M1 Informatique AIGLE

HMIN201

M1 TER

TER : Software Heritage

Rapport Final

Groupe Bajonim

Bachar Rima,
bachar.rima@etu.umontpellier.fr
Joseph Saba,
joseph.saba@etu.umontpellier.fr
Tasnim Shaqura,
tasnim.shaqura@etu.umontpellier.fr

Encadrant :
Jessie Carbonnel

Responsable de l'UE :
Mattieu Lafourcade

27 mai 2019

Table des matières

Chapitre 1

Introduction

Les logiciels sont actuellement omniprésents dans tous les aspects de notre vie quotidienne ; ils constituent l'un des piliers de l'héritage humain et doivent être préservés contre toute suppression et tout endommagement. Archiver leurs codes source s'avère ainsi une tâche primordiale. En effet, le code source d'un logiciel constitue un artefact logiciel essentiel dans le domaine des connaisances scientifiques, culturelles, et techniques. D'autre part, le code source est facilement lisible et compréhensible par les humains, et peut être transformé en fichiers exécutables. À ce titre là, des plateformes ont déjà été proposées telles que `The Internet Archive` et `UNESCO Persist`. Toutefois, ces plateformes se concentraient plutôt sur la préservation des fichiers exécutables au lieu du code source [1][2].

1.1 Description de Software Heritage

`Software Heritage` est une initiative lancée par **INRIA** [1], soutenue par l'**UNESCO** et visant « la collecte, la conservation et le partage de code source de tous les logiciels publiquement accessibles depuis n'importe quelle plateforme d'hébergement de code source »[3].

Son architecture consiste en un *framework* permettant de retrouver le code source des logiciels susmentionnés et de les ingérer au sein de l'archive universel de `Software Heritage`. En particulier, les **Listers** en constituent une partie centrale : il s'agit de *crawlers* configurés pour parcourir des dépôts de code source, « *mapper* » leurs modèles à des modèles intégrables à l'infrastructure, et reporter l'ingestion de leur contenu à d'autres composants du *framework*. L'ingestion du contenu d'un dépôt « *listé* »au sein de l'archive est effectuée par des composants spécifiques, les **Loaders**. Enfin, la planification des tâches du *listing* et du *loading* est régulée par un **Scheduler**, un composant interagissant avec une queue de tâches asynchrones opérée par un serveur `Celery`.

Il faut préciser que les plateformes d'hébergement embarquent chacune des dépôts de code source à structures différentes, ce qui nécessite la création d'un **Lister** dédié pour chaque plateforme. Par ailleurs, les différentes ver-

1. **Institut National de Recherche en Informatique et Automatique**

sions d'un logiciel et leurs métadonnées associées sont gérées par un gestionnaire de version, ce qui nécessite la création d'un **Loader** dédié pour chaque gestionnaire. Actuellement, tous les **Listers** et **Loaders** ont été créés uniquement par l'équipe de `Software Heritage`. Les **Listers** développés l'ont été pour les plateformes d'hébergement les plus populaires (`Github`, `Bitbucket`, ...). De même, les **Loaders** développés l'ont été pour les gestionnaires de version les plus populaires (`Git`, `SVN`, `Mercurial`, ...).

1.2 Contexte du TER

Dans le cadre de ce projet, encadré par Jessie Carbonnel, du module **HMIN201** désignant le TER, encadré par Mathieu LaFourcade, notre objectif final consiste à créer un **Lister** pour une plateforme de développement ciblée. Ainsi, les tâches nécessaires à effectuer afin d'accomplir ce but peuvent être énumérées de la manière suivante :

- Lire et comprendre les articles et tutoriels écrits par l'équipe de `Software Heritage` ;
- Analyser différentes plateformes d'hébergement afin d'en cibler une ;
- Concevoir et développer un **Lister** pour la plateforme choisie ;
- Répliquer localement l'environnement de `Software Heritage` afin de tester le **Lister** développé ;
- Faire une *Pull Request* afin d'intégrer le **Lister** testé au dépôt de développement de `Software Heritage` sur `GitHub`.

1.3 Plan du rapport

Nous commençons ce rapport par une courte description de `Software Heritage`, suivie par la spécification du contexte du stage. Ensuite, nous détaillerons la problématique générale traitée par `Software Heritage` et la sous-problématique particulière adressée par notre projet.

Par la suite, nous fournirons une explication technique détaillée de l'infrastructure de `Software Heritage` et de son fonctionnement, l'étape fondamentale sur laquelle se base notre méthodologie, et nous terminerons la section par le planning prévisionnel du projet. Après, nous élaborerons nos approches pour la conception d'un **Lister** et son implémentation, ainsi que les résultats obtenus.

