distributed coordination models for client/server computing. *Computer*, 28(4):14–22, 1995. 9

[19] Maha Alsayasneh. *On the identification of performance bottlenecks in multi-tier distributed systems*. PhD thesis, Université Grenoble Alpes [2020-....], 2020. 9

[20] Erik Daniel and Florian Tschorsch. Ipfs and friends: A qualitative comparison of next generation peer-to-peer data networks. *IEEE Communications Surveys & Tutorials*, 24(1):31–52, 2022. 9

[21] Protocol Labs. “InterPlanetary File System”. https://ipfs.io/. [Online; accessed 2023-10-10]. 11

[22] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems 16, 2 (May 1998), 133-169. Also appeared as SRC Research Report 49. This paper was first submitted in 1990, setting a personal record for publication delay that has since been broken by [60].*, May 1998. ACM SIGOPS Hall of Fame Award in 2012. 13

[23] Kevin Babitz. “Syncthing” . https://towardsdatascience.com/the-strange-story-of-the-paxos-algorithm-52a9f3f53ae0. [Online; accessed 2023-10-10]. 13

[24] etcd. “Raft implementation in Go” . https://github.com/etcd-io/raft. [Online; accessed 2023-10-10]. 13, 83, 88

[25] Christian Deyerl and Tobias Distler. In search of a scalable raft-based replication architecture. In *Proceedings of the 6th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC ’19, New York, NY, USA, 2019. Association for Computing Machinery. 14

[26] “gRPC”. https://grpc.io/. [Online; accessed 2023-10-10]. 15

[27] “Protocol Buffers”. https://protobuf.dev/. [Online; accessed 2023-10-10]. 15

[28] Mark S. Miller. "Promise Pipelining". http://www.erights.org/elib/distrib/pipeline.html. [Online; accessed 2023-10-10]. 16

[29] Juan Cruz Viotti and Mital Kinderkhedia. A Survey of JSON-compatible Binary Serialization Specifications. *arXiv e-prints*, page arXiv:2201.02089, January 2022. 17

[30] Kenton Varda. "Discussing Cap'n Proto Backing". https://news.ycombinator.com/item?id=25587781. [Online; accessed 2023-10-10]. 17

[31] Juan Cruz Viotti and Mital Kinderkhedia. A Benchmark of JSON-compatible Binary Serialization Specifications. *arXiv e-prints*, September 2022. 17

[32] Mark S. Miller. "CapTP: The Four Tables". http://www.erights.org/elib/distrib/captp/4tables.html. [Online; accessed 2023-10-10]. 17

[33] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. *SIGPLAN Not.*, 52(6):185–200, jun 2017. 21

[34] Harsh Joshi. Analysis of web assembly technology in cloud and backend. *International Research Journal of Modernization in Engineering Technology and Science*, 2022. 21

[35] Wenwen Wang. How far we've come – a characterization study of standalone webassembly runtimes. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*, pages 228–241, 2022. 21

[36] Benedikt Spies and Markus Mock. An evaluation of webassembly in non-web environments. In *2021 XLVII Latin American Computing Conference (CLEI)*, pages 1–10, 2021. 21

[37] "WebGPU in Wasm via Emscripten". https://github.com/juj/wasm_webgpu#webgpu-in-wasm-via-emscripten-or-dawn. [Online; accessed 2023-10-10]. 22

[38] Daniel Lehmann, Johannes Kinder, and Michael Pradel. Everything old is new again: Binary security of WebAssembly. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 217–234. USENIX Association, August 2020. 23

[39] Adrian Cole. "Introducing wazero from Tetrate". https://tetrate.io/blog/introducing-wazero-from-tetrate/. [Online; accessed 2023-10-10]. 24

[40] Dave Cheney. "cgo is not Go". https://dave.cheney.net/2016/01/18/cgo-is-not-go. [Online; accessed 2023-10-10]. 24

[41] "Wazero - Compiler engine". https://github.com/tetratelabs/wazero/blob/main/internal/engine/compiler/RATIONALE.md#compiler-engine. [Online; accessed 2023-10-10]. 25

[42] Stealth Rocket. "WASI-Go". https://github.com/stealthrocket/wasi-go. [Online; accessed 2023-10-10]. 25

[43] Frank Denis. "Performance of WebAssembly runtimes in 2023". https://00f.net/2023/01/04/webassembly-benchmark-2023/. [Online; accessed 2023-10-10]. 25

[44] Christine Spang. "Subprocess to FFI: Memory, Performance, and Why You Shouldn't Shell Out". https://github.com/spang/subprocess-to-ffi. [Online; accessed 2023-10-10]. 26

[45] Adrian Cole. "WebAssembly for the Backend ". https://speakerdeck.com/adriancole/webassembly-for-the-backend-gophercon-israel-2023. [Online; accessed 2023-10-10]. 26

[46] Object Management Group. "CORBA". https://www.corba.org/. [Online; accessed 2023-10-10]. 26

[47] S. Vinoski. Corba: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 35(2):46–55, 1997. 26

[48] David P Anderson. Boinc: A system for public-resource computing and storage. In *Fifth IEEE/ACM international workshop on grid computing*, pages 4–10. IEEE, 2004. 26

[49] Yoichi Hirai. Defining the ethereum virtual machine for interactive theorem provers. In *Financial Cryptography and Data Security: FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers 21*, pages 520–535. Springer, 2017. 27

[50] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. Sledge: A serverless-first, light-weight wasm runtime for the edge. In *Proceedings of the 21st International Middleware Conference*, Middleware '20, page 265–279, New York, NY, USA, 2020. Association for Computing Machinery. 27

