CCA - COMPETENCE CENTRE
HTL Anichstraße

Diploma Thesis

Planning and MVP implementation of the Mize data engine

Handed in by

Sebastian Moser

Handed in at

Higher Technical College - and Experimental Institute Anichstraße

Elektronik und Technische Informatik

Supervisor

Oberprantacher-Zimmermann Stefan, Ing. Dipl.-Päd. BEd.

Innsbruck, March 2025

Delivery note: Supervisor:

Date:

Kurzfassung /Abstract

0.1 Abstract

This thesis is about the planning of the data engine called Mize, which is part of a larger software platform with the same name. In the beginning already existing data engine systems with similar capabilities are examined, which leads to the conclusion, that nothing that fully matches the requirements of the Mize data engine exists. The main goals of mize are, to make data accessible as if it were stored locally on the device, even if it is not and give this access through one simple and universal API, like Filesystems did, back when multiple devices per user were not a thing yet.

To achieve this mize has a struct called an Instance, through which all data can be accessed and also modified. Each place data is needed, there is an Instance. They all can talk to one another in a Peer-to-Peer way with one Instance being the owner of every peace of data, to be able to correctly coordinate multiple changes to the data from multiple sources in a simple way. This instance is implemented with the programming language Rust and compiled in different ways, to form libraries, that can be used by any programming language and run on any computer system. For that the packaging part of mize also makes up a part of this thesis.

In addition to planning the mize data engine, a MVP of it is also implemented, to show that the core planned concepts can work.

0.2 Kurzfassung

Die vorliegende Diplomarbeit beschäftigt sich mit der Planung von der Daten Engine mit dem Namen Mize und darum herum einer Software Plattform mit dem gleichen Namen. Am Anfang werden Daten Engines mit ähnlichen Fähigkeiten untersucht, was zum Schluss führt, dass es nichts gibt, was komplett mit den Anforderungen der Mize Daten Engine übereinstimmt. Die Hauptziele von Mize sind Daten abrufbar zu machen, als ob sie lokal auf dem jeweiligen Gerät sind, auch wenn sie das nicht sind und das über eine einfache und universelle API, so wie Filesysteme das waren zu Zeiten, wo es mehrere Geräte pro Benutzer noch nicht gegeben hat.

Um das zu erreichen, hat Mize eine Struktur mit dem Namen Instanz, durch welche alle Daten abrufbar und modifizierbar sind. Überall wo Daten gebraucht werden, ist eine solche Instanz. All diese Instanzen können mit dem Peer-to-Peer Prinzip miteinander reden, wobei eine Instanz immer der Inhaber von einem Datenstück ist. Dies macht es möglich, eine korrekte Koordinierung von mehreren Änderungen der Daten von mehreren Quellen einfach zu gestalten. Diese Instanz ist mit der Programmiersprache Rust implementiert und kompiliert in unterschiedlichen Wegen um Software Bibliotheken zu erhalten, welche von jeder Programmiersprache

verwendet werden können und auf jedem Computer System ausgeführt werden können. Daher macht der Packetierungsteil von Mize auch einen Teil dieser Arbeit aus.

Zusätzlich zur Planung der Mize Daten Engine wurde auch ein MVP davon implementiert, um zu zeigen, dass die Kernkonzepte funktionieren können.

0.3 Project outcome

All core concepts of the Mize data engine are planned, thought out and documented in this thesis. The MVP implementation can make data accessible as if it were local across Instances on the same Linux system connected through Unix Sockets. The connection into the webview of the Mize Explorer is implemented, but not fully working yet. The connection using Websockets is not yet implemented. With the invocation of one command Mize and all it's modules can be compiled for Linux x86_64, Linux aarch64, WebAssembly for running in browsers and Windows x86_64 and then automatically uploaded to a public HTTP serer.

The MVP implementation can be found on Github: [14], with the other parts of the Mize platform being across the repositories: [15], [33], [16], [18], [7] and [17]. The source code of this documentation can be found at [8] and the self built tool, which was used to build this document is [10].

Declaration of the independence of the work

STATUTORY DECLARATION

I declare under penalty of law that I have written this work independently and without outside assistance, that I have not used any sources or resources other than those indicated, and that I have identified all passages taken verbatim and in their content from the sources used. My work may be made publicly available unless a confidentiality notice is present.

Location, Date Sebastian Moser

Contents

1 Introduction

One of the most fundamental aspects of creating software application is managing Data. In modern times, when people no longer have only one computer at a fixed place at home, but many and mobile ones (smartphones, smartwatches, smart cars, E-Bikes, all the smarthome appliances, ...), it is also desirable that this data is available on all device of a person. All smart devices have some sort of connectivity hardware built into them, which could be Bluetooth, WIFI, LoRa, Ethernet and more.

1.1 The current Data management Situation

1.1.1 The example of a small life enhancement

Let's assume for example you have two unused function keys on your keyboard and would like to use those to dimm your smart desk lamp. Looking at what the hardware is capable of (theoretically), this would work as follows: The desktop computer could, whenever one of those keys is pressed, send a Frame with the new brightness value through it's Ethernet connection to the Router, which would forward it to the smart lamp via wifi and the lamp then changes the voltage going to the LEDs acordingly. In practise, when we take the software running on the devices into account, implementing such a small life enhancement, would require way to much work, self written software and programming skill to make it work.

