Software Heritage
Rapport de stage de fin d'étude

Contribution à un logiciel libre visant à archiver la totalité du code publiquement accessible de manière pérenne

Quentin Campos
Université Paris-Est Marne-la-Vallée

Mardi 6 Septembre 2016

Encadré par
Stefano Zacchiroli
Roberto Di Cosmo

Avril - Septembre 2016, Université Paris-Est Marne-la-Vallée

Stage encadré par Stefano Zacchiroli et Roberto Di Cosmo

Soutenu le 6 Septembre 2016

Table des Matières

Introduction

Les logiciels libres fournissent une base largement utilisée pour concevoir des programmes de plus en plus élaborés. Software Heritage a pour ambition de collecter et d'archiver de manière pérenne le code source publiquement disponible - majoritairement du code libre, ou du code appelé à le devenir dans le futur.

Archiver le code source, c'est conserver l'ADN des logiciels, et permettre de les préserver, les recompiler, ou de les étudier. A cette échelle, c'est à la fois un travail nécessaire pour conserver notre héritage digital, et une porte ouverte vers la prochaine échelle de recherche en génie logiciel.

La conservation des données n'est pas le seul objectif de Software Heritage. Il rend aussi accessible publiquement le code archivé. Cette bibliothèque publique permet de conserver *ad vitam aeternam* un lien vers toutes les versions existantes d'un logiciell. C'est un élément clef dans les science pour la reproductibilité scientifique comme dans l'industrie où la tracabilité du code critique est d'une importance capitale.

Software Heritage est un projet ambitieux oeuvrant pour la conservation de notre patrimoine et appelé à devenir, à l'instar du W3C, une organisation indépendante portée par des acteurs internationaux.

J'ai rejoint pendant mon stage l'équipe de Software Heritage après un an de développement. La possibilité de rejoindre un projet libre de cette importance à un stade aussi jeune de son développement a constitué une opportunité incomparablement enrichissante sur de nombreux aspects.

Ce rapport décrit plus en détail les enjeux et le fonctionnement de Software Heritage : son infrastructure et architecture logicielle ; ainsi que mes contributions au

projet et activités au sein de l'équipe.

1. Contexte du stage

Mon stage s'est déroulé entre Avril et Septembre 2016 au sein de l'Inria Paris. Il s'agit d'un stage de fin d'étude réalisé en conclusion de mon M2 à l'université Paris-Est Marne-la-Vallée. Cette section apporte des informations sur le contexte de ce stage et sur le projet auquel j'ai participé.

1.1 Inria

Fondé en 1967 dans le cadre du plan calcul, l'Inria (à l'origine IRIA [1]) est un institut de recherche public rattaché au ministère de l'éducation nationale, de l'enseignement supérieur et de la recherche. Le président actuel en est Antoine Petit.

Actuellement, l'Inria est constitué de huit centres de recherche. Le stage présenté dans ce rapport s'est déroulé dans les locaux de Paris, rattachés au centre de Rocquencourt, premier centre Inria à avoir ouvert ses portes.

1.2 Projet

Le projet Software Heritage, débuté en 2015, vise à collecter la totalité du code source publiquement accessible et à l'archiver de manière pérenne. Il est né au sein de l'Inria et à été publiquement annoncé le 30 juin 2016 lors d'une conférence de presse dans les locaux du centre de Paris.

Les motivations à créer un tel projet sont diverses :

Avant que l'informatique ne soit répandue comme elle l'est aujourd'hui, inscrire la connaissance sur des tablettes, parchemins, livres, ... permettait de sauvegarder

1. Institut de Recherche en Informatique et en Automatique.

cette connaissance. Aujourd'hui, toute nos nouvelles connaissances sont digitales. Sauvegarder les logiciels, c'est sauvegarder les outils qui nous permettent de faire avancer la science.

Lorsqu'une expérience est réalisée, afin que son résultat puisse être validé, il faut qu'elle puisse être reproduite par des pairs. Si cette expérience est le résultat d'un calcul ou d'une simulation informatisée, reproduire l'expérience nécessite de posséder le logiciel utilisé, qui pourrait être perdu dans quelques années (hébergement sur site personnel du chercheur, par exemple).

La recherche en informatique est elle aussi affectée, car une telle quantité de projets logiciels, dont l'historique de développement est conservé, ouvre de nouvelles portes dans l'étude du développement en lui-même, du modèle des logiciels libres, l'évolution d'un projet, ...

1.3 Équipe

Software Heritage est le produit d'une petite équipe constituée de quatre membres permanents, accompagnés par des conseillers scientifiques.

Roberto Di Cosmo (rdicosmo)
le directeur général est professeur à l'université Paris Diderot et président de l'IRILL.

Stefano Zacchiroli (zack)
est directeur de la technologie de Software Heritage, en détachement de l'université Paris Diderot, et à été Debian Project Leader trois années consécutives de 2010 à 2013.

Nicolas Dandrimont (olasd) et Antoine Dumont (ardumont)
développent le coeur de Software Heritage et sont tous les deux contributeurs dans des projets Open Source.

Jordi Bertran de Balanda (jbertran)
est le second stagiaire de Software Heritage, en M1 à l'université Pierre et Marie Curie, Paris

2. Software Heritage

Ce chapitre présente Software Heritage d'un point de vue plus technique en indiquant l'infrastructure mise en place, et l'architecture logicielle du projet.

2.1 Data Model

Le but de Software Heritage étant de conserver de manière pérenne la totalité du code source écrit et publiquement accessible, les données archivées par le logiciel ne doivent jamais être écrasées. Les accès aux données se font en *append-only*.

Lorsqu'un contenu est mis à jour, la version déjà présente dans Software Heritage n'est pas écrasée, mais une nouvelle entrée est crée.

2.1.1 Stockage des fichiers

Les données manipulées par Software Heritage sont adressables par contenu, c'est à dire qu'on peut retrouver leur identifiant (leur adresse) à partir de leur contenu. Dans le cas de Software Heritage, nos objets sont identifiés par leur sha1.

Les fichiers sources sauvegardés sont stockés sur disque dans une arborescence de répertoires qui dépend de leur identifiant.

2.1.2 Base de données

La base de données contient les méta-données qui structurent les fichiers sources en commits, révisions, projets (voir schéma complet de la base de donnée en annexe A.4 p.46).

2.2 Infrastructure

Software Heritage est un projet ambitieux qui requiert beaucoup de développement, mais aussi des ressources matérielles conséquentes, en terme de stockage principalement. Actuellement, deux machines physiques sont utilisées.

2.2.1 Serveurs Inria

Dans le centre de Roquencourt, dont dépend l'antenne de Paris où est situé l'équipe Software Heritage se situent les deux machines physiques du projet, **Louvre** et **Banco**. Louvre est un hyperviseur hébergeant les machines virtuelles du projet. Banco est la machine sur laquelle sont stockées les sauvegardes.

Les machines virtuelles sont gérées par `puppet`, un utilitaire qui permet de déployer et mettre à jour automatiquement les paquets requis pour que la machine puisse remplir son rôle.

Uffizi

contient les fichiers sources sauvegardé par Software Heritage. C'est sur cette machine que fonctionne le `Storage` et sont enregistrés les fichiers source archivés.

Prado

héberge la base de donnée PostgreSQL. Contrairement aux fichiers source, elle est enregistrée sur des disques durs SSD afin d'accélérer la vitesse d'exécution des requêtes critiques.

Getty

fait fonctionner le système de journalisation de Software Heritage qui enregistre les ajouts à la base. Des clients peuvent ensuite lire ces informations pour effectuer des opérations particulières sur tous les contenus nouvellement ajoutés (analyse du contenu, classification, ...)

2.2.2 Ouverture au cloud

Software Heritage est né au sein de l'Inria, mais son objectif à long terme est de devenir une organisation indépendante qui pourra perdurer *ad vitam aeternam* pour conserver l'héritage logiciel de l'humanité.

Software Heritage ne peut donc pas remplir son rôle en n'ayant que des serveurs chez Inria, tant pour des raisons de sécurité évidente (les données sont toutes au même endroit) que pour les risques humains ou administratifs (la disparition éventuelle de l'Inria ne devant pas mettre en péril le projet).

L'objectif est de trouver des partenaires disposés à fournir à Software Heritage un hébergement qui permettra à Software Heritage de s'émanciper d'ici quelques années de l'Inria.

2.3 Architecture logicielle

Les trois objectifs de Software Heritage conduisent beaucoup le développement du projet. Cette influence est visible dans les catégories de modules du projet.

