



LINGI1341
Computer networks :
Truncated Reliable Transport Protocol

Edited by
Groupe 136
Simon Messens & Guillaume Legat
82751700 - 39751400
Université catholique de Louvain

October 31, 2019

Part I

1 Architecture générale

Tout programme un minimum volumineux se doit d'être segmenté en plusieurs fichiers *.c* (accompagné de son camarade le *.h*) afin d'améliorer la lisibilité et de structurer au mieux le projet. Nous allons donc commencer par décrire ces différents fichiers un par un.

packet_implem.c

Nous avons commencé le projet comme il nous l'était conseillé dans les consignes, par nous occuper de la tâche *Inginious : Format des segments du projet de groupe*. Nous commençons donc par créer une structure "pkt" dans laquelle nous traduirons une entrée binaire en variables qui lui seront assignées. Nous avons aussi la fonction *encode(...)* qui marche à l'inverse, à partir de variable nous reconstruisons une sortie binaire. Pendant le décodage nous effectuons aussi une série de petits tests afin de déterminer si le packet a été corrompu pendant le transfert.

read.c

Ensuite, nous nous sommes penchés sur la tâche *Envoyer et recevoir des données*. Dont nous avons pris trois méthodes pour les regrouper dans notre fichier *read.c*. Et la dernière fonction, *read_write_loop(...)* a été reprise en partie dans *selective.c*.

Nous avons donc, *real_address(...)* qui va nous permettre de prendre en argument une adresse sous forme de tableau de *char* et la mettre dans une *struct sockaddr_in6*, qui est un format utilisé ensuite pour la *socket* que l'on voudra créer.

create_socket(...) va prendre en argument l'adresse et le port de la source et du destinataire. Il va ensuite créer un *socket* pour ensuite lui *bind* la source si il y en a une et lui *connect* le destinataire si il y en a un.

Nous finissons *read.c* avec *wait_for_client(...)* qui va servir à connecter une *socket* qui n'avait pas de destinataire à sa création. Pour se faire, nous utilisons la méthode *recvfrom(...)* qui va attendre qu'un message soit envoyé à notre *socket*, pour ensuite récupérer les informations de l'expéditeur de ce message et finalement le connecter.

receiver.c

Le *receiver* est notre *main*, c'est par lui que tout commence, sa première partie est consacrée à la lecture des arguments, ceux-ci permettront à l'utilisateur de modifier le comportement du programme. Nous appelons ensuite les différents

méthodes de *read.c* décrites plus haut afin de créer un socket qui est la connection entre nous et le *sender*. Quand la connection est établie, il appellera *selective(...)* dans *selective.c* qui prendra en charge la suite des festivités. C'est avec le décriptage des arguments de notre *main* que nous savons si les messages d'erreurs de notre programmes devront être affiché sur la sortie d'erreur standrad ou écrit dans le fichier *log.txt*.

selective.c

La méthode principale de ce fichier est *selective(...)*, elle s'occupe de réceptionner les paquets du *sender* et de gérer quelle réaction il faut avoir ensuite.

Pour commencer, elle crée notre buffer qui aura comme rôle d'être notre window de réception. Nous entrons ensuite dans une grande boucle *while* dont on ne sortira que si la connection est finie et qui va lire et traiter un paquet à chaque passage.

Au début de la boucle, nous appelons *read_sock(...)* qui est assez semblable à la méthode *read_write_loop(...)* d'Inginious sans le concept d'écriture et de boucle. Elle s'occupe de lire sur la socket le paquet envoyé et de les stocker dans un buffer.

De retour dans *selective(...)*, nous décodons les datas reçus du buffer en paquet grace à la méthode *pkt_decode(...)* de *packet_implem.c*. Nous testons ensuite si il est bien de type DATA au sinon, nous l'ignorons. Nous vérifions aussi que sa length n'est pas nulle, auquel cas, cela voudrait dire que le sender veut se déconnecter, nous lui envoyons donc un ack confirmant la réception de sa déconnexion et nous sortons de la boucle while.

Nous envoyons nos paquets d'acknowledgment en utilisant notre méthode *send_ack(...)* qui va s'occuper de créer un nouveau *pkt_t* avec la taille de window qu'il nous reste, le bon type (*PTYPE_NACK* si *payload tronqué*, *PTYPE_ACK* sinon, une length nulle, le timestamp identique au timestamp du dernier packet reçu dans l'ordre et le seqnum correspondant à celui du prochain paquet qu'on aimerait recevoir. Ce paquet est ensuite encoder grace à *pkt_encode(...)* (décrit plus haut) et envoyé sur la socket pour le sender.

Ensuite, nous vérifions si le seqnum du packet reçu est celui attendu, dans ce cas, nous l'écrivons dans le fichier de sortie puis vérifions notre buffer si le paquet suivant n'y est pas stocké. Si c'est le cas, nous l'écrivons à son tour dans le fichier de sortie, libérons la place du buffer, incrémentons notre window et parcourons notre buffer à la recherche du suivant et ainsi de suite. Pour ensuite envoyer un ack par rapport au dernier paquet envoyé.

Si le paquet n'est pas dans l'ordre mais qu'il reste dans le spectre de notre window, il est alors stocké dans notre buffer pour plus tard, notre window est décrémentée et nous envoyons un ack identique au précédant.

Finalement, si il est en désordre et qu'il ne rentre même pas dans la window, nous envoyons un ack identique au précédant et nous ignorons ce paquet de données.