[51] Suborbital. "Sat". `https://github.com/suborbital/sat`. [Online; accessed 2023-10-10]. 27

[52] Suborbital. "E2 Core". `https://github.com/suborbital/e2core/`. [Online; accessed 2023-10-10]. 27

[53] "WaPC". `https://wapc.io/`. [Online; accessed 2023-10-10]. 27

[54] Erlang. "Documentation: Processes". `https://www.erlang.org/doc/reference_manual/processes.html`. [Online; accessed 2023-10-10]. 27

[55] Erlang. "Documentation: link". `https://www.erlang.org/doc/man/erlang#link-1`. [Online; accessed 2023-10-10]. 27, 70

[56] Erlang. "Documentation: Distributed Erlang System". `https://www.erlang.org/doc/reference_manual/distributed.html#distributed-erlang-system`. [Online; accessed 2023-10-10]. 27

[57] Kenton Varda. "Re: [capnproto] capt'n proto schema for webassembly". `https://wapc.io/`. [Online; accessed 2023-10-10]. 28

[58] Andrew Dona-Couch. “rust-wasm-capnproto-example”. https://github.com/couchand/rust-wasm-capnproto-example. [Online; accessed 2023-10-10]. 28

[59] Wazero Slack Channel. “In response to process scheduling question” . https://gophers.slack.com/archives/C040AKTNTE0/p1694376620726659?thread_ts=1694362325.275289&cid=C040AKTNTE0. [Online; accessed 2023-10-10]. 35, 36, 67

[60] “Matrix - Go Cap’n Proto”. https://matrix.to/#/#go-capnp:matrix.org. [Online; accessed 2023-10-10]. 35, 36

[61] “Wetware”. https://github.com/wetware/pkg. [Online; accessed 2023-10-10]. 36, 111

[62] “Wetware Fork”. https://github.com/mikelsr/pkg. [Online; accessed 2023-10-10]. 36, 94, 111

[63] “Wetware Web Crawler”. https://github.com/mikelsr/ww-webcrawler. [Online; accessed 2023-10-10]. 36, 83, 85, 86, 91, 95, 111

[64] “raft-capnp”. https://github.com/mikelsr/raft-capnp. [Online; accessed 2023-10-10]. 36, 83, 87, 111

[65] “Wetware Raft Cap’n Proto Example”. https://github.com/mikelsr/ww-raft-example. [Online; accessed 2023-10-10]. 36

[66] “wzprof”. https://github.com/stealthrocket/wzprof. [Online; accessed 2023-10-10]. 38

[67] “Matplotlib”. https://matplotlib.org/. [Online; accessed 2023-10-10]. 41

[68] John D. Garrett. garrettj403/SciencePlots. September 2021. 41

[69] William Kennedyo. “Scheduling In Go : Part II - Go Scheduler”. https://www.ardanlabs.com/blog/2018/08/scheduling-in-go-part2.html. [Online; accessed 2023-10-10]. 45

[70] "epoll(7) — Linux manual page". https://man7.org/linux/man-pages/man7/epoll.7.html. [Online; accessed 2023-10-10]. 46

[71] Ralph Caraveo. "The new kid in town — Go's sync.Map". https://medium.com/@deckarep/the-new-kid-in-town-gos-sync-map-de24a6bf7c2c. [Online; accessed 2023-10-10]. 48

[72] Go. "WASI support in Go"". https://go.dev/blog/wasi. [Online; accessed 2023-10-10]. 55

[73] WebAssembly. "WASI Sockets Proposal"". https://github.com/WebAssembly/wasi-sockets#asynchronous-apis. [Online; accessed 2023-10-10]. 56

[74] Chris O'Hara et al. "Go runtime: implement wasip1 netpoll". https://go-review.googlesource.com/c/go/+/493357. [Online; accessed 2023-10-10]. 56

[75] Haehyun Cho, Jinbum Park, Joonwon Kang, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. Exploiting uses of uninitialized stack variables in linux kernels to leak kernel pointers. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020. 59

[76] Go Contributors. "The Go Programming Language - Git Repository" . https://github.com/golang/go/. [Online; accessed 2023-10-10]. 60

[77] codefromthecrypt. "Wazero wasi: fix nonblocking sockets on *NIX". https://github.com/tetratelabs/wazero/pull/1503. [Online; accessed 2023-10-10]. 60

[78] Erlang. "Documentation: monitor". https://www.erlang.org/doc/man/erlang#monitor-2. [Online; accessed 2023-10-10]. 74

[79] "Wazero Conformance". https://github.com/tetratelabs/wazero#conformance. [Online; accessed 2023-10-10]. 76

[80] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, 1984. 81

[81] Alfonso De la Rocha, David Dias, and Yiannis Psaras. Accelerating content routing with bitswap: A multi-path file transfer protocol in ipfs and filecoin. *San Francisco, CA, USA*, page 11, 2021. 82

[82] Alexandre Beslic. "proton". `https://github.com/abronan/proton`. [Online; accessed 2023-10-10]. 87

[83] "HTTP Dummy Server". `https://github.com/mikelsr/httpdummy`. [Online; accessed 2023-10-10]. 95

[84] Denis Bakhvalov. *Performance Analysis and Tuning on Modern CPUs*. 2020. 100

[85] Sasko Ristov, Radu Prodan, Marjan Gusev, and Karolj Skala. Superlinear speedup in hpc systems: Why and when? In *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 889–898, 2016. 103

[86] "doc: runtime/pprof: block profiler documentation needs some love #14689 ". `https://github.com/golang/go/issues/14689`. [Online; accessed 2023-10-10]. 107