RAPPORT TECHNIQUE RENDU 3

UNIVERSITE DE PICARDIE JULES VERNE

UFR DES SCIENCES

LICENCE 3 INFORMATIQUE – PARCOURS INFORMATIQUE

SAE Concevoir protocole de communication entre un serveur et clients

Réalisé par :

Mohamed Yassine ALOUI

Benassila Erwan

1. Introduction

Ce rapport technique documente le processus de développement de protocole Trivial File Transfer Protocol (TFTP) en C, en se concentrant sur le contrôle de flux et la gestion des connexions réseau. TFTP est un protocole léger utilisé pour transférer des fichiers sur un réseau.

Il est élaboré par Benassila Erwan et Mohamed Yassine Aloui dans le cadre du SAE4, sous la supervision de monsieur UTARD GIL pour l'année universitaire 2023-2024.

2. Présentation du Projet

Le projet consiste en l'implémentation d'une application de transfert de fichiers utilisant le protocole TFTP (Trivial File Transfer Protocol) basé sur UDP (User Datagram Protocol). Le but principal est de créer un système de transfert de fichiers simple et efficace permettant à un client de demander la lecture ou l'écriture de fichiers depuis ou vers un serveur.

Davantage de détails sur le projet sont disponibles dans le cahier des charges, lequel est inclus en annexe pour référence.

3. Fonctionnalités et Implémentations

3.1. Fonctionnalités du Client TFTP

Le client TFTP présente les fonctionnalités suivantes :

- Lecture de Fichier (RRQ) : Permet au client de demander la lecture d'un fichier depuis le serveur.
- Écriture de Fichier (WRQ) : Permet au client d'envoyer un fichier vers le serveur.
- Gestion des Erreurs : Gère les erreurs de transmission et d'accès aux fichiers en envoyant des paquets d'erreur appropriés.

3.2. Fonctionnalités du Serveur TFTP

Le serveur présente les fonctionnalités suivantes :

- Gestion des Requêtes (RRQ et WRQ): Traite les requêtes de lecture et d'écriture de fichiers en répondant avec les données du fichier demandé ou en attendant les données à écrire.

- ❖ Gestion des Erreurs : Gère les erreurs de transmission et d'accès aux fichiers en envoyant des paquets d'erreur appropriés.

Analyse et Explication technique :

D'après le RFC135 cinq types de paquet qui existe chaque paquet a un rôle spécifique et des nombres de bit spécifique :

- ❖ Paquet RRQ : opcode =1 :
  - Il contient [opcode(2bit), nomfichier,0(fin de nom fichier), Mode de transfére,0]
  - Ce paquet sert à demander au serveur d'envoyer le fichier x.
  - Les 0 sont des \O (pour indiquer la fin de chaine de caractère)
- ❖ Paquet WRQ : opcode =2 :
  - Il contient [opcode(2bit), nomfichier,0(fin de nom fichier), Mode de transfére,0]
  - Ce paquet sert à demander au serveur de recevoir le fichier x.
- ❖ Paquet DATA : opcode=3
  - La structure de paquet de data est [opcode (2bit) |numéro de block(2bit) |data (n bit)]
  - Le numéro de block est très important, il sert à identifier le paquet et à la gestion des erreurs.
  - Le paquet DATA va contenir les données de fichier et peut être envoyer plusieurs fois soit de serveur vers client (cas de get) ou l'inverse (cas de put)
- ❖ Paquet ACK
  - Ce paquet sert à valider la réception de paquet WRQ ou bien à chaque envoie de paquet DATA l'émetteur doit attendre ce paquet pour envoyer la prochaine DATA
  - Opcode=4
  - [Opcode (2bits) |numéro de block]
- ❖ Packet ERROR:
  - [Opcode(2bit) |code d'erreur (2bit) |message d'erreur|0(fin string)]
  - Opcode=5
  - Ce paquet permet d'envoyer un message d'erreur avec un code spécifique pour chaque type d'erreur

En se basant sur ces paquets on a implémenté notre code suivant cette logique :

1)Échange entre le Serveur et le Client (RRQ - Read Request)

Lorsqu'un client envoie une requête de lecture (RRQ - Read Request) à un serveur TFTP, l'échange de paquets se déroule comme suit :

