Projet de réseaux informatiques : TRTP : Truncated Reliable Transport Protocol

Groupe 70 : Julien Launois, Hugo Fichère

Octobre 2019

1 Introduction

Ici est présenté notre implémentation du **receiver**.

2 Gestion des connexions concurrentes

Chaque connexion est gérée via une structure *Connection* qui stocke tout ce qui concerne un **sender** distant. Ceci comprend : l'adresse ipv6 du sender (**SOCKADDR_IN6**), le descripteur de fichier de sortie, un tableau de packets **pkt_t*** qui fait office de fênetre de reception, un gros buffer de plusieurs **Mo** ainsi que d'autres variables facilitant la gestion de la connexion tel qu'un entier étant toujours égal au numéro de séquence désiré.
Au lancement, le programme crée un tableau de *struct Connection* avec autant de *slots* que demandés en argument (100 par défaut). Après l'initialisation du **socket** d'écoute, le programme passe dans une boucle infinie qui, à chaque itération, attend un packet (grâce à la méthode **select**). Si celui-ci est prêt (et donc qu'il y a des données en attente). On appelle notre fonction **handle_reception()** qui s'occupera de recevoir, décoder le packet, et de l'attribuer à une *Connection* du tableau.
Si l'adresse de laquelle a été reçue le packet n'est pas parmi les adresses des *Connection*s déjà présentes, alors on alloue une nouvelle *Connection* dans le tableau. Enfin, on appelle la fonction **co_handle_new_pkt()** qui s'occupe de soit rejeter le packet, soit le mettre dans la fenêtre de réception (de la *Connection*) et d'envoyer un packet de retour **ack** adéquat.
Enfin, si on a toujours rien recu après 2 secondes, on envoie de nouveau le même packet. Après 10 secondes d'inactivité, on considère la connexion comme perdue et on la ferme.

3 Implémentation de la fenêtre de réception

La fenêtre de réception est, comme mentionné plus tôt, un attribut de la structure *Connection*. C'est est en réalité un tableau de **pkt_t***. Chaque emplace-

ment de ce tableau correspond à un **numéro de séquence** (*seqnum*) différent. Chaque packet complet et cohérent ayant un *seqnum* valide est donc copié à l'emplacement concerné dans ce tableau. Les packets se retrouve ainsi dans le bon ordre. S'il arrive qu'un packet soit invalide, tronqué ou perdu, on laisse un emplacement vide pour qu'il puisse s'y mettre par après. Si le prochain packet a le *seqnum* suivant, il sera mis à la position d'après dans le tableau. Lorsqu'on recevra le packet manquant on le stockera à l'emplacement laissé vide.
Si le dernier packet complète un trou, ou plus généralement si c'est celui qu'on attend, alors on copie la séquence entière de packets de la fenêtre dans le "gros buffer" de plusieurs mégabytes (ici 8 MB). Ceci fait, on supprime les packets de la fenêtre et on incrémente un index qui parcourt les emplacements de manière cyclique. Ainsi, le prochain packet sera stocké à la position de l'index dans la fenêtre et les suivants dans les cases adjacentes. Si cet index arrive à la taille de la fenêtre (31) il revient à 0. L'index pointe toujours un emplacement vide correspondant au plus petit *seqnum* que l'on a toujours pas reçu.
A la fin de la connexion ou lorsque le gros buffer est plein, on l'écrit dans le fichier que représente le descripteur de fichier de la *Connection* correspondante.

4 Les générations d'acquittements

À la fin du traitement de chaque packet reçu, on envoie un packet de retour avec le plus petit *seqnum* que l'on pas encore reçu. La taille de la fenêtre de réception envoyée sera toujours maximale (= 31) pour plus de rapidité. En effet, si tous les packets sont envoyés correctement on pourra toujours les gérés. Sinon, on continuera d'envoyer le *seqnum* manquant qui finira ainsi par être envoyé. Le packet sera un *nack* ou un *ack* en fonction du dernier reçu. Bien qu'en pratique, on pourrait toujours envoyer un *ack* vu qu'on ignore les packets tronqués comme s'ils étaient corrompus. Enfin, on envoie toujours le même *timestamp* que le dernier packet reçu du moins si il est consistant (c-à-d complet ou tronqué mais pas corrompu).

5 La partie critique du code

Les tests d'interopérabilité (avec le groupe 111) ont mis en évidence le fait que la transmission des fichiers était particulièrement longue pour des fichiers un peu plus volumineux (environ 10 minutes pour 6 MB = 10 kB/s). Il s'est avéré que cela venait de notre **receiver**. En discutant avec les membres de ce groupe, nous avons trouvé la source du problème. Initialement, dès que le packet désiré était reçu, on ouvrait le fichier, et on écrivait dedans. Cet appel système augmentait drastiquement la durée d'exécution du programme. Il suffisait de stocker plusieurs mégaoctets en mémoire vive pour les écrire dans les fichiers beaucoup plus rarement. De là découle le choix d'implémenter un gros buffer.
Au final, la seule partie vraiment critique reste probablement la copie de chaque *payload* dans ce buffer. En dehors, de ça le programme ne fait pas de calculs

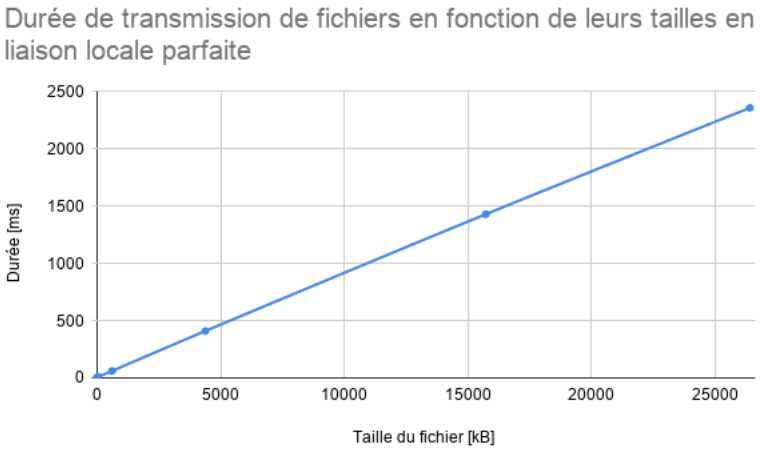
lourds. Les packets sont immédiatement et rapidement géré dès qu'ils sont reçus.

