Conception d'un système de "tagging" des fichiers avec Rust

Steven Liatti

Projet de bachelor - Prof. Florent Glück

Hepia ITI 3ème année

1er juillet 2018

Résumé Le but de ce projet est de concevoir et développer un "moteur de gestion de tags" pouvant gérer des dizaines de milliers de fichiers et tags associés de manière efficace, en Rust. Le stockage des tags utilisera le mécanisme des "extended attributes" disponibles dans la plupart des systèmes de fichiers modernes. Le moteur d'indexation devra surveiller les fichiers modifiés, créés, ou supprimés afin d'indexer les tags avec un minimum de latence (temps réel). Si le temps le permet, le système développé sera intégré à un environnement desktop choisi (Gnome, KDE, etc.).

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

Table des matières

1 Introduction **9**

1.1 Motivations . . . 9

1.2 Buts . . . 9

2 Analyse de l'existant **10**

2.1 Applications utilisateur . . . 10

2.1.1 TMSU . . . 10

2.1.2 Tagsistant . . . 10

2.1.3 TaggedFrog . . . 11

2.1.4 TagSpaces . . . 11

2.2 Fonctionnalités disponibles dans l'OS . . . 12

2.2.1 Windows . . . 12

2.2.2 macOS . . . 13

3 Architecture **15**

3.1 Gestion des tags . . . 15

3.2 Indexation des fichiers et des tags . . . 15

3.2.1 Indexation avec table de hachage et arbre . . . 15

3.2.2 Indexation avec un graphe et une table de hachage . . . 19

3.3 Surveillance du FS . . . 22

3.4 Recherche par tags . . . 22

4 Analyse technologique **23**

4.1 Rust . . . 23

4.1.1 Installation . . . 23

4.1.2 Cargo et Crates.io . . . 23

4.1.3 Généralités . . . 24

4.1.4 Structures de données . . . 28

4.1.5 Traits et généricité . . . 30

4.1.6 Énumérations et *pattern matching* . . . 32

4.1.7 Collections . . . 33

4.1.8 *Ownership*, *Borrowing* et références . . . 34

4.1.9 Gestion des erreurs . . . 37

4.1.10 Tests . . . 39

4.1.11 Concurrence et threads . . . 39

4.1.12 *Unsafe* Rust . . . 42

4.2 Extended attributes . . . 42

4.2.1 Petites manipulations . . . 43

4.3 `inotify` 44
4.4 Sockets 46

5 Réalisation **47**
5.1 Tag Manager 47
5.2 Tag Engine 47

6 Protocole de test **48**
6.1 *Benchmarks* 48
6.2 Mesure de performances 48

7 Discussion/résultats **49**

8 Conclusion **50**

9 Références **51**

Table des figures

1 TaggedFrog en utilisation [1] 11
2 TagSpaces en utilisation 12
3 Vue et gestion d'un tag dans le Finder macOS [2] 13
4 Un annuaire représenté comme une table de hachage - [3] 16
5 Représentation sous forme d'arbre d'une hiérarchie de fichiers et répertoires . . 17
6 Canal de communication entre threads - [4] 41

