Rapport technique

Etape 3: Serveur TFTP Multi-clients

Licence 3 informatique - MIAGE

Encadrant : Mme B.WAFA & M G.UTARD

Binôme : Karamoko SAMASSA & Henry Andrew MEDINA CONDO

Année : 2025–2026

Table de Matières

1. Introduction et objectifs

1.1 Contexte du projet

Le protocole TFTP (Trivial File Transfer Protocol), défini par la RFC 1350, est un protocole de transfert de fichiers simplifié fonctionnant au-dessus d'UDP. Si les étapes précédentes nous ont permis de maîtriser les mécanismes de base (RRQ, WRQ, DATA, ACK), l'étape actuelle vise à transformer le serveur en un système robuste et concurrent.

1.2 Objectifs de l'étape 3

1. **Parallélisme et concurrence**: Implémenter deux modèles d'architecture radicalement différents pour gérer les clients:
    - **Le modèle multithread**: Allocation d'une ressource d'exécution dédiée (pthread) pour chaque transfert.
    - **Le modèle monothread** (Multiplexage): Utilisation de la primitive système select() pour surveiller plusieurs sessions de transfert au sein d'une boucle unique.
2. **Fiabilité sur UDP**: Assurer la résilience du protocole face à la perte de paquets grâce à des mécanismes de *timeout* et de *retransmission* côté serveur.
3. **Cohérence des données**: Mettre en place des mécanismes de verrouillage (Mutex et verrous logiques) pour empêcher les conflits de lecture/écriture lorsqu'un fichier est sollicité par plusieurs utilisateurs en même temps.

2. Détails de l'implémentation

2.1 Approche Multi Threadée

L'idée centrale est de transformer un serveur séquentiel en un serveur concurrent capable de traiter des dizaines de requêtes simultanément sans bloquer le port d'écoute principal (69). Pour chaque nouvelle requête (RRQ ou WRQ), le serveur délègue le travail à un **thread ouvrier** indépendant.

A. Architecture : de la version Séquentielle à la version multi-Client

Le passage au multithread ne s'est pas limité à l'ajout de pthread_create. Il a permis d'atteindre une conformité totale avec la RFC 1350, alors que la version précédente ne répondait que partiellement aux exigences de gestion des erreurs.

1. **Standardisation du Code** : Contrairement à l'ancienne version qui manipulait des offsets manuels dans des tampons bruts (*buffer[2] = ...*), nous utilisons désormais des structures packées dans *tftp_protocole.h.* L'usage de *__attribute__((packed))* garantit que la disposition binaire en mémoire correspond exactement au standard réseau, rendant le code plus lisible.

2. **Libération du Port 69** : Dans la version séquentielle, le port 69 était monopolisé pendant toute la durée d'un transfert, rendant le serveur indisponible. Le thread principal ne fait que "réceptionner" la demande.

3. **Le TID (Transfer Identifier)** : La RFC impose que le transfert de données s'effectue sur un port différent du port de réception. L'architecture respecte cela en effectuant un *bind* sur le port **0** dans chaque thread, générant un TID unique par client.

B. Mécanisme de Transfert Identifier (TID)

Chaque transfert est désormais une entité isolée, gérée par son propre thread ouvrier.

Contexte de Transfert: Toutes les informations nécessaires (socket TID, descripteur de fichier, adresse client) sont encapsulées dans une structure *tftp_context_t*. Cela évite toute collision de variables globales entre deux clients simultanés.

Robustesse Individuelle: Chaque thread gère ses propres retransmissions et ses propres timeouts via *setsockopt*. Une perte de paquet sur le Client A n'aura aucun impact sur la vitesse de transfert du Client B.

C. Gestion de la synchronisation des fichiers

Le développement de ce mécanisme repose sur trois impératifs techniques :

Atomicité de l'accès au registre : Avant d'accéder à un fichier, chaque thread doit consulter une liste globale des verrous actifs. Cette consultation est protégée par un mutex global pour éviter une *race condition* où deux threads créeraient simultanément deux verrous distincts pour un même fichier.

Intégrité des états (Compteurs) : L'état d'occupation d'un fichier (nombre de lecteurs actifs, présence d'un écrivain) est encapsulé dans une structure dédiée. Toute modification de ces compteurs est atomique pour prévenir toute corruption de l'état logique du serveur.