2.3.1 Loaders (collect)

Lorsqu'un code source à été collecté, il peut s'agir d'un projet encore en développement, qui est donc en train d'évoluer. Le but des loaders est de parcourir à intervalle régulier des sources pour vérifier si le code source qu'elles contiennent à été mis à jour.

Par exemple, les dépôts git sur Github sont vérifiés à un intervalle qui dépend de leur fréquence de mise à jour : à chaque passage, si le dépôt n'a pas été mis à jour, la durée avant la prochaine vérification est multipliée par deux. A l'inverse, s'il a été mis à jour, la durée est divisée par deux.

Le rôle des loaders est donc de vérifier ces sources, et lorsqu'elles ont subi des modifications, de télécharger les nouvelles informations et de les normaliser afin de les mettre dans la base de donnée et le système de fichiers de Software Heritage.

2.3.2 Storage et Base de Données (preserve)

Le stockage de Software Heritage est conçu pour conserver de manière pérenne les fichiers sauvegardés. Lorsque les données ont été téléchargées par les loaders et normalisés afin d'être intégrés dans le modèle de Software Heritage, les modules `swh.storage` et `swh.objstorage` sont en charge d'assurer la conservation de l'intégrité des objets.

La base de donnée est sauvegardée régulièrement et les fichiers sources répliqués sur d'autres serveurs.

Vérification du contenu

L'objectif étant de conserver les contenus des fichiers, un tel système est fortement sensible aux *bit-rote*s, c'est à dire l'inversion d'un bit sur un disque qui peut subvenir occasionellement.

Comme les contenus enregistrés dans Software Heritage sont adressables par contenu, leur identifiant dépend de leur contenu. Si une erreur survient et corrompt leur contenu, l'erreur peut-être détéctée.

Afin de détécter ce genre d'erreurs, les stockages (Banco et Uffizi) hébergent aussi un *daemon* qui en arrière plan vérifie l'intégrité des contenus en séléctionant au hasard des objets à vérifier.

Les erreurs ayant une probabilité égale de se produire sur n'importe quel fichier (récent ou ancien), choisir au hasard la cible des vérifications donne aux fichiers une probabilité égale d'être vérifiés. L'heuristique est que, après une période de temps suffisement longue, tous les fichiers devraient avoir été vérifiés au moins une fois.

2.3.3 Web view (share)

La *web-view* est l'ensemble des éléments qui permettent d'accéder à Software Heritage, et donc au contenu sauvegardé, depuis Internet. Le Site Web de Software Heritage [1], ainsi que l'API publique [2] constituent cette vue.

2.4 Travail effectué

Le travail effectué lors de mon stage se limite au stockage des données et à leur préparation pour la distribution. Les chapitres suivants contiennent des informations plus détaillées sur ces deux aspects et sur les choix de développement.

1. `www.softwareheritage.org`
2. A la fin de mon stage (Septembre 2016), l'API n'est pas publique car l'infrastructure n'est pas suffisante pour répondre à une grande quantité de requêtes.

3. Archiver

Software Heritage manipule de grandes quantités de données qui doivent être répliquées sur plusieurs serveurs. Le but de l'archiver est de gérer la synchronisation des noeuds de stockage, afin que l'ensemble du réseau contienne toujours au moins N copies de chaque fichier source.

3.1 Motivations

Se référer aux annexes A.1 p.37 pour les spécifications de la première version et A.2 p.40 pour celles de la seconde.

Bien que les deux aient été développées entièrement, cette section ne décrit que la seconde car leur fonctionnement est très similaire.

3.2 Fonctionnement

L'archiver est constitué de trois classe qui orchestrent la copie des fichiers à partir des informations de copie de la base de donnée

3.2.1 Base de données

Dans la base de donnée de l'archiver de Software Heritage (Voir 3.1 p.15), la table `content_archive` contient pour chaque objet référencé un objet json de la forme suivante :

```
1   {
2       "node name": {
3           "status": "<status>",
4           "mtime": "<last update time>"
```

```
5         }
6     }
```

Il s'agit d'une dénormalisation de la table représentée par `content_status` sur le schéma 3.1 p.15.

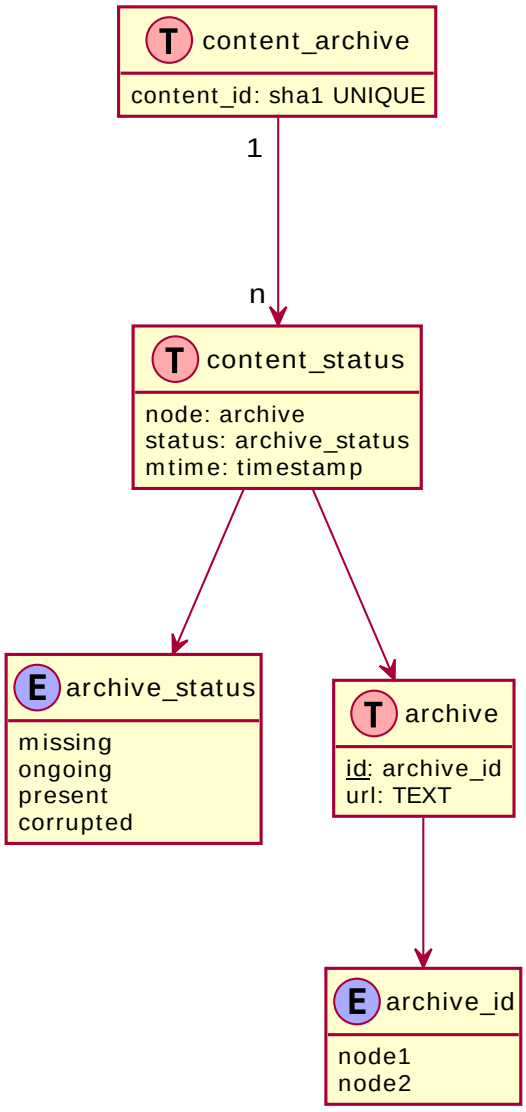

FIGURE 3.1 – Schéma de la base de donnée de l'archiver

Si un noeud est connu de l'archiver (contenu dans sa base de donnée) mais n'est pas renseigné pour un contenu, le contenu est considéré absent sur ce noeud.

3.2.2 ArchiverDirector

Le directeur (`ArchiverDirector`) a pour rôle d'itérer sur les objets référencés par la base de donnée et de créer des paquets avec ceux qui sont susceptibles de nécessiter une copie supplémentaire.

Sont *susceptibles* de nécessiter une copie supplémentaire les objets qui ont moins de noeuds sur lequel ils sont présent (au sens strict, uniquement ceux pour qui le champ `archive_status` est `'present'`) que requis par la configuration de l'archiver.

Ces contenus nécessitant d'être archivés sont assemblés par lot et ces lots mis dans une file d'attente où ils seront consommés par des workers.

3.2.3 ArchiverWorker

Lorsqu'un worker[^worker-archiver] consomme un des lots mis dans la file par le director, il instancie un `ArchiverWorker` dont le rôle est de parcourir le lot pour préparer la copie.

Pour chaque objet que le director a sélectionné précédemment, le worker vérifie qu'il est toujours nécessaire de le copier. Le worker s'assure donc que l'object n'a toujours pas le nombre requis de copies, en prenant en compte plus finement le statut `'ongoing'`

Si un objet est `'ongoing'` mais que le délai depuis la dernière mise à jour du status (calculé avec le champ `'mtime'`) est inférieur au délai maximum autorisé par le worker, une copie est déjà en cours ; on considère qu'elle va réussir et que le contenu sera présent sur le noeud, il est donc considéré comme tel lorsque le worker compte le nombre de copies existantes pour savoir s'il conserve cet objet.

A l'inverse, si le délai est écoulé, la copie est considérée comme un échec et le noeud n'est pas considéré comme contenant l'objet.

Pour chaque contenu qui est toujours candidat à être copié, le worker choisit au hasard un noeud source parmi ceux qui contiennent déjà l'objet, et un noeud destination parmi ceux sur lequel il est absent. Si plus d'une copie est requise, le worker choisit autant de couples (source, destination) que nécessaire.

Pour chaque couple (source, destination), le worker regroupe les objets qui sont associé au couple. Il effectue alors pour chaque contenu un contrôle d'intégrité de l'objet sur son noeud source. Si une erreur est détectée (objet absent ou contenu corrompu), il met à jour la base de donnée avec le nouveau status (`'missing'` ou `'corrupted'`), logue l'erreur, et retire l'objet de sa liste.

Pour les objets restants, le worker met à jour leur entrée dans la base de donnée pour indiquer que leur copie est en cours avec le status `'ongoing'`, puis il instancie localement un `ArchiverCopier` et le fait effectuer la copie avant de passer au couple de noeuds suivant.

