École Polytechnique de l'Université de Tours
64, Avenue Jean Portalis
37200 TOURS, FRANCE
Tél. +33 (0)2 47 36 14 14
www.polytech.univ-tours.fr

Département Informatique
3e année
2013 - 20014

Rapport de projet

Développement d'un logiciel de téléchargement parallélisé pour la protection du copyright des mangas

Encadrant
Mathieu Delalandre
mathieu.delalandre@gmail.com

Université François-Rabelais, Tours

Étudiants
Anthony DEMARCY
anthony.demarcy@etu.univ-tours.fr
Florian MONTALBANO
florian.montalbano@etu.univ-tours.fr

DI3 2013 - 2014

Version du 8 juin 2014

Table des matières

Table des figures

Introduction

Aujourd'hui, par le biais d'Internet, il est très facile de se procurer diverses oeuvres soumises à des droits d'auteur et supposées payantes, et ce gratuitement et en toute illégalité. Le piratage de telles oeuvres est depuis bien longtemps devenu un sujet d'inquiétude majeur auprès des industriels et artistes qui sont à leur origine, pour des raisons financières évidentes. Un secteur très touché par ce phénomène est celui des ***mangas*** : ces bandes dessinées japonaises qui sont traditionnellement publiées sous formes de chapitres dans des magazines spécialisés au Japon sont aujourd'hui très prisées sur Internet.

La vente de tels magazines a atteint son paroxysme en 1995, année à partir de laquelle ce moyen de vente connût un éternel déclin. Le marché du manga valait plus de 5 milliards de dollars en 2009, et les éditeurs entendent continuer à le voir prospérer. A plus petite échelle, ce sont de nombreux auteurs de mangas (appelés ***mangakas***) dont le métier serait menacé sur le moyen terme par une prolifération excessive de ces pratiques de téléchargement. C'est pourquoi la lutte contre cette distribution libre d'oeuvres *copyrightées* est devenu un enjeu de taille.

Deux formes de distribution illégale des mangas sont à distinguer :

— D'un côté, les ***scanlations*** qui sont des versions scannées des originaux, puis éditées par des fans afin d'être traduites 'bénévolement' dans une multitude de langues. Ces travaux sont accessibles gratuitement au plus grand nombre.

— D'un autre côté, le **piratage** qui consiste à obtenir le manga le plus tôt possible (parfois même avant sa sortie) afin de le scanner et de le proposer en tant que support parfaitement conforme à l'original, étant revendus par la suite.

L'objectif de ce projet (réalisé en collaboration avec l'université préfectorale d'Osaka) est de concevoir un système complet capable d'aider au téléchargement de masse de fichiers *copyrightés* afin qu'ils puissent être ultérieurement soumis à analyse. Ce système se décompose ainsi en deux parties majeures :

— Un ***web crawler*** qui repère des 'liens suspects' et les enregistre en tant que **cibles** dans une base de données prévue à cet effet.

— Un **programme de téléchargement parallélisé** qui analyse les cibles préalablement trouvées par le *web crawler*, et en télécharge plusieurs simultanément tout en gérant une notion de priorité afin d'ordonnancer ces téléchargements.

Le travail de notre binôme résidait dans le développement du programme de téléchargement parallélisé, le *web crawler* étant réalisé par un autre groupe d'étudiants.

Étude du système

1.1 Le Web Crawler

Les ***web crawlers*** sont des programmes capables d'arpenter des pages web, souvent dans des buts d'indexation (moteurs de recherche) ou de récupération de contenu. C'est précisément ce deuxième cas qui nous intéresse dans le cadre de notre projet.

1.1.1 Le fonctionnement de base d'un web crawler

Un ***web crawler*** traverse les pages web grâce à des **liens**, comme le ferait un utilisateur normal. Il arrive ainsi sur une nouvelle page, qu'il analyse (les éléments analysés diffèrent selon le but initial du web crawler), puis note l'ensemble des liens qu'ils considèrent comme pertinents afin d'être explorés par la suite. Cette problématique en apparence simple cache de nombreux pièges : par exemple, un tel programme essaie d'éviter à tout prix de revisiter une page qu'il a déjà explorée (pour des raisons d'efficacité évidentes). Le risque majeur lié à cela est l'enfermement du robot dans une boucle infinie où ce dernier revisite une même suite de pages, encore et encore.

Une solution communément utilisée afin d'éviter ce genre de problèmes est l'emploi d'une **table de hachage** contenant le hachage de chacun des liens déjà visités. A la visite de chaque nouvelle page, son URL est hachée selon une fonction produisant un entier unique qui y est associé, et un marqueur est mis à cette case de la table pour indiquer que cette URL a été visitée. Ainsi, si le robot venait à tenter d'explorer de nouveau cette même page, l'URL serait identique et conduirait donc à un hachage identique. Il consulterait donc la même case de la table que lors de la première visite de la page, et y apercevrait le marqueur, renonçant donc à la réexplorer. Le choix de la technique de hachage est variable et dépend du web crawler, Google utilise par exemple son propre algorithme dénommé CityHash [1] pour ses robots d'indexation.

Afin d'être performant, un *web crawler* fonctionne la plupart du temps avec différentes "***branches d'exploration***" effectuées en simultané grâce à un système de parallélisation (**multithreading**). Ainsi, chacune de ces branches est gérée sur un fil d'exécution (**thread**) qui lui est propre, démultipliant la vitesse d'exploration du robot. Cependant, la table de hachage mentionnée précédemment doit être commune à l'ensemble des *threads* : son accès doit donc être contrôlé et géré intelligemment afin d'éviter les problèmes de partage de ressources habituels liés au *multithreading*. Cela peut s'effectuer par l'utilisation de primitives de synchronisation simples telles que des **sémaphores** ou des **exclusions mutuelles**, ou de façon plus sophistiquée avec des **moniteurs**.

Il est bon de remarquer qu'un robot comme celui requis dans notre système est très peu consommateur en puissance processeur, mais peine davantage à démultiplier les fils d'exécution à cause de la connexion Internet qui le relie aux serveurs. Paradoxalement, l'emploi du *multithreading* sur une telle application sert justement à contrebalancer la perte de temps liée à l'accès aux pages. Cela se manifeste d'autant plus lorsque l'on tente d'accéder à des serveurs distants et que la moindre page requiert plusieurs secondes d'attente (ce qui aurait des conséquences dramatiques sur un *web crawler* sans parallélisme).

1. Plus d'informations sur CityHash : https://code.google.com/p/cityhash/

1.1.2 Les pièges rencontrés par des web crawlers

Les *web crawlers* d'indexation comme ceux de Google sont 'polis' : ils prennent en compte un fichier standardisé (robots.txt [2]) afin d'être au courant des règles que les gérants du site ont fixé sur son exploration. Ainsi, avec un contenu approprié, un gérant de site peut interdire à ces robots 'polis' de visiter leur site. En lisant le fichier robots, ceux-ci prendront connaissance de l'interdiction et ne poursuivront pas leur exploration. Cependant, dans le contexte du système que nous souhaitons réaliser, il n'y a aucun intérêt à prendre en compte un tel fichier au vu des objectifs de notre web crawler. Pire encore, le prendre en compte serait la porte ouverte pour tous les sites pirates afin d'interdire la venue de notre robot et donc de se protéger définitivement des contrôles qu'il opère. C'est pourquoi notre web crawler a tout intérêt à ne pas être 'poli'.

Malgré tout, notre robot reste indésirable aux yeux de ces sites possédant des contenus illégaux, et c'est ainsi que divers stratagèmes ont été inventés au fil du temps pour contrer les web crawlers : ce sont les **bots traps** ('pièges à robots' en anglais).

Le type de **bot trap** le plus courant et le plus simple consiste en l'inclusion d'un lien invisible dans les pages du site qui amène sur une page spéciale. Comme le lien est invisible, aucun utilisateur n'est supposé pouvoir atteindre cette page (sauf en examinant minutieusement le code source). Seuls les robots peuvent atteindre cette page spéciale qui, une fois demandée au serveur, bloque l'IP du client (ce qui est réalisable avec un simple code PHP) afin de lui empêcher toute navigation. Notre robot se retrouvera donc banni du site, empêchant toute exploration (et a fortiori, toute détection de cibles pour notre système). Ces pièges sont brutaux, mais non nuisibles au web crawler en lui-même.

