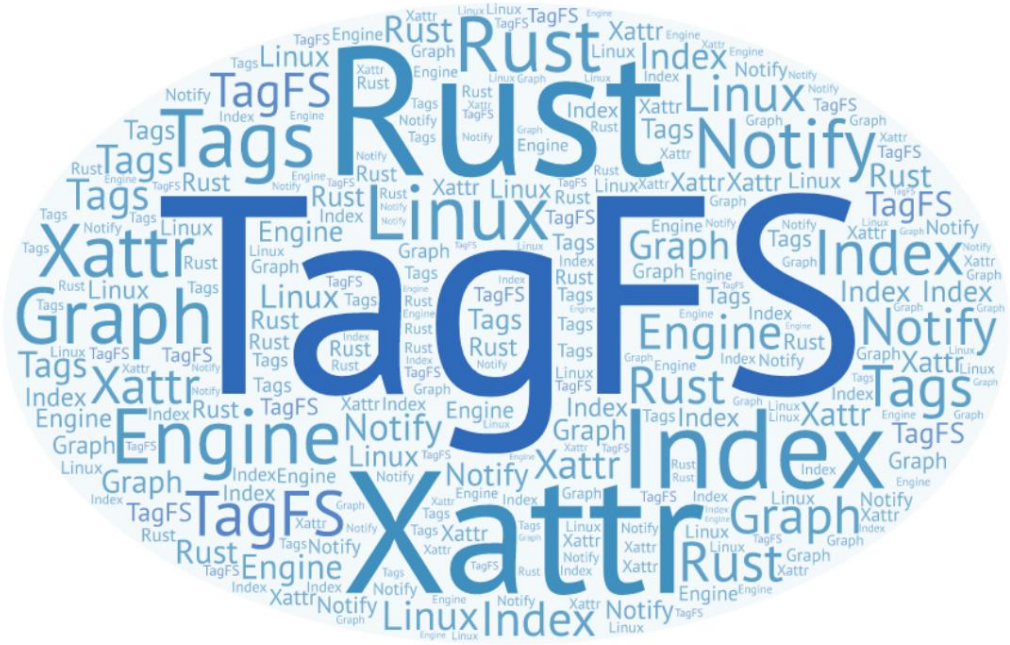
h e p i a
Haute école du paysage, d'ingénierie
et d'architecture de Genève

Hes·SO GENÈVE
Haute Ecole Spécialisée
de Suisse occidentale

TagFS
File labeling system with Rust



Bachelor thesis presented by

Mr Steven LIATTI

for obtaining the title of Bachelor of Science HES-SO in

Information Technology Engineering with orientation in Complex software and systems

September 2018

Responsible HES Professor

Florent LUCKY

h e p i a
Haute école du paysage,
et d'architecture de Genè

spring 2018
Session de bachelor

INFORMATION TECHNOLOGY ENGINEERING

ORIENTATION – COMPLEX SOFTWARE AND SYSTEMS

FILE TAGGING AND INDEXING ENGINE WITH RUST

Description :

The existing file *tagging* systems found in the Linux world today use databases and/or additional files to store metadata for tagged files. The problem related to the fact that the files and their metadata are not stored in the same entity (ie the file), complicates the implementations and makes them sub-optimal in terms of complexity, performance, latency and robustness.

An alternative solution to these traditional approaches would be to take advantage of the *extended attributes* present in most of the file systems available in the Linux kernel (ext4, xfs, etc.). The *extended attributes* allow, for each file, to store or extract a set of metadata, all in an efficient way.

The primary goal of this project is to design and develop a "tag management engine" that can easily and efficiently manage tens or hundreds of thousands of files and associated *tags* .
The secondary goal of the project is to implement it in the Rust language. Indeed, the Rust language is a modern system language, a worthy successor to the C language, and designed to be extremely reliable, robust and efficient.

Indexing mechanisms will be studied and file monitoring systems will be investigated.
Indeed, the system will have to make sure to monitor the files modified, created, or deleted in order to index the tags with a minimum of latency, the goal being to offer performances as close to real time as possible.

Finally, if time permits, the developed system will be integrated with a popular file manager in the Linux world (Nautilus, Thunar, Konqueror, etc.).

Required work :

- State of the art.
- Study and handling of the Rust language and familiarization with it in the system context of the project.
- Analysis of *extended attributes ;* in particular, study of their behavior during operations common accesses (cp, mv, rsync, copies from a file manager, etc.) in order to avoid inadvertent loss of tags in case certain operations do not maintain certain attributes.
- Analysis and selection of file system notification systems.
- Analysis and selection of indexing and search algorithms. • System design and implementation.
- Analysis of the implemented system (performance measurements, etc.).
- Discussion of the use of the Rust language versus a traditional C implementation.
- Demonstrator.
- If time permits: integration with a file manager.

Candidate :

M. LIATTI STEVEN

Field of study: ITI

Responsible teacher(s):

GLUCK Florent

In collaboration with :
Bachelor's thesis subject to an internship agreement in a company: **no**
Bachelor thesis subject to a confidentiality agreement: **no**

h e p i a
Haute école du paysage, d'ingénierie
et d'architecture de Genève

Spring 2018
Session de bachelor

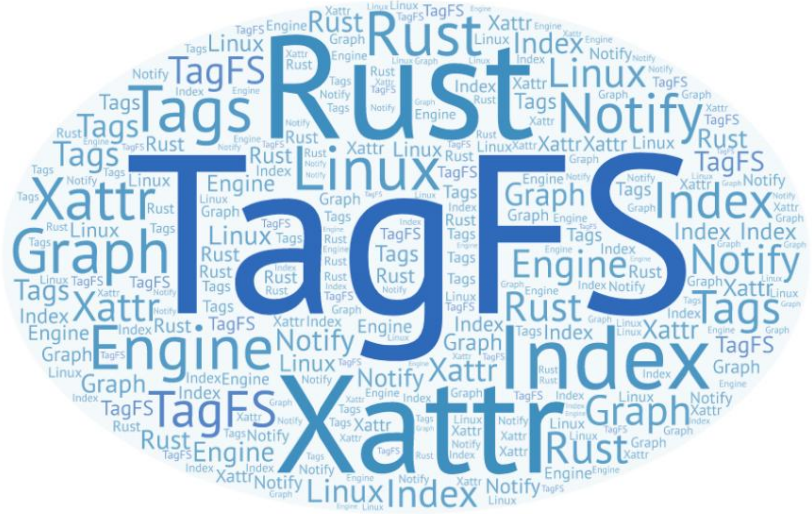
Résumé :

With the increase in computing power and storage capacity for a reasonable price, our personal computers manage very large quantities of files, in the order of hundreds of thousands or a million files. How then can you quickly find a file that matches certain criteria without remembering its location on disk? A solution to this problem is to give the possibility to the user to affix one or more labels, or "tags", on his files and to provide him with an interface with which he can easily find his files. It is important that tags are stored in file attributes so that they are not lost during file manipulations. The user must be able to quickly find his files by searching by tags.

With the aim of solving these problems, the solution proposed in this project is a duo of programs, Tag Manager and Tag Engine, forming the TagFS system. The user can manage tags associated with files and perform file searches by tags using Tag Manager.
Tag Engine works in the background indexing and monitoring the user's tree of files and tags in real time and with very little latency.

To carry out this work, many technologies had to be studied and implemented, first of all the Rust programming language. Rust is a modern, reliable and powerful language, a worthy successor to C. The mechanisms for indexing and monitoring the file system as well as the extended attributes for storing tags have also been analyzed.

The whole project is available on GitHub, at the following address: https://github.com/stevenliatti/tagfs.



Candidate :

M. LIATTI STEVEN
Field of study: ITI

Responsible teacher(s):

Happy Florentine

Bachelor's thesis subject to an internship agreement in a company: no
Bachelor thesis subject to a confidentiality agreement: no

Contents

Shapes table

Table of tables

Table of source code listings

Typographic conventions

When writing this document, the following typographical conventions were adopted.

- All words borrowed from the English or Latin language have been written in italics.
- Any reference to a file (or directory) name, path, parameter usage, variable, user-usable command, or source code snippet is written in monospaced font.
- Any file or code snippet is written in the following format:

```
1 fn main() {
2         println!("Hello, world!");
3 }
```

- In listings, lines preceded by a "$" are executed in a shell.

Document structure

This document begins with the introduction, recalling the motivations of the project and the goals to be achieved. It then discusses existing applications providing a service similar to the purpose of the project, but with some limitations for the most part. It continues with a section on the software architecture of the final system, with its various components and needs. Afterwards, a technological analysis of the Rust language and the tools used for the technical realization is carried out. The following section illustrates the technical realization itself, with the two programs designed. The system is tested in the following section, explaining the protocol and the test results. Finally, the document ends with a discussion section of the results, project status, future improvements and differences between Rust and C and a final concluding section. The references are at the very end of the document.

Thanks

My thanks go first and foremost to my companion, Marie Bessat, for her infinite patience and encouragement. I would also like to thank my family and friends for their support throughout my studies and during the realization of this work. I would like to thank all my teachers for their lessons and especially Mr. Florent Glück for monitoring the project and his valuable advice as well as Mr. Orestis Malaspinas for advice on the Rust language. I would also like to thank Mr. Joël Cavat, ITI assistant at hepia, for having advised me and lent me his very good course on Graphs and Networks by Mr. Jean-Francois Hêche, whom I thank indirectly. Finally, I would like to thank my classmates for their camaraderie and help during my studies and my bachelor thesis.

Acronyms

API Application Programming Interface, Programming Interface: services offered by a producer program to other consumer programs. 17, 49, 51, 53, 55, 64

CLI Command Line Interface, Command line interface: text-based human-machine interface: the user enters commands in a terminal and the computer responds by executing the user's commands and displaying the result of the operation . 13, 28, 47, 53

FS File System, File System: logical organization of physical files on disk. 12–14, 18–21, 24, 25, 27, 47, 49–51, 64, 67, 85

GUI Graphical User Interface, Graphical interface: means of interacting with software where controls and objects can be manipulated. Opposes the command line interface (CLI). 15

OS Operating System, Operating system: software layer between the hardware of a computer and the user applications. Offers abstractions for managing processes, files, and devices among others. 11, 13, 16, 39, 47, 53, 64, 76, 85

SYSCALL System call, System call: when a program needs privileged access to certain parts of the operating system (file system, memory, peripherals), it asks the kernel to perform the desired operation for it. 49–53

XATTR Extended Attributes, Extended attributes: see section 4.2. 11–13, 17, 18, 47–49, 53, 55, 60, 63

1 Introduction

1.1 Motivations

With the increase in computing power and growing mass storage capacity at a reasonable price, our personal computers manage very large quantities of files, in the order of a thousand or a million files. Whether it's pictures, documents or music, the file storage reserves seem endless. This raises the question of the organization of these numerous files. How should we order our personal photos? By date, by place, by theme? These are three good answers, but unfortunately for the user, today's operating systems (OS) only offer one native way of organizing files: the classic hierarchy of directories, sub-directories and files, in the form of a tree structure.

How to quickly find the photo of your cat sleeping on your balcony at the beginning of its life in a mass of more than 10,000 images? How to classify your repertoire of studies, a repertoire for the courses, another for the practical work, or for each course, two sub-directories "theory" and "practice"? How to retrieve a song without knowing its title or the artist but knowing the genre? One solution to these problems is to give the user the possibility of affixing one or more labels, or "tags", to his files and to provide him with an interface with which he can easily find his files. He must keep control over his files and be able to manipulate them as he always has. Tags should be stored with files, so they won't be lost if there's a big change in the system. The use of extended attributes, or extended attributes (XATTR), is the most natural way to meet this last need: the tags thus "travel" with the files. We will see that there are applications that partly solve this problem. Finally, this interface must be efficient and reliable.

These two qualifiers, powerful and reliable, sum up the Rust programming language.
Rust is a modern programming language with a growing and passionate community. It performs very well, close to or better than C depending on the situation. It is reliable thanks to its strict rules on the use of memory and its very intelligent compiler.
It is a language totally adapted to our situation. Through this work, the reader will be able to get an illustration of the possibilities offered by Rust.

The purpose of this work is therefore to design and develop a tag management engine that meets the needs mentioned above and to learn the notions of Rust necessary for its realization.

1.2 Goals

In more detail, the goals of this project are:

— Study and appropriate the Rust language for the realization of a system application under Linux.

— List existing applications for tagging files.

— Study the XATTRs during common manipulations on the files.

— Explore file system (FS) monitoring methods.

— Analyze ways to index a file tree.

— Design and implement a system that responds to the motivations. It must be efficient, usable in real time and manage many files, directories and tags.

— Measure the performance of this system.

— Carry out a demonstration.

2 Analysis of the existing

In this section, we will analyze the main existing solutions, whether in the form of user applications or integrated directly into an operating system (OS). Jean-Francois Dockes also lists them with advantages and disadvantages on his site [1]. What we are looking for is an open source application, running on Linux, storing tags in extended attributes (XATTR), not modifying files and performing well.

2.1 User Apps

2.1.1 TMSUs

TMSU [2] is a command-line tool (CLI) that allows assigning tags to files and performing searches by tags. We start by initializing TMSU in the chosen directory. One command lists the tags associated with one or more files and another lists the files that have the given tag(s). TMSU offers the possibility for the user to "mount" a virtual Filesystem (FS) with FUSE (Filesystem in UserSpacE). The tool is fast and efficient, but it has some flaws:

— No GUI.

— Dependency on FUSE to mount the virtual FS.

— Storage of tags in an SQLite database: if the database is lost, the tags are equally.

2.1.2 Tagsistant

Tagsistant [3] is another command-line (CLI) tag management tool. It depends on FUSE and a database (SQLite or MySQL) to work. As for TMSU, it is necessary to give a directory to Tagsistant for its internal use. Inside it, there are different directories:

/

alias –- Directory containing the most common requests archive –- Directory listing relation files –- Directory containing relations between tags and stats files –- Directory containing information on the use of Tagsistant store –- Directory where tags are managed files and added tags tags -- Tag management directory

Each directory has a specific role. Everything is done with the terminal and usual commands (cp, ls, mkdir, etc.). In Tagsistant, a directory created in the tags directory corresponds to a tag. We finally end up with a tree structure of tags and files

[4]. Although this tool is powerful from a speed of execution point of view, it has the shortcomings of TMSU as well as new ones:

— No GUI.

— Dependency on FUSE to mount the virtual FS.

— Storage of tags in a database: if the database is lost, the tags are also.

— Use of the different directories not very intuitive.

— All files are contained in a single directory and their names are modified for the internal needs of the application. Obligation to go through the application to access the files.

2.1.3 TaggedFrog

TaggedFrog [5] is a program available on Windows only and does not share its sources. Its inner workings are not documented. The interface is nice, you can add files by Drag & Drop. The interface gradually creates a "cloud" of tags, as can be found on certain websites. You can perform searches on tags and files. Presumably TaggedFrog maintains a database of tags associated with files, which again does not meet our needs.

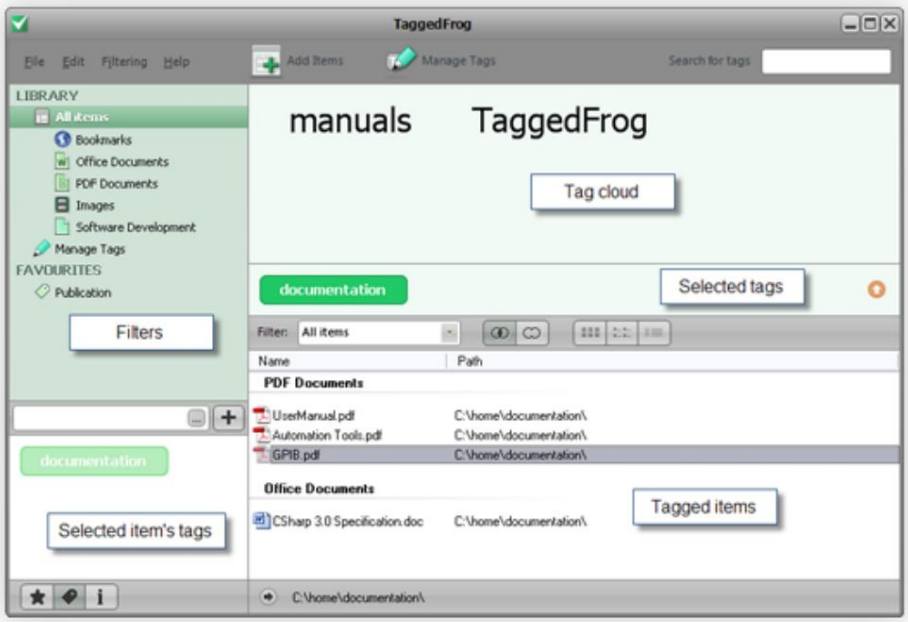


Figure 1 – TaggedFrog in use [5]

2.1.4 TagSpaces

TagSpaces [6] is a program with a graphical interface (GUI) allowing to label its files with tags. The application is pleasant to use, we start by connecting a location that will act as a destination directory for the files. You can add or create files from the application. Existing files added from the application are copied into the directory (thus creating a duplicate). On the left panel is the tag management area. TagSpaces automatically adds certain so-called "intelligent" tags to files newly created with the application (eg a tag with the creation date).

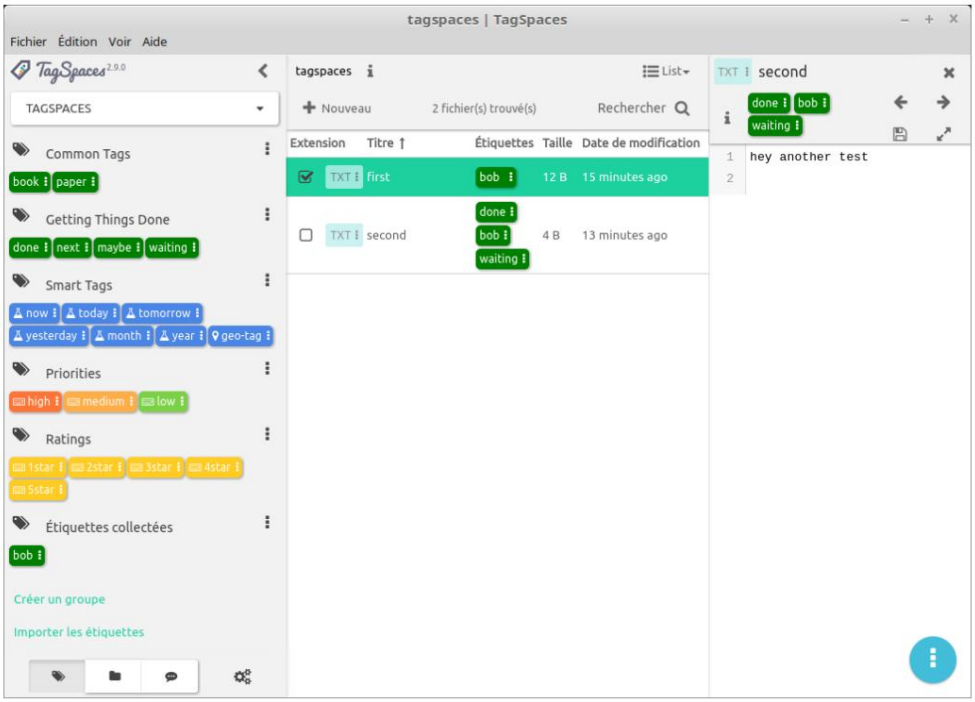


Figure 2 – TagSpaces in use

Overall, the application is functional and user friendly. However, two black points are to be deplored:

1. The application copies the already existing files selected by the user, which creates an additional constraint in the management of his personal files.
2. TagSpaces stores tags directly in the file name, thereby changing its name [7]. Although handy when syncing using a cloud service, the file becomes dependent on TagSpaces. If the user decides to change his name without respecting the internal nomenclature, he risks losing the tags associated with the file.

