Projet réseau

Rapport de projet

Erwan Kessler; Yann Meyer

6 Mai 2020

Contents

1 Introduction

L'objet de ce rapport est de présenter et d'expliquer l'implémentation effectuée dans le cadre du projet du module Réseau et Système Avancé (RSA).
En effet, ce projet consiste en l'implémentation d'un client et d'un serveur TFTP.
Cette paire client/serveur doit permettre, dans un premier temps la lecture d'un fichier situé sur le serveur par le client. Puis, dans un deuxième temps, la fonctionnalité d'écriture d'un fichier sur le serveur par le client sera ajoutée.
Conformément au sujet, ces deux opérations s'effectuent uniquement en mode *octet*, ne gèrent pas les paquets d'erreurs et le serveur ne peut gérer la connexion de plusieurs clients.

2 Principe de développement

Dès le début du projet, nous avons identifié une symétrie du travail a effectuer lors de l'implémentation de la paire client/serveur.
En effet, le fonctionnement du serveur en mode *WRQ* est très similaire à celui du client en mode *RRQ*. Inversement, le fonctionnement du serveur en mode *RRQ* est proche de celui du client en mode *WRQ*. Ainsi, nous avons décidé d'introduire un fichier nommé *utils.c*. Ce fichier contient les fonctions nécessaires à la fois au client et au serveur. Ces fonctions sont donc le plus générique possible afin de faciliter leur utilisation dans les deux contextes différents.
Ce choix de conception s'est avéré particulièrement judicieux, nous permettant de gagner du temps.

3 Serveur

Le serveur peut doit recevoir de la part des clients un nom de fichier qui servira soit à l'écriture soit à la lecture du dit fichier. Par la suite, suivant le code d'opération, l'une des deux suivantes opérations est effectuée. Le serveur serveur ne s'arrête pas après un transfert mais ne peut pas gérer plus d'un client à la fois.

3.1 WRQ

Cette opération permet d'écrire dans un fichier présent sur le serveur le contenu envoyé par le client, pour cela, il est d'abord impératif de vérifier que le client a bien accès à la zone d'écriture cible et que le fichier n'existe pas déjà sur le serveur. Dans les cas contraires, un message d'erreur est renvoyé (conformément au RFC, 06 pour un fichier déjà existant et 02 pour une violation d'accès, les autres erreurs n'ont pas été implémentées).
Une fois ces précautions prises, le serveur envoie un *ACK* et reçoit chaque paquet de données qu'il confirme avec le bon *block id* (les id de blocs vont de 1 à 99 et puis repassent à 1). Si on reçoit deux paquets identiques on sait que notre *ACK* n'est pas arrivé, on défausse alors simplement le paquet doublon.

3.2 RRQ

Cette opération permet de lire un fichier du serveur. Pour cela, on vérifie l'existence du fichier, confirme la réception du nom du fichier au client et on envoie les paquets de données en incrémentant le *block id*. Cette opération est plus délicate que la précédente car on doit s'assurer que le client reçoit bien les données. Pour cela on attend avec un *select* pendant 5 secondes que le client envoie son *ACK*. Si cela n'arrive pas, alors on renvoie le paquet jusqu'à 5 fois, si on obtient aucune réponse on suppose alors que le client s'est déconnecté et on rompt la connexion avec le client.
Si le client envoie les bons *ACK* alors le serveur envoie toute les données par bloc de 512 octets jusqu'à ce qu'il ne reste plus de données, le dernier bloc est donc de taille différente de 512.

4 Client

Le client doit pouvoir envoyer et récupérer un fichier du serveur, il n'est actif que durant le transfert et quitte immédiatement en cas de paquet d'erreur ou de fin de transmission.

4.1 GET

Cette opération permet de demander la lecture d'une ressource sur le serveur.
Le client créé donc un en-tête *RRQ* avec le nom du fichier ainsi que le mode d'opération (le sujet spécifie que celui ci est toujours égal à *octet*). Le client crée alors un fichier local portant le même nom est va écrire dedans les données reçues du serveur. Il envoie un *ACK* à chaque fois qu'il a reçu un paquet et s'il reçoit deux fois le même paquet alors son *ACK* originel a été perdu, il le renvoie donc et attends la réponse du serveur. La taille du paquet reçu finit par passer en dessous de 512, à ce moment là le client sait que toutes les données ont été récupérées et ferme la connexion.

4.2 PUT

Cette opération permet d'envoyer un fichier au serveur.
Une fois l'en-tête WRQ envoyé, il est important d'attendre que le serveur envoie un *ACK*, si cela n'est pas le cas, la requête est renvoyée. Une fois que la demande d'envoi a été effectué, le fichier est ouvert localement en lecture seule et pour chaque paquet de 512 envoyé, les *ACK* du serveur sont vérifiés. Si, durant 5 secondes le serveur ne répond pas, alors la donnée est renvoyée et cela jusqu'à 5 fois. Si le serveur ne répond pas, le client quitte la connexion, sinon il continue l'envoi augmentant progressivement le numéro de bloc de 1 à 99 (puis retourne à 1 si nécessaire).
Une fois le fichier entièrement envoyé le client ferme la connexion.

5 Tests et simulation

Afin de pouvoir tester l'intégralité des fonctionnalités de ce projet, nous avons du ajouter un générateur de pseudo aléatoire déterminant si oui ou non un paquet *ACK* est perdu. Ce seuil est fixé à 20% afin de pouvoir facilement visualiser les pertes artificielles de paquet sans compromettre la capacité de notre programme à transférer les fichiers. En effet, le timeout est fixé à 5 secondes et le client comme le serveur interrompent la connexion après 5 échecs consécutifs.
Pour chaque paquet échangé, la probabilité que la transmission n'aboutisse pas est donc de $0.2^5 = 0.00032$. Pour une transmission relativement courte, on peut donc raisonnablement considérer que la transmission aboutira.

6 Gestion de projet et répartition des heures de travail

En raison des circonstances sanitaire, ce projet a été effectué dans sa quasi-totalité à distance.
Ainsi, nous avons du mettre en place un canal de communication virtuel. Nous donc avons décidé de mettre en place un serveur *Discord* dédié au projet RSA.
Toutefois, lors des phases du projet nécessitant un travail en collaboration tel que la mise en place des fonctions utilitaires et le débogage, nous avons aperçu les limites du travail à distance.

Tâches	Erwan	Yann
Lecture et compréhension de la RFC	0.5h	2h
Développement utilitaires	2.5h	3h
Implémentation serveur	2h	10h
Implémentation client	12h	1h
Débogage	1.5h	2h
Tests	3.5h	2h
Rapport	1h	2h
Total	23h	22h

7 Conclusion

Ainsi ce projet nous a permis de comprendre en profondeur les protocole UDP, TFTP ainsi que les fonctions C associés à la création d'une paire client/serveur compatible IPv6. Nous avons donc pu concrétiser le travail réalisé dans la partie réseau du module RSA.
Ce projet nous a également permis de nous familiariser avec la lecture d'une RFC nécessitant la connaissance d'un langage particulier et une attention particulière aux détails.

8 Ressources

- https://tools.ietf.org/html/rfc1350
- https://linux.die.net/man/3/getaddrinfo
- https://linux.die.net/man/2/sendto
- https://linux.die.net/man/2/bind
- https://linux.die.net/man/2/select
- https://linux.die.net/man/2/recvfrom
- https://linux.die.net/man/2/connect