Évitement de l'interblocage (Deadlock) : Le protocole de verrouillage suit un ordre strict : verrouillage du registre global, puis verrouillage spécifique au fichier, puis relâchement immédiat du registre pour ne pas bloquer les requêtes visant d'autres fichiers.

```c
typedef struct VerrouFichier {
    char nom_fichier[256];

    pthread_mutex_t mutex;          /* Protège nb_lecteurs et en_ecriture */
    pthread_cond_t  cond;           /* Réveille les threads en attente    */
    int nb_lecteurs;                /* Nombre de lecteurs actifs          */
    int en_ecriture;                /* 1 si un écrivain est actif         */

    int ref_count;                  /* Nb de threads utilisant ce verrou  */
    struct VerrouFichier *suivant;
} VerrouFichier;
```

Pour ouvrir le fichier en lecture, on doit s’assure que personn

Situation (Client A / Client B)	Nature du Conflit	Risque Théorique	Comportement du Serveur
Lecture / Lecture (RRQ / RRQ)	Aucun	Aucun	Accès Parallèle : Le serveur autorise les deux threads à lire simultanément. Le débit global est optimisé.
Lecture / Écriture (RRQ / WRQ)	Lecteur vs Écrivain	Lecture de données incohérentes (Dirty Read).	Mise en attente : L'écriture est bloquée jusqu'à ce que tous les lecteurs aient libéré le fichier.
Écriture / Lecture (WRQ / RRQ)	Écrivain vs Lecteur	Le lecteur accède à un fichier en cours de modification.	Priorité à l'Exclusion : Le thread de lecture est suspendu par une variable de condition jusqu'à la fin de l'écriture.
Écriture / Écriture (WRQ / WRQ)	Écrivain vs Écrivain	Écrasement mutuel et corruption irréversible du fichier.	Exclusion Mutuelle : Un seul thread possède le verrou exclusif. Le second est mis en file d'attente.

2.2 Approche Mono Threadée

A. La Gestion des États de connexion

L'absence de threads dédiés impose la dématérialisation de la pile d'exécution. Pour suivre l'avancement de chaque transfert une structure de données 'ClientContext' est utilisée pour mémoriser l'état volatil de chaque session. Ce mécanisme permet au serveur de "reprendre" chaque transfert exactement là où il s'était arrêté lors de l'itération précédente. Les information stockés incluent:

- **L'état logique**: Identification de la phase actuelle via une énumération (STATE_RRQ, STATE_WRQ ou STATE_NONE).
- **Le contexte protocolaire**: Le descripteur de socket spécifique, le dernier numéro de bloc validé, ainsi que la pointeur du fichier ouvert ('FILE *p').
- **La persistance temporelle**: L'horodatage de la dernière activité ('last_activity') pour le calcul des timeouts.

```c
typedef struct {
    int sockfd;
    struct sockaddr_in client_addr;
    socklen_t addr_len;

    ClientState state;
    char filename[256];
    FILE *fp;

    uint16_t block_num;      // Next block to send (RRQ) or expected block (WRQ)
    char buffer[MAX_BUF];    // Data buffer
    int buffer_len;

    time_t last_activity;
    int retries;

    bool active;
} ClientContext;
```

B. Mécanisme de la primitive 'select()'

Le coeur algorithmique repose sur un boucle de surveillance active utilisant la primitive système 'select()'. Le processus se déroule en trois phases cycliques:

1. **Construction du set de descripteurs**: À chaque cycle, l'ensemble 'readfds' est dynamiquement reconstruit pour inclure la socket d'écoute (port 69) et l'ensemble des sockets client actifs.

2. **Surveillance synchrone**: L'appel système bloque le processus jusqu'à ce qu'un événement réseau survienne ou de que le délai de garde (1 seconde) expire. Cette temporisation courte est nécessaire pour permettre au serveur de sortir du blocage et d'effectuer la vérification des timeouts.

3. **Aiguillage des événements**: Si une activité est détectée sur le descripteur principal, la fonction 'handle_new_request()' initialise une nouvelle structure de contexte. Si l'activité concerne une socket client, 'handle_client_io' exécute le traitement du paquet spécifique (lecture ou écriture d'un bloc) avant de rendre immédiatement le contrôle à la boucle principale.

