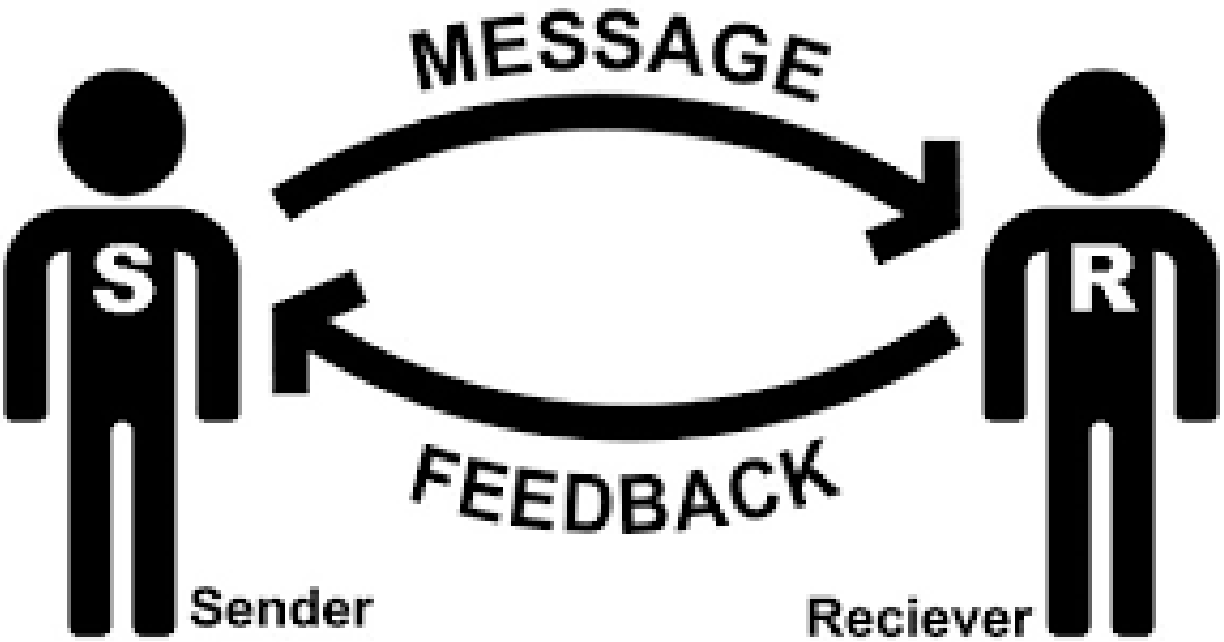
LINGI1341

Rapport Projet TRTP

22 octobre 2018



Loïc Quinet (2808-1600)
Jonathan Thibaut (2454-16-00)

Année académique 2018-2019

1 Introduction

La société Crousti-Croc a installé récemment un datacenter dans la région. Afin de lui transmettre toutes les données utiles à son utilisation, l'entreprise a besoin d'une implémentation correcte en langage C. Cette implémentation doit répondre à certains critères comme par exemple, aucune fuite de mémoire, utilise la stratégie d'un selective repeat, et permettant la troncation des payloads. A la demande de l'entreprise, le système doit être performant sur IPv6.

Pour la satisfaire, nous avons donc créé une implémentation de cette transmition de données. Dans le suite de ce rapport figureront nos choix de conception et comment nous avons construit notre implémentation. Mais aussi, les difficultés rencontrées, les performances de notre programme et l'évaluation de ce dernier.

2 Construction de l'implémentation

Les demandes de la société étaient très claires. Et nous avons essayé d'y répondre du mieux possible.
Tout d'abord, nous avons utilisé les moyens mis en place pour nous aider à démarrer le projet, c'est à dire Inginious, pour apprendre à utiliser les fonctions nécessaires à la création du sender et du receiver. Une fois cette tâche réussie, nous étions capables de créer une conversation entre un sender et un receiver sans accusé de réception. De plus, une deuxième tâche nous a permis de créer la structure de données utile pour envoyer des packets ainsi que remplir ces derniers, les encoder dans un buffer pour les envoyer ou encore décoder un buffer reçu et remplir la structure correctement.
Une fois ces deux tâches réussies nous avons pu commencer à créer notre implémentation du sender et receiver. Pour ce faire, nous avons décidé de les créer en plusieurs étapes.
Pour commencer, nous avons simplement envoyé des données du sender vers le receiver sans accusé de réception ou renvoi des données. Une fois cette étape réussie, nous avons ajouté des accusés de réception. Ce n'est que par la suite que l'implementation du selective repeat rentre en compte ainsi que le renvoi des données. Pour tester si notre code fonctionnait, nous avons ouvert deux terminaux et avons exécuté la fonction sender dans l'un et receiver dans l'autre. Nous avons alors mis des printf à chaque étape effectuée pour voir si toutes les étapes se passaient correctement.
Pour ce qui est de la troncation des données, nous utilisons la fonction crc32 qui permet de calculer le crc d'une donnée. Ce crc est contrôlé à chaque envoi et réception d'un packet. Chaque packet envoyé contient minimum un crc qui sécurise le header de ce dernier. Si le packet contient un payload, alors un deuxième crc est calculé et permet de s'assurer que le payload reçu est correct. Il nous est donc possible de transmettre des données sans craindre les troncations. Par exemple, si le crc calculé est différent de celui reçu, le packet sera automatiquement supprimé et un packet PTYPE_NACK sera envoyé pour signifier que le packet est tronqué.