A l'inverse, d'autres pièges plus subtils visent à nuire au robot en l'enfermant dans une boucle d'exploration sans fin. Cela est réalisable grâce à l'inclusion dans le site web d'un 'module de génération de fausses pages' qui créé une quantité quasi-infinie de pages web ayant toutes une URL différente et se suivant les unes aux autres sous la forme d'une chaîne : ainsi, une fois que le robot a 'mordu à l'hameçon' en atteignant la première page du piège, chacune des pages suivantes ne comportant qu'un unique lien, celui-ci est tenté de suivre ce lien et va alors explorer des milliers de pages vides de toute pertinence.

Les bot traps sont donc un réel problème au parcours des *web crawlers*, car s'ils ne font pas attention, ils peuvent tomber dans ces pièges par mégarde et être privés d'un contenu pertinent.

1.1.3 Les résultats émis par notre web crawler

Comme mentionné en introduction, notre rôle dans le projet n'est pas de s'occuper du *web crawler*, mais de l'application de téléchargement parallélisé (qui sera appelée ***downloader*** dans la suite de ce rapport). Les recherches que nous avons menées sur les *web crawlers* étaient donc uniquement à but informatif, afin de mieux cerner le fonctionnement global de notre système. Concrètement, seules les **cibles** retournées par le web crawler et les données qu'elles proposent nous intéressent dans le cadre du développement du *downloader*.

De ce point de vue, une grande liberté nous a été laissée afin de proposer notre propre structure pour les cibles, indépendemment de celle réalisée par nos collègues s'occupant du *web crawler*. Le but du *downloader* étant d'ordonnancer et d'effectuer ces téléchargements en exploitant le principe de *multithreading* afin d'en effectuer plusieurs simultanément, nous avons très vite nécessité la présence de **métadonnées** servant à classer les cibles entre elles.

2. Voir la partie "Références" en fin de rapport pour plus de détails

En plus de contenir une **URL**, une cible pourrait alors contenir l'ensemble des métadonnées suivantes :

— **Niveau de priorité** (arbitraire, pouvant être défini par exemple en fonction du site de provenance)
— **Taille du fichier**
— **Popularité du fichier**
— **Débit du serveur**
— ...

Une somme pondérée de l'ensemble de ces métadonnées serait alors réalisée pour chacune des cibles, lui attribuant un **score** qui représente l'importance de la cible aux yeux de l'ordonnanceur. Les cibles ayant les plus hauts scores seraient ainsi traitées en premier par le *downloader*.

Une fois les sorties du *web crawler* étudiées, nous avons pu attaquer le vif du sujet et réfléchir au fonctionnement interne du *downloader*, qui est le coeur de notre projet.

1.2 Le Downloader

L'application que nous avons dû développer dans le cadre de ce projet devait répondre à deux problématiques majeures : une problématique **d'ordonnancement** et une autre de **parallélisation** des téléchargements effectués.

1.2.1 Ordonnancement et répartition

Lorsque le downloader est exécuté, il doit tout d'abord récupérer les **cibles** depuis une source extérieure (qui était une base de données dans le système théorique qui nous a été présenté). Une fois celles-ci acquises dans un ordre quelconque, il doit trouver un ordre qu'il considère comme optimal pour les traiter : c'est ici qu'intervient **l'ordonnanceur** (ou ***scheduler***). Ce module a pour vocation de trier les cibles précédemment acquises dans un certain ordre qui sera celui utilisé pour leur traitement. Comme évoqué précédemment, le *scheduler* va utiliser les **métadonnées** afin de calculer un score pour chacune des cibles, qui seront ensuite triées par ordre de score croissant.

Une fois cette liste ordonnée établie, c'est au tour du ***dispatcher*** de commander la réalisation des téléchargements à l'URL indiquée par chacune des cibles. Ainsi, si le *downloader* n'exploitait pas le principe de parallélisme, le dispatcher se contenterait de prendre les cibles de la liste ordonnée une à une et d'actionner la procédure de téléchargement pour chacune d'entre elles (en attendant patiemment que le téléchargement précédent soit terminé pour s'attaquer à la cible suivante).

Cependant, pour concevoir une application de téléchargement de masse performante, il est inconcevable de ne pas effectuer de mise en parallèle des différents téléchargements. C'est pourquoi nous nous sommes penchés sur cette problématique.

1.2.2 Gestion du multithreading

Lorsqu'un téléchargement est effectué, un flux de données circule sur le réseau, consommant une proportion de la **bande passante** de la connexion Internet utilisée pour effectuer le téléchargement. Plus le débit de téléchargement est élevé, plus la bande passante utilisée sera grande, et il arrive quelquefois que le serveur envoyant les données soit suffisamment rapide pour envoyer les données à un débit tel que

l'ensemble de la bande passante soit consommée. Cependant, ce cas est loin d'être une généralité, et de la bande passante non utilisée aurait pu servir à effectuer un second (voire plus) de téléchargements en **parallèle**.

C'est ici que le ***multithreading*** intervient en permettant d'effectuer plusieurs téléchargements en parallèle (chacun formant un fil d'exécution à part entière). Les ressources processeur sont ainsi partagées entre ces différents ***threads*** (ce qui n'est en rien pénalisant, puisqu'un tel programme est très peu consommateur sur cet aspect), mais cela permet surtout de maximiser l'utilisation de la bande passante à notre disposition, permettant de fait d'améliorer le rendement de l'application (qui peut être calculé en octets acquis par unité de temps).

C'est le rôle du *dispatcher* que de créer ces *threads* dans l'ordre donné au préalable par le *scheduler* afin de mettre en place ce procédé de parallélisation des tâches. Dans l'idéal, il aurait été possible d'optimiser la création de ces *threads* en analysant divers paramètres liés à l'état de la connexion et du processeur afin d'autoriser au dispatcher de créer un nombre optimal de *threads*. Par souci de simplicité, nous avons décidé de n'autoriser le dispatcher à créer que 3 *threads*, ce qui permet déjà dans la plupart des cas de maximiser l'utilisation de la connexion.

Il est également bon de noter qu'ici, aucun partage de données n'est effectué entre ces *threads*, car chacun doit effectuer une tâche qui lui est propre. Physiquement parlant, certaines ressources sont partagées (telles que la connexion réseau, l'accès au disque dur), mais ces aspects sont gérés par le système d'exploitation, et aucun système de synchronisation autre que le *dispatcher* n'est nécessaire pour sécuriser d'éventuelles variables partagées comme c'était le cas pour les *web crawlers*.

L'aspect *multithreading* de notre projet est celui qui nous a majoritairement poussé à nous orienter vers une **implémentation en Java**, car sa gestion est grandement simplifiée par la bibliothèque standard du langage. Gérer efficacement la parallélisation dans d'autres langages plus bas niveau (comme le C ou C++) aurait requis des bibliothèques externes plus ou moins bien documentées pour pouvoir être effectué dans le temps qui nous était imparti.

1.2.3 Étude des requêtes HTTP

Chacun des threads que nous venons d'évoquer doit, de son côté, s'occuper de télécharger la cible qui lui est confiée. Pour cela, nous avons dû nous intéresser aux mécaniques qui se cachent derrière un téléchargement effectué par n'importe quel navigateur web afin de pouvoir reproduire ce comportement dans le cadre de notre downloader. C'est pourquoi nous nous sommes intéressés au fonctionnement des requêtes HTTP qui permettent de communiquer avec un serveur web.

HyperText Transfer Protocol (HTTP) fût conçu en 1996 dans sa première version, et a très peu évolué. En effet, la version que nous utilisons aujourd'hui en permanence lorsque l'on navigue sur Internet est la version 1.1 achevée en 1999, sans qu'elle ait évoluée depuis. Les requêtes que formalisent ce protocole permettent une très grande variété d'actions, allant de la demande d'une page ou d'un fichier à un serveur jusqu'à l'envoi de données d'un formulaire ou de fichiers complets.

