

Projet 1 - Rapport

Antoine Gennart

October 27, 2018

1 Introduction

Dans le cadre du cours LINGI1341 : *Computer networks : information transfer*, il nous a été demandé de réaliser un protocole de transfert fiable basé sur des segments UDP.

Ce rapport décrit l'architecture globale du projet, et répond aux questions imposées dans le cadre du projet.

2 Architecture

- **sender.c** Implémente la fonction principale du sender.
- **receiver.c** Implémente la fonction principale du receiver.
- **pkt.c** Implémente toute la structure des packets, ainsi que toutes les fonctions pour manipuler ces packets.
- **network.c** Implémente la connexion UDP entre deux machines.
- **utils.c** Contient un certain nombre de fonction utiles et qui n'avaient pas leur place dans les autres fichiers.
- **fifo.c** Implémente la mémoire FIFO (*first in first out*) qui va être utilisé par le receiver pour gérer les *acknowledgment*

3 Questions

3.1 Que mettez-vous dans le champs **Timestamp**, et quelle utilisation en faites-vous?

Le champs timestamp contient un nombre de microsecondes. Il est calculé comme suit :

$$\text{timestamp} = t_{\text{epoch}} \% (2^{32} - 1) [\mu s]$$

ou t_{epoch} correspond au temps écoulé depuis l'époque en microsecondes et $\%$ représente l'opération modulo.

Le timestamp étant enregistré sur 32 bits, il est limité dans sa précision. Utiliser la valeur calculée ci dessus correspond à un cycle d'environ 1 heure entre chaque réinitialisation du timestamp.

3.2 Comment réagissez-vous à la réception de paquets **PTYPE_NACK**?

Lorsque le *sender* reçoit un packet de type **PTYPE_NACK**, il sait que le packet n'a pas été correctement transmit au *receiver*, il va donc directement renvoyer un packet sans prendre en compte la valeur du retransmission timeout.

Lorsque le *receiver* reçoit un packet de type **PTYPE_NACK**, il sait que le réseau a tronqué le paquet, il va donc envoyer un *not acknowledgement* au lieu *acknowledgment*. Anisi le *sender* sera au courant de la situation et pourra renvoyer un bon packet au plus vite.

3.3 Comment avez-vous choisi la valeur du retransmission timeout?

La valeur du *retransmission timeout* est calculé via l'algorithme de *Van Jacobson*. Comme décrit dans le cours (p. 159 du syllabus), il est initialisé de la manière suivante :

$$\begin{aligned} srtt &= rtt \\ rttvar &= \frac{rtt}{2} \\ rto &= srtt + 4 \cdot rttvar \end{aligned}$$

Lorsque d'autres mesures du rtt arrivent, nous pouvons mettre à jours les valeurs de rtt, srtt et rto comme suit:

$$\begin{aligned} rttvar &= (1 - \beta) \cdot rttvar + \beta \cdot (srtt - rtt) \\ srtt &= (1 - \alpha) \cdot srtt + \alpha \cdot rtt \\ rto &= srtt + 4 \cdot rttvar \end{aligned}$$

Dans le cadre du projet, j'utilise les valeurs conseillées dans le syllabus, à savoir $\alpha = \frac{1}{8}$ et $\beta = \frac{1}{4}$. Ces valeurs particulières permettent de remplacer des *floating point operations* par des *bits shifts*, ce qui est beaucoup moins coûteux pour le processeur.

3.4 Quelle est la partie critique de votre implémentation, affectant la vitesse de transfert?

La partie la plus critique est l'attente d'un *acknowledgment* lorsque qu'un packet a été perdu. Cela nous fait perdre le temps d'un *retransmission timeout*. Sur un réseau local, on mesure un retransmission timeout inférieur à 1 milliseconde.

3.5 Quelles sont les performances de votre protocole?

Une première mesure de la performance du protocole a été faite en mesurant la vitesse de transfert en fonction de la taille de la *sliding window*. Deux sets de mesures ont été réalisés, un premier utilisant le simulateur de lien proposé par le professeur, un second exécuté localement sur mon ordinateur. Chaque mesure représentée dans la figure 1 est la moyenne de 20 mesures réalisées pour la *sliding window* correspondante.

