LINFO1341 - Projet 1 :
Truncated Reliable Transport Protocol

Charlotte Léonard, 67061700
Serena Lucca, 29541700

Mars 2021

Mécanisme de fenêtre de réception

La fenêtre de réception du `receiver` est implémentée par un buffer de pointeurs vers des paquets. Pour chaque paquet reçu, on vérifie si le numéro de séquence rentre dans la fenêtre des numéros de séquence autorisés. Si c'est le cas mais que le paquet n'est pas celui qu'on attendait, on crée un nouveau paquet à l'endroit du tableau où celui ci doit se trouver puis on recopie toute la structure du paquet reçu dans le nouveau paquet grâce à `memcpy()`. Etant donné que le paquet à été copié dans un nouveau pointer, on peut supprimer sans problème le paquet reçu juste après avoir envoyé l'acquittement.
Si le paquet reçu est déja dans la windows alors on passe juste à l'étape suivante (envoyer un acquittement).
Lorque l'on reçoit enfin le paquet avec le numéro de séquence attendu, on imprime les paquets contenus dans la window dans l'ordre jusqu'à tomber sur un élément `NULL`. Cet élément correspond à l'emplacement du prochain paquet attendu. Il suffit maintenant de mettre à jour la window en ramenant tout les paquets restant dans la window (ceux qui n'ont pas été imprimés) vers le début de la window en recréant chaque paquet avec `memcpy()` à son nouvel index. Chaque paquet qui a été imprimé ou qui a été déplacé vers une nouvelle case doit être supprimé et remis à `NULL` afin de pouvoir continuer à remplir correctement le buffer.

Mécanisme de fenêtre d'envoi

Le `sender` lit les données qu'il doit envoyer tant qu'il a de la place dans sa window. Il ajoute les paquets au fur et a mesure dans sa window et les garde tant qu'ils n'ont pas été acquittés. Lorsque le `sender` reçoit un acquittement, il retire les éléments de la window jusqu'à l'élément corespondant au numéro de séquence de l'acquittement exclu. La structure de la window du `sender` est une structure similaire a une liste chainée.

Génération des acquittements

Chaque paquet reçu par le `receiver`, si il est valable (non corrompu) et que son numéro de séquence rentre dans la fenêtre des numéros de séquence autorisés, génère un acquittement. Si le paquet est tronqué, on envoie un paquet de type `PTYPE_NACK` sinon on envoie un paquet de type `PTYPE_ACK` avec le numéro de séquence égal au numéro de séquence du paquet attendu.

Fermeture de la connexion

Le `sender` sait qu'il a envoyé tout les paquets lorsqu'il arrive au bout du fichier qu'on lui donne en argument ou qu'on le lui indique par l'entrée standard. Une fois que tout les paquets on été envoyé le `sender` envoie un dernier paquet de taille 0 et le met ensuite dans sa window. Il termine un fois que ce dernier paquet est acquitté. Pour ce qui est du `receiver`, le transfert est terminé uniquement après bonne réception d'un paquet de taille 0 avec son numéro de séquence égal au numéro de séquence attendu.

Champ Timestamp

Le champ timestamp nous permet de calculer la durée des round trip times des paquets dans le réseau. Le `sender` encode dans le champ timestamp le moment auquel le paquet est envoyé et en recevant l'acquittement il peut comparer le champ timestamp a sa clock pour en déduire le rtt de ce paquet.

Retransmission Timer

Le `sender` réinitialise la valeur du timer de retransmission après chaque envoi d'un paquet `PTYPE_DATA` et retransmet le dernier paquet de sa window uniquement si le dernier paquet envoyé n'a pas été acquitté et si aucun paquet n'a été envoyé pendant 0.2 seconde. Cette valeur est approximativement aléatoire, nous l'avons choisie par essais erreurs et elle a bien fonctionné lors de nos tests, elle permet de ne pas retransmettre trop vite et de ne pas trop ralentir le programme.

Réception d'acquittement de paquets tronqués

Lorsque le `sender` reçois un paquet de type `PTYPE_NACK` dont le numéro de séquence correspond au numéro de séquence d'un paquet de la window d'envoi, le `sender` renvoie ce paquet et le garde dans sa window tant qu'il n'est pas acquitté.

Receiver injoignable

Nous n'avons pas eu le temps d'implementer une stratégie gérant le cas ou le `receiver` ne peut traiter l'ouverture de connexion. Pour l'instant le sender va simplement envoyer ses paquets et retransmettre indéfiniment en l'attente d'une réponse. Pour contrer ce problème, nous pourrions faire en sorte que le `sender` attende une seconde après l'envoi du premier paquet puis si aucune réponse n'est reçue durant cette seconde il recommence en doublant le temps d'attente qui est donc à 2 secondes maintenant et ainsi de suite jusqu'à disons 2 secondes d'attentes. A la fin de cette période d'attent, on pourrait déclarer que le `receiver` est injoignable et donc décider de fermer le sender. Si une réponse du `receiver` est obtenue dans le temps imparti alors on peut supposer que la connexion est bien établie et on continue avec le processus habituel.

