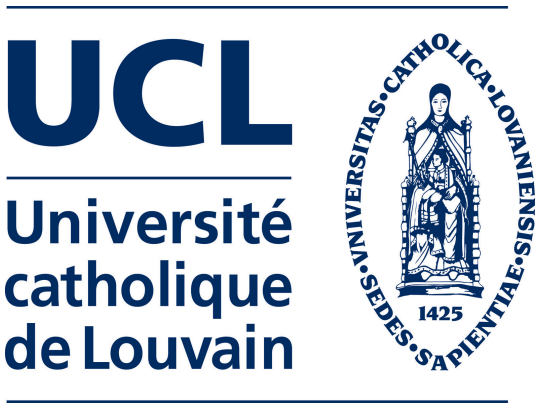

LINGI-1341

Computer Networks : Information Transfer

Rapport du projet - Implémentation d'un protocole de transport sans pertes

Groupe n°2

Auteurs : Nicolas Sias, Youri Mouton

2017-2018

1 Introduction

Conjointement aux cours théorique et aux travaux pratiques, il nous a été demandé de réaliser un protocole de transport en C utilisant *UDP*. Ce protocole doit être fiable, appliquant la stratégie du *Selective Repeat*. De plus, notre projet doit également fonctionner avec *IPV6*.
L'implémentation de ce protocole requiert la conception de deux programmes de reception (*receiver*) et d'envoi (*sender*), ceux-ci permettant le transfert de données unidirectionnel.

Dans ce rapport, nous allons détailler l'architecture de notre implémentation et le fonctionnement des programmes d'envoi et reception. Par la suite, nous expliquerons notre stratégie de tests et les améliorations possibles. Enfin, nous terminerons par une brève conclusion.

2 Architecture

Notre implémentation comporte huits fichiers :

1. **common.h** : fichier header commun à tous les autres fichiers C, ce dernier détermine les fonctions utilisées dans l'implémentation, les constantes importantes précompilées (les *defines*), la structure d'un paquet et ses flags d'état ainsi que quelques macros utilitaires.
2. **receiver.c** : implémente la reception des données de notre protocole.
3. **sender.c** : implémente l'envoi des données de notre protocole.
4. **utils.c** : regroupe l'ensemble des fonctions utilitaires, facilitant la lisibilité de notre code.
5. **pkt.c** : décrit les méthodes concernant la structure du paquet. Son code est inspiré par la tâche *INGinious-Format des segments du projet de groupe*.
6. **net.c** : implémentation les fonctionnalités concernant la connexion entre le *receiver* et le *sender*.
7. **minqueue.h** : décrit la structure et les fonctions de la *priority queue* utilisée par receiver.c.
8. **minqueue.c** : implémente les fonctionnalités de la *priority queue*.

3 Fonctionnement des programmes

3.1 Receiver

Après interprétation des arguments d'entrées et validation de la connexion avec le *sender*, le *receiver* entame la reception et l'écriture des données. Le programme se termine après l'envoi du dernier ACK. Cet ACK est déterminé par le dernier paquet envoyé par le sender. Un paquet est considéré comme dernier si son champ length est à zéro et son seqnum est équivalent au seqnum de l'avant-dernier paquet reçu.

3.1.1 Description de la méthode receive_data

Cette méthode contient l'ensemble de notre stratégie de reception. Elle regroupe plusieurs variables importantes à la stratégie de *selective repeat* :

— **window_size** : le nombre de place vide dans le buffer de réception, initialisé à 31.
— **status** : l'état du paquet reçu. Le receiver ignore les paquets considérés comme non-valides par *status*.
— **min_missing_pkt** : le seqnum du paquet minimum manquant, initialisé à 0.
— **pkt_queue** : la priority queue orientée minimum contenant les paquets reçus et triant automatiquement ceux-ci en fonction de leur champ seqnum. Si la priority queue reçoit un paquet dont le seqnum est équivalent à un autre paquet dans pkt_queue, il ignorera le paquet.

La méthode **receive_data** consiste en une boucle ne pouvant se terminer que si la méthode envoie le dernier ACK. Après réception d'un paquet, celui-ci est vérifié. Si ce dernier est correct, le programme appelera la méthode **fill_window**.

La méthode **fill_window** consiste à décrémenter *window_size* si ce paquet n'est pas hors-séquence. Après décrémentation, le paquet est placé dans pkt_queue.

Si la variable window_size a été décrémentée, le receiver déclenchera la méthode **write_packet**. Celle-ci écrit l'ensemble des paquets dans le buffer jusqu'au *min_missing_pkt* exclus. Elle vérifie aussi si le prochain paquet écrit est

Si une écriture a été effectuée, le receiver appelera la méthode **send_response**. Celle-ci envoie un ACK au sender avec un seqnum équivalent au seqnum du dernier paquet écrit.

On répète ce processus jusqu'à la réception du paquet clôturant la connexion.

