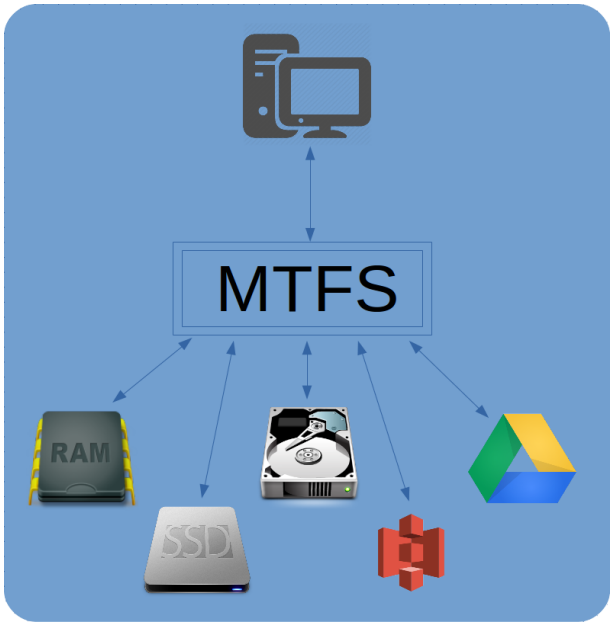
h e p i a
Haute école du paysage, d'ingénierie
et d'architecture de Genève

Hes·SO GENÈVE
Haute Ecole Spécialisée
de Suisse occidentale

Multi Tier Filesystem



Thèse de Bachelor présentée par

Monsieur David WITTWER

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

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

Septembre 2017

Professeur HES responsable TB
Florent GLÜCK

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

printemps 2017
Session de bachelor

INGÉNIERIE DES TECHNOLOGIES DE L'INFORMATION

ORIENTATION – LOGICIELS ET SYSTÈMES COMPLEXES

INFRASTRUCTURE DE STOCKAGE MULTI-TIERS

Descriptif :

Avec l'augmentation massive de données produites, l'accès aux données et le stockage de celles-ci devient une problématique critique. Certaines données nécessitent être accédées avec une latence minimum alors que d'autres types de données, typiquement des données rarement accédées, peuvent être stockées sur des supports à latence élevée. Une infrastructure de type multi-tiers organise les données en *pools* de stockage aux propriétés spécifiques, notamment en termes de capacité (plus ou moins élevées), d'accès (latence, débit). Typiquement, un pool à très haute capacité de stockage sera caractérisé par une latence élevée, un faible débit (typiquement *tapes*) et un faible coût par GB. Au contraire, un disque SSD sera caractérisé par une latence faible, un débit élevé et un coût important par GB.
Plusieurs constructeurs proposent des solutions commerciales d'infrastructures de stockage multi-tiers « tout en un » comprenant matériel et logiciel. Des solutions open-source existent, telles qu'Alluxio (http://www.alluxio.org) mais celle-ci implique une modification des applications pour en profiter (typiquement nécessitant l'écriture d'un back-end spécifique).
Le but de se travail est de concevoir et réaliser une infrastructure de stockage multi-tiers gérant de manière transparente la migration des données vers le pool le plus approprié. La migration des données suivra une politique de migration configurable selon les cas d'accès (temps, utilisateur, groupe, application, etc.).

Travail demandé :

- Analyse de l'existant
- Spécification de l'architecture ; celle-ci doit répondre au minimum au besoins ci-dessous :
  - Gestion de *pools* de stockage aux propriétés et caractéristiques propres : création, destruction, modification, ajout de ressources, etc.
  - Les différents *pools* sont typiquement caractérisés par leur capacité, latence et débit
    - Par exemple, on pourra considérer un *pool «* haute performance » basé sur des disques SSD, un *pool «* standard » basé sur des disques mécaniques et un *pool «* archivage » basé sur bandes. Les données utilisées fréquemment migreront automatiquement vers le pool « haute performance » alors que les données rarement accédées migreront vers le pool « archivage ».
  - La politique de migration des données doit être configurable selon les scénarios d'accès, par exemple : fréquence d'accès, utilisateur, groupe, application, etc.
  - Déterminer si la gestion des données est à réaliser au niveau bloc ou fichier (i.e. module kernel ou FUSE)
    - Justification de la gestion des données choisie ; selon l'approche choisie, certains aspects seront plus ou moins réalisables (exemple : migration des données vers le Cloud, etc.)
- Conception et implémentation de l'architecture
- Démonstrateur

Candidat :

M. Wittwer David

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 2017
Session de bachelor

Résumé :

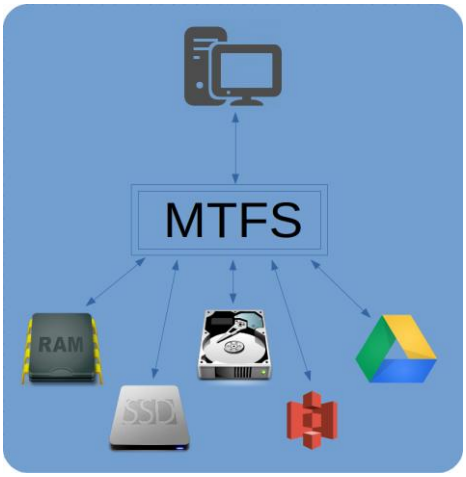
Avec l'augmentation massive de données produites, l'accès aux données et le stockage de celles-ci devient une problématique critique. Certaines données nécessitent d'être accédées avec une latence minimum alors que d'autres types de données, typiquement des données rarement accédées, peuvent être stockées sur des supports à latence élevée.

L'utilisation du SSD et HDD ensemble sur une même machine est devenu assez commun mais l'ajout de bande magnétique ou *Cloud* n'est pas parfait, le premier est relativement lent d'accès et bien que le deuxième soit de plus en plus répandu, dans la plupart des cas les fichiers sur le *Cloud* sont juste une copie des fichiers locaux ou doivent être téléchargés pour être lu.

Plus le nombre de support est grand plus il devient compliqué d'organiser les fichiers pour les répartir en fonction de leurs caractéristiques (fréquence d'accès, taille, etc.). De plus, il faut se rappeler ou les fichiers ont été placés pour ne pas perdre du temps à les chercher. Pour finir dans le cas de fichiers volumineux où seulement une partie est accédée régulièrement, c'est tout le fichier qui doit se situer sur le même espace de stockage.

Dans le but de résoudre ces problèmes, la solution proposée est un système de fichiers, capable de gérer plusieurs espaces de stockage et de migrer automatiquement les données d'un espace à l'autre en fonction de règles spécifiées lors de sa création. Les fichiers ne sont plus obligatoirement sur un seul espace de stockage mais peuvent être répartis en plusieurs morceaux sur l'ensemble des espaces de stockage ce qui permet dans le cas du *Cloud* de pouvoir accéder au fichier sans que la totalité ne soit nécessairement téléchargée. Ce système est appelé MTFS (*Multi Tier Filesystem*).

Les services de stockage utilisés par MTFS sont gérés avec des *plugins*. L'interface nécessaire à leur implémentation étant fournie, tout développeur tiers peut intégrer un nouvel espace de stockage au système.



Deux *plugins* ont été réalisés pour les besoins de ce projet. Le premier utilise les espaces de stockage comme le SSD ou le HDD, l'autre utilise l'espace de stockage *Cloud* Amazon S3. Ce dernier utilise le SDK fourni par Amazon.

L'intégralité du projet est sous licence GNU GPLv3 et est disponible sur GitHub à l'adresse suivante :
https://github.com/Dawen18/mtfs

Candidat :

M. WITTWER DAVID
Filière d'études : ITI

Professeur(s) responsable(s) :

Glück 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

Table des matières

Table des figures

Listings

Acronymes

— **MTFS** *Multi Tier File System*
— **FS** *File System*
— **FUSE** *Filsystem in UserSpacE*
— **AWS** *Amazon Web Services*
— **S3** *Simple Storage Service*
— **API** *Application Programming Interface*
— **SDK** *Software Development Kit*
— **QoS** *Quality of service*

Préface

Présentation

Le présent mémoire synthétise l'étude menée dans le cadre de l'obtention du Bachelor en ingénierie des technologies de l'information, spécialisation logiciels et systèmes complexes d'hepia, suivit en cours du soir.

Tout d'abord, ce document explique la problématique du stockage des données. Ensuite il présente l'état actuel des systèmes équivalents à celui proposé. Dans un troisième chapitre, une introduction sur le fonctionnement d'un système de fichiers et la façon dont il est géré sur Linux est effectuée. Le quatrième chapitre explique le fonctionnement du *framework* FUSE. Le cinquième chapitre spécifie le fonctionnement théorique du système de fichiers conçu. Il est suivi par l'architecture logicielle choisie pour implémenter ce système de fichiers puis par l'implémentation des deux *plugins* de base. Le chapitre suivant présente la méthodologie de test et la validation du système ainsi que les mesures de performances. Puis vient le chapitre sur les améliorations possibles et enfin, le dernier chapitre dresse un bilan du travail accompli.

Conventions typographiques

Ce document suit des règles typographiques ainsi présentées afin de faciliter la lecture :

— l'*italique* est employé pour les termes anglais.

— une police à `chasse fixe` est utilisée pour les commandes ou sortie standard de commande.

— le ***gras italique*** est utilisé pour les termes dédiés à MTFS

Remerciements

Je tiens à remercier ma compagne pour son soutient quotidien sans failles, ses encouragements et sa compréhension. Je remercie également ma sœur pour la relecture intégrale de ce document ainsi que pour tous ces précieux conseils. Je remercie aussi tous mes professeurs qui m'ont beaucoup appris, ce qui m'a permis de réaliser ce travail et particulièrement M. Glück pour ces précieux conseille durant la réalisation de ce travail.

Chapitre 1

Introduction

Actuellement, il existe de nombreux moyens de stocker ses données : Disque SSD ou HDD, bande magnétique, *cloud*, etc., ayant chacun à la fois des avantages mais également des inconvénients. On peut toutefois faire une constatation : plus l'espace de stockage est grand, plus le temps d'accès aux données est long.

L'utilisation du SSD et HDD ensemble sur une même machine est devenu assez commun mais l'ajout de bande magnétique ou *cloud* n'est pas parfait, le premier est relativement lent d'accès et bien que le deuxième soit de plus en plus répandu, dans la plupart des cas les fichiers sur le *cloud* sont juste une copie des fichiers locaux ou doivent être téléchargés pour être lu.

L'utilisation de ces différents espaces de stockage ensemble sur une même machine devient assez contraignant pour plusieurs raisons.

Premièrement plus le nombre d'espaces de stockage est grand plus il devient compliqué d'organiser les fichiers pour les répartir en fonction de leurs caractéristiques (fréquence d'accès, taille, etc). De plus il faut se rappeler ou les fichiers on été placés pour ne pas perdre du temps à les chercher.

Pour finir dans le cas de fichiers volumineux où seulement une partie est accédée régulièrement, c'est tout le fichier qui doit se situer sur le même espace de stockage.

Dans le but de résoudre ces problèmes, la solution proposée est un système de fichiers, capable de gérer plusieurs espaces de stockage et de migrer automatiquement les données d'un espace à l'autre en fonction de règles spécifiées lors de sa création. Les fichiers ne sont plus obligatoirement sur un seul espace de stockage mais peuvent être répartis en plusieurs morceaux sur l'ensemble des espaces de stockage ce qui permet dans le cas du *cloud* de pouvoir accéder au fichier sans que la totalité ne soit nécessairement téléchargée. De plus L'utilisateur n'a pas connaissance de l'existence de ces multiples espaces de stockage et de leur temps d'accès et capacités différents, il ne voit qu'un seul et même espace de stockage. Pour finir, un espace de stockage peut en tout temps être ajouté ou supprimé du système.

Ce système de fichiers est appelé *Multi Tier File System* (MTFS) et il se base sur le *framework* FUSE. La gestion des espaces de stockage type disque dur (HDD, SDD) ainsi que du cloud (Amazon S3) est effectuée nativement. De plus, une interface est fournie pour y implémenter n'importe quel autre espace de stockage supplémentaire.

Chapitre 2

État de l'art

Au moment de la rédaction de ce mémoire (mi 2017) plusieurs systèmes plus ou moins semblables ont été répertoriés.

Parmi ces systèmes, nous trouvons des systèmes open source comme fscop[3] ou btier[4]. Le premier n'a pas été mis à jour depuis 2009 et que le second est en version beta, non actualisé depuis août 2015. Les deux systèmes peuvent utiliser uniquement les espaces de stockage qui peuvent être montés sur Linux, sont des modules kernel et effectuent la migration par fichiers complets par opposition à la migration au niveau de blocs du système de fichiers. Pour finir dans les projets open source, il y a Alluxio[5]. Alluxio est un projet sous licence Apache 2.0, c'est un serveur de fichier en Java capable de fonctionner avec plusieurs *cloud* existants. La migration des données est effectuée par fichier (ou objet) et est uniquement en fonction de la fréquence d'accès. Pour accéder à un FS (*filesystem*) Alluxio il faut passer par le client fournit ou utiliser une des API (*Application Programming Interface*) fournie. La Figure 2.1 illustre l'architecture de Alluxio avec les interfaces pour les clients et les adaptateurs pour les espaces de stockage.

Google a également déposé un brevet [6] pour un système qu'ils ont appelé *Multi-tiered filesystem*. Toutefois, ce système est un serveur de fichiers et nécessite que chaque application qui veut utiliser ce système doive utiliser un protocole précis. La gestion des fichiers est découpée en deux parties, les méta données et les données utilisateurs, puis le système est constitué de plusieurs niveaux de stockage. Contrairement au système développé pendant ce travail, ce système n'est pas adapté à un ordinateur privé et ne parle pas de la possibilité de gérer de nouveaux espaces de stockage.