Comme nous pouvons le voir, la présence d'un simulateur de lien basique¹ n'affecte pas de manière significative le transfert. Par contre la taille de la *sliding window* est très importante pour la vitesse de transfert. On observe un pique lorsque la sliding window fait une taille de 2.

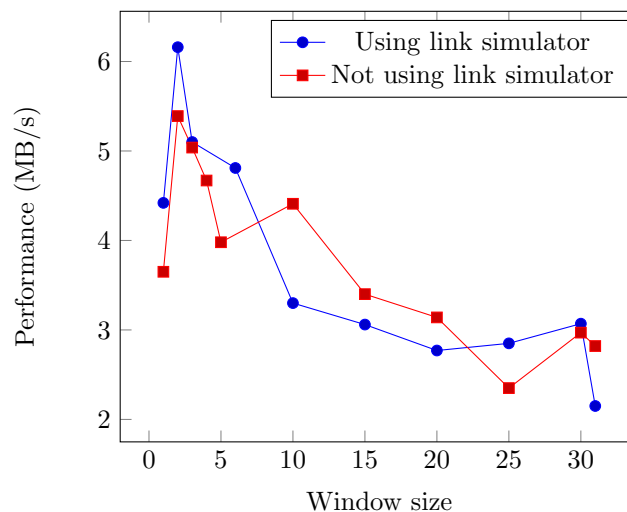


Figure 1: Mesure de la performance en fonction de la taille de la *sliding window*

Une seconde mesure de la performance du protocole vise à mesurer l'efficacité de l'algorithme de *Van Jacobson* décrit dans la section 3.3. Comme on peut l'observer dans la figure 2, le RTO² commence à une valeur élevée, et va se mettre à jour automatiquement à chaque fois que des packets sont reçus. Il ne sera jamais stable, étant donné qu'à chaque fois que le RTO sera respecté, il diminuera, jusqu'à ce qu'il soit trop petit. A ce moment, il sera augmenté.

3.6 Quelles sont les stratégies de test que vous avez utilisées?

La stratégie de test consiste à envoyer et recevoir un fichier, pour ensuite comparer le fichier d'entrée avec le fichier de sortie. Si ces deux fichiers sont différents, le test échoue.

Une seconde stratégie consiste à réaliser exactement la même chose que précédemment, mais en utilisant un simulateur de lien prévu à cet effet.

¹Il simule un lien presque sans défauts

²Retransmission timeout

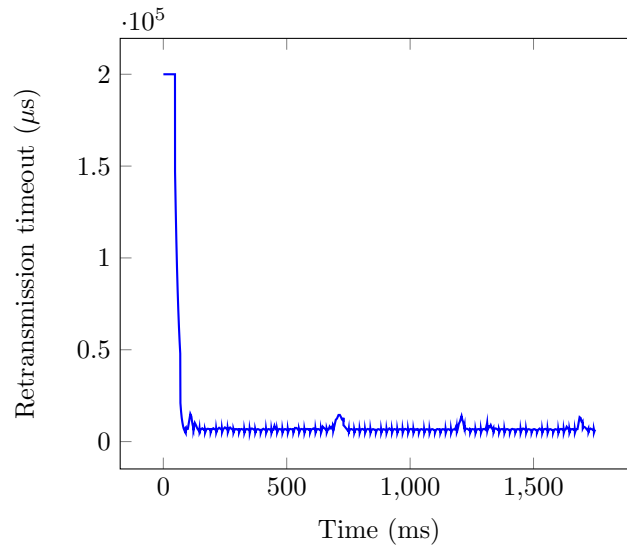


Figure 2: *Retransmission timeout* en fonction du temps.

4 Séances d'interopérabilité

Groupe de Losseau : Il a été découvert que le receiver ne renvoyait pas le numéro de séquence attendu par le sender, mais le numéro de séquence attendu diminué de 1.

Groupe de Quinet : Il a été découvert que la fonction du sender attendait un packet de taille trop grande pour un *ack*, la fonction de décodage d'un packet renvoyait donc l'erreur `E_NOMEM`. Cette erreur a été corrigée en soustrayant l'espace réservé au *crc2* lorsque le payload était absent (cas d'un *ack*).

Autre groupe : Il a été découvert que le code du sender ne prenait pas en compte l'entrée standard, et que le sender ne prenait pas en compte qu'un fichier puisse être entré en paramètre.

5 Conclusion