Implémentation d'un protocole de transport sans pertes
Rapport suite aux tests d'interopérabilité

Gustin Simon et Pinon Brieuc
11711400 - 68411400

October 2016

1 Introduction

Dans le cadre du cours de réseaux `LINGI1341`, il nous a été demandé d'implémenter un protocole de transport sans perte basé sur l'`UDP` et l'`IPV6`. Le protocole doit aussi protéger l'intégrité des paquets via des `CRC`. La stratégie utilisée doit être `go-back-n` augmenté d'un `selective repeat` implémenté côté récepteur. De plus, le protocole doit supporter des tailles de fenêtres de réception variables. Finalement, le protocole utilise aussi une méthode `graceful disconnect` pour terminer la connexion.

Après l'implémentation de notre protocole, des tests d'interopérabilité avec les programmes d'autres groupes ont été réalisés (cf. section A en annexe).

On notera `nom_du_fichier` pour indiquer `nom_du_fichier.c` et son *header*.

2 Structure du code

Le `Makefile` permet de produire les deux exécutables (`sender` et `receiver`). Le premier a pour fichier de base `sender` et le deuxième a `receiver`. Chacun de ces fichiers a ses propres dépendances.

Détails des fichiers et de leur utilisation :

- `sender` Fonction `main` pour l'exécutable `sender`. Ouverture et fermeture des fichiers d'input et d'output, appel des fonctions définies dans d'autres fichiers pour établir la connexion et transmettre les données spécifiés au récepteur.
- `receiver` Fonction `main` pour l'exécutable `receiver`. Ouverture et fermeture des fichiers d'input et d'output, appel de fonctions définies dans d'autres fichiers pour établir la connexion et recevoir les données de l'émetteur.
- `createConnection` Fonctions qui s'occupent de la résolution d'adresses, ainsi que de la création du socket pour l'émetteur ou le receveur et de la connexion du socket côté récepteur.
- `packet` Définition des paquets tel que transmis sur la connexion. Utilisé par les fonctions d'envoi et de réception de paquets.
- `senderReadWriteLoop` Gère l'émission de paquets de données depuis l'entrée vers le récepteur, et la réception d'acquittements. Utilisé par `sender`.
- `senderManagePacket` S'occupe de l'encodage des paquets de données et le décodage des acquittements, fourni aussi les fonctions pour gérer les différents buffers de paquets. Utilisé par `sender`.
- `receiverReadWriteLoop` Gère la réception de paquets de données et l'émission d'cquittements. Utilisé par `receiver`.
- `receiverManagePacket` Fonction pour le décodage de paquets de données, la création de paquet d'acquittement, et la gestion des paquets dans les buffers. Utilisé par `receiver`.
- `timer` Gestion des timers. Utilisé par `senderReadWriteLoop`, `senderManagePackets` ainsi que `receiverManagePackets`.

3 Spécificités de l'implémentation

Différentes décisions d'implémentations ont été prises lors de l'implémentation de ce projet.

3.1 Structures de données utilisée pour les buffers de réception et d'émission

L'implémentation se doit d'utiliser des buffers pour placer les paquets quelque part entre le moment où ceux-ci son créés et envoyé vers l'interlocuteur, ou entre le moment où ceux-ci sont décodés et le moment où ils sont écrits sur le fichier de sortie (côté récepteur). Nous devons en effet attendre que les fichiers sur lesquels on veut faire des opérations d'entrée/sortie soient prêts et nous ne voulons pas mettre le programme en pause le temps d'attendre.

Nous avons décidé d'implémenter des tableaux circulaires, soit des tableaux avec une variable indiquant un début et le nombre d'éléments qu'il contient et où le dépassement d'un tableau par la droite renvoie au début du tableau. Ceux-ci permettent d'implémenter un créateur (remplisseur) et un décodeur (videur) sous le principe `FIFO` avec des conditions sur le nombres maximal de paquets créés mais non-décodés, ce qui correspond exactement à nos besoins.

Nous aurions aussi pu utiliser des listes chainées, qui auraient eu l'avantage de ne pas avoir de taille maximale (dans les limites de la mémoire physique de la machine). Cependant, ces structures prennent beaucoup plus de temps pour atteindre un nœud, puisqu'il faut en parcourir une partie pour y arriver. Étant donné que nous faisons beaucoup d'opérations sur les différents éléments présents dans ces buffers, nous avons préféré utiliser des tableaux circulaires.

