Serveur Multithreadé en C

IRIE Joel
OLBORSKI Nathan

9 décembre 2021

1 Rappel des objectifs de l'existant

L'objectif de ce projet est de partir d'un serveur fournit codé en C, et en utilisant la bibliothèque pthread de multithreader ce serveur pour lui permettre de traiter plusieurs connexions en même temps.

Le code fournit nous propose un serveur, un client basique capable d'uploader un fichier, un fichier spin qui sert à patienter sur le serveur, et deux autres fichiers qui definissent des fonctions utilisées pour clarifier la gestion des connexions dans le code du serveur ou du client et enfin un Makefile qui gère la compilation du projet.

Pour modifier le code et transformer ce serveur en un serveur multithreadé les objectifs sont les suivants :

- Mettre en place une fonction qui sera appelée lors de la création d'un thread afin d'accepter et de gérer les connexions à travers ce thread (un thread-master)
- Mettre en place un ensemble de thread de taille fixe qui déterminera la quantité de thread simultanés maximale sur le serveur (les thread-worker)
- Faire en sorte que les threads soient correctement crées pour gérer chaque connexion sans que plusieurs threads essayent de gérer la même connexion et modifient la même ressource en même temps

La partie la plus importante du code fournie est la suivante :

```
int listen_fd = open_listen_fd_or_die(port);
    while (1) {
  struct sockaddr_in client_addr;
  int client_len = sizeof(client_addr);
  int conn_fd = accept_or_die(listen_fd, (sockaddr_t *) &client_addr, (socklen_t *) &
    client_len);
  request_handle(conn_fd);
  close_or_die(conn_fd);
    }
```

C'est dans cette boucle while que le serveur va gérer les connexions, listen_fd est un entier utilisé dans accept_or_die qui permet de déterminer le port sur lequel on écoute les connexions entrantes, cette connexion par un client va elle même être décrite par l'entier conn_fd. Il suffit ensuite d'appeler les fonctions request_handle et close_or_die sur cet entier pour gérer la connexion (forward les requêtes) puis la fermer. C'est dans les fichiers io_helper.* et request.* qu'on trouve les définitions de ces fonctions et la gestion d'erreur.

On précisera donc simplement que les connexions se font avec des stream sockets, une fois que le sokcet est ouvert un flux entrant et un flux sortant sont autorisés et les fonctions accept_or_die, request_handle et close_or_die agissent sur ce socket pour nous simplifier les différentes étapes de la connexion.

2 Les différents composants du code (1 diag, 1page)

L'architecture de notre solution finale est la suivante :

Décrivons avec plus de détail le processus qui a lieu lors d'une requête effectuée par le client :

- On commence par envoyer une requête http à l'aide du binaire compilé du wclient.c la requête envoyé est donc une requête HTTP car le code fournit envoie ses requêtes de cette manière,

