Ecole Polytechnique de l'Université François Rabelais 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

2016-2017

Sujet ASR #6 : Stratégie d'indexation pour un robot Web parallèle

Tuteur académique
Mathieu Delalandre

Étudiants
Paul Fayoux (DI5)
Valérian Menin (DI5)

20 janvier 2017

Liste des intervenants

Nom	Email	Qualité
Paul Fayoux	paul.fayoux@etu.univ-tours.fr	Étudiant DI5
Valérian Menin	valerian.menin@etu.univ-tours.fr	Étudiant DI5
Mathieu Delalandre	mathieu.delalandre@univ-tours.fr	Tuteur académique, Département Informatique

Avertissement

Ce document a été rédigé par Paul Fayoux et Valérian Menin susnommés les auteurs.

L'Ecole Polytechnique de l'Université François Rabelais 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

Paul Fayoux et Valérian Menin, *Sujet ASR #6 : Stratégie d'indexation pour un robot Web parallèle*, Projet Architecture, Système et Réseaux, Ecole Polytechnique de l'Université François Rabelais de Tours, Tours, France, 2016-2017.

```
@mastersthesis{
    author={Fayoux, Paul and Menin, Valérian},
    title={Sujet ASR #6 : Stratégie d'indexation pour un robot Web parallèle},
    type={Projet Architecture, Système et Réseaux},
    school={Ecole Polytechnique de l'Université François Rabelais de Tours},
    address={Tours, France},
    year={2016-2017}
}
```

Document mis en forme sous LaTeX
grâce à la classe de document `polytech.cls` V2.1.10 (2016/12/08)

Table des matières

Table des figures

Introduction

Les robots d'indexation sont des composants centraux pour le Web. Ils permettent la récupération de contenu hétérogène sur le Web et le moissonnage de portails d'information. Néanmoins la récupération de données à des fins d'exploitation n'est sans soulever des problèmes de copyright. D'un côté des données publiquement postées sur le Web à des fins utilisateurs / commerciales sont libres de droit. De l'autre côté, de nombreux serveurs mettent en place des stratégies de détection afin de limiter le moissonnage automatique de leur portail d'information par les robots d'indexation.

L'alternative consiste alors à mettre en place des stratégies d'indexation. Par exemple, en répartissant les requêtes sur un intervalle de temps donné, en mixant de fausses et de vraies requêtes, en distribuant le robot sur différentes machines afin de diversifier les accès en IP, en doublant les requêtes, etc. Ceci ne va pas sans soulever des problèmes de synchronisation au sein du robot, au niveau centralisé et réparti. Ce sujet se propose d'explorer certains aspects de cette problématique, en particulier avec la mise en œuvre d'un ordonnanceur central pour la répartition des requêtes et la mise en place d'une architecture client-serveur pour leur détournement. Le sujet visera un cas d'usage d'indexation de portails images e.g. `http://www.bedetheque.com/`

1 Acteurs et contexte

Dans le cadre de notre spécialité Architecture Système et Réseau, nous (messieurs Paul Fayoux et Valérian Menin) avons eu l'opportunité de faire un projet. Le projet de stratégie d'indexation pour un robot Web parallèle proposé par monsieur Mathieu Delalandre nous nous a intéressé et nous avons donc choisi de le traiter.

Nous avons déjà eu l'opportunité de travailler sur un projet de robot d'indexation en python avec la librairie *pysoup*. Pour ce projet nous avions pour but de récupérer toutes les personnes présentes sur Wikipédia puis de télécharger de façon automatique les k premières images correspondantes à une recherche d'image sur différents moteurs de recherche (Baidu, Bing et Yahoo, car nous étions en Chine et ne pouvions avoir accès à Google qu'avec un VPN).// Nous avions donc commencé à voir des blocages liés à certains sites et voulions découvrir plus en détail ces moyens.

