LINGI1341 : Réseaux informatiques - Projet 1

de Voghel Brieuc - 5910 1600
van de Walle Nicolas - 2790 1600

12 septembre 2021

Introduction

Dans le cadre de l'installation d'un nouveau data-center de Crousti-Croc, il nous a été demandé de fournir une implémentation d'un protocole TRTP (i.e. Truncrated Reliable Transfer Protocol) basé sur des segments UDP dans le langage C.

Ce programme permet de communiquer entre un `sender` et un `receiver` sur deux machines distantes. Ces deux fonctions prennent comme arguments une adresse IPv6 et un port UDP formant une connexion sur laquelle transiteront les messages. Les données transférées peuvent provenir d'un fichier passé en argument ou de l'entrée standard.

Dans ce rapport sont décrits la structure du code et les choix d'implémentation. Une évaluation qualitative et quantitative de l'implémentation est également présente. Les performances et la méthodologie utilisée y sont analysées.

1 Structure du code

Afin de faciliter la gestion, la création et le debug du code, celui-ci a été divisé en différents dossiers et fichiers et applique le principe de responsabilité unique [1] (adapté à un langage de programmation non orienté objet tel que C évidemment). Chaque fichier pourrait être comparé à une classe et joue un rôle bien précis. Voici, ci-dessous, la structure des dossiers :

```
/
├── src/
│   ├── packet/
│   │   ├── Makefile
│   │   └── packet.c (et .h associé)
│   ├── socket/
│   │   ├── Makefile
│   │   ├── create_socket.c (et .h associé)
│   │   ├── read_write_loop_receiver.c (et .h associé)
│   │   ├── read_write_loop_sender.c (et .h associé)
│   │   ├── real_address.c (et .h associé)
│   │   └── wait_for_client.c (et .h associé)
│   ├── stack/
│   │   ├── Makefile
│   │   └── stack.c (et .h associé)
│   ├── receiver.c
│   └── sender.c
├── tests/
│   ├── Makefile
│   ├── packetTest.c
│   └── stackTest.c
├── Makefile
├── receiver
└── sender
```

1. https://fr.wikipedia.org/wiki/Principe_de_responsabilit%C3%A9_unique

Le dossier `src` contient donc, comme spécifié dans les consignes, les différents fichiers sources du programme. Celui-ci se décompose donc en trois dossiers :

— **`packet`** : il s'agit ici de l'ensemble des fonctions permettant d'agir sur un packet (dont l'implémentation est discutée ci-après). Elles correspondent à ce qui était demandé dans l'exercice Inginious que l'on peut retrouver à l'adresse suivante : `https://inginious.info.ucl.ac.be/course/LINGI1341/format-des-segments`

— **`socket`** : il s'agit ici de l'ensemble des fichiers contenant les fonctions demandées dans l'exercice Inginious disponible à l'adresse suivante : `https://inginious.info.ucl.ac.be/course/LINGI1341/envoyer-et-recevoir-des-donnees`. Ces fonctions permettent de récupérer les adresses IPv6 d'un hôte sur base de son hostname, créer un socket afin de communiquer entre 2 hôtes, attendre la connexion du pair et envoyer ou recevoir des données depuis le pair.

— **`stack`** : il s'agit ici d'un ensemble de fonctions permettant la création et l'interaction avec une liste doublement chaînée qui sera utilisée pour y stocker des paquets (`pkt_t`). Le fonctionnement de cette liste sera développé dans la section 3.3

Le deuxième dossier que l'on peut retrouver à la racine du projet est le dossier `tests` qui contient les différentes suites de tests ayant permis le développement du programme. Ceux-ci peuvent être exécutés grâce à la commande `make tests` depuis la racine.

Vient ensuite le `Makefile` « général » chargé de générer les différents exécutables (`sender` et `receiver`). Celui-ci fera appel, aux différents `Makefile`'s spécifiques situés dans les dossiers des sources. Cela permet de faciliter l'ajout ou la suppression de dépendances sans pour autant devoir retrouver, au niveau du `Makefile` général, le différents points à ajouter ou supprimer.

2 Algorithme

Lors du lancement de l'exécutable `sender` en utilisant une des commandes suivantes,

```
./sender [-f INPUTFILE] HOSTNAME PORT
./sender HOSTNAME PORT < INPUTFILE
```

une connexion est initiée (grâce à HOSTNAME et PORT) et le fichier INPUTFILE (ou l'entré standard si ce fichier n'est pas spécifié) est lu et décomposé en paquets de 512 Bytes qui sont ensuite stockés dans un buffer avec un numéro de séquence `seqnum` $\in [0; 255]$.

