Networking – part I

Luis D. Pedrosa & Jean-Cédric Chappelier 2017

Cette semaine vous allez devoir mettre en place tous les composants de base de l'infrastructure de DHT (revoir si nécessaire le document général de présentation), y compris quelques outils à la ligne de commande (des programmes, donc) pour pouvoir interagir avec cette infrastructure.

Même si à ce stade l'infrastructure mise en place restera relativement simple (un seul serveur, $N{=}W{=}R{=}1$; revoir si nécessaire le fichier de description principal), le fait que nous préparions tous les composants en fait un travail conséquent. Il est important pour votre efficacité de bien vous organiser, module par module (typiquement : `system` –> `node` –> `client` –> `network` ; revoir le diagramme général d'organisation du projet) et de bien tester *systématiquement* vos implémentations.

Le but de cette semaine est donc d'atteindre une infrastructure uniquement *locale*, sur un seul serveur, sans redondance, en utilisant les tables de hachage implémentées la semaine passée. Le client interagira avec ce seul serveur au moyen de requêtes UDP vers `localhost` (IP 127.0.0.1) sur le port 1234.

Nous commençons ici par décrire les outils de plus haut niveau avant de décrire les modules d'implémentation plus bas niveau. Le travail de cette semaine étant conséquent, il est important de bien vous répartir ce travail ; par exemple, l'un s'occupant des outils et l'autre des modules plus bas niveau (mais ce n'est qu'*un* exemple d'organisation parmi d'autres ; un autre exemple pouvant être de faire `pps-launch-server` et `network` d'un coté et `node`, `client`, `pps-client-get` et `pps-client-put` de l'autre).

I. Description des outils de la ligne de commande

Les trois outils à la ligne de commande utilisés pour cela seront : `pps-launch-server`, `pps-client-put` et `pps-client-get`, détaillés ci-dessous. Il faudra modifier votre `Makefile` pour que par défaut il compile ces trois outils.

Ces trois commandes devront répondre aux spécifications suivantes :

- `pps-launch-server` lance un serveur sur `localhost` (IP 127.0.0.1) sur

le port 1234 (voir `config.h`) ; elle ne reçoit aucune entrée et ne produit aucune sortie ; utilisez le flot d'erreur (`stderr`) pour vos messages de debug si nécessaire (voir la « fonction » `debug_print` dans `error.h`).

- `pps-client-put` a pour but d'ajouter des paires clé-valeur au serveur ; elle attend en boucle (jusqu'à la fermeture de son flot d'entrée standard) des séquences de deux lignes : en premier, la clé (un seul caractère, sur une ligne), puis ensuite la valeur (un entier, sur une autre ligne) ; en cas de succès d'ajout, `pps-client-put` affiche le message « `OK\n` » sur la sortie standard ; elle affiche par contre le message « `FAIL\n` » en cas d'erreur ; **respectez scrupuleusement ces messages** (`\n` ne doit être affiché explicitement, mais symbolise un retour à la ligne) ;

  nous spécifions totalement les affichages à effectuer sur la sortie standard et ceux-ci ne doivent en aucun cas être modifiés ; par contre, vous êtes libre d'utiliser le flot d'erreur comme bon vous semble pour vos propres messages ; nous ignorerons simplement ce flot d'erreur lors de nos corrections ;

- `pps-client-get` a pour but de lire une valeur associée à une clé depuis le serveur ; elle attend en boucle (jusqu'à la fermeture de son flot d'entrée standard) une ligne contenant une clé à lire (un seul caractère) ; en cas de succès d'ajout, `pps-client-get` affiche la valeur reçue suivant le format « `"OK %d\n"` » ; elle affiche par contre le message « `FAIL\n` » en cas d'erreur (ce qui, à ce stade du projet ne devrait pas se produire – voir la note ci-dessous – mais doit être envisagé dans le cas général).

