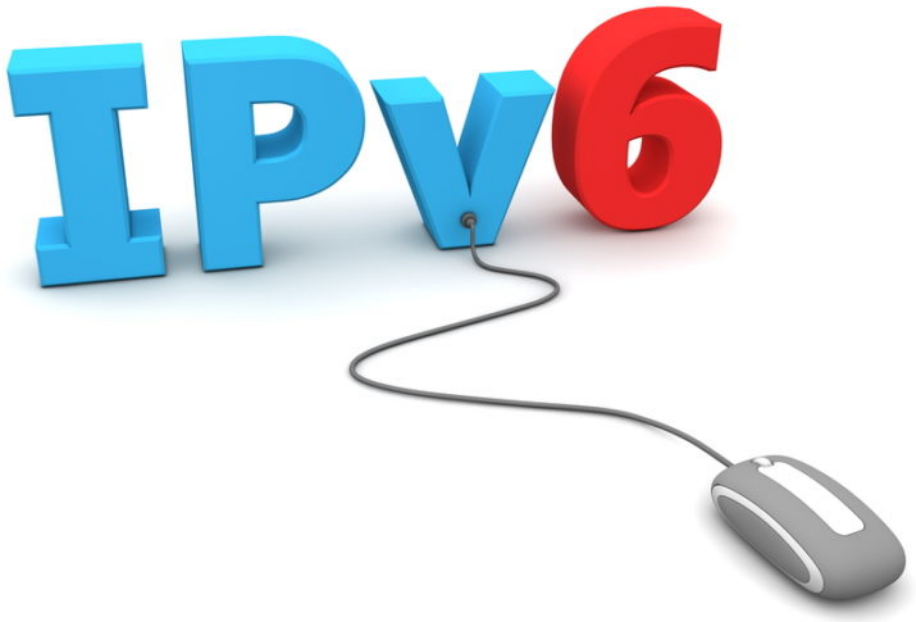
UCLouvain

ÉCOLE POLYTECHNIQUE DE LOUVAIN

LINGI1341 : Réseaux informatiques
Projet TRTP

De Keersmaeker	François	7367-1600
Gérard	Margaux	7659-1600



Louvain-la-Neuve
2 Novembre 2018

1 Introduction

Dans le cadre du cours de réseaux informatiques, nous avons été amenés à développer, dans le langage C, un protocole de transport basé sur des segments UDP. Ce protocole doit être fiable, utiliser la méthode du *selective repeat* et permettre la troncation de payload. Pour réaliser ce protocole, nous avons implementé deux programmes principaux permettant respectivement d'envoyer des données et d'en recevoir sur le réseau. Ce rapport décrit l'architecture générale de notre projet et répond à quelques questions techniques concernant le timestamp, les paquets de type NACK, le retransmission timer mais aussi les tests que nous avons implémenté pour rendre notre programme fonctionnel.

2 Architecture générale

Le projet est constitué de deux fichiers exécutables principaux, `sender` et `receiver`. Ces fichiers et leur fonctionnement seront décrits en détail dans la suite de ce rapport.

2.1 Sender

Le `sender` permet d'envoyer des données sur le réseau, lues soit depuis un fichier, soit depuis l'entrée standard. Pour ce faire, il englobe ces données dans un paquet contenant diverses informations utiles, puis encode ce paquet en binaire avec la fonction `pkt_encode()`. Ensuite, il est en mesure d'envoyer ce paquet sur le réseau grâce à la fonction `sendto()`. Le `sender` doit stocker tous les paquets qu'il envoie dans un buffer de paquets au cas où il serait amené à les renvoyer. Un paquet peut être retiré de ce buffer uniquement lorsque l'acquittement contenant le numéro de séquence de ce paquet à été réçu. Le `sender` a une taille de fenêtre d'envoi définie. Lorsqu'un acquittement est reçu, la fenêtre est décalée, ce qui permet d'envoyer un nouveau paquet sur le réseau.

Si le `sender` ne reçoit pas l'acquittement d'un certain paquet, il doit renvoyer ce dernier lorsque le retransmission time-out est écoulé. Il doit également renvoyer le paquet dans le cas où le `receiver` envoie un paquet de type NACK. En effet, cela signifie que le paquet envoyé a été corrompu lors de la transmission et qu'il ne peut dès lors pas être décodé par le `receiver`.

2.2 Receiver

Le `receiver` permet de recevoir des paquets qui ont été envoyés sur le réseau par le `sender`. Lorsqu'il a reçu un paquet, il le décode grâce à la fonction `pkt_decode()`. Lorsque le paquet reçu a été analysé et s'avère valide, le `receiver` envoie un acquittement au `sender` contenant le numéro de séquence du prochain paquet attendu. Il affiche ensuite les données contenues dans le paquet décodé, soit dans un fichier, soit sur la sortie standard. Comme le `sender`, le `receiver` a une fenêtre glissante de réception des paquets. Lorsqu'un paquet est reçu, la fenêtre se décale. Grâce à la méthode du *selective repeat*, si un paquet hors séquence est reçu, il est stocké si il est contenu dans la fenêtre de réception, sinon il est ignoré.