2.2 Features available in the OS

2.2.1 Windows

Starting with Windows Vista, Microsoft gave users the ability to add metadata to files; among these metadata are the tags. There is a feature called Search Folder which allows to create a virtual directory containing the result of a search on filenames or other criteria [8]. Since Windows 8, the user has the possibility of adding metadata to certain types of files (those of the office suite for example), including tags. He can then perform targeted searches via the Windows file explorer of the meta:value type [9]. It's a shame that Windows doesn't support more file types, like PDFs or .txt files.

2.2.2 macOS

macOS has its own system for labeling files. It has been integrated since version OS X 10.9 Mavericks. From the file explorer, the user has the possibility to add, modify, delete and search for tags. Files can have multiple tags associated with them. A color code makes it easier to remember and visualize the assigned tags.
In the file explorer, the tags are found on the bottom side, to access them more quickly.

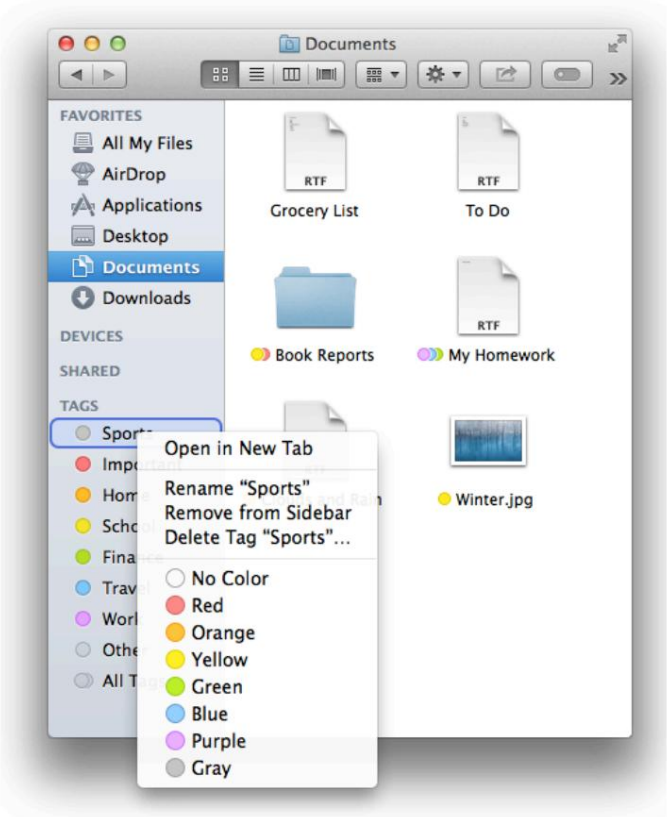


Figure 3 – View and management of a tag in the macOS Finder [10]

When clicking on a tag, a Spotlight search is performed. Spotlight is macOS' internal search engine. Spotlight keeps an index of tags, providing quick access to matching files [11]. All these tags can be synchronized on the different "iDevices" via iCloud. Finally, a settings menu allows the management of tags (display, deletion, etc.) [10], [12]. The implementation of this system uses XATTRs to store tags. The different tags are found in the kMDItemUserTags attribute, listed one after the other. Via the Terminal, using the mdls command, we can display the list of tags associated with a file, named "Hello" for the example:

```
1 % mdls -name kMDItemUserTags Hello 2
kMDItemUserTags = (
3       Green,
4       Red,
5       Essential
6 )
```

Listing 1 - mdls listing file tags on macOS [13]

Here, the Hello file is tagged with three tags, "Green", "Red" and "Essential". The fact that the indexing is carried out with Spotlight implies a re-indexing of the files in the case of a name change for a given tag under macOS. The FSEvents system framework provides a partial solution: it is an API (also used by Spotlight) which offers applications the possibility of being notified if a change has taken place on a directory (one event every 30 seconds). FSEvents maintains logs of these changes in files, so applications can retrieve the history of changes whenever they want [14]. The only negatives of this implementation are that it's not open source, only exists on macOS, and it's not written in Rust. But it is the solution towards which this project tries to get closer because it corresponds to the criteria on the XATTRs and the internal functioning.

3 Architecture

This section presents the software architecture chosen to implement the system. He is composed of four distinct entities, described in the following subsections.

3.1 Tag management

The physical management of tags stored in extended attributes (XATTR) is a feature that is independent of the rest of the system. As explained in section 4.2, system tools exist to manipulate file XATTRs. However, to provide a higher level of abstraction, consistency on tag naming for indexing and more comfort for the end user, a tool becomes necessary for tag management. This tool comes, in its basic form, as a command-line program. It must, at a minimum, offer the possibility of reading the tags contained in the files and adding and deleting the tags given as input by the user. He must be able to handle several tags and files simultaneously. From an algorithmic and data structures point of view, this part is not particularly difficult.

3.2 Indexing files and tags

Indexing files and associated tags is one of the two pillars of the system. An index of relationships between tags and files must be created. The term "index" used here does not take on its exact literary meaning, id is the list of important terms of a book with the list of pages in which they appear. In our situation, it is closer to its use for databases: it is a data structure allowing to quickly find the data, in our case, to quickly recover the relation between tags and files. Two architectures have been imagined for the indexing of tags and files, they are described in the following two subsections. Table 1 lists the events that occur during normal use of the File System (FS) by assigning them a number. These numbers are taken up and used to lighten the reading of tables 2 and 3.

Event Number	
1	Adding a tag to a file or directory
2	Removing a tag from a file or directory
3	Renaming a tag
4	Adding a file to the watched tree
5	Adding a directory to the watched tree
6	Moving or renaming a file in the watched tree
7	Moving or renaming a directory in the monitored tree
8	Deleting a file from the watched tree
9	Deleting a directory from the watched tree

Table 1 - Events occurring on the FS

The following explanations mention the notion of time complexity (the number of operations necessary to accomplish the algorithm, depending on the size of the inputs) with the notation of Landau, or Big O [15].

3.2.1 Indexing with hash table and tree

The first version of the indexing architecture, featuring two donation structures born, is as follows:

1. A hash table (or hashmap) associating a tag (its name, in the form of a character string) to a set (in the mathematical sense) of file paths on disk.
2. A tree, corresponding to the tree structure of files, with the directories, sub-directories and files as nodes. The directory to watch represents the root of the tree. Links between nodes represent the contents of a directory. The node data contains the name of the file or directory and all the tags associated with the file.

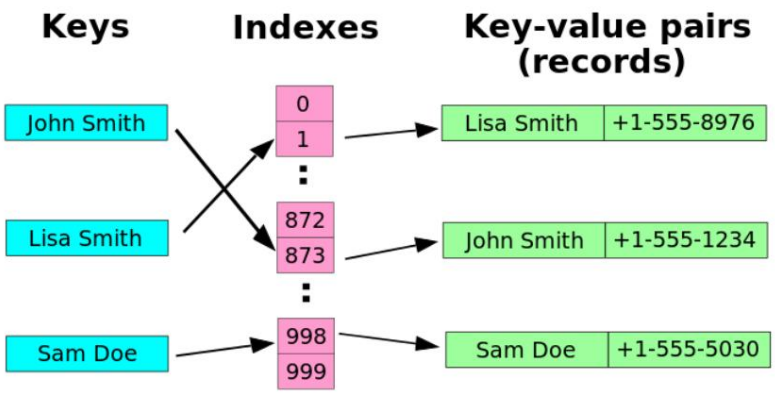
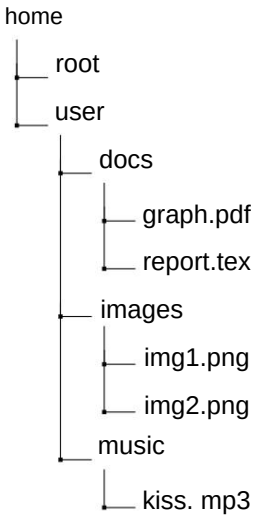


Figure 4 – A directory represented as a hash table - [16]

A hash table is an associative array. The components of the association are the "key", linked to one or more "values". To insert, access or delete an entry from the table, it is necessary to calculate the "hash" of the key, id is its unique hash. In figure 4, we see the keys in blue, the result of the hash in red and the associated values in green. The risk that two or more keys produce the same hash is called a "collision", which is why a good hash table implementation must not only use a good hash function but also a way to solve collisions. This is how the three operations above can be performed, on average, in constant time (O(1)) and in the worst case (if the collisions follow one another) in linear time (O(n)) . In our case, using a hash table to store the relationship between a tag and its files is effective when a search by tags is requested. Moreover, by associating a set of file paths, set operations (union, intersection) can be performed when a search involving several tags is performed.

The tree, in the computer sense, is a representation of the FS hierarchy in our case. Consider the following hierarchy as an example:



It can be obtained using the tree command under Linux for example. The same representation in the form of a tree is illustrated in Figure 5:

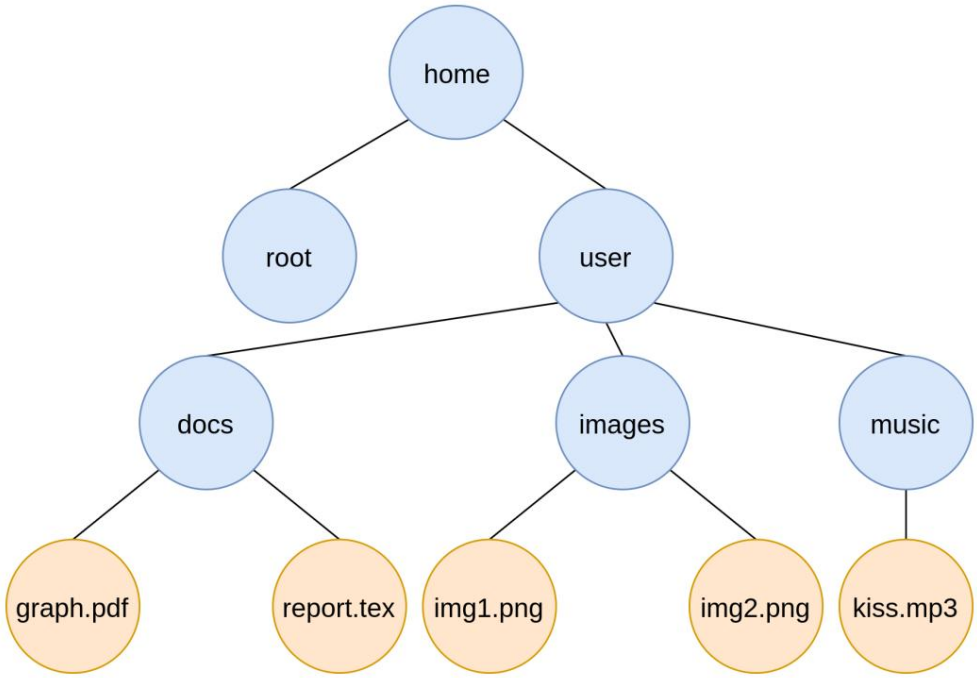


Figure 5 – Tree representation of a hierarchy of files and directories

The "home" node represents the root of the tree. Each node represents either a file (in orange) or a directory (in blue) on disk. Each directory can be seen as a subtree of the main tree. From a programmatic point of view, a node would be defined, at a minimum, as a data structure containing a "data" field (in our case, the name of the file/directory and all of its tags) and a "children" field. , a list or a set of pointers to child nodes. In this case, only the directory nodes point to child directory or file nodes, the files would only have an empty list of pointers.

Table 2 gives, for each event occurring on the FS, the operations to be executed for the two data structures (the hash table and the tree) and an approximation of the complexity. The events are represented by their numbers, in table 1. The following variables are defined:

— c = Constant operation.

— p = Shaft depth.

— t = Name of tags.

Number	Operations on the hashmap	Tree operations
1	If tag not present, add tag as key and add file path to set -> O(c)	Browse the tree in search of the file and add the tag to the set of existing tags -> O(p ÿ c)
2	Remove the file from all file paths associated with the tag -> O(c)	Browse the tree in search of the file and remove the tag from the set of existing tags -> O(p ÿ c)
3	Delete key and re-insert new key and associated set -> O(c)	For the set of paths retrieved with the hashmap, modify the corresponding nodes -> O(t ÿ c)
4	For all file tags, add the tag if necessary and associate the file path -> O(t ÿ c)	Browse the tree in search of the parent directory of the file, add the new node and all of its tags -> O(pÿc)
5	Same as previous line	Browse the tree in search of the parent directory, add the new node and all of its tags, then, recursively, add its children (subdirectories and files) -> ÿ $O(p^2$ ÿ c)
6	For all tags in the file, associate the new file path -> O(t ÿ c)	Browse the tree in search of the parent and change the link of the node with its parent / simple renaming of the name in the label -> O(p ÿ c)
7	For all tags of all subdirectories and files, associate the new file path -> O(tÿ p ÿ c)	Same as previous line
8	For all file tags, remove file path -> O(tÿc)	Traverse the tree in search of the parent and delete the link and the node -> O(p ÿ c)
9	For all tags in all subdirectories and files, remove file path -> O(t ÿ p ÿ c)	Browse the tree in search of the re parent directory, delete the node, all of its tags, and recursively, its children (subdirectories and files) -> ÿ $O(p^2$ ÿ c)

Table 2 – Operations and complexity, first architecture

This version was partly abandoned and adapted for two major reasons:

1. Having two interrelated data structures increases the complexity of update operations (adding, moving, deleting files and tags).

2. The implementation turned out to be more difficult than expected, due to some constraints of Rust (see section 7.2).

3.2.2 Indexing with a graph and a hash table

During the implementation of this part of the program, a new architecture was imagined. It uses the basics of the previous one, but simplifies the data structure. Rather than maintaining two different structures, this solution proposes a main data structure, supported by a secondary structure, optional, but nevertheless effective:

1. A graph, with a node representing either a directory, a file or a tag.
   Each node is a data structure comprising a name and type and is uniquely identified. Thanks to this identifier, the nodes are easily accessible.
2. A hash table, associating the name of a tag with its unique identifier as graph node.

A graph represents a network of nodes that can be linked to each other. Jean François Hêche, professor at the heig-vd, gives the definitions of undirected and directed graphs in his course on "Graphs and Networks". Let's start by defining what an undirected graph is: "An undirected graph is a structure made up of a set V, whose elements are called the vertices or nodes of the graph, and a set E, whose elements are called the edges of the graph, and such that each edge is associated with a pair of vertices of V called the endpoints of the edge." (Ash, page 1, [17]). Figure 6 shows an example of such a graph.

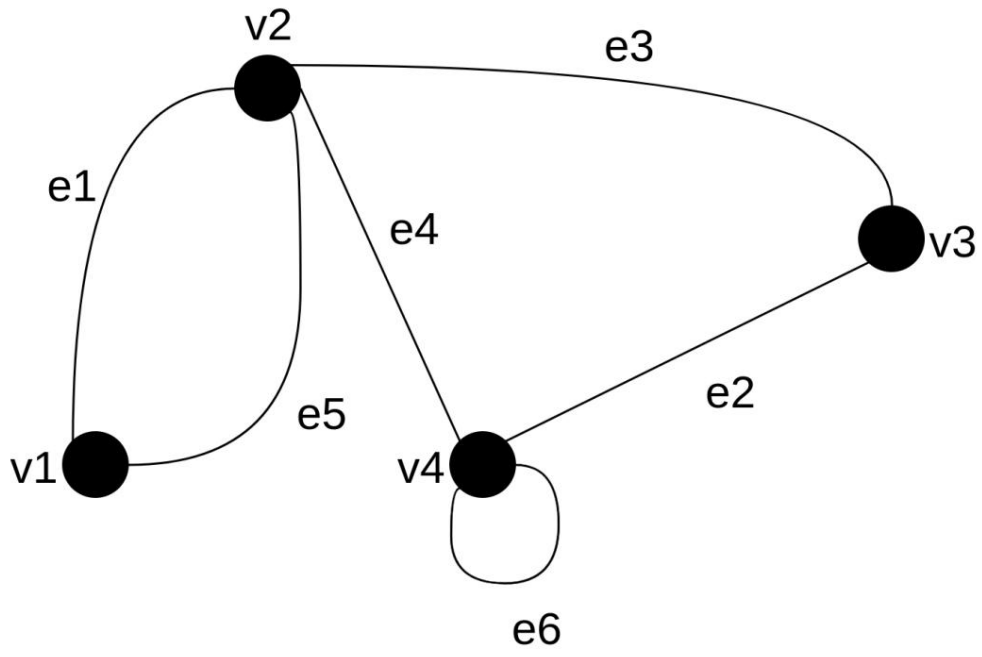


Figure 6 – Undirected graph

A directed graph is similar to an undirected graph. The only difference is that a direction is given to the link between two nodes and this link is no longer called "edge" but "arc".
Figure 7 shows an example of such a graph.

