LINGI1341 - Projet 1

Protocole de transfert fiable

Le 31 octobre 2014

Cours — LINGI1341 – Réseaux informatiques
Auteurs — Sundeep Dhillon (6401-11-00)
— Romain Henneton (4783-12-00)
Professeur — Olivier Bonaventure
Année académique — 2014–2015
Programme d'études — FSA13BA et SINF13BA

Table des matières

Introduction

Dans le cadre de notre cours de réseaux informatiques, on nous a confié un projet consistant à la création d'un protocole UDP de transfert fiable en utilisant un *selective repeat*. Ce projet est composé de deux objectifs principaux : être capable de créer ce protocole de transfert et effectuer des tests avec les protocoles créés par d'autres groupes d'étudiants (et éventuellement, apporter des modifications à notre code suite à ces tests).

1 Structure du programme

Il nous a été demandé de créer un protocole de transfert gérant le *selective repeat*.
Pour cela, deux options s'offraient à nous, à savoir :

- Utiliser la fonction `"select"` toute faite, gérant les *timer* et les *acknowledgements* automatiquement.
- Implémenter ce *selective repeat* via des threads (un *listener* et un *timer*).

Nous avons décidé d'utiliser la seconde méthode. En effet, il nous semblait plus intéressant de comprendre fondamentalement comment se passait le *selective repeat* plutôt que d'utiliser une fonction toute faite, et ceci afin qu'en cas d'erreur, nous puissions cibler le problème facilement vu que nous utilisons nos propres fonctions et donc, une implémentation de plus bas niveau, spécifique à notre code.

1.1 Sender

Ce dernier est composé de trois blocs principaux, à savoir :

- `sender primaire`
- `ackreceiver`
- `resender`

Dans le **sender primaire**, nous commençons par récupérer les différentes options et informations telles que le port, le hostname ainsi que les options facultatives. Ensuite, nous créons un tableau contenant les paquets à envoyer dans l'ordre. Par la suite, un sémaphore est utilisé pour limiter le nombre de paquets envoyés simultanément (notion de *window*).

Ce sémaphore est initialisé à 1 et à la réception du premier *acknowledgement* (voir **ackreceiver**), nous mettons à jour la taille de la *window* maximale grâce à l'information *window* située à l'intérieur du paquet *acknowledgement*. Il est donc possible d'envoyer au départ, un seul paquet et par la suite, le nombre de paquets est adapté au *buffer* de réception, le tout de manière séquentielle.

Pendant ce temps, deux threads s'exécutent en parallèle : l'**ackreceiver** qui écoute sur le *socket*, réceptionne les *acknowledgements*, les traite de façon à indiquer au **resender** quel(s) paquet(s) doit/doivent être ré-envoyé(s) et quel(s) paquet(s) a/ont été correctement réceptionné (afin de gérer les pertes de paquets dûes à *sber*, et à *splr*) ainsi que d'indiquer quand le *sender* et les différents threads doivent s'arrêter. Lorsqu'un *acknowledgement* correct est reçu, l'**ackreceiver** effectue un/plusieurs sempost(s) afin de mettre à jour la *window* d'envoi.

Le **resender** est un thread qui vérifie, via un timer, quels paquets doivent être ré-envoyés. Pour cela, il mémorise le numéro de séquence du premier élément de la *window* avant le lancement du *timer* et le compare avec le numéro de séquence du premier élément après le *timer*. Si un *acknowledgement* a été correctement reçu par l'**ackreceiver**, alors cette valeur sera différente (mise à jour de la variable `"elemnow"`), et le *timer* sera relancé. Par contre, si ces deux numéros de séquence sont égaux, cela signifie que le premier paquet de la *window* n'a pas été correctement reçu et doit donc etre ré-envoyé. Avant de ré-envoyer les paquets futurs, le **resender** vérifie chaque fois ce numéro `"elemnow"`.