3.2.4 ArchiverCopier

L'`ArchiverCopier` est initialisé avec une source, une destination, et une liste d'objets qui ont été sélectionnés par le worker pour être copiés de l'un à l'autre.

Le copier ouvre la connexion avec les noeuds, et pour chaque objet, fait une requête pour le récupérer depuis la source, puis l'ajoute à la destination.

A l'issue de la copie, si tout s'est bien passé, la base de donnée est mise à jour pour refléter la présence des objets nouvellement copiés. Si une erreur s'est produite en revanche, tout le lot du copier est annulé. Leur statut `'ongoing'` expirera et ils seront de nouveau planifiés pour une copie.

4. Object Storage

Lorsque les fichiers source sont capturés par un loader (voir 2.3.1 p.12), ils sont enregistrés dans l'**object storage** dont le but est de conserver les fichiers et d'en restituer leur contenu à la demande.

4.1 Principe et fonctionnement

L'object storage de Software Heritage est un *blob-storage* clef-valeur dans lequel chaque *blob* est référencé par son sha1 (voir 2.1 p.10).

Initialement, l'object storage était une classe entièrement dépendante du storage, et n'existait pas sans elle (voir figure 4.1 p.19). Au cours du développement, l'object storage a pris de plus en plus d'importance et d'indépendance jusqu'à être déplacé dans son propre package python (depuis `swh.storage.objstorage` vers `swh.objstorage`.

4.2 Les modifications

L'object storage est le composant le plus au coeur de Software Heritage, car c'est lui qui est en charge de stocker les fichiers sources bruts, dont les méta-données sont dans la base de donnée. Pour cette raison, c'est un composant qui est appelé à évoluer souvent pour devenir de plus en plus riche lorsque de nouvelles fonctionnalités sont nécessaires.

Cette section retrace les modifications effectuées sur l'object storage au cours de mon stage, ainsi que les raisons qui ont motivé ces changements.

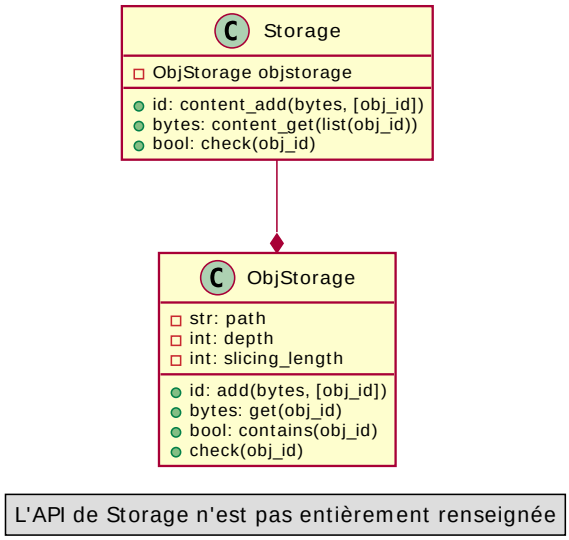

FIGURE 4.1 – Architecture initiale du Storage

4.2.1 RemoteObjStorage API

Créer l'archiver (voir 3 p.14) nécessite d'avoir sur la machine de sauvegarde un moyen de recevoir les objets copiés et de les stocker de la même manière que sur la machine source : un serveur http qui puisse recevoir les requêtes nécessaires et les déléguer à un *blob-storage*.

Comme les données sauvegardées n'ont pas besoin d'être manipulées mais uniquement présentes, déployer un serveur `Storage` (qui possède déjà une version remote http) implique une surcouche contingente.

Afin de remplir les objectifs de l'archiver tout en étant efficace, il a fallu ajouter un serveur http pour l'object storage. Le schéma 4.2 p.20 montre le fonctionnement de cette nouvelle API.

4.2.2 Interface ObjStorage

L'introduction de ce `RemoteObjStorage` (voir section précédente 4.2.1 p.19) qui possède les mêmes fonctionnalités et une API similaire à l'`ObjStorage`, induit la nécessité d'une interface commune afin que les modifications d'API n'empêchent pas le duck typing de fonctionner correctement et les deux types d'object storage d'être interchangés de manière invisible.

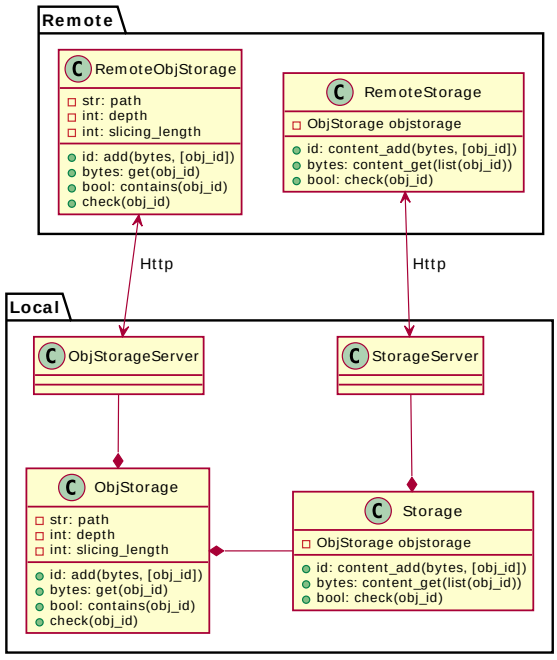

FIGURE 4.2 – API http pour ObjStorage

On introduit donc une interface dont vont hériter les deux classes d'object storage, et le code initial de `ObjStorage` est déplacé dans la classe `PathSlicingObjStorage` (voir schéma 4.3 p.21).

En python, le duck-typing rend inutile la présence d'interfaces puisque l'existence des méthodes est vérifiée en runtime. En réalité, les interfaces sont des classes abstraites dont toutes les méthodes lèvent l'exception `NotImplementedError`. Si une implémentation de la super classe ne respecte pas son contrat, l'erreur est levée et explique l'absence d'implantation de la méthode.

```
1    class ObjStorage():
2        ...
3        def get(self, obj_id):
4            """ Retrieve the content referenced by the given id. """
5            raise NotImplementedError('Class %s must implement "get"
                 method' % type(self))
6        ...
```

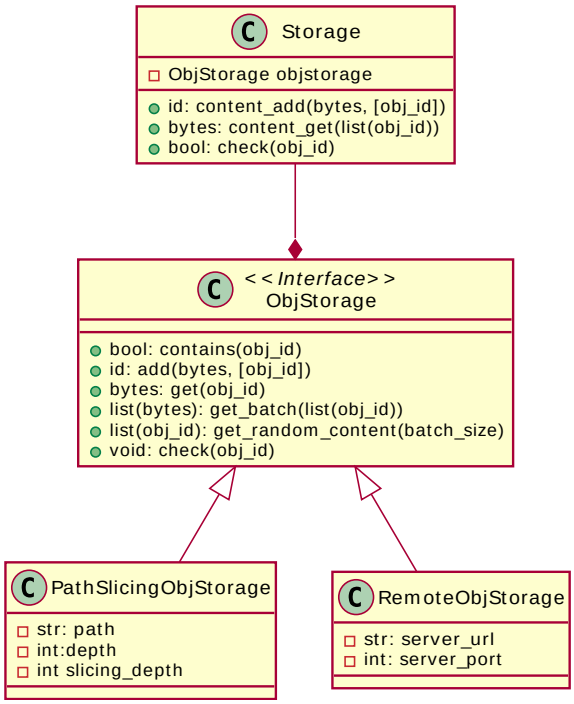

FIGURE 4.3 – Ajout de l'interface ObjStorage

Toutefois, cette solution seule n'est pas suffisante. L'erreur est détectée au cours de l'exécution et n'apporte pas la même sécurité qu'une interface dans un langage compilé.

Python, depuis la version 2.6 contient un module `abc` qui fournit des fonctionnalités utilisées pour faire des classes de base abstraites (Abstract Base Class). Ces classes permettent de forcer l'implantation des méthodes dans les classes filles, sans quoi une exception est levée à la création de l'objet de la classe fille. Cela permet de capturer cette erreur plus tôt et de ne pas la manquer, si la couverture des tests ne permettait pas de détecter l'erreur.

```
1    import abc
2
3    class ObjStorage(metaclass=abc.ABCMeta):
4        ...
5        @abc.abstractmethod
6        def get(self, obj_id):
7            """ Retrieve the content referenced by the given id. """
```

```
8             raise NotImplementedError('Class %s must implement "get"
                  method' % type(self))
9         ...
```

La méthode conserve toutefois son exception `NotImplementedError` dans le cas ou la classe fille redéfinit la méthode mais fait appel à la méthode de la classe parent.

```
1     class PathSlicingObjStorage(ObjStorage):
2         def get(self, obj_id):
3             super().get(obj_id)
```

4.2.3 PathSlicingObjStorage paramétré

La classe `PathSlicingObjStorage` est l'implantation de `ObjStorage` qui effectue concrètement le stockage des fichiers sur le disque (voir 2.1 p.10).

