KPI Report for the Wormhole Project

Introduction

The Wormhole project aims to provide a decentralized data storage solution, offering a secure and simple alternative to centralized and cloud-based systems. This report presents the key performance indicators (KPIs) that have been achieved so far, in accordance with the guidelines of Epitech.

Evaluating and Integrating New Technologies

Regular Technology Watch

Our project pushed us to stay aware of current tech trends, starting with general sources like social media. But as development moved forward, we had to dive into more specific and technical areas that were directly relevant to what we were building.

We split our attention across a few key aspects of the project.

We first looked into what the blockchain community was doing. Even though Wormhole isn't a blockchain solution, we share some of the same goals, like decentralization and resilience. The Blockworks blog gave us a broad view of that ecosystem, while the IPFS blog was even more relevant since IPFS is also a non-blockchain decentralized filesystem.

A big part of our work was making sure Wormhole integrates well with Linux. So we followed blog posts and documentation around the Linux kernel and FUSE (kernel layer used to give files to the operating system). One of the most useful resources was a Systemd Conference talk, which really made us rethink how we approached system integration with systemd.

Finally, we spent time researching the fast-evolving world of distributed filesystems. The more we looked, the more different tools and approaches we found, each with their own way of solving the same core problems. We made an effort to stay in touch with ongoing developments in projects like:

- Ceph
- GlusterFS
- Tahoe-LAFS
- SeaweedFS
- IPOP
- IPFS
- MooseFS

This research helped us refine our ideas and stay grounded in real-world solutions.

Research Documentation

We had to conduct multiple research during our project development, starting with the programming language. Our programming language prerequisites were precise.

- **High data integrity**
  (no unpredictable behavior, even in the event of an error)
- **Native support for Linux, Windows and possible MacOS support**
  The mesh must be able to be composed of several different systems, to increase the network access.
- **High-performance**
  Relatively close to the machine (without overlay).
  Need to process large quantities of data quickly.
- **Permissive**
 We are creating a new technology, so our tools must not be limited by what already exists.

Our different candidates were C, C++, Go and Rust. We started by studying a few articles and benchmarks, like this. We built a general understanding of the state of the different languages we studied regarding our specific needs.

	C	C++	Go	Rust
Safety	- -	-	+	++
Performance	+++	++	+	++
Permissive	+++	+++	+	++

After this general study we decided to test our intuitions with a one month Rust prototype attempting to verify the feasibility of our project in Rust. And after this prototype we were confident in our choice because of theses specific aspect of the Rust programing language:

- **"Security-first" policy:**
  The language has a very important focus on memory safety and producing bug free code. This ensures no unpredictable behavior, even in the event of developer error.
- **Supported by all major platforms.**
- **High performance:**
  Compiled in machine language
  Performance between C and C++ on most benchmarks
  More performant than Go
- **Few overhead:**
  Low-level language making it compatible with filesystem technologies like FUSE (similar to C++ or C)