Il existe aussi des systèmes propriétaires ayant été mis en place par plusieurs fabricants de systèmes informatiques tels que Quantum[7] et IBM. Quantum a développé un système appelé StorNext[1], ce système est un serveur de fichiers constitué d'un rack principal qui est indispensable et d'un ou plusieurs rack optionnels. La Figure 2.2 illustre ce système. Aucune précision n'a été trouvée quant à la configuration de la migration. Le système développé par IBM (Spectrum Scale) est semblable à StorNext dans le sens où c'est un serveur de fichier et que l'on peut ajouter des racks pour étendre la taille de l'espace de stockage. Pour la migration des données, aucune précision n'a été trouvée quand a la granularité mais il est précisé que le

FIGURE 2.1 – Architecture Alluxio

but est uniquement de diminuer au maximum le temps d'accès.

Finalement, des articles ont également été écrits à ce propos. Le premier article[8] présente un nouveau design pour un système de fichiers distribué qui connaît des supports de stockage hétérogènes (par exemple, mémoire, SSD, HDDs, NAS) avec différentes capacités et caractéristiques de performance. Le design proposé est de nouveau client-serveur et la granularité est aussi sur l'intégralité du fichier. Le deuxième article[9] aborde les notions de QoS[10] (*Quality of service*) dans un système *multi-tier* et propose des pistes pour leur amélioration mais toujours dans une optique client-serveur.

En comparaison à ces systèmes, MTFS n'est pas du tout selon l'architecture client-serveur mais peut être monté aussi bien sur un *desktop* que sur un serveur. De plus MTFS effectue une migration par fragments de fichier ce qui répartir mieux les données et peut gérer plusieurs types de règles de migration. Pour finir la gestion des espaces de stockage est externalisée ce qui permet d'ajouter un nouveau volume à chaud même dans le cas où un nouvel espace de stockage vient d'être créé. Par exemple admettons qu'Infomaniak décide de faire un *cloud* comme Amazon, il suffit qu'un développeur (ou Infomaniak dans notre exemple) décide d'implémenter le *plugin* pour MTFS fonctionnant avec leur espace de stockage, d'installer ce *plugin* et le système actuel de MTFS peut sans problème être agrandi pour utiliser ce nouveau *cloud*.

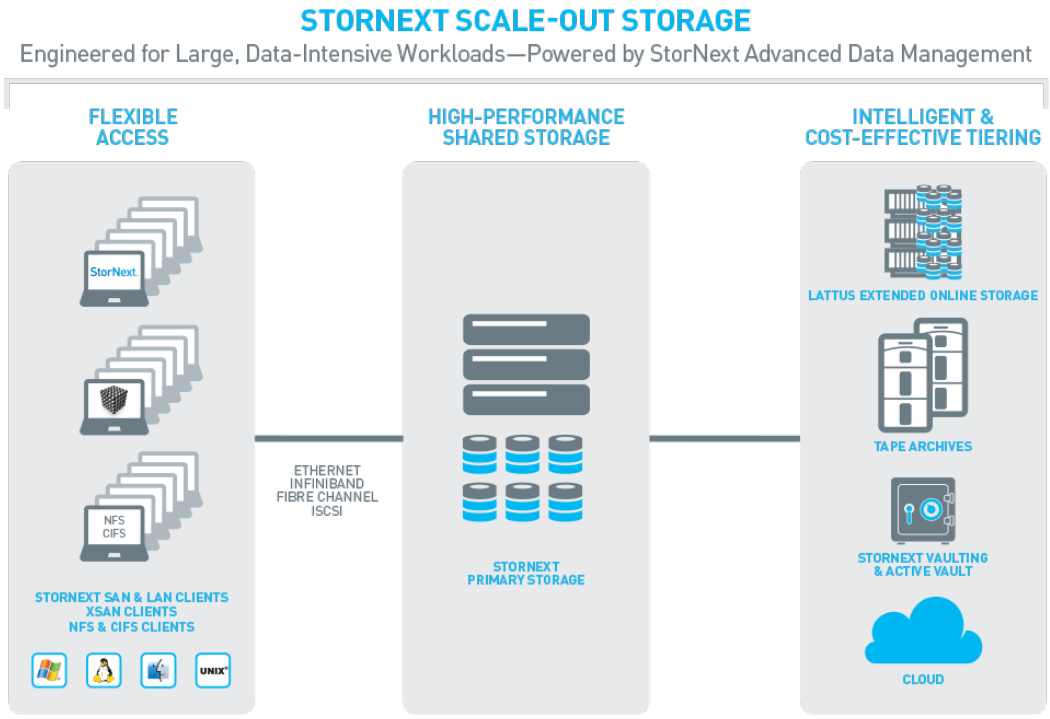


FIGURE 2.2 – Gestion des données de StorNext[1]

Chapitre 3

Qu'est qu'un système de fichiers

3.1 Introduction

> "Un système de fichiers (abrégé « FS » pour *File System*, parfois *filesystem* en anglais) ou système de gestion de fichiers (SGF) est une façon de stocker les informations et de les organiser dans des fichiers sur ce que l'on appelle, en génie logiciel, des mémoires secondaires (pour le matériel informatique, il s'agit de mémoire de masse comme un disque dur, un disque SSD, un CD-ROM, une clé USB, une disquette, etc.). Une telle gestion des fichiers permet de traiter, de conserver des quantités importantes de données ainsi que de les partager entre plusieurs programmes informatiques. Il offre à l'utilisateur une vue abstraite sur ses données et permet de les localiser à partir d'un chemin d'accès." (source : wikipedia[11])

Pour bien comprendre comment fonctionne un FS il faut savoir comment sont structurés les fichiers. Un fichier est constitué de métas données qui permettent de l'identifier et de connaître ses propriétés comme sa taille, sa date de création ou de dernier accès, son propriétaire ou encore son type et ses droits d'accès [1]. Il contient aussi bien entendu les données brutes, celle que l'utilisateur voit. Sur Linux presque tout est fichier. Un dossier n'est en réalité rien d'autre qu'un fichier particulier, au lieu d'avoir les données brutes il y a ce que l'on appelle des entrées qui représentent le contenu du dossier. Les métas données sont stockés dans ce que l'on appelle un *inode* et le contenu dans un bloc.

Étant donné qu'il existe beaucoup de FS différents, les développeurs de noyau Linux ont développé un FS virtuel (VFS) qui permet de faire abstraction du vrai FS. La Figure 3.1 illustre l'architecture du noyau Linux au niveau des systèmes de fichiers. Notre FS étant prévu pour fonctionner sur Linux, les sections suivantes vont présenter rapidement le fonctionnement de VFS.

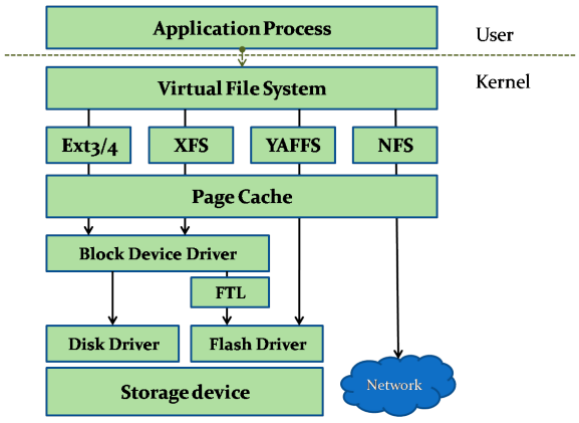


FIGURE 3.1 – Architecture du noyau Linux côté système de fichiers

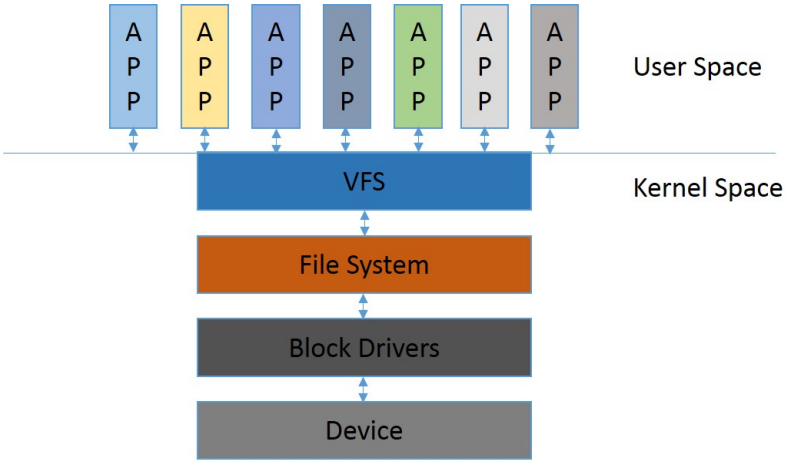


FIGURE 3.2 – Architecture de VFS

3.2 Le VFS

Pour que Linux puisse fonctionner avec plusieurs FS, il faut ajouter une couche d'abstraction interne qui permet d'accéder à n'importe quel système de fichiers du moment que du code *kernel* a été écrit pour gérer ce FS. VFS est situé entre les applications et ce code *kernel* comme illustré dans la Figure 3.2.

Ce système de fichiers fonctionne avec des *inode* et des blocs. Dans les répertoires, chaque fichier est représenté par une *directory entry*. Cette manière d'organiser les données, utilisée dans plusieurs FS, est celui choisi pour MTFS.

3.2.1 *Inode*

L'*inode* est une structure de données qui permet de stocker les informations d'un fichier à l'exception de son nom, comme les droits d'accès ainsi que sa date de création. C'est dans l'*inode* que l'on spécifie quels sont les blocs qui contiennent les données du fichier.

1. lecture, écriture, exécution

3.2.2 *Directory entry*

Chaque entrée (fichier, dossier, lien, etc) dans un dossier est représentée par une *directory entry*. Cette structure de données contient le nom de l'entrée ainsi que le numéro de son *inode*. Les différentes entrées d'un dossier sont stockées dans un bloc comme les données brutes d'un fichier.

3.2.3 *Data Block*

Ce bloc contient les données du fichier ou les *directory entry*. Il fait une taille fixe déterminée par l'espace de stockage et le système de fichiers.

Chapitre 4

Présentation de FUSE

4.1 Introduction

Une des premières contraintes, est que l'espace de stockage de MTFS (Multi Tier File System) soit vu par l'utilisateur comme un seul disque dans lequel il peut organiser ses fichiers comme il le souhaite. Pour ce faire, il faut que le système d'exploitation intègre ce système comme n'importe quel *File System* (FS). Dans le cas de Linux, il existe deux solutions : Soit développer du code *kernel*, soit utiliser FUSE (code utilisateur).

Le code *kernel*, même s'il est plus rapide à l'exécution[1], à comme limite que si le code est buggé, c'est tout le système d'exploitation qui plante ou qui peut être instable, donc les applications utilisateurs aussi. Le code utilisateur lui ne peut en aucun cas faire planter le *kernel*. De plus le code *kernel* ne peut pas utiliser les librairies utilisateur mais uniquement les librairies *kernel* qui sont beaucoup plus limitées. Pour finir Le code *kernel* doit impérativement être écrit en C pur et pas en C++.

FUSE permet d'implémenter un système de fichiers, en espace utilisateur, ce qui permet donc un accès à toutes les librairies utilisateur. Cette implémentation peut être dans n'importe quel langage (C, C++,python,etc) du moment qu'une interface FUSE a été développé pour le langage en question. Dans le cas où le programme se bloquerait, cela n'impacterait en rien le *kernel*, l'application serait juste tuée (terminée) par le *kernel*.

C'est pour ces raisons que FUSE a été choisi pour implémenter MTFS. FUSE met à disposition deux API (*Application Programming Interface*) différentes, la version haut niveau qui est synchrone et travaille avec le nom de la ressource demandée, puis la version bas niveau qui est asynchrone et avec laquelle ce sont les numéros d'*inode* qui sont utilisés à la place des noms. La version bas niveau a été choisie et ce, principalement car elle est asynchrone mais aussi car elle est conseillée par les développeurs de FUSE pour les applications qui on besoin de plus de rapidité.

Les sous-sections suivantes expliquent plus précisément le fonctionnement de FUSE.

1. Pas de changement de contexte dû aux appels système

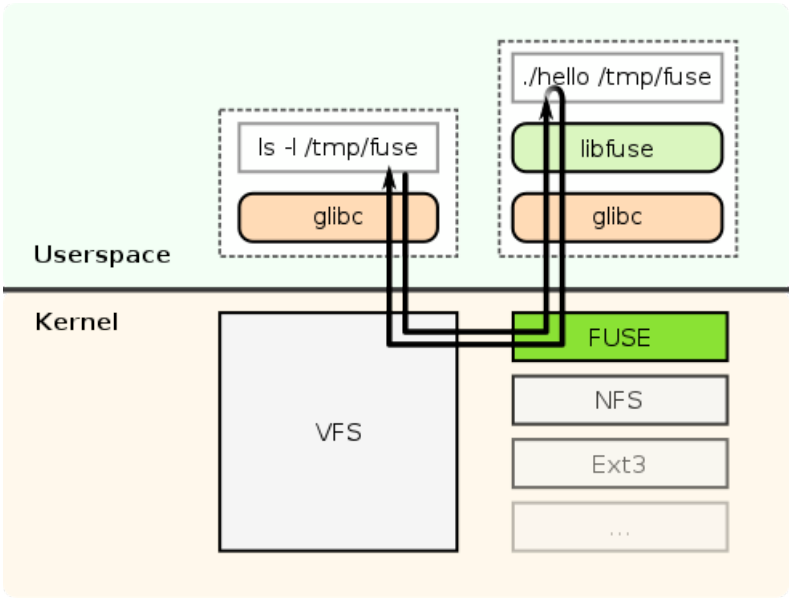


FIGURE 4.1 – Schéma de fonctionnement de FUSE
(source : Wikipedia)

4.2 Architecture

Comme l'illustre la figure Figure 4.1, un FS FUSE s'exécute dans l'espace utilisateur et de ce fait, apporte plusieurs avantages :

— n'importe quel utilisateur peut monter un FS FUSE ;
— un blocage ne perturbe en rien le *kernel* ;
— grand confort pour le développement dû à l'accès aux librairies et au choix du langage non limité à C.