Lors de l'ajout d'un fichier à l'object storage, le sha1 de son contenu est calculé et lui sert d'identifiant. Cet identifiant définit l'arborescence dans laquelle le fichier va être stocké sur le disque.
Par exemple, un fichier `foo.txt` dont le sha1 est `abcdef1234` est stocké sur le disque sous `<storage root>/ab/cd/ef/abcdef1234`.

La création d'une telle arborescence permet de limiter le nombre de fichiers dans un même répertoire, nombre qui à notre échelle est mal supporté par le système de fichier XFS.

Cependant, pour accéder à un fichier, il est nécessaire d'ouvrir trois répertoires, c'est à dire accéder à trois inodes et passer par autant d'indirections. En passant à un système de fichier comme Ext4 pour qui le nombre de fichiers dans un répertoire n'est pas un facteur de ralentissement, il est possible de créer un stockage sur disque avec moins de découpe et donc moins de profondeur.

Seulement, la place disponible sur le stockage ne permet pas de faire une migration *en place* de tous les fichiers en une seule fois. Il est donc nécessaire de faire le déplacement partition par partition, au nombre de 16. Cette transition lente nécessite d'avoir simultanément plusieurs object storage dont la manière de découper l'identifiant diffère.

La classe `PathSlicingObjStorage` à donc été modifiée de manière à ce que la manière dont le sha1 est découpé soit passé comme un argument.

Au lieu de faire des découpes régulières, le `PathSlicingObjStorage` prend un argument une chaîne de caractère dont le contenu définit la taille des découpes. Cette syntaxe est basée sur le *slice* des iterables de python.

Par exemple, la chaîne `"0:2/2:4/4:6"` correspond à la découpe initiale de l'identifiant, où le premier répertoire est nommé en fonction des deux premiers caractères (de 0 jusqu'à 2 exclu), puis des deux suivants (de 2 jusqu'à 4 exclu), et ainsi de suite.

La découpe choisie pour le nouveau stockage sur Ext4 est `"0:1/0:5/"`, où le premier caractère ne représente que les points de montages virtuels des seize partitions.

Cet argument est donc parsé par le `PathSlicingObjStorage` et produit des informations pour la découpe de l'identifiant.

```
1   class PathSlicingObjStorage(ObjStorage):
2       def __init__(self, root, slicing):
3           # La chaine est parsee de telle sorte a produire
4           # une liste de couple (debut, fin) pour chaque
5           # partie.
6           self.root = root
7           self.bounds = [
8               slice(*map(int, sbounds.split(':')))
9               for sbounds in slicing.split('/')
10              if sbounds
11          ]
```

```
1       def _obj_dir(self, hex_obj_id):
2           """ Compute the path of an object
3           Args:
4               hex_obj_id (str): hex representation of the object id
5           Returns:
6               path to the directory where the content should be.
7           """
8           # Ces couples sont utilises pour former des morceaux
9           # d'identifiant qui sont assembles pour former un path
10          slices = [hex_obj_id[bound] for bound in self.bounds]
11          return os.path.join(self.root, *slices)
```

Ainsi, il est possible d'avoir plusieurs object storage qui ont une manière de stocker les fichiers différentes, et d'effectuer une transition partition par partition afin de ne pas dupliquer la totalité du contenu avant d'effacer l'ancienne version.

Lorsqu'une partition est migrée, et les deux object storage mis en place, si un contenu est ajouté qui correspond à une partition migrée, il doit être ajouté au nouvel object storage, mais pas à l'ancien. De même, lorsqu'un contenu est demandé, il doit être récupéré peu importe sur quel stockage il est.

4.2.4 ObjStorage demultiplexer

Le rôle d'un demultiplexer est de recevoir des requêtes, et de les répartir à ses composants. En demultiplexant les fonctions de `ObjStorage` vers les deux instances de `PathSlicingObjStorage`, il devient possible d'avoir de manière invisible les deux object storage fonctionnant en parallèle, comme illustré sur le schéma 4.4 p.24.

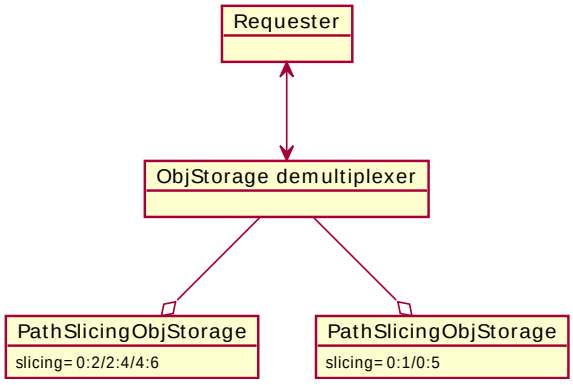

FIGURE 4.4 – Exemple de répartition des requêtes

On introduit donc une nouvelle classe héritant de `ObjStorage` dont le rôle est de dupliquer les requêtes et de les déléguer à plusieurs object storages (voir schéma 4.5 p.24)

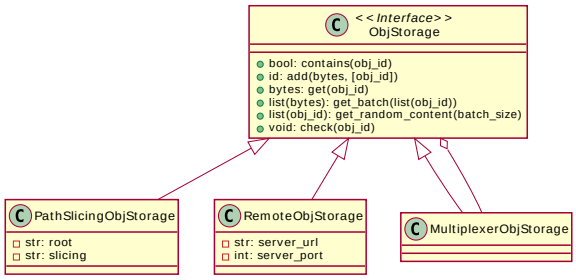

FIGURE 4.5 – Ajout du Multiplexer dans la hiérarchie ObjStorage

En revanche, lorsqu'un contenu est ajouté et qu'il doit être placé dans une partition qui a déjà été migrée, il ne faut l'ajouter que dans la nouvelle version de l'object storage. Il est donc nécessaire de pouvoir filtrer les accès aux storages. A cette fin, on ajoute un nouvel intermédiaire entre le multiplexer et l'object storage qui est capable de filtrer les appels de l'API et d'empêcher certaines actions d'être effectuées sur les object storages filtrés.

La classe `FilterObjStorage` est donc une classe qui sert de proxy pour un object storage, qui hérite de `ObjStorage` pour en conserver l'API et qui autorise certaines actions en bloquant les autres. Les méthodes d'un `FilterObjStorage` ne font aucune action lorsque celle demandée n'est pas autorisée, ou délèguent à un champ de type `ObjStorage` les actions à effectuer.

```
1     # This is an example that does not cover the full API
2     class FilterObjStorage(ObjStorage):
3         """ Base class of filter """
4         def __init__(self, objstorage):
5             self.delegate = objstorage
6
7         # Functions do nothing by default
8         def get(self, obj_id):
9             return
10
11        def add(self, bytes, obj_id):
12            return
13
14    class ReadObjStorage(FilterObjStorage):
15        """ Filter that only allows read operations. """
16        def get(self, obj_id):
17            return self.storage.get(obj_id)
18
19        # self.add is not defined so it does nothing.
```

Comme un filtre bloque certaines actions puis délègue à un `ObjStorage`, il est possible de chaîner les filtres afin de créer une chaine de délégation qui permet d'appliquer plusieurs filtres et d'avoir un résultat complexe de manière invisible.

Les types de filtre crées sont les suivants (voir schéma 4.6 p.26).

— **ReadObjStorageFilter** n'autorise que les opérations de lecture, faisant de l'object storage un stockage en lecture seule.

— **PrefixIdObjStorageFilter** autorise les opérations sur des objets dont l'identifiant (le sha1) contient un préfixe donné en argument.
— **RegexIdObjStorageFilter** autorise les opérations lorsque l'identifiant de l'objet correspond à l'expression régulière passée en argument.

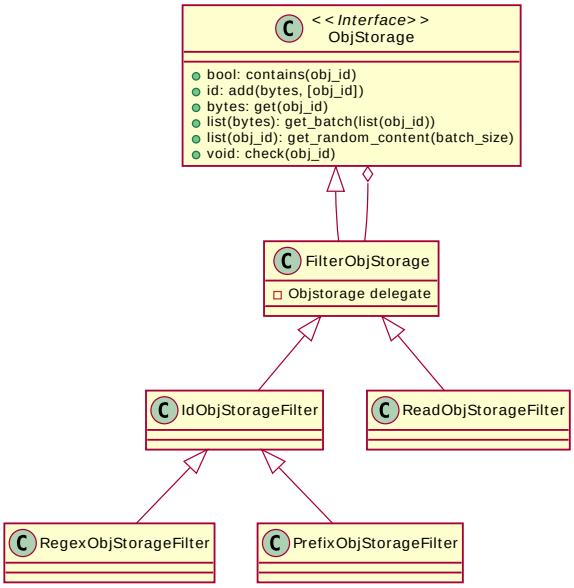

FIGURE 4.6 – Architecture du filtre d'object storage

Ces trois types de filtre permettent de faire fonctionner notre scénario d'utilisation, comme illustré sur le schéma 4.7 p.27.