Pour conclure, nous comparerons les versions prévisionnelle et finale du planning, puis nous discuterons les difficultés rencontrées et les perspectives du projet. Finalement, nous listerons un bilan du projet en citant ses apports.

Chapitre 2

Problématique

2.1 La diaspora du code source

2.2 La fragilité du code source

2.3 Software Heritage en tant que solution

Current status et roadmap de SWH

2.4 Notre contribution

Chapitre 3

Analyse

3.1 Terminologie et fonctionnement de Software Heritage

3.1.1 Modèle des données

Le modèle des données de `Software Heritage` est centré sur la notion de stockage d'« **artefacts logiciels** » et leurs **informations de provenance** correspondantes, hébergés sur des **plateformes d'hébergement de code source**[3].

Plateformes d'hébergement de code source

Les **plateformes d'hébergement de code source** sont destinées à être « *crawlées* » par des **Listers** et ingérées au sein de l'archive universel de `Software Heritage` par des **Loaders**[3]. Ces plateformes sont catégorisées de la manière suivante :

***forges* de développement collaboratif :** `GitHub`, `GitLab`, `BitBucket`, ...

dépôts d'un gestionnaire de paquets : `PyPI`[1], `CPAN`[2], `npm`[3], ...

distributions logicielles FOSS[4] **:** `Debian`, `Fedora`, `FreeBSD`, ... - other types : e.g.

autres : par exemple les **URL**s[5] personnelles et celles désignant des collections de projets institutionnels non hébergées sur des *forges*.

Artefacts logiciels

Définition (**Artefact Logiciel**)**.**
Selon Le grand dictionnaire terminologique, un **artefact logiciel** désigne

1. ***Python Package Index***
2. ***Comprehensive Perl Archive Network***
3. ***Node Package Manager***
4. ***Free and Open-Source Software***
5. ***Uniform Resource Locator***

tout « *module d'information utilisé ou produit lors de la conception d'un logiciel* ».

Dans le cadre de `Software Heritage`, pour tout logiciel hébergé sur une plateforme d'hébergement de code source, il existe plusieurs **artefacts logiciels** qui sont assez récurrent lors du développement du logiciel, et qui constituent les composants de base de l'archive[3]. Ces artefacts peuvent être catégorisés de la manière suivante :

1. *file contents* ou *blobs* ;
2. *directories* ;
3. *revisions* ou *commits* ;
4. *releases* ou *tags*.

Définition (**Blob**)**.**
Le **contenu binaire du code source** (*i.e. les octets*), sans aucune métadonnée associée (même pas le nom du blob). C'est Un artefact récurrent à travers différentes versions d'un même logiciel, différents répertoires du même projet, voire même différents projets.

Définition (**Directory**)**.**
Une **liste récursive d'entrées nommées** pointant vers d'autres aretfacts (*i.e. des blobs ou d'autres directories*). C'est Un artefact associé à des métadonnées divers (*e.g. bits de permission, estampilles de modification, ...*).

Définition (**Revision**)**.**
Une **version du *directory* racine du logiciel** tel qu'il est capturé par un **gestionnaire de version** (*i.e. un commit*), contenant la totalité du **code source** du projet désigné par le logiciel. C'est un artefact associé à des métadonnées divers (*e.g. message de commit, estampilles, versions précédentes, ...*).

Définition (**Release**)**.**
Une **revision stable** qui pourra être mise en production (*i.e. un project milestone*). C'est un artefact associé à des métadonnées divers désignant les **métadonnées d'une revision** et d'autres (*e.g. nom du release, version du release, signatures digitales, ...*).

Informations de provenance des données

Dans le cadre de `Software Heritage`, pour tout logiciel hébergé sur une plateforme d'hébergement de code source, les **informations retournées** par

le *crawling* de celui-ci sont appelées les informations de provenance (*provenance information*)[3]. Ces informations peuvent être catégorisées de la manière suivante :

1. *software origins* ;
2. *projects* ;
3. *snapshots* ;
4. *visits*.

Définition (**Software Origin**)**.**
Un **ensemble de références** pointant vers les endroits de récupération des **artefacts logiciels** d'un logiciel, archivés dans `Software Heritage`. Il s'agit d'une **paire** <*type*, *url*> :

type : le **type de l'origine** (un **gestionnaire de version** tel que `Git` ou `SVN`, un **paquet source** tels que `DSC`[6], ...)

url : une **adresse URL canonique** désignant l'**adresse de l'origine** (une **adresse clonable** par un **gestionnaire de version**, ou **téléchargeable** telle qu'un *tarball* téléchargé via `wget`).

