Rapport de projet : TRTP
LINGI1341

BELLON Guillaume, BEYRAGHI Vahid
30151600, 32381600

Octobre 2018

1 Introduction

Il y a quatre 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.

2.1 Sender

Lorsqu'un sender est lancé, il va créer un socket, un lien entre l'envoyeur et le receveur des données, afin d'établir une connexion. Il va ensuite lire sur stdin (ou sur un fichier par l'option f) les données à envoyer et créer un paquet de données composé d'un header contenant des informations telles que le type du paquet (dans ce cas-ci DATA_TYPE), 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. Dans ce paquet, il y a aussi un champ TimeStamp, mais celui-ci n'a pas d'utilité dans notre code.

Une fois le paquet créé, il sera envoyé vers le sender via le socket, puis stocké dans une window le temps de recevoir un accusé de réception. Si l'accusé de réception arrive et est de type ACK, le paquet est supprimé de la window. Si le paquet a été tronqué durant le transfert, l'accusé de réception est de type NACK et on renvoie alors le paquet. Si aucun accusé de réception n'a été reçu après un temps RTT, on renvoie le paquet. Plusieurs paquets sont envoyés et stockés en même temps.
Si un EOF est lu, le sender attend que sa window soit vide (et donc qu'il ai reçu tous les accusés de réception) avant d'envoyer un paquet de longueur nulle et de payload vide signalant que le receiver peut se déconnecter. Il va dès lors renvoyer ce paquet jusqu'à recevoir un ACK. Si un ACK est reçu ou si rien n'est reçu pendant 10 secondes, le sender se déconnecte, ferme le sfd et le FileDirector. Le programme est terminé.

2.2 Receiver

Le receiver lancé, il va lui aussi créer un socket avec les mêmes caractéristiques que le sender afin d'établir la connexion et va ensuite recevoir les paquets de données échangés. Si un paquet reçu est tronqué, on vérifie si son seqnum est dans la liste des seqnums attendus. Si non, on le discard, si oui on renvoie un NACK au sender, indiquant que le paquet a été tronqué. Si le paquet n'a pas eu de souci durant le transfert et si son seqnum n'est pas le premier seqnum attendu mais est bien dans la liste des seqnum attendus, le receiver va stocker ledit paquet dans un buffer (s'il n'y est pas déjà) et renvoyer un ACK du premier seqnum attendu. Si le paquet n'a pas eu de souci durant le transfert et si son seqnum est le premier seqnum attendu, il récupère le payload du paquet et le traite en l'envoyant sur la sortie. Le receiver va alors regarder dans son buffer si les seqnum des paquets stockés correspondent au nouveau senqnum attendu, les traiter si c'est le cas, les enlever du buffer et envoyer l'ACK du prochain seqnum attendu. Si un paquet reçu a été corrompu durant le transfert, on le discard.
Si un paquet traité a une longueur de payload égale à 0, le receiver comprend que le sender demande une déconnexion et lui renvoie un ACK avant de se déconnecter.

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 programmes 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.

- Le sender devant avoir une taille de window variant en fonction des disponibilités du receiver, nous avons fait en sorte que, lorsque le sender reçoit un ACK ou NACK, il prenne connaissance du nombre de place libre dans le buffer du receiver via le champ window et n'envoie pas de nouveaux paquets tant que le nombre de paquets stockés dans la window dépasse ou égale ce nombre. Cela permet de ne pas saturer le réseau alors que le receiver n'est pas apte à recevoir plus de données.

- 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.

- Le paquet de longueur 0 indiquant l'atteinte du EOF n'est envoyé par le sender que lorsque sa window est vide, c'est-à-dire que lorsque toutes les données envoyées ont reçu leur ACK. Cela évite de perdre des données et permet aussi au receiver de pouvoir se déconnecter facilement

sans chercher à vider son buffer. Le sender lance alors un timer de 10 secondes et se déconnecte s'il reçoit un ACK, s'il a une erreur en essayant d'envoyer le RTT sur le socket (signifiant que la connexion est rompue) ou si le timer atteint les 10 secondes.

- Le selective repeat a été implémenté au moyen du buffer dans le receiver. Le buffer stocke les paquets dans la séquence de seqnum attendus en renvoyant le ACK du premier seqnum attendu et, une fois que ce dernier arrive, traite les seqnums suivants déjà stockés dans le buffer (si leur seqnum sont les bons). Le sender possède une méthode qui supprime de sa window les paquets dont le numéro de séquence est inférieur au seqnum du dernier ACK reçu, ce qui permet de vider la window plus vite.

4 Tests

Il nous a également été demandé d'implémenter divers tests afin de vérifier le bon fonctionnement de notre implémentation. Un code de vérification a été implémenté au moyen de CUnit, permettant de tester nos différentes méthodes et programmes. Ces test exécutent sender et receiver simultanément au moyen d'un fork et transfèrent des fichiers différents sur des connexions différentes. Tests réalisés :

- Test d'envoi d'un fichier de 1kb sur une connexion parfaite et sur une connexion avec 25% d'erreur, 25% de perte, 100ms de délais et 200ms de jitter.
- Test d'envoi d'un fichier de 100kb sur une connexion parfaite et sur une connexion avec 1% d'erreur et 1% de perte.
- Test d'envoi d'un fichier de 1kb sur une connexion avec 20% de paquets tronqués, 5% de corruption, 5% de perte, 100ms délais et 200ms de jitter sur l'aller et le retour.
- Test d'envoi d'un fichier de 100kb sur une connexion avec 2% de paquets tronqués, 1% de corruption, 1% de perte, 100ms délais et 200ms de jitter sur l'aller et le retour.
- Test des fonctions de classe Window
- Test des fonctions de la classe Common_Lib

5 Conclusion

Après ces 4 semaines d'implémentation, nous sommes fiers du résultat final. En effet nous n'avons pas de fuites de mémoires et avons respecté (à notre sens) toutes les consignes du cahier des charges. Quelques améliorations peuvent encore malgré tout être apportées à notre code, telles qu'un RTT variable au cours du temps et changeant en fonction de la congestion du réseau, ou encore trouver une utilité au champ TimeStamp de la structure du paquet de données.