6 Fermeture de la connexion

Lorsqu'on reçoit un packet valide mais dont la longueur du *payload* est nulle, on considère que c'est la fin du message. Si il reste des packets non-reçus, on les demande via un **ack**. Dès qu'on les a tous reçus, on ferme la connexion en libérant la structure *Connection* correspondante dans le tableau.

7 Performances

En local, pour une connexion parfaite la vitesse est constante et s'approche des 10 MB/s. Le durée de transmission est donc proportionnelle à la taille du fichier reçu.



Grâce au programme **Linksimulator** nous pouvons simuler des connexions non-idéales. En ajoutant un taux d'erreurs de 5% la vitesse devient très variable, elle est divisée de 10 à 100 fois (jusqu'à 6 secondes de transmission pour un fichier de 605 kB contre 60 ms en liaison parfaite). Idem avec un taux de packets tronqués de 5% et pire encore quand les deux sont combinés. Ce phénomène est évidemment accentué lorsqu'on diminue encore artificellement la qualité des liens en augmentant ces pourcentages. Cette lenteur est cependant due au *sender* utilisé car il arrive qu'il attende que son *timeout* soit terminé pour envoyer de nouveau un packet. On observe pas cette diminution de performance pour le *sender* du groupe 19 par exemple.
On obtient des résultats similaires avec des pertes.
Si on ajoute un délai, la transmission est plus lente également, comme attendu.

On retiendra donc que même si le réseau est imparfait, le fichier arrive toujours en entier à destination au final tant que la connexion reste active.

8 Fuites mémoire et erreurs

L'utilisation de **valgrind** a mis en évidence un nombre important de fuites mémoires. Ces dernières étaient proportionnelles au nombre de packets reçus. Les packets stockés dans la structure *Connection* n'étaient pas bien libérés. Des *malloc* de **pkt_t*** étaient effectués à chaque réception sans qu'ils soient correctement libérés.
Toutes ces fuites ont été fixées, nous libérons à présent les packets uniquement en fin de connexion sans les réallouer à chaque nouveau packet mais plutôt en les modifiant.
Il arrive parfois cette erreur :
double free or corruption (!prev)
Aborted (core dumped).
Cependant, elle est si rare qu'il a été impossible d'identifier son origine avec certitude.

9 Fonctionnalités additionnelles

Le programme accepte deux arguments supplémentaires :
- **-p** : pour imprimer les logs de débogage, ces derniers apparaissent à chaque packet reçu et affectent donc la vitesse de réception. Certains logs sont cependant affichés (même sans l'argument) dans la sortie d'erreur standard pour indiqués les débuts et fins de connexions, ces derniers n'affectant pas la vitesse d'exécution globale.
- **-t** [secondes] : pour décider après combien de temps sans packet entrant le programme doit prendre fin. Sans cet argument le programme reste en vie indéfiniment et l'utilisateur doit le fermer manuellement à l'aide la commande **CTRL + V**

10 Nos tests

Via CUnit, nous testons quelques fonctions d'encodage et de décodage des packets. Puis nous exécutons le programme avec 4 fichiers de différentes tailles envoyés en même temps suivi d'un test utilisant **Linksimulator** avec 5% d'erreurs, de packets tronqués et de pertes ajouté à un délai de 100 ms pouvant lui même varier de 100 ms. (Ne pas oublier de taper **CTRL + V** lorsque ces tests sont terminés pour fermer **Linksimulator** et passer aux tests d'encodage CUnit.)

11 Annexe

11.1 Tests d'interopérabilité

Les tests d'interopérabilité furent effectués avec les groupes 111 et 19 (dans cet ordre). Ils ont été faits entre ordinateur de la salle Intel et également en wifi. Le wifi étant en moyenne plus lent et présentant quelques rares erreurs même sans le **Linksimulator**.

11.1.1 Groupe 111

Un des fichiers que ce *sender* nous a envoyé pesait environ 6 MB, ce qui a mis en évidence un bug lorsque la fenêtre arrivait au *seqnum* numéro 256. À partir de là, les packets était ignoré. Ce bug est maintenant fixé.
Il nous a aussi permis de nous rendre compte de la lenteur de notre programme due à l'ouverture intempestive du fichier comme mentionné plus haut. Et ainsi, la réception du fichier de 6 MB passa de plusieurs minutes à quelques secondes.

11.1.2 Groupe 19

Les tests effectués avec ce groupe furent plutôt concluants. Même lorsque le **Linksimulator** ajoutait des erreurs et des packets tronqués, il était plus rapide que le *sender* de référence. Excepté cependant, lorsqu'il y a un délai important ou des pertes car ce *sender* n'envoie qu'un packet à la fois peu importe la taille de fenêtre envoyée.
Ces tests ont permis de résoudre une **Segmentation fault** lorsque le premier packet était corrompu.