Sa grande souplesse réside dans sa structure, qui se décompose en deux grandes parties : un **en-tête** (header) et un **corps** (body). Voici un exemple de demande de page à un serveur :

```
GET /wiki/Wikip%C3%A9dia:Accueil_principal HTTP/1.1 [CRLF]
Host: fr.wikipedia.org [CRLF]
Connection: close [CRLF]
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X; de-de) AppleWebKit/523.10.3 (KHTML, like
Gecko) Version/3.0.4 Safari/523.10 [CRLF]
Accept-Encoding: gzip [CRLF]
Accept-Charset: ISO-8859-1,UTF-8;q=0.7,*;q=0.7 [CRLF]
Accept-Language: de,en;q=0.7,en-us;q=0.3 [CRLF]
[CRLF]
```

FIGURE 1.1 – Exemple de requête HTTP pour obtenir la page d'accueil du site Wikipedia

Une requête de page comme celle montrée ci-dessus ne possède aucun corps, l'en-tête suffisant à donner les informations nécessaires. Un en-tête HTTP est structuré en lignes (**CRLF** signifiant 'Carriage Return - Line Feed', une association de deux caractères représentant un saut de ligne), et la fin de cet en-tête est facilement repérable à **deux sauts de ligne consécutifs**.

Du point de vue de son contenu, on observe que cet en-tête débute par GET, ce qui signifie que l'on souhaite obtenir un fichier du serveur (ici, ce fichier est une page web). Un serveur pouvant abriter plusieurs hôtes, il est important de le préciser avec le champ **Host**. D'autres informations s'ensuivent, permettant de donner des détails au serveur afin de lui permettre d'affiner la page exacte qui va nous être envoyée. Dans le cadre d'un téléchargement de fichier archive comme nous comptons le faire, des champs tels que **User-Agent** (qui représente "l'identité" du navigateur ou robot qui explore le site) n'ont que peu d'influence.

En réponse à cette requête, le serveur envoie une réponse suivant la même structure, avec des champs différents :

```
Status: HTTP/1.1 200 OK [CRLF]
Server: nginx/1.1.19 [CRLF]
Date: Fri, 06 Jun 2014 12:50:15 GMT [CRLF]
Content-Type: [CRLF]
Content-Length: [CRLF]
Connection: close [CRLF]
[CRLF]
<!DOCTYPE html>
<html lang="fr" dir="ltr" class="client-nojs">
<head>
<meta charset="UTF-8" />
    . . .
</html>
```

FIGURE 1.2 – Exemple de réponse HTTP suite à la requête de la page d'accueil

On remarque surtout la présence d'un corps, qui est le contenu de la page en elle-même. Il en va de même pour le téléchargement d'un fichier : le corps de la réponse contiendra les octets du fichier.

C'est ce procédé que nous avons utilisé afin de pouvoir télécharger des fichiers au moyen de notre downloader. Nous avons pu, suite à ces recherches menées sur HTTP, établir un modèle de requête minimaliste qui sera utilisé par le downloader pour toute la suite du projet :

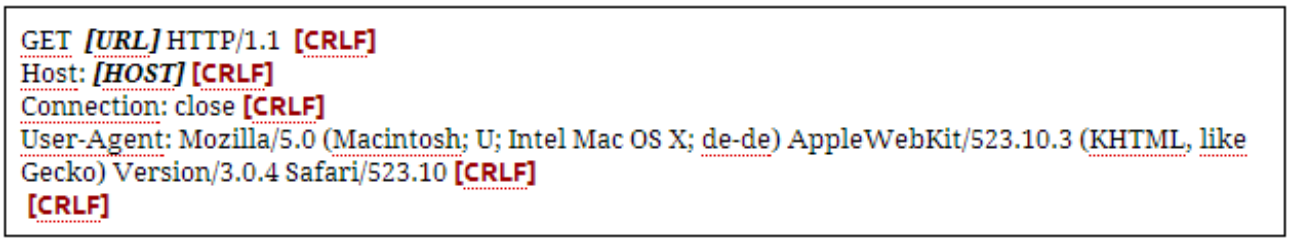

FIGURE 1.3 – Modèle de requête HTTP utilisé par le downloader

Il suffit alors uniquement de remplacer Il suffit alors uniquement de remplacer **[HOST]** par la partie 'Hôte' de l'URL de la cible (par exemple, *fr.wikipedia.org*), et **[URL]** par tout ce qui suit dans l'URL. En se connectant au serveur web correspondant à l'URL cible sur le port 80 (qui est le port standard prévu à cet effet) et en envoyant ladite requête, le serveur est supposé renvoyer une réponse qui contient dans son corps les données du fichier demandé, permettant d'en générer une copie locale. Dès lors, le *downloader* est capable d'effectuer ce qu'on attend de lui : des téléchargements parallélisés préalablement ordonnancés à l'aide de métadonnées.

1.3 Vue d'ensemble du système

En conclusion de cette analyse du système, le système se décompose en deux majeures parties que sont le robot de recherche (web crawler) et le **programme de téléchargement parallélisé** (downloader). Ces deux éléments étant à la fois distincts mais liés, il est important de comprendre les intéractions ayant lieu entre eux.

Le *web crawler* arpente ainsi Internet à la recherche de fichiers d'archive qu'il détecte comme suspects et les enregistre comme **cibles** accompagnées de **métadonnées** qui permettront leur classification. Le *downloader*, de son côté, récupère ces cibles, évalue un **score** qui permet de les interclasser et d'effectuer un ordonnancement de leur traitement, puis remet cette liste ordonnée à son dispatcher qui 'actionne' les requêtes de téléchargement au sein de threads séparés.

A chaque fois qu'un thread termine son travail, il génère sur le disque de la machine une copie du fichier qu'il vient de télécharger afin qu'il soit analysable par la suite avec d'autres outils dont ne nous connaissons pas les modalités de fonctionnement (cela n'étant pas un problème, puisqu'il est totalement indépendant du *downloader*).

Voici un diagramme représentant de manière synthétique ce fonctionnement :

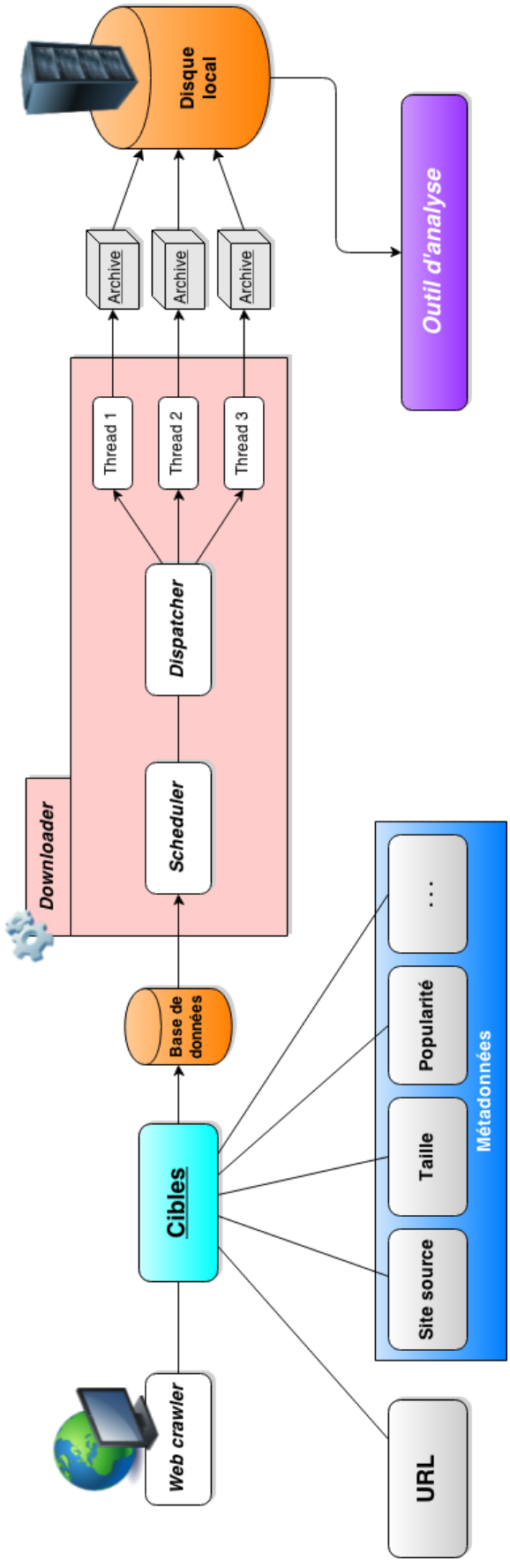

FIGURE 1.4 – Diagramme synthèse du système

Implémentation du downloader en Java

2.1 Architecture générale du programmer et style de code

Comme évoqué dans l'analyse du système, nous avons choisi Java comme langage pour l'implémentation de notre *downloader* pour plusieurs raisons :