Par ailleurs, nous avons des serveurs personnels et nous souhaitions savoir comment mieux les administrer pour éviter certaines attaques. Nous étions donc intéressés pour comprendre

comment sont effectuées ces attaques pour ensuite pouvoir les éviter.

2 Objectifs

L'objectif de notre projet était d'étudier les différentes stratégies qui peuvent être mis en place afin de pouvoir indexé des données rapidement. Notre projet ne se base donc pas sur la mise en place d'un robot web, mais sur le support qui pourrait lui permettre d'avoir de meilleur performance sans se faire bloquer par les serveurs.

Pour cela nous avons étudié les stratégies proposé par le sujet et nous avons cherché à effectuer des requêtes via une application que nous avons développer. En changeant le comportement de l'application nous avons pu observé des différences de performances et nous avons aussi pu remarquer sous quel condition nous pouvions nous faire jeter des serveurs.

A force de recherche nous avons trouvé intéressant de voir quel mécanisme il était possible de mettre en place sur un serveur afin de bloquer les différentes attaques. Nous possédions des serveurs sur lesquels nous avons pu testé certain de ces mécanismes de défense.

3 Base méthodologique

Nous travaillons avec GitHub pour gérer les versions de nos sources.

Lors de la période de projet, nous avons pu à plusieurs reprises faire le point sur nos avancées, les difficultés rencontrées ainsi que les nouvelles directions envisagées.

Notre encadrant de projet nous a conseillé de développer en java car c'est un langage populaire et surtout la visualisation des performances est assez simple grâce à de nombreux outils disponibles.

1 Description générale

1 Environnement du projet

Durant notre projet, nous avons pu rencontrer notre encadrant à plusieurs reprises pour discuter des différentes orientations possibles pour continuer.

Ce projet devait nous permettre de tester les différentes stratégies d'attaque. Nous avons donc d'abord cherché à mettre en place une application qui nous permettrait de faire des requêtes à volonté, puis nous avons cherché a étudié les différents comportements en modifiant des paramètres. Nous avions une certain liberté concernant la mise en place, et l'étude des stratégies.

2 Caractéristiques des utilisateurs

Notre projet étant d'étudier les différentes stratégies possibles pour faire de l'indexation de page web. Les utilisateurs pourront être des personnes voulant tester les stratégies pour ensuite les mettre en place sur leur robot, ou bien des personnes voulant tester les mécanismes de défense de leur serveur.

3 Fonctionnalités de l'application

L'application que nous avons développée est réalisé en Java, toute machine possédant un environnement d'exécution Java peut faire tourner le programme.

Notre programme n'établit pas de stratégie en particulier, il propose à l'utilisateur une interface graphique lui permettant de lancer une attaque. L'utilisateur a le choix de renseigner certain paramètre afin de configurer un type d'attaque.

Les paramètres renseignables par l'utilisateur sont les suivants :

— l'url de la page web à attaquer
— le nombre de threads à utiliser
— le nombre de requêtes à effectuer
— le port de connexion de la machine
— le IP de la machine qui prendra le relais

— le port de connexion de la machine qui prendra le relais

Le nombre de threads, et le nombre de requête permette déjà à l'utilisateur de choisir entre une attaque maximisant le nombre de connexion simultané (ce qui se rapproche du Déni de Service Deny of Service (DoS)), ou une attaque favorisant les performances de téléchargement en minimisant le nombre de connexion simultané.

Les trois derniers paramètres (les deux ports et l'IP) permettent de mettre en place un relais entre les machines. Une première machine peut lancer l'attaque, puis lorsqu'elle échoue, elle peut passer le relais à la machine suivante et ainsi de suite jusqu'à ce que l'on revienne sur la première.

