Rapport du projet 1
Implémentation d'un protocole de transport sans pertes

Alexis Nootens Philippe Stormme

29 octobre 2015

1 Introduction

Dans le cadre du premier projet du cours de réseaux informatiques, nous avons travaillé à une solution pour rendre le transport de donnée sur des segments UDP aussi fiable qu'en TCP. Pour ce faire nous avons implémenté un protocole de transport de données et nous l'avons integré dans une application d'envoi d'un fichier entre deux hôtes. Ce document contient l'explication de notre solution, les choix qui ont été effectués et l'explication de notre méthode de test afin de vérifier cette solution.

2 Implémentation

Dans la poursuite d'un développement itératif et incrémental, l'implémentation du programme s'est déroulée en deux étapes détaillées ci-dessous.

2.1 Les paquets

Tout d'abrord, nous avons défini une structure représentant les paquets qui seront échangés sur le réseau. Un paquet est constitué d'un en-tête, d'un payload et d'une somme de contrôle.

— L'en-tête contient des informations sur le paquet et sur le programme qui l'a envoyé.
— Le payload constitue les données à transférer.
— La somme de contrôle permet de vérifier qu'à l'arrivée, le paquet est bien tel qu'il était lors de son envoi.

Nous avons ensuite ajouté des fonctions qui nous permettraient d'utiliser cette structure par la suite, notamment une fonction permettant de traduire le contenu de la structure en données pouvant être envoyées sur le réseau et une autre permettant de retransformer ces données sous la forme de la structure.

2.2 Les programmes sender et receiver

Une fois les deux premières étapes complétées, nous sommes passés à l'écriture des programmes sender et receiver. Ceux-ci reçoivent en argument une adresse et un port, ainsi qu'éventuellement un nom de fichier. Tous deux commencent par établir une connexion entre eux avant d'ouvrir, s'il y a lieu, le fichier dont le nom a été passé en argument et de lancer la boucle principale du programme qui ne s'interrompra qu'en cas d'erreur (irrésolvable) ou au terme du transfert.

2.2.1 Le programme sender

Le programme sender dispose d'un buffer dans lequel il stocke chaque paquet envoyé en attendant la réception d'un ACK dont le numéro de séquence est supérieur à celui du paquet, à la réception de ce ACK, le paquet est retiré du buffer. Dans sa boucle principale, sender, si son buffer n'est pas plein, lit 512 octets de l'entrée standard ou du fichier d'entrée, l'encode dans un paquet de données, stocke celui-ci dans le buffer et l'envoie sur le socket.

Ensuite un appel à select permet de vérifier si un paquet de contrôle (ACK, NACK) a été reçu, en cas de réception d'un ACK, le buffer est délesté des éléments reçus, en cas de réception d'un NACK, le paquet correspondant est renvoyé. Pour finir, le temps écoulé depuis l'envoi des paquets du buffer est calculé et les envois restés sans réponse pendant un certain temps sont répétés.

Les trois actions précédentes sont répétées jusqu'à ce que le fichier soit lu entièrement ou que l'utilisateur envoie un EOF sur l'entrée standard.

A propos de la valeur du timeout, nous la fixons de la façon suivante : au lancement du programme, on fixe un temps court et arbitraire qui servira de délai maximum pour la réception du premier ACK / NACK.
Ensuite, à chaque réception d'un paquet envoyé par le receiver, le délai est modifié pour valoir la somme de la moitié de son ancienne valeur et de la moitié du temps passé à attendre le dernier ACK / NACK. Quand ce délai maximum est dépassé, le paquet pour lequel aucun ACK n'a été reçu avec le seqnum le plus faible est renvoyé.
Cette solution a été sélectionnée car elle semble minimiser temps de transfert et duplication des paquets, en tous cas par rapport aux autres solutions considérées.

2.2.2 Le programme receiver

Le programme receiver dispose d'un buffer dans lequel il stocke les paquets hors-séquence reçus, lors de la réception d'un paquet en séquence, les paquets du buffer le suivant directement sont extraits et écrits sur stdout ou le fichier cible s'il y en a un.

Dans sa boucle principale, le receiver utilise un appel select pour vérifier si un paquet est disponible sur le socket. Quand c'est le cas, le paquet est décodé, si celui-ci est valide et est en séquence, il est écrit dans le fichier cible ou la sortie standard et un ACK est envoyé vers sender, s'il est valide et hors séquence, il est stocké dans le buffer. Si au contraire le paquet est invalide, alors un NACK est envoyé au sender.

Lorsqu'un paquet de données avec un payload de longueur nulle est reçu, le transfert est considéré comme terminé.

3 Architecture

Les fichiers sender.c et receiver.c sont les fichiers principaux, ils contiennent chacun une fonction `main` et l'essentiel des fonctions leur permettant d'assurer leur rôle lors du transfert d'un fichier.

Ces deux fichiers principaux utilisent tous deux les fonctions contenues dans packet.c et socket.c, le premier servant à manipuler les paquets et le second à établir une connection entre le sender et le receiver.