We would have to install software like AutoHotkey [4], that can react to any keypresses and run some code, that we would have to write ourselves, to update the brightnes value, by sending for example a http request to the IP of the lamp. We could also be unlucky and have a lamp, which only talks to the cloud of the manufacturer and has no http server through wich it allows changes of it's brightnes value. This would require finding documentation of the APIs of the cloud or even reverse engineering them, to be able to tell the cloud, to change the brightnes value. The cloud could also add delays of up to multiple seconds.

All this trouble exists because: Firstly the operating system on the desktop computer does not have functionality to map two keys to increment and decrement some

data value (in our case the brightness value of the lamp). Secondly there is no such magic "data value", that can be modified on one device (the desktop computer) and have an effect on some other (the lamp).

1.1.2 Lack of general data management systems or standards

A long time ago, when Unix and the command line was how computers were used, there was the filesystem as the only data management API. Every application on the system just opened, read and wrote files, that mostly just contained readable text. One part of the Unix philosophy was "Everything is a file", so even if you wanted to find out for example your IP address, you would read a file in a special filesystem. This allowed for a great flexibility for the user as to what programm they would use to view or modify the content of a file. And programms can be combined in the so called Unix pipes ([11]), which allowed for endless possibilities of what to do with the data of one or many files.

Nowadays, the data stored by programms is more complex than just text and also every Person has many computer systems in there life, makig it nececary to have data available across devices. Those added requirements, lead to every application implementing it's data management in their own way. The problem with this is, that integrating with other appliations becomes a feature, that the developers need to implement and not a thing that's possible from the beginning. Many modern apps proudly show on their websites all the other applications, that they can integrate with, but those are never all apps, which deal with the same kind of data. For example, in our HTL you have four sources, where homework items could be specified by a teacher, those being Microsoft Teams, Webuntis, Moodle and Schoolfox and it is not possible to show all of your homework in one single list. No you need to visit all three websites/apps and check manually.

Anoter example from IoT, where this problem really limits the possibilities of what you can do with IoT devices. My family has solar panels, an inverter, a battery, a heat pump and a electric car charging box installed at our home. Both the inverter and the battery connect to our WIFI network, that they use to talk to their respective clouds, allowing us to view data about them from anywhere we have an Internet connection. The problem however is that this situation does not allow, or makes it extreamly difficult to get this data in realtime into a local Grafana instance, that would be my tool of choiche to show and analyze the graphs of our Energy system. There is also no way, to tell the car charger to charge the car, whenever the inverter would be selling power to the grid. The car charger uses bluetooth to talk to the app that came with it and has the option to connect it to their cloud if we wanted to. The car even has LTE hardware to always be able to talk to it's cloud and for our heat pump the company installed an extra LTE router in our home, so that the heat

pump could talk to, you guessed it, their cloud. The data of our energy system is spread across five highly incompatible data management systems and there is no single place to view and modify all important data of all components. In my view this is a ridiculous situation and my family and everyone else who I tell about this agrees.

Every application having it's own data management system allows developers to implement data management in a way, that works best for the use case of their application, but also many applications fail badly at a proper data management implementation. For example our Inverter at least in the past, sent it's data into the cloud encrypted with AES128, but the passphrase was made of all ones ([25]) and also it takes five minutes for the newest data to be shown on the cloud's website.

1.1.3 A look at existing Data Management Systems

ICloud

If you have all Apple devices and just turn on ICloud, you already get many things, that the Mize data engine would enable. A picture you take on your IPhone is immediately abailable on your Mac, you mirror the screen of your IPhone to your Mac and you see the battery levels of all your devices on every device. The problem with the Apple Ecosystem is however, that it only works with Apple devices and data can only be stored on Apple's servers, Integrations with all other systems you use personally or for work is also non existent. Also Apples ecosystem is not particularly good at allowing creative combinations of tools by users, like Unix pipes do for example.

InstantDB

InstantDB is an awesome system, to manage data for a modern application. It has nice APIs to access and update data, does so in realtime and can deal with clients going offline for some time. All data is stored on a server, that can also be self hosted. The Problem with InstantDB is, that it is made for developers, to create apps with and not users wanting to store theeir personal data using it. Also it only has a client to be used with JavaScript, making it not usable in embedded scenarios on micro controllers.

Java Hibernate

Hibernate is a data management library for Java. It is only for Java and therefore already can't fulfil the goals of the Mize data engine.

DotNET Entity Framework

The DotNET Entity Framework is the same as Hibernate, but for DotNET languages like `C#` or `F#`.

1.2 Data Management with the Mize data engine

The Mize data engine tries to be a general data management system, just like the Unix filesystem was a long time ago, but being able to deal with modern requirements like syncronisation between many devices and a more complex structure of data. It should be usable by application developers as a library for data management, but at the same time allowing users to then interact with this data outside of this one application, for example to view it in a file explorer like app, or use some data from one application in some other one. Mize should also be flexible enought to be used in any data management scenario, from large data centers, to small micro controllers in IoT applications. Changes in data will be procesed in real time and just like InstantDB Mize will handle the merges of data, when one Instance goes offline for some time.

1.3 Plans for more than just data management

The Mize data engine is one part of the software platform also called Mize. Those two names are the same, because data management is the most important part of this platform.

1.3.1 What is a Software Platform

A software Platform is a technology, that allows developers to create, package, deploy, distribute and run a software application. Examples of software platforms are operating systems like Windows, Android or any Linux distribution, cloud environments like AWS or Azure and also development frameworks like Electron or SDL.

1.3.2 Other Components of the Mize Platform

To make up a complete software platform Mize as in addition to the data engine some other components.

