Guillaume Flandre Vivien Schilis Julien Topçu

PROJET - SYSTEMES D'EXPLOITATION

SERVEUR WEB

IFIPS INFORMATIQUE – CYCLE INGENIEUR 1 - 2007

SOMMAIRE

INTRODUCTION

Le projet qui nous est confié est de développer un serveur web, basé sur le protocole HTTP 1.1, pouvant gérer un maximum de connexions. Des mesures de performance seront faites afin d'évaluer les capacités de notre serveur.

Les serveurs web sont des applications qui demandent une attention particulière sur plusieurs critères. On peut citer notamment la performance, la gestion de la mémoire, le nombre de connec- tions établies. Lors de sa mise en production, le serveur web doit assumer une demande massive de connexions à la seconde et doit être prêt a dialo- guer correctement avec ses clients.

Aujourd'hui certains serveurs web se distinguent de ces différentes optimisations. On peut citer par exemple Apache qui est le plus utili- sé en production sur les machines UNIX notam- ment, mais aussi l'arrivée de Lighttpd (alias Lighty) qui se révèle très léger et très performant.

LE SERVEUR WEB

CONCEPTION DU SERVEUR

Comme tout serveur, par définition, notre serveur a été développé comme un démon prêt à recevoir des requêtes respectant le protocole de communication client-serveur HTTP. Une partie du protocole HTTP 1.1 a été développée, à savoir la méthode GET, la gestion des pages HTML et des fichiers binaires comme les images du type PNG, JPG, GIF.

Un processus principal va recevoir les demandes de connexions de façon linéaire afin d'établir une Socket de communication entre le client et le serveur. Si celui-ci a put être créé, un proces– sus propre au dialogue http va alors être lancé et va assurer l'échange des données entres les deux entités sous forme de requêtes.

Pour gérer ces requêtes, deux structures principales ont été mise en place.

```
/* Dans request.h */

typedef struct http_request

{

	int socket;



	http_method method;

	http_version version;



	char* host;

	char* uri;

	char* accept;

	char* accept_encoding;

	char* user_agent;

	char* connection;

	char* if_modified_since;

	char* entity;

}http_request_t;
```

```
/* Dans response.h */

typedef struct http_response

{

	int socket;


	char* entity;


	int content_length;

	char* content_type;


	http_status_code code;


	char* host;

	char* accept;

	char * lastmodif;

	int cache_age;

	int must_be_revalidate;


}http_response_t;
```

Le serveur en attente de données va recevoir la requête du client sous forme textuelle. Lorsqu'il a reçu la totalité de la requête (ligne vide), celui ci va « parser » les informations afin de récupérer celles qu'il sait traiter et va créer une structure de réponse à partir de la requête. Si le serveur sait traiter l'URI demandé (fichier présent, extension connue, etc...), celui-ci va alors remplir le corps de la réponse et l'envoyer au client. Si la mention "keep-alive" n'a pas été trouvée auparavant, il quitte le processus de dialogue.

GESTION MULTI-PROCESSUS

Une première version avec des solutions de *fork* fut au départ adoptée. Cependant il s'est avéré que cette solution était très coûteuse lorsque l'on commençait à recevoir un grand nombre de connexions.

Nous nous sommes donc arrêté sur une version multi-thread qui permet de mieux tenir les grosses charges. De plus, le partage de la mémoire avec le cache s'est avéré indispensable.

A chaque connexion établie, la fonction de dialogue est appelée à l'aide d'un *pthread_create*.

```
while (1)
	{
		sock_connected = httpd_socket_accept
				(spbok_socket_contact, (struct sockaddr *)&address);

		if (sock_connected < 0) {
			perror ("accept");
			return -1;
		}

		pthread_create (&thr, NULL, httpd_process, (void*)sock_connected);
		pthread_detach(thr);
	}
```

Pour ne pas qu'il y ait des zombies et libérer rapidement l'espace mémoire, un timeout permet de sortir du processus. Si au bout d'un certain temps le serveur ne reçoit plus de données de la part du client, il ferme le Socket et quitte le processus.

GESTION DU CACHE

Structure du cache

Le cache est un tableau en variable globale. Les éléments du tableau sont organisés de cette manière : nom du fichier, contenu du fichier, taille du fichier en octets, verrou sur fichier, statistiques d'accès, date de dernière modification, durée de validité avant revalidation et un indicateur booléen que nous expliquerons plus tard. La taille du tableau du cache est initialisée grâce à une valeur n par défaut, on alloue le tableau avec cette taille. Ensuite on remplie le cache, selon sa configuration, de m éléments. Une variable stockera la « taille du tableau » qui sera le nombre d'éléments réels dans le cache (ici m). Au cours de l'exécution du serveur, si m=n, on réallouera le tableau avec une taille plus grande.