Une boucle va ensuite être lancée et exécutera différentes actions :

— **Envoi des paquets** : sur base du numéro de séquence à envoyer, le programme récupérera le paquet associé du buffer et l'enverra sur le `socket` de la connexion.

— **Réception des réponses** : le programme récupérera également, sur le `socket`, les réponses envoyées par le `receiver` qui peuvent être de type `ACK` ou `NACK`.

De son coté, le `receiver`, qui a été lancé avant (ou au maximum 10s après) le `sender` au moyen d'une des commandes suivantes,

```
./receiver [-f OUTPUTFILE] HOSTNAME PORT
./receiver HOSTNAME PORT > OUTPUTFILE
```

va recevoir les paquets et envoyer des réponses en conséquence (`ACK` ou `NACK`).

3 Choix d'implémentation

3.1 Utilisation du champ timestamp

Le champ `timestamp` présent dans les paquets envoyés sont initialisés avec l'heure Posix (en secondes) juste avant l'envoie du paquet. Ce champ sera accédé afin de savoir quand le paquet à été envoyé pour savoir si son RTO (i.e. retransmission time out - ou compte à rebours déterminant si un paquet non acquitté doit être renvoyé) s'est écoulé.

Ce champ timestamp permet donc aussi de savoir si un paquet a déjà été envoyé. En effet, ce champ timestamp n'est différent de 0 que si le paquet a été envoyé. Les numéros de séquence étant cycliques, nous pouvons donc ne pas confondre différents paquets portant le même identifiant.

Le RTO à été déterminé en fonction de la latence du réseau. Comme celui-ci varie entre 0ms et 2000ms, nous avons décidé de prendre 4 secondes de RTO. Ceci permet à un paquet d'être envoyé et acquitté en partant du principe que son temps de traitement par le `sender` et le `receiver` est négligeable. Cette valeur ne varie pas dans le temps.

3.2 Utilisation de paquets PTYPE_NACK

Les paquets de type PTYPE_NACK (i.e. acquittement négatif ou plus simplement `NACK`) sont utilisés en parallèle des paquets de type PTYPE_ACK qui, eux, signalent au `sender` que tous les paquets qui ont été envoyés (jusqu'au `seqnum` communiqué exclu) ont bien été reçus et que les paquets à partir de ce `seqnum` compris sont attendus. Le `sender` sait qu'il peut donc oublier ces paquets acquittés.

Un `NACK` signale au `sender` qu'un packet a bien été reçu mais n'est pas valide, il peut avoir été tronqué ou corrompu par le réseau.

Dans le premier cas, le `sequnum` sera égal à celui du paquet tronqué. Si le paquet a été corrompu, on ne peut croire les informations qu'il contient. Le `seqnum` envoyé correspond donc au prochain paquet attendu par le `receiver`.

3.3 Buffer

Chaque programme (i.e. `sender` et `receiver`) utilise un buffer de type liste doublement chaînée a des fins qui lui sont propres :

- **`sender`** : Le buffer contient l'ensemble des paquets envoyés ou non mais n'ayant pas été acquittés avec un `ACK`. L'idée est que, si le `receiver` exige une retransmission d'un packet qui n'a pas encore été acquitté, qu'il ait été envoyé ou non, le `sender` est capable de le (re)trouver et le (re)transmettre.
- **`receiver`** : Le buffer contient les éléments appartenant à la fenêtre de réception n'ayant pas encore été écrits dans le fichier de sortie. C'est-à-dire que, si on attend un certain numéro de séquence mais que l'on reçoit les suivants (appartenant à la fenêtre glissante), on les enregistre et les imprime une fois que tous les paquets peuvent être imprimés dans l'ordre.

3.4 Fast-Retransmit

Afin d'optimiser la vitesse de transfert, nous avons décidé de mettre en place un système de Fast-Retransmit. C'est à dire que si un même `ACK` (i.e. avec le même numéro de séquence) est reçu 3 fois par le sender, celui-ci va ré-envoyer immédiatement le paquet portant ce numéro de séquence et donc pas attendre l'expiration du RTO.

3.5 Test-driven development

Afin de s'assurer que le code rédigé était correct, nous avons utilisé les processus de développement TDD [2] (i.e. Test Driven Development) et Bug Driven Development. En effet, avant d'implémenter les fonctions relatives au buffer (`stack`), nous avons défini les besoins et donc les spécifications des différentes fonctions à implémenter, rédigé les tests assurant le bon fonctionnement de celles-ci et puis finalement écrit le code du buffer qui devait passer les tests pour s'assurer qu'il soit correct. Si par la suite, nous constations des bugs qui n'étaient pas pris en compte, nous créions d'abord les tests avant de résoudre le problème.

