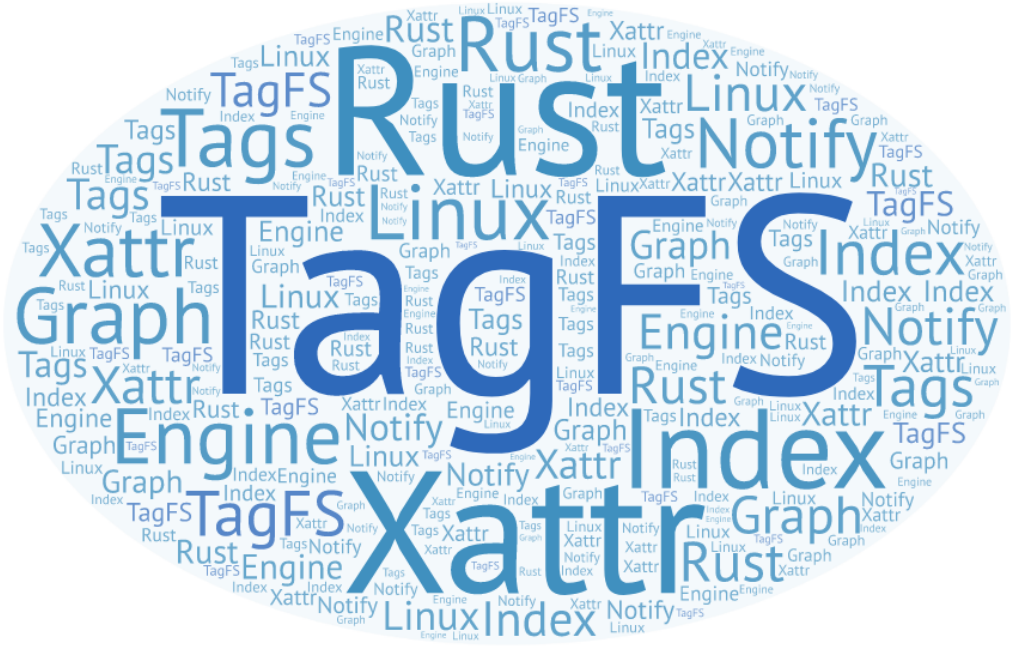
h e p i a
Haute école du paysage, d'ingénierie
et d'architecture de Genève

Hes·SO GENÈVE
Haute Ecole Spécialisée
de Suisse occidentale

TagFS
Système d'étiquetage des fichiers avec Rust



Thèse de Bachelor présentée par

Monsieur Steven LIATTI

pour l'obtention du titre de Bachelor of Science HES-SO en

Ingénierie des technologies de l'information avec orientation en Logiciels et systèmes complexes

Septembre 2018

Professeur HES responsable

Florent GLÜCK

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

printemps 2018
Session de bachelor

INGÉNIERIE DES TECHNOLOGIES DE L'INFORMATION

ORIENTATION – LOGICIELS ET SYSTÈMES COMPLEXES

MOTEUR DE TAGGING ET D'INDEXATION DES FICHIERS AVEC RUST

Descriptif :

Les systèmes de *tagging* de fichiers existants que l'on peut trouver dans le monde Linux aujourd'hui utilisent des bases de données et/ou des fichiers aditionnels pour stocker les meta données des fichiers taggés. La problématique liée au fait que les fichiers et leurs méta-données ne sont pas stockés dans la même entité (i.e. le fichier), complique les implémentations et les rend sub-optimales en terme de complexité, performance, latence et robustesses.
Une solution alternative à ces approches traditionnelles serait de tirer parti des *extended attributes* présents dans la plupart des systèmes de fichiers disponibles dans le noyau Linux (ext4, xfs, etc.). Les *extended attributes* permettent, pour chaque fichier, de stocker ou extraire un ensemble de méta-données, le tout de manière efficace.
Le but premier de ce projet est de concevoir et développer un « moteur de gestion de tags » pouvant aisément et efficacement gérer des dizaines ou des centaines de milliers de fichiers et *tags* associés.
Le but secondaire du projet, est d'en réaliser l'implémentation dans le langage Rust. En effet, le langage Rust est un langage système moderne, digne successeur du langage C, et conçu pour être extrêment fiable, robuste et performant.
Les mécanismes d'indexation seront étudiés et les systèmes de surveillance de fichiers seront investigués. En effet, le système devra s'assurer de surveiller les fichiers modifiés, créés, ou supprimés afin d'indexer les tags avec un minimum de latence, le but étant d'offrir des performances aussi proches du temps réel que possible.
Enfin, si le temps le permet, le système développé sera intégré à un gestionnaire de fichiers populaire dans le monde Linux (Nautilus, Thunar, Konqueror, etc.).

Travail demandé :

- Etat de l'art.
- Etude et prise en main du langage Rust et familiarisation avec celui-ci dans le contexte système du projet.
- Analyse des *extended attributes* ; en particulier, étude de leur comportement lors des opérations d'accès courantes (cp, mv, rsync, copies depuis un gestionnaire de fichier, etc.) afin d'éviter la perte involontaire de tags au cas où certaines opérations ne maintienderaient pas certains attributs.
- Analyse et sélection des systèmes de notification du système de fichiers.
- Analyse et sélection des algorithmes d'indexation et de recherche.
- Conception et implémentation du système.
- Analyse du système implémenté (mesures de performances, etc.).
- Discussion de l'utilisation du langage Rust par rapport à une implémentation traditionnelle en C.
- Démonstrateur.
- Si le temps le permet : intégration à un gestionnaire de fichiers.

Candidat :

M. LIATTI STEVEN

Filière d'études : ITI

Professeur(s) responsable(s) :

GLUCK Florent

En collaboration avec :
Travail de bachelor soumis à une convention de stage en entreprise : **non**
Travail de bachelor soumis à un contrat de confidentialité : **non**

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

Printemps 2018
Session de bachelor

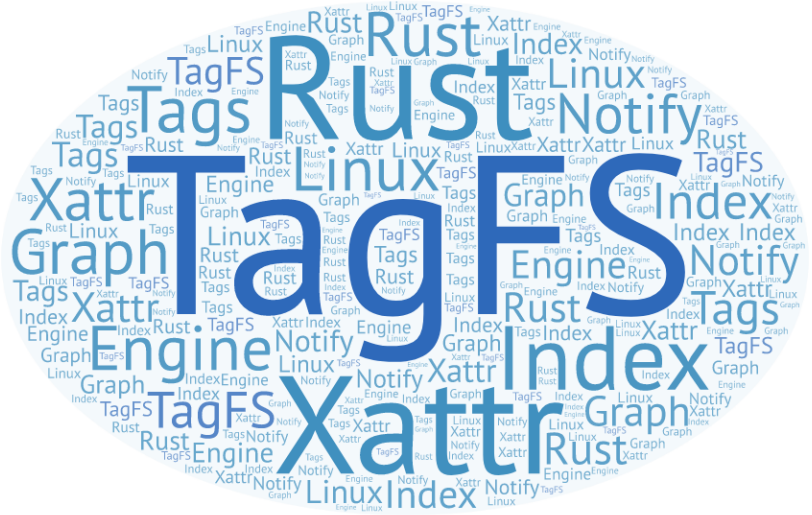
Résumé :

Avec l'augmentation de la puissance de calcul et la capacité de stockage pour un prix raisonnable, nos ordinateurs personnels gèrent des quantités de fichiers très importantes, de l'ordre de centaines de milliers ou du million de fichiers. Comment alors retrouver rapidement un fichier correspondant à certains critères sans se rappeler de son emplacement sur le disque ? Une solution à ce problème est de donner la possibilité à l'utilisateur d'apposer une ou plusieurs étiquettes, ou "tags", sur ses fichiers et de lui fournir une interface avec laquelle il pourra aisément retrouver ses fichiers. Il est important que les tags soient stockés dans les attributs des fichiers pour qu'ils ne soient pas perdus lors des manipulations des fichiers. L'utilisateur doit pouvoir rapidement retrouver ses fichiers par recherche par tags.

Dans le but de résoudre ces problèmes, la solution proposée dans ce projet est un duo de programmes, Tag Manager et Tag Engine, formant le système TagFS. L'utilisateur peut gérer les tags associés aux fichiers et exécuter des recherches de fichiers par tags grâce à Tag Manager. Tag Engine travaille en arrière-plan en indexant et surveillant l'arborescence des fichiers et tags de l'utilisateur en temps réel et avec très peu de latence.

Pour réaliser ce travail, de nombreuses technologies ont dû être étudiées et mises en œuvre, en premier lieu le langage de programmation Rust. Rust est un langage moderne, fiable et performant, digne successeur de C. Les mécanismes d'indexation et de surveillance du système de fichiers ainsi que les attributs étendus pour le stockage des tags ont également été analysés.

Tout le projet est disponible sur GitHub, à l'adresse suivante : https://github.com/stevenliatti/tagfs.



Candidat :

M. LIATTI STEVEN
Filière d'études : ITI

Professeur(s) responsable(s) :

Glück Florent

Travail de bachelor soumis à une convention de stage en entreprise : non
Travail de bachelor soumis à un contrat de confidentialité : non

Table des matières

Table des figures

Table des tables

Table des listings de code source

Conventions typographiques

Lors de la rédaction de ce document, les conventions typographiques ci-dessous ont été adoptées.

- Tous les mots empruntés à la langue anglaise ou latine ont été écrits en *italique*.
- Toute référence à un nom de fichier (ou répertoire), un chemin d'accès, une utilisation de paramètre, variable, commande utilisable par l'utilisateur, ou extrait de code source est écrite avec une police d'écriture à `chasse fixe`.
- Tout extrait de fichier ou de code est écrit selon le format suivant :

```rust
fn main() {
    println!("Hello, world!");
}
```

- Dans les listings, les lignes précédées d'un "$" sont exécutées dans un shell.

Structure du document

Le présent document commence par l'introduction, rappelant les motivations du projet et les buts à atteindre. Il aborde ensuite les applications existantes fournissant un service similaire au but du projet, mais avec quelques limitations pour la plupart. Il poursuit par une section sur l'architecture logicielle du système final, avec ses différents composants et besoins. Après, une analyse technologique du langage Rust et des outils utilisés pour la réalisation technique est réalisée. La section suivante illustre la réalisation technique en elle-même, avec les deux programmes conçus. Le système est testé dans la section d'après, expliquant le protocole et le résultat des tests. Finalement, le document se termine par une section de discussion des résultats, de l'état du projet, des améliorations futures et des différences entre Rust et C et une dernière section de conclusion. Les références se trouvent à la toute fin du document.

Remerciements

Mes remerciements vont en premier lieu à ma compagne, Marie Bessat, pour son infinie patience et ses encouragements. Je remercie aussi ma famille et mes proches pour leur soutien tout au long de mes études et lors de la réalisation de ce travail. Je tiens à remercier tous mes professeurs pour leurs enseignements et tout particulièrement M. Florent Glück pour le suivi du projet et ses précieux conseils ainsi que M. Orestis Malaspinas pour les conseils sur le langage Rust. J'aimerais remercier aussi M. Joël Cavat, assistant ITI à l'hepia, pour m'avoir conseillé et m'avoir prêté son très bon cours sur les Graphes et Réseaux de M. Jean-Francois Hêche, que je remercie indirectement. Je tiens finalement à remercier mes collègues de classe pour leur camaraderie et aide durant mes études et mon travail de bachelor.

Acronymes

API *Application Programming Interface*, Interface de programmation : services offerts par un programme producteur à d'autres programmes consommateurs. 17, 49, 51, 53, 55, 64

CLI *Command Line Interface*, Interface en ligne de commande : interface homme-machine en mode texte : l'utilisateur entre des commandes dans un terminal et l'ordinateur répond en exécutant les ordres de l'utilisateur et en affichant le résultat de l'opération. 13, 28, 47, 53

FS *File System*, Système de fichiers : organisation logique des fichiers physiques sur le disque. 12–14, 18–21, 24, 25, 27, 47, 49–51, 64, 67, 85

GUI *Graphical User Interface*, Interface graphique : moyen d'intéragir avec un logiciel où les contrôles et objets sont manipulables. S'oppose à l'interface en ligne de commande (CLI). 15

OS *Operating System*, Système d'exploitation : couche logicielle entre le matériel d'un ordinateur et les applications utilisateurs. Offre des abstractions pour la gestion des processus, des fichiers et des périphériques entre autres. 11, 13, 16, 39, 47, 53, 64, 76, 85

SYSCALL *System call*, Appel système : lorsqu'un programme a besoin d'un accès privilégié à certaines parties du système d'exploitation (système de fichiers, mémoire, périphériques), il demande au noyau d'exécuter l'opération voulue pour lui. 49–53

XATTR *Extended Attributes*, Attributs étendus : voir section 4.2. 11–13, 17, 18, 47–49, 53, 55, 60, 63

1 Introduction

1.1 Motivations

Avec l'augmentation de la puissance de calcul et la capacité de stockage de masse grandissante à un prix raisonnable, nos ordinateurs personnels gèrent des quantités de fichiers très importantes, de l'ordre du millier ou du million de fichiers. Que ce soit des images, des documents ou de la musique, les réserves de stockage de fichiers semblent sans fin. Dès lors se pose la question de l'organisation de ces nombreux fichiers. Comment doit-on ordonner ses photos personnelles ? Par date, par lieu, par thème ? Ce sont trois bonnes réponses, mais malheureusement pour l'utilisateur, les systèmes d'exploitation (OS) d'aujourd'hui ne proposent qu'une seule manière native d'organiser ses fichiers : la classique hiérarchie de répertoires, sous-répertoires et fichiers, sous la forme d'une arborescence.

Comment retrouver rapidement la photo de votre chat dormant sur votre balcon au début de sa vie dans une masse de plus de 10'000 images ? Comment classer son répertoire d'études, un répertoire pour les cours, un autre pour les travaux pratiques, ou pour chaque cours, deux sous-répertoires "théorie" et "pratique" ? Comment récupérer une chanson sans connaître son titre ni l'artiste mais en connaissant le genre ? Une solution à ces problèmes est de donner la possibilité à l'utilisateur d'apposer une ou plusieurs étiquettes, ou "tags", sur ses fichiers et de lui fournir une interface avec laquelle il pourra aisément retrouver ses fichiers. Il doit garder le contrôle sur ses fichiers et pouvoir les manipuler comme il l'a toujours fait. Les tags doivent être stockés avec les fichiers, pour qu'ils ne soient pas perdus en cas de grand changement dans le système. L'utilisation des attributs étendus, ou *extended attributes* (XATTR), est la manière la plus naturelle de répondre à ce dernier besoin : les tags "voyagent" ainsi avec les fichiers. Nous verrons qu'il existe des applications résolvant en partie ce problème. Finalement, cette interface doit être performante et fiable.

Ces deux qualificatifs, performant et fiable, résument le langage de programmation Rust. Rust est un langage de programmation moderne, fort d'une communauté grandissante et passionnée. Il est très performant, proche ou meilleur que C selon les situations. Il est fiable grâce à ses règles strictes sur l'utilisation de la mémoire et son compilateur très intelligent. C'est un langage totalement adapté à notre situation. À travers ce travail, le lecteur pourra se faire une illustration des possibilités offertes par Rust.

Le but de ce travail est donc de concevoir et développer un moteur de gestion des tags répondant aux besoins cités précédemment et d'apprendre les notions de Rust nécessaires à sa réalisation.

1.2 Buts

Avec plus de détails, les buts de ce projet sont les suivants :

— Étudier et s'approprier le langage Rust pour la réalisation d'une application système sous Linux.

— Répertorier les applications existantes permettant d'étiqueter les fichiers.

— Étudier les XATTR lors des manipulation courantes sur les fichiers.

— Explorer les méthodes de surveillance du système de fichiers (FS).

— Analyser les moyens d'indexer une arborescence de fichiers.

— Concevoir et implémenter un système répondant aux motivations. Il devra être performant, utilisable en temps réel et gérer de nombreux fichiers, répertoires et tags.

— Mesurer les performances de ce système.

— Réaliser une démonstration.

2 Analyse de l'existant

Dans cette section, nous allons analyser les principales solutions existantes, qu'elles soient sous la forme d'applications utilisateur ou intégrées directement dans un système d'exploitation (OS). Jean-Francois Dockes en dresse également une liste avec avantages et inconvénients sur son site [1]. Ce que nous recherchons est une application *open source*, fonctionnant sur Linux, stockant les tags dans les *extended attributes* (XATTR), ne modifiant pas les fichiers et performante.

2.1 Applications utilisateur

2.1.1 TMSU

TMSU [2] est un outil en ligne de commande (CLI) qui permet d'attribuer des tags à des fichiers et d'exécuter des recherches par tags. On commence par initialiser TMSU dans le répertoire choisi. Une commande liste les tags associés à un ou plusieurs fichiers et une autre liste les fichiers qui possèdent le ou les tags donnés. TMSU offre la possibilité à l'utilisateur de "monter" un système de fichiers (FS) virtuel avec FUSE (Filesystem in UserSpacE). L'outil est rapide et efficace, mais il comporte quelques défauts :

— Pas d'interface graphique.

— Dépendance à FUSE pour monter le FS virtuel.

— Stockage des tags dans une base de données SQLite : si la base est perdue, les tags le sont également.

2.1.2 Tagsistant

Tagsistant [3] est autre outil en ligne de commande (CLI) de gestion de tags. Il dépend de FUSE et d'une base de données (SQLite ou MySQL) pour fonctionner. Comme pour TMSU, il faut donner un répertoire à Tagsistant pour son usage interne. À l'intérieur de ce dernier, se trouvent différents répertoires :

```
/
├── alias -- Répertoire contenant les requêtes les plus courantes
├── archive -- Répertoire listant les fichiers
├── relations -- Répertoire contenant les relations entre les tags et fichiers
├── stats -- Répertoire contenant des infos sur l'utilisation de Tagsistant
├── store -- Répertoire où sont gérés les fichiers et ajoutés les tags
└── tags -- Répertoire de gestion des tags
```

Chaque répertoire a un rôle bien précis. Tout se fait avec le terminal et des commandes usuelles (`cp`, `ls`, `mkdir`, etc.). Dans Tagsistant, un répertoire créé dans le répertoire `tags` correspond à un tag. On se retrouve finalement avec une arborescence de tags et de fichiers

[4]. Bien que cet outil soit performant d'un point de vue de la rapidité d'exécution, il comporte les défauts de TMSU ainsi que des nouveaux :

— Pas d'interface graphique.

— Dépendance à FUSE pour monter le FS virtuel.

— Stockage des tags dans une base de données : si la base est perdue, les tags le sont également.

— Utilisation des différents répertoires peu intuitive.

— Tous les fichiers sont contenus dans un seul répertoire et leur nom est modifié pour les besoins internes de l'application. Obligation de passer par l'application pour accéder aux fichiers.

2.1.3 TaggedFrog

TaggedFrog [5] est un programme disponible sur Windows uniquement et ne partage pas ses sources. Son fonctionnement interne n'est pas documenté. L'interface est agréable, on peut ajouter des fichiers par *Drag & Drop*. L'interface créé au fur et à mesure un "nuage" de tags, comme on peut le retrouver sur certains sites web. On peut exécuter des recherches sur les tags et les fichiers. On peut supposer que TaggedFrog maintient une base de données des tags associés aux fichiers, ce qui ne correspond à nouveau pas à nos besoins.

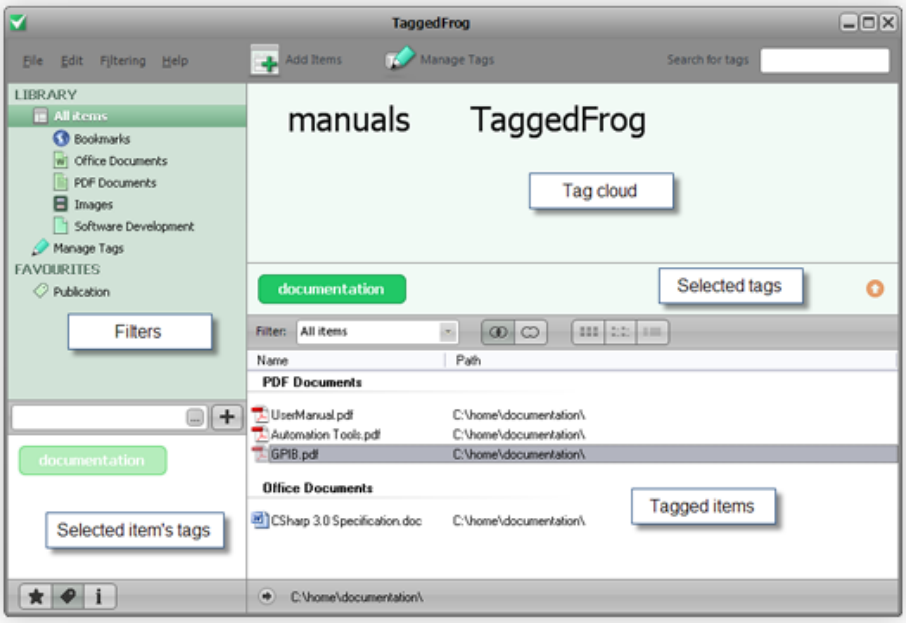


FIGURE 1 – TaggedFrog en utilisation [5]

2.1.4 TagSpaces

TagSpaces [6] est un programme avec une interface graphique (GUI) permettant d'étiqueter ses fichiers avec des tags. L'application est agréable à utiliser, on commence par connecter un emplacement qui fera office de répertoire de destination aux fichiers. On peut ajouter ou créer des fichiers depuis l'application. Les fichiers existants ajoutés depuis l'application sont copiés dans le répertoire (cela créé donc un doublon). Sur le panneau de gauche se situe la zone de gestion des tags. TagSpaces ajoute automatiquement certains tags dits "intelligents" aux fichiers nouvellement créés avec l'application (par exemple un tag avec la date de création).

