Kévin
Vythelingum

Serveur HTTP haute disponibilité

Introduction

Nous nous intéressons au cas d'un serveur gérant un pool de sous-serveurs HTTP ayant pour but de servir sous forme HTML des fichiers "texte" contenus dans un répertoire pré-établi. Plus précisément, à partir d'un squelette de serveur en langage C, nous ferons l'établissement des connexions et la gestion du cycle de vie des sous-serveurs.

L'objectif du TP est de :

- décrire le cycle de vie d'une socket TCP
- forger des réponses HTTP
- interpréter des requettes HTTP

Déroulement

La base

Il s'agit tout d'abord de compléter le fichier *reseau.c* au niveau des trois étapes fondamentales de mise en place d'une socket, à savoir :

- la création de la socket
- l