Configuration du cache

Le cache se configure à l'aide du fichier *cache.conf* disponible dans le répertoire doc. L'utilisateur du serveur peut spécifier dans ce fichier le nombre et le nom des fichiers à mettre en cache. Pour chaque fichier, l'utilisateur doit spécifier la durée de validité en secondes de celui-ci dans le cache ainsi que sa popularité. Un seuil de popularité minimum devra être spécifié, il permettra au serveur de déterminer si une page « mérite » d'être mise en cache au cours de son exécution. Nous conseillons donc l'utilisateur d'y faire figurer les pages d'accueil des sites hébergés ainsi que tous les objets, comme par exemple des images, relatifs à ces pages.

Initialisation du cache

Le serveur va lire le fichier de configuration du cache s'il y en a un. Pour chaque nom de fichier présent dans le fichier de configuration, on va le charger en mémoire ainsi que sa date de dernière modification. On limite grâce à cela les accès disques. S'il n'existe pas de fichier de configuration, le cache reste vide. Le serveur dispose d'une valeur par défaut de popularité de fichier, il va donc au fur et à mesure de son exécution créer un cache.

Accès au cache

A chaque requête GET, on va vérifier si le fichier demandé est disponible en cache. Si tel est le cas, le serveur va donc créer un message à partir de la représentation en mémoire du fichier demandé en y incluant la durée de validité de celui-ci ainsi que sa date de dernière modification. Cependant, à chaque accès à un fichier en cache, on incrémente ses statistiques d'accès. Comme plusieurs clients peuvent vouloir accéder à un même fichier en cache, on va donc disposer d'un mutex pour chaque fichier. Chaque client devra attendre que le mutex soit libéré pour pouvoir accéder à la représentation en cache du fichier désiré.

Si le fichier désiré n'est pas dans le cache, le serveur va le chercher sur le disque. Il va aussi créer dans le cache, une représentation vide du fichier et initialiser ses statistiques d'accès (la représentation « vide » signifie qu'on ne charge pas le contenu du fichier en mémoire pour l'instant). Chaque fichier dispose d'une variable « booléenne » permettant de savoir si sa représentation en cache est une représentation vide ou non. Lorsqu'un client désire accéder à un fichier en cache disposant d'une représentation vide, on incrémente les statistiques d'accès de ce fichier mais le serveur agit comme-ci le fichier n'était pas présent de le cache. C'est le thread de mise à jour du cache qui s'occupera de charger le contenu du fichier en mémoire (cf partie mise à jour).

Ajout d'un élément dans le cache

Comme expliqué plus haut, on ajoute un élément dans le cache lorsqu'un client désire accéder à un fichier non présent dans le cache. Ceci nécessite d'incrémenter la variable de taille du tableau, comme plusieurs clients peuvent demander un fichier non disponible en cache, nous utilisons là aussi un mutex. Cependant nous utilisons la fonction *pthread_mutex_trylock* et non pas *pthread_mutex_lock*. *pthread_mutex_trylock*, comme son nom l'indique, essaye de verrouiller un mutex. Si le mutex est déjà verrouillé, le thread ne se met pas en sommeil, il continue son exécution. Si le thread arrive à verrouiller le mutex, on effectue l'ajout, sinon on lui donne 2 autres chances de le verrouiller (en rendant la main avec *usleep* entre chaque essai). L'ajout est effectué si et seulement si le nombre d'éléments déjà présents dans le tableau n'est pas égal à la taille allouée du tableau. Il n'y a que le thread de mise à jour qui réalloue le tableau de cache. Les ajouts ne peuvent créer de doublons car après avoir pris le mutex gérant l'ajout d'un élément, on vérifie si le fichier est déjà présent dans le cache même sous forme de représentation « vide ».

