INGI1341 - Implémentation d'un protocole de transfert fiable

Ortegat Pierre (1954-14-00) &
Dubray Alexandre (1178-14-00)

2 novembre 2016

I Le *sender*

I.1 L'architecture générale

Nous passerons les détails de l'établissement de la connexion pour nous concentrer sur l'essentiel du protocole et des choix d'implémentations du côté du *sender*. Supposons donc une connexion établie à un hôte distant. Faisons abstraction des contraintes (c.f code source) et supposons l'envoie d'une suite $S_1, S_2, \dots S_N$ de segment de données. Chaque segment envoyé sera stocké dans une queue [1].

Un des avantages d'une queue est que, une allocation dynamique de la mémoire permet une utilisation minimale de celle-ci lorsque le réseau est peu actif. En outre, nous gardons facilement cette queue ordonnée par ordre de *retransimission timer* [2]. Ainsi, lorsque nous parcourons notre queue pour retransmettre les paquets, nous savons que lorsque nous en rencontrons un qui n'a pas encore *time out*, alors ceux qui suivent non plus.

Lorsque nous recevons un acquittement, la première chose à faire est, si possible, retirer des paquets de la queue. Ensuite, il y a plusieurs possibilités, en fonction du champ *window* du segment. Soit k la valeur de ce champ.

— Si $k > 0$, alors on peut envoyé jusqu'à k (k compris) nouveau paquets.
— Si $k = 0$, alors le *receiver* n'a plus de place libre dans son buffer de réception et aucune nouvelles données ne peut être envoyée.. Entre alors en scène un système de temporisateur grâce à l'appel système alarm(). Le temps de ce temporisateur est fixé à quatre secondes (deux étant le RTT maximum). À chaque

1. Nous avons utilisé l'implémentation d'une *"tail queue"* définie dans la librairie *BSD libc* (TAILQ)

2. Remarquons que nous pourrions faire le raisonnement qui suit avec les numéros de séquences. L'opération optimisée ne serait donc plus la retransmission des paquets mais les acquittements

fois qu'un signal SIGALRM est intercepté, le dernier paquet envoyé et renvoyé. Ainsi, si les acquittements du *receiver* se perdent, on peut le "réveiller".

Lorsqu'il n'y a plus de données à lire sur l'entrée standard, le segment de fin de connexion est envoyé si et seulement si toutes les données ont été envoyées ET acquittée. Ainsi, même si il y a des problèmes pour fermer la connexion, toutes les données auront été envoyée.

I.2 Le *retransmission timeout*

Remarquons dans un premier temps, que le champ *timestamp* des segments envoyés sont tous nuls. En effet, les élément de la queue dans laquelle nous stockons les segments sont en fait une structure qui contient le segment et le temps (via une struct timeval) auquel le segment devra être retransmit.

Nous avons décider d'implémenter un *retransmission timer* (RT0) dynamique. L'avantage par rapport à un RTO fixe est que l'hôte réagit mieux aux variations du réseaux. Lorsqu'un acquittement est reçu pour un paquet, le (RT0) est diminuer.

Lorsqu'un segment doit être renvoyé, le RTO est augmenté. Une retransmission peut être due à deux facteurs

- Une congestion du réseau. Alors, il est important de d'augmenter le RTO pour éviter d'envoyer trop de segment sur le réseau et, ce faisant, le saturer encore plus.
- Une perte du segment. Celle-ci ne peut être devinée lorsque l'on augmente le RTO. Le fait d'augmenter le RTO alors que le réseau n'est pas congestionné n'est pas un problème vu que, lorsque le segment sera acquitté, le RTO sera diminué.

II Le *receiver*

II.1 Architecture générale

Comme précédemment dit, nous passerons les détails de la création de la connexion et nous nous concentrerons sur la façon dont le receiver va traiter les paquets qu'il reçoit.

La façon dont le receiver se comporte est plutôt simple et peut se décomposer en quelques étapes. Pour commencer, le receiver va établir la connexion et se mettre en attente de donnée sur la connexion. Dès qu'une donnée arrive il va tenter le la décoder en un paquet. Si le paquet n'est pas valide, il va être libéré de la mémoire pour en attendre un qui n'est pas corrompu. Si le paquet a su être créé avec succès, le receiver va le confronter à son buffer interne. A partir de ce point il y a plusieurs situation possible :

— le paquet est avant la windows actuelle (le numéro de séquence du paquet est plus petit que le numéro de séquence attendu) : dans ce cas, le paquet doit juste être confirmé au sender, il semblerait qu'un ack se soit perdu.
— le paquet est après la windows actuelle le numéro de séquence du paquet est plus grand que le numéro de séquence attendu) : ce cas-ci est fort semblable au précédent : le sender semble ne pas avoir encore réalisé qu'un paquet a été perdu. Pour résoudre cela, il suffit de transmettre un ack avec le paquet actuellement attendu au sender pour l'informer de quelles sont les informations actuellement manquantes pour continuer.
— le paquet a un numéro de séquence repris dans le buffer et il n'est pas déjà dans le buffer : ce cas ci est le cas de base, il suffit d'insérer le paquet dans le buffer et de signaler au sender ou on en est dans la réception des données en envoyant un ack avec comme numéro de séquence le prochain paquet attendu.
— le paquet peut être inséré dans le buffer mais est déjà présent : cas encore plus simple que le précédent : il semblerait qu'il y aie des erreurs de transmission. On droppe le paquet et on renvoie un ack.

Une fois que le paquet a été confronté au buffer, nous allons regarder si il est possible de sortir des données du buffer sur le file descriptor que l'on a (qui peut être stdout comme un fichier, ça ne change rien). Si le paquet qu'il nous manquait (c-a-d le paquet suivant les données que l'on a déjà sortit) est présent dans le buffer nous allons sortir toutes les données continues du buffer et définir la prochaine donnée voulue sur la donnée juste après la dernière que l'on a sortit.

