Louis Lefebvre : louis.lefebvre@imt-atlantique.net
Lucas Rakotoarivony : lucas.rakotoarivony@imt-atlantique.net

PROJET OS

RAPPORT TECHNIQUE

Version - 1.0

Année scolaire 2021-2022
Date de remise du rapport : 12 décembre 2021

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

RAPPEL DES OBJECTIFS ET DE L'EXISTANT

Ce projet a pour objectif de développer un serveur web concurrentiel capable de répondre à de multiples requêtes simultanément.

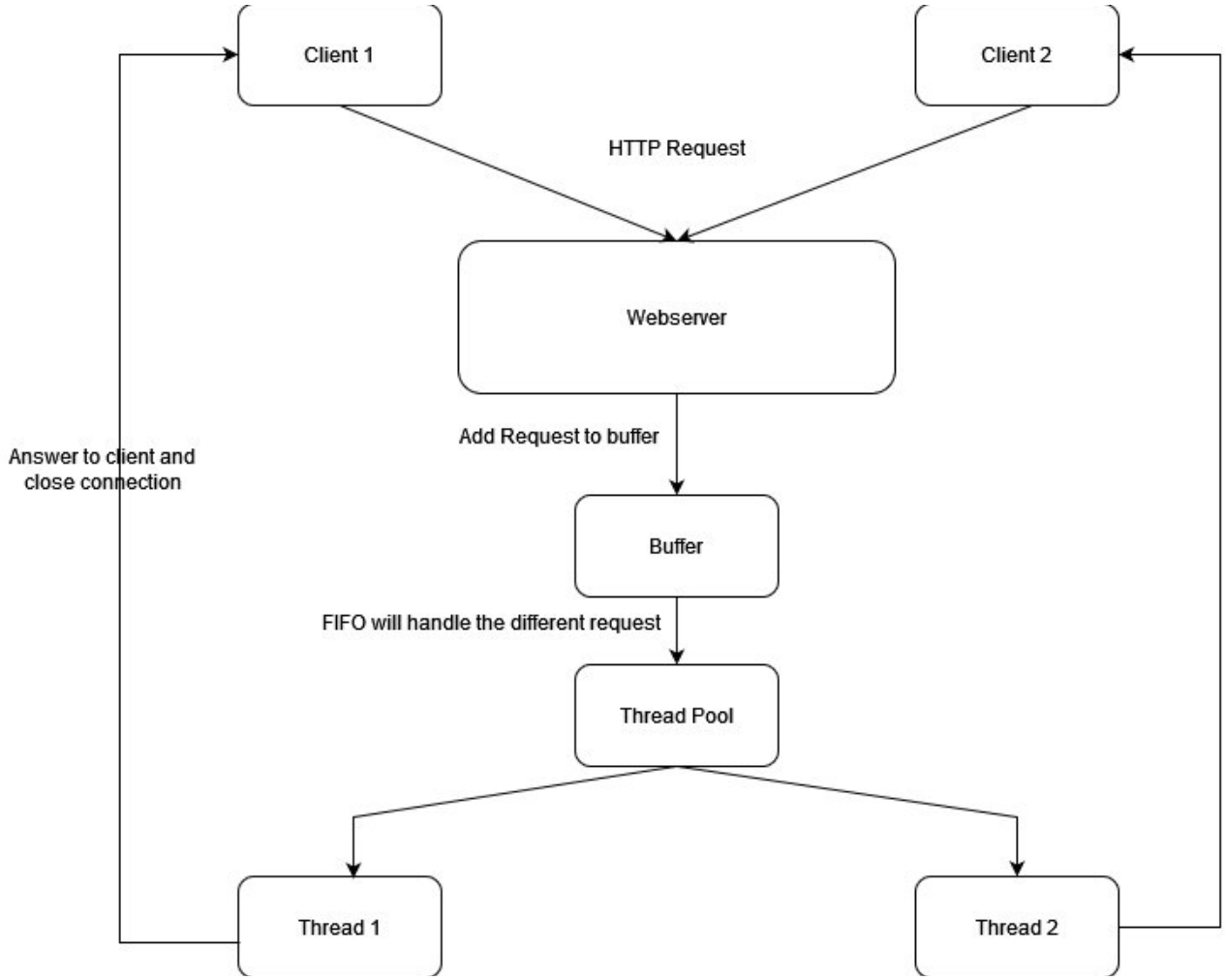
Pour cela nous disposons du code minimal d'un serveur web fonctionnel mais non concurrentiel. Ce serveur n'utilisant qu'un seul thread, l'objectif principal de ce projet est d'implémenter un web serveur multithread pour gérer de nombreuses requêtes.