Définition (**Project**)**.**
Une **entité abstraite** associée à des **software origins** divers, avec leurs métadonnées correspondantes. De plus, un projet peut être **versionné** et **imbriqué** dans une **hiérarchie de projets**, et permet de générer des **ressources de développement** (*e.g. websites, issue trackers, mailing lists, software origins, ...*).

Définition (**Snapshot**)**.**
Une **snapshot à un instant donné** d'un/plusieurs **point(s) d'entrée** d'un logiciel référencé par un **software origin** :

— s'il s'agit d'un **gestionnaire de version** : **points d'entrée** = les **branches** de développement (*e.g. une snapshot de la branche principale, une autre snapshot de la branche de features ...*) ;
— s'il s'agit d'une **distribution de paquets source** : **points d'entrée** = les **suites**[7] de développement (*e.g. une snapshot pour la dernière version d'un paquet source pour la suite stable*).

Définition (**Visit**)**.**
Un **lien** entre un **software origin** et une **snapshot**, créé lors de la consultation du **software origin**, permettant d'enregistrer le **moment de son consultation** et un **snapshot entier** de son état.

6. ***Debian Source Control***
7. différents niveaux de maturité d'un paquet source logiciel

Structure de données

La structure de données à utiliser pour implanter l'archive doit permettre la déduplication de certains **artefacts logiciels** et **informations de provenance** tels que les **blobs**, les **directories**, les **revisions**, les **releases** et les **snapshots**. Cette déduplication est essentielle afin d'assurer une préservation à long terme et un stockage efficace. Pour ce faire, `Software Heritage` ont adopté le modèle d'un **graphe orienté acyclique Merkel**[3] ou **Merkel DAG** [8] (cf. Figure 3.1).