4 Structure générale du système

On distinguera les deux possibilités de topologie qui pourront être mis en place :

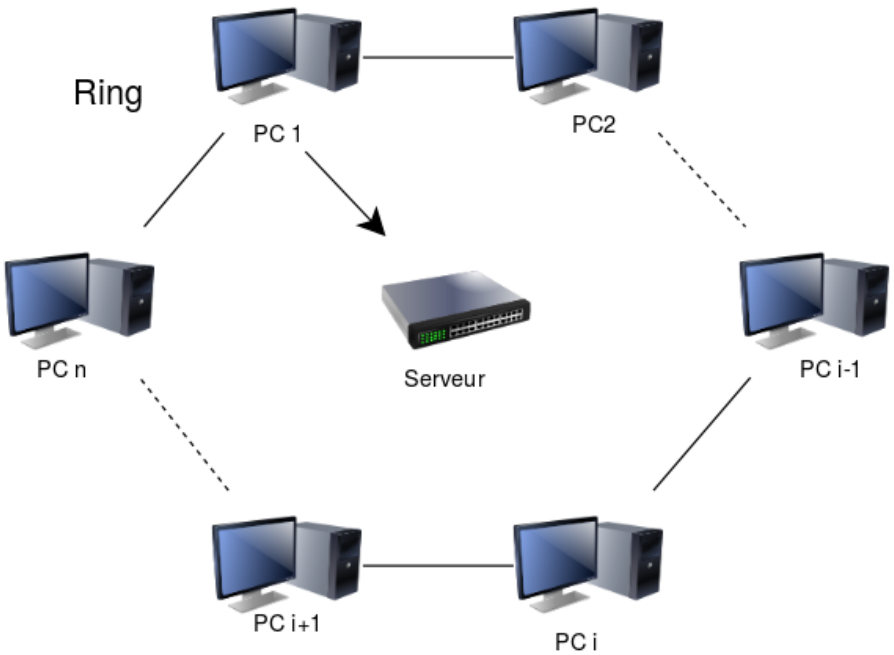

Figure 1 – *Topologie en anneau*

La topologie en anneau est utilisée dans le but de faire du web scraping (aussi appelé harvesting). Le programme est donc lancée sur toutes les machines mais s'exécute de façon à ce qu'une seule des machine ne travail. On cherche à avoir une information de bonne qualité pour par exemple récupérer les données d'un site web.

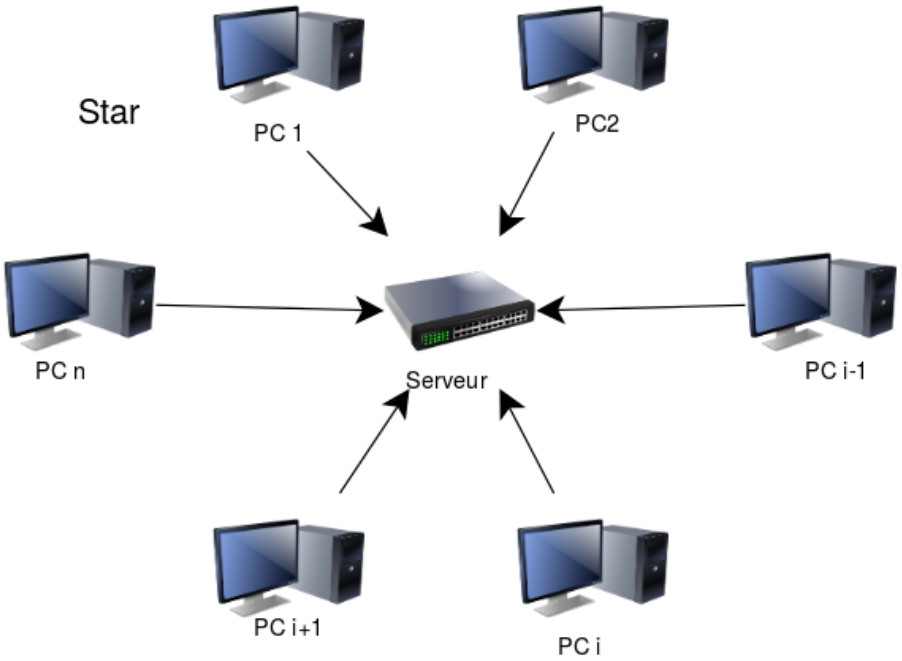

Figure 2 – *Topologie en étoile*

La topologie en étoile peut-être utilisée dans le but de surcharger le serveur pour des attaques de type Dénis de Service, mais peut aussi être utilisée dans le but de récupérer des données plus rapidement.