Gui Engine

The gui engine's job is to display a set of Items to the screen and allow uses to interact with them. It is part of the mme repository ([16]).

CLI Engine

The CLI Engine is the user interface to interact with items from a command line. It has it's own repository called "mme-cli" ([17]).

User Commands Engine

Every modern gui application has some sort of command palete, where users can search through all availavle commands, this avoids users needing to search through menue trees to find the right command. The user command engine is basically that and in the repository called "comandr" ([7]).

Init Engine

A large portion of code that is written for an application is code, that runns at startup and brings the memory into the right state for the application to begin operation. This code I call "setup code" or "init code" and is the reason, why large applications often take so extreamly long to start up. The Init Engine will try to optimize the amount of code that needs to run on startup, therefore decreasing the time, untill an application becomes usable. It will be part of the Victorinix repository [33].

Option Engine

The Option Engine is about managing configuration, using a standrad way to define configuration options and their documentation. The same kind of options will also be used to describe detauls about computer systems, like explained in 3.2.1. This will also be part of the Victorinix repository ([33]). It should also eventually be possible to describe massive cloud deployments as a set of such options and therefore have deployments defined declaratively, instead of having to set them up imperatively.

Run any software anywhere

This is the end goal of the Victorinix repository. It will try to use options from the Option Engine to figure out, how to get from the current computer system's environment to an environment, which can run the programm in question. This could include containerisation, virtualisation or even emulation.

1.3.3 Mize OS

Those components of the Mize platform are quite similar to the main components of an Operating System Userspace (which does not include ther kernel). So with enough work, there is a possibility, that Mize will turn into a Operating System Platform, that will mainly use the Linux or BSD Kernel, but also be able to be installed onto a kernel from Windows, or other future kernels.

1.4 Diploma thesis goals

The goal of this diploma thesis is the planning of the main concepts of the Mize data engine, which are described in detail in the chapter 3.1. Some of the higher level concepts like the "Mize tag system", the "Mize link system" and "Mounting Items" got some thought, but are not core concepts of the data engine. Also not covered is the the "Mize Explorer" mme for short and the other parts of the larger Mize platform. All of those would break the boundaries of length for a diploma thesis of a single person. However a MVP of the data engine, that shows that those core concepts work in practise is a goal.

2 History of Mize

The Idea of Mize was everything but immediate, it evolved over about 4 years. First thoughts about something like that came to mind shortly after learning to program with python in first class of the HTL. This history is documented in this chapter.

2.1 Coding Wiki

When learning to code I ran into the problem, that you forget all the names of functions, modules, concepts and even the names of keywords. And also what they do. So to help with that I wanted to make an application, where I could put all those things and be able to look them up, when needed by searching. All those items would have a type, for example keyword, python-internal-function, python-module, ... Later I realized, that also all "documentation items" from other languages and frameworks could be added to such a searchable system and work on a "personal db" was also started, where personal notes, quotes and anything else could be stored and would have a type of what it is, like note, quote, date and so on.

The implementation of this predecessor of Mize, never become close to being usable. I gave up when I couldn't get HTML forms to work, this was a struggle due to a lack of understanding for HTTP and networking in general. The source code was later uploaded to Github for the purpose of documenting the history of Mize ([6]).

2.2 Items Project

This next iteration already had the concept of an Item, which has a certain type, which was reflected in it's name. The realization in this iteration was, that every peace of data could be represented as a JSON document, even for a file, you could encode it's contents with Base64 ([5]) and have that stored with also the metadata of the file in a JSON document. So the plan was to make a webapp, as that was the new thing I learned just then, that could show JSON documents using different React ([24]) Components depending on the `_type` field of the document. The backend

was a server also written in JavaScript, who talked to a MongoDB ([19]) database where the JSON documents were stored.

This project was already able to show all the Moodle ([28]) courses, in which a Moodle account was a member in, with it's own GUI, but the implementation was not future proof at all. Also the requirement to have a working MongoDB somewhere to setup this system was annoying, I wanted to just be able to run an executable, which would also handle all the storing of data and not require any external services.

2.3 Mize

For the third iteration I thought, that to make something proper, it would need to be implemented in a lower level language. So initially C was conceived, but then I found and learned Rust, which turned out to be a great language choice for this project, also because of it's memory safety guarantees at compile time ([27]). The goal was to refine the ideas from the Items Project and implement them in a proper way, that would also be future proof, extensible and have a simple core with many add on modules for all extra functionality.

Initially the messages of the protocol had a custom encoding logic, where the interpretation of each byte was hardcoded into the decoding function. The plan with this was to save bandwidth, because we don't have the overhead of JSON. This turned out to be a stupid idea, as a value of type u64 ([30]) with a value of 1 was encoded as a 1 with seven 0 bytes following. After realizing this, at commit `9825c5242b7a71bbd119e83af95b2f77af16571f` the messages were switched to be simple JSON documents.

The next big change of Mize was the focus on portability, which is explained in more detail in 3.1.6, as it is already in the time frame of the diploma thesis. This also facilitated the change of the protocol's encoding to Cbor.

3 Implementing MiZe

3.1 The Data engine

This chapter is about the implementation of the so called "Data engine" that will form the basis of the Mize platform. Other projects or products with the same or similar goals were examined in 1.1.3. My previous attempts at making a general data management system and how the idea came to be were looked at in 2. In this chapter we look at the latest attempt, which will hopefully also be the last. It incorporates all the learnings from previous ones as well as a lot more experience with computer systems and programming.

