Rapport de projet 1 : Truncated Reliable Transport Protocol

Adrien Banse, Marine Branders

1 Introduction

Il nous a été demandé d'écrire un programme permettant de recevoir un fichier sous forme de paquets, eux-mêmes d'une certaine structure bien particulière. Notre programme doit fonctionner malgré une liste de malheurs qui peut arriver aux paquets tels que :

- Un délais constant d'envoi
- Un délais variable d'envoi
- La troncature des paquets (les paquets sont envoyés sous forme tronquée)
- La corruption de certains paquets.
- La perte de certains paquets

Nous pensons avoir atteint ces objectifs, ce rapport explique comment nous avons procédé.

2 Architecture générale

Le programme receiver prend en argument au minimum un nom de domaine ou une adresse IPv6 et un port correspondant à un numéro de port UDP. Il peut également prendre en argument les options d'un format de fichier et d'un nombre maximal de connexions simultanées avec des sources.

Lors du lancement du programme receiver, un appel à la fonction `create_socket_real_adress()` est effectué. Cette fonction retourne le file descriptor du socket sur lequel le receiver devra écouter les informations provenant des différentes sources.

Si aucun problème n'est survenu, un appel à la fonction principale `read_write_loop()` sera effectué. Cette fonction est une boucle infinie écoutant sur le socket et traitant les informations envoyées par de potentielles sources. Cette boucle ne s'arrêtera que si un problème survient. Dans ce cas-là, le programme receiver s'arrêtera.

L'explication du fonctionnement de la fonction `read_write_loop()` est détaillée dans la section des choix d'implémentation.

Notre programme se compose de trois librairies : la librairie `packet`, la librairie `packet_queue`, la librairie `socket`. La première gère encodage et le décodage des paquets transmis sur le réseau. La seconde correspond à l'implémentation d'une liste chainée pour gérer la fenêtre de réception d'un receiver en stockant les paquets par ordre de numéros de séquence. La dernière comporte toutes les autres fonctions nécessaires au receiver pour communiquer avec plusieurs sources.

3 Choix de conception/d'implémentation

3.1 Connexions simultanées

Lors du lancement du programme, un socket est créé afin d'écouter sur l'adresse donnée en argument ou toutes les adresses si " : :" est précisé comme adresse. Ce socket sert à envoyer et recevoir de l'information sur une port fourni en argument.

- **Structure** `src_queue_t`

Une fois le socket créé, la fonction `read_write_loop()` est lancée pour écouter dans une boucle infinie, l'information arrivant sur le socket. Une structure `src_queue_t` est initialisée au début de l'appel à cette fonction. Cette structure permettra de stocker les différentes sources occupées à envoyer de l'information au receiver sous forme de liste chainée, de contrôler le nombre de sources simultanées et de savoir quel numéro devra prendre le fichier de sortie lorsqu'une nouvelle connexion s'établira.

- **Structure** `source_t`

Lorsque de l'information est détectée sur le socket, un appel à la fonction `recvfrom()` est effectué afin de récupérer l'adresse de la source. Il s'agit alors de savoir si un échange d'informations était déjà en cours avec cette source. Pour cela, la fonction `get_source()` permet de trouver la structure `source_t` associée à cette adresse si elle déjà en train d'échanger de l'information avec le receiver. Cette structure contient un file descriptor du fichier de sortie dans lequel les paquets sont écrits, l'adresse de la source et une liste chainée servant à stocker les paquets reçus mais pas encore écrits dans le fichier (explications plus détaillées de son implémentation au point 3.2).

- **Nombre maximal de sources/ajout d'une source**

Si la source n'est pas encore répertoriée et que le numéro de séquence du paquet vaut 0, la source va être ajoutée à la liste des sources à condition que le nombre maximal de sources simultanées ne soit pas dépassé. Pour cela, la structure `src_queue_t` contient le nombre de sources connectées (qui est incrémenté à chaque nouvelle source) et le nombre maximal de sources simultanées.

- **Fermeture de connexion/suppression d'une source**

Lorsque le programme écrit dans un fichier le dernier paquet (c'est-à-dire avec un champ `length` à 0), la source associée à ce paquet et ce fichier est supprimée de la liste de sources connectées au receiver car le transfert de données est terminé. Si le paquet d'acquittement du paquet indiquant la fin du fichier est perdu ou corrompu, la source va nous renvoyer ce dernier paquet. La source va être détectée comme étant nouvelle mais le paquet reçu aura un champ `length` à 0. Dans ce cas précis, la source ne sera pas ajoutée et un paquet d'acquittement du paque reçu sera généré.

3.2 Gestion des paquets

- **Réception d'un paquet**

Lorsque de l'information est détectée sur le socket, un appel à la fonction `recvfrom()` est effectué comme expliqué précédemment et cette fonction stocke dans un buffer (`bufRead`) l'information transmise. Ce buffer est alors décodé à l'aide de la fonction `pkt_decode()`. Cette fonction indiquera si l'information reçue correspond bien au format exigé pour le transfert d'informations.