Dans le cas d'une attaque dénis de service chacune des machines vont chercher à créer le plus de connexion simultanée possible, mais si chaque machine ne crée des connexions que les unes après les autres cela permet d'avoir un résultat rapide sans être bloqué par le serveur (on sera perçu comme étant plusieurs utilisateurs courtois).

Pour cela une des stratégies consiste à mettre en place un serveur ordonnanceur qui dispatch le travail du robot sur plusieurs machines, qui se chargeront de faire les requêtes, chaque résultat de requête doit ensuite être renvoyé à l'ordonnanceur qui se chargera des traitements.

2 Analyse des différentes stratégies

Nous avons étudié deux types de stratégie en faisant tout d'abord des requêtes séquentielles (web scrap), puis des requêtes simultanés en utilisant du multi-threading. Nous avons testé ces mécanismes sur une seule machine avant de faire la même chose avec plusieurs machines.

1 Exécution séquentiel de requêtes (mono-thread)

Dans un premier temps, nous avons testé de faire plusieurs requêtes de façon séquentiel. Nous avons remarqué qu'au bout d'un grand nombre de requêtes nous nous faisions jeter par le serveur.

Dans notre cas, nous avions commencé par effectuer des recherches sur le site http://www.bedetheque.com/. Notre objectif était de simuler du scraping.

1.1 Le web scraping

Le web scraping consiste à extraire du contenu de sites web. Cette attaque est assez classique et très répandue en particulier par les robots d'indexation ou ceux effectuant du pillage de données.

Le scraping est considéré comme une visite dérangeante et intrusive, car le principe est de prendre des informations du site pour les utiliser sur un autre site. Nous pouvons citer Google news comme étant un outil journalistique effectuant du web scraping pour faire ses articles alors que certains sites de média ne les autorisent pas à le faire.

Notre premier objectif était de pouvoir récupérer des données depuis le site http://www.bedetheque.com/.

Comme le montre les Figure 1et Figure 2, le site est une banque de données d'image importante. Nous voulions donc savoir s'il était possible de télécharger automatiquement toutes ces images.

Les sites internet contrôlent normalement l'utilisation du scraping via un fichier robots.txt qui décrit les répertoires des pages indexables et les User-Agents des robots autorisés (par exemple : on peut choisir d'autoriser uniquement les robots de Google). Cependant, ce fichier n'est qu'une règle rarement respectée et peu de pays ont créé des lois donnant une véritable valeur à ce fichier. Cette règle est souvent ignorée par les robots qui souhaitent récupérer un maximum de

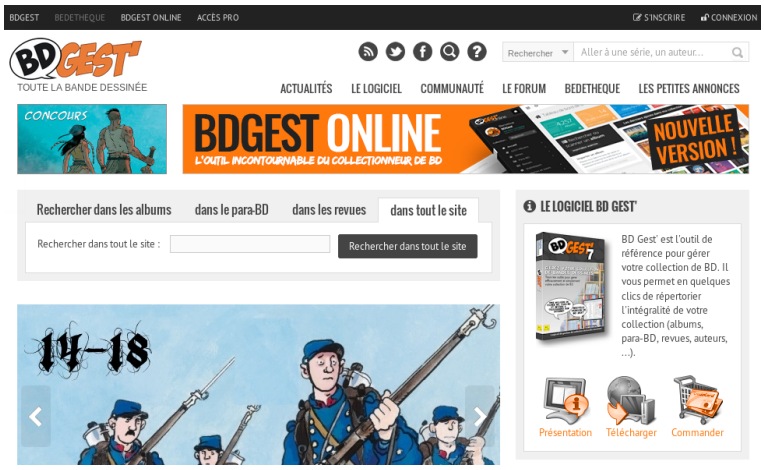

Figure 1 – *capture d'écran de l'accueil du site*

Figure 2 – *capture d'écran de résultat de recherche*

données pour l'indexation.
Les informations pour les robots sont situées à la racine du site comme nous pouvons le voir sur le site de la bedetheque : `http://www.bedetheque.com/robots.txt`.

