Rapport de projet : TRTP
LINGI1341

BELLON Guillaume, BEYRAGHI Vahid
30151600, 32381600

Octobre 2018

1 Introduction

Il y a deux semaines, il nous a été demandé de réaliser un projet qui nous permettrait de nous familiariser avec la communication entre plusieurs machines via un réseau. Nous devions réaliser un programme, écrit en C, qui permettrait d'envoyer ou de recevoir des données au moyen d'un nouveau protocole : le Truncated Reliable Transport Protocol (ou TRTP). Ce protocole doit être fiable et basé sur des segments UDP, tout en ne contenant aucune fuite de mémoire. Il doit être basé sur une implémentation adoptant la stratégie du selective repeat et doit permettre la troncation du payload, tout en fonctionnant sur un protocole IPv6. Nous vous proposons de découvrir le fonctionnement général ainsi que les choix de conception posés lors de l'élaboration de ce code.

2 Fonctionnement général

Deux programmes nous étaient demandés : un programme sender et un programme receiver.
Le programme sender est, comme son nom l'indique, chargé d'envoyer des données. On lui spécifie en argument l'adresse IPv6 ou le nom du domaine du receiver, ainsi que le numéro du port UDP sur lequel le receiver est lancé, puis, le sender enverra les données reçues depuis l'entrée standard vers le receiver. Un autre argument optionnel peut être rajouté : [-f X]. Si cette option est présente, X contient le chemin vers le fichier contenant les données à envoyer, et les données sont transmises sans passer par stdin.
Receiver, quant à lui, va aussi prendre en argument l'adresse IPv6 ou le nom de domaine où accepter la connexion, et le numéro de port UDP sur lequel écouter. Il prend également l'argument optionnel [-f X] qui, si spécifié, redirigera la sortie du receiver vers le fichier de pathname X. Si f n'est pas spécifié, la sortie de receiver sera affichée sur stdout.

Lorsqu'un sender est lancé, il va lire les données à envoyer via stdin (ou via un fichier par l'option f) et créer un paquet de données, composé d'un header contenant des informations telles que le type du paquet, la longueur du payload ou encore si le paquet a été tronqué. Le paquet de données est également composé d'un payload qui contient de manière opaque les données à envoyer. On retrouve également deux CRC qui permettent de s'assurer que, ni le header, ni le payload n'ont été corrompus. Une fois le paquet créé, il va créer un socket, un lien entre l'envoyeur et le receveur des données, afin d'y envoyer le paquet.
Le receiver lancé, il va lui aussi créer un socket avec les mêmes caractéristiques que le sender afin d'établir la connexion. Il va alors recevoir le paquet de données, vérifier si il n'a été modifié durant le transfert, puis en extraire le payload sur stdout (ou sur un fichier par l'option f). Il va également renvoyer au sender un ACK pour indiquer que le paquet est bien arrivé.

3 Approche du problème et choix de conception

Pour nous aider dans ce projet, il nous a été demandé de faire des exercices sur Inginious. La plupart des codes demandés dans ces exercices permettaient d'être directement intégrés dans le projet, ce qui réduisait le champs des possibilités d'implémentation des sender et receiver. Nous avons donc basé notre implémentation de la manière suivante :
Lorsque le sender reçoit de l'information à envoyer, il crée un socket afin d'établir la connexion entre l'émission et la réception grâce à la fonction create_socket qui prend en argument le nom du domaine/adresse IPv6 et le port sur lequel émettre. Le socket créé, il restera le même jusqu'à la fin de l'envoi. Le receiver va également créer un socket avec ces mêmes paramètres. Une fois fait, le sender crée un paquet de donnée sous la forme d'une structure pkt_t. La structure pkt_t contient un header, un timestamp, un CRC1, un payload et un CRC2.