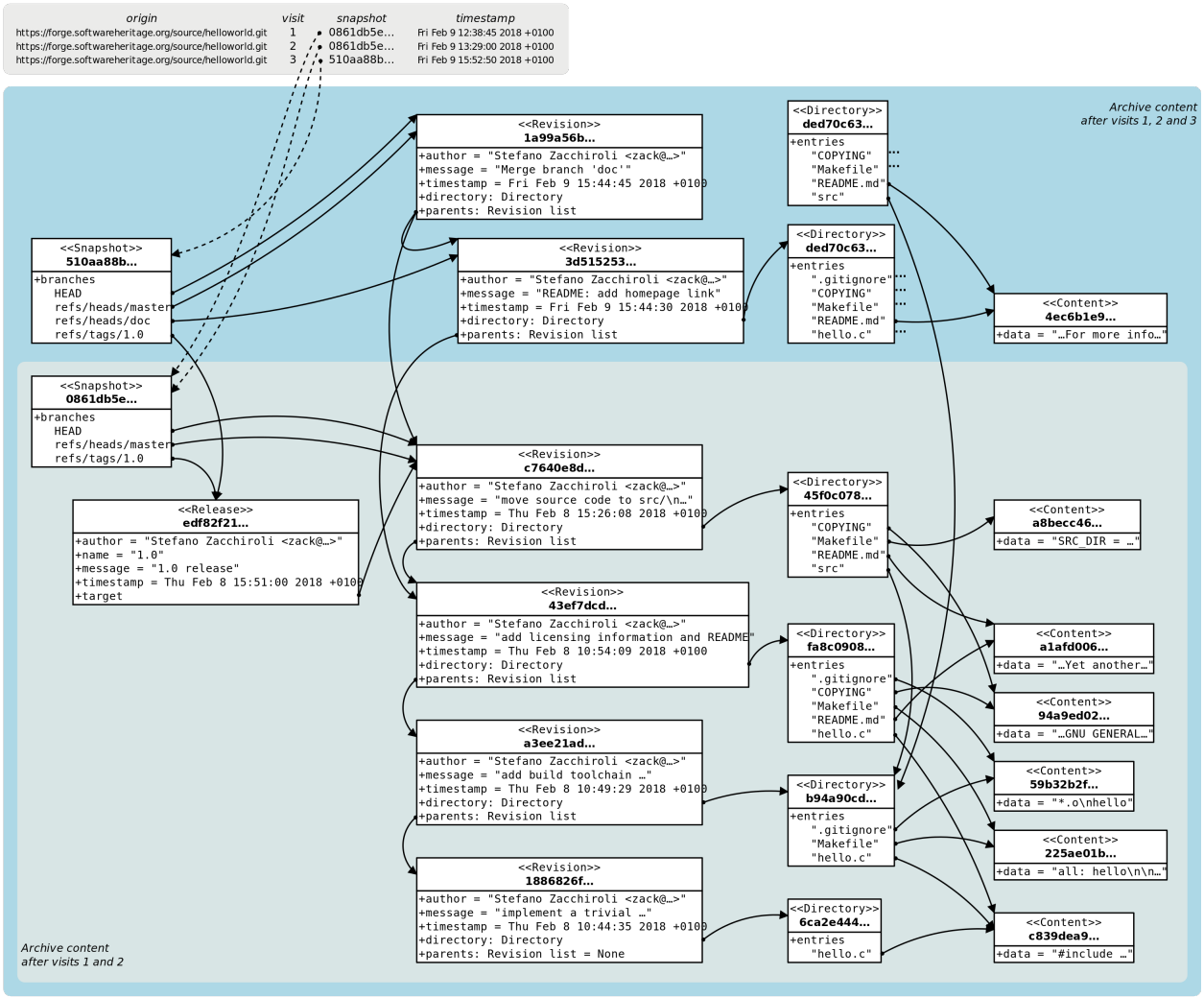

FIGURE 3.1 – **Merkel DAG** de `Software Heritage`[4]

8. ***Direct Acyclic Graph***

Le **Merkel DAG** est composé de noeuds et d'arcs tels que :

1. **noeuds** – chaque noeud :
   — désigne un **artefact logiciel unique** ;
   — est identifié par un **identifieur intrinsèque** désignant un digest cryptographique calculé à partir du noeud et son contenu. Ceci implique qu'un **software origin** sera ajouté au **Merkel DAG**, uniquement quand celui-ci ne contient pas déjà un noeud ayant le même identifiant. Cette propriété du **Merkel DAG** assure une **déduplication native** à l'archive implanté ;
   — contient l'ensemble des **métadonnées** qui lui sont propres (*e.g. messages de commit, estampilles, noms de fichiers, ...*) ;
   — contient des pointeurs vers les identifiants des **noeuds enfants** en format canonique.
2. **arcs** :
   — les **directories** pointent sur des **blobs** et d'autres **directories** ;
   — les **revisions** pointent sur des **directories** et les **revisions** précédentes ;
   — les **releases** pointent sur des **revisions** ;
   — les **snapshots** pointent sur des **releases** et des **revisions**.

Un noeud du **Merkel DAG** désignant une **revision** est présenté dans la figure 3.2. On voit bien l'identifiant intrinsèque du noeud, ainsi que celui du noeud désignant le **directory racine** pointé par la **revision**, et celui du noeud désignant la **revision** précédente. De plus, on voit la date d'ajout, l'auteur, le commiteur, le message et la date du commit de la **revision**.

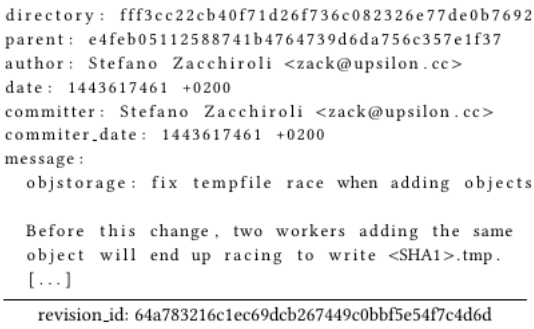

FIGURE 3.2 – Un noeud du **Merkel DAG** de `Software Heritage`[3]

Identifieurs intrinsèques

Afin de pouvoir manipuler les différents **artefacts logiciels** à archiver, l'archive a besoin de les identifier et de les référencer. Pour ce faire, les

identifieurs doivent être uniques, persistents, et intrinsèques. De plus, ils doivent supporter la gestion des versions, ainsi que l'identification à differents niveaux de granularité (*i.e. d'une snapshot à un blob*). Ainsi, l'équipe de `Software Heritage` a adopté les identifieurs **IDO** [9], permettant de satisfaire ces besoins[5].

Les **IDO**s sont régis par la **syntaxe BNF** suivante :

```
<identifier> ::= "swh" ":" <scheme_version> ":" <object_type> ":" <object_id> ;
<scheme_version> ::= "1" ;
<object_type> ::=
"snp" (* snapshot *)
| "rel" (* release *)
| "rev" (* revision *)
| "dir" (* directory *)
| "cnt" (* content *)
;
(* intrinsic object id, as hex-encoded SHA1 *)
<object_id> ::= 40 * <hex_digit> ;
<hex_digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
| "a" | "b" | "c" | "d" | "e" | "f" ;
```

Par ailleurs, les **IDO**s peuvent être complementés par des **informations contextuelles**, dont deux types sont supportés pour le moment :

