Rapport PR6_IRC_UDP

Raphaëlle Gauchée et Joseph Priou

12 Mai 2019

1 Introduction

Ce document est le compte rendu de notre projet de programmation réseau que nous avons fait au sein du cours de Programmation Réseau assuré par Juliusz Chroboczek pendant le dernier semestre de notre 3ème année de Licence de Mathématiques-Informatiques.

2 Manuel

2.1 Sujet

Nous avons implémenté un chat en pair à pair en UDP, qui fonctionne grâce à un système de voisins et un protocole d'inondation. Vous pouvez trouver le sujet précis du projet au sein de notre projet git.

2.2 Compilation

Pour compiler le programme, il suffit de taper "make" dans le dossier du projet. Nous avons pour seule dépendance la librairie ncurses.

2.3 Exécution

Notre exécutable s'appelle "./irc_udp". Le programme demande à l'utilisateur d'entrer un nom puis l'utilisateur peut taper les messages qu'il veut dans la console. L'écran est séparé en deux parties, celle ou l'utilisateur tape ses messages et celle ou il peut voir les messages envoyé sur le chat.

Une liste d'option est donné si vous lancez le programme avec une option inconnue, par example "./irc_udp test". Pour quitter le programme, il suffit de taper "QUIT".

```
Help:
./irc_udp --no-ncurses --with-logs --no-juliusz-init
          --no-port --port NUM_PORT

    --no-ncurses : launch the application in raw mode
    --with-logs : launch the application with logs on the screen
    --no-juliusz-init : launch the application without juliusz connexion
    --no-port : launch the application with no port specified
    --port NUM_PORT : launch the application with port NUM_PORT specified

If some contradicory informations are given, the last one is kept
```

3 Implémentation

3.1 Environnement

Le programme implémente toute la partie minimale demandée, donc la liste des voisins, l'inondation et l'envoie de message. L'utilisateur est défini par un environnement, qui contient un id tiré au hasard, une socket pour toutes les communications, une seconde socket pour la découverte de voisins en multicast (local), le temps actuel et cinq listes : une pour les voisins, une pour les voisins potentiels, une pour les messages reçus, une pour les messages dont on attend un acquittement et une avec les TLV qu'on doit envoyer.

```
typedef struct      s_env{
    uint64_t       id;
    int            sock;
    int            sock_multicast;
    struct timeval now;
    t_bool         need_neighbours;
    t_list *       li_neighbours;
    t_list *       li_potential_neighbours;
    t_list *       li_messages;
    t_list *       li_acquit;
    t_list *       li_buffer_tlv_ip;
}                   t_env;
```

Cet environnement est alloué au début du programme et est stockée dans une variable globale

3.2 Gestion de l'interface utilisateur

Nous avons décidé de séparer la partie réseau de la partie interaction avec l'utilisateur. L'une de nos premières actions consiste à fork notre programme afin que le processus fils gère la partie I/O de notre projet. Puis on communique avec ce processus à l'aide de pipe.

Dans le cas où le programme est lancé sans ncurses, un troisième processus est mis en place pour gérer le read bloquant de stdin. Nous ne voulons surtout pas changer les paramètres de notre terminal.

Toute l'implémentation de notre gestions des processus et de nos interfaces graphiques se situent dans le dossier "src/ui/".

3.3 Détails d'implémentation

Nous avons favorisé le code pratique au code optimisé, c'est pourquoi notre programme ressemble à une game loop dont le temps d'arrêt est variable (maîtrisé par un select). Certaines étapes pourraient être plus optimisés à l'aide d'algorithmes ou de structures de données plus adaptées, nous avons choisi de faire au plus simple et de faire une implémentation simple et complète.

3.3.1 Structures de données

La seule structure de données abstraite est une simple liste chaînée, avec laquelle nous avons beaucoup de fonctions de manipulations, à savoir add, remove, size, find, findall, removeif, etc... Voir le fichier t_list.h

3.3.2 Déroulement du programme

Notre programme se déroule en trois phases:

- Phase 0 : Gestion du réveil, par le réseau, par le multicast, par une entrée par l'utilisateur ou par un timeout
- Phase 1 : Analyse de notre environnement. Comment on réagit par rapport à notre environnement
- Phase 2 : On construit puis on envoie tous nos messages

Pour gérer les conditions d'initialisations, on commence à la phase 1, et on effectue les phases dans l'ordre dans une boucle. C'est la fonction core_loop qui effectue la boucle qui se situe dans le fichier "src/core/core_loop.c".

3.3.3 Découverte de voisins

On attend de recevoir des TLV neighbours pour avoir des voisins potentiels. Si on a moins d'un certain nombre de voisins, on envoie des TLV hello court à des potentiels voisins dont on a pas envoyé de hello court depuis un certain temps. Quand on reçoit le premier hello long d'un voisin potentiel on lui renvoie un hello long pour confirmer la relation de symétrie. À chaque tour de boucle il y a un certain pourcentage de chance que nous même on envoie à un voisin aléatoire, un TLV neighbour d'un autre voisins aléatoire.