1.2 Receiver

Comme pour le **sender**, le **receiver** récupère tout d'abord les différentes options, lit le socket en prenant différents paramètres tels que le *hostname* et le *port*. Par la suite, il écoute sur le *socket* (`receivefrom`) pour récupérer les paquets envoyés par le **sender**. Lorsqu'un paquet est reçu, trois cas se présentent à nous :

- Soit le CRC est incorrect et aucun *acknowledgement* n'est envoyé.
- Soit le CRC est correct, mais le numéro de séquence n'est pas le bon (le numéro de séquence est supérieur au numéro de séquence attendu). Dans ce cas-là, le paquet est stocké dans un *buffer* (notion de *window*).
- Si le CRC est correct et que le numéro de séquence du paquet est le numéro de séquence attendu, alors le paquet est imprimé sur *stdout* ou dans le *filename* spécifié dans les options.
  De plus, on vérifie si les numéros de séquence suivants sont présents dans le *buffer* ou non. Si oui, ils sont écrits jusqu'à arriver à un numéro de séquence manquant.
  Une fois tout cela effectué, un *acknowledgement* portant le numéro du dernier paquet écrit + 1 est renvoyé au **sender**. Si un paquet de taille inférieur à 511 bytes est écrit, cela signifie que c'était le dernier et le **receiver** se termine.

2 Tests d'interopérabilité

Nous avons réalisé des tests d'interopérabilité entre notre code et celui de deux autres groupes.

2.1 Premier test

Ce premier test a eu lieu en collaboration avec le groupe composé de *Charles Jacquet* ainsi que *Stéphane Kimmel*. Assez étonnament, nous avons pu constater que tout fonctionnait de manière correcte du premier coup. Nous avons donc pu bien vérifier que l'implémentation de notre *sender* était indépendante de l'implémentation du *receiver* et par conséquent, générique et théoriquement ré-utilisable par n'importe quel groupe.
En effet, notre *receiver* couplé avec leur *sender* fonctionnait sans problème apparent et il en est de même pour notre *sender* couplé avec leur *receiver*.

2.2 Second test

Ce dernier a eu lieu en collaboration avec le groupe composé de *Sandrine Romainville* et *Jolan Werner*. Après après avoir effectué un test, nous avons pu constater que ce dernier ne

pouvait aboutir. Pourquoi ? Nous avons pu déceler un petit problème de CRC. En effet, leurs paquets n'ont pas été faits de la même manière que chez nous et par conséquent, vu que les bits ne sont pas stockés de la même manière pour le CRC, cela justifie l'échec du test.

3 Remarques, limitations et difficultés rencontrées

1. Dans un paquet, nous pouvons stocker au maximum 511 bytes. En effet, la consigne étant de pouvoir envoyer un input sur *stdin*, nous devons conserver le dernier caractère du *payload* libre pour pouvoir y insérer le caractère *\0* indiquant la fin de la chaîne de caractères.

2. De plus, ce qui découle de cette première remarque est le fait que nous ne pouvons envoyer que des fichiers *.txt* et non pas de fichiers binaires (qui seraient de toute manière inaffichables sur *stdout*). Il pourrait être intéressant de vérifier par la suite le type d'input, et de s'adapter au type .txt et/ou binaire.

3. Pour implémenter `sber`, nous avons utilisé un *flag* indiquant si un caractère a été modifié (nous modifions uniquement le premier caractère du *payload*) et lorsque le paquet a été envoyé, nous remettons ce caractère modifié à sa valeur initiale afin d'éviter que le paquet reste indéfiniment corrompu et que le *resender* puisse l'envoyer correctement.

4. Lors de nos tests, nous avons vérifié que le *selective repeat* fonctionnait correctement (en regardant l'ordre dans lequel les paquets arrivaient, que le buffer fonctionnait bien...etc). Cependant, nous avons supprimé les `printf` nécessaires à ce debug car si le *payload* reçu doit être affiché sur *stdout* (pas d'option *file*), alors le texte sur *stdout* contiendrait à la fois le texte de debug ainsi que les différentes données envoyées, ce qui n'est pas l'objectif recherché.

5. Nous avons utilisé des `mutex` pour protéger le *socket* de descripteur de fichier du *send* ainsi que l'entier `"elemenow"` utilisé dans les différents threads (exclusion mutuelle).

3.1 Extension(s)

Nous n'avions pas bien compris l'utilisation de l'option *delay* dans les consignes intiiales et pensions qu'il s'agissait du temps d'attente entre deux envois de paquets.
Pour corriger ce problème, nous pourrions réalisé une fonction alternative `sendtobis` qui attend un certain temps avant d'envoyer le paquet. Au lieu d'utiliser *sendto* dans la fonction principale avec le *resender*, nous pourrons donc créer un thread pour chaque paquet à envoyer, utilisant cette fonciton `sendtobis` et ce, afin d'implémenter correctement le *delay* qui est le temps de propagation sur le câble.