— Sa **simplicité** (Java offre de nombreuses facilités avec l'immense collection de fonctions et d'objets qu'il propose, notamment pour la gestion du *multithreading* et des connexions réseau)

— Sa **portabilité** (n'ayant aucune information sur la machine sur laquelle ce programme pourrait hypothétiquement être utilisé, générer du code portable est un atout)

— Sa **robustesse** (avec une bonne gestion des exceptions et les outils qui sont fournis avec un IDE comme Eclipse, le debugging est aisé)

— Son **accessibilité** (n'étant pas des programmeurs chevronnés, il est pratique d'avoir un langage tolérant aux erreurs et simple dans sa syntaxe)

De plus, le fait qu'il s'agisse d'un langage très répandu permet de facilement se documenter sur de nombreux sujets, et facilite sa maintenance par d'autres personnes. C'est également dans cette optique que nous avons choisi d'écrire du code en **anglais** (y compris les commentaires) afin de le rendre compréhensible par des personnes non francophones. Nous avons également pris soin de documenter l'ensemble des méthodes créées et de commenter le code de façon parcimonieuse.

Dans cet esprit, nous avons souhaité effectuer un **découpage modulaire** propre de notre programme, afin de distinguer plusieurs parties indépendantes dans leur fonctionnement, et qui une fois 'emboîtées' produisent le logiciel final. Nous avons donc réfléchi aux différentes possibilités qui s'offraient à nous, et avons abouti aux modules suivants :

— **Target** : Classe permettant la représentation en mémoire des cibles générées par le web crawler.

— **InputModule** : Lit une source de données contenant des cibles, et génère une liste non ordonnée les contenant.

— **Scheduler** : Ordonne une liste de cibles grâce aux métadonnées qu'elles contiennent.

— **Dispatcher** : Utilise une liste ordonnée de cibles (comme celles générées par le module Scheduler) afin de répartir leur traitement sur 3 threads différents et de gérer leur exécution.

— **RunnableDownload** : Tâche exécutable sous la forme d'un thread, effectuant le téléchargement en lui-même.

Dans le système 'théorique' que nous venons d'analyser, les cibles étaient supposées se situer dans une base de données. Cependant, en pratique, nous n'avions aucune base de données permettant d'effectuer de quelconques tests et le modèle des cibles n'était pas fixé (et une grande liberté nous a donc été laissée quant à la détermination des métadonnées et au calcul du score). C'est pourquoi nous avons opté pour l'utilisation d'un **fichier texte** comme support d'entrée pour les cibles.

Cependant, le découpage modulaire est ici capital : en effet, avoir isolé le module d'entrée du reste du programme permet d'unifier le type de données qu'il produit (ici, une liste non ordonnée de cibles) sans pour autant fixer d'où celles-ci proviennent. Le reste du programme utilisera donc la liste produite par ce module sans se soucier de comment elle l'a acquise (principe d'encapsulation, ou encore de la 'boîte noire'). Ainsi, si le programme devait gérer l'obtention des cibles depuis une base de données plutôt que depuis un fichier texte, seul le code du module d'entrée **InputModule** devrait être modifié, et l'ensemble du programme serait inchangé.

Qui plus est, en plus des bénéfices 'physiques' de cette séparation, le code gagne beaucoup en 'logique', et donc en lisibilité et en maintenabilité. Ce sont quelques raisons (parmi d'autres) qui justifient l'importance que nous avons attribué à ce découpage en modules.

Voici un diagramme montrant les interactions entre ces différents modules :

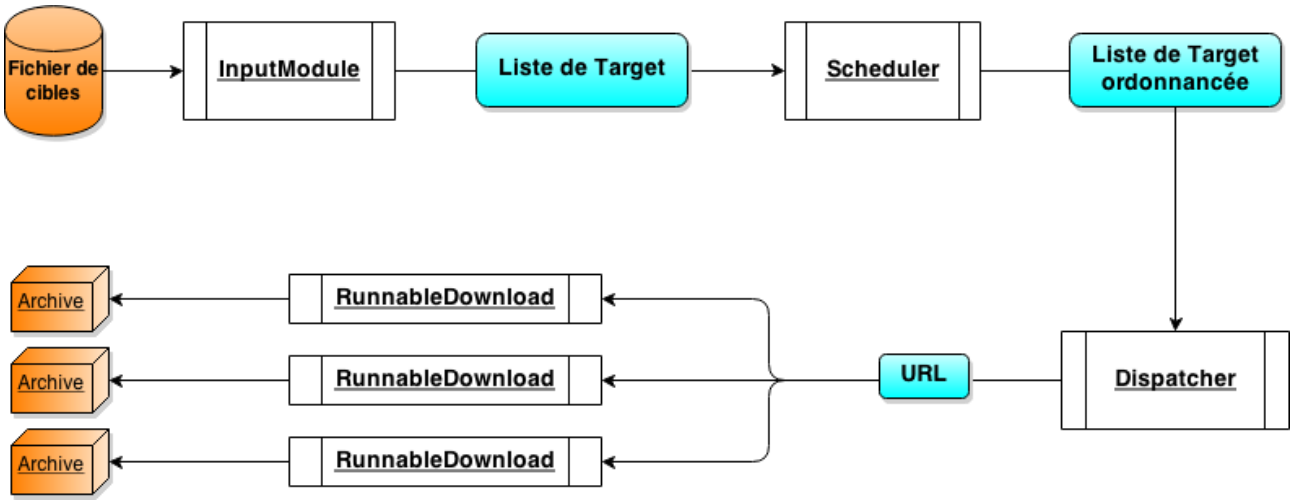

FIGURE 2.1 – Diagramme d'interaction entre les modules

2.2 Structure de données des cibles

Le downloader traite en permanence des cibles, et c'est pourquoi nous avons dû concevoir une structure de données permettant de les représenter efficacement en mémoire. C'est à ces fins que nous avons conçu la classe **Target** qui, une fois instanciée, représente une cible préalablement trouvée par le web crawler. Cette classe doit donc permettre d'accéder à tous les éléments utiles d'une telle cible (comme son URL ou ses métadonnées), et nous avons choisi d'y intégrer un système de calcul de score dynamique, ce qui sera d'une grande utilité pour le **Scheduler**.

Afin de ne pas complexifier inutilement le programme, nous avons fait le choix d'implémenter très peu de métadonnées différentes, ce qui simplifie également la formule de calcul de score. Nous aborderons cependant les perspectives envisageables à ce sujet plus loin dans ce rapport.

Nous nous sommes donc contentés des attributs (variables membres) suivants pour la classe Target :

```
// === ATTRIBUTES ==========================================================

        private URL _url;
        private double _score;

        // Metadata
        private int _fileSize;
        private int _importance;
```

FIGURE 2.2 – Déclaration des attributs de la classe Target

De façon prévisible, on aperçoit l'URL de la cible et son score calculé à partir des métadonnées, auxquels il n'était pas possible d'échapper. On remarque cependant que seules deux métadonnées ont été implémentées :

— **fileSize**, qui représente la taille du fichier en kilo-octets, est une valeur concrète et objective.

— **importance**, qui représente 'l'importance' que doit porter le fichier aux yeux du **Scheduler**, est de son côté une valeur **abstraite** qui peut être considérée comme la pondération de différentes autres métadonnées.