Mise à jour du cache

Un thread est chargé de mettre périodiquement le cache à jour. Il a pour mission de vérifier si la date de dernière modification du fichier sur le disque a changé par rapport à celle de la représentation en cache. Si c'est le cas on recharge le fichier en mémoire.
Pour les fichiers ayant une représentation « vide », le thread va vérifier s'ils ont atteint le seuil de popularité et le cas échéant, charge ses fichiers en mémoire.
La mise à jour du cache interdit donc la consultation du cache. Une variable booléenne (« *cache_locked* ») permet d'indiquer si le cache est en cours de mise à jour, un mutex est associé à cette variable. Lorsqu'un thread associé à un client désire consulter un fichier en cache, il doit prendre le mutex de cache_locked et lire sa valeur puis le rendre immédiatement. Si le cache est en cours de mise à jour, le thread n'ira pas le consulter.

Le thread de mise à jour, quant à lui, va prendre le mutex de cache_locked et positionner cet indicateur de mise à jour à vrai puis rendra le mutex de manière à ce qu'un autre thread puisse consulter la valeur de cet indicateur. Il existe une variable « *client_here* » qui est le nombre de threads qui actuellement consultent le cache. Cette variable est aussi protégée par un mutex, les threads associés aux clients vont l'incrémenter en entrant dans le cache et vont la décrémenter en y sortant. Le thread de mise à jour va donc consulter périodiquement la valeur de client_here (en la figeant grâce au mutex) en rendant la main

(*usleep*) jusqu'à ce que cette valeur soit nulle. Ayant positionné la valeur de *cache_locked* à vrai (cache en cours de mise à jour), il ne peut y avoir de thread qui entre en consultation du cache pendant que le thread de mise à jour attend que le nombre de clients dans le cache s'annule. Par sécurité, le thread de mise à jour va prendre le mutex associé à chacun des fichiers en cache puis va prendre le mutex d'ajout d'un élément dans le cache. Un thread associé à un client décrémente *client_here* avant d'ajouter un élément de manière à ne pas bloquer le thread de mise à jour. Ce thread « client » n'aura pas son exécution bloqué s'il désire faire un ajout alors que le thread de mise à jour a pris le mutex sur l'ajout car il effectue un *try_lock* sur celui-ci et abandonne l'ajout s'il n'arrive pas à prendre le mutex.

Si le tableau du cache est plein, le thread de mise à jour effectue une réallocation de celui-ci de taille + taille initiale d'allocation. Il y recopie ensuite les valeurs précédemment dans le tableau et initialise par défaut les valeurs restantes qui sont des cases vides (celles comprises entre taille et taille + taille initiale d'allocation).

Revalidation d'un fichier

Lors d'une requête de revalidation d'un fichier, on va tester si la date de dernière modification de ce fichier en cache est différente de celle présente dans la requête. Si ce n'est pas le cas, on envoie un message NOT MODIFIED code 304 dans la norme HTTP 1.1 au client. Sinon on envoie le contenu modifié du fichier ainsi que sa nouvelle date de modification. Lorsque que le cache est en cours de mise à jour, la consultation de celui-ci est interdite. Pour répondre aux requêtes de revalidation de fichier à ce moment là, on dispose d'une procédure fictif_cache qui va vérifier directement sur le disque la date de dernière modification du fichier désiré. Puis on effectue comme précisé ci-dessus à l'exception que lorsqu'on envoie le contenu du fichier, on accède directement au disque (le cache étant indisponible à ce moment là).

Enregistrement des données du cache

Lors de la fermeture du serveur, le serveur va écrire dans le fichier de configuration du cache l'ensemble des noms de fichiers en cache dans la popularité est supérieure ou égale au seuil de popularité avec leur durée de validité. De cette manière à l'exécution suivante le cache sera plus complet.

EXTRAS

Des petites fonctionnalités ont été ajoutées au serveur web afin de rendre plus facile et plus conviviale la gestion du serveur. On retrouve par exemple une fonction de “log”, classique aux serveurs web unix, qui va permettre de retracer plus facilement les connexions établies ou rejetées, les requêtes ou les problèmes survenus. On aurait pu ajouter une gestion “syslog” en option qui aurait été appréciable.

La gestion des configurations peut se faire via un fichier de configuration pour changer le port ou la racine du serveur par exemple. Ces options peuvent être fournies en ligne de commande par ailleurs (voir *spbok --help*).

