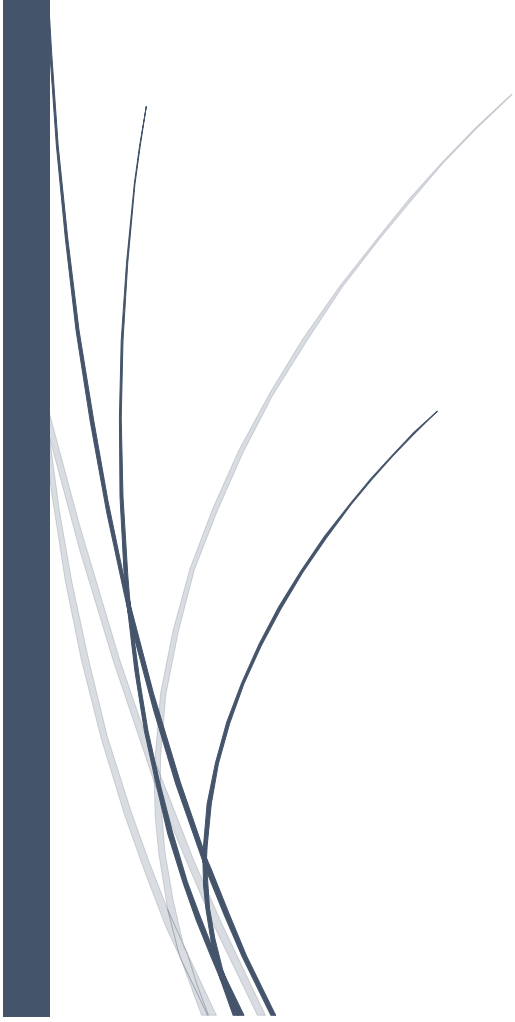
11/11/2015

Système de fichier basé sur les tags

Projet de semestre



Anthony Ramirez
HEPIA ITI 3ÈME ANNÉE

1 Table des matières

Table des matières

2 Cahier des charges

2.1 Descriptif

Avec le volume de données personnelles en constante augmentation, les utilisateurs se retrouvent confrontés à une quantité gigantesque d'information difficile à classifier et à rechercher. Le paradigme du chemin d'accès/fichier est trop limitant pour organiser efficacement ses données de manière. Par exemple, comment retrouver une photo particulière dans une collections de 50'000 photos disséminées dans des centaines de répertoires ? Ce projet se divise en 2 étapes. La première étape vise à recenser les systèmes de tagging de fichiers existants afin d'en déterminer un flexible, robuste et performant (ex: robustesse aux déplacements fichiers). La deuxième étape est de développer un mécanisme permettant de réaliser un tagging de fichiers automatisé en se basant sur divers critères pertinents (extension, mime type, EXIF pour les images, etc.). Au terme du projet, un utilisateur devrait être capable de retrouver ses données beaucoup plus efficacement qu'avec une approche classique.

2.2 Travail demandé

- Recherche des différents « desktop search engines » existants, en particulier les implémentations remplissants les critères ci-dessous :
    - Implémentations encore activement développées
    - Multi-plateformes pour les 3 plateformes considérées : Linux, Mac, Windows
    - Code source disponible et libre.
- Etude et tests des systèmes de notification de fichier pour les 3 plateformes considérées.
- Implémentation d'une couche d'abstraction multi-plateforme indépendante du système de notification sous-jacent.
- Proposition d'un modèle de stockage permettant de stocker un ensemble de tags pour chaque fichier. Ce modèle doit pouvoir « scaler » à un très grand nombre de fichiers et être rapide d'accès. De plus, il doit remplir les critères ci-dessous :
    - Ajouter, renommer et supprimer un/des tags doit être très rapide
    - Persistance des tags en cas de déplacement ou de copies de fichiers
    - La création d'index (s'il y en a une) doit prendre peu de temps (quelques minutes au maximum) et doit consommer très peu de ressources (CPU et RAM).
    - Aucuns problèmes de synchronisation entre les fichiers et les tags associés (sauf très temporairement).
- Implémentation du modèle de stockage.
- Implémentation d'applications simples pour la manipulation des tags (ajout, renommage, suppression, etc.).
- Si le temps le permet : inférence de tags simple automatique (pour populer un ensemble de fichiers avec des tags « initiaux »).
- Démonstrateur.

3 Introduction

Avec l'agrandissement croissant de la taille des disques durs et du nombre de fichiers que nous devons gérer quotidiennement, l'architecture hiérarchique traditionnelle composée de dossiers contenant des fichiers commence à montrer ses limites. L'idée de ce projet est de proposer une manière différente de visualiser ses fichiers afin de les retrouver plus facilement et de pouvoir modifier son architecture à la volée.