***origin* :** l'**URL software origin** du logiciel associé.

***line numbers of interest* :** le numéro d'une ligne, ou un intervalle.

Les **informations contextuelles** sont régies par la **syntaxe BNF** suivante :

```
<identifier_with_context> ::= <identifier> [<lines_ctxt>] [<origin_ctxt>] ;
<lines_ctxt> ::= ";" "lines" "=" <line_number> ["-" <line_number>] ;
<origin_ctxt> ::= ";" "origin" "=" <url> ;
<line_number> ::= <dec_digit> + ;
<url> ::= (* RFC 3986 compliant URLs *) ;
```

Les tables 3.1 et 3.2, contiennent un ensemble d'exemples d'**IDO**s avec/sans des informations contextuelles pour différents types de noeuds du **Merkel DAG**.

IDO	Type de Noeud
`swh:1:cnt:94a9ed024d3859793618152ea559a168bbcbb5e2`	**blob**
`swh:1:dir:d198bc9d7a6bcf6db04f476d29314f157507d505`	**directory**
`swh:1:rev:309cf2674ee7a0749978cf8265ab91a60aea0f7d`	**revision**
`swh:1:rel:22ece559cc7cc2364edc5e5593d63ae8bd229f9f`	**release**
`swh:1:snp:c7c108084bc0bf3d81436bf980b46e98bd338453`	**snapshot**

TABLE 3.1 – Exemples d'**IDO**s de différents noeuds du **Merkel DAG** de `Software Heritage`

IDO	Type de Noeud
`swh:1:cnt:41ddb23118f92d721 8099a5e7a990cf58f1d07fa; origin=https://github.com/chrislgarry...; lines=64-72/`	**blob** avec l'**URL** du **software origin** et l'**intervalle des lignes** d'intérêt du code source désigné
`swh:1:dir:c6f07c2173a458d09 8de45d4c459a8f1916d900f; origin=https://github.com/id-Software/Qua...`	**directory** avec l'**URL** du **software origin**

TABLE 3.2 – Exemples d'**IDO**s avec d'**informations contextuelles** de différents noeuds du **Merkel DAG** de `Software Heritage`

3.1.2 Architecture conceptuelle et flot des données

Flot d'ingestion des données

L'architecture à adopter pour la plateforme doit permettre de « *crawler* » une liste de plateformes d'hébergement de code source et d'archiver leur contenu. Pour ce faire, l'équipe de `Software Heritage` ont adopté une architecture conceptuelle[3] divisant la tâche en deux sous-tâches, effectuées respectivement par deux composants de base : le *listing* des plateformes d'hébergement par des **Listers** et le *loading* de leur contenu au sein de l'archive de par des **Loaders** (cf. Figure 3.3).

Listing

Le *listing* d'une plateforme d'hébergement de code source consiste à énumérer les **software origins** qui lui sont associés (e.g. des dépôts sur `GitHub` ou `BitBucket`, des paquets source individuels de `PyPI` ou `Debian`, ...). Pour chaque plateforme, un **Lister** dédié doit être créé afin de « *mapper* »les modèles des **software origins** vers des modèles équivalents intégrables au sein