4.3 Principe de fonctionnement

Un FS FUSE est constitué de deux parties :

1. Les *handlers* : il s'agit d'une structure C qui contient des pointeurs vers des fonctions. Ces fonctions seront appelées par le *framework* lors d'un accès au FS et décrivent le fonctionnement de ce dernier. Par exemple lors de l'ouverture d'un fichier déclenche le *handler* `open` ;

2. Le programme principal : il s'agit du programme démarré par l'utilisateur pour monter le FS. Ce programme effectue les opérations nécessaires pour préparer le FS, puis crée une session FUSE qui sera exécutée grâce à `fuse_session_loop`. L'exécution de fuse est détachée du terminal ayant exécuté le programme.

```
FUSE library version: 3.0.0
unique: 1, opcode: INIT (26), nodeid: 0, insize: 56, pid: 0
```

FIGURE 4.2 – Exemple de sortie avec FUSE en mode debug

4.4 Handlers

Les handlers sont les opérations appelées par FUSE lorsque l'utilisateur utilise le FS. Pour l'API bas niveau, toutes ces opérations sont asynchrones et il faut donc appeler des fonctions de FUSE pour transmettre les réponses. De plus un *thread* est créé pour chaque requête de l'utilisateur, donc si une requête est longue à s'exécuter cela n'impactera pas les autres utilisateurs.

FUSE peut-être exécuté en mode *debug*, ce mode ne détache pas le processus du terminal et affiche les *handlers* appelés ainsi que leur réponse. La Figure 4.2 est un exemple de ce que donne fuse en mode *debug*. La première ligne est juste la version de la librairie FUSE. La deuxième ligne est constituée de 5 blocs. Le premier est l'identifiant unique de la requête, le deuxième est le *handler appelé*, le troisième est le numéro d'*inode* transmit au *handler*, le deux derniers ne sont pas utiles pour l'implémentation du FS mais correspondent à la taille du *buffer* interna à FUSE et au pid de la tâche actuelle.

FUSE est livré avec plusieurs exemples de FS dont *passthrought_ll.c*. Lorsqu'il est monté, le point de montage réagit comme si c'était un alias de la racine mais en lecture seule. Utilisé avec le mode *debug*, il permet d'étudier les *handlers* utilisés.

Les sous-sections suivantes détaillent les *handlers* utilisés pour diverses commandes communes. Le point de montage est `/tmp/pass` et toutes les commandes exécutées le sont depuis ce répertoire.

4.4.1 Commande `ls`

`ls` permet de lister le contenu d'un dossier.

Cette commande est la plus complète et la Figure 4.3 l'illustre.

```
unique: 11, opcode: GETATTR (3), nodeid: 1, insize: 56, pid: 12313
   unique: 11, success, outsize: 120
unique: 12, opcode: OPENDIR (27), nodeid: 1, insize: 48, pid: 12313
   unique: 12, success, outsize: 32
unique: 13, opcode: READDIRPLUS (44), nodeid: 1, insize: 80, pid: 12313
   unique: 13, success, outsize: 176
unique: 14, opcode: READDIR (28), nodeid: 1, insize: 80, pid: 12313
   unique: 14, success, outsize: 16
unique: 15, opcode: RELEASEDIR (29), nodeid: 1, insize: 64, pid: 0
   unique: 15, success, outsize: 16
```

FIGURE 4.3 – sortie FUSE pour la commande `ls`

— Le premier *handler* appelé est `getattr`. Ce h*andler* permet de récupérer les attributs (droits d'accès, propriétaire, etc.) d'un *inode*, soit dans notre exemple, le numéro un (c'est l'*inode* racine de FUSE) ;

— *opendir* : Ce *handler* est appelé pour préparer la lecture du dossier. Il permet au FS d'effectuer des opérations en vue de la lecture du dossier ;
— *readdirplus* : *Handler* principal de cette commande, il est appelé, et dispose en ses paramètres de la taille maximale du *buffer* de réponse. Cet appel est donc réalisé plusieurs fois si toutes les *directory entry* prennent plus de place que le *buffer*. Un dernier appel est fait (ici via READDIR) où il faut répondre par un *buffer* vide afin de bien indiquer que la lecture est terminée ;
— *releasedir* Pour finir ce *handler* est appelé pour dire au FS que la lecture est terminée.

4.4.2 Commande `cd`

`cd` permet de changer de dossier de travail. Les messages de debug sont visibles en Figure 4.4
Commande exécutée : `cd home/`

— Le premier *handler* appelé est le même que pour `ls` ;
— *lookup* sur le dossier home pour vérifier que le dossier existe ;
— *access* sur le dossier home pour demander si l'utilisateur dispose du droit d'accès.

```
unique: 5, opcode: GETATTR (3), nodeid: 1, insize: 56, pid: 14266
   unique: 5, success, outsize: 120
unique: 6, opcode: ACCESS (34), nodeid: 1, insize: 48, pid: 14266
   unique: 6, success, outsize: 16
```

FIGURE 4.4 – Messages de débug de FUSE pour la commande `cd`

4.4.3 Commande `mkdir`

`mkdir` permet de créer un dossier.

Cette commande a été exécutée avec l'implémentation de MTFS car `passthrought_ll` est un FS en lecture seule.

```
unique: 41, opcode: LOOKUP (1), nodeid: 1, insize: 44, pid: 14751
   unique: 41, error: -2 (No such file or directory), outsize: 16
unique: 42, opcode: MKDIR (9), nodeid: 1, insize: 52, pid: 14751
   unique: 42, success, outsize: 144
unique: 43, opcode: LOOKUP (1), nodeid: 1, insize: 44, pid: 2913
   unique: 43, success, outsize: 144
```

FIGURE 4.5 – Sortie de FUSE pour la commande `mkdir`

La Figure 4.5 illustre les *handlers* appelés. Le premier *handler* est `LOOKUP`, il permet de vérifier que le dossier n'existe pas déjà. Puis vient `MKDIR` qui crée le dossier et enfin de nouveau `LOOKUP` pour vérifier que le dossier a bien été crée.

4.4.4 Commande `touch`

`touch` permet de créer un fichier s'il n'existe pas ou de mettre à jour sa datte d'accès s'il existe.

```
unique: 24, opcode: LOOKUP (1), nodeid: 1, insize: 45, pid: 13302
   unique: 24, error: -2 (No such file or directory), outsize: 16
unique: 25, opcode: MKNOD (8), nodeid: 1, insize: 61, pid: 13302
   unique: 25, success, outsize: 144
unique: 26, opcode: OPEN (14), nodeid: 140736951632880, insize: 48, pid: 13302
   unique: 26, success, outsize: 32
unique: 30, opcode: SETATTR (4), nodeid: 140736951632880, insize: 128, pid: 13302
   unique: 30, success, outsize: 120
```

FIGURE 4.6 – Sortie de FUSE pour la commande `touch`

Comme illustré sur la Figure 4.6 cette commande commence aussi par `LOOKUP`. Vient ensuite `MKNOD` qui crée le fichier puis `OPEN` et `SETATTR` qui permettent respectivement d'ouvrir le fichier et de mettre à jour les attributs de celui-ci dans le cas du `touch`, la date d'accès. Si `LOOKUP` retourne les infos d'un fichier `MKNOD` n'est pas appelé car le fichier existe.

4.4.5 Commande `rm`

`rm` supprime un fichier ou un dossier [2]

```
unique: 37, opcode: LOOKUP (1), nodeid: 1, insize: 45, pid: 14681
   unique: 36, error: -38 (Function not implemented), outsize: 16
   unique: 37, success, outsize: 144
unique: 38, opcode: UNLINK (10), nodeid: 1, insize: 45, pid: 14681
   unique: 38, success, outsize: 16
```

FIGURE 4.7 – Sortie de FUSE pour la commande `rm`

Pour finir, cette commande illustrée par la Figure 4.7 vérifie si le fichier existe via `LOOKUP` et le supprime avec `UNLINK`. Dans le cas de la suppression d'un dossier le *handler* `RMDIR` est appelé à la place de `UNLINK`.

4.4.6 *handlers* principaux

On peut tirer un bilan de ces différentes commandes et définir des *handlers* qui sont plus prioritaires que d'autres. L'API bas niveau de FUSE définit 42 *handlers*, parmi ceux-ci le choix a été fait d'implémenter uniquement ceux nécessaires au commandes précédentes et à la lecture-écriture des fichiers. Ce qui donne 17 *handlers* à implémenter [3]

- `access`
- `lookup`
- `getattr`
- `setattr`
- `open/opendir`
- `readdir/readdirplus`
- `release/releasedir`
- `mknod/mkdir`
- `read`
- `write`
- `rmdir`
- `unlink`

2. avec l'option `-r`
3. les liens ont été mis de côté dans un premier temps

4.5 Implémentation d'un *handler*

Les *handlers* sont exécutés de manière asynchrone, le retour de ce dernier est donc systématiquement `void`. Par ailleurs, pour que FUSE associe les réponses aux requêtes, un paramètre de type `fuse_req_t` est transmis au *handler* et doit être retransmis lors de la réponse. Le Listing 4.1 est l'implémentation du *handler* `OPEN` dans le programme d'exemple `passthrought_ll.c`

```
1  static void lo_open(fuse_req_t req, fuse_ino_t ino,
2            struct fuse_file_info *fi) {
3    int fd;
4    char buf[64];
5
6    sprintf(buf, "/proc/self/fd/%i", lo_fd(req, ino));
7    fd = open(buf, fi->flags & ~O_NOFOLLOW);
8    if (fd == -1) return (void) fuse_reply_err(req, errno);
9
10   fi->fh = fd;
11   fuse_reply_open(req, fi);
12 }
```

Listing 4.1 – Implémentation du *handler* `OPEN` de `passthrought_ll.c`

Chapitre 5

Spécification de MTFS

5.1 Introduction

Dans les sections suivantes, les principaux mécanismes de MTFS (Multi Tier File System) sont abordés. Des recommandations quant à l'implémentation sont également abordées, mais la façon dont les données sont stockées reste libre à l'implémentation.

5.2 Terminologie

Les termes suivants sont utilisés pour parler de MTFS

1. ***Volume*** : Ce terme désigne un espace de stockage géré par MTFS. Par exemple un disque dur ou un stockage Dropbox.
2. ***Pool*** : Ce terme désigne une agrégation de ***volumes*** possédant une règle de migration commune.
3. ***Plugin*** : Il s'agit du système qui va gérer la lecture et écriture des données sur un système de stockage (HDD, SSD, S3, Dropbox, FTP, etc.).

À noter que les identifiants des ***pools*** et ***volumes*** sont placés "entre guillemets".

5.3 Contraintes

5.3.1 Transparence pour l'utilisateur

La transparence pour l'utilisateur est la contrainte principale qui est à l'origine du *File System* (FS). Cette propriété implique que l'utilisateur n'ait pas à se soucier de la façon dont sont stockées les données, à partir du moment où cela respecte l'arborescence qu'il utilise.

5.3.2 Volumes de stockage hétérogènes

Étant donné qu'il y a plusieurs types de stockage avec des spécificités (performance, taille et méthode d'accès) différentes, il faut que l'utilisateur voie MTFS comme un seul et unique espace de stockage. L'utilisateur ne doit pas avoir à se soucier de l'endroit où sont stockées ces données. Étant donné qu'il peut y avoir des ***volumes*** sur le *cloud* l'utilisateur doit quand même pouvoir accéder à ces données de la même manière que si elles étaient stockées localement.

5.3.3 Migration des données

C'est la deuxième contrainte à l'origine du FS. Il faut que les données soient réparties sur les ***volumes*** en fonction des règles établies lors de la construction du système, ainsi qu'elles soient automatiquement déplacées si les règles du ***volume*** ne sont plus satisfaites (changement de propriétaire, accès plus fréquent, etc.). Cela doit également s'effectuer de manière totalement transparente pour l'utilisateur, et sans bloquer les données.

5.3.4 Redondance

La redondance est importante pour assurer l'intégrité des données en cas de perte définitive ou temporaire d'un ***volume***, afin que le système puisse continuer à être opérationnel. Cette redondance doit être répartie sur des ***volumes*** différents tout en respectant les règles définies.

5.3.5 Indépendance du système de fichiers

Chaque ***volume*** est différent dû à son ***plugin*** et à ses propres limitations, mais elles ne doivent pas limiter MTFS. Par exemple un ***volume*** qui utilise un HDD formaté en MinixFS qui a une taille maximale de nom de fichier de 14 caractères ne doit pas imposer cette limite à MTFS. Il en va de même pour la taille maximum des fichiers.

5.3.6 Droits d'accès

Étant donné que MTFS est destiné à être utilisé par plusieurs utilisateurs différents, il est nécessaire que les droits d'accès soient gérés. Pour des questions de simplicité et de compatibilité, suivent les mêmes règles que celles utilisées ceux sur Unix/Linux.

5.3.7 Asynchronicité

MTFS ne doit pas empêcher un utilisateur d'accéder à des données parce qu'il n'a pas fini de traiter la requête d'un autre utilisateur. L'implémentation doit alors éviter d'être bloquante quand cela est possible grâce à l'utilisation de flots d'exécution concurrents (asynchronicité) à l'aide de *threads*. Dans le cas d'un même utilisateur, il récupère la main uniquement lorsque son opération est terminée. Soit tout est en cache et en attente d'être persisté sur les volumes, soit tout est sur les volumes.

5.3.8 *Scalability* et flexibilité

Le système doit être redimensionnable en tout temps, sans nécessiter son démontage et en limitant le plus possible d'impacter l'utilisateur. Il faut pouvoir ajouter ou supprimer un volume ainsi que modifier les règles de migration en tout temps y compris lorsque le volume est monté.