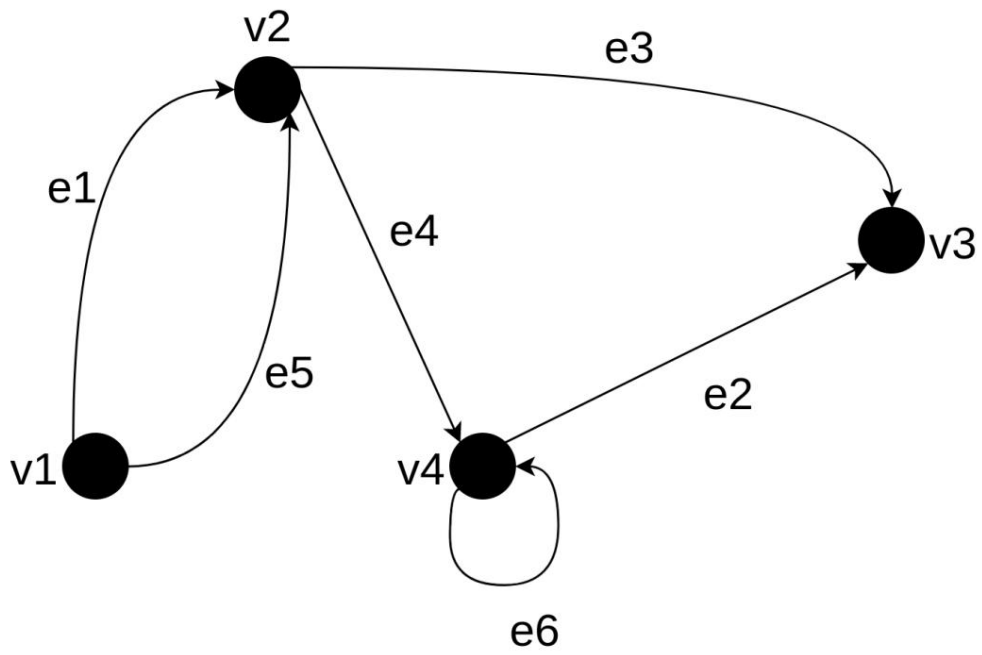


Figure 7 – Directed graph

A graph is therefore a set of nodes connected by edges or arcs, depending on whether the graph is oriented or not. In our case, the use of a graph is not so far from that of a tree. Moreover, according to graph theory, "a tree is a cycle-free and connected graph" (Hêche, page 33, [17]). "Without cycle" means that a traversal of the graph is possible in such a way that the starting and ending nodes are different. "Connected" defines a graph such that for each pair of nodes in the graph there is a path connecting them.
The use of such a graph faithfully represents the tree structure of the FS (the schema of a tree is kept) and greatly simplifies the operations when events occur on the latter. Adding the tags as graph nodes maintains a single consistent data structure and decreases the number of different operations required when updating the FS. The traversal of this graph is done according to the given file path, keeping the unique identifier of the node corresponding to the root directory, the traversal is done from the root to the final node of the file path.

The hash table used in this version can be seen as a "cache" for accessing tag nodes. Indeed, we could do without this hash table and when access to a tag is requested, search the entire graph for the tag in question. However, this last operation quickly becomes significant when the graph has a large number of nodes. In addition, this hashmap is accessed much less often than in the first version of the architecture, because it is updated only during operations on tags and no longer on those linked only to files and directories (potentially heavier operations) .

As for subsection 3.2.1, table 3 gives for each event occurring on the FS, the operations to be executed for the two data structures (the graph and the hash table) and an approximation of the complexity. The events are represented by their numbers, in table 1. The following variables are defined:

— c = Constant operation.

— p = Depth of the graph.

— t = Name of tags.

We can see that the operations on the hash table are few and often optional, which results in a gain on the number of total operations.

Number	Operation on the graph	Operation on the hashmap
1	Browse the graph looking for the file, if necessary create the tag node, and link the file node to the tag node -> O(p ÿ c)	If not existing, add the name of the tag as a key and its identifier in the graph as theirs -> O(c)
2	Browse the graph in search of the file node and remove the link between tag node and file. If the tag node is not connected to any other node, delete it -> O(p ÿ c)	If the tag node is not connected to any other node, delete the entry -> O(c)
3	Obtain the identifier thanks to the hashmap and rename the corresponding node -> O(c)	Delete the entry and recreate one with the new name and same identifier -> O(c)
4	Browse the graph in search of the parent directory, add the new node. For existing tags, link the new node, if not create the corresponding new tag node -> O(p ÿ t ÿ c)	For each tag, operation identical to number 1
5	Browse the graph in search of the parent directory, add the new node. For existing tags, link the new node, if not create the corresponding new tag node. Repeat for the subtree -> ÿ O(p ^{2} ÿ t ÿ c) Traverse the graph looking for the parent and change the link of the node with its parent /	For each tag, operation identical to number 1
6	simple renaming of the name in the label -> O(p ÿ c)	No operation required
7	Identical to number 6	No operation required
8	Browse the graph in search of the file node and remove the links between tag nodes and parent node -> O(p ÿ t ÿ c)	For each tag in the file, delete the tag node if it no longer has links to other nodes -> O(t ÿ c)
9	Browse the graph in search of the directory node and remove the links between tag nodes and parent node. Repeat for subtree -> ÿ O(p ^{2} ÿ t ÿ c)	For each sub-directory or sub-file, operation identical to number 8

Table 3 – Operations and complexity, second architecture

3.3 FS Monitoring

Monitoring of FS and associated tags is the second pillar of the system. Initial indexing is mandatory, but it is also necessary to constantly monitor the file tree to keep this index up to date. To achieve this, we will use inotify (see section 4.3), paying particular attention to the following events:

— IN_ATTRIB: change on tags (addition, deletion, renaming).

— IN_CREATE: creation of file/directory in the monitored directory. Add a new monitoring if directory.

— IN_DELETE: deletion of a file/directory in the monitored directory.

— IN_DELETE_SELF: deletion of the monitored directory.

— IN_MOVE_SELF: deletion of a file/directory in the monitored directory.

— IN_MOVE_FROM: moving/renaming the directory (old name).

— IN_MOVE_TO: moving/renaming the directory (new name).

One thread takes care of listening to the events of the FS and writes them in a buffer while another will update the graph to reflect the changes that have occurred by reading in this same buffer (simple producer-consumer pattern).

3.4 Requests for tags and files

Once FS monitoring is in place, the system should be able to respond to user queries. Through a command line tool, the user has the possibility to:

— Request the list of files and directories associated with one or more tags. The query can be in the form of a simple logical expression (with the logical operators "and" and "or").

— Request the list of existing tags.

— Renaming a tag.

To exchange requests and responses, client and server communicate by sockets, with a very simple protocol format (described in table 4) to distinguish the type of a request.

4 Technological analysis

4.1 Rust

This section introduces the Rust programming language and some of its mechanisms, through some examples, which are either absolutely necessary to start programming with Rust, or used in the code of this project. Rust is a strongly typed, compiled, high-performance, multi-paradigm language. It can be used, among other things, for system-oriented programming, to create command-line programs (CLI) or to create web applications. With an active community, many packages and modules are available on Crates.io [18] and many discussions are present on the reddit [19] dedicated. For more details, the excellent book [20] produced by the maintainers of Rust will be able to give more detailed and precise information to the reader eager to know about Rust. Another book [21], more specialized, guides the beginner in Rust in the design of linked lists, because not trivial in Rust because of its constraints.

4.1.1 Installation

Installing Rust on Linux and macOS is very simple. Prerequisites are a C compiler (some Rust libraries require one) and the data transfer tool curl. Then just open a terminal and enter the commands in Listing 2. The second command is optional and only necessary if you want to use Rust without logging out of the shell
fluent.

```
1 $ curl https://sh.rustup.rs -sSf | sh 2 $ source
$HOME/.cargo/env
```

Listing 2 – Installing Rust on Linux or macOS

On Windows, the procedure is a little longer but just as simple, make sure you have the C++ build tools for Visual Studio 2013 or higher. Rust installs its compiler, rustc, which allows you to compile a source file (.rs) into an executable file. However, we won't talk about it any further, the compilation will be done with the Cargo package and compilation manager (see subsection 4.1.2). For more details, refer to chapter 1.1 of the book [20]. Are we (I)DE yet? [22] gives an overview of text editors compatible with the Rust development chain. Regarding the bachelor project presented here, all the code was written with Visual Studio Code.

4.1.2 Cargo and Crates.io

Cargo is Rust's built-in build-and-run system and package manager. From the terminal, its main commands allow you to create a new pro

jet (cargo new myproject), to compile it (cargo build), to run it (cargo run) or to generate the associated documentation (cargo doc). When a new project is created with Cargo, a Cargo.toml file is generated (like the package.json file with Node.js and npm) with the following minimal content:

```
1 [package] 2
name = "myproject"
3 version = "0.1.0"
4 authors = ["Firstname Lastname <me@mail.com>"]
5
6 [dependencies]
```

Listing 3 – Contents of the Cargo.toml file

The "package" section contains information about the project itself. The "depen dencies" section lists the packages that our application depends on, called "crates" by the Rust community. Thousands of crates are available on Crates.io [18]. Other sections can be added to the Cargo.toml file to customize compilation commands, create workspaces from several crates or add specific commands for example. A subchapter (1.3) and a whole chapter (14) are dedicated to Cargo in the Rust book [20] and it also has full documentation [23] (like the book dedicated to Rust).

4.1.3 General

Comments Any text written after two consecutive slashes "//" is considered a comment in Rust. The multiline syntax that exists in C for example is not taken into account. By putting three consecutive slashes "///", the commenter tells the compiler that this comment is part of the documentation (which can be generated with Cargo, see subsection 4.1.2).

Variables First of all, to declare a variable in Rust, you have to use the let keyword .
A variable is by default declared immutable, id is that it cannot be modified in the rest of the code. Although Rust is a strongly typed language, its compiler can in most cases infer the correct type of the variable, either by analyzing the assigned value or by analyzing the first use of the variable (arguments of a function, insertion of data in the case of collections). However, there is the possibility of explicitly declaring the type of the variable by indicating it before the "=". To declare a variable mutable, you must add the keyword mut before its name. Then, the constants are declared with the const keyword and their type must be indicated. The main difference between constants and immutable variables is that a constant cannot be the result of a value

calculated when the program is run. Finally, a variable can be "hidden" or "shadowed": a new declaration with let and the same name overwrites the previous value and can be of a different type. Listing 4 shows some cases of variable declarations:

```
1 // Declaration of a variable "x", immutable and of inferred type i32
2 let x = 3;
3 // Declaration of a variable "y", mutable and of inferred type bool 4 let mut y = true;

5 // Declaration of a variable "z", mutable and of type declared char
6 let mut z : char = 'A';
7 // The 64-bit floating-point PI constant 8 const PI: f64 =
3.1415; 9 // Shadowing. The first declaration creates a
variable named "answer" 10 // of type i32 and value 42, while the second overwrites the previous
variable 11 // by taking its name and is of type String.

12 let answer = 42;
13 let answer = answer.to_string();
```

Listing 4 - Examples of variable declarations in Rust

For more details, refer to chapter 3.1 of the book [20].

Types There are two families of types in Rust:

— Scalars: integers, floating point numbers, booleans and characters.

— Compounds (two primitive types): tuples and arrays.

Integers can be signed or unsigned and on 8, 16, 32, 64 bits or depending on the architecture of the processor (i8, u8, i16, u16 , i32 , u32, i64, u64, isize, usize).
Decimal numbers have two possibilities, either on 32 bits or on 64 bits (f32 or f64). The classic bool type can take two values, true or false. Finally, the last scalar primitive type, char, stores a Unicode character between single quotes. The first compound primitive type is the tuple. It is a grouping of several values which can be of different types. When declaring, variable names, types, and values of a tuple are enclosed in parentheses and separated by commas. Finally, the array type, or array: classic type grouping together several values of the same type this time. An array has a fixed size, determined at compile time. The declaration of the values of an array is done between square brackets "[ ]". To access a value of the array, use the syntax array[i] where i is a valid index of the array (between zero included and the size of the array not included). Some examples are given in Listing 5.

```
1 let myint: i32 = 1234 ; 2 let
mychar: char = 'a'; 3 let myfloat: f64
= 2.0; 4 // Declaration of a tuple 5 let
tuple: (char, u32, f64) = ('c', 42, 2.8);
6 // Destructuring the tuple into three distinct variables 7 let (letter,
age, score) = tuple; 8 // Declaration of an array


9 let myarray = ['a', 'b', 'c', 'd', 'e', 'f']; 10 let x = myarray[4]; // x is
'e'
```

Listing 5 - Some Rust Primitive Types

For more details, refer to chapter 3.2 of the book [20].

Functions As in C, any program has the main() function as its entry point. A function declaration begins with the fn keyword , followed by the name of the function, a list of any parameters, and any return types. When a function has a return value, the last line of the function that does not have a semicolon at its end is evaluated as an expression and returned. The return keyword nevertheless exists if the function must return in very specific cases (in a condition for example). Variables declared inside the function are not accessible from outside the function. Function arguments are passed by copy by default (see subsection 4.1.8 for details). The following listing gives an example of the same function, in two more or less short versions.

These functions expect two integers and also return an integer.

```
1 fn x_plus_y_plus_one(x : i32, y : i32) -> i32 {
2         let x_plus_y = x + y;
3         x_plus_y + 1
4 }
5
6 fn x_plus_y_plus_one_short(x : i32, y : i32) -> i32 {
7         x + y + 1
8 }
```

Listing 6 – Examples of functions in Rust

For more details, refer to chapter 3.3 of the book [20].

Control structures Like any programming language, Rust has control structures to manage conditions and repetitions (loops). There is the classic if condition { ... } else if other_condition { ... } else { ... } with a notable difference compared to C: it is possible to affect a variable with an if, as in Listing 7. There is no switch ... case per se in Rust, we will see the match ... case in subsection 4.1.6. As far as loops are concerned, there are three of them: loop (infinite loop), while (loop with initial condition) and for, loop to traverse collections mainly by their iterators (see subsection 4.1.7).

```
1 // Example of a simple if ... else
2 let name = "fred";
3 if name == "fred" {
4         println!("Hello buddy !");
5 }
6 else {
7         println!("Hello World !");
8 }
9
10 // Example of assigning a variable with an if. Here, n will be 42
11 let condition = true;
12 let n = if condition { 42 }
13 else { 66 };
```

Listing 7 - Examples of conditions in Rust

```
1 // Infinite loop
2 loop
3     { println!("Forever");
4 }
5 // Loop with condition
6 let mut x = 0;
7 while x < 10 {
8         println!("{}", x);
9         x = x + 1;
10 }
11 // Path of an array 12 let myarray =
[1, 2, 3, 4, 5]; 13 for elem in myarray.iter()
{ println!("value : {}", elem);
14
15 }
```

Listing 8 - Examples of loops in Rust

For more details, refer to chapter 3.5 of the book [20].

Organization of files and modules A program written in Rust can be split into several files and modules. A module can contain declarations of functions, structures and their implementations (see section 4.1.4), etc. The keyword for declaring a module is mod. The code inside the module is by default private, to make it accessible outside the module, the pub prefix is available. To use a module within another or in main.rs, it must be imported with the use keyword . By default, any module is defined in a project's src/ lib.rs file. If many modules are declared, it is possible to mention them in src/lib.rs in this way: mod mymodule; and create a file with the same name as the module (in this example, src/mymodule.rs) containing the code in question. A module declaration can contain others as well, creating a hierarchy of modules. For more details, refer to chapter 7 of the book [20].

4.1.4 Data structures

As in C, Rust grants the possibility to the programmer to define his own compound types, the structs. Declaring and instantiating a structure is done like in C with some shortcuts available. A structure without field names is also available, called "tuple struct". To access the fields of a structure, just use the dot notation (player.name).

```
1 // Structure defining a character in a video game 2 struct Player { name: String,
class: String,
3
4
5           life: i32,
6           active: bool
7 }
8 // Création d'une variable Player 9 let player_one
= Player { name: String::from("Groumf"), class:
10          String::from("Wizard"),
11
12          life: 100,
13          active: true
14 };
15
16 // Structure sans noms aux champs 17 struct
Coordinates(f64, f64); 18 let geneva =
Coordinates(46.2016, 6.146);
19
20 // Structure vide ()
21 struct Nil;
```

Listing 9 - Examples of Structures in Rust

The particularity of structures, compared to C, is that it is possible to define methods attached to structures, like methods in Java, without obtaining a class stricto sensu of object-oriented languages, even if the final result is very similar.

To define methods to a structure, it is necessary to declare a block of code starting with the keyword impl followed by the name of the structure and braces. Inside this block are defined functions related to the structure. In Listing 10, we see the declaration of the Player framework and its implementation, with three methods (with function syntax) for creating a new character, for it to attack another, and for it to greet. The only difference between a method and a function is that a method which is called on a variable of the type of the structure with the dotted notation expects the parameter self as the first parameter, obligatorily. self refers to the variable itself, like this in Java. We can see that self and the other parameters have a syntax not yet described (& and &mut), subsection 4.1.8 gives further explanation.

```
struct Player {
    name: String, class: String, life: i32, force: i32
}

impl Player { fn
    new(name : String, class : String) -> Player {
        Player { name, class, life : 100, force : 10 }
    }

    fn attack(&self, other : &mut Player) {
        other.life = other.life - self.force;
    }

    fn say_hi(&self) { println!
        ("Hi, I'm {}, powerfull {} !", self.name, self.class);
    }
}

fn main() {
    let player_one = Player::new(String::from("Groumf"),
        String::from("Wizard")); let mut
    player_two = Player::new(String::from("Trabi"),
        String::from("Bard"));
    player_one.attack(&mut player_two);
    player_one.say_hi();
}
```

Listing 10 – Block impl of a structure in Rust

For more details, refer to chapter 5 of the book [20].

4.1.5 Traits and genericity

Traits in Rust are the equivalent of interfaces in Java. It is a way of defining an abstract behavior that a type could follow. Listing 11 defines a "vehicle" trait (Vehicle) that implement the "bicycle" (Bicycle) and "plane" (Plane) structures. The Vehicle trait gives the signature of a single function, description() which describes the variable of the type in question. Both structs implementing the trait must also implement all the functions of the trait.

```
1 trait Vehicle { fn description(&self); }
2
3 struct Bicycle { wheels : u8, passengers : u8 } 4 impl Vehicle for
Bicycle { fn description(&self) {
5
6                println!("I'm a bicycle, I have {} wheels and \ can carry {}
7                    passengers.", self.wheels, self.passengers);
8
9            }
10 }
11
12 struct Plane { engines : u8, passengers : u16, fuel : String } 13 impl Vehicle for Plane { fn
description(&self) {
14
15               println!("I'm a plane, I have {} engines and \ can carry {}
16                   passengers. I fly with {}.", self.engines, self.passengers,
17                   self.fuel);
18           }
19 }
20
21 fn main() {
22           let bicycle = Bicycle { wheels : 2, passengers : 1 }; bicycle.description();
23           let plane = Plane { engines : 1, passengers : 100, fuel :
24           String::from("kerosene") };
25
26           plane.description();
27 }
```

Listing 11 - Rust Trait Implementations

Traits can be declared generic, just like functions. Genericity is not a concept reserved for Rust, many programming languages use it.

This is a way to avoid repeating the same code for different types of data, but which would have similarities. Let's take the example of a function that adds two numbers. Arguments could be either integers or floating point numbers.

The method for adding these two types is the same. But for a strongly typed language like Rust, it is necessary to precisely define the types of function arguments.

This is where genericity comes in: the declaration of a function expects a generic type, usually named T, and handles it like a real type. Many types of bookstore

standard are generic, like the Option and Result types (see subsection 4.1.6). For more details, refer to chapter 10 of the book [20].