Voici un exemple possible d'interaction (`^D` signifie que l'utilisateur tape les touches contrôle (`Ctrl`) et `D` en même temps, ce qui ferme le flot d'entrée standard) :

```
prompt> make
prompt> ./pps-client-put ## échoue car aucun serveur n'existe
a
1
FAIL
^D
prompt> ./pps-launch-server & ## lance un serveur en tâche de fond
prompt> ./pps-client-put ## fonctionne cette fois
a
1
OK
b
42
OK
^D
prompt> ./pps-client-get
a
OK 1
```

```
x
OK 0
^D
```

Note : remarquez qu'à ce stade une clé dont on n'a pas encore donné de valeur (clé « inexistante », `x` dans l'exemple ci-dessus) retournera simplement la valeur 0. Nous n'avons en effet pour le moment aucun moyen de distinguer entre une clé n'ayant jamais été introduite et une clé stockant la valeur 0. Ce comportement sera modifié plus tard lorsque nous aurons le moyen de distinguer les deux.
[**fin de note**]

II. Modules bas niveaux

Nous avons modularisé l'infrastructure de ce projet de la façon suivante :

- `system.[ch]`: fournissant les outils « système » de plus bas niveau pour ouvrir un socket ou rechercher une adresse IP ;
- `node.[ch]`: fournissant, pour les clients, la représentation des nœuds de l'anneau des clé ; à ce stade, tout simplement un serveur ;
- `client.[ch]`: fournissant les fonctions d'initialisation et de terminaison d'un client ;
- `network.[ch]`: fournissant l'implémentation des fonctions de communication client-serveur des requêtes `put` (écriture clé-valeur) et `get` (lecture de valeur).

II.1. system

Le module « système » est pratiquement identique à celui de la semaine 3 ; nous avons juste adapté les codes d'erreur. Pour rappel, il fournit les outils :

- `int get_socket(time_t temps)` permettant d'obtenir un socket réseau pour la communication en précisant son temps d'attente (« timeout », en secondes) ;
- `error_code get_server_addr(const char* IP, uint16_t port, struct sockaddr_in* adresse)` permettant d'obtenir l'adresse, au sens « objet interne » (`struct sockaddr_in`), d'une adresse IP et d'un port donnés ;
- `error_code bind_server(int socket, const char *IP, uint16_t port)` permettant d'associer une communication réseau (représentation externe : adresse IP, port) à un socket réseau (représentation interne).

Note : voir si nécessaire les fichiers `.h` pour plus de détails.
[**fin de note**]

Comme en semaine 03. ces fonctions vous sont fournies en l'état ; vous n'avez pas à les modifier, simplement à les utiliser.

II.2. node

Ce module est très simple et sert uniquement à bien modulariser le code, offrir les fonctions nécessaires à la représentation des nœuds dans les clients (pour le moment, et jusqu'à la semaine 11, chaque nœud est simplement un serveur différent) . Pour l'instant, simplement les fonctions d'initialisation et de fin d'utilisation d'un serveur par un client.

Pour la fermeture (`node_end()`), il n'y a strictement rien à faire ici. Vous y mettrez à l'avenir, si nécessaire, ce que vous estimerez devoir faire lorsqu'un client n'utilise plus un nœud.

Pour l'initialisation, il suffit à ce stade du projet d'appeler `get_server_addr()` de la couche « system ».

Vous devrez, bien sûr, au préalable définir le type `node_t` dans `node.h` pour y mettre ce dont vous avez besoin.

II.3. client

Le module « client » n'est pas non plus très développé à ce stade du projet. Il ne contient lui aussi que deux fonctions : l'initialisation et la fin d'utilisation d'un client.

Vous devez au préalable définir le type `client_t` dans `client.h`, lequel regroupe pour le moment un nom (de type `const char*`), un `node_ t` (serveur avec lequel le client va communiquer) et le socket (`int`) via lequel communiquer.

Pour des raisons d'uniformité avec la suite du projet, nous avons de plus introduit le type `client_init_args_t` permettant d'initialiser un client à partir de l'environnement. Pour le moment, il contient simplement un pointeur sur un client et un nom (de type `const char*`).

Concernant `client.c` :

- pour la fin d'un client (`client_end()`), il n'y a rien de plus à ce stade que d'appeler `node_end()` ;
- pour l'initialisation, le nom du client sera simplement celui reçu par l'argument de type `client_init_args_t` et il faudra initialiser le serveur correspondant ainsi qu'obtenir un socket avec un délai (« *timeout* ») d'une seconde.

