LINGI1341 - Rapport Projet 1

Navarre Louis 12351600 et Plancq Hadrien 56661600

October 2018

1 Gérer les timing

Lors de l'envoi de n'importe quel paquet, qu'il soit un *acknowledgement* ou un paquet transportant des données, nous définissons le champ *Timestamp* au moment de l'envoi à l'aide la fonction *time_t time()*. Grâce à cela, on peut facilement voir quels paquets doivent être retransmis pour cause de timeout. C'est ce que fait notre fonction *int check_timeout(list_t* list, int sfd)*, elle retransmet tous les paquets ayant dépassé le RETRANSMISSION_TIMER, en changeant encore une fois leur *Timestamp* lors du nouvel envoi. Nous avons fixé le retransmission timeout à 2 secondes car cette valeur permet de suivre ce qu'il se passe dans le programme lors de son exécution, notamment pour les tests d'interopérabilité. Pour plus de facilité, cette valeur reste fixe quelque soit la vitesse du réseau.

2 Réception d'un paquet NACK

Notre gestion des paquets NACK est très basique, mais fonctionnelle. Lorsque le `sender` reçoit un paquet de type NACK, il va simplement libérer la mémoire allouée pour ce paquet, donc ignorer son comportement. Le paquet ainsi tronqué par le réseau sera traité donc comme un paquet qui n'a pas reçu d'accusé de réception correspondant. Celui-ci sera envoyé à nouveau lorsque son *retransmission timer* est en timeout.En ce qui concerne le `receiver`, en cas de réception d'un paquet tronqué, va simplement le *discard* et envoyer l'accusé de type NACK correspondant. Nous voyons donc que même si la solution est basique et qu'elle réduit l'efficacité du programme, elle est totalement fonctionnelle.

3 Performances du protocole

Notre protocole est fortement ralentit lorsqu'un paquet se perd. En effet, le RETRANSMISSION_TIMER de notre programme est fixé à 2 secondes. La vitesse de transfert dépend donc directement de la probabilité qu'a un paquet d'atteindre son objectif. Pour un paquet qui ne se perd pas en chemin, l'envoi et la réception s'effectuent de façon quasi instantanée.

En terme de performances, nous pouvons assez facilement déduire que le programme a une complexité de $\Theta(n)$. En effet, doubler la taille de la donnée à envoyer va doubler le temps d'envoi. Ci-dessous un tableau des différents tests effectués pour en venir à ce résultat. Pour ces tests, la probabilité de perte d'un paquet par le réseau a été fixée à 10%.

Taille à envoyer	**Temps d'envoi**
50 kB	8 sec
100 kB	16 sec
150 kB	26 sec
200 kB	34 sec

4 Implémentation du sender

main • process_sender	S'occupe de récupérer les arguments comme le nom du domaine du receveur, le numéro de port, ou encore la présence ou non d'un fichier à lire
process_sender • read_to_list • check_ack • check_timeout • final_send	Fonction générale. Coordonne la lecture-mise en paquet-envoie, la vérification des timeout ainsi que la réception des accusés de réception. La fonction regarde s'il y a quelque chose à lire, ou un accusé de réception à récupérer, mais dans tous les cas, les paquets en timeout sont retransmis et la vérification de fin de transfert est vérifiée à son tour
read_to_list • send_packet	Est appelée si il y a quelque chose à lire sur l'entrée. Si la window n'est pas remplie, alors la fonction lit ce qu'il y a en entrée, crée un paquet adapté et ajoute ce paquet dans la window. Elle envoie ensuite ce paquet, et met à jour le nouveau numéro de séquence. Si la fin du fichier est détectée (EOF), alors n'envoie pas le paquet mais met à jour pkt_fin pour l'envoie final
send_packet	Met à jour le timestamp et envoie le paquet sur le socket
check_ack • packet_checked	Décode le paquet reçu sur le socket file descriptor. Récupère la valeur de la window. Si le paquet est de type NACK, alors le paquet est simplement ignoré (et la mémoire libérée évidemment), si c'est un ACK, alors la fonction appelle packet_checked
packet_checked	Libère toute la mémoire associée aux paquets dont le numéro de séquence est plus petit que le numéro passé en argument (celui qui est reçu dans le ACK)
check_timeout • send_packet	Parcourt toute la liste chainée (servant de window) et envoie les paquets dont le timestamp est trop ancien.
final_send • send_packet	Cette fonction n'est appelée que lorsque la fin du fichier a été détectée dans read_to_list, et vérifie si la window est vide. Si oui, alors nous pouvons assumer que toutes les données ont été envoyées et bien reçues par le receveur. Dans ce cas, la fonction envoie le dernier paquet indiquant que le transfert est fini.