4.1.6 Enumerations and pattern matching

Enumerations are another way to design your own types. As in C, an enumeration lists all the possible variants of a value of the same type. The classic example of an enumeration are the days of the week. Seven different cases, without possible evolutions. Enumerations in Rust take on their full meaning in combination with pattern matching: a control structure resembling a switch ... box in C but with a side borrowed more from functional programming (we find them elsewhere in Scala). All possible cases of an enumeration must be treated with a match (hence the default clause _). The block of code to the right of each => can be returned, like a function (see Listing 12).

```
1 enum Direction {
2         North, South, East, West
3 }
4
5 fn print_direction(direction : Direction) { match direction {
6
7             Direction::North => println!("Go North"), Direction::South
8             => println!("Go South"), => println!("Go East or West") //
9             _  clause par défaut
10        }
11 }
```

Listing 12 – Defining an enum and using it with pattern matching in Rust

Rust has a very powerful enumeration in the standard library: Option (recopied in Listing 13). It replaces the infamous NULL in C or other languages. It simply removes countless bugs often encountered due to NULL. If a variable exists, it ends up in the Some case of the option, if it does not exist, in the None case. This enumeration is also generic (see subsection 4.1.5), it accepts any type of data.

```
enum Option<T> {
        Some(T),
        None,
}
fn process(value : Option<u32>) {
        match value {
                Some(data) => println!("{}", data), None =>
                println!("Error, no data")
        }
}
```

Listing 13 – The Option enumeration and its use with pattern matching in Rust

For more details, refer to chapter 6 of the book [20].

4.1.7 Collections

This subsection describes the two most used collections in Rust, namely vectors (Vec) and associative hash tables (HashMap). A vector is an array that has no fixed size, elements can be added or removed from it. It is the equivalent of ArrayList in Java. A vector is generic (see subsection 4.1.5), it can contain any type of data, but only one type at a time. Many methods exist to manipulate a vector, either to add or remove elements from it or to convert it to other forms. A macro, vec! is available to quickly create a vector (comma separated elements between "[ ]"). Listing 14 gives some examples:

```
// Declaration of a vector with new, then with the macro let v: Vec<char> =
Vec::new(); let mut v = vec!['a', 'b', 'c']; // Add element to vector v.push('d');
// Access to the second element of the vector, in two different ways let second:
&char = &v[1]; let second: Option<&char> = v.get(1); // (immutable) traversal
of the vector with the syntax for .. in ..
for i in &v {
        println!("{}", i);
}
```

Listing 14 – Examples of declarations and uses of a vector

A HashMap is an associative hash table, such as exists in Java. It is, like the vector, generic (see subsection 4.1.5), it accepts any type of data. It is an efficient data structure for quickly accessing information. Listing 15 gives some examples:

```
1 // Declaration of a hashmap with new 2 let h:
HashMap<char, u32> = HashMap::new(); 3 // Add key-value
pair to hashmap 4 h.insert('a', 97); 5 // Access to the value
associated with the key 'a', returns an Option 6 let a:
Option<&u32> = h.get('a'); 7 // (immutable) traversal of the hashmap with the syntax for ..
in .. 8 for (key, value) in &h { println!("{}: {}", key, value);
9
10 }
```

Listing 15 – Examples of declaration and use of a HashMap

For more details, refer to chapter 8 of the book [20].

4.1.8 Ownership, Borrowing and References

The unique feature of Rust is undoubtedly ownership, or "possession".
Ownership is defined by these three rules:

— Each variable is said to be the "owner" of a value.

— There can only be one owner for a value.

— When the owner is destroyed or changes range, the value is destroyed.

Before continuing on this notion, a brief reminder on the use of memory is necessary. The operating system (OS) provides a program with two different memory areas to store its variables: the stack (stack) and the heap (heap). The stack access mechanism is simple and fast and stores variables whose size is fixed and known at compile time. Primitive type variables (see paragraph 4.1.3) are stored in the stack.
When a variable whose size is not known in advance (high type, for example collections) is declared, it is placed in the heap. A program wanting to store a variable on the heap is obliged to ask the OS to find an available space large enough to store the variable and to give it a pointer to this area, to find it later.

When a value is assigned to a variable, it is held by that variable until it is destruction. Listing 16 illustrates an example:

```
1 {
2         // my_vec is the "owner" of this vector let my_vec = vec![3, 2,
3         1]; ... // my_vec is used
4
5 } // the scope of my_vec ends here, my_vec is then deleted
```

Listing 16 - Scope a variable in Rust

The my_vec variable is on the stack, but the data it points to is on the heap. When my_vec goes out of range, the data in the stack and in the heap will be freed. If my_vec is assigned to another variable, the ownership of the data is transferred to this new variable, my_vec will no longer be accessible and usable after the assignment, as shown in Listing 17. This scenario does not occur with types primitives whose size in memory is known at compile time and where a copy of the data is made. For an advanced type to be copied in this way, it must implement the Copy trait.

```
1 {
2         // No problem here, a and b are of i32 primitive type, of // fixed and known size, the
3         value of a is copied into b.
4         let a = 10;
5         let b = a;
6
7         let mut my_vec = vec![3, 2, 1]; let other_vec
8         = my_vec; my_vec.push(42); // Error, the
9         value has been moved (move)
10 }
```

Listing 17 – Transfer of ownership and Rust

In line 8 of Listing 17, the variable my_vec is invalidated, but not the data to which it points. Only the pointer to this data is transferred to other_vec.
This is where the second rule of ownership applies, there cannot be more than one owner of the same value at the same time. To make a real copy of the data from one vector to another, or in general for an advanced type, you must call the clone() method, which copies the data entirely in memory. This ownership transfer situation also occurs during calls to functions. If a parameter is given to a function, the function takes ownership of it, as shown in Listing 18.

```
fn main() {
    let mut my_vec = vec![3, 2, 1];
    print_vec(my_vec); // The function takes possession of the vector my_vec.push(42); //
    Error, the value has been moved (move)
}

fn print_vec(v : Vec<i32>) { println!("My
    super vector : {:?}", v);
}
```

Listing 18 - Transferring ownership to a function in Rust

To overcome this problem of transfer of ownership during calls to functions, two solutions existent :

1. The function must return the possessed value. This is impractical if the purpose of the function is to return the result of an operation. It can return a tuple made up of the grabbed value(s) and the result of its operation. This way of doing things is cumbersome and not recommended.
2. The function can "borrow" the variable in different ways (see below).

Fortunately, functions can "lend" variables to each other by borrowing.
Two types of variable references are available:

1. Les références immutables (syntax &ma_variable).
2. Mutable references (syntax &mut my_variable).

Listing 19 shows an example of immutable and mutable borrowing.

```
fn main() {
    let mut my_vec = vec![3, 2, 1]; // The function
    immutably borrows the vector ref_immutable(&my_vec); // The function mutable
    borrows the vector ref_mutable(&mut my_vec);



}

fn ref_immutable(v : &Vec<i32>) {
    println!("My super vector : {:?}", v);
}

fn ref_mutable(v : &mut Vec<i32>) {
    v.push(42);
}
```

Listing 19 – Borrowing variables between functions in Rust

These references are conceptually close to pointers in C (Rust also accepts the dereferencing of variables with the symbol *) but obey two fundamental rules: 1. At any time, there can be either a single mutable reference, or several references immutable, but not both at the same time.

2. References must always be valid.

Thanks to references, Rust avoids concurrency problems on pointed values as well as invalid pointers. For more details, refer to chapter 4 of the book [20]. There are also other types of pointers, called "intelligent". The book on Rust dedicates an entire chapter to them (chapter 15 of the book [20]).

4.1.9 Error handling

Notwithstanding its very restrictive compiler, which detects many compile-time errors, Rust has advanced run-time error handling. It distinguishes two types: recoverable errors and unrecoverable errors (not like in languages like Java where the concept of exceptions mixes these two types of errors). For the latter, Rust provides a macro, panic!, which abruptly stops the program and prints an error message to standard output indicating at which line the program crashed. For recoverable errors, a more elegant way exists: like the Option enum seen in Section 4.1.6, the Result enum was designed to handle runtime errors (see Listing 20).

```
enum Result<T, E> {
        Ok(T),
        Err(E),
}

fn process(value : Result<u32, std::io::Error>) {
        match value {
                Ok(data) => println!("u32 value : {}", data), Err(error) => println!
                ("Error : {}", error)
        }
}
```

Listing 20 – The Result enumeration and its use with pattern matching in Rust

Many functions in the standard library and in the community return Result, especially when opening a file. As these kinds of operations are common, Rust provides two functions equivalent to performing a match on a Result, unwrap() and expect(). The only difference between the two is that with the second function, a custom message is expected as an argument. Listing 21 shows the different ways to process a Result when opening a file:

```
fn main() {
        // Version 1 let f
        = File::open("test.txt"); let f = match f { Ok(file)
        => file,

                Err(error) => {
                        panic!("Error on opening file : {:?}", error)
                },
        };
        // Version 2
        let f = File::open("test.txt").unwrap(); // Version 3

        let f = File::open("test.txt").expect("Error on opening test.txt");
}
```

Listing 21 - Opening a file and processing it in Rust

For more details, refer to chapter 9 of the book [20].

4.1.10 Tests

When a new library project is created with cargo new mylib --lib, a unit tests is de facto added to the lib.rs file, as in Listing 22:

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);
    }
}
```

Listing 22 - Test module added automatically

Each function preceded by the #[test] attribute is considered by the compiler as a test. To run the tests, Cargo provides a command, cargo test. When this command is executed, all tests are performed in parallel, without guaranteeing a predefined execution order. A summary of the tests performed, passed, and failed is printed to standard output when the command completes. To perform our tests, Rust provides some
macros :

— assert! : expects a bool type argument.

— assert_eq! : expects two arguments of the same type, to check that they are equal.

— assert_ne! : inverse of the previous one, checks the inequality of the two given arguments.

There is also another attribute, #[should_panic], for performing functions that test if a piece of code should "panic", thus failing. Finally, the killer feature of testing in Rust is that the cargo test command also checks sample code in our code documentation, to keep documentation up to date with our code. For more details, refer to chapter 11 of the book [20].

4.1.11 Concurrence et threads

Rust provides an implementation of threads in its standard library. A thread in Rust corresponds to a system thread. Listing 23 gives an example of creating a thread. The spawn() method returns a handler to terminate the thread properly with the join() method.

```
use std::thread;

fn main() {
    let handle = thread::spawn(|| {
        println!("Hello from thread !");
    });

    println!("Hello from main !");
    handle.join().unwrap();
}
```

Listing 23 - Creating a Thread in Rust

A thread can take ownership of a variable if the move keyword is added to the spawn() call. The variable is no longer available in the function that called the thread. Listing 24 shows an example of this situation.

```
use std::thread;

fn main() {
    let my_vec = vec!['a', 'b', 'c']; let handle =
    thread::spawn(move || {
        println!("{:?}", my_thing);
    });
    handle.join().unwrap();
}
```

Listing 24 - Creating a thread in Rust and passing a variable

To communicate between threads, a message mechanism is available. The threads communicate through a channel according to the Multiple Producer, Single Consumer topology ("multiple producer, single consumer", see figure 8): it is possible that several threads send (produce) messages in the channel but only one thread can receive (consume). Listing 25 gives an example.

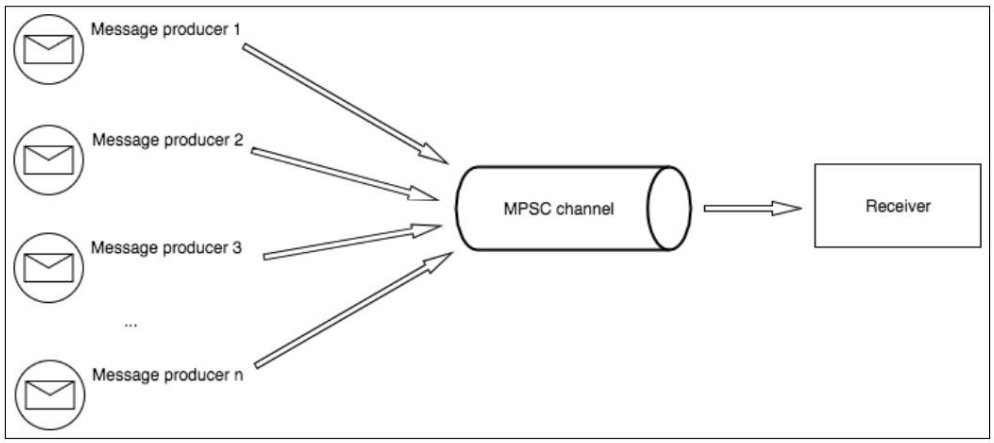


Figure 8 – Canal de communication entre threads - [24]

```rust
use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();

    let new_tx = mpsc::Sender::clone(&tx);
    thread::spawn(move || { let val = String::from("Hello
        from first thread"); new_tx.send(val).unwrap();

    });

    thread::spawn(move || { let val
        = String::from("Hello from second thread"); tx.send(val).unwrap();

    });

    for received in rx {
        println!("Message from threads: {}", received);
    }
}
```

Listing 25 – Message passing with two producers and one consumer in Rust

Lower level primitives (Mutex) as well as traits are available to manipulate Rust threads more finely. For more details, refer to chapter 16 of the book [20].

An article illustrating the use of threads in Rust by creating a simple hash calculation program gives a good basis for building your own program [24].

4.1.12 Unsafe Rust

The Rust compiler is not only very verbose on errors but also very restrictive. This constraint is the guarantee for the programmer to obtain safe and reliable code on memory management. However, there are times when code must violate Rust's memory rules and best practices. This is particularly the case for low-level code (kernel manipulation for example). For these scenarios, Rust can run in unsafe mode. Any block of code that needs to be unsafe must be prefixed with the unsafe keyword . It is then the responsibility of the programmer to check that the code is not buggy or dangerous. For more details, refer to chapter 19.1 of the book [20].

4.2 Extended attributes

4.2.1 Operation of XATTRs

Extended attributes (XATTR), or "extended attributes" in French, are a way to attach metadata to files and folders in the form of space.name:value pairs.
The namespace, or namespace in English, defines the different classes of attributes. As part of this project, the focus is on the ext4 [25] file system (FS) on Linux, there are currently four namespaces or classes: user, trusted, security and system.
The space that interests us is user. This is where the user or the application, provided it has the usual UNIX rights on the files, can manipulate the XATTRs. The other three namespaces are used, among other things, for ACL access lists (system), kernel security modules (security) or by root (trusted) [26] [27]. The name is a string and the value can be a string or binary data. XATTRs are stored in files. Many FS manage their use: ext2-3-4, XFS, Btrfs, UFS1-2, NTFS, HFS+, ZFS. These FS are used by the four most popular OS: Windows, macOS, Linux and FreeBSD. Windows uses XATTR notably in its management of Unix permissions in the Linux shell integrated into Windows 10 [28]. macOS, as seen in section 2.2.2, uses them among other things in its tag management system. The xattr command allows you to manipulate them. Under Linux, there are three: attr, getfattr and setfattr. Under Linux with ext2-3-4, each attribute has a data block (1024, 2048 or 4096 bytes) [27]. Apple and freedesktop.org advocate reverse DNS notation for naming attributes [29], [30] because any process can modify attributes in user space. Prefixing the name of the program to the name of the attribute, for example user.myprogram.myattribute, reduces the risk of another application using the same attribute name. Unfortunately, most Linux CLI tools for manipulating files like cp, tar, etc. do not take into account the attributes with their default syntax, it is necessary to specify additional arguments [31].

4.2.2 Behavior during common access operations

To verify the portability of the XATTRs, some tests were carried out between an SSD acting as a Linux Mint 18.2 Sonya system disk with two USB keys (8 and 64 GB) and a network location mounted in NFS. Listing 26 shows the output of the df command, which returns the usage of the various storage locations, in order: the system disk, in ext4, the 8 GB key formatted once in FAT32, then again in NTFS, the 64 GB key formatted in ext4 and finally a virtual machine under Debian 9 mounted in NFS.