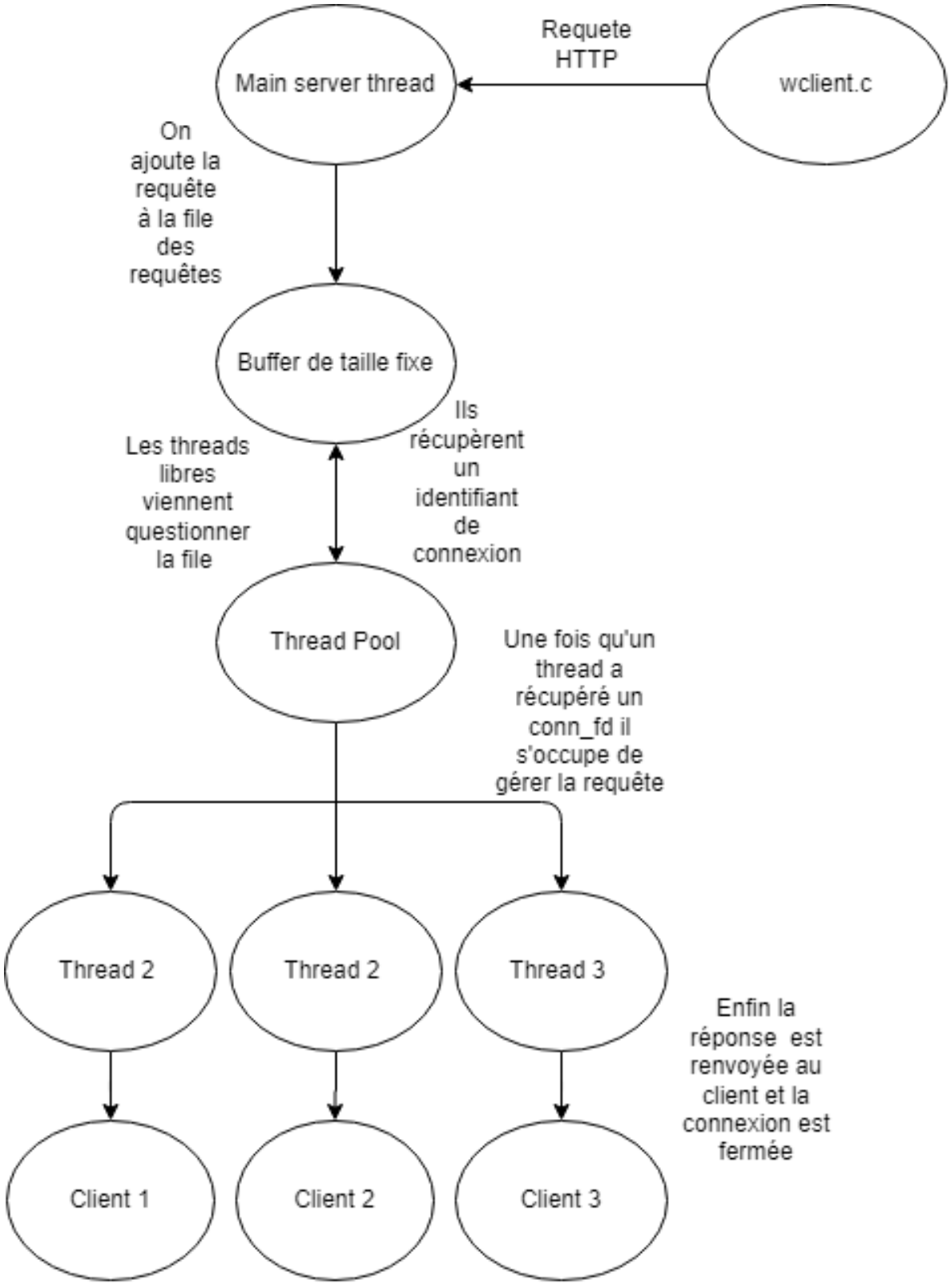

FIGURE 1 – Architecture de notre solution.

pour simuler une connexion réaliste on choisi comme fichier a fournir spin.cgi pour faire durer la connexion plus longtemps et ainsi simuler les temps de chargement

— Cette requête va être traitée par le serveur qui va lui associer un identifiant de connexion et placer cet identifiant dans le buffer, une fois dans le buffer cet identifiant sera récupéré par un thread travailleur afin de traiter la connexion. Si le buffer est plein et qu'on reçoit de nouvelles connexions le serveur ne devrait pas pouvoir accepter ces connexions
— Au fur et à mesure que le buffer se remplit, les différents threads vont récupérer les différents identifiants de connexion présents dans le buffer, les retirer du buffer (afin de faire de la place pour que le serveur puisse accepter de nouvelles connexions) et gérer la connexion, ce qui correspond à accepter le message du client et à lui renvoyer une réponse
— S'assurer que les différents threads qui modifient la file ne modifient pas les valeurs en même temps ce qui pourrait amener à une erreur de traitement, ceci revient à résoudre un problème du producteur/consommateur qui requiert l'utilisation de mutex et de variables conditionnelles que nous allons détailler dans la partie suivante

Les différents fichiers présents dans notre dossier avant la compilation sont les suivants :

