Ecole Polytechnique de l'Université de Tours
Département Informatique
64 avenue Jean Portalis
37200 Tours, France
Tél. +33 (0)2 47 36 14 14
polytech.univ-tours.fr

Projet Architecture, Système et Réseaux

2019-2020

Crawler XMLTV

Tuteur académique
Mathieu Delalandre

Étudiants
Lucien Le Guellec (DI5)
Romain Hérault (DI5)

8 janvier 2020

Liste des intervenants

Nom	Email	Qualité
Lucien Le Guellec	lucien.leguellec@etu.univ-tours.fr	Étudiant DI5
Romain Hérault	romain.herault2@etu.univ-tours.fr	Étudiant DI5
Mathieu Delalandre	mathieu.delalandre@univ-tours.fr	Tuteur académique, Département Informatique

Avertissement

Ce document a été rédigé par Lucien Le Guellec et Romain Hérault susnommés les auteurs.

L'Ecole Polytechnique de l'Université de Tours est représentée par Mathieu Delalandre susnommé le tuteur académique.

Par l'utilisation de ce modèle de document, l'ensemble des intervenants du projet acceptent les conditions définies ci-après.

Les auteurs reconnaissent assumer l'entière responsabilité du contenu du document ainsi que toutes suites judiciaires qui pourraient en découler du fait du non respect des lois ou des droits d'auteur.

Les auteurs attestent que les propos du document sont sincères et assument l'entière responsabilité de la véracité des propos.

Les auteurs attestent ne pas s'approprier le travail d'autrui et que le document ne contient aucun plagiat.

Les auteurs attestent que le document ne contient aucun propos diffamatoire ou condamnable devant la loi.

Les auteurs reconnaissent qu'ils ne peuvent diffuser ce document en partie ou en intégralité sous quelque forme que ce soit sans l'accord préalable du tuteur académique et de l'entreprise.

Les auteurs autorisent l'école polytechnique de l'université François Rabelais de Tours à diffuser tout ou partie de ce document, sous quelque forme que ce soit, y compris après transformation en citant la source. Cette diffusion devra se faire gracieusement et être accompagnée du présent avertissement.

Pour citer ce document

Lucien Le Guellec et Romain Hérault, *Crawler XMLTV:* , Projet Architecture, Système et Réseaux, Ecole Polytechnique de l'Université de Tours, Tours, France, 2019-2020.

```
@mastersthesis{
    author={Le Guellec, Lucien and Hérault, Romain},
    title={Crawler XMLTV: },
    type={Projet Architecture, Système et Réseaux},
    school={Ecole Polytechnique de l'Université de Tours},
    address={Tours, France},
    year={2019-2020}
}
```

Document mis en forme sous LaTeX
grâce à la classe de document polytech.cls V2.2.5 (2017/05/02)

Table des matières

Table des figures

1 Introduction

1 Acteurs, enjeux et contexte

Ce document est le rapport concernant le projet ASR de Lucien Le Guellec et de Romain Hérault dont le client et tuteur pédagogique était M. Mathieu Delalandre.

Ce projet entre dans le cadre de la création d'une future *start-up* appelée Todd.tv dont l'objectif est de concevoir un guide TV disposant de fonctionnalités supplémentaires par rapport aux guides déjà présents sur internet. Ces fonctionnalités peuvent être d'afficher des informations supplémentaires sur un futur programme (des actualités sur les acteurs par exemple) ou indiquer si le programme est présentement interrompu par une publicité. Le site proposerait aussi une application permettant de faciliter l'utilisation de la télévision en prévenant l'utilisateur quand un programme reprend après une publicité.

Noter projet intervient dans la partie guide, pour récupérer des images de programmes afin par exemple d'en faire des vignettes.

2 Objectifs

Pour ce faire, il nous fallait télécharger l'ensemble des images d'illustration de programmes télévisés présentes sur le site internet de Télérama afin de les inclure dans l'éventuel futur guide TV. Fort heureusement, il existe déjà un document XML qui répertorie tous ces programmes, avec l'URL des images en question, et qui est notamment mis à jour quotidiennement sur le site XMLTV France. Notre programme devait donc tourner continuellement et télécharger ce fichier XMLTV quand il est mis à jour, télécharger les images qui y sont listées et les redimensionner pour les garder dans un dossier tenu à jour, sans avoir à retélécharger toutes les images quotidiennement, puisque la liste des programmes à venir n'est renouvelée que partiellement.

Sur ce projet, le langage Java a été imposé.

2 Analyse et architecture

1 Architecture de l'application

Nous avions d'abord réalisé un programme téléchargeant le fichier XMLTV, le parsant et créant des *threads* pour télécharger toutes les images. L'opération pouvait être lancée périodiquement, et il ne se passait rien entre les exécutions. Néanmoins, nous avons voulu rendre les différentes parties du programme plus indépendantes, avec un fonctionnement continu naturel plutôt que des exécutions ponctuelles. Plutôt qu'un algorithme qui appelle les différentes fonctions de lecture du fichier XMLTV puis le téléchargement des images les unes après les autres, nous avons divisé notre programme en modules.

2 Problème des producteurs et des consommateurs

Cette architecture modulaire pose cependant un problème de communication entre les modules, qui devait être plus souple que pour notre première version. En effet, chaque module aurait besoin de récupérer des données provenant d'autres modules et fournir de nouvelles données à un autre module. Par exemple, un module se charge d'extraire une liste de liens, un autre utilise ces liens pour télécharger des images, un troisième module se charge de redimensionner ces images, etc.