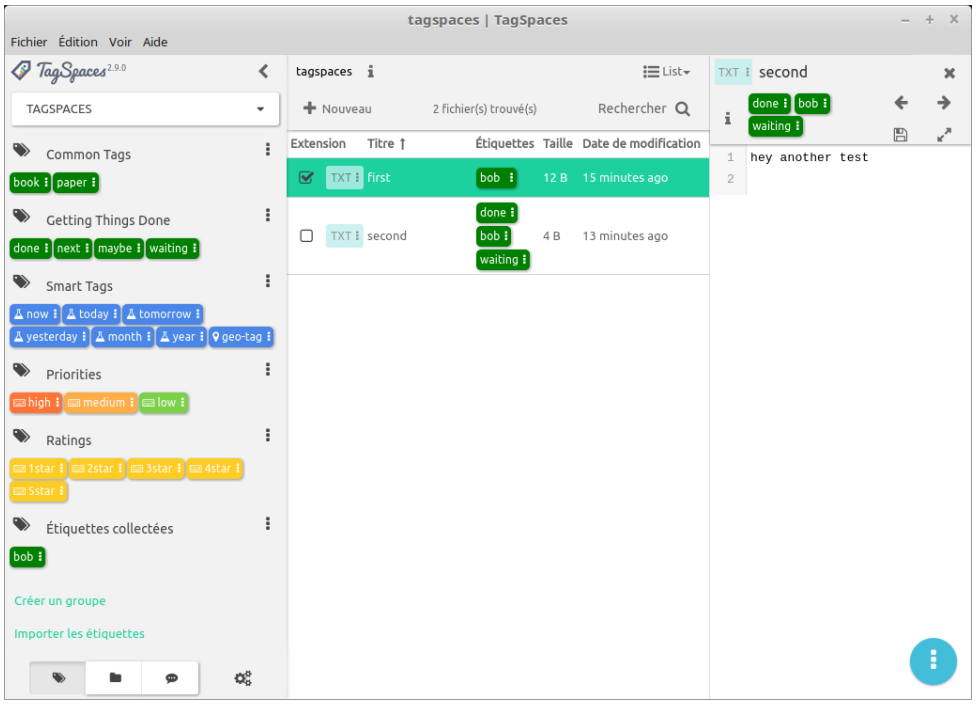


FIGURE 2 – TagSpaces en utilisation

Globalement, l'application est fonctionnelle et *user friendly*. Cependant, deux points noirs sont à déplorer :

1. L'application copie les fichiers déjà existants sélectionnés par l'utilisateur, ce qui créé une contrainte supplémentaire dans la gestion de ses fichiers personnels.
2. TagSpaces stocke les tags directement dans le nom du fichier, modifiant ainsi son nom [7]. Bien que pratique dans le cas d'une synchronisation à l'aide d'un service cloud, le fichier devient dépendant de TagSpaces. Si l'utilisateur décide de changer son nom sans respecter la nomenclature interne, il risque de perdre les tags associés au fichier.

2.2 Fonctionnalités disponibles dans l'OS

2.2.1 Windows

À partir de Windows Vista, Microsoft a donné la possibilité aux utilisateurs d'ajouter des méta-données aux fichiers ; parmi ces méta-données se trouvent les tags. Il existe une fonctionnalité appelée *Search Folder* qui permet de créer un répertoire virtuel contenant le résultat d'une recherche sur les noms de fichiers ou d'autres critères [8]. Depuis Windows 8, l'utilisateur a la possibilité d'ajouter des méta-données à certains types de fichiers (ceux de la suite office par exemple), dont des tags. Il peut par la suite exécuter des recherches ciblées via l'explorateur de fichiers Windows du type `meta:value` [9]. C'est dommage que Windows ne prenne pas en compte davantage de types de fichiers, comme les PDFs ou les fichiers `.txt`.

2.2.2 macOS

macOS possède son propre système pour étiqueter des fichiers. Il est intégré depuis la version OS X 10.9 Mavericks. Depuis l'explorateur de fichiers, l'utilisateur a la possibilité d'ajouter, modifier, supprimer et rechercher des tags. Les fichiers peuvent avoir plusieurs tags associés. Un code couleur permet de plus facilement se souvenir et visualiser les tags attribués. Dans l'explorateur de fichiers, les tags se retrouvent sur le bas côté, pour y accéder plus rapidement.

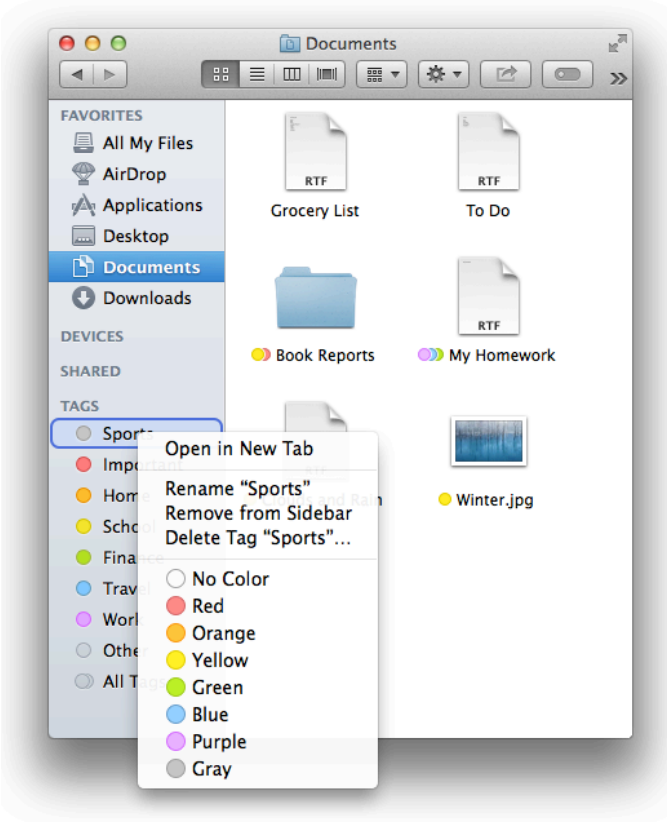


FIGURE 3 – Vue et gestion d'un tag dans le Finder macOS [10]

Lorsque l'on clique sur un tag, une recherche Spotlight est effectuée. Spotlight est le moteur de recherche interne à macOS. Spotlight garde un index des tags, fournissant un accès rapide aux fichiers correspondants [11]. Tous ces tags peuvent se synchroniser sur les différents "iDevices" via iCloud. Finalement, un menu de réglages permet la gestion des tags (affichage, suppression, etc.) [10], [12]. L'implémentation de ce système utilise les XATTR pour stocker les tags. Les différents tags se trouvent dans l'attribut `kMDItemUserTags`, listés les uns à la suite des autres. Via le Terminal, à l'aide de la commande `mdls`, nous pouvons afficher la liste des tags associés à un fichier, nommé "Hello" pour l'exemple :

```
% mdls -name kMDItemUserTags Hello
kMDItemUserTags = (
    Green,
    Red,
    Essential
)
```

Listing 1 – `mdls` listant les tags d'un fichier sous macOS [13]

Ici, le fichier `Hello` est étiqueté avec trois tags, "Green", "Red" et "Essential". Le fait que l'indexation est réalisée avec Spotlight implique une réindexation des fichiers dans le cas d'un changement de nom pour un tag donné sous macOS. Le framework système `FSEvents` donne une solution partielle : c'est une API (utilisée également par Spotlight) qui offre aux applications la possibilité d'être notifiées si un changement a eu lieu sur un répertoire (un événement toutes les 30 secondes). `FSEvents` maintient des logs de ces changements dans des fichiers, les applications peuvent ainsi retrouver l'historique des changements quand elles le souhaitent [14]. Les seuls points négatifs de cette implémentation, c'est qu'elle n'est pas *open source*, qu'elle n'existe que sur macOS et qu'elle n'est pas écrite en Rust. Mais c'est la solution vers laquelle ce projet essaie de se rapprocher car elle correspond aux critères sur les XATTR et le fonctionnement interne.

3 Architecture

Cette section présente l'architecture logicielle choisie pour implémenter le système. Il est composé de quatre entités distinctes, décrites dans les sous-sections suivantes.

3.1 Gestion des tags

La gestion physiques des tags stockés dans les attributs étendus (XATTR) est une fonctionnalité indépendante du reste du système. Comme expliqué dans la section 4.2, des outils système existent pour manipuler les XATTR des fichiers. Cependant, pour offrir un plus haut niveau d'abstraction, une cohérence sur le nommage des tags pour l'indexation et plus de confort pour l'utilisateur final, un outil devient nécessaire pour la gestion des tags. Cet outil se présente, sous sa forme de base, comme un programme en ligne de commande. Il doit, au minimum, offrir la possibilité de lire les tags contenus dans les fichiers et ajouter et supprimer les tags donnés en entrée par l'utilisateur. Il devra pouvoir manipuler plusieurs tags et fichiers simultanément. D'un point de vue algorithmique et structures de données, cette partie n'est pas particulièrement ardue.

3.2 Indexation des fichiers et des tags

L'indexation des fichiers et des tags associés est l'un des deux piliers du système. Il faut créer un index des relations entre les tags et les fichiers. Le terme "index" utilisé ici ne prend pas son sens littéraire exact, *id est* la liste des termes importants d'un livre avec la liste des pages dans lesquels ils apparaissent. Dans notre situation, il se rapproche plus de son utilisation pour les bases de données : c'est une structure de données permettant de retrouver rapidement les données, dans notre cas, récupérer rapidement la relation entre tags et fichiers. Deux architectures ont été imaginées pour l'indexation des tags et des fichiers, elles sont décrites dans les deux sous-sections suivantes. La table 1 liste les événements qui se produisent lors d'une utilisation normale du système de fichiers (FS) en leur attribuant un numéro. Ces numéros sont repris et utilisés pour alléger la lecture des tables 2 et 3.

Numéro	Événement
1	Ajout d'un tag à un fichier ou à un répertoire
2	Suppression d'un tag d'un fichier ou d'un répertoire
3	Renommage d'un tag
4	Ajout d'un fichier dans l'arborescence surveillée
5	Ajout d'un répertoire dans l'arborescence surveillée
6	Déplacement ou renommage d'un fichier dans l'arborescence surveillée
7	Déplacement ou renommage d'un répertoire dans l'arborescence surveillée
8	Suppression d'un fichier de l'arborescence surveillée
9	Suppression d'un répertoire de l'arborescence surveillée

TABLE 1 – Événements survenant sur le FS

Les explications suivantes mentionnent la notion de **complexité en temps** (le nombre d'opérations nécessaires à l'accomplissement de l'algorithme, en fonction de la taille des entrées) avec la notation de Landau, ou *Big O* [15].

3.2.1 Indexation avec table de hachage et arbre

La première version de l'architecture de l'indexation, comportant deux structures de données, est la suivante :

1. Une table de hachage (ou *hashmap*) associant un tag (son nom, sous forme de chaîne de caractères) à un ensemble (au sens mathématique) de chemins de fichiers sur le disque.
2. Un arbre, correspondant à l'arborescence des fichiers, avec comme noeuds les répertoires, sous-répertoires et fichiers. Le répertoire à surveiller représente la racine de l'arbre. Les liens entre les noeuds représentent le contenu d'un répertoire. Les données du noeud contiennent le nom du fichier ou du répertoire et l'ensemble des tags associés au fichier.

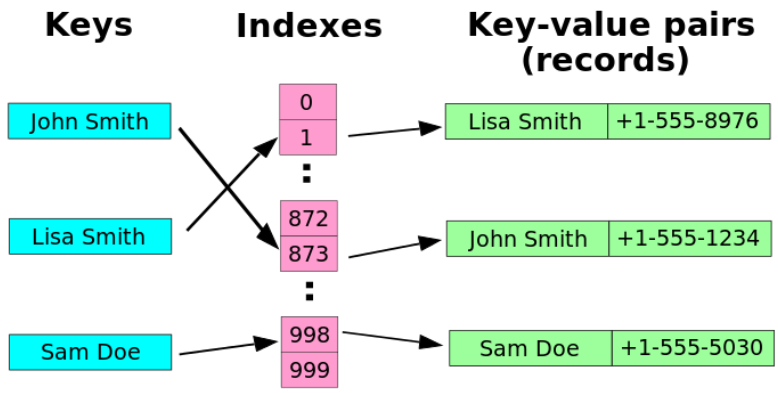


FIGURE 4 – Un annuaire représenté comme une table de hachage - [16]

Une table de hachage est un tableau associatif. Les composantes de l'association sont la "clé", reliée à une ou plusieurs "valeurs". Pour insérer, accéder ou supprimer une entrée de la table, il faut calculer le "hash" de la clé, *id est* son empreinte unique. Sur la figure 4, nous apercevons les clés en bleu, le résultat du hash en rouge et les valeurs associées en vert. Le risque que deux clés ou plus produisent une même empreinte s'appelle une "collision", c'est pour cela qu'une bonne implémentation d'une table de hachage doit non seulement utiliser une bonne fonction de hachage mais aussi une manière de résoudre les collisions. C'est ainsi que les trois opérations ci-dessus peuvent être réalisées, en moyenne, en temps constant (O(1)) et dans le pire des cas (si les collisions s'enchainent) en temps linéaire (O(n)). Dans notre cas, l'utilisation d'une table de hachage pour stocker la relation entre un tag et ses fichiers est efficace lorsqu'une recherche par tags est demandée. De plus, en associant un ensemble de chemins de fichiers, des opérations ensemblistes (union, intersection) peuvent être réalisées lorsque une recherche impliquant plusieurs tags est effectuée.

L'arbre, au sens informatique, est une représentation de la hiérarchie du FS dans notre cas. Prenons comme exemple la hiérarchie suivante :

```
home
├── root
└── user
    ├── docs
    │   ├── graph.pdf
    │   └── report.tex
    ├── images
    │   ├── img1.png
    │   └── img2.png
    └── music
        └── kiss.mp3
```

Elle peut être obtenue grâce à la commande `tree` sous Linux par exemple. La même représentation sous forme d'un arbre est illustrée sur la figure 5 :

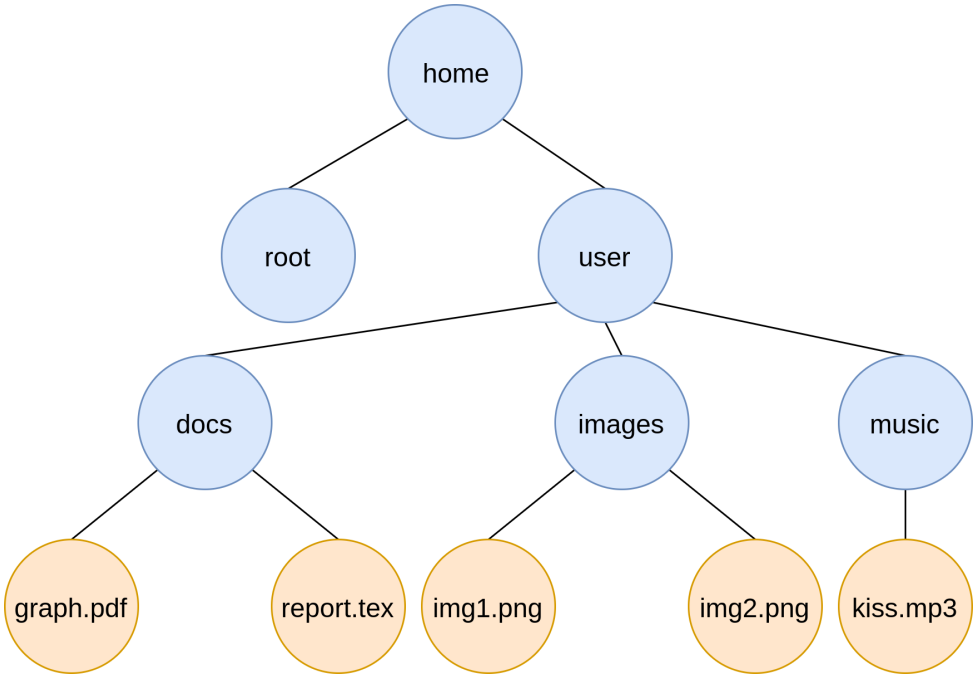


FIGURE 5 – Représentation sous forme d'arbre d'une hiérarchie de fichiers et répertoires

Le noeud "home" représente la racine de l'arbre. Chaque noeud représente soit un fichier (en orange), soit un répertoire (en bleu) sur le disque. Chaque répertoire peut être vu comme un sous-arbre de l'arbre principal. Du point de vue programmatoire, un noeud serait défini, au minimum, comme une structure de données contenant un champ "données" (dans notre cas, le nom du fichier/répertoire et l'ensemble de ses tags) et un champ "enfants", une liste ou un ensemble de pointeurs vers les noeuds enfants. Dans le cas présent, seuls les noeuds répertoires pointent vers des noeuds répertoires ou fichiers enfants, les fichiers n'auraient qu'une liste vide de pointeurs.

La table 2 donne, pour chaque événement survenu sur le FS, les opérations à exécuter pour les deux structures de données (la table de hachage et l'arbre) et une approximation de la complexité. Les événements sont représentés par leurs numéros, en table 1. Les variables suivantes sont définies :

— c = Opération constante.

— p = Profondeur de l'arbre.

— t = Nombre de tags.

Numéro	Opérations sur la *hashmap*	Opérations sur l'arbre
1	Si tag non présent, ajouter le tag comme clé et ajouter le chemin du fichier à l'ensemble -> $O(c)$	Parcourir l'arbre à la recherche du fichier et ajouter le tag à l'ensemble des tags existants -> $O(p * c)$
2	Supprimer le fichier de l'ensemble des chemins de fichiers associés au tag -> $O(c)$	Parcourir l'arbre à la recherche du fichier et supprimer le tag de l'ensemble des tags existants -> $O(p * c)$
3	Supprimer la clé et réinsérer la nouvelle clé et l'ensemble associé -> $O(c)$	Pour l'ensemble des chemins récupérés avec la *hashmap*, modifier les noeuds correspondants -> $O(t * c)$
4	Pour tous les tags du fichier, ajouter au besoin le tag et lui associer le chemin de fichier -> $O(t * c)$	Parcourir l'arbre à la recherche du répertoire parent du fichier, ajouter le nouveau noeud et l'ensemble de ses tags -> $O(p*c)$
5	Identique à la ligne précédente	Parcourir l'arbre à la recherche du répertoire parent, ajouter le nouveau noeud et l'ensemble de ses tags, puis, récursivement, ajouter ses enfants (sous-répertoires et fichiers) -> $\approx O(p^2 * c)$
6	Pour tous les tags du fichier, associer le nouveau chemin de fichier -> $O(t * c)$	Parcourir l'arbre à la recherche du parent et changement du lien du noeud avec son parent / simple renommage du nom dans l'étiquette -> $O(p * c)$
7	Pour tous les tags de tous les sous-répertoires et fichiers, associer le nouveau chemin de fichier -> $O(t * p * c)$	Identique à la ligne précédente
8	Pour tous les tags du fichier, supprimer le chemin de fichier -> $O(t*c)$	Parcourir l'arbre à la recherche du parent et suppression du lien et du noeud -> $O(p * c)$
9	Pour tous les tags de tous les sous-répertoires et fichiers, supprimer le chemin de fichier -> $O(t * p * c)$	Parcourir l'arbre à la recherche du répertoire parent, supprimer le noeud, l'ensemble de ses tags, et récursivement, ses enfants (sous-répertoires et fichiers) -> $\approx O(p^2 * c)$

TABLE 2 – Opérations et complexité, première architecture

Cette version a été en partie abandonnée et adaptée pour deux raisons majeures :

1. Avoir deux structures de données interdépendantes augmente la complexité des opérations de mise à jour (ajout, déplacement, suppression de fichiers et tags).