Vitesse de transfert

Dans le cas d'un lien parfait, la vitesse d'exécution de notre implémentation en principalement liée au transfert en lui même, les fontions de notre code sont négligeables pas rapport au round trip time moyen. Cependant si le lien n'est pas parfait, la vitesse est ralentie par la retransmission qui attend l'expiration du timer. Le temps attendu par le timer étant plus élevé qu'un round trip time moyen ca impacte considérablement la vitesse de transfert.

Performances

Le premier test de performance est réalisé sur un réseau fiable. Sur la Figure 1 on peut voir le temps de transfert total en fonction de la taille de fichier avec des fichiers de taille exponentiellement croissante, de 50 kB a 26 MB. On peut observer sur la Figure 1 que, avec une échelle logarithmique, le temps d'envoi est linéaire, on peut donc conclure que le temps d'envoi est linéairement proportionel a la taille du fichier. On peut également observer que les deux premiers paquets ont le même temps de transfert, cela est du au fait qu'il font tout les deux moins de 512 octets et donc un seul paquet a été envoyé durant ces deux transferts, leurs temps de transfert sont donc logiquement approximativement les mêmes.
Sur la Figure 2, on peut observer le temps de transfert en fonction des pertes sur le réseau. On peut voir que plus les pertes augmentent plus le temps augmente et qu'il augmente en suivant un caractère exponentiel.
Le même résultat peut etre observé sur la Figure 3, l'augmentation de l'erreur ralentit aussi le transfert de manière

exponentielle.
Sur la Figure 4 on remarque que le temps de transmission reste plus ou moins dans le même ordre de grandeur. Il y a des variations mais rien de significatif. Les troncations n'influencent pas le temps de transmission.

