Antoine St-Denis
STDA01059305
st-denis.antoine@courrier.uqam.ca

Devoir 3

Bruteforceur XPath parallélisé

INF5171-20
18 décembre 2017

Document d'analyse remis à

Guy Tremblay

Université du Québec à Montréal

Le problème

Plusieurs applications web vont utiliser un fichier XML comme base de donnée. Ces applications vont généralement utiliser XPath, le langage pour faire des requêtes sur le fichier XML. Lorsqu'une application prend des données d'un utilisateur pour aller faire une requête sur ce fichier XML et que ces données ne sont pas assainies, il y a risque d'injection.

C'est lorsqu'il a possibilité d'injection que vient la possibilité du *XML Crawling,* ou exploration XML. Cette technique consiste à utiliser les fonctions: **count**() (pour trouver le nombre d'enfants d'un noeud), **string-length**() (pour trouver la longueur du nom d'un noeud ou de son texte) et **substring**() (pour trouver le nom du nom ou du texte).

L'injection ne nous permet que de retourner une réponse positive ou fausse, nous obligeant d'utiliser les fonctions plus hautes de manière méthodique et graduelle.

Par exemple:

```
<?xml version="1.0" encoding="UTF-8"?>
<data>
 <users>
   <user id="1">hentouane</user>
   <user id="2">admin</user>
   <user id="3">root</user>
 </users>
 <privateInfo>
  <creditCards>
    <creditCard id="1">123456789</creditCard>
  </creditCards>
 </privateInfo>
 <flags>
 </flags>
 <uqam>
 </uqam>
</data>
```

Sans connaître le fichier XML, l'approche serait la suivante:

count(/*[1]) = 1
-> Faux
count(/*[1]) = 2
-> Faux
count(/*[1]) = 3
-> Faux
count(/*[1]) = 4
-> Vrai
(Nous savons maintenant que le premier noeud possède quatres noeuds enfants: users, privateInfo, flags, uqam)

string-length(name(/*[1])) = 1
-> Faux
...
string-length(name(/*[1])) = 4
-> Vrai
(Le nom du premier noeud, data, est de longueur 4)

substring(name(/*[1]), 1, 1) = 'a'
-> Faux
...
substring(name(/*[1]), 1, 1) = 'd'
-> Vrai
(La première lettre du nom du premier noeud est 'd')
substring(name(/*[1]), 1, 2) = 'da'
-> Vrai
(Le 2 premières lettre du nom du premier noeud sont 'da')
...
substring(name(/*[1]), 1, 4) = 'data'
-> Vrai
(Nous avons trouvé le nom du premier noeud, nous pouvons chercher les 4 autres)

Comme on peut l'observer, ce sont des opérations qui pourraient s'avérer longues dans le cas d'un gros fichier à explorer. L'idée de mon travail était donc de paralléliser l'exploration XML afin de donner de meilleurs résultats.

L’approche

Pour résoudre ce problème, j’ai pris l’approche de d’abord utiliser l'interpréteur Jython qui, tel que JRuby, interprète le code Python en utilisant la JVM de Java, permettant du vrai parallélisme (tandis que le CPython est limité par le *GIL - Global Interpreter Lock)*.

J’ai décidé d’utiliser Python en raison de mon expérience avec ce langage, le très grand nombre de ressources en lignes pour aider à son utilisation et pour comparer avec mon utilisation de Ruby pendant ce cours.

J’ai opté pour l’approche Coordonateur/Travailleur. La tâche ici se trouve à être l’exploration d’un noeud et de son texte. Sans avoir accès à une librairie comme PRuby, l’idée en arrière de l’utilisation était de pouvoir utiliser une répartition dynamique de manière que je trouvais plutôt élégante et facilement applicable en Python.

Pseudocode, tel que adapté des notes du cours:

```
THREAD travailleur( sac_tâches )
        WHILE true DO
                obtenir une tâche du sac_tâches # Bloquant!
                IF il restait une tâche à exécuter
                THEN
                        explorer_noeud()
                        tâche.terminé? = true
                END
        END
END
```

Le sac de tâches ici fonctionne avec un *Queue* de la librairie *threading* de Python. Cette structure de donnée est de type *FIFO* et synchronise l’accès aux tâches, simplifiant sa gestion. Le reste des responsabilités du sac de tâches étaient la création des travailleurs, de la *Queue* ainsi que l’ajout de tâches.