2. L'implémentation s'est avérée plus difficile que prévue, du fait de certaines contraintes de Rust (voir section 7.2).

3.2.2 Indexation avec un graphe et une table de hachage

Pendant l'implémentation de cette partie du programme, une nouvelle architecture a été imaginée. Elle reprend les bases de la précédente, mais simplifie la structure de données. Plutôt que de maintenir deux structures différentes, cette solution propose une structure de données principale, secondée par une structure secondaire, optionnelle, mais néanmoins efficace :

1. Un graphe, avec un noeud représentant soit un répertoire, soit un fichier soit un tag. Chaque noeud est une structure de données comportant un nom et type et est identifié de manière unique. Grâce à cet identifiant, les noeuds sont facilement accessibles.
2. Une table de hachage, associant le nom d'un tag à son identifiant unique en tant que noeud du graphe.

Un graphe représente un réseau de noeuds qui peuvent être reliés les uns aux autres. Jean-François Hêche, professeur à la heig-vd, donne les définitions des graphes non orientés et orientés dans son cours sur les "Graphes et Réseaux". Commençons par définir ce qu'est un graphe non orienté : "Un graphe non orienté est une structure formée d'un ensemble V, dont les éléments sont appelés les sommets ou les noeuds du graphe, et d'un ensemble E, dont les éléments sont appelés les arêtes du graphe, et telle qu'à chaque arête est associée une paire de sommets de V appelés les extrémités de l'arête." (Hêche, page 1, [17]). La figure 6 montre un exemple d'un tel graphe.

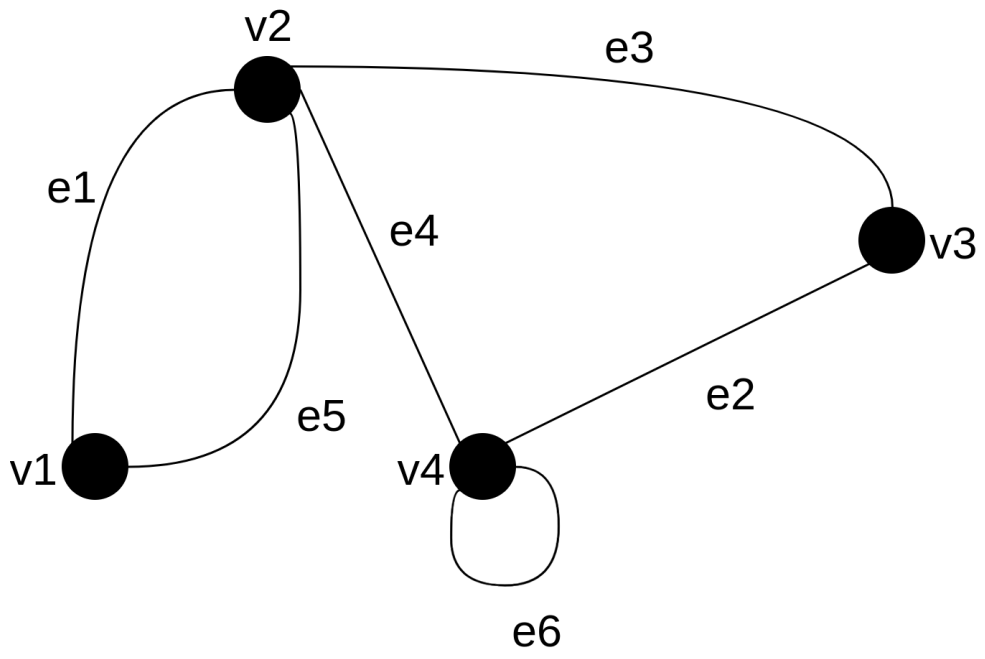


FIGURE 6 – Graphe non orienté

Un graphe orienté est semblable à un graphe non-orienté. La seule différence est qu'une direction est donnée au lien entre deux noeuds et ce lien ne se nomme plus "arête" mais "arc". La figure 7 montre un exemple d'un tel graphe.

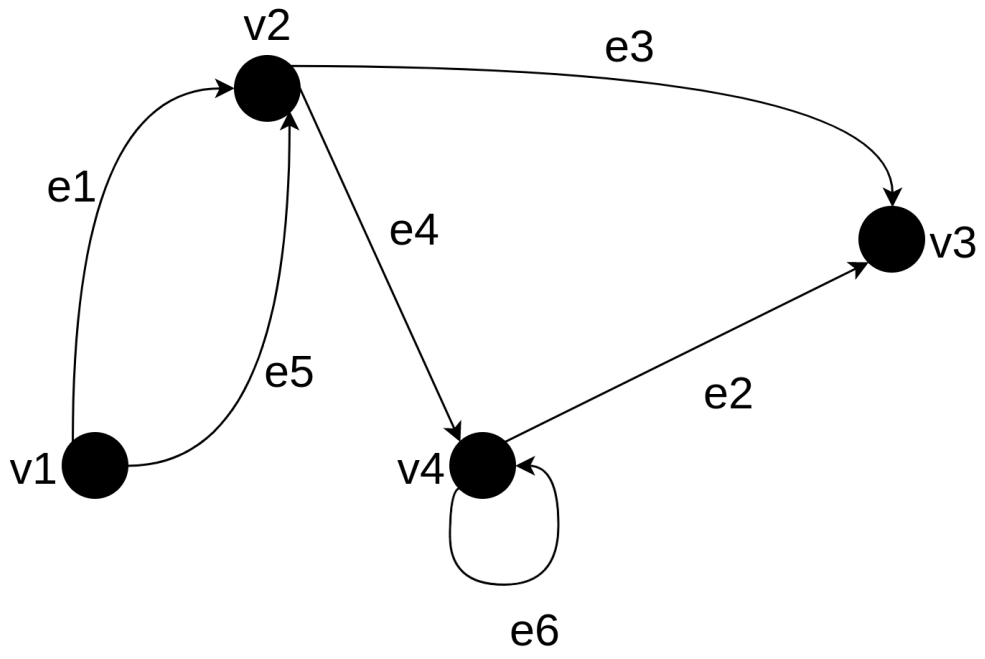


FIGURE 7 – Graphe orienté

Un graphe est donc un ensemble de noeuds reliés par des arêtes ou des arcs, selon si le graphe est orienté ou non. Dans notre cas, l'utilisation d'un graphe n'est pas si éloignée de celle d'un arbre. Par ailleurs, selon la théorie des graphes, "un arbre est un graphe sans cycle et connexe" (Hêche, page 33, [17]). "Sans cycle" signifie qu'un parcours du graphe est possible de telle sorte à ce que le noeud de départ et d'arrivée soient différents. "Connexe" définit un graphe tel que pour chaque paire de noeuds du graphe il existe un chemin les reliant. L'utilisation d'un tel graphe représente fidèlement l'arborescence du FS (on garde le schéma d'un arbre) et simplifie grandement les opérations lorsque des événements surviennent sur ce dernier. Le fait d'ajouter les tags comme noeuds du graphe maintient une unique structure de données cohérente et diminue le nombre d'opérations différentes nécessaires lors de la mise à jour du FS. Le parcours de ce graphe se fait en fonction du chemin de fichier donné, en gardant l'identifiant unique du noeud correspondant au répertoire racine, le parcours se fait de la racine vers le noeud final du chemin de fichier.

La table de hachage utilisée dans cette version peut être vue comme un "cache" d'accès aux noeuds tags. En effet, nous pourrions nous passer de cette table de hachage et lorsqu'un accès à un tag est demandé, rechercher dans tout le graphe le tag en question. Cependant, cette dernière opération devient rapidement conséquente lorsque le graphe comporte de très nombreux noeuds. De plus, cette *hashmap* est accédée bien moins souvent que dans la première version de l'architecture, car elle est mise à jour uniquement lors des opérations sur les tags et non plus sur celles liées seulement aux fichiers et répertoires (opérations potentiellement plus lourdes).

Comme pour la sous-section 3.2.1, la table 3 donne pour chaque événement survenu sur le FS, les opérations à exécuter pour les deux structures de données (le graphe et la table de hachage) et une approximation de la complexité. Les événements sont représentés par leurs numéros, en table 1. Les variables suivantes sont définies :

— c = Opération constante.

— p = Profondeur du graphe.

— t = Nombre de tags.

Nous pouvons constater que les opérations sur la table de hachage sont peu nombreuses et souvent facultatives, ce qui se traduit par un gain sur le nombre d'opérations totales.

Numéro	Opération sur le graphe	Opération sur la *hashmap*
1	Parcourir le graphe à la recherche du fichier, si besoin créer le noeud tag, et relier le noeud fichier au noeud tag -> $O(p * c)$	Si non existant, ajouter le nom du tag comme clé et son identifiant dans le graphe comme valeur -> $O(c)$
2	Parcourir le graphe à la recherche du noeud fichier et supprimer le lien entre noeud tag et fichier. Si le noeud tag n'est relié à aucun autre noeud, le supprimer -> $O(p * c)$	Si le noeud tag n'est relié à aucun autre noeud, supprimer l'entrée -> $O(c)$
3	Obtenir l'identifiant grâce à la hashmap et renommer le noeud correspondant -> $O(c)$	Supprimer l'entrée et en recréer une avec le nouveau nom et le même identifiant -> $O(c)$
4	Parcourir le graphe à la recherche du répertoire parent, ajouter le nouveau noeud. Pour les tags existants, lier le nouveau noeud, sinon créer le nouveau noeud tag correspondant -> $O(p * t * c)$	Pour chaque tag, opération identique au numéro 1
5	Parcourir le graphe à la recherche du répertoire parent, ajouter le nouveau noeud. Pour les tags existants, lier le nouveau noeud, sinon créer le nouveau noeud tag correspondant. Répéter pour la sous-arborescence -> $\approx O(p^2 * t * c)$	Pour chaque tag, opération identique au numéro 1
6	Parcourir le graphe à la recherche du parent et changer le lien du noeud avec son parent / simple renommage du nom dans l'étiquette -> $O(p * c)$	Pas d'opération requise
7	Identique au numéro 6	Pas d'opération requise
8	Parcourir le graphe à la recherche du noeud fichier et supprimer les liens entre noeuds tags et noeud parent -> $O(p * t * c)$	Pour chaque tag du fichier, supprimer le noeud tag s'il n'a plus de liens vers d'autres noeuds -> $O(t * c)$
9	Parcourir le graphe à la recherche du noeud répertoire et supprimer les liens entre noeuds tags et noeud parent. Répéter pour la sous-arborescence -> $\approx O(p^2 * t * c)$	Pour chaque sous répertoire ou sous fichier, opération identique au numéro 8

TABLE 3 – Opérations et complexité, deuxième architecture

3.3 Surveillance du FS

La surveillance du FS et des tags associés est le deuxième pilier du système. L'indexation initiale est obligatoire, mais il est également nécessaire de surveiller en permanence l'arborescence des fichiers pour garder cet index à jour. Pour y parvenir, nous allons utiliser `inotify` (voir section 4.3), en surveillant tout particulièrement les événements suivants :

— IN_ATTRIB : changement sur les tags (ajout, suppression, renommage).

— IN_CREATE : création de fichier/répertoire dans le répertoire surveillé. Ajouter une nouvelle surveillance si répertoire.

— IN_DELETE : suppression d'un fichier/répertoire dans le répertoire surveillé.

— IN_DELETE_SELF : suppression du répertoire surveillé.

— IN_MOVE_SELF : suppression d'un fichier/répertoire dans le répertoire surveillé.

— IN_MOVE_FROM : déplacement/renommage du répertoire (ancien nom).

— IN_MOVE_TO : déplacement/renommage du répertoire (nouveau nom).

Un thread s'occupe d'écouter les événements du FS et les inscrit dans un buffer tandis qu'un autre va mettre à jour le graphe pour répercuter les changements survenus en lisant dans ce même buffer (simple *pattern* producteur-consommateur).

3.4 Requêtes de tags et fichiers

Une fois que la surveillance du FS est en place, le système doit pouvoir répondre à des requêtes de la part de l'utilisateur. À travers un outil en ligne de commande, l'utilisateur a la possibilité de :

— Demander la liste des fichiers et répertoires associés à un ou plusieurs tags. La requête peut être sous la forme d'une expression logique simple (avec les opérateurs logiques "et" et "ou").

— Demander la liste des tags déjà existants.

— Renommer un tag.

Pour échanger les requêtes et les réponses, client et serveur communiquent par sockets, avec un format de protocole très simple (décrit dans la table 4) pour distinguer le type d'une requête.

4 Analyse technologique

4.1 Rust

Cette section présente le langage de programmation Rust et certains de ses mécanismes, à travers quelques exemples, qui sont soit absolument nécessaires pour commencer à programmer avec Rust, soit utilisés dans le code de ce projet. Rust est un langage multi paradigmes, fortement typé, compilé et performant. Il peut être utilisé en autres pour de la programmation orientée système, pour créer des programmes en ligne de commande (CLI) ou pour créer des applications web. Fort d'une communauté active, de nombreux packages et modules sont disponibles sur Crates.io [18] et de nombreuses discussions sont présentes sur le reddit [19] dédié. Pour plus de détails, l'excellent livre [20] réalisé par les mainteneurs de Rust saura donner de plus amples et précises informations au lecteur avide de connaissances sur Rust. Un autre livre [21], plus spécialisé, guide le débutant à Rust dans la conception de listes chaînées, car non triviales en Rust de par le fait de ses contraintes.

4.1.1 Installation

L'installation de Rust sur Linux et macOS est très simple. Les prérequis sont un compilateur C (certaines librairies Rust en nécessitent un) et l'outil de transfert de données `curl`. Il suffit ensuite d'ouvrir un terminal et d'entrer les commandes du listing 2. La deuxième commande est facultative est n'est nécessaire que si l'on veut utiliser Rust sans se déconnecter du shell courant.

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

Listing 2 – Installation de Rust sur Linux ou macOS

Sur Windows, la procédure est un peu plus longue mais tout aussi simple, il faut s'assurer d'avoir les C++ build tools pour Visual Studio 2013 ou supérieur. Rust installe son compilateur, `rustc`, qui permet de compiler un fichier source (`.rs`) en fichier exécutable. Nous n'allons cependant pas en parler davantage, la compilation se fera avec le gestionnaire de paquets et de compilation Cargo (voir sous-section 4.1.2). Pour plus de détails, se référer au chapitre 1.1 du *book* [20]. Le site Are we (I)DE yet ? [22] donne un aperçu des éditeurs de texte compatibles avec la chaîne de développement Rust. En ce qui concerne le projet de bachelor présenté ici, tout le code a été écrit avec Visual Studio Code.

4.1.2 Cargo et Crates.io

Cargo est le système de compilation et exécution et le gestionnaire de paquets intégré à Rust. Depuis le terminal, ses commandes principales permettent de créer un nouveau pro-

jet (`cargo new myproject`), de le compiler (`cargo build`), de l'exécuter (`cargo run`) ou de générer la documentation associée (`cargo doc`). Lorsqu'un nouveau projet est créé avec Cargo, un fichier `Cargo.toml` est généré (à la manière du fichier `package.json` avec Node.js et npm) avec le contenu minimal suivant :

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

Listing 3 – Contenu du fichier `Cargo.toml`

La section "package" contient les informations sur le projet en lui-même. La section "dependencies" liste les paquets dont dépend notre application, appelés "*crates*" par la communauté Rust. Des milliers de *crates* sont disponibles sur Crates.io [18]. D'autres sections peuvent être ajoutées au fichier `Cargo.toml` pour personnaliser les commandes de compilation, créer des workspaces à partir de plusieurs *crates* ou ajouter des commandes spécifiques par exemple. Un sous-chapitre (1.3) et un chapitre entier (14) sont dédiés à Cargo dans le livre de Rust [20] et il dispose également d'une documentation complète [23] (comme le *book* dédié à Rust).

4.1.3 Généralités

Commentaires Tout texte écrit après deux slashs consécutifs "//" est considéré comme un commentaire en Rust. La syntaxe multiligne qui existe en C par exemple n'est pas prise en compte. En mettant trois slashs consécutifs "///", le commentateur indique au compilateur que ce commentaire fait partie de la documentation (qui peut être générée avec Cargo, voir sous-section 4.1.2).

Variables Tout d'abord, pour déclarer une variable en Rust, il faut utiliser le mot-clé `let`. Une variable est par défaut déclarée immutable, *id est* qu'elle ne peut pas être modifiée dans la suite du code. Bien que Rust soit un langage fortement typé, son compilateur sait dans la plupart des cas inférer le bon type de la variable, soit en analysant la valeur attribuée, soit en analysant la première utilisation de la variable (arguments d'une fonction, insertion de données dans le cas des collections). Il existe toutefois la possibilité de déclarer explicitement le type de la variable en l'indiquant avant le "=". Pour déclarer une variable mutable, il faut lui ajouter le mot-clé `mut` avant son nom. Ensuite, les constantes sont déclarées avec le mot-clé `const` et leur type doit obligatoirement être indiqué. La différence principale entre les constantes et les variables immutables est qu'une constante ne peut être le résultat d'une valeur

calculée à l'exécution du programme. Enfin, une variable peut être "masquée" ou "obscurcie" (*shadowed*) : une nouvelle déclaration avec `let` et le même nom écrase la précédente valeur et peut être de type différent. Le listing 4 montre quelques cas de déclarations de variables :

```
// Déclaration d'une variable "x", immutable et de type inféré i32
let x = 3;
// Déclaration d'une variable "y", mutable et de type inféré bool
let mut y = true;
// Déclaration d'une variable "z", mutable et de type déclaré char
let mut z : char = 'A';
// La constante PI de type flottant 64 bits
const PI : f64 = 3.1415;
// Shadowing. La première déclaration crée une variable nommée "answer"
// de type i32 et de valeur 42, alors que la deuxième écrase la
// précédente variable en lui prenant son nom et est de type String.
let answer = 42;
let answer = answer.to_string();
```

Listing 4 – Exemples de déclarations de variables en Rust

Pour plus de détails, se référer au chapitre 3.1 du *book* [20].

Types Il existe deux familles de types en Rust :

— Les scalaires : nombres entiers, nombres à virgule, booleans et caractères.

— Les composés (deux types primitifs) : les tuples et les tableaux.

Les entiers peuvent être signés ou non signés et sur 8, 16, 32, 64 bits ou dépendant de l'architecture du processeur (`i8`, `u8`, `i16`, `u16`, `i32`, `u32`, `i64`, `u64`, `isize`, `usize`). Les nombres à virgule ont deux possibilités, soit sur 32 bits, soit sur 64 bits (`f32` ou `f64`). Le type `bool`, classique, peut prendre deux valeurs, `true` ou `false`. Enfin, dernier type primitif scalaire, `char`, stocke un caractère Unicode entre simples guillemets. Le premier type primitif composé est le tuple. C'est un regroupement de plusieurs valeurs qui peuvent être de différents types. Lors des déclarations, les noms de variables, les types et les valeurs d'un tuple sont contenues entre parenthèses et séparées par des virgules. Enfin, le type tableau, ou *array* : classique type regroupant plusieurs valeurs du même type cette fois. Un *array* a une taille fixe, déterminée à la compilation. La déclaration des valeurs d'un tableau se fait entre crochets "[ ]". Pour accéder à une valeur du tableau, il faut utiliser la syntaxe `array[i]` où `i` est un indice valide du tableau (entre zéro compris et la taille du tableau non compris). Quelques exemples sont donnés dans le listing 5.

```rust
let myint : i32 = 1234;
let mychar : char = 'a';
let myfloat : f64 = 2.0;
// Déclaration d'un tuple
let tuple : (char, u32, f64) = ('c', 42, 2.8);
// Destructuration du tuple en trois variables distinctes
let (letter, age, score) = tuple;
// Déclaration d'un tableau
let myarray = ['a', 'b', 'c', 'd', 'e', 'f'];
let x = myarray[4]; // x vaut 'e'
```

Listing 5 – Quelques types primitifs de Rust

Pour plus de détails, se référer au chapitre 3.2 du *book* [20].

Fonctions Comme en C, tout programme a comme point d'entrée la fonction `main()`. Une déclaration de fonction commence par le mot-clé `fn`, est suivi du nom de la fonction, de la liste des éventuels paramètres et des éventuels types de retour. Lorsqu'une fonction a une valeur de retour, la dernière ligne de la fonction qui n'a pas de point-virgule à sa fin est évaluée comme une expression et est retournée. Le mot-clé `return` existe néanmoins si la fonction doit retourner dans des cas bien précis (dans une condition par exemple). Les variables déclarées dans la fonction ne sont pas accessibles depuis l'extérieur de la fonction. Les arguments de la fonction sont passés par copie par défaut (voir la sous-section 4.1.8 pour plus de détails). Le listing suivant donne l'exemple d'une même fonction, en deux versions plus ou moins courtes. Ces fonctions attendent deux entiers et retournent également un entier.

```rust
fn x_plus_y_plus_one(x : i32, y : i32) -> i32 {
    let x_plus_y = x + y;
    x_plus_y + 1
}

fn x_plus_y_plus_one_short(x : i32, y : i32) -> i32 {
    x + y + 1
}
```

Listing 6 – Exemples de fonctions en Rust

Pour plus de détails, se référer au chapitre 3.3 du *book* [20].

