UFR D'Informatique

Université de Paris / Université Paris Diderot

Projet de réseau

Tchat via inondation fiable

Auteurs :

Étienne Marais

Xavier Durand

Enseignant :

Juliusz Chroboczek

Année académique 2018-2019

Table des matières

Introduction

Ce document est le rapport du projet de programmation réseau.
Ce projet consiste à implémenter un tchat en réseau à l'aide du principe d'inondation fiable.
L'inondation fiable fonctionne de la façon suivante :

- Tous les membres du protocole sont liés en pair à pair.
- Lorsqu'un membre reçoit une donnée, il l'inonde à tous ses pairs symétriques.
- Il attend un message d'acquittement de tous les pairs qu'il a inondés, et continue d'envoyer les données tant qu'il n'a pas reçu d'acquittement.
- Pour que deux pairs soient pairs entre eux, on procède à un échange de données d'identification (id).

Nous avons ensuite défini que les données transmises sur le réseau seraient sous la forme de TLV, c'est à dire de la forme :

Type	Length	Value ···

avec le type et la taille sur un octet.
Pour identifier que nous communiquons bien avec les personnes implémentant le protocole, nous avons défini que tout TLV était encapsulé dans un datagramme contenant le champ *magic* à 93 et le champ *version* à 2.
Enfin, nous allons vous présenter comment nous avons implémenté l'ensemble du protocole et les extensions que nous lui avons apportées.

1

Manuel

1.1 Prérequis

Pour pouvoir lancer le programme, il faut que les bibliothèques C suivantes soient installées :

- **ncurses** : interface graphique
- **math.h** : librairie math de c
- **pthread** : utilisation de threads

Il est aussi nécessaire d'avoir les programmes suivants pour pourvoir compiler ou afficher les rapports de couverture de code :

- **Makefile** : sert pour compiler le programme
- **lcov** : permet d'obtenir la *couverture* du programme (surtout pour le développement)
- être sur un ordinateur sous Linux : nous ne savons pas si le programme marche sur Windows (l'API socket étant légèrement différente) ou Mac, mais il ne marche pas sur téléphone.

1.2 Compilation

Tout d'abord, il faut télécharger le dossier. Vous pouvez soit déziper le .zip soit cloner le dossier git. Pour cela, il faut utiliser *git*, et lancer, dans le dossier que vous voulez, la commande :

```
git clone https://github.com/kolibs/Floodus.git
```

Ensuite, vous vous mettez dans le dossier *Floodus/*, puis vous lancez la commande *make floodus*, et enfin vous exécutez le fichier *floodus* :

```
cd Floodus
make floodus
./floodus
```

Vous pouvez, de plus, obtenir la *couverture du code* de notre programme en lançant les commandes suivantes :

```
make coverage
```

Cela vous permet de savoir quelles parties de notre code vous avez testé. Il faut avoir préalablement lancé le programme. Ensuite, pour voir le résultat, vous affichez dans votre navigateur préféré le fichier "coverage.html/index.html".

Enfin, il est possible de voir toute la documentation du programme en lançant la commande :

```
make doc
```

Et ensuite, toujours avec votre navigateur préféré, vous ouvrez le fichier "doc/html/index.html".

2

Notre programme en globalité

Notre programme se sépare en 4 parties importantes.

2.1 Le lecteur

Cette partie du programme ne s'occupe que de lire les données qui arrivent sur la socket de communication.
Lorsque celui-ci reçoit une donnée, il la transfère au noyau central. C'est le noyau principal qui s'occupe ensuite de redistribuer les tâches. Il faut savoir que lorsque le lecteur reçoit des données, nous interprétons ces données à la volée, de façon synchrone, à la différence de l'écrivain.

2.2 L'écrivain

Le rôle de ce module est de gérer de tout ce qui est écriture sur la socket. L'écriture se fait de manière asynchrone. Pour cela, nous maintenons un buffer d'écriture sous la forme d'une liste chaînée de :

```
struct buffer_node_t {
    ip_port_t dest;              /* couple ip-port de la destination */
    data_t *tlvs;                  /* tableau de TLV à envoyer */
    size_t tlvlen;                 /* taille du tableau TLV */
    struct buffer_node_t *next;    /* nœud suivant du buffer */
};
```

Nous avons choisi de faire l'écriture de façon asynchrone car on ne sait pas si la socket est toujours disponible en écriture. En effet, elle peut être bloqué par la lecture. Nous pouvons ici procéder à l'agrégation des données, simplement, mais nous vous détaillerons cela dans une autre partie.

