LING1341 : Projet Réseaux 2018

Liliya Semerikova Alexandre Gobeaux

22 octobre 2018

1 Introduction

Dans le cadre du cours de Réseaux donné en 2018 à l'EPL par Olivier Bonaventure, il nous a été demandé d'écrire deux programmes : *sender* et *receiver*. Ces programmes permettront de faire un transfert de données entre deux machines distantes.

2 Architecture

L'architecture générale de notre projet est composée de la façon suivante. Nous avons deux programmes : *sender* et *receiver*. L'utilisateur entre sur la ligne de commande un fichier potentiel précédé de l'argument *-f* et le *hostname* et le *port* de receiver. Le fichier passé en argument pour sender sera le contenu transféré vers receiver tandis que le fichier passé en argument pour receiver sera le fichier sur lequel le contenu reçu écrit.

Le programme va alors lire le fichier tout en envoyant et créant des paquets encodés avec des *payloads* d'une taille maximale de 512 octets. A son tour, receiver reçoit les paquets et vérifie leur validité avec le *crc*. Pour la suite, le *receiver* et le *sender* communiquent en *selective repeat*. Si le paquet est considéré comme bon, *receiver* le sauvegarde. Juste après, il envoie un ack à *sender* en indiquant le numéro du paquet attendu. Dans le cas opposé, le paquet est ignoré et on envoie également un *ack* avec le numéro du paquet attendu. Dans le cas du paquet tronqué, nous envoyons un *nack* avec cette fois-ci le numéro de séquence du paquet reçu. On répète la procédure jusqu'à la fin du fichier.

3 Choix de conception

Pour sauvegarder tous les *payloads* à envoyer, nous avons créé un *buffer* modélisé par une queue (FIFO) dans le *sender*. Chaque nœud de ce *buffer* possède un paquet et le moment auquel le paquet est envoyé. La queue est de taille limitée et la limitation est indiquée par la *window* que *receiver* nous envoie avec chaque *acknowledgement*. Dans ce FIFO nous avons un pointeur vers le premier élément d'une queue - *head*, mais aussi pour le dernier - *last*. Le fait de mettre un pointeur vers le dernier élément, nous permet réduire la complexité de *O(n)* à *O(1)* lorsqu'on *push* un élément dans ce *buffer*.

Ensuite, nous avons aussi deux *buffers* en mode FIFO dans le *receiver*. Le premier sert à sauvegarder les *acks* pour les renvoyer. Le deuxième est là pour sauvegarder les paquets à écrire dans le fichier ou sur la sortie standard. La particularité de cette dernière queue (FIFO) est qu'elle *push* selon le numéro de séquence. Donc nous obtenons à la fin une queue triée. Nous commençons à lire dès que notre queue commence par un paquet avec 0 comme numéro de séquence. Par après, si le paquet suivant possède un numéro +1 par rapport à paquet précédant, on l'écrit. Sinon, on attend que le paquet arrive. Si, par contre, le numéro de séquence est 255, le prochain paquet qu'on attend doit avoir 0 comme le numéro de séquence.

Dès qu'on reçoit un *ack* avec le prochain numéro de séquence attendu, nous allons supprimer tous les paquets dans le *sender* avec le numéro de séquence plus petit que le numéro de séquence reçu. Quand nous le faisons, nous nous rendons compte que le numéro 255 est plus petit que 0, si le paquet avec le numéro de séquence égal à 0 se trouve après le paquet avec le numéro 255.

Si *receiver* nous envoie un *nack*, nous remettons le temps de ce paquet à 0. La même chose se produit lorsque le *retransmission timer* est expiré pour un des fichiers. Remettre le *timer* à 0 permet à notre programme de savoir qu'il faut renvoyer ces paquets.

La valeur du *retransmission timer* pour le premier paquet est mise à 4100ms. C'est la valeur du pire cas qu'on peut avoir selon l'énoncé car la latence maximale pour un paquet vaut 2000 ms. Les 100 ms rajoutées sont une borne au temps de calcul de decode. Ensuite, dès que le premier paquet arrive, nous cherchons le moment auquel le paquet a été envoyé. Grâce à ceci, nous allons remettre le temps le plus adapté pour notre cas. C'est-à-dire qu'on utilise comme *retransmission timer* le double de la différence de temps entre la réception de l'ack et l'envoi du paquet. On double cette différence pour avoir une sécurité.

Pour le moment, la taille de window choisie change en fonction de la place qu'il nous reste dans le *buffer*, comme c'était précisé dans l'énoncé. Mais selon les nouveaux changements, nous comptons le modifier et faire une taille de *window* variable selon la difficulté de calcul ou encore selon la congestion du réseaux. En effet, si on reçoit on paquet tronqué, ça veut dire que le réseau est chargé. Effectivement, c'est à ce moment qu'il faudrait changer la taille de la *window* et envoyer moins de paquets. Nous ne gérons pas encore ce cas-là, mais nous comptons le faire avant la soumission finale.

4 Débogage et tests

Nous avons réalisé des tests qui nous permettent de vérifier le bon fonctionnement de notre queue. Mais aussi, en lançant *sender* et *receiver* sur un fichier de 905616 caractères et en comparant le fichier de sortie avec le fichier d'entrée en utilisant la commande *diff*. Nous avons obtenu des fichiers identiques. Par contre, si on essaie d'envoyer un fichier de moins de 512 octets, *sender* n'arrive pas à se déconnecter. Mais nous comptons régler ce problème par la suite.

5 Évaluation qualitative

En faisant les tests avec Valgrind, nous nous sommes rendus compte qu'il y a encore des fuites de mémoire dans notre programme. Les commandes utilisées pour cela sont :

`$ valgrind -leak-check=full -show-leak-kinds=all -track-origins=yes -v ./sender -f text.txt ::1 12345` et `$ valgrind -leak-check=full -show-leak-kinds=all -track-origins=yes -v ./receiver -f output.txt ::1 12345`

6 Conclusion

A ce stade-là, notre projet n'est pas parfait. Mais nous arrivons à gérer l'envoi d'un fichier à condition qu'il n'y ait pas de pertes. Pour la suite, nous comptons l'améliorer en ajoutant le changement de taille de *window* en fonction des

difficultés de calculs. Mais aussi gérer les cas particuliers, comme par exemple le fait qu'un ack peut arriver alors que le paquet n'existe plus sur le buffer. Enfin, nous comptons créer des tests pour nos sender et receiver avec linksim pour assurer un transfert de données fiable.