We also had to choose which operating systems to support in priorities. Our project is foremost directed at enterprises so we had to find what operating systems enterprises servers are using.
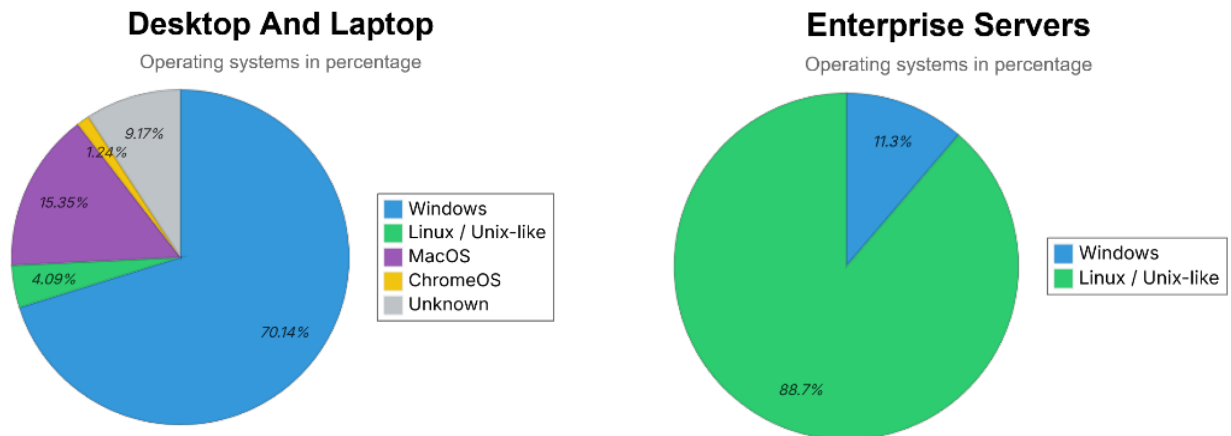
After investigation, we found that supporting windows and linux would be the most efficient to support enterprise servers and enough desktop and laptop to access the said servers.

We also had to choose what native integration technology we would choose.
This is the choice we made:

Choice of Linux integration: Fuse (via Fuser)

- Fuse (Filesystem in Userspace). Linux kernel module.
- Allows file system creation by a user program.
- Fuser (Rust library). Port of the Libfuse C library to Rust.
- It also supports MacOS.

Choice of Windows integration: WinFsp (via WinFsp-RS)

- WinFsp (Windows File System Proxy). Kernel module for Windows.
- Allows file system creation by a user program.
- Rust library of the same name.

On the opposite of the other research the choice was pretty simple. For the linux and rust side, fuse-rs and Fuser were the only two relevant libraries to use Fuse through Rust. We chose Fuser because the fuse-rs hasn'nt been maintained for 5 years.

For windows only WinFsp-RS was relevant as a maintained Rust framework.

Practical Application of New Technologies

At the beginning of the EIP, we took a week to create a first prototype, a POC, to demonstrate the project's technical feasibility.

We then asked Axel to produce a UML diagram to redesign our whole project architecture to be more scalable. It took a full sprint of design. But it allowed us to plan ahead a new architecture from scratch instead of just continuing the gradual improvement. And this has been a key part of the development because it allowed us to fix our very quickly growing complexity, even if it meant rewriting a large part of the code.
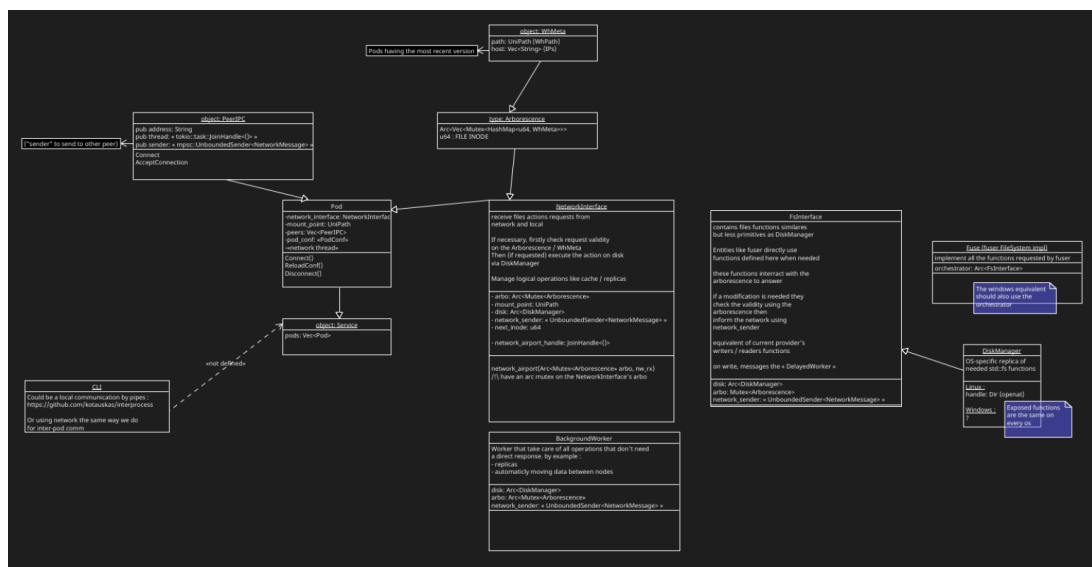
Today, a functional prototype of Wormhole has been published and is usable under controlled conditions. This marks a significant milestone in the project's development.