5.4 Architecture conceptuelle

5.4.1 Principe de fonctionnement

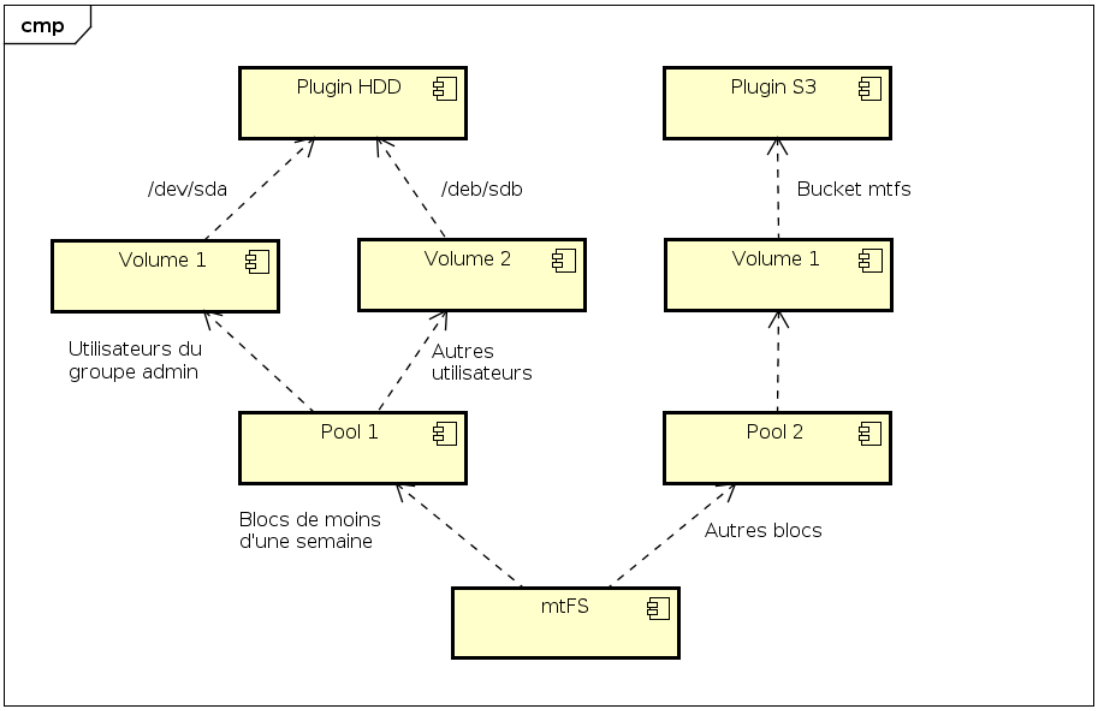


FIGURE 5.1 – Architecture générale de MTFS

Les relations entre les composants principaux de MTFS sont illustré en Figure 5.1. Nous pouvons observer la présence de deux ***volumes*** "1", néanmoins pas dans le même pool : ce ne sont pas les mêmes. Pour le *plugin* HDD, il s'agit de l'utilisation de deux fois le même *plugin* mais pas avec le même disque dur : ce n'est donc pas le même objet.

Dans un système comme celui ci-dessus, lorsque sont créés des fichiers en tant qu'utilisateur appartenant au groupe "admin", tous les blocs et *inodes* seront dirigés dans le ***volume*** "1" du ***pool*** "1", soit sur `/dev/sda`. Si l'utilisateur est différent, ils iront sur `/dev/sdb`. Au bout d'une semaine, tous les blocs et *inodes* n'ayant pas été accédés seront automatiquement déplacés dans le ***volume*** "1" du ***pool*** "2", soit sur Amazon dans le *bucket* mtfs.

5.4.2 Configuration

Les paramètres configurables du système sont :

— La taille des blocs de données
— La taille des différents caches
— Le type de migration
— Les règles de migration
— La redondance des données

La configuration s'effectue via un exécutable fourni avec le système. En aucun cas il ne faut modifier le fichier de configuration car cela serait traité comme une corruption de ce dernier.

5.4.3 Migration des données

Les données doivent automatiquement être déplacées quand elles ne respectent plus les règles établies lors de la création du système et ce de manière totalement transparente pour l'utilisateur.

5.4.4 Répartition des données

Pour MTFS, un fichier est constitué de plusieurs blocs de taille personnalisable lors de la création d'un stockage MTFS. Les blocs ne sont pas nécessairement tous stockés sur le même ***volume***. Dans le cas d'un gros fichier pour lequel uniquement la fin est régulièrement lue, le bloc contenant la fin de fichier sera dans un ***volume*** rapide alors que les autres pourraient être dans un ***volume*** *cloud*. De plus, les *inodes* peuvent aussi être stockés sur un autre ***volume***.

5.4.5 *Pools* et *volumes*

Volume

Un ***volume*** est identifié de manière unique au sein d'un ***pool*** et est associé à un ***plugin***.

Le *plugin* doit implémenter les primitives suivantes pour l'accès aux données :

— `add` ;
— `del` ;
— `get` ;
— `put` ;

Le *plugin* doit également implémenter d'autres primitives pour le fonctionnement interne de MTFS :

— `getSuperblock` ;
— `putSuperblock` ;
— `getName` ;
— `attach` ;
— `detach` ;

La fonction de ces primitives est détaillé au sous-section 6.2.3

Pools

Un ***pool*** contient un ou plusieurs ***volumes***. A chaque ***volume*** est associé une règle pour savoir si les données peuvent être dirigées, ou non, dans ce volume. Chaque ***pool*** est identifié de manière unique au sein du système.

5.4.6 *Meta*fichiers

Les descriptions des données dans les sous-sections suivantes sont au format JSON. Ce format a été choisi plutôt que le format binaire pour faciliter la compréhension, le débogage et car il existe des librairies pour le parser facilement. Cela n'implique en aucun cas qu'elles doivent obligatoirement être stockées de cette manière et une fois le système stable il serait judicieux de passer à un format binaire. Chaque bloc est référencé de manière unique au sein d'un ***volume*** et c'est l'identifiant du pool, du volume et du bloc qui font l'identifiant unique au sein du système. Les différents blocs (*inode, dir block* et *data*) ont chacun leur espace de nommage pour les identifiants, c'est-à-dire qu'il peut tout à fait y avoir un *inode* `"1"` et un *data block* `"1"`.

Superblock

Ce bloc définit le système avec tous les ***pools*** et les ***volumes*** ainsi que leurs identifiants respectifs. Pour chaque ***volume*** il y a aussi la configuration du ***plugin***. Le *superblock* est utilisé pour monter le système. Lors de la création du système il est copié sur tous les volumes, les copies servent de vérification contre la corruption et aussi comme point d'entrée de récupération en cas de faille de la machine hôte. Le Listing 5.1 est le *superblock* utilisé pour les tests du chapitre 8. On peut voir qu'il est constitué d'un seul ***pool*** qui contient deux ***volumes***, le premier est un SSD et le deuxième un *bucket* AWS S3. Tous les niveaux de caches ont été mit à 0 pour que l'accès aus ***volumes*** soit le plus direct possible et la taille des blocs de données et de répertoire est de 512 octets. Pour finir, dans la dernière partie, la clé `rootInodes` contient une liste des copies de l'*inode* racine. Cette liste est primordiale pour effectuer une récupération du système (c.f. sous-section 5.5.5).

```
{
    "inodeCacheSize": 0,
    "directoryCacheSize": 0,
    "blockCacheSize": 0,
    "blockSize": 512,
    "redundancy": 1,
    "migration": 0,
    "pools": {
        "1": {
            "migration": 0,
            "volumes": {
                "1": {
                    "highLimit": 120,
                    "plName": "block",
                    "params": {
                        "devicePath": "/dev/sdb7",
                        "fsType": "ext4"
                    }
                },
                "2": {
                    "lowLimit": 60,
                    "plName": "s3",
                    "params": {
                        "bucket": "mtfs",
                        "region": "eu-central-1"
                    }
                }
            }
        }
    },
    "rootInodes": [
        {
            "poolId": 1,
            "volumeId": 1,
            "id": 0
        },
        {
            "poolId": 1,
            "volumeId": 2,
            "id": 0
        }
    ]
}
```

Listing 5.1 – Fichier *superblock* utilisé pour les tests chapitre 8

Inode

L'*inode* avec l'identifiant "0" est réservé pour l'*inode* racine et ne doit en aucun cas être attribué par un plugin. Un *inode* contient les métadonnées suivantes :

- `mode` : les droits d'accès et le type de fichier ;
- `uid` : l'utilisateur propriétaire du fichier ;
- `gid` : le groupe propriétaire du fichier ;
- `size` : la taille du fichier ;
- `linkCount` : nombre de lien pointant l'*inode* ;
- `atime` : dernière date d'accès ;
- `dataBlocks` : blocs de données du fichier illustré en Figure 5.2. C'est un tableau de tableau, Le premier tableau contient le ième bloc du fichier ou dossier, c'est le bloc recherché lors de la lecture du fichier ou dossier. Le deuxième tableau contient les redondances du ième bloc. Le contenu de ces blocs est identique. Par exemple un fichier d'une taille de 1000 octets avec une taille de bloc de 512 octets et une redondance de 2 contiendra 4 blocs de données. Le data bloc pourrait alors être le suivant :

```
"dataBlocks": [
    [0, 1],
    [2, 3]
]
```

Les blosc 0 et 1 sont identiques et le 2 et 3 le sont aussi.

Dans la plupart des FS avec *inode*, il y a des blocs de données directes, indirects voir même double indirect comme illustré dans la Figure 5.3. Ceci est dù au fait que la taille d'un *inode* est fixe. Dans le cas de MTFS la taille de l'*inode* n'est pas fixe donc aucun besoin de blocs indirects. Cependant dans le cas d'un fichier contenant beaucoup de blocs de données l'*inode* peut devenir considérablement grand, ce qui peut ralentir la lecture du fichier. Pour pallier ce problème l'*inode* est fragmenté. Le premier fragment contient toutes les métas données citées plus haut mais pas l'intégralité des *data blocks*, ce sont les autres fragment qui contiennent la suite du tableau des *data blocks*. Le fragment à accéder peut-être obtenu via une formule expliquée en section 5.5.1. La taille d'un fragment devrait être la même que la taille d'un bloc de données mais le choix de cette taille est laissée libre à l'implémentation.

Un exemple d'*inode* est donnée au Listing 5.2. Dans cet exemple on peut voir qu'il n'y a qu'un seul bloc de données avec aucune redondance. Le mode est en décimal, en octal cela donne 40775, 4 correspond à un dossier ce qui explique le `linkCount` à 2, 0 permet de voir qu'il n'y a pas de droit d'endossement et enfin 775 correspond au droit lire, écrire et exécuter pour l'utilisateur et le groupe propriétaire, et lire et exécuter pour les autres.

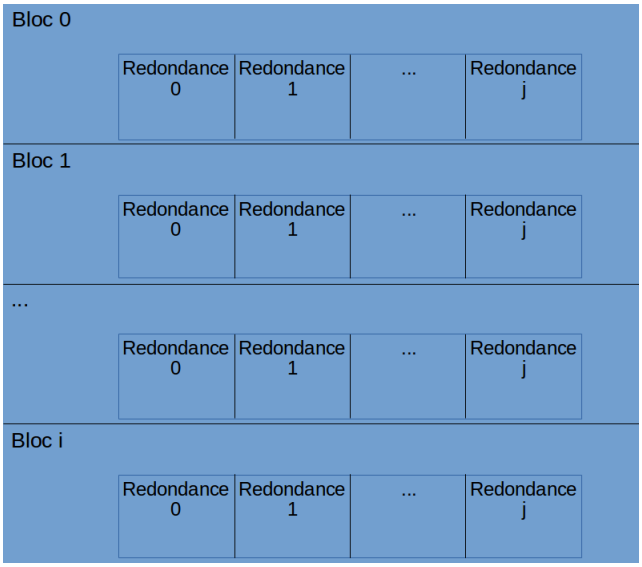


FIGURE 5.2 – illustration du tableau des *data blocks* d'un inode

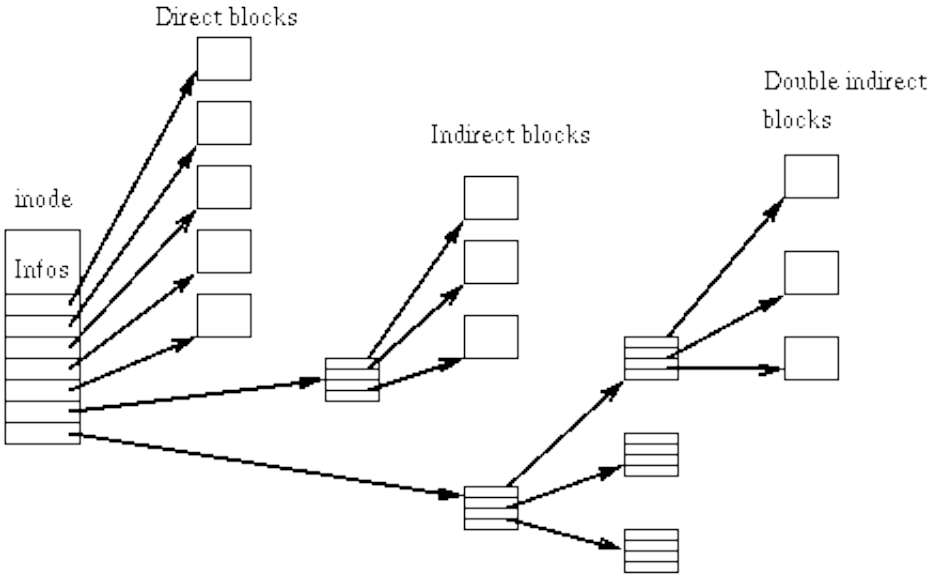


FIGURE 5.3 – Exemple de FS avec indirection de bloc (Source Wikipedia [2])

```
{
    "mode": 16893,
    "uid": 0,
    "gid": 1001,
    "size": 0,
    "linkCount": 2,
    "atime": 1502633936,
    "dataBlocks": [
        [
            {
                "poolId": 1,
                "volumeId": 1,
                "id": 0
            }
        ]
    ]
}
```

Listing 5.2 – Exemple de fichier inode

directory block

Ces blocs sont remplis par les entrées des différents fichiers d'un dossier. La taille de ce bloc est fixe et est la même que celle des blocs de données. Les *directory block* sont en fait des blocs de données avec un format particulier. le Listing 5.3 est un exemple de *directory block*. On peut voir qu'il contient uniquement un fichier appelé foo et que son *inode* est le numéro "2" situé dans le ***pool*** "1", ***volume*** "1".

```
{
    "foo": [
        {
            "poolId": 1,
            "volumeId": 1,
            "id": 2
        }
    ]
}
```

Listing 5.3 – Exemple de fichier *directory block*

data block

Ces blocs sont les données bruts du fichier et font une taille fixe spécifiée lors de la configuration d'un espace de stockage MTFS.

meta block

Le *meta block* est un bloc nécessaire au fonctionnement de MTFS uniquement. Chaque bloc (*inode, directory block* et *data block*) à son *meta block*. Le Listing 5.4 illustre son contenu. `referenceId` correspond aux identifiants des blocs qui référence le bloc auquel est attaché ce *meta block*, cela permet lorsque le bloc est déplacé de changer son pointeur. `lastAccess` est utile pour les migrations en fonction de la fréquence d'accès. Il peut paraître redondant avec celui dans l'*inode* mais celui-ci est pour chaque bloc alors que dans l'*inode*, c'est pour tout le fichier.

```
{
    "referenceId": [
        {
            "poolId": 1,
            "volumeId": 1,
            "id": 0
        },
        {
            "poolId": 1,
            "volumeId": 2,
            "id": 0
        }
    ],
    "lastAccess": 1502633936
}
```

Listing 5.4 – Exemple de fichier meta

5.5 Limitations du système

5.5.1 Taille du système

Chaque type de bloc (*inode*, *directory* ou *data*) est dans un espace de nommage différent. C'est a dire qu'il peut y avoir un *inode* "1" mais aussi un *directory block* "1" et un *data block* "1". L'identifiant de ces blocs fait 64 bits ce qui nous donne un nombre maximal de 2^{64} par type de bloc.

