LINGI1341 - Computer networks : information transfer
Implémentation d'un protocole de transport sans pertes
Rapport

Legat Guillaume
Mottet Sébastien

October 2016

1 Architecture Générale

Afin de rendre notre projet le plus lisible possible nous avons décidé de séparer notre code en un fichier .c associé à un fichier .h pour chaque grande fonctionnalité du programme. Voici une brève description des différents fichiers, de leur rôle et de leur fonctionnement global. Nous décrirons l'implémentation du sender et du receiver plus bas dans ce rapport mais si vous désirez plus de détails sur l'implémentation et le fonctionnement des différentes fonctions, nous vous invitons à lire les spécifications et les commentaires présents dans le code source.

Connection_and_transfer.c

Ce fichier contient les différentes fonctions permettant la connexion entre les deux hôtes. La fonction *real_address* permet de convertir une String en une structure *sockaddr_in*6 utilisable pour bind et connect le socket. La fonction *create_socket* permet de bind et connect les deux hôtes et $wait\_for\_client$ attend le premier paquet pour finaliser la connexion en indiquant à l'hôte jouant le rôle de receiver de savoir d'où arrive les paquets. L'implémentation de ces fonctions est quasi identique que celle des fonctions remises sur Inginious.

Fonctions_communes.c

Ce fichier contient les fonctions inclassables communes au sender et au receiver. En l'occurence il n'y a qu'une fonction, *read_entries* qui traite les entrées de l'utilisateur afin d'en tirer les informations utiles.

Packet_implem.c

Ce fichier contient les différentes fonctions permettant de traiter les paquets. Les fonctions principales sont *pkt_encode* et *pkt_decode* qui permettent de passer d'une structure représentant un paquet à un tableau de char prêt à l'envoi et inversement. L'implémentation et les spécifications sont exactement les mêmes que celles se trouvant sur Inginious.

Sender.c

Ce fichier contient la fonction main du sender. Le rôle de la main est d'établir la connexion et de lancer le sender s'appuyant sur la stratégie du selective repeat. Nous commençons par utiliser la fonction *read_entries* afin de reccueillir les différents arguments entré par l'utilisateur. Nous établissons ensuite la connexion faisant appel à *create_socket*. Quand le lien est fait, nous appelons *selective_repeat_send* afin que l'échange commence.

Receiver.c

Nous agissons pareillement que dans *sender.c* à l'exception faites que nous appelons *selective_repeat_receive*.

Selective_repeat_sender.c

Ce fichier est, avec *Selective_repeat_receiver.c*, le coeur du projet. Il se charge de de découper le contenu à envoyer en paquet et s'assure de la bonne réception de ceux-ci par le receiver.
La fonction *selective_repeat_send* s'articule autour d'une boucle qui ne s'arrête que lorsque tout les paquets ont été envoyé et "acké". Grâce à la fonction select nous determinons quand est-ce qu'on peut lire ou écrire sur le socket.
Si on peut écrire sur le socket, on lit jusqu'à 512 bytes (taille max du payload) sur le fichier à transferer, on crée une structure paquet pour associer les informations utiles aux bytes lus (length,seqnum,...). Ensuite, à l'aide de la fonction pkt_encode on transforme la structure en un tableau de char que l'on envoie sur le réseau. Les paquets envoyés sont stockés dans un buffer en attente de leur acknowledgment. Une structure *time_val* placée dans un autre buffer leur est également associée afin d'enregistrer leur moment d'envoi
Si on peut lire sur le socket, on lit 12 bytes (soit la taille d'un ack). On transforme les données reçues en une structure paquet afin de pouvoir extraire le seqnum. Nous retirons ensuite les paquets du buffer ayant un seqnum parmis les x seqnum précédant le seqnum contenu dans l'ack (x est la taille de la fenêtre du sender. Celle-ci est déterminée lors de la réception du premier paquet en lisant la window de celui-ci). Les $time_val$ associées aux paquets retirés le sont aussi.
Régulièrement, les structures $time_val$ des paquets sont lues. Si un paquet est dans le buffer depuis plus de n ms, il est consideré comme perdu et envoyé à nouveau.
Lorsque la boucle s'est arreté, il reste à signaler au receiver la fin du transfert pour qu'il puisse s'arrêter. Le sender envoie donc un paquet de payload nul. Il attend un ack du receiver avant de se déconnecter. Si l'ack n'arrive pas avant n ms, le paquet de déconnexion est réenvoyé. Si la fonction *send* retourne -1, c'est que le receiver s'est déconnecté et que l'ack s'est perdu. Dans ce cas, le sender s'arrête également.

