Rapport UE
Systèmes d’exploitation

Concurrency - web server

Étudiants: Abdallah ZERKANI, Jann SÁNCHEZ
Professeurs: Guillaume DOYEN, Renzo NAVAS

Date: 01/12/2024

Rappels des Objectifs du Projet

Le but de ce projet est de transformer un serveur web basique en un serveur multi-threadé.

Ce projet est l'occasion de travailler sur du code existant en le lisant, le comprenant, et en modifiant ce code pour y ajouter de nouvelles fonctionnalités.

Dans un premier temps, l'idée est de comprendre le fonctionnement d'un serveur web simple grâce au code de base fourni (comprendre les mécanismes du protocole HTTP et de la gestion des connexions via TCP).

Il s'agit ensuite de rendre le serveur concurrent pour améliorer ses performances en exécutant plusieurs requêtes simultanément, à la manière des serveurs "modernes".

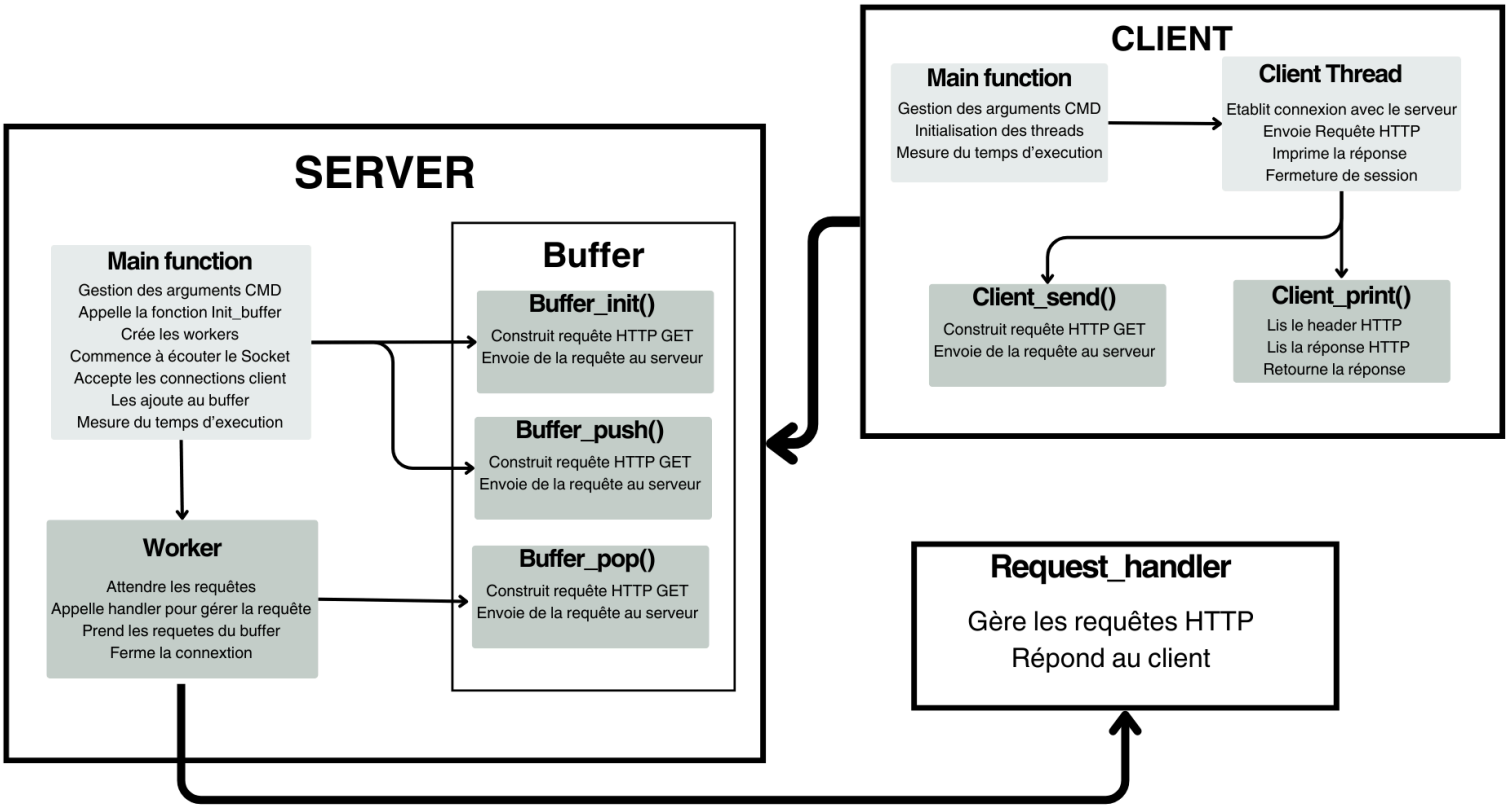
Le thread principal se concentre uniquement sur l'acceptation des connexions et la mise en file des requêtes dans le buffer. Les threads travailleurs se réveillent lorsqu'une requête est disponible, la traitent, et renvoient une réponse au client. Cette architecture permet une meilleure gestion des ressources et assure que des requêtes courtes ne soient pas bloquées par des requêtes longues.