3.1.1 The Item

Each piece of data is called an Item in this data engine. An Item works much like a mixture of the well established data storage concepts File and Folder. Files hold a list of bytes and Folders hold a list of links to other files of folders. An Item has both a list of bytes (also called the value of an Item) and a list of links to other Items.

The Type System

In an ordinary filesystem the type of the data in a file is defined by the string after the last dot or by the first few bytes of the file being a "magic value" ([23]). An Item of the Mize data engine, in addition to having a list of bytes and a list of links, also has a type. The type of an Item is stored as a string at the path "type". This type string is a with space separated list of the names of one ore multiple types. A type can specify how to interpret the bytes in the value of the Item and also what sub Items at what paths with what types the Item needs to have. This includes multiple levels, so also the sub Items of sub Items and so on are defined by a type. The type of the sub Item is then whatever the parent Item defines it to be and the type of the parent with a slash and the path, of where the sub Item is at, added to it. There can however only be one type, that says how to interpret the value bytes. All the other types can only have path definitions or further define the interpretation.

This type system allows Items to have really precise types, where some application needs to add special data to it, but also if an application only deals with a very generic type, it can just ignore the more specific types in the type string.

Lets look at the type system with the example of a note in my Obsidian Vault ([21]). It has the type string: "Note ObsidianNote MarkdownNote File MarkdownFile LinuxFile PosixFile NFSv4ACL".

- The type "Note": defines, that the bytes in the value should be interpreted as a UTF-8 string. Applications that want to just modify or display the text of the note can see the Item as just that, a string. The type Note can therefore be seen as just an alias for the type "String".
- "MarkdownNote": further details that the string is actually Markdown source code. Also markdown notes can have yaml properties at the top of the source, so MarkdownNote also defines, that there is a path "properties", where all properties are mapped into.
- "ObsidianNote": is the type, which has paths, to put all information that is specific to the obsidian note taking application ([20]). For example in Obsidian every note is part of a so called vault ([21]) so the type ObsidianNote will include a link at the path obsidian_vault to the Item, that is a the vault the note belongs to.
- "File": Obsidian stores every note as just a markdown file on your system. So our note can also be seen as a "File" stored on some filesystem. ([9]) This type means the value of our note Item is the content of the File. And we have sub Items at certain paths for file metadata, that files have on all platforms.
- "PosixFile": There is file metadata unique to UNIX systems like the permission if a user is able to execute this file ([32]). Things like this are again stored in sub Items at certain paths, which give an application access to this UNIX specific data.
- "NFSv4ACL": The file that stores our note, could have special attributes called ACLs or access control lists. There was an attempt to standardize ACLs in the POSIX standard, but that was withdrawn ([22]). Many POSIX operating systems, including Linux, implement the ACLs as defined in the NFSv4 standard but because they are not part of the POSIX standard, they should not be part of the "PosixFile" type, but rather in their own type.

3.1.2 The Instance

The Instance is the main concept of the Mize data engine. Everywhere some code needs to access, store or update some data an Instance will be present and the interacting with data will be done through methods of the Instance. There is a Rust

struct called Instance that holds all necessary state and implements those methods in the file `./src/core/instance/mod.rs` of the Mize source code.

3.1.3 The Namespace

Every Instance has a Namespace associated with it, which will make it uniquely identifiable anywhere. For Instances that are more consuming data or user facing the Namespace will be a UUID. When you create an Instance and don't specifically configure some Namespace a random UUID will be generated. For Instances that are supposed to own data a domain should be used. So I will have for example one Instance, that is my home server, which is reachable from the internet under the domain "c2vi.dev" and also has the Namespace "c2vi.dev".

Additionally to having one Namespace, an Instance can also be part of one Namespace. An Instance on my local laptop would be setup to be part of the Namespace "c2vi.dev". This would make every Item stored on and owned by my home server Instance by default and more importantly means I address the same Items across all my devices, because all my devices are part of the same Namespace.

3.1.4 The MizeId

Each Item needs to be identifiable somehow. This is what the MizeId is for. It is essentially a path, but with two extra concepts.

1) The first element of the path is usually the so called "store part". It is generated by the storage part of the Mize data engine, which can be swapped out for different implementations. Depending on the what storage part is in use and user configuration it can be just an incrementing number like github issues, a UUID, random Base64 strings like YouTube videos or something like Snowflake ([2]) from X.com.
2) Before the first part there is the optional Namespace. It can be omitted, when you want to address an Item on the Namespace your Instance is part of. To address Items from another Namespace you add this Namespace to the front of the MizeId separated with a colon.

A MizeId without a Namespace can for example look like `0/inst/config`. And one with a Namespace like `462acca5-81aa-4da2-bddc-da00d126ba9a:22/type` or `c2vi.dev:0`.

A MizeId can also be represented as a URL ([34]), which would use the scheme named Mize, the authority part would be the Namespace, then the store part as the first element behind the slash and finally rest of the path look as following:

`mize://<namespace>/rest/of/path`. Parameters will be ok to be used for URL like MizeIds and even non URL MizeIds, but for now there was not yet found a use for them in the Mize data engine.

3.1.5 Network of Instances

It is important for Instances to be able to talk to one another, since one of the main goals of the Mize platform is that any data is usable on any device just like it was local.

Topology

The Topology is implemented in a peer-to-peer way. An Instance can establish a connection to some other Instance using one of many transport layers like tcp, quick, websockets, ipc sockets, Bluetooth, serial, shared memory, canbus and usb through which messages about data are then exchanged.