Il faudra bien sûr gérer correctement tous les cas d'erreur. Il pourra être utile à ce sujet d'utiliser les « fonctions » fournies dans `erreur.h` (par exemple `M_EXIT_IF`, `M_EXIT_IF_ERR`, etc.).

II.4. network

La couche « network » est la couche-outil la plus haute, modélisant la communication client-serveur. Elle offre deux fonctions pour la gestion de ces communications : `network_put` pour l'échange de requêtes lors de l'écriture clé-valeur et `network_get` pour la lecture de valeur.

Chacune de ces deux fonctions a un comportement similaire : envoi de la requête du client au serveur, puis réception (et gestion) de la réponse du serveur.

Notez bien qu'avec le protocole utilisé (UDP), il n'est pas garanti que les message soient délivrés. Si la réponse du serveur n'est pas arrivée dans le temps imparti (qui est réglé grâce à la fonction `get_socket()` du module « system »), considérer la requête comme ayant échoué (renvoyer `ERR_NETWORK`).

Nous vous conseillons, pour écrire ces fonctions, d'adopter (comme toujours !) une approche modulaire (décomposez, DÉcomposez, DÉCOMPOSEZ !).

Pour envoyer des requêtes au serveur, utilisez la fonction `sendto()` (`man sendto`). Pour lire les requêtes reçues en retour du part serveur, utiliser la fonction `recvfrom()` (`man recvfrom`).

III. Implémentation des outils de la ligne de commande

III.1. `pps-launch-server`

La réalisation de la commande `pps-launch-server` ne nécessite que le module « system ».

Utiliser `bind()` (`man 2 bind`) pour faire le lien entre le socket obtenu par `get_socket()` et l'objet adresse obtenu par `get_server_addr()` (`struct sockaddr` ; **attention** ici à la différence de type ! On peut sans autre convertir un *pointeur* vers `struct sockaddr_in` en un *pointeur* vers `struct sockaddr`, mais pas dans l'autre sens).

Ensuite, un serveur est en fait une boucle infinie qui lit puis exécute les requêtes des clients qui le contactent, puis leur répond. Pour lire les requêtes adressées au serveur, utiliser la fonction `recvfrom()` (`man recvfrom`). Pour répondre, utiliser la fonction `sendto()` (`man sendto`).

Clients et serveurs utilisent des messages simples pour communiquer :

- les requêtes d'écriture (envoyées par `pps-client-put`) sont des messages de 5 octets (5 `char`s) résultants simplement de la concaténation de la clé (1 `char`, donc 1 octet) et de sa valeur, sur 32 bits au format réseau (utiliser `htonl()` ; revoir si nécessaire le travail effectué en semaine 3 et/ou la manpage correspondante) ; le serveur répond positivement en envoyant simplement un datagramme vide ;
- les requêtes de lecture (envoyées par `pps-client-get`) sont des messages de 1 octet, la clé ; le serveur répond positivement en envoyant un datagram de 4 octets, la valeur correspondante sur 32 bits au format réseau.

III.2. `pps-client-put`

La réalisation de la commande `pps-client-put` nécessite la couche « network » et donc aussi les modules « client », « node » et « system ».

Après initialisation (voir `client.h`), un client est simplement un programme qui attend des « commandes » sur son flot d'entrée standard. Les « commandes » attendues par `pps-client-put` sont simplement des paires clé–valeur (voir l'exemple d'interaction client-serveur donné plus haut, section I). Il ignorera tout autre entrée (utilisez très simplement `scanf` à ce stade).

Pour chaque paire correctement lue, utilisez simplement `network_put` (voir `network.h`) pour communiquer avec le serveur et afficher sur le terminal la réponse correspondante.

III.3. `pps-client-get`

`pps-client-get` fonctionne de façon très similaire à `pps-client-put` à la différence que ses « commande » se sont constituées que de la clé.

IV. Tests et debuging

Tester et debuger les communications réseau n'est pas facile. Nous vous avons pour cela écrit un petit outil `log-packets.c` que vous trouverez dans le répertoire `provided/` de votre dépôt Git. Cet outil permet de substituer notre propre version de `sendto()` à celle du système (fonction utilisée pour envoyer des messages sur des sockets). Nous y avons ajouté l'écriture systématique, dans un fichier `packets.log`, de tous les messages envoyés. Voici comment l'utiliser.