3.1.2 Gestion des cas particuliers

— **Gestion des paquets tronqués** : un paquet tronqué sera automatiquement ignoré après l'envoi d'un NACK au sender. Ce NACK contiendra un seqnum équivalent à min_missing_pkt.
— **Paquet corrompu** : la variable *status* vérifie l'état de la méthode pkt_decode. Après décodage, elle vérifie si le CRC1 et CRC2 du nouveau paquet reçu sont valides.
— **Paquet hors séquence** : On considère qu'un paquet est hors séquence lorsque le nombre de paquets supposés entre *min_missing_pkt* et ce paquet n'est pas compris entre 0 et 31. Si un paquet est hors séquence, il est ignoré.

3.1.3 Remarques

Le programme Receiver à été écrit par Nicolas Sias sur les machines Linux de la salle Intel au Réaumur. De plus, le *receiver* ne contient pas de fuites mémoires, comme rapporté par Valgrind.

3.2 Sender

Le programme Sender permet l'envoi des paquets au Receiver par la stratégie du Selective Repeat. Les données à envoyer sont stockées dans une fenètre glissante. Chaque fois qu'un acquittement est reçu pour au moins l'entrée la plus ancienne, la fenètre de réception sera glissée en fonction du nombre de paquets à acquitter. Cela permet de gérer les acquittements cumulatifs aisément. L'implémentation du Sender permet également de gérer les paquets perdus, délayés, corrompus ou tronqués. Tous les arguments du simulateurs de lien *link_sim* ont été vérifiés avec des paramètres choisis aléatoirement.

3.2.1 Les fonctions et procédures de Sender

1. *array_slide* : Cette fonction va glisser un tableau de taille $size$, n fois.
2. *get_payload* : Lit à une position donnée une quantité de données depuis un fichier. Cette fonction est typiquement appelée à chaque fois que la fenètre glissera.
3. *slide_window* : Cette procédure va supprimer les paquets acquittés et ajouter de nouveaux paquets prêts à être envoyés.
4. *make_window* : Cette fonction prépare une première fenètre d'envoi de taille 32. Appelée une fois avant de rentrer dans la boucle d'envoi.
5. *send_terminating_packet* : Cette procédure contient le code un peu compliqué permettant la fermeture grâcieuse du Receiver et du Sender en envoyant des paquets dont le champs $length == 0$ et en attendant une réponse appropriée.
6. *send_data* : Procédure contenant la boucle principale d'envoi des paquets au Receiver.

3.2.2 Cas particulers

1. Delay/Jitter : Lorsqu'un acquittement met du temps à arriver, l'utilisation de $poll\left(3\right)$ permet de continuer à envoyer d'autres paquets avant d'écouter à nouveau un acquittement
2. Loss : Lorsqu'un paquet est perdu il sera renvoyé après les deux secondes de time-out ou lors de la réception de l'acquittement approprié
3. Cut : Lorsqu'un acquittement est tronqué on l'ignore et on continue à envoyer et écouter. Si un paquet tronqué est requiert par l'envoi d'un NACK par le receiver on retourne en arrière dans la fenètre d'envoi afin d'envoyer ce paquet tronqué
4. Error : Si un paquet est corrompu la même stratégie que pour le Cut est utilisée

3.2.3 Stratégie de sortie

La procédure $send\_terminating\_packet$ permet de finir la transmission de manière gracieuse. Si notre paquet est corrompu, perdu, tronqué et que le Receiver est toujours connecté il se renverra et attendra un acquittement de ce dernier. Dans le cas ou l'acquittement n'arriverait pas, la procédure vérifie que le Receiver est toujours connecté par un appel non bloquant à $recv2$ et vérification d'erreur $ENOTCONN$, permettant au Sender de terminer son exécution si le Receiver, ayant envoyé un acquittement pour le paquet de termination s'est arrêté mais ce dernier s'est perdu, corrompu ou tronqué. Cette procédure n'est pas le plus bel exemple de code dans le protocole mais elle est robuste.

3.2.4 Remarques

Le programme Sender à été écrit par Youri Mouton, certifié sans fuites ou accès douteux en mémoire par l'utilisation du programme $valgrind\ --leak-check = full$. De plus, Sender à été écrit et testé sur les machines $CentOS$ de la salle Intel au Réaumur.

4 Questions

4.1 Que mettez-vous dans le champs Timestamp, et quelle utilisatin en faites-vous ?

Nous mettons la valeur en microsecondes du temps récupéré par $gettimeofday$ tenant sur 32 bits dans le champs timestamp. Cela nous permet de calculer le round-trip-time à la réception d'un ack pour calculer un retransmission-timeout approprié, comme décrit plus bas.

4.2 Comment réagissez-vous à la réception de paquets PTYPE_NACK ?

Le programme Sender va simplement renvoyer la fenètre en commençant par le plus ancien paquet non acquitté à la manière $Go - Back - N$.