Regarding tutorials for the project, we inspired ourselves a lot from this example file from the official fuser codebase. Even if it's not a traditional tutorial, it allowed us to find how to implement Linux filesystems with fuser. Implementing a filesystem has lots of room for small errors allowing weird behaviours to slip in, this showed us what was the idiomatic way of doing things.

Engagement in Tech Communities

We didn't discuss a lot about our project in tech communities yet, as we want the project to be a bit more mature or "user ready" before the announcement. This would allow us to have a greater impact directly by leveraging hype, rather than just throwing out ideas that are not yet done.

However, we were active in the communities of tech that we are using, like the libraries Fuser or WinFsp-rs where we have two contributions:

- Correction of an error type in Fuser
- Memory leak correction in WinFsp-rs

Protecting and Leveraging our Technology

Legal protection

Our project is open source. Everyone can see the source code, and use it for free. This changes rather radically the type of legal protections we need, as we are not protecting our software against thieves. You can see why we made this choice in the economic plan section.

However, even though our project is free of access, we had to set a certain number of boundaries defined in our licence. We chose the AGPL licence. You can see a nice summary of it on the official GNU website.
This licence forces everyone that modifies the code to publish it under the same licence, if the edited version is distributed or used in a service publicly available to users. The published version can of course be used back by us to improve Wormhole.

Technical protection

As our solution is itself a security solution (securing your files through redundancy), we must ensure that our code is safe if we don't want to become a security threat ourselves. We need to:

- Ensure integrity of the data we keep
- Ensure we don't create doors for cyberattacks

Protection of data integrity

Here is what we do and use to ensure that no data is corrupted or made missing:

Using the Rust language:

As you saw in the research documentation section, Rust is designed in such a way that it doesn't allow any undefined behavior.
This means that every piece of data must be well defined, and can't go missing without it being clear and expected. The same goes for conditional actions, if a choice must be made, the developer is forced to stipulate behavior in every case.

This contrasts with other system languages like C or C++. As they have some more freedom on how the developer code things, but also make it easy to corrupt data (notably pointers management) or fall in unexpected behavior if the developer missed a case.

Using code reviews:

While Rust forces us to define behavior, it doesn't prevent us from writing *wrong* behavior. That's where code reviews steps in. Before publishing a feature, we have at least one other

member of the team checking what was changed, ensuring another layer of safety for the published code.

Using code testing:

We create unit tests and functional tests to ensure that our code has a correct behavior.

Using websockets:

Websocket is the technology that we use to send data between computers. It is a mature technology, and uses safe protocols like TCP. This ensures that sent data is received.

Protection against Cyberattacks

Open source project:

While hackers can freely read the code to find flaws, so can our user base. This approach has proven to be effective on almost every widely adopted open source project.

Compatibility with operating system security

As Wormhole is only a layer managing data, it stays compatible with operating system level security, like disk encryption or antiviruses.

Futur SSL support

SSL is similar to HTTPS, but for websockets. Supporting this will allow communication between two Wormhole instances to be encrypted and secure.

Market Surveillance and Audit

Wormhole distinguishes itself from its competitors in several ways:

Comparison of Distributed File Systems

Characteristic	Gluster FS	Moose FS	IPFS
Disk caching	Less integrated	No	Yes
Complexity	Moderate installation	More complex	light or moderate installation
Target	Businesses/ individuals	Data centers	Collaborative cloud

The main difference point overall, is that Wormhole aims to be a simple yet effective solution, working "out of the box". We are not mainly targeting petabyte size storage like Gluster or

Moose FS, nor having the same performance as solutions that use complex setups with their own partitions like Ceph.