4.2.5 Accès Cloud

Le but de Software Heritage nécessite d'avoir plusieurs serveur qui peuvent servir de sauvegarde, ou d'un autre point d'accès. Pour cette raison, l'utilisation de clouds distants (voir 2.2.2 p.11) représente une addition importante.

Afin de normaliser les accès aux object storages, il n'est pas envisageable d'utiliser directement l'API des clouds pour y stocker des fichiers, mais il est préférable d'utiliser celle de l'object storage. Une nouvelle implantation de `ObjStorage` sert d'intermédiaire pour accéder aux clouds, en utilisant la librairie Apache Libcloud qui permet de plus de faire abstraction du cloud utilisé en masquant les différentes API Http, comme indiqué sur le schéma 4.8 p.28.

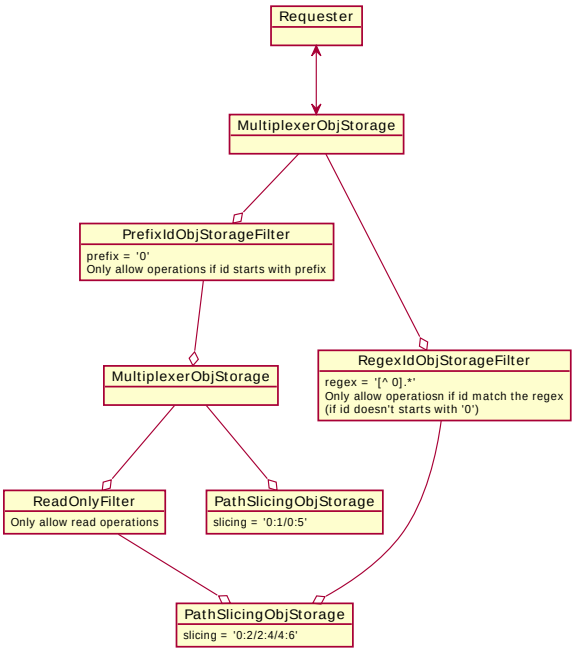

FIGURE 4.7 – Fonctionnement de l'object storage filter

4.2.6 Extraction du package

Avec tous ces ajouts, le module `swh.storage.objstorage` s'étoffe et devient suffisamment important pour justifier qu'il soit contenu dans son propre package sous `swh.objstorage`.

Extraire le code de l'object storage depuis `swh.storage.objstorage` vers `swh.objstorage` se fait vers un dépôt différent (`swh-environment/swh-objstorage`). Ce dépôt doit contenir tous les commits relatifs à l'object storage afin d'en conserver tout l'historique de développement.

— Cloner le dépôt initial
— Utiliser `git --subdirectory-filter` pour ne conserver que le contenu de `swh/storage/objstorage`
— Changer l'url remote du dépôt pour celle d'un nouveau dépôt
— Publier les modifications sur le nouveau dépôt qui est initialisé avec uniquement les fichiers souhaités.

```
1  git clone
       https://forge.softwareheritage.org/diffusion/DOBJS/swh-objstorage.git
```

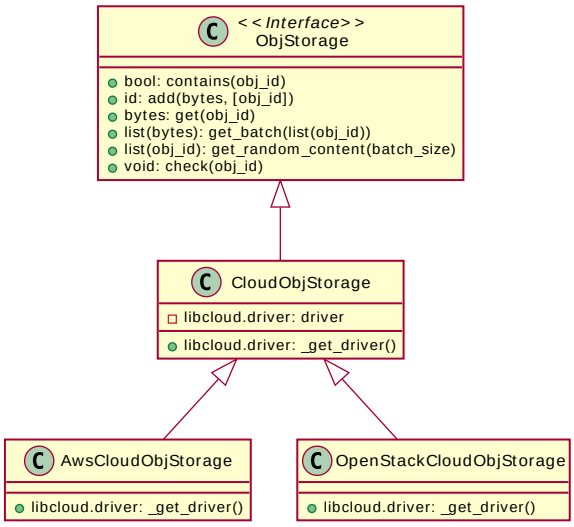

FIGURE 4.8 – Ajout d'un object storage se connectant à un cloud

```
cd swh-storage
git filter-branch --prune-empty --subdirectory-filter objstorage
git remote set-url remote
    https://forge.softwareheritage.org/diffusion/DOBJS/swh-objstorage.git
# Effectuer les autres operations pour agencer le depot ici.
git push -u origin # Publier le nouveau depot
```

Après ces opérations, le nouveau dépôt `swh-objstorage` est initialisé avec le contenu requis.

4.2.7 Changement de l'API

Au cours du développement d'autres fonctionnalités, le besoin s'est fait sentir de récupérer des contenus par lot. Dans le cas d'un object storage local, itérer sur les requêtes suffit. Mais pour les versions remote de l'object storage, il s'agit d'une grande perte d'efficacité car les requêtes Http sont multipliées.

Afin de permettre des requêtes par lot, l'API de `ObjStorage` est enrichie de la méthode `list(bytes): get_batch(list(obj_id))`. Celle-ci possède une implantation par défaut dans `ObjStorage` (qui devient conceptuellement une classe abstraite plutôt qu'une interface, cette nuance n'existant pas en python) qui suffit pour la plupart des cas d'utilisation.

```
    get_batch(self, obj_ids):
```

```
        for obj_id in obj_ids:
            try:
                yield self.get(obj_id)
            except ObjNotFoundError:
                yield None
```

Calquée sur `content_get` de la classe `Storage`, cette méthode est un générateur qui produit les contenus des storages ou `None` si l'id n'existe pas dans l'object storage.

Cette méthode est redéfinie dans `RemoteObjStorage` afin de faire transiter la requête en une seule fois vers le serveur.

4.2.8 Initialisation des object storages

Toute l'architecture basée sur `ObjStorage` ne contient qu'une seule classe ayant réellement un impact sur le disque, les autres ne faisant que déléguer ces opérations.

Les classes qui utilisent un object storage (`Storage`, archiver, ...) sont toutes configurées par un fichier externe. Ce fichier doit contenir des informations sur l'object storage utilisé. Cette configuration est complexe et nécessite un langage de description structuré. Le fait que les object storages délèguent à un autre object storage les actions permet d'écrire le fichier de configuration comme un arbre.

Par exemple, la configuration suivante donne le résultat du schéma 4.4 p.24.

```
 storages:
     cls: multiplexer
     args:
         storages:
             - cls: pathslicing
               args:
                   root: /srv/swh/objects
                   slicing: 0:1/0:5
             - cls: pathslicing
               args:
                   root: /srv/swh/objects2
                   slicing: 0:2/2:4/4:6
```

L'architecture issue de `ObjStorage` est conçue pour être facilement extensible, en contenant un grand nombre de classes abstraites intermédiaires, permettant à chaque type de storage d'être spécialisé avec peu d'ajout de code. Le module permettant

de parser ce fichier de configuration et de créer un `ObjStorage` doit donc aussi être facile à étendre afin qu'un nouvel `ObjStorage` nécessite peu de modifications pour être instancié.

La configuration d'un object storage est de la forme :

```
cls: <class identifier>
args:
    <class's __init__ args1 name>: <value>
    ...
    <class's __init__ argsN name>: <value>
```

Ce qui permet, à l'aide d'une simple table `identifiant -> classe` d'obtenir la classe python à partir de l'identifiant, et d'appeler son constructeur avec les arguments.

```
_STORAGE_CLASSES = {
    'pathslicing': PathSlicingObjStorage,
    'remote': RemoteObjStorage,
}


def get_objstorage(cls, args):
    """ Create an ObjStorage using the given implementation class.

    Args:
        cls (str): objstorage class unique key contained in the
            _STORAGE_CLASSES dict.
        args (dict): arguments for the required class of objstorage
            that must match exactly the one in the `__init__` method
                of the class.
    Returns:
        subclass of ObjStorage that match the given `cls` argument.
    Raises:
        ValueError: if the given storage class is not a valid
            objstorage key.
    """
    try:
        return _STORAGE_CLASSES[cls](**args)
    except KeyError:
        raise ValueError('Storage class %s does not exists' % cls)
```

5. SWH Vault

Le but à long terme de Software Heritage est de mettre à disposition publiquement les objets archivés. Ces objets peuvent être des fichiers sources simples, mais aussi des tarballs, ou des dépôts.

