RE203 - Projet de Réseau

Rapport Final

Maxime Bellier Jean-Michaël Celerier
Julien Chaumont Bazire Houssin Sylvain Vaglica

Groupe 3

15/05/2013

IPB
ENSEIRB
MATMECA
BORDEAUX

Introduction

Ce rapport final pour le projet de réseau RE203 fait suite au rapport intermédiaire délivré le 23 avril dernier. Il sera ici question de l'implémentation mise en œuvre par notre groupe, l'explication du sujet ayant été développée précédemment.

1 Le bloc commun

Étant donné que le développement des deux parties du projet (contrôleur et routeur) a été réalisé respectivement dans les langages C et C++, il a été décidé d'établir une sorte bibliothèque commune sur laquelle se baser. Cette partie comprend :

- une invite de commande
- un système de gestion des fichiers de configuration
- des fonctions de traitement des messages reçus et à envoyer
- une API pour la gestion des sockets
- quelques fonctions utilitaires

Chacun de ces blocs est indépendant des autres, ce qui assure une intégration facilitée dans le contrôleur comme dans le routeur. Une telle organisation permet d'introduire un certain degré d'abstraction tout en limitant la redondance de code entre les deux composantes du projet.

1.1 Le prompt

Fichiers : `common/prompt.c`, `common/prompt.h`

L'invite de commande, ou *prompt* en Anglais, permet la communication entre le contrôleur ou le routeur, et l'utilisateur. Il suffit pour cela de regarder constamment sur l'entrée standard et de traiter les commandes saisies par l'utilisateur. Afin de ne pas bloquer le programme sur l'écoute du fichier `stdin`, il faut pouvoir exécuter cette tâche en parallèle, d'où l'utilisation de la bibliothèque `pthread` ici.

Les fonctions définies dans les fichiers sus-mentionnés assurent la création de ce thread et l'écoute sur l'entrée standard. Le paramètre à fournir à la fonction d'initialisation est un pointeur de fonction. Lorsqu'une information arrive, elle est mise en mémoire et convertie en un `Message` [1], et c'est à la fonction dont le pointeur est passé en paramètre de gérer les actions à effectuer suite à la réception de ce message.

De cette manière, contrôleur et routeur ont seulement à gérer la fonction de traitement des messages utilisateur (chaque programme devant comprendre des commandes qui lui sont spécifiques), mais la tâche de récupération des instructions de l'utilisateur est quant à elle unifiée.

1.2 Traitement des fichiers de configuration

Fichiers : `common/config.c`, `common/config.h`

Le sujet demandait à ce que routeur comme contrôleur puissent être paramétrés depuis un fichier de configuration. Ce module permet la lecture de ces fichiers. La bibliothèque C `T-Rex` [2], qui permet la manipulation d'expressions régulières, a été utilisée dans ce cadre afin de faciliter la reconnaissance des différentes lignes possibles dans les fichiers de configuration.

Dans le code du routeur (respectivement du contrôleur), il suffit d'appeler la fonction `config__readRouter` (respectivement `config__readController`). En interne, ces fonctions ouvrent le fichier de configuration indiqué, le lisent ligne par ligne, identifient les mots-clés à l'aide des expressions régulières,

1. Se référer à la partie 1.3
2. Publiée sous licence libre *zlib* par Alberto Demichelis, consultable à l'adresse `http://sourceforge.net/projects/tiny-rex/`

et remplissent une variable de type `Configuration` avec les informations contenues dans le fichier. Cette structure de données est déclarée de la manière suivante :

```
enum SoftwareType { ROUTER, CONTROLLER };

typedef struct Configuration
{
        enum SoftwareType type;
        char controllerAddress[20];
        int  routerPort;
        int  controllerPort;
        int  pollTimeoutValue;
        int  controllerUpdateInterval;
        int  routerUpdateInterval;
        int  defaultTTLValue;
        int  defaultPingPacketCount;
        int  defaultPacketTimeoutValue;
        int  defaultDVTimeoutValue;
} Configuration;
```

A la fin de la lecture, il ne reste plus qu'à retourner au programme appelant un pointeur vers la structure remplie, qui n'aura alors qu'à consulter la valeur de ces différents attributs.

Notez que la structure `Configuration` contient des attributs concernant aussi bien le contrôleur que le routeur. C'est au programme de veiller à n'utiliser que les attributs le concernant, les autres attributs n'ayant aucune garantie d'avoir été initialisés à une valeur cohérente (précisons cependant qu'il n'y a aucune raison pour que le contrôleur ait besoin des informations propres aux routeurs, et inversement).