3 Utilisation de Timestamp

Dans les packets envoyés, plusieurs champs sont indispensables comme le type, la fenêtre ou le numéro de séquence. Mais parmi ceux-ci se trouve aussi un champ Timestamp, qui n'a pas encore de fonctionnalité pré-définie. Il nous est donc laissé libre pour l'utiliser comme bon nous semble. Plusieurs idées nous sont venues en tête quant a l'utilisation de ces 32 bits.

Nous avons tout d'abord pensé à y mettre une deuxième fois un des crc. Par exemple celui

du payload s'il y en a un. Nous pensons que cette solution peut avoir un intérêt dans le sens où si un des bits du crc venait à être modifié lors de l'envoi, ce dernier ne serait plus correct et le packet devrait être supprimé. Mais si le timestamp contient aussi le crc, il est possible de comparer ce dernier et finalement se rendre compte que le payload reçu est correct. Il ne nous faudrait dès lors pas renvoyer le packet.

Une deuxième idée est d'utiliser le timestamp pour agrandir le payload envoyable de 4 octets. Mais cette idée ne nous semble pas très utile. En effet, si le payload est trop grand, il suffit de le séparer et de l'envoyer en plusieurs packets. Surtout que le timestamp ne fait que 4 octets ce qui n'est rien à côté des 512 du payload.

Pour conclure, nous préférons l'utilisation du timestamp comme deuxième crc, ce qui nous permet de diminuer le nombre de renvois. En effet, nous pensons que cette solution nous rapportera plus que d'envoyer des plus longs payloads. Malgré tout, cette solution ne sera codée que lorsque l'ensemble du projet sera achevée.

4 Réception d'un packet PTYPE_NACK

La réception d'un tel packet signifie que le packet n'est pas arrivé correctement au receveur et qu'il a été tronqué. Nous savons dès lors que le packet a été supprimé par le receveur et que ce dernier attend toujours de le recevoir. Dans ce cas, deux solutions s'offrent à nous. Soit, nous attendons que le timer de renvoi automatique arrive à 0 et renvoie le packet correspondant au numéro de séquence reçu dans le PTYPE_NACK, soit nous remettons le timer à zéro et renvoyons directement le packet souhaité. Pour une question d'optimisation, c'est la deuxième solution qui est préférable. C'est donc cette dernière que nous avons mise en pratique dans notre code.

5 Valeur de retransmission du timeout

Dans le cas où un packet est perdu, un timer de retransmission de ce dernier est inclus dans notre code. Le choix de la durée de ce timer est très important. Un timer trop court renverra le packet alors que celui-ci avait bien été reçu et que l'accusé de réception allait être envoyé ou est envoyé. Par contre un timer trop long peut bloquer le transfert de données en attendant que ce packet arrive. Il est donc primordial de bien le choisir.
Pour ce faire, nous nous sommes renseignés sur le débit d'une connexion ADSL standard qui est de plus ou moins 1,2 Mbits/s en upload et 7,2 Mbits/s en download (données trouvées sur le site : https ://www.ariase.com/fr/vitesse/observatoire-debits.html le 20-10-2018). De plus, le packet maximal que nous pouvons envoyer est de 4224 bits et la réponse maximale sera de 96 bits car cette dernière ne contient aucun payload. En supposant que le temps de transmission est de 10 millisecondes et sachant que c'est le temps d'upload qui détermine la vitesse de l'envoi, nous obtenons un RTT de 0,0236 secondes. Comme tous les réseaux n'ont pas le même débit, nous allons prendre un coefficient de sécurité de 300 pourcent. Ce qui nous donne un RTT de 0,0708. Nous mettrons dès lors notre timer à 0,0708 ce qui permet l'envoi de données sur la majeur partie des wifis.

6 Partie critique du code pour la vitesse de ce dernier

Pour ce qui est de la partie critique de notre implémentation, nous avons pu remarquer que le temps mis pour faire la connexion entre le sender et le receiver et les autres fonctions

utiles pour l'envoi de données ne prenait que très peu de temps à s'exécuter. Nous sommes donc arrivés à la conclusion que ce qui prennait le plus de temps dans notre code devait être le recopiage par memcpy de toutes les données des buffers vers les packets et inversément. En effet, à chaque encodage et décodage, l'ensemble des données à transmettre ou transmises ont été recopiées deux fois minimum. Il en va de soi que pour de nombreux packets, cette tâche demandera un certains temps pour être exécutée.