Avant tout, il faut le compiler, sous forme de bibliothèque dynamique (`.so`). Utilisez pour cela le `Makefile` fourni :

```
make -f Makefile.log-packets log-packets.so
```

Pour remplacer le `sendto()` système par le nôtre, il suffit de faire précéder la commande utilisée par « `LD_PRELOAD=./log-packets.so` ». Par exemple, pour logger tous les messages du serveur, lancez-le comme ceci :

```
LD_PRELOAD=./log-packets.so ./pps-launch-server
```

Faites, bien sûr, de même avec les clients si vous voulez voir les messages qu'ils envoient.

Si l'on reprend par exemple l'exemple du début, vous pourriez faire la chose suivante :

- ouvrez DEUX terminaux côte à côte ;
  on note `TERM1>` les commandes que vous tapez dans le premier, et `TERM2>` celles dans le deuxième ;
- entrez ensuite :

```
TERM1> LD_PRELOAD=./log-packets.so ./pps-launch-server &

TERM2> tail -F packets.log

TERM1> LD_PRELOAD=./log-packets.so ./pps-client-put
a 1
OK
b 42
OK
^D
TERM1> LD_PRELOAD=./log-packets.so ./pps-client-get
a
OK 1
x
OK 0
^D
```

(vous pourriez même avoir trois terminaux et lancer chacun des clients dans un terminal séparé, sans avoir besoin de terminer `put` avant de lancer `get`) ;

- observez en parallèle le contenu du fichier `packets.log` évoluer dans le deuxième terminal ;
  il devrait au final contenir quelque chose comme (les numéros de ports peuvent être différents) :

```
127.0.0.1 1234 127.0.0.1 56104
0.0.0.0 45258 127.0.0.1 1234 61 00 00 00 01
127.0.0.1 1234 127.0.0.1 45258
0.0.0.0 45258 127.0.0.1 1234 62 00 00 00 2A
127.0.0.1 1234 127.0.0.1 45258
0.0.0.0 46641 127.0.0.1 1234 61
127.0.0.1 1234 127.0.0.1 46641 00 00 00 01
```

```
0.0.0.0 46641 127.0.0.1 1234 78
127.0.0.1 1234 127.0.0.1 46641 00 00 00 00
```

- terminez en tapant :
- `kill %1` dans le premier terminal (termine le serveur) ;
- et `Ctrl-C` dans le deuxième (termine la lecture continue du fichier `packets.log`).

Nous vous conseillons par ailleurs de compiler en mode « debug » (ajoutez `-DDEBUG` à vos commandes de compilation) et d'utiliser la « fonction » `debug_print()` fournie dans `error.h` pour afficher ce qui se passe dans vos programmes.

V. Conseil et rendu

ATTENTION : il y a *beaucoup* de sources d'erreurs possibles dans toutes ces fonctions. Il faut bien systématiquement vérifier tous les arguments, et aussi les valeurs de retour des fonction outils utilisées dans une fonction de plus haut niveau. Il faut imaginer que chaque fonction peut être testée séparément avec les entrées « *bizarres* » (ce que nous ne nous priverons pas de faire). Si vous avez des question, n'hésiter pas à profiter des séances d'exercices pour demander.

Concernant le rendu, comme la semaine passée le travail de cette semaine ne sera pas évalué en tant de tel (c.-à-d. seul), mais devra faire partie du premier rendu intermédiaire du projet (délai : le dimanche 22 avril 23:59). Néanmoins, nous vous conseillons fortement de travailler régulièrement et faire systématiquement des tags hebdomadaires (si votre travail correspondant est opérationnel) afin de profiter de nos comptes-rendus de tests. Le tag correspondant à cette semaine est `week05`.

Pour cette semaine, il faudrait alors ajouter les sept fichiers suivants au répertoire `done/` de votre dépôt GitHub (de groupe) : `node.c`, `client.c`, `network.c`, `pps-launch-server.c` , `pps-client-put.c` et `pps-client-get.c` ; puis mettre à jour les fichiers `Makefile`, `client.h` et `node.h` ; et enfin « tagger » avec le tag `week05` puis « pousser » le résultat vers GitHub : `git push --tags`.