École Polytechnique de Louvain

LINGI1341 - Réseaux informatiques

Groupe 111 - SENDER

Projet TRTP

2019-2020

Hadrien Libioulle : 10041700
Louis Laurent : 67081700

1 Architecture

Notre programme commence dans la classe **main.c** qui s'occupe de gérer les arguments et de lancer la fonction *scheduler()* dans notre classe **sender.c**. *scheduler()* est l'armature du programme et s'occupe d'appeler les autres fonctions dans le bon ordre :

- On a d'abord un appel à *init()* qui crée et connecte le socket, et initialise les variables globales.
- Ensuite, le gros du programme démarre :
  - un while qui boucle tant que toute l'input n'a pas été lue. On commence la boucle par un appel aux fonctions *emptySocket()*, *removeFromSent()* et *resendExpiredPkt()*. La première lit et traite les paquets reçus, la seconde s'occupe de mettre à jour notre fenêtre d'envoi contenant les paquets à renvoyer si ils expirent, et la dernière parcourt cette même fenêtre et renvoie les paquets dont le timer de retransmission a expiré.
  - Ensuite, si la fenêtre de réception du receiver le permet, des données sont lues dans l'input, encodées et envoyées sur le réseau.
  - Lorsque l'input est vide, on vide le buffer correspondant à la fenêtre d'envoi et on envoie le dernier paquet avec un champ *length* set à 0. On s'assure de la bonne réception du dernier paquet puis on return dans la fonction *main()* qui va fermer les file descriptors et libérer la mémoire.

2 Choix d'implémentation

2.1 Échec de la connexion avec le receiver

En cas d'échec de connexion avec le receiver dans *init()*, on renvoie le paquet 0 pendant 30 secondes (délai de deadlock) toutes les secondes (délai de retransmission). Si le *receiver* finit par répondre, alors l'échange démarre comme si le *receiver* avait répondu depuis le début.

2.2 Utilisation des timestamps

Les timestamps sont utilisés pour vérifier qu'un paquet reçu ne s'est pas perdu sur le réseau et ne nous est parvenu que bien plus tard. On peut imaginer qu'un paquet de type PTYPE_ACK avec un numéro de séquence jugé comme adéquat soit reçu, mais qu'il soit trop vieux c'est-à-dire un ancien paquet d'il y a plusieurs cycles. On vérifie donc que les paquets reçus soit au moins plus récents que le premier paquet dans notre fenêtre d'envoi.
Pour une amélioration future, on peut imaginer une utilisation des timestamps qui permettrait de fixer la valeur du timer de retransmission d'un paquet afin d'optimiser les performances. Pour l'instant notre timer de retransmission a une valeur fixe de 1 seconde.

2.3 Choix du délai de retransmission

Au début, celui-ci est de 1s. Ce choix provient de tests : un délai plus long a tendance à ralentir la connexion sauf si le délai est plus élevé que 1s. Par exemple, si le délai est de 2s (sans cut rate, loss rate ou error rate), un timer de retransmission de 1s ou 3s n'impacte pas les performances. Le délai impacterait les performances dans ce cas-ci, s'il y avait des pertes, corruptions, ... Nous avons estimé qu'il fallait privilégié les performances sur un réseau avec moins de 1 secondes de délai que de mieux gérer le cas extrême d'un réseau avec beaucoup de délai, pertes, corruptions, ... Idéalement, le délai devrait être choisi dynamiquement pour être légèrement supérieur au délai moyen du réseau et pour tenir compte des pertes/corruptions/toncations.

2.4 Réaction à un NACK

Lorsque l'on reçoit un paquet de type PTYPE_NACK dans *EmptySocket()* et qu'il est présent dans notre fenêtre d'envoi, il est immédiatement renvoyé.

2.5 Retransmission rapide de paquets

Lorsque l'on vide le socket dans *EmptySocket()*, si l'on reçoit plusieurs paquets de type PTYPE_ACK (plus de 2 pour être exact) avec le même numéro de séquence, on le renvoie directement avant de continuer à vider le buffer. Cela permet d'augmenter drastiquement les performances : sans cette fonctionnalité, le paquet "bloquant" qui n'a pas été correctement reçu par le receiver va nous empêcher d'envoyer un nombre optimal de paquet pendant la prochaine itération de la boucle.

2.6 Fermeture de la connexion

Pour fermer la connexion, on envoie tous les paquets sauf le dernier, on vide le buffer qui correspond à la fenêtre d'envoi, c'est-à-dire on reçoit tous les acknowledgements pour ces paquets. On envoie ensuite le dernier paquet et on attend que sa réception soit confirmée par le receiver. Normalement, le dernier paquet devrait être envoyé avec tous les autres paquets. Ensuite, on attendrait la confirmation de réception de tous les paquets en même temps, mais pour une raison inconnue, cela nous causait une erreur. Nous avons opter pour un quick fix pour des raisons de temps.

3 Tests

3.1 Valgrind

L'exécution de valgrind entre dans une VM Ubuntu vers la machine ***Utapau*** n'a montré aucune fuite mémoire (sauf des *invalidReads* liés à getaddrinfo).

3.2 Tests unitaires CUnit

Nous avons réalisé une série de tests CUnit pour les fonctions élémentaires. Ceux-ci testent :

- *addToBuffer()*
- *isUsefulAck()*
- *getFromBuffer()*
- *resendExpiredPkt()*
- *removeFromSent()*

Pour les réaliser un certain nombre de fois, il est utile d'utiliser la commande suivante :

Listing 1 – Répète la commande 60 fois toutes les secondes

```
#!/bin/bash
for n in {1..60}; do ./tests.out; sleep 1; done
```

Occasionnellement, certaines assertions de CUnit échouent mais cela est dû à des cas limites mal vérifiés dans les tests plutôt qu'à une erreur de la fonction testée.