Ce problème appartient à la problématique producteur/consommateur qui se présente lorsque l'on a une ou plusieurs entités (les producteurs) qui produisent des éléments et une ou plusieurs autres entités qui les utilisent (des consommateurs). C'est un problème de synchronisation de ressources et il peut apparaître notamment lorsque l'on fait du *multi-threading*, comme dans notre cas : chaque type de données qui peut être communiqué d'un module à l'autre peut être produit par n'importe quel *thread* du module producteur, et consommé par n'importe quel *thread* du module consommateur. Il faut donc une interface unique entre chaque module, accessible simultanément à plusieurs producteurs (qui l'enrichissent) et plusieurs consommateurs (qui s'y servent).

3 Blocking queue

Nous avons trouvé une solution à notre problème dans le *package* java.util.concurrent : l'interface *BlockingQueue*. Une *BlockingQueue* est une file dans laquelle différents *threads* peuvent prendre ou déposer des objets. L'avantage d'utiliser un tel objet est que les méthodes permettant de récupérer ou déposer des objets dans la files sont bloquantes. Ainsi, si l'on veut déposer un objet dans la file mais qu'elle est déjà pleine, alors la méthode attend qu'une place se libère avant de déposer l'objet. De la même manière, si l'on veut récupérer un objet dans la file mais qu'elle est vide, alors la méthode se met en attente jusqu'à ce qu'un objet y soit déposé.

Un autre avantage de l'interface *BlockingQueue* est qu'elle est automatiquement synchronisée, on peut donc utiliser une *BlockingQueue* avec plusieurs *threads* sans risque.

L'interface *BlockingQueue* possède plusieurs implémentations en Java. Celle que nous avons utilisée est la *LinkedBlockingQueue*. Il s'agit d'une file dont les éléments sont ordonnés selon la règle FIFO (*first-in-first-out*).

Utilisation

Dans notre programme, chaque module est exécuté au sein d'un ou de plusieurs *threads* qui lui sont propre. Ces *threads* produisent et/ou consomment des données qui transitent à travers les *BlockingQueue*.

Les *BlockingQueue* nous permettent aussi de faciliter l'utilisation du *multi-threading* dans les cas où l'on souhaite optimiser les performances.

Par exemple, on utilise du *multi-threading* pour télécharger les images. On crée plusieurs *threads* qui téléchargent chacun une image et recommencent avec une autre image lorsqu'ils ont fini. Pour avoir le lien de l'image qu'il doit télécharger, un *thread* se sert directement dans une *BlockingQueue*. Ainsi, lorsqu'un *thread* prend un lien dans la file, il en est automatiquement enlevé et les autres *threads* ne le voient plus. De plus lorsque la file est vide, les *threads* de téléchargement se mettent en attente jusqu'à ce qu'il y ait des nouveaux liens, qui les réveillent automatiquement quand ils sont ajoutés à la *BlockingQueue*.

3 Conception détaillée

Notre programme est divisé en différents modules, qui correspondent essentiellement à des *threads* ou des groupes de *threads* et communiquent principalement par des *blocking queues*, comme montré sur la Figure 1 que nous allons détailler.

1 Téléchargement du XMLTV

Le module Downloader s'occupe de télécharger périodiquement le fichier XMLTV via le site XMLTV France et de le parser.

Pour faire cela, on étend la classe *TimerTask* qui nous permet de programmer l'exécution de notre code avec un *timer*. Ainsi, le code sera appelé en boucle, et tous les appels seront séparés par un laps de temps défini dans le fichier de configuration. L'appel d'une *TimerTask* se fait dans un *thread* séparé.

On peut paramétrer l'heure du premier téléchargement dans le fichier de configuration (balises <*start*>) et indiquer la durée entre deux téléchargements (balise <*refreshTime*>).

Un mécanisme a été mis en place pour éviter de retélécharger le fichier XMLTV si celui-ci n'a pas été mis à jour sur le site web : on se fie à la date de dernière mise à jour sur le site web. Dès qu'on télécharge le fichier, on sauvegarde la date courante, et lorsque l'on veut télécharger de nouveau le fichier, on vérifie que la date sauvegardée est antérieure à la date de mise à jour sur le site web. Si c'est la cas, on télécharge le fichier, sinon, on ne fait rien. On peut ne pas utiliser ce mécanisme et retélécharger systématiquement le fichier grâce a la balise <*forceMode*> dans le fichier de configuration.

Une fois téléchargé, le fichier est stocké à l'emplacement indiqué dans le fichier de configuration (balise <*XMLTVFile*>).

On fait appel au parseur directement après le téléchargement, dans la même *TimerTask*.

2 Parseur

Le fichier XMLTV étant un fichier assez lourd (environ 80 Mo), il faut utiliser un parseur efficace. Par ailleurs, il n'est pas nécessaire de s'arrêter sur chaque balise, étant donné que les seules choses qui nous intéressent dans ce fichier sont les liens vers les images de programmes.