- Client (Initiateur) -> Serveur (Récepteur) :
  - Le client envoie une requête RRQ au serveur, spécifiant le nom du fichier à lire et le mode de transfert.
- Serveur -> Client :
  - Le serveur répond en envoyant le premier paquet de données (DATA) contenant une partie du fichier demandé au client.
- Client -> Serveur :
  - Le client envoie un paquet d'accusé de réception (ACK) au serveur pour confirmer la réception du premier paquet de données.
- Répétition des étapes 2 et 3 :
  - Le serveur envoie les paquets de données restants contenant le reste du fichier en séquence.
  - Le client envoie un paquet d'accusé de réception (ACK) à chaque fois qu'il reçoit un paquet de données.
- **Fin de la transmission :**
  - Une fois que tous les paquets de données ont été envoyés par le serveur et confirmés par le client, le transfert est considéré comme terminé.

2)Échange entre le Serveur et le Client (WRQ - Write Request)

  - Lorsqu'un client envoie une requête d'écriture (WRQ - Write Request) à un serveur TFTP, l'échange de paquets se déroule comme suit :
- **Client (Initiateur) -> Serveur (Récepteur) :**
  - Le client envoie une requête WRQ au serveur, spécifiant le nom du fichier à écrire et le mode de transfert.
- Serveur -> Client :

- Le serveur répond en envoyant un paquet d'accusé de réception (ACK) au client pour confirmer la réception de la requête.

❖ Client -> Serveur :
  - Le client envoie le premier paquet de données (DATA) contenant une partie du fichier à écrire sur le serveur.

❖ Serveur -> Client :
  - Le serveur envoie un paquet d'accusé de réception (ACK) au client pour confirmer la réception du premier paquet de données.

❖ Répétition des étapes 3 et 4 :
  - Le client envoie les paquets de données restants contenant le reste du fichier en séquence.
  - Le serveur envoie un paquet d'accusé de réception (ACK) à chaque fois qu'il reçoit un paquet de données.

❖ Fin de la transmission :
  - Une fois que tous les paquets de données ont été envoyés par le client et confirmés par le serveur, le transfert est considéré comme terminé.

Pour implémenter la connexion entre le serveur et le client, on a utilisé des sockets UDP, qui sont des sockets de datagramme offrant une communication bidirectionnelle et non connectée.

Dans mon code, j'ai d'abord créé un socket côté serveur en utilisant la fonction socket() avec les paramètres AF_INET pour la famille d'adresses IPv4 et SOCK_DGRAM pour le type de socket datagramme.

Ensuite, j'ai lié ce socket à une adresse et un port en utilisant la fonction bind(), spécifiant l'adresse IP du serveur et le numéro de port sur lequel le serveur écoute les demandes de connexion.

Pour le rôle de la structure struct sockaddr_in server_address, elle représente l'adresse du serveur dans le code. Cette structure contient des informations sur l'adresse IP et le port du serveur.

elle est utilisée pour spécifier l'adresse IP du serveur ainsi que le port sur lequel le serveur attend les demandes de connexion.

Elle est utilisée lors de la liaison du socket à une adresse avec la fonction bind() et lors de l'envoi de paquet dans le code(recv_from et send_to).

Les fichier sources

Le dossier src contient Deux Fichiers client.c et serveur.c

Le code source de client.c contient :

main(): La fonction principale du programme. Elle crée une socket UDP, initialise l'adresse du serveur, puis appelle les fonctions get() ou put() en fonction de la commande passée en argument (get pour récupérer un fichier du serveur, put pour envoyer un fichier au serveur).

get(): Cette fonction est responsable de l'envoi d'une requête de lecture (RRQ) au serveur pour récupérer un fichier. Elle appelle send_RRQ() pour envoyer la requête, puis attend et traite les paquets DATA envoyés par le serveur.

put(): Cette fonction est responsable de l'envoi d'une requête d'écriture (WRQ) au serveur pour envoyer un fichier. Elle appelle send_WRQ() pour envoyer la requête, puis envoie les paquets DATA au serveur en réponse aux ACK reçus en appellant la fonction **transferer_DATA()**.

send_RRQ(): Envoie une requête de lecture au serveur avec le nom du fichier demandé.

send_WRQ(): Envoie une requête d'écriture au serveur avec le nom du fichier à écrire.

send_ACK(): Envoie un paquet d'acquittement (ACK) au serveur avec le numéro de bloc correspondant.

transferer_DATA(): Transfère les données du fichier entre le client et le serveur. Pour les requêtes de lecture, cette fonction envoie des paquets DATA au serveur en réponse aux ACK reçus. Pour les requêtes d'écriture, elle reçoit les paquets DATA du client et les écrit dans un fichier.