La piste explorée est d'utiliser un système de tagging géré par l'utilisateur. Le but est que celui-ci peut ajouter des tags sur les fichiers pour les organiser, ce qui a pour effet de créer un réseau de tags avec des fichiers au bout. L'avantage est de sélectionner très rapidement les fichiers qu'on veut en donnant les tags appropriés, mais aussi d'avoir plusieurs moyens d'accéder à la même ressource. Par exemple, une photo qui serait à la fois reliée à un voyage et au fait que ça soit une photo de montagne.

4 Système orienté utilisateur vs. Tagging automatique

Le choix de faire un système de tag orienté utilisateur, à l'inverse des tags automatiques qui tentent de classer le fichier par rapport à son contenu, est tout d'abord de pouvoir coller le plus possible aux besoins d'une personne. Ensuite, cela permet une plus grande flexibilité puisque chacun peut ajouter les tags qu'il désire. Pour finir, c'est aussi un moyen plus sûr car l'utilisateur a le contrôle sur tout (le système n'ajoute rien automatiquement).

Cependant, chaque fichier possèdent des attributs intéressants qu'il serait dommage de négliger. Les quatre principaux étant son nom, les dates de création et de dernière modification et son extension. Il serait aussi intéressant de sauvegarder toutes les dates de modification afin de savoir sur quoi l'on a travaillé un jour donné.

5 Analyse de l'existant

Il existe de nombreux programmes sur le marché qui proposent ce genre de fonctionnalité, chacun ayant ses avantages et ses inconvénients. J'ai donc commencé par rechercher s'il existait un programme qui répondait exactement à mes besoins ou qui s'en approchait et dont je pourrais m'inspirer.

5.1 Windows

Windows possède la plus grande palette de « search desktop engine », avec des outils très performants dans la recherche de fichiers sur le disque tels qu'*Everything* ou Copernic. Néanmoins, je n'ai trouvé aucun programme sérieux qui implémente des tags.

5.2 Linux

La communauté Linux est la plus active dans le domaine puisqu'en plus de projet comme le *NEPOMUK Semantic Desktop*, qui dépasse largement le cadre de ce projet mais qui inclut un système de tagging, on peut trouver des outils comme Tagsistant ou Tracker. Les deux répondent au cahier des charges, implémentent des tags et sont robustes au déplacement de fichier. Leur principal problème se trouve dans l'utilisation au quotidien selon moi.

Tagsistant propose une recherche basé sur des dossiers : On va chercher dans le dossier virtuel du tag désiré pour trouver notre fichier. Il est aussi possible de cumuler les tags et de préciser la recherche avec des attributs (par exemple une partie du nom du fichier). La principale critique que je ferais à ce programme est que la recherche n'est pas très intuitive : il faut toujours utiliser le fichier qu'on a configuré au départ.

Tracker implémente une recherche de fichier classique, avec une longue construction d'index à l'installation, ainsi qu'un système de tag. Un peu comme pour Tagsistant, la recherche, qui se fait cette fois en ligne de commande, pêche un peu. En effet, elle permet de **trouver** ses fichiers mais pas forcément de les **utiliser**. Étant donné que ce n'est pas la fonctionnalité principal du programme, elle semble un peu gadget.

5.3 Mac

Comme pour beaucoup de choses dans l'univers Mac, Apple a pris les devants pour proposer un système de tagging très performant aux utilisateurs. *Spotlight*, le système de recherche de Mac OSX, permet d'ajouter facilement des tags avec une interface GUI, le seul à en posséder une, et de les retrouver très facilement dans sa barre de recherche universelle. Cet outil est très facile à utiliser, on ajoute ses tags rapidement et la recherche est également extrêmement simple.

La seul critique que je ferais à *Spotlight*, un peu comme *Tracker*, est que la fonctionnalité de tagging semble gadget et n'est pas faite pour être utilisée au quotidien. Cela sert plutôt à retrouver quelques fichiers importants rapidement que de s'en servir comme système de fichier à part entière.

5.4 Critiques

Dans mes recherches, j'ai repéré deux problèmes récurrents dans pratiquement toutes ces solutions. Le premier est particulièrement visible dans le résultat de mes recherches pour Windows : l'accent est mis sur des programmes qui recherchent des fichiers directement sur le disque de la manière la plus rapide possible, et non pas sur une autre manière d'organiser les fichiers (qui amènerait à les retrouver plus rapidement).

Le second, qui est lié au premier, est que même les systèmes qui implémentent la fonction de tagging ne sont pas focalisés sur celle-ci, ce qui fait qu'elle a un côté un peu « gadget ». La seule exception à ça est *Tagsistant*, dont c'est la fonctionnalité principale.