Liste des tables

1 Cas d'utilisation et opérations, première architecture 18
2 Cas d'utilisation et opérations, deuxième architecture 21

Table des listings de code source

1 `mdls` listant les tags d'un fichier sous macOS [5] . . . 14
2 Installation de Rust sur Linux ou macOS . . . 23
3 Contenu du fichier `Cargo.toml` . . . 24
4 Exemples de déclarations de variables en Rust . . . 25
5 Quelques types primitifs de Rust . . . 26
6 Exemples de fonctions en Rust . . . 26
7 Exemples de conditions en Rust . . . 27
8 Exemples de boucles en Rust . . . 28
9 Exemples de structures en Rust . . . 29
10 Bloc `impl` d'une structure en Rust . . . 30
11 Implémentations d'un trait en Rust . . . 31
12 Définition d'une `enum` et son utilisation avec un *pattern matching* en Rust . . . 32
13 L'énumération `Option` et son utilisation avec un *pattern matching* en Rust . . . 33
14 Exemples de déclarations et utilisations d'un vecteur . . . 33
15 Exemples de déclaration et utilisation d'une `HashMap` . . . 34
16 Portée d'une variable en Rust . . . 35
17 Transfert de l'*ownership* en Rust . . . 35
18 Transfert de l'*ownership* vers une fonction en Rust . . . 36
19 Emprunts de variables entre fonctions en Rust . . . 37
20 L'énumération `Result` et son utilisation avec un *pattern matching* en Rust . . . 38
21 Ouverture d'un fichier et son traitement en Rust . . . 38
22 Module de test ajouté automatiquement . . . 39
23 Création d'un thread en Rust . . . 40
24 Création d'un thread en Rust et passage d'une variable . . . 40
25 *Message passing* avec deux producteurs et un consommateur en Rust . . . 41
26 Output de `df -Th` : le disque système, les clés USB et le NFS . . . 43
27 Copie sur clé USB 8 Go, FAT32 . . . 43
28 Copie sur clé USB 8 Go, NTFS . . . 43
29 Copie sur clé USB 64 Go, ext4 . . . 44
30 Copie sur l'emplacement réseau distant, NFS . . . 44
31 Structure `inotify_event` - [6] . . . 45

Conventions typographiques

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

- Tous les mots empruntés à la langue anglaise ont été écrits en *italique*
- Toute référence à un nom de fichier (ou dossier), 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 :

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

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

Structure du document

Remerciements

Acronymes

API *Application Programming Interface*, Interface de programmation : services offerts par un programme producteur à d'autres programmes consommateurs. 14, 44, 47

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. 8, 9

FS *File System*, Système de fichiers : organisation logique des fichiers physiques sur le disque. 2, 10, 11, 16, 19, 20, 22, 42, 44, 45

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). 9

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. 2, 10, 12, 34, 42

SYSCALL *System call*, Appel système : lorsqu'un programme a besoin d'un accès privilégié à certaines parties du système d'exploitation (système de fichier, mémoire, périphériques), il demande au noyau d'exécuter l'opération voulue pour lui. 44, 45, 47

XATTR *Extended Attributes*, Attributs étendus : voir section 4.2. 9, 15, 42–44, 47

1 Introduction

XATTR CLI GUI

1.1 Motivations

1.2 Buts

[7]

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 OS. Jean-Francois Dockes en dresse également une liste avec avantages et inconvénients sur son site [7].

2.1 Applications utilisateur

2.1.1 TMSU

TMSU [8] 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 dossier 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 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 également.

2.1.2 Tagsistant

Tagsistant [9] est autre outil 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 [10]. 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 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 [1] 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.

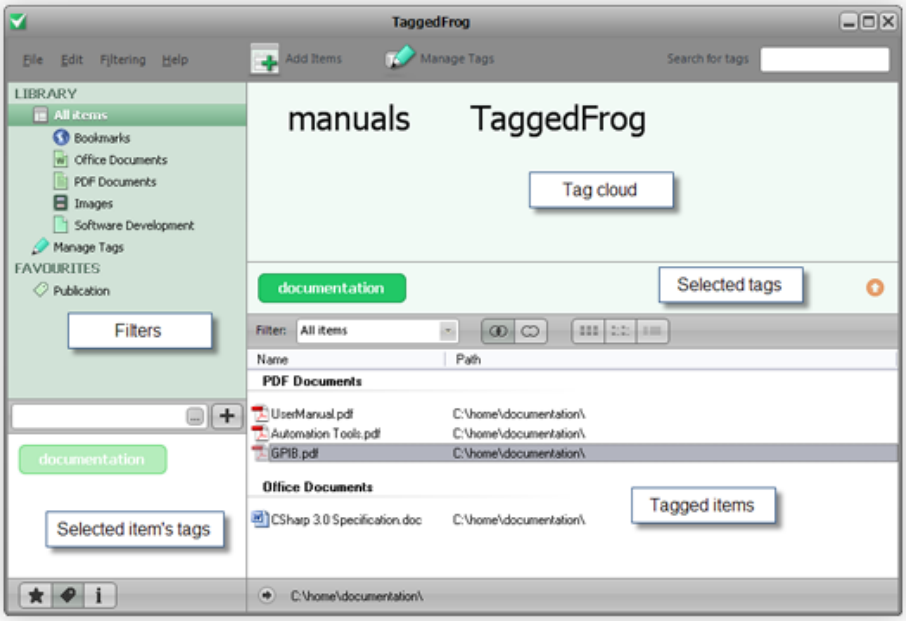

FIGURE 1 – TaggedFrog en utilisation [1]

2.1.4 TagSpaces

TagSpaces [11] est un programme avec une 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 dossier 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 dossier

(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).