2.3 Le noyau central

Ce module s'occupe de répartir toutes les tâches à faire entre les différents modules. Il s'occupe aussi de lancer l'inondation et de maintenir les listes de voisins. Il donne la main au lecteur lorsque celui-ci a des données à lire, et à l'écrivain lorsque celui-ci a des données à écrire grâce à une boucle de contrôle. Il est séparé en deux parties qui seront détaillées plus loin :

- "l'inondeur" : il s'occupe de tout ce qui touche à l'inondation.
- le voisin : il s'occupe de tout ce qui est relation de voisinage et maintien de la liste des voisins.

2.4 Les timeurs

Ce module correspond à deux threads qui sont lancés au début du programme et qui vont s'occuper de compter le temps qu'il reste avant d'envoyer les TLVs *neighbour* et les TLV *hello long* et *hello court*.
Dans notre implémentation, nous avons choisi d'envoyer les TLV *neighbour* à tous les voisins symétriques toutes les minutes.
Pour les TLV *hello long*, nous les envoyons toutes les 30 secondes. Les TLV *hello court* sont envoyés en même temps que les *hello long* à tous les voisins possibles s'il n'y a pas assez de voisins (a)symétriques (nous avons fixé ce minimum à 8).

Le deuxième thread qui envoie les ip/port des voisins à l'ensemble des voisins symétriques prend soin de ne pas envoyer sa propre ip à chacun des voisins.

Nous avons choisi d'utiliser les threads pour une partie de l'implémentation afin de réduire les calculs sur le temps pour des actions répétées périodiquement (30 et 60 secondes). En effet, nous avons pu automatiser ces tâches pour s'exécuter à intervalle régulier. Cela nous a aussi permis de découvrir une des fonctionnalités du C que nous n'avions pas vu en cours. Enfin, nous avons dû utiliser des mécanismes de verrous pour éviter des accès en concurrence des données et nous assurer que nous n'avions pas de famine.

Enfin le reste du temps est contrôlé par une boucle et un *pselect*.

3

Parties implémentées

Nous allons, dans cette partie, expliquer l'implémentation du programme.

3.1 Sujet minimal

3.1.1 Hashmap

Tout d'abord, parlons en détail de comment nous avons choisi de réaliser la partie minimale. Il faut noter que nous avons écrit nous-même une hashmap que nous allons grandement utiliser dans tout le programme.
Cette hashmap est implémentée de telle façon à ce qu'elle soit un tableau de listes chaînées de couple (clé, valeur). Tous les éléments (clés et valeurs) contenus dans la hashmap sont stockés selon la structure suivante :

```
struct iovec {
    void  *iov_base;    /* Adresse de début */
    size_t iov_len;     /* Nombre d'octets à transférer */
};
```

Ensuite, la fonction de hachage est une fonction opérant sur les bits des données. Enfin, nous avons fixé la taille de la hashmap à 4096 éléments.

3.1.2 Maintien de la liste des voisins

Dans le cadre de ce projet, il est nécessaire de maintenir deux listes de voisins :

- la liste des voisins potentiels
- la listes des voisins courants (asymétriques et symétriques)

La liste des voisins potentiels contient les voisins dont nous n'avons pas eu de réponse depuis longtemps ou que nous avons reçu par un TLV Neighbour. Nous nous servons de cette liste

afin d'augmenter le nombre de voisins courants si nous avons moins de 8 voisins courants. Les seuls informations que nous possédons sur ces voisins sont le port et l'ip.

La liste des voisins courants contient, elle, les voisins qui ont envoyé un "hello" récemment. Les voisins ayant envoyés un hello long depuis plus de deux minutes sont conservés dans la liste des voisins courants mais sont considérés comme asymétriques lors du fonctionnement du programme. Nous ne leur envoyons pas de données pour l'inondation ou via les TLV neighbour (par sûreté).

Ces deux listes sont maintenues dans des hashmaps permettant d'avoir un accès direct à leurs valeurs. Les voisins courants sont stockés dans la structure suivante :

```
struct neighbour_t {
     u_int64_t id; /* Identifiant unique */
     struct timespec hello; /* Dernier hello reçu */
     struct timespec long_hello; /* Dernier hello long */
 };
```

3.1.3 Inondation fiable

Ce module a pour rôle de s'occuper de la partie inondation du protocole.
Pour cela, nous maintenons une liste chaînée triée, selon le temps restant avant la prochaine inondation, des messages que nous recevons et que nous envoyons. Elle est composée de :

```
struct message_t {
    struct timespec send_time; /* temps absolu à partir duquel on peut
    envoyer le message */
   u_int8_t count;          /* compteur du nombre d'envois effectués */
    u_int8_t type;             /* type du message */
    data_t *content;           /* contenu du message */
   u_int64_t id;             /* id de celui qui a envoyé le message */
    u_int32_t nonce;           /* nounce rendant le message unique */
    hashmap_t *recipient;      /* ensemble de ceux à qui on doit
    envoyer le message sous forme ( (ip,port) -> (ip,port) ) */
    struct message_t *next;    /* message suivant dans la liste */
}
```