```
1 Sys. of files 2 /dev/sda2          Type Size Used Avail Uti% Mounted on
                                     ext4        451G      334G 96G 78% /
3 /dev/sdg1  4 /                     vfat        7.7g      4.0K 7.7G 1% /media/pc/cle1 41M 7.7G 1% /
dev/sdg1  5 /dev/                    fuseblock 7.7g        media/pc/cle1 33G 23G 59% /media/pc/
sdg1  6                              ext4          59G     cle2
192.168.1.21:/home/user nfs4                       916G    198G 673G 23% /mnt/debian
```

Listing 26 – Output of df -Th: the system disk, the USB keys and the NFS

The approach is as follows: an XATTR in the user space with the name author and the value steven is added to the file file.txt with attr. This file is copied with cp taking care to preserve the attribute (option --preserve=xattr). Once copied, we try to read the same attribute, always with attr. The results can be seen in the 27 listings,
28, 29 and 30:

```
45  ~ $ attr -s author -V steven file.txt
46 The "author" attribute set to a value of 6 bytes for file.txt:
47 steven
48  ~ $ cp --preserve=xattr file.txt /media/pc/cle1
49 cp: setting attributes for '/media/pc/key1/file.txt': Operation no
   supported
```

Listing 27 – Copy to 8 GB USB flash drive, FAT32

```
54  ~ $ attr -s author -V steven file.txt
55 The "author" attribute set to a value of 6 bytes for file.txtÿ:
56 steven
57  ~ $ cp --preserve=xattr file.txt /media/pc/cle1 ~ $ cd /media/pc/cle1
58
59 /media/pc/cle1 $attr -g author file.txt 60 The "author" attribute
had a value of 6 bytes for file.txtÿ:
61 steven
```

Listing 28 - Copy to 8GB USB stick, NTFS

```
~ $ attr -s author -V steven file.txt
The "author" attribute set to a value of 6 bytes for file.txtÿ:
steven
~ $ cp --preserve=xattr file.txt /media/pc/cle2 ~ $ cd /media/pc/cle2

/media/pc/cle2 $attr -g author file.txt The "author" attribute
had a value of 6 bytes for file.txtÿ:
steven
```

Listing 29 - Copy to 64 GB USB flash drive, ext4

```
~ $ cp --preserve=xattr file.txt /mnt/debian
cp: setting attributes for '/mnt/debian/file.txt': Operation no
supported
```

Listing 30 - Copy to remote network location, NFS

We see that the operation is unsuccessful on the key in FAT32 and on the location network mounted in NFS while it succeeds on USB keys in NTFS and ext4.

Two other little experiments were done with the mv command and copying/moving files with Linux Mint's Nemo file explorer. Both of these operations preserve the XATTRs by default.

4.3 inotify

Under Linux, a tool (included in the kernel) dedicated to FS monitoring exists, inotify [32]. As the name suggests, inotify gives an application the ability to be notified about filesystem-level events. A programming interface (API) in C exists and offers the following system calls (SYSCALL):

1. int inotify_init(void) : initializes an inotify instance and returns a descriptor of file.
2. int inotify_add_watch(int fd, const char *pathname, uint32_t mask): this function expects the file descriptor returned by inotify_init, a file path or directory to watch and a binary mask consisting of the events to watch (see below). It returns a new file descriptor that can be read with SYSCALL read().
3. int inotify_rm_watch(int fd, int wd): inverse call to the previous one, removes the monitoring of the file descriptor wd from the inotify instance returned by fd.

inotify is used as follows: you must initialize the instance, add the files and directories for monitoring with the desired event mask and, generally, in a loop, call the SYSCALL read() with the returned file descriptor as an argument by inotify_init(). Each successful call to read() returns the structure available for listing
31 :

```
1 struct inotify_event {
2         int     wd;             /* Watch descriptor */
3         uint32_t mask;          /* Event mask */
4         uint32_t cookie; /* Cookie unique d'association des
5                                 events (for rename(2)) */
6         uint32_t len;           /* Size of the name field */ name[]; /
7         char    * Optional null-terminated name */
8 };
```

Listing 31 – Structure inotify_event - [32]

The mask field can take the following values (multiple values allowed, separated by logical "or" -> "|"):

— IN_ACCESS: file access.

— IN_ATTRIB: change to file attributes.

— IN_CLOSE_WRITE: file opened for writing closed.

— IN_CLOSE_NOWRITE: file opened for writing closed.

— IN_CREATE: file/directory creation.

— IN_DELETE: deletion of a file/directory.

— IN_DELETE_SELF: deletion of the monitored directory itself.

— IN_MODIFY: modification of a file/directory.

— IN_MOVE_SELF: deletion of a file/directory.

— IN_MOVE_FROM: moving/renaming the directory (old name).

— IN_MOVE_TO: moving/renaming the directory (new name).

— IN_OPEN: opening a file.

— IN_ALL_EVENTS: macro combining all previous events.

The cookie field of the inotify_event structure takes on its full meaning during IN_MOVE events: a unique number is generated to link these two sub-events, which are actually only one. inotify therefore offers a very good basis for FS monitoring. However, it has some limitations:

- No recursive monitoring of a directory: if a complete tree must be monitored, a dedicated monitoring must be added for each sub-directory.
- File paths may change between event emission and event processing ment.
- inotify only allows userspace directory monitoring by default.
- There is no way to discriminate which process or user generated an event.

There are several system tools that use inotify [33]:

- incron: equivalent of cron, but the execution of the tasks is not done according to a schedule given, but according to a given event on a file.
- lsyncd: synchronization tool, based on rsync. Synchronization is performed on each change in the monitored directory to a list of remote locations configured in advance.
- iwatch: triggering a command according to an inotify event.
- inotify-tools: two commands allowing to use inotify directly in the terminal :
  - inotifywait: executes a wait on an event, before continuing the thread of execution.
  - inotifywatch: returns a list of events from monitored directories.

For more information, the inotify man page exists [32] and a very good two-part article on the additions of inotify over dnotify (its predecessor) [34] and its limitations by Michael Kerrisk [35] .

Finally, there is also a newer API to receive notifications from FS, fanotify [36]. It erases some of inotify's flaws (in particular access to mounted devices, such as USB keys), but has a major flaw for this project: there is no support for file creation, deletion and movement events and directories. fanotify cannot therefore be used for this project.

4.4 Sockets

Sockets (literally "socket" in French) are a means of communication between different processes, whether they are on the same machine or in a network. There are several types of sockets, the two best known are "local" sockets (type AF_UNIX or AF_LOCAL), only possible on the same machine because they are linked by a special file on the FS of the machine, and IP sockets (type AF_INET or AF_INET6) which are linked by pairs of IP addresses and ports. During a socket communication between two processes, one of the processes assumes the role of server and the other of client. The server must, in order, execute the following SYSCALLs:

1. Create the socket with socket().

2. Bind the socket to a listening address with bind().
3. Listen for an incoming connection with listen().
4. Accept an incoming connection with accept().
5. Receive messages with read(). 6. Emit messages with write().
7. Close the connection with close().

The client, for its part, must execute the following SYSCALLs:

1. Create the socket with socket().
2. Connect to a listening server with connect().
3. Receive messages with read().
4. Emit messages with write().
5. Close the connection with close().

Figure 9 summarizes the procedure for initializing and using sockets.

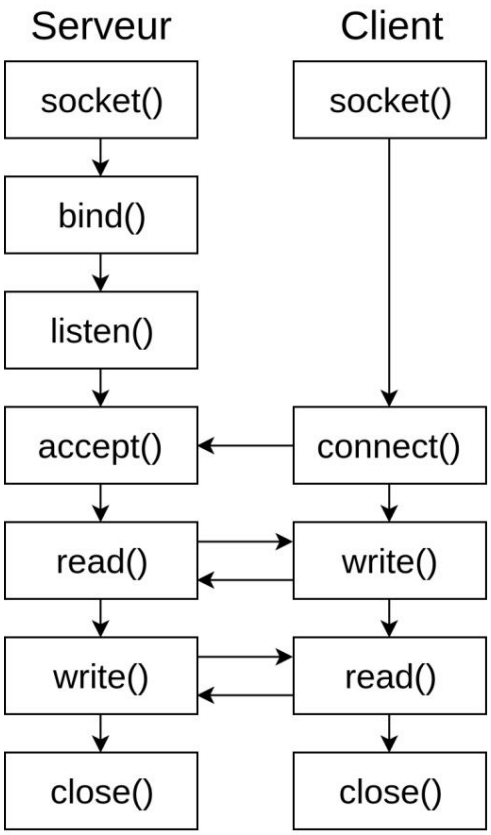


Figure 9 – Procedure for initializing and using sockets

When establishing the connection, a channel is established between the two processes, whatever the server writes is received by the client and vice versa. For more information, the sockets man page is available [37].

5 Achievement

5.1 Tag Manager

5.1.1 Program and code description

The first realization of this project is a command line tool (CLI), written in Rust, allowing to easily list, add and delete tags to files and directories (with a recursive option for these) and to execute requests to the server (Tag Engine, subsection 5.2) to list existing tags, rename a tag and request the list of files corresponding to given tags. This tool depends on two crates available on crates.io : clap [38] and xattr [39]. Tags are stored in an extended attribute (XATTR) named user.tags and are separated from each other by commas.

clap (Command Line Argument Parser for Rust) is a library for parsing the arguments of a CLI program. It analyzes and validates the arguments provided by the user. It has several syntaxes to define the arguments of the commands and options expected by our program. On clap's github repository, in the examples folder, several usage examples are provided. To illustrate its use, Listing 32 repeats example 01a_quick_example.rs with most of the comments truncated and layout changes. From lines 2 to 13, the expected arguments and information about the application are defined with among others the version of the program, the name of the author, etc. In this example, the arguments are defined from a character string respecting a very specific format. clap automatically generates program help from the defined arguments if it is run without any of the required arguments. From line 15, the received arguments are used. The value_of() method returns the value of an argument present at runtime. It is therefore easy to use the given arguments as program variables. clap also gives the possibility to group the arguments. Only one argument of a group can be present at runtime, which avoids many conditions for detecting and excluding arguments.

xattr is a Rust application programming interface (API) for retrieving, listing, adding/editing, and deleting XATTRs hooked to files with Rust. It is essentially a wrapper of the C system calls (SYSCALL) provided by Linux and other operating systems (OS) to manipulate XATTRs. Note that the functions offered do not follow symbolic links (it is the same for Tag Manager itself). Four functions are available: get(), list(), set() and remove(). All expect the name of the file and, depending on the case, the name of the XATTR as well as its value.

```
fn main() {
    let matches = App::new("MyApp") .version("1.0")

        .author("Kevin K. <kbknapp@gmail.com>") .about("Does
        awesome things") .args_from_usage( "-c, --
        config=[FILE] 'Sets a custom config file' <output>
            'Sets an optional output file' -d... 'Turn debugging information on'")


        .subcommand(SubCommand::with_name("test")
            .about("does testing
            things") .arg_from_usage("-l, --list 'lists test values'")) .get_matches();


    if let Some(o) = matches.value_of("output") { println!("Value for
        output: {}", o);
    }
    if let Some(c) = matches.value_of("config") { println!("Value for
        config: {}", c);
    }

    match matches.occurrences_of("d") {
        0 => println!("Debug mode is off"), 1 => println!
        ("Debug mode is kind of on"), 2 => println!("Debug mode
        is on"), 3 |
            _ => println!("Don't be crazy"),
    }

    if let Some(matches) = matches.subcommand_matches("test") {
        if matches.is_present("list") { println!
            ("Printing testing lists..."); } else {

            println!("Not printing testing lists...");
        }
    }
}
```

Listing 32 - Example of using clap (truncated comments) - [40]

Tag Manager consists of two files. The first, main.rs contains the definitions and detections of the arguments supplied by the user with clap (see listing 33), the calls to the functions manipulating the tags of the files and the connection socket part, requests and waiting for a response from the server (Tag Engine, subsection 5.2). The second file, lib.rs, contains the public API for retrieving, assigning, renaming, and deleting tags for a given file and a test module for those functions. Note that in the output of the program, there is no distinction between files and directories. As an example, Listing 34 shows code for the del_tags() function that removes given tags from a file. It preserves existing tags that should not be deleted and completely removes the XATTR in case of an empty tag array. The use of Option and Result enums in association with pattern matching are used whenever possible.

```
43          the file \"myfile\"\n                    \
44          tag_manager -f myfile -d work  to the                => Delete the tag \"work\" \
45          file \"myfile\"\n  \ tag_manager -f
46          myfolder -r -s geneva => Set the tag \"geneva\" \ to the folder \"myfolder\" and his subtree\n
47          \ => Show files corresponding to query\  tag_manager -q bob AND the list of existent tags\n
48          to \"n tag_manager -R old_name new_name  let matches =                tag_manager -l  => Rename the tag \"old_name\"
49          App::new("tag_manager") .help(help) .group(ArgGroup::with_name("ops").args(&["set",
50                                                                  "del"])) .group(ArgGroup::with_name("queries")
51
52
53
54
55                  .args(&["list", "query",
56
57
58
59          "rename"])) .arg(Arg::with_name("set").short("s").long("set") .takes_value(true).multiple(true)) .arg(Arg::w
60          .arg(Arg::with_name("files").short("-f").long("--
61                  files") .takes_value(true).multiple(true).required(false))
62          .arg(Arg::with_name("recursive").short("-r")
```

Listing 33 - Declaration of arguments in main.rs

```
55 pub fn del_tags(file: &str, tags_to_del: &HashSet<String>, recursive:
   bool) {
56      recursion(file, recursive, Delete, tags_to_del); match
57      check_existent_tags(file) {
58          Ok(res) => match res {
59              Some(mut tags) => { //
60                  Delete only the given tags for tag in
61                  tags_to_del { tags.retain(|ref e| e != &tag);
62
63                  }
64                  // To avoid to let an empty array of tags if tags.is_empty()
65                  {
66                      match xattr::remove(file, ATTR_NAME) {          _   => () }
67                  }
68                  else {
69                      xattr::set(file, ATTR_NAME,
70                          &hash_set_to_vec_u8(&tags)) .expect("Error
71                           when (re)setting tag(s)");
72                  }
73              }, _     => ()
74          },
75          Err(err) => {
76              eprintln!("Error for file \"{}\" : {}", file, err);
77              return;
78          }
79      }
80      println!("Tag(s) {:?} for file {:?} have been deleted", tags_to_del, file);
81
82 }
```

Listing 34 - Code for del_tags() function in lib.rs

Socket communication is achieved using the standard UnixStream library, equivalent to AF_UNIX sockets in C (see subsection 4.4). The sockets address file is by default written in /tmp/tag_engine. The request format respects the small protocol described in table 4. The start of the request is formed by the three-character code. The query listing the files according to a logical expression of the tags accepts the AND and OR operators, with the logical precedence of the first over the second (see subsection 5.2 for more details). Operators and operands (tags) must be separated by a space.

Query	Code Example
Files and directories corresponding to a logical expression of tags	0x0 0x0 tag1 OR tag2 AND tag3
List of existing tags	0x1 0x1
Renaming a tag	0x2 0x2 old_name new_name

Table 4 – Protocol format for requests to the Tag Engine server

Figure 10 summarizes how the program works:

1. Analysis of arguments.
2. Execution of the commands (either on the files, or request to the server).
3. Printing of results.

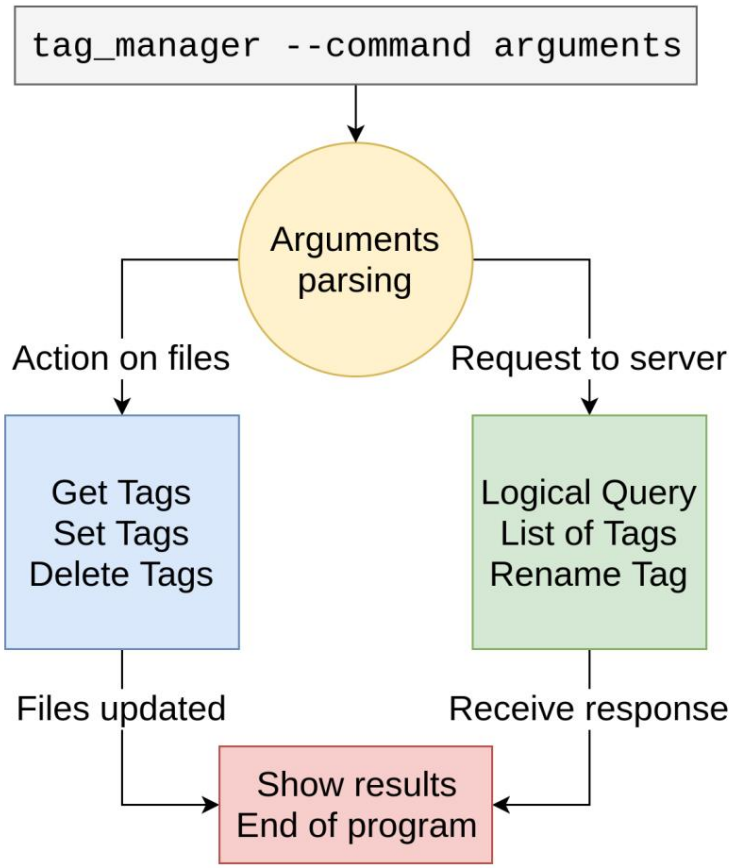


Figure 10 – Tag Manager operation diagram

5.1.2 Using the program and examples

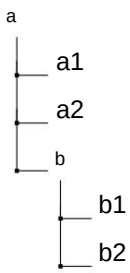
The use of the different arguments of the program is summarized in table 5. The arguments are divided into two groups, the first to directly manipulate the files and their tags, the second to execute requests to the Tag Engine server. For the first

group, the -f or --files argument is required. For the operations of the second group, it is obviously necessary that the Tag Engine server is launched.

Operation	Arguments	Example
Show help	-h	tag_manager -h
Show tags of one or more files	-f or --files	tag_manager -f file1 file2
Show folder tags recursively	-r or --recursive	tag_manager -f myfolder -r
Assign tags to one or more fi les	-s ou --set	tag_manager -f file1 file2 -s bob fred
Remove tags from one or more files	-d or --del	tag_manager -f file1 file2 -d bob fred
List cor files respondents to a tag request	-q or --query	tag_manager -q bob AND fred
List exis tags so many	-l or --list	tag_manager -l
Rename a tag	-R or --rename	tag_manager -R old_name new_name

Table 5 – Usage and arguments expected by Tag Manager

Listing 35 illustrates the uses and returns of Tag Manager. First, Tag Manager is used to recursively list tags of given files and subdirectories, assign in_a and myfiles tags to different files and directories, list existing tags, and query on both tags. The tree structure used for the example consists of the following four files and two directories:



```
1 $ ./tag_manager -f a -r 2 File
"a" has no tags 3 File "a/a1" has
no tags 4 File "a/b" has no tags
5 File "a/b/b1" has no tags 6 File
"a/a2" has no tags 7 $ ./
tag_manager -f a/* -s in_a 8 Tag(s)
{"in_a"} for file "a/a1" have been setted 9
Tag(s) {"in_a"} for file "a/a2" have been setted 10 Tag(s) {"in_a"}
for file "a/b" have been setted 11 $ ./tag_manager -f a/b a/b/b1
a/b/b2 -s myfiles 12 Tag(s) {"myfiles"} for file "a/b" have been
setted 13 Tag(s) {"myfiles"} for file "a/b/b1" have been setted 14
Tag(s) {"myfiles"} for file "a/b/b2" have been setted 15 $ ./
tag_manager -f a -r 16 File "a" has no tags 17 Tag(s) ["in_a"] for file "a/
a1"



18 Tag(s) ["in_a", "myfiles"] for file "a/b"
19 Tag(s) ["myfiles"] for file "a/b/b1"
20 Tag(s) ["myfiles"] for file "a/b/b2"
21 Tag(s) ["in_a"] for file "a/a2" 22 $ ./
tag_manager -l 23 in_a

24 myfiles
25 $ ./tag_manager -q myfiles AND in_a 26 /
home/stevenliatti/a/b
```

Listing 35 - Examples of using Tag Manager

5.2 Tag Engine

5.2.1 Program and code description

The second practical realization of this project is a program indexing and monitoring a tree structure of directories, files and associated tags. This program also acts as a server for requests issued from Tag Manager. It is divided into several files and modules:

— graph.rs: module relating to the construction and maintenance of the graph of tags, files and directories (see paragraph petgraph).

— lib.rs: groups the other modules and contains the dispatcher() function, called during an event on the monitored tree (see notify paragraph).

— main.rs: program entry point, initializes shared variables, the server socket thread and listens indefinitely on events occurring on the monitored tree

(see paragraph Mutex and multiple references).

— parse.rs: module for converting an infix expression to a postfix (see paragraph Analyzing a logical expression).

— server.rs: module building the socket server and responding to the various requests sent from Tag Manager (see the Sockets server paragraph).

This program depends on four external crates. The first is tag_manager (the library of functions handling XATTRs, see subsection 5.1), made during this project. The other three are available on crates.io, these are walkdir [41], petgraph [42] and notify [43].

walkdir is a library for efficiently and recursively traversing a tree of files. It offers different structures, including WalkDir and DirEntry. WalkDir expects a directory path and returns an iterator listing each subdirectory and file contained in the starting directory. Each entry of this iterator is represented by the DirEntry structure, which holds methods for obtaining information about the entry (the full path, the metadata, if it's a directory or a file, its name, etc.) . Listing 36 illustrates an example of traversing a directory and displaying information for each entry encountered. walkdir is used to perform the first scan of the provided tree and build the graph (see listing 38).