La dernière critique que j'emmétrais est que je n'ai trouvé aucune solution à la fois viable, open-source et surtout cross-platform.

6 Solutions envisagées

Après avoir fait le tour des programmes existants, j'ai pu analyser ce qu'il manquait à ceux-ci et donc ce sur quoi je devais me focaliser. Il fallait donc trouver un moyen de proposer les fonctionnalités de base d'un système de fichier basé sur les tags et dans l'idéal que cela soit cross-platform.

La première solution était de se concentrer sur Linux et de me baser sur le code de *Tagsistant* ou *Tracker* étant donné qu'ils répondent déjà au cahier des charges et qu'ils sont open-source. Le risque étant que je passe plus de temps à comprendre le code et la logique derrière qu'à vraiment développer quelque chose. De plus, ils implémentent des fonctionnalités qui sortent du cadre et qui complexifient le programme alors que ce n'est pas nécessaire.

C'est pourquoi j'ai décidé de partir de zéro en basant mon programme sur des composants que j'ai choisis moi-même. J'ai donc du rechercher les API disponibles pour chaque système.

6.1 Interception des évènements du système de fichier

Afin de rendre mon programme robuste, il doit pouvoir intercepter les modifications faites par l'utilisateur sur un fichier comportant un tag et mettre à jour ses informations en conséquence. Étant donné que le but est de travailler sur plusieurs OS différents, j'ai étudié les API disponibles pour chacun d'eux.

Windows implémente un composant .NET **FileSystemWatcher** qui permet de définir un fichier à monitorer et récupérer les événements de modification de ses attributs. Les fonctionnalités sont donc sommaire et ne concerne que les modifications (pas d'ouverture, lecture etc…).

Dans le monde Linux, **inotify** possède les mêmes fonctionnalités en ajoutant les actions qui n'entrainent pas de modification (lecture, ouverture, fermeture etc…). De plus, il existe plusieurs outils très puissants tels qu'**incron**, **lsyncd** ou **iwatch** qui se basent sur ce principe. Il existe des APIs pour Python, Java, Perl, PHP et Ruby mais limitée à Linux.

Pour MacOS, la librairie **FSEvents** permet de faire sensiblement la même chose que le **FileSystemWatcher** de Windows. Le programme « s'enregistre » pour recevoir les notifications d'un dossier. Le fait de voir les modifications d'un fichier en particulier n'est disponible que depuis la version 10.7 cela dit. L'API est utilisable en

Ces solutions répondent à mes besoins hormis le fait qu'elles sont spécifiques à un OS et absolument pas cross-platform.

6.2 Création d'un index

La recherche de fichier sur le disque de manière « traditionnelle » n'entrant pas dans le cadre du cahier des charges, j'ai choisis de ne pas faire d'index car cela aurait complexifié inutilement le programme. Le seul index conservé en mémoire est celui des tags.

7 Solutions choisies

Comme expliqué au point précédent, j'ai trouvé plusieurs APIs qui correspondait à mes besoins mais qui avait le principal défaut de ne pas être cross-platform. Je me suis donc tourné vers des solutions plus générales.

L'API **watchdog** permet de récupérer les opérations de création, déplacement, modifications et suppression des fichiers et dossiers sur Linux, Windows, MacOS et BSD. Cette solution est en fait une implémentation en Python des solutions décrites plus haut, plus une partie polling qui est commune à tous les OS mais qui ne sert qu'en dernier recours à causes des mauvaises performances.

7.1 Justification

J'ai choisi cette API principalement parce qu'elle est cross-platform, répond à mes besoins, que le langage Python m'est familier et possède une bonne communauté, et parce qu'elle utilise des outils natifs derrière que je connais puisque ça correspond au résultat de mes recherches.

7.2 Limitations

Le problème de vouloir travailler sur plusieurs OS est qu'il faut sacrifier la spécificité de chacun, donc **watchdog** se limite aux fonctionnalités de base de la détection d'évènement du système de fichier.

7.2.1 Déplacement hors de la zone surveillée

L'autre gros problème, qui est inhérent à l'API **inotify**, est qu'il est impossible de détecter lorsqu'on fichier est déplacé hors de la zone « surveillée » par le watchdog. Il est donc obligatoire de suivre une série de répertoire de manière récursive pour éviter de rater le déplacement d'un fichier et de le confondre avec sa suppression.

7.2.2 Pas de notification de la suppression d'un répertoire suivi

Lorsqu'on supprime un répertoire suivi, aucune notification de suppression n'est générée puisqu'il n'est pas considéré comme étant dans la zone surveillée. La solution serait de suivre automatiquement le répertoire parent. Étant donné que j'ai trouvé ce bug tard, je n'ai pas eu le temps de tester et de mettre en place cette solution mais cela ne demanderait pas beaucoup de temps pour le faire.