3.3.4 Processus d'inondation

Quand on reçoit un message, on rajoute dans notre liste des messages à acquitter à tous nos voisins hormis celui qui vient d'envoyer le message. On attend au moins 1 seconde avant d'envoyer le message. Le temps entre chaque renvoie de message est tiré aléatoirement entre 2^{n-1} et 2^n, avec n le nombre de silence reçu. On effectue le même algorithme si nous voulons envoyer un message.

3.4 Architecture

Voici notre architecture de fichier.

```
doc/
inc/
src/
 core/
    env_manip/
    parse_tlv/
    process_step/
 struct/
    data/
    network/
    protocol/
 ui/
 utils/
```

Le dossier "doc/" contient la documentation de notre projet, à savoir le déroulement des 3 phases, le reste est documenté dans les headers ou a été dit à l'oral entre nous, dans ce cas, il est présent dans ce rapport.

Le dossier "inc/" contient nos fichiers headers.

Le dossier "src/" contient notre code source.

Le sous-dossier "core/" contient l'implémentation principale de notre programme, et est réparti en 3 sous dossiers : "env_manip", "parse_tlv" et "process_step". Le dossier "env_manip" contient toutes les actions atomiques que nous pouvons faire sur notre environnement à savoir effacer un voisin, chercher un message par un acquittement, etc... Le dossier "parse_tlv" analyse ce que nous avons reçu sur le réseau, et réagit en conséquence : ajout d'un voisin potentiel, renvoi d'un hello long pour assurer la bidirectionnalité d'une relation... Le dossier "process_step" contient les étapes qui définissent la phase 1, comme la suppression de voisins trop vieux, ou le renvoi de messages non acquittés... etc.

Le sous-dossier "struct/" contient l'implémentation de nos structures que nous utilisons réparti dans 3 sous-dossiers : "data/" représente nos structures de données abstraites, nous n'avons que la simple liste chaînée t_list, "network/" contient des structures de données qui nous permettent de rajouter une couche d'abstration nécessaire pour la construction et l'envoi de messages :

"t_iovec_builder" ou "t_tlv_builder". Enfin le sous-dossier "protocol/" contient les structures de données relatives à l'implémentation du protocole d'inondation

Le sous-dossier "ui/" contient notre gestion des processus ainsi que les interfaces utilisateurs.

Le sous-dossier "utils/" contient toutes sorte de fichiers utilitaires qui ne peuvent pas être placé ailleurs car utiles à beaucoup d'endroits.

3.5 Extension

Nous avons fait quelques extensions, on souhaite faire la différence entre les extensions que nous avons faites, et des extensions que nous pouvons faire facilement avec notre base de code.

3.5.1 Extensions faites

- Concurrence : nous gérons plusieurs inondations en même temps. Nos structures de données sont pensées pour être asynchrone et non bloquantes sur une action donnée.
- Agrégation (partielle) : Nous gérons les TLV et avons un algorithme d'optimisation de la répartition des TLV sur plusieurs messages si besoin il y a, par rapport à un Global MTU qui est fixé à 1024. Nous n'avons pas fait la découverte d'un PMTU.
- Optimisation (partielle) : Nous tirons un temps aléatoire pour optimiser l'inondation d'un message parmi nos voisins.
- Sécurité de l'implémentation : Nous vérifions soigneusement toutes les informations que l'on nous donne et affichons un message d'erreur si quelque chose d'inattendue se produit
- Découverte Multicast : Nous pouvons découvrir des voisins en multicast local.
- 2 interfaces graphiques : Nous pouvons lancer notre programme dans 2 interfaces graphiques : une textuelle uniquement sans gestion d'affichage entre ce que l'on tape et ce qu'on affiche, et une à l'aide de ncurses où une séparation nette est faite entre le ce que l'on veut écrire et ce que l'on veut afficher.
- Un système de gestion des logs : À chaque utilisation, nous créons un fichier de logs qui permet de retracer l'exécution du programme. Nous pouvons aussi afficher les logs pendant l'éxécution dans l'interface graphique.

3.5.2 Extensions faciles à implémenter

- Calcul du PMTU
- Gestion des voisins obtenus en locales (ne pas les envoyer en Neighbour notamment.)
- Fragmentation
- Données non textuelles
- Transfert de fichiers

Nous n'avons pas fait le calcul du PMTU ni la gestion des voisins locaux par manque de temps, ni les 3 dernières car elles ne sont pas représentatives de notre capacité à pouvoir faire de la programmation réseau.

3.6 Outils externes

Nous avons utilisé ncurses pour l'interface graphique, git pour le gestionnaire de versions, github pour héberger le code source ainsi que uncrustify pour que notre code reste épuré et commum.

3.7 Bugs

Nous avons quelques bugs d'affichage pour le mode ncurses (calcul de padding). Il est également possible que ncurses ne marche pas chez vous, nous n'avons pas réussi à trouver tous les bugs...

4 Conclusion

Le programme est un chat en réseau UDP qui implémente donc un protocole d'inondation, une gestion des voisins, et une interface utilisateur. On peut ainsi communiquer avec toutes les personnes connecté sur le chat. Le programme interopère avec les autres implémentations de ce même protocole.