Projet de programmation réseau

miaouchat : « Groupe de discussion par inondation fiable »

Félix Desmaretz + Louis Gavalda

Nous avons implementé le protocole spécifié dans le sujet proposé par Juliusz Chroboczek afin de créer un groupe de discussion pair à pair (ou *chat P2P*, ça sonne mieux).

Ce rapport rend compte des choix qui ont été faits et détaille la logique de notre implémentation.

Contents

0.1 Utilisation

0.1.1 Compilation (tl;dr: `make && ./miaouchat`)

Il faut bien sûr compiler le programme avant de pouvoir le lancer. Les commandes suivantes sont à exécuter à la racine du projet et ont les effets suivants :

- `make` génère un fichier exécutable nommé `miaouchat` : c'est notre programme, fraîchement compilé, bien joué ! Vous pouvez ensuite le lancer en tapant `./miaouchat` dans votre terminal.
- `doxygen Doxyfile` génère automatiquement la documentation (que ce compte rendu vient compléter) à partir de commentaires présents directement dans les sources. Ensuite, la commande `./manual` vous permet d'ouvrir ladite documentation dans un nouvel onglet de votre navigateur.

0.1.2 Interface graphique (tl;dr: on peut discuter avec les autres gens)

Lorsque **miaouchat** se lance, une fenêtre apparaît. Rien d'extravagant : nous avons voulu nous rapprocher du style minimaliste des interfaces IRC d'antan, dans lesquelles une discussion est réduite à l'essentiel.

En haut à droite s'affiche le nombre de pairs/voisins symétriques auxquels notre programme est connecté. Au milieu, la plus grande partie de la fenêtre contient la liste des messages reçus (stockée seulement en mémoire). Pour discuter, c'est enfantin : il suffit d'écrire un message dans le champ de texte (en bas de la fenêtre) puis d'appuyer sur Entrée pour l'envoyer.

L'heure d'envoi est spécifiée avant chaque message, ainsi que — normalement — le pseudo de l'émetteur. On a donc un affichage qui ressemble plus ou moins à :

```
[14:26:44] <bob> mais c'est top !!
[14:26:55] <alice> comme tu dis ;-)
[14:27:02] <bob> au top
[14:27:06] <bob> miaou
[14:27:18] <alice> wouf
[14:27:25] <bob> ...
```

Des messages à propos de l'exécution du programme peuvent également s'afficher (en orange).

1 Sous le capot

1.1 Particularités du projet (tl;dr: protocole bien pensé pour l'usage)

On peut remercier l'auteur du sujet d'avoir réfléchi sérieusement à son protocole. La spécification y est relativement précise, et des conseils sont mêmes fournis afin de nous guider dans l'implémentation.

Il s'agit d'un protocole pair à pair dans lequels les messages sont « inondés » sur le réseau.

Chaque membre du réseau reçoit l'intégralité des messages, qu'ils s'adressent à lui ou non. (Un membre, ou nœud, est un logiciel qui exécute une implémentation du protocole ; il forme un réseau avec l'ensemble des autres membres avec qui il est « connecté ».)

Lorsqu'un message adressé à un récepteur précis est émis, ce message est transmis à chacun des voisins de l'émetteur via **UDP**. De façon récursive, chacun transmet à son tour le message à ses voisins jusqu'à ce que tous les membres du réseau l'aient finalement reçu.

Une telle manière de procéder induit évidemment des limitations, et c'est pourquoi on se contente d'utiliser ce protocole pour échanger des messages textuels. Sans même parler de fragmentation, si un message/fichier lourd devait s'échanger entre deux utilisateurs, il se verrait copié par chacun des membres du réseau et cela constituerait un terrible gaspillage, notamment de bande passante.

Notre programme, dans sa partie réseau, s'appuie exclusivement sur **UDP** qui, comme vous le savez, est dépourvu de contrôle d'erreurs (réémission, fragmentation, etc.). Pour construire un véritable groupe de discussion, il faut donc gérer tout un tas de choses (« à qui est destiné tel ou tel message ? » ; « qui est autorisé à m'envoyer un message ? »).

Les données traitées par l'application sont échangées sous forme de **TLV** (pour *Type, Length, Value*) dont les formats sont précisément définis.

