LINGI1341

Rapport de projet1

Implémentation d'un protocole de transfert sans pertes

Momin Charles Iserentant Merlin

30 octobre 2015

1 Architecture générale

Nos deux programmes sont chacun composés de 3 fichiers :

— ***socket.h* (commun aux deux)**
Comprend l'ensemble des fonctions relatives à l'utilisation des sockets.
— ***packet.h* (commun aux deux)**
Comprend l'ensemble des fonctions relatives à la création et à l'utilisation des packets.
— ***sender.c***
Le programme de l'émetteur. Son rôle est de lire le fichier et de le transmettre via le socket.
— ***receiver.c***
Le programme de réception. Son rôle est de lire sur le socket afin de réceptionner le fichier.

Les deux programmes ont été réalisés afin d'implémenter le protocole de transfert sans pertes du *selective repeat*.

1.1 Le programme d'émission : sender.c

Le principe du *sender* est simple. Le programme boucle sur une lecture de STDIN suivie d'un envoi sur le socket des données si les conditions d'envoi sont satisfaites. Une analyse des arguments réalisée au préalable permet de savoir si STDIN doit être redirigée vers un fichier ou non. Le fait de lire sur STDIN nous a permis de ne faire qu'une fonction générale de lecture/envoi pour le sender, l'entrée standard pouvant être redirigée à notre guise. L'interception des acquis se réalise aussi dans la boucle d'envoi. Dans le cas d'une réception de packet sur le socket, le type de celui-ci est analysé et le programme réagit en conséquence. Dans notre cas, les types nous interressant étaient le *ACK* et *NACK*. Deux réactions sont alors réalisées :

— Packet de type **ACK**
Les différents itérateurs tels que la window sont mis à jours en fonction du numéro de séquence reçu pour l'acquittement. Les ressources occupées par les packets acquis sont libérées des buffers de packet (*window[]*) et de timer (*timer_buffer[]*).
— Packet de type **NACK**
Le programme va chercher le packet non-acquis dans son buffer et le retransmet.

Lors du premier envoi d'un packet, un timer lui est attribué [1]. À chaque début de boucle, les différents timers sont vérifiés afin de ne pas dépassé le délai de retransmission. Si cela venait à se produire, le packet est réenvoyé et son timer est remis à 0. De plus, dans le cas de l'envoi d'un fichier via les paramètres *-f/-filename*, un calcul de la longueur totale du fichier est réalisée avant de commencer le transfert. La longueur des données envoyées est enregistrée ce qui permet de savoir exactement quand doit s'arrêter la boucle. Une fois la fin de fichier atteinte, le programme envoi un packet de taille nulle et attend l'acquis de ce dernier.

1.2 Le programme de réception : receiver.c

Tout comme le *sender*, le principe du *receiver* est assez simple. Le programme boucle sur une lecture du socket suivie d'une écriture sur STDOUT des données ainsi récupérées. Une fois de plus, l'écriture n'est pas directement réalisée dans un fichier, mais sur la sortie standard elle même redirigée vers un fichier si nécessaire. Cela nous permet une fois de plus de pouvoir rediriger facilement la destination d'écriture des données reçues. Contrairement au *sender*, le *receiver* ne reçoit pas de packet de type *ACK* ou *NACK* mais en envoi. Les situations suivantes peuvent être la source d'un envoi de packet :

— Packet de type **NACK**

1. Sous la forme d'une structure *(*Timer). Voir *sender.c* pour plus de détails

— Réception d'un packet au numéro de séquence non attendu. Par exemple la réception du n.seq[4] alors que le n.seq[2] était initialement attendu.

— Packet de type **ACK**

— Réception d'un packet au numéro de séquence attendu. Si celui-ci a été perdu ou retardé avant sa réception, le numéro d'acquis envoyé est celui du dernier packet présent dans le buffer envoyé.

Le *receiver* n'a aucun timer. Lors de la réception d'un packet hors-séquence, celui-ci est placé dans un buffer. Ce buffer est vérifié à chaque réception de packet, afin de pouvoir le vider dès la réception du packet attendu. Lorsqu'il se vide, les différents itérateurs sont mis à jours et un acquis est envoyé. Les ressources du buffer sont quant à elles libérées. Les packets qui sont hors-window sont tout simplement nié. Enfin, notre programme n'envoi pas de packet *NACK* en cas de corruption du packet reçu : en effet la corruption nous empêche d'être sûr que les informations reçues sur le packets sont correctes. On ne peux donc pas envoyer de numéro de séquence, n'étant pas sûr du résultat.

