Kevin Youna
Tomek Lecocq
Karim Yahiaoui
Valloris Cylly
Nguyen Luan

M2 IAP-LP & IAP-IMPAIRS
Systèmes Avancés

TP3 – Stockage Objets

Le but de TP était de proposer une solution de stockage possédant les trois opérations : « put », « get » et « delete » .

Notre solution est découpée en plusieurs sections. Tout d'abord nous avons le code de la partie stockage, en C++. Ensuite il y a un client minimal, utilisable en ligne de commande, lui aussi implémenté en C++. Et finalement le serveur REST, implémenté en JavaScript (nodejs) qui permet d'accéder aux objets à partir d'une connexion HTTP.
Le serveur REST n'implémente que le GET et le PUT

Nous avons choisis C++ pour profiter de sa librairie standard un peu plus étoffée que celle du C tout en conservant la compatibilité avec les bibliothèques C. De plus ce langage offre des performances similaires au C.
Pour le serveur nous avons préféré d'utiliser JavaScript car il était très simple d'implémenter un serveur HTTP en utilisant ce langage et quelques modules nodejs. De plus, nodejs s'interface assez facilement avec les modules externes en C/C++.

Le choix d'utiliser des UUID est simplement pour la facilité d 'utilisation car il existe déjà des librairies pour manipuler ces clés sur Linux.

La lecture des fichiers se fait toujours avec un mappage du fichier en mémoire. Pour l'écriture (lors d'un PUT par exemple) nous la faisons à l'aide d'appels write.

Comment vont être stockés les objets ? Quel format de données sur disque allez-vous utiliser ?

Les objets sont enregistrés dans des fichiers, ceux-ci sont créés et placé dans une arborescence lors de la création de l'objet. On a choisi d'avoir une arborescence (au lieu de mettre tous les fichiers dans le même dossier) par souci de performance.
Un objet correspond toujours à un fichier dont le nom est sa clé et sa valeur le contenu du fichier.

Comment allez-vous « indexer » les objets (passer de la clé aux données stockées pour les GET et les DELETE ?

L'arborescence est créée à partir de la clé de l'objet. Les clés sont au format UUID.
Par exemple pour la clé `8f40c608-3f64-496e-817a-c5618843665b` :

```
8f40
└── c608
    └── 3f64
        └── 496e
            └── 817a
                └── c561
                    └── 8843
                        └── 66
                            ├── 5b
                            └── index
```

L'index est un fichier qui permet de savoir si l'objet est présent dans le système de stockage. Il fait la taille (en bits) du nombre d'objets qu'il pourrait y avoir au plus dans son répertoire. Ici c'est donc 256 bits (ou 32 octets). Le dernier octet de l'UUID permet de connaître la position de l'objet dans l'index.
Dans l'exemple au dessus, pour que l'objet soit vu par notre programme, il faut donc que le 91ème bit (`0x5b`) du fichier `index` soit à 1.

Comment allez-vous vérifier que les données ne sont pas corrompues ?

On ne vérifie pas l'intégrité des objets. On pourrait utiliser un filesystem comme ZFS ou Btrfs pour que le système de fichiers le fasse à notre place.

Quelle interface allez-vous offrir pour accéder à ce service? (librairie ? Commande ? Autres...)

On propose une interface en ligne de commande pour le client basique. Il y a également une API REST si on lance le serveur HTTP.
Il est possible d'adapter notre solution assez facilement pour en faire une librairie (ça serait assez similaire à ce que fait le wrapper pour JavaScript/nodejs).

Comment allez-vous vous assurer que votre système (les données que l'utilisateur vous confie) résiste aux crash de votre service, du système, de la machine? Quelles précautions allez-vous prendre ?

Lors d'un PUT, l'objet n'est ajouté dans l'index qu'après avoir fini de l'ajouter. Donc lors d'un crash on ne risque pas d'avoir des objets incomplets reconnus par le programme. Par contre on peut avoir des fichiers de données d'objets non-indexés prenant de la place pour rien. Il faudrait faire un outil pour les traquer et les supprimer.

Allez-vous vous appuyer sur un système de fichiers existants où allez-vous gérer les disques directement sans système de fichiers?

On utilise le système de fichier existant. Tous nos tests on été faits sur le filesystem ext4, mais ça devrait pouvoir fonctionner sur n'importe quel système de fichier supportant des noms longs et n'ayant pas de petites limites au niveau de la taille maximale des fichiers ou du nombre de répertoires.
On peut d'ailleurs gérer plusieurs disques (locaux ou sur le réseau) s'ils sont montés dans l'arborescence avec la bonne nomenclature.

Votre service doit-il s'appuyer sur un ensemble de processus? Sur un seul processus multi-thread? Sur un seul processus mono-thread? Utilisation de librairies d'entrées sorties asynchrones ?

Le programme de base (en ligne de commande) utilise un seul processus mono-thread avec une utilisation des entrées sorties synchrones. Il a été prévu d'utiliser des file locks à l'aide de fnctl pour que plusieurs clients puissent travailler sur la même arborescence d'objets.
L'API REST est gérée par le serveur node.js, le nombre de processus et de threads est laissé au choix de celui-ci. (Nous ne connaissons pas les details de l'implémentations de nodejs et de son fonctionnement).

Quelles ressources (CPU / mémoire) la machine physique hébergeant votre service devrait-elle raisonnablement avoir?

Nous avons testé sur nos machines personnelles (CPU 2 coeurs, 4GB de RAM) et ça fonctionne même sur de gros fichiers (jusqu'à 1GB) . Nous n'avons pas testé la charge avec une très grosse quantité d'objets et requêtes à la fois.