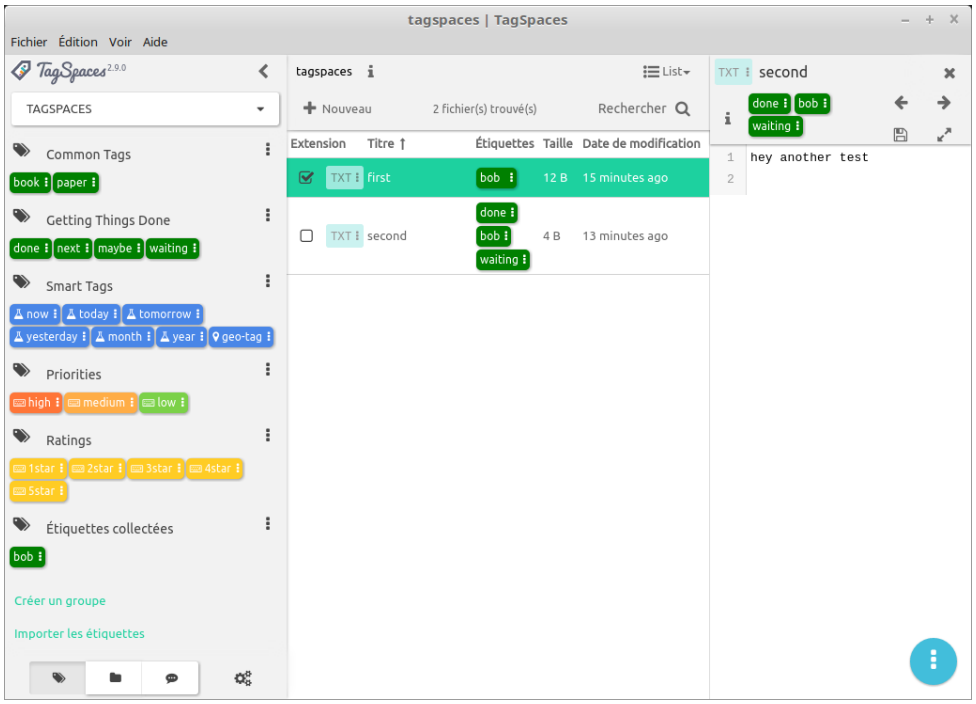

FIGURE 2 – TagSpaces en utilisation

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 [12]. 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 dossier virtuel contenant le résultat d'une recherche sur les noms de fichiers ou d'autres critères [13]. 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 recherche de l'explorateur de fichiers Windows du type `meta:value` [14]. 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.