Ce processus a également été utilisé pour le développement des fonctions `pkt_encode()` et `pkt_decode()` du fichier `packet.c` [3]. Nous avions des attentes spécifiques en fonction des paramètres entrés, nous avons juste du insérer celles-ci dans des tests et le tour était joué.

4 Tests d'interopérabilité

Durant la séance d'interopérabilité du 19 octobre, nous avons été en mesure d'envoyer et recevoir des paquets avec les 5 groupes avec qui nous avons essayé. Cela a même permis de debugger le programmes de certains d'entre eux afin de trouver où se trouvaient leurs problèmes. L'implémentation du protocole répondait donc aux attentes et était compatible avec celle des autres groupes.

2. https://technologyconversations.com/2013/12/20/test-driven-development-tdd-example-walkthrough/
3. Le test relatif a été ajouté dans le code après coup mais a été utilisé durant la réalisation de la tâche Inginious https://inginious.info.ucl.ac.be/course/LINGI1341/format-des-segments

5 Changements apportés depuis la première soumission

La première soumission, comme expliqué dans le rapport qui lui était associé, était fonctionnelle mais ne répondait pas encore à l'ensemble des points du cahier des charges. Depuis lors, voici les différentes modifications apportées au code et à la logique des programmes :

— **`Limite de paquets`** : Le code permet maintenant d'envoyer des fichiers représentant plus de 255 paquets. Les numéros de séquence peuvent donc être réutilisés lors de l'envoi de nouveaux paquets.

— **`Utilisation des NACK`** : Les `NACK` ne sont plus utilisés à chaque fois qu'un paquet qui n'était pas celui attendu est reçu mais uniquement lorsque le paquet a été tronqué ou corrompu. Dans les autres cas, on envoie juste un ACK avec le numéro de séquence attendu. Cela nous a donc permis d'implémenter un « fast retransmit » qui permet donc de renvoyer un paquet sans attendre que son RTO n'expire.

— **`Entrée standard`** : Le cas où le fichier à envoyer se trouve sur l'entrée standard est maintenant pris en compte. L'ensemble de l'entrée standard est considérée comme un fichier.

6 Robustesse et évaluation du code

Afin de déterminer la robustesse de l'implémentation, nous allons, dans le cas de l'envoi d'un fichier image d'une taille de 155 835 Bytes, compter le nombre de packets de type PTYPE_DATA qui doivent être envoyés en fonction du pourcentage de paquets perdus. Il va de soi que le minimum est de $ceil\left(\frac{155835}{512}\right) = 305$ paquets sans compter celui marquant la fin de la connexion. La taille de la fenêtre de réception est de 31.

Pourcentage de paquets perdus (%)	Nombre de PTYPE_DATA envoyés	Durée de l'envoi (s)
0	306	0.038
5	336	5.59
10	462	21.58
25	615	54.46
50	1110	154.63

On constate dès lors que le nombre de paquets PTYPE_DATA à envoyer dépend du nombre de paquets perdus et la durée de transmission augmente en parallèle.

Ces longues durées peuvent être expliquées par le fait que le retransmission timer doit entrer en jeu lorsque le pourcentage de paquets perdus est élevé (et le fast retransmit n'est pas suffisant). Ce qui veut dire que le programme attend 4s avant de réenvoyer des paquets.

Conclusion

En cinq semaines, nous avons pu (1) analyser la problématique qui nous a été posée et (2) développer une version du programme répondant le plus possible au cahier des charges. En structurant dès le départ notre développement, nous avons pu attaquer l'implémentation en différentes phases facilitant la répartition du travail et la compréhension de chaque partie.

Il est tout de même à noter que notre programme présente quelques faiblesses lorsque le delai de transfert des paquets n'est pas constant (lorsqu'on utilise une valeur élevée de `jitter` avec le simulateur de liens [4]).

Néanmoins, les codes proposés pour `sender` et `receiver` implémentent correctement le protocole TRTP. Ils sont fonctionnels, gèrent les cas non traités dans le code rendu lors de la première soumission et ne présente donc que la faiblesse énoncée ci-dessus.

L'analyse de robustesse a aussi pu nous montrer que la quantité de paquets envoyée est raisonnable dans le cas de pertes plus ou moins élevées, comme discuté précédemment.

4. `https://github.com/oliviertilmans/LINGI1341-linksim`