But what we can provide **is an effective solution of large and safe data storage for companies, yet without requiring an entire team to maintain.**
Our solution has no minimal requirement besides having at least two computers, and can be installed on users endpoints as well as on dedicated servers. Can be used by humans as well as by other softwares.

You can see more results of our technology watch in the regular technology watch section.

Rapid prototyping to secure the idea

As you can see in the practical application of new technologies section, we firstly created a first prototype to ensure the project feasibility.

We then worked to quickly have a functional prototype, even if it cuts some corners. It is today publicly available on our Github, and we are now working to improve it (stability and features).

Functional and Technical Audits

- **Functional Audit**: Flexibility, simplicity, and compatibility are the strengths.
- **Technical Audit**: Rust is suitable, native integrations anticipated, and focus on network algorithms and architectures.

Economic Plan

We chose to be an open source project for a few reasons:

- Large visibility and unrestricted community adoption
- Possibility to keep the project alive even if we one day stop maintaining it.

At first sight, it has the quite important drawback that no one pays us for using the project. But looking into it more, being closed source or paid also have drawbacks:

- Losing the open source perks
  - No free contribution
  - No free security review
  - Impossible large scale adoption by the community
- Few companies would actually pay for an unknown service not widely used

That's why we chose to stay open source. If the project popularity explodes, we can still leverage the success, ideas does not lack:

- Prestations for companies (Advice, installation and maintain at low cost)
- Starting our own cloud storage service to add nodes to your cluster

- Renting a small part of the storage of large companies to smaller companies (in exchange to a reduction of our presentation cost)
- And much more…

Collaborating with Technical Experts

Several collaborations have been conducted to enrich the project.

When starting the project, we had meetings with BPCE Infogérance et Technologies and Grant Thornton, two large companies using sensitive data. We asked them about the methods they use to protect their data integrity and availability. We saw that they use rather radical means:

BPCE-IT is a bank, and therefore uses a [Mainframe](#) server to process data. The Mainframes are quite the opposite of decentralisation, as they aim to have the most processing power in one single place, to avoid the difficulties of synchronizing separate concurrent servers.

Their integrity / availability technique is hardware. They have two or more mainframes, but only one is running. The second stores copies and can start quickly if the first one fails. The Mainframe themselves have built-in hardware redundancies (many processors, and data mirrored on many disks).

Although this method works, it is insanely expensive at this scale (using only a fraction of the power) and can be considered quite "bruteforce".

Grant Thornton is a law firm. They also have many smaller servers, but also use the same technique of using a fraction of the power while the other half is on standby.

For long term backups, they use [magnetic tape](#). Those are nice long term storage solutions, but are not used at runtime, only for backups. They are stored elsewhere and only used again to restore the system if every other security measure fails.

We also gathered experience from our past internships at Safran (industry), Nokia (telecommunications) and IParcus (cyber-security audit startup).

All this knowledge led us to this conclusion:
The market critically lacks a decent storage framework, so much that large companies spend insane amounts of money just to store their data.

And this is what led us to create Wormhole, made to fill this gap. We aim to be the same little revolutions that Docker and Kubernetes were.

Eric Gilon is a Lead data analyst and Ex-Local IT Support at BeLux. He helped us by participating in a first beta test plan. His professional experience made him a perfect fit to represent a real user of our product and made him able to give us valuable insight of the professional usage of our tool, especially regarding the onboarding process.

You can see proof of the exchange with BPCE-IT and Eric Gilon as an image next to this file. Unfortunately, the exchange with Grand Thornton was a phone call, and we don't have any proof anymore.

As you can see in the engagement in tech communities section, we also conducted two open source contributions to libraries used in the project.