TELECOM Nancy

Réseaux et Systèmes Avancés (RSA)

Projet de Réseaux

Rapport de projet : MyTFTP

Réalisé par :
DARBON Frantz
KESLICK Malaury

Professeurs :
BADONNEL Rémi
CHRISMENT Isabelle

2020

Table des matières

1 Introduction

Le but de ce projet est de créer un client et un serveur UDP implémentant un protocole de transfert de fichiers de type TFTP. Ce travail a été fait en laguage C, Frantz DARBON étant responsable de la partie serveur, et Malaury KESLICK de la partie client.

Dans un premier temps, nous allons vous présenter la manière dont nous avons géré ce projet, puis nous expliciterons nos choix d'implémentation avant de finalement conclure.

2 Gestion de projet

2.1 Matrice SWOT

Voici la matrice SWOT (présentant les forces ($STRENGHTS$), les faiblesses ($WEAKNESSES$), les opportunités ($OPPORTUNITIES$) et les menaces ($THREATS$)) de notre groupe projet.

STRENGTHS

1. Bonne entente au sein du groupe
2. L'un des membres du groupe ayant un grand intérêt pour le C

WEAKNESSES

1. L'un des membres du groupe ayant des difficultés en C

OPPORTUNITIES

1. Permet de mieux comprendre les cours de Réseaux
2. Permet d'entretenir de bons réflexes en language C

THREATS

1. Second semestre de deuxième année très chargé en terme de projets
2. Confinement impactant la motivation et l'efficacité du travail personnel

2.2 Distribution du travail

Comme évoqué dans l'introduction, et comme demandé dans le sujet, nous nous sommes chacun concentré sur une partie du modèle client-serveur. Frantz DARBON s'est chargé de la partie serveur, et Malaury KESLICK de la partie client. Nous nous entretenions régulièrement via audio-conférences afin de statuer de l'avancé de chacun, et de nous assurer que nos parties pourraient au final s'associer comme elles sont censées le faire. Nous avons finalement tous les deux élaborer ce rapport. Voici le tableau récapulatif du temps passé sur ce projet.

Frantz Darbon	Malaury Keslick
Conception serveur : 15h Rapport : 1h	Conception client : 12h Rapport : 2h
Total : 16h	Total : 14h

3 Choix d'implémentation

Dans cette partie nous allons vous expliciter la manière dont nous avons créés notre client et notre serveur UDP qui implémentent le protocole de transfert de fichiers TFTP.

3.1 Principe global

3.1.1 Mise en place de UDP

La mise en place du protocole UDP est résumé par le schéma suivant :

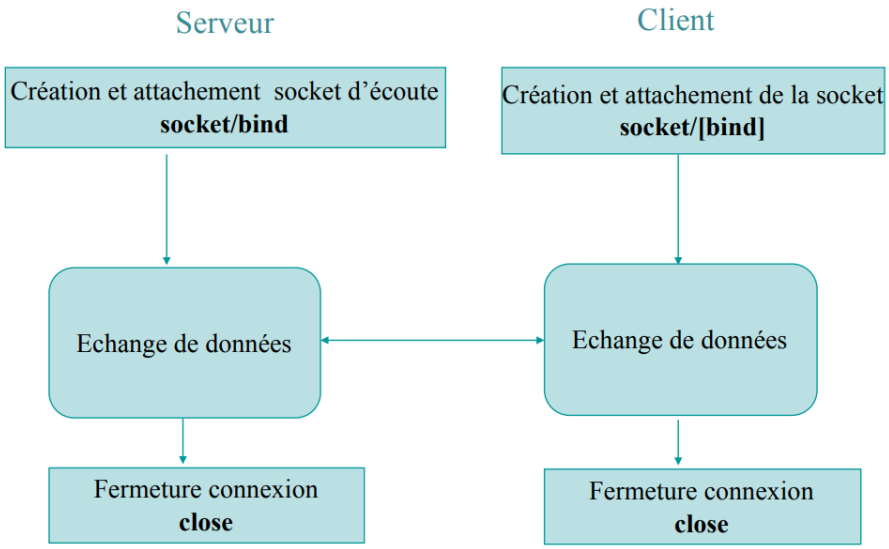

FIGURE 1 – Principe du protocole UDP

3.1.2 Fonctionnement de TFTP

Le protocole de transfert de fichier TFTP (**T**rivial **F**ile **T**ransfert **P**rotocol) fonctionne de la manière suivante :