Ces deux métadonnées nous ont semblées suffisantes pour faire les tests qui s'imposaient vis à vis du **Scheduler**. Il est également bon de noter que nous avons tenu à respecter le **principe d'encapsulation**, et que nous avons par conséquent défini tous nos attributs comme **privés** et permis leur accès au moyen de méthodes accesseurs.

Lorsqu'une cible est instanciée, son constructeur est appelé, initialisant de façon convenable ses attributs en fonction des paramètres qui lui sont donnés. Cependant, nous tenions à ce que cette classe soit capable de calculer elle-même son score pour faciliter la tâche du **Scheduler**. C'est pourquoi nous avons créé la méthode **computeScore()** qui met à jour l'attribut score en fonction des métadonnées de l'objet. Ainsi, dans le constructeur de l'objet ainsi que dans tous les accesseurs modifiant une métadonnée, cette méthode computeScore() est automatique rappelée, permettant de toujours garder le score à jour de manière totalement transparente pour l'utilisateur de la classe.

Voici un exemple de code d'accesseur en écriture (ici, celui de **fileSize**) qui reflète cette logique :

```
public void setFileSize(int size)
{
        if(size < 0)
                size = 0;

        _fileSize = size;
        this.computeScore();
}
```

FIGURE 2.3 – Implémentation d'un accesseur en écriture de la classe Target

Et voici le code du constructeur de la classe, qui suit le même principe :

```
public Target(URL url, int fileSize, int importance)
{
	_url = url;

	if(fileSize < 0)
		fileSize = 0;
	_fileSize = fileSize;

	if(importance < 0)
		importance = 0;
	_importance = importance;

	// Update the score of the target, as we modified metadata
	this.computeScore();
}
```

FIGURE 2.4 – Implémentation du constructeur de la classe Target

Dans ces deux méthodes est appelée la fonction de calcul du score afin de mettre ce dernier à jour. En l'état, notre fonction **computeScore()** se contente de faire une simple pondération entre **l'inverse de la taille du fichier** et **l'importance**. Ainsi, plus un fichier est important et petit en taille (et donc en théorie rapide à télécharger), plus il sera priorisé par le **Scheduler**. Le détail mathématique de cette fonction sera donné plus loin dans ce rapport.

Dans l'intérêt de ce même **Scheduler**, nous avons dû rendre des cibles comparables entre elles, afin qu'elles puissent être triées (la valeur de comparaison étant bien évidemment le score). C'est dans ce but que nous avons défini que la classe **Target** implémentait l'interface standard Java Comparable. Cela nous a amené à implémenter la méthode **compareTo(...)** qui compare deux cibles et renvoie leur ordre relatif. Son code est le suivant :

```
public int compareTo(Target other) {
	if(this._score > other._score)
		return -1;
	else if(this._score < other._score)
		return 1;
	return 0;
}
```

FIGURE 2.5 – Implémentation de l'interface Comparable au moyen de la fonction compareTo

Ainsi, comme on peut le voir dans ses embranchements conditionnels, la fonction renvoie -1 (la cible courante est 'plus petite' que la cible 'other') si le score de la cible courante est supérieur à celui de l'autre cible, et 1 dans le cas contraire. La logique voudrait que qu'une cible soit 'plus grande' si son score est plus grand, mais cette astuce sera expliquée par la suite. Le cas où 0 est retourné signifie une égalité parfaite entre les deux éléments, et dans ce cas précis, leur positionnement relatif est indifférent.

Grâce à l'implémentation de cette interface, nous avons pu drastiquement simplifier le code du Scheduler en le réduisant à une unique ligne :

```
public static void scheduleTargets(ArrayList<Target> targetList)
{
        // We use the fact that Target implements the Comparable interface to use this
        // sorting method
        Collections.sort(targetList);
}
```

FIGURE 2.6 – Implémentation du Scheduler

En effet, la fonction **Collections.sort(ArrayList<T> list)** est capable de trier **list** uniquement si le type T des données qu'elle contient implémente l'interface **Comparable**. Grâce à cette interface, cette fonction est assurée que le type T implémente la méthode **compareTo(...)** qu'elle utilise successivement sur les éléments afin d'appliquer un dérivé de l'algorithme de tri fusion. Cette façon de faire est donc à la fois élégante et très performante. Cependant, la fonction **Collections.sort(...)** trie naturellement par ordre croissant, et nous souhaitons obtenir une liste triée par ordre décroissant (où la cible ayant le plus haut score se situe en premier). C'est pourquoi nous avons 'inversé' la logique de **compareTo(...)** comme mentionné lors de son analyse (en renvoyant qu'une cible est 'plus petite' si son score est plus grand).

Ainsi, un simple appel vers cette fonction **scheduleTargets(...)** suffit à trier la liste non ordonnée de cibles passée en paramètre, et ce selon les scores calculés automatiquement dans le constructeur et les accesseurs en écriture de **Target**. On obtient alors une liste ordonnancée des cibles à traiter qui n'attend qu'à être confiée au dispatcher afin que les téléchargements soient bel et bien effectués.

2.3 Implémentation du dispatcher

Le *dispatcher* est une partie capitale de notre *downloader* qui gère la majorité du travail, puisque c'est lui qui s'occupe d'amorcer les téléchargements tout en faisant bien attention à prendre en compte la notion de *multithreading*.

Pour gérer le nombre de *threads* en action et les limiter à un certain nombre, il nous a semblé judicieux d'utiliser un **sémaphore**. En effet, ces primitives de synchronisation fonctionnant au moyen d'un compteur sont très efficaces pour représenter une ressource présente en quantité limitée. Ici, les ressources initialement en présence sont 3 *threads* 'allouables', puis à chaque nouvelle création d'un thread, on requiert l'accès à l'une de ces trois instances de la ressource. Si une est présente, on décrémente le compteur du sémaphore, et à l'inverse, si aucun n'est disponible (si 3 téléchargements sont déjà en cours en parallèle), le sémaphore 'bloque' le dispatcher jusqu'à ce que l'un des téléchargements soit terminé et 'libère un thread'.

Cette façon de gérer de le multithreading impose de fixer un nombre de threads maximaux, mais il sera étayé plus loin dans ce rapport les améliorations possibles à un tel fonctionnement, en s'adaptant dynamiquement à plusieurs paramètres.

Le code de la fonction qui s'occupe d'effectuer la répartition des threads est la suivante :

```
public static void dispatchTargets(ArrayList<Target> targets)
{
	for(Target t : targets)
	{
		// Wait for semaphore to allow downloading
		try {
			System.out.println(t.getURL() + " acquiring.");
			_threadSemaphore.acquire();
		} catch (InterruptedException e) {
			e.printStackTrace();
		}

		Thread thread = new Thread(new RunnableDownload(t.getURL()));
		thread.start();
	}
}
```

FIGURE 2.7 – Implémentation du Dispatcher

Ce code plutôt simple est un exemple flagrant de l'efficacité de Java à gérer les problématiques de multithreading et de synchronisation comparé à d'autres langages plus bas niveau. En effet, des objets **Thread** et **Semaphore** sont nativement présents et permettent de gérer ces aspects avec beaucoup de simplicité. Ainsi, le comportement des sémaphores détaillé précédemment est pratiquement entièrement géré par la seule méthode **acquire()**. Cette méthode bloquante ne permet d'atteindre la création et l'exécution d'un thread (situés quelques lignes plus bas) seulement si une instance de la ressource 'threads disponibles' est présente.

La tâche qui est confiée au thread est un objet de type **RunnableDownload**, qui est notre dernier module. Cette classe nécessite la seule URL de l'emplacement du fichier et s'occupe de tout le travail de connexion au serveur web et de téléchargement, comme nous allons le voir en détail.

2.4 Téléchargement via requête HTTP

La gestion des téléchargements grâce à des requêtes HTTP s'est révélé être l'aspect le plus technique (en terme de programmation) de l'ensemble du projet. Il a fallu judicieusement utiliser l'ensemble des ressources à notre disposition afin que le parallélisme devienne un atout plutôt qu'une surcharge inutile. En effet, par exemple, une mauvaise gestion du disque dur peut entraîner une quantité extrême d'accès disque, ce qui ralentit considérablement l'ensemble des *threads* qui doivent patienter chacun leur tour afin d'écrire sur le disque dur.