Structures de contrôle Comme tout langage de programmation, Rust possède des structures de contrôle pour gérer les conditions et les répétitions (boucles). Il y a le classique `if condition { ... } else if autre_condition { ... } else { ... }` avec une différence notable par rapport à C : il est possible d'affecter une variable avec un `if`, comme dans le listing 7. Il n'y a pas de `switch ... case` à proprement parler en Rust, nous verrons le `match ... case` à la sous-section 4.1.6. En ce qui concerne les boucles, elles sont au nombre de trois : `loop` (boucle infinie), `while` (boucle avec condition initiale) et `for`, boucle pour traverser les collections par leurs itérateurs principalement (voir sous-section 4.1.7).

```
1  // Exemple d'un 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 // Exemple d'affectation d'une variable avec un if. Ici, n vaudra 42
11 let condition = true;
12 let n = if condition { 42 }
13 else { 66 };
```

Listing 7 – Exemples de conditions en Rust

```rust
// Boucle infinie
loop {
    println!("Forever");
}
// Boucle avec condition
let mut x = 0;
while x < 10 {
    println!("{}", x);
    x = x + 1;
}
// Parcours d'un tableau
let myarray = [1, 2, 3, 4, 5];
for elem in myarray.iter() {
    println!("value : {}", elem);
}
```

Listing 8 – Exemples de boucles en Rust

Pour plus de détails, se référer au chapitre 3.5 du *book* [20].

Organisation des fichiers et modules Un programme écrit en Rust peut être découpé en plusieurs fichiers et modules. Un module peut contenir des déclarations de fonctions, de structures et leurs implémentations (voir section 4.1.4), etc. Le mot-clé pour déclarer un module est `mod`. Le code à l'intérieur du module est par défaut privé, pour le rendre accessible en dehors du module, le préfixe `pub` est disponible. Pour utiliser un module au sein d'un autre ou dans `main.rs`, il faut l'importer avec le mot-clé `use`. Par défaut, tout module est défini dans le fichier `src/lib.rs` d'un projet. Si de nombreux modules sont déclarés, il est possible de les mentionner dans `src/lib.rs` de cette manière : `mod mymodule;` et de créer un fichier ayant le même nom que le module (dans cet exemple, `src/mymodule.rs`) contenant le code en question. Une déclaration de module peut en contenir d'autres également, créant ainsi une hiérarchie de modules. Pour plus de détails, se référer au chapitre 7 du *book* [20].

4.1.4 Structures de données

Comme en C, Rust octroie la possibilité au programmeur de définir ses propres types composés, les `struct`. La déclaration et l'instanciation d'une structure se font comme en C avec quelques raccourcis disponibles. Une structure sans noms de champs est également disponible, appelée "tuple struct". Pour accéder aux champs d'une structure, il suffit d'utiliser la notation pointée (`player.name`).

```rust
// Structure définissant un personnage dans un jeu vidéo
struct Player {
    name: String,
    class: String,
    life: i32,
    active: bool
}
// Création d'une variable Player
let player_one = Player {
    name: String::from("Groumf"),
    class: String::from("Wizard"),
    life: 100,
    active: true
};

// Structure sans noms aux champs
struct Coordinates(f64, f64);
let geneva = Coordinates(46.2016, 6.146);

// Structure vide ()
struct Nil;
```

Listing 9 – Exemples de structures en Rust

La particularité des structures, par rapport à C, est qu'il est possible de définir des méthodes rattachées aux structures, à la manière des méthodes en Java, sans pour autant obtenir une classe *stricto sensu* des langages orientés objets, même si le résultat final est très semblable. Pour définir des méthodes à une structure, il est nécessaire de déclarer un bloc de code commençant par le mot-clé `impl` suivi du nom de la structure et d'accolades. À l'intérieur de ce bloc sont définies des fonctions en relation avec la structure. Dans le listing 10, nous voyons la déclaration de la structure `Player` et de son implémentation, comportant trois méthodes (avec la syntaxe des fonctions) pour créer un nouveau personnage, qu'il puisse en attaquer un autre et qu'il puisse saluer. La seule différence entre une méthode et une fonction est qu'une méthode qui est appelée sur une variable du type de la structure avec la notation pointée attend le paramètre `self` comme premier paramètre, obligatoirement. `self` réfère à la variable elle-même, comme `this` en Java. Nous pouvons voir que `self` et les autres paramètres ont une syntaxe non décrite pour l'instant (`&` et `&mut`), la sous-section 4.1.8 donne de plus amples explications.

```rust
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 – Bloc `impl` d'une structure en Rust

Pour plus de détails, se référer au chapitre 5 du *book* [20].

4.1.5 Traits et généricité

Les traits en Rust sont l'équivalent des interfaces en Java. C'est une manière de définir un comportement abstrait que pourrait suivre un type. Le listing 11 définit un trait "véhicule" (`Vehicle`) qu'implémentent les structures "vélo" (`Bicycle`) et "avion" (`Plane`). Le trait `Vehicle` donne la signature d'une seule fonction, `description()` qui décrit la variable du type en question. Les deux structures implémentant le trait doivent également implémenter toutes les fonctions du trait.

```rust
trait Vehicle { fn description(&self); }

struct Bicycle { wheels : u8, passengers : u8 }
impl Vehicle for Bicycle {
    fn description(&self) {
        println!("I'm a bicycle, I have {} wheels and \
            can carry {} passengers.",
            self.wheels, self.passengers);
    }
}

struct Plane { engines : u8, passengers : u16, fuel : String }
impl Vehicle for Plane {
    fn description(&self) {
        println!("I'm a plane, I have {} engines and \
            can carry {} passengers. I fly with {}.",
            self.engines, self.passengers, self.fuel);
    }
}

fn main() {
    let bicycle = Bicycle { wheels : 2, passengers : 1 };
    bicycle.description();
    let plane = Plane { engines : 1, passengers : 100,
        fuel : String::from("kerosene") };
    plane.description();
}
```

Listing 11 – Implémentations d'un trait en Rust

Les traits peuvent être déclarés génériques, comme les fonctions d'ailleurs. La généricité n'est pas un concept réservé à Rust, de nombreux langages de programmation l'utilisent. C'est une manière d'éviter de répéter un même code pour des types de données différents, mais qui auraient des similitudes. Prenons l'exemple d'une fonction faisant l'addition de deux nombres. Les arguments pourraient être soit des nombres entiers, soit des nombres à virgule. La méthode pour additionner ces deux types est la même. Mais pour un langage fortement typé comme Rust, il est nécessaire de définir précisément les types des arguments des fonctions. C'est là qu'entre en jeu la généricité : la déclaration d'une fonction attend un type générique, usuellement nommé `T`, et le manipule comme un type réel. De nombreux types de la librairie

standard sont génériques, comme les types `Option` et `Result` (voir sous-section 4.1.6). Pour plus de détails, se référer au chapitre 10 du *book* [20].

4.1.6 Énumérations et *pattern matching*

Les énumérations sont une autre manière de concevoir ses propres types. Comme en C, une énumération liste toutes les variantes possibles d'une valeur d'un même type. L'exemple classique d'une énumération sont les jours de la semaine. Sept cas différents, sans évolutions possibles. Les énumérations en Rust prennent tout leur sens en combinaison avec les *pattern matching* : une structure de contrôle ressemblant à un `switch ... case` en C mais avec un côté davantage emprunté à la programmation fonctionnelle (on les retrouve d'ailleurs en Scala). Tous les cas possibles d'une énumération doivent être traité avec un `match` (d'où la clause par défaut _). Le bloc de code à droite de chaque => peut être retourné, comme pour une fonction (voir listing 12).

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

Listing 12 – Définition d'une `enum` et son utilisation avec un *pattern matching* en Rust

Rust possède dans la librairie standard une énumération très puissante : `Option` (recopiée dans le listing 13). Elle remplace le tristement fameux `NULL` en C ou d'autres langages. Elle supprime simplement d'innombrables bugs souvent rencontrés à cause de `NULL`. Si une variable existe, elle se retrouve dans le cas `Some` de l'option, si elle n'existe pas, dans le cas `None`. Cette énumération est de plus générique (voir sous-section 4.1.5), elle accepte tout type de données.

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

Listing 13 – L'énumération `Option` et son utilisation avec un *pattern matching* en Rust

Pour plus de détails, se référer au chapitre 6 du *book* [20].

4.1.7 Collections

Cette sous-section décrit les deux collections les plus utilisées en Rust, à savoir les vecteurs (`Vec`) et les tables de hachage associatives (`HashMap`). Un vecteur est un tableau qui n'a pas de taille fixe, des éléments peuvent lui être ajoutés ou enlevés. C'est l'équivalent des `ArrayList` en Java. Un vecteur est générique (voir sous-section 4.1.5), il peut contenir tout type de données, mais un seul type à la fois. De nombreuses méthodes existent pour manipuler un vecteur, que ce soit pour lui ajouter ou enlever des éléments ou pour le convertir vers d'autre formes. Une macro, `vec!` est disponible pour rapidement créer un vecteur (éléments séparés par des virgules entre "[ ]"). Le listing 14 donne quelques exemples :

```rust
// Déclaration d'un vecteur avec new, puis avec la macro
let v: Vec<char> = Vec::new();
let mut v = vec!['a', 'b', 'c'];
// Ajout d'un élément au vecteur
v.push('d');
// Accès au deuxième élément du vecteur, de deux manière différentes
let second: &char = &v[1];
let second: Option<&char> = v.get(1);
// Parcours (immutable) du vecteur avec la syntaxe for .. in ..
for i in &v {
    println!("{}", i);
}
```

Listing 14 – Exemples de déclarations et utilisations d'un vecteur

Une `HashMap` est une table de hachage associative, tel qu'il en existe en Java. Elle est, comme le vecteur, générique (voir sous-section 4.1.5), elle accepte tout type de données. C'est une structure de données efficace pour accéder rapidement à une information. Le listing 15 donne quelques exemples :

```
// Déclaration d'une hashmap avec new
let h: HashMap<char, u32> = HashMap::new();
// Ajout d'une paire clé-valeur à la hashmap
h.insert('a', 97);
// Accès à la valeur associée à la clé 'a', retourne une Option
let a: Option<&u32> = h.get('a');
// Parcours (immutable) de la hashmap avec la syntaxe for .. in ..
for (key, value) in &h {
    println!("{}: {}", key, value);
}
```

Listing 15 – Exemples de déclaration et utilisation d'une `HashMap`

Pour plus de détails, se référer au chapitre 8 du *book* [20].

4.1.8 *Ownership*, *Borrowing* et références

La caractéristique unique de Rust est sans aucun doute l'*ownership*, ou "possession". L'*ownership* est défini par ces trois règles :

— Chaque variable est dite le "possesseur" (*owner*) d'une valeur.

— Il ne peut y avoir qu'un seul *owner* pour une valeur.

— Lorsque l'*owner* est détruit ou change de portée, la valeur est détruite.

Avant de continuer sur cette notion, un bref rappel sur l'utilisation de la mémoire est nécessaire. Le système d'exploitation (OS) met à disposition d'un programme deux zones mémoire différentes pour stocker ses variables : la pile (*stack*) et le tas (*heap*). Le mécanisme d'accès à la pile est simple et rapide et stocke des variables dont la taille est fixe et connue à la compilation. Les variables de type primitifs (voir paragraphe 4.1.3) sont stockées dans la pile. Lorsqu'une variable dont la taille n'est pas connue à l'avance (type évolué, par exemple les collections) est déclarée, elle placée dans le tas. Un programme voulant stocker une variable sur le tas est obligé de demander à l'OS de chercher un espace disponible assez grand pour stocker la variable et qu'il lui donne un pointeur vers cette zone, pour la retrouver par la suite.

Lorsqu'une valeur est assignée à une variable, elle est détenue par cette variable jusqu'à sa destruction. Le listing 16 illustre un exemple :

```rust
{
    // my_vec est le "propriétaire" de ce vecteur
    let my_vec = vec![3, 2, 1];
    ... // my_vec est utilisé
} // la portée de my_vec se termine ici, my_vec est alors supprimé
```

Listing 16 – Portée d'une variable en Rust

La variable `my_vec` se trouve sur la pile, mais les données vers lesquelles elle pointe se trouvent dans le tas. Lorsque `my_vec` sera hors de portée, les données dans la pile et dans le tas seront libérées. Si `my_vec` est assigné à une autre variable, l'*ownership* des données est transféré à cette nouvelle variable, `my_vec` ne sera plus accessible et utilisable après l'affectation, comme illustré au listing 17. Ce cas de figure ne se présente pas avec les types primitifs dont la taille en mémoire est connue à la compilation et où une copie des données est effectuée. Pour qu'un type évolué puisse être copié de cette façon, il doit implémenter le trait `Copy`.

```rust
{
    // Ici pas de problèmes, a et b sont de type primitif i32, de
    // taille fixe et connue, la valeur de a est copiée dans b.
    let a = 10;
    let b = a;

    let mut my_vec = vec![3, 2, 1];
    let other_vec = my_vec;
    my_vec.push(42); // Erreur, la valeur a été déplacée (move)
}
```

Listing 17 – Transfert de l'*ownership* en Rust

À la ligne 8 du listing 17, la variable `my_vec` est invalidée, mais pas les données vers lesquelles elle pointe. Il n'y a que le pointeur vers ces données qui transféré vers `other_vec`. C'est ici que la deuxième règle de l'*ownership* s'applique, il ne peut y avoir plus d'un *owner* d'une même valeur au même moment. Pour réaliser une vraie copie des données d'un vecteur à un autre, ou de manière générale pour un type évolué, il faut appeler la méthode `clone()`, qui copie entièrement les données en mémoire. Cette situation de transfert d'*ownership* survient également lors des appels à des fonctions. Si un paramètre est donné à une fonction, la fonction en prend possession, comme illustré au listing 18.

```rust
fn main() {
    let mut my_vec = vec![3, 2, 1];
    print_vec(my_vec); // La fonction prend possession du vecteur
    my_vec.push(42); // Erreur, la valeur a été déplacée (move)
}

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

Listing 18 – Transfert de l'*ownership* vers une fonction en Rust

Pour palier à ce problème de transfert de possession lors d'appels aux fonctions, deux solutions existent :

1. La fonction doit retourner la valeur possédée. Ce n'est pas pratique si le but de la fonction est de retourner le résultat d'une opération. Elle peut retourner un tuple formé de la ou les valeurs accaparées et du résultat de son opération. Cette manière de faire est lourde et non conseillée.
2. La fonction peut "emprunter" la variable de différentes manières (voir ci-dessous).

Heureusement, les fonctions peuvent se "prêter" les variables par *borrowing* ("emprunt"). Deux types de références aux variables sont disponibles :

1. Les références immutables (syntaxe `&ma_variable`).
2. Les références mutables (syntaxe `&mut ma_variable`).

Le listing 19 montre un exemple d'emprunt immutable et mutable.

```rust
fn main() {
    let mut my_vec = vec![3, 2, 1];
    // La fonction emprunte de manière immutable le vecteur
    ref_immutable(&my_vec);
    // La fonction emprunte de manière mutable le vecteur
    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 – Emprunts de variables entre fonctions en Rust

Ces références sont proches conceptuellement des pointeurs en C (Rust accepte d'ailleurs le déréférencement des variables avec le symbole $*$) mais obéissent à deux règles fondamentales :

1. À tout moment, il peut y avoir soit une seule référence mutable, soit plusieurs références immutables, mais pas les deux en même temps.
2. Les références doivent toujours être valides.

Grâce aux références, Rust évite les problèmes de concurrence sur les valeurs pointées ainsi que les pointeurs invalides. Pour plus de détails, se référer au chapitre 4 du *book* [20]. Il existe également d'autres types de pointeurs, dits "intelligents". Le livre sur Rust dédie un chapitre entier à eux (chapitre 15 du *book* [20]).

4.1.9 Gestion des erreurs

Nonobstant son compilateur très restrictif, détectant à la compilation de nombreuses erreurs, Rust a une gestion avancée des erreurs survenant à l'exécution. Il distingue deux types : les erreurs récupérables et les erreurs irrécupérables (non pas comme dans les langages comme Java ou le concept d'exceptions mélange ces deux types d'erreurs). Pour ces dernières, Rust offre une macro, `panic!`, stopant abruptement le programme et affichant un message d'erreur sur la sortie standard en indiquant à quelle ligne le programme a planté. Pour les erreurs récupérables, une manière plus élégante existe : à la manière de l'énumération `Option` vue à la section 4.1.6, l'énumération `Result` a été conçue pour gérer les erreurs au *runtime* (voir listing 20).

```rust
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 – L'énumération `Result` et son utilisation avec un *pattern matching* en Rust

De nombreuses fonctions de la librairie standard et de la communauté retournent des `Result`, notamment lors de l'ouverture d'un fichier. Comme ce genre d'opérations sont courantes, Rust met à disposition deux fonctions équivalentes à réaliser un `match` sur un `Result`, `unwrap()` et `expect()`. La seule différence entre les deux est qu'avec la deuxième fonction, un message personnalisé est attendu en argument. Le listing 21 montre les différentes façons de traiter un `Result` lors de l'ouverture d'un fichier :

```rust
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 – Ouverture d'un fichier et son traitement en Rust

Pour plus de détails, se référer au chapitre 9 du *book* [20].

4.1.10 Tests

Lorsqu'un nouveau projet *librairie* est créé avec `cargo new mylib --lib`, un module de tests unitaires est *de facto* ajouté au fichier `lib.rs`, comme dans le listing 22 :

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

Listing 22 – Module de test ajouté automatiquement

Chaque fonction précédée par l'attribut `#[test]` est considérée par le compilateur comme un test. Pour exécuter les tests, Cargo met à disposition une commande, `cargo test`. Lorsque cette commande est exécutée, tous les tests sont effectués en parallèle, sans garantir un ordre d'exécution prédéfini. Un résumé des tests effectués, réussis et échoués est affichés sur la sortie standard à la fin de l'exécution de la commande. Pour réaliser nos tests, Rust fournit quelques macros :

— `assert!` : attend un argument de type `bool`.

— `assert_eq!` : attend deux arguments de même type, pour vérifier qu'ils sont égaux.

— `assert_ne!` : inverse du précédent, vérifie l'inégalité des deux arguments donnés.

Il existe également un autre attribut, `#[should_panic]`, pour réaliser des fonctions qui testent si un morceau de code doit "paniquer", donc échouer. Finalement, la *killer feature* des tests en Rust, c'est que la commande `cargo test` vérifie également le code donné en exemple dans la documentation de notre code, pour garder une documentation à jour avec notre code. Pour plus de détails, se référer au chapitre 11 du *book* [20].

4.1.11 Concurrence et threads

Rust fournit une implémentation des threads dans sa librairie standard. Un thread en Rust correspond à un thread système. Le listing 23 donne un exemple de création d'un thread. La méthode `spawn()` retourne un *handler* pour terminer le thread proprement avec la méthode `join()`.

```rust
use std::thread;

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

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

Listing 23 – Création d'un thread en Rust

Un thread peut prendre possession d'une variable si le mot-clé `move` est ajouté à l'appel de `spawn()`. La variable n'est plus disponible dans la fonction ayant appelé le thread. Le listing 24 montre un exemple de cette situation.

```rust
use std::thread;

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

Listing 24 – Création d'un thread en Rust et passage d'une variable

Pour communiquer entre threads, un mécanisme de messages est disponible. Les threads communiquent à travers un canal selon la topologie *Multiple Producer, Single Consumer* ("multiple producteur, unique consommateur", voir figure 8) : il est possible que plusieurs threads envoient (produisent) des messages dans le canal mais un seul thread peut les recevoir (consommer). Le listing 25 donne un exemple.

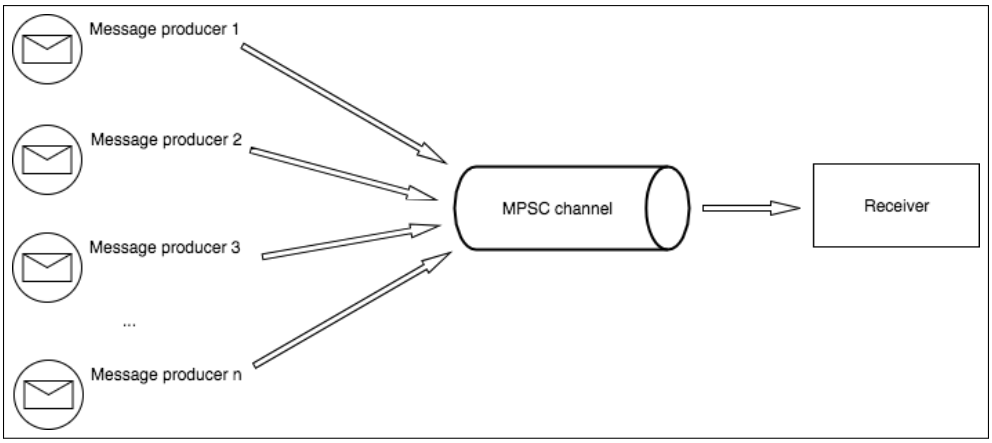


FIGURE 8 – Canal de communication entre threads - [24]

```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* avec deux producteurs et un consommateur en Rust

Des primitives plus bas niveau (Mutex) ainsi que des traits sont disponibles pour manipuler plus finement les threads en Rust. Pour plus de détails, se référer au chapitre 16 du *book* [20]. Un article illustrant l'utilisation des threads en Rust par la réalisation d'un programme simple de calcul de hash donne une bonne base pour construire son propre programme [24].