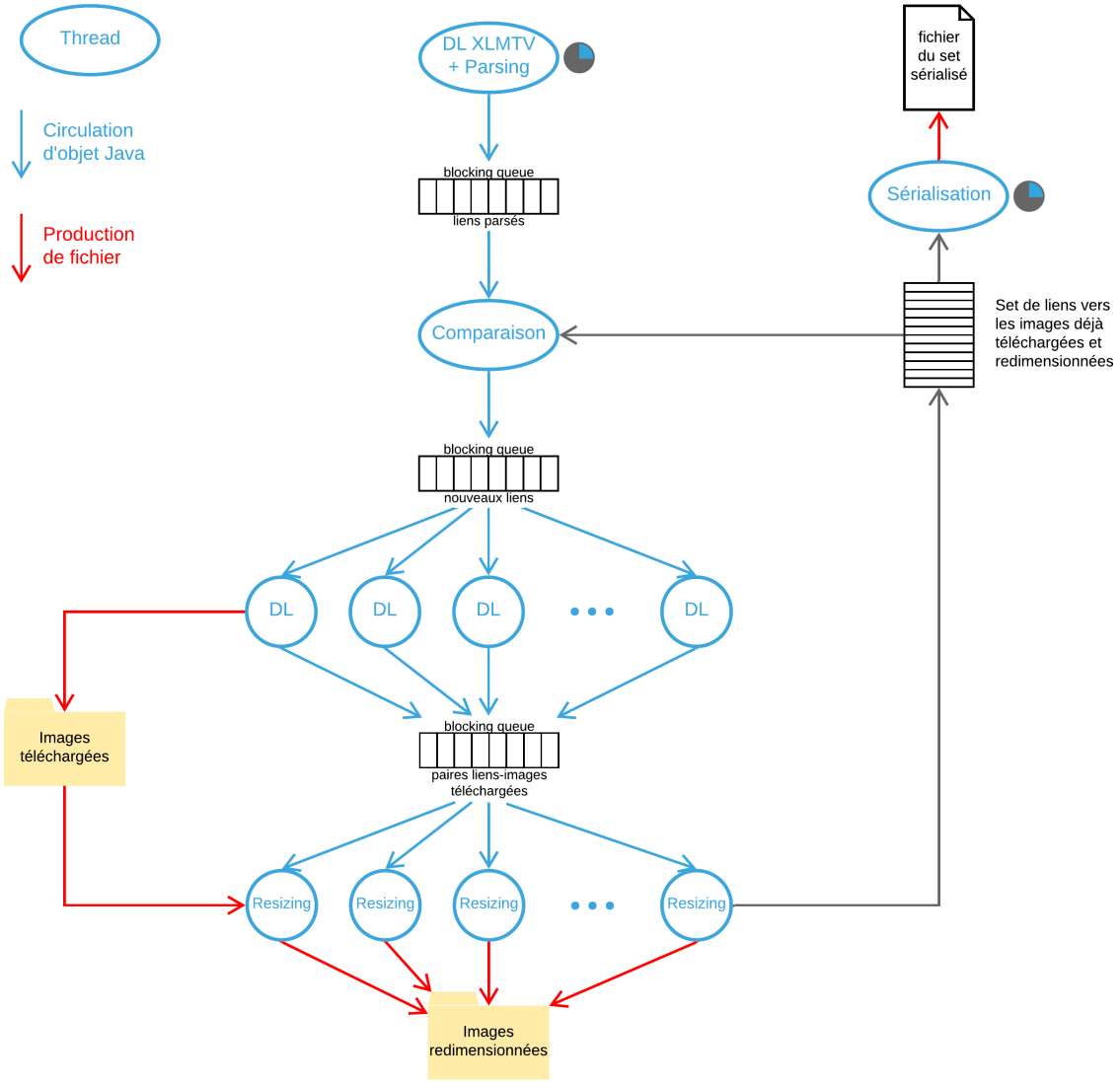


Figure 1 – *Structure globale de l'application*

Le parseur que nous utilisons est un parseur SAX. Il lit le fichier séquentiellement et appelle un événement à chaque fois qu'il rencontre une balise qui nous intéresse. Il a l'avantage de ne pas stocker l'intégralité du fichier dans la mémoire et d'être très rapide. De plus, le faible nombre de balises sur lesquelles nous lui demandons de s'arrêter rend l'utilisation du parseur optimale.

Les liens que l'on recherche se trouvent sous le paramètre *src* de la balise <*icon*>, elle-même située dans la balise <*programme*>. Il y a une petite difficulté car il ne faut pas prendre les liens des balises <*icon*> situées sous les balises <*rating*> (elles aussi sous <*programme*>) car celles-ci ne représentent pas des programmes.

Le parseur ajoute les liens dans un *HashSet*, ce qui nous permet de supprimer automatiquement les doublons, puis lorsque le *parsing* est terminé, on transfère les liens du *HashSet* dans la *BlockingQueue* « parsedLinks ».

3 Comparateur

Le module de comparaison est lancé dans un seul *thread* (l'opération est très légère) qui tire un par un les liens parsés par le module précédent dans la *BlockingQueue* et vérifie s'ils sont déjà contenus dans le *set* des liens d'images déjà traitées, c'est-à-dire téléchargées et redimensionnées. On verra comment est construit ce *set* dans la partie *Resizing* (Section 5). Si le lien n'est pas contenu dans le *set*, il est inséré dans la *BlockingQueue* suivante, à destination du module de téléchargement.

4 Téléchargement des images

Le module de téléchargement est quant à lui lancé dans un nombre de *threads* défini par le fichier de configuration (balise <*downloader.multithreading.threadsNumber*>), dont l'optimum peut varier selon la puissance de la machine et la vitesse de connexion, comme on le verra dans la partie sur les tests (Chapitre 4). Chaque *thread* est un consommateur de la *BlockingQueue* des nouveaux liens, qui ont été sélectionnés par le module de comparaison, et un producteur de la *BlockingQueue* suivante. En effet, il télécharge l'image dont il a tiré le lien dans le dossier des images téléchargées, et forme un objet *Pair* constitué du lien vers l'image téléchargée, et de l'objet *fichier* correspondant à l'image elle-même, dans son dossier. Cet objet peut enfin être inséré dans la *BlockingQueue* à destination du module de redimensionnement.

