LINGI1341 - Rapport Projet 1

Navarre Louis 1235-1600 et Plancq Hadrien 5666-1600

Novembre 2018

1 Introduction

Ce rapport est relatif au projet de groupe pour le cours LINGI1341 Computer Networks, pour l'année académique 2018-2019. Ce projet a pour but d'implémenter un protocole fiable pour envoyer des données entre deux machines (pouvant être différentes) à travers un réseau pouvant provoquer par exemple des pertes ou des corruptions de données.

2 Gérer les timing

Lors de l'envoie d'un paquet de type `PTYPE_DATA`, nous définissons le champ *timestamp* à l'aide de la fonction `time_t time()`. Pour être complet, précisons que nous ne prenons que les 32 premiers bits de la valeur retournée (initialement 64). Ainsi, nous pouvons facilement voir quels paquets doivent être retransmis pour cause de *timeout*. Si `(uint32_t) time(NULL) - pkt_get_timestamp(paquet) >= RETRANSMISSION_TIMER` alors nous renvoyons le paquet, en mettant évident à jour son *timestamp*. Comme valeur de `RETRANSMISSION_TIMER`, nous avons décidé de garder une valeur fixe de 2 secondes, qui est la latence maximale de transmission d'un paquet. Même si cela peut diminuer les performances (nous aurions pu faire une "moyenne" du temps de transmission), nous préférons miser sur la fiabilité du programme en prenant la valeur maximale de latence.

3 Réception d'un paquet NACK

Lorsque le `sender` reçoit un paquet de type `PTYPE_NACK`, celui-ci va simplement ignorer le paquet et libérer la mémoire associée à ce paquet. Ce paquet sera retransmis lorsqu'il sera en situation de *timeout*. Néanmoins, dans la réception de ce type de paquet, le `sender` apprend que la taille de la *window* du `receiver` a été divisée par deux. En effet, lorsque ce dernier reçoit un paquet de type tronqué, nous supposons que le réseau est quelque peu surchargé. Pour diminuer le trafic et soulager le réseau, le `receiver` va donc diviser la taille de sa *window* par deux, et cette valeur sera transmise au `sender` dans le paquet de type `PTYPE_NACK`.

4 Performances du protocole

Notre protocole est fortement ralentit lorsqu'un paquet se perd. En effet, le RETRANSMISSION_TIMER de notre programme est fixé à 2 secondes. La vitesse de transfert dépend donc directement de la probabilité qu'a un paquet d'atteindre son objectif. Pour un paquet qui ne se perd pas en chemin, l'envoi et la réception s'effectuent de façon quasi instantanée.

En terme de performances, nous pouvons assez facilement déduire que le programme a une complexité de $\Theta(n)$. En effet, doubler la taille de la donnée à envoyer va doubler le temps d'envoi. Ci-dessous un graphe reprenant les différents tests effectués, la probabilité de perte d'un paquet par le réseau a été fixée à 10% pour ces tests. Pour être plus précis, nous utilisons le programme `linkesim` avec les paramètres suivants: `-l 20 -d 30 -j 30 -c 05 -e 10 -R`. Nous remarquons directement le caractère linéaire de la complexité ($\Theta(n)$). Nous remarquons également une forte divergence de certains points par rapport à la droite de régression

linéaire, c'est encore une fois lié au fait que notre protocole est très dépendant du nombre de paquets perdus lors du transfert.

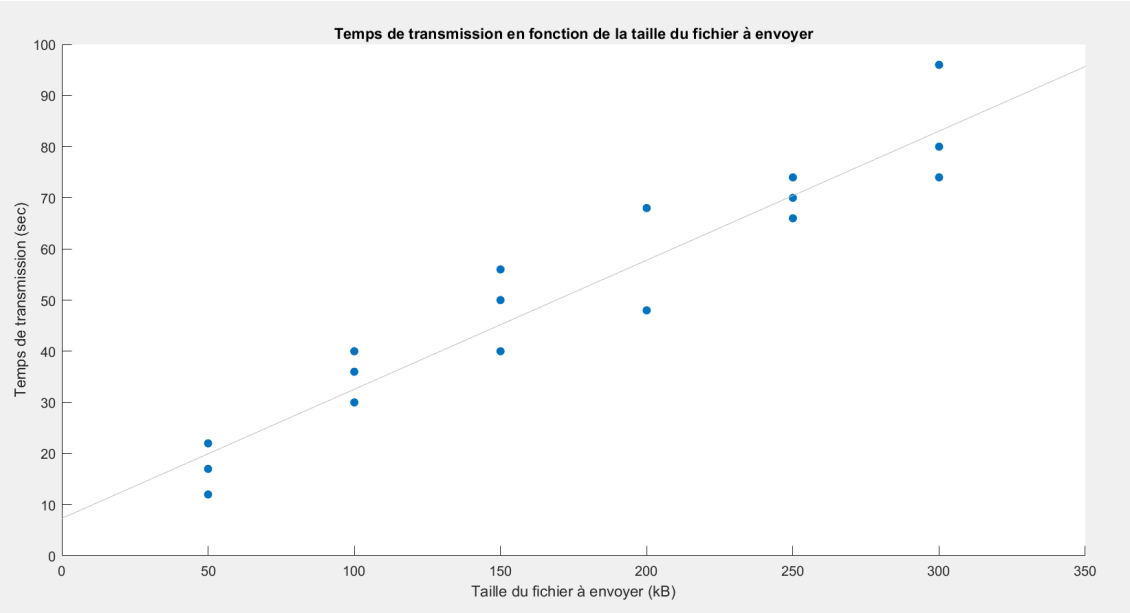

Figure 1: Performances du protocole

5 Implémentation du sender

La figure 2 montre le fonctionnement de notre `sender`. Tous les noms de fonction repris dans cette figure sont les fonctions visibles dans notre code source.