```
1 use walkdir::WalkDir;
2
3 fn main() {
4     for entry in WalkDir::new("/home") { let entry =
5         entry.unwrap(); println!("{}",
6         entry.path().display());
7     }
8 }
```

Listing 36 - Traversing a directory with walkdir

petgraph is a graph representation library. It provides different data structures to represent a graph and modules for manipulating and traversing graphs.
There are three data structures:

1. Graph<N, E, Ty = Directed, Ix = DefaultIx> : represents a graph and its data in the form of two adjacency lists (two Vec, one for the nodes and the other for the arcs or edges) . The nodes are of the generic type N, the arcs or edges of the generic type E, the graph is oriented by default (Directed type) and the type for the index (the numerical identifier of a node and of an arc or edge in the respective vectors, determining the maximum size of the graph) is by default u32 (which allows more than four billion nodes, 4'294'967'296 precisely).

2. StableGraph<N, E, Ty = Directed, Ix = DefaultIx> : similar to Graph, with a notable difference, it keeps the identifiers of nodes and arcs or edges removed.
   This structure also has fewer methods.
3. GraphMap<N, E, Ty> : represents a graph and its data in the form of an associative table, whose keys are the nodes of generic type N, with the obligation for this type to be compliant for the use as a key. Allows to test the existence of a node in constant time with the counterpart of not being able to store more than one node with the same data.

Before continuing the explanations on the use of petgraph, let's briefly review the architecture chosen to represent the tree structure of files and tags to monitor. A graph node can be either a file, or a directory, or a tag. The link between a directory and a sub-directory or a file is symbolized by an arc starting from the directory in question.
The link between a tag and a directory or a file is also an arc starting from the tag in question. An arc has no data to save, so it has no effective type. To speed up access to a tag in the graph from its name, a hashmap associating the name of the tag with its identifier in the graph is maintained.

The StableGraph structure was chosen because of its retention of identifiers when deleting nodes and arcs. Indeed, during tag or file deletion operations, to maintain consistency in the hashmap of the tags associated with the identifiers of the graph, these identifiers must not change. Otherwise, this hashmap would not be usable. Listing 37 shows the data structures used for our graph. A graph node is represented by the Node structure, containing the name of the node and its type (tag, file or directory). An arc is simply defined by an empty type, named Nil. petgraph offers many methods for StableGraph: access, addition and deletion of nodes and arcs, an iterator on the neighbors of a node (with direction indication), search for arcs between two nodes, etc. The make_graph() function listed in listing 38 is the primary function of our module, it creates the graph and the hashmap with walkdir from the root directory path. It returns the graph containing the files, directories and tags found, the hashmap filled in and the identifier of the root node, starting point for the following updates of the graph (on the files and directories). The make_subgraph() function, called in the loop, makes sure to correctly add the nodes according to the path, to avoid

to create the same node twice or for a child node to be created before its parent (there is no guarantee of traversing the file tree in parent-child order).

```
16 #[derive(Debug, Clone)] 17 pub
struct Nil;
18
19 #[derive(Debug, Clone)] 20 pub
enum NodeKind {
21          Tag,
22          File,
23          Directory
24 }
25
26 #[derive(Clone)]
27 pub struct Node {
28          pub name : String, pub
29          kind : NodeKind
30 }
31
32 pub type MyGraph = StableGraph<Node, Nil>;
```

Listing 37 – Structures for graph nodes and edges in src/graph.rs

```
83 pub fn make_graph(path_root : String, base_path : String)
84         -> (MyGraph, HashMap<String, NodeIndex>, NodeIndex) { let mut
85         graph : MyGraph = StableGraph::new(); let mut tags_index =
86         HashMap::new(); let local_root = local_path(&mut path_root.clone(),
87         base_path.clone()); let root_index = graph.add_node(
88
89
90                 Node::new(local_root, NodeKind::Directory)
91         );
92         update_tags(path_root.clone(), &mut tags_index,
93                 &mut graph, root_index);
94         let mut is_root = true;
95
96         for entry in WalkDir::new(path_root).into_iter()
97                 .filter_map(|e| e.ok()) { if is_root {
98
99                         is_root = false;
100                        continue;
101                }
102                let mut path = entry.path().display().to_string(); let path =
103                local_path(&mut path, base_path.clone()); make_subgraph(root_index,
104                &mut tags_index, &mut graph,
105                        path, base_path.clone());
106        }
107        (graph, tags_index, root_index)
108 }
```

Listing 38 – Fonction make_graph() dans src/graph.rs

For each file and directory, the update_tags() function, listed in Listing 39, compares the tags in the XATTRs to the tags present in the graph and updates the relations if necessary.

```
218 pub fn update_tags(path : String,
219         tags_index : &mut HashMap<String, NodeIndex>, graph : &mut
220         MyGraph, entry_index : NodeIndex) { let existent_tags =
221         get_tags(graph, entry_index); let fresh_tags = match
222         tag_manager::get_tags(&path) {
223             Some(tags) => tags,
224             None => HashSet::new()
225         };
226         remove_tags(existent_tags.difference(&fresh_tags), tags_index,
227             graph, entry_index); add_tags(fresh_tags.difference(&existent_tags),
228         tags_index, graph, entry_index);
229
230 }
```

Listing 39 – Fonction update_tags() dans src/graph.rs

notify is a cross-platform FS event notification library. It uses different implementations depending on which OS it is used on. On Linux, it relies on inotify. It expects a file or directory path. It also has a recursive option for monitoring a directory. It offers two distinct APIs:

— Debounced API (default): returns all events with pre-processing done by notify, grouping some events into one, for example: renaming a file, single create event when creating a file rather than a create+write+chmod. The events are sent after a delay (defined at creation), precisely to be able to group them upstream if necessary. Each event is a member of the DebouncedEvent enumeration.

— Raw API: returns all events without pre-processing by notify and immediately. It has the advantage of being exhaustive but more logical processing must be performed (especially for renaming events). Each event is contained in the RawEvent structure, itself containing the path of the file having undergone the event (path), the event in question (op) and a "cookie" making the link between two sub-events forming part of a single rename event (see section 4.3 for more details).

Debounced API is used here to help detect rename events. Both APIs require the creation of a communication channel between two or more threads: notify will act as a producer by emitting events. In an infinite loop, the events needed to update the graph are caught and processed. This is the game

channel consumer thread. The graph updating events are as follows:

— File or directory creation.

— Modification of attributes (in our case, tags).

— Delete file or directory.

— File or directory renaming.

The detail of this mechanism is available in Listing 40, lines 14 to 33.

Mutex and multiple references Since the main thread, consuming events emitted by notify, and the server socket thread, listening on incoming requests, both need to read and modify the graph and the associated hashmap, the basic rule of a single owner per value is not respected. Moreover, the threads must not access these two variables at the same time, at the risk of ending up with inconsistent data.

The solutions to both of these problems are atomic multiple references, or Atomic Reference Counting (Arc) and locks, or Mutex. The former allow multiple concurrent and concurrent owners at a value, at the cost of a slight performance degradation for memory safety issues. The second are classic locks in concurrent programming, like those existing in C for example. In Listing 40, we can see in lines 5 and 6 the graph and the hashmap being wrapped in a Mutex, itself wrapped in an Arc, and in lines 7 and 8 the creation of a reference to this graph and hashmap thanks to the Arc::clone() function. From there, if the main thread wishes to use the graph or the hashmap, it must take the lock, as in lines 23 and 24 of this same listing. The socket server thread must do the same on its side, this is the reason why in line 11 it receives the variables graph and tags_index defined in lines 5 and 6.

```
// Initialization of shared variables: graph and hashmap
// Multiple references possible thanks to Arc
// Accès concurrent grâce à Mutex let (graph,
tags_index, root_index) = make_graph(path, base_path); let graph = Arc::new(Mutex::new(graph));
let tags_index = Arc::new(Mutex::new(tags_index)); let main_graph = Arc::clone(&graph); 8
let main_tags_index = Arc::clone(&tags_index);



// Launch the server socket in a separate thread 10 thread::spawn(move ||
{ server(base_path, &graph, &tags_index);

});
// Initialisation de la surveillance avec notify 14 let (tx, rx) = channel();
let mut watcher = watcher(tx, Duration::from_secs(1)).unwrap(); 16
watcher.watch(path, RecursiveMode::Recursive).unwrap(); loop {



        match rx.recv() {
            match event {
                Create(_) | Chmod(_) | Remove(_) | Rename(_, _) => {
                    // Taking the lock on the graph and the hashmap // for possible
                    modifications let mut ref_graph = main_graph.lock().unwrap();
                    let mut ref_tags_index = main_tags_index.lock().unwrap();



                    // dispatcher takes care of performing the right action // depending on
                    the event
                    dispatcher(event, &mut ref_tags_index,
                        &mut ref_graph, root_index, base_path);
                }
            }
        }
}
```

Listing 40 – Tag Engine main.rs function (reduced and simplified, non-functional)

Sockets server The src/server.rs file contains all the logic for processing requests from Tag Manager. As seen in Listing 40 on lines 10-12, the server is started in a separate thread. Therefore, Tag Engine can listen on both FS events and requests from Tag Manager. Three types of queries are implemented, the detail is available in table 4 of subsection 5.1.1. The code received is converted into an enum, RequestKind, visible in Listing 41, to facilitate manipulation by pattern matching.

```
1 enum RequestKind {
2         Entries(String),
3         Tags,
4         RenameTag(String)
5 }
```

Listing 41 - RequestKind enumeration in server.rs file

Each request is first parsed with the parse_request() function which determines if the request is valid and what type it is. Locks are then taken to access the graph and the hashmap to construct the response. Functions reading and writing requests and responses manipulate a UnixStream.

There is an interesting mechanism to show in the request_tags() function copied in Listing 42. The function itself is not very complex, it takes the lock on the hashmap of the tags associated with the identifiers of the nodes of the graph and returns a vector containing all the tags, id is the keys of the associative table. It is this last operation which is particularly powerful, in lines 5 and 6 of the function. To generate the entries vector from the tags_index keys, the collect() function is used. This function allows you to transform one iterator into another, very simply.

```
1 fn request_tags(tags_index_thread : &Arc<Mutex<HashMap<String, NodeIndex>>>,
2         stream : &mut UnixStream) { println!("Request for Tags"); let tags_index =
3         tags_index_thread.lock().unwrap(); let mut entries : Vec<String> =
4         tags_index.keys() .map(|key| key.clone()).collect();
5
6
7         entries.sort();
8         write_response(entries, stream);
9 }
```

Listing 42 - Illustration of the use of the collect() function

Parsing a logical expression The file src/parse.rs mainly contains a function transforming an infix logical expression into a postfix (also called "reverse Polish notation" [44]). In the context of this program, it is used by the request providing one or more tags, separated by logical "and" or "or" to retrieve the list of corresponding files and directories. The infix expression type is the "natural" way in mathematics or logic to declare the sequence of operators and operands of a calculation or Boolean logical expression. The following examples are more explicit: the infix logical expression bob OR fred AND max would result in the postfix expression bob fred max AND OR. The algorithm used to implement this function is available here [45]. Listing 43 shows the two enums used for the conversion. An Arg is either an operand with a name, or an "AND" or "OR" operator. The compare() method, inspired by Java, compares the two operators to give priority to "AND" over "OR". This method illustrates the power of pattern matching, destructuring two variables at the same time in an elegant way.

```
5 #[derive(Debug, Clone, PartialEq)] 6 pub enum
Operator { AND, OR } 7 impl Operator { fn
compare(&self, other : &Operator) -> i8 {
8
9                   match (self, other) {
10                          (&AND, &OR) => 1,
11                          (&OR, &AND) => -1,
12                          _   => 0
13                  }
14          }
15 }
16
17 #[derive(Debug, Clone, PartialEq)] 18 pub enum
Arg {
19          Operand(String),
20          Operator(Operator)
21 }
```

Listing 43 – Operator and Arg enumerations and compare() method

The reason for manipulating a postfix rather than an infix expression is that the postfix evaluation algorithm is much simpler to implement, it only requires a stack to store the operators [44]. The algorithm is available in Listing 44.

```
for each token in the postfix expression:
    if token is an operator:
        operand_2 <-- pop from the stack operand_1
        <-- pop from the stack
        result <-- evaluate token with operand_1 and operand_2 push result back
        onto the stack
    else if token is an operand:
        push token onto the stack
result <-- pop from the stack
```

Listing 44 – Postfix expression evaluation algorithm - [44]

It is thus not necessary to establish an expression parsing grammar, as for a source code parser or regular expressions. For the simple evaluation of an expression comprising only two different operators, this solution is largely satisfactory and efficient. The function converting the infix expression to postfix is called infix_to_postfix() and is found in the src/parse.rs file and the function implementing the algorithm for evaluating a postfix expression is called expression_to_entries() and is found in the server.rs file. The operands of the expression are tags and the two operators are "AND" and "OR". "AND" takes precedence over "OR", like multiplication over addition. For each tag, a set in the mathematical sense of the files and directories it points to is made and placed on the stack. When an operator occurs, the last two sets of inputs are popped and the corresponding set operation is applied to them (an intersection for an "AND" and a union for an "OR"). The resulting new set is again pushed onto the stack. At the end of the algorithm, the final set is returned. Figure 11 summarizes how the program works:

1. Thread main, maintaining graph and hashmap and monitoring tree.
2. Sockets server thread responding to requests.

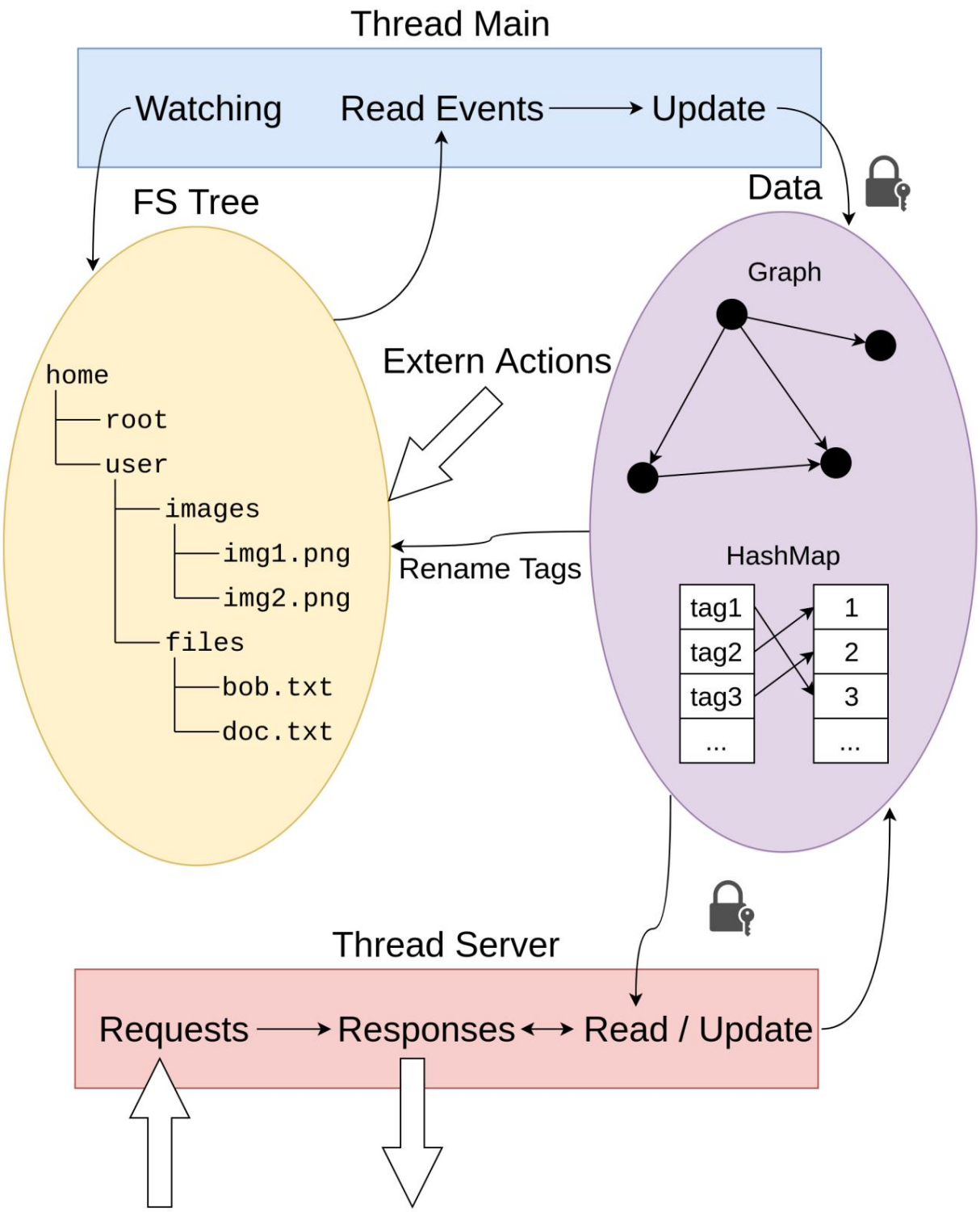


Figure 11 – Tag Engine operation diagram

5.2.2 Using the program and examples

The program expects a valid absolute path pointing to a directory. It takes an optional argument, -d or --debug, which prints the state of the graph and hashmap to standard output and writes two files, graph.dot and graph.png, after each event that occurs. Listing 45 shows the Tag Engine outputs corresponding to tree operations manipulated by Tag Manager (see section 5.1.2 and Listing 35). We see in lines 11 to 16 the addition of the "in_a" and "myfiles" tags and in lines 30 and 31 the requests for tags and entries made. Both files also represent the state of the graph. The graph.dot file respects the syntax of a tool called Graphviz [46], dedicated to graph modeling and visualization. It includes several engines for generating graphs from text or raw data, including dot used here. From the file graph.dot, available in listing 46, figure 12 is obtained. petgraph gives the possibility of generating a file respecting the semantics expected by dot to draw a graph. We can see that the result obtained agrees with the tree structure obtained.