4.1.12 *Unsafe* Rust

Le compilateur Rust est non seulement très verbeux lors d'erreurs mais également très restrictif. Cette contrainte est la garantie pour le programmeur d'obtenir un code sûr et fiable sur la gestion de la mémoire. Cependant, il arrive que du code doive transgresser les règles et bonnes pratiques de Rust sur la mémoire. C'est notamment le cas pour du code bas niveau (manipulation du kernel par exemple). Pour ces cas de figure, Rust peut fonctionner en mode *unsafe* ("non sûr"). Tout bloc de code qui nécessite d'être non sûr doit être précédé du mot-clé `unsafe`. Il est alors de la responsabilité du programmeur de vérifier que le code n'est pas buggé ou dangereux. Pour plus de détails, se référer au chapitre 19.1 du *book* [20].

4.2 Extended attributes

4.2.1 Fonctionnement des XATTR

Les *extended attributes* (XATTR), ou "attributs étendus" en français, sont un moyen d'attacher des méta-données aux fichiers et dossiers sous forme de paires `espace.nom:valeur`. L'espace de nom, ou *namespace* en anglais, définit les différentes classes d'attributs. Dans le cadre de ce projet, l'accent est mis sur le système de fichiers (FS) ext4 [25] sous Linux, il existe actuellement quatre espaces de noms ou classes : `user`, `trusted`, `security` et `system`. L'espace qui nous intéresse est `user`. C'est là que l'utilisateur ou l'application, pour autant qu'il ait les droits usuels UNIX sur les fichiers, peut manipuler les XATTR. Les trois autres espaces de noms sont utilisés entre autres pour les listes d'accès ACL (`system`), les modules de sécurité du kernel (`security`) ou par `root` (`trusted`) [26] [27]. Le nom est une chaine de caractères et la valeur peut être une chaine de caractères ou des données binaires. Les XATTR sont stockés dans les fichiers. De nombreux FS gèrent leur usage : ext2-3-4, XFS, Btrfs, UFS1-2, NTFS, HFS+, ZFS. Ces FS sont utilisés par les quatre OS les plus répandus : Windows, macOS, Linux et FreeBSD. Windows utilise les XATTR notamment dans sa gestion des permissions Unix dans le shell Linux intégré à Windows 10 [28]. macOS, comme vu à la section 2.2.2, les utilise entre autres dans son système de gestion des tags. La commande `xattr` permet de les manipuler. Sous Linux, il en existe trois : `attr`, `getfattr` et `setfattr`. Sous Linux avec ext2-3-4, chaque attribut dispose d'un bloc de données (1024, 2048 ou 4096 bytes) [27]. Apple et freedesktop.org préconisent la notation DNS inversée pour nommer les attributs [29], [30] car n'importe quel processus peut modifier les attributs dans l'espace utilisateur. En préfixant du nom du programme au nom de l'attribut, par exemple `user.myprogram.myattribute`, on diminue le risque qu'une autre application utilise le même nom d'attribut. Malheureusement, la plupart des outils CLI Linux pour manipuler les fichiers comme `cp`, `tar`, etc. ne prennent pas en compte les attributs avec leur syntaxe par défaut, il faut spécifier des arguments supplémentaires [31].

4.2.2 Comportement lors des opérations d'accès courantes

Pour vérifier la portabilité des XATTR, quelques tests ont été réalisés entre un SSD faisant office de disque système à Linux Mint 18.2 Sonya avec deux clés USB (de 8 et 64 Go) et un emplacement réseau monté en NFS. Le listing 26 montre la sortie de la commande `df`, qui renvoie l'utilisation des différents emplacements de stockage, dans l'ordre : le disque système, en ext4, la clé de 8 Go formatée une fois en FAT32, puis une autre fois en NTFS, la clé de 64 Go formatée en ext4 et finalement une machine virtuelle sous Debian 9 montée en NFS.

```
Sys. de fichiers        Type   Taille Utilisé Dispo Uti% Monté sur
/dev/sda2               ext4     451G    334G   96G  78% /
/dev/sdg1               vfat     7.7G    4.0K  7.7G   1% /media/pc/cle1
/dev/sdg1               fuseblk  7.7G     41M  7.7G   1% /media/pc/cle1
/dev/sdg1               ext4      59G     33G   23G  59% /media/pc/cle2
192.168.1.21:/home/user nfs4     916G    198G  673G  23% /mnt/debian
```

Listing 26 – Output de `df -Th` : le disque système, les clés USB et le NFS

La démarche est la suivante : un XATTR dans l'espace `user` avec comme nom `author` et comme valeur `steven` est ajouté au fichier `file.txt` avec `attr`. Ce fichier est copié avec `cp` en prenant garde à préserver l'attribut (option `--preserve=xattr`). Une fois copié, on tente de lire le même attribut, toujours avec `attr`. Les résultats sont visibles dans les listings 27, 28, 29 et 30 :

```
~ $ attr -s author -V steven file.txt
L'attribut "author" positionné à une valeur de 6 octets pour file.txt :
steven
~ $ cp --preserve=xattr file.txt /media/pc/cle1
cp: setting attributes for '/media/pc/cle1/file.txt': Opération non
supportée
```

Listing 27 – Copie sur clé USB 8 Go, FAT32

```
~ $ attr -s author -V steven file.txt
L'attribut "author" positionné à une valeur de 6 octets pour file.txtă:
steven
~ $ cp --preserve=xattr file.txt /media/pc/cle1
~ $ cd /media/pc/cle1
/media/pc/cle1 $ attr -g author file.txt
L'attribut "author" avait une valeur de 6 octets pour file.txtă:
steven
```

Listing 28 – Copie sur clé USB 8 Go, NTFS

```
~ $ attr -s author -V steven file.txt
L'attribut "author" positionné à une valeur de 6 octets pour file.txtă:
steven
~ $ cp --preserve=xattr file.txt /media/pc/cle2
~ $ cd /media/pc/cle2
/media/pc/cle2 $ attr -g author file.txt
L'attribut "author" avait une valeur de 6 octets pour file.txtă:
steven
```

Listing 29 – Copie sur clé USB 64 Go, ext4

```
~ $ cp --preserve=xattr file.txt /mnt/debian
cp: setting attributes for '/mnt/debian/file.txt': Opération non
supportée
```

Listing 30 – Copie sur l'emplacement réseau distant, NFS

On constate que l'opération est infructueuse sur la clé en FAT32 et sur l'emplacement réseau monté en NFS alors qu'elle réussit sur les clés USB en NTFS et ext4.
Deux autres petites expériences ont été menées avec la commande `mv` et la copie/déplacement de fichiers avec l'explorateur de fichiers Nemo de Linux Mint. Ces deux opérations conservent par défaut les XATTR.

4.3 `inotify`

Sous Linux, un outil (inclu au noyau) dédié à la surveillance du FS existe, `inotify` [32]. Comme son nom l'indique, `inotify` donne la possibilité à une application d'être notifiée sur des événements au niveau du système de fichiers. Une interface de programmation (API) en C existe et offre les appels systèmes (SYSCALL) suivants :

1. `int inotify_init(void)` : initialise une instance `inotify` et retourne un descripteur de fichier.
2. `int inotify_add_watch(int fd, const char *pathname, uint32_t mask)` : cette fonction attend le descripteur de fichier renvoyé par `inotify_init`, un chemin de fichier ou répertoire à surveiller et un masque binaire constitué des événements à surveiller (voir plus loin). Il retourne un nouveau descripteur de fichier qui pourra être lu avec le SYSCALL `read()`.
3. `int inotify_rm_watch(int fd, int wd)` : appel inverse du précédent, supprime la surveillance du descripteur de fichier `wd` de l'instance `inotify` retournée par `fd`.

`inotify` s'utilise comme suit : il faut initialiser l'instance, ajouter les fichiers et répertoires pour la surveillance avec le masque des événements voulus et, généralement, dans une boucle, appeler le SYSCALL `read()` avec comme argument le descripteur de fichier renvoyé par `inotify_init()`. Chaque appel abouti à `read()` retourne la structure disponible au listing 31 :

```
1 struct inotify_event {
2     int      wd;       /* Descripteur de surveillance */
3     uint32_t mask;     /* Masque d'événements */
4     uint32_t cookie;   /* Cookie unique d'association des
5                           événements (pour rename(2)) */
6     uint32_t len;      /* Taille du champ name */
7     char     name[];   /* Nom optionnel terminé par un nul */
8 };
```

Listing 31 – Structure `inotify_event` - [32]

Le champ `mask` peut prendre les valeurs suivantes (multiples valeurs autorisées, séparées par des "ou" logiques -> "|") :

- IN_ACCESS : accès au fichier.
- IN_ATTRIB : changement sur les attributs du fichier.
- IN_CLOSE_WRITE : fichier ouvert en écriture fermé.
- IN_CLOSE_NOWRITE : fichier ouvert en écriture fermé.
- IN_CREATE : création de fichier/répertoire.
- IN_DELETE : suppression d'un fichier/répertoire.
- IN_DELETE_SELF : suppression du répertoire surveillé lui-même.
- IN_MODIFY : modification d'un fichier/répertoire.
- IN_MOVE_SELF : suppression d'un fichier/répertoire.
- IN_MOVE_FROM : déplacement/renommage du répertoire (ancien nom).
- IN_MOVE_TO : déplacement/renommage du répertoire (nouveau nom).
- IN_OPEN : ouverture d'un fichier.
- IN_ALL_EVENTS : macro combinant tous les événements précédents.

Le champ `cookie` de la structure `inotify_event` prend tout son sens lors des événements IN_MOVE : un numéro unique est généré pour faire le lien entre ces deux sous-événements, qui ne sont en réalité qu'un seul. `inotify` offre donc une très bonne base pour la surveillance du FS. Il possède cependant quelques limitations :

— Pas de surveillance récursive d'un répertoire : si une arborescence complète doit être surveillée, il faut pour chaque sous-répertoire ajouter une surveillance dédiée.

— Les chemins de fichiers peuvent changer entre l'émission d'un événement et son traitement.

— `inotify` ne permet que la surveillance de répertoires en espace utilisateur par défaut.

— Il n'y pas de moyen de discriminer quel processus ou utilisateur a généré un événement.

Il existe plusieurs outils système qui utilisent `inotify` [33] :

— `incron` : équivalent de `cron`, mais l'exécution des tâches se fait non pas selon un horaire donné, mais selon un événement donné sur un fichier.

— `lsyncd` : outil de synchronisation, basé sur `rsync`. La synchronisation est effectuée à chaque changement dans le répertoire surveillé vers une liste d'emplacements distants configurés à l'avance.

— `iwatch` : déclenchement d'une commande selon un événement `inotify`.

— `inotify-tools` : deux commandes permettant d'utiliser `inotify` directement dans le terminal :

  — `inotifywait` : exécute une attente sur un événement, avant de continuer le fil d'exécution.

  — `inotifywatch` : retourne une liste d'événements des répertoires surveillés.

Pour plus d'informations, la page de `man` sur `inotify` existe [32] et un très bon article en deux parties sur les ajouts de `inotify` par rapport à `dnotify` (son prédécesseur) [34] et sur ses limitations par Michael Kerrisk [35].
Pour conclure, il existe également une API plus récente pour recevoir les notifications du FS, `fanotify` [36]. Elle gomme quelques défauts d'`inotify` (notamment l'accès aux périphériques montés, tels que les clé USB), mais comporte un défaut de taille pour ce projet : il n'y pas le support les événements de création, suppression et déplacement de fichiers et répertoires. `fanotify` ne peut donc pas être utilisé pour ce projet.

4.4 *Sockets*

Les *sockets* (littéralement "prise" en français) sont un moyen de communication entre différents processus, qu'ils soient sur la même machine ou en réseau. Il existe plusieurs types de *sockets*, les deux plus connues sont les sockets "locales" (type `AF_UNIX` ou `AF_LOCAL`), uniquement possibles sur la même machine car reliées par un fichier spécial sur le FS de la machine et les sockets IP (type `AF_INET` ou `AF_INET6`) qui sont reliées par des paires d'adresses IP et ports. Lors d'une communication socket entre deux processus, un des processus endosse le rôle de serveur l'autre de client. Le serveur doit, dans l'ordre, exécuter les SYSCALL suivants :

1. Créer la socket avec `socket()`.

2. Associer la socket à une adresse d'écoute avec `bind()`.
3. Écouter l'arrivée d'une connexion avec `listen()`.
4. Accepter une connexion entrante avec `accept()`.
5. Recevoir les messages avec `read()`.
6. Émettre les messages avec `write()`.
7. Fermer la connexion avec `close()`.

Le client, de son côté, doit exécuter les SYSCALL suivants :

1. Créer la socket avec `socket()`.
2. Se connecter à un serveur en écoute avec `connect()`.
3. Recevoir les messages avec `read()`.
4. Émettre les messages avec `write()`.
5. Fermer la connexion avec `close()`.

La figure 9 résume la procédure d'initialisation et d'utilisation des sockets.

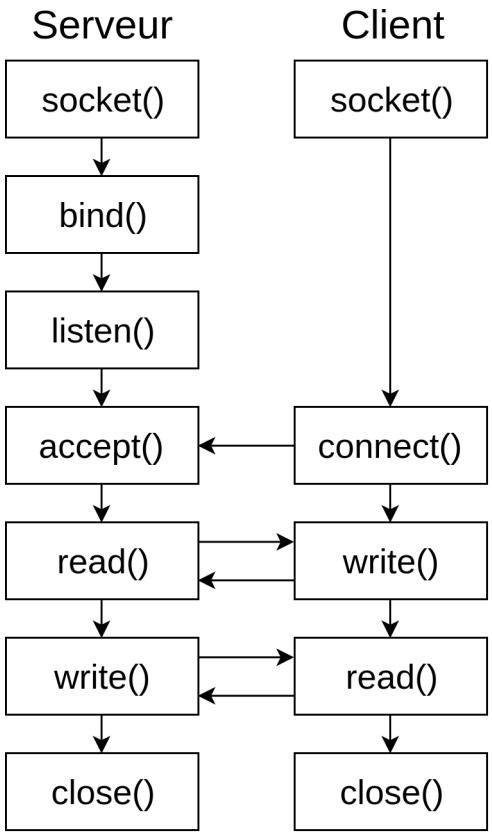


FIGURE 9 – Procédure d'initialisation et d'utilisation des sockets

Lors de l'établissement de la connexion, un canal est établi entre les deux processus, tout ce que le serveur écrit est reçu par le client et inversement. Pour plus d'informations, la page de man des sockets est disponible [37].

5 Réalisation

5.1 Tag Manager

5.1.1 Description du programme et du code

La première réalisation de ce projet est un outil en ligne de commande (CLI), écrit en Rust, permettant de facilement lister, ajouter et supprimer des tags à des fichiers et répertoires (avec une option récursive pour ces derniers) et d'exécuter des requêtes vers le serveur (Tag Engine, sous-section 5.2) pour lister les tags existants, renommer un tag et demander la liste des fichiers correspondant à des tags donnés. Cet outil dépend de deux *crates* disponibles sur crates.io : clap [38] et xattr [39]. Les tags sont stockés dans un *extended attribute* (XATTR) nommé `user.tags` et sont séparés entre eux par des virgules.

clap (Command Line Argument Parser for Rust) est une librairie pour *parse* les arguments d'un programme CLI. Elle analyse et valide les arguments fournis par l'utilisateur. Elle dispose de plusieurs syntaxes pour définir les arguments des commandes et options attendues par notre programme. Sur le dépôt github de clap, dans le dossier *examples*, plusieurs exemples d'utilisation sont fournis. Pour illustrer son utilisation, le listing 32 reprend l'exemple `01a_quick_example.rs` avec la plupart des commentaires tronqués et des adaptations de mise en page. Des lignes 2 à 13, les arguments attendus et les informations sur l'application sont définis avec entre autres la version du programme, le nom de l'auteur, etc. Dans cet exemple, les arguments sont définis à partir d'une chaîne de caractères respectant un format bien spécifique. clap génère automatiquement une aide au programme à partir des arguments définis s'il est lancé sans aucun des arguments obligatoires. À partir de la ligne 15, les arguments reçus sont utilisés. La méthode `value_of()` retourne la valeur d'un argument présent à l'exécution. Il est donc aisé d'utiliser les arguments donnés en tant que variables du programme. clap donne également la possibilité de regrouper les arguments. Un seul argument d'un groupe peut être présent à l'exécution, ce qui évite de nombreuses conditions de détection et d'exclusion des arguments.

xattr est une interface de programmation (API) en Rust pour récupérer, lister, ajouter/modifier et supprimer des XATTR accrochés à des fichiers avec Rust. C'est essentiellement un *wrapper* des appels système (SYSCALL) en C fournis par Linux et d'autres systèmes d'exploitation (OS) pour manipuler les XATTR. À noter que les fonctions offertes ne suivent pas les liens symboliques (il est fait de même pour Tag Manager lui-même). Quatre fonctions sont disponibles : `get()`, `list()`, `set()` et `remove()`. Toutes attendent le nom du fichier et selon les cas le nom du XATTR ainsi que sa valeur.

```rust
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 – Exemple d'utilisation de clap (commentaires tronqués) - [40]

Tag Manager est constitué de deux fichiers. Le premier, `main.rs` contient les définitions et détections des arguments fournis par l'utilisateur avec clap (voir listing 33), les appels aux fonctions manipulant les tags des fichiers et la partie socket de connexion, requêtes et attente de réponse du serveur (Tag Engine, sous-section 5.2). Le deuxième fichier, `lib.rs`, contient l'API publique pour récupérer, attribuer, renommer et supprimer les tags pour un fichier donné et un module de test de ces fonctions. À noter que dans les *output* du programme, il n'y a pas de distinction entre fichiers et répertoires. À titre d'exemple, le listing 34 montre le code de la fonction `del_tags()` qui supprime les tags donnés d'un fichier. Elle préserve les tags existants qui ne doivent pas être supprimés et supprime totalement le XATTR en cas de tableau de tags vide. L'usage des `enum Option` et `Result` en association avec des *pattern matching* sont utilisées dès que possible.