1.3 Le système des `Messages`

Fichiers : `common/messages.c`, `common/messages.h`

Les `Messages` correspondent à une structure de données relative aux informations reçues et transmises par le contrôleur, le routeur ou l'utilisateur. Ce module permet notamment :

- L'allocation et la libération dynamique d'un message
- L'échappement d'une chaîne de caractères (caractères "*" et "\"), à utiliser lors de l'envoi d'un `Message`, et l'opération inverse pour la réception
- Le parsing, c'est-à-dire la reconnaissance de la commande contenue dans le `Message` (là aussi, la bibliothèque `T-Rex` est mise à contribution)
- La gestion du TTL

La structure `Message` contient toutes les informations nécessaires au traitement ultérieur.

```
typedef struct Message
{
        enum MessageType type;   //Type du message (login, poll, ping, load, ...)
        char *s_parameter;       //Paramétre de la commande (identifiants, ...)
        int  n_parameter;        //Paramétre de la commande (numéro de port, ...)
        char *node1, *node2;     //Identifiants de l'émetteur et du récepteur
        int  seqnum;             //Numéro de séquence
        enum Acceptance accept; //Type de retour
} Message;
```

1.4 La gestion des sockets

1.4.1 La structure `Client`

Fichiers : `common/client.c`, `common/client.h`

La structure `Client` sert de base à la gestion des communications par socket. Elle permet l'association d'un routeur par son identifiant à une socket (qui contient à la fois l'adresse IP et le port à utiliser pour établir une communication avec ce client). Les fonctions fournies avec ce module permettent la récupération du port, de l'adresse et de l'identifiant d'un client, la mise à jour de l'identifiant ou encore la comparaison de deux `Client`s.

La fonction `getsockname` de la bibliothèque `sys/socket.h`, en addition des fonctions `inet_ntoa` et `ntohs`, permet de récupérer séparément, et sans difficulté majeure, l'adresse et le numéro de port depuis une socket.

1.4.2 Les sockets

Fichiers : `common/net.c`, `common/net.h`, `common/sock.h`

Il s'agit certainement du pilier du projet, puisque c'est ce module qui assure la communication entre les routeurs et/ou le contrôleur en réseau. L'idée était en fait de réaliser une couche d'abstraction au-dessus de la bibliothèque `sys/socket.h`.

La structure de données utilisée ici se nomme `network`, et représente le réseau.

```
struct network_s{
  short status; //Réseau ouvert ou fermé
  SOCKET server;//Socket du serveur
  int max;      //Plus grand identifiant de socket (utilisé par select)
  Client *clients; //Clients connectés
  unsigned int nb_clients; //Taille du tableau précédent

  /* Fonctions d'événement */
  //Fonction à appeler lorsque l'entrée standard a été utilisée
  input_event_function input_event;
  //Fonction à appeler lors de la connexion d'un client
  connection_event_function connection_event;
  //Fonction à appeler lors de la déconnexion d'un client
  disconnection_event_function disconnection_event;
  //Fonction à appeler lors de la réception d'un message
  message_event_function message_event;
};
```

Une fois les fonctions d'initialisation appelées, la fonction `network__update` permet l'écoute *non bloquante* (à l'aide de la fonction de multiplexage de la librairie standard `select`) sur le port indiqué. A chaque événement (*ie* connexion ou déconnexion d'un client, réception d'un message, ...), la fonction continue le traitement en analysant le type de l'événement qui l'aura réveillé et en appelant les fonctions correspondantes dont les pointeurs sont indiqués dans la structure `network` utilisée. Ce sera le rôle de ces fonctions de gérer les actions à mettre en place, ce qui dépend évidemment de s'il s'agit du contrôleur ou d'un routeur, et doivent donc être implémentées et initialisées directement depuis le programme utilisant le module.

2 Le contrôleur

En se basant sur le code du bloc commun, il est plus facile d'écrire un code propre et bien organisé. Cette seconde partie détaille l'implémentation du contrôleur.

2.1 Gestion des graphes

Fichiers : `controller/graphlib.c`, `controller/graphlib.h`

La première action à effectuer sur le contrôleur est le chargement de la topologie d'un réseau sous la forme d'un fichier au format *dot*. A cet effet, il a fallu mettre en place un système de gestion de graphes facilitant les interactions entre le format *dot* et la représentation en mémoire : la librairie de graphes de *graphviz*[3] était incontestablement la plus apte à répondre à ce problème. Dans un souci de propreté du code, une couche d'abstraction a été, ici aussi, mise en place au dessus de *graphviz*.