Afin d'atteindre cet objectif, nous allons tout d'abord devoir comprendre l'architecture de base du serveur web non concurrent déjà fourni et le prendre en main. En effet, ce serveur web est fonctionnel mais très peu commenté ce qui peut complexifier la compréhension de ce code.

Par la suite, il va falloir réussir à ajouter de la concurrence via l'utilisation de plusieurs threads et un thread pool. Cela représentera le principal travail de ce projet, pour cela nous allons utiliser la librairie pthread de c.

Enfin, ce serveur web pourra être complexifié en développant notamment une structure de gestion de requête de type FIFO ou en prenant en compte les différentes extensions des fichiers demandés afin d'éviter des failles de sécurité.

DIAGRAMME DES DIFFÉRENTS COMPOSANTS DU CODE AVEC EXPLICATION



Notre code s'articule selon le diagramme précédent

Tout d'abord on initialise un web serveur et on peut définir un certain nombre de thread, une taille de buffer et un port d'écoute. Par défaut, on définit le nombre de threads à 5 , une taille de buffer de 10 et le port d'écoute est le port 10000.

Dans un premier temps, un certain nombre de clients (dans cet exemple 2 mais cela peut être généralisé), envoie des requêtes de type HTTP au web serveur. Les informations et différentes fonctions liés à cette partie sont présentes dans le fichier wserver.c.

Ces requêtes vont ensuite être stockés dans une file de requête (ou buffer) pour être ensuite pris en charge par la thread pool du serveur web quand des threads libre seront disponible.

On utilise une structure de type FIFO (First In First Out) pour faire en sorte que les requêtes arrivés en premier soit traités en priorité. Cette structure FIFO est implémenté dans le fichier pool.c ainsi que les différentes fonctions liées au multi threading. Les fichiers pool.c et pool.h représentent nos principaux ajouts par rapport à la version de base du projet.

Une fois la requête dans le thread pool, un thread sera attribué à la requête (en respectant la structure FIFO) et elle sera donc traité en parallèle des autres requêtes si le nombre de thread disponible est suffisant pour gérer toutes ces requêtes. Pour l'attribution des requêtes par thread, les threads libres vont interroger la file de requête pour savoir si des requêtes sont en attente de traitement et si oui traiter en priorité la première arrivé selon la structure FIFO. Le thread va également récupérer un identifiant de connexion lié à une requête pour montrer que ce thread est en charge de cette requête

La requête sera ensuite traité grâce aux différentes fonctions définies dans le fichier request.c. Les requêtes seront traités en fonction de différents attributs (dynamique ou statique, GET ou POST) et une réponse sera produite en fonction de la requête. Une erreur peut également apparaître si les paramètres de la requête étaient incorrects.

Une fois la requête traitée et une fois qu'une réponse est disponible, le web serveur envoie la réponse au client de manière parallèle aux autres réponses grâces aux différents threads. La connexion avec le client est ensuite coupée et le serveur web libère ensuite le thread chargé de cette requête afin de le rendre disponible pour une autre requête ou de gérer une autre requête présente dans le buffer.

PRÉSENTATION EN PSEUDO-CODE DES DIFFÉRENTS ALGORITHMES ESSENTIELS DE L'APPLICATION

webServeur :

```
lire port
lire nbThread
lire tailleBuffer
pool <- creerPool(nbThread, tailleBuffer)
ouvrePort(port)
//On ouvre la pool et le port à partir des arguments donnés lorsque le serveur est lancé
boucle infinie
    requête <- accepteRequete(portOuvert, adresseClient, tailleAdresse)
    si requête faite alors
        ajouterRequete(requete)
//Si une requête est faite, on accepte la connection et on envoie la requête dans la pool de thread
```

gestion de la pool(pool) :

```
tant que pool.débutPile = pool.finPile alors
    bloqueJusquaCondition(pool.nonVide)
//Si la pile FIFO est vide, on a pas de tâche à donner aux threads donc on attend
tache <-pool.pile[pool.débutPile]
pool.débutPile <- (pool.débutPile + 1) mod pool.tailleMax
pool.threadPret <- pool.threadPret + 1
```

```
//Il y a des tâches dans la pile donc on retire la première
débloqueThread(pool.mutex)
effectueTache(tache)
verouilleThread(pool.mutex)
pool.threadPret <- pool.threadPret - 1
débloqueThread(pool.mutex)
```

ajoutRequete(pool, routine, arg) :

```
bloqueThread(pool.mutex)
si pool.débutPile = pool.finPile alors
    signaleCondition(pool.nonVide)
//On signale si la pool est vide ou pleine
req.routine <- routine
req.arg <- arg
//On met la requête sous la bonne forme et on l'ajoute à la pile
pool.pile[pool.finPile] <- req
pool.finPile <-( pool.finPile + 1) mod pool.tailleMax
debloqueThread(pool.mutex)
```

Structure FIFO :
On travaille avec une pile de type FIFO. Pour cela on stocke dans un tableau de taille fixe deux compteurs, pour connaître où sont le début et la fin de la pile. Quand une requête arrive à la fin de la pile, on l'ajoute à la fin. Quand une requête est traitée, on la retire du

début. Si on arrive aux limites du tableau, on utilise un modulo pour revenir au début. Notre tableau fonctionne donc comme une boucle.