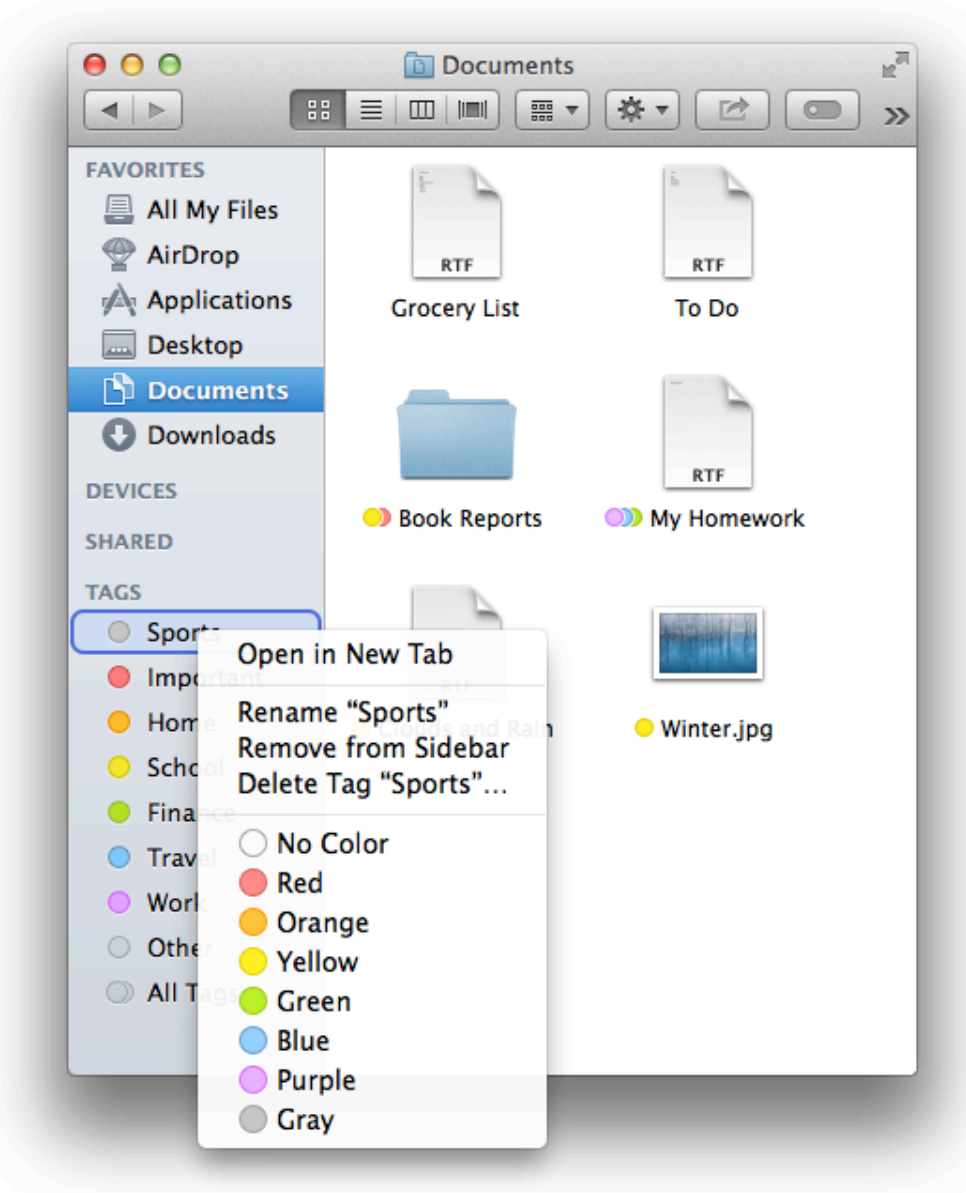

FIGURE 3 – Vue et gestion d'un tag dans le Finder macOS [2]

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 [15]. 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.) [2], [16]. L'implémentation de ce système utilise les extended attributes (voir section 4.2) 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 [5]

Ici, ce 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 dossier (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 [17].

3 Architecture

Le système global 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 XATTR est une fonctionnalité indépendante du reste du système. Comme vu 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. Deux architectures ont été imaginées pour l'indexation des tags et des fichiers.

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 chaine 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 dossiers, sous-dossiers et fichiers. Le dossier à surveiller représente la racine de l'arbre. Les liens entre les noeuds représentent le contenu d'un répertoire. L'étiquette du noeud contient le nom du fichier ou du répertoire et l'ensemble des tags associés au fichier.