9. ***Identifiers for Digital Objects***

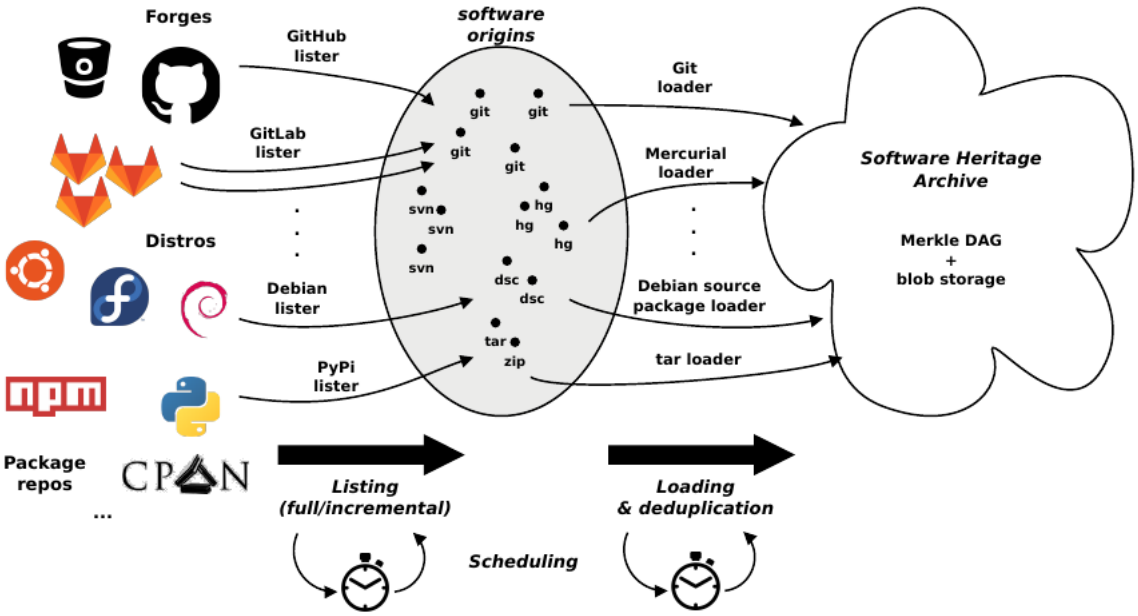

FIGURE 3.3 – L'architecture conceptuelle de la plateforme de `Software Heritage`[3]

de l'architecture de `Software Heritage`[3].

En outre, il existe deux techniques de *listing* :

full listing **:** collecter la liste entière des **software origins** associée à une plateforme d'hébergement de code, afin s'assurer de n'en rater aucun. Il s'agit d'une technique à utiliser une seule fois initialement, et d'une manière moins fréquente ultérieurement en raison de son aspect chronophage, surtout quand la plateforme est relativement grande.

incremental listing **:** collecter uniquement l'ensemble des **software origins** qui ont été modifiés ou ajoutés depuis le dernier *listing*. Il s'agit d'une technique à privilégier et à utiliser régulièrement, suite au premier *full listing*, pour la mise à jour des noeuds correspondant aux **software origins** au sein du **Merkel DAG**.

De plus, il existe deux styles de *listing* :

pull style **:** l'archive consulte régulièrement les plateformes d'hébergement de code en vue de lister leurs **software origins**. Cette technique est assurée par défaut par les **Listers** apropriés.

push style **:** les plateformes d'hébergement collaborant avec `Software Heritage`, si proprement configurées, notifient l'archive à chaque modification de leurs **software origins** associés. Cette technique permet

de minimiser le décalage entre la version archivée et la version hébergée d'un **software origin**.

Loading

Le *loading* du contenu des **software origins** d'une plateforme d'hébergement de code, correspond à l'extraction de leurs **artefacts logiciels** associés et leur ingestion au sein de l'archive. Pour chaque type de **software origin**, un **Laoder** dédié doit être créé afin d'« *ingérer* »les **artefacts logiciels** et **snapshots** associés, en assurant la contrainte de déduplication des noeuds au sein du **Merkel DAG**[3] (e.g. un **Loader** pour chaque gestionnaire de version tels que `Git` ou `SVN`, un **Loader** pour chaque format d'un paquet source tels que `Debian source packages` ou `tarballs`, ...).

Scheduling

Les tâches de *listing* et de *loading* occurrant régulièrement, un composant permettant de planifier leurs occurrences s'avère ainsi primordial. Il s'agit du composant **Scheduler**, permettant de **synchroniser** ces tâches dans une **queue de tâches asynchrones** opérée par un **serveur** `Celery`[3].

Le **Scheduler** est implémenté selon les stratégies d'*adaptive scheduling* et d'*exponential backoff*, s'appuyant la notion d'**actions fructueuses** ou *fruitful actions*. Ces stratégies permettent d'équilibrer entre la mise à jour du contenu de l'archive et la surcharge des plateformes concernées (`Software Heritage` et les plateformes d'hébergement de code consultées), surtout lors du *loading* des **software origins** listés associés à une plateforme assez large.

Définition (**Fruitful Action**)**.**
Une **action**, désignant une **tâche périodique à planifier** (*i.e. listing* ou *loading*), est considérée **fructueuse** si la visite associée à l'action retourne de **nouvelles informations** depuis la **dernière visite** :

***fruitful listing* :** lors de la découverte de nouveaux **software origins** à « *lister* » ;

***fruitful loading* :** lors du changement de l'état d'un **software origin** consulté depuis la dernière visite.

Définition (**Adaptive Scheduling and Exponential Backoff**)**.**
La **stratégie** d'**Adaptive Scheduling** permet d'**augmenter la fréquence** des visites d'une **action** quand celle-ci est **fructueuse**, et de la **diminuer** dans le cas contraire. Le **taux** de cette augmentation/diminution est spécifié par la **stratégie** d'**Exponential Backoff**, indiquant de le **doubler** en cas d'une **augmentation** et de le **diviser par deux** en cas d'une **diminution**.