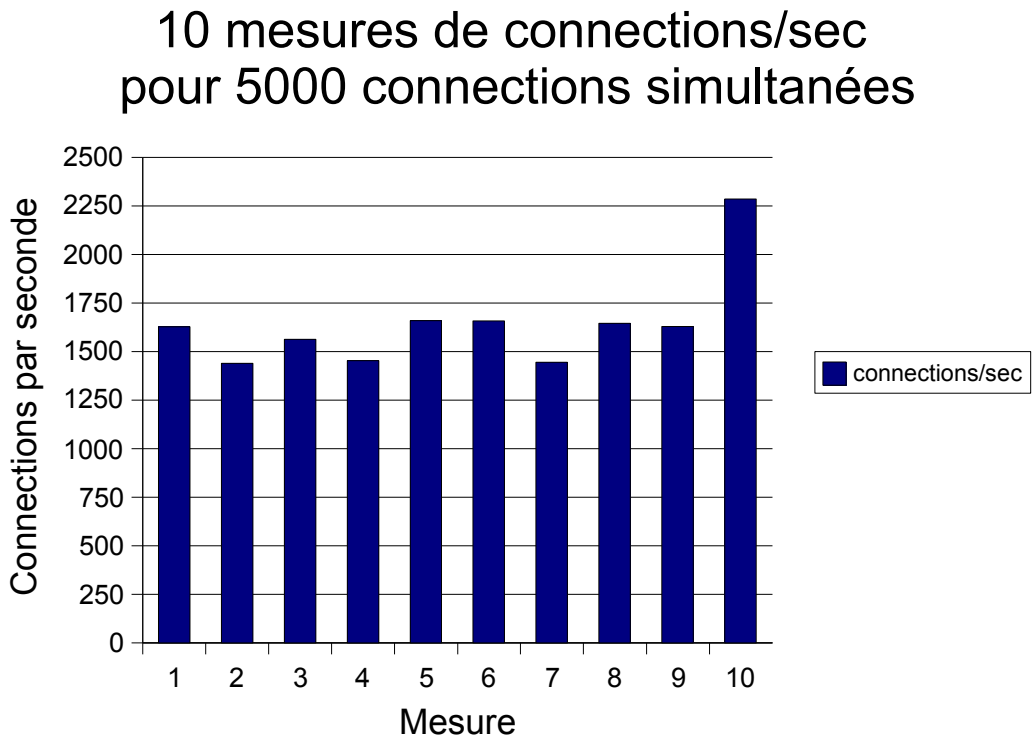
Exemple de fichier de configuration (voir *doc/spbok.conf*) :

```
################################################

# Configuration file for Springbok

# Commented variables will get default values

################################################


 port 80


# log  "/var/log/springbok"


# listen_on 127.0.0.1


 max_connections 1000;


 document_root "/var/www/springbok"
```

LES PERFORMANCES

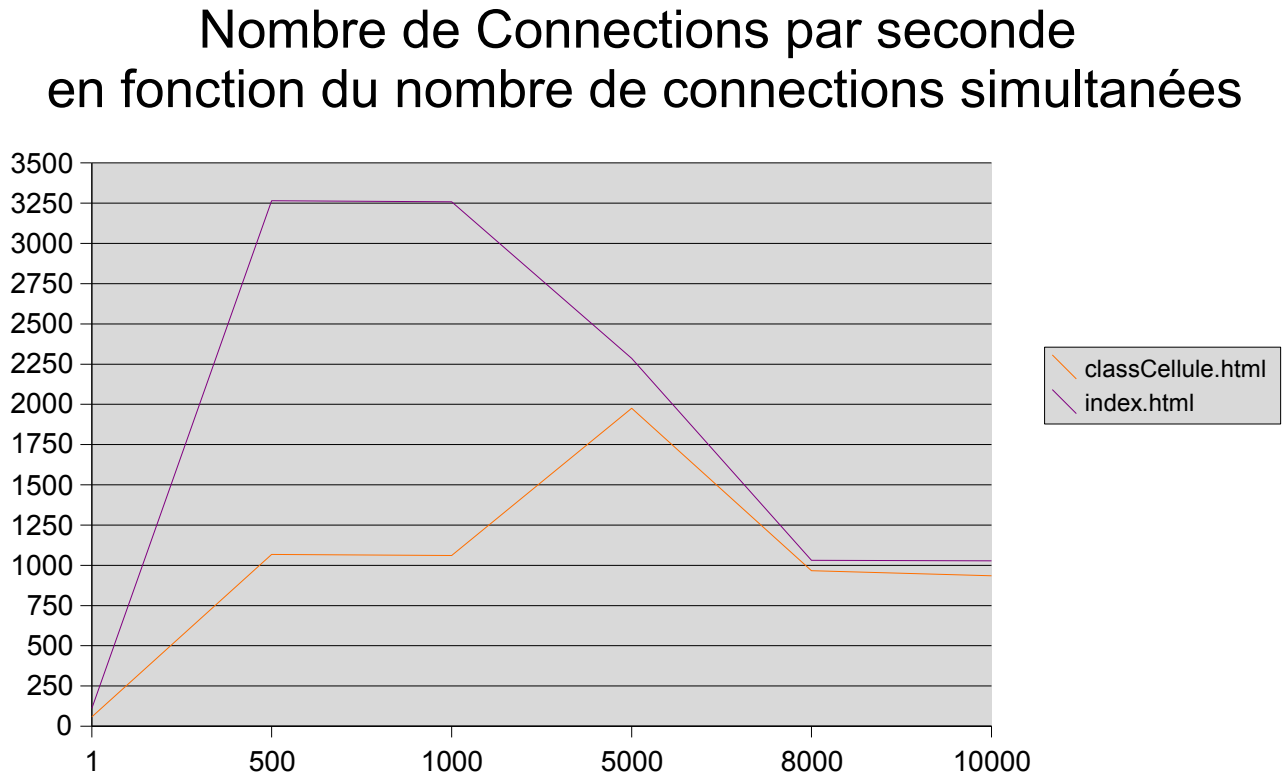
Les mesures de performance suivantes ont été effectuées sur un Centrino 1,73 GHz, 533 Mhz FSB, 2 MB de cache à l'aide de *httperf*.