2 Partie critique de l'implémentation

La partie critique se passe à la réception de donnée, du côté du *receiver*. Nous avons implémenté notre programme de telle façon que les données reçues soient directement écrites dans le fichier. Cela ralenti potentiellement le programme qui doit donc traiter en plus d'un envoi d'acquis potentiel, écrire sur le fichier. Une des pistes possibles d'amélioration aurait été de placer le packet dans un buffer qui aurait été traité lors d'une pause potentielle au niveau de la réception de données. Un autre point influençant nos performances est l'envoi direct de donnée (lorsque possible) du côté du *sender*. Nous avons remarqué lors de nos tests que lorsque la window est mise à jour, par exemple lors de l'incrémentation de la borne maximale, le packet correspondant au numéro de séquence de la borne maximale était directement envoyé après l'incrémentation. Cela fait donc appraître des envois de packets de type "$ACK[4] \Rightarrow SEND[8]$" provoquant la mise obligatoire en buffer du côté *receiver*. Un des moyens d'éviter ça aurait été de passer par un buffer alternatif du côté *sender*, par exemple sous la forme du file, ce qui aurait permis un envoi ordonné des données.

3 Choix de la valeur du timeout

Nous avons procédé expérimentalement en effectuant des mesures de temps entre l'envoi et la réception de l'acquis. Nous tombions à une moyenne de 85ms en réseau local. Pour confirmer nos résultats, nous avons utiliser la commande ping [2] afin de trouver un temps moyen de réponse en réseau local. Nous obtenions une gamme de [70ms-116ms] de temps de réponse, ce qui concordait avec nos résultats. Nous avons décidé de prendre une large marge et de mettre le timer à 150ms. Voici-ci dessous quelques relevés de nos tests via la commande ping :

```
PING 192.168.1.10 (192.168.1.10) 56(84) bytes of data.
64 bytes from 192.168.1.10: icmp_seq=1 ttl=64 time=115 ms
64 bytes from 192.168.1.10: icmp_seq=2 ttl=64 time=107 ms
64 bytes from 192.168.1.10: icmp_seq=3 ttl=64 time=112 ms
64 bytes from 192.168.1.10: icmp_seq=4 ttl=64 time=128 ms
64 bytes from 192.168.1.10: icmp_seq=5 ttl=64 time=48.4 ms
64 bytes from 192.168.1.10: icmp_seq=6 ttl=64 time=73.4 ms
64 bytes from 192.168.1.10: icmp_seq=7 ttl=64 time=94.1 ms
64 bytes from 192.168.1.10: icmp_seq=8 ttl=64 time=116 ms

--- 192.168.1.10 ping statistics ---
8 packets transmitted, 8 received, 0% packet loss, time 14020ms
```

2. Pour plus d'information `http://www.computerhope.com/unix/uping.htm`

La meilleure des stratégie serait d'adapter le timout en fonction de la moyenne du flux de bytes envoyés.

4 Stratégie de tests utilisée

De nombreux tests locaux ont été effectués afin de tester le bon fonctionnement du code.

4.1 *socket.h* et *packet.h*

Les fonctions contenues dans les fichiers *socket.h* et *packet.h* ont été testées via des tests unitaires réalisés avec cunit. Ces tests visaient à tester premièrement le bon fonctionnement générale des fonctions, et deuxièmement si les cas limites avaient été gérés. Le mieux étant évidement d'en faire un maximum, nous pensons qu'il aurait été mieux d'en faire plus afin de vérifier un maximum de subtilités de programmation. Néanmoins nous en avons déjà une bonne batterie. Une fonction de *socket.h* n'as cependant pas été testée via des tests unitaires car elle demande une connexion externe au programme testé.