- Le client envoie une demande d'autorisation de lecture du fichier X (ou d'écriture, mais nous nous concentrerons ici uniquement sur le cas de la lecture) au serveur ;
- Une fois que le serveur a reçu la demande, il récupère le fichier à transmettre et l'envoi par paquets de 512 octets (un paquet fait alors au maximum 516 octets, cf. schéma suivant) ;

— Après chaque paquet envoyé, le serveur attends de recevoir un acquittement de la part du client, preuve de bonne réception du paquet ;
— Un paquet de moins de 516 octets est signe qu'il est le dernier à être envoyé/reçu ;
— Un time-out est mis en place afin de ne pas perdre de paquets(cf sous partie *Mise en place du timer*).

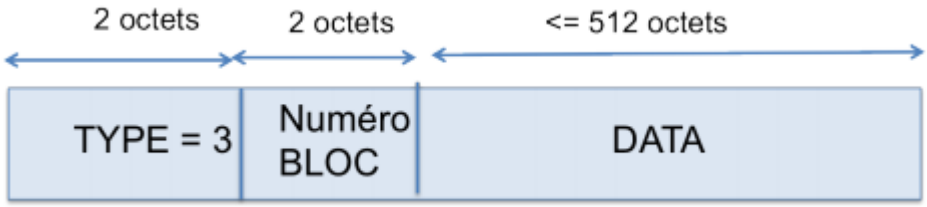

FIGURE 2 – Format d'un paquet envoyé par le serveur

3.1.3 Mise en place du timer

Un timer côté serveur, initialisé à 5 secondes, est lancé à chaque envoi de paquet par le serveur. On considère qu'un paquet est perdu dès lors que l'acquittement du dernier paquet envoyé n'est pas reçu avant la fin du timer.
La valeur du timer est adaptée via un principe de dichotomie. Ce dernier, pour chaque acquittement reçu, est mis à jour comme étant la moyenne entre le dernier RTT et l'ancienne valeur du timer :

$$newTimer = \frac{oldTimer + lastRTT}{2} \tag{1}$$

3.2 Spécificités du serveur

3.2.1 Principe général

Voici le principe de fonctionnement du serveur :

- Le *main* prend en argument une apresse IP et un numéro de port,
- Le serveur crée la socket de dialogue avec le client,
- Le serveur attend la réception du paquet de demande de lecture d'un fichier (RRQ) que le client lui envoie (cf. partie suivante) (*rcvfrom*),
- A la réception du paquet, le serveur le formate en une structure C (facilitant la manipulation des données),
- Le serveur récupère les données du fichier demandé, les stocke en tas dans une String puis la formate afin de créer les différents segments de données à envoyer au client,
- Après envoie du premier segment (*sendto*), le serveur lance le timer et attend l'acquittement validant la réception du premier paquet par le client (*recvfrom*). Principe de **stop-and-wait**,
- Si cet acquittement est reçu, le serveur envoie alors le paquet suivant (*sendto*) et met à jour la valeur du timer,
- Si l'acquittement n'est pas reçu avant la fin du timer, le dernier paquet envoyé est alors considéré comme perdu. Le serveur procède alors au renvoi du dernier paquet (*sendto*),
- On répète ces opérations jusqu'à l'envoi total du fichier.

Pour toutes les étapes précédemment explicitées, des tests sont réalisés afin de vérifier le bon formatage, la bonne allocation en mémoire et la bonne utilisation des données.

