

# LINGI1341 - Réseaux informatiques :

## Projet 1

### Implémentation d'un protocole de transport sans pertes

BASTIN Julien

October 2017

## 1 Introduction

Il nous a été demandé dans le cadre du cours *LINGI1341 - Réseaux Informatiques* d'implémenter en langage **C** un protocole de transport utilisant des segments **UDP**. Ce protocole doit permettre de réaliser des transferts fiables de fichiers en utilisant la stratégie du **selective repeat** et doit fonctionner avec **IPv6**. Pour implémenter ce protocole il nous est demandé de réaliser deux programmes : un d'envoi et un de réception de données. Voici quelques spécifications de ces programmes.

## 2 Le champs Timestamp

Le champs timestamp est un élément d'un paquet de données que nous devons envoyer. J'ai choisi d'inclure dans ce champs une valeur différente en fonction du type de paquet envoyé :

- s'il s'agit d'un paquet contenant des données (de type `PTYPE_DATA`) : la valeur de ce champs est le temps écoulé depuis le premier janvier 1970 à minuit converti en millisecondes.
- s'il s'agit d'un acquittement (que ce soit `PTYPE_ACK` ou `PTYPE_NACK`) : la valeur de ce champs est le temps écoulé entre le moment de l'envoi du paquet par le *sender* et le moment de réception du paquet par le *receiver*.

Une fois un paquet d'acquittement reçu par le *sender*, celui peut calculer la valeur de retransmission. Cette fonctionnalité n'est cependant pas encore implémentée par faute de temps.

## 3 Réception de paquets `PTYPE_NACK`

La stratégie choisie ici est la retransmission directe du packet qui a été tronqué. En effet la réception d'un tel paquet indique que le *receiver* a bien reçu le paquet mais que celui-ci a été tronqué. Ceci nous assure donc que le *receiver* ne recevra pas une seconde fois le paquet en question et renvoyer directement le paquet est plus rapide que d'attendre la fin du temps imparti de ce dernier.

## 4 Valeur du retransmission timeout

Pour le moment dans le programme cette valeur est égale à 2000 millisecondes car c'est la valeur maximale de la latence du réseau. Néanmoins ceci n'est pas du tout optimal et il serait plus judicieux de recalculer cette valeur à chaque réception d'un acquittement ; acquittement qui contient la différence entre le temps de réception et d'émission du paquet (voir Timestamp). Cette fonctionnalité sera disponible dans le programme final.

## 5 Partie critique

Examinons d'abord la stratégie du *receiver*.

À la réception d'un paquet le *receiver* regarde d'abord la fiabilité ce celui-ci. S'il est tronqué un paquet de type PTTYPE\_NACK est envoyé directement, s'il n'a pas le bon format il est ignoré et s'il est fiable une suite d'opération est lancée :

1. si le numéro de séquence est hors de la fenêtre de réception alors le paquet est ignoré, sinon il est stocké dans le buffer.
2. si le numéro de séquence correspond au numéro de séquence attendu (qui équivaut au premier élément de la fenêtre de réception) alors on écrit dans le fichier de sortie tous les paquets stockés à la suite du premier élément de la fenêtre.
  - (a) On décale la fenêtre de réception pour que le premier élément soit après le dernier élément écrit.
  - (b) On envoie un acquittement correspondant au numéro de séquence du dernier paquet écrit dans le fichier de sortie.

On peut voir avec cette stratégie qu'aucune étape ne ralentit la vitesse de transfert de donnée. De plus l'acquittement cumulatif lui permet de ne pas envoyer des acquittement pour chaque paquet et est donc un gain de temps.

Examinons ensuite la stratégie du *sender*.

Tout d'abord le *sender* a la possibilité de recevoir des informations du *receiver* ainsi que lui en envoyer grâce à la fonction *select()*. Ceci est un gain de temps et il est possible de traiter un acquittement ainsi qu'un envoi de donnée en même temps ; c'est ce que fait le *sender* ici. Le seul point noir du *sender* est qu'il parcourt à chaque fois la fenêtre d'envoi entière pour tester les temps de transmission et ne recalcule pas ceux-ci pour s'adapter à la vitesse du trafic.

## 6 Stratégie de tests

Par manque de temps, seul des tests manuels ont été effectués en local. Voici les différents tests réalisés et leurs résultats :

- Lancement du *sender* et du *receiver* en local et sans fichier, donc en échangeant de l'information via l'entrée standard : tout se déroule parfaitement bien.
- Lancement du *sender* et du *receiver* en local avec un fichier de taille inférieure à 512 octets passé en entrée du *sender* : tout se déroule parfaitement bien.
- Lancement du *sender* et du *receiver* en local avec un fichier de taille supérieure à 512 octets passé en entrée du *sender* : pour une raison qui n'a pas encore été trouvée les  $n-1$  premiers paquets (avec  $n$  = le nombres de paquets à envoyer pour envoyer entièrement le fichier) ne s'envoie pas correctement et ensuite s'en suit une boucle de réception du  $n^{i\text{eme}}$  paquet par le *receiver* alors que le *sender* ré-envoi des paquets suite au timeout.

## 7 Test d'interopérabilité

J'ai réalisé un test avec le groupe 25. Voici les différentes choses que nous avons remarqué :

- Lors du test en local de mon sender avec le receiver de l'autre groupe sans fichier passé en argument, on a remarqué que mon sender marche sauf pour l'envoi du dernier paquet qui doit fermer le transfert.
- Lors du test de pc à pc on a remarqué que mon sender n'arrive pas à trouver le pc de l'autre groupe et que mon receiver décode mal les données. Au contraire leur sender arrivait à atteindre mon pc mais leur sender envoyait un premier paquet avec tr à 1 pour calculer le rtt, ce qui n'est pas interopérable.

Nous n'avons dès lors pas sus effectuer plus de tests car mon sender et mon receiver ne fonctionnaient pas correctement.

## 8 Changements effectués

Un premier changement effectué est l'utilisation du champ `timeout`. En effet à la première soumission je ne l'utilisais pas et maintenant il me sert à adapter le rtt pour plus grand efficacité.

Le second changement que j'ai réalisé est la résolution des problème d'encodage et décodage des paquets. À la première soumission je m'étais rendu compte que soit l'encodage soit le décodage de paquet contenant un payload de taille maximale (512 bytes) ne marchait pas. Après de longues recherches j'ai trouvé que c'est l'encodage qui ne fonctionnait pas, j'ai donc modifié la fonction `pkt_encode` de la classe `packet_imlem.c`.

Un troisième changement a été effectué sur la stratégie d'envoi et de réception ainsi que sur le décalage des fenêtres de réception et d'envoi. Ce changement a été réalisé suite à un envoi de fichier qui ne se terminait pas correctement lors d'un test.

Un quatrième changement effectué est la redirection des logs sur la sortie `stderr`. On sait dès lors récupérer les logs lors du passage de l'argument `2 > fichier.log`.

Un cinquième changement effectué est la possibilité d'arrêter la conversation lorsqu'on ne passe pas de fichier avec l'argument `-f` au sender via `CTRL+D`. Ceci permet aussi de passer des fichiers via `stdin`, la fin du fichier est alors détectée et la conversation se termine.

Le dernier changement effectué est la création de tests via la commande `maketest`. Ces tests sont les mêmes qu'effectués manuellement lors de la première soumission (à savoir l'envoi de 3 fichiers de taille croissante). Le résultat de ces tests est désormais positif contrairement à la première soumission. Un quatrième test a aussi été rajouter pour tester le transfert de fichier via `stdin`.