Chaque bloc du FS est de taille "blocksize" ce qui implique qu'un méta fichier doit aussi faire cette taille au maximum ou alors être scindé, "blocksize" étant un paramètre donné lors de la création d'un espace de stockage MTFS. Pour MTFS la taille des *inodes* et des *directory block* n'est pas fixe et ils sont en conséquent scindé dès qu'ils dépassent la taille "blocksize".

Un identifiant de bloc est structuré de la manière suivante :

— identifiant du ***pool*** (32 bits)

— identifiant du ***volume*** (32 bits)

— identifiant bloc (64 bits)

Soit une taille totale de 128 bits.

Les tailles maximales théoriques sont donc les suivantes :

fichier

Un *inode* est composé des éléments suivants :

— Le type (32 bits)
— L'utilisateur propriétaire (32 bits)
— Le groupe propriétaire (32 bits)
— La taille du fichier (64 bits)
— Le nombre de liens sur le fichier (32 bits)
— La date d'accès (64 bits)
— Les blocs de données : c'est un tableau d'identifiant et un identifiant fait 128 bits.

La taille minimale d'un *inode* en considérant aucun bloc de données est de 256 bits. Pour une taille de bloc de 4096 octets, le nombre de blocs de données de l'inode est donné par la formule suivante :

$$\frac{4096 * 8 - 256}{128} = blocs$$

$$\frac{4096 * 8 - 256}{128} = 254$$

Ce qui nous donne 254 blocs soit une taille de fichier maximal de

$$254 * blocksize = 254 * 4096 * 8 = 8323072bits$$

Un fichier de taille maximale de 8 Mo est beaucoup trop petit. Pour résoudre cette limitation les *inodes* sont fragmentés en plusieurs blocs. L'identifiant du fragment est noté sur 64 bits ce qui nous fait 2^{64} fragments par *inode*. Le numéro du fragment à accéder est donné par la formule suivante :

$$fragment = \frac{numeroBloc * 128 + 256}{blocksize}$$

$$= \frac{255 * 128 + 256}{4096 * 8} = \frac{32869}{32768} = 1.0039$$

C'est donc le fragment 1 qu'il faut prendre. Avec ces fragments la taille maximale d'un fichier en nombre de blocs est donné de la façon suivante :

$$nbBloc = nombreDeFragments * identifiantsParFragment - 256$$

$$= 2^{64} * \frac{blocksize}{128} - 256 = 2^{64} * \frac{4096 * 8}{128} - 256 = 2^{64} * 256 - 256 = 4.72 * 10^{21}$$

Le nombre maximal théorique de bloc de données dans un fichier est donc de $4.72 * 10^{21}$ dans le cas où il n'y a pas de redondance. Si une redondance de 2 est configurée cela veut dire que chaque bloc de données existe en deux exemplaires donc on divise par deux le nombre de blocs

utiles dans un *inode*. Sachant que le nombre de blocs dans un même volume est de 2^{64} soit $1.84 * 10^{19}$ il faut impérativement que les données soient réparties sur plusieurs ***volumes***.

dossier

Comme vu précédemment un *inode* peut contenir $4.72 * 10^{21}$ blocs de données. MTFS ne limitant pas la taille d'un nom de fichier il devient compliqué de calculer le nombre maximum d'entrée que peux contenir un dossier. Toutefois dans le fichier `limits.h`[1] du noyau Linux[2] une taille maximale pour les noms de fichiers est fixée à 255 caractères. Cette taille sera prise pour les calculs. Pour les calculs le cas sans redondance est considéré. La taille d'une entrée est donc la suivante :

$$255 * tailleCaractre + tailleIdentifiant = 255 * 8 + 128 = 2168bits$$

En prenant une taille de bloc de 4096 octets, nous obtenons un nombre d'entrées par bloc de :

$$\frac{4096 * 8}{2168} = 15.1$$

Soit 15 entrées par bloc car on ne peut pas fractionner une entrée.

$$15 * 4.72 * 10^{21} = 7.08 * 10^{22}$$

Ce qui nous donne en théorie un total $7.08 * 10^{22}$ entrées possibles dans un dossier.

Volume

La taille des méta données d'un bloc est la suivante :

— `referenceId` : 128 bits

— `lastAccess` : 64 bits

Ce qui nous donne une taille de 192 bits dans le cas d'aucune redondance. ATTENTION le cas des liens physiques est mis de côté car non implémenté pour l'instant mais la taille du champ `referenceId` grandit proportionnellement au nombre de liens sur un *inode*. Sachant que chaque type de bloc a un identifiant sur 64 bits. Sachant aussi que chaque bloc des méta données est d'une taille de 192 bits , pour une taille de bloc toujours de 4096 octets, la taille maximale d'un volume peut être calculé de la manière suivante :

$$blocksize * (3 * 2^{64}) + metasSize * (3 * 2^{64})$$

$$= 4096 * 8 * (3 * 2^{64}) + 192 * (3 * 2^{64}) = 1.82 * 10^{24} bits$$

Un volume plus grand que 1.82 Yo ne pourra pas être intégralement utilisé par un ***volume***

1. `/include/uapi/linux/limits.h`
2. vérifié en version 4.4 à 4.12

MTFS. Cela ne veut pas pour autant dire que les volumes plus petits ne sont pas pris en charge.

MTFS

les calculs ci-dessous sont toujours en considérant une taille de bloc de 4096 octets. Le nombre maximum de ***volumes*** dans un ***pool*** est de 2^{32} et le nombre maximum de pool du système est aussi de 2^{32}. Donc la taille maximale théorique nécessaire à un espace de stockage MTFS est la suivante.

$$2^{32} * 2^{32} * 1.82 * 10^{24} = 3.35 * 10^{43} = 3.35 * 10^{19} Yo$$

Ce chiffre est toutefois à prendre avec précaution car il dépend de la façon dont sont stockées les données. Par exemple l'*inode* Listing 5.2 au format JSON occupe 136 octets d'espace disque soit 1088 bits, ce qui est nettement plus que les 256 bits annoncé ci-dessus. Ceci est dû au format JSON car chaque caractère prend un octet d'espace disque.

La taille utile d'un espace de stockage MTFS est la suivante :

$$2^{32} * 2^{32} * 2^{64} * blocksize = 2^{32} * 2^{32} * 2^{64} * 4096 * 8 = 1.11 * 10^{43} = 1.11 * 10^{19} Yo$$

5.5.2 Accès concurrents

Lorsque plusieurs utilisateurs accèdent au même fichier en même temps en lecture ce n'est pas un problème mais le problème survient lorsque au moins l'un d'eux veut écrire. Il faut que l'écriture soit atomique, c'est-à-dire qu'aucune autre opération n'intervienne tant que l'écriture n'est pas terminée. Pour résoudre ce problème, dans une première version de MTFS les fichiers écrits sont "verrouillés" tant qu'ils ne sont pas intégralement écrits, pour éviter une lecture incohérente.

5.5.3 Montage unique

Étant donnée qu'il y a des volumes locaux dans un espace de stockage MTFS, il n'est pas possible d'utiliser la même configuration sur deux machines différentes, et ce même si les volumes configurés sont uniquement des *cloud* car il n'y a pas de communication entre les clients et lors de l'attribution d'un nouveau bloc il n'est pas demandé au ***plugin*** de vérifier que le bloc soit réellement disponible.

Il y a cependant une configuration possible pour que plusieurs machines accèdent à un espace de stockage MTFS. La première solution est de faire un serveur de fichier qui expose son espace de stockage MTFS. Par exemple un serveur peut monter un système MTSF dans le dossier `/mnt/mtfs` et partager ce dossier via le protocole NFS[12]

5.5.4 Politiques de migrations

Lors de la configuration du système par l'administrateur il est possible qu'un mauvais choix mène un utilisateur à ne plus avoir de place pour stocker ces fichiers.

Il est aussi possible que les politiques de migration mènent toutes les redondances d'un bloc sur le même système de stockage ce qui ne donne plus aucun intérêt à effectuer de la redondance (voir aussi sous-section 5.5.5).

Prenons par exemple la configuration suivante :

- redondance : 2
- ***pool*** 1 : donnée de moins d'une semaine
  - ***volume*** 1 : SSD uniquement pour les utilisateurs du groupe admin
  - ***volume*** 2 : HDD pour tous les utilisateurs
- ***pool*** 2 : plus d'un semaine
  - ***volume*** 1 : S3 uniquement pour les utilisateurs groupe admin
  - ***volume*** 2 : Bandes magnétiques pour tous les utilisateurs

Avec cette configuration-ci, un utilisateur n'appartenant pas au groupe admin, dispose d'un espace de stockage égal au volume de la bande magnétique plus du HDD et rien de plus, car il n'a pas le droit d'accéder aux autres volumes. Une fois le HDD plein MTFS refusera la création de nouveaux fichiers car cela forcerait à les placer sur la bande magnétique ce qui va à l'encontre de la règle des fichiers de plus d'une semaine. Une fois la bande magnétique pleine MTFS ne déplacera plus les blocs du HDD à la bande magnétique, cela enfreindra la règle de moins d'une semaine mais les blocs ne seront pas perdus et une erreur devra être signalée à l'administrateur.

5.5.5 *Recovery*

Il existe une possibilité de perte des données, dans le cas où la configuration ne répartit pas correctement les redondances.

Pour illustrer ces propos reprenons la configuration proposée ci-dessus. Un bloc appartenant au groupe admin de plus d'une semaine sera dans le ***volume*** 1 du ***pool*** 2 et sa redondance sera placée prioritairement sur un ***volume*** autre que S3, donc il ira sur la bande magnétique. Dans ce cas aucun problème si un des deux ***volume*** vient à défaillir, le deuxième est la avec la redondance. Le problème survient dans le cas d'un utilisateur n'appartenant pas au groupe admin. Le bloc ira sur la bande magnétique et sa redondance aussi, car les règles l'interdisent d'aller ailleurs. En aucun cas les règles ne doivent être outrepassées. Un travail de vérification devrait alors être fait pas l'implémentation lors de la création de l'espace de stockage MTFS pour avertir l'utilisateur qu'il y a un risque.

Chapitre 6

Implémentation de MTFS

6.1 Introduction

Ce chapitre présente une implémentation du système de fichiers (FS) MTFS en l'intégrant au mieux avec le *framework* FUSE.

En raison du temps imparti, certaines fonctionnalités de MTFS n'ont pas été implémentés mais la conception globale permet une implémentation future aisée.

L'implémentation a été effectué pour C++14 minimum.

6.2 Architecture logicielle

6.2.1 Fonctionnement

L'implémentation doit mettre en place les différents mécanismes de la spécification de MTFS en répondant aux contraintes suivantes :

— assurer de bonnes performances ;
— permettre l'ajout de ***plugin*** sans modification du système existant ;
— assurer une facilité de configuration et d'utilisation ;

Afin de satisfaire au mieux ces exigences, le développement est découpé en 5 modules :

— *Config* : Module qui permet de configurer ainsi que de monter un FS MTFS ;
— *Plugin system* : Module qui gère les différents *plugin* en les chargeant/libérant et définit leur interface ;
— *Wrapper* : Module permettant l'ajout facile de *handlers* et des retours à FUSE de code d'erreur par défaut ;
— *Core* : Module principal du système, c'est lui qui est utilisé par FUSE pour accéder aux fichiers ;
— *Migrator* : Module qui effectue la migration des blocs.

Outre ces différents modules, dans le cas où il y a plusieurs blocs à lire/écrire sur les volumes, l'opération est réalisée de manière asynchrone pour des questions de performance à l'aide d'un *threadPool*[13].