4.2 *sender* et *receiver*

Le comportement de chacun des programmes étant différents, nous avons jugé utile des les tester de façon séparée. Nous avons fait ça dans le but de vérifier que les spécifications de chacun étaient bien respectées avant d'essayer de les faire intéragir. Pour ces deux-là, nous avons eu recours à un ensemble de tests "manuels". Ceux-ci se faisait au moyen de débugage et d'outils tels qu'un simulateur de lien linksim [3]. Des résultats obtenus via ce simulateur sont présentés ici :

```
/link_sim -p 12345 -P 12365 -d 1000 -e 25 &
[6] 5483
@@ Using random seed: 1445936118
@@ Using parameters:
.. port: 12345
.. forward_port: 12365
.. delay: 1000
.. jitter: 0
.. err_rate: 25
.. cut_rate: 0
.. loss_rate: 0
.. seed: 1445936118

./receiver -f rec ::1 12365
[SEQ   0] Delayed packet by 1936 ms
[SEQ   0] Delayed packet by 283 ms
[SEQ   0] Sent packet
[SEQ   1] Delayed packet by 1561 ms
[SEQ   2] Corrupting packet
...
[SEQ   2] Sent packet
 Error: a problem occured during 'decode()'
...
[SEQ   9] Sent packet
[SEQ   7] Delayed packet by 1069 ms
[SEQ  10] Delayed packet by 1443 ms
...
[SEQ   8] Delayed packet by 546 ms
```

3. Source : `https://github.com/oliviertilmans/LINGI1341-linksim`

```
[SEQ   8] Sent packet
[SEQ  12] Delayed packet by 1681 ms
[SEQ  11] Delayed packet by 243 ms
[SEQ  14] Sent packet
...
[SEQ  15] Delayed packet by 1645 ms
[SEQ  15] Sent packet
[SEQ  19] Delayed packet by 1402 ms
[SEQ  18] Sent packet
[SEQ  17] Delayed packet by 583 ms
[SEQ  17] Sent packet
[SEQ  19] Sent packet
```

Et on peux vérifier que le packets envoyés arrivent bien entièrement à destination :

```
md5sum merlon.txt rec
10eb07bd057749f5176d43031f9112f3  merlon.txt[9235bytes]
10eb07bd057749f5176d43031f9112f3  rec[9235bytes]
```

Le simulateur de lien était un bon outil permettant de montrer les failles probables de notre implémentation en faisant varier les différents paramètres proposés. Nous avons entre autre remarqué qu'une faille existait dans notre programme suite à un deadlock apparaissant lors de l'exécution de nos programmes sous certains paramètres du simulateur.

5 Annexes

Des corrections ont été apportées au code suite au test d'interopérabilité S7. Tout d'abord, nous avons procédé à deux tests d'interopérabilité avec deux groupes différents

- **Groupe de *Alexandre Olikier* et *Louis Reynders***
  Notre *receiver* fonctionnait correctement, notre *sender* smblait avoir un problème.
- **Groupe de *Maxime Hulet* et *Olivier Adant***
  Notre *receiver* fonctionnait correctement, notre *sender* semblait avoir un problème.

Les points problématiques suivants ont alors été identifiés et corrigés :

- **Mauvais traitement des paramètres**
  Afin de différencier le paramètre définissant le port de celui définissant l'adresse IPv6 utilisée, nous le passions dans la fonction *atoi(const char * str)*. Nous pensions que celle-ci renverrait le code d'erreur '0' avec un paramètre du format d'une adresse IPv6. Ceci n'étant pas le cas, le paramètre de l'adresse était mis par défaut à la valeur *::*, ce qui était problématique pour le sender. Nous avons donc opté pour une comparaison en fonction de la longueur du paramètre.
- **Memory leaks**
  Nous avons remarqué une présence de memory leak, malgré avoir appliqué de nombreux free(). Après une révision du code, nous avons remarqué qu'il en manquait à plusieurs endroit, et les avons donc ajouté.
- **Affichage sur STDOUT(Moindre problème)**
  Notre *sender* affichait du texte sur la sortie standard, ce qui n'était selon les consignes pas permis.
- **Manque de rapport d'erreur sur STDERR(Moindre problème)** Les informations affichées sur STDERR était simples, trop générales et peu explicites. Nous avons donc ajouté plus d'informations concernant les codes d'erreurs obtenus

Un dernier test a alors été effectué avec le groupe de *Emilio Gamba* et *Jean-Baptiste Macq*. Les deux programmes ont alors fonctionné correctement. Enfin, un test d'envoi via deux machines de la salle intel a été effectuée pour notre propre implémentation, afin de vérifier l'envoi d'un "gros" fichier de 9235 bytes. Il a été réalisé avec succès.