5 Resizing

Comme le module de téléchargement, le module de redimensionnement s'exécute dans plusieurs *threads*, dont le nombre est également défini dans le fichier de configuration (balise <*resizer.multithreading.threadsNumber*>) et dont l'optimum dépend aussi des performances du système. Chaque *thread* extrait une paire lien-image de la *BlockingQueue* alimentée par les *threads* de téléchargement et redimensionne l'image grâce à la fonction *resize* de la bibliothèque *org.imgscalr.Scalr*. En plus du fichier, cette fonction prend une taille unique en paramètre, qui est la longueur du côté du carré dans lequel l'image redimensionnée sera inscrite. Si l'image est plus large que longue, sa largeur sera égale à cette taille et sa longueur plus courte, et vice versa. La taille en question est définie dans le fichier de configuration (balise <*resizer.imageSize*>). Enfin, l'image originale est supprimée du dossier, afin de gagner de la place, l'image redimensionnée est enregistrée dans un dossier dédié et le lien qui correspond est ajouté au *set* des liens vers les images déjà téléchargées et redimensionnées.

6 Serialisation

Le dit *set* est périodiquement sérialisé, ce qui permet au programme de l'ouvrir et de le lire au démarrage, si le programme a été arrêté. Ainsi, lors de la prochaine mise à jour du dossier d'images, il a toujours la version du set qui correspond à l'état de son dossier. Comme pour le téléchargement du fichier XMLTV, la périodicité de ce *thread* est permise par la classe *TimerTask*, qu'il étend. La période est définie dans le fichier de configuration par la balise <*serializer.refreshTime*>.

7 Logger

Le *logger* est une simple classe contenant des méthode statiques. L'emplacement du fichier de logs est paramétrable dans le fichier de configuration (balise <*logger.logPath*>). Il suffit ensuite de faire appel à sa méthode *writeLogs(texte)* pour écrire directement le texte passé en paramètre à la suite du fichier de logs. La classe *Logger* se charge d'ajouter automatiquement la date et l'heure de l'appel à la méthode.

Dans notre programme, on fait appel au *logger* lorsque l'on attrape une erreur ou lorsque l'on télécharge un nouveau fichier XMLTV.

Il est compliqué d'enregistrer une trace de ce que font les autres modules étant donné qu'ils tournent continuellement (ce ne sont pas des processus avec un « début » et une « fin »). Pour une analyse plus poussée du comportement des autres modules, on peut utiliser les composants du module *Analytics*.

8 Analytics

Le module *Analytics* nous permet de suivre en temps réel ou a posteriori le déroulement de l'application.

Il s'agit d'un *thread* tournant parallèlement au programme et qui lit à une fréquence définie dans le fichier de configuration (balise <*analytics.refreshTime*>) la taille des différentes *BlockingQueues*, du *HashSet* de liens dont les images sont téléchargées et redimensionnée et la taille des dossiers des images téléchargées et redimensionnées.

Ce modules fournit ensuite plusieurs moyens pour accéder à ces informations.

8.1 Displayer

La classe *Displayer* permet d'afficher en temps réel les informations du module *Analytics*. Il s'agit d'une petite interface qui affiche les tailles des différentes *BlockingQueues*, du *HashSet* des liens et le nombre de fichiers contenus dans les dossiers des images téléchargées et des images redimensionnées.

Cet affichage est actualisé en temps réel, ce qui permet de voir le déroulement du programme (Figure 2).

8.2 CSVExporter

Ce deuxième outil du module *Analytics* permet de récupérer après coup un fichier CSV contenant les informations récupérée dans la classe *Analytics*. Ce fichier contient les tailles des *BlockingQueues*, du *HashSet* des liens et le nombre de fichiers contenus dans les dossiers des images téléchargées et des images redimensionnées pour tous les instants où le module *Analytics* a récupéré les informations.

Pour ne pas avoir un fichier trop gros et contenant des informations inutiles, on n'écrit pas dans le fichier lorsque les tailles des *BlockingQueues* des liens parsés, des liens comparés et des liens téléchargés sont toutes égales à 0 (cela correspond aux instant entre le redimensionnement de la dernière image téléchargée et le prochain téléchargement de fichier XMLTV, où le programme ne fait rien de spécial).

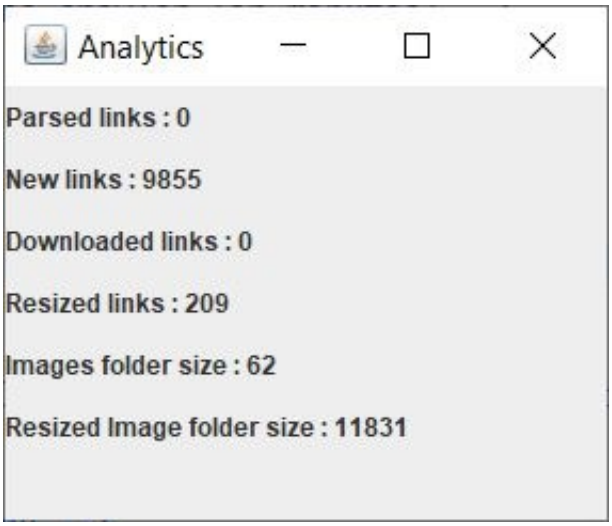


Figure 2 – *Interface permettant de visualiser en temps réel la taille des queues et des dossiers d'images*

L'avantage de récolter des informations à travers un fichier CSV est que cela nous permet de faire des graphiques sur les données. Ainsi, on peut vérifier si notre programme a fonctionné comme attendu et extraire des nouvelles informations sur le contenu des fichier XMLTV. Grâce à ces courbes, on peut par exemple estimer la proportion de nouveaux liens au cours du temps.