Finalement, une fois que nous avons traité touts les paquets, on envoie un ack si nécésaire et nous vérifions si nous avons un paquet de type EOF. Si c'est le cas, nous allons ack le paquet eof et quitter.

III Partie critique

La partie critique de notre implémentation, celle qui défini a rapidité de nos programme est le temps définie par les retransmission timer. En effet, en réseau parfait, nous allons à une vitesse fort convenable mais nos programmes se trouvent fortement ralentis dès que le réseau a des pertes. En effet on commencera à observer des moments de "blanc" durant lesquels, le sender n'a pas eu les ack nécessaire pour continuer et attend que ses times expirent.

IV Les tests

Pour les tests, nous considérons trois cas de fichiers. Le premier est le plus logique, un fichier remplis de caractère aléatoire. Le deuxième type est un fichier dont tout les bits sont à 1 et le dernier est un fichier dont tout les bits sont à 0.

En outre, nous considérons diverse taille de fichier, allant de 300 bytes jusque 40000bytes.

Nous avons deux scripts de test. Le premier ne simule pas de réseau (i.e. les pertes, les délais,etc.) Il permet entre autre de vérifier l'exactitude du programme en faisant abstraction du réseaux. Via ce script on peut donc vérifier que notre programme reprend les fonctionnalités de bases.

Le deuxième script simule un réseau. Ce test permet donc de déterminer si notre programme prend bien en compte les différents éléments du réseau, comme les pertes de paquets, les délais, les altérations de paquets, etc.

Les deux scripts fonctionne de la même manière. Pour chaque fichier ils vont lancer le transfert, si une erreur survient ou que le fichier est corrompu, le script s'arrête et renvoie 1. Si les tests sont réussis, le script renvoie 0.

I Résultat des tests d'interopérabilité

Pour l'intégralité de nos tests d'interopérabilité, nous avons divisé les tests en quatre sous tests, tous fait avec un fichier aléatoire de 262 144 bytes ($512 * 512$) générés aléatoirement. Après chaque sous test, nous vérifions que le transfert s'est déroulé correctement avec md5sum. Voici la liste de nos tests :

1. Nous testons le sender de l'autre groupe avec notre receiver
2. Nous testons le receiver de l'autre groupe avec notre sender
3. Nous testons, avec linksim activé, leur sender et notre receiver
4. Nous testons, avec linksim activé, leur receiver et notre sender

A chaque fois que nous devions utiliser linksim, il était réglé de la façon suivante :

— 10% de pertes
— 10% de corruption
— 50 ms de délai
— 25 ms de jitering

Voici la liste de nos tests d'interopérabilité ainsi que leur résultat :

— Groupe 84 (Sébastien Strebelle & Julian Roussieau) : un première série de test découvre un problème de leur coté, mais, après un rapide fix de leur part, tous les tests passent (ils avaient mal compris en quoi consistait l'EOF).
— Groupe 47 (Céline Deknop & Adrien Hallet) : tous les tests passent du premier coup, aucune erreur.
— Groupe

II Changement du *sender*

Plusieurs changements/améliorations ont été apportée à l'algorithme du *sender* par rapport à la première version remise. Ces changements visaient particulièrement à augmenter les performances, diminuer l'utilisation de la mémoire ou encore avoir un meilleurs *retransmission timer*. Voici les différents changement opérer :

— Utilisation du champ *timestamp*. Dans la première version, le *timestamp* n'était pas utilisé et nous gardions, parallèlement au paquet envoyé, une `struct timeval` pour garder en mémoire le moment où le le paquet a été envoyé. Dans la version finale, nous utilisons le champ *timestamp* de la manière suivante : nous récupérons le temps via l'appel gettimeofday(). Nous prenons les 12 bits de poids faibles du champs `tv_sec` et les 20 bits de poids faibles du champ `tv_usec` pour les combiner dans le champs *timestamp* (les 12 bits des secondes seront les 12 bits de poids fort).
Le principal avantage d'utiliser cette technique est que l'on gagne de la mémoire. En effet, pour chaque paquet stocker dans la queue, on ne doit plus garder en mémoire une `struct timeval`.

— Amélioration de la diminution du *retransmission timeout* (RTO). Dans la première version, l'augmentation et la diminution du RTO se faisaient par rapport à une fraction de celui-ci. Cependant, lorsque l'on diminue le RTO (quand un paquet est acquitté), la diminution se fait maintenant par rapport à une fraction du temps qu'il restait au paquet avant qu'il ne *time out*. Par exemple, si un paquet est acquitté $1\mu s$ avant qu'il ne *time out*, il faut moins diminuer le RTO que si il est acquitté 1 seconde avant de *time out*.
— Amélioration de la vérification des anciens acquittement. La vérification des segment se fait maintenant via trois axes :

1. Vérification de l'acquittement via le CRC
2. Vérification que l'acquittement correspond à un paquet qui peut potentiellement se trouver dans la fenêtre de réception.
3. Vérification que ce n'est pas un ancien segment grâce au champ timestamp qui nous donne une indication de quand le paquet a été envoyé. Si il a été envoyé il y a plus de MSL seconde, l'acquittement est considérer comme un ancien acquittement.

— Enfin, le *timer* utilisé par l'appel `select()` a été optimisé. Il est paramétré au temps restant avant que le premier paquet insérer dans la queue (celui qui se trouve à la tête de la queue) doive être renvoyé. Nous savons donc que si l'appel à `select()` renvoie 0, on peut directement renvoyé le paquet et refaire un appel à `select()`.