En général, un crawler web va parcourir un site pour en faire une topologie alors que le scraper cherche à recueillir des données. Dans notre cas, nous cherchions à récupérer des images mais nous pouvons imaginer un scraper allant sur un site pour analyser les prix d'une entreprise puis faire une analyse sur ces prix, ainsi, une entreprise concurrente pourrait aligner ses prix juste en dessous. Ceci est un fait, de nombreuses entreprises et organisations utilisent ce genre

d'outil pour obtenir des informations.

Un scraper cherche à obtenir des informations ciblées. Dans notre cas, nous avions choisi de cibler les images représentants des couvertures de bandes dessinées afin d'avoir des requêtes qui demande au serveur d'envoyé beaucoup de données. On espérait alors se faire jeter plus rapidement, car il était possible qu'un serveur s'aperçoit que nous lui faisions trop souvent des requêtes.

1.2 Résultat de nos tests

Nous voulions dans un premier temps avoir des temps de requêtes rapide. Le but n'était donc pas de surcharger le travail du serveur mais plutôt d'optimiser nos demandes pour obtenir les informations désirées.

Nous avons donc fait un flux de requêtes sur le site pour effectuer des recherches. Puisqu'il ne nous était pas demandé de faire un Crawler, mais de simplement tester la stratégie d'attaque, nous avons simplement lancé plusieurs fois la même recherche en boucle afin de voir le comportement.

Nous avons ainsi remarqué qu'au cours de l'attaque certaines des requêtes étaient rejetées, cependant rien ne bloquait réellement le programme.

1.3 Mécanisme de défense

Nous savons qu'il est possible qu'un serveur mette en place un système regardant le nombre de requête qu'un utilisateur peu faire par seconde sur son serveur, cependant le fait que les requêtes soit faites séquentielle-ment ne perturbe pas suffisamment le serveur pour que ce type d'attaque soit dangereux pour lui. Ainsi nous n'avons rencontré aucun serveur mettant en place ce type de protection et ceci est logique puisque cela pourrait être nuisible aux utilisateurs.

1.4 Utilisation de plusieurs machines

Nous avons ensuite réalisé les mêmes tests en attaquant simultanément un même serveur en lançant plusieurs machines. Puisque chacune des machines sont vu comme de simple utilisateurs aucune d'entre elles n'est rejetées. On peut donc déjà valider la stratégie consistant à mettre en place un ordonnanceur qui partagera des requêtes entre chacune des machines.

2 Attaque en maximisant le nombre de connexion (multi-threading)

Nous avons ensuite cherché à mettre en place du multi-threading, afin de pouvoir lancé au sein d'une même machine plusieurs requêtes et donc plusieurs connexions au serveur web. Ce type d'attaque se rapproche d'attaque de déni de service Deny of Service (DoS) très souvent bloqué par les sites web.

2.1 Le déni de service

Le déni de service consiste a attaqué un serveur en le submergeant de connexion. Quand le serveur reçoit une requête il établit une connexion avec le client puis ils vont répondre au client en envoyant plusieurs paquets. Si plusieurs clients lui envoient beaucoup de requêtes

beaucoup de connexion vont s'établir et le serveur pourra être en difficulté de répondre à toutes les requêtes ce qui provoque alors le blocage des requêtes des autres utilisateurs.