7.3 Choix du langage

Étant donné le besoin de portabilité et les APIs disponibles, le Python s'est imposé comme choix logique. De plus, j'ai déjà pas mal d'expérience avec ce langage donc c'était plus pratique.

8 Analyse fonctionnelle

8.1 Fonctionnalités utilisateur

8.1.1 Ajouter un tag

Commande : addtag tag file [file] [file] …

8.1.2 Supprimer un tag

Commande : removetag tag file [file] [file] …

8.1.3 Rechercher par tag

Commande : searchfile tag

Options :

- --npi

  Permet de passer en mode notation polonaise inverse.

  Ex : searchfile --npi hiver montagne @and neige @not @and été @or

De base la commande applique un ET logique sur les tags passés en paramètre par l'utilisateur. La notation polonaise inverse permet une plus grande liberté sur la recherche que l'on veut faire. Étant donné que le shell interprète les caractères & et |, il est nécessaire de les mettre entre guillemets. Le NOT permet d'avoir une négation et est un opérateur unaire.

La commande donnée en exemple effectue la recherche ((hiver & montagne) & !neige) | été

8.2 Fonctionnalités système

Les fonctionnalités systèmes représentent le travail que le daemon fait de manière automatique en réaction à un événement déclenché par le système.

8.2.1 Déplacement de fichier

Lors d'un déplacement, le chemin du fichier et/ou son nom sont mis à jour dans la base.

Le fonctionnement normal du watchdog est que lorsqu'on déplace un fichier hors du scope de dossiers qu'il suit, il considère que le fichier n'existe plus et qu'il a été supprimé donc il en efface toutes traces dans la base de données.

8.2.2 Suppression de fichier

Lorsqu'un fichier est supprimé, le daemon met à jour la base de donnée en supprimant l'entrée du fichier, ses relations avec des tags et vérifie si ces derniers ou le dossier parent du fichier sont toujours utilisés. Si ce n'est pas le cas, il les supprime aussi de la base.

Si c'est un dossier qui est supprimé, le daemon va stopper le scheduler du watchdog sur ce dossier, supprimer les fichiers contenus dans ce dossier ainsi que leurs liens avec des tags et vérifier si ces tags ont encore des liens avec des fichiers.

Pour finir il va également supprimer ce dossier de la base.

8.2.3 Création de fichier et tags automatiques

L'événement de création de fichier dans les répertoires suivis est détecté et, pour l'instant, la date du jour est ajoutée automatiquement comme tag sous la forme « DD-MM-YYYY » ainsi que l'extension. Il est très facile de rajouter d'autres tags automatiques par la suite.

8.2.4 Logging

Étant donné que, par définition, le daemon n'a pas d'accès à un terminal, il était important de garder un log des actions qu'il effectue. J'ai donc du tout au long du développement mettre en place un maximum de log pour suivre les opérations de ce dernier.

Il est aussi important d'avoir une trace des modifications effectuée par le watchdog étant donné qu'elles ne sont pas directement contrôlées par l'utilisateur. Chaque événement est donc logé avec l'heure et la date à laquelle il a été déclenché.

9 Architecture et implémentation

9.1 Schéma

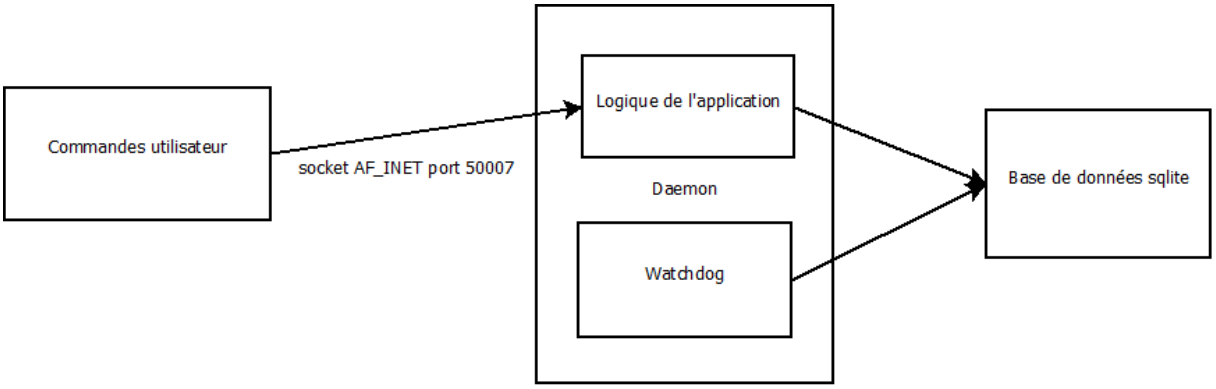


Figure 1 - Architecture générale du projet

9.2 Watchdog

Le travail du watchdog est d'intercepter les événements de **création**, **déplacement** et **suppression** sur les répertoires que l'on veut suivre. Il va ensuite appeler une fonction de callback avec en paramètre le fichier source, le type d'événement et, en fonction de celui-ci, le fichier destination ou non.