— io_helper.c, io_helper.h, request.h et spin.cgi n'ont pas été modifés
— request.c a été modifié au niveau de la fonction request_handle afin d'empêcher l'utilisation du caractère '..' dans le nom du fichier contenu dans la requête afin d'empêcher l'utilisateur de parcourir tous les fichiers du serveur lors de la connexion d'un client. Cet ajout est donc présent pour des raisons de sécurité car c'est une faille qu'un attaquant pourrait exploiter afin de récupérer /etc/shadow ou /etc/passwd Cette implémentation n'est pas optimale car il existe des caractères spéciaux ou d'autre subterfuges pour réussir à accéder à des fichiers sensibles malgré notre ajout, mais par soucis de temps nous ne sommes pas allé chercher plus loin dans cette direction.
— wclient.c n'a pas été modifié non plus même si il était possible de le modifier afin d'avoir plus de détails dans nos requêtes, en affichant le temps que la connexion a mis à s'effectuer par exemple
— script_test est le script bash que nous avons utiliser pour simuler plusieurs connexions successives
— Afin d'implementer la structure de file FIFO (first in first out) nous avons créé un header local avec les deux fichiers queue.c et queue.h
— Makefile a été légèrement modifié, il a fallu rajouter le tag pthread dans la construction du binaire wserver afin de prendre en compte cette librairie, et également rajouter la construction de l'objet queue.o et son utilisation dans la construction de wserver.o afin que notre header local soit bien pris en compte
— Enfin wserver.c est le fichier qui a été le plus modifié afin d'implementer le multithreading du serveur et nous allons détailler les modifications dans la section suivante

3 Pseudo code et explications des algos essentiels (1 diag, 1 page)

Le diagramme en Figure 2 présente un cycle de fonctionnement lorsqu'un client se connecte, comme nous nous intéressons à la partie que nous avons fait par nous même nous ne détaillerons plus comment les connexions sont gérés car ce sont toujours les fonctions accept_or_die, request_handle et close_or_die qui s'occupe de ce travail. Nous allons maintenant commencer par présenter l'approche qui nous a permis d'arriver à cette solution à partir de l'existant.

Il faut d'abord savoir que pour multithreader un serveur il faut faire en sorte que les opérations effectuées par les fonctions fournies soit encapsulé dans une fonction qui sera appelée lors de la création d'un thread pour associer à ce thread la tâche de gérer les connexions, on se retrouve donc avec la fonction connection_handle suivante :

```
void* connection_handle(void* connection_descriptor){
    request_handle(connection_descriptor);
    close_or_die(connection_descriptor);
}
```

Sans prendre en compte les problèmes de type ou de renvoi, on a bien une fonction qu'on peut appeler lors de la création d'un thread, avec comme argument l'identifiant de connexion, afin de gérer