Grâce à un type de TLV particulier renvoyé (répondu) à la réception d'un message, on s'assure que le même message n'est pas envoyé plusieurs fois au même voisin.

1.2 C pas sorcier (tl;dr: aïe c'est bas niveau)

Il était imposé d'écrire le programme en C. Chouette language. Pour de nombreuses raisons, il est fort adapté à l'apprentissage de la programmation réseau (on se trouve à « bas niveau »).

Du point de vue de son architecture, notre programme est simple, il colle au protocole. Ceci dit, en C, puisqu'il faut notamment gérer la mémoire, il est pratique de commencer par créer tout un tas de types de données, de routines, pour se simplifier la vie par la suite. Les concepts à manipuler ont beau être simples, la logique élémentaire, il nous a fallu écrire plusieurs milliers de lignes de code ; là où une implémentation en Python (par exemple) aurait requis tout au plus quelques centaines de lignes.

Pas de souci avec ça, hein, au contraire, ça nous fait des choses à détailler dans ce compte rendu.

Mais tout de même : coder en C prend du temps.

1.3 Décisions prises (tl;dr: interface avec GTK)

Nous avons décidé d'utiliser des threads, un pour la partie serveur, un pour la partie qui écoute les entrées utilisateur et un qui s'occupe d'envoyer de l'inondation.

miaouchat, c'est le nom que nous avons donné à notre logiciel.

Par contre, nous tenions à avoir une vraie interface graphique, bien propre, afin que **miaouchat** ne consiste pas seulement en une *proof of concept* mais soit vraiment agréable à utiliser pour des personnes qui n'ont pas l'habitude du terminal.

C'est GTK qui nous a paru être le meilleur choix, car GTK est une librairie...

- installée presque partout ;
- écrite en C, et elle s'utilise donc nativement dans ce langage ;
- éprouvée (elle existe depuis plus de vingt ans) ; on dispose de suffisamment d'exemples pour se dépatouiller et obtenir ce que l'on souhaite, contrairement à beaucoup d'autres solutions exotiques ;
- qui permet d'obtenir un résultat satisfaisant — en terme d'esthétique — relativement simplement.

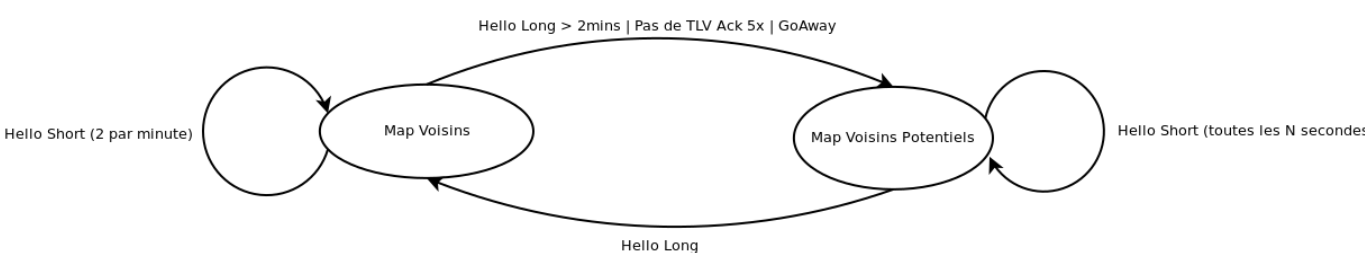
1.4 Beaucoup de fichiers (tl;dr: c'est bien rangé)

Fichier (.h/.c)	Contient	Notes
dllist	fonctions de manipulation de listes doublement chaînées de type `dllist_t`.	
interface		
main	la fonction main() : c'est le chef d'orchestre	
map	sources de la hashmap.	merci à l'auteur : rxi, sur Github
msg	méthodes de manipulation et transformations des TLV et messages.	
msg_list		
neighbour	implémentent les différentes listes de voisins et leur manipulations.	
neighbour_map	implémentent les différentes listes de voisins et leur manipulations.	
serialization		
shared_resources		
test	tests écrits durant le développement.	aucun intérêt.
types	types non standards, afin de les centraliser et d'éviter les conflits d'importation.	
worker		