3.2.2 Récupération des données du fichier

Dès la réception de la RRQ et donc du nom de fichier associé, un appel à *char* retrieveData(char* filename, char* data)* est effectué. Cette fonction permet :

- L'ouverture du fichier. Si ce dernier n'existe pas, une erreur est soulevée,
- La récupération du contenu du fichier ligne par ligne en supposant qu'une ligne ne fait pas plus de 2048 caractères,
- De récupérer des données dont la taille est limitée par celle du tas. En effet, chaque ligne du fichier source est concaténée à une chaîne de caractères initialisée à 2048 caractères et agrandie par tranche de 2048 au besoin (*realloc*).

3.2.3 Création des segments de données

Une fois la donnée récupérée, un appel à *segment** createDataSegments(char* data)* est effectué. Cette fonction permet :

— Le découpage de la donnée en bloc de 512 octets maximum. Dans le cas où la donnée à une taille qui est un multiple de 512, un bloc additionnel vide est créé (permet au client de savoir qu'il s'agit de la fin du contenu du fichier demandé).
— D'ajouter le type "segment de données" (03) à l'en-tête de chaque segment.
— D'ajouter le numéro de bloc à l'en-tête de chaque segment.

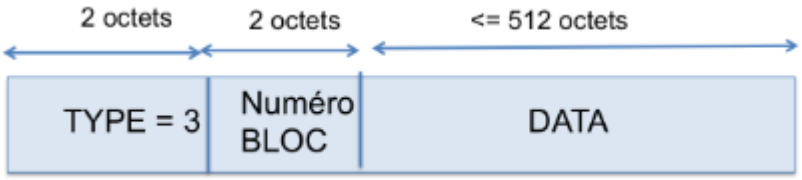

FIGURE 3 – Format des segments de données

Les segments ainsi créés sont alors stockés dans un tableau de *segment*. L'adresse de ce dernier est ensuite retourné au *main*. La fonction *char* segToData(segment* seg)* permet le formatage de chaque segment en chaîne de caractères pour envoi (*sendto*).

3.2.4 Réception d'acquittement

Pour chaque acquittement reçu, le serveur vérifie l'intégrité du paquet et envoie ensuite le prochain segment de données. Comme dit précédemment, il met également à jour le timer.

3.3 Spécificités du client

3.3.1 Principe général

Voici le principe de fonctionnement du client :

- Le main prend en argument une apresse IP, un numéro de port et un chemin de fichier ;
- Le client crée la socket de dialogue avec le serveur ;
- Le client crée le paquet de demande de lecture concernant le fichier passé en paramètre (cf. partie suivante), et l'envoie au serveur (*sendto*) ;
- Le client attend donc de recevoir les paquets de données envoyés par le serveur (*recvfrom*), et stocke les données reçues dans un fichier local nommé *output.txt* (*fputs*) ;
- Après réception d'un paquet, le client envoie un acquittement, preuve de bonne réception des données (*sendto*).
- Si la partie données du dernier paquet reçu est plus petite que 512 octets, alors le client envoie un dernier acquittement, avant de fermer la socket de dialogue et le fichier dans lequel les données sont maintenant toutes stockées ;
- Un time-out est mis en place afin de ne pas perdre de paquets (cf partie *Mise en place du timer*).

Comme pour le serveur, pour toutes les étapes précédemment explicitées, des tests sont réalisés afin de vérifier le bon formatage, la bonne allocation en mémoire et la bonne utilisation des données.

3.3.2 Création de la requête d'accès au fichier

Après récupération du nom de fichier souhaité (passé en argument du main) et mise en place de la socket de dialogue avec le serveur, le client doit envoyer le message contenant la demande d'accès au fichier en question. Le paquet à envoyer a le format suivant :