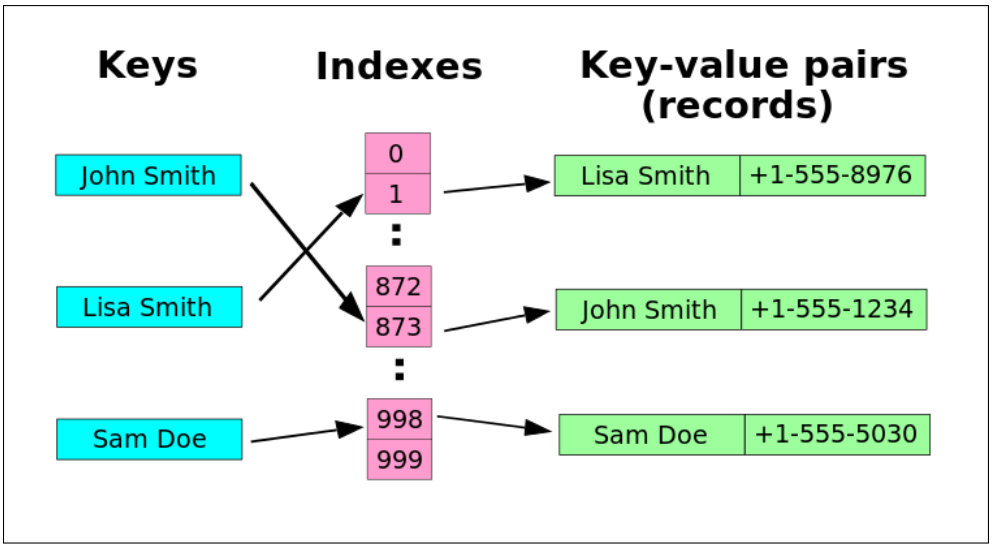

FIGURE 4 – Un annuaire représenté comme une table de hachage - [3]

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 lorsque 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 :

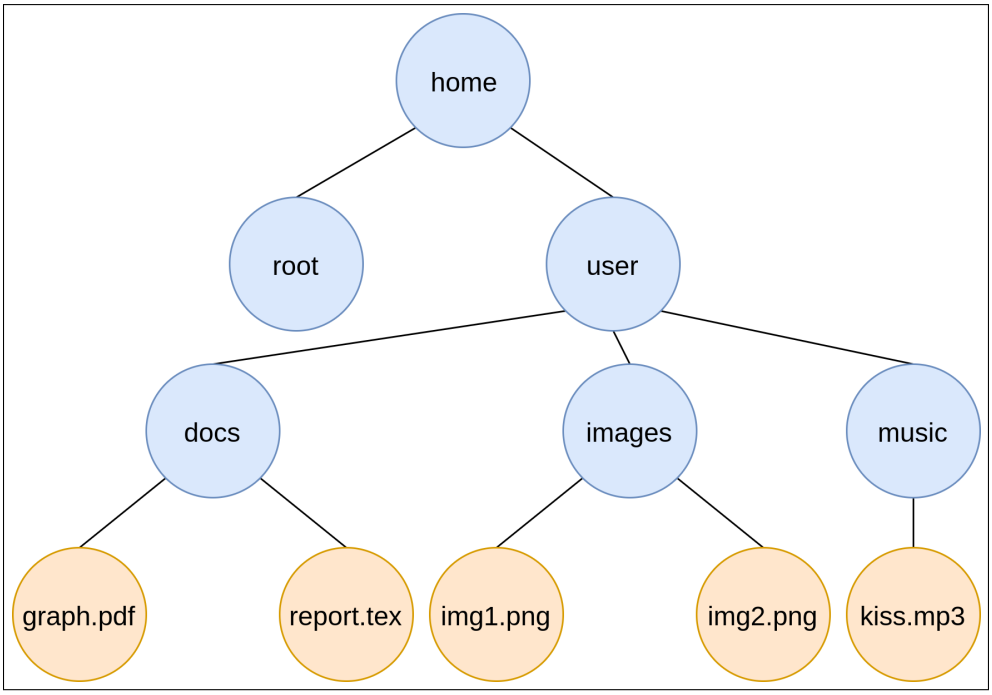

FIGURE 5 – Représentation sous forme d'arbre d'une hiérarchie de fichiers et répertoires

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 définit, 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 1 donne un aperçu des différents cas de figure de l'utilisation du système de fichiers. Pour chaque cas d'utilisation, l'opération correspondante pour les deux structures de données (la table de hachage et l'arbre) est donnée, avec une approximation de la complexité de l'opération en utilisant la notation *Big O*. Les variables suivantes sont définies :

— $c = Operation\ constante.$

— $p = Profondeur\ de\ l'arbre.$

— $t = Nombre\ de\ tags.$

Cas d'utilisation	**Opération *hashmap***	**Opération arbre**
Ajout d'un tag à un fichier/répertoire	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 -> $O(p * c)$
Suppression d'un tag d'un fichier/répertoire	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 -> $O(p * c)$
Ajout d'un fichier	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 -> $O(p * c)$
Ajout d'un répertoire	Opération identique à l'ajout d'un fichier	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) -> $\approx O(p^2 * c)$
Déplacement /renommage d'un fichier	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 -> $O(p * c)$
Déplacement /renommage d'un répertoire	Pour tous les tags de tous les sous-répertoires et fichiers, associer le nouveau chemin de fichier -> $O(t * p * c)$	Parcourir l'arbre à la recherche du parent et changement du lien du noeud avec son parent / simple renommage du nom dans l'étiquette -> $O(p * c)$
Suppression d'un fichier	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 -> $O(p * c)$
Suppression d'un répertoire	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 1 – Cas d'utilisation et opérations, 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 5.2).