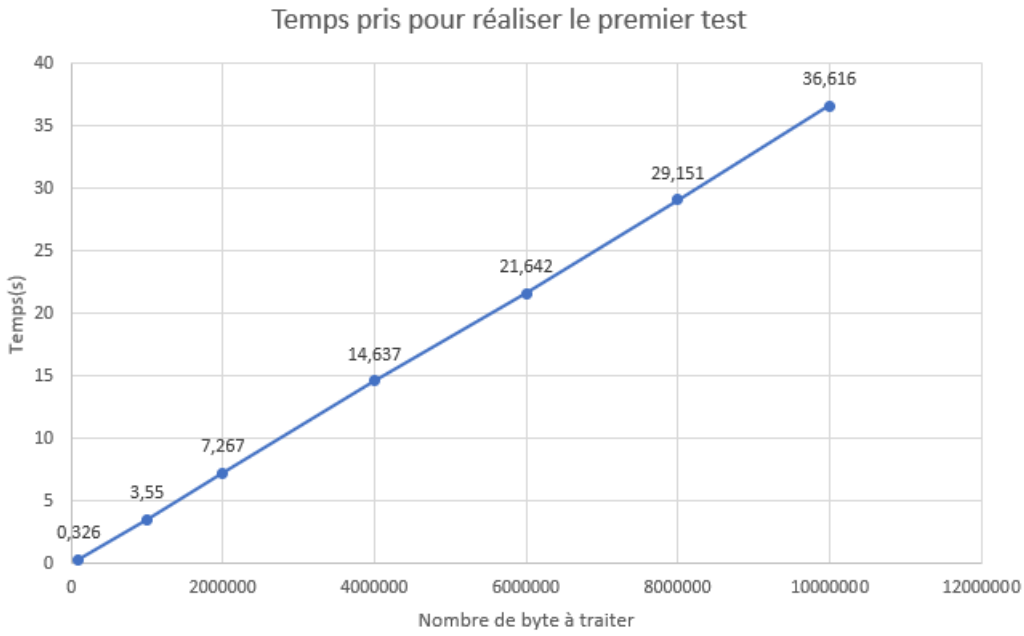
1. Header : Contient le type de paquet, l'indication de troncature, la taille de la window, le numéro de séquence du paquet et la longueur du payload attribué.
2. Timestamp : Nous n'avons pas défini d'utilité à ce champ.
3. CRC1 : Résultat de l'application du CRC32 sur le header, permet de vérifier que le header n'a pas été corrompu durant le transfert.
4. Payload : Contient les données à transmettre.
5. CRC2 : Résultat de l'application du CRC32 sur le payload, permet de vérifier que le payload n'a pas été corrompu durant le transfert.

Une fois le pkt_t créé, le sender va envoyer ce dernier via le socket créé puis, va stocker le pkt_t, son numéro de séquence et le temps auquel il a été créé dans un buffer nommé window. Le but de cette manipulation est de placer le pkt_t en attente d'un acknowledgment du receiver. Si ce dernier arrive, on delete le pkt_t de la window, si il n'arrive pas avant le RTT ou si un nack est reçu, on renvoie le pkt_t en question en réinitialisant son temps de création.
Le receiver lui va attendre un pkt_t via le socket. Une fois reçu, on vérifie qu'il n'est pas tronqué (auquel cas on renvoie un nack et on discard le paquet), si son numéro de séquence correspond à un numéro attendu dans la window (au cas contraire on le discard) et s'il n'a pas été corrompu (auquel cas on le discard aussi). Si le numéro de séquence du pkt_t reçu n'est pas au début de la séquence attendue, mais bien dans l'ensemble des numéros attendus, le receiver va le stocker dans un buffer et renvoyer un ack indiquant qu'il attend toujours le premier pkt_t. Si le numéro de séquence du pkt_t reçu correspond à celui attendu, on le traite, on envoie au sender un ack puis on incrémente le numéro de séquence attendu. On regarde ensuite dans le buffer si on ne peut pas traiter les pkt_t stockés afin de vider le buffer.
Le sender regarde de manière régulière si les RTT des pkt_t stockés ne sont pas dépassés. Si non, il va regarder si sa window est pleine et, si non, envoyer un nouveau paquet de données. Si la window est pleine, il va attendre des acks ou nacks du receiver et agir en conséquence. Si le sender atteint le EOF de l'entrée et que sa window est vide, il transfert un EOF au receiver et termine son exécution. Le receiver termine également son exécution après réception du EOF.
La valeur du RTT a été choisie à 4 secondes étant donné que la latence du réseau pour acheminer un paquet varie entre 0 et 2 secondes. Il faut donc poser un RTT d'au moins le temps d'un aller-retour.

4 Tests

Il nous a également été demandé d'implémenter divers tests afin de vérifier le bon fonctionnement de notre implémentation. Plusieurs tests ont été réalisés. Un test a été codé en utilisant la librairie de tests CUnit, les autres via les fonctions main des différentes classes.

Le premier test consiste à créer des fichiers de taille variable, d'envoyer ces fichiers via le programme, et de comparer le fichier entré avec le fichier de sortie. La figure ci-dessous montre le temps pris par le programme test en fonction du nombre de bytes qu'il doit traiter. Si ces temps semblent long, c'est surtout car le test doit d'abord écrire un fichier de la taille demandée (c'est la partie qui prend le plus de temps). On constate que l'augmentation du temps est équivalente au nombre de bytes demandé (si on multiplie par 10 le nombre de bytes, on fait de même pour le temps).



Nous avons également fait quelques programmes de test internes aux fonctions, tel que les tests des fonctions de la window. D'autres tests ont également été envisagés comme un test qui applique des défauts sur la connexion (troncature, corruption, perte de données, ...) mais n'ont pas été réalisés par manque de temps.

5 Conclusion

Malgré une implémentation correcte et un fonctionnement global, il reste quelques améliorations à apporter à notre code. Il reste encore à gérer quelques fuites de mémoires et tester tous les cas possibles d'erreurs afin de vérifier la robustesse du programme. Nous devons également utiliser le Timestamp et gérer le fait que les windows sont de taille variable, ainsi que tester l'intéropérabilité de notre implémentation.