TESTS EFFECTUÉS ET ÉTAT DU CODE

Actuellement, notre serveur web fonctionne de manière concurrentiel avec plusieurs threads et en respectant une structure FIFO (First In First Out). Cela signifie que ce sont les requêtes arrivés en premier qui sont traités en priorité et que les requêtes ne pouvant être traités immédiatement sont stockés dans un buffer.

Pour vérifier que notre serveur web est bien capable de gérer plusieurs requêtes simultanément nous avons utilisé la page spin.cgi. En effet, cette page permet, à l'aide d'un paramètre, d'occuper un thread pendant plusieurs secondes. Nous avons donc effectué plusieurs tests où on définissait un web serveur avec N threads, on lançait ensuite de manière quasi simultané à l'aide d'un script bash ou manuellement sur navigateur N+2 fois la page spin.cgi avec comme paramètre 10 pour que la requête dure 10 secondes. Et comme prévu, notre serveur web a géré les N premières requêtes en 10 secondes et s'est ensuite occupé des 2 requêtes restantes. Ce test a donc bien démontré que notre serveur web était concurrentiel.

Notre serveur web implémente également la gestion des paramètres demandés en effet la commande

./wserver -d . -p 8003 -t 8 -b 16

crée bien un serveur à la racine sur le port 8003 avec 8 threads et 16 buffers.

Cependant, par manque de temps, nous n'avons pas pu implémenter toutes les fonctionnalités que nous souhaitions. En effet, nous avons effectué des tests pour vérifier la robustesse de notre serveur par rapport à des attaques de type DOS. Pour cela, nous avons utilisé le fichier client_FIFO.sh qui lance de multiples requêtes via wclient. Nous avons observé que après un certains nombre de requêtes (environ 200) pour une raison inconnue notre serveur crashait. Cela consiste une piste d'amélioration pour améliorer notre serveur dans le futur.

De plus, lors de nos tests de serveur sur différentes infrastructures., nous avons remarqué que notre web serveur ne fonctionnait pas sur WSL. Nous avons discuté de ce problème avec certains de nos camarades travaillant également sur ce projet et nous avons constaté que ce problème était récurrent dans les développements de serveur web concurrentiel. Nous n'avons pas encore trouvé la source de ce problème et une solution pour l'éviter.

Globalement, notre serveur web fonctionne bien de manière concurrentiel avec toutes les options demandés dans le cadre du projet. Il existe cependant des erreurs qui peuvent apparaître dans des conditions particulières mais dans le cas d'un usage classique cela ne devrait pas poser de problème.

CONCLUSION : APPRENTISSAGES ET DIFFICULTÉS RENCONTRÉES

Durant ce projet, nous avons améliorer nos compétences en C à travers la compréhension du code et l'implémentation de nouvelles fonctionnalités. En effet, lors du début de ce projet, nos compétences en C étaient très limités. Il est intéressant dans le cadre de notre formation de se familiariser avec un langage de programmation de bas niveau tel que le C.

De plus, nous avons également appris à gérer des threads et un serveur web de type concurrentiel grâce à la gestion des différentes requêtes dans une structure de type FIFO. Nous avons choisi une structure FIFO car elle facilement compréhensible et de nombreux exemples de ce type de structure sont déjà disponible en ligne.

La principale difficulté rencontrée est la compréhension du code initial. En effet, comme mentionné précédemment, le code de base est très peu commenté et complexe. De plus le C étant un langage de bas niveau complexe, il nous a fallu beaucoup de temps pour comprendre la syntaxe et l'utilité de chaque blocs de code.

L'implémentation du serveur avec plusieurs threads fut complexe mais à l'aide du cours Processes and Threads de cet UE, cela ne fut pas trop difficile à développer à partir du code initial.

Pour conclure, durant ce projet nous avons développer un serveur web concurrentiel à partir d'un serveur web non concurrentiel. Ce projet nous a notamment permis de développer nos compétences en C durant l'implémentation de la structure multi threads.