Si le `receiver` reçoit un paquet tronqué, il envoie un paquet de type NACK au `sender` contenant le numéro de séquence du paquet tronqué. En effet, si le bit de troncage est égal à 1, cela signifie que le paquet a perdu son payload lors de la transmission et qu'il ne contient donc aucune donnée.

2.3 Champs des paquets de données

Les différents champs des paquets de données sont les suivants :

- Type : 2 bits, représente le type du paquet de données (réelles données, acquittement ou non-acquittement).
- TR : 1 bit, est mis à 1 si le paquet a été tronqué.
- Window : 5 bits, indique la taille de la fenêtre de réception de l'émetteur du paquet.
- Seqnum : 1 byte, indique le numéro de séquence du paquet dans le cas d'un paquet de données, ou le numéro de séquence du paquet reçu dans le cas d'un acquittement.
- Length : 2 bytes, inidique la longueur du payload dans le cas d'un paquet de données.
- Timestamp : 4 bytes, signification laissée à notre appréciation.
- CRC1 : 4 bytes, checksum calculé sur le header. Permet de vérifier si le paquet a été corrompu ou non lors de la transmission. La valeur du CRC1 est calculée à l'envoi et à la réception. Si les deux valeurs diffèrent, le paquet est ignoré car il a été corrompu.
- Payload : jusqu'à 512 bytes, représente les données à transmettre.
- CRC2 : 4 bytes, représente le checksum calculé sur le payload. Permet de vérifier si le paquet a été altéré lors de la transmission. La valeur du CRC2 est calculée à l'envoi et à la réception. Si les deux valeurs diffèrent, le paquet est ignoré car il a été corrompu.

3 Timestamp

La signification du champ `timestamp` du header était laissée à notre appréciation. Nous avons donc décidé de remplir ce champ avec l'heure d'envoi du paquet, en millisecondes. Cette information nous permettrait de mettre à jour la valeur du RTO au fil de l'exécution du programme. En effet, lorsqu'on reçoit un paquet ACK pour un certain paquet DATA, on peut calculer la différence entre les deux `timestamp` de ces paquets pour connaître la durée nécessaire pour envoyer un paquet DATA et recevoir le paquet ACK correspondant. Ensuite, on peut définir un RTO légèrement supérieur à cette durée, pour qu'il ait le temps de recevoir le paquet ACK mais qu'il soit le plus efficace possible. Malheureusement, nous n'avons pas eu suffisamment de temps pour implémenter ce raisonnement au sein de notre code.

4 Paquets de type NACK

Lorsque le `receiver` reçoit un paquet tronqué, c'est-à-dire un paquet dont la payload a été perdu lors de la transmission ou corrompu, il va renvoyer un paquet de type NACK au `sender`. Ce paquet aura pour numéro de séquence le numéro de séquence du paquet tronqué. Après la réception de celui-ci, le `sender` doit renvoyer le paquet de données qui portait ce numéro de séquence. Il faut donc que le `sender` garde en mémoire les paquets envoyés tant qu'il n'a pas reçu de paquet de type ACK pour ces paquets de la part du `receiver`.

5 Retransmission Time-out

Nous avons choisi un temps de retransmission standard de 4.5 secondes. En effet, l'énoncé stipule que le temps maximal pour qu'un paquet soit envoyé est de 2 secondes. Le temps d'aller-retour maximal du paquet est donc de 4 secondes. En fixant le temps de retransmission à 4.5 secondes, nous sommes certains de ne jamais dupliquer de paquets. Nous utilisons la fonction `select` pour renvoyer certains paquets lorsque le temps de retransmission expire. Nous aurions pu optimiser le RTO en appliquant l'algorithme décrit dans les notes de cours. Néanmoins, par manque de temps, nous avons préféré nous concentrer sur d'autres aspects de l'implémentation du projet, celui-ci n'étant pas une priorité.

6 Performances

Notre code ne nous permet pas d'envoyer plusieurs paquets simultanément sur le réseau ce qui implique que notre programme n'est pas optimal en terme de rapidité. Néanmoins, nous gérons une grande partie des cas limites qui pourraient se produire sur le réseau en réalité. Notre protocole de transport est donc plutôt fiable. Pour améliorer la rapidité de notre protocole, nous aurions pu utiliser des threads afin d'envoyer plusieurs paquets simultanément sur le réseau.