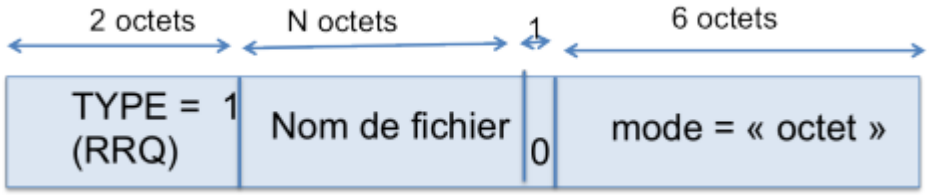

FIGURE 4 – Format du paquet de demande d'accès au fichier

Dans la fonction *char *create_request(char *filename)*, on renvoi donc simplement une chaîne de caractère de la forme "01filename0octet0", les deux premiers caractères

étant signe que l'on demande l'accès en lecture, et le mode étant choisi à "octet" dans l'énoncé. On utilise la fonction *char *strcat(char *dest,char *ajout)* de la librairie *string.h*, qui permet de concaténer des chaînes de caractères, afin de créer cette chaîne de caractère morceau par morceau.

3.3.3 Création d'un acquittement

Lorsque le client reçoit un paquet de données, il doit envoyer un acquittement prouvant la bonne réception de ce dernier. Un acquittement doit avoir le format suivant :

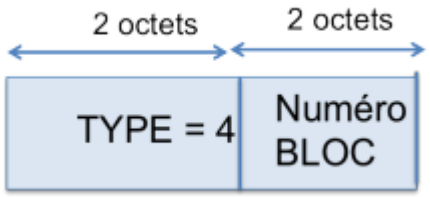

FIGURE 5 – Format d'un acquittement

On procède alors de la même manière que pour la création du segment de demande d'accès au fichier. Dans la fonction *char *create_ack(int acknumber)*, on passe le numéro de bloc reçu en paramètre dans une chaîne de caractères (grâce à la fonction *int sprintf(char *dest, const char *format, int number)*), puis on concatène cette chaîne à la chaîne "04", signe que l'on envoie un acquittement. Cette fois encore, la concaténation est gérée par la fonction *char *strcat(char *dest,char *ajout)* de la librairie *string.h*.

3.4 Simulation

La simulation de test que nous proposons via le makefile de notre git est le suivant :

- Nous avons enregistré au format texte les 8 premières pages du document *THE TFTP PROTOCOL* fourni dans l'enoncé du sujet dans le document test.txt ;
- En lançant *make*, puis *make run_server* dans un premier terminal, et enfin *make run_client* dans un second terminal, la communication se lance telle que décrite dans les parties précédentes ;
- Dans chacun des terminaux correspondants s'affiche la succession d'actions faite par la partie concernée (cf. captures d'écran ci-dessous) ;
- Le résultat de la réception des blocs de données par le client est stockée dans le fichier output.txt à la racine du dépôt. Lorsque la communication est coupée, les fichiers test.txt et output.txt sont donc identiques.

