LING1341 : Projet Réseaux 2018

Liliya Semerikova Alexandre Gobeaux

11 mai 2018

1 Introduction

Dans le cadre du cours de Réseaux donné en 2019 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 le *hostname* et le *port* de receiver, ainsi qu'un fichier d'entrée et pour le receiver aussi un fichier de sortie.

Le programme va alors lire le fichier tout en envoyant et créant des paquets encodés avec des *payload* au plus de 512 octets. A son tour, receiver reçoit les paquets et les vérifie avec le *crc*. Pour la suite, le *receiver* et le *sender* "se parlent" en mode de *selective repeat*. Si le paquet est considéré comme bon, *receiver* le sauvegarde. Juste après, il envoie un ack à *sender* en indiquant le numéro de suivant paquet attendu. Dans le cas opposé, le paquet est ignoré et on envoie toujours le *ack* en tenant le compte du prochain fichier attendu. Dans le cas du paquet tronqué, nous envoyons un *nack*. On répète la procédure jusqu'à la fin du fichier.

3 Choix de conception

Pour sauvegarder tous les *payload* à envoyer, nous avons créé un *buffer* en mode FIFO dans le *sender*. Chaque nœud de ce *buffer* possède un paquet et le temps d'envoi de ce paquet. La queue est de taille limitée et la limitation est indiquée par le *window* que *receiver* nous envoie avec chaque *ack*. 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)* vers *O(1)* quand on fait *push* un élément dans ce *buffer*.

Ensuite, nous avons aussi deux *buffers* en mode FIFO dans le *receiver*. Le premier sert à sauvegarder les *ack* pour les renvoyer. Le deuxième est là pour sauvegarder les paquets à écrire dans le fichier ou sur la sortie standard. La particularité de ce FIFO c'est qu'il *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, les numéro de séquence est à 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, on se rend 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*, dans ce cas nous remettons le temps de ce paquet à 0. On fait pareil si on se rend compte que le *transmission timer* est expiré pour un des fichiers. Le fait que le *timer* est remis à 0, notre programme sait qu'il faut les renvoyer.

La valeur de *retransmission timer* pour le premier fichier est mis à 4100ms. C'est la valeur du pire cas qu'on peut avoir selon les énoncés, car latence est mis à 2000 ms et on prend aussi en compte le temps de traitement des données. Ensuite, dès que le premier fichier arrive, nous allons chercher le temps à quel il était envoyé dans le buffer. Et selon ça, nous allons remettre le temps le plus adapté pour notre cas. C'est à dire le temps à quel on a reçu le fichier moins le temps d'envoi et la différence on multiplie par 2 pour en être sur.

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 le *window* variable selon la difficulté de calcule. En effet, si on reçoit on paquet tronqué, ça veut dire que le réseau est changé. Effectivement, c'est le temps de diminuer *window* et envoyer moins de paquets. Pour le moment on ne gère pas encore ce cas-là, mais on compte le faire avant la soumission finale.

4 Débogage et tests

5 Évaluation

5.1 Qualitatif

En faisant les tests avec Valgrind, nous nous sommes rendus compte qu'il y a encore des fuites de mémoire dans notre programme. La commande utilisée pour cela est :

```
$ valgrind -leak-check=full -show-leak-kinds=all -track-origins=yes -v ./main - -d output
```

Cependant, ces fuites sont de l'ordre de quelques centaines de *bytes* et ne sont donc pas réellement importantes.

5.2 Quantitatif

Voici le temps d'exécution de notre code. On observe bien que multi-threader est bénéfique pour la rapidité du programme. Le nombre optimal de *threads* est entre 1 et 2 *threads* par cœur sur le processeur.

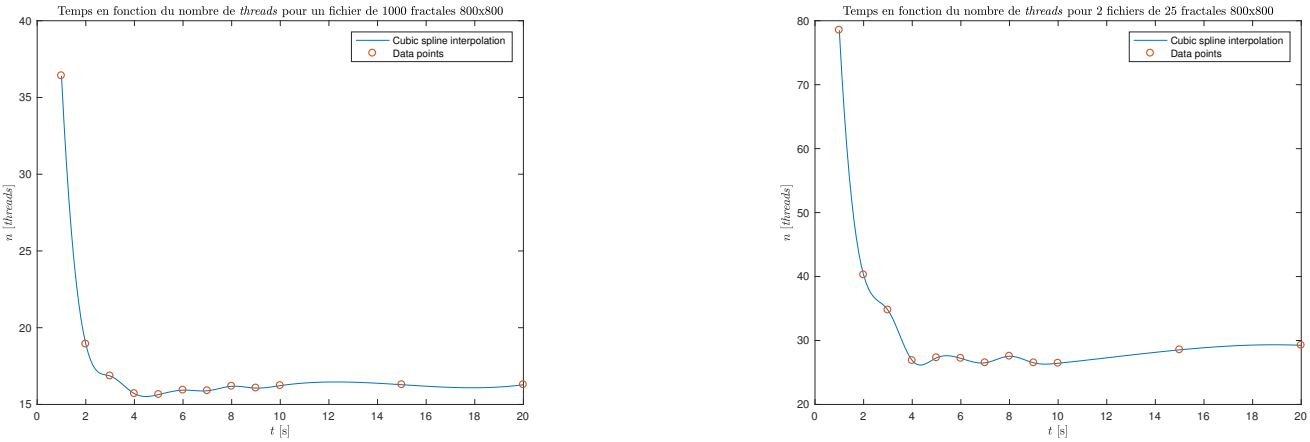

FIGURE 1 – Données et interpolations cubiques de celles-ci.

6 Conclusion

Ce projet nous a permis de nous familiariser avec le langage C ainsi qu'avec les *threads*, les sémaphores, les mutexs ou encore Linux. En plus, il s'agissait d'une opportunité d'utiliser la gestion de versions avec `git` ou encore les outils de débogage tels que Valgrind et `gdb`.