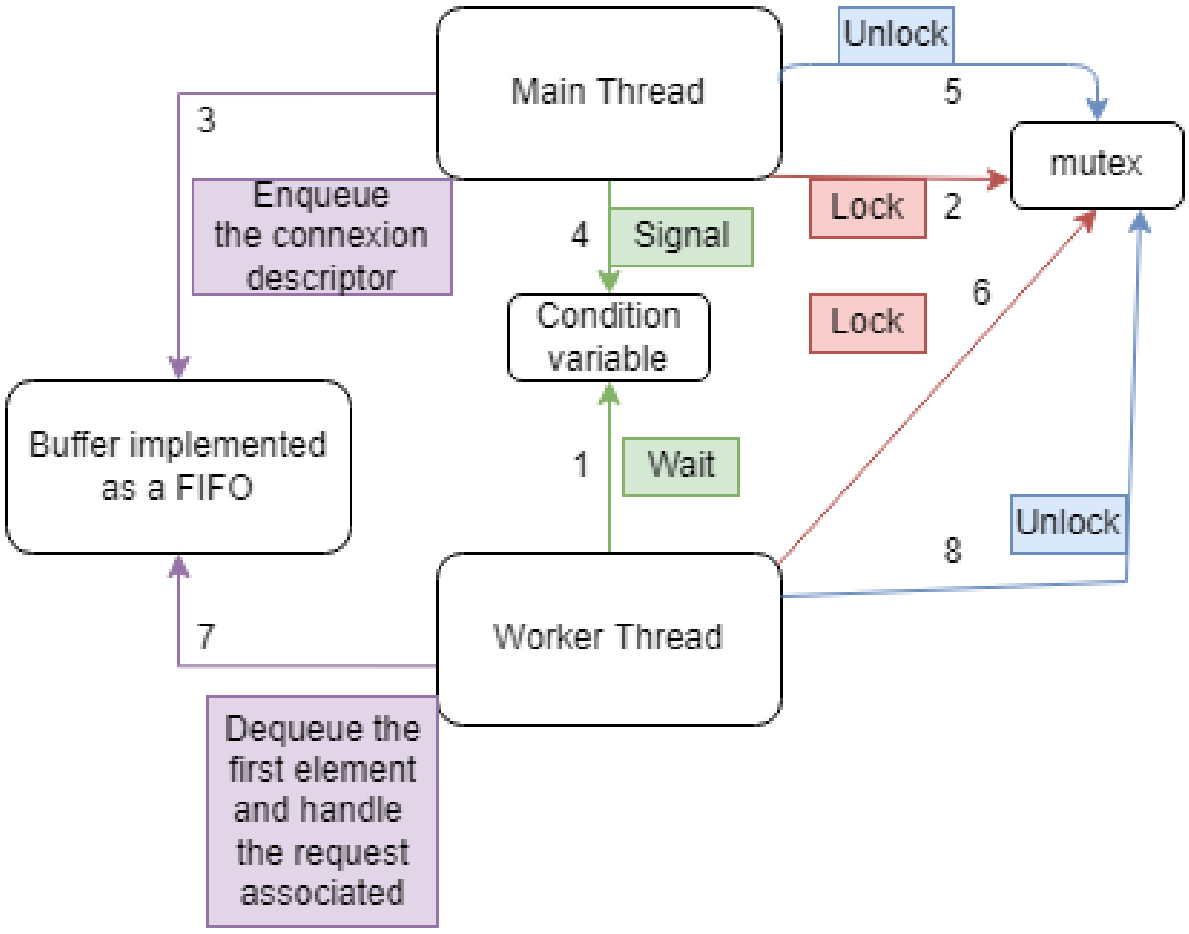

FIGURE 2 – Présentation d'un cycle de fonctionnement pour gérer une requête

une connexion et de la fermer. Même si notre serveur est désormais multithreadé cela ne suffit pas à remplir les conditions demandées, en effet si on veut utiliser un ensemble de thread de taille fixe afin d'empêcher notre machine de créer un thread à chaque fois qu'un client se connecte on ne peut pas se contenter d'appeler la fonction connection_handle car il faut désormais ranger les différents identifiants de connexion dans une structure de données à laquelle les threads vont devoir accéder pour récupérer ces identifiants. Commençons par détailler la structure de donnée utilisée pour le buffer :

```
struct noeud {
    int* head;
    struct noeud* tail;
    };

typedef struct noeud noeud_t;

void enqueue(int *conn_fd);
int* dequeue();
```

Ici on définit une structure appellée liste chainée qui tient son nom du fait que lorsqu'on regarde un noeud de la liste on a accès à la valeur de ce noeud et à la liste chainée à partir du noeud d'après (la tail).

Les deux fonctions décrites à la suite respectent une implémentation FIFO, First in First out, ce qui veut dire que si on ajoute 3 élements e1, e2, e3 à la liste alors les 3 appels consécutifs de dequeue renverront dans l'ordre e1, e2, e3, le premier élément à entrer dans la liste est le premier à sortir, ou encore le premier client qui se connecte au serveur est celui qui voit sa requête traitée en premier.

Une fois que cette structure de donnée est mise en place les deux dernières choses à faire sont de s'assurer que cette strucutre de données soit modifié correctement par les différents threads, et aussi de faire en sorte que les threads ne questionnent pas constamment la liste pour savoir si il y a des connexions à traiter afin d'optimiser les performances du serveur.