Lorsque nous recevons un TLV de type *data*, nous regardons si le message n'est pas déjà dans la liste des messages à inonder.
Si oui, nous répondons par un TLV *ack*, et nous enlevons la source de la hashmap *recipient*. Dans le cas, où la source est déjà enlevée, nous envoyons quand même un TLV *ack* car cela signifie que le précédent *ack* n'a pas été reçu.
Si non, nous créons un nouveau message que nous initialisons avec les informations contenues dans le TLV *data*. Nous mettons à *send_time* la valeur du temps courant + 1 et à *recipient* tous les voisins symétriques du moment sauf la source du TLV *data*. Ensuite, nous ajoutons le message dans la liste des messages à inonder. Puis, nous traitons le TLV *data* en fonction de son type. Enfin, nous envoyons un TLV *ack*.

L'inondation procède de la façon suivante :

1. La liste des messages est triée par le temps d'envoi. Pour savoir s'il y a une donnée à inonder, nous regardons juste la tête de la liste.

2. Lorsque le temps de la tête de lecture est dépassé, nous ajoutons au buffer d'écriture un TLV *data* avec pour destination chaque membre de *recipient*.

3. Si le message a été inondé trop de fois (ici nous avons fixé à 6), nous envoyons un TLV *go_away* à tous les voisins restants (avec le bon code d'erreur) dans *recipient* et nous supprimons le message de la liste.

Notons qu'ici, un message peut rester dans la liste même s'il n'a plus de voisins à inonder car nous voulons garder les messages que nous avons déjà reçu en mémoire pendant un certain temps.

3.2 Extensions

Nous allons maintenant vous présenter les différentes extensions que nous avons mises en place dans ce projet.

3.2.1 Interface graphique à l'aide de *NCurses*

Pour permettre à l'utilisateur d'utiliser le protocole nous avons choisi de mettre en place une interface graphique qui permet de dialoguer en mode texte. Elle est composée d'un tampon d'écriture, dont la limite, d'abord fixée à 242, a été étendue avec l'apparition des TLV de

taille 220. La deuxième partie de l'interface graphique est un cadre qui affiche les informations à l'utilisateur. Les messages de l'utilisateur sont affichés en bleu, de même que les informations fournies par le système. Ceux des autres pairs sont affichés en blanc.
Dans le cas où le programme a été compilé avec le flag de DEBUG à 1, les messages pour débuguer sont aussi affichés dans cette zone. En outre, l'interface offre la possibilité d'effectuer un certain nombre de commandes pour l'utilisateur.

Il est possible d'obtenir des informations sur ses interfaces (ip, etc), son port et son nom en utilisant :

```
/i
```

Par défaut, l'utilisateur se connecte sur l'interface graphique avec le nom *Nobody*. L'utilisateur peut choisir de changer de surnom dans la limite de 30 caractères (le reste est tronqué) en utilisant la commande :

```
/s [name]
```

Afin de permettre à l'utilisateur d'ajouter manuellement des voisins courants (par défaut nous nous connectons à Juliusz), nous avons implémenté une commande de connexion :

```
/c [ip ou petite url] [port]
```

Enfin pour pouvoir stopper proprement l'interface graphique et nettoyer la mémoire, nous avons écrit une commande d'arrêt :

```
/q
```

3.2.2 Calcul de PMTU

A chaque envoi de datagrammes, nous déterminons à l'aide d'une socket connectée temporairement quel est le PMTU. Ici, nous ne prenons en compte que le calcul synchrone (par le système) du PMTU : nous ne calculons pas le PMTU à l'aide des messages ancillaires de façon asynchrone.
Lorsque nous avons le PMTU, nous remplissons le datagramme à envoyer avec le maximum de TLV possible.

3.2.3 Gestion des exceptions

Nous gérons toutes les exceptions possibles que peut nous renvoyer le programme. Notamment, lorsque l'utilisateur fait un *ctrl+c*, nous catchons le signal pour pouvoir quitter le

programme proprement (libération de mémoire).
Ensuite, lorsque nous avons une exception système, nous arrêtons le programme en envoyant des TLV *go_away* avec le code 1, si possible. Enfin, lorsque nous recevons une exception réseau (par exemple *network unreachable*), nous testons la connexion pendant 5 secondes avant de quitter le programme si cela n'aboutit pas.

3.2.4 Concurrence

Grâce au système de buffer d'écriture, nous pouvons envoyer des données de façon asynchrone, et donc procéder à plusieurs inondations en même temps.

3.2.5 Agrégation des tlvs

À l'aide du buffer d'écriture, nous pouvons procéder à une agrégation des données à envoyer à un même destinataire.
Nous avons défini dans la partie d'avant (section 2.2) le buffer de l'écrivain. Nous remarquons le champ *dest* dans *buffer_node_t*. Ce champ permet, à chaque ajout d'un TLV à envoyer sur le réseau, d'agréger les données pour un même destinataire. Nous avons défini la limite de 2800 octets de TLV par noeud du buffer, pour ne pas inonder toujours la même personne si nous avons beaucoup de données à transmettre.