3.1.3 L'archive

D'une part, la **conception logique** de l'**archive** de `Software Heritage` consiste en un **Merkel DAG**. D'autre part, l'**implémentation physique** de l'**archive** combine plusieurs technologies de stockage, en raison des différentes tailles de stockage des noeuds du **Merkel DAG**.

Stockage des noeuds BLOB

Les **blobs** occupent la **majorité de l'espace de stockage**, étant donné qu'ils contiennent la totalité du code source. Afin d'assurer l'espace et les méchanismes de stockage convenables, l'équipe de `Software Heritage` a introduit un composant **ObjectStorage** gérant ces tâches[3]. Il s'agit d'une **table de hachage**, associant chaque **blob** à son **IDO** correspondant utilisé comme **clé**. Ceci permet la **distribution du stockage** dans un **cluster de tables de hachage** et de profiter des avantages associées.

En cas de **collision** (*i.e. lorsque le calcul des **IDOs** de deux objets différents donne un résultat identique*), l'archive utilise plusieurs **algorithmes de checksum** avec des contraintes d'unicité. Par conséquent, l'archive peut détecter les collisions avant l'ingestion d'un nouvel **artefact logiciel**. Parmi les algorithmes de checksum utilisé, nous notons `SHA1` et `SHA256`.

Stockage des autres noeuds

Les autres noeuds du **Merkel DAG**, notamment les **directories**, **revisions**, **releases** et **snapshots**, sont stockés chacun dans une **base de données relationnelle** `PostgresSQL` dédiée. Chaque **tuple** d'une table de base de données est identifié par l'**IDO** du **noeud correspondant** et contient son contenu[3]. Ceci permet la **distribution du stockage** dans un **cluster de tables de base de données** et de profiter des avantages associées.

Réplication des noeuds

À chaque type de noeud du **Merkel DAG** est associé un **log de changement** ou *feed change* persistant, détaillant l'ensemble des **changements effectués** sur les noeuds[3]. Un tel outil est idéal pour la **réplication** des noeuds : après une opération de **réplication entière** d'un **log de changement**, les mirroirs peuvent rester à jour facilement par rapport à l'archive principal par **réplication incrémentale**.

Politique de rétention

La **politique de retention** permet de **contrôler la réplication des noeuds**, afin d'assurer un **système tolérant aux pannes** et la **longue préservation** des **artefacts logiciels**[3]. Pour ce faire, la politique actuelle précise la nécessité d'avoir deux mirroirs locaux du composant **ObjectStorage** entier, et une troisième sur un cloud publique. Afin d'assurer le respect de cette politique, un **composant** de l'infrastructure de `Software Heritage` :

- suit le nombre et la localisation des mirroirs de chaque **noeud archivé** ;
- vérifie régulièrement l'**adhérence des noeuds archivés** à la **politique de retention** ;
- crée des **réplications supplémentaires** d'un noeud archivé en cas de manque de mirroirs pour assurer son adhérence à la **politique de retention**.

Récupération automatique des objects corrompus

En cas d'un noeud corrompu, un composant de l'infrastructure `Software Heritage`[3] :

- choisit **régulièrement** un ensemble **aléatoire** de **noeuds archivés** à **vérifier** ;
- re-calcule l'**IDO** de chaque noeud de l'ensemble choisi, ainsi que celui de chacun de ses **mirroirs**, afin de vérifier leur **intégrité** ;
- en cas de **violation d'une contrainte d'intégrité** par l'une des copies du noeud archivé, tous les mirroirs du noeud concerné seront **vérifiés dynamiquement**. Au cours de la vérification, les **versions corrompues** du noeud concerné seront **automatiquement remplacées** par un **mirroir** vérifiant la **contrainte d'intégrité** parmis ceux choisis.

