A4 2cm1.5cm2cm1.5cm

Rapport du projet de Programmation Répartie
Un serveur HTTP

AYRAULT Maxime 3203694 & CARISTAN Mathis 3000038

04/01/2017

Notre programme prend en entrée un numéro de port *num_port*, un nombre de client *num_client*, et un volume de donnée limite par adresse IP par dix secondes *num_cpt*. Voici les commandes à saisir pour compiler et exécuter le programme :

```
$> touch .depend
$> make clean depend all
$> ./serveur.x 2500 5 5
```

Question 1 - Structure du serveur ✓

Nous détaillons ici dans un premier temps la mise en place du serveur, et son corps principal. Le programme commence par vérifier qu'il a été appelé avec suffisamment de paramètres. Ensuite, il effectue les préparations classiques pour un serveur TCP, à savoir, création d'une socket, et mise en écoute sur cellec-ci. En plus de cela, nous avons choisi d'utiliser la fonction `select()`, afin de pouvoir quitter proprement le serveur par le biais de l'entrée standard (en tapant 'QUIT' ou 'QUIT').
Lorsqu'un client se présente, le serveur vérifie qu'il peut accepter un nouveau client. Si c'est le cas, il va créer une thread qui s'occupera de dialoguer avec le client par le biais d'une socket dédiée. Dans le cas où il y aurait trop de clients simultanés, le serveur ignore simplement la connexion, et continue à attendre sur le `select()`.

La thread prend comme argument une `struct Client`, qui contient les informations nécessaires à la communication avec le client, ainsi qu'à sa gestion. Originalement[1], la thread avait le comportement suivant :

- Recevoir la requête sur la socket dédiée,
- Vérifier que la requête est correcte (commence par GET /, et finit par HTTP/1.x),
- Identifier le chemin, le nom et l'extension du fichier,
- Vérifier les droits d'accès aux fichiers,
- Parser le fichier *mime.types*, à la recherche du type de fichier approprié,
- Envoyer l'en-tête de réponse,
- Envoyer le contenu du fichier,
- Fermer la connexion.

Question 2 - Journalisation ✓

Pour gérer la journalisation, nous nous sommes inspirés d'un fonctionnement orienté objet. Nous avons créé une `struct Loginfo` qui contient toutes les informations qui doivent être écrites dans le fichier de log. Nous utilisons des fonctions pour remplir cette structure, et pour écrire ce qu'elle contient dans le journal. Ceci nous a permis d'assurer facilement que le remplissage de la structure était *thread-safe*. En effet, il est par exemple important de faire attention lors de la manipulation des fonctions liées au temps notamment, dans un contexte multi-threadé.
De la même manière, la gestion des accès concurrents au fichier est gérée par la méthode d'écriture dédiée, `WriteLog()`.

[1] Le comportement a été modifié à la question 4

Question 3 - Fichier exécutable ✓

Lors du traitement de la requête, nous avons vu plus haut que le serveur récupérait les statistiques du fichier demandé. Nous pouvons ainsi tester si le fichier est exécutable, si c'est le cas, le serveur va alors effectuer un `fork()`, et le fils exécutera le fichier.

Nous avons mis en place un systeme de communication interprocessus afin de pouvoir récuperer les information émisent par le fils pour le renvoyer au père qui enverra le tout au client.
Nous avons utilisé pour cela des tube només.
Ils ont pour nom `synchro` et `retour` suivi par le numero de thread du père.
–`synchro` est ouvert en lecture dans le fils et en ecriture dans le père, le fils attends de recevoir un feu vert `'1'` avant de commencer à faire quelque chose.
–`return` est ouvert en ecriture pour le fils et en lecture pour le père. Le père envoie au client tout ce que le fils lui donne. Le fils remplace la sortie standard `stdout` par le tube avec `dup2` puis execute le fichier executable avec `execl`.
Tout ce que le programme ecrit sur la sortie stardard et du coup redirigé vers le port d'écriture du tube.
Une fois tout ça fini le père detruit les deux tubes avec unlink, et reprend ce qu'il doit faire en temps normal.

Question 4 - Requêtes persistantes ✓

Pour pouvoir gérer les requêtes persistantes, nous avons modifié l'implémentation de notre serveur. Ainsi, la thread précédemment créée qui gérait le client, et la requête, ne s'occupe plus de la requête. De manière similaire à la boucle principale du serveur, la thread reste en attente de requêtes de la part du client. En recevant une requête, la thread (qu'on appelle par la suite *thread client*, va à son tour créer une thread (qu'on appellera *thread requête*), qui gérera la requête. Dans le cas ou plusieurs thread requête s'exécutent pour un même client, il faut veiller à la synchronisation des

réponses. Pour cela, nous avons choisi d'utiliser un système de mutex. Les threads vont traiter la requête, mais avant de pouvoir envoyer la réponse au client, elle doivent verrouiller un mutex personnel. Celui-ci est verrouiller dès sa création, et n'est déverrouiller par la thread requête précédente, que quand celle-ci a finit d'envoyer sa propre réponse. Remarquons que cette méthode nécessite, une attention particulière pour certains cas, notamment la première requête pour laquelle le thread client ne doit pas verrouiller le mutex après la création (puisque cette requête n'est précédée par aucune autre).
Nous avons choisi d'utiliser ce système plutôt qu'un système basée sur les signaux UNIX, pour une raison principale : il est possible dans de très rares cas que la synchronisation ne fonctionne pas, si le signal arrive précisément entre le démasquage de celui-ci, et le `suspend()`, créant alors un blocage.

Question 5 - Contrer le déni de services ✗

Nous n'avons pas réussi à mettre en place notre système pour contrer le déni de service. Voici, l'idée sur laquelle il se basait.
Lors de la réception d'une requête de la part d'un client, une fonction va parser le fichier de log à la recherche des entrées correspondant à ce client. Elle va ensuite additionner le poids de toutes les requêtes qui ont déjà été effectuées par ce client au cours des 10 dernières secondes. Dans le cas où poids de ces requêtes excèderait le maximum autorisé, ce client serait bloqué pendant 10 secondes. Si le client venait à ré-émettre une requête pendant ces 10 secondes, le compteur serait réinitialisé. À la fin du compteur, un SIGALRM, serait émis, et le client débloqué.