4 Tests

Afin de valider le fonctionnement du logiciel et de l'optimiser, nous avons procédé à des tests de mise en œuvre dans des conditions réelles, et à des tests de performances.

1 Exécution continue

Le but de notre projet étant de développer un programme qui tourne en continu pendant une longue période, il fallait le tester dans des conditions similaires pour s'assurer qu'elles n'entraîneraient pas des crashs ou des bugs. Heureusement, nous possédons une machine personnelle qui est utilisée comme serveur et qui nous a servi à tester notre programme en continu sur plusieurs semaines. Le serveur est une machine tournant 24/7 sous l'OS Freenas.

Grâce au module Analytics, nous avons pu récupérer des informations sur le déroulement de notre programme et vérifier qu'il fonctionnait bien pendant de longues durées (plusieurs semaines), sans être sujet à des bugs ni à des crashs.

Nous pouvons aussi utiliser les informations récoltées dans notre fichier CSV pour connaître l'heure approximative à laquelle le fichier CSV est mis à jour, ainsi que le nombre de nouvelles images ajoutées chaque jour.

Nous pouvons voir que les téléchargements ont été effectués aux alentours de 5h10. Comme nous avons paramétré notre application de manière à ce qu'elle ne s'active que toutes les heures, nous pouvons en déduire que le site XMLTV France met à jour son fichier quotidiennement, entre 4h10 et 5h10.

Grâce au graphique (Figure 1), nous pouvons voir qu'une grosse partie des images sont téléchargées au premier lancement de l'application, et qu'un nombre très réduit d'images est téléchargé chaque jour. Si il y a près de 10000 images qui sont téléchargées lors de la première exécution, il n'y a en moyenne que 400 nouveaux programmes ajoutés par jour sur les deux premières semaines. On peut aussi voir sur la Figure 2 que le nombre de nouveaux programmes diminue au cours du temps.

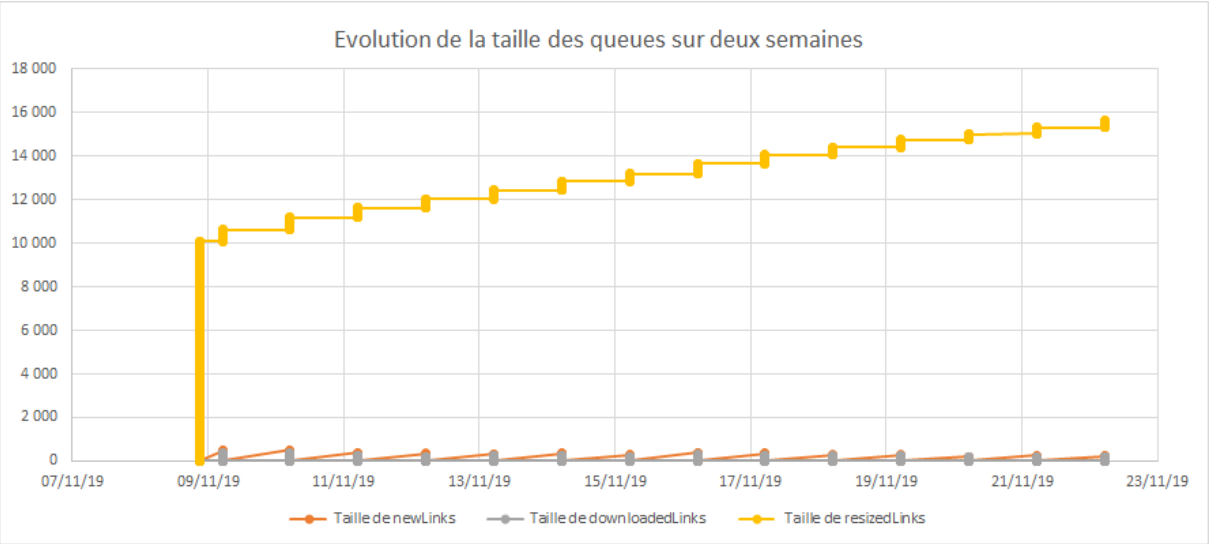


Figure 1 – *Évolution de la taille des queues sur deux semaines d'exécution continue*