3.1.4 Architecture technique

3.1.5 Diagrammes de séquence

3.2 Méthodologie

3.3 Planning Prévisionnel

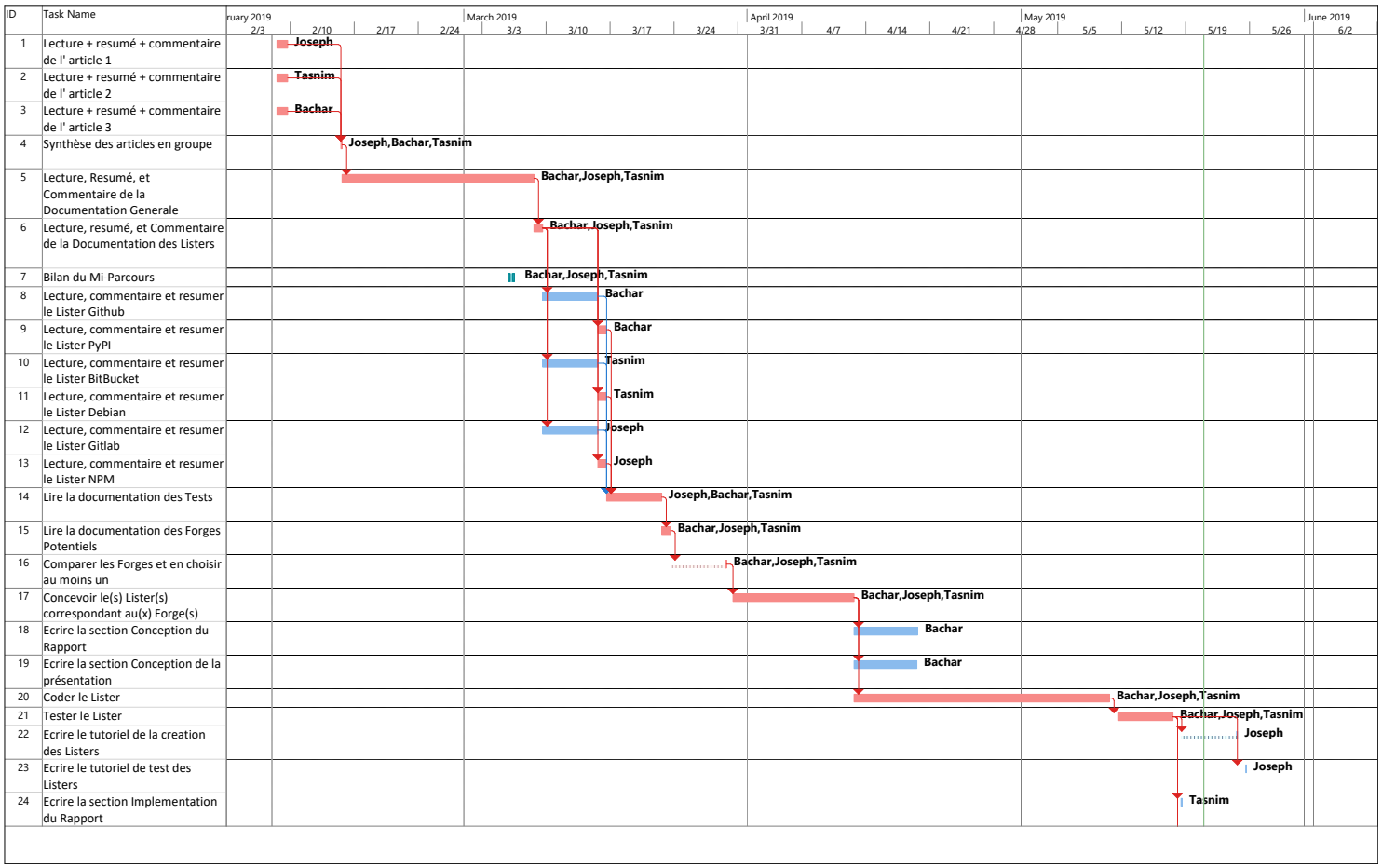

FIGURE 3.4 – Planning prévisionel

Chapitre 4

Conception

design de la solution proposée (diagrammes + explications)

Chapitre 5

Implémentation

les technos qu'on a utilisé
bibliotheques
Outils (e.g. XML parsers)
Launchpad client

Chapitre 6

Résultats

pull request ?

Chapitre 7

Conclusion

7.1 Planning final

7.2 Difficultés rencontrées

7.3 Perspectives

7.4 Bilan et apports du TER

annexes
resumés
code

Bibliographie

[1] The internet archive software collection. https://archive.org/details/software&tab=about. Accessed : 2019-05-23.

[2] About persist : Unesco persist programme. https://unescopersist.org/about/. Accessed : 2019-05-23.

[3] Roberto Di Cosmo and Stefano Zacchiroli. Software Heritage : Why and How to Preserve Software Source Code. In *iPRES 2017 - 14th International Conference on Digital Preservation*, pages 1–10, Kyoto, Japan, September 2017.

[4] Software heritage documentation. https://docs.softwareheritage.org/devel/_images/swh-merkle-dag.svg. Accessed : 2019-05-23.

[5] Roberto Di Cosmo, Morane Gruenpeter, and Stefano Zacchiroli. Identifiers for Digital Objects : the Case of Software Source Code Preservation. In *iPRES 2018 - 15th International Conference on Digital Preservation*, pages 1–9, Boston, United States, September 2018.