Le déni de service est une attaque punie par la loi en France, selon l'article *323* du code pénal : "*Le fait d'entraver ou de fausser le fonctionnement d'un système de traitement automatisé de données est puni de cinq ans d'emprisonnement et de 75 000 euros d'amende*".

2.2 Résultat de nos tests

Nous avons donc testé, avec tout d'abord une seule machine uniquement, d'attaquer le site web `http://www.bedetheque.com/` pour voir comment il réagirait. Il se trouve que leur site web a mis en place un système de protection contre ces attaques, ainsi en testant avec 4 threads en parallèles nous étions déjà blacklistés et nous étions redirigés vers une page web contenant ce texte : "*Vous utilisez sans doute un programme qui scanne la bedetheque. Votre IP a été bloquée pour préserver les ressources du serveur, car ce genre de script pénalise l'ensemble des utilisateurs du site. Vous pouvez utiliser gratuitement BDGest Online pour gérer votre collection de BD, comics et mangas. Pour plus d'information, contactez info@bdgest.com.*"

Nous avons donc par la suite fait d'autre tentative sur nos serveurs et site web personnels. Les résultats statistiques sur les temps d'exécution d'une requête et sur le débit, montre que plus ils y a de threads plus les débits sont faibles et plus les requêtes vont mettre du temps. Ceci s'explique par le fait que chaque requête vont se partager la bande passante, celle-ci atteint rapidement son maximum lors du lancement d'une attaque.

Nous avons ainsi pu remarquer que certain hébergeur web ne proposait pas de protection contre ce type d'attaque, et concernant nos serveurs nous avons pu mettre en place des mécanismes de protection.

2.3 Mécanisme de défense

Les serveurs web Apache et Nginx propose des configurations permettant d'être protégé contre les attaques de type Deny of Service (DoS).

Dans Nginx les paramètres suivant permettre d'être protégé en limitant le nombre de requête et le nombre de connexion qu'un utilisateur peut faire par seconde.

- *limit_req* : permet de limiter le nombre de requetes maximum par IP et par seconde
- *limit_conn* : permet de limiter le nombre de connexions maximum par IP

Dans Apache le mode **mod-evasive** permet de renseigner des paramètres similaires :

- DOS*PageCount* : définit le nombre de fois ou une page peut être appelée par la même adresse IP avant que celle-ci ne soit bloquée.
- DOS*SiteCount* : définit le nombre de fois ou un site peut être demandé par la même adresse IP avant que celle-ci ne soit bloquée.
- DOS*PageInterval* : détermine un intervalle en secondes qui autorise l'affichage de la même page avant un blocage.
- DOS*SiteInterval* : détermine un intervalle en secondes qui autorise l'affichage d'un même site avant un blocage.
- DOS*BlockingPeriod* : détermine la durée de blocage (en secondes).

Si un administrateur met en place ces règles, il peut ensuite mettre en place l'outil **Fail2ban** afin de bloquer pendant autant de temps qu'il le souhaite les IP qui ont tenté de faire des attaques.

3 Conception de l'application

1 L'interface de notre application

Voici l'interface de notre application après avoir lancé une attaque sur le site web www.padovapizza.com.

Le champ url permet à l'utilisateur de renseigner l'adresse url du site sur lequel il veut faire des requêtes. On trouve les champs "Nb Thread" et "Nb Requete" permettant de fixer le nombre de thread à créer et le nombre de requête à faire, ici il y a un seul thread et on veut faire 1000 requêtes.

On a renseigné une IP qui correspond à l'adresse local de notre machine et le port 4024 dans les champs "IP du client" et "Port du client", nous avons effectivement lancé l'application sur un seul machine. Dans ce cas l'application transférera les données de l'attaque au port 4024 de la machine.

Nous avons aussi renseigné le champ "Port du serveur" qui correspond au port sur lequel notre application va écouter afin de recevoir une demande de relais d'une autre application, nous avons démarré l'écoute via le bouton "Start server" c'est pour cela que le message "serveur démarré correctement" s'affiche.