Dans la pratique, le programme agit sur une variable globale de type `Agraph_t`. Cette structure contient en fait l'arbre correspondant à la topologie du réseau chargé, où chaque nœud est un routeur, et où chaque arête indique le voisinage de deux routeurs et est labellisée du poids du chemin représenté. *graphviz* offre toutes les fonctions nécessaires au traitement d'une telle variable, de son initialisation à l'aide d'un fichier *.dot* à sa sauvegarde, et évidemment l'ajout/-suppression/modification de tout nœud ou arête.

2.2 Table des routeurs

Fichiers : `controller/info_table.c`, `controller/info_table.h`

Un structure de table de hachage est utilisée dans le code du contrôleur. Son rôle est de faire le lien entre l'identifiant d'un routeur (son nom) et la structure `Client` qui lui est associée et qui permet la communication. Pour cela, cette partie du code source se base sur la bibliothèque *Hash-Table*[4] en C, qui a le grand avantage d'être très simple d'utilisation, en plus d'avoir une documentation bien exhaustive. Après initialisation de la structure, celle-ci est mise à jour à chaque connexion et déconnexion d'un routeur sur le réseau. Dans le cas d'une connexion, rappelons que le bloc de gestion des sockets gère de lui-même l'allocation et l'initialisation d'une structure `Client`.

Travailler sur une telle structure de données permet d'optimiser la complexité temporelle de la recherche des informations de connexion d'un routeur donné. Comme pour le graphe du réseau, le contrôleur travaille sur une variable globale pour gérer la table de hachage.

2.3 Traitement des commandes

Fichiers : `controller/exec.c`, `controller/exec.h`

Comme évoqué précédemment, les commandes envoyées par l'utilisateur et par les routeurs sont respectivement réceptionnées par le *prompt* et par le module de gestion des sockets. Le code propre au contrôleur doit ensuite être capable de traiter ces commandes.

Pour cela, les deux fonctions `exec__prompt_message` et `exec__sock_message`, appelés par les modules sus-cités, ont pour rôle de traiter l'action demandée, à savoir :

- Dans le cadre d'un message venant du *prompt* :
  - Chargement de la topologie du réseau depuis un fichier vers une structure de graphe
  - Enregistrement de la structure de graphe courante dans un fichier

3. Publiée sous licence libre Eclipse, consultable à l'adresse `http://www.graphviz.org/`
4. Publiée sous licence libre GNU/GPL par Ankur Shrivastava, consultable à l'adresse `https://github.com/ankurs/Hash-Table/`

  - Affichage du graphe du réseau dans le terminal
  - Ajout/modification/suppression d'une connexion dans le graphe du réseau
  - Déconnexion d'un routeur
  - Demande de l'arrêt du contrôleur
- Dans le cadre d'un message venant d'une socket :
  - Connexion d'un routeur
  - Demande de *poll*
  - Déconnexion d'un routeur

Ces fonctions se chargent simplement, selon la nature de la demande, d'appeler les fonctions des autres modules (par exemple l'ajout d'une arête dans le graphe lors de la demande d'ajout d'une connexion entre deux routeurs, ou encore l'ajout des informations de connexion d'un routeur à la table de hachage lors de la connexion d'un routeur au réseau), en faisant dans certains cas quelques vérifications d'intégrité (est-ce que le routeur indiqué existe ? le pointeur passé en paramètre est-il valide ? ...).

Précisions sur la demande de *poll*

Le *poll*, qui consiste à renvoyer au routeur appelant la liste de ses voisins ainsi que le coût de leurs liaisons respectives, utilise directement le graphe du réseau. Pour construire cette liste, il faut parcourir la liste des voisins du routeur dans le graphe. La librairie *graphviz* facilite cette opération en proposant des fonctions de parcours des arêtes d'un sommet donné (initialisation avec `agfstedge`, puis parcours avec `agnxtedge`). Il ne reste plus qu'à récupérer le nom du sommet au bout de l'arête courante, le poids de ladite arête, et concaténer le tout selon la syntaxe demandée.

Notons tout de même que, même si la topologie chargée indique la propriété de voisinage sur deux routeurs donnés, il se peut que le second routeur ne se soit pas encore connecté au réseau. Dans ce cas, il ne faut pas l'ajouter au voisinage puisque celui-ci est, pour une durée indéterminée, injoignable. Cette difficulté est aisément contournée en ajoutant une information booléenne dans chaque nœud du graphe, indiquant si oui ou non le routeur associé est connecté (initialisé à 0 puis mis à 1 lors de la connexion, et enfin à 0 lors de la déconnexion).