Description des différents buffers :

- Dans `senderManagePacket`, pour stocker les paquets créés par l'émetteur mais pas encore envoyés ou pour lesquels on n'a pas encore reçu d'acquittement, nous avons un buffer appelé `bufPktToSend`. C'est un buffer d'une taille assez conséquente afin de pouvoir préparer un grand nombre de paquets dans le cas où les opérations d'entrée/sortie ne seraient pas possibles pendant un certain temps sur le socket par exemple.

- Dans `receiverManagePacket`, pour stocker les paquets reçus et prêts à être écrits dans le fichier de sortie côté récepteur, nous avons le buffer `dataPktInSequence`. Ce buffer permet par sa taille de stocker plus de paquets qu'une solution où seul un buffer tel que décrit ci-après ne permettrait de le faire.

- Pour recevoir les paquets hors séquence mais dans la fenêtre de réception, nous avons le buffer `bufOutOfSequencePkt`. Dès que les paquets manquants avec un numéro de séquence plus ancien sont reçus, les paquets devenant "en séquences" sont mis dans le buffer précédent, où ils attendront d'être écrits dans le fichier de sortie. Ce buffer est utilisé dans `receiverManagePacket`.

3.2 Envoi d'acquittement

Bien que nous ayons testé une solution avec un buffer pour l'envoi des acquittements, nous avons finalement opté pour une seule variable globale pour cette tâche. Nous avons donc une variable globale qui permet de garder en mémoire l'acquittement le plus récent. Un acquittement étant préparé à chaque fois qu'un paquet arrive chez le receveur, cette variable globale est souvent mise à jour. Quand le socket est disponible pour envoyer l'acquittement préparé, le receveur l'envoie. Dans le cas d'un réseau parfait, tous les acquittements sont donc envoyés. Par contre, pour un réseau plus lent, ou dans le cas où le socket serait souvent indisponible pour des opérations d'écriture, le receveur aura le temps de recevoir d'autres paquets de données, et donc de remplacer le dernier acquittement par un acquittement plus récent avant de l'envoyer.

Si on ne reçoit pas de nouveau paquet de données pendant un certain temps, nous avons un timer (de 2 secondes dans notre cas) qui se termine et le dernier acquittement est renvoyé à l'émetteur.

Comme dit précédemment, nous avions testé une solution avec un buffer qui gardait les acquittements pas encore envoyés. Quand c'était possible, nous envoyions tous les acquittements du buffer (du plus ancien au plus récent) puis nous effacions ceux-ci, sauf le plus récent de façon à pouvoir le renvoyer quand le timer se finissait. Bien que cette solution fonctionne, elle ralentissait le réseau, ainsi que l'exécution du programme d'émission. En effet, ce dernier attend la réception d'un acquittement pour

mettre à jour son buffer de paquets à envoyer (ou à ré-envoyer). En recevant plusieurs acquittements l'un après l'autre, il doit les traiter un par un et mettre son buffer à jour à chaque traitement. On perd donc l'intérêt des acquittements cumulatifs. Avec notre solution finale, seul le dernier acquittement préparé est envoyé à l'émetteur, qui ne doit donc mettre à jour son buffer de paquets à envoyer qu'une seule fois (en sachant que les acquittements sont cumulatifs), ce qui lui permet donc d'être plus rapide et de moins utiliser le réseau.

3.3 Gestion de l'endianness

Pour gérer les différences d'endianness et donc d'interprétation des paquets envoyés ou reçus entre machine, nous écrivons octet par octet dans les buffers de flux de données à l'aide de shifts binaires lors du codage et décodage. La machine connaissant sa propre endianness, les opérations de bit shiftings permettent de prendre en compte les différences d'endianness facilement sans devoir utiliser des fonctions telles que `htons` ou `ntohl`. Nous mettons donc le byte de plus grand poids en premier dans le flux de données lors de l'encodage et le premier byte que nous prenons du flux de données lors du décodage est le byte de plus grand poids (et vice-versa). Les fonctions qui s'occupent de cette tâche se trouvent au début des fichiers `packets`.

Remarquons que pour garder le code simple, nous avons décidé d'écrire sur les différents flux de données le champ `timestamp` des paquets dans l'endianness du réseau (comme tous les autres champs de la structure `pkt_t`). Cette décision n'a pas de conséquence sur le bon fonctionnement du programme puisqu'il faut simplement que l'émetteur sache lire et écrire ce champ de façon cohérente (le receveur ne fait que le transmettre sans le lire).

3.4 Arrêt

Pour la gestion de l'arrêt, dès que l'émetteur reçoit `EOF` sur son fichier à transmettre, il met la variable `EOFRead` à `true` et crée un paquet avec une taille de payload à zéro à transmettre au récepteur. Le fichier d'entrée ne sera alors plus lu. L'émetteur continue d'envoyer ses paquets restants jusqu'à ce que tous les acquittements lui confirme qu'ils ont été reçus. Quand le paquet de fin de connexion est envoyé la variable `EOFPktSent` est mise à `true`. Une fois ce dernier paquet acquitté, la variable `EOFPktAck` est mise à `true` ce qui permet à l'émetteur de sortir de sa boucle principale et de quitter le programme.