Ils utilisent aussi locales.h et argpars.c. Le fichier locales.h contient une structure contenant des variables globales aussi bien utiles à sender qu'à receiver. Quant à argpars.c, il contient une unique fonction permettant de lire et interpréter les arguments du programme, leurs valeurs étant ensuite stockées dans la structure définie dans locales.h et utilisées pendant toute la durée du transfert. Le fichier argpars.c est inclus directement dans sender.c et receiver.c car sa seule raison d'être est de permettre d'éviter de dupliquer son contenu.

Le dossier `osdep` et son contenu permettent, sous OSX, de traduire les fonctions issuent de endian.h dans leur version BSD sans altérer le code.

4 Tests internes

Dans l'optique de porter nos tests sur différentes implémentations du projet en provenance d'autres étudiants. Nous nous sommes limités à vérifier les seules interfaces qui nous ont été fournises. (*i.e.* les paquets et les executables `sender`, `receiver`). Ainsi les tests sont divisés en deux modus operandi :

1. Utilisation du framework CUnit pour tester les méthodes `encode` et `decode` des paquets.
2. Utilisation d'un script shell pour assurer, avec l'aide de checksums, qu'un fichier généré aléatoirement et envoyé par `sender` est le même que celui reçu par `receiver` sur une liaison parfaite et une liaison qui retarde, alterne, voir coupe les paquets.

Concernant les tests sur les fonctions encode et decode avec CUnit. Le résultat de la fonction encode est testé uniquement pour un paquet valide, on suppose que l'encodage d'un paquet invalide constitue une erreur du programme et non de la fonction encode. En revanche, pour la fonction decode, on teste le résultat pour un paquet valide ainsi que le résultat pour des paquets invalides pour les raisons suivantes :

— crc invalide
— longueur incohérente
— paquet coupé
— type invalide

À la suite des tests unitaires, nous effectuons un exercice global de transfert en script shell se découpant en deux périodes : La première consistant à transférer un fichier de 100 megabytes sur une liaison loopback parfaite, la seconde est de transférer un fichier d'1 megabyte sur une liaison aux propriétées suivantes :

— Perte de 5% des paquets
— Corruption de 5% des paquets
— Coupure de 5% des paquets
— Délai de transmission variant dans l'intervalle de 0.5 à 1 seconde

Ces exercices se déroulent séquentiellement dans l'ordre énoncé et sont vérifié à chaque périodes par une comparaison des checksums entre le fichier envoyé et le fichier reçu.

Pour lancer la suite de test, le reviewer peut exécuter la règle `make test` depuis le dossier principale. Le script de test octroi un temps d'exécution de deux minutes au test sur une liaison parfaite, et quatres minutes sur une liaison imparfaite. Au délà de ces temps, le test sera considéré comme échoué. Ces valeurs ont été choisies par expérimentation sur notre solution et de la discution du temps d'opération des solutions de nos collègues.

5 Performances

En l'absence d'erreurs de transmission, la vitesse du transfert parait optimale et ne semble affectée que par des éléments extérieurs au programme (vitesse / charge du réseau ou d'écriture / lecture du disque-dur, ...).

En cas d'erreurs de transmission fréquentes, la vitesse du transfert est considérablement diminuée, une partie des paquets doit en effet être renvoyée et ce seulement une fois détectée, ce sont donc les paquets intégralement perdus qui ont l'impact le plus important sur la vitesse du transfert, en dehors des délais qui naturellement augmentent le temps passé par les paquets entre sender et receiver, sans que rien ne puisse y être fait.

La quantité d'erreurs lors du transfert a un impact sur la vitesse de transmission, mais par paliers, 10 pourcent de corruption n'affectent pas la vitesse du transfert davantage que 5 pourcent, mais 90 pourcent causeraient un ralentissement bien plus important.

6 Tests d'interopérabilité

Nos tests d'interopérabilité ont été effectués avec le groupe 90 regroupant Alexis Macq et Maxime Piraux. Ainsi que le groupe 55 regroupant Florian Felten et Julien Sterbelle.

Les tests suivant ont été effecté dans le sens sender → receiver et receiver → sender.

6.1 Groupe 90

1. Transfert d'un fichier texte léger (<1MB)
2. Transfert d'un fichier vidéo webm (4.2MB)

Les deux cas ce sont montré concluant, les fichiers envoyés étant bien les fichiers reçus. Le test n'ont pas nécessité plus d'appronfondissement et notre projet d'aucune modification.

6.2 Groupe 55

1. Discussion unilatérale par transmission séquentielle de message.
2. Transfert de /dev/urandom jusqu'à plus soif.

Encore ici, les tests se sont montrés concluant et n'ont nécéssités aucune modification du programme.

6.3 Modification du projet

Une modification fut néanmoins effectuée dans la fonction `real_address()` après s'être rendu compte que nous essayons de libérer le résultat alloué par `getaddrinfo()` même si cette dernière échouait.

7 Conclusion

Nous sommes fiers de soumettre une solution fonctionnelle et complète remplissant les conditions de l'énoncé. Mais nous sommes aussi personnellement et particulièrement satisfaits de l'amélioration de notre compréhension du protocole UDP, et TCP par extension. La différence de performance de transfert entre un réseau parfait et imparfait nous a ouvert les yeux sur la difficulté de maintenir une connexion efficace entre deux hôtes distants.