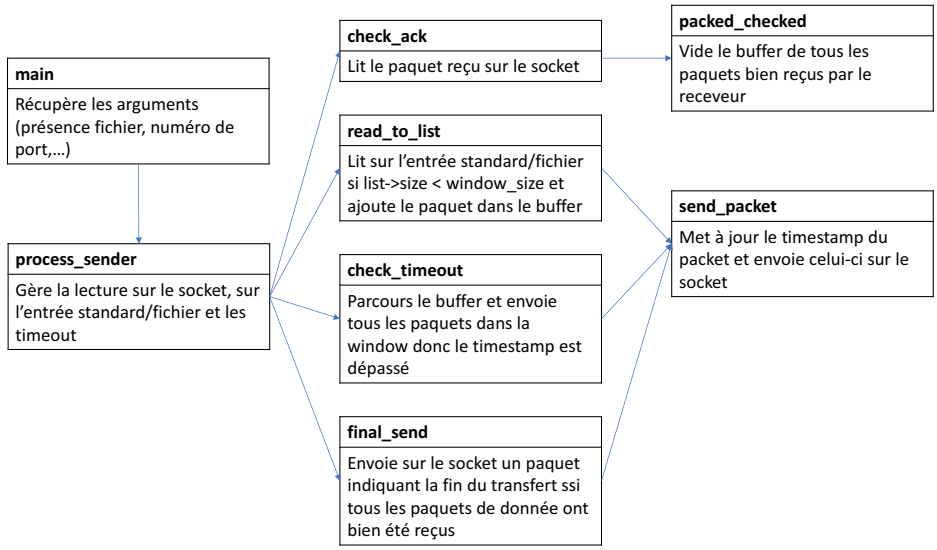

Figure 2: Déroulement du `sender`

6 Implémentation du receiver

La figure 3 montre le fonctionnement de notre `receiver`. Tous les noms de fonction repris dans cette figure sont les fonctions visibles dans notre code source.

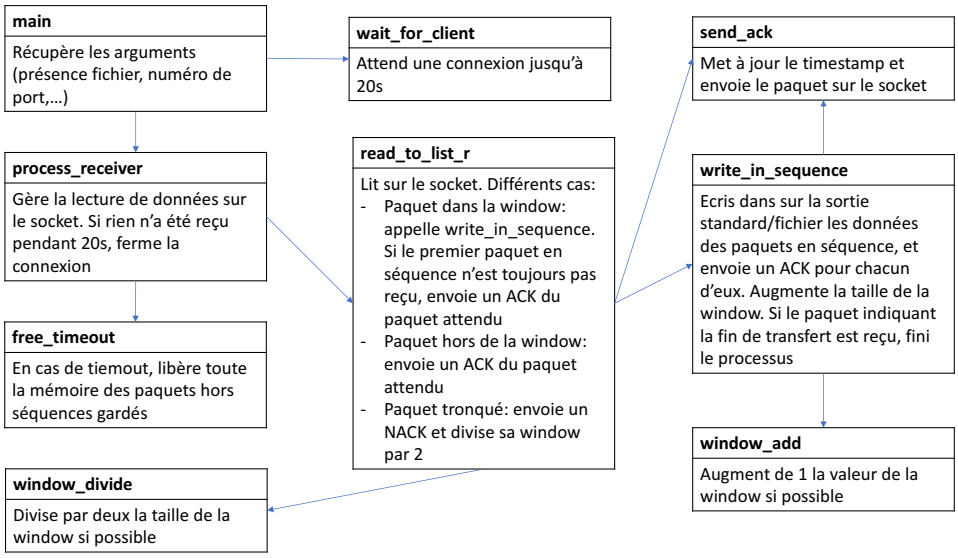

Figure 3: Déroulement du `receiver`

7 Tests effectués

7.1 Tests unitaires

Pour tester les fonctions de gestion des paquets (fonctions de `packet_implement.c`) ainsi que les fonctions relatives aux listes chaînées (`nyancat.c`), nous avons écrit des tests unitaires en utilisant CUNIT. Il ne nous a pas semblé utile de tester d'autres fonctions à l'aide de CUNIT car un test fourni abordé au point suivant permet de le faire.

