LINGI1341 : Projet TRTP

Receiver Groupe 64

Oliva Justin 33011600 Vlaicu Patriciu Vasile 32341700

Octobre 2019

1 Organisation du programme

Le programme est décomposé en plusieurs parties. Le fichier principal est receiver, il est chargé de récupérer les arguments et de lancer la main. Le fichier buffer contient toutes les fonctions nécessaires à la gestion du buffer. Ce dernier utilise le fichier packet_implem qui gère les packets. Enfin le fichier functions contient des functions utilitaires utilisées dans receiver.

2 Structures de données

Une première structure de données est celle représentant les paquets(pkt_t). En effet les paquets sont des chaînes de bits non distinctes ce qui n'est pas pratique, cette structure permet donc à la réception d'un paquet de le décoder et d'assigner une valeur à tous les champs présents dans le protocol(type, tr, window etc...). L'encodage d'un paquet à partir de la structure vers une chaîne de bits est également possible.

Une seconde structure fdItemList est une liste chaînée de fdItem. Elle sert principalement à différencier les senders lors des connexions simultanées mais également à maintenir à jour l'état d'envoi de leur fichier. Un fdItem contient une struct sockadd_in6 *sender qui permet de stocker l'adresse ipv6 d'un sender ainsi qu'un int fd qui est un filedescriptor associé à ce sender. On stocke également le dernier seqnum reçu(lastseqnum), le dernier seqnum acquitté(lastack) et le dernier seqnum écrit(writtenseqnum).

Enfin la structure bufferItemList qui est une liste chaînée de bufferItem. C'est un buffer d'une taille qui vaut $30 * senderlimit$ qui permet de stocker tous les paquets en attente d'écriture. Un bufferItem mémorise simplement un paquet(packet) ainsi que le filedrescriptor(fd) du fichier dans lequel il doit être écrit. Chaque filedrescriptor correspond à un sender donc tous les paquets sont écrits au bon endroit.

3 Fonctionnement du programme

Au lancement du programme, les arguments vont d'abord être récupérés pour assigner la senderlimit et le format des noms de fichier. Le socket UDP d'écoute est créé, le buffer est initialisé, on entame ensuite une boucle infinie d'écoute sur le socket listener.

Lors de la réception d'un paquet la fonction receive_packet est appelée, elle ajoute le sender à la liste si il n'y est pas déjà et ajoute le paquet au buffer si ce n'est pas un doublon. L'acquittement du plus grand seqnum reçu en séquence est ensuite effectué. Lorsque nous reçevons le dernier paquet d'un sender, ce dernier est enlevé de la liste et l'écriture de son fichier se termine. L'envoi du fichier est alors terminé.

Si nous ne souhaitont plus recevoir de fichiers, il suffit simplement d'effectuer un ctrl+c, le receiver va alors être fermé et le buffer libéré. De cette manière notre implémentation ne comprend aucune fuite de mémoire.

4 Réponses aux questions

4.1 Comment maintenez-vous l'état des différentes connections concurrentes ?

Grâce à la liste chainée fdItemList nous pouvons différencier chaque connexion et les gérer indépendamment. Une trace des seqnums reçus est maintenue pour chacune d'entres-elles.

4.2 Quelle est votre technique pour traiter les connexions concurrentes ?

L'appel système poll() est utilisé pour gérer toutes les connexions sur le socket listener. Il écoute en permanence le socket et timeout toutes les 1 microsecondes. De cette manière si nous reçevons des paquets provenant de n'importe quel sender ceux-ci vont toujours être ajoutés au buffer. La seule condition est que le buffer dispose de place, si il est rempli alors on va d'abord effectuer une écriture pour libérer de la place. Lorsque poll() timeout ce temps d'attente est utilisé pour écrire les paquets reçus. Cette stratégie fonctionne du moment que le receiver est assez rapide pour gérer toutes les connexions.