5.1 Fonctionnement

Afin de mettre à disposition au téléchargement un dépôt, Software Heritage doit être capable de reformer à partir des données normalisées de sa base de donnée un *bundle* téléchargeable. Le module qui permet d'effectuer cette opération est le **Vault Software Heritage** (Voir spécification en annexe A.3 p.44).

En fonction du type de contenu requis, le *bundle* doit contenir des données différentes et être produit à l'aide de son propre cooker.

— **Un répertoire** est compilé simplement en ajoutant les fichiers contenus dans une archive compressée.
— **Une tarball** peut être compilée simplement sous la forme d'un répertoire compressé.
— la requêtes pour **un dépôt** s'effectue en demandant un commit particulier du dépôt. Le *bundle* produit doit contenir tout l'historique de commits du dépôt jusqu'à celui demandé. Une telle requêtes demande un travail de compilation plus conséquent.

Étant donné la durée nécessaire à produire un *bundle*, les requêtes ne peuvent pas résulter en une attente active de la réponse.

5.2 API de manipulation des bundles

Le Vault met à disposition une API Http qui permet d'interagir avec les composants chargés de la gestion des bundles.

5.2.1 Requête d'un bundle

L'API du **Vault** permet de faire la requête pour un *bundle*. Lorsque cette requête est reçue, elle est ajoutée dans une file d'attente et retourne une réponse `201 CREATED`. Cette file d'attente est consommée par des workers qui produisent les *bundles*.

5.2.2 Fabrication d'un bundle

Les modules chargés de fabriquer les *bundles* sont les **cookers**, qui consomment les requêtes de la file d'attente et produisent les bundles associés, qui sont ajoutés dans un cache contenant les bundles disponibles au téléchargement.

5.2.3 Téléchargement d'un bundle

Lorsqu'un *bundle* est demandé au téléchargement, s'il n'a pas déjà été compilé par un cooker, l'API retourne une erreur `404 NOT FOUND`. Sinon, son contenu est envoyé.

6. Methodologie de travail

6.1 Prises de décisions

Software Heritage est un projet libre. Toutes les discussions relatives à son développement sont publiquement accessibles.

Le principal moyen de communication immédiat est le canal irc `#swh-devel` hébergé sur freenode.net. Les discussions relatives au développement, bien qu'il soit plus rapide de les faire à l'oral permettent d'être transparent et d'intégrer plus facilement de potentiels nouveaux contributeurs. Les discussions plus conséquentes nécéssitant des interventions élaborées passent par la liste de diffusion `swh-devel@inria.fr`.

6.2 Encadrement

Au cours de mon stage, je n'ai pas uniquement été encadré par mon tuteur Stefano, mais par toute l'équipe de développement, très attentive à mes questions.

Des réunions informelles sont organisées à chaque fois qu'une nouvelle tâche m'est assignée, au cours de laquelle mon tuteur m'explique la tâche à effectuer et éventuellement les algorithmes à employer.

Pendant le développement, des entrevues sont effectuées à intervalles régulier pour que je puisse faire part de mon avancement.

Occasionellement lors des étapes importantes du projet, nous participons à des réunions d'équipe où chacun d'entre nous explique la tâche sur laquelle il travaille et ses avancées, afin que les autres membres puissent prendre connaissance du travail de chacun, et apporter un regard différent sur les méthodes utilisées.

6.3 Suivi des tâches

Lorsque je dois effectuer une tâche, elle est créé (par moi ou la personne l'assignant) sur la forge de Software Heritage [1] et m'est assignée.

Il est alors de ma responsabilité de créer les sous-tâches que j'estime nécéssaire afin de fragmenter mon travail et de permettre au membre de l'équipe m'encadrant sur cette tâche de correctement suivre mes avancées.

6.4 Revue du code

Chaque modification que j'apporte au code est envoyée vers la forge, puis reviewée via une interface web par un membre permanent de l'équipe, qui peut commenter (globalement ou sur une ligne spécifique) ma contribution.

A l'issue de ces commentaires, ma proposition est acceptée ou rejetée. Si elle est acceptée, le `git push` se débloque pour cet arbre de commit, me permettant de placer mon code sur le dépôt.

En cas de rejet, je dois mettre à jour ma diff et soumettre à nouveau ma proposition en prennant en compte les remarques et commentaires de mon reviewer.

1. `https://forge.softwareheritage.org`

Conclusion

Collecter le code source que nous écrivons aujourd'hui c'est, de même que nous conservons déjà de notre mieux les livres, un moyen de s'assurer que notre héritage digital ne disparaît pas ; et que notre connaissance comme nos logiciels, qui se construisent sur la base de ceux que nous possédons déjà, ne peuvent pas perdre leurs preuve ou leurs racines.

Software Heritage n'est pas qu'un projet libre. C'est une initiative ambitieuse, portée par un institut public et dont l'objectif est de devenir à terme une organisation internationale indépendante.

Le stage effectué à la fin du master est une conclusion aux années d'études effectuées à l'université. C'est l'occasion de mettre en application ses compétences dans une situation réelle, au sein d'une véritable équipe et sur un projet concret, afin d'acquérir une expérience différente.

Travailler sur projet tel que Software Heritage constitue en soi une opportunité enrichissante sur le plan technique. Avoir rejoint l'équipe pendant la période de l'annonce publique du projet n'a fait que rendre encore plus passionnante cette expérience.

Être présent pendant ce lancement m'a permis de découvrir depuis l'intérieur les mécanismes mis en place pour rendre public le développement : transparence des décisions de design, outils pour faciliter l'intégration de nouveaux développeurs et communication avec la communauté.

Réaliser mon stage au sein d'une équipe engagée et passionnée par les problématiques du logiciel libre me permet de connaître maintenant beaucoup mieux le fonctionnement d'un tel projet et m'a réellement passionné pour Software Heritage, pour lequel je souhaite continuer à contribuer.

Software Heritage, en collectant une telle quantité de code, donne la possibilité d'effectuer des analyses à une échelle nouvelle, et ouvre la porte à un nouvel ordre de grandeur de recherche en génie logiciel. Cette perspective m'a fortement intéressé au cours de mon stage et me motive désormais à poursuivre cette voie vers une thèse dans cette problématique.

Remerciements

Ce rapport est la conclusion de mon stage, et c'est en synthétisant les mois passés au sein de Software Heritage que je peux prendre la pleine mesure de ce qui m'a été apporté par une équipe dévouée, pédagogue et attentive. Je souhaite donc remercier très chaleureusement toute l'équipe de Software Heritage pour son encadrement, sa présence, et l'environnement de travail agréable et stimulant qu'ils ont crée.

Je souhaite remercier les membres de l'équipe avec qui j'ai travaillé, Nicolas Dandrimont et Antoine Dumont, dont les revues de code m'ont permis de profiter de leur expérience continuellement pendant cinq mois, en commentant systématiquement mon code et me fournissant ainsi des informations basées sur des exemples concrets.

Je tiens à remercier très chaleureusement Roberto Di Cosmo pour avoir travaillé plus que je ne peux l'imaginer à combattre les aléas administratifs mais avoir conservé en toutes circonstances une écoute attentive et bienveillante. Ainsi que Stefano Zacchiroli pour sa présence, sa passion et sa pédagogie dont j'ai pu profiter tout au long de mon stage.

A. Annexes

A.1 Software Heritage Archiver (Version 1)

The Software Heritage (SWH) Archiver is responsible for backing up SWH objects as to reduce the risk of losing them.

Currently, the archiver only deals with content objects (i.e., those referenced by the content table in the DB and stored in the SWH object storage). The database itself is lively replicated by other means.

A.1.1 Requirements

— **Master/slave architecture**

There is 1 master copy and 1 or more slave copies of each object. A retention policy specifies the minimum number of copies that are required to be "safe".

— **Append-only archival**

The archiver treats master as read-only storage and slaves as append-only storages. The archiver never deletes any object. If removals are needed, in either master or slaves, they will be dealt with by other means.

— **Asynchronous archival.**

Periodically (e.g., via cron), the archiver runs, produces a list of objects that need to be copied from master to slaves, and starts copying them over. Very likely, during any given archival run other new objects will be added to master ; it will be the responsibility of *future* archiver runs, and not the current one, to copy new objects over.