2 Gestion de la connections concurrentes

Nous n'avons pas eu le temps d'implémenter les connexions concurrentes mais voici comment nous avons penser la structurer. Tout d'abords un connexion externe engendre la creation d'un fichier de sortie dont le nom comporte un numero de série et l'incrémentation d'un counter. Ce counter servira de garder le compte des connexion concurentes. Une fois les sockets et les filenames créer nous utilisons un appel `select()` afin de lire en continu chaque socket. Le point positif de l'utilisation de `select()`, c'est que ce n'est pas une fonction bloquant. Cela nous permettrait de gérer plusieurs connexion en même temps sans pour autant gérer du multithreading. Le programme s'arreteera quand le counter retombera a 0.

3 Mecanisme de fenêtre de réception

Comme expliqué plus haut dans *selective.c*, notre fenêtre de réception est constituée de 2 parties : un *buffer* et une variable *window*. Le buffer est un tableau de *pkt_t* de 32 places qui accueillera les paquets dont le numéro de séquence n'est pas celui attendu mais qui reste dans le spectre de la window, c'est à dire qui est compris entre le numéro de séquence attendu et ce dernier + window.

La window represente le nombre de places dans notre buffer qui sont encore libres. Elle sera incrémentée à chaque fois que nous ajoutons un paquet dans le buffer et décrémentée quand nous en retirons un. C'est cette window que nous mettons dans nos acquittements comme demandé dans l'énoncé du projet.

Et pour finir, quand nous recevons un paquet dans l'ordre (avec le numéro de séquence que l'on attend), nous regardons si notre buffer ne contient pas le paquet suivant et ainsi de suite.

4 Génération des acquittements

Elle est gérée dans *selective.c* et plus précisément dans notre méthode *send_ack(...)* qui va commencer par créer un nouveau paquet vierge, lui attribuer le numéro de séquence du prochain paquet attendu, la window comme décrite ci-dessus au point précédent, le timestamp du dernier paquet écrit dans le fichier et le type *PTYPE_ACK* si le paquet reçu n'était pas tronqué ou *PTYPE_NACK* si il l'était. Nous pourrons ensuite l'encoder (grace à notre méthode *pkt.encode(...)*) et l'envoyer sur la socket vers le sender.

5 Gestion de la fermeture de la connexion

Nous comprenons que le sender désir se déconnecter quand il nous envoie un paquet de type *PTYPE_DATA* mais avec une length de valeur 0. Nous envoyons ensuite un ack correspondant à son paquet pour dire au sender qu'on a bien reçu

sa demande de déconnexion. Après ça, nous sortons de la grande boucle while (citée dans la partie *selective.c*), nous fermons la socket et les fichiers ouverts puis terminons le programme.

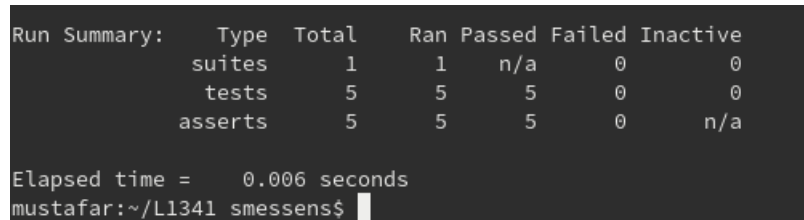
6 Stratégies de tests utilisées

Nous utilisons CUnit afin de tester notre programme. Ca va nous permettre de réaliser différents tests et de vérifier si on les passe bien. Nous avons également linksim qui nous permet de modifier la qualité du lien entre sender et receiver comme on le souhaite. Nous réalisons donc 5 tests :

- **Test 1** : c'est le test basique, nous utilisons un lien parfait.
- **Test 2** : c'est le test d'erreur, c'est cette fois si le simulateur de lien va simuler des corruptions dans les paquets de données, nous utilisons un taux de 30%.
- **Test 3** : C'est le test de délai, nous simulons un lien comportant un délai de 50 ms.
- **Test 4** : C'est le test des pertes, le lien simulé perdra des paquets envoyés à un taux de 30%.
- **Test 5** : Ce test combine toutes les difficultés précédentes. Le lien aura 15% de corruptions, 25 ms de délai et 15% de perte de paquets.

Nous vérifions la similitude entre le fichier qui est lu par le sender et celui écrit par le receiver grâce à la commande *sha512sum*.

Les résultats des tests sont décrits par la Figure 1. Elle montre bien que nous les passons tous sans soucis.



```
Run Summary:  Type  Total   Ran Passed Failed Inactive
                suites    1     1   n/a     0      0
                tests     5     5     5     0      0
                asserts    5     5     5     0     n/a

Elapsed time = 0.006 seconds
mustafar:~/L1341 smessens$
```

Figure 1: Capture d'écran des résultats des tests

7 Résultats d'interopérabilité

Malheureusement, nous n'avons pas eu le temps de réaliser des tests d'interopérabilité avec d'autres groupes. Malgré tout, grâce à la version de sender mise à disposition sur moodle, nous avons tout de même pu tester notre programme d'abord

en interne puis avec deux machines différentes en salle intell. Et durant ces échanges de données, notre implémentation fonctionnait bien et permettait une bonne réception des paquets envoyés par le sender même avec des liens intentionnellement endommagé par *linksim*.