Si ce dernier acquittement n'est pas reçu (parce qu'il a été perdu sur le réseau par exemple), l'émetteur continue à envoyer le dernier paquet jusqu'à ce qu'un timer de 10 fois le temps de retransmission se finisse. À ce moment-là, l'émetteur considère que l'acquittement a été perdu et arrête son exécution. La valeur de ce timer peut être facilement changée puisque nous l'avons définie à l'aide de `define`.

Dans le cas d'un très mauvais réseau, il est donc possible que ce dernier paquet, bien qu'envoyé 10 fois, ne soit pas reçu par le récepteur. Dans ce cas, la prochaine fois que celui-ci voudra lire sur le socket pour recevoir un paquet, la permission lui sera refusée et le programme s'arrêtera.

Dès que le récepteur essaye d'écrire le paquet de fin de transmission sur le fichier de sortie (il ne le fait pas puiqu'il détecte qu'il a une taille nulle), il met sa variable `EOFPktReceived` à `true`. Il arrête alors de lire le socket, écrit le dernier acquittement sur le socket (si celui-ci est disponible) et s'arrête.

4 Paramètres et Vitesse de transfert

4.1 TimeStamp

Nous avons pris la décision de ne pas utiliser ce champ de la structure `pkt_t`, notre programme fonctionne tout simplement bien sans. Ce champ représente cependant une piste d'amélioration pour notre programme, celui-ci permettrait de mesurer les temps d'aller-retour pour les paquets et ainsi de fixer de manière dynamique notre timer de retransmission.

4.2 Timer de retransmission

Nous avons décidé de mettre la valeur du timer de retransmission à la même valeur que le délai maximal du réseau auquel nous pouvons être confronté, à savoir 2 secondes. Un timer de 4 secondes nous aurait

permis de ne retransmettre un paquet que quand on est sûrs que celui-ci a été perdu. Nous avons décidé de façon arbitraire de diviser ce temps par deux, ce qui nous a semblé être un bon compromis pour ne pas congestionner le réseau si celui-ci est mauvais, et ne pas attendre trop longtemps avant de retransmettre un paquet qui aurait été perdu sur un bon réseau. Pour trouver une meilleure valeur, il nous aurait fallu avoir une distribution statistique du réseau que nous utiliserons. Dans notre cas, nous devions implémenter un protocole qui marcherait sur tout réseau ayant un *round-trip time* de moins de 4 secondes. Remarquons que ceci n'empêchera pas notre programme de fonctionner sur un réseau avec un plus grand *round-trip time*, il sera simplemant moins optimisé.

Nous pouvons cependant facilement changer la valeur de ce timer puisque nous l'avons définie à l'aide d'un `define`.

Une piste d'amélioration serait de mettre à jour dynamiquement la valeur de ce timer par exemple en utilisant le champ timestamp pour savoir si un paquet à été reçu ou non, ou si le réseau est congestionné.

4.3 Vitesses de transfert

La vitesse de transfert des paquets dépend énormément de la vitesse et de la qualité du réseau. De notre côté nous pouvons jouer sur le temps de retransmission et la taille de la fenêtre. La taille de la fenêtre étant très limitée (32) et dépendant indirectement de la taille de la fenêtre, on ne s'intéresse qu'à l'évolution du temps de transmission pour des fréquences de retransmission différentes. Le temps