Nous avons lancé l'attaque en appuyant sur le bouton "Start", l'ensemble des statistiques ou des messages d'erreur qui auront lieu pendant l'attaque s'affiche dans le panel de gauche. On y retrouve en première entrée les paramètres que l'utilisateur a renseigné, ensuite les nombres de requête effectué, le débit moyen et le temps moyen des requêtes à chaque instant ou l'affichage s'est rafraîchi.

Tout en bas de la fenêtre une barre de progression permet de savoir combien de requête il reste à faire.

2 Architecture de notre application

Notre application a été réalisé de la façon suivante. Nous avons :

— Une classe **Windows** chargé de l'interface. Cette classe permet à l'utilisateur de lancer le thread **SocketConnexionServer** avec l'action "Start Server" et de lancer ou arrêter le thread **Thread_print** avec les actions "Start" et "Stop".

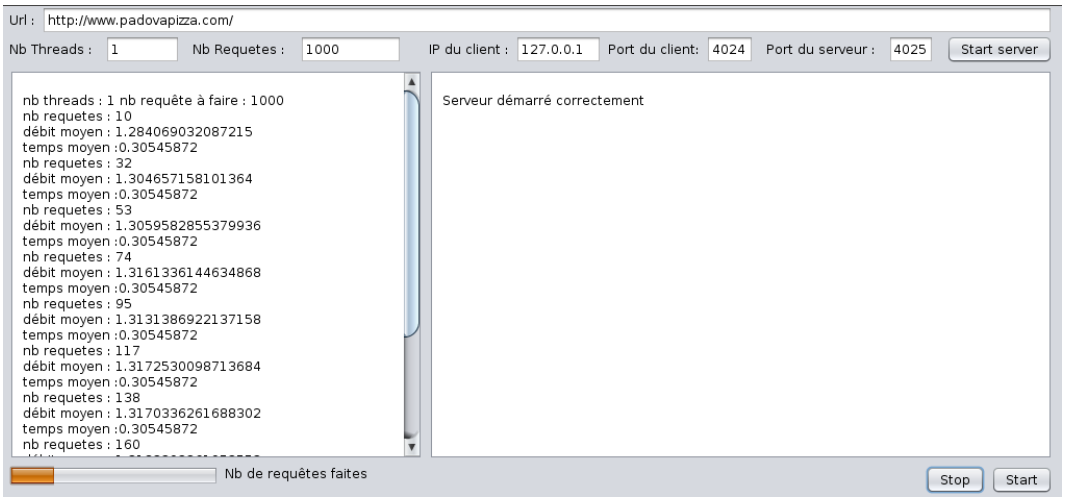

Figure 1 – *Interface de l'application*

— Une classe **Thread_print** qui implémente l'interface **Runnable** chargé de lancer les autres threads **Thread_requestUrl** chargé d'envoyer les requêtes via des objets **WebRobot**.
— La classe **WebRobot** qui défini un objet permettant de stocker les informations concernant une requête et possédant des méthodes permettant d'envoyer la requête. Cet objet possède les méthodes permettant de stocker les pages web reçu par les requêtes, cette fonctionnalité n'est pas utilisée dans notre programme (car nous n'avons pas besoin de stocker les pages web).
— La classe **SocketConnexion** qui contient les méthodes permettant de lancer des threads **SocketConnexionClient** et des threads **SocketConnexionServer**.
— La classe **SocketConnexionClient** qui permet de lancer une connexion en tant que client à un serveur à une adresse IP et un port donné. Cette classe nous sert a envoyé les informations de l'état d'une attaque à une autre machine.
— La classe **SocketConnexionServer** qui permet de lancer un thread qui écoutera sur un port TCP donné. Cette classe nous permet de récupérer le message envoyé par une autre machine qui vient d'être bloqué pendant son attaque. Après avoir reçu le message qui contient l'état de l'attaque, le thread **SocketConnexionServer** lance le thread **Thread_print** afin de reprendre l'attaque avec les paramètres du message.
— La classe **RobotWebException** nous permet de gérer des erreurs particulières ayant lieu dans les méthodes de la classe **WebRobot**.