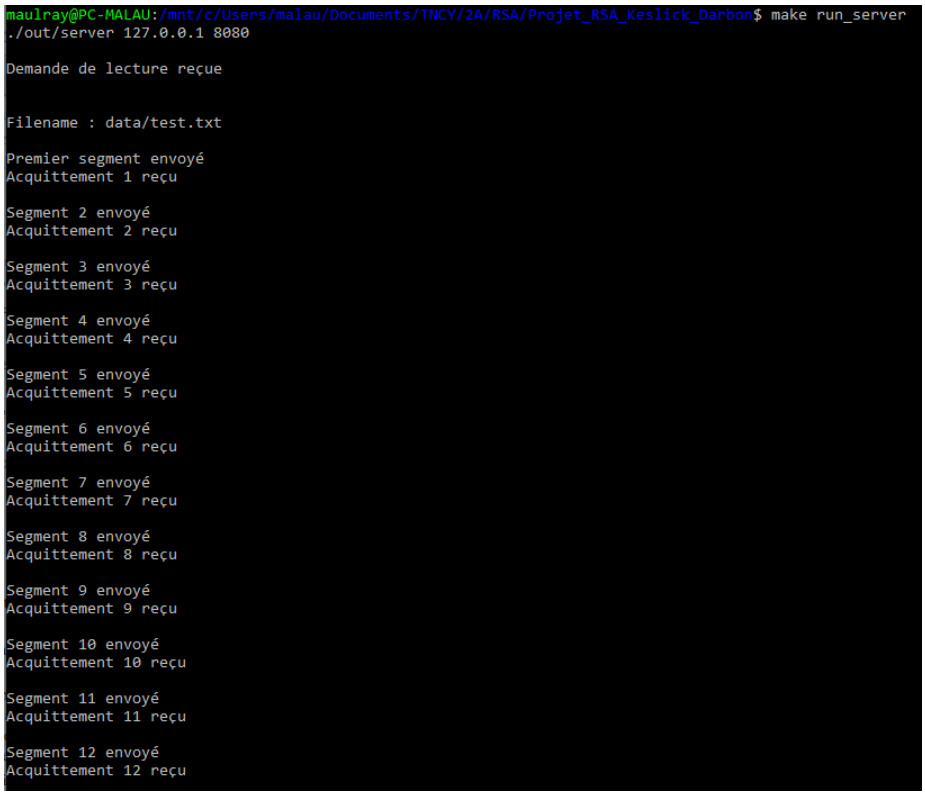

FIGURE 6 – Extrait de l'affichage serveur durant l'exécution

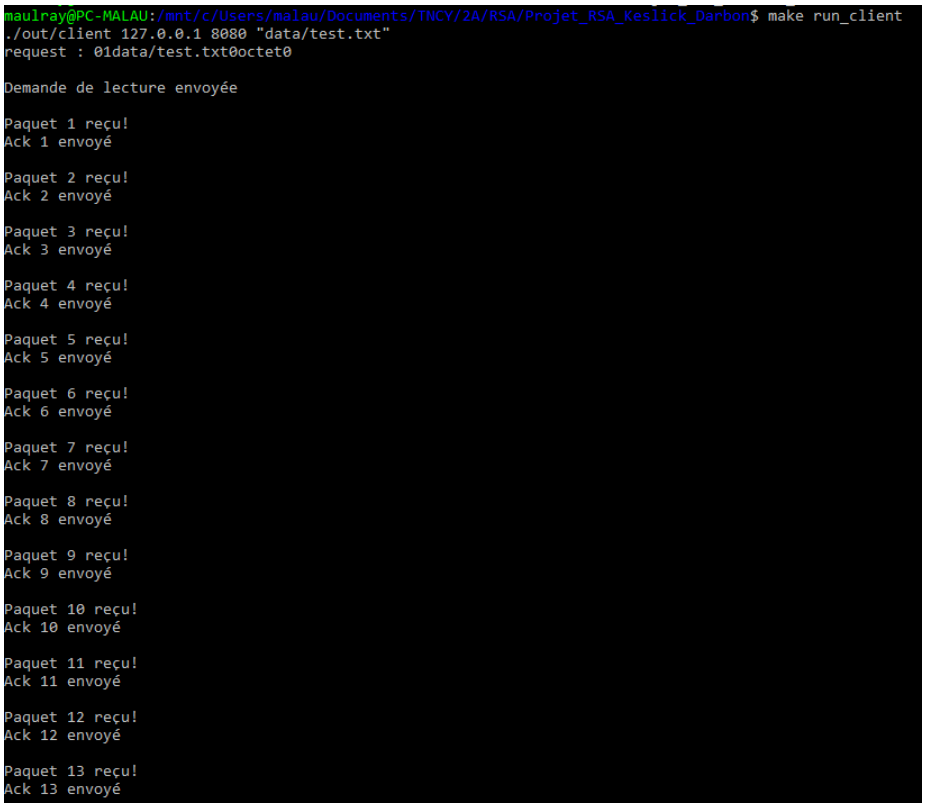

FIGURE 7 – Extrait de l'affichage client durant l'exécution

4 Conclusion

Nous vous avons donc présenté la manière dont nous avons mis en place notre client-serveur UDP implémentant un protocole de transfert de fichiers TFTP. Ce projet nous aura permis de mieux comprendre la partie Réseaux du module RSA de ce semestre.