Le code source de serveur.c contient :

main :

Crée un socket UDP pour le serveur.

Lie le socket à une adresse IP et un port spécifié.

L'ip et le port sont entrée par l’utilisateur lors d’exécution de programme

Appelle la fonction ecouter_packet pour écouter les paquets provenant des clients.

ecouter_packet :

Fonction principale qui écoute les paquets entrants et appelle les fonctions appropriées en fonction de l'opcode du paquet.

Décide si le paquet entrant est une requête de lecture (RRQ) ou d'écriture (WRQ) et appelle les fonctions transferer_DATA ou recevoir_DATA respectivement.

Gère les erreurs d'opcode en envoyant un paquet d'erreur approprié avec la fonction send_ERROR.

transferer_DATA :

Fonction qui gère le transfert de données vers un client.

Ouvre le fichier demandé en mode lecture.

Divise le fichier en paquets de données (DATA) et les envoie au client.

Attend la réception d'un paquet d'acquittement (ACK) du client avant d'envoyer le prochain paquet.

recevoir_DATA :

Fonction qui gère la réception de données d'un client.

Reçoit les paquets de données (DATA) envoyés par le client et les écrit dans un fichier local.

Envoie un paquet d'acquittement (ACK) au client pour chaque paquet de données reçu.

send_ERROR :

Fonction utilitaire pour envoyer un paquet d'erreur (ERROR) au client ou l’inverse en cas de problème.

Utilisée par la fonction ecouter_packet pour gérer les erreurs d'opcode et envoyer des messages d'erreur appropriés.

Communication Client-Serveur

La communication entre le client et le serveur s'effectue en utilisant des sockets UDP. Le client envoie une requête (RRQ ou WRQ) au serveur, qui répond avec des paquets DATA ou ACK en fonction de la requête. Le client et le serveur échangent des paquets jusqu'à ce que le transfert de fichier soit complet.

Le dossier doc contient la documentation de projet.

4. Difficultés Rencontrées

4.1. Boucle infinie dans le serveur et dans le client lors d'exécution

On a rencontré des problèmes de boucle infinie dans notre code.

Débogage prolongé : On a passé 10 jours à déboguer notre code pour trouver des solutions à ces problèmes.

On a investi beaucoup de temps et d'efforts dans la résolution de ces problèmes, ce qui est une partie essentielle du processus de développement logiciel. En fin de compte, l'expérience acquise grâce à la résolution de ces problèmes nous aidera à améliorer nos compétences en programmation et à éviter des erreurs similaires à l'avenir.

4.2. Gestion des Timeout

Une autre difficulté a été la gestion des timeouts lors de l'attente de réponses aux requêtes TFTP. Des mécanismes de temporisation ont dû être mis en place pour éviter les blocages du programme en cas de non-réponse du serveur.

5. Conclusion

En conclusion, l'implémentation du TFTP avec UDP a été un défi technique qui a permis d'approfondir la compréhension des paquets et de types de paquet et surtout les socket UDP.

Étape 3 : Serveur TFTP Multi-clients

Version monothreadé, en utilisant select () pour gérer les requêtes de plusieurs clients en parallèle : réalisée par Mohammed yassine aloui

Dans le cadre de notre implémentation du serveur TFTP, nous avons utilisé la fonction système select pour surveiller l'activité sur plusieurs descripteurs de fichiers en même temps. La fonction select nous permet de détecter quels descripteurs de fichiers ont une activité d'entrée/sortie prête à être lue ou écrite sans bloquer le processus.

Voici comment nous avons utilisé select dans notre code :

1)Initialisation des ensembles de descripteurs de fichiers :

Avant d'appeler select, nous initialisons deux ensembles de descripteurs de fichiers : master et read_fds. L'ensemble master contient tous les descripteurs de fichiers que nous voulons surveiller, tandis que read_fds est un ensemble temporaire que nous utilisons pour stocker les descripteurs de fichiers prêts à être lus.