— **Integrity at archival time.**

Before archiving objects, the archiver performs suitable integrity checks on them. For instance, content objects are verified to ensure that they can be decompressed and that their content match their (checksum-based) unique identifiers. Corrupt objects will not be archived and suitable errors reporting about the corruption will be emitted.

Note that archival-time integrity checks are *not meant to replace periodic integrity checks* on both master and slave copies.

— **Parallel archival**

Once the list of objects to be archived has been identified, it SHOULD be possible to archive objects in parallel w.r.t. one another.

— **Persistent archival status**

The archiver maintains a mapping between objects and the locations where they are stored. Locations are the set {master, slave_1, ..., slave_n}.

Each pair is also associated to the following information :

— **status** : 3-state : *missing* (copy not present at destination), *ongoing* (copy to destination ongoing), *present* (copy present at destination)
— **mtime** : timestamp of last status change. This is either the destination archival time (when status=present), or the timestamp of the last archival request (status=ongoing) ; the timestamp is undefined when status=missing.

A.1.2 Architecture

The archiver is comprised of the following software components :

— archiver director
— archiver worker
— archiver copier

Archiver director

The archiver director is run periodically, e.g., via cron.

Runtime parameters :

— execution periodicity (external)
— retention policy
— archival max age
— archival batch size

At each execution the director :

1. for each object : retrieve its archival status
2. for each object that is in the master storage but has fewer copies than those requested by the retention policy :
3. if status=ongoing and mtime is not older than archival max age then continue to next object
4. check object integrity (e.g., with swh.storage.ObjStorage.check(obj_id))
5. mark object as needing archival
6. group objects in need of archival in batches of archival batch size
7. for each batch :
8. set status=ongoing and mtime=now() for each object in the batch
9. spawn an archive worker on the whole batch (e.g., submitting the relevant celery task)

Note that if an archiver worker task takes a long time (t > archival max age) to complete, it is possible for another task to be scheduled on the same batch, or an overlapping one.

Archiver worker

The archiver is executed on demand (e.g., by a celery worker) to archive a given set of objects.

Runtime parameters :

— objects to archive

At each execution a worker :

1. create empty map { destinations -> objects that need to be copied there }
2. for each object to archive :
3. retrieve current archive status for all destinations
4. update the map noting where the object needs to be copied
5. for each destination :
6. look up in the map objects that need to be copied there
7. copy all objects to destination using the copier
8. set status=present and mtime=now() for each copied object

Note that :

— In case multiple jobs where tasked to archive the same of overlapping objects, step (2.2) might decide that some/all objects of this batch no longer need to be archived to some/all destinations.

— Due to parallelism, it is also possible that the same objects will be copied over at the same time by multiple workers.

Archiver copier

The copier is run on demand by archiver workers, to transfer file batches from master to a given destination.

The copier transfers all files together with a single network connection. The copying process is atomic at the file granularity (i.e., individual files might be visible on the destination before *all* files have been transferred) and ensures that *concurrent transfer of the same files by multiple copier instances do not result in corrupted files*. Note that, due to this and the fact that timestamps are updated by the director, all files copied in the same batch will have the same mtime even though the actual file creation times on a given destination might differ.

As a first approximation, the copier can be implemented using rsync, but a dedicated protocol can be devised later. In the case of rsync, one should use –files-from to list the file to be copied. Rsync atomically renames files one-by-one during transfer ; so as long as –inplace is *not* used, concurrent rsync of the same files should not be a problem.

A.1.3 DB structure

Postgres SQL definitions for the archival status :

```
CREATE DOMAIN archive_id AS TEXT;

CREATE TABLE archives (
```

```
  id   archive_id PRIMARY KEY,
  url  TEXT
);

CREATE TYPE archive_status AS ENUM (
  'missing',
  'ongoing',
  'present'
);

CREATE TABLE content_archive (
  content_id  sha1 REFERENCES content(sha1),
  archive_id  archive_id REFERENCES archives(id),
  status      archive_status,
  mtime       timestamptz,
  PRIMARY KEY (content_id, archive_id)
);
```

A.2 Software Heritage Archiver (Version 2)

The Software Heritage (SWH) Archiver is responsible for backing up SWH objects as to reduce the risk of losing them.

Currently, the archiver only deals with content objects (i.e., those referenced by the content table in the DB and stored in the SWH object storage). The database itself is lively replicated by other means.

A.2.1 Requirements

— **Peer-to-peer topology**

Every storage involved in the archival process can be used as a source or a destination for the archival, depending on the blobs it contains. A retention policy specifies the minimum number of copies that are required to be "safe".

Althoug the servers are totally equals the coordination of which content should be copied and from where to where is centralized.

— **Append-only archival**

The archiver treats involved storages as append-only storages. The archiver never deletes any object. If removals are needed, they will be dealt with by other means.

— **Asynchronous archival.**

Periodically (e.g., via cron), the archiver runs, produces a list of objects that need to have more copies, and starts copying them over. The decision of which storages are choosen to be sources and destinations are not up to the storages themselves.

Very likely, during any given archival run, other new objects will be added to storages ; it will be the responsibility of *future* archiver runs, and not the current one, to copy new objects over if needed.

— **Integrity at archival time.**

Before archiving objects, the archiver performs suitable integrity checks on them. For instance, content objects are verified to ensure that they can be decompressed and that their content match their (checksum-based) unique identifiers. Corrupt objects will not be archived and suitable errors reporting about the corruption will be emitted.

Note that archival-time integrity checks are *not meant to replace periodic integrity checks*.

— **Parallel archival**

Once the list of objects to be archived has been identified, it SHOULD be possible to archive objects in parallel w.r.t. one another.

— **Persistent archival status**

The archiver maintains a mapping between objects and the locations where they are stored. Locations are the set {master, slave_1, ..., slave_n}.

Each pair is also associated to the following information :

— **status** : 3-state : *missing* (copy not present at destination), *ongoing* (copy to destination ongoing), *present* (copy present at destination), *corrupted* (content detected as corrupted during an archival).

— **mtime** : timestamp of last status change. This is either the destination archival time (when status=present), or the timestamp of the last archival request (status=ongoing) ; the timestamp is undefined when status=missing.

A.2.2 Architecture

The archiver is comprised of the following software components :

— archiver director

— archiver worker

— archiver copier

Archiver director

The archiver director is run periodically, e.g., via cron.

Runtime parameters :

— execution periodicity (external)

— retention policy

— archival max age

— archival batch size

At each execution the director :

1. for each object : retrieve its archival status
2. for each object that has fewer copies than those requested by the retention policy :
3. mark object as needing archival
4. group objects in need of archival in batches of `archival batch size`
5. for each batch :

6. spawn an archive worker on the whole batch (e.g., submitting the relevant celery task)

Archiver worker

The archiver is executed on demand (e.g., by a celery worker) to archive a given set of objects.

Runtime parameters :

— objects to archive
— archival policies (retention & archival max age)

At each execution a worker :

1. for each object in the batch
2. check that the object still need to be archived (#present copies < retention policy)
3. if an object has status=ongoing but the elapsed time from task submission is less than the *archival max age*, it count as present, as we assume that it will be copied in the futur. If the delay is elapsed, otherwise, is count as a missing copy.
4. for each object to archive :
5. retrieve current archive status for all destinations
6. create a map noting where the object is present and where it can be copied
7. Randomly choose couples (source, destination), where destinations are all differents, to make enough copies
8. for each (content, source, destination) :
9. Join the contents by key (source, destination) to have a map {(source, destination) -> [contents]}
10. for each (source, destination) -> contents
11. for each content in content, check its integrity on the source storage
    — if the object if corrupted or missing
        — update its status in the database
        — remove it from the current contents list
12. start the copy of the batces by launching for each transfer tuple a copier
    — if an error occured on one of the content that should have been valid, consider the whole batch as a failure.
13. set status=present and mtime=now for each successfully copied object

Note that :

— In case multiple jobs where tasked to archive the same of overlapping objects, step (1) might decide that some/all objects of this batch no longer need to be archived.
— Due to parallelism, it is possible that the same objects will be copied over at the same time by multiple workers. Also, the same object could end having more copies than the minimal number required.

Archiver copier

The copier is run on demand by archiver workers, to transfer file batches from a given source to a given destination.

The copier transfers files one by one. The copying process is atomic at the file granularity (i.e., individual files might be visible on the destination before *all* files have been transferred) and ensures that *concurrent transfer of the same files by multiple copier instances do not result in corrupted files*. Note that, due to this and the fact that timestamps are updated by the worker, all files copied in the same batch will have the same mtime even though the actual file creation times on a given destination might differ.

The copier is implemented using the ObjStorage API for the sources and destinations.