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

Nous avons implémenté la gestion du retranmission timeout telle qu'expliquée dans le syllabus page 159. L'algorithme de Jacobson mis en place utilise le Smoothed Round-Trip-Time et la variance de RTT, rttvar. Initialement le RTO est mis à 2 secondes. Lors du premier acquittement reçu, le RTO est calcué comme ceci :

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

Le calcul du RTO aux acquittements suivants, les paramètres $\alpha = \frac{1}{8}$ et $\beta = \frac{1}{4}$:

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

Le calcul du RTO est limité à deux secondes comme défini dans l'énoncé du projet.

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

Le retransmission timeout de 0.2 secondes ralentit probablement dramatiquement les performances que nous pourrions atteindre. Encore une fois, la séance d'interopérabilité nous éclairera sur les méthodes que nous pourrions utiliser.

5 Stratégie de tests

5.1 Shell Scripts

Nous avons utilisés des $Shell$ scripts afin de lancer les programmes Sender, Receiver et $link\_sim$ automatiqument. Ils sont disponibles dans le dossier $test$. Lors du développement des programmes Sender et Receiver nous avons découvert beaucoup de paramètres du simulateur de liens pour lesquels les programmes Sender et Receiver ne fonctionnaient pas correctement et la $seed$ pour chacuns de ces tests ont été gardés afin de reproduire ces tests lors de l'implémentation du futures fonctionnalités. Ces $seed$ ont formé nos cas de tests unitaires locaux et nous ont bien servi à nous rendre compte d'erreurs fatales.
Par exemple :

- Le premier paquet corrompu ou perdu causant des erreurs
- Le paquet de termination de connection perdu causant le bloquage du Receiver
- L'acquittement du paquet de termination de connection perdu causant le bloquage du Sender
- L'acquittement de paquets hors séquence
- ...

6 Séances d'interopérabilité

1. **Lundi 23/10, groupe 15** Nous avons remarqué que le sender et receiver du groupe 15 envoyaient uniquement des fichiers textes. De plus, notre paquet et acquittement terminant la connexion étaient erronés. En effet, nous avions considéré que le dernier paquet et acquittement possédaient un champ seqnum équivalent au dernier paquet contenant des données.
2. **Lundi 23/10, groupe 33** Leur sender considérait des acquittements corrects comme non-valides. De plus, leur receiver ralentissait fortement le processus de transfert suite à un timeout trop important. Nos paquets et acquittements terminant la connexion n'étaient pas encore corrigés durant cette séance.
3. **Mercredi 25/10, groupe 100** La window du paquet terminant la connexion possédait un champ window de taille zéro. Après correction de ce bug, nous n'avons eu aucun problème sans link_sim. Avec tout les paramètres activés sur link_sim, notre sender ralentissait le processus de transfert.
4. **Vendredi 29/10, groupe 43** Durant cette séance, nous avons utilisé uniquement le sender. En effet, le control flow de notre receiver était en cours de réécriture. Aucun bug n'a été recensé durant cette séance, avec ou sans link_sim.

Suite à nos séances d'interopérabilité, quelques modifications ont été apportés :

- **receiver.c** Le paquet de termination attendu et l'acquittement envoyé ont été corrigés, permettant notre receiver de fonctionner avec les sender d'autres groupes. Le receiver a été repensé suite à la découverte d'un soucis dans notre control flow : le receiver s'arrêtait brusquement si les paquets commençant et terminant la connexion étaient corrompus ou possédaient des délais de transmission trop importants. Nous avons donc changé la structure du code, ce qui nous a permis de le clarifier et de supprimer des lignes obsolètes. Conceptuellement, les fonctions restent les mêmes, mais celles-ci sont plus lisibles et propres. Nous avons aussi implémenter le fast retransmit du côté du receiver : ce dernier envoie toutes les deux cents milisecondes des acquittements si aucun paquet n'a été reçu.
- **sender.c** L'acquittement attendu lors de l'envoi du paquet de termination à été corrigé, permettant notre sender de fonctionner avec les receiver des autres groupes correctement. La calcul dynamique du retransmission timeout à été implémenté par l'utilisation du timestamp et de l'algorithme de Jacobson. De plus, le sender renverra un paquet après avoir reçu trois fois un même ack, pouvant indiquer qu'un paquet est manquant avant que le time-out soit atteint, permettant de rendre plus rapide l'envoi de fichiers.
- **Makefile** Les logs sont rendus optionels par l'utilisation de l'argument -DDEBUG

7 Conclusion

Ce projet aura été un challenge décidemment intéressant, nous permettant de mettre en application des concepts appris en cours théorique de réseaux. L'implémentation en groupe de deux à aussi été l'occation d'apprendre à manier le langage C un peu mieux, ce qui n'est pas une mince affaire. Nous avons pu remarquer que la séance d'interopérabilité était très utile et a pu nous permettre de nous rendre compte d'erreurs de compréhension de l'énoncé. De plus, ces séances nous ont inspiré pour l'implémentation du calcul du retransmission timeout.