La Figure 6.1 illustre la communication entre les différents modules de MTFS

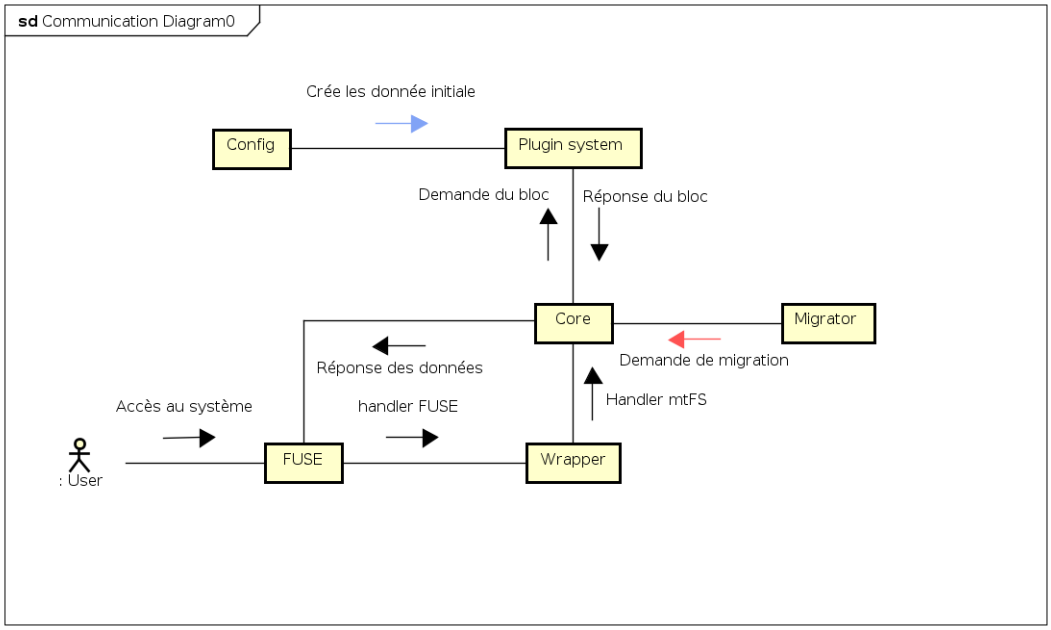


FIGURE 6.1 – Diagramme de communication de mtFS

6.2.2 *Config*

Ce module contient les deux exécutables nécessaires au fonctionnement de MTFS à savoir `MTFSCreate` et `MTFSMount`. Le but de ces exécutables est de faciliter la configuration et le montage du système mais aussi de détecter les erreurs ou incohérences de configuration.

`MTFSCreate`

`MTFSCreate` est l'exécutable qui permet de créer un système de stockage MTFS. Cet exécutable a plusieurs options qui permettent de configurer correctement l'espace de stockage, les principales sont :

- `-n` : Créé un nouvel espace de stockage et prend en paramètre son nom ;
- `--add/--del` : Ajoute où supprime un ***volume*** ou ***pool***. Add retourne l'identifiant du ***pool***/***volume*** créé ;
- `-p/-v` : Spécifie le ***pool***/***volume*** concerné par la modification en prenant en paramètre son identifiant ;
- `-c` : Permet de configurer un ***plugin*** ou une politique de migration à l'aide d'une *string* ou d'un fichier JSON donné en paramètres. Cette option nécessite de spécifier le ***pool***/***volume*** concerné via l'option -p/-v ;
- `--install` : Analyse la configuration pour détecter les incohérences où risque de blocage et l'installe si elle est conforme.

```
usage: ./mtfsMount [options] <mountpoint> <configName>

    -h   --help            print help
    -V   --version         print version
    -d   -o debug          enable debug output (implies -f)
    -f                     foreground operation
    -s                     disable multi-threaded operation
    -o clone_fd            use separate fuse device fd for each thread
                           (may improve performance)
    -o allow_other         allow access by all users
    -o allow_root          allow access by root
    -o auto_unmount        auto unmount on process termination
```

FIGURE 6.2 – Aide de MTFSMount

`MTFSMount`

`MTFSMount` est l'exécutable qui permet le montage d'un FS MTFS. Les options sont celles définies par FUSE et en plus le nom de la configuration à monter. le Figure 6.2 illustre la sortie de l'exécutable avec l'option `-h`. les options recommandées sont `-o allow_other` et `-o auto_umount`. Par exemple pour les tests du chapitre 8 la ligne suivante permet le montage du système

```
./mtfsMount -o allow_other -o allow_root /tmp/mtfs test
```

6.2.3 *Plugin manager*

Étant donné que MTFS ne peut pas implémenter tous les systèmes des stockages qui existent ainsi que ceux à venir, un système de *plugins* a été mis en place. Cela permet à un développeur indépendant ou à un *provider* de *Cloud* de développer l'intégration de leur système dans MTFS.

Les *plugins* nécessaires au fonctionnement d'un espace de stockage MTFS sont chargés au montage de ce dernier ce qui implique que ces *plugin* doivent être compilés comme des librairies dynamiques. Le chargement des *plugins* est géré par l'objet `PluginManager`. La librairie `dlfcn.h` est utilisée pour charger un *plugin* via les fonctions `dlopen` et `dlsym`. Le Listing 6.1 montre comment le *plugin* est chargé ligne 2 et le chargement du symbole `createObj` ligne 10.

La classe abstraite `Plugin` est incluse dans le *package* de MTFS et sert d'interface pour les différents *plugins*. Cette classe contient les méthodes suivantes.

— `attach` : Monte le *plugin* à l'aide des paramètres reçus ;
— `detach` : Démonte le *plugin*. Cela permet de dire au *plugin* qu'il doit finir toutes ces tâches en cours ;
— `add` : Alloue un nouveau bloc ;
— `del` : Supprime un bloc ;
— `get` : Récupère un bloc ;
— `put` : Écrit un bloc ;

— `getSuperblock` : Récupère le super bloc ;
— `putSuperblock` : Écrit le super bloc.

Les fonctions `add, del, get` et `put` prennent en paramètre les type de bloc concerné et, si c'est le méta bloc qui est concerné. La fonction `attach` reçoit en paramètre les données nécessaires au fonctionnement du *plugin* et en plus la taille de bloc utilisé pour le système et le chemin absolu vers un dossier qui peut être utilisé par le *plugin* pour son fonctionnement mais, uniquement pour des données qui n'ont pas besoin d'être persistantes.

Chaque *plugin* doit en plus implémenter trois fonctions qui servent à son intégration à MTFS. Ces trois fonctions sont les suivantes :

— `createObj` : Créé l'objet du *plugin* ;
— `destroyObj` : Détruit l'objet ;
— `getInfo` : Récupère les infos du *plugin* comme son nom et les paramètres nécessaires au montage de ce dernier.
— `getName` : Récupère le nom du *plugin* tel qu'il doit être noté lors de la configuration d'un espace de stockage MTFS.

```
1  //    Open plugin
2      void *library = dlopen(path.c_str(), RTLD_LAZY);
3      if (!library) {
4        cerr << "Cannot load plugin '" << name << "': " << dlerror() << endl;
5        return nullptr;
6      }
7      dlerror();
8
9  //    Load createObj symbol
10     plugin.createObj = (pluginSystem::Plugin *(*)()) dlsym(library, "createObj"
       );
11     char *dlsym_error = dlerror();
12     if (dlsym_error) {
13       cerr << "Cannot load symbol create: " << dlsym_error << endl;
14       return nullptr;
15     }
```

Listing 6.1 – Extrait du **PluginManager** pour charger une librairie

6.2.4 *Wrapper*

Le *wrapper* est le module qui permet aux *handlers* FUSE de correspondre à des fonctions du *Core* de MTFS. Ce module est générique et peut-être utilisé dans n'importe quelle implémentation d'un FS FUSE en C++. Il permet une implémentation future de *handler* aisée et aussi un comportement par défaut pour chaque *handler*[1]. Il est constitué de trois objets : `FuseCallback`, `Fusebase` et `MTFSFuse`.

1. en l'occurrence le retour du code d'erreur `ENOSYS` qui spécifie que la fonction n'est pas implémentée.

`FuseCallback`

Cet objet est celui qui lie les *handlers* FUSE à ses méthodes. Il contient un objet `FuseBase` qui est appelé pour chaque fonction.

`FuseBase`

`FuseBase` est l'objet de base pour toutes les implémentations de FUSE. Il contient une méthode `run` qui définit l'objet `FuseBase` du `FuseCallback` comme lui-même et qui exécute la fonction qui permet à FUSE de démarrer. Une méthode pour chaque *handler* FUSE est définie et retourne à FUSE le code erreur `ENOSYS` (*Function not implemented*).

`MtfsFuse`

Cet objet hérite de `FuseBase` et redéfinit uniquement les fonctions qui sont implémentées par le système de fichiers. C'est le fichier à modifier pour ajouter le traitement de *handlers* par FUSE.

6.2.5 *Core*

C'est le module principal. Il s'occupe d'accéder aux blocs pour les transmettre à FUSE. Ce module est constitué de 5 objets indispensables et peut-être complété par d'autre objets gérant le cache. Pour des questions de temps, l'objet Cache n'a pas été implémenté mais l'interface et l'utilisation par les autres objets sont implémentés. Les sous sections suivantes détaillent les 5 objets indispensables.

`Mtfs`

L'objet `Mtfs` effectue les opérations demandées par FUSE et y répond. C'est dans cet objet qu'est la logique de MTFS. Si une requête impacte plusieurs blocs, ceux-ci sont accédés de manière concurrente à l'aide d'un *thread pool*[13] basé sur la librairie Boost[14].

`PoolManager`

Il contient les différents ***pools*** et permet de répartir les données sur ces derniers en fonction des règles établies. Son rôle est aussi de verrouiller les blocs qui sont en cours de modification pour éviter qu'ils ne soient lus ou écrits, ce qui provoquerait un état inconsistant des données. Pour finir il maintient à jour une table des déplacements utilisée lorsqu'un bloc est déplacé lors de la migration.

`Pool`

La répartition de données au sein du ***pool*** est effectuée par cet objet. Il contient uniquement une liste des ***volumes*** et ainsi que leurs règles respectives.

`Volume`

C'est l'objet qui contient le *plugin*. Il tient à jour le *timestamp* du dernier accès de chaque bloc.

`Rule`

L'objet `Rule` est une classe abstraite pour les règles de migration. Cet objet permet d'implémenter n'importe quel type de règle. L'implémentation de base fournit deux règles, une qui permet de définir les utilisateurs et groupes qui sont autorisés ou bloqués et la deuxième définit la fréquence d'accès des blocs d'un volume. Par exemple tous les blocs accédé il y a moins de 2 jours sont dans un SSD puis jusqu'à une semaine sur un HDD et enfin ceux de plus d'un mois sur AWS S3. Dans cette première implémentation les règles ne peuvent pas être combinées dans un même volume. La règle des temps d'accès est la mieux intégrée à MTFS.

6.2.6 *Migrator*

Ce module est un *Daemon* interne à MTFS. Dans notre cas c'est un *thread* qui fonctionne en arrière-plan de MTFS. Il se réveille à intervalles réguliers pour effectuer la migration des données selon les règles prédéfinies. Si aucune migration n'a eu lieu au bout d'un certain nombre de réveils, le temps entre chaque réveil augmente, de même à chaque fois qu'il y a des migrations [2], le temps diminue jusqu'à une certaine limite (2 minutes minimums et 20 minutes maximum).

6.3 Interaction entre les objets

Le diagramme de séquence de la Figure 6.3 illustre les fonctions utilisées lors de la commande `ls`. Pour une question de lisibilité, le diagramme n'affiche pas les fonctions utilisées entre le `PoolManager` et le `Plugin`, ces dernières servent uniquement à récupérer le bloc dans le bon pool puis dans le bon volume. Sur ce diagramme on peut voir les *handlers* appelé successivement [3]. Le *handler* `opendir` déclenche l'obtention d'un nombre de bloc qui est définit en tant constante de la classe `Mtfs`. Suite à ça le *handler* `readdirplus` récupère les blocs suivants puis les parcourt tous pour construire la liste des fichiers du dossier. Pour finir, le *handler* `releasedir` libère la mémoire allouée dans `opendir`. L'obtention des blocs se fait en parallèle via un *thread pool*[13].

2. Un nombre de bloc est définit comme seuil pour diminuer le temps d'attente entre chaque réveil
3. le `readdir` présent en sous-section 4.4.1 a été enlevé car il se déroule de la même façon que `readdirplus`

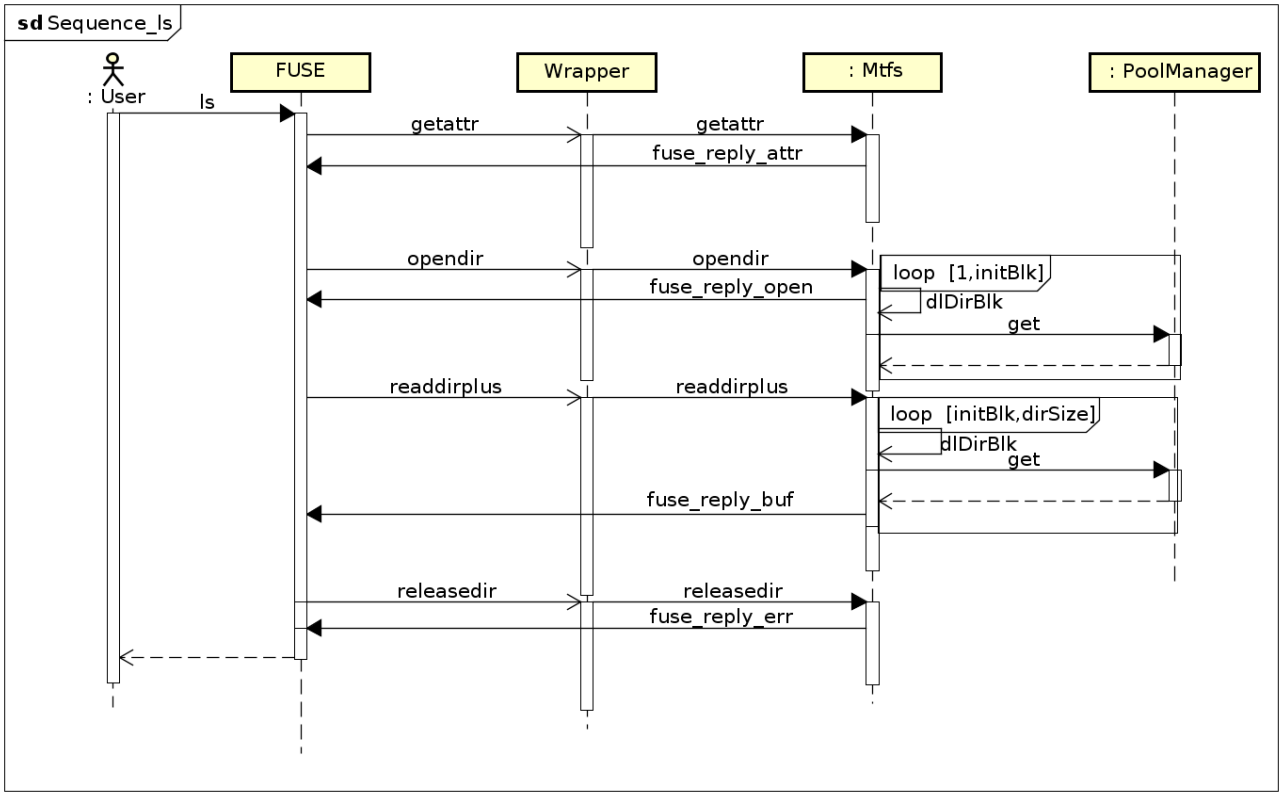


FIGURE 6.3 – Diagramme de séquence pour la commande `ls`

La Figure 6.4 quant à elle démontre le déroulement de la commande `mkdir`. Dans le *handler* `mkdir`, on remarque que l'inode est ajouté en premier puis ensuite le *directory block*, le `add` du *directory block* a lieu uniquement s'il n'y a pas encore de *directory block* dans le répertoire ou si le dernier est plein. Tous les blocs sont ajoutés et écrit autant de fois qu'il y a de redondance. On peut aussi observer que le premier `lookup` répond via `fuse_reply_err` avec le code ENOSYS [4] alors que le deuxième répond via fuse_reply_entry. Ceci est dû au fait que dans le premier cas le dossier n'existait pas encore alors que dans le deuxième cas il existe.