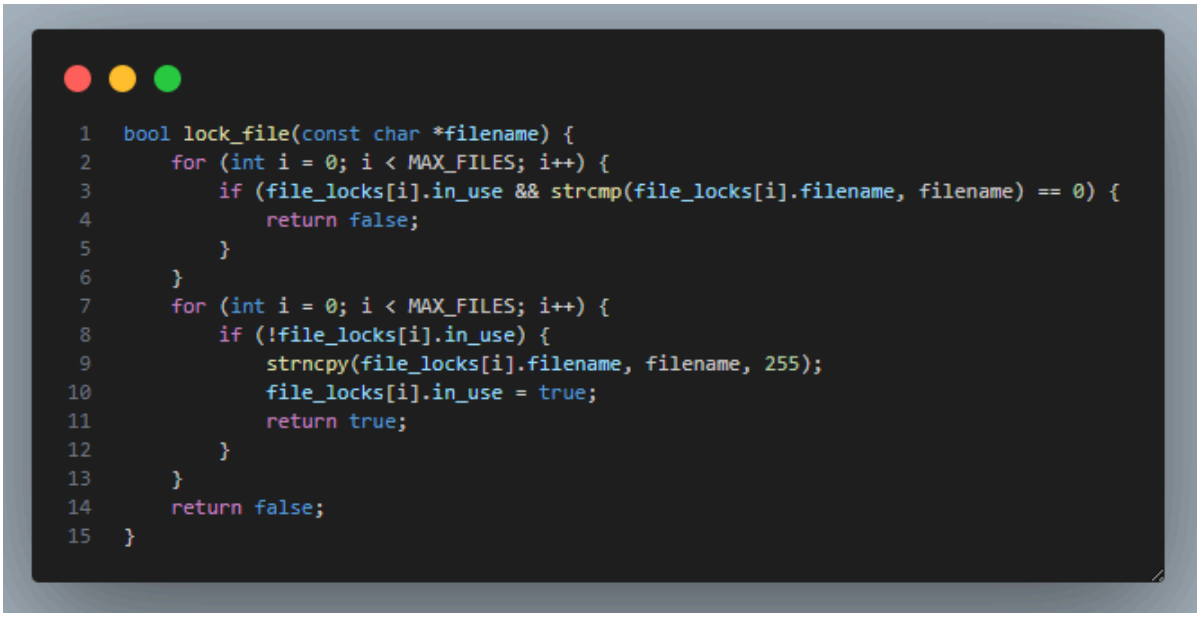
```c
while (1) {
    fd_set readfds;
    FD_ZERO(&readfds);
    FD_SET(server_fd, &readfds);
    int max_fd = server_fd;
    for (int i = 0; i < MAX_CLIENTS; i++) {
        if (clients[i].active) {
            FD_SET(clients[i].sockfd, &readfds);
            if (clients[i].sockfd > max_fd) max_fd = clients[i].sockfd;
        }
    }
    struct timeval tv = {1, 0};
    int activity = select(max_fd + 1, &readfds, NULL, NULL, &tv);

    if (activity < 0 && errno != EINTR) {
        perror("select");
        continue;
    }
    if (activity > 0) {
        if (FD_ISSET(server_fd, &readfds)) {
            handle_new_request(server_fd);
        }
        for (int i = 0; i < MAX_CLIENTS; i++) {
            if (clients[i].active && FD_ISSET(clients[i].sockfd, &readfds)) {
                handle_client_io(i);
            }
        }
```

C. Cohérence et synchronisation logicielle

Bien que le modèle mono threadé élimine par nature les conditions de concurrence liées à la mémoire vive, la gestion de l'intégrité du système de fichiers demeure impérative.

- **Verrouillage impérative**: Avant toute validation de requête, le serveur consulte une table de verrous logiques ('file_locks').

- **Exclusion Mutuelle:** Si un fichier fait déjà l'objet d'un transfert actif (notamment en écriture), le serveur rejette systématiquement toute nouvelle sollicitation.

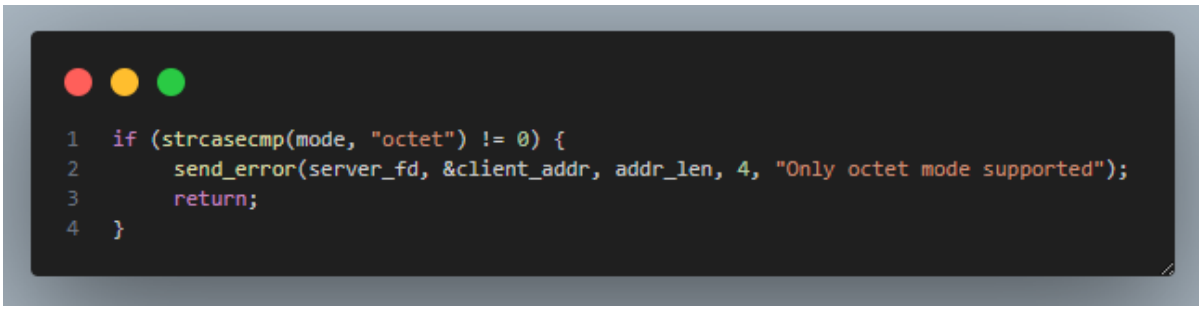


```
bool lock_file(const char *filename) {
    for (int i = 0; i < MAX_FILES; i++) {
        if (file_locks[i].in_use && strcmp(file_locks[i].filename, filename) == 0) {
            return false;
        }
    }
    for (int i = 0; i < MAX_FILES; i++) {
        if (!file_locks[i].in_use) {
            strncpy(file_locks[i].filename, filename, 255);
            file_locks[i].in_use = true;
            return true;
        }
    }
    return false;
}
```

2.3 Analyse de l'interopérabilité et Mode de Transfert

Lors des tests avec des clients TFTP standards, nous avons choisi d'implémenter une validation stricte du mode de transfert.