Il va ensuite effectuer les opérations détaillées au point 7.2 selon l'événement déclencheur. Par contre, le watchdog s'exécute et lance le callback dans un thread séparé du daemon. En effet, lorsque ce dernier démarre, il ne fait que crée un **observer** qui va se charger de suivre les dossiers qu'il trouve dans la base de données grâce à des **schedulers** (un par dossier). Ensuite le watchdog fonctionne de manière autonome.

Il est possible de suivre un dossier et ses sous-dossiers mais j'ai fait le choix de ne pas utiliser cette option pour garder un contrôle plus strict sur les dossiers suivis et éviter d'avoir des informations non pertinente (déclenchement d'événement sur des fichiers qui ne sont pas dans la base).

Le daemon enregistre la fonction **update_database** comme callback unique pour les trois événements qui nous intéressent. On peut aussi utiliser une fonction différente pour chacun mais j'ai fait le choix de réunir tout en une fonction pour des questions de simplicité.

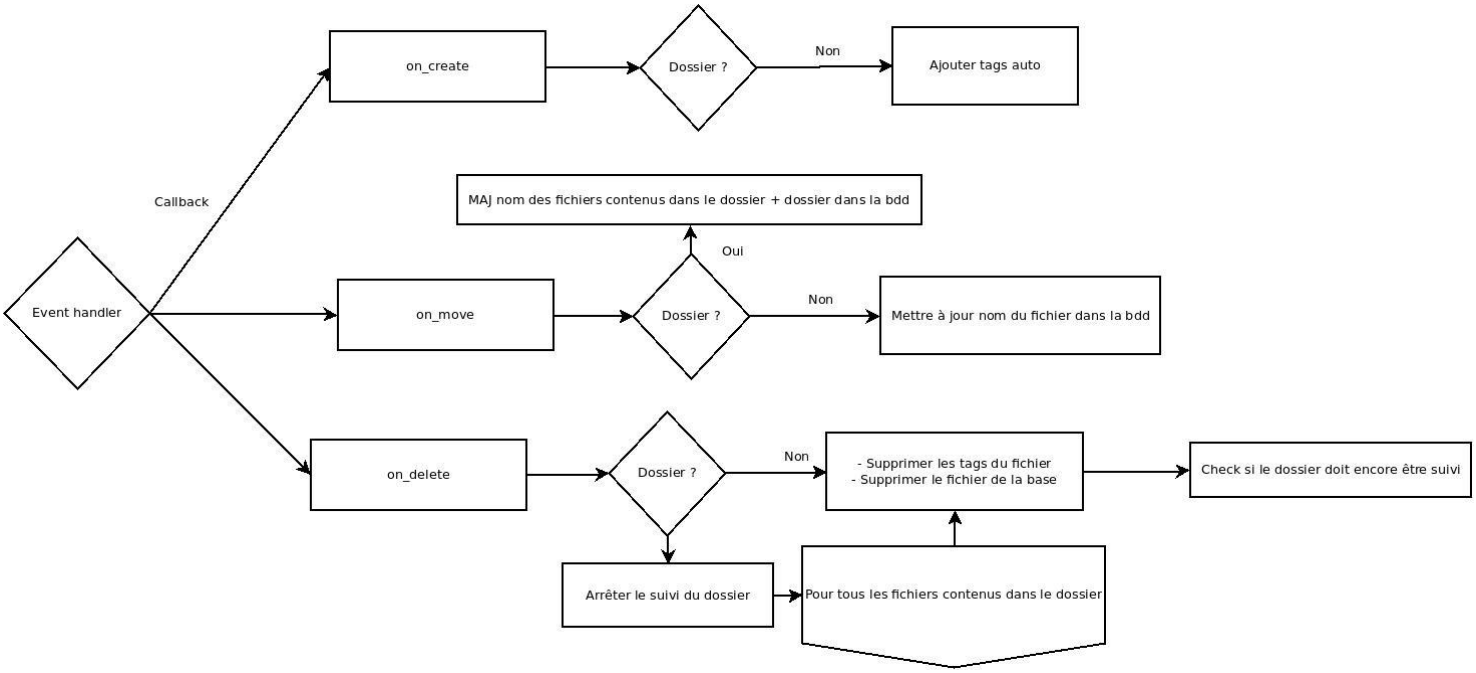


Figure 2 - Architecture du watchdog

9.3 Protocole

L'utilisateur communique utilise les différentes commandes mise à disposition pour donner des instructions au daemon. Celui-ci se charge d'ajouter ou de supprimer des tags dans la base, avec toutes les actions que cela implique.