4.3 Comment avez-vous implémenté le mécanisme de fenêtre de réception ?

Le programme fait en sorte que la fenêtre de reception soit toujours comprise entre 0 et 31. Lors de l'aquittement la fênetre envoyée vaut $31 - ((buffersize - 1)/senderlimit)$ ainsi on n'excède jamais la fenêtre de réception.

4.4 Quel est votre stratégie pour la génération des acquittements ?

La génération d'acquittement dépend du seqnum du packet reçu. Lorsqu'il n'y a pas de perte et que les paquets sont reçus en séquence la fonction send_ack() va simplement acquitter le $seqnum + 1$. Si il y a une perte alors l'écriture va être bloquée jusqu'au packet perdu, on va accepter les paquets suivants et acquitter le paquet précédent le paquet perdu à chaque réception. Une fois le paquet perdu reçu, on va acquitter la plus grand séquence possible

présente dans le buffer avec ack_seq(), l'écriture sera alors débloquée pour cette séquence. On va itérer cette opération pour toutes les pertes jusqu'à ce qu'on reviennent à une reception en séquence.

4.5 Quelle est la partie critique de votre implémentation, affectant la vitesse de transfert ?

L'acquittement du 255ème paquet reçu force l'écriture de tous les autres paquets de ce sender présent dans le buffer afin de pouvoir réinitialiser les acquittements à partir de 0. Si on combine celà à des pertes alors le receiver peut rester bloquer un petit moment au 255ème seqnum ce qui peut affecter négativement la vitesse de transfert. De manière générale, la réception de paquets perdus nécessite des opérations supplémentaires ralentissant possiblement le transfert.

4.6 Comment gérer vous la fermeture de la connexion ?

Le sender est supprimé de notre liste lorqu'il nous a envoyé son dernier paquet qui contient donc un signalement eof.

4.7 Quelles sont les performances de votre implémentation ?

4.7.1 Temps d'exécution

Notre programme a des performances assez bonnes. Quoi qu'il arrive nous allons écrire dans un fichier que lorsque nous vidons notre buffer / dépassons le seqnum 255. Ce n'est clairement pas la méthode la plus efficace temporellement (il serait sûrement possible d'écrire les packets au fur et à mesure de leur réception), cependant elle facilite l'implémentation du code.

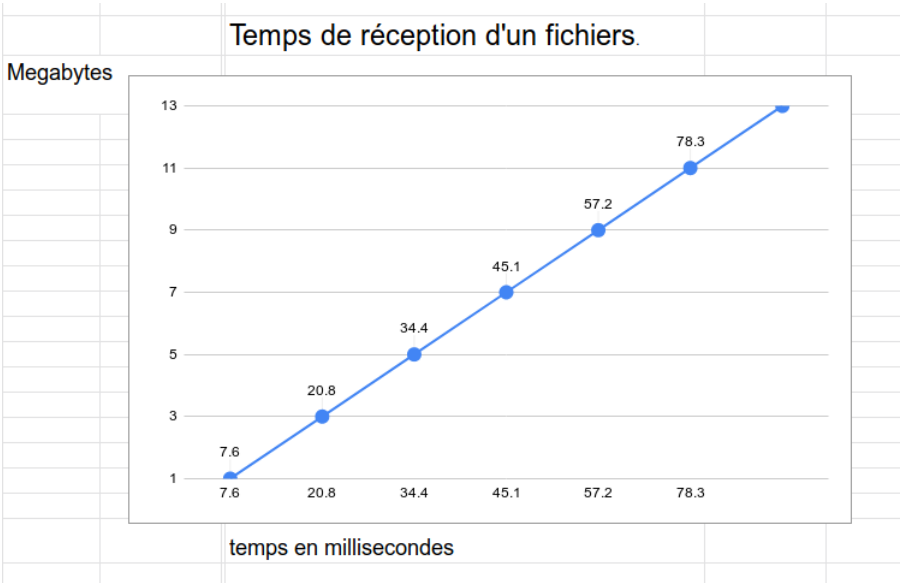

Figure 1: Temps(ms) de réception en fonction de la taille du fichier(mb)

4.7.2 Performances

Notre programme fonctionne sans faute sur un réseau en conditions parfaites, cependant il se peut qu'il rencontre des problèmes sur un réseau imparfait à cause du jitter / de délais trop varié entre l'envoi des packets lorsqu'on reçoit des plus gros fichiers (dépassent le megabyte par exemple).