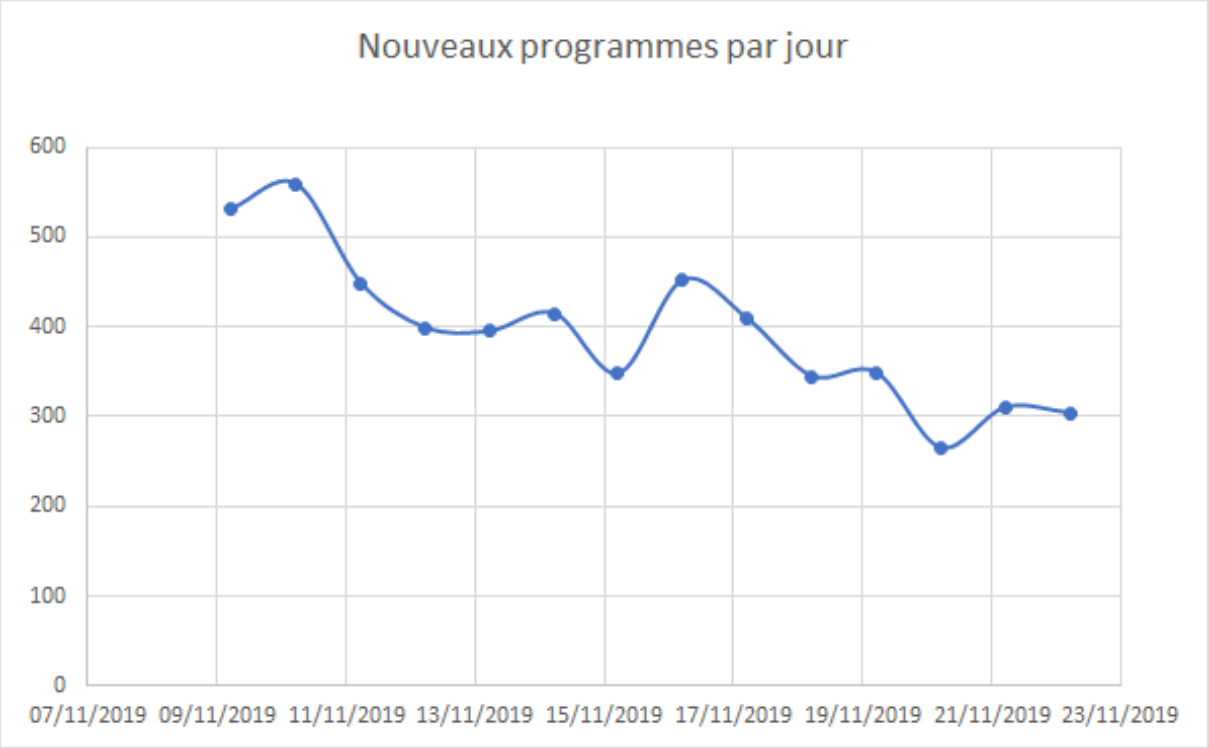


Figure 2 – *Nombre de nouveaux programmes au cours du temps*

2 Recherche de la configuration optimale

2.1 Présentation des tests

Nous avons aussi procédé à des tests de performance qui servent aussi à calibrer nos paramètres. L'objectif de ces tests était de voir l'impact du nombre de *threads* alloués aux différents modules sur le temps de téléchargement et de traitement de notre application.

Les deux modules dont il est possible de modifier le nombre de *threads* sont ceux s'occupant du téléchargement des images et de leur redimensionnement.

Le principe de ces tests est de faire varier le nombre de *threads* de téléchargement avec un nombre de *threads* de redimensionnement fixe, puis de faire l'inverse en faisant varier le nombre de *threads* de redimensionnement avec un nombre de *threads* de téléchargement fixe. En observant les temps d'exécutions ainsi que l'évolution en temps réel des tailles des queues, nous pouvons déterminer combien de *threads* sont nécessaires à ces deux modules, sur la machine de test.

2.2 Mise en oeuvre des tests

Nous avons tout d'abord mis en oeuvre ces tests sur un ordinateur portable connecté par Wi-Fi au réseau de l'école. Le problème de cette configuration est que nous avons été très vite limité par les performances de la machine et la vitesse de téléchargement du réseau. En effet, le téléchargement était trop lent pour que nous puissions voir un changement en faisant varier le nombre de *thread* de redimensionnement. Un seul *thread* de redimensionnement suffisait à gérer à lui seul le flux d'image téléchargé. De plus, cette configuration n'est pas forcément représentative de la configuration finale sur laquelle notre application sera déployée, qui se fera sur un serveur plus puissant et doté d'une connexion fibre.

Nous avons donc procédé à une deuxième série de test sur une autre machine avec une connexion fibrée.

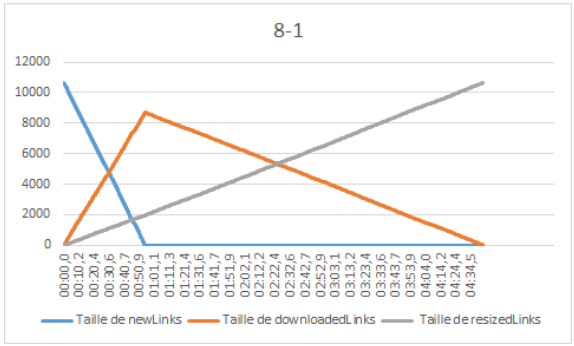
La machine sur laquelle nous avons fait nos tests possédait 6 coeurs et le débit descendant était de 300 Mb/s.

Nous avons choisis une nomenclature de la forme x-y où x est le nombre de *threads* du module téléchargement et y le nombre de *threads* du module redimensionnement. Ainsi le graphique 4-8 indique les tailles des queues lorsqu'il y a 4 *threads* de téléchargement et 8 *threads* de redimensionnement.