Le sujet précise également que la liste retournée doit être nulle si le voisinage du routeur n'a pas été modifié depuis le dernier *poll*. Pour vérifier cette propriété, on utilise là aussi une information booléenne dans chaque nœud du graphe, qui précise si le voisinage du routeur associé a été ou non modifié depuis la dernière demande de mise à jour. Celle-ci est d'abord mise à 0 avant le premier *poll*, puis à 1 après chaque nouvelle demande. La mise à 0 est également effectuée sur tous les routeurs voisins d'un routeur venant de se connecter ou de se déconnecter, ainsi que sur tous les routeurs concernés par un ajout, suppression ou modification d'arête. De cette façon, il n'y a qu'à vérifier la valeur de ce booléen pour savoir si la procédure doit être faite en entier, ou s'il suffit de renvoyer une liste vide.

2.4 Résumé du fonctionnement du contrôleur

L'algorithme suivant résume brièvement le fonctionnement du contrôleur. L'intégralité du traitement des commandes, de la réception à l'exécution puis à la génération de la réponse, est exécuté dans les différents sous-modules.

```
initialization(graph, hash_table, network);
while is_opened(network) do
    /* these instructions are executed in a separated thread                */
    message = prompt__new_message();
    if message__is_not_empty(message) and message__is_valid(message) then
        exec__prompt_message(message);
    end
    /* and these ones are executed in the main thread                       */
    message = network__new_message();
    if message__is_not_empty(message) and message__is_valid(message) then
        exec__network_message(message);
    end
    /* in this way, both are executed in the same time without creating any
       deadlock                                                             */
end
close(graph, hash_table, network);
```

3 Les routeurs

De même que pour le contrôleur, une partie du code du routeur se base sur le code commun présenté précédemment. Néanmoins, le défi repose dans le fait que le routeur est écrit en C++, et que bien qu'il y ait un fort lien de parenté entre les deux langages, il est parfois nécessaire de connaître quelques astuces pour pouvoir interfacer les deux codes.

De plus, il a été fait en sorte que le code suive une architecture MVC (Modèle-Vue-Contrôleur), en séparant les classes qui servent à l'entrée/sortie et celles qui servent à la gestion du logiciel. La majorité des appels passe par une classe contrôleur.

3.1 La classe Router

Fichiers : `router/router.cpp`, `router/router.hpp`
Classe : `Router`

Cette classe est la classe principale du programme. Elle est instanciée dans le *main*, et instancie par la suite toutes les autres classes du logiciel ; c'est le contrôleur (au sens MVC) du logiciel.

La classe `Router` contient notamment la table de routage, ainsi que la plupart des classes qui gèrent les messages, et manipule les threads qui servent à communiquer périodiquement avec le contrôleur et les autres routeurs, par le biais des commandes *poll* et *vector*).

3.2 La table de routage : RouteTable

Fichiers : `router/routetable.cpp`, `router/routetable.hpp`, `router/entry.cpp`, `router/entry.hpp`
Classes : `Router`, `Entry`

La table de routage a été implémentée sur la base d'un simple tableau associatif (l'objet `map` du C++). Pour un routeur donné, on associe à chacun de ses voisins (désigné par son nom) une structure `Entry` qui contient les différentes informations nécessaires à la communication avec ce client.

`Entry` contient un pointeur vers un `Client`, présenté précédemment, la distance entre deux nœuds, ainsi que le prochain saut à effectuer pour rejoindre un nœud.

3.3 Gestion des messages

Fichiers : `router/event.cpp`, `router/event.hpp`, `router/exec.cpp`, `router/exec.hpp`
Classes : `Event`, `Exec`

Grâce à l'architecture choisie, la gestion de la communication est sensiblement semblable à celle utilisée dans le routeur.

Le prompt (classe`Prompt`), ainsi que la pseudo-classe `net` de la partie commune, communiquent avec la classe `Exec`, qui est chargée de la compréhension et de l'application des actions requises lors de la réception d'un message.

L'utilisation du même système d'événements qu'en C a posé des problèmes d'incompatibilité de prototypes : en effet, la classe `Event` contient les événements à exécuter en cas d'opération sur le réseau ; mais le prototype attendu par la fonction C est incompatible avec une méthode d'instance, qui contient un paramètre caché `this`.

Il a donc été nécessaire d'avoir une variable globale vers la classe `Router` pour accéder à toutes les informations nécessaires (table de routage en particulier) dans la classe `Event`. Ce n'est certes pas une bonne pratique de programmation, mais cela permet de n'avoir qu'un seul code pour le routeur et le contrôleur.