4. *No such file or directory*

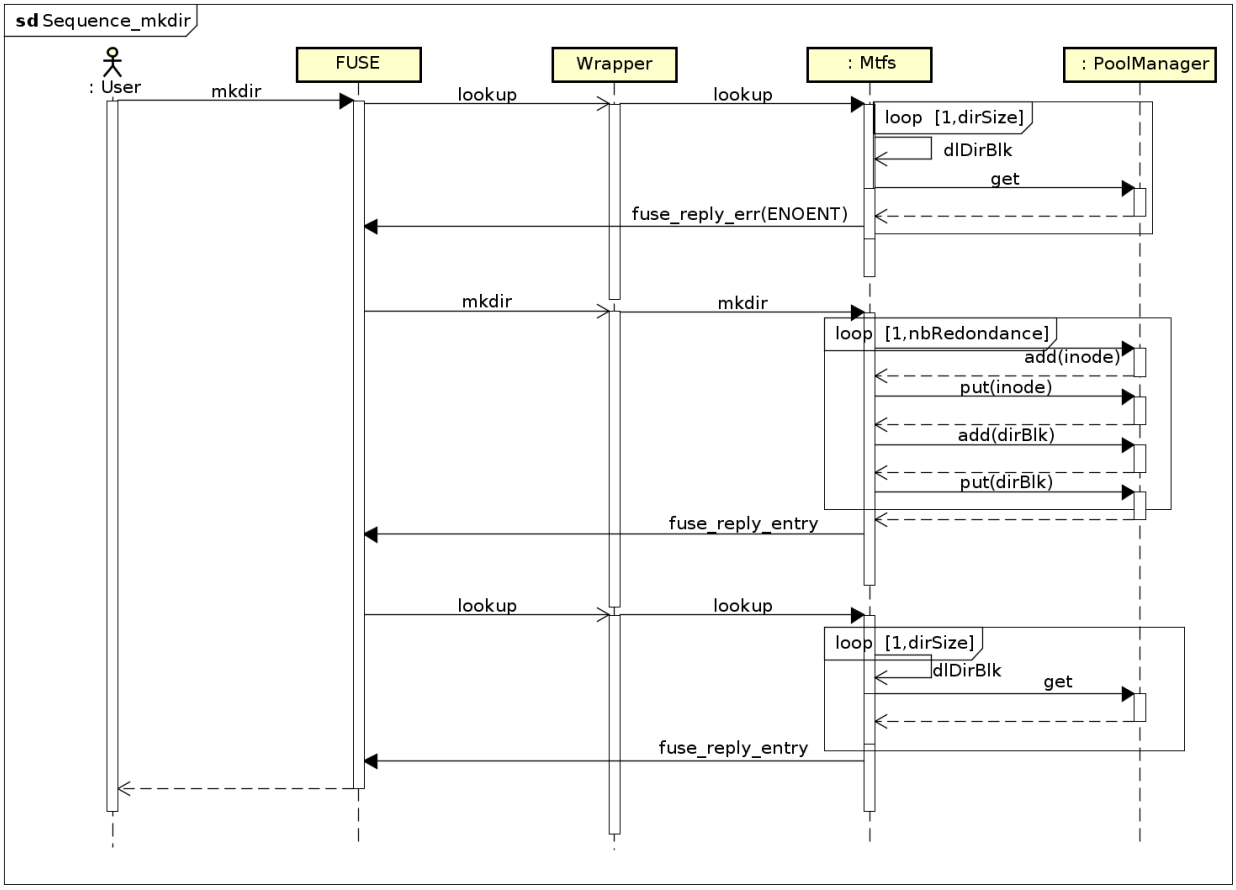


FIGURE 6.4 – Diagramme de séquence pour la commande `mkdir`

Chapitre 7

Implémentation des *plugins*

7.1 Introduction

Le système de base de MTFS (*Multi Tier Filesystem*) est conçu avec deux *plugins*. Le premier permet d'utiliser les espaces de stockage comme les SSD ou HDD ou tout autre système qui peur être monté sur Linux avec la commande `mount`. Le deuxième *plugin* permet d'intégrer un *bucket* d'Amazon S3 (*Simple Storage System*).

7.2 *Plugin* block

Comme dit précédemment ce *plugin* peut utiliser tous les espaces de stockage qui peuvent être montés sur Linux. Étant donné que la commande `mount` est utilisée dans ce *plugin* cela impose à MTFS d'être exécuté avec les droits `root`. Aucune librairie particulière n'est nécessaire au fonctionnement de ce *plugin*. Dans un premier temps et pour des questions de facilité de débogage, les différents blocs sont écrits au format JSON, excepté le *data block* qui lui est écrit au format binaire.

Comme illustré dans le Listing 5.1, pour son fonctionnement le *plugin* a besoin des informations suivantes :

— `devicePath` : C'est le chemin d'accès du fichier spécial de l'espace de stockage à utiliser. Par exemple `/dev/sda1` ;
— `fsType` : C'est le FS de l'espace de stockage. Il doit impérativement être pris en charge par Linux. Par exemple Ext4 ;

L'espace de stockage pour ce *plugin* est monté dans le *home* des *plugin* de MTFS dans un dossier `BlockDevice` plus le nom du fichier spécial. Par exemple `/dev/sda1` sera monté dans `${PLUGIN_HOME}/BlockDevice/sda1`.

7.3 *Plugin* s3

Ce *plugin* permet l'intégration d'un *bucket* Amazon S3 dans MTFS. Il est basé sur le SDK (*Software Development Kit*) C++ fourni par Amazon[15]. Pour le fonctionnement de ce *plugin* il faut donc installer la libraire `aws-cpp-sdk-s3` comme décrit dans l'annexe . Pour fonctionner, le *plugin* a besoin de connaître les clés d'accès de AWS. Ces clés doivent être inscrites dans le fichier suivant `/.aws/credentials`, et le contenu du fichier est celui illustré dans le Listing 7.1. Outre ces informations le *plugin* a besoin des données suivantes illustrées dans le Listing 5.1 :

— `region` : C'est la région AWS dans laquelle est située le *bucket*. Par exemple`eu-central-1`.

— `bucket` : C'est le nom du *bucket* é utiliser. Par exemple `mtfs` ;

Le principe de fonctionnement est le suivant. Pour l'*upload*, création du fichier représentant les données en local et *upload* sur S3. Pour le *download*, récupération du fichier sur S3 et extraction des données du fichier pour transmettre à MTFS.

```
1 [default]
2 aws_access_key_id = your_access_key_id
3 aws_secret_access_key = your_secret_access_key
```

Listing 7.1 – Fichier `credentials` pour le SDK AWS

Chapitre 8

Tests et performance

Tous les tests ont été effectués sur la machine suivante : Asus X750JB avec Intel Core I7-4700 HQ 2.5Ghz avec 12 GB de RAM. Cette machine possède 4 cœurs physiques et 8 logiques. L' espace de stockage utilisée est un SSD Samsung 850Pro. Pour correctement comparer les temps d'accès, la taille des blocs logiques du SSD a été récupérée à l'aide de la commande **fdisk -l /dev/sdb**. La taille de bloc est de 512 octets.

Ce qui nous donne la configuration suivante :

— taille des blocs : 512 ;
— pool 1 :
  — volume 1 : SSD Samsung 850Pro pour les blocs de moins de deux minutes ;
  — volume 2 : Cloud Amazon S3 à Francfort pour les autres blocs.

Les tests suivants ont été effectués et sont détaillés dans les sections suivantes :

— Test de validation des *plugins* à l'aide du *framework* googletest[16]
— Test de validation de la migration des données via un script.
— Test de vitesse de lecture/écriture via programme C++

Pour effectuer les tests, MTFS est compilé dans un mode prévu à cet effet qui écrit dans un fichier tous les blocs attribués ainsi que les déplacements effectués.

8.1 Validation des *plugins*

La validation des *plugins* est effectuée via un programme en C++ et le *framework* googletest. Ce *framework* permet de définir des préconditions à chaque test et aussi définit plusieurs types de tests via des macros. De plus, ce *framework* effectue un affichage coloré des différents tests ce qui permet de voir très rapidement ceux qui ne sont pas passés.

8.2 Validation de la migration

Pour valider la migration un script (c.f. Annexe 3.6.1) python est exécuté. Il effectue les actions suivantes :

```
unique: 33, opcode: LOOKUP (1), nodeid: 1, insize: 45, pid: 14724
   unique: 32, error: -38 (Function not implemented), outsize: 16
MtFS [DEBUG] LOOKUP: test
MtFS [DEBUG] POOL_MANAGER: get dir block p:1 v:1 i:0
MtFS [DEBUG] POOL_MANAGER: get inode p:1 v:1 i:1
   unique: 33, success, outsize: 144
unique: 34, opcode: OPEN (14), nodeid: 140589983079056, insize: 48, pid: 14724
   unique: 34, success, outsize: 32
unique: 35, opcode: GETATTR (3), nodeid: 140589983079056, insize: 56, pid: 14724
   unique: 35, success, outsize: 120
unique: 36, opcode: READ (15), nodeid: 140589983079056, insize: 80, pid: 14724
MtFS [DEBUG] POOL_MANAGER: get dat block p:1 v:1 i:0
   unique: 36, success, outsize: 37
```

FIGURE 8.1 – Contenu les logs avant la migration lors de la lecture d'un fichier

```
unique: 41, opcode: LOOKUP (1), nodeid: 1, insize: 45, pid: 14961
MtFS [DEBUG] LOOKUP: test
MtFS [DEBUG] POOL_MANAGER: get dir block p:1 v:2 i:0
MtFS [DEBUG] POOL_MANAGER: get inode p:1 v:2 i:1
   unique: 41, success, outsize: 144
unique: 43, opcode: OPEN (14), nodeid: 140589983081712, insize: 48, pid: 14961
   unique: 43, success, outsize: 32
unique: 44, opcode: GETATTR (3), nodeid: 140589983081712, insize: 56, pid: 14961
   unique: 44, success, outsize: 120
unique: 45, opcode: READ (15), nodeid: 140589983081712, insize: 80, pid: 14961
MtFS [DEBUG] POOL_MANAGER: get dat block p:1 v:2 i:0
   unique: 45, success, outsize: 37
```

FIGURE 8.2 – Contenu les logs après la migration lors de la lecture d'un fichier

— Écriture d'un fichier sur MTFS et récupération dans les logs des numéros de blocs.
— Attend que la migration soit effectuée (via un minuteur).
— Lit le fichier sur MTFS et vérifie dans les logs que le volume soit bien celui d'AWS.
— Attend de nouveau la migration.
— Effectue une dernière lecture pour vérifier dans les logs que le fichier soit bien sûr le SSD.

On peut constater en Figure 8.1 linge 4, 5 et avant-dernière ligne, que l'identifiant du volume est le "1" ce qui correspond au SSD. Dans la Figure 8.2 les identifiants des volumes ont changé et sont ceux su volume S3. Le retour au SSD n'est pas affiché mais le résultat est semblable à celui en **??**. Ces informations permettent de valider la migration des données.

8.3 Tests de vitesse

Lors de ces tests un bug a été rencontré lors de l'écriture des fichiers de plus de 256 Ko(c.f. sous-section 9.1.2). Toutefois, des valeur ont quand même pu être prises pour de plus petites tailles. La Figure 8.3 illustre les mesures prises. Le programme de test ouvre le fichier avec les *flags* `O_DIRECT` et `O_SYNC`. Ces *flags* permettent de minimiser l'effet du cache et s'assure que les opérations sur le système de fichiers sont réalisées de manière synchrone. Le but est de minimiser le plus possible les gains du cache et de mesurer le temps que prend réellement l'opération d'écriture. On peut observer que Ext4 est plus lent que MTFS. Ceci est dû au fait que Ext4 effectue plus d'opérations que MTFS lors de l'écriture d'un fichier. Sans les *flags* Ext4 est 10 fois plus rapide que ce que l'on peut observer sur la Figure 8.3.