- **Contrainte technique**: Le serveur vérifie explicitement la présence du mode 'octet' via la fonction 'strcasecmp'.
- **Observation**: Si un client teste d'utiliser le mode 'mail' ou 'netascii', ou s'il envoi des options de négociation (RFC 2347) que le serveur ne reconnaît pas encore, celui-ci renvoie un erreur de type 4 (Illegal TFTP operation).
- **Justification**: Cette approche garantit l'intégrité des fichiers binaires transférés, évitant toute conversion de fin de ligne non souhaitée qui pourrait corrompre les fichiers dans le répertoire './tftp'.



```
if (strcasecmp(mode, "octet") != 0) {
    send_error(server_fd, &client_addr, addr_len, 4, "Only octet mode supported");
    return;
}
```

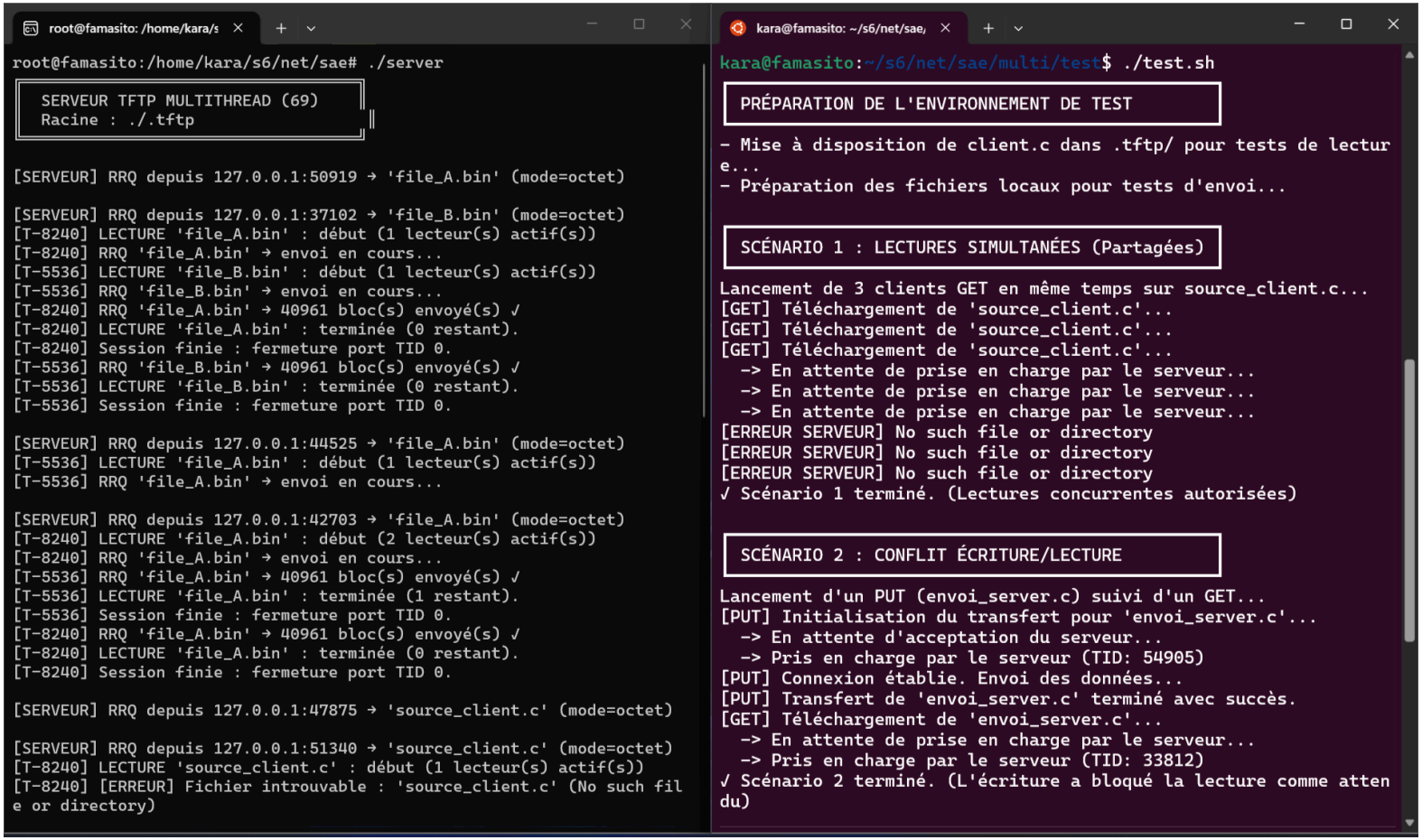
3. Résultats et Preuves Expérimentales

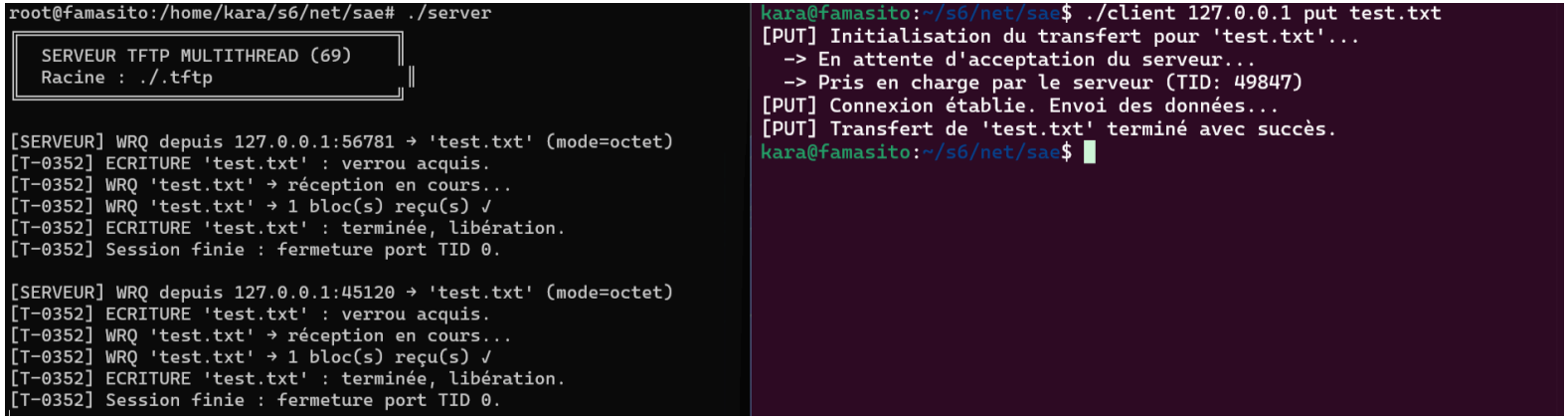
3.1 Resultat : Multi-Client

Pour garantir la fiabilité du serveur face à des accès concurrents et valider l'implémentation du modèle Lecteurs-Écrivains, une suite de tests automatisés a été développée. Cette phase permet de passer d'une vérification unitaire à une validation de stress-test en conditions réelles.

Nous avons pu simuler le multi-Client avec des scripts afin de mettre à l'épreuve notre serveur :

- **Script Shell (*`test.sh`*)** : Ce script orchestre des scénarios de haut niveau en utilisant les fonctionnalités du shell pour lancer des processus clients en arrière-plan. Il permet de vérifier rapidement le comportement du serveur face à des mélanges de requêtes GET et PUT.
- **Simulateur C (*`test_multithreading.c`*)** : Ce programme utilise la bibliothèque `pthread` pour générer des threads clients. Chaque thread exécute une instance du binaire `./client` via la fonction `system()`, permettant un contrôle temporel très précis (nanosleep) sur l'arrivée des requêtes. Ainsi, nous avons testé les situations qui exposent le serveur à la concurence.