Nous avons fait 10 mesures du nombre de connections par seconde pour le fichier *Index.html* de 961 octets et ceci pour 5000 connections simultanées.



On a une valeur moyenne de 1640,4 connections par seconde. Il n'y a pas beaucoup d'écart autour de cette valeur à l'exception de la barre 10 beaucoup plus grande que les autres.

Nous avons ensuite comparé le nombre de connections par seconde sur les fichiers *index.html* et *classCellule.html* 35633 octets en fonction du nombre de connections simultanées (ces fichiers sont tous deux présents en cache.



On remarque que certaines valeurs ne sont pas assez significatives, comme par exemple toutes celles inférieures à 5000 connections simultanées car la capacité étant supérieure à 1000 connections par seconde, 1000 connections ne suffisent pas pour avoir une valeur de connections par seconde assez fine. On a cependant la courbe d'*index.html* au dessus de celle de *classCellule.html*, ceci est expliqué par la taille nettement inférieure à celle de *classCellule.html*. On remarque aussi qu'au voisinage de 10000 connections simultanées, on tend pour les 2 courbes vers une valeur proche de 900 connections par seconde.

Remarques

Nous avons remarqué qu'on obtenait des performances allant de 2500 à 5000 connections par seconde en modifiant la priorité du serveur dans l'ordonnanceur pour 5000 connections sur *index.html*. Ceci explique la pointe à plus de 2250, le processus du serveur devait disposer de plus de ressources à ce moment là.

Le temps de réponse en local est de environ 1 ms pour toutes les mesures. Nous n'avons donc pas juger nécessaire de mesurer le temps de génération d'une réponse.

Nous n'avons pas pu mettre en évidence l'atout des directives de cache que l'on donne au client car nous n'avons pas trouvé d'outils permettant de mesurer le nombre de transactions issues d'un grand nombre de clients connectés en simultané et gérant le cache. De plus httperf ne permet pas de tester les sessions de connections sur notre serveur car il n'émet pas de requêtes contenant de Keep-Alive.
Mais nous avons pu voir grâce à l'utilisation de client comme firefox, que le nombre de requêtes était grandement diminué en utilisant les directives telles que expire et must-revalidate dans les réponses émises.

Une amélioration du serveur serait de passer aux polls ce qui diminuerait le nombre de thread et donc augmenterai le nombre de connections par seconde.

CONCLUSION

Travailler sur ce projet nous à permis de connaître en profondeur un outil familier qui peut paraître complexe par la dimension qu'il a pris ces dernières années au niveau mondial. Il en ressort que le protocole HTTP est facile à mettre en oeuvre, pour une gestion basique des pages web en tout cas.

Devoir gérer de bonnes performances sur la communication réseau nous à fait prendre conscience du travail réalisé sur les solides serveur HTTP. Prévoir les erreurs venant de l'extérieur, et s'adapter aux capacité de la machine nous fait comprendre que ce sont de nouvelles attentions sur lesquelles il faut se pencher lorsqu'il s'agit de développer des applications réseaux de ce type.