2)Boucle de surveillance avec select :

Nous utilisons une boucle infinie while(1) pour maintenir notre serveur en attente de nouvelles connexions. À chaque itération de la boucle, nous appelons select pour surveiller les descripteurs de fichiers en lecture et spécifions un timeout de 5 secondes.

Pendant l'attente, select surveille tous les descripteurs de fichiers contenus dans l'ensemble master et attend qu'au moins un descripteur soit prêt à être lu. Une fois qu'une activité est détectée ou que le timeout spécifié est écoulé, select se termine et retourne.

Traitement des événements :

Nous utilisons la macro FD_ISSET pour vérifier si le descripteur de fichier du serveur est prêt à être lu. Si tel est le cas, cela signifie qu'il y a une nouvelle demande de connexion entrante du coup on appelle la fonction écouter paquet (expliqué ci-dessus)

Version multi-threadé, en utilisant les threads pour gérer les requêtes de plusieurs clients en parallèle : réalisée par Erwan Benassila

Dans le cadre de notre implémentation du serveur TFTP, nous avons utilisé le multithreading pour pourvoir traiter les requêtes client en parallèle de la boucle principal et en traiter plusieurs en même temps.

Voici comment est géré le multi-threadind dans notre code :

1)Création de thread client :

Lorsqu’une requête client est reçu, on crée une structure ThreadArgs avec les données reçues de la requête client ainsi qu’un socket client nouvellement

créer et on crée un thread client en utilisant pthread_create qui récupérer la structure et appelant la fonction client_thread.

2)Traitement de la requête dans le thread :

La fonction client_thread appelé, elle récupère les données de la structure passé en paramètre, utilise l'opcode contenu dans ces données pour appeler transferer_DATA ou recevoir_DATA avec en paramètre les autres données récupérées de la structure et donc on exécute une de ces fonctions.

3)Libérer les ressources :

Maintenant que la requête a été traité, on libère les ressources qui ne sont plus nécessaire au programme donc le socket client ainsi que les données de la structure de données utilisé dans le thread et puis on met fin au thread en utilisant pthread_exit.

Étape 4 : Extension du Protocole TFTP

Afin d'implémenter **l'option bigfile** qui Permettre de traiter des fichiers sans limitation de taille :

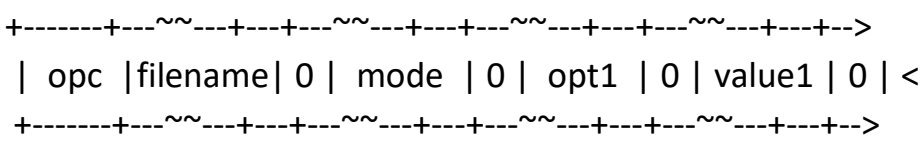
On doit suivre l'RFC 2347 et implémenter la Gestion des Options et Négociation des Paquets RRQ/WRQ et OACK

Cette négociation va permettre au serveur et client de s'accorder sur des options importantes telles que la gestion des transferts des fichiers de grande taille, un aspect important dans le contexte actuel où les fichiers à transférer peuvent dépasser la capacité standard du protocole TFTP.

Négociation des Options

La négociation des options débute dès que le client envoie une requête de lecture (RRQ) ou d'écriture (WRQ) au serveur, incluant potentiellement des options supplémentaires comme "bigfile".

Donc la structure de paquet RRQ/WRQ devient comme la suite :



```
  >-------+---+---~~---+---+
< optN  | 0 | valueN | 0 |
  >-------+---+---~~---+---+
```

Ainsi, nous allons ajouter des champs "option" et "valeur" dans nos paquets. Dans notre cas, l'option1 sera "bigfile" avec une valeur de "1", signifiant que le client demande un RRQ/WRQ pour un fichier de très grande taille.

À la réception de cette requête, le serveur examine les options proposées et répond avec un paquet OACK (Option Acknowledgment) pour confirmer les options acceptées. Si l'option demandée n'est pas disponible, le serveur envoie un paquet d'erreur avec un nouveau code d'erreur, le code 8, pour informer le client de l'indisponibilité de cette option.