Comme aperçu dans le code du Dispatcher qui précède, la classe **RunnableDownload** représente une tâche exécutable par un objet **Thread**. Cela est rendu possible grâce au fait que **RunnableDownload** implémente l'interface standard **Runnable** qui impose l'existence de la seule fonction **run()** qui représente la tâche à exécuter. Ainsi, l'objet Thread est assuré que la classe **RunnableDownload** implémente cette méthode run() et peut donc l'appeler au sein d'un nouveau fil d'exécution, mettant en place le mécanisme de parallélisme.

Le code de cette méthode **run()** étant plutôt long, nous allons le voir en plusieurs étapes et détailler chacune d'entre elles.

```
// Find the IP address corresponding to the target host name
String IPAddress = InetAddress.getByName(_targetURL.getHost()).getHostAddress();

// Try to establish a connection with distant web server
Socket socket = new Socket(IPAddress, 80);

// Check for connection
if(!socket.isConnected())
{
	System.out.println("[EXCEPTION] Connection to server failed at <" + IPAddress + ">.");
	socket.close();
	throw new Exception();
}
```

FIGURE 2.8 – Connexion au serveur web

Cette première partie est une mise en place, elle se charge de se connecter au moyen d'un Socket au serveur web sur le port 80. Cependant, pour pouvoir s'y connecter, il faut d'abord connaître l'adresse IP du serveur, ce qui n'est pas notre cas (puisque nous ne possédons que l'URL y aboutissant). C'est pourquoi nous devons résoudre le lien grâce à une requête DNS afin d'obtenir l'adresse IP correspondant au serveur désigné par l'hôte contenu dans l'URL : cela est réalisé de façon entièrement hermétique par la fonction statique **InetAddress.getByName(URL)**. Une fois cette IP récupérée, il ne nous reste qu'à effectuer une tentative de connexion, et si celle-ci échoue, on envoie une exception gérée à la fin de la fonction dans un bloc **catch** adapté.

Dans le cas (majoritaire) où la connexion est établie avec succès, ce fil d'exécution se charge de préparer un espace adapté pour recevoir le fichier sur le disque, ainsi que d'associer un flux à l'entrée du socket (afin de capturer les données provenant du serveur). Nous avons choisi que les fichiers téléchargés soient placés dans une arborescence où chaque dossier représente un site, et ces fichiers sont placés dans le dossier correspondant au site d'où ils proviennent. Cela est réalisé par le code suivant :

```
// Prepare a file on the disk where to write the contents
File dir = new File(BASE_LOCATION + _targetURL.getHost());
if(!dir.exists())
	dir.mkdirs();

String filename = _targetURL.getPath().substring(1).replace('/', '_');
if(filename.isEmpty())
	filename = "index.html";
File file = new File(BASE_LOCATION + _targetURL.getHost() + "/" + filename);
if(!file.exists())
	file.createNewFile();

InputStream is = socket.getInputStream();
FileOutputStream fos = new FileOutputStream(file);
```

FIGURE 2.9 – Mise en place des fichiers et du flux d'entrée

Le travail effectué par cette section est plutôt rébarbatif, car elle doit s'assurer de la présence des répertoires où est censé arriver le fichier, puis créer le fichier en lui-même avant de 'connecter' le flux entrant du socket à un flux de sortie vers le fichier. Il est évidemment possible que le code de gestion du système de fichiers génère des exceptions (dans le cas où le programme n'aurait par exemple pas les droits suffisants pour créer des fichiers à cet emplacement), ce qui est également intercepté par les blocs **catch** en fin de fonction.

Maintenant que tout est en place, il nous suffit de générer la requête HTTP appropriée suivant le modèle donné précédemment dans ce document, puis de l'envoyer au serveur :

```
PrintWriter request = new PrintWriter( socket.getOutputStream() );
request.print( "GET " + _targetURL.getPath() + " HTTP/1.1\r\n" +
               "Host: " + _targetURL.getHost() + "\r\n" +
               "Connection: close\r\n" +
               "User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X; de-de)
               AppleWebKit/523.10.3 (KHTML, like Gecko) Version/3.0.4 Safari/523.10\r\n" +
               "\r\n" );
request.flush();
```

FIGURE 2.10 – Envoi de la requête HTTP

Ce code utilise un objet d'écriture via buffer (**PrintWriter**) qui est 'connecté' au flux sortant du socket, afin que ce qui est écrit soit envoyé directement au serveur. Nous générons ensuite la requête exactement comme le modèle précédemment établi le spécifie, puis forçons le vidage des buffers avec la méthode **flush()** afin de s'assurer que tout soit bien envoyé.

Il ne nous reste désormais qu'à attendre une très probable réponse du serveur, puis la traiter afin d'en récupérer le corps et le placer dans le fichier préalablement préparé. Pour cela, nous devons nous 'débarasser' de l'en-tête de la réponse afin de ne stocker que le corps dans la réponse, et c'est ici qu'interviennent les **deux sauts de lignes consécutifs** qui marquent la fin de cet en-tête.

Afin de la trouver efficacement, nous utilisons une file (liste FIFO - 'first in, first out') dans laquelle, au fur et à mesure de la lecture séquentielle de l'en-tête, nous plaçons systématiquement chaque caractère lu tout en retirant le plus ancien élément de la file afin de conserver uniquement 4 caractère. Nous pouvons alors vérifier à chaque itération si les 4 derniers caractères lus (le contenu de la file) équivalents aux caractères spéciaux : **CR LF CR LF** (correspondant à deux sauts de ligne).

Le code de séparation de l'en-tête est le suivant :

```
LinkedList<Integer> fourLastBytes = new LinkedList<Integer>();
int c = -1;
boolean foundHeaderEnd = false;

// Insert 4 fake characters to make its size equal to 4 from the start
fourLastBytes.add(0); fourLastBytes.add(0); fourLastBytes.add(0); fourLastBytes.add(0);

// Insert and remove characters appropriately from the FIFO list
while (!foundHeaderEnd && (c = is.read()) != -1)
{
        fourLastBytes.add(c);
        fourLastBytes.removeFirst();
```

```
        // Search for a double CRLF to detect the end of the HTTP header
        if(fourLastBytes.get(0) == 0x0D && fourLastBytes.get(1) == 0x0A
        && fourLastBytes.get(2) == 0x0D && fourLastBytes.get(3) == 0x0A)
                foundHeaderEnd = true;
}

if(!foundHeaderEnd)
{
        socket.close();
        fos.close();
        System.out.println("[EXCEPTION] Invalid HTTP response from the server <" +
                                _targetURL.getHost() + ">.");
        throw new Exception();
}
```

FIGURE 2.11 – Séparation de l'en-tête du corps de la réponse HTTP

L'objet **LinkedList** est un conteneur de la bibliothèque standard de Java servant à représenter les listes FIFO, ce qui correspond exactement à notre besoin. Le reste du code suit la logique présentée à l'instant, au détail près que nous devions gérer le cas où le serveur renvoie une réponse 'non-standard' qui ne possède pas une mise en forme classique (et donc où le programme ne trouverait pas le double saut de ligne).

Le plus compliqué étant fait, il ne reste plus qu'à copier le corps de la réponse HTTP dans un fichier pour finaliser le téléchargement. Cependant, l'effectuer caractère par caractère obligerait à effectuer des accès disque très nombreux et systématiques, ce qui augmente les possibilités de 'blocage' des threads en attente d'avoir accès au disque dur pour pouvoir y écrire leur fichier.

