# Implémentation d'un protocole de transport sans perte

**Groupe 33 :** Samuel MONROE, Rémy VOET

Octobre 2016

## 1 Introduction

Ce rapport fait état de notre travail sur l'implémentation d'un protocole de transport sans perte, via notamment la mise en place d'une stratégie de selective repeat en utilisant le protocole UDP.

## 2 Architecture du programme

### 2.1 Initialisation du receiver et sender

La première étape du programme est l'initialisation de deux entités communiquantes (receiver & sender) selon les paramètres suivants :

— Hostname ou IPv6
— Numéro de port
— Fichier optionnel comme source d'input (sender) ou destination de l'ouput (receiver)

Nous avons pour cela créé une fonction *read_args* commune aux deux entités et utilisant *getopt* afin d'initialiser les données nécessaires à la communication.
En l'absence d'un nom de fichier optionnel, ce sont les file-descriptor de STDIN/STDOUT qui sont utilisés respectivement pour le sender et le receiver.

### 2.2 Connexion des deux entités

Pour cette étape, nous avons en grand partie repris les éléments produits afin d'accomplir la tâche INGInious "Envoyer et recevoir des données".

1. **real_address** est utilisé par les deux entités afin de résoudre le hostname acquis précédemment en une addresse IPv6 valide.

2. **create_socket** crée un socket et permet au sender de se connecter sur celui-ci via l'addresse/port obtenus précedemment, et permet au receiver de lier ce socket sur son addresse et port.

3. **wait_for_client** est enfin utilisé par le receiver afin d'attendre un message en provenance du sender. Cette détection est effectuée via une lecture indempotente sur le socket.

## 2.3 Boucle de communication

Cette partie du programme est la plus complexe et celle ayant nécessité le plus de travail, les éléments précédents, ainsi que le formatage des paquets échangés, ayant été accomplis au travers des tâches INGInious.

Avant de rentrer dans la boucle de communication, les deux entités initialisent un tableau de buffers destinés à contenir temporairement les paquets échangés, afin que le sender puisse les retransmettre si besoin, et que le receiver puisse écrire ces paquets sur fichier dans l'ordre de leurs numéros de séquence.

### 2.3.1 Receiver

En plus du buffer de paquets, le receiver tient à jour un compteur de numéros de séquence utilisé pour l'envoi des acquittements.

#### Lecture sur le socket

Si le receiver dispose de suffisamment de place sur son buffer, il récupère un paquet sur le socket.
Ce paquet est ensuite placé sur le buffer à l'indice donné par $numSequence \ \% \ tailleFenetre$.
Si le numéro de séquence du paquet correspond au numéro suivant attendu, le compteur est mis à jour pour envoyer l'acquittement. Sinon cet acquittement est envoyé avec un numéro de séquence qui correspond au dernier paquet non-reçu afin d'en obtenir la retransmission.

#### Ecriture des fichiers sur le buffer

Puisque les paquets sont disposés sur le tableau de buffers selon un ordre précis et ordonné, un indice d'écriture parcourt ce buffer lors de la boucle.
Si l'indice est positionné sur un paquet stocké valide, celui-ci est écrit sur le fichier, l'indice est incrémenté et l'emplacement précédent est vidé.
Si l'indice est positionné sur un emplacement mis à **NULL**, c'est que le paquet correspondant à cet emplacement n'a pas encore été reçu. Le processus d'écriture pourra reprendre lorsque cet emplacement aura été rempli.

### 2.3.2 Sender

**Ecriture sur le socket**

Au tout début de la communication, le sender considère une window valant 1.
En fonction de la valeur de la window, le sender envoie autant de paquet que possible en construisant ceux-ci via les données obtenues sur le descripteur de fichier. Un **timestamp** est ajouté à ces paquets avant envoi en utilisant la valeur t courante du temps système.
Ces paquets envoyés sont stockés sur le buffer d'envoi de la même manière qu'ils sont stockés par le receiver en réception.

**Retransmission des paquets**

Cette retransmission se base sur la valeur des **timestamp** associés aux paquets retenus dans le buffer d'envoi.
Une boucle parcourt les paquets du buffer et compare chaque timestamp avec le temps courant système.
Si le temps courant est plus grand que le timestamp du paquet additionné à la valeur du timeout, ce paquet est renvoyé au receiver.

La valeur du **timeout** a été établie de façon à être supérieur au RTT calculé selon la latence donnée par l'énoncé du projet.
Cette latence est indiquée comme variant dans l'intervalle **[0,2000]**(ms). Le Round-Trip-Time peut donc être évalué à ~4000ms au maximum. Un **Timeout** de 5000ms nous donne alors une marge confortable afin d'établir un timer de retransmission.

**Réception des acquittements**

Lors de la réception d'un acquittement envoyé par le receiver, l'acquittement cumulatif est géré en positionnant un indice sur le buffer de paquets envoyé et en parcourant vers l'arrière ce tableau.
Une comparaison sur les numéros de séquence permet de libérer uniquement les paquets concernés par l'acquittement cumulatif.

# 3 Partie critique et vitesse de transfert

La partie critique qui affecte la vitesse de transfert est sans aucun doute la valeur du timeout, qui a été prévue pour pallier au pire cas possible de latence et ce qui ralenti fortement la vitesse de retransmission et par conséquent le temps total nécessaire au transfert.
Une possibilité d'amélioration de ce problème serait un recalcul du timeout en fonction de la latence constatée sur le réseau.

Une autre partie qui affecte la vitesse de transfert de l'application de sans doute la stratégie de retransmission mise en place.
Celle-ci se base sur une boucle parcourant le tableau de paquets stockés, ce à chaque passage

dans la boucle de communication.
L'utilisation de threads et de la fonction **ualarm** pourrait peut-être être une alternative à cette boucle et de permettre aux threads de relancer l'écriture du paquet concernés lorsqu'un signal d'alarme est lancé après expiration du timeout.

# 4 Stratégie de test

Afin de pouvoir tester de manière isolée les différents composants (fonctions) du programme, nous avons utilisé la librairie **CUnit**.
Des tests unitaires couvrent donc les différents cas possible d'exécution des fonctions et leurs valeurs de retour.

De manière plus globale et afin de tester l'exécution entière des deux programmes, nous utilisons **link_sym** dans un script bash afin d'observer le comportement du protocole dans un environnement avec pertes et latence.

# 5 Conclusion

Nous sommes parvenus à une solution fonctionnelle même en poussant les caractéristiques du simulateur de réseau vers des conditions très peu idéales.

Nous n'avons cependant pas su identifier un bug survenant dans de rares cas, qui sera l'objet de notre correction pour la seconde remise en addition des éléments qui seront soulevés suite au test d'interopérabilité.

# 6 Test d'interopérabilité et modifications

Les tests d'interopérabilité effectués le mercredi 26/10 ont été concluants, nous sommes parvenus à réaliser des échanges de données sans pertes avec les autres groupes.
Les seuls problèmes identifiés lors de ces tests étaient les suivants :

— Notre sender envoyait le dernier paquet de fin de fichier **eof** avant que le receiver ait signalé la bonne réception des paquets précédents.

— L'utilisation de l'utilitaire **valgrind** a pointé de nombreuses fuites de mémoire dans notre application.

Les modifications du code ont donc concerné ces fuites de mémoires, liées principalement à des non-libérations de buffers.

Concernant le problème d'envoi de fin de fichier, nous avons modifié le *sender* afin que celui-ci attende de recevoir l'acquittement portant le numéro de séquence attendu de ce paquet de fin de fichier, le sender étant dès lors certain que les données précédentes ont été reçues avant de terminer le transfert.