3.2.2 Indexation avec un graphe et une table de hachage

Pendant l'implémentation de cette partie du programme (voir section 5.2), 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.

Les explications suivantes sont appuyées par le cours de Jean-François Hêche, professeur à la heig-vd, sur les "Graphes et Réseaux". La définition d'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, [18]).
La définition d'un graphe orienté : "Un graphe 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 arcs du graphe, et telle qu'à chaque arc est associé un couple de sommets de V (c.-à-d. un élément de $V \times V$) appelés les extrémités de l'arc." (Hêche, page 3, [18]).
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, [18]). "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 liées. 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, elle 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, le tableau 2 donne un aperçu des opérations sur les deux structures de données lorsque le FS est manipulé. Les variables suivantes sont définies :

— $c = Operation\ constante.$

— $p = Profondeur\ du\ graphe.$

— $t = Nombre\ de\ tags.$

Cas d'utilisation	**Opération graphe**	**Opération *hashmap***
Ajout d'un tag à un fichier/répertoire	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 -> $O(c)$
Suppression d'un tag d'un fichier/répertoire	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 -> $O(c)$
Ajout d'un fichier	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 -> $O(p*t*c)$	Identique à l'ajout d'un tag à fichier
Ajout d'un répertoire	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 -> $\approx O(p^2*t*c)$	Identique à l'ajout d'un tag à fichier
Déplacement /renommage d'un fichier/répertoire	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
Suppression d'un fichier	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 -> $O(t*c)$
Suppression d'un répertoire	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 à la suppression d'un fichier

TABLE 2 – Cas d'utilisation et opérations, deuxième architecture

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.

3.3 Surveillance du FS

La surveillance du FS et des tags associés est le deuxième pilier du système. L'indexation initiale est nécessaire, 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 inscrire 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 Recherche par tags

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 CLI ou pour créer des applications web. Fort d'une communauté active, de nombreux packages et modules sont disponibles sur Crates.io [19] et de nombreuses discussions sont présentes sur le reddit [20] dédié. Pour plus de détails, l'excellent livre [21] réalisé par les mainteneurs de Rust saura donner de plus amples et précises informations au lecteur avide de connaissances sur Rust. Un autre livre [22], 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 la commande suivante :

```
1 $ curl https://sh.rustup.rs -sSf | sh
```

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* [21]. Le site Are we (I)DE yet ? [23] 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 projet (`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 :

```
[package]
name = "myproject"
version = "0.1.0"
authors = ["Firstname Lastname <me@mail.com>"]

[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 [19]. 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 [21] et il dispose également d'une documentation complète [24] (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 :

```rust
// 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* [21].

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 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, `chat` 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* [21].

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* [21].

Structures de contrôle Comme tout langage de programmation, Rust possède des structures de contrôle pour gérer les consditions 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* [21].

Organisation des fichiers et modules Un programme écrit en Rust a la possibilité d'être découpé en plusieurs fichiers (autres que `main.rs`) 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* [21].

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("Thief"));
    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* [21].

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.

```
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* [21].

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* [21].

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* [21].

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 q'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. L'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)
} // la portée de my_vec se termine ici, my_vec est alors supprimé
```

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

À la ligne 8 du listing 17, la variable `my_vec` est invalidée, mais 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* [21]. Il existe également d'autres types de pointeurs, dits "intelligents". Le livre sur Rust dédie un chapitre entier à eux (chapitre 15 du *book* [21]).