There is however a quite hard problem that exists in such an architecture. How does an Instance know what other Instances need to know about some change in some data? What if two Instances what to update the same data at the same time? This problem gets even harder when an Instance is offline for some time. This can happen if the hardware the Instance is running on is turned off, has no power or no connection to the other Instance. Systems with a distributed peer-to-peer architecture can get quite complex, because of that. There is for example the concept of CRDTs ([1]) which stands for "conflict free replicated data types". CRDTS were explored for use in this project, but not used because of their complexity. The possibility to add CRDT functionality later is planned for.

Because a Server-Client architecture is so much simpler, Mize uses an architecture similar to that on top of the peer-to-peer connections. There is always one Instance that has ownership of an Item, it's master Instance. With this there is always one Instance knowing the newest state of an Item and all updates to an Item have to eventually go through this one Instance. With the ability to send special "maybe updates" to peers directly. Such updates may then be reverted or changed by the master Instance of the Item. This is useful to display updates to the user faster, in case the master Instance is further away network wise, or if it is not reachable at all. The master Instance of an Item can also change dynamically if this is needed.

In stark contrast to the established Server-Client model, not all Items of a Namespace have to have the same master Instance. This will be useful for larger deployments where the traffic for all the Items of the Namespace would be too much for one server to handle. In such a deployment the intended Topology is as follows. There

are frontend Instances that are distributed across the globe, acting as sort of a CDN. Users connect to the frontend Instance, that is geographically closest to them, which is done using the DNS System. And then there are backend Instances that are the actual master Instances of the hosted Items. Each frontend Instance has to take care of only some of the users of the service and each backend Instance has to be master of only some of the Items, or even only one Item, if necessary. No single Instance has to deal with all the traffic. With this setup also services with a lot of demand should be possible.

Cbor

Cbor (= Concise Binary Object Representation) is a data encoding format similar to JSON, but binary instead of Text based, making it not human readable. A binary encoding scheme was chosen instead of the Text based JSON for two advantages. Firstly less overhead is added by the encoding itself and secondly it supports encoding arbitrary byte sequences. With JSON such arbitrary data sequences would be encoded into Base64 first, which further increases the size of encoded data by a third.

Protocol

In order to exchange data and changes to data over the connections between Instances, a protocol is needed. The protocol of the Mize Data engine works by exchanging different types of Messages between the Instances. A Message is made up of Cbor encoded bytes, that represent the Cbor type called “map”. A Cbor map maps keys to values and is comparable to an Object in JSON. The value mapped by the key “1” is the Command field or the type of the message. Different types of messages then have different keys that they need. The key “2” denotes the MizeId, that the message is about.

Types of Messages At the current state, where simple data exchanges can take place, only eight types of messages are implemented, as seen in `./src/core/proto.rs` - `CMD_GET` ... sent to get the data of the Item specified by the id field. - `CMD_GIVE` ... sent in reply to a `CMD_GET` containing the data of an Item. - `CMD_UPDATE` ... sent by the Instance with ownership of an Item, to tell other Interfaces about changes in that Item. - `CMD_UPDATE_REQUEST` ... sent to request a change in an Item where the sending Instance is not the owner of. - `CMD_SUB` ... sent to the Instance with ownership, to let it know, that you want to be notified of updates to an Item. - `CMD_GET_SUB` ... the same as sending a `CMD_SUB` and a `CMD_GET` as separate messages, because getting the data and subscribing to updates are often needed at the

same time. - `CMD_CREATE` ... ask an Instance to create an Item, which it will then be the owner of. - `CMD_CREATE_REPLY` ... sent in reply to a `CMD_CREATE` containing the MizeId of the newly created Item.

The fact, that a message is just a Cbor map, makes the protocol extremely extensible. This is by design and inspired by the HTTP Headers, which are of a similar data structure that allowed HTTP to extend it's capabilities by defining many new Headers.

3.1.6 Portability

Before commit `a16c2f92217c79445650ce1ce2e8ef6391e849c3` the implementation plan was to have one server, which was written in Rust, that would take care of storing all Items and handling updates and so on. There would then be client implementations in any language Mize can be used with. This implementation plan can be seen in older commits of the Mize repository and was already syncing data between the server and a JavaScript client implementation.

Around the above mentioned commit it was realized, that a lot of the logic is implemented multiple times. Once in the server and once in the client. The client for example also has to store the Items even if only in ram. But it may be even desirable for it to store them on disk, to be kept across restarts. Also all the logic to update the data of Items has to be re implemented in every client language.

One library for all languages

The question then was, can a, what was since then called Instance, be written in a language, that can then be used as a library by any other language. So that this Instance code would become the server as well as the client. It turns out, that almost any programming language can somehow interact with a C library. C is a very old language that is still widely used today. Any language needs to call C functions from the C standard library if it wants anything from the Operating System anyway, so almost all languages have a way to call functions from a shared C Library. In Java you can for example use `System.loadLibrary("libname")` which defines special to Java exported C functions. In python there is the CDLL function and then you can call any normal C function of the shared library. This works the same in any language I can find.

JavaScript running under for example NodeJs can use shared C libraries in the same way as any other language, but what about JavaScript that runs in the Browser? There is a project called asm.js ([31]), which can compile C into a JavaScript, using only a subset of JavaScript's operations, expressions and functions in order to

run faster. Using this we can compile our Mize Instance C Library to JavaScript and use it in the Browser. In recent years a standard called WebAssembly was developed, which allows to run C and most other compiled languages in the Browser. WebAssembly is as the name suggests an Assembly like language, a list of instructions similar to what a CPU would execute. The instructions that WebAssembly defines are however optimized to be run by a virtual machine, making it also similar to the byte code Java is compiled into, which is then run by the JVM (= Java virtual machine). Such virtual WebAssembly machines are now part of every modern web browser.