```
37      let matches = App::new("tag_manager")
38          .help(help)
39          .group(ArgGroup::with_name("ops").args(&["set", "del"]))
40          .group(ArgGroup::with_name("queries")
41              .args(&["list", "query", "rename"]))
42          .arg(Arg::with_name("set").short("s").long("set")
43              .takes_value(true).multiple(true))
44          .arg(Arg::with_name("del").short("d").long("del")
45              .takes_value(true).multiple(true))
46          .arg(Arg::with_name("files").short("-f").long("--files")
47              .takes_value(true).multiple(true).required(false))
48          .arg(Arg::with_name("recursive").short("-r")
49              .long("--recursive"))
50          .arg(Arg::with_name("query").short("-q").long("--query")
51              .takes_value(true).multiple(true))
52          .arg(Arg::with_name("list").short("-l").long("--list")
53              .takes_value(false))
54          .arg(Arg::with_name("rename").short("-R").long("--rename")
55              .number_of_values(2))
56          .get_matches();
```

Listing 33 – Déclaration des arguments dans `main.rs`

```rust
pub fn del_tags(file: &str, tags_to_del: &HashSet<String>, recursive: bool) {
    recursion(file, recursive, Delete, tags_to_del);
    match check_existent_tags(file) {
        Ok(res) => match res {
            Some(mut tags) => {
                // Delete only the given tags
                for tag in tags_to_del {
                    tags.retain(|ref e| e != &tag);
                }
                // To avoid to let an empty array of tags
                if tags.is_empty() {
                    match xattr::remove(file, ATTR_NAME) { _ => () }
                }
                else {
                    xattr::set(file, ATTR_NAME,
                        &hash_set_to_vec_u8(&tags))
                        .expect("Error when (re)setting tag(s)");
                }
            }, _ => ()
        },
        Err(err) => {
            eprintln!("Error for file \"{}\" : {}", file, err);
            return;
        }
    }
    println!("Tag(s) {:?} for file {:?} have been deleted",
        tags_to_del, file);
}
```

Listing 34 – Code de la fonction `del_tags()` dans `lib.rs`

La communication socket est réalisée grâce la librairie standard `UnixStream`, équivalent aux sockets `AF_UNIX` en C (voir sous-section 4.4). Le fichier adresse des sockets est par défaut écrit dans `/tmp/tag_engine`. Le format des requêtes respecte le petit protocole décrit dans la table 4. Le début de la requête est formé du code à trois caractères. La requête listant les fichiers selon une expression logique des tags accepte les opérateurs `AND` et `OR`, avec la précédence logique du premier sur le second (voir sous-section 5.2 pour plus de détails). Les opérateurs et opérandes (les tags) doivent être séparés par un espace.

Requête	Code	Exemple
Fichiers et répertoires correspondants à une expression logique de tags	`0x0`	`0x0 tag1 OR tag2 AND tag3`
Liste des tags existants	`0x1`	`0x1`
Renommage d'un tag	`0x2`	`0x2 old_name new_name`

TABLE 4 – Format du protocole des requêtes au serveur Tag Engine

La figure 10 résume le fonctionnement du programme :

1. Analyse des arguments.
2. Exécution des commandes (soit sur les fichiers, soit requête vers le serveur).
3. Impression des résultats.

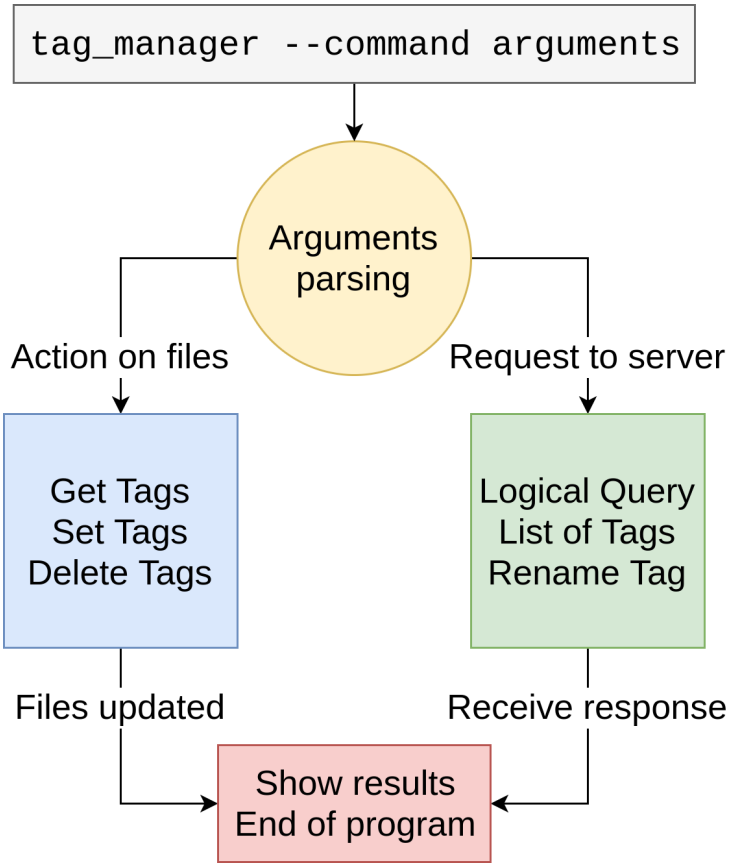


FIGURE 10 – Schéma de fonctionnement de Tag Manager

5.1.2 Utilisation du programme et exemples

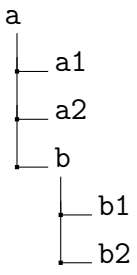
L'utilisation des différents arguments du programme est résumée dans la table 5. Les arguments sont divisés en deux groupes, le premier pour manipuler directement les fichiers et leurs tags, le deuxième pour exécuter des requêtes vers le serveur Tag Engine. Pour le premier

groupe, l'argument `-f` ou `--files` est obligatoire. Pour les opérations du deuxième groupe, il faut évidemment que le serveur Tag Engine soit lancé.

Opération	Arguments	Exemple
Afficher l'aide	`-h`	`tag_manager -h`
Afficher les tags d'un ou plusieurs fichiers	`-f` ou `--files`	`tag_manager -f file1 file2`
Afficher les tags d'un dossier récursivement	`-r` ou `--recursive`	`tag_manager -f myfolder -r`
Attribuer des tags à un ou plusieurs fichiers	`-s` ou `--set`	`tag_manager -f file1 file2 -s bob fred`
Supprimer des tags à un ou plusieurs fichiers	`-d` ou `--del`	`tag_manager -f file1 file2 -d bob fred`
Lister les fichiers correspondants à une requête de tags	`-q` ou `--query`	`tag_manager -q bob AND fred`
Lister les tags existants	`-l` ou `--list`	`tag_manager -l`
Renommer un tag	`-R` ou `--rename`	`tag_manager -R old_name new_name`

TABLE 5 – Utilisation et arguments attendus par Tag Manager

Le listing 35 illustre les usages et retours de Tag Manager. Tout d'abord, Tag Manager est utilisé pour lister récursivement les tags des fichiers et sous-répertoires donnés, attribuer les tags `in_a` et `myfiles` à différents fichiers et répertoires, lister les tags existants et faire une requête sur les deux tags. L'arborescence utilisée pour l'exemple est constituée des quatre fichiers et deux répertoires suivants :



```
$ ./tag_manager -f a -r
File "a" has no tags
File "a/a1" has no tags
File "a/b" has no tags
File "a/b/b1" has no tags
File "a/a2" has no tags
$ ./tag_manager -f a/* -s in_a
Tag(s) {"in_a"} for file "a/a1" have been setted
Tag(s) {"in_a"} for file "a/a2" have been setted
Tag(s) {"in_a"} for file "a/b" have been setted
$ ./tag_manager -f a/b a/b/b1 a/b/b2 -s myfiles
Tag(s) {"myfiles"} for file "a/b" have been setted
Tag(s) {"myfiles"} for file "a/b/b1" have been setted
Tag(s) {"myfiles"} for file "a/b/b2" have been setted
$ ./tag_manager -f a -r
File "a" has no tags
Tag(s) ["in_a"] for file "a/a1"
Tag(s) ["in_a", "myfiles"] for file "a/b"
Tag(s) ["myfiles"] for file "a/b/b1"
Tag(s) ["myfiles"] for file "a/b/b2"
Tag(s) ["in_a"] for file "a/a2"
$ ./tag_manager -l
in_a
myfiles
$ ./tag_manager -q myfiles AND in_a
/home/stevenliatti/a/b
```

Listing 35 – Exemples d'utilisation de Tag Manager

5.2 Tag Engine

5.2.1 Description du programme et du code

La deuxième réalisation pratique de ce projet est un programme indexant et surveillant une arborescence de répertoires, fichiers et tags associés. Ce programme fait également office de serveur pour les requêtes émises depuis Tag Manager. Il est divisé en plusieurs fichiers et modules :

— `graph.rs` : module relatif à la construction et maintenance du graphe des tags, fichiers et répertoires (voir paragraphe **petgraph**).

— `lib.rs` : regroupe les autres modules et contient la fonction `dispatcher()`, appelée lors d'un événement sur l'arborescence surveillée (voir paragraphe **notify**).

— `main.rs` : point d'entrée du programme, initialise les variables partagées, le thread socket serveur et écoute indéfiniment sur les événements survenus sur l'arborescence surveillée (voir paragraphe **Mutex et références multiples**).

— `parse.rs` : module de conversion d'une expression infixe vers postfixe (voir paragraphe **Analyse d'une expression logique**).

— `server.rs` : module construisant le serveur socket et répondant aux différentes requêtes émises depuis Tag Manager (voir paragraphe **Serveur sockets**).

Ce programme dépend de quatre *crates* externes. Le premier est tag_manager (la librairie des fonctions manipulant les XATTR, voir sous-section 5.1), réalisé au cours de ce projet. Les trois autres sont disponibles sur crates.io, il s'agit de walkdir [41], petgraph [42] et notify [43].

walkdir est une librairie pour parcourir efficacement et de manière récursive une arborescence de fichiers. Elle offre différentes structures, dont `WalkDir` et `DirEntry`. `WalkDir` attend un chemin de répertoire et retourne un itérateur listant chaque sous-répertoire et fichier contenu dans le répertoire de départ. Chaque entrée de cet itérateur est représenté par la structure `DirEntry`, qui détient des méthodes pour obtenir des informations sur l'entrée (le chemin complet, les méta-données, si c'est un répertoire ou un fichier, son nom, etc.). Le listing 36 illustre un exemple de parcours d'un répertoire et de l'affichage des informations pour chaque entrée rencontrée. walkdir est utilisé pour réaliser le premier balayage de l'arborescence fournie et constituer le graphe (voir listing 38).

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

Listing 36 – Parcours d'un répertoire avec walkdir

petgraph est une librairie de représentation de graphes. Elle fournit différentes structures de données pour représenter un graphe et des modules de manipulation et parcours de graphes. Les structures de données sont au nombre de trois :

1. `Graph<N, E, Ty = Directed, Ix = DefaultIx>` : représente un graphe et ses données sous forme de deux listes d'adjacences (deux `Vec`, un pour les noeuds et l'autre pour les arcs ou arêtes). Les noeuds sont de type générique `N`, les arcs ou arêtes de type générique `E`, le graphe est par défaut orienté (type `Directed`) et le type pour l'index (l'identifiant numérique d'un noeud et d'un arc ou arête dans les vecteurs respectifs, déterminant la taille maximale du graphe) est par défaut `u32` (ce qui autorise plus de quatre milliards de noeuds, 4'294'967'296 précisément).
2. `StableGraph<N, E, Ty = Directed, Ix = DefaultIx>` : semblable à `Graph`, avec une différence notable, elle garde les identifiants des noeuds et arcs ou arêtes supprimés. Cette structure comporte également moins de méthodes.
3. `GraphMap<N, E, Ty>` : représente un graphe et ses données sous forme d'une table associative, dont les clés sont les noeuds de type générique `N`, avec l'obligation pour ce type d'être conforme pour l'utilisation en tant que clé. Permet de tester l'existence d'un noeud en temps constant avec la contrepartie de ne pas pouvoir stocker plus qu'un noeud avec une même donnée.

Avant de continuer les explications sur l'utilisation de petgraph, faisons un bref rappel de l'architecture choisie pour représenter l'arborescence des fichiers et tags à surveiller. Un noeud du graphe peut être soit un fichier, soit un répertoire, soit un tag. Le lien entre un répertoire et un sous-répertoire ou un fichier est symbolisé par un arc partant du répertoire en question. Le lien entre un tag et un répertoire ou un fichier est également un arc partant du tag en question. Un arc n'a pas de données à sauvegarder, il n'a donc pas de type effectif. Pour accélérer l'accès à un tag dans le graphe à partir de son nom, une *hashmap* associant le nom du tag à son identifiant dans le graphe est maintenue.

La structure `StableGraph` a été choisie en raison de sa rétention des identifiants lors des suppressions de noeuds et d'arcs. En effet, lors des opérations de suppression de tags ou de fichiers, pour maintenir cohérente la *hashmap* des tags associés aux identifiants du graphe, il ne faut pas que cesdits identifiants changent. Dans le cas contraire, cette *hashmap* ne serait pas utilisable. Le listing 37 montre les structures de données utilisées pour notre graphe. Un noeud du graphe est représenté par la structure `Node`, contenant le nom du noeud et son genre (tag, fichier ou répertoire). Un arc est simplement défini par un type vide, nommé `Nil`. petgraph offre de nombreuses méthodes pour `StableGraph` : accès, ajout et suppression de noeuds et d'arcs, un itérateur sur les voisins d'un noeud (avec indication du sens), recherche d'arcs entre deux noeuds, etc. La fonction `make_graph()` listée au listing 38 est la fonction première de notre module, elle crée le graphe et la *hashmap* avec walkdir à partir du chemin du répertoire racine. Elle retourne le graphe contenant les fichiers, répertoires et tags trouvés, la *hashmap* remplie et l'identifiant du noeud racine, point de départ pour les mises à jour suivantes du graphe (sur les fichiers et répertoires). La fonction `make_subgraph()`, appelée dans la boucle s'assure d'ajouter correctement les noeuds en fonction du chemin, pour éviter

de créer deux fois le même noeud ou pour qu'un noeud enfant ne soit créé avant son parent (il n'y a pas de garantie de parcourir l'arborescence des fichiers dans un ordre parent-enfants).

```rust
#[derive(Debug, Clone)]
pub struct Nil;

#[derive(Debug, Clone)]
pub enum NodeKind {
    Tag,
    File,
    Directory
}

#[derive(Clone)]
pub struct Node {
    pub name : String,
    pub kind : NodeKind
}

pub type MyGraph = StableGraph<Node, Nil>;
```

Listing 37 – Structures pour les noeuds et les arcs du graphe dans `src/graph.rs`

```rust
pub fn make_graph(path_root : String, base_path : String)
    -> (MyGraph, HashMap<String, NodeIndex>, NodeIndex) {
    let mut graph : MyGraph = StableGraph::new();
    let mut tags_index = HashMap::new();
    let local_root = local_path(&mut path_root.clone(),
        base_path.clone());
    let root_index = graph.add_node(
        Node::new(local_root, NodeKind::Directory)
    );
    update_tags(path_root.clone(), &mut tags_index,
        &mut graph, root_index);
    let mut is_root = true;

    for entry in WalkDir::new(path_root).into_iter()
        .filter_map(|e| e.ok()) {
        if is_root {
            is_root = false;
            continue;
        }
        let mut path = entry.path().display().to_string();
        let path = local_path(&mut path, base_path.clone());
        make_subgraph(root_index, &mut tags_index, &mut graph,
            path, base_path.clone());
    }
    (graph, tags_index, root_index)
}
```

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

Pour chaque fichier et répertoire, la fonction `update_tags()`, listée au listing 39, compare les tags dans les XATTR aux tags présents dans le graphe et met à jour les relations si besoin.

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

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

notify est une librairie multiplateforme de notifications d'événements sur le FS. Elle utilise différentes implémentations selon sur quel OS elle est utilisée. Sur Linux, elle repose sur `inotify`. Elle attend un chemin de fichier ou répertoire. Elle dispose également d'une option récursive pour la surveillance d'un répertoire. Elle offre deux API distinctes :

— *Debounced* API (par défaut) : retourne tous les événements avec un pré-traitement effectué par notify, regroupant certains événements en un seul, par exemple : le renommage d'un fichier, événement *create* unique lors de la création d'un fichier plutôt qu'un *create* + *write* + `chmod`. Les événements sont envoyés après un délai (défini à la création), pour justement pouvoir les regrouper en amont si nécessaire. Chaque événement est un membre de l'énumération `DebouncedEvent`.

— *Raw* API : retourne tous les événements sans pré-traitement par notify et de manière immédiate. Elle a l'avantage d'être exhaustive mais davantage de traitement logique doit être effectué (notamment pour les événements de renommage). Chaque événement est contenu dans la structure `RawEvent`, contenant elle-même le chemin du fichier ayant subi l'événement (`path`), l'événement en question (`op`) et un "cookie" faisant le lien entre deux sous-événements faisant partie d'un seul événement de renommage (voir section 4.3 pour plus de détails).

Debounced API est utilisée ici pour faciliter la détection d'événements de renommage. Les deux API nécessitent la création d'un canal de communication entre deux threads ou plus : notify aura le rôle d'un producteur en émettant les événements. Dans une boucle infinie, les événements nécessaires à la mise à jour du graphe sont attrapés et traités. C'est la partie

thread consommateur du canal. Les événements mettant à jour du graphe sont les suivants :

— Création de fichier ou répertoire.

— Modification des attributs (dans notre cas, les tags).

— Suppression de fichier ou répertoire.

— Renommage de fichier ou répertoire.

Le détail de ce mécanisme est disponible au listing 40, des lignes 14 à 33.

Mutex et références multiples Étant donné que le thread `main`, consommant les événements émis par notify, et le thread socket serveur, écoutant sur les requêtes entrantes, ont besoin tous deux de lire et modifier le graphe et la *hashmap* associée, la règle de base d'un seul *owner* par valeur n'est pas respectée. Par ailleurs, il ne faut pas que les threads accèdent en même temps à ces deux variables, au risque de se retrouver avec des données incohérentes. Les solutions à ces deux problèmes sont les références multiples atomiques, ou *Atomic Reference Counting* (`Arc`) et les verrous, ou `Mutex`. Les premières autorisent plusieurs *owner* simultanés et concurrents à une valeur, au prix d'une légère dégradation de performances pour des questions de sécurité mémoire. Les seconds sont des verrous classiques en programmation concurrente, comme ceux existant en C par exemple. Dans le listing 40, nous pouvons voir aux lignes 5 et 6 le graphe et la *hashmap* être enveloppés dans un `Mutex`, lui-même enveloppé dans un `Arc` et aux lignes 7 et 8 la création d'une référence à ce graphe et *hashmap* grâce à la fonction `Arc::clone()`. À partir de là, si le thread `main` désire utiliser le graphe ou la *hashmap*, il lui faut prendre le verrou, comme aux lignes 23 et 24 de ce même listing. Le thread serveur socket doit faire de même de son côté, c'est la raison pour laquelle à la ligne 11 il reçoit les variables `graph` et `tags_index` définies aux lignes 5 et 6.

```
// Initialisation des variables partagées : graphe et hashmap
// Multiples références possibles grâce à 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);
let main_tags_index = Arc::clone(&tags_index);
// Lancement du socket serveur dans un thread séparé
thread::spawn(move || {
    server(base_path, &graph, &tags_index);
});
// Initialisation de la surveillance avec notify
let (tx, rx) = channel();
let mut watcher = watcher(tx, Duration::from_secs(1)).unwrap();
watcher.watch(path, RecursiveMode::Recursive).unwrap();
loop {
    match rx.recv() {
        match event {
            Create(_) | Chmod(_) | Remove(_) | Rename(_, _) => {
                // Prise du verrou sur le graphe et la hashmap
                // pour modifications éventuelles
                let mut ref_graph = main_graph.lock().unwrap();
                let mut ref_tags_index = main_tags_index.lock()
                    .unwrap();
                // dispatcher s'occupe de réaliser la bonne action
                // en fonction de l'événement
                dispatcher(event, &mut ref_tags_index,
                    &mut ref_graph, root_index, base_path);
            }
        }
    }
}
```

Listing 40 – Fonction `main.rs` de Tag Engine (réduite et simplifiée, non fonctionnelle)

Serveur sockets Le fichier `src/server.rs` contient toute la logique des traitements de requêtes provenant de Tag Manager. Comme vu sur le listing 40 aux lignes 10 à 12, le serveur est lancé dans un thread séparé. De ce fait, Tag Engine peut écouter à la fois sur les événements du FS et sur les requêtes provenant de Tag Manager. Trois types de requêtes sont implémentés, le détail est disponible dans la table 4 de la sous-section 5.1.1. Le code reçu est converti en une `enum`, `RequestKind`, visible dans le listing 41, pour faciliter la manipulation par *pattern matching*.

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

Listing 41 – Énumération `RequestKind` dans le fichier `server.rs`

Chaque requête est tout d'abord analysée avec la fonction `parse_request()` qui détermine si la requête est valide et de quel type il s'agit. Les verrous sont ensuite pris pour accéder au graphe et à la *hashmap* pour construire la réponse. Les fonctions lisant et écrivant les requêtes et réponses manipulent un `UnixStream`.

Il y a un mécanisme intéressant à montrer dans la fonction `request_tags()` recopiée dans le listing 42. La fonction en elle-même n'est pas très complexe, elle prend le verrou sur la *hashmap* des tags associés aux identifiants des noeuds du graphe et retourne un vecteur contenant tous les tags, *id est* les clés de la table associative. C'est cette dernière opération qui est particulièrement puissante, aux lignes 5 et 6 de la fonction. Pour générer le vecteur `entries` à partir des clé de `tags_index`, la fonction `collect()` est utilisée. Cette fonction permet de transformer un itérateur en un autre, très simplement.

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

Listing 42 – Illustration de l'utilisation de la fonction `collect()`

Analyse d'une expression logique Le fichier `src/parse.rs` contient principalement une fonction transformant une expression logique infixe en postfixe (appelée également "notation polonaise inverse" [44]). Dans le cadre de ce programme, elle est utilisée par la requête fournissant un ou plusieurs tags, séparés par des "et" ou des "ou" logiques pour récupérer la liste des fichiers et répertoires correspondants. Le type d'expression infixe est la manière "naturelle" en mathématiques ou en logique de déclarer la séquence d'opérateurs et opérandes d'un calcul ou d'une expression logique booléenne. Les exemples suivants sont plus explicites : l'expression logique infixe `bob OR fred AND max` se traduirait en l'expression postfixe `bob fred max AND OR`. L'algorithme utilisé pour implémenter cette fonction est disponible ici [45]. Le listing 43 montre les deux `enum` utilisée pour la conversion. Un `Arg` est soit un opérande avec un nom, soit un opérateur "AND" ou "OR". La méthode `compare()`, inspirée de Java, compare les deux opérateurs pour donner la priorité à "AND" sur "OR". Cette méthode illustre la puissance des *pattern matching*, destructurant deux variables en même temps de manière élégante.