C'est pourquoi nous avons dû mettre en place un système de buffer intermédiaire afin d'y faire transiter jusqu'à 1Mo de données (ce choix est arbitraire, mais suffisamment grand pour être efficace) qui sont ensuite écrites en une seule fois sur le disque. Une fois l'ensemble de cette 'copie' effectuée, il ne reste plus qu'à fermer les différents flux (socket et fichier) pour terminer proprement cette tâche :

```
// Now that we found the end of the HTTP header, we can copy the contents
// of the request answer. We write it into the previously setup local file.
byte[] buffer = new byte[BUFFER_SIZE];
int amountRead = 0;
while ((amountRead = is.read(buffer, 0, BUFFER_SIZE)) != -1)
        fos.write(buffer, 0, amountRead);

socket.close();
fos.close();
System.out.println("Target download finished : " + _targetURL);

// This thread has ended, decrement the amount of threads
Dispatcher._threadSemaphore.release();
```

FIGURE 2.12 – Copie du corps de la réponse HTTP vers un fichier et finalisation du téléchargement

On remarque également que la dernière ligne de code de cette exemple consiste à relâcher une instance de la ressource 'Threads allouables' puisque ce thread s'apprête à se terminer. Ainsi, un éventuel autre

téléchargement en attente a la possibilité de débuter son exécution, ou si aucun n'est en attente, le compteur du sémaphore s'incrémente.

Nous avons volontairement omis le code du module **InputModule** (se chargeant de lire un fichier de cibles afin d'instancier les objets correspondants en mémoire) car il s'agit d'un code plutôt massif et sans grand intérêt technique (il se contente de lire dans le fichier et d'en analyser le contenu), nous avons donc jugé qu'il n'apportait pas suffisamment d'informations utiles pour être placé dans cette partie liée à l'implémentation. Nous l'avons cependant mis en annexe pour ceux qui seraient intéressés quant à son contenu.

Outre cette petite exception, nous avons couvert la quasi-globalité du code produit dans le cadre de ce projet. Nous allons désormais analyser un exemple d'exécution en fixant une entrée et en observant les mécanismes du programme se mettre en action.

2.5 Exemple d'exécution

Pour cet exemple d'exécution du programme, nous avons pris en compte le fichier de cibles suivant :

```
1 http://upload.wikimedia.org/wikipedia/commons/d/da/Herbert_Garland_DYK.png;45;0
2 http://polytech.univ-tours.fr/medias/fichier/m2ri-eme_1397481514777-pdf;1899;0
3 http://www.synopsite.com/file/rda14.zip;86426;0
4 https://github.com/rainmeter/rainmeter/releases/download/v3.1.0.2190/Rainmeter-3.1.exe;9383;10
5 http://download.tuxfamily.org/notepadplus/6.6.4/npp.6.6.4.bin.zip;6272;0
6 http://www.synopsite.com/file/rda05.zip;18432;0
```

FIGURE 2.13 – Contenu du fichier d'entrée

L'ensemble des 'importances' des cibles ont volontairement été définies comme nulles, excepté pour une cible (n°4) qui pèse 9383 Ko (ce qui est relativement massif au vu des autres cibles). Si cette cible avait eu la même importance que toutes les autres, elle aurait été dépréciée car plutôt volumineuse, et donc traitée parmi les dernières. Cependant, l'importance de 10 qui lui a été fixée sera capable de compenser cela et servira de démonstration quant au bon fonctionnement de la méthode de calcul de score.

Lorsque l'on exécute le programme, celui-ci nous donne la sortie présente sur la page suivante (colorisée afin de simplifier la lecture).

Au début de la sortie, on remarque la présence du résultat de la lecture du fichier source par le programme : celui-ci nous confirme les cibles qu'il a trouvé dans le fichier, et affiche également le score qu'il leur a associé. On observe donc que la petite image PNG possède un score de plus de 356, ce qui est bien au delà de toutes les autres cibles (ce qui semble cohérent, puisque sa taille était bien inférieure à tous les autres fichiers). Puis, en deuxième place vient une cible intéressante : le fichier EXE auquel nous avons donné une importance supérieure avec un score de 19. Il passe ainsi devant de nombreuses autres cibles bien plus légères que lui, ce qui confirme le bon fonctionnement du système de ce point de vue.

```
=============== Multithreaded downloader ===============
Target found : http://upload.wikimedia.org/wikipedia/commons/d/da/Herbert_Garland_DYK.png (score = 356.17391304347825).
Target found : http://polytech.univ-tours.fr/medias/fichier/m2ri-eme_1397481514777-pdf (score = 8.623157894736842).
Target found : http://www.synopsite.com/file/rda14.zip (score = 0.18957038888310365).
Target found : https://github.com/rainmeter/rainmeter/releases/download/v3.1.0.2190/Rainmeter-3.1.exe (score = 19.20545
Target found : http://download.tuxfamily.org/notepadplus/6.6.4/npp.6.6.4.bin.zip (score = 2.6118284712258886).
Target found : http://www.synopsite.com/file/rda05.zip (score = 0.8888406661964954).

http://upload.wikimedia.org/wikipedia/commons/d/da/Herbert_Garland_DYK.png acquiring.
https://github.com/rainmeter/rainmeter/releases/download/v3.1.0.2190/Rainmeter-3.1.exe acquiring.
http://polytech.univ-tours.fr/medias/fichier/m2ri-eme_1397481514777-pdf acquiring.
http://download.tuxfamily.org/notepadplus/6.6.4/npp.6.6.4.bin.zip acquiring.
Starting target download : http://polytech.univ-tours.fr/medias/fichier/m2ri-eme_1397481514777-pdf
Starting target download : https://github.com/rainmeter/rainmeter/releases/download/v3.1.0.2190/Rainmeter-3.1.exe
Starting target download : http://upload.wikimedia.org/wikipedia/commons/d/da/Herbert_Garland_DYK.png
Target download finished : http://upload.wikimedia.org/wikipedia/commons/d/da/Herbert_Garland_DYK.png
http://www.synopsite.com/file/rda05.zip acquiring.
Starting target download : http://download.tuxfamily.org/notepadplus/6.6.4/npp.6.6.4.bin.zip
Target download finished : https://github.com/rainmeter/rainmeter/releases/download/v3.1.0.2190/Rainmeter-3.1.exe
http://www.synopsite.com/file/rda14.zip acquiring.
Starting target download : http://www.synopsite.com/file/rda05.zip
Target download finished : http://polytech.univ-tours.fr/medias/fichier/m2ri-eme_1397481514777-pdf
Starting target download : http://www.synopsite.com/file/rda14.zip
Target download finished : http://download.tuxfamily.org/notepadplus/6.6.4/npp.6.6.4.bin.zip
Target download finished : http://www.synopsite.com/file/rda05.zip
Target download finished : http://www.synopsite.com/file/rda14.zip
=============== END OF EXECUTION ===============
```

FIGURE 2.14 – Sortie émise par le programme

La deuxième partie de la sortie (qui est colorisée) sert à suivre le déroulement du programme sur ses différents fils d'exécution. Les lignes rouges sont les **acquisitions de threads**, les jaunes représentent le **démarrage de la tâche de téléchargement** et les vertes signalent la **fin du téléchargement d'une cible**. On observe alors que jamais plus de 3 téléchargements ne sont présents en même temps, et qu'un nouveau téléchargement ne se lance que lorsqu'un autre vient de se terminer.

Le thread principal ne s'achève que lorsque l'ensemble des threads de téléchargement se sont achevés. Une fois le programme terminé dans son ensemble, on peut aller dans le répertoire 'C :/DOWNLOADS/' (modifiable par le biais d'une simple constante dans le code) pour confirmer la présence des fichiers téléchargés. Ceux-ci sont disposés selon l'arborescence suivante :

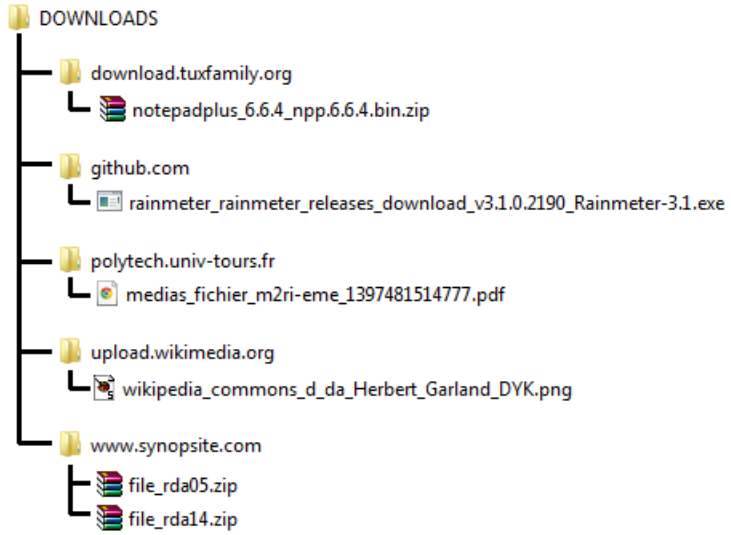

FIGURE 2.15 – Arborescence de fichiers obtenue une fois l'exécution achevée

A titre informatif, cette exécution s'est réalisée en 1 minute et 26 secondes, pour un total de 110 Mo de données téléchargées avec une connexion ayant un débit de 1.35 Mo/s en moyenne. Cela nous donne une **utilisation théorique de la connexion** aux alentours des **95%**, ce qui est très satisfaisant.