Pour faciliter la réception, le programme envoie d'abord la taille des données, puis dans un deuxième temps les données elles-mêmes. Lorsqu'il y a plusieurs fichiers ou tag, le client met en forme les différents noms dans une seule chaîne de caractère pour tout envoyer d'un coup.

9.4 Clients

Les fonctionnalités côté client ont été divisées en plusieurs scripts python pour simplifier l'utilisation et éviter d'avoir une seule commande avec un nombre immense d'options qui peuvent en plus être contradictoires.

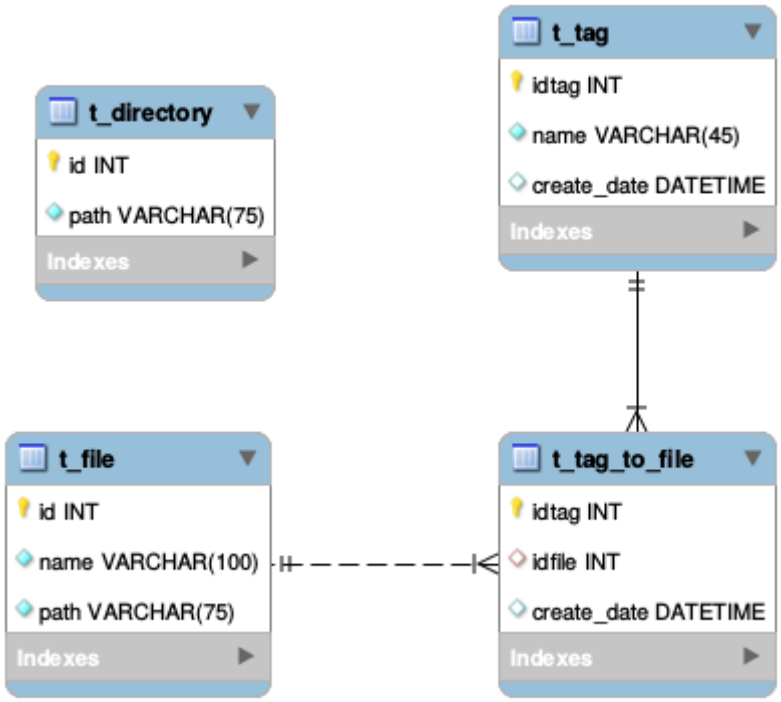
9.5 Base de données

La base de données est extrêmement simple. J'ai choisi d'utiliser SQLite comme SGBD afin d'avoir une base de données sous forme de fichier et pour sa légèreté. Le but est de stocké la liste des tags, des fichiers tagués, les liens entre les deux et la liste des dossiers suivis.

Une chose importante à noter dans le code lors de la connexion à la base de données, si la base de données est modifiée par un autre processus après que la connexion ait été ouverte, ces modifications ne sont pas visibles. C'est pourquoi je suis obligé de fermer puis ouvrir une connexion à chaque événement détecté par le **watchdog** ou à chaque appel au **daemon**.

La création du fichier contenant la base de données se fait grâce au script **install.py**. J'ai choisi cette solution pour uniformiser l'installation sur les différents OS et pour éviter certains problèmes de droits d'accès aux fichiers du **daemon**.

La table **t_tag_to_file** possède une contrainte **ON DELETE CASCADE** sur ses deux clés étrangères **idtag** et **idfile**, ainsi lorsqu'on supprime un fichier ou un tag de la base, le lien est automatiquement supprimé.



9.6 Daemon

Le daemon contient toutes la logique de l'application. C'est donc entièrement ici que les accès à la base de données se font. Il est commandé par les programmes utilisateur à travers un socket. Chaque commande contient l'instruction à effectuer et les arguments nécessaires. Il s'occupe aussi de configurer et de lancer le **watchdog**.

Au lancement, il crée un socket, lit la liste de dossiers dans la base de données, crée un **scheduler** pour chacun d'eux et démarre le **watchdog**.

Après le démarrage, le **daemon** est une boucle infinie qui attend sur un socket. Lorsqu'il reçoit une connexion, il lit le premier message qui contient la taille du message qu'il va recevoir. Ce premier message a une longueur fixe de 4 caractères.