3.2.6 Utilisation message ancillaire

Dans notre programme, nous traitons une partie des messages ancillaires, notamment pour les adresses sources (pendant la réception) et les adresses destinations (pendant l'envoi) du message. Nous vérifions, notamment, que l'interface sur laquelle nous recevons les données est bien celle à laquelle nous nous attendions. Nous faisons de même pour l'ip.
On procède ainsi à une sécurité du protocole.

3.2.7 Sécurité de l'implémentation

Notre implémentation est robuste au non-respect du protocole défini. Si nous recevons des datagrammes qui n'ont pas le bon format, ou, si nous avons des TLV qui n'ont pas le bon format, alors nous envoyons un TLV *warning* avec un message correspondant à la source.

3.2.8 Fragmentation

La dernière extension mise en place est la possibilité d'envoyer des données textuelles allant jusqu'à 60Ko (environ) à l'aide du TLV *data* de type 220.

Nous avons introduit la structure suivante :

```
struct big_data_t {
    u_int8_t *content;      /* contenu du gros message */
    u_int16_t contentlen;   /* taille du gros message */
    u_int16_t read_nb;      /* nombre d'octets remplis */
   struct timespec end_tm; /* temps limite de conservation du message en mémoire */
    u_int8_t type;          /* type du gros message */
};
```

Elle correspond à la structure utilisée pour chaque gros message.
À chaque réception d'un TLV *data* de type 220, nous regardons si nous n'avons pas déjà lu le TLV. Si non, nous nous situons dans deux cas différents :

1. si le gros message n'existe pas, nous créons un buffer contenant le contenu du TLV
2. s'il existe déjà, nous le remplissons avec les nouvelles données reçues

Le champ *end_tm* permet de ne garder un gros message qu'un certain temps (ici 5 minutes).
Ensuite, lorsque le message est rempli, nous l'interprétons en fonction du type :

- Si le type vaut 0 alors cela correspond à un texte en utf8 qu'on affiche alors à l'écran de l'utilisateur
- Si le type est autre, on ne fait rien. Cependant, une extension possible serait de pouvoir transmettre des fichiers binaires, des images, des gifs, des fichiers textes, etc... Ces extensions sont ajoutables. Cependant, le protocole n'étant pas conçu, à l'origine, pour ce genre de transfert et l'interface ne permettant pas de les afficher proprement, nous ne sommes pas allés plus loin.

4

Commentaires et extensions possibles

Voici une liste non-exhaustive des extensions que nous n'avons pas eu le temps d'implémenter :

- Adresses multiples : détection qu'un pair a plusieurs adresses ip sur le réseau, et n'envoyer les données qu'à une seule de ses adresses.
- Sécurité cryptographique : permettre de transmettre des données cryptées. Le protocole est basé sur le principe que tout le monde reçoit toutes les données, donc une cryptographie propre à l'implémentation n'est pas forcément adaptée à ce protocole (cela reste subjectif).
- Utilisation multicast : envoi des TLVs *hello court*, *hello long* et *neighbour* via un multicast.
- Transfert de fichiers quelconques.
- Interface graphique : avoir une interface graphique plus *user-friendly* et plus interactive avec le programme.

Ce protocole est très bien pour comprendre le principe d'inondation. De plus, la structure de l'inondation utilisant les TLV, cela rend le protocole modulable avec la possibilité de l'étendre grâce à des extensions de TLV. Le fait d'ignorer les TLV de type inconnu permet à certains pairs d'utiliser les extensions sans gêner le fonctionnement global de l'inondation. Néanmoins, il ne permet pas d'avoir plusieurs conversations différentes avec des groupes de personne différents. Il est conçu pour une discussion avec l'ensemble des pairs. Enfin, la taille des TLV étant codée sur un octet, cela ne permet pas de transmettre beaucoup d'informations, malgré la fragmentation qui risque de vite saturer les autres pairs.

Conclusion

Pour conclure, nous pouvons dire que le projet nous a permis d'implémenter un protocole de communication pair-à-pair et de mettre en pratique les techniques d'utilisation de la communication UDP. Cela nous a montré les avantages qu'offre ce protocole de couche transport tel que la modularité tout en nous confrontant à ses inconvénients. En outre, nous avons pu constater la réalité du réseau et à son principe de non-fiabilité qui nous oblige à anticiper les absences de pairs ou une rupture du réseau soudaine. Enfin, ce projet nous aura permis d'effectuer en groupe un projet en langage C, proche du système, complétant, ainsi l'enseignement du semestre 5.