En ce qui concerne la corruption des paquets, nous utilisons le fonction `crc32` pour appliquer un checksum sur le CRC1 et le CRC2 du paquet reçu par le `receiver`. Nous avons remarqué lors du test avec le simulateur de lien que le calcul des CRC n'était pas très précis. En effet, si un paquet est corrompu sur plusieurs bits, notre code va le détecter et l'ignorer correctement. Ainsi, le `sender` va renvoyer à nouveau le paquet qui avait été tronqué après le temps de retransmission. Néanmoins, lorsqu'un paquet est corrompu sur un seul bit, notre code ne détecte pas la corruption et le `receiver` envoie donc un acquittement tout à fait normal au lieu d'ignorer le paquet.

Pour résoudre les fuites de mémoire auxquelles nous avons fait face durant l'implémentation du projet, nous avons utilisé le programme `valgrind`. Nous sommes parvenus à supprimer toutes les fuites de mémoire dans la fonction `receiver` mais il nous reste quelques petits problèmes dans la fonction `sender`.

7 Inter-opérabilité

Nous avons réalisé des tests d'inter-opérabilité avec plusieurs groupes différents mais nous ne retiendrons que les 2 plus intéressants dans ce rapport.

- **Florian Damhaut et Benoit Loucheur :** au début, leur `sender` ne recevait pas les ACK de notre `receiver`, mais ce problème a été réglé plutôt rapidement. En ce qui concerne le leur, la transmission se déroulait correctement tant que moins de 256 paquets étaient échangés. Le cas échéant, l'ordre des données était corrompu.

- **Soukéina Bojabza et Augustin Delecluse :** comme pour le premier groupe, leur `sender` ne recevait pas les ACK de notre `receiver`. Par la suite, nous avons résolu ce problème et notre code fonctionnait correctement. En ce qui concerne leur code, le `receiver` a un problème de mémoire lorsque nous testons le code en local. Lorsque nous le testons avec le simulateur de lien, le code ne réagit pas correctement lorsqu'il y a des paquets qui sont douteux.

8 Modifications après la première soumission

Lors de la remière soumission, nous n'avions pas eu le temps de tester notre code sur les ordinateurs de la salle Intel. C'est pourquoi notre code fonctionnait en local mais pas en salle Intel. En effet, le `receiver` n'envoyait pas l'acquittement correctement car nous avions un problème au niveau de l'adresse et nous avions pas mal d'erreurs au niveau de la gestion de la mémoire.

Nous avons également ajouté une structure `struct ack` qui permet de gérer plus facilement les paquets de type ACK, ainsi que les fonctions `encode()` et `decode()` correspondantes. Ces dernières sont cependant très proches des fonctions utilisées pour les paquets de type DATA.

9 Tests

Afin de rendre nos codes fonctionnels, nous avons testé chaque fonction individuellement. Par la suite, nous avons réalisé un test général reprenant l'ensemble de nos fonctions. Il permet de simuler l'envoi d'un paquet sur le réseau par le `sender`, la réception de celui-ci par le `receiver` et l'envoi d'un acquittement par le `receiver`. Pour ce faire, nous avons connecté 2 terminaux de notre ordinateur sur le même port pour qu'ils soient en mesure de communiquer et de s'échanger de l'information sous forme de paquets.

Nous avons joint des tests qui permettent d'expérimenter le `sender` et le `receiver` dans 2 cas :

- la lecture et l'écriture dans un fichier
- la lecture sur l'entrée standard et l'écriture sur la sortie standard

Pour ce faire, il suffit d'appeler la commande `make tests` dans le dossier parent et de lancer les éxécutables suivants dans deux terminaux séparés :

- `sender_test_fichier` et `receiver_test_fichier` pour la lecture et l'écriture dans un fichier.
- `sender_test_stdin` et `receiver_test_stdout` pour la lecture depuis l'entrée standard et l'écriture depuis la sortie standard.

En outre, nous avons inclus un test permettant de voir le comportement du protocole lorsque les paquets envoyés sont tronqués et/ou perdus. Nous avons pour ceci utilisé le simulateur de lien. Le `receiver` de ce test est lancé automatiquement lors de l'appel à `make tests`. Il suffit de lancer le `receiver` dans un autre terminal à l'aide de la commande `./sender ::1 12345` pour tester le protocole.

Par ailleurs, nous avons écrit des tests CUnit qui permettent de vérifier l'implémentation de certaines fonctions de base.

Enfin, nous avons également testé notre code grâce au simulateur Linksim qui était fourni sur Moodle. Il s'avère que notre implémentation du projet passe tous les tests sauf celui de la corruption, comme expliqué dans la section *Performances*.

10 Conclusion

Notre code est fonctionnel. Il gère les pertes et la troncation des paquets correctement grâce à un retransmission timer que nous avons fixé. Cependant, nous aurions aimé apporter quelques améliorations à notre code. En effet, nous aurions pu appliquer l'algorithme décrit dans les notes de cours pour calculer un retransmission timer variable et optimal grâce à la valeur que nous avons stockée dans le timestamp. Nous aurions également pu implémenter plus efficacement la gestion des paquets corrompus. Néanmoins, notre projet passe la plupart des tests que nous avons effectués et s'avère fiable.