It might seem that the logical conclusion would be that the Instance code has to be written in C, but this is not the case. It is not just easy to use C shared libraries in many languages, many languages also make it easy to make C like shared Libraries with them. Rust is one of them. In Rust a function can be declared with a special `extern "C"` and `#[no_mangle]` directives, which will make the compiler compile it in a C way. You can then also tell the Rust compiler to create a C shared library in addition to the rlib (= Rust library) file.

JVM languages

Languages that run in the JVM (= Java Virtual Machine, [26]) like Java and Kotlin are not compiled to machine code, but to JVM byte code, which is then executed by the JVM. This enables a compiled program to run on any system which has a JVM. JVM programs can use C like libraries, which contain machine code, through the use of the JNI (= Java Native Interface [13]). This however removes the portability of JVM applications. It is possible to include multiple versions of the C library and choose the right one for the system at runtime, but this is far from a clean solution. There is a project called “Asmble” ([3]), which claims it can transpile WebAssembly Instructions to JVM bytecode. This would allow to distribute the Mize library for JVM Languages as only JVM bytecode, improving portability by a lot.

Cleaner Codebase

The decision that the Rust code for the Instance should run anywhere to be uses as a library by any programming language, made it necessary to separate the logic of the Instance itself (which will be the same no matter the environment and language) from the logic that interacts with the platform beneath, on which the Instance code runs on. Where the platform beneath could be an operating system, the browser, the hardware directly in case of embedded systems without an operating system, the JVM and many more. This restriction, that logic, that is part of the Instance can not do any direct interaction with for example the operating system, made the code

a lot cleaner. All the Code that is part of the Instance logic, is in the folder `src/core` and all logic, that has to do with a particular platform is bound in `src/platform`.

3.2 Packaging and Distributing

Software Packaging is the process of getting all parts of a Software Application into a format, that can be used to run the Application (or use the library) on a certain user's system. Distributing is about how you then get the application or library to the user. ([29])

It is a goal of Mize to be able to run on any computer system, from large servers to laptops to micro controller powered smardHome devices. To run Mize under Windows, Linux or MacOS it should be a simple as downloading the right executable file and running it. No installation required and it should just work. To achieve this some thought needs to go into the packaging and distributing part of Mize.

As mentioned in 3.1.6, the core logic code that is written in Rust will be used everywhere and by any programming language where Mize is used. This means, that if for example someone writes a program in python, which uses the Mize library, the machine code of the library may need to be different depending on system the python program is executed on. Things like this and many more need to be taken into account, when thinking about how to package and distribute Mize.

3.2.1 Differences between computer systems Mize should be able to run on

There are countless details/characteristics that can be different between a user system (also called the target system) and the system that is used to develop in this case the Mize platform. Also user systems might vary across many of said details. The software you write is only use full to the user, if it runs on whatever system the user has.

A broad summarily of such characteristics of a computer system follows.

OS Kernel

A kernel is the main component of an operating system and the only one that actually interacts with hardware. In order to be able to talk to hardware it needs to run with higher privilege than all other programs of the operating system and all

user programs. A kernels job also includes memory management, giving device agnostic access to hardware devices, process management and file systems.

The two most used kernels are the Windows Kernel, which is the most popular option for desktop computers and Linux, the most popular option when it comes to server systems and smartphones. Two other frequently used kernels are XNU (standing for X not Unix), the one used and developed by Apple for all of their operating systems and the FreeBSD Kernel development by the FreeBSD Foundation.

The differences important for distributing our software are the apis used by user space programs to interact with the kernel, also called system calls or syscalls, and the file formats executable programs and libraries need to be in, so that their kernel can execute/load the machine code.

Another possibility that can be considered for distributing software is systems without a kernel or small micro and realtime kernels. Such scenarios are often found in embedded devices and are also called "bare metal" systems.

Syscalls A userspace program has to use syscalls to do anything that is not modifying it's own memory pages. Running another program, reading a file, using a network interface, allocating more memory, ... are all things somehow done via a syscall. In order to invoke a syscall a program has to set some CPU registers according to the syscall spec and then execute the "syscall" instruction.

Unix based kernels like Linux, FreeBSD Kernel and XNU take the approach of having a small number of syscalls and using special files and ioctls to provide all needed functionality. This comes to around 100 syscalls. The Windows Kernel has multiple thousands of syscalls and they can change with every release of Windows. Unix kernels keep their syscalls the same forever, or at least stay backwards compatible by only adding extra ones.

File Formats The file containing the executable code we distribute has to be in the right format for the kernel to load it into memory and start executing it as a new process or a dynamic library.

The Executable Linkable Format (= ELF) is what used by Linux and FreeBSD kernels. Windows and the former DOS have the Portable Executable format and the Apple kernel uses a file format called Mach-O.

CPU architecture

The CPU architecture defines what instructions a CPU can execute and how they need to be structured. Common examples for CPU architectures are x86, arm, avr and riscv, but many more exist.

OS userspace

Windows solves their ever changing syscalls by providing system libraries like for example kernel32.dll and user32.dll, which programs should use to interact with the kernel. Also when using Unix kernels syscall aren't invoked by your code directly, there is a so called "standard library" for that. Besides having functions, that directly call syscalls standard libraries also have a lot of functionality that makes interacting with the kernel more friendly. Multiple of such standard libraries exist. The most used one is glibc, made as part of the GNU project. Others are Musl, bionic, newlib and some more.