3.3 Tests manuels

De plus nous avons réalisé une série de tests manuels qui impliquaient de lancer notre *sender* avec le *receiver* de référence, en premier lieu sur nos laptops, puis entre nos laptops et les machines de la salle Intel [1] et enfin entre des machines de la salle Intel. Dans les 3 cas, nous avons lancé les 2 programmes sans le simulateur de liens puis avec celui-ci. Lorsque nous avons utlisé le simulateur de liens, nous avons testé plusieurs configurations :

- Uniquement du **délai** ;

1. Nous avons utilisé les machines ***Utapau***, ***Hoth Palpatine*** et ***Amidala***

- Uniquement **délai** et **jitter** ;
- Uniquement un **cut rate** non nul ;
- Uniquement un **error rate** non nul ;
- Uniquement un **lost rate** non nul ;
- Une combinaison de tout

Lors du test où l'on combine tous les paramètres de *link_sim*, Nous avons veillé à ne pas mettre des paramètres réseau trop extrêmes dans link_sim pour que les tests soient d'une part réaliste et d'autre part qu'ils se terminent dans un temps raisonnable ($cut\ rate + loss\ rate + error\ rate \leq 10\%$ et un délai ≤ 200 [ms] avec un jitter ≤ 20 [ms]). Nous avons testé d'envoyer le fichier *medium_input.txt* avec un délai de 2000 [ms] et un jitter de 20 [ms] ce qui met en moyenne 4 minutes 30 secondes. Nous n'avons donc pas jugé utile de mettre ces tests dans les graphes car ils sont particulièrement lents (théoriquement, il n'est pas possible d'aller beaucoup plus vite car le *sender* passe son temps à attendre étant donné que les réponses mettent 2 secondes à arriver ce qui est largement supérieur au temps mis pour envoyer une fenêtre de 31 paquets).

Les tests manuels que nous avons exécutés se sont donc révélés concluants, nous permettant de mettre en avant quelques erreurs (notamment pour l'implémentation de la gestion des timestamps dans les ack. qui ne fonctionnait pas ainsi que des soucis de performances dans certains cas). Finalement, nous avons corrigés toutes les erreurs que nous avons pu déceler (erreurs lors de l'envoi du dernier paquet, attente inutile dans notre 1ère implémentation liée à un mauvais algortithme d'envoi, ...) avec ces tests.

4 Performances

Pour le graphes montrant les performances, nous nous sommes limités à 3 tests dans 2 cas (Sur 1 même machine et sur 2 machines différentes). Tests effectués :

1. Réseau parfait
2. 5% de perte, 5% de corruption et 5% de troncation
3. 20ms. de délai et 10ms. de jitter avec 5% de perte, 5% de corruption et 5% de troncation

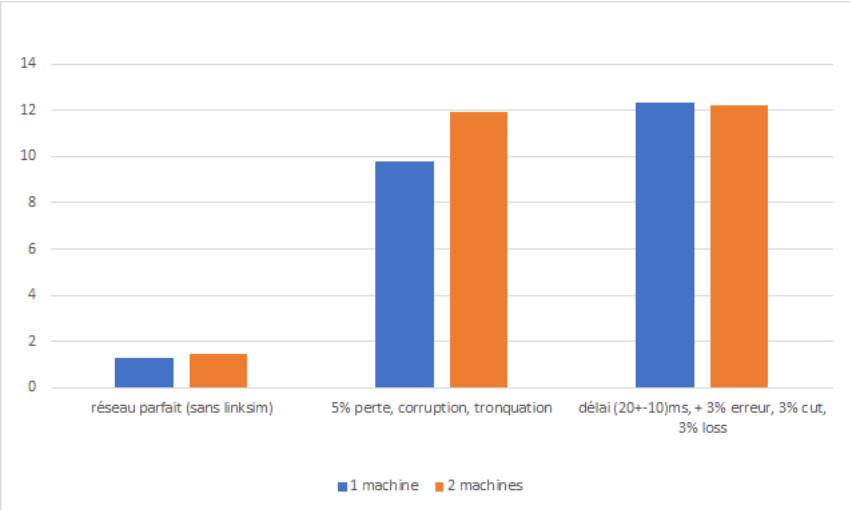

FIGURE 1 – Graphes montrant les performances

5 Annexe

5.1 Interopérabilité

Nous avons effectué des tests d'interopérabilité avec les groupes 70, 92 et 138 ainsi qu'avec le *receiver* de référence. Les tests se sont avérés concluants et nous ont permis de mettre en lumière quelques problèmes :

- Le dernier paquet (avec Length = 0 et donc sans payload) possédait un CRC_2 ce qui n'était pas correct.
- Nous ne gérions pas le timestamp provenant d'un acknowlegment.

Sinon globalement, notre *sender* était bien interopérable avec ces 3 groupes (avec des vitesses qui varient significativement car celles-ci dépendent fortement de la "compatibilité" des algorithmes d'envoi/réception du *sender* et *recevier* utilisés).

5.2 Changements apportés au code depuis la première soumission

- Ajout de l'utilisation des timestamps provenant des paquets de type PTYPE_ACK.
- Les tests CUnit peuvent (*enfin*) être utilisés sur les machines de la salle Intel (précédemment, il y avait des *segmentation faults* ou des erreurs de *free()*)
- Gestion d'un receiver qui ne répondrait pas tout de suite (le sender réessaye l'envoi du premier paquet pendant 30 secondes).
- Suppression du CRC_2 que l'on envoyait avec un paquet sans *payload*