Ces fonctionnalités s'implémentent de la manière suivante :

Dans la boucle principale du serveur :

```
while (1) {
        int conn_fd = accept_or_die(args);
        mutex_lock();
        enqueue(p_conn_fd);
        signal(condition)
        mutex_unlock();
    }
```

La nouvelle fonction appelée lors de la création d'un thread devient :

```
void* thread_worker(void *arg){
    while(1){
        mutex_lock();
        if queue == EMPTY {
            wait(condition);
        }
        int conn_fd = dequeue();
        mutex_unlock();
        if (conn_fd != NULL){
        connection_thread_handler(conn_fd);
        }
    }
```

L'utilisation d'un mutex et d'une variable conditonelle nous permet de résoudre le problème du producteur/consommateur de la manière suivante :

On souhaite s'assurer qu'un thread ne modifie la structure de donnée que si il possède le mutex, pour le main thread qui s'occupe uniquement de remplir la file avec les identifiants de connexion cela se traduit par l'appel à mutex_lock() avant d'ajouter des valeurs dans la file. En revanche si on utilisait simplement un appel à mutex_lock dans notre worker thread le problème rencontré serait que nous serions obligé de constamment questionner la file afin de savoir si de nouveaux identifiants de connexion ont été ajouté à la file et cela consommerait beaucoup de ressources inutilement. La solution à ce problème est donc d'utiliser une variable conditionnelle dans notre implémentation le

worker thread va attendre sur cette variable conditionnelle si il voit que la file est vide, ainsi quand le main thread ajoutera un identifiant à la file, l'appel à signal() va réveiller TOUS les threads qui dorment sur cette condition qui vont ainsi recevoir le mutex tour à tour pour accomplir leur tâche. Il faut alors s'assurer qu'on puisse faire des appels dequeue() lorsque la file est vide, ici tout fonctionne correctement car dequeue() renvoie NULL si la file est vide le thread retourne donc attendre si il a été réveillé mais que la file était de nouveau vide lorsqu'il à reçu le mutex et essayé de dequeue.

Donc on aura bien le fonctionnement suivant : lors du lancement du serveur le buffer et le thread pool sont initialisée selon les valeurs par défaut ou celles données par l'utilisateur et tous les threads vont recevoir comme tâche d'attendre que la condition soit signalée ce qui sera le cas lorsqu'on aura des clients qui se connectent au serveur.

NB : Dans notre implémentation la structure de donnée externe est une liste chainée, qui, par nature n'a pas de taille maximum (on ajoute des éléments à la chaine), cependant il n'est pas impossible de rajouter à notre liste chainée une variable taille_max qui définirait la taille max de notre liste et refuserait d'ajouter des éléments à la file si la taille de cette dernière a atteint la taille maximale autorisée. Le problème avec cette implémentation est que nous n'avons pas réussi à utiliser l'input de l'utilisateur lors du lancement du serveur afin de redéfinir cette taille maximale. La solution serait alors d'utiliser des arrays avec un jeu d'indice afin de simuler une file fifo. Par exemple, en gardant en mémoire le nombre d'éléments présents dans le tableau on ajoute un élément à la suite des éléments déjà présent dans le tableau lors de l'ajout d'une valeur (si le tableau n'est pas déjà plein) et lorsqu'on souhaite récupérer une valeur on récupère le premier élément du tableau, on décale tous les éléments du tableau vers la gauche et on n'oublie pas de décrémenter le compteur d'objets dans le tableau. Malheureusement par un soucis de temps nous n'avons pas eu le temps de mettre en place cette implémentation et avons donc décidé de garder notre liste chainée, qui présente tout de même l'avantage d'être plus rapide en terme de compléxité que la solution présentée avec des tableaux.

4 Tests effectués (1 page)

Pour la totalité des tests on utilise le script suivant :

```
if [[ $# -ne 1 ]]; then
        echo "mauvaise utilisation"
        exit 1
fi
NUMBER=$1

for (( i=0; i<$NUMBER; i++)) do
        ./wclient 127.0.0.1 10000 /spin.cgi?5
done
exit 0
```

Ce script très simple prend en argument un seul paramètre n (un entier) et appelle le client n fois vers le serveur local vers le port 10000 qui est le port par défaut de notre serveur.

Le fichier spin.cgi qui était fourni dans le code de base permet de simuler une connexion "réaliste" d'un client qui n'aurait pas forcément un débit de connexion très rapide, ou tout simplement un client qui se connecte à un site web qui ne possède pas juste une page de garde en HTML mais pleins de scripts JS et d'appels vers différentes API ce qui est le cas de beaucoup de site et qui ralenti beaucoup la réponse à la demande du client. L'appel ./spin.cgi ?5 précise à notre script de test que l'argument à fournir à spin.cgi est 5, chaque connexion attendra donc 5 secondes sur le serveur avant de se déconnecter

Lors du développement de notre solution la grande majorité des modifications étaient apportées au fichier wserver.c la façon la plus simple de tester la validité de notre solution est de compiler notre code et de faire tourner le serveur en local sur le port 10000 avant d'appeler le script ci dessus afin de tester plusieurs connexions quasi simultanées.