Le thread qui s'occupe de l'ordonnancement des requêtes est donc le thread **Thread_print**, puisqu'elle lancera les autres threads chargées d'effectuer ces requêtes. Le nombre de requête que l'utilisateur renseigne correspond au nombre de requêtes totales qui seront effectuées. Afin de gérer les threads qui effectue les requêtes, nous avons utilisé un **Station-service** ainsi le nombre de requête à faire est partagé entre chaque thread de façon a ce que chaque thread aient à peu près la même charge de travail. Dans le cas où le nombre de requête à faire est inférieur au nombre de threads alors autant de threads que de nombre de requête seront créées et chacune ne feront qu'une seule requête.

Un certain nombre d'objets sont partagés entre le thread principal **Windows** et le thread **Thread_print** et aussi entre la classe **Thread_print** et les autres classes. Nous avons fait attention à géré correctement la concurrence sur ces objets en utilisant des variables **Volatile** et des variables de type **Atomic**.

Notamment afin d'afficher à l'utilisateur des indicateurs concernant le temps moyen et le débit moyen mis par chaque requête, nous avons mis en place deux tableaux **AtomicLongArray** partagés entre toutes les threads **Thread_requestUrl** et le thread **Thread_print**. Un des tableaux

est utilisé afin de connaître le temps moyen des requêtes d'un thread, l'autre tableau est utilisé afin de connaître le débit moyen. Ainsi chaque thread **Thread_requestUrl** va modifier sa case du tableau seul le thread **Thread_print** lis les données de façon périodique (nous avons fixé une période de 3 secondes) et affiche les statistiques à l'utilisateur. Un compteur du nombre de requête terminé, sous forme d'**AtomicInteger** est aussi partagé afin de connaître l'état d'avancement de l'attaque.

4 Conclusion

Au départ de ce projet nous n'avions pas connaissance des outils java permettant de lancer des requêtes sur des serveurs web. Ce projet a été une opportunité pour nous d'en apprendre plus sur ces mécanismes et de voir les comportements lorsque l'on introduit du multi-threading.

Il est important de savoir que les mécanismes que nous avons étudiés ne sont pas a utilisé n'importe comment ni n'importe ou. Le déni de service est un des cas d'attaque bien cerné par la loi Française (et dans la plupart des pays). Il reste vrai que beaucoup de pratique non courtoise n'ont pas encore été légiféré notamment sur l'utilisation du "robot.txt" par les robots d'indexations.

Nous avons pu observer que le multi-threading dégradait les performances en termes de téléchargement, mais maximisait l'utilisation de la bande passante. La solution qui semble la plus intéressante pour mettre en place un robot d'indexation reste la mise en place de plusieurs machines qui vont faire les requêtes puis envoyer les résultats à une autre machine chargé des traitements.

Acronymes

DoS Deny of Service. 4, 8, 9

Sujet ASR #6 : Stratégie d'indexation pour un robot Web parallèle

Résumé

Le projet avait pour but d'étudier les différentes stratégies d'attaque qui pourrait être utile dans le cadre de la mise en place d'un robot d'indexation. Nous avons pour cela mis en place un programme nous permettant d'étudier ces mécanismes en lançant des requêtes sur une url donnée. Nous avons aussi étudié les différents moyens permettant de lutter contre ce type d'attaque.

Mots-clés

web, serveur, attaque, dos, scraping web

Abstract

Our project aims to discover several strategy who could be useful in the puropose of a web crawler. We manage to try those strategy by developping an application that can send request to a web server. We also search the way to protect a web server against those attack.

Keywords

web, server, attack, dos, web scraping

Tuteur académique
Mathieu Delalandre

Étudiants
Paul Fayoux (DI5)
Valérian Menin (DI5)