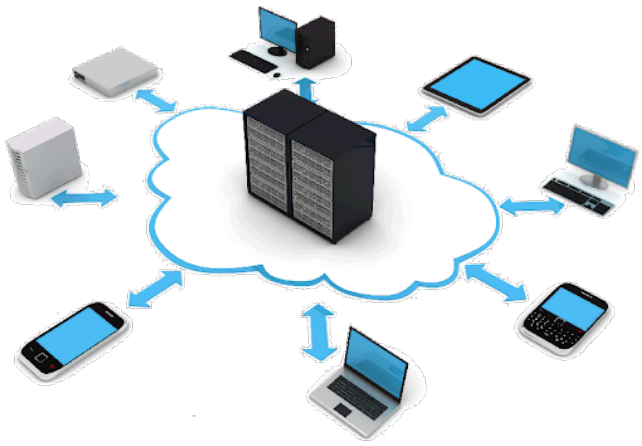
Université catholique de Louvain

LINFO1341

Computer networks : information transfer

Rapport du Projet 1 : TRTP



Andy Laurez NOMA : 1029-17-00
Maxime Jacques de Dixmude Slavic NOMA : 6205-15-00

Professeurs :
Bonaventure Olivier

27 mars 2021

UCLouvain

1 Introduction

Dans la cadre du cours LINFO1341, il nous a été demandé de réaliser un projet consistant à implémenter, en *C*, un *protocole* permettant le transfert de données avec *UDP*. Nous devons donc écrire un programme **Sender** et un **Receiver** sans fuite de mémoire et en utilisant la stratégie du *selective repeat*.

2 Informations Importantes

Vous prouvez trouver dans l'archive de notre projet deux versions du sender et du receiver. En effet, nous voulions apporter des changements afin d'amélioré notre programme avec la séance d'interopérabilité, cependant avec nos changements notre sender ne fonctionne plus et nous n'avons pas réussi à regler le problème avec la remise finale, nous avons donc mis les fichiers de notre deuxième version dans *senderV2.c* et *receiverV2.c*.

Les changements entre nos 2 versions sont surtout utiles pour la fenêtre de réception et la stratégie pour la génération des acquittements , c'est 2 points important seront donc détaillé en 2 sections dans ce rapport.

3 Architecture Générale

3.1 Sender

Le sender fonctionne à l'aide d'une structure *bufWindow* contenant :

- char* tableau[33] : un pointeur vers un tableau contenant les adresses des packets encodés.
- size_t tabOfSize[33] : un tableau de size_t qui permet, une fois associé au tableau précédent, de savoir la longueur du packet encodé.
- int next : afin d'optimiser un minimum la mémoire, nous avons choisi d'implémenter un tableau circulaire, d'où la présence d'un pointeur (une tête de lecture en quelque sorte) permettant de savoir où est le prochain élément à envoyer.
- int size : comme son nom le laisse penser, cette variable permet de connaitre le nombre d'élément que contient le tableau. Extrêment important car on ne nettoie pas les tableaux dés lors que les valeurs contenues ne sont plus utile.
- int sizeMax : on aurait pu se passer de cette variable mais pour notre facilité d'implémentation elle nous étais indispensable. Elle permet tout simplement de savoir combien d'élément les tableaux peuvent contenir au maximum.

Il est interressant de noter que nous aurions du implémenter ceci directement en variable global et non en structure pour la simple et bonne raison que ça diminue la lisibilité du code et ensuite car ce n'est pas nécessaire (on a pas de thread ou autre à gérer).

Maintenant, passons à l'architecture de la *main*. Tout d'abord, on gère les arguments grâce à *getopt()*. Ensuite, on se connecte avec le *receiver*. Un fois ceci fais, on rentre dans la première boucle *while()*. Dans cette dernière, on choisi l'activité grâce à *select()* qui nous permet de savoir s'il y a quelque chose à lire sur la socket. Si cette dernière est choisie on décode le packet reçu et 3 cas sont possibles,

le cas idéal : un packet valide qui a un numéro de séquence dans la porté de type ACK, on va donc acquitter les numéros de séquence et vérifier le RTT. Deuxième cas, packet valide de type NACK (expliqué en 3.6). Et enfin, le pire cas : un packet qui n'a pas de sens qui est donc tout simplement ignoré. Ensuite, on envoie les packets selon ce qui était demander au dernier ACK.

Une fois la première boucle finie on passe à la deuxième, celle-ci n'as qu'un seul but : gérer le dernier packet de longueur 0 pour marquer la fin du transfert.

3.2 Receiver

Notre receiver fonctionne avec un buffer fonctionnant avec une liste chainée qui sera toujours triée en fonction du prochain seqnum non-écrit par le receiver (Seqw), la liste sera toujours tirée comme ceci : [Seqw, Seqw+1, ..., 255, 0, ..., Seqw-1].

Une fois que le sender est connecté et envoie des paquets, le receiver recoit les paquets grâce à la fonction *select()*, il traite les paquets en vérifiant qu'ils sont bien encodé (CRC correct, etc) et va ensuite les ajouter dans son buffer si ils sont correctes et qu'il rentre dans la window (afin de gagner de l'espace de stockage nous stockons dans le buffer du receiver uniquement les champs Seqnum, Timestamp, Payload, Length et pas le paquet entier).

Tout cela se fait dans la fonction *receivedata()*, nous gérons à la fin de cette méthode les acquittement (Voir section stratégie d'ACK) et appelons la méthode *sendack()* pour envoyer un ACK.

Pour ajouter un paquet dans notre buffer nous avons la méthode *addbuffer()* qui vérifie si le seqnum est bien dans la window et si c'est le cas appel la méthode *pushorder()* pour que le packet soit bien ajouté dans le bonne ordre dans le buffer.