root@famasito: /home/kara/s
root@famasito:/home/kara/s6/net/sae# ./server
SERVEUR TFTP MULTITHREAD (69)
Racine : ./.tftp
[SERVEUR] RRQ depuis 127.0.0.1:50919 → 'file_A.bin' (mode=octet)
[SERVEUR] RRQ depuis 127.0.0.1:37102 → 'file_B.bin' (mode=octet)
[T-8240] LECTURE 'file_A.bin' : début (1 lecteur(s) actif(s))
[T-8240] RRQ 'file_A.bin' → envoi en cours...
[T-5536] LECTURE 'file_B.bin' : début (1 lecteur(s) actif(s))
[T-5536] RRQ 'file_B.bin' → envoi en cours...
[T-8240] RRQ 'file_A.bin' → 40961 bloc(s) envoyé(s) ✓
[T-8240] LECTURE 'file_A.bin' : terminée (0 restant).
[T-8240] Session finie : fermeture port TID 0.
[T-5536] RRQ 'file_B.bin' → 40961 bloc(s) envoyé(s) ✓
[T-5536] LECTURE 'file_B.bin' : terminée (0 restant).
[T-5536] Session finie : fermeture port TID 0.
[SERVEUR] RRQ depuis 127.0.0.1:44525 → 'file_A.bin' (mode=octet)
[T-5536] LECTURE 'file_A.bin' : début (1 lecteur(s) actif(s))
[T-5536] RRQ 'file_A.bin' → envoi en cours...
[SERVEUR] RRQ depuis 127.0.0.1:42703 → 'file_A.bin' (mode=octet)
[T-8240] LECTURE 'file_A.bin' : début (2 lecteur(s) actif(s))
[T-8240] RRQ 'file_A.bin' → envoi en cours...
[T-5536] RRQ 'file_A.bin' → 40961 bloc(s) envoyé(s) ✓
[T-5536] LECTURE 'file_A.bin' : terminée (1 restant).
[T-5536] Session finie : fermeture port TID 0.
[T-8240] RRQ 'file_A.bin' → 40961 bloc(s) envoyé(s) ✓
[T-8240] LECTURE 'file_A.bin' : terminée (0 restant).
[T-8240] Session finie : fermeture port TID 0.
[SERVEUR] RRQ depuis 127.0.0.1:47875 → 'source_client.c' (mode=octet)
[SERVEUR] RRQ depuis 127.0.0.1:51340 → 'source_client.c' (mode=octet)
[T-8240] LECTURE 'source_client.c' : début (1 lecteur(s) actif(s))
[T-8240] [ERREUR] Fichier introuvable : 'source_client.c' (No such file or directory)
kara@famasito: ~/s6/net/sae/
kara@famasito:~/s6/net/sae/multi/test$ ./test.sh
PRÉPARATION DE L'ENVIRONNEMENT DE TEST
- Mise à disposition de client.c dans .tftp/ pour tests de lecture...
- Préparation des fichiers locaux pour tests d'envoi...
SCÉNARIO 1 : LECTURES SIMULTANÉES (Partagées)
Lancement de 3 clients GET en même temps sur source_client.c...
[GET] Téléchargement de 'source_client.c'...
[GET] Téléchargement de 'source_client.c'...
[GET] Téléchargement de 'source_client.c'...
-> En attente de prise en charge par le serveur...
-> En attente de prise en charge par le serveur...
-> En attente de prise en charge par le serveur...
[ERREUR SERVEUR] No such file or directory
[ERREUR SERVEUR] No such file or directory
[ERREUR SERVEUR] No such file or directory
✓ Scénario 1 terminé. (Lectures concurrentes autorisées)
SCÉNARIO 2 : CONFLIT ÉCRITURE/LECTURE
Lancement d'un PUT (envoi_server.c) suivi d'un GET...
[PUT] Initialisation du transfert pour 'envoi_server.c'...
-> En attente d'acceptation du serveur...
-> Pris en charge par le serveur (TID: 54905)
[PUT] Connexion établie. Envoi des données...
[PUT] Transfert de 'envoi_server.c' terminé avec succès.
[GET] Téléchargement de 'envoi_server.c'...
-> En attente de prise en charge par le serveur...
-> Pris en charge par le serveur (TID: 33812)
✓ Scénario 2 terminé. (L'écriture a bloqué la lecture comme attendu)


root@famasito:/home/kara/s6/net/sae# ./server
SERVEUR TFTP MULTITHREAD (69)
Racine : ./.tftp
[SERVEUR] WRQ depuis 127.0.0.1:56781 → 'test.txt' (mode=octet)
[T-0352] ECRITURE 'test.txt' : verrou acquis.
[T-0352] WRQ 'test.txt' → réception en cours...
[T-0352] WRQ 'test.txt' → 1 bloc(s) reçu(s) ✓
[T-0352] ECRITURE 'test.txt' : terminée, libération.
[T-0352] Session finie : fermeture port TID 0.
[SERVEUR] WRQ depuis 127.0.0.1:45120 → 'test.txt' (mode=octet)
[T-0352] ECRITURE 'test.txt' : verrou acquis.
[T-0352] WRQ 'test.txt' → réception en cours...
[T-0352] WRQ 'test.txt' → 1 bloc(s) reçu(s) ✓
[T-0352] ECRITURE 'test.txt' : terminée, libération.
[T-0352] Session finie : fermeture port TID 0.
kara@famasito:~/s6/net/sae$ ./client 127.0.0.1 put test.txt
[PUT] Initialisation du transfert pour 'test.txt'...
-> En attente d'acceptation du serveur...
-> Pris en charge par le serveur (TID: 49847)
[PUT] Connexion établie. Envoi des données...
[PUT] Transfert de 'test.txt' terminé avec succès.
kara@famasito:~/s6/net/sae$

3.2. Resultat : Mono-thread

La validation de l'implémentation basée sur la primitive *select()* a été réalisée à l'aide d'un script d'automatisation Shell. Ce protocole de test a pour objectif de démontrer que l'unique fil d'exécution du serveur est capable de gérer la concurrence et de maintenir l'intégrité du système de fichiers sans les mécanismes classiques de multi-threading.

A. Architecture du Protocole de Test (*test_select.sh*)

Le script de test a été conçu comme un environnement de contrôle rigoureux, simulant des conditions d'accès réseau simultanées. Son fonctionnement se décompose en quatre phases stratégiques :

1. **Vérification du Multiplexage**: En lançant deux processus clients en arrière-plan (&) pour télécharger des fichiers différents (*archivo_A.txt* et *archivo_B.txt*), le script vérifie la capacité du serveur à alterner entre les descripteurs de fichiers dans la boucle select. Le succès de cette étape confirme que le traitement d'un client ne bloque pas l'entrée en service d'un autre.