7.2 Tests fournis

Notre programme a également été testé avec le script de test disponible sur Moodle (*linkesim*. Nous avons donc pu tester notre protocole avec des pertes, modifications, troncations, délais de nos paquets. Une autre partie critique du programme est le passage des numéros de séquence de 255 à 0. Pour pouvoir vérifier que notre implémentation fonctionne, nous avons décidé de transférer des fichiers d'environ 150kB, pour être sûr que nous effectuons ce passage. En effet, puisque la taille maximale du *payload* d'un paquet est de 512 bytes, il y aura environ 290 numéros de séquences utilisés; si les tests fonctionnent pour un fichier de cette taille, c'est que nous gérons correctement ce passage délicat. Ainsi, un test que nous effectuions était le suivant: `-l 20 -d 30 -j 30 -c 05 -e 10 -R`.

7.3 Tests de mémoire

L'utilisation de *valgrind* indique que notre implémentation ne comporte aucune fuite de mémoire lors des majeurs tests effectués. Ainsi, lorsque tout se passe correctement (dans le sens où le `sender` ou `receiver` ne se fait pas arrêter manuellement), nous n'avons détecté aucune fuite de mémoire, même losque le programme doit traiter avec des pertes/corruptions/délais des paquets par le réseau.

7.4 Test d'erreurs

En plus de *valgrind* pour examiner les différentes erreurs d'exécution (par exemple, *double free*), nous avons testé tous nos fichiers d'extension `.h` et `.c` avec `cppcheck`, et nous n'avons détecté aucune erreur.

Résultats. Pour valider les résultats, nous nous attendions à, au minimum, 10 succès consécutifs. Ces tests nous ont permis de trouver l'un ou l'autre bug, mais, à la fin, nous obtenions 100% de réussite.

8 Retour sur l'interopérabilité

Les tests d'interopérabilité nous ont permis de trouver trois erreurs majeurs dans notre implémentation. Précisons que nous avons tout d'abord effectué ces tests dans des conditions parfaites (aucune perte,...), puis en utilisant le programme `linkesim`. La première concernait la gestion de la fin de transmission. En effet, dû à une mauvaise compréhension du cahier des charges, le numéro de séquence du paquet indiquant la fin de transmission ne portait pas le bon numéro de séquence, ce qui empêchait une bonne fermeture de notre `sender` et de notre `receiver`. Un deuxième bug trouvé fut notre mauvaise gestion du passage des numéros de séquence de 255 à 0. Par exemple, lorsque notre buffer contenait des paquets de donnée de numéro de séquence dans $[0, 31]$, et que nous recevions un ACK de 253, tous ces paquets de donnée étaient considérés comme bien reçus; cela posait évidemment un grand problème lorsqu'il y a des pertes de paquets. La dernière erreur était que, lors de l'envoie d'un paquet de type `PTYPE_ACK` ou `PTYPE_NACK` par le `receiver`, nous mettions comme valeur de *timestamp* le temps où ce paquet à été envoyé. Or, nous devions, comme indiqué dans le cahier des charges, mettre comme *timestamp* la valeur du *timestamp* du dernier paquet de type `PTYPE_DATA` reçu.

9 Autres améliorations par rapport à la première soumission

Outre les différentes modifications apportées pour corriger les bugs répertoriés précédemment dans ce rapport, nous avons aussi effectué quelques modifications/améliorations à notre programme, même si, précisons-le, elles ne furent pas nombreuses. La plus importante d'entre elles fut la gestion de la fenêtre de réception du `sender` et du `receiver`. Pour le premier, nous avons choisi une taille de fenêtre de réception fixe et égale à 0, puisque le `sender` n'a pas de fenêtre de réception: il traite directement les paquets lus sur le *socket*. Pour le second, nous partons d'une taille égale à 1. Ensuite, pour chaque paquet bien reçu en séquence, nous agrandissons la fenêtre de réception de celui-ci d'une unité (macro `WINDOW_ADD`) si possible. Pour chaque paquet de données reçu étant tronqué par le réseau, comme mentionné dans la section 3, nous divisons la taille de la fenêtre de réception par deux (macro `WINDOW_DIVIDE` si celle-ci est supérieure à 1).

Par rapport à la première soumission, nous avons décidé de ne finalement pas changer notre `RETRANSMISSION_TIMER`, et de garde celui-ci constant et égal à 2sec. Changer cela nous aurait amené à changer grandement notre implémentation sans apporter assez de valeur à nos yeux, mis à part peut-être au niveau des performances. Nous avons aussi décidé de garder notre manière de finir nos programmes `sender` et `receiver`. En effet, lors des tests d'interopérabilité, nous avons remarqué que cette manière de procéder est efficace et fonctionnait dans tous les cas.

10 Conclusion

Même si certaines choses ne sont pas optimisées, comme la gestion de notre `RETRANSMISSION_TIMER` qui reste constant, nous estimons que notre implémentation respecte le cahier des charges dans sa totalité, et que notre protocole fonctionne comme demandé. Ce projet nous a permis de développer des compétences plus pratiques par rapport aux réseaux informatiques, et, même si ce projet n'est pas parfait, nous pensons proposer un programme efficace, certes peut-être pas optimal au niveau des performances, mais sûr et fiable.