Le receiver appelle la méthode *writerec()* pour vider son buffer, cette méthode appelle la fonction *popqueue()* qui retire le premier élément du buffer si il correspond au seqnum attendu. La première fonction va écrire les paquets du buffer tant qu'il correspond au seqnum du paquet précédent + 1. La méthode retourne le seqnum du dernier packet écrit, qui nous sera utile pour envoyer le bon acquittement.

4 Choix De Conception

4.1 Fenêtre de réception

Dans notre **première version** la window du receiver est initilisée à 1 et nous comptions la modifier en fonction de la qualité du réseau.

Pour la **deuxième version**, nous avons choisi de mettre une fenêtre de réception de 31 au receiver (Valeur maximale autorisée pour la window). Cette valeur peut être facilement changée car elle est stockée dans la variable globale *wind* du receiver.

Nous pourrions d'ailleurs améliorer notre programme en imaginant de changer cette valeur en fonction du réseaux (surcharge, fiabilité, etc).

4.2 Stratégie pour la génération des acquittements

Notre statégie pour gérer les ACK est que nous attendons de recevoir tout les paquets que le sender nous envoie et ensuite on lui acquitte le seqnum du dernier paquet écrit + 1 quand il ne nous envoie plus rien ou si on a reçu le dernier pacquets avec un length de valeur 0 et biensur que tout les paquets précédents sont correctes.

Il est à noté que nous avions une autre stratègie au départ mais grâce à la session d'interopérabilité nous avons décidé d'opter pour cette autre stratégie. (Voir section Séance d'interopérabilité)

4.3 Fermeture de la connexion

Pour la fermeture de la connexion, le receiver recoit le paquet PTYPE DATA avec une length de 0, ce qui annonce la fin du transfert, il va donc acquitter ce paquet (seqnum + 1 comme pour tout les paquets PTYPE DATA) et pourra se déconnecter. Le sender, lui attend de recevoir l'acquitement pour ensuite terminer.

4.4 Champ Timestamp

Nous insérons dans le champ Timestamp de chaque paquet l'heure en nanoseconde avant d'envoyer le paquet (grâce à *clockgettime* et la structure *timespec*), nous pouvons grâce à cela savoir le **round-trip time** du paquet lorsque nous recevons l'aquis du paquet par le receiver.

4.5 Valeur du timer de retransmission

Nous avons choisi arbitrairement 3 fois la valeur du plus grand RTT.

4.6 Réception de paquets PTYPE NACK

Lorsqu'un packet valide de type NACK est receptionné par le sender, ce dernier va renvoyer un nombre de paquet égal au nombre indiqué dans le champ window du packet avec les numéros de séquences commencant par le dernier acquitté. Attention donc, le Nack n'acquitte pas !

4.7 Stratégie pour l'ouverture de la connexion

Pour l'ouverture de la connexion, nous avons une fonction *real_address* permettant de convertir une chaîne de caractères représentant soit un nom de domaine soit une adresse IPv6, en une structure *sockaddr_in6* et nous avons ensuite une fonction *create_socket* permettant de créer un socket, ainsi que de le lier à certains ports et adresses. Le receiver va donc attendre que le sender se connecte et lui envoie des paquets.

4.8 Partie critique de notre implémentation

Nous avons décidé de rendre dans les fichiers principales notre première version car nous nous sommes rendu compte que la deuxième version, bien que meilleur dans l'idée et dans les performances ne fonctionnait pas toujours et avait des erreurs qui survenait parfois aléatoirement (avec les mêmes tests).

4.9 Séance d'interopérabilité

La séance d'interopérabilité nous a été utile dans la réalisation de notre projet, surtout pour notre receiver. En effet, nous nous sommes rendu compte que notre stratégie d'acquittement utilisée lors de notre première soumission n'était pas optimale. Notre première statègie était que dés qu'on recevait un paquet correcte avec le bon seqnum attendu, nous envoyons un acquis. Dans un réseau parfait qui renvoie les paquets dans l'ordre c'est une perte de temps car on ne va jamais recevoir des paquets faux ou qui ne sont pas dans le bon ordre. Nous avons donc changer notre stratégie et utilisons celle décrite dans la section "3.2 Stratégie pour la génération des acquittements". En changeant notre stratégie nous nous sommes rendu compte que notre sender ne fonctionnait pas bien et n'était pas interopérable, nous avons donc fait de petites modifications dessus.

5 Performance et Tests

En ce qui concerne la performance, nous avons tout d'abord tester notre programme avec *valgrind* afin de vérifier les fuites de mémoires, après avoir corriger des erreurs évidentes, il nous reste de petites fuites que nous n'avons pas réussi à régler. A chaque fois que nous lancons notre programe, quel que soit la taille du fichier transféré, le receiver a à chaque test exactement 1 free en moins que son nombre d'allocs et le sender en a exactement 4 de moins.

Nous avons faits des tests avec des fichiers (présents dans le dossier *test/*) de tailles différentes et néccesittant donc un nombre de paquet à envoyer différents.

Nous comptons par la suite réaliser plus de tests automatisés, avec nottamment le simlink ainsi qu'un script python pour plot un graph qui mesure le temps en fonction de la taille du fichier à transférer.

6 Conclusion

Nous avons donc réalisé ce protocole TRTP en réspectant les consignes et les spécifications du protocole. Nous avons su l'améliorer au fur et à mesure des jours et avons pu changer certaines stratègies afin de l'optimiser. Nous n'avons cependant pas pu encore terminer notre deuxième version qui est bien meilleur dans l'idée et dans l'optimisation du protole même si il y a toujours des façons d'amélioré ce protocole et il ne sera jamais parfait.