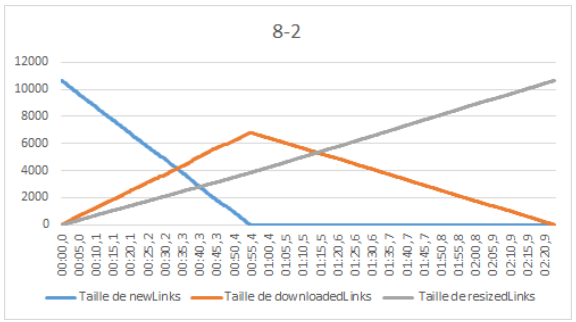
Nous pouvons voir sur les graphiques que lorsque le nombre de *threads* de redimensionnement est trop faible (Figure 3a, Figure 3b et Figure 3c), le programme va continuer de redimensionner les images alors qu'elles seront déjà toutes téléchargées, ce qui en plus de prendre plus de temps, occupe plus de mémoire. A partir de 8 *threads* (Figure 3d et Figure 3e), le redimensionnement rattrape le téléchargement des images, ce qui fait qu'il y a très peu d'images qui restent en attente dans la queue des downloadedLinks.

En recommençant le test en faisant varier les *threads* de téléchargement et en gardant 8 *threads* de redimensionnement, nous pouvons voir que le nombre de *threads* de téléchargement a un impact important sur la vitesse de téléchargement. Ainsi, avec un seul *thread*, le téléchargement de toutes les images prend plus de 3 minutes alors qu'il ne prend qu'une minute et 15 secondes avec 8 *threads*.

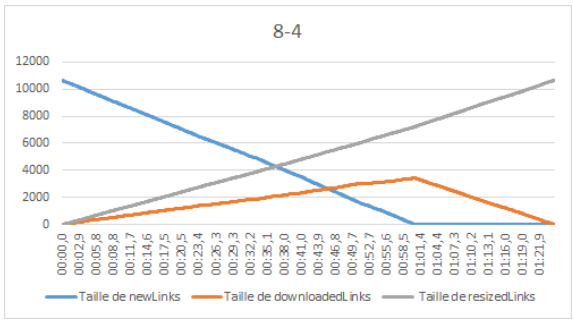
Nous pouvons donc voir qu'avoir plus de *threads* nous permet d'améliorer significativement la vitesse de notre programme. En revanche, prendre trop de *threads* peut aussi nuire aux performances du programme : dans notre cas, les temps avec 16 *threads* sont un peu plus longs qu'avec 8.



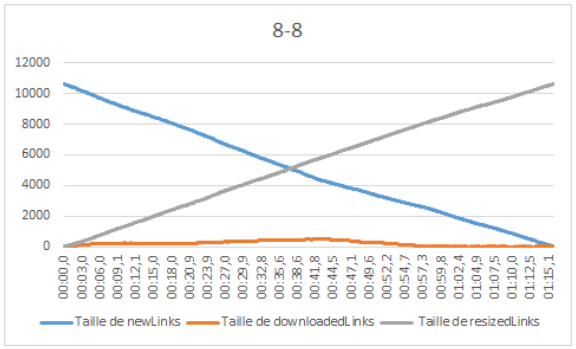
(a) *8-1*



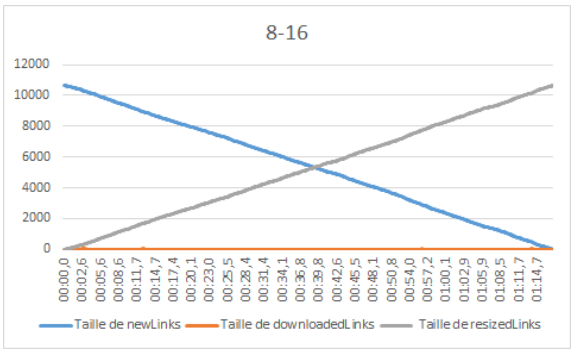
(b) *8-2*



(c) *8-4*

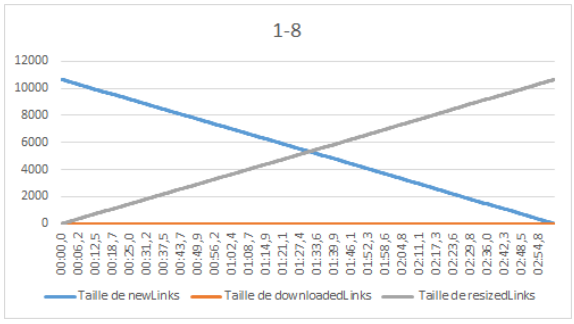


(d) *8-8*

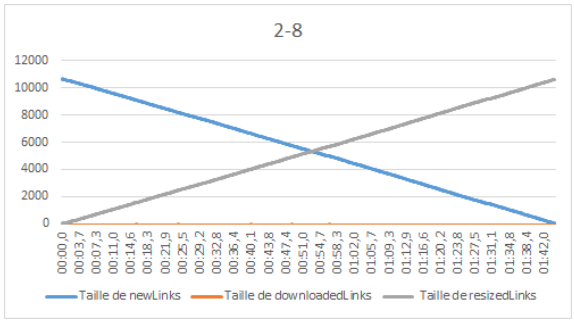


(e) *8-16*

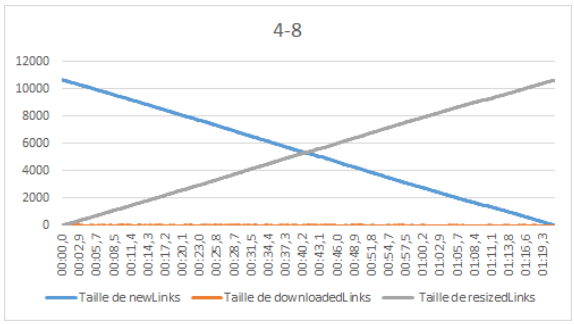
Figure 3 – *Variations du nombre de threads de redimensionnement avec 8 threads de téléchargement*