7 Performance de notre code

Pour tester les performances de notre code, nous avons décidé de le faire en envoyant des données. Pour commencer, nous allons envoyer un certains nombres de bytes et enregistrer le temps mis pour qu'ils arrivent à destination. Nous allons répéter ce scénario plusieurs fois et ce en augmentant à chaque fois le nombre de bytes envoyée. Nous pourrons par la suite analyser la complexité de notre algorithme.

Nombre de bytes envoyés	temps [ms]
5928	6
11858	7
17788	16
23718	27
29643	28
41508	41
53368	38
65228	42
77988	46
88949	50
100808	44
112668	45
124528	45
148248	42

Comme nous pouvons le voir, le temps nécéssaire à la transmission des packets n'est pas constant. En effet, il se peut qu'envoyer un plus grand nombre de bytes prennent moins de temps. Cela peut s'expliquer par la perte de packet. En effet, il se peut que lors d'un test, aucun packet ne soit perdu, tandis que pour un autre certains se perdent et doivent par conséquent être renvoyés. Ce qui implique un plus grand délai. Il est donc difficile de pouvoir tirer des conclusions sur la complexité de notre code. Malgré tout, nous pouvons remarquer que le temps de transmission a tendance à se stabilisé lorsque le nombre de bytes échangés devient élevé. Ce qui est logique car la proportion de packet perdu a tendance à s'uniformiser plus le nombre de packet envoyé est grand.

8 Stratégie de test utilisée

Tout d'abord, pour débugger notre code, nous avons utilisé des printf. L'usage de ces derniers nous permet de vérifier que notre code passe bien par toutes les étapes nécessaires à son bon fonctionnement, tant chez le sender que chez le receiver.
Ensuite, à chaque avancée, nous testions que l'envoi et la réception de données fonctionnaient toujours malgré les modifications ou améliorations faites et si cette dernière fonctionne aussi. Ce qui nous permet de tout de suite savoir d'où vient notre problème.

Enfin, quelques tests Cunit ont été créés pour tester le bon fonctionnement des fonctions contenues dans packet.implem.c.

9 Problèmes rencontrées

Au cours de ce projet, nous avons pu remarquer par moment notre difficulté à comprendre le fonctionnement de certaines fonctions. Les man pages bien que relativement bien expliquées et parfois avec des exemples, ne nous ont parfois pas permis de comprendre directement toutes les utilités ou spécificités des fonctions utilisées. Ce n'est qu'après quelques lectures que nos lanternes se sont enfin éclairées pour nous permettre de réussir à implémenter le chat sur Inginious et notre sender et receiver. Nous pensons que ce problème se résoudra par lui-même à force d'utiliser et de lire les man pages des fonctions souhaitées.
Un autre problème rencontré a été de visualiser et de mettre toutes les étapes dans l'ordre mais aussi comment les implémenter. Pour se faire, nous avons d'abord fait un croquis de notre solution avant de l'implémenter et de la tester pour voir si cette dernière était correcte. Fort heureusement, elle nous a bel et bien permis d'envoyer nos packets de données.

10 Pistes d'amélioration

Tout d'abord, n'ayant pas eu assez de temps, nous n'avons pas mis en place la solution proposée plus haut quant à l'utilisation du timestamp. Nous avons donc des données qui sont échangées mais qui ne servent à rien. Nous perdons donc en efficacité.

Ensuite, nous pouvons aussi améliorer notre façon de créer les windows. En effet, pour l'instant, nous avons mis une taille fixe mais la solution optimale serait d'implémenter des windows dynamiques. C'est à dire que nous commençons avec une taille de fenêtre de 1 et si la transmission n'est pas saturée, on augmente la taille de la fenêtre jusqu'à ce qu'elle le soit. Une fois saturée, on la diminue et ainsi de suite pour tendre vers la taille de fenêtre optimale.

11 Conclusion

Pour conclure, notre programme permet d'envoyer des données entre un sender et un receiver. Comme demandé par la société, notre programme contrôle que le header et le payload reçus sont corrects grâce à l'usage de CRC. Si un des deux n'est pas correct, alors le packet est supprimé et il sera renvoyé par le sender. De plus, le selective repeat est implémenté et un temps de retransmission est intégré au programme pour renvoyer les packets perdus. Enfin, lorsqu'un packet est reçu correctement, un accusé de réception est envoyé par le receiver au sender.
La comparaison entre la demande du client et ce dont notre programme est capable nous permet de dire que ce dernier répond à la demande de la société. L'implémentation n'est probablement pas parfaite et les tests effectués ne couvrent pas tout le code mais nous pensons que le code reste fonctionnel et correspond au comportement du programme souhaité par le client.