2.6 Optimisations possibles

Le programme obtenu est, comme le montre l'exemple précédent, fonctionnel et remplit le cahier des charges du projet (qui demandait le développement d'un logiciel de **téléchargement parallélisé** intégrable au sein d'un système donné). Cependant, le temps qui nous était imparti pour réaliser ce projet et nos connaissances techniques nous ont poussé à effectuer des compromis, bien que nous ayons de nombreuses idées pour améliorer et optimiser le fonctionnement du programme.

Tout d'abord, concernant les métadonnées et le calcul de score. La fonction actuelle est la suivante :

$$score = ((importance + 1) * 2^14)/(taille + 1);$$

Cette fonction est simpliste mais à rempli son office, car nous avons volontairement limité le nombre de métadonnées en les regroupant toutes dans une unique valeur abstraite, l'**importance**. D'autres métadonnées pourraient être imaginées (comme celles déjà données dans la partie d'analyse du système), et cette fonction objectif pourrait être grandement affinée, donnant un ordonnancement d'autant plus performant. Une étude poussée serait nécessaire pour arriver à un résultat de qualité.

De la même manière, dans une volonté de simplicité, nous avons fixé le nombre maximal de threads à une valeur constante, qui a été arbitrairement fixée à 3. Cependant, l'objectif de la parallélisation étant de pousser au maximum l'utilisation de la connexion tout en gardant des performances optimales du côté du CPU et de l'accès aux ressources, il pourrait être intéressant de farie varier dynamiquement ce nombre maximal de threads en fonction d'une analyse régulière de l'état de la connexion, ainsi que de la puissance CPU disponible ou encore de statistiques telles que le temps moyen d'attente pour accéder au disque au sein des threads.

Cette modification permettrait d'obtenir un rendement optimal en toutes circonstances, ce qui est un atout de taille une fois intégré dans un système concret.

Pour pouvoir être intégré dans un tel système, il serait intéressant de rendre le programme plus **autonome**. En effet, en l'état, il nécessite qu'un administrateur l'exécute, celui-ci traite une liste fixe de cibles puis termine son exécution. Au sein du système final, il serait préférable que l'application soit lancée une unique fois et récupère les cibles depuis une base de données et sous la forme d'un **flux**. De cette manière, l'ajout de cibles dans la base de données par le web crawler permettrait au downloader de la traiter pratiquement instantanément à partir du moment où elle apparaît dans la base de données.

Cela modifierait la structure du programme de manière profonde, mais le découpage en modules permet encore une fois de rentre une telle migration plus aisée.

Conclusion

En conclusion, nous sommes parvenus à concevoir un programme répondant au problème initial et avons réussi à atteindre des performances honorables. En ce sens, nous pensons être parvenus à notre but.

Ce projet a été un travail enrichissant à plusieurs niveaux :

— L'aspect d'analyse du système nous a obligé à nous appliquer des contraintes réalistes en nous renseignant sur l'architecture d'un système concret et ses finalités. Nous avons dû nous documenter sur de nombreux aspects qui nous étaient inconnus pour la plupart afin de pouvoir avoir une vue d'ensemble de celui-c
— De façon plus pragmatique, ce projet a perfectionné nos connaissances sur le langage Java, nous poussant à découvrir certaines facettes de sa bibliothèque et à mettre en pratique les aspects **multithreading** et **réseau**, qui sont tous deux très importants dans les logiciels d'aujourd'hui.
— Enfin, ce projet a amélioré notre faculté à collaborer afin de trouver des solutions à nos problèmes, ce qui est très important en vue des métiers de l'ingénieur.

Nous avons également tenu à soigneusement documenter et commenter le code produit afin qu'il puisse éventuellement évoluer dans le cadre de futurs projets, et également afin de pouvoir plus facilement corriger de possibles bugs que nous aurions laissé passer.

Comme évoqué dans la partie finale, le logiciel peut être amélioré sur de très nombreux aspects, mais peu de travail serait nécessaire à son intégration dans un système concret.

Annexes

A.1 Références

Informations sur les standards d'exclusion des robots (**robots.txt**) :

- https://developers.google.com/webmasters/control-crawl-index/docs/robots_txt

Spécification du protocole HTTP :

- http://www.w3.org/Protocols/rfc2616/rfc2616.html

Les images des contenus de fichiers (entrée et sortie dans l'exemple d'exécution) ont été prises dans l'éditeur de texte libre **Notepad++** :

- http://www.notepad-plus-plus.org/

Les diagrammes de ce rapport ont été réalisés grâce à l'outil gratuit **draw.io** :

- https://www.draw.io/

A.2 Documents annexes

```
public static ArrayList<Target> loadFromDatabase()
{
	ArrayList<Target> targets = new ArrayList<Target>();

	try {
		File inputFile = new File("C://targets.txt");
		BufferedReader br = new BufferedReader(new FileReader(inputFile));

		int lineID=0;
		String line;
		while ((line = br.readLine()) != null) {
			// Each line of the file corresponds to one target
			lineID++;
			String[] split = line.split(";");

			if(split.length < 3) {
				System.out.println("[WARNING] Target on line " + lineID + " has
					not enough parameters.");
			}
			else {
				try {
					// Parse the line to find target URL and metadata
					URL url = new URL(split[0]);
					int fileSize = Integer.parseInt(split[1]);
					int importance = Integer.parseInt(split[2]);

					Target t = new Target(url, fileSize, importance);

					System.out.println("Target found : " +
						t.getURL().toExternalForm() +
						" (score = " + t.getScore() + ").");
					targets.add(t);

				} catch (MalformedURLException e) {
					System.out.println("[WARNING] Target URL \"" + split[0] +
						"\" is malformed (on line " + lineID + ").");
				} catch (NumberFormatException e) {
					System.out.println("[WARNING] Target parameter could not be parsed
						as an integer (on line " + lineID + ").");
				}
			}
		}
		br.close();
	} catch (FileNotFoundException e) {
		System.out.println("[ERROR] Input file couldn't be opened.");
	} catch (IOException e) {
		System.out.println("[ERROR] A problem occured while reading input file.");
	}
	return targets;
}
```

FIGURE A.1 – Code de la fonction de lecture de fichiers d'InputModule

Développement d'un logiciel de téléchargement parallélisé pour la protection du copyright des mangas

Département Informatique
3e année
2013 - 20014

Rapport de projet

Résumé : Ce rapport détaille notre démarche dans un projet consistant en l'étude d'un système de protection de copyright pour les mangas, ainsi que le développement d'une application de téléchargement parallélisé destiné à être intégrée dans ce même système.

Mots clefs : copyright, manga, téléchargement, parallélisme

Abstract: This report details our work on a project aiming to analyze a manga copyright protection system, along with the development of a multithreaded downloader which will be integrated inside of this system.

Keywords: copyright, manga, downloading, multithread

Encadrant
Mathieu Delalandre
mathieu.delalandre@gmail.com

Université François-Rabelais, Tours

Étudiants
Anthony DEMARCY
anthony.demarcy@etu.univ-tours.fr
Florian MONTALBANO
florian.montalbano@etu.univ-tours.fr

DI3 2013 - 2014