2. **Épreuve d'Exclusion Mutuelle (File Locking) :** Le script tente de forcer une condition de course (*race condition*) en demandant l'accès à un fichier déjà en cours de transfert. Il capture ensuite la sortie d'erreur pour valider la réception du message "File busy".

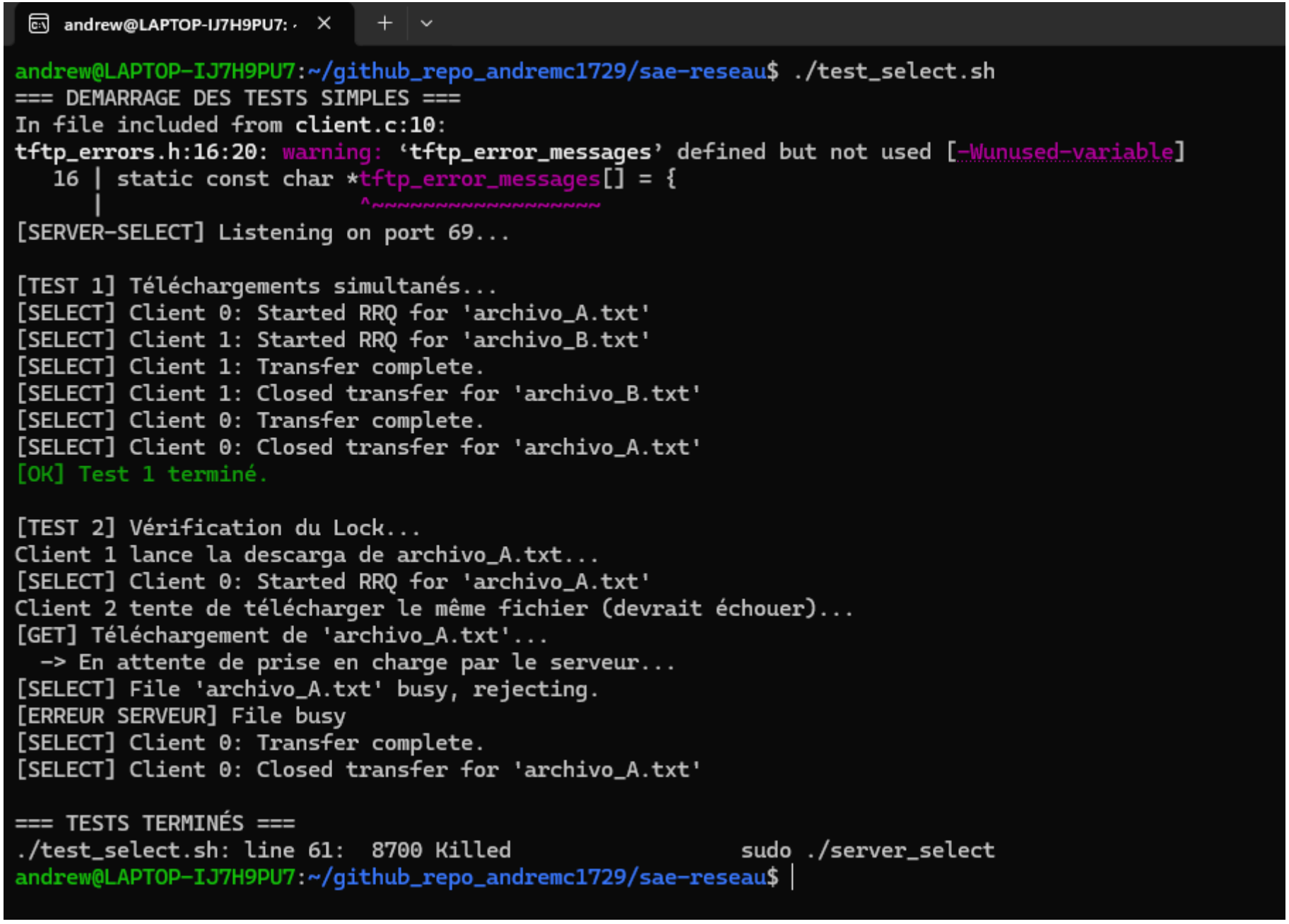
B. Analyse du comportement du serveur (*server_select.c*)

L'exécution du script a permis de confirmer la validité des choix d'implémentation suivants :

- **Efficacité de la structure**: Le script démontre que le tableau de contextes permet de conserver l'état de chaque transfert (numéro de bloc, pointeur de fichier, temporisation) de manière étanche.

