LINGI1341 - Rapport du premier projet

Minh Phuong Tran Gilles Peiffer

22 octobre 2018

1 Introduction

Pour ce projet, il nous a été demandé de réaliser un protocole de transport en C utilisant UDP. Il doit être fiable, en fonctionnant en *selective repeat* et avec IPv6.

Deux programmes devaient être écrits pour ceci :

— `receiver`, détaillé à la section 2.1 ;

— `sender`, détaillé à la section 2.2.

Ce rapport doit permettre au lecteur de comprendre la philosophie de conception utilisée pour ce projet, ainsi que de se faire une idée des performances du projet et des tests mis en œuvre pour valider l'exactitude des résultats.

2 Structure générale

2.1 `receiver`

Le `receiver` commence en interprétant les arguments donnés au programme, puis ensuite tente d'établir une connexion avec le `sender`. Ensuite, il commence à recevoir les paquets envoyés par ce `sender` en écrivant leurs données sur la sortie spécifiée (soit la sortie standard, soit un fichier passé en argument avec l'option `-f`). Après chaque paquet reçu correctement, le `receiver` envoie un `ACK` de sorte à notifier le `sender` de la réception correcte de ce paquet. Le `receiver` utilise une stratégie de *selective repeat*, et est donc capable de sauvegarder des paquets reçus hors séquence, et d'éviter leur retransmission en utilisant des acquis cumulatifs.

Lorsque le `receiver` reçoit un paquet avec un champ `length` nul, dont le numéro de séquence est égal au numéro de séquence du dernier acquis envoyé, il sait qu'il doit terminer la transmission.

Après cette méthode `main`, un appel à la fonction `receive_data` permet d'effectuer la plupart des opérations nécessaires sur le paquet :

— vérification de l'exactitude des informations (CRC correct, numéro de séquence dans la fenêtre de réception,...) ;

— récupération des données utiles (le *payload*) ;

— appel aux *fonctions d'acquittement* et d'écriture ;

Afin de pouvoir recevoir des données "dans le mauvais ordre", le `receiver` utiliser un buffer de réception basé sur le principe de fonctionnement d'une file de priorité minimale telle qu'implémentée par Olivier Tilmans dans son simulateur de lien. Grâce à cette structure de données, il est possible d'effectuer de façon efficace les opérations de `push` et `pop` pour écrire les données sur la sortie spécifiée. Il a fallu modifier légèrement cette structure afin d'éviter de dupliquer des paquets du coté du `receiver`.

Finalement, l'envoi des acquittements se fait selon la consigne du document `statement.pdf` fourni sur le site Moodle du cours.

2.2 `sender`

Le `sender` prend comme argument le domaine ou l'adresse IPv6 du `receiver` ainsi que le port UDP sur lequel celui-ci écoute. Il prend aussi comme argument optionnel un fichier contenant les données à envoyer au receiver (sans lequel, les données sont lues directement sur `stdin`).

Afin de pouvoir envoyer des données au `receiver`, nous avons divisé le programme en plusieurs étapes :

— la connexion avec le `receiver` ;

— la lecture des données à envoyer et l'encodage de ces données dans des structures `pkt_t` ;

— l'envoi de ces `pkt_t` suite à une reconversion en une chaîne de caractères ;

— la gestion des ACK et des NACK et le réenvoi du paquet si aucun ACK correspondant n'a été reçu avant l'intervalle de temps TIMEOUT ;

— la déconnexion avec le `receiver`.

Une des fonctions principales dans le `sender` qui mérite d'être mentionnée est la fonction `send_data`. En effet, dans cette fonction, la lecture des données n'empêche pas le fait de recevoir des ACK de la part du `receiver`. Cela permet d'envoyer les données au fur et à mesure en remplissant les paquets à leur capacité maximale sans avoir à lire toutes les données avant de les diviser en paquets pour les envoyer ensuite.

