INGI1341 - Implémentation d'un protocole de transfert fiable

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

26 octobre 2016

Note : pour compiler, notre projet utilise cmake. Il est présent par défaut sur les machines la salle mais pas forcément sur toutes les distributions linux.

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.

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

— 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 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

Notre stratégie de tests, bien que fort simple suffit pour nous révéler touts les problèmes que l'on pourrait observer lors d'un transfert utilisant notre programme. En effet, quoi de plus simple que de tester si un fichier a été transféré correctement pour tester un programme de transfert de fichier ? C'est à cette idée que nous nous sommes raccrochés et nous l'avons appliqué tout le long de nos tests avec différentes tailles de fichier et différentes configuration.