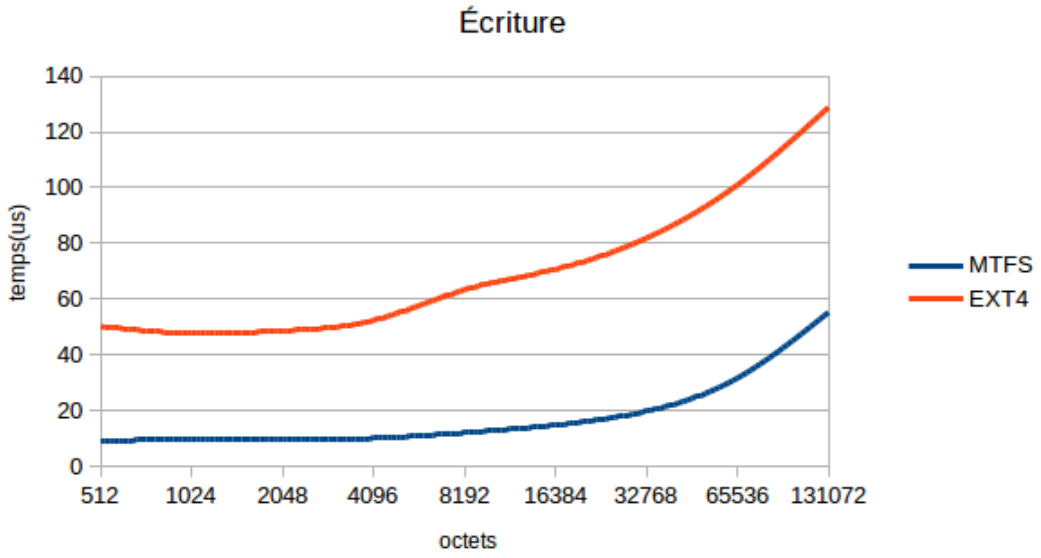


FIGURE 8.3 – Temps d'écriture d'un fichier en fonction de la quantité d'octets écrit. Les mesures ont été réalisées sur un disque SSD formaté en ext4 et sur MTFS.

Comme l'illustre la Figure 8.4, il y a nettement moins de différences entre les deux FS lors de la lecture. Ceci est dû au fait que lors de la lecture aucune opération n'est faite en plus de la lecture du contenu. Là encore sans les *flags*, Ext4 est 10 fois plus rapide que sur le graphique.

Pour obtenir les graphiques Figure 8.3 et Figure 8.4 un échantillon de 5000 par taille de fichier a été relevé, puis la médiane de ces derniers a été faite.

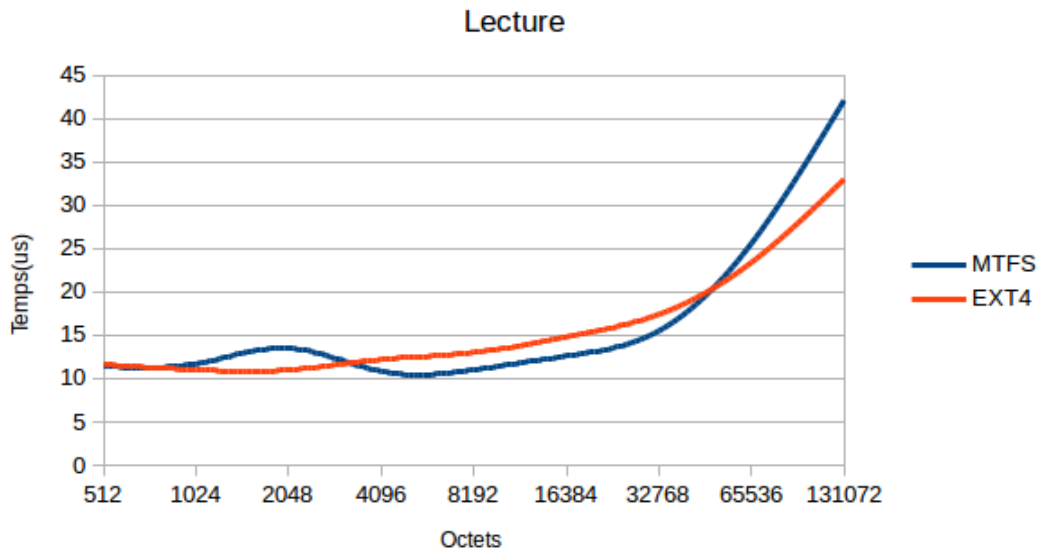


FIGURE 8.4 – Temps de lecture d'un fichier en fonction de la quantité d'octets écrit. Les mesures ont été réalisées sur un disque SSD formaté en ext4 et sur MTFS.

Chapitre 9

Discussion et perspectives

Les résultats obtenus démontrent de bonnes capacités du système. Dans l'implémentation actuelle, ce qui limite les performances est l'absence d'un système de cache. Le cache permettrait de considérer une action (comme l'écriture d'un bloc) comme terminée avant que les données ne soient réellement sur le ***volume***. En contrepartie, il faut implémenter une solution qui permet de ne pas perdre les données du cache qui ne sont pas encore écrites sur le ***volume***. Une solution à envisager serait le mécanisme de journal[17].

Dans l'état actuel de MTFS la gestion de la mémoire n'est pas optimale. Pour améliorer celle-ci deux solutions sont envisagées, soit l'utilisation de pointeurs intelligents, soit l'utilisation de pools de mémoire.

9.1 Problèmes rencontrés

Ces tests ont permis de constater que lors du montage du système FUSE crée déjà 20 *threads*. De plus durant les tests le nombre de *thread* maximum constaté est de 600, ce qui est anormal. Cela ne bloque en rien le système, car ce sont tous des *threads* qui ont terminé leurs tâches mais ils occupent de la place mémoire. Il faut modifier l'implémentation de sorte à ne plus utiliser les *threads pools* mais effectuer les opérations de manière synchrone. Ceci pour confirmer que le surplus de *thread* est effectivement créé par l'implémentation. Et enfin, procéder par paliers en réutilisant petit à petit les *thread pool* tout en vérifiant leur destruction une fois leur tâche terminée.

9.1.1 FUSE

La documentation de FUSE n'étant pas très précise, j'ai dû à plusieurs reprises aller lire le code source pour comprendre ce que le *handler* avait en paramètre, ou comment formater la réponse à transmettre. Par exemple, pour le *handler* `setattr` je n'arrivais pas à comprendre comment savoir quels sont les attributs à modifier. Il y a en paramètre un entier `to_set` dont la documentation dit ceci :

> "bit mask of attributes which should be set"

Mais à aucun moment il n'est précisé quel bit correspond à quel changement. En regardant le code j'ai finalement trouvé la réponse qui est visible au Listing 9.1

```
/* 'to_set' flags in setattr */
#define FUSE_SET_ATTR_MODE  (1 << 0)
#define FUSE_SET_ATTR_UID (1 << 1)
#define FUSE_SET_ATTR_GID (1 << 2)
#define FUSE_SET_ATTR_SIZE  (1 << 3)
#define FUSE_SET_ATTR_ATIME (1 << 4)
#define FUSE_SET_ATTR_MTIME (1 << 5)
#define FUSE_SET_ATTR_ATIME_NOW (1 << 7)
#define FUSE_SET_ATTR_MTIME_NOW (1 << 8)
#define FUSE_SET_ATTR_CTIME (1 << 10)
```

Listing 9.1 – Fichier `credentials` pour le SDK AWS

Un autre problème rencontré avec FUSE a été de comprendre le fil d'exécution et la paralélisation des *handlers*. Dans une première version de MTFS c'est l'objet `Mtfs` du *core* qui s'occupait de démarrer des *thread* pour effectuer les actions. Puis après quelques tests il c'est avéré que FUSE faisait déjà ce travail ce qui a permit de simplifier le code de `Mtfs`.

9.1.2 Écriture de fichier

Dans la version actuelle de MTFS un bug ne permet pas d'écrire des fichiers plus gros que 256 Ko. Ce bug empêche d'utiliser le *benchmark* IOzone[18] pour les tests de performance du FS.

9.2 Améliorations futures

Bien que les objectifs du cahier des charges soient atteint il reste plusieurs améliorations possibles.

La résolution des bugs constatés en section 9.1 est une priorité.

Ensuite, il faut effectuer une vérification plus précise de la configuration avant son installation. Ceci, dans le but de mieux détecter les risques évoqués en sous-section 5.5.4 et sous-section 5.5.5.

Une autre amélioration possible serait l'exécutable `mtfsCreate`. Actuellement, il permet de créer complètement un système, mais il pourrait être intéressant d'intégrer une option permettant de lister les *plugin* ainsi que leur dépendance en matière de configuration. Il serait aussi intéressant d'ajouter une option pour installer un *plugin*, ceci dans le bug d'éviter au maximum a l'utilisateur de devoir aller dans le *home* de MTFS.

Une fois le système plus stable, il faudra se passer du format JSON pour l'inscription des données sur le disque. Un format binaire est à privilégier car plus compacte. Étant donné que les *inode* et *directory blocs* ont des tableaux dynamiques il faut penser à ajouter un champ lors de l'inscription en format binaire, ceci dans le but de connaître la taille du tableau lors de la lecture. Sans ça lors de la lecture on ne peut pas savoir combien de bytes il faut lire. La

solution la plus portable est de définir dans chaque structure de données devant être inscrit sur le disque, une fonction `toBinary` qui implémenterai cette conversion, et permettrai au *plugin* de directement l'utiliser. De plus cela ajouterait une possibilité de passer d'un format JSON en mode *debug* à un format binaire en mode *release*.

Au niveau de *core* une amélioration utile serait l'ajout du cache. Actuellement une classe abstraite existe pour définir le comportement du cache mais aucune classe abstraite n'a été implémentée. L'implémentation permet l'utilisation d'un cache par type de bloc. Cela permet d'avoir des caches de tailles différentes et avec chacun leur espace mémoire. Malgré ceci un seul cache pour tous les types de blocs est tout à fait possible.

Une autre piste intéressante serait au niveau de l'écriture des blocs. Il faudrait trouver un moyen plus élégant que le simple verrouillage du bloc écrit, une piste intéressante serait le *copy on write*[19].

Pour finir, l'ajout de *handler* comme par exemple `readlink`, `symlink` et `link` serait un plus pour le FS.

Chapitre 10

Conclusion

D'un point de vue personnel, réaliser ce projet m'a beaucoup appris. J'ai pu mettre en pratique ce que j'ai appris tout le long de ma formation. Cela m'a aussi permis d'apprendre le *framework* FUSE que je ne connaissais pas et qui ouvre d'immense possibilité en matière d'implémentation de FS. J'ai aussi largement approfondi mes compétences en programmation C++ qui est un langage que j'apprécie particulièrement. Cela m'a aussi permis de me rendre compte toutes les opérations qui sont continuellement effectuées sur un FS.

Malgré le problème d'écriture des fichiers de plus de 256 Ko le cahier des charges est respecté et le système est fonctionnel. Dans un premier temps je me consacrerais à mon travail et à mon master mais je pense continuer à travailler sur ce projet dans mon temps libre car je pense qu'il a de grandes capacités et répond à un besoin présent. Tout le monde peut apporter des améliorations à MTFS étant donné que l'intégralité du code source est disponible sur github à l'adresse suivante `https://github.com/Dawen18/mtfs` et est soumit à la licence GNU GPLv3.

Bibliographie

[1] Quantum. Stornext. http://www.quantum.com/products/scale-out-storage/stornext-data-management/index.aspx. [En ligne ; Page disponible le 16-août-2017].

[2] Wikipedia. Inode pointer structure — wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php, 2016. [En ligne ; Page disponible le 16-août-2017].

[3] fscops. https://code.google.com/archive/p/fscops/. [En ligne ; Page disponible le 16-août-2017].

[4] btier. https://sourceforge.net/projects/tier/. [En ligne ; Page disponible le 16-août-2017].

[5] Alluxio. http://www.alluxio.org. [En ligne ; Page disponible le 16-août-2017].

[6] C.J. Aston, M.S. Laker, T.E. Willis, N. Berrington, M.A. Dorey, C.F. Garbagnati, and S. Shottan. Multi-tiered filesystem, September 23 2014. US Patent 8,843,459.

[7] Quantum. http://www.quantum.com. [En ligne ; Page disponible le 16-août-2017].

[8] H. Herodotou. *Towards a distributed multi-tier file system for cluster computing*. May 2016.

[9] Efficient qos for multi-tiered storage systems. In *Presented as part of the 4th USENIX Workshop on Hot Topics in Storage and File Systems*, Boston, MA, 2012. USENIX.

[10] Wikipedia. Quality of service — wikipedia, the free encyclopedia, 2017. [En ligne ; Page disponible le 16-août-2017].

[11] Wikipédia. Système de fichiers — wikipédia, l'encyclopédie libre. http://fr.wikipedia.org/w/index.php?title=Syst%C3%A8me_de_fichiers&oldid=138468779, 2017. [En ligne ; Page disponible le 16-août-2017].

[12] Wikipedia. Network file system — wikipedia, the free encyclopedia, 2017. [En ligne ; Page disponible le 16-août-2017].

[13] thread pool. http://threadpool.sourceforge.net/index.html. [En ligne ; Page disponible le 16-août-2017].

[14] Boost. http://www.boost.org/. [En ligne ; Page disponible le 16-août-2017].

[15] Amazon. Aws sdk for c++. https://github.com/aws/aws-sdk-cpp, 2017. [En ligne ; Page disponible le 16-août-2017].

[16] Google. googletest. https://github.com/google/googletest, 2017. [En ligne ; Page disponible le 16-août-2017].

[17] Wikipedia. Journaling file system — wikipedia, the free encyclopedia, 2017. [En ligne ; Page disponible le 16-août-2017].

[18] IOzone. Iozone filesystem benchmark. http://www.iozone.org/, 2017. [En ligne ; Page disponible le 16-août-2017].

[19] Wikipedia. Copy-on-write — wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Copy-on-write&oldid=788173851, 2017. [En ligne ; Page disponible le 16-août-2017].