Afin de réaliser cette tâche, nous avons choisi d'implémenter une *queue* FIFO de paquets. Ainsi, nous ne sommes pas obligés de connaître le nombre de paquets à envoyer à l'avance.

La gestion des ACK et des NACK ainsi que la déconnexion se font selon les consignes.

3 Questions

3.1 Utilisation du champ `timestamp`

Pour la soumission finale, nous avons décidé de mettre le temps de la fonction `clock.gettime` dans le timestamp, nous voulions que le `receiver` vérifie cette valeur afin d'ignorer les paquets trop anciens. Malheureusement, nous nous sommes rendus compte que les autres groupes n'utilisent peut-être pas leur timestamp de la même manière, ce qui empêcherait l'interopérabilité. Actuellement, nous n'utilisons donc pas la valeur retournée.

Nous aurions pu par contre l'utiliser pour fournir une estimation du *round-trip time* afin de mieux calibrer notre temps de retransmission.

3.2 Réception d'un packet de type `PTYPE_NACK`

Lorsque le `sender` reçoit un packet de type `PTYPE_NACK`, il envoie de nouveau les paquets que le `receiver` a ignorés à cause de leur troncature.

Nous avons implémenté l'algorithme "*Additive Increase, Multiplicative Decrease*" tel que celui présenté dans les notes de cours afin de gérer les cas de congestion. En effet, lorsqu'il y a congestion, le `receiver` reçoit des paquets tronqués par le réseau. Suite à cela, le `receiver` modifie la taille de sa window (la divise par 2) et renvoie un NACK au `sender`, qui lui réajuste sa window en fonction. S'il n'y a pas de congestion, le `receiver` augmente la taille de sa window de 1 à condition qu'elle ne dépasse pas la valeur maximale, à savoir 31.

3.3 Valeur du *retransmission timeout*

Comme dit à la section 3.1, la valeur du retransmission timeout pourrait être ajustée de manière intelligente en utilisant la valeur du champ `timestamp` des paquets. Pour cela, il faudrait que le sender calcule le RTT afin de diminuer ou d'augmenter le retransmission timeout pour optimiser le *throughput* du protocole.

Cependant, dans notre projet, cette fonctionnalité n'est pas encore implémentée, et notre retransmission timeout a une valeur fixe de 200 ms. Cette valeur a été choisie d'après nos recherches sur Internet.

Nous avons cependant ajouté une *fast retransmission* à notre `receiver`. En effet, lorsque le `sender` lui envoie un paquet dont le champ `seqnum` est en dehors de la window du `receiver`, celui-ci renvoie un ACK avec comme seqnum le seqnum du début de la window du `receiver`.

3.4 Partie critique de l'implémentation

Pour la deuxième soumission, nous nous sommes rendu compte que le RTO affecte le plus notre performance. Une façon d'améliorer ce comportement serait d'utiliser le champ `timestamp` pour calculer le *round-trip time* de sorte à mieux calibrer le RTO par la suite.

Nous avons également ajouté un seuil d'inactivité. En effet, si le `receiver` perçoit que le `sender` (ou vice-versa) n'est pas actif depuis un laps de temps supérieur au seuil, celui-ci se déconnecte automatiquement.

3.5 Stratégies de test

Pour tester notre implémentation, les tests INGInious ont été très utiles : en effet, ils ont permis de tester l'exactitude de quelques parties de notre programme, auxquelles nous n'avons par la suite pas apporté beaucoup de changements :

— la récupération d'arguments en ligne de commande avec la fonction `getopt` grâce à l'exercice "Interpreter des arguments en ligne de commande" ;

— l'encodage et le décodage des structures avec l'exercice "Encoder et décoder des structures" ;

— la connexion au *socket* et l'échange de données pour l'exercice "Envoyer et recevoir des données" ;

— les fonctions de gestion des paquets grâce à l'exercice "Format des segments du projet de groupe".