Si le paquet reçu n'est pas valide, soit la source n'était pas encore connue et le paquet va alors être ignoré, soit la source était déjà répertoriée et un paquet d'acquittement avec le prochain numéro de séquence va être généré.

- **Traitement d'un paquet valide**

Si la source du paquet échangeait déjà avec le receiver ou qu'elle a pu être ajoutée, le paquet va alors être traité à l'aide de la fonction `pkt_treat()`. Si le numéro de séquence du paquet reçu est dans la fenêtre de réception, celui-ci va être insérer dans la liste chainée des paquets à écrire. Sinon, le paquet sera être ignoré. Dans les 2 cas, un paquet d'acquittement avec le prochain numéro de séquence sera généré.

- **Structure** `pkt_queue_t`

La liste chainée des paquets reçus à stocker avant d'être écrits dans le fichier de sortie est implémentée via la structure `pkt_queue_t`. Celle-ci contient le premier noeud de la liste, le nombre d'éléments dans la liste, le prochain numéro de séquence attendu ainsi que le dernier timestamp reçu.

- **Fenêtre de réception**

Afin de déterminer si un paquet reçu est dans la fenêtre de réception, la fonction `pkt_treat()` va faire appel à la fonction `seqnumOK()`. Cette dernière va vérifier que la différence entre le seqnum reçu et le prochain seqnum attendu est positive et inférieure à la taille de la fenêtre, c'est-à-dire 31 moins le nombre de paquets stockés dans la liste chainée.

Remarque : Pour assurer la continuité entre les numéros de séquence lorsqu'on passe de 255 à 0, il suffit d'ajouter 256 aux numéros de séquence inférieurs à 32 lors du calcul expliqué précédemment.

La fonction renverra -1 si le paquet doit être ignoré et un entier positif correspondant à la différence calculée sinon.

- **Stockage d'un paquet**

Lorsqu'un paquet fait partie de la fenêtre de réception, celui-ci va être ajouté à la liste chainée des paquets à stocker. L'insertion du paquet se fera de manière à ce que les numéros de séquence soient dans l'ordre.

Remarque : Comme pour le calcul pour déterminer si un numéro de séquence fait partie de la fenêtre de récception, il faudra ajouter 256 aux numéros de séquence inférieurs à 32 pour s'assurer qu'ils soient insérés à la fin de la liste et non au début.

- **Mise à jour du prochain numéro de séquence attendu**

Lorsque le numéro de séquence d'un paquet reçu correspond au prochain numéro de séquence attendu d'une source, il faut mettre à jour ce prochain numéro de séquence attendu. Pour cela, la liste chainée est parcourue en incrémentant de un ce nombre tant que le numéro de séquence est déjà présent dans la liste et qu'il y a toujours des paquets dans la liste.

- **Génération des acquittements**

Chaque fois qu'un paquet est reçu, un paquet d'acquittement ou de non-acquittement doit être envoyé à la source, à condition que la connexion avec la source ait été établie. Pour cela, la fonction `pkt_ack()` n'a qu'à récupérer le dernier timestamp reçu et le prochain numéro de séquence attendu qui sont stockés dans la structure `pkt_queue_t` de la source.

- **Ecriture des paquets dans le fichier de sortie**

L'avantage de notre implémentation de la structure `pkt_queue_t` est que les paquets sont mis dans l'ordre à écrire dans le fichier. Dès lors, il suffit de regarder si le premier paquet de la liste chainée possède un numéro de séquence inférieure au prochain numéro de séquence attendu (en tenant toujours compte du passage de 255 à 0). Si c'est le cas, le paquet est retiré de la liste et son champ `Payload` est écrit dans le fichier associé à la source. L'opération est répétée jusqu'à ce qu'on ne puisse plus écrire de paquet.

4 Stratégie de tests

4.1 Tout au long du développement - réseau parfait

Durant le développement du programme, nous avons testé chaque implémentation de manière unitaire. Par exemple, en ce qui concerne les listes chaînées utilisées, nous avons à chaque fois testé les cas extrêmes qui pourraient faire bugger notre programme.

Pour écrire ces tests unitaires nous avons utilisé `CUnit`.

En ce qui concerne le programme de manière plus générale, nous avons surtout fonctionné avec des images, car cela nous a permis de vérifier en un clin d'oeil si nous étions dans la bonne voie.

4.2 Candidat solution - réseau non-parfait

Lorsque nous pensions être face à un candidat solution (c'est-à-dire qui fonctionne dans un réseau parfait pour tout type de fichiers), nous avons commencé à tester notre programme dans un réseau non-parfait.

Pour cela, nous avons pu compter sur le programme de simulation de lien `link_sim` (écrit par l'équipe didactique). Nous avons également pu compter sur l'implémentation de référence d'un `sender` (écrite également par l'équipe didactique). Nous avons dès lors écrit 6 scripts `bash` qui testent notre programme dans les cas suivants :

- Délais des paquets.
- Inversion des paquets. Un délais est effectué et celui-ci est variable, ce qui a comme conséquence que les paquets n'arrivent pas dans le bon ordre.
- Troncature des paquets.
- Corruption des paquets. Des bytes sont parfois corrompus au sein des paquets.
- Perte des paquets. Certains paquets sont perdus.