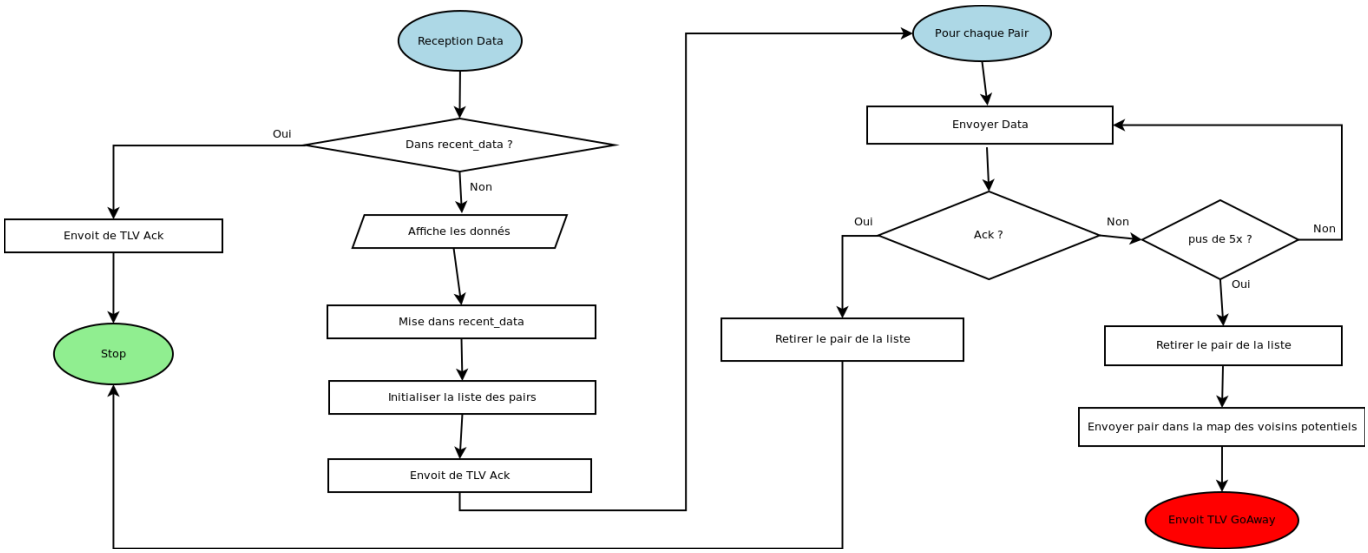
2 Logique de fonctionnement

Les diagrammes ci-dessous illustrent la logique de notre programme.

2.0.1 Relation entre voisins



2.1 Réception d'un message



2.2 Structures de stockage que l'on a

- une hashmap de messages reçus.
- une liste contenant les clés des messages ordonées par ordre d'arrivée.
- une hashmap de voisins courants.
- une hashmap de voisins potentiels.

Chaque voisins courant a une liste de messages qui lui reste à innonder.

2.3 Découverte de voisins

On peut passer en paramêtre à la messagerie une string contenant l'addresse et une autre le port d'un pair. Ensuite, à l'aide d'un getaddrinfo(), on récuperera les addrinfo* correspondant à toutes les paires possible à cette paire adresse/port qui seront stockées soit dans la hashmap de voisins courants si on arrive à y envoyer un TLV HELLO court, sinon dans la hasmap des voisins potentiels. Lorsqu'on reçoit un TLV HELLO court, on regarde si l'emetteur fait partie de nos hashmaps de voisins, sinon on récupère sont adresse et son port et on le passe à la fonction précédente.

2.4 Inondation

Lorsque l'on reçoit ou envoit un message, on place tout d'abort le TLV DATA dans la liste des

messages à émettre de chaque voisins en initialisant le champ `send_time` à la data à laquelle l'envoit est prévu. Dans le thread `worker`, on a une boucle qui à lieu toutes les 0.5 secondes et qui regarde la liste de messages à émettre chaque voisin. Si un message à une date d'emission (`send_time`) antérieure à la date actuelle, alors il est stocké dans un tableau de TLV DATA. lorsque la taille des TLV dans ledit tableau s'apprête à dépasser le PMTU du voisin considéré, on les envoit en un seul message. On réinitialise la date d'émission prévue par la même occasion.

~

Merci d’avoir lu.

Félix & Louis