Une fois qu'il a reçu le message, il effectue un parsing, récupère les arguments et exécute les instructions appropriées. A la fin, la connexion du socket et celle avec la base de données sont fermées.

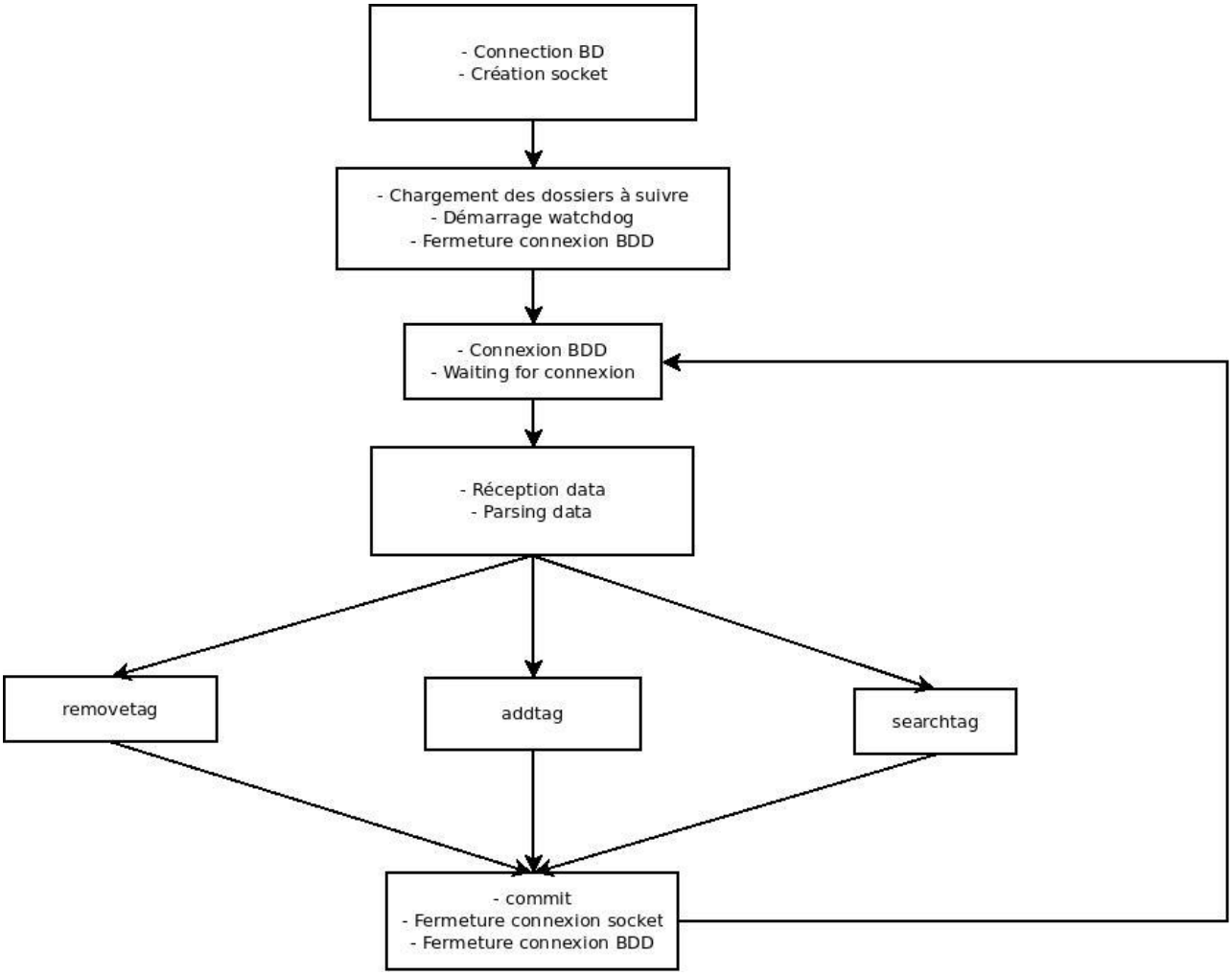


Figure 3 - Architecture du daemon

10 Performances et tests

Ces tests ont été réalisés sur une machine Ubuntu 14.4 installée sur le disque dur (Pas une VM).

10.1 Ajout d'un tag sur 2500 fichiers dans un seul répertoire

Le contexte de ce test de performance est le tagging d'un dossier de manière récursive avec un tag unique. Ce dossier contient 2500 fichiers qui eux-mêmes n'ont aucun tag avant le test.