Résultats

Temps d'exécution

Taille d'échantillon	**Séquentiel**	**Parallèle-1**	**Parallèle-2**	**Parallèle-4**	**Parallèle-8**
8	8.456	7.837	4.851	4.100	3.918
16	13.496	13.614	8.248	6.627	6.077
32	24.565	23.978	14.394	17.975	11.356

Explications:

Ici, on peut observer que le programme parallélisé a généralement une meilleure performance avec un plus grand nombre de travailleurs (Parallèle-1 signifie 1 travailleur, Parallèle-2 a 2 travailleurs, etc).

Accélérations absolues

Taille d'échantillon	**Parallèle-1**	**Parallèle-2**	**Parallèle-4**	**Parallèle-8**
8	1.079	1.743	2.062	2.158
16	0.991	1.636	2.037	2.221
32	1.024	1.706	1.367	2.163

Explications:

L'accélération semble devenir meilleure avec le plus de travailleurs ajoutés. À noter ici que les tests ont été effectués sur une machine à 4 coeurs. De plus, l'exécution à un seul travailleur reste très semblable à l'exécution séquentielle.

Accélérations relatives

Taille d'échantillon	**Parallèle-2**	**Parallèle-4**	**Parallèle-8**
8	1.616	1.911	2.000
16	1.651	2.054	2.240
32	1.666	1.334	2.111

Explication:

L'accélération relative semble s'enligner avec l'accélération absolue, pointant à une augmentation de l'efficacité par la parallélisation des opérations.

Temps d'exécution en CPython

Taille d'échantillon	**Séquentiel en CPython**
8	2.905
16	5.009
32	8.245

Explication:

L'exécution en CPython semble beaucoup plus efficace. Ce phénomène s'explique par l'utilisation de la JVM, qui est un processus assez lourd pour le démarrage. Il faudrait se rendre dans un document assez complexe et à une taille assez considérable pour que le programme parallèle Jython performe mieux que celui en CPython. L'utilisation d'une machine à nombreux coeurs viendrait aider aussi.

Conclusion

Par rapport à la version séquentielle utilisant le même interpréteur, la version parallèle de l'explorateur XML est plus efficace, surtout en augmentant le nombre de travailleurs et pour un document avec un plus grand nombre de noeuds.

La stratégie de test

Pour tester le programme, des tests d'intégrations ont été utilisés. Afin d'avoir une utilisation la plus près d'une réelle utilisation, un serveur local était lancé avec plusieurs pages PHP propres à la largeur du document (test-8.php, test-16.php, test-32.php). C'est avec ce serveur que le programme communiquait par protocol HTTP. Par la réponse du serveur, le programme pouvait voir si la commande était bonne ou non.

Les différentes tailles de documents utilisaient une même structure, de sorte à ce que les différents tests ne diffèrent que de par leur taille et non pas par leur profondeur.

Par exemple, le fichier user-pass-8.xml:

```
<?xml version="1.0" ?>
<data>
  <users>
      <user>0</user>
      <user>01</user>
      <user>02</user>
      <user>03</user>
      <user>1</user>
      <user>10</user>
      <user>11</user>
      <user>12</user>
  </users>
  <passwords>
      <password>12345</password>
      <password>abc123</password>
      <password>password</password>
      <password>computer</password>
      <password>123456</password>
      <password>tigger</password>
      <password>1234</password>
      <password>a1b2c3</password>
  </passwords>
</data>
```

Tous les autres fichiers comportent la même base, en plus de d'autres noeuds *user* et *password*.

Pour les mesures, chaque test était exécuté 3 fois et la mesure minimum était choisie.
Les résultats, soit les fichiers XML générés par l'exploration, étaient vérifiés manuellement afin d'assurer un bon résultat.

Conclusion

Concernant le langage Python, j'ai trouvé que son utilisation m'était plus aisé que pour le Ruby. J'ai aussi pu utiliser un *IDE (PyCharm)*, ce qui a grandement aidé ma productivité, alors que normalement je codais mes devoirs à distance, directement sur Japet. Le déboggage s'est avéré beaucoup plus facile aussi, à cause du déboggueur de *Pycharm*. Au final, Python reste un langage de script et les seules raisons pour lesquelles je le préfère au Ruby sont les innombrables librairies disponibles en ligne, de même que sa gestion des dépendances aisées pour les environnements unix.

Le programme résultant n'est pas tout à fait satisfaisant. En effet, la version CPython reste généralement plus efficace dans le cas de fichiers moins volumineux. Cependant, l'utilisation réelle du programme par une communication à distance plutôt que local pourrait donner des résultats plus intéressant.