Par la suite, des outils plus avancés tels que `cppcheck`, `valgrind` et `gdb` ont été utilisés pour débugger notre code et pour trouver les fuites de mémoire que nous avons toutes corrigées.

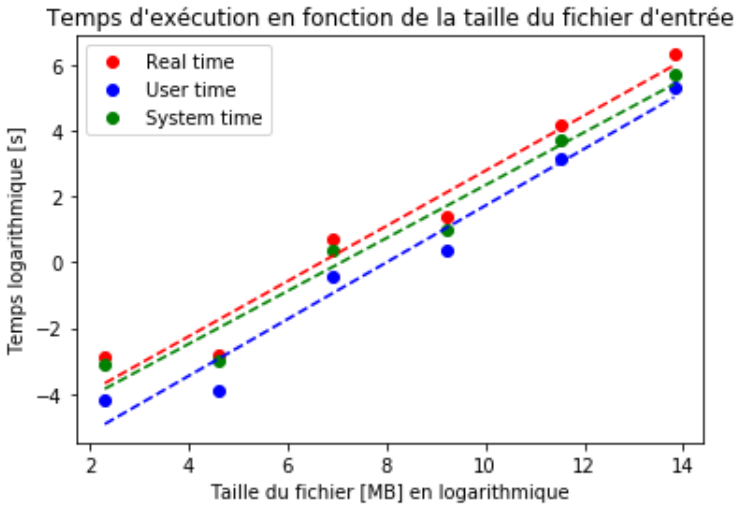
3.5.1 Simulateur de liens

Nous avons également utilisé le simulateur de liens proposé sur Moodle afin de tester notre implémentation.

Nous avons inclus dans notre dossier de tests quelques variations au fichier `test.sh` fourni, de sorte à jouer sur les paramètres suivants : le délai, les pertes, la corruption, la troncature et le jitter.

3.5.2 Test unitaires

Afin de tester les blocs atomiques de notre projet, nous avons écrit quelques tests unitaires, autant pour le `sender` que pour le `receiver`. Comme attendu, ces tests s'exécutent correctement. Ils sont exécutables avec `make tests`.



3.6 Interopérabilité

3.6.1 Groupe Gobeaux - Semerikova

Nos deux programmes étaient presque compatibles dès le premier test. Nous avons du enlever le calcul fait avec le timestamp puisque l'autre groupe ne l'utilisait pas de la même manière que nous. Après ce petit changement, le test d'interopérabilité est passé sans souci, autant dans un sens que dans l'autre.

3.6.2 Groupe Mulders - Reniers

En testant avec ce groupe, nous avions eu au départ des problèmes avec leur `receiver`. Après une inspection de leur code, et un petit fix au calcul de la window, cet échange aussi s'est passé sans problèmes sur les machines de la salle Intel.

3.7 Performances

Grâce à l'utilisation d'une file de priorité minimale, nous arrivons à effectuer les opérations de `push` et `pop` en $\mathcal{O}(\lg n)$, et l'opération `peek` en $\mathcal{O}(1)$.

Lors de nos tests, le programme est capable de transmettre plusieurs megabytes en moins d'une seconde, permettant notamment de transférer rapidement des vidéos.

Afin de mieux quantifier la performance de nos programmes, nous avons analysé le temps d'exécution en fonction de la taille du fichier d'entrée (avec un délai de 10 ms, un jitter de 10 ms, une perte de 10 %, un taux d'erreur de 10% et un taux de troncation de 10%).

4 Conclusion

Après un petit *wake-up call* pour la première soumission, nous n'avons pas pris de pause après celle-ci, de sorte à avoir un projet majoritairement fonctionnel lors des tests d'interopérabilité. Par la suite, nous avons découvert quelques erreurs supplémentaires en jouant sur les paramètres de `test.sh`. Ces problèmes ont permis d'améliorer de façon itérative notre projet, de sorte à le rendre plus que respectable en fin de course.