Selective_repeat_receiver.c

Ce fichier est, avec *Selective_repeat_sender.c*, le coeur du projet. Il se charge de réceptionner les paquets envoyés par le sender, de vérifier que leur contenu n'a pas été corrompu et de les écrire dans le bon ordre. La fonction s'articule également autour d'une boucle qui ne s'arrête que lorsqu'un paquet de taille nul (signal de fin de transmission) est reçu. Grâce à la fonction select nous determinons quand est-ce qu'on peut lire ou écrire sur le socket.
Si on peut à la fois écrire et lire sur le socket, on lit jusqu'à 524 bytes (taille max d'un paquet) et on utilise la fonctio *pkt_decode* afin d'obtenir une structure paquet. Si la fonction *pkt_decode* retourne E_CRC c'est que le paquet est corrompu. Dès lors, il est écarté. On regarde maintenant le seqnum du paquet. Si celui-ci est celui du paquet attendu, le payload est directement écrit sur la sortie et on écrit également tout les payload des paquets qui étaient stockés car arrivé dans le désordre et qui sont maintenant dans le bon order (ex : si 1 était attendu et que 2,3 et 4 étaient stockés. A l'arrivé de 1, les paquets 2,3 et 4 seront également écrits). Si le paquet reçu n'est pas celui attendu mais se trouve dans la fenêtre, il est stocké dans un buffer. Si le paquet reçu ne se trouve pas dans la fenêtre, il est écarté. Dans tout les cas, un acknowledgment signalant le paquet attendu est envoyé.
Si un paquet de taille nul est reçu, le receiver envoie un ack, sort de la boucle et s'arrête.

2 Champs Timestamp et son utilisation

Nous avons choisi de ne pas utiliser le timestamp parceque nous avons pu implémenter notre receiver et sender sans et surtout parceque ca pose de gros problèmes de portabilité. Effectivement, si les autres groupes utilisent le timestamp, leur façon de l'utiliser aurait sans doute été très différente de la nôtre et ce fait aurait très certainement impliqué de gros changements d'implémentation lors des tests de portabilité.

3 Choix de la valeur de retransmission timeout

La latence du réseau pour acheminer un paquet étant de maximum 2s, l'aller-retour paquet/ack est de maximum 4s. Nous considérons que l'envoi de l'ack après réception du paquet est quasiment instantané. Par conséquent, nous retransmettons les paquets lorsqu'ils sont depuis plus de 4s dans le buffer. Ce timeout ralentissant fortement les tests, libre à vous de modifier la valeur de $TIME\_OUT$ se trouvant dans *selective_repeat_sender.h*.

4 Partie critique affectant la vitesse de transfert

Aucune des fonctions utilisées est bloquantes et par conséquent aucune partie du code affecte très fortement la vitesse de transfert. Cependant, la partie du code du receiver qui cherche les paquets dans l'ordre dans le buffer et qui les écrit sur la sortie peut prendre un temps relativement important. Surtout si il y a beaucoup de paquet dans l'ordre. La partie du code du sender qui renvoie les paquets restés trop longtemps dans le buffer peut également prendre beaucoup de temps si il faut renvoyer beaucoup de paquets. Mais ces parties semblent inévitables.

5 Stratégie de tests

Nous avons établi cinq tests différents afin que nous puissions nous assurer que notre solution gère les difficultés qu'elle pourrait rencontrer, une par une, et ensuite toutes en même temps. Chaque test execute un script qui lance le sender, le receveir et link_sim. Lorsque leur exécution se termine, on compare le fichier envoyé et reçu pour voir si ils sont bien identiques.

- TEST 1 : lien fiable
  Nous commençons par tester notre protocole avec un lien fiable. Afin de s'assurer que tout fonctionne dans un cas idéal.
- TEST 2 : pertes
  Grace au simulateur de lien, nous avons lancer notre projet tout en simulant des défaults. Dans ce test ci, nous avons simuler une perte de 20% des paquets.
- TEST 3 : délais
  Dans ce troisième test, nous avons simulé un délais de 50 millisecondes avec un écart de 30 environ.
- TEST 4 : corruptions de données
  Ensuite nous avons donné à notre protocole un lien avec un taux d'erreur de 20%.
- TEST 5 : toutes les imperfections possible
  Pour finalement finir sur un test qui reprend toutes les imperfections en même temps et ce dans les deux sens de transmission possibles.
  Ces dernières sont donc :

  1. un délais de 50 millisecondes et un écart de 30 millisecondes
  2. un taux d'erreur de 15%
  3. un taux de perte de 15%
  4. et tout ça dans les deux sens

  Le résultat de nos tests est résumé sur la Figure 1, elle nous montre bien que notre protocole passe tous nos tests sans soucis et qu'il est fiable. De plus le fichier d'entrée (test_in.txt) peut être modifié afin de tester des fichiers de longueurs différentes.

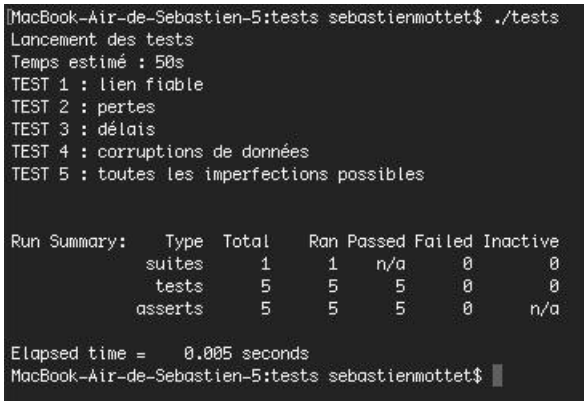

Figure 1: Résultat de notre série de test