Userspace wise Windows makes it easy for us, as there is only one Userspace. Linux however has over 1000 different distributions. Some are very similar, like Debian and Ubuntu and some do things completely differently, like Android or NixOS.

Available Hardware and Drivers

On some systems there will be special hardware for example rendering graphics or decoding a video stream. Our software should take advantage of that if available.

3.2.2 Implementing the distributing part

Now that we know what all needs to be taken into account for the packaging of this project, the following paragraphs document the implementation of the distributing and packaging part of Mize.

Cross Compiling

Cross Compiling is the process of building software that should run on a computer system that is different, in regards to one or more of the characteristics mentioned in 3.2.1, to the system used to compile the software.

Copying the source code of Mize onto every system we want to release for and build it on there, is just not practical. That would require to have one of every

kind of system and take way too long. So cross compiling is certainly a thing we need to implement into the distributing process. Cross compiling usually means installing some sort of cross toolchain and running that to compile the source code. Every system we want to compile for, will of course need it's own toolchain installed. This imperative approach firstly takes a lot of manual effort and secondly it often happens, that all the installed cross toolchains and also toolchain local to the system interfere with one another, which can produce unexpected errors during compilation. Those errors then have nothing to do with something being wrong in any of the compiled source code, but rather in the toolchains and how they were installed.

Nix

Nix is the name of a package manager and a domain specific language, that is used by this package manager to define packages.

The first major difference to most other package managers is that a package is not defined as a set of attributes like the version, the name, list of dependency packages. With Nix however a package is a function in this Nix language. The parameters of this function are, every dependency, compilation options, the compiler and many "build functions" special to the targeted system and the language of the project. The functions that defines a package then calls one of those "build functions" passing it things like name, version, source-code and other metadata of the package.

With most package managers apart from Nix a package defines the path it is installed into. The package manager simply runs the install code of the project using for example `make install` for make based projects. This method leads to a significant problem when you'd want to install two different versions of a package, both versions will want to install files to the same path. The Nix package manager installs a package only into a path that is unique to that package. Such a path looks for example like this: `/nix/store/<hash>-<name>-<version>/`, where `<hash>` is a hash of all used "inputs" (the arguments passed to "build functions"), therefore being only the same if you install a package with exactly matching dependencies, compiler options, target system, and so on.

How Nix helps with cross compiling

Nix runs all builds in a sandboxed environment, where only the needed programs, dependencies and toolchains are installed. This fully eliminates errors that arise because of the toolchain, how it is installed or interference with other installed toolchains or programs. All the details off this sandboxed environment and what toolchain and dependencies are installed is decoratively defined in the package

definitions using the Nix DSL. It is also worth noting, that nix uses hashing, to make sure to use exactly the same exact versions of toolchain, dependencies, programs, making those environments fully reproducible.

The Module System

Only the core parts of the Mize platform are in the Mize library itself. All other functionality will be provided by external modules. Also a type can provide a module, which contains functionality for this type, for example code, to check if a update is valid. A module is a folder containing a `mize_module.nix` file, in which you define with the Nix DSL, what the module does and how to build it.

The distributing process

After the nix definitions are written, it becomes possible to run one command, that builds all versions of Mize and all modules for all computer systems. The command is `nix build github:c2vi/mize#dist` and can be run on any Linux system, with the only thing, that needs to be installed being Nix. This creates a path in the Nix store, which can then simply be rsync'd onto a webserver, where users can download the correct executables and other files. There is a shell script in the Mize repository called "deploy", which does basically that and also checks, if the path is successfully created by the nix build invocation with `[[ "$path" != "" ]]` and only then runs the rsync command. `ocih` is the hostname of the host where the webserver is located. This host can only be accessed, when this script is run on my local machine, where it can access the right private key.

```
path=$(nix build .#dist -L -v --print-out-paths $@)
[[ "$path" != "" ]] && rsync -rv $path/* ocih:host/data/my-website \
--rsync-path="sudo rsync"
```

Appendix

All the code from the relevant repositories is included as a single zip file, that contains a folder for each repository. The repositories are:

- The main Mize Repository, containing the data engine: [14]
- MME, the Mize Explorer: [16]
- Victorinix: [33]
- MME Presenters: [18]
- Modules for Mize: [15]
- Command engine: [7]
- Command line explorer for Mize: [17]
- Archive of the Coding Wiki project: [6]
- Archive of the ItemsProject: [12]
- Source code of this document: [8]
- The tool used to build this document from it's source code: [10]

Bibliography

[1] *About CRDTs • Conflict-free Replicated Data Types*, [Online; accessed 24. Mar. 2025], Oct. 2024. [Online]. Available: https://crdt.tech.

[2] *Announcing Snowflake*, [Online; accessed 12. Mar. 2025], Mar. 2025. [Online]. Available: https://blog.x.com/engineering/en{_}us/a/2010/announcing-snowflake.

[3] *asmble*, [Online; accessed 14. Mar. 2025], Mar. 2025. [Online]. Available: https://github.com/cretz/asmble.

[4] *AutoHotkey*, [Online; accessed 18. Mar. 2025], Feb. 2025. [Online]. Available: https://www.autohotkey.com.

[5] *Base64 - MDN Web Docs Glossary: Definitions of Web-related terms | MDN*, [Online; accessed 24. Mar. 2025], Feb. 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/Base64.