```
1 $ ./tag_engine /home/stevenliatti/a -d 2 graph
StableGraph {
3           Ty: "Directed", node_count: 6, edge_count: 5, edges: (0, 1),
4           (1, 2), (1, 3), (0, 4), (0, 5), node weights: {
5
6               0: Directory "a", 1: Directory "b", 2: File "b1",
7               3: File "b2", 4: File "a2", 5: File "a1"
8           },
9           free_node: NodeIndex(4294967295),free_edge: EdgeIndex(4294967295)
10 }, tags_index {} 11
chmod : "a/a1" 12 chmod :
"a/a2"
13 chmod : "a/b"
14 chmod : "a/b"
15 chmod : "a/b/b1"
16 chmod: "a/b/b2"
17 graph StableGraph {
18          Ty: "Directed", node_count: 8, edge_count: 11, edges: (0, 1),
19          (1, 2), (1, 3), (0, 4), (0, 5), (6, 5), (6, 4),
20              (6, 1), (7, 1), (7, 2), (7, 3),
21          node weights: {
22              0: Directory "a", 1: Directory "b", 2: File "b1",
23              3: File "b2", 4: File "a2", 5: File "a1",
24              6: Tag "in_a", 7: Tag "myfiles"
25          },
26          free_node: NodeIndex(4294967295),free_edge: EdgeIndex(4294967295)
27 }, tags_index { "in_a":
28          NodeIndex(6), "myfiles": NodeIndex(7)
29 }
30 Request for Tags
31 Request for Entries "myfiles AND in_a"
```

Listing 45 - Example of using Tag Engine

```
1 digraph {
2       0 [label="Directory \"a\""] 1
3       [label="Directory \"b\""] 2 [label="File
4       \"b1\""]
5       3 [label="File \"b2\""]
6       4 [label="File \"a2\""]
7       5 [label="File \"a1\""]
8       6 [label="Tag \"in_a\""] 7
9       [label="Tag \"myfiles\""]
10      0 -> 1
11      1 -> 2
12      1 -> 3
13      0 -> 4
14      0 -> 5
15      6 -> 5
16      6 -> 4
17      6 -> 1
18      7 -> 1
19      7 -> 2
20      7 -> 3
21 }
```

Listing 46 – Dot file produced by petgraph

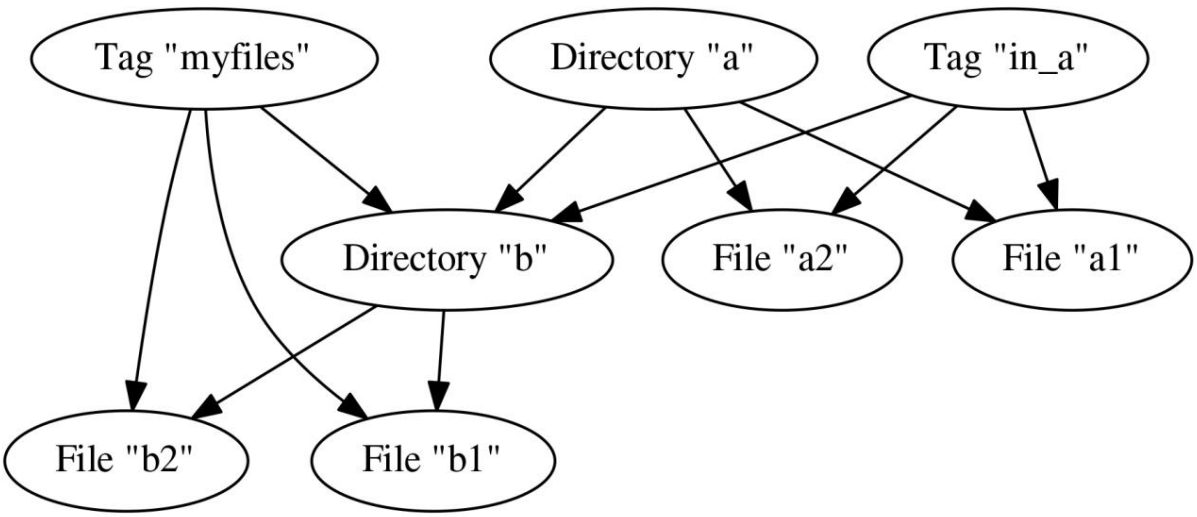


Figure 12 – Image of the graph obtained with the dot command

5.3 TagFS

Figure 13 summarizes the overall operation of the TagFS system, including Tag Manager and Tag Engine:

1. Tag management with Tag Manager.
2. Tree and server indexing and monitoring with Tag Engine.

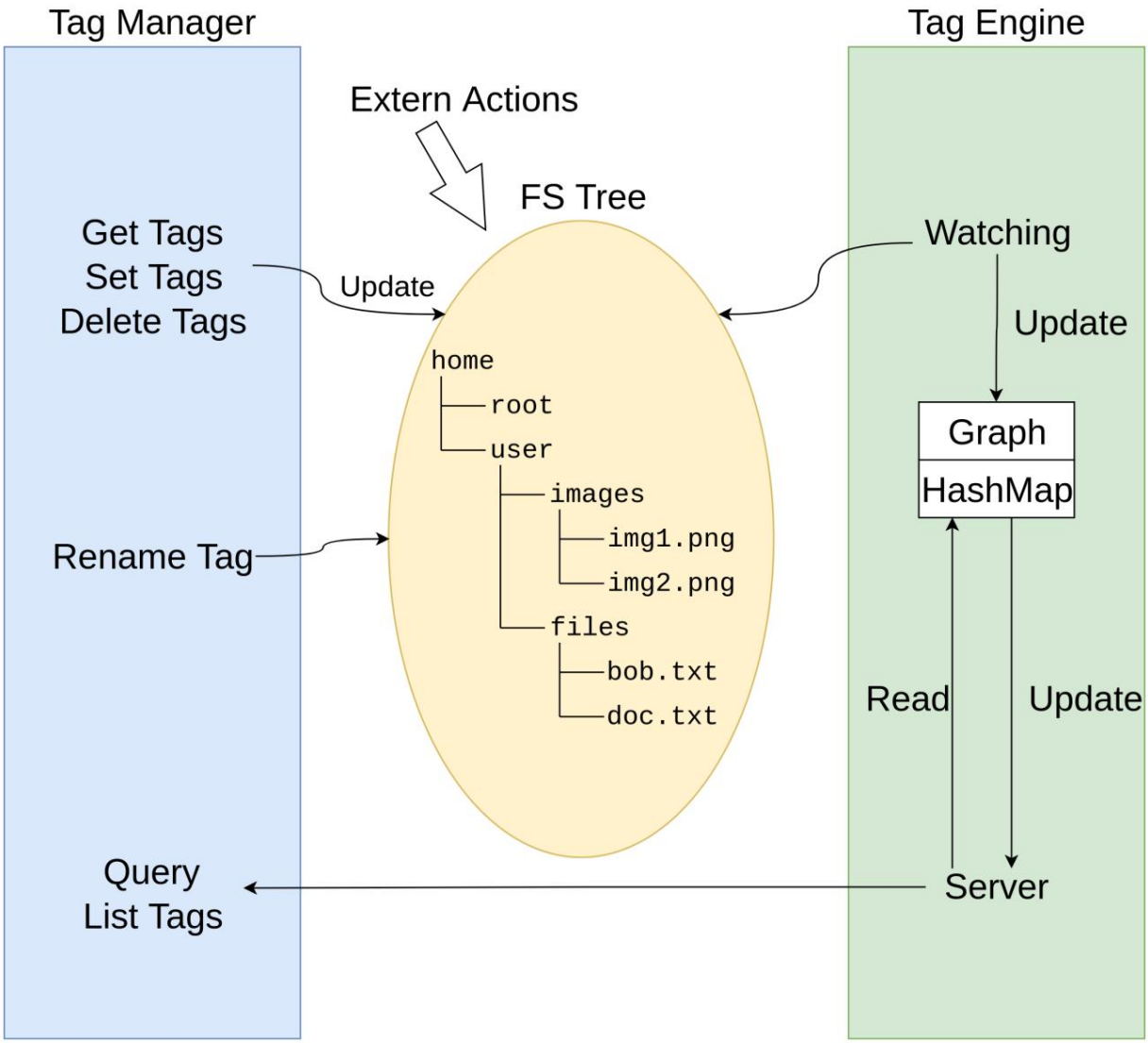


Figure 13 – Global Flow Diagram of TagFS

6 Tests

6.1 Performance Metrics

Execution time measurements were made with a slightly modified version of Tag Engine: the main() function was truncated from the launch of the server socket thread and the infinite loop listening on the events occurring on the tree on

vigil. Thus, the program traverses the tree structure only once and builds the graph and the associated hash table. This modification was made in order to measure several executions of the program (with Rust's Duration type) for a given directory to achieve an average time. It is illustrated in Listing 47.

```
fn main() {
    // ...
    let now = Instant::now(); let (graph,
    tags_index, root_index) =

        tag_engine::graph::make_graph( String::from(absolute_path_root), base_path.clone()
        );
    let new_now = Instant::now(); let elapsed
    = new_now.duration_since(now); println!("{}",
    elapsed.as_secs() as f64 +
        elapsed.subsec_nanos() as f64 * 1e-9);
}
```

Listing 47 – modified Tag Engine main.rs to measure execution time

In all, 200 executions were carried out, 100 with the program compiled in bug mode (cargo build, not optimized) and 100 with release mode (cargo build --release, with maximum optimizations). Cargo and rustc versions are: cargo 0.26.0 (41480f5cc 2018-02-26) and rustc 1.25.0 (84203cac6 2018-03-25). The target directories differ greatly in their number of sub-directories and files contained, ranging from five directories and 863 files to several thousand directories and a hundred thousand files (15,172 directories and 112,046 files precisely). These directories do not contain tags. Table 6 lists the directories used and their contents.

Directory	Player name tories	Number of files
Android	15’172	112’046
android-studio	3’331	13’287
bin	553	9’306
Documents	15’442	64’486
Dropbox	2’377	8’659
Images	5	863
Music	135	1’352

Table 6 – Directories used for execution time measurements

The bash script used to perform these 200 executions is shown in Listing 49. It writes for each directory two files containing the measurements of the 100 executions for the two compiled versions of the program. The last loop of the script executes the average.m file, found in Listing 48, which averages the 100 measurements. The machine used to compile and run the measurements has the following hardware and software characteristics (the commands lscpu, lshw, uname -r and lsb_release -a were used):

— Processeur : Intel(R) Core(TM) i7-3770K CPU @ 3.50GHz, boost @ 3.90GHz, x86_64, 4 coeurs, 8 threads.

— RAM: 4x4 GB DDR3 1333 MHz.

— Motherboard: Asus Maximus V Formula, Chipset Intel(R) Z77.

— System disk: Samsung 850 EVO Basic 500 GB.

— OS : Linux Mint 18.2 Sonya, kernel 4.15.0-24-generic.

```
1 arg_list = argv(); 2 filename =
arg_list{1}; 3 data_file =
importdata(filename); 4 data = data_file(3:102); 5 m =
mean(data); 6 fid = fopen(filename, "a"); 7 fprintf(fid,
"%f\n", m); 8 fclose(fid);
```

Listing 48 - Octave script to calculate run average

```
#!/bin/bash

iter=100
for entry in ~/*
do
        name=$(echo $entry | tr / -)

        find $entry -type d | wc -l > measures/release$name.txt find $entry -type f |
        wc -l >> measures/release$name.txt for i in ‘seq 1 $iter‘

        do
                target/release/tag_engine $entry >> measures/release$name.txt
        done

        find $entry -type d | wc -l > measures/debug$name.txt find $entry -type f
        | wc -l >> measures/debug$name.txt for i in ‘seq 1 $iter‘

        do
                target/debug/tag_engine $entry >> measures/debug$name.txt
        done
done

for entry in measures/*
do
        octave average.m $entry
done
```

Listing 49 - Bash script to run 100 runtime measurements

The averages obtained are represented in FIG. 14. The y-axis represents the average execution time of the program. The abscissa axis represents the seven directories used for this test. For each directory, there is a measure of the program compiled in debug mode (blue bar) and a measure of the program compiled in release mode (orange bar). Note that the two directories containing the most files, namely Android and Documents, are those whose execution time is the longest. Overall, the less entries a directory contains, the less execution time it will take. With two directories equivalent in terms of entries, the variations that may occur are most certainly due to the specifics of the files contained, such as their size on disk.

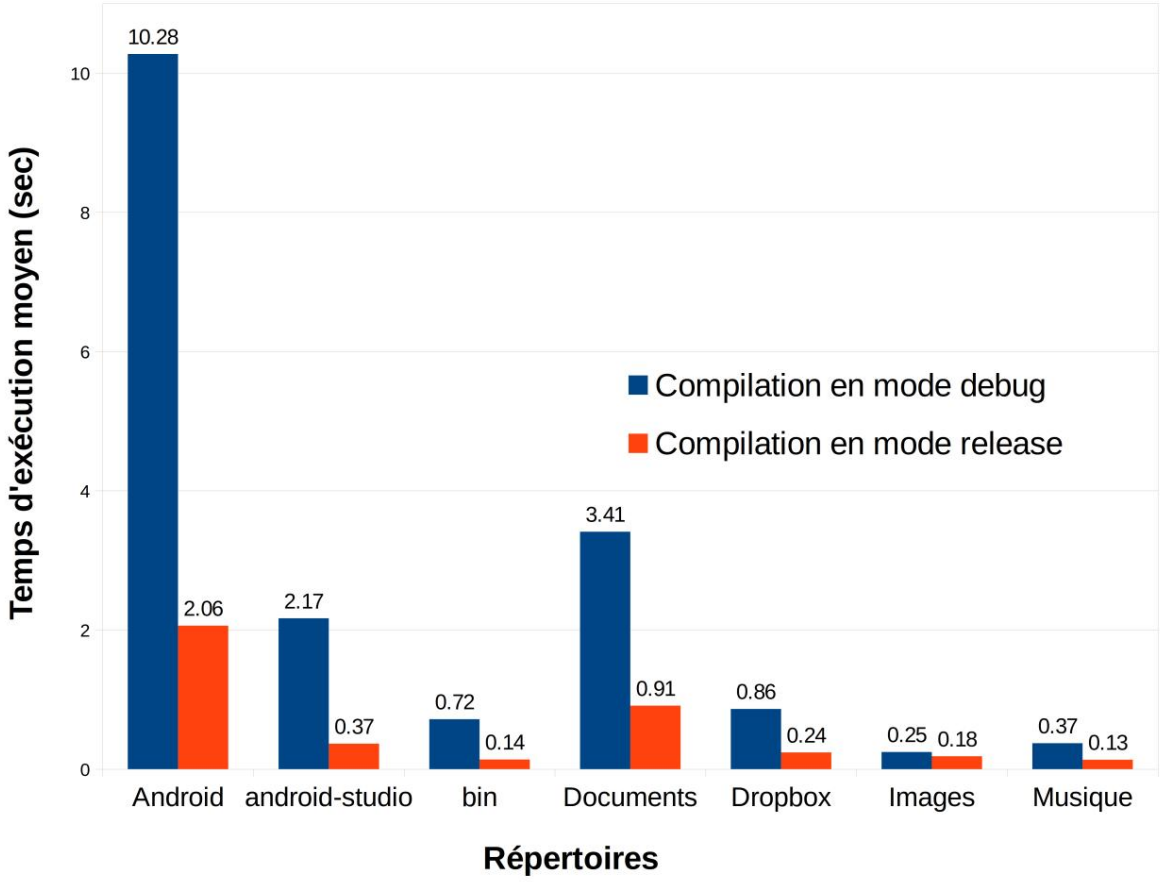


Figure 14 – Execution time according to the directory and by type of compilation mode

It is interesting to note that the execution time ratio between the unoptimized and the optimized version can be significant. Figure 15 illustrates these time relationships. We see that the difference varies between 5.93 for android-studio and 1.33 for Images, generally speaking again the directories with the most entries have the highest ratios. The Images directory containing few elements, fewer operations are executed, so it is more difficult to optimize this reduced number of operations during the execution of the program. The lesson to be learned from these reports is that the Rust compiler is capable of great code optimizations.

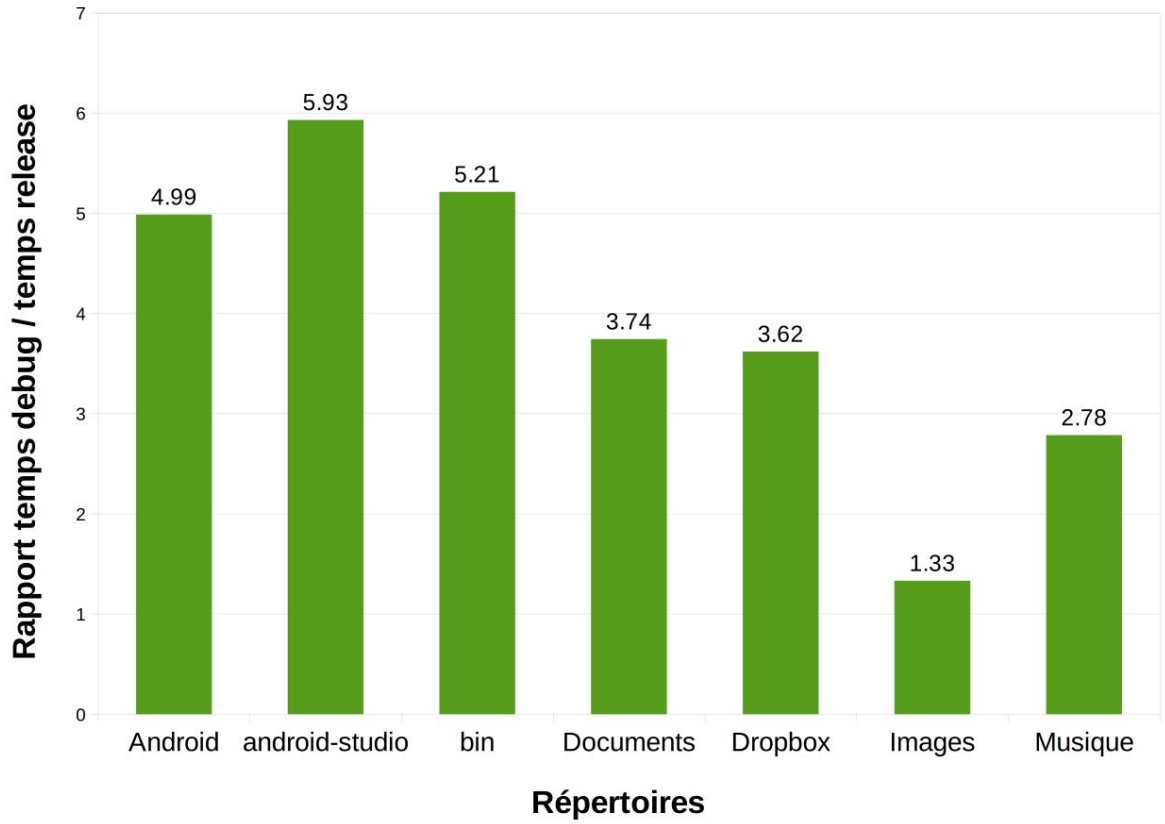


Figure 15 – Relationship between execution time in debug and release mode

7 Discussion

7.1 Rust VS C

In this subsection, we will discuss the system implementation in Rust rather than in C. The advantages are numerous and the disadvantages are acceptable.

7.1.1 Advantages of Rust over C

Rust's main advantage over C is the memory safety guarantee. As long as the code compiles, it will be safe in terms of memory allocation/deallocation and concurrency (logic errors are still the responsibility of the programmer, however).
This guarantee is provided by Rust's compiler, which could be defined as "intelligent".
Clever because almost all memory-related errors are caught at compile time.
He even goes further by being very verbose about the errors that have occurred, often suggesting the way to solve the problem. What over time have become best practices to know when programming in C, Rust forces programmers to implement before the code is even compiled. The following code listings show two examples where Rust behaves "better" than C. Listings 50 and 51 show a function, written in C and Rust, respectively, that returns an undefined pointer. In C, the detection of the error would occur at the execution of the program (although according to the C compiler used, a warning, warning, will be displayed to the programmer), with the famous Segmentation fault. The Rust equivalent would refuse to compile this code, because of the broken lifetime rule.