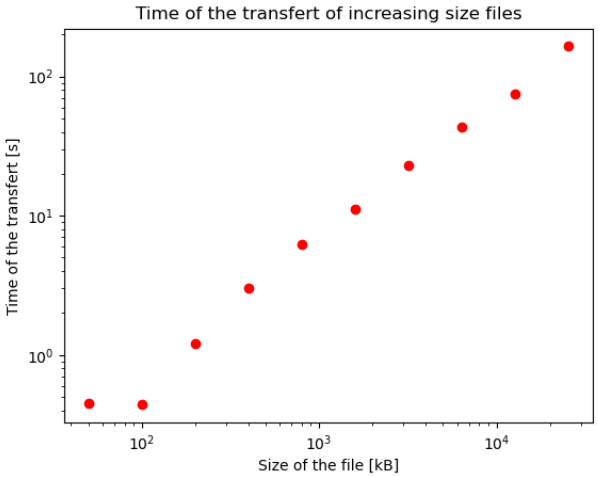

FIGURE 1 – Durée du transfert en fonction de la taille des fichiers

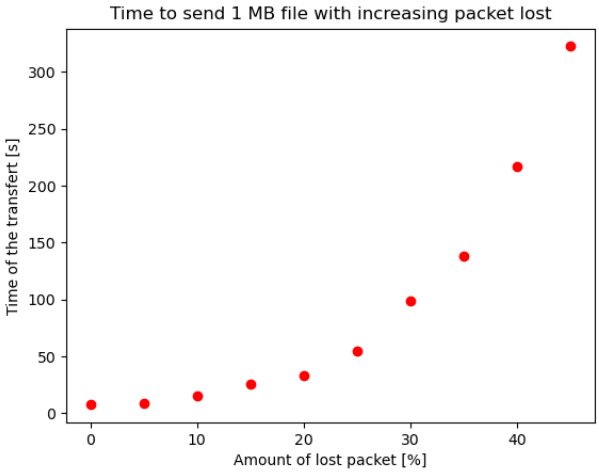

FIGURE 2 – Durée du transfert de 1 MB avec une augmentation de perte

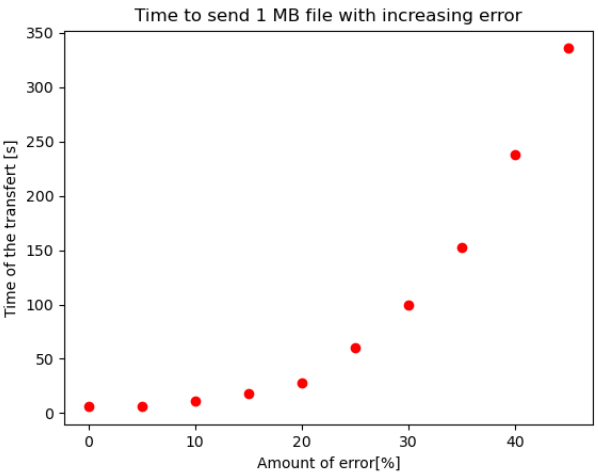

FIGURE 3 – Durée du transfert de 1 MB avec une augmentation d'erreur

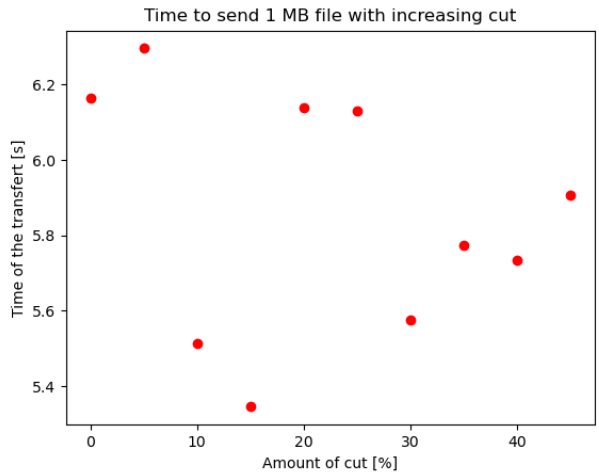

FIGURE 4 – Durée du transfert de 1 MB avec une augmentation de troncation

Tests

Pour réaliser notre suite de tests, nous nous sommes grandement inspirées du test fourni dans le squelette du projet ainsi que du test fourni sur le moodle du cours. Nos tests sont relativement simples, nous envoyons un fichier binaire relativement gros sur différents types de lien puis nous vérifions si le fichier reçu est le même que le fichier envoyé. Pour le premier test, notre fichier est envoyé sur un lien parfait, le deuxième sur un lien avec du delay et du jitter, le troisième sur un lien avec des pertes, des corruptions et des troncations de paquets et le dernier test reprend toutes les imperfections possibles sur le lien.

Chaque test crée un fichier dans le dossier `input`, un fichier dans le dossier `output` et 5 fichiers dans le dossier `logs_and_stats`. Ces 5 fichiers sont les logs du lien, du sender et du receiver ainsi que les stats du sender et du receiver. Le fichier le plus interessant selon nous est celui contenant les logs du lien. En effet, on peut y observer chaque paquet envoyé par le sender et par le receiver et l'ordre dans lequel tout se passe.
Notons que dans notre suite de tests les valeurs des paramètres du lien sont relativement basses (10% de pertes de corruptions et de troncations et 20ms de delay et de jitter) cependant nous avons testé notre implémentation avec des valeurs bien plus élevées (jusqu'à 50% et 100ms) et tout fonctionne toujours très correctement seulement tout prend énormément plus de temps (plusieurs dizaines de minutes) donc nous ne vous conseillons pas de modifier de manière exagérée les paramètres.
Nos tests peuvent prendre jusqu'à plus d'une minute à s'exécuter donc laissez leur le temps.

Pistes d'améliorations

Tout d'abord, nous ne prenons pas en charge les noms de domaines comme input mais uniquement les adresses IPv6 à cause d'un problème lors de l'implémentation.
Nous avons remarqué que notre sender envoie un grand nombre de paquet, cela est du au fait qu'il renvoie un paquet pour chaque ack valable reçu sans se poser plus de questions. Avec plus de temps, nous pourrions essayer d'optimiser cela notamment en utilisant mieux le champ timestamp. Cependant, cette façon de faire nous permet d'être très robuste par rapport aux autres receivers donc nous avons décidé de la laisser ainsi pour le moment.
Jusqu'à présent, nous n'avons testé nos implémentations que sur un lien parfait ou sur un lien imparfait dans le sens du sender vers le receiver. Nous n'avons pas eu suffisament de temps que pour tester en profondeur notre implémentation sur un lien imparfait du receiver vers le sender (ou dans les 2 sens en même temps). Théoriquement, notre sender étant assez robuste, il devrait pouvoir s'en sortir car il vérifie correctement la validité des ack reçus et il possède un timer de retransmission dans le cas où rien ne se passe pendant trop longtemps. Nous n'avons malheureusement pas pu vérifier cela en pratique.

Interopérabilité

Nous avons pu tester notre implémentation avec 3 groupes. Nous avons commencé avec le groupe 135, notre sender avec leur receiver fonctionnait très bien mis à part qu'ils mettaient le numéro de sequence du tout dernier paquet (de taille 0) au même numéro de séquence que l'avant dernier paquet envoyé donc la connexion ne terminait pas.
Lorsque nous avons essayé notre receiver avec leur sender on a remarqué que nous ne renvoyions pas le timestamp du paquet reçu donc ils plantaient dès le début, nous avons rapidement corrigé cela.
Avec le second groupe (150), nous avons remarqué que notre receiver essayait de vérifier si le deuxième crc32 du tout dernier paquet alors que celui ci n'en a même pas donc nous avons vite corrigé cela et le reste s'est très bien passé.
Nous avons eu exactement le même problème en testant avec le groupe 31 car nous nous étions séparées pour tester chacune notre implémentation avec un groupe différent, une fois les changements effectués tout s'est très bien passé.