Pour vérifier le bon fonctionnement de notre solution

Afin de déterminer de manière efficace d'où venaient les différents problèmes rencontrés lors du développement nous avons utilisé de nombreuses occurrences de la fonction printf pour déterminer à quel étape le serveur arrêtait de fonctionner mais également pour déterminer si le multi-threading

fonctionnait bien à l'aide de $printf(\%ls, pthread\_self())$/ ce qui va afficher l'id du thread qui appelle cette ligne de code, ainsi on pouvait voir qu'un thread différent traitait nos requêtes et donc que le serveur est bien multithreadé

5 Conclusion et difficultés rencontrées

Même si le travail demandé pour ce projet pouvait paraître simple lorsqu'on réfléchit aux différentes étapes à mettre en place, le fait de travailler avec un langage aussi intransigeant que C a rajouté beaucoup de difficultés au projet et donc une bien meilleure compréhension de ce langage à la fin du projet.

La réalisation de ce projet nous a également permis de mieux comprendre la compilation de projet en C avec plusieurs fichiers dont des headers locaux, nous avons du modifier le fichier Makefile afin de faire en sorte que le projet se compile bien, en prenant en compte la bibliothèque pthread par exemple.

Parmi les difficultés rencontrés, de nombreux problèmes venaient des types qui n'étaient pas les bons lors de l'appel de fonctions posix comme pthread_create, en effet les différents cast à effectuer pour avoir le bon type de variable à chaque appel de fonction requiert une bonne compréhension des pointeurs. De plus le code source a été difficile à aborder au début à cause des différentes assertions présentent dans les headers qui modifiaient le nom des fonctions mais finalement ce fut agréable de se sentir à l'aise sur ces notions à la fin du projet.

La difficulté majeure que j'aimerai mentionner et qui aura bloqué notre progression pendant plusieurs jours et n'est toujours pas résolu à ce jour est la suivante ;

Pendant la totalité du projet toutes les compilations on été effectuées sous Ubuntu 20.04 avec la version la plus récente de gcc. Cependant selon le poste de travail Ubuntu était supporté soit par une machine virtuelle soit par Windows Subsystem for Linux.

Or il s'avère que notre serveur n'a jamais réussi à accepter une seule connexion sans rencontrer une erreur fatale sous WSL. A l'époque où nous avons rencontré ce problème il faut savoir que notre serveur créait encore un thread à chaque connexion et le problème était que lors qu'un client essayait de se connecter l'appel à la fonction pthread_create() causait un Segmentation fault et fermait le serveur. Le première hypothèse était que nous accédions à de la mémoire protégée en l'occurence nous avions eu la même erreur lorsque nous n'utilisions pas de pointeur vers l'identifiant de connexion. Cependant le fait que le serveur fonctionne correctement sur une machine virtuelle est pas sous WSL alors que le code source était identique était tout de même étrange. Pour plus d'information j'ai décidé de recompiler le serveur en utilisant GDB le GNU Project debugger ce qui m'a permis de savoir exactement quels appels causaient notre erreur.

Les résultats fournis montraient bien que l'erreur survenait au moment de l'appel de pthread_create() mais le problème était que l'adresse à laquelle se situait le pointeur n'était pas reconnu par le système et donc le thread ne pouvait pas être créé. Ma supposition est que l'adresse donnée à un pthread_t à peine initialisé est conservé par windows est donc la fonction pthread_create n'avait pas accès à cet adresse.

Finalement c'était un sujet assez intéressant, car peu compliqué en terme de concept (en supposant être déjà familier avec le fonctionnement d'un serveur HTTP) mais qui nous a permis de vraiment apprendre C à travers l'expérience. Cependant le langage C reste relativement compliqué à aborder, par exemple les structures de données ne sont pas utilisées comme dans les autres langages, mais cela fut très intéressant pour apprendre un langage à bas niveau.