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*.

```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 fonctions, 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* [21].

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* [21].

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 6) : 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.

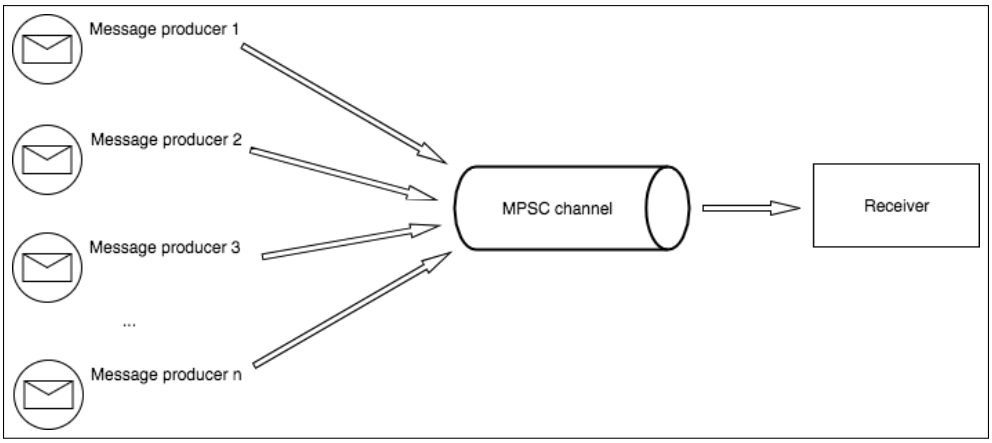

FIGURE 6 – Canal de communication entre threads - [4]

```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* [21].

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* [21].

4.2 Extended attributes

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 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 extended attributes. 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 extended attributes 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 extended attributes 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 le 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 [31].

4.2.1 Petites manipulations

Pour vérifier la portabilité des XATTR, quelques tests ont été réalisés entre un SSD faisant office de disque système à Linux Mint avec 2 clés USB (de 8 et 64 Go) et un emplacement réseau monté en NFS. Le listing suivant 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 les suivants :

```
~ $ 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` [6]. 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 API en C existe et offre les 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 (1), ajouter les fichiers et répertoires pour la surveillance avec le masque des événements voulus (2) 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 suivante :

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

Listing 31 – Structure `inotify_event` - [6]

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` [32] :

— `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 [6] et un très bon article en deux parties sur les ajouts de `inotify` par rapport à `dnotify` (son prédécesseur) [33] et sur ses limitations par Michael Kerrisk [34].

4.4 Sockets

5 Réalisation

5.1 Tag Manager

La première réalisation de ce projet est un outil en ligne de commande, écrit en Rust, permettant de facilement lister, ajouter et supprimer des tags à des fichiers et dossiers. Il fait usage de deux *crates* disponibles sur crates.io : clap [35] et xattr [36]. Clap (Command Line Argument Parser for Rust) est une librairie pour parser les arguments d'un programme en ligne de commande. Xattr est une API en Rust pour récupérer, lister, ajouter/modifier et supprimer des XATTR avec Rust. Elle utilise les SYSCALL en C fournis par Linux.

5.2 Tag Engine

[37]

6 Protocole de test

6.1 *Benchmarks*

6.2 Mesure de performances

7 Discussion/résultats

8 Conclusion

9 Références

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

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

[3] 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.

[4] Marcin Baraniecki. Multithreading in rust with mpsc (multi-producer, single consumer) channels. https://blog.softwaremill.com/multithreading-in-rust-with-mpsc-multi-producer-single-consumer-channels-db0fc91a novembre 2017. Consulté le 18.06.2018.

[5] 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.

[6] inotify - monitoring filesystem events. http://man7.org/linux/man-pages/man7/inotify.7.html, septembre 2017. Consulté le 13.06.2018.

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

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

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

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

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

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

[20] r/rust. https://www.reddit.com/r/rust/. Consulté le 05.05.2018.

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

[22] 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.

[23] Manuel Hoffmann. Are we (i)de yet ? https://areweideyet.com/. Consulté le 25.04.2018.

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

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

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

[34] Michael Kerrisk. Filesystem notification, part 2 : A deeper investigation of inotify. https://lwn.net/Articles/605128/, juillet 2014. Consulté le 13.06.2018.

[35] Kevin Knapp. clap. https://crates.io/crates/clap, mars 2018. Consulté le 14.05.2018.

[36] Steven Allen. xattr. https://crates.io/crates/xattr, juillet 2017. Consulté le 14.05.2018.

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