La difficulté d'implémentation réside dans la gestion des ressource partagée (**consumer-producer problem**), il faut bien gérer l'utilisation des outils de gestion d'accès (mutex, variables de condition) afin que l'accès au buffer par le thread principal et les workers ne pose pas problème.

Après avoir mis cela en place, on testera le fonctionnement du multithreading grâce au code *spin.c* qui génère une requête et attend un certain temps avant de terminer le processus. Si le multithreading fonctionne, générer 3 fois cette requête avec 5 secondes d'attente devrait prendre 5 secondes environ et 15 secondes avec le serveur de base.

Le serveur doit pouvoir être configuré avec différents paramètres passés en ligne de commande, notamment le nombre de threads et la taille du buffer. Cela permettra de tester diverses configurations et d'adapter le serveur en fonction des charges. L'objectif est de trouver la ratio thread/buffer qui optimise les performances du serveur.

Diagramme des différents composants du code avec explication



Serveur

Composant	Function	Description
Serveur	Main Function()	• Cette fonction reçoit les arguments envoyés par l'utilisateur lorsque le serveur est démarré, ces arguments ont une valeur par défaut si l'utilisateur ne donne pas de valeurs. • Appelle la fonction Buffer_init() pour initialiser le buffer en accord avec les arguments sélectionnés. • Création des threads • Configuration du socket pour commencer à écouter les connections entrantes
Buffer	Buffer_init()	• Initialisation de la structure de données
	Buffer_push()	• Ajouter une nouvelle connexion au buffer

		s'il y a de la place disponible
	Buffer_pop()	• Retirer la connexion du buffer pour être traité par un thread
Threads	Worker()	• Attendre les requêtes de connexion dans le buffer • Chaque thread traite les requêtes HTTP en appelant la fonction request_handler() • Ferme la connexion une fois le traitement terminé

Client

Composant	Function	Description
Client	Main Function()	• Reçoit les arguments envoyés par l'utilisateur • Création des threads pour générer plusieurs requêtes simultanées
Utilisateurs	Client Thread()	• Etablir la connexion avec le serveur web • Appeler la fonction client_send() pour envoyer les requêtes HTTP • Appeler fonction client_print() pour recevoir la réponse. • Clôturer la connexion après avoir reçu la réponse.
	Client_send()	• Construit et envoie une enquête HTTP valide au serveur
	Client_print()	• Lire l'entête de la réponse HTTP • Lire et afficher le contenu de la réponse

Pseudo-code des différents algorithmes essentiels de l'application

Algorithm 1 Server HTTP

```
N: Number of worker threads
B: Size of the buffer
C: Buffer
S_running: Server running status (True/False)
Initialize the server with parameters: N, B
Create buffer C of size B
Create N worker threads
while S_running do
    Accept new connection conn
    if C is full then
        Reject conn
    else
        Push conn into C
    end if
end while
Stop worker threads
Clean up resources and terminate the server
```

Algorithm 2 Worker Thread Function

```
while S_running do
    conn = Pop a connection from C
    if conn = −1 then
        Exit the thread
    end if
    Process the HTTP request (using request_handle(conn))
    Send HTTP response to the client
    Close conn
end while
```

Explication

Algorithme 1 : On commence par initialiser les paramètres de base du serveur en définissant le nombre de threads de travail `N`, la taille du buffer `B` pour stocker les connexions, et une file d'attente `C` pour les connexions en attente. L'état du serveur est ensuite initialisé à "en fonctionnement" (variable `Running` à True).

On crée ensuite un buffer de taille `B`, qui est utilisé pour stocker les connexions qui arrivent lorsque le serveur est occupé. La taille du buffer limite le nombre de connexions en attente, ce qui empêche le serveur d'être

surchargé. Le serveur crée également `N` threads de travail qui traiteront les connexions une fois qu'elles auront été acceptées et mises dans le buffer.

Le thread principal, en boucle dans l'algorithme 1, attend des connexions clients. Lorsqu'une nouvelle connexion `conn` arrive, le serveur vérifie si le buffer est plein en utilisant `if C is full`. Si le buffer est plein, cela signifie que le serveur est surchargé, et la connexion est rejetée. Si le buffer n'est pas plein, la connexion `conn` est insérée dans le buffer, où elle sera récupérée par un des threads de travail.