Dans ces scripts `bash`, nous comparons à chaque fois tous les bytes du fichier d'entrée et du fichier de sortie, afin de vérifier la fiabilité de notre programme.

5 Performances du programme

5.0.1 En réseau parfait

Nous ne parvenons pas à obtenir un graphe satisfaisant du temps d'exécution. En effet, il est logique que ces temps d'exécution dépendent de l'état du réseau.

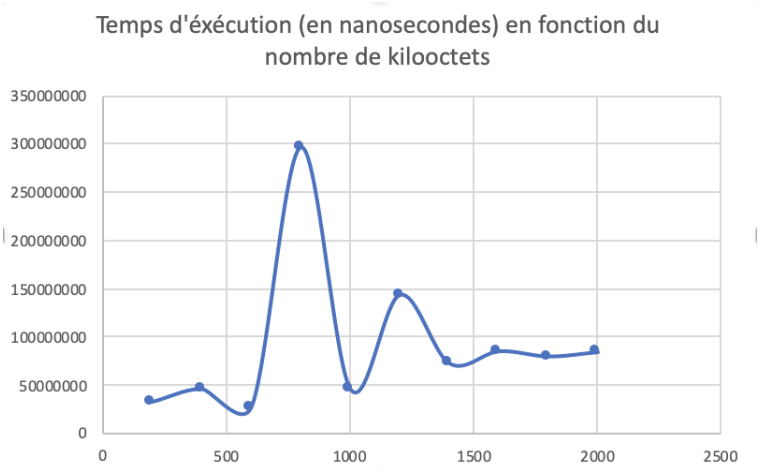

FIGURE 1 – Graphe du temps d'exécution

5.0.2 En réseau non-parfait

En réseau non-parfait, tout dépend des paquets que le simulateur de lien décide de tronquer, corrompre, ... Il est donc également impossible de sortir des graphes satisfaisants. Nous pouvons seulement affirmer dans le cas du délai que le programme est ralenti linéairement en fonction du nombre de paquets.

6 Conclusion

L'implémentation du receiver répond à tous les critères demandés. Elle sait gérer plusieurs connexions simultanées, utilise le selective repeat pour communiquer avec les sources. Elle peut également recevoir des informations de manière sécurisée sur un réseau non-parfait, c'est-à-dire avec des paquets perdus, tronqués, corrompus, retardés et dont l'ordre n'est pas assuré.

Certaines optimisations auraient pu être effectuées. Dans le cas où une source est inactive pendant trop longtemps, celle-ci pour être supprimée de la liste des sources enregistrées. Pour cela, il faudrait ajouter un timer dans la structure `source_t` correspondant au temps depuis le moment où le dernier paquet a été reçu de cette source. Il suffirait de vérifier que tous les timers sont inférieurs à une borne de temps maximale fixée avant chaque appel à la fonction `select()` et de remettre à zéro le timer d'une source chaque fois qu'un paquet est reçu de celle-ci.

7 Annexes

7.1 Résultats des tests d'interopérabilité

Les tests d'inter-opérabilité ont été effectués avec les groupes 13 et 101. Plusieurs petites modifications ont été apportées à notre implémentation suite à des erreurs remarquées lors de ces tests en réseau non-parfait.

Lorsque le réseau n'était pas parfait, c'est-à-dire que les paquets ne sont pas reçus dans l'ordre suite à des retards, des pertes ou des erreurs, les paquets étaient écrits dans l'ordre reçu et non dans l'ordre original.

Le problème venait de la fonction `canWrite()` de la librairie `socket` qui ne comparait pas correctment le premier numéro de séquence de la liste et le prochain numéro de séquence attendu.

Des problèmes de continuité des numéros de séquence lors du passage de 255 à 0 étaient également rencontrés en réseau non-parfait. Des modification s ont dues être apportées aux fonctions `add()` de la librairie `packet_queue` et `canWrite()` de la librairie `socket`.

7.2 Changements effectués

- **Ecriture des paquets dans le fichier de sortie**

Dans la première version de notre implémentation, au maximum un paquet était était écrit dans le fichier à chaque fois qu'un paquet était reçu. Désormais, tous les paquets pouvant être écrits sont écrits d'une seule fois pour éviter que les paquets s'accumulent et que la taille de la fenêtre de réception soit optimisée.

- **Fermeture de connexion en réseau non-parfait**

Lorsque le paquet reçu signifiant la fin du fichier est écrit dans le fichier de sortie, la source était supprimée de notre liste de sources. Si le paquet d'acquittement de ce dernier paquet est perdu ou corrompu, la source va renvoyer ce paquet. Dans notre première version, cette source était considérée comme une nouvelle source et un nouveau file descriptor lui était associé. Un nouveau fichier de 0 byte était donc créé. Désormais, avant d'ajouter une nouvelle source.