Commande : ./addtags tag /home/anthony/mydir

```
[24-03-2016 16:30:36] Request received
tagging directory
1229
Insertion tag
Insertion fichier
tag added
(...)
[24-03-2016 16:30:38] Commande executée
Waiting for connexions
```

Résultat : 2 secondes

10.2 Suppression d'un tag sur 2500 fichiers dans un répertoire

Pour ce test j'ai repris le dossier précédent et simplement supprimé les tags des fichiers. La commande est lancée sur le dossier et donc appliquée récursivement sur les fichiers à l'intérieur.

Commande : ./removetag tag /home/anthony/mydir

```
[24-03-2016 16:31:00] Request received
tag removed
(...)
tag removed
[24-03-2016 16:31:04] Commande executée
Waiting for connexions
```

Résultat : 4 secondes

La différence de temps avec la commande précédente s'explique par un accès à la base de données supplémentaire et le fait qu'il faille tester si le tag est toujours utilisé à chaque fois qu'on supprime le lien entre celui-ci et un fichier.

10.3 Suppression d'un répertoire contenant 2500 fichiers tagués

```
DB connected
Socket listening on 50007
Waiting for connexions
[09-03-2016 22:12:46] /home/alekzander/testdir/cursor (1'303e copie).png deleted
...
[09-03-2016 22:13:21] Traitement terminé
```

Résultat : 35 secondes

10.4 Opération sur fichiers tagués

Étant donné qu'il s'agit d'une opération similaire à la suppression, mais en moins couteuse en ressource, je n'ai pas fait de tests. Ceux-ci aurait forcément été meilleurs que le test de suppression.

10.5 Suivi d'un grand nombre de répertoire

Le nombre de répertoire suivi n'a aucune influence sur les performances, seul le nombre d'événement entre en compte. Étant donné que l'événement de modification n'est pas surveillé, qu'on fasse une action sur 100 fichiers dans un répertoire ou 100 fichiers dans 100 répertoires ne fait aucune différence. Néanmoins, le tagging récursif d'un dossier a été optimisé par rapport au tagging « classique » d'un grand nombre de fichiers.

11 Installation

11.1 Windows

- Installer Python 2.7 selon docs.python.org/2/using/windows.html
- Mettre pip à jour : python –m pip install –U pip
- Installer le watchdog : python –m pip install watchdog
- Exécuter le script python d'installation fourni

11.2 Linux

- apt-get install pip
- pip install watchdog
- pip install daemonize
- Exécuter le script python d'installation fourni

12 Discussions

12.1 Compatibilité avec les OS leaders du marché

J'ai pu mettre en place et tester mon programme sur les versions 7 et 10 de Windows ainsi que sur Ubuntu 14.4. Je n'ai malheureusement pas pu me procurer de Mac pour effectuer mes tests mais l'API watchdog est disponible sur Mac et le module subprocess utilisé pour crée un daemon sur Windows est également disponible donc par extension je suis confiant sur la compatibilité avec MacOS.

12.2 Améliorations possibles

12.2.1 Mise en mémoire des informations

Actuellement la plupart des informations (fichiers et dossiers suivis par exemple), sont dans la base de données, ce qui fait que le nombre de requêtes (qui sont potentiellement lentes) peut exploser et ralentir le daemon. En gardant ces informations en mémoire, l'accès serait beaucoup plus rapide et le coût moindre. La base de données servirait alors que de backup pour éviter de perdre les informations lorsqu'on éteint le daemon.

Cependant, étant donné la nature même du daemon, celui-ci accède à la base tourne en tâche de fond donc tant qu'il peut suivre, il n'est pas forcément intéressant d'apporter cette modification puisqu'elle n'apporterait rien pour l'utilisateur.

12.3 Bugs

Les principaux bugs sont plutôt liés à des limitations des APIs décrites plus haut dans le rapport.

13 Conclusion

13.1 État du travail

Le travail rendu possède les fonctions de base prévues et est multi-plateforme donc il remplit le cahier des charges de ce côté-là. Les tags sont « robustes » dans la limite des possibilités offertes par l'API, c'est-à-dire qu'ils suivent lorsqu'il y a un déplacement (pour autant que cela reste dans la zone surveillée par le watchdog).

13.2 Commentaires personnels

J'ai apprécié travailler sur ce projet mais j'ai eu beaucoup de mal à m'y mettre. La motivation a fait défaut vers le milieu et il m'est difficile d'avancer quand je travaille par petites tranches de quelques heures. De plus, étant donné le temps à disposition, le travail de semestre a toujours passé en dernière position parmi mes priorités par rapport aux autres travaux à rendre.

14 Annexes

14.1 Code

14.1.1 Daemon.py

14.1.2 Removetags.py

14.1.3 Addtags.py

14.1.4 Searchfile.py

14.1.5 Install.py

14.1.6 Start_daemon.py