5 Implémentation du receiver

main • process_receiver • wait_for_client	S'occupe de récupérer les arguments comme le nom du domaine du receveur, le numéro de port, ou encore la présence ou non d'un fichier à écrire
process_receiver • read_to_list_r	Fonction générale. Regarde si des données ont été mises à jour sur le socket file descriptor. Si c'est le cas, alors la fonction appelle `read_to_list_r`. Si, au bout de 20 secondes, rien n'a été mis à jour, alors la fonction va se terminer, et le programme aussi.
wait_for_client	Permet de connecter le `receiver` au socket du `sender`.
read_to_list_r • send_packet • write_in_sequence	Est appelée si il y a quelque chose à lire sur le socket. Lit les données du socket, et ajoute le paquet à sa window. Appelle ensuite la fonction `write_in_sequence` uniquement si le paquet entre dans sa window. La valeur de retour de cette fonction permet de savoir si le paquet reçu est dans la séquence attendue. Si oui alors rien n'est fait. Si non, alors un accusé de type ACK est envoyé pour avertir le `sender` que le `receiver` attend toujours le premier paquet en séquence. Si la valeur de retour de la fonction appelée indique la fin du transfert, alors la fonction retourne aussi une valeur à `process_receiver` permettant de terminer la fin du transfert. Si le paquet a été tronqué, alors crée un accusé de type NACK et l'envoie sur le socket avec `send_ack`.
send_ack	Met à jour le timestamp et envoie l'accusé sur le socket
write_in_sequence • send_ack	Parcours la liste chainée servant de window. Ecrit sur la sortie standard/sur le fichier les données des différents paquets qui rentrent dans la séquence attendue, et libère la mémoire associés à chacun d'eux. Envoie un accusé de type ACK à chaque paquet bien traité. Si le premier paquet en séquence est toujours attendu, alors la fonction n'écrit rien, mais retourne une valeur permettant à la fonction `read_to_list_r` de traiter ce cas particulier. Si le paquet final est lu dans cette fonction et qu'il n'y a plus d'autre paquet à traiter, alors la fonction retourne une valeur indiquant que la fin de la procédure.

6 Tests effectués

6.1 Tests unitaires

Pour tester les fonctions de gestion des paquets (fonctions de `packet_implement.c`) ainsi que les fonctions relatives aux listes chaînées (`nyancat.c`), nous avons écrit des tests unitaires en utilisant CUNIT. Il ne nous a pas semblé utile de tester d'autres fonctions à l'aide de CUNIT car un test fourni abordé au point suivant permet de le faire.

6.2 Tests fournis

Notre programme a également été testé en utilisant le fichier test disponible sur Moodle. Nous avons pu tester l'efficacité de notre programme lorsque les paquets avaient jusqu'à 50% de chance de se perdre en chemin ainsi qu'avec un fichier pesant jusqu'à 157697 octets (ce qui permet de parcourir tous les numéros de séquence de la window et de tester le passage du numéro de séquence 255 au numéro de séquence 0). Rarement, mais nous préférons le mentionner, le test indique que le `receiver` ne s'est pas fermé correctement à la fin du test. Ce n'est pas à cause de notre protocole de fermeture mais bien parce que le test fourni regarde l'état du `receiver` 5 secondes après la fermeture du `sender`, or notre protocole de fermeture prend 20 secondes à s'effectuer. Si on change le test pour que l'état du `receiver` soit testé 20 secondes après la fermeture du `sender`, on remarque que notre protocole de fermeture est efficace dans 100% des cas testés.

6.3 Tests de mémoire

L'utilisation de *valgrind* indique que notre implémentation ne comporte aucune fuite de mémoire lors des majeurs tests effectués. Par exemple, notre test majeur fut de s'échanger un fichier de 150kB avec des pertes de 50%, à succès.

7 Améliorations éventuelles

7.1 Gestion des paquets tronqués et NACK

Dans notre implémentation, notre gestion des paquets tronqués et des NACK qui s'en suivent est très basique (voir section ci-dessus). Nous aurions peut-être pu décider que, lorsqu'un NACK est reçu, on envoie à nouveau spécifiquement ce paquet, mais nous avons décidé de simplement attendre un timeout pour le renvoyer. Cette solution n'est pas optimale (puisque nous devons alors attendre la fin du timeout du paquet) mais elle fonctionne. Il s'agit juste ici de rendre plus complet l'implémentation et de rendre le programme plus efficace.

7.2 Gestion de la fin de programme

Pour l'instant, lorsque le `sender` a fini d'envoyer ses données, il envoie le paquet final (avec une longueur de *payload* de 0) et se ferme directement, sans attendre un accusé de réception du `receiver`. Il aurait peut-être été plus ludique de traiter ce paquet comme un paquet normal, c'est-à-dire d'attendre un accusé, et d'envoyer à nouveau le paquet en cas de timeout. D'un autre point de vue, nous nous retrouvons ici dans *un problème des deux armées*, et nous avons donc choisi la solution simple: le `sender` envoie son ultime paquet et se déconnecte. Le `receiver`, quand à lui, s'il reçoit le paquet, envoie un dernier accusé et se ferme. Si ce paquet est perdu, alors le `receiver` attendra au maximum 20 secondes, et au bout de ces 20 secondes sans nouveau paquet reçu, celui-ci se fermera simplement.

7.3 Le *retransmission timer*

Comme dit plus haut dans ce rapport, nous avons décidé de garder un *retransmission timer* constant, d'une part par simplicité mais, surtout, parce que cela nous est utile pour les différents tests d'interopérabilité. Le plus gros problème par rapport à ce *retransmission timer* constant, c'est que notre programme est beaucoup plus lent, car, en cas de perte d'un paquet, le programme devra attendre 2 secondes avant d'envoyer à nouveau ces paquets. Une manière d'arranger cela serait de partir d'un *retransmission timer* de 2 secondes (par exemple), et de changer cette valeur au fur et à mesure du programme en fonction du temps moyen d'envoi d'un paquet - réception de l'accusé correspondant; par exemple, si le temps moyen est de k millisecondes, nous pourrions avoir un *retransmission timer* de $max(2000, 2k)$ millisecondes.

8 Conclusion

Même si certaines choses peuvent être améliorées, comme la gestion du *retransmission timer*, nous estimons que notre programme est fonctionnel. Nous vérifierons cela lors des tests d'interopérabilité.