- **Mécanisme de Verrouillage Logique**: Contrairement à une approche par threads où les clients pourraient être mis en attente, le serveur mono-thread utilise une table file_locks. Le test Shell confirme que si un slot in_use est détecté pour un nom de fichier identique, le serveur émet immédiatement un paquet d'erreur (Code 0), préservant ainsi la linéarité des opérations d'écriture et de lecture.

- **Gestion des Timeouts et Retransmissions**: Grâce à la temporisation d'une seconde de l'appel select(..., &tv), le serveur sort du blocage pour exécuter check_timeouts(). Le script a validé que les clients inactifs sont proprement déconnectés et que les ressources (slots et sockets) sont libérées après MAX_RETRIES.


andrew@LAPTOP-IJ7H9PU7:~/github_repo_andremc1729/sae-reseau$ ./test_select.sh
=== DEMARRAGE DES TESTS SIMPLES ===
In file included from client.c:10:
tftp_errors.h:16:20: warning: ‘tftp_error_messages’ defined but not used [-Wunused-variable]
16 | static const char *tftp_error_messages[] = {
| ^~~~~~~~~~~~~~~~~~~~
[SERVER-SELECT] Listening on port 69...
[TEST 1] Téléchargements simultanés...
[SELECT] Client 0: Started RRQ for 'archivo_A.txt'
[SELECT] Client 1: Started RRQ for 'archivo_B.txt'
[SELECT] Client 1: Transfer complete.
[SELECT] Client 1: Closed transfer for 'archivo_B.txt'
[SELECT] Client 0: Transfer complete.
[SELECT] Client 0: Closed transfer for 'archivo_A.txt'
[OK] Test 1 terminé.
[TEST 2] Vérification du Lock...
Client 1 lance la descarga de archivo_A.txt...
[SELECT] Client 0: Started RRQ for 'archivo_A.txt'
Client 2 tente de télécharger le même fichier (devrait échouer)...
[GET] Téléchargement de 'archivo_A.txt'...
-> En attente de prise en charge par le serveur...
[SELECT] File 'archivo_A.txt' busy, rejecting.
[ERREUR SERVEUR] File busy
[SELECT] Client 0: Transfer complete.
[SELECT] Client 0: Closed transfer for 'archivo_A.txt'
=== TESTS TERMINÉS ===
./test_select.sh: line 61: 8700 Killed sudo ./server_select
andrew@LAPTOP-IJ7H9PU7:~/github_repo_andremc1729/sae-reseau$

Conclusion

- La migration d'une conception séquentielle vers une conception multithreadée à l'aide de la bibliothèque pthread a permis le découplage efficace du port d'écoute principal 69, en déléguant chaque flux de données à des threads indépendants qui fonctionnent avec leurs propres identifiants de transfert (TID), assurant ainsi une disponibilité totale du serveur face à de multiples requêtes simultanées.

- La sécurité des données a été assurée grâce à un mécanisme de synchronisation basé sur des mutex et des variables de condition qui met en œuvre une politique d'exclusion mutuelle sélective, permettant un accès parallèle pour plusieurs lectures, mais suspendant de manière déterministe toute opération conflictuelle lorsqu'un thread demande une autorisation d'écriture sur la même ressource.

- L'utilisation de structures encapsulées telles que tftp_context_t et la configuration de sockets non bloquantes à l'aide de setsockopt garantissent que chaque transfert est une entité isolée, permettant ainsi que la gestion des erreurs, des retransmissions et des délais d'attente d'un client spécifique n'interfère pas avec les performances ou la stabilité des autres sessions actives.

- L'implémentation du serveur via select() démontre qu'il est possible de gérer la concurrence de manière extrêmement efficace avec un seul fil d'exécution. Cette approche, bien que plus exigeante en termes de rigueur algorithmique (gestion manuelle des états et des verrous), offre une empreinte mémoire minimale et une stabilité remarquable.

Annexes

Guide d'exécution et de test

A. Exécution du Serveur Multi-thread

a. Lancement du serveur (avant *sudo chmod 777 .tftp*)

./server

b. Utilisation standard avec le client (exemple avec get)

./client 127.0.0.1 get archivo_prueba.txt 69

B. Tests de Concurrence (Serveur Multi-thread)

a. Assurez-vous que le serveur est en cours d'exécution (./server).

b. Dans un terminal séparé, lancez le script Shell :

chmod +x test_multi.sh

./test_multi.sh

C. Exécution du Serveur Mono-thread (Multiplexage select)

a. Lancement du serveur

./server_select

b. Utilisation standard avec le client (exemple avec get)

./client 127.0.0.1 get archivo_prueba.txt 69

D. Validation automatisée (Protocole de test complet) du Serveur Mono-thread

a. Lancement du serveur

./server_select

b. Validation automatisée (Protocole de test complet)

chmod +x test_select.sh

./test_select.sh