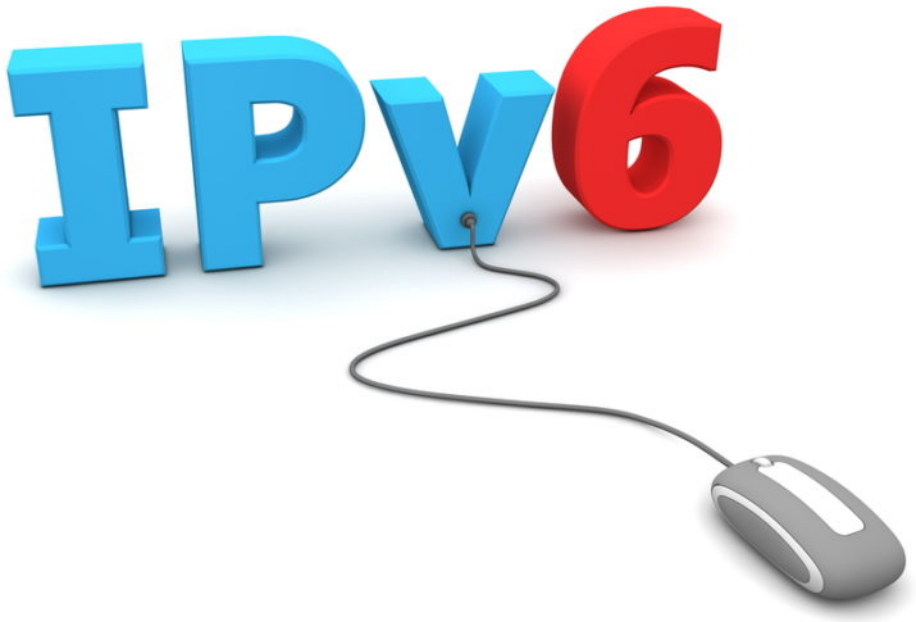
UCLouvain

ÉCOLE POLYTECHNIQUE DE LOUVAIN

LINGI1341 : Réseaux informatiques
Projet TRTP

De Keersmaeker	François	7367-1600
Gérard	Margaux	7659-1600



Louvain-la-Neuve
Octobre 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 principaux, `sender` et `receiver`. Ces fichiers et leur fonctionnement seront décrits en détails 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 encode l'information dans un paquet contenant diverses informations utiles grâce à 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 timer 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 paquet reçu. 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 n'est pas fiable et qu'il peut éventuellement engendrer des erreurs au niveau de l'information transmise.

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, ...
- 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

Pour l'instant, nous n'utilisons pas le champ timestamp mais par la suite, il pourrait contenir une donnée qui nous aiderait à optimiser le temps de retransmission. En effet, lorsqu'un paquet est envoyé sur le réseau, il pourrait contenir l'heure à laquelle il a été envoyé comme valeur du timestamp. Lorsque le `sender` recevra l'aquittement ce de même paquet (qui contiendra toujours le timestamp), il sera alors en mesure de calculer le RTT.

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 Timer

Nous avons choisi un temps de retransmission standard de 2.5 secondes. Par la suite, nous souhaitons le rendre plus efficace en appliquant un algorithme d'amélioration du RTO comme décrit dans les notes de cours. En ce qui concerne la retransmission des paquets perdus ou corrompus, nous les gérons grâce à la fonction `select()`.

6 Performances

Pour l'instant, notre code ne nous permet pas d'envoyer plusieurs paquets simultanément sur le réseau ce qui implique que notre programme n'est pas très performant 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.

7 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.

Nous avons également écrit des tests CUnit qui permettent de vérifier l'implémentation des fonctions suivantes :

- pkt_new
- pkt_get_type
- pkt_set_type
- pkt_get_tr
- pkt_set_tr
- pkt_get_window
- pkt_set_window
- pkt_get_payload
- pkt_set_payload
- pkt_get_seqnum
- pkt_set_seqnum
- pkt_get_length
- pkt_set_length
- ajout_buffer
- retire_buffer
- pkt_encode

8 Conclusion

Notre code est fonctionnel. Il gère les pertes et la corruption des paquets correctement grâce à un retransmission timer que nous avons fixé. Cependant, il nous reste encore quelques amélirations à apporter à notre code. En effet, nous pourrions appliquer l'algorithme décrit dans les note de cours pour calculer un retransmission timer variable mais optimisé. Pour arriver à cet objectif, nous allons nous aider du champ timestamp, que nous n'utilisons pas à l'heure actuelle. Nous espérons que les séances d'interopérabilité nous aideront à optimiser notre code afin que ce dernier soit opérationnel et surtout performant.