Algorithme 2 : Les threads de travail permettent de gérer des requêtes HTTP. Chaque thread de travail récupère une connexion depuis le buffer et traite la requête correspondante. Un thread commence par tenter de "pop" une connexion de la file d'attente, ce qui signifie qu'il extrait une connexion à traiter. Si la file est vide (indiquée par `conn = -1`), le thread se termine, ce qui peut se produire lorsque le serveur arrête son fonctionnement.

Une fois une connexion extraite, le thread traite la requête HTTP. Cela inclut la gestion de la requête via la fonction `request_handle(conn)` et l'envoi de la réponse HTTP au client. Une fois le traitement terminé, le thread ferme la connexion avant de passer à la suivante.

Le serveur principal et les threads de travail fonctionnent de manière complémentaire (producteur-consommateur):

- Le serveur principal **accepte** les connexions et les **place** dans le buffer.
- Les workers **consomment** les connexions du buffer et les traitent simultanément. Chaque thread est responsable de traiter une requête.

Le thread principal est le producteur qui génère les connexions, et les threads de travail sont les consommateurs qui les traitent.

Tests effectués et état du code

Nous avons effectué plusieurs tests pour notre serveur web concurrent, nous avons également mis en place plusieurs méthodes afin d'effectuer les tests dans de bonnes conditions:

1) **Test du multi-threading:** Pour tester le bon fonctionnement du multi-threading nous avons tout simplement utilisé le code *spin.c,* ce script attend un certain temps après avoir reçu la réponse à sa requête, ce qui permet de ralentir le traitement des requêtes afin de constater la différence entre serveur simple et un serveur effectuant du multi-threading.

2) **Mesure de temps:** Pour avoir une comparaison quantitative entre un serveur utilisant le multi-threading et un serveur simple, nous avons ajouté sur le code client une mesure du temps d'exécution, cela permet d'obtenir le temps nécessaire, en millisecondes, pour gérer *n* requêtes client. Le but étant de constater que les requêtes sont effectivement gérées plus rapidement par le serveur multi-thread.

3) **Ecriture d'un script (temp_moyenne.sh):** Le but de ce script est de générer N nombre de requêtes M fois de suites pour retourner le temps moyen pour N requêtes clients. Cela permet de mesurer la performance de notre serveur web plus efficacement, en réduisant les interférences causées par les aléas des autres processus en cours d'exécution sur la machine.

4) **Séparer client et serveur via un réseau local:** On a connecté 2 machines au même réseau afin d'isoler la machine qui gère le côté du client et la machine qui gère le serveur pour éviter que les requêtes client ne prennent des ressources au serveur.

5) Taille Buffer et nombre de threads : Nous avons aussi essayé de trouver quel ratio entre la taille du buffer et le nombre de thread donnait les résultats les plus concluants en utilisant le script mentionné précédemment.

Le code actuel permet bel et bien de répondre à plusieurs requêtes sur notre serveur de manière simultanée. Cependant, un blocage peut se produire lorsque le nombre de requêtes envoyées devient très élevé (> 1000). Le

fonctionnement du code est donc partiel, car il fonctionne seulement pour un nombre limité de requêtes clients simultanées.

Conclusion : apprentissages et difficultés rencontrées

Grâce à l'utilisation du script *spin.c* nous avons pu confirmer que le serveur parvient à gérer plusieurs requêtes simultanément. Ce test a permis de simuler une charge sur le serveur, en introduisant un délai volontaire dans le traitement des requêtes. Pour 3 requêtes de 5 secondes, le serveur répond en 5 secondes environ et en 15 secondes pour le serveur sans multi-threading.

Toutefois, le serveur se bloque parfois lorsque trop de requêtes sont envoyées simultanément. Une explication possible de ce problème est une surcharge du système qui entraîne des dégradations de performance mais il est difficile d'affirmer avec certitude que c'est dû à cela.

Les résultats des mesures de temps n'ont pas permis de dégager de conclusion nette quant à l'impact de la taille du buffer ou du nombre de threads sur les performances du serveur. En effet, les durées observées pour de petites configurations (par exemple, un buffer de taille 2 et 2 threads) sont assez proches de celles observées avec des configurations plus larges. Cela pourrait s'expliquer par plusieurs facteurs. Une explication possible est le fait qu'un nombre faible de threads et un petit buffer pourraient suffire à gérer un nombre réduit de requêtes. La plus-value du multi-threading aurait donc lieu pour un nombre de requêtes simultanées plus élevées, dans des ordres de grandeur qu'on a du mal à tester étant donné les blocages côté serveur pour un grand nombre de requêtes.

Concernant les apprentissages, ce projet fut pour nous deux ludique et très formateur. Nous ne nous étions jamais plongés dans le code source d'un serveur web pour le configurer et l'optimiser, cela nous a ainsi permis de comprendre en profondeur comment les différents éléments d'un serveur web fonctionnaient et cela a solidifié nos compétences en programmation C par la même occasion.