```
5  #[derive(Debug, Clone, PartialEq)]
6  pub enum Operator { AND, OR }
7  impl Operator {
8      fn compare(&self, other : &Operator) -> i8 {
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 – Énumérations `Operator` et `Arg` et méthode `compare()`

La raison de manipuler une expression postfixe plutôt qu'infixe est que l'algorithme d'évaluation postfixe est bien plus simple à implémenter, il ne nécessite qu'une pile pour stocker les opérateurs [44]. L'algorithme est disponible dans le 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 – Algorithme d'évaluation d'une expression postfixe - [44]

Il n'est ainsi pas nécessaire d'établir une grammaire d'analyse d'expressions, comme pour un analyseur de code source ou d'expressions régulières. Pour l'évaluation simple d'une expression comportant deux opérateurs différents uniquement, cette solution est largement satisfaisante et efficace. La fonction convertissant l'expression infixe vers postfixe se nomme `infix_to_postfix()` et se trouve dans le fichier `src/parse.rs` et la fonction implémentant l'algorithme d'évaluation d'une expression postfixe se nomme `expression_to_entries()` et se trouve dans le fichier `server.rs`. Les opérandes de l'expression sont des tags et les deux opérateurs sont "AND" et "OR". "AND" a la priorité sur "OR", comme la multiplication sur l'addition. Pour chaque tag, un ensemble au sens mathématique des fichiers et répertoires vers lesquels il pointe est réalisé et est placé sur la pile. Lorsqu'un opérateur survient, les deux derniers ensembles d'entrées sont dépilés et l'opération ensembliste correspondante leur est appliquée (une intersection pour un "AND" et une union pour un "OR"). Le nouvel ensemble ainsi obtenu est de nouveau poussé sur la pile. À la fin de l'algorithme, l'ensemble final est retourné. La figure 11 résume le fonctionnement du programme :

1. Thread `main`, maintenant graphe et *hashmap* et surveillant l'arborescence.
2. Thread serveur sockets répondant aux requêtes.

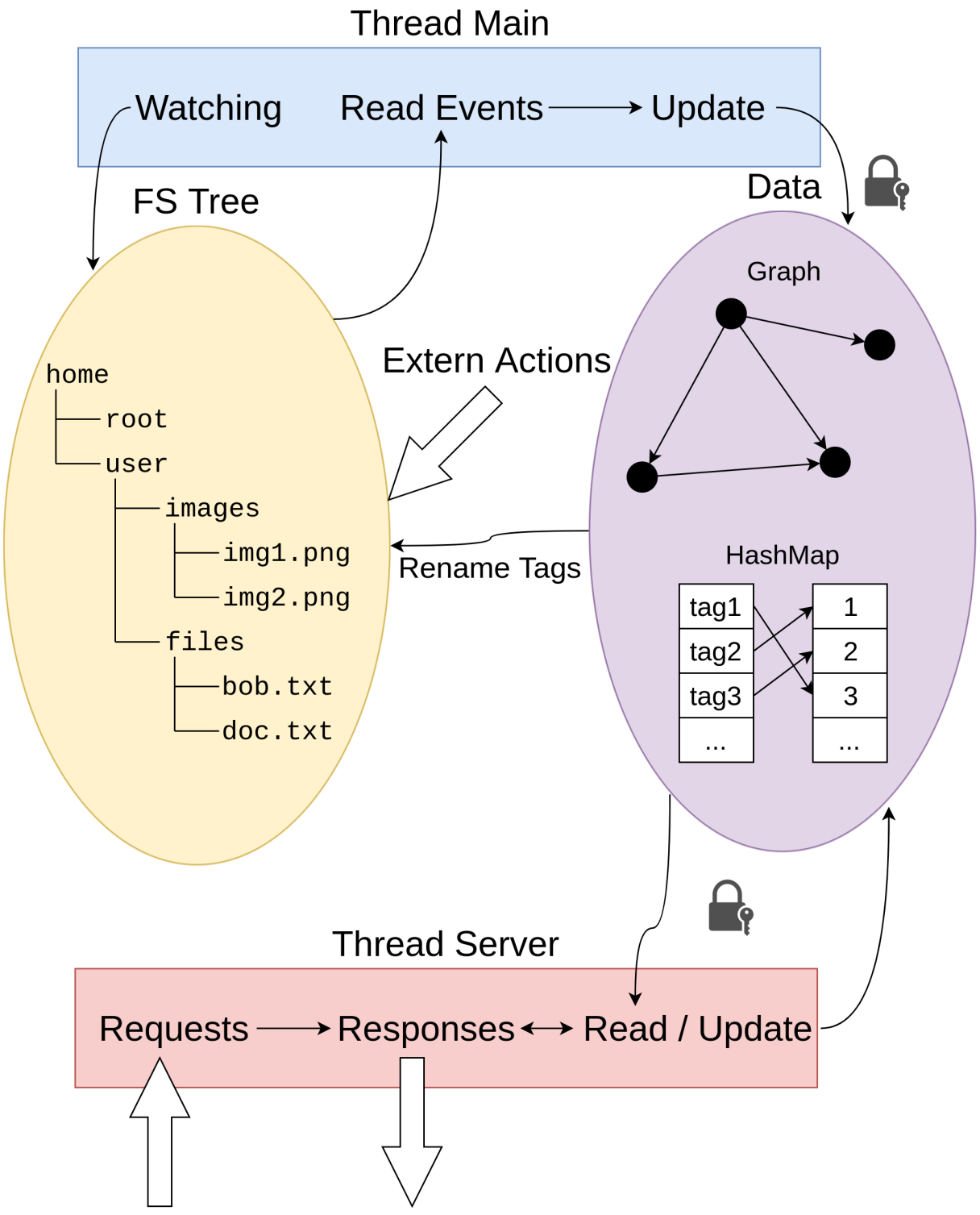


FIGURE 11 – Schéma de fonctionnement de Tag Engine

5.2.2 Utilisation du programme et exemples

Le programme attend un chemin absolu valide pointant vers un répertoire. Il admet un argument facultatif, `-d` ou `--debug`, qui imprime sur la sortie standard l'état du graphe et de la *hashmap* et écrit deux fichiers, `graph.dot` et `graph.png`, après chaque événement survenu. Le listing 45 montre les *outputs* de Tag Engine correspondants aux opérations sur l'arborescence manipulée par Tag Manager (voir section 5.1.2 et le listing 35). On aperçoit aux lignes 11 à 16 l'ajout des tags "in_a" et "myfiles" et aux lignes 30 et 31 les requêtes de tags et d'entrées effectuées. Les deux fichiers représentent également l'état du graphe. Le fichier `graph.dot` respecte la syntaxe d'un outil appelé Graphviz [46], dédié à la modélisation et visualisation de graphes. Il comprend plusieurs moteurs de génération de graphes à partir de données texte ou brutes, dont `dot` utilisé ici. À partir du fichier `graph.dot`, disponible au listing 46, la figure 12 est obtenue. petgraph donne la possibilité de générer un fichier respectant la sémantique attendue par `dot` pour dessiner un graphe. Nous pouvons voir que le résultat obtenu concorde avec l'arborescence obtenue.

```
$ ./tag_engine /home/stevenliatti/a -d
graph StableGraph {
    Ty: "Directed", node_count: 6, edge_count: 5,
    edges: (0, 1), (1, 2), (1, 3), (0, 4), (0, 5),
    node weights: {
        0: Directory "a", 1: Directory "b", 2: File "b1",
        3: File "b2", 4: File "a2", 5: File "a1"
    },
    free_node: NodeIndex(4294967295),free_edge: EdgeIndex(4294967295)
}, tags_index {}
chmod : "a/a1"
chmod : "a/a2"
chmod : "a/b"
chmod : "a/b"
chmod : "a/b/b1"
chmod : "a/b/b2"
graph StableGraph {
    Ty: "Directed", node_count: 8, edge_count: 11,
    edges: (0, 1), (1, 2), (1, 3), (0, 4), (0, 5), (6, 5), (6, 4),
        (6, 1), (7, 1), (7, 2), (7, 3),
    node weights: {
        0: Directory "a", 1: Directory "b", 2: File "b1",
        3: File "b2", 4: File "a2", 5: File "a1",
        6: Tag "in_a", 7: Tag "myfiles"
    },
    free_node: NodeIndex(4294967295),free_edge: EdgeIndex(4294967295)
}, tags_index {
    "in_a": NodeIndex(6), "myfiles": NodeIndex(7)
}
Request for Tags
Request for Entries "myfiles AND in_a"
```

Listing 45 – Exemple d'utilisation de Tag Engine

```
digraph {
    0 [label="Directory \"a\""]
    1 [label="Directory \"b\""]
    2 [label="File \"b1\""]
    3 [label="File \"b2\""]
    4 [label="File \"a2\""]
    5 [label="File \"a1\""]
    6 [label="Tag \"in_a\""]
    7 [label="Tag \"myfiles\""]
    0 -> 1
    1 -> 2
    1 -> 3
    0 -> 4
    0 -> 5
    6 -> 5
    6 -> 4
    6 -> 1
    7 -> 1
    7 -> 2
    7 -> 3
}
```

Listing 46 – Fichier `dot` produit par petgraph

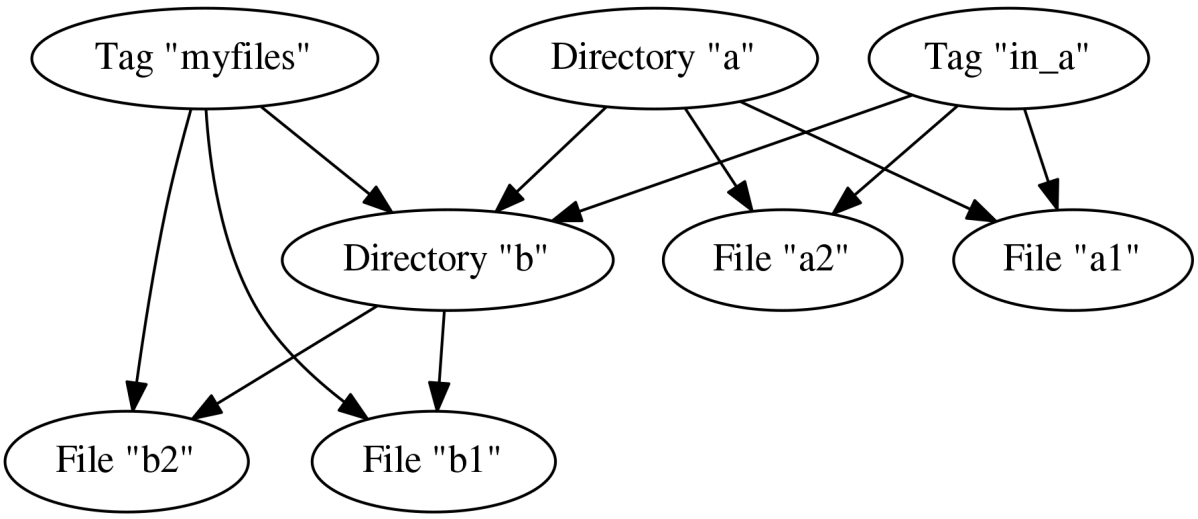


FIGURE 12 – Image du graphe obtenue avec la commande `dot`

5.3 TagFS

La figure 13 résume le fonctionnement global du système TagFS, comportant Tag Manager et Tag Engine :

1. Gestion des tags avec Tag Manager.
2. Indexation et surveillance de l'arborescence et serveur avec Tag Engine.

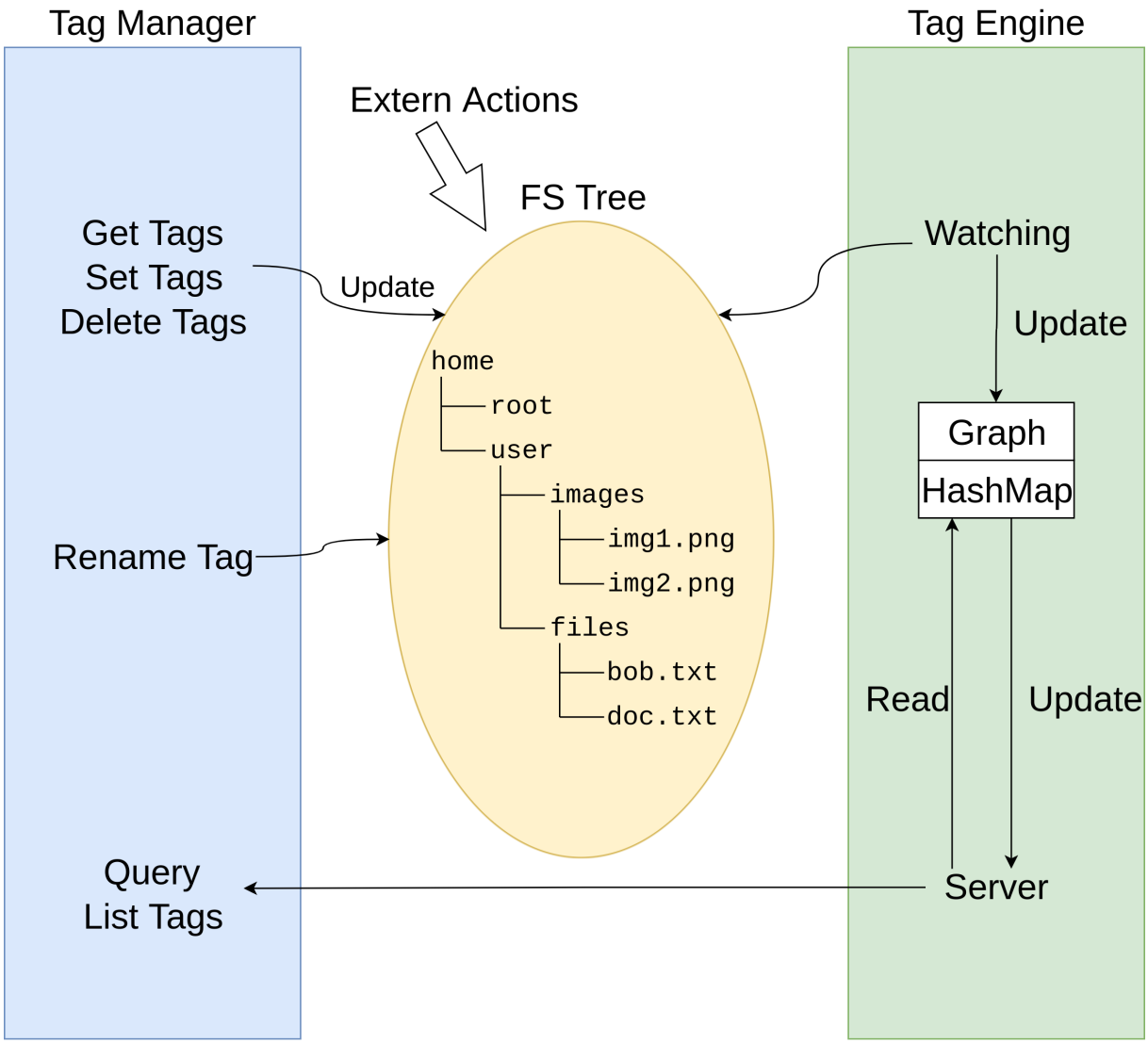


FIGURE 13 – Schéma de fonctionnement global de TagFS

6 Tests

6.1 Mesures de performances

Des mesures de temps d'exécution ont été réalisées avec une version de Tag Engine légèrement modifiée : la fonction `main()` a été tronquée à partir du lancement du thread socket serveur et de la boucle infinie écoutant sur les événements survenus sur l'arborescence surveillée. Ainsi, le programme parcoure une seule fois l'arborescence et construit le graphe et la table de hachage associée. Cette modification a été faite dans le but de mesurer plusieurs exécutions du programme (avec le type `Duration` de Rust) pour un répertoire donné pour réaliser une moyenne du temps. Elle est illustrée au listing 47.

```
1  fn main() {
2      // ...
3      let now = Instant::now();
4      let (graph, tags_index, root_index) =
5          tag_engine::graph::make_graph(
6              String::from(absolute_path_root), base_path.clone()
7          );
8      let new_now = Instant::now();
9      let elapsed = new_now.duration_since(now);
10     println!("{}", elapsed.as_secs() as f64 +
11         elapsed.subsec_nanos() as f64 * 1e-9);
12 }
```

Listing 47 – `main.rs` de Tag Engine modifié pour mesurer le temps d'exécution

En tout, 200 exécutions ont été réalisées, 100 avec le programme compilé en mode *debug* (`cargo build`, non optimisé) et 100 avec le mode *release* (`cargo build --release`, avec optimisations maximum). Les versions de Cargo et rustc sont les suivantes : cargo 0.26.0 (41480f5cc 2018-02-26) et rustc 1.25.0 (84203cac6 2018-03-25). Les répertoires cibles diffèrent grandement dans leur nombre de sous-répertoires et fichiers contenus, allant de cinq répertoires et 863 fichiers à plusieurs milliers de répertoires et une centaine de milliers de fichiers (15'172 répertoires et 112'046 fichiers précisément). Ces répertoires ne contiennent pas de tags. Le tableau 6 dresse les répertoires utilisés et leur contenu.

Répertoire	Nombre de répertoires	Nombre de fichiers
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
Musique	135	1'352

TABLE 6 – Répertoires utilisés pour les mesures de temps d'exécution

Le script `bash` utilisé pour réaliser ces 200 exécutions est montré au listing 49. Il écrit pour chaque répertoire deux fichiers contenant les mesures des 100 exécutions pour les deux versions compilées du programme. La dernière boucle du script exécute le fichier `average.m`, disponible au listing 48, qui fait la moyenne des 100 mesures. La machine utilisée pour compiler et exécuter les mesures a les caractéristiques matérielles et logicielles suivantes (les commandes `lscpu`, `lshw`, `uname -r` et `lsb_release -a` ont été utilisées) :

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

— Mémoire vive : 4x4 Go DDR3 1333 MHz.

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

— Disque système : Samsung 850 EVO Basic 500 Go.