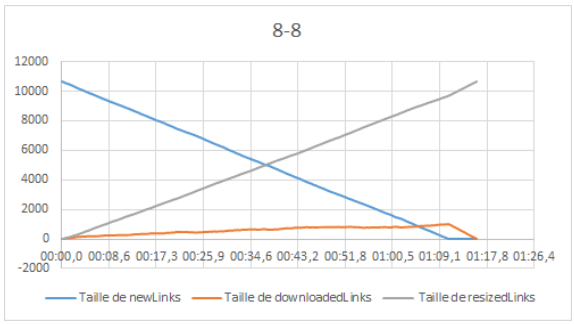
(a) *1-8*



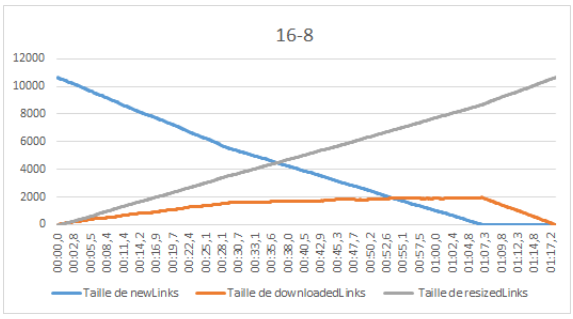
(b) *2-8*



(c) *4-8*



(d) *8-8*



(e) *16-8*

Figure 4 – *Variations du nombre de threads de téléchargement avec 8 threads de redimensionnement*

5 Conclusion

Nous avons réussi à mener à bien notre projet jusqu'au bout, ce qui nous permet de fournir un programme fonctionnel et correspondant aux spécifications demandées.

Nous avons mis en place une architecture modulaire, dont chaque partie est indépendante. Ainsi, il est possible de modifier ou de remplacer chacun de nos modules sans que cela n'ait un impact sur les autres. Cette architecture a été possible grâce à l'utilisation des *blocking queues* qui nous permettent de transférer nos données d'un module à l'autre facilement et de résoudre simplement le problème de producteurs/consommateurs multiples.

Enfin, nous avons pu tester notre programme sur un serveur dans des conditions réalistes, ce qui nous a permis de vérifier qu'il fonctionnait correctement et de récupérer des données nous permettant de faire des statistiques quant à l'intensité du travail effectué chaque jour.

Annexes

A Fichier de configuration

```xml
<?xml version="1.0" encoding="UTF-8"?>

<config>
  <website>
    <URL>
      <!--The home page of xmltv.fr website-->
      <home>https://www.xmltv.fr</home>

      <!--The link to the XMLTV file-->
      <file>https://www.xmltv.fr/guide/tvguide.xml</file>
    </URL>
  </website>
  <crawler>
    <!--The time between two activations of the crawler (in minutes) -->
    <refreshTime>10</refreshTime>

    <start>
      <!-- The hour when the program will start -->
      <hours>17</hours>
      <minutes>41</minutes>
      <seconds>00</seconds>
    </start>

    <!--Forces the crawler to download the file even if the file is up to date -->
    <forceRecovery>true</forceRecovery>

    <!--The path where the XMLTV file is stored -->
    <XMLTVFile>
      <path>C:/Temp/XMLTV/xmltv.xml</path>
    </XMLTVFile>

  </crawler>
  <downloader>
    <!-- The path of the folder where the images will be temporary stored  -->
    <imagesFolderPath>C:/Temp/Images</imagesFolderPath>
    <multithreading>
    <!-- The number of threads that download the images (8 recommended)-->
      <threadsNumber>8</threadsNumber>
    </multithreading>
  </downloader>
```

```
41    <resizer>
42      <!-- The path of the folder where we store our final images -->
43      <resizedImagesFolderPath>C:/Temp/ImagesResized</resizedImagesFolderPath>
44      <!-- The maximum size of images -->
45      <imageSize>224</imageSize>
46      <!-- The number of threads that resize the images (8 recommended) -->
47      <multithreading>
48        <threadsNumber>8</threadsNumber>
49      </multithreading>
50    </resizer>
51
52    <serializer>
53      <!--The interval of time between two saves in minutes-->
54      <refreshTime>15</refreshTime>
55      <!-- The name of the file where we store our downloaded links -->
56      <serializedLinksPath>links.dat</serializedLinksPath>
57    </serializer>
58
59    <logger>
60      <!-- The path of the log file -->
61      <logPath>C:/Temp/Logs/Logs.txt</logPath>
62    </logger>
63
64    <analytics>
65      <!-- The path of the csv file -->
66      <CSVpath>C:/Temp/CSV/csv.csv</CSVpath>
67      <!-- Activate or desactivate the interface -->
68      <GUI>true</GUI>
69      <!-- refresh time  of the Analyzer module -->
70      <refreshTime>100</refreshTime><!-- In ms -->
71    </analytics>
72
73  </config>
```

Crawler XMLTV

Résumé

Un robot d'indexation qui récupère les images d'illustration des émissions télévisées sur le site de Télérama grâce à une architecture modulaire utilisant des queues bloquantes pour répondre au problème de producteurs/consommateurs multiples.

Mots-clés

producteurs/consommateurs, robot d'indexation, queues bloquantes, XMLTV

Abstract

A web crawler getting TV show thumbnails from *Télérama*'s website, using a modular achitecture with blocking queues to solve the multiple producer–consumer problem.

Keywords

producer–consumer, web crawler, blocking queues, XMLTV

Tuteur académique
Mathieu Delalandre

Étudiants
Lucien Le Guellec (DI5)
Romain Hérault (DI5)