```
1 int* undefined_pointer() { int data[10]
2         = {114, 117, 115, 116, 105, 115, 98, 101, 115, 116}; return &data[3];
3
4 }
```

Listing 50 - Creating an undefined pointer in C

```
1 fn undefined_pointer() -> &i32 {
2         let data = [114, 117, 115, 116, 105, 115, 98, 101, 115, 116];
3         &data[3]
4 }
```

Listing 51 - Creating an undefined pointer in Rust

Another example is unauthorized access to memory. Listings 52 and 53 show a function, written in C and Rust respectively, that prints an element of an array at an incorrect index to standard output. In C, "no problem", the function will print this

that she finds at this memory address, without complaining. This kind of error is potentially a vulnerability that can be exploited for malicious purposes. In Rust, this error wouldn't be caught at compile time, but at least the program would crash on execution (panicked) and the error could be fixed.

```
1 void print_data() {
2       int data[10] = {114, 117, 115, 116, 105, 115, 98, 101, 115, 116}; printf("%d\n", data[10]);
3
4 }
```

Listing 52 - Unauthorized access to memory in C

```
1 fn print_data() {
2       let data = [114, 117, 115, 116, 105, 115, 98, 101, 115, 116]; println!("{}", data[10]);
3
4 }
```

Listing 53 - Unauthorized access to memory in Rust

We can also mention that Rust is sometimes faster than C for certain tasks, or almost as fast. The Computer Language Benchmarks Game [47] compares Rust and C, C++, Go, and Java one-by-one on runtimes, amount of memory, and CPU load for a variety of applications. Rust compared to C and C++ is sometimes faster and often a little behind. Rust compared to Go and Java is still the fastest. We can conclude that security can literally and figuratively rhyme with speed.

Another great advantage of Rust over C is its error handling and the absence of NULLs. Tony Hoare, a computer science researcher, is the inventor of NULL and today calls it his billion-dollar mistake [48]. NULL allows many unexpected errors, despite the fact that the idea is not bad (it is convenient to declare a variable containing no value). In Rust, NULL does not exist, instead a very powerful enumeration is available, Option (see Listing 13). It thus avoids handling empty data.

Then come several advantages that are less decisive, but nevertheless very practical in the everyday life of the programmer. Examples include the great Cargo tool and the crate source available at crates.io, containing thousands of libraries made and tested by the community. The unit tests of the program are integrated directly into the language and their execution is facilitated with Cargo. The standard Rust library is well supplied, especially with collections (vectors, hash tables, etc.). Finally, if it were necessary to finish, the management of generic types directly included in the language is a very practical asset.

7.1.2 Disadvantages of Rust compared to C

Rust nevertheless has two defects, more or less disabling depending on the time and the situation, linked to each other. The first is the language learning curve which can be long and daunting. Rust is governed by certain binding rules, simple but not easy to pin down in all situations. This is particularly the case for the rules of ownership, borrowing and references (see subsection 4.1.8), quite unique to Rust. The second defect stems from the first: it is sometimes necessary to review certain algorithms or data structures that are easily implemented in C or other similar languages. This is notably what happened during the implementation of Tag Engine and its data structure to contain the tree structure of directories, files and tags (see subsection 7.2 for more details). Other small flaws can be mentioned, such as the lack of default arguments when declaring functions or the overloading of methods or functions. These are not defects strictly speaking, but facilities that it would have been desirable to have.

Finally, the major obstacle to the mass adoption of the language (like many other new languages) is the lack of support from a major company (even if Mozilla uses it for its Firefox browser) and the "laziness " programmers to turn away from C or C++.

7.2 Problems encountered

The main problem encountered during this work was the implementation of a tree in Rust for the needs of Tag Engine. The simple implementation of a tree node in C is similar to that described in Listing 54.

```
1 struct Node {
2     int data; // the type is not important for the example
3     struct Node** children;
4 };
```

Listing 54 – Implementing a tree structure in C

It is a recursive structure and relatively easy to understand: a node consists of data of a chosen type and an array of pointers to child nodes. This kind of declaration is impossible in Rust, because the compiler needs to know the exact size of the data at compile time (avoid infinite recursive structures). One solution is to wrap the child nodes in a Box -like smart pointer (see chapter 15 of the book [20]). But another problem will arise when using nodes, because Rust does not allow multiple mutable references to the same value. To solve this last problem, the use of Rc<T>, or reference counting. This is another smart pointer, allowing multiple ownerships for a single value. We then meet with

a nesting of advanced pointers, as in Listing 55, a solution proposed by Rust Leipzig [49] :

```
use std::rc::Rc;
use std::cell::RefCell;

struct Node<T> {
        previous: Rc<RefCell<Box<Node<T>>>>, //
                                  ^^      ^^
                                  |  |    |  |
                                  |  |    |  - The next Node with generic 'T'
                                  |  |    |
        // // // //              |  |    - Heap allocated memory, needed if 'T' is a trait object.
        //                       |  |
        //                       |  - A mutable memory location with
        //                       |      dynamically checked borrow rules.
        //                       |      Needed because 'Box' is immutable.
        //                       |
        //                    - Reference counted pointer, will be
        //                        dropped when every reference is gone.
        //                        Needed to create multiple node references.

        next: Vec<Rc<RefCell<Box<T>>>>,
        data: T,
        // ...
}
```

Listing 55 - Rust node structure with smart pointers - [49]

Not only is this solution not very readable, but it also has the defect of removing all the advantages offered by the Rust compiler on the constraints of scope and lifetime of variables. A solution that seems to solve all the above problems is described in Listing 56. It uses an arena, a memory region [50]. Rather than having pointers between nodes of the tree, this solution uses a collection to store the data (here a vector) and the relations between nodes are held thanks to the identifiers in this same collection (here the indices of the vector). Thus, the problem of lifetime of values and multiple ownerships is resolved, because the arena is the sole holder of the data (therefore of the nodes). The nodes remain accessible from outside the arena by keeping their identifiers in the arena.

```rust
pub struct Arena<T> {
    nodes: Vec<Node<T>>,
}

pub struct Node<T> {
    parent: Option<NodeId>,
    previous_sibling: Option<NodeId>, next_sibling:
    Option<NodeId>, first_child: Option<NodeId>,
    last_child: Option<NodeId>,

    /// The actual data which will be stored within the tree
    pub data: T,
}

pub struct NodeId {
    index: help,
}
```

Listing 56 - Structure of a node in Rust with an arena - [49]

The crate petgraph used to store data for files, directories and tags in a tree structure works like this: it uses two vectors, one for nodes, the other for arcs, to store data. This crate perfectly met the needs of the new architecture for indexing files, directories and tags: it provides an implementation of a graph and allows access to nodes from outside the graph. For more details on this subsection, these various articles address this issue and give more details on possible solutions [21] [51] [49] [52] [53] [54].

7.3 Results and future improvements

The results of the performance measurements carried out in subsection 6.1 show that the initial traversal of the graph is relatively fast, at least it takes place with very little latency even for a directory containing more than 15,000 directories and more than 100' 000 files (a running average of two seconds to scan the whole tree). It is reasonable to say that this operation is among those most cumbersome for a given tree, the program can therefore be labeled efficient. However, to be completely sure, it would be necessary to carry out additional tests during longer uses.

Although the specification was met, the state of the system at render time was not not perfect, here is a non-exhaustive list of improvements that could be made:

— Integration with a Linux file manager (Nautilus, Nemo, etc.) for handling tags (adding, deleting) and for executing queries. An alternative would be to create a similar interface in the form of a web application, for example.

— Transformation of Tag Engine into Daemon (a process detached from the parent shell and which does not print its results on standard output). It could run at OS startup and write its results and indexes to files.

— Ability to add new directory paths to monitor after initial launch of Tag Engine. Currently the program launches with a single directory path to monitor. This requires modifying the management of threads and data in Tag Engine. The problem can be partially solved by giving Tag Engine a directory high enough in the hierarchy to monitor a larger number of files.

— Management of removable devices and their FS. inotify does not monitor the /media folder the same as any other directory. It is therefore necessary to be able to detect the connection of a storage device and add a new watch to the mounted directory.

— Carry out more tests, more global than unit and performance tests.

— Create a cache of the last logical requests sent to the server, to speed up the response. The cache should be cleared if the tree or hashmap changes.

8 Conclusion

The two main objectives of this Bachelor project were to study and appropriate the Rust language and to design an efficient and user friendly tag management engine. Rust is a modern, reliable and powerful language. Its strengths are as numerous as the new concepts it introduces compared to a language like C. With an active and serious community, there is the hope that it will be adopted by more and more developers for many types of applications. Although studying Rust took up a lot of the time allotted to this work, it was not wasted time. The constraints imposed by Rust should be a beneficial standard for many languages.

Other topics have been studied, such as indexing methods, extended file attributes, or file system monitoring systems. The limits of extended attributes have been shown (in particular regarding incompatibility with certain file systems or network shares) and of the inotify notification system, chosen for this project. Nevertheless, these two technologies, with the association of Rust, made it possible to surpass existing tag management applications in certain respects. The requested specifications have been completed and questions about the use of Rust in this type of application verified. Thanks to this system, the user can now label his personal files without fear of losing them and can find them easily and quickly using simple logical queries.

This project is the culmination of my studies at hepia, it made me discover a new language full of potential, taught good programming practices and made me progress in the process of designing and creating a system application. The whole project is available at this address: https://github.com/stevenliatti/tagfs.

9 References

[1] Jean-Francois Dockes. Extended attributes and tag file systems. https://www. lesbonscomptes.com/pages/tagfs.html, July 2015. Accessed 04.05.2018.

[2] Paul Ruane alias oniony. Tmsu. https://tmsu.org/. Accessed on 18.05.2018.

[3] Tagsistant. Tagsistant : semantic filesystem for linux. http://www.tagsistant.net/. Accessed on 18.05.2018.

[4] Tagsistent. http://www.tagsistant.net/documents-about-tagsistant/0-8-1-howto , March 2017. Accessed 18.05.2018.

[5] Andrei Marukovich. Taggedfrog - quick start manual. http://lunarfrog.com/projects/taggedfrog/quickstart. Accessed on 18.05.2018.

[6] Tag Spaces. Your offline data manager. https://www.tagspaces.org/. Consulted the 18.05.2018.

[7] TagSpaces. Organize your data with tags. https://docs.tagspaces.org/tagging. Accessed on 18.05.2018.

[8] Greg Shultz. ders An in-depth look at windows vista's virtual fol https://www.techrepublic.com/ technology. an- article/ in-depth-look-at-windows-vistas-virtual-folders-technology/, tobre 2005. Consulté le 21.05.2018. oc

[9] Russell Smith. Manage documents with windows rer using tags and file properties. manage- explode https://www.petri.com/documents-with-windows-explorer-using-tags-and-file-properties, avril 2015. Consulté le 21.05.2018.

[10] Apple team. Os x : Tags help you organize your files. https://support.apple.com/ en-us/HT202754, février 2015. Consulté le 08.05.2018.

[11] John Siracusa. mac os x 10.4 tiger-spotlight. https://arstechnica.com/gadgets/2005/04/macosx-10-4/9/ , April 2005. Accessed 08.05.2018.

[12] John Siracusa. Os x 10.9 mavericks: The ars technica review - tags. https://arstechnica.com/gadgets/2013/10/os-x-10-9/8/ , October 2013. Accessed 08.05.2018.

[13] John Syracuse. OS X 10.9 Mavericks : The Ars Technica Review - Tags Implementation. https://artechnica.com/gadgets/2013/10/os-x-10-9/9/, October 2013. Accessed on 08.05.2018.

[14] John Siracusa. mac os x 10.5 leopard: the ars technica review - fsevents. https://arstechnica.com/gadgets/2007/10/mac-os-x-10-5/7/ , October 2007. Accessed 08.05.2018.

[15] Wikipedia. Big O notation. https://en.wikipedia.org/wiki/Big_O_notation, June 2018. Accessed 2018-06-18.

[16] Wikipedia. A directory represented as a hash table. https://en. wikipedia.org/wiki/Hash_table#/media/File:HASHTB08.svg, June 2015. Accessed on 23.06.2018.

[17] Jean-François Heche. Graphs & Networks. February 2012. Accessed 13.06.2018.

[18] The Rust community crate registry. https://crates.io/. Accessed on 05.05.2018. [19] r/rust. https://www.reddit.com/r/rust/. Accessed on 05.05.2018.

[20] Rust Team. The rust programming language, 2nd edition. https://doc.rust-lang. org/stable/book/second-edition/. Consulté le 25.04.2018.

[21] Computational Geometry Lab. Learning rust with entirely too many linked lists. http: //cglab.ca/~abeinges/blah/too-many-lists/book/. Consulté le 28.04.2018.

[22] Manuel Hoffman. Are we (i)de yet? https://areweideyet.com/. Consulted the 25.04.2018.

[23] The Cargo Book. https://doc.rust-lang.org/cargo/. Accessed on 05.05.2018.

[24] Marcin Baraniecki. Multithreading in rust with mpsc (multi-producer, single consumer) channels. https://bit.ly/2AbJELg, November 2017. Accessed 18.06.2018.

[25] Wikipedia. ext4. https://en.wikipedia.org/wiki/Ext4, may 2018. Consulted the 18.06.2018.

[26] Jeffrey B. Layton. Extended file attributes rock! http://www.linux-mag.com/id/8741/ , June 2011. Accessed 09.05.2018.

[27] Andreas Gruenbacher. attr(5) - linux man page. https://linux.die.net/man/5/attr. Accessed on 09.05.2018.

[28] Jack Hammons. Wsl file system support. https://blogs.msdn.microsoft.com/wsl/2016/06/15/wsl-file-system-support/, June 2016. Accessed 21.05.2018.

[29] John Siracusa. Mac os x 10.4 tiger - extended attributes. https://arstechnica.com/ gadgets/2005/04/macosx-10-4/7/, avril 2005. Consulté le 08.05.2018. [30] freedesktop.org. Guidelines for extended attributes. https://www.freedesktop.org/ wiki/CommonExtendedAttributes/, mai 2018. Consulté le 04.05.2018.

[31] Jean-Francois Dockes. Extended attributes : the good, the not so good, the bad. https://www.lesbonscomptes.com/pages/extattrs.html, juillet 2014. Consulté le 04.05.2018.

[32] inotify - monitoring filesystem events. http://man7.org/linux/man-pages/man7/inotify.7.html, September 2017. Accessed 13.06.2018.

[33] Denis Dordoigne. Exploiting inotify is simple. https://linuxfr.org/news/exploit-inotify-it-is-simple, November 2014. Accessed 13.06.2018.

[34] Michael Kerrisk. Filesystem notification, part 1 : An overview of dnotify and inotify. https://lwn.net/Articles/604686/, juillet 2014. Consulté le 13.06.2018.

[35] Michael Kerrisk. Filesystem notification, part 2: A deeper investigation of inotify. https://lwn.net/Articles/605128/ , July 2014. Accessed 13.06.2018.

[36] fanotify - monitoring filesystem events. http://man7.org/linux/man-pages/man7/fanotify.7.html, September 2017. Accessed 13.06.2018.

[37] socket - create an endpoint for communication. http://man7.org/linux/man-pages/ man2/socket.2.html, septembre 2017. Consulté le 13.06.2018.

[38] Kevin Knapp. clap. https://crates.io/crates/clap, March 2018. Accessed on 14.05.2018.

[39] Steven Allen. xattr. https://crates.io/crates/xattr, July 2017. Accessed on 14.05.2018.

[40] Kevin Knapp. 01a quick example.rs. https://github.com/kbknapp/clap-rs/blob/ master/examples/01a_quick_example.rs, mars 2018. Consulté le 29.06.2018.

[41] Andrew Gallant. walkdir. https://crates.io/crates/walkdir, February 2018. Accessed on 24.05.2018.

[42] petgraph. https://crates.io/crates/petgraph, March 2018. Accessed 24.05.2018.

[43] Daniel Faust, Felix Saparelli, Joe Wilm, George Israel Peña, Michael Maurizi, and Pierre Baillet. notify. https://crates.io/crates/notify, November 2017. Accessed 24.05.2018.

[44] Wikipedia. Reverse Polish notation. https://en.wikipedia.org/wiki/Reverse_Polish_notation , June 2018. Accessed 2018-06-29.

[45] Premshree Pillai. Infix - Postfix. http://scriptasylum.com/tutorials/infix_postfix/algorithms/infix-postfix/ , 2002. Accessed 29.06.2018.

[46] Welcome to graphviz. http://www.graphviz.org/. Accessed on 08.06.2018.

[47] Rust versus C gcc fastest programs. https://benchmarksgame-team.pages.debian. net/benchmarksgame/faster/rust.html. Consulté le 03.07.2018.

[48] Tony Hoare. Null references : The billion dollar mistake. https://www.infoq.com/ presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare, août 2009. Consulté le 03.07.2018.

[49] Rust Leipzig. Idiomatic tree and graph like structures in rust. https://rust-leipzig. github.io/architecture/2016/12/20/idiomatic-trees-in-rust/, December 2016. Accessed 08.06.2018.

[50] Wikipedia. Region-based memory management. https://en.wikipedia.org/wiki/ Region-based_memory_management, juin 2018. Consulté le 18.06.2018.

[51] Nick Cameron. Graphs and arena allocation. https://aminb.gitbooks.io/ rust-for-c/content/graphs/ index.html, juin 2015. Consulté le 08.06.2018.

[52] Russell Cohen. Why writing a linked list in (safe) rust is so damned hard. https://rcoh. me/posts/ rust-linked-list-basically-impossible/, février 2018. Consulté le 08.06.2018.

[53] Simon Sapin. Borrow cycles in rust: arenas vs. drop-checking. https://exyr.org/2018/rust-arenas-vs-dropck/ , February 2018. Accessed 08.06.2018.

[54] Matthias Endler. Of boxes and trees - smart pointers in rust. https://matthias-endler.de/2017/ boxes-and-trees/ , August 2017. Accessed 08.06.2018.