4.8 Quelle(s) stratégie(s) de tests avez-vous utilisée(s) ?

Pour tester notre programme nous avons utilisé CUnit.
Nous avons décidé de tester notre structure bufferItemList contenant des bufferItem et la méthode receive_packet() qui reprend la majorité des méthodes de notre projet.

4.8.1 BufferItemList

Nous avons décidé dans un premier temps de tester notre structure bufferItemList contenant des bufferItem à l'aide des méthodes de test test_add_packet() ; et test_remove_packet() ;

Le test_add_packet vérifie que lors de l'ajout d'un nouvel élément bufferItem dans la liste chaînée celui-ci s'ajoute dans le bon ordre : premièrement en fonction du file decriptor et secondement en ordre croissant.

Le test_remove_packet vérifie simplement le bon fonctionnement de la fonction remove_packet.

4.8.2 receive_packet

La dernière méthode que nous testons est la méthode receive_packet() .
Cette méthode reprend la majorité des méthodes de notre projet. Si elle fonctionne correctement cela signifie que la réception et l'écriture des packets fonctionnent correctement.

Pour tester la méthode nous créons des packets avec différents numéros de séquence et nous simulons leur réception par la méthode receive_packet().
Cette opération nous permet aussi de vérifier le bon fonctionnement des méthodes d'encodage et décodage de packets.
Nous vérifions les valeurs de retour de modified_receive_packet pour savoir si elle traite correctement nos cas-limites tel que le dépassement du seqnum 255 mais aussi les cas classiques comme la réception d'un packet en séquence, la réception d'un paquet perdu et ainsi de suite.

4.9 Annexes

4.9.1 Tests d'interopérabilité

Nous avons effectué deux tests d'interopérabilité (groupe 93 et groupe 137).

Groupe 137: Le premier test a été effectué avec le groupe 137. Nous avons effectué un test simple d'envoi de fichier de petite taille (sans simulation de lien avec problèmes de connexions) car le sender du groupe pouvait possiblement se bloquer à cause d'un segfault si le fichier dépassait une certaine taille. Le test s'est déroulé correctement, le fichier envoyé par le sender était identique à celui écrit par le receiver (sha512sum identique).

Groupe 93: Les tests effectués avec ce groupe ont été plus extensifs. Nous avons premièrement testé l'envoi d'une image 4k sans simulation de liens fautifs. L'image s'est correctement transférée sur notre machine.
En second lieu nous avons testé l'envoi d'un fichier vidéo au format .avi d'une durée d'environ 2 heures et d'une taille d'à peu près 1.5gb. La vidéo s'est correctement transférée.

Pour les tests avec des liens fautifs nous avons mis toutes les options à 1 (delay, jitter, error, truncation, etc.). C'est dans ce cas que nous avons rencontré des problèmes : Notre receiver semblait demander en boucle l'envoie d'un même paquet jusqu'au timeout. Les membres du groupe 93 nous ont aussi signalé que notre receiver envoyait des acknowledgement packets avec une window de 0. Nous avons donc remarqué que notre implémentation gérait mal les liens fautifs sur de longs fichiers.

Le problème a été réglé lorsque nous avons corrigé certains cas de notre méthode d'ajout d'éléments dans la bufferlist. Précédemment il était possible d'ajouter plusieurs fois le même paquet, ce qui provoquait des erreurs dans tout notre code.