Pour information, les messages à gérer au prompt sont *ping*, *route* et *message*, mais d'autres ont été ajoutés, tels que *quit* pour quitter le logiciel, et *poll* pour forcer la mise à jour du voisinage.

`Exec` contient la logique des opérations à réaliser (par exemple, si le message est de type `neighborhood` et que son niveau d'acceptation est `OK`, alors on ne fait rien, sinon on fait le traitement nécessaire sur le message). Mais la plus grande partie du traitement se déroule dans des classes séparées : `PromptActions` et `SockActions`. `PromptActions` contient les actions à réaliser pour un message rentré à la main, et `SockActions` en fait de même pour un message reçu par le réseau.

A la différence de `Exec`, ces deux classes manipulent directement les fonctions contenues dans `net.c`. Ce sont elles qui forment les messages à envoyer par le réseau, en remplissant notamment les membres de la structure Message.

3.4 Classes annexes

Fichiers : `router/display.cpp`, `router/display.hpp`, `router/exceptions.hpp`
Classe : `Display`, `Exceptions diverses et variées`

`Display` sert à l'affichage des données du programme, des réponses aux messages, des messages d'erreurs, tel que spécifié dans le sujet. `Exceptions` est une liste de dérivés de `std::exception`, instanciés de manière à pouvoir s'en servir très simplement ; et se trouve être entièrement contenue dans un fichier header.

Par exemple, voici l'utilisation de l'exception pour l'absence d'un routeur dans la table de routage : `Unknown destination` (la casse est importante).

```
throw unknownDest; // envoi
catch(UnknownDest&) // récupération
```

La gestion des exceptions de cette manière ajoute à la clarté du code.

3.5 Diagramme de classes

Afin de récapituler plus simplement cette partie, le diagramme de classes du routeur est donné en figure 1.

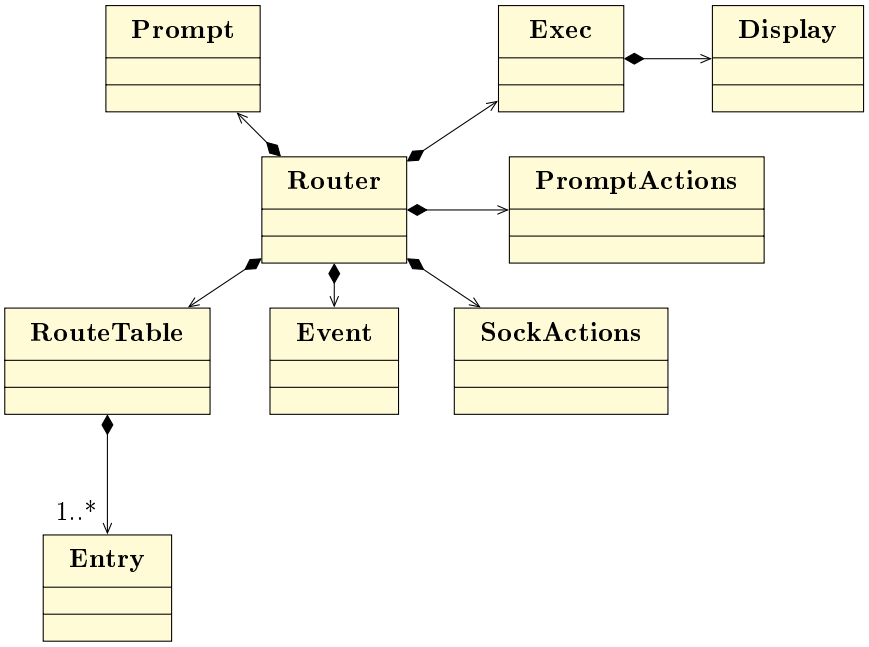

FIGURE 1 – Diagramme de classes du routeur

Conclusion

Les programmes proposés répondent aux différents prérequis du sujet, ce qui était à vrai dire l'objectif principal. Durant le développement, l'accent a notamment été mis sur le soin à apporter au code. En conséquence, le code source suit une architecture MVC, est documenté, et même optimisé en ce qui concerne la redondance de code (grâce au code commun aux deux sous-programmes). Cette organisation du code a aussi permis de mieux organiser l'équipe sur les différentes tâches à réaliser, et ce, bien plus finement qu'une simple répartition "routeur-contrôleur".

Par ailleurs, les tests effectués, sur un même ordinateur comme sur plusieurs en réseau, ont démontré la stabilité et la fonctionnalité du projet.