A.2.3 DB structure

Postgres SQL definitions for the archival status :

```
-- Those names are sample of archives server names
CREATE TYPE archive_id AS ENUM (
  'uffizi',
  'banco'
);

CREATE TABLE archive (
  id   archive_id PRIMARY KEY,
  url  TEXT
);

CREATE TYPE archive_status AS ENUM (
  'missing',
  'ongoing',
  'present',
  'corrupted'
);

CREATE TABLE content_archive (
  content_id sha1 unique,
  copies jsonb
);
```

Where the content_archive.copies field is of type jsonb and contains datas about the storages that contains (or not) the content represented by the sha1

```
{
```

```
    "$schema": "http://json-schema.org/schema#",
    "title": "Copies data",
    "description": "Data about the presence of a content into the storages",
    "type": "object",
    "Patternproperties": {
        "^[a-zA-Z1-9]+$": {
            "description": "archival status for the server named by key",
            "type": "object",
            "properties": {
                "status": {
                    "description": "Archival status on this copy",
                    "type": "string",
                    "enum": ["missing", "ongoing", "present", "corrupted"]
                },
                "mtime": {
                    "description": "Last time of the status update",
                    "type": "float"
                }
            }
        }
    },
    "additionalProperties": false
}
```

A.3 Software Heritage Vault

Software source code **objects**—e.g., individual source code files, tarballs, commits, tagged releases, etc.—are stored in the Software Heritage (SWH) Archive in fully deduplicated form. That allows direct access to individual artifacts but require some preparation, usually in the form of collecting and assembling multiple artifacts in a single **bundle**, when fast access to a set of related artifacts (e.g., the snapshot of a VCS repository, the archive corresponding to a Git commit, or a specific software release as a zip archive) is required.

The **Software Heritage Vault** is a cache of pre-built source code bundles which are assembled opportunistically retrieving objects from the Software Heritage Archive, can be accessed efficiently, and might be garbage collected after a long period of non-use.

A.3.1 Requirements

— **Shared cache**

The vault is a cache shared among the various origins that the SWH archive tracks. If the same bundle, originally coming from different origins, is requested, a single entry for it in the cache shall exist.

— **Efficient retrieval**

Where supported by the desired access protocol (e.g., HTTP) it should be possible for the vault to serve bundles efficiently (e.g., as static files served via HTTP, possibly further proxied/cached at that level). In particular, this rules out building bundles on the fly from the archive DB.

A.3.2 API

All URLs below are meant to be mounted at API root, which is currently at `https://archive.softwareheritage.org/api/1/`. Unless otherwise stated, all API endpoints respond on HTTP GET method.

A.3.3 Object identification

The vault stores bundles corresponding to different kinds of objects. The following object kinds are supported :

— directories
— revisions
— repository snapshots

The URL fragment `:objectkind/:objectid` is used throughout the vault API to fully identify vault objects. The syntax and meaning of :objectid for the different object kinds is detailed below.

Directories

— object kind : directory
— URL fragment : directory/ :sha1git

where :sha1git is the directory ID in the SWH data model.

Revisions

— object kind : revision
— URL fragment : revision/ :sha1git

where :sha1git is the revision ID in the SWH data model.

Repository snapshots

— object kind : snapshot
— URL fragment : snapshot/ :sha1git

where :sha1git is the snapshot ID in the SWH data model. (**TODO** repository snapshots don't exist yet as first-class citizens in the SWH data model ; see References below.)

A.3.4 Cooking

Bundles in the vault might be ready for retrieval or not. When they are not, they will need to be **cooked** before they can be retrieved. A cooked bundle will remain around until it expires ; at that point it will need to be cooked again before it can be

retrieved. Cooking is idempotent, and a no-op in between a previous cooking operation and expiration.

To cook a bundle :

— POST /vault/ :objectkind/ :objectid

Request body : **TODO** something here in a JSON payload that would allow notifying the user when the bundle is ready.

Response : 201 Created

A.3.5 Retrieval

— GET /vault/ :objectkind

(paginated) list of all bundles of a given kind available in the vault ; see Pagination. Note that, due to cache expiration, objects might disappear between listing and subsequent actions on them.

Examples :

— GET /vault/directory

— GET /vault/revision

— GET /vault/ :objectkind/ :objectid

Retrieve a specific bundle from the vault.

Response :

— 200 OK : bundle available ; response body is the bundle

— 404 Not Found : missing bundle ; client should request its preparation (see Cooking)

A.3.6 References

— Repository snapshot objects

— Amazon Web Services, API Reference for Amazon Glacier ; specifically Job Operations

A.3.7 TODO

— **TODO** pagination using HATEOAS

— **TODO** authorization : the cooking API should be somehow controlled to avoid obvious abuses (e.g., let's cache everything)

— **TODO** finalize repository snapshot proposal

A.4 Schéma relationnel

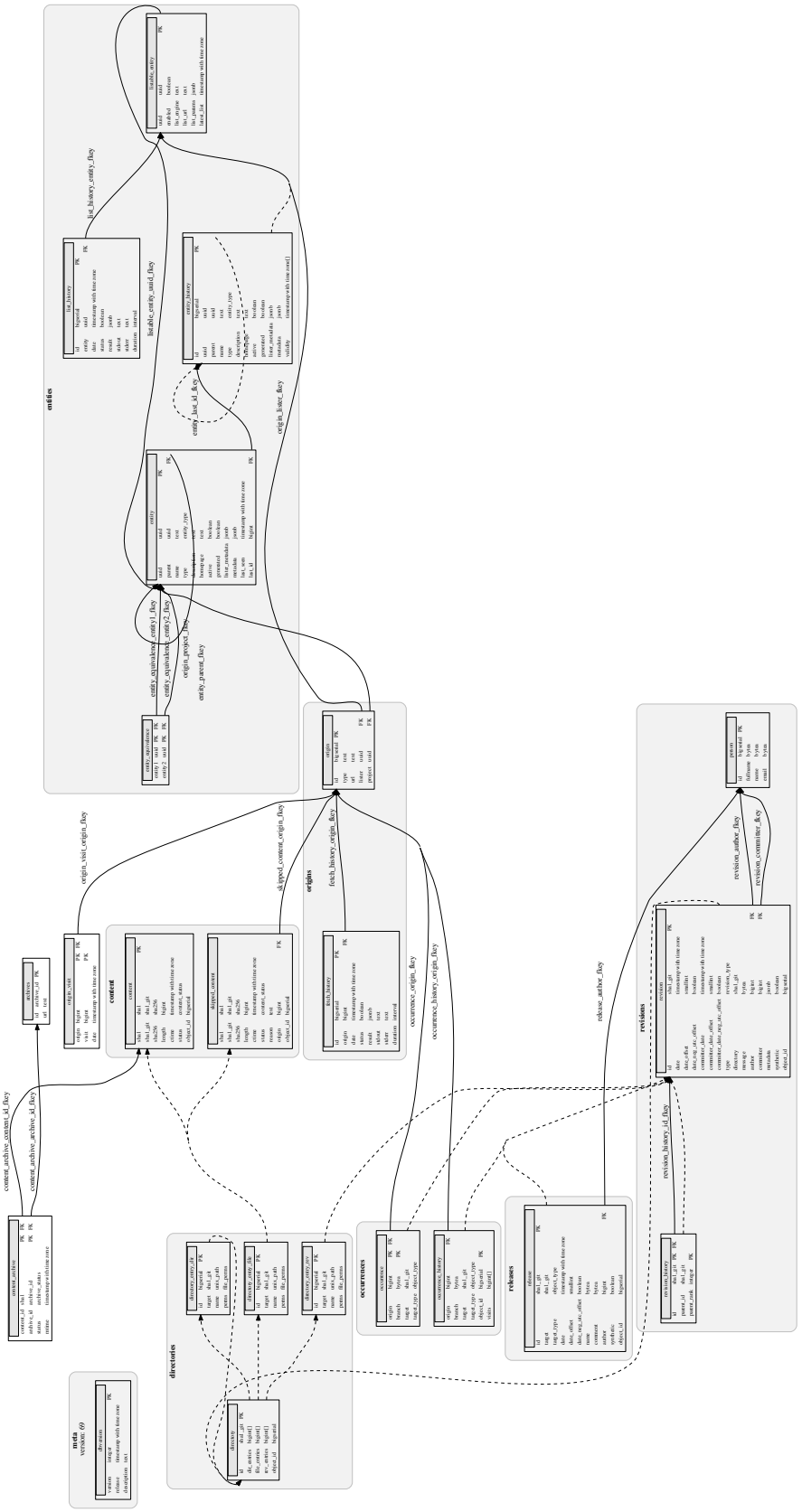

FIGURE A.1 – Schéma relationnel de la base de donnée Software Heritage