Structure de nouveau paquet OACK :

```
+-------+---~~---+---+---~~---+---+---~~---+---+---~~---+---+
|  opc  |  opt1  | 0 | value1 | 0 |  optN  | 0 | valueN | 0 |
 +-------+---~~---+---+---~~---+---+---~~---+---+---~~---+---+
```

L'opcode du paquet OACK est 6.

Si le client reçoit le paquet OACK :

- Dans le cas d'une opération de mise en ligne (put), il commence par envoyer les paquets de DATA.
- Dans le cas d'une opération de téléchargement (get), il envoie un paquet ACK au serveur. Ce dernier, lorsqu'il reçoit le paquet ACK, commence à envoyer les paquets de DATA.

Si le client reçoit dans l'OACK une option non demandée initialement, il envoie un paquet d'erreur avec le code 8.

Explication plus détaillée sur la logique de négociation implémenté :

- ❖ Cas de "put"
  1. Le client envoie une requête WRQ au serveur, incluant l'option "bigfile" avec la valeur "1" pour indiquer un fichier de grande taille.

2. Le serveur analyse la requête et les options proposées. S'il accepte l'option "bigfile":
3. Envoie un paquet OACK au client pour confirmer l'acceptation de l'option.
4. À la réception du paquet OACK, le client commence à envoyer les paquets de DATA séquencés pour transférer le fichier.
5. Pour chaque paquet de DATA reçu, le serveur envoie un paquet ACK correspondant pour confirmer la réception.

Si le serveur n'accepte pas l'option "bigfile" ou une autre erreur survient :

- Le serveur envoie un paquet d'erreur avec le code 8 au client pour signaler l'indisponibilité de l'option ou l'erreur rencontrée.

Cas de "get" (Téléchargement d'un fichier)

1. Le client envoie une requête RRQ au serveur, incluant l'option "bigfile" avec la valeur "1".
2. Le serveur vérifie les options proposées et, s'il accepte l'option "bigfile":
3. Envoie un paquet OACK au client pour confirmer.
4. Après avoir reçu le paquet OACK, le client envoie un paquet ACK avec le numéro de bloc 0 pour signaler qu'il est prêt à recevoir le fichier.
5. Le serveur commence alors à envoyer les paquets de DATA séquencés, contenant les segments du fichier.
6. Le client envoie un paquet ACK pour chaque paquet de DATA reçu pour confirmer la réception et demander le segment suivant jusqu'à la fin du fichier.
- Si à n'importe quel point une option non demandée est reçue dans l'OACK ou une erreur est détectée, le client ou le serveur peut envoyer un paquet d'erreur avec le code 8 pour signaler le problème et interrompre la transaction.

Gestion des Fichiers de Grande Taille

Le traitement des fichiers de grande taille est facilité par l'option "bigfile". Lorsqu'un fichier à transférer dépasse la taille standard que TFTP peut normalement gérer, le client et le serveur utilisent cette option pour signaler leur capacité à gérer de tels fichiers. Cela permet de fragmenter le fichier en multiples paquets DATA, chacun étant suivi d'un paquet ACK, jusqu'à la fin du transfert.

Numérotation des Blocs : Chaque bloc de données est assigné un numéro de bloc unique, débutant à 1 et s'incrémentant pour chaque bloc successif. Cette numérotation joue un rôle clé dans l'ordre de transmission et la vérification de l'intégrité des données reçues.

Facilitation de la Retransmission : En cas d'erreur de transmission ou de perte de paquets, le numéro de bloc permet au serveur et au client de s'identifier et de retransmettre spécifiquement les blocs manquants ou erronés, sans nécessiter la retransmission de l'ensemble du fichier.

Transmission et Accusés de Réception (ACK)

La transmission des blocs de données suit un protocole strict où chaque bloc envoyé par l'émetteur (serveur ou client) est acquitté par un paquet ACK du recepteur, indiquant la bonne réception :

Séquence de Transmission : Les blocs de données sont envoyés séquentiellement. L'émetteur attend un accusé de réception pour chaque bloc avant de procéder à l'envoi du bloc suivant.

Intégrité du Transfert : Les ACK servent de confirmation que chaque bloc a été correctement reçu et enregistré par le client, garantissant ainsi l'intégrité du fichier transféré.