— 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 – Script Octave pour calculer la moyenne des exécutions

```bash
#!/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 – Script bash pour exécuter 100 mesures de temps d'exécution

Les moyennes obtenues sont représentées sur la figure 14. L'axe des ordonnées représente le temps moyen d'exécution du programme. L'axe des abscisses représente les sept répertoires utilisés pour ce test. Pour chaque répertoire, il y a une mesure du programme compilé en mode *debug* (barre bleue) et une mesure du programme compilé en mode *release* (barre orange). On remarque que les deux répertoires contenant le plus de fichiers, à savoir `Android` et `Documents`, sont ceux dont le temps d'exécution est le plus long. Globalement, moins un répertoire contient d'entrées, moins le temps d'exécution sera long. À deux répertoires équivalents en termes d'entrées, les variations qui peuvent survenir sont très certainement dûes aux spécificités des fichiers contenus, comme leur taille sur le disque.

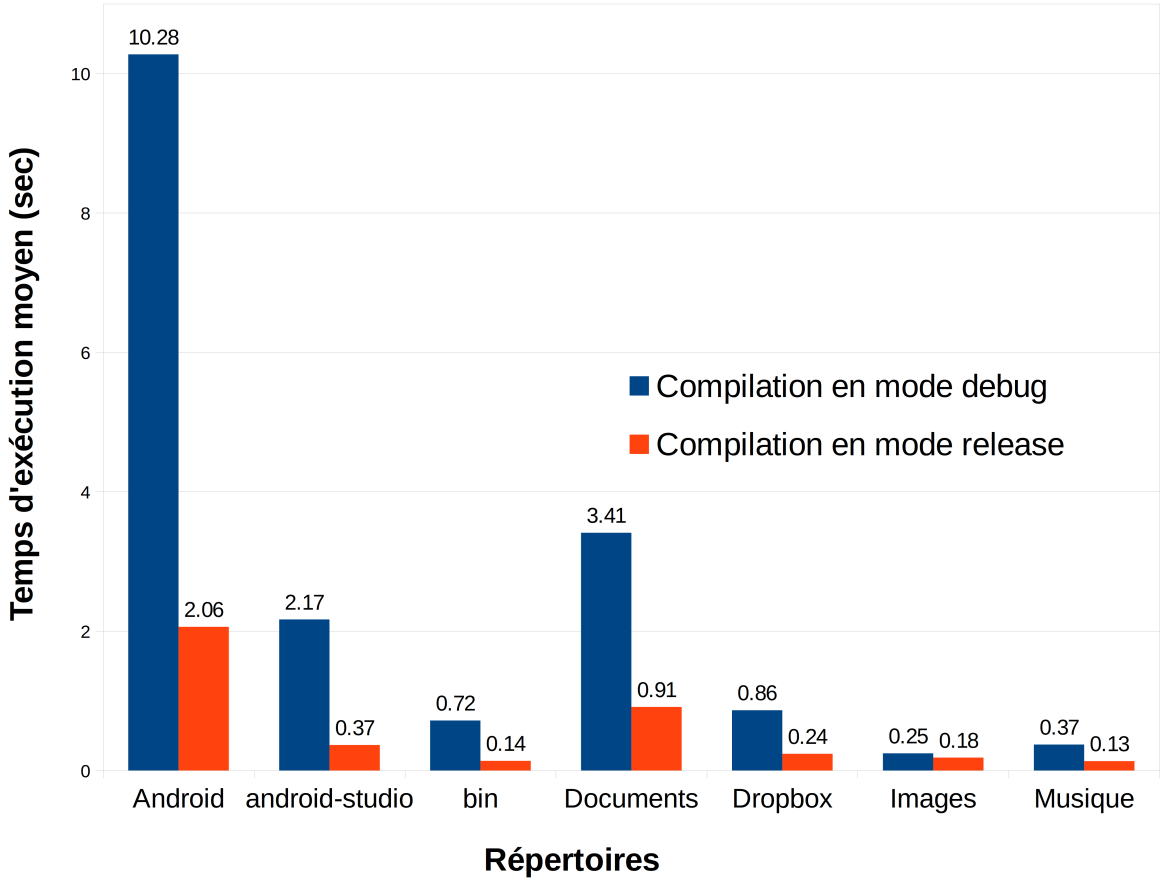


FIGURE 14 – Temps d'exécution en fonction du répertoire et par type de mode de compilation

Il est intéressant de remarquer que le rapport de temps d'exécution entre la version non-optimisée et optimisée peut être important. La figure 15 illustre ces rapports de temps. Nous voyons que la différence varie entre 5.93 pour `android-studio` et 1.33 pour `Images`, de manière générale ce sont à nouveau les répertoires avec le plus d'entrées qui ont les rapports les plus importants. Le répertoire `Images` contenant peu d'éléments, moins d'opérations sont exécutées, il est donc plus difficile d'optimiser ce nombre réduit d'opérations durant l'exécution du programme. La leçon à tirer de ces rapports est que le compilateur Rust est capable de grandes optimisations sur le code.

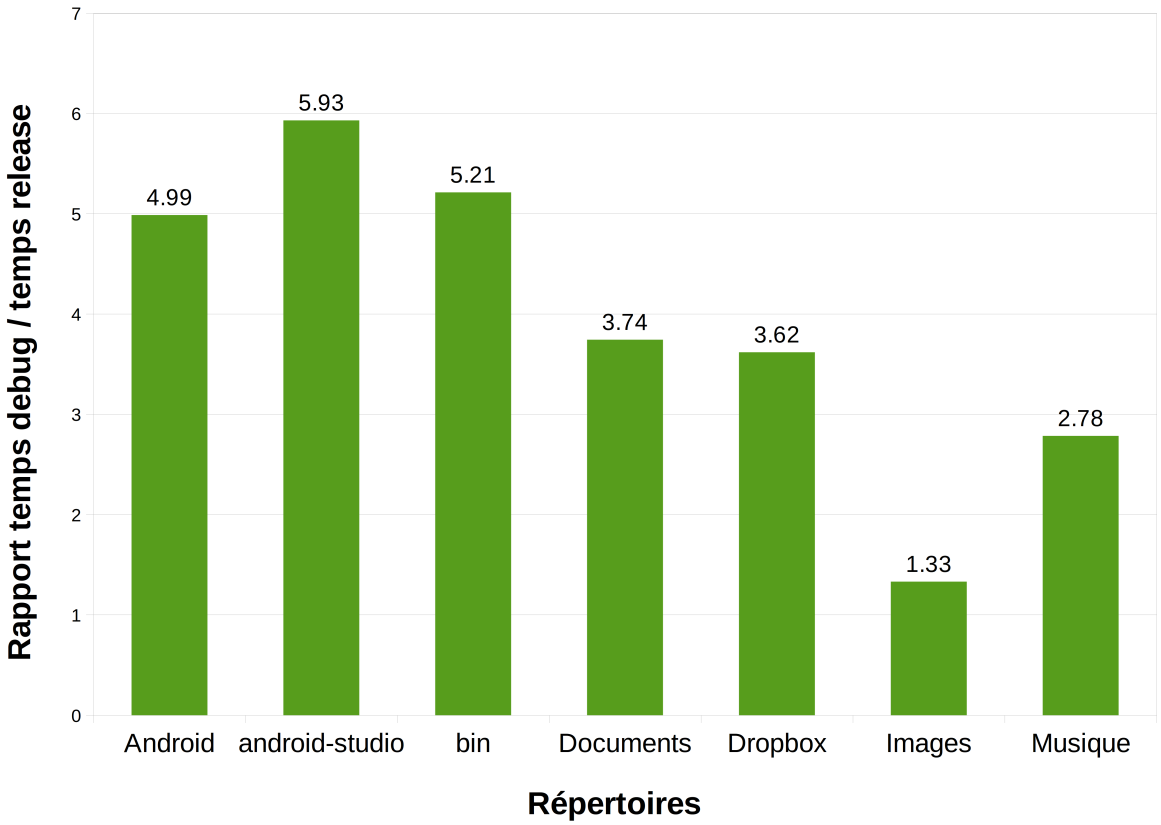


FIGURE 15 – Rapport entre le temps d'exécution en mode *debug* et *release*

7 Discussion

7.1 Rust VS C

Dans cette sous-section, nous allons discuter de l'implémentation du système en Rust plutôt qu'en C. Les avantages sont nombreux et les inconvénients sont acceptables.

7.1.1 Avantages de Rust par rapport à C

L'avantage principal de Rust par rapport à C est la garantie sur la sécurité de la mémoire (*safe*). Tant que le code compile, il sera sûr en termes d'allocation/désallocation mémoire et concurrence (les erreurs de logique sont néanmoins toujours à la charge du programmeur). Cette garantie est fournie par le compilateur de Rust, qu'on pourrait définir d'"intelligent". Intelligent car la quasi-totalité des erreurs liées à la mémoire sont détectées à la compilation. Il va même plus loin en étant très verbeux sur les erreurs survenues, en suggérant souvent le moyen de résoudre le problème. Ce qui, avec le temps, sont devenues des bonnes pratiques à connaître lorsque nous programmons en C, Rust force les programmeur à les implémenter avant même que le code soit compilé. Les listings de code suivants illustrent deux exemples où Rust se comporte "mieux" que C. Les listings 50 et 51 montrent une fonction, respectivement écrite en C et en Rust, qui retourne un pointeur indéfini. En C, la détection de l'erreur surviendrait à l'exécution du programme (bien que selon le compilateur C utilisé, un avertissement, *warning*, sera affiché au programmeur), avec le fameux *Segmentation fault*. L'équivalent Rust refuserait de compiler ce code, à cause de la règle de la durée de vie non respectée.

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

Listing 50 – Création d'un pointeur indéfini en C

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

Listing 51 – Création d'un pointeur indéfini en Rust

Un autre exemple est l'accès non autorisé à de la mémoire. Les listings 52 et 53 montrent une fonction, respectivement écrite en C et en Rust, qui imprime sur la sortie standard un élément d'un tableau à un index non correct. En C, "aucun problème", la fonction imprimera ce

qu'elle trouve à cette adresse mémoire, sans se plaindre. Ce genre d'erreurs est potentiellement une vulnérabilité qui peut être exploitée à des fins malicieuses. En Rust, cette erreur ne serait pas détectée à la compilation, mais au moins le programme s'arrêterait abruptement à son exécution (*panicked*) et l'erreur pourrait être corrigée.

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

Listing 52 – Accès non autorisé à de la mémoire en C

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

Listing 53 – Accès non autorisé à de la mémoire en Rust

Nous pouvons également citer que Rust est parfois plus rapide que C pour certaines tâches, ou presque aussi rapide. Le site *The Computer Language Benchmarks Game* [47] dresse une comparaison entre Rust et, un à un C, C++, Go et Java sur les temps d'exécution, quantité de mémoire et charge processeur pour une variété d'applications. Rust comparé à C et C++ est parfois plus rapide et souvent distancé de peu. Rust comparé à Go et Java est toujours le plus rapide. On peut en conclure que sécurité peut rimer au propre comme au figuré avec rapidité.

Un autre grand avantage de Rust sur C est sa gestion des erreurs et l'absence de `NULL`. Tony Hoare, un chercheur en informatique, est l'inventeur de `NULL` et l'appelle aujourd'hui son erreur à un milliard de dollars [48]. `NULL` autorise de nombreuses erreurs inattendues, malgré le fait que l'idée n'est pas mauvaise (il est pratique de déclarer une variable ne contenant pas de valeur). En Rust, `NULL` n'existe pas, à la place une énumération très puissante est disponible, `Option` (voir listing 13). Elle évite ainsi de manipuler des données vides.

Viennent ensuite plusieurs avantages moins déterminants, mais toutefois bien pratiques dans la vie de tous les jours du programmeur. Citons par exemple le formidable outil Cargo et la source de *crates* disponible sur crates.io, contenant des milliers de librairies réalisées et éprouvées par la communauté. Les tests unitaires du programme sont intégrés directement au langage et leur exécution est facilitée avec Cargo. La librairie standard de Rust est bien fournie, notamment avec les collections (vecteurs, tables de hachage, etc.). Enfin, s'il fallait terminer, la gestion des types génériques directement inclue au langage est un atout bien pratique.

7.1.2 Inconvénients de Rust par rapport à C

Rust a malgré tout deux défauts, plus ou moins handicapants selon le temps et la situation, liés l'un à l'autre. Le premier est la courbe d'apprentissage du langage pouvant être longue et décourageante. Rust est régi par certaines règles contraignantes, simples mais pas faciles à cerner dans toutes les situations. C'est notamment le cas des règles d'*ownership*, de *borrowing* et des références (voir sous-section 4.1.8), assez uniques à Rust. Le deuxième défaut découle du premier : il faut parfois revoir certains algorithmes ou structures de données facilement implémentables en C ou autres langages similaires. C'est notamment ce qu'il s'est produit durant l'implémentation de Tag Engine et sa structure de données pour contenir l'arborescence des répertoires, fichiers et tags (voir sous-section 7.2 pour plus de détails). D'autres petits défauts peuvent être mentionnés, comme par exemple le manque d'arguments par défaut lors des déclarations de fonctions ou la surcharge de méthodes ou fonctions. Ce ne sont pas des défauts à proprement parler, mais des facilités qu'il aurait été souhaitable de disposer. Finalement, le frein majeur à l'adoption massive du langage (comme bien d'autres nouveau langages) est le manque de soutien de la part d'une importante société (même si Mozilla l'utilise pour son navigateur Firefox) et la "flemme" des programmeurs de se détourner de C ou C++.

7.2 Problèmes rencontrés

Le principal problème rencontré durant ce travail a été l'implémentation d'un arbre en Rust pour les besoins de Tag Engine. L'implémentation simple d'un noeud d'un arbre en C est similaire à celle décrite au listing 54.

```
1 struct Node {
2     int data; // le type n'est pas important pour l'exemple
3     struct Node** children;
4 };
```

Listing 54 – Implémentation de la structure d'un arbre en C

C'est une structure récursive et relativement facile à comprendre : un noeud est constitué d'une donnée d'un type choisi et d'un tableau de pointeurs vers des noeuds enfants. Ce genre de déclaration est impossible en Rust, car le compilateur a besoin de connaître la taille exacte des données à la compilation (éviter les structure récursives infinies). Une solution est d'envelopper les noeuds enfants dans un pointeur intelligent de type `Box` (voir chapitre 15 du *book* [20]). Mais un autre problème surviendra au moment de l'utilisation des noeuds, car Rust ne permet pas plusieurs références mutables vers une même valeur. Pour résoudre ce dernier problème, l'utilisation des `Rc<T>`, ou *reference counting*. C'est un autre pointeur intelligent, qui autorise plusieurs *ownership* pour une seule valeur. On se retrouve alors avec

une imbrication de pointeurs avancés, comme dans le listing 55, solution proposée par Rust Leipzig [49] :

```rust
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 – Structure d'un noeud en Rust avec pointeurs intelligents - [49]

Non seulement cette solution n'est pas très lisible mais en plus elle a le défaut de supprimer tous les avantages qu'offre le compilateur Rust sur les contraintes de portée et durée de vie des variables. Une solution qui semble résoudre tous les problèmes précédents est décrite au listing 56. Elle utilise une *arena*, une région mémoire [50]. Plutôt qu'avoir des pointeurs entre noeuds de l'arbre, cette solution utilise une collection pour stocker les données (ici un vecteur) et les relations entre noeuds sont tenues grâce aux identifiants dans cette même collection (ici les indices du vecteur). Ainsi, le problème de durée de vie des valeurs et des multiples *ownership* est résolu, car l'*arena* est la seule détentrice des données (donc des noeuds). Les noeuds restent accessibles depuis l'extérieur de l'*arena* en gardant leurs identifiants dans l'*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: usize,
}
```

Listing 56 – Structure d'un noeud en Rust avec une *arena* - [49]

Le *crate* petgraph utilisé pour stocker les données des fichiers, répertoires et tags d'une arborescence fonctionne de cette manière : il utilise deux vecteurs, l'un pour les noeuds, l'autre pour les arcs, pour stocker les données. Ce *crate* répondait parfaitement aux besoins de la nouvelle architecture pour l'index des fichiers, répertoires et tags : il fournit une implémentation d'un graphe et permet l'accès aux noeud de l'extérieur du graphe. Pour plus de détails sur cette sous-section, ces différents articles abordent cette problématique et donnent davantage de détails sur les possibles solutions [21] [51] [49] [52] [53] [54].

7.3 Résultats et améliorations futures

Les résultats des mesures de performances réalisés à la sous-section 6.1 montrent que le parcours initial du graphe est relativement rapide, tout du moins se déroule avec très peu de latence même pour un répertoire contenant plus de 15'000 répertoires et plus de 100'000 fichiers (une moyenne d'exécution de deux secondes pour scanner toute l'arborescence). Il est raisonnable de dire que cette opération est parmi celles les plus lourdes pour une arborescence donnée, le programme peut donc être labélisé performant. Cependant, pour s'en assurer totalement, il faudrait réaliser des tests supplémentaires lors de plus longues utilisations.

Bien que le cahier des charges ait été rempli, l'état du système au moment du rendu n'était pas parfait, voici une liste non exhaustive d'améliorations qui pourraient être apportées :

— Intégration à un gestionnaire de fichiers Linux (Nautilus, Nemo, etc.) pour la manipulation des tags (ajout, suppression) et pour exécuter les requêtes. Une alternative serait de réaliser une interface similaire sous forme d'application web par exemple.

— Transformation de Tag Engine en *Daemon* (un processus détaché du shell parent et qui n'imprime pas ses résultats sur la sortie standard). Il pourrait se lancer au démarrage de l'OS et écrire ses résultats et indexes dans des fichiers.

— Possibilité d'ajouter des nouveaux chemins de répertoires à surveiller après le lancement initial de Tag Engine. Actuellement, le programme se lance avec un seul chemin de répertoire à surveiller. Il faut pour cela modifier la gestion des threads et des données dans Tag Engine. Le problème peut être partiellement résolu en donnant à Tag Engine un répertoire suffisamment haut dans la hiérarchie pour surveiller un plus grand nombre de fichiers.

— Gestion des périphériques amovibles et de leurs FS. `inotify` ne permet pas de surveiller le dossier `/media` de la même manière qu'un autre répertoire. Il faut donc pouvoir détecter le branchement d'un périphérique de stockage et ajouter une nouvelle surveillance sur le répertoire monté.

— Réaliser davantage de tests, plus globaux que des tests unitaires et de performances.

— Créer un cache des dernières requêtes logiques adressées au serveur, pour accélérer la réponse. Le cache devrait être effacé en cas de modification de l'arbre ou de la *hashmap*.

8 Conclusion

Les deux objectifs principaux de ce projet de Bachelor étaient d'étudier et s'approprier le langage Rust et de concevoir un moteur de gestion de tags efficace et *user friendly*. Rust est un langage moderne, fiable et performant. Ses atouts sont aussi nombreux que les nouveaux concepts qu'il introduit par rapport à un langage comme C. Fort d'une communauté active et sérieuse, il y a l'espoir qu'il soit adopté par de plus en plus de développeurs pour de nombreux types d'applications. Bien que l'étude de Rust ait pris une grande partie du temps alloué à ce travail, ce ne fut pas du temps perdu. Les contraintes imposées par Rust devraient être un standard bénéfique pour de nombreux langages.

D'autres sujets ont été étudiés, comme les méthodes d'indexation, les attributs étendus des fichiers, ou les systèmes de surveillance du système de fichiers. Les limites des attributs étendus ont été montrées (notamment sur l'incompatiblité avec certains systèmes de fichiers ou partages réseaux) et du système de notification `inotify`, choisi pour ce projet. Néanmoins, ces deux technologies, avec l'association de Rust, ont permis de surpasser sur certains points les applications existantes de gestion des tags. Le cahier des charges demandé a été rempli et les interrogations sur l'usage de Rust dans ce genre d'applications vérifiées. Grâce à ce système, l'utilisateur peut maintenant étiqueter ses fichiers personnels sans avoir peur de les perdre et peut les retrouver facilement et rapidement à l'aide de requêtes logiques simples.

Ce projet est le couronnement de mes études à hepia, il m'a fait découvrir un nouveau langage plein de potentiel, enseigné de bonnes pratiques de programmation et m'a fait progresser dans la démarche de conception et réalisation d'une application système. Tout le projet est disponible à cette adresse : https://github.com/stevenliatti/tagfs.

9 Références

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

[2] Paul Ruane alias oniony. Tmsu. https://tmsu.org/. Consulté le 18.05.2018.

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

[4] Tagsistant. Tagsistant 0.8.1 howto. http://www.tagsistant.net/documents-about-tagsistant/0-8-1-howto, mars 2017. Consulté le 18.05.2018.

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

[6] TagSpaces. Your offline data manager. https://www.tagspaces.org/. Consulté le 18.05.2018.

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

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

[9] Russell Smith. Manage documents with windows explorer using tags and file properties. https://www.petri.com/manage-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/, avril 2005. Consulté le 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/, octobre 2013. Consulté le 08.05.2018.

[13] John Siracusa. Os x 10.9 mavericks : The ars technica review - tags implementation. https://arstechnica.com/gadgets/2013/10/os-x-10-9/9/, octobre 2013. Consulté le 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/, octobre 2007. Consulté le 08.05.2018.

[15] Wikipedia. Big O notation. https://en.wikipedia.org/wiki/Big_O_notation, juin 2018. Consulté le 18.06.2018.

[16] Wikipédia. Un annuaire représenté comme une table de hachage. https://fr.wikipedia.org/wiki/Table_de_hachage#/media/File:HASHTB08.svg, juin 2015. Consulté le 23.06.2018.

[17] Jean-François Hêche. *Graphes & Réseaux*. février 2012. Consulté le 13.06.2018.

[18] The Rust communitys crate registry. https://crates.io/. Consulté le 05.05.2018.

[19] r/rust. https://www.reddit.com/r/rust/. Consulté le 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 Hoffmann. Are we (i)de yet ? https://areweideyet.com/. Consulté le 25.04.2018.

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

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

[25] Wikipedia. ext4. https://en.wikipedia.org/wiki/Ext4, mai 2018. Consulté le 18.06.2018.

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

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

[28] Jack Hammons. Wsl file system support. https://blogs.msdn.microsoft.com/wsl/2016/06/15/wsl-file-system-support/, juin 2016. Consulté le 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, septembre 2017. Consulté le 13.06.2018.

[33] Denis Dordoigne. Exploiter inotify, cest simple. https://linuxfr.org/news/exploiter-inotify-c-est-simple, novembre 2014. Consulté le 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/, juillet 2014. Consulté le 13.06.2018.

[36] fanotify - monitoring filesystem events. http://man7.org/linux/man-pages/man7/fanotify.7.html, septembre 2017. Consulté le 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, mars 2018. Consulté le 14.05.2018.

[39] Steven Allen. xattr. https://crates.io/crates/xattr, juillet 2017. Consulté le 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, février 2018. Consulté le 24.05.2018.

[42] petgraph. https://crates.io/crates/petgraph, mars 2018. Consulté le 24.05.2018.

[43] Daniel Faust, Félix Saparelli, Joe Wilm, Jorge Israel Peña, Michael Maurizi, and Pierre Baillet. notify. https://crates.io/crates/notify, novembre 2017. Consulté le 24.05.2018.

[44] Wikipédia. Reverse Polish notation. https://en.wikipedia.org/wiki/Reverse_Polish_notation, juin 2018. Consulté le 29.06.2018.

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

[46] Welcome to graphviz. http://www.graphviz.org/. Consulté le 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/, décembre 2016. Consulté le 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 v.s. drop-checking. https://exyr.org/2018/rust-arenas-vs-dropck/, février 2018. Consulté le 08.06.2018.

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