[6] *coding-wiki*, [Online; accessed 27. Mar. 2025], Mar. 2025. [Online]. Available: https://github.com/c2vi/coding-wiki.

[7] *comandr*, [Online; accessed 24. Mar. 2025], Mar. 2025. [Online]. Available: https://github.com/c2vi/comandr.

[8] *dipl*, [Online; accessed 24. Mar. 2025], Mar. 2025. [Online]. Available: https://github.com/c2vi/dipl.

[9] freeCodeCamp, "What Is a File System? Types of Computer File Systems and How they Work – Explained with Examples," *FreeCodeCamp*, Jan. 2022. [Online]. Available: https://www.freecodecamp.org/news/file-systems-architecture-explained.

[10] *htldoc*, [Online; accessed 24. Mar. 2025], Mar. 2025. [Online]. Available: https://github.com/c2vi/htldoc.

[11] *Introduction to Pipes in UNIX systems | Logdy*, [Online; accessed 24. Mar. 2025], Mar. 2025. [Online]. Available: https://logdy.dev/blog/post/introduction-to-pipes-in-unix-systems.

[12] *items-projekt*, [Online; accessed 24. Mar. 2025], Mar. 2025. [Online]. Available: https://github.com/c2vi/items-projekt.

[13] *Java Native Interface*, [Online; accessed 14. Mar. 2025], Jan. 2024. [Online]. Available: https://www.baeldung.com/jni.

[14] *mize*, [Online; accessed 24. Mar. 2025], Mar. 2025. [Online]. Available: https://github.com/c2vi/mize.

[15] *mize-modules*, [Online; accessed 24. Mar. 2025], Mar. 2025. [Online]. Available: https://github.com/c2vi/mize-modules.

[16] *mme*, [Online; accessed 24. Mar. 2025], Mar. 2025. [Online]. Available: https://github.com/c2vi/mme.

[17] *mme-cli*, [Online; accessed 24. Mar. 2025], Mar. 2025. [Online]. Available: https://github.com/c2vi/mme-cli.

[18] *mme-presenters*, [Online; accessed 24. Mar. 2025], Mar. 2025. [Online]. Available: https://github.com/c2vi/mme-presenters.

[19] *MongoDB: The World's Leading Modern Database*, [Online; accessed 24. Mar. 2025], Mar. 2025. [Online]. Available: https://www.mongodb.com.

[20] *Obsidian*, [Online; accessed 10. Mar. 2025], Mar. 2025. [Online]. Available: https://obsidian.md.

[21] *Obsidian Vault*, [Online; accessed 11. Mar. 2025], Mar. 2025. [Online]. Available: https://help.obsidian.md/vault.

[22] *POSIX Access Control Lists on Linux*, [Online; accessed 11. Mar. 2025], Nov. 2019. [Online]. Available: https://www.usenix.org/legacy/publications/library/proceedings/usenix03/tech/freenix03/full{_}papers/gruenbacher/gruenbacher{_}html/main.html.

[23] S. Purohit, "Beneath the Bytes: A Deep Dive into Magic Numbers for File Identification," *Medium*, Jan. 2024, ISSN: 4213-1214. [Online]. Available: https://medium.com/@shailendrapurohit2010/beneath-the-bytes-a-deep-dive-into-magic-numbers-for-file-identification-4bff213121c4.

[24] *React*, [Online; accessed 24. Mar. 2025], Mar. 2025. [Online]. Available: https://react.dev.

[25] *Reverse-engineering an encrypted IoT protocol*, [Online; accessed 23. Mar. 2025], Feb. 2024. [Online]. Available: https://smlx.dev/posts/goodwe-sems-protocol-teardown.

[26] S. Rosado, *Understanding the JVM: Java Virtual Machine - Samantha Rosado - Medium*. China: Medium, Aug. 2023, ISBN: 978-707331123. [Online]. Available: https://medium.com/@sammierosado/understanding-the-jvm-java-virtual-machine-b70f73311237.

[27] *Rust*, [Online; accessed 22. Mar. 2025], Mar. 2025. [Online]. Available: https://www.kadasolutions.ch/blog/rust-a-modern-language-for-safe-and-fast-general-purpose-programming.

[28] *Startseite | Moodle.org*, [Online; accessed 24. Mar. 2025], Mar. 2025. [Online]. Available: https://moodle.org/?lang=de.

[29] *Summary of what Software Packaging and Distribution is by ChatGPT*, [Online; accessed 24. Mar. 2025], Mar. 2025. [Online]. Available: https://chatgpt.com/share/67e0ac0b-ef64-800e-808f-20ac12c9cf2a.

[30] *u64 - Rust*, [Online; accessed 22. Mar. 2025], Mar. 2025. [Online]. Available: https://doc.rust-lang.org/std/primitive.u64.html.

[31] *Understanding asm.js*, [Online; accessed 12. Mar. 2025], Nov. 2024. [Online]. Available: https://www.sitepoint.com/understanding-asm-js.

[32] *Unix / Linux - File Permission / Access Modes*, [Online; accessed 11. Mar. 2025], Mar. 2025. [Online]. Available: https://www.tutorialspoint.com/unix/unix-file-permission.htm.

[33] *victorinix*, [Online; accessed 24. Mar. 2025], Mar. 2025. [Online]. Available: https://github.com/c2vi/victorinix.

[34] *What is a URL?* [Online; accessed 12. Mar. 2025], Mar. 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn{_}web{_}development/Howto/Web{_}mechanics/What{_}is{_}a{_}URL.