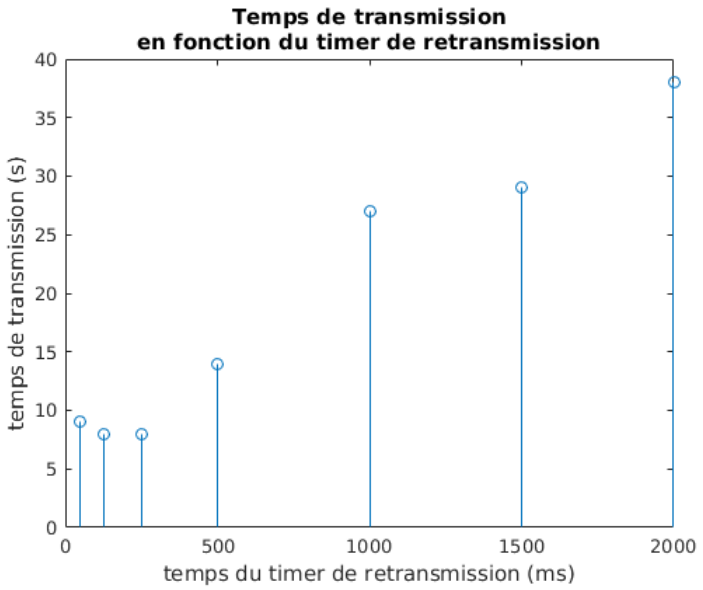

Figure 1: fichier de 1Mo, délai de 50ms, taux de perte 1%

de transmission est linéaire par morceau, stable à la valeur du délai et linéaire éloigné. La linéarité est facile à comprendre les temps testés sont presque à des échelles différentes du temps de délai, le temps de retransmission domine donc le temps total de transmission. Au contraire proche du temps de délai le temps de transmission se stabilise car le délai de transmission devient dominant (et il est constant).

5 Tests

Avant de faire un programme dédié au test de notre programme, nous avons fait beaucoup de tests "à la main" en transférant des fichiers ou des chaines de caractères entrées sur `stdin` avec notre programme, aussi bien en simulant un réseau parfait qu'en simulant des pertes, délais et erreurs à l'aide du programme `link_sim` qui nous a été fourni.

Nous avons ensuite écrit un script `bash` qui lance plusieurs fois les programmes en transférant différents fichiers sur des réseaux simulés avec différents paramètres. Le programme `link_sim` qui simule les réseaux utilise des fonctions aléatoires, ce qui fait qu'à chaque fois que notre script est lancé,

les tests sont légèrement différents. On peut lancer ces tests directement en lançant le script ou grâce au `makefile` avec la règle `tests`. Enfin, nous avons testé notre code avec le script `bash` fourni sur Moodle, qui fait le même genre de tests que notre script.

Nous avons lancé ces tests un certain nombre de fois et n'avons jamais eu d'erreur de transfert, nous pouvons donc avancer que notre programme fonctionne au moins dans la majorité des cas.

Remarquons tout de même que la fonction `getaddrinfo` utilisée pour résoudre un nom d'hôte en adresse IP (version 6) nous renvoie très souvent une erreur sur nos ordinateurs personnels. Nos tests ont donc été lancé en spécifiant des adresses plutôt que des noms d'hôtes.

6 Conclusion

Les objectifs sont atteint notre code est fonctionnel même avec des connexions de mauvaise qualité selon nos tests et le code/rapport est lisible.

Outre les améliorations déjà citées dans ce rapport, nous pourrions en donner d'autres possibles mais ne faisant pas partie du projet. L'introduction d'acquittements sélectifs serait intéressante, notre code serait prêt à la recevoir, ceux-ci amélioreraient grandement la vitesse de transfert sur les connexions à fort taux de perte de paquets mais avec une fenêtre de réception importante. Une gestion dynamique de la congestion du réseau serait aussi possible via `timestamp` et une mesure du taux de perte.

A Tests d'interopérabilité

Les tests d'interopérabilité que nous avons faits se sont plutôt bien passés. Nous avons tenté de transférer plusieurs gros fichiers dans les deux sens en utilisant notre programme pour un des deux interlocuteurs et le programme d'un autre groupe pour l'autre interlocuteur. Nous avons effectué ces tests avec 3 groupes différents.

Après que les tuteurs aient réglé un problème avec les ports des ordinateurs des salles Intel, nous avons pu voir que le transfert se faisait sans problème dans les deux sens avec deux des trois groupes. Un de ces groupes avait par contre un problème avec leur programme d'envoi.

Les fichiers que nous nous sommes transférés comportaient notamment des fichiers de 1 Mo. Pour vérifier que les fichiers étaient bien les mêmes après le transfert sur les deux ordinateurs différents, nous utilisions le programme `md5sum` qui calcule la somme *md5* du fichier.

Le seul problème que nous avons rencontré était le fait que lorsque notre récepteur créait un fichier sur les ordinateurs des salles Intel, celui-ci n'avait pas les bonnes permissions. Ainsi, en tant qu'utilisateur, nous n'avions pas la permission de lire ou d'écrire dans le fichier que notre programme venait de créer, chose qui n'est jamais arrivé en utilisant nos ordinateurs personnels. Nous avons donc ajouté les flags `S_IRUSR` et `S_IWUSR` dans l'appel de la fonction `open` afin que l'utilisateur ait bien ces deux permissions quel que soit l'ordinateur utilisé.

À part cela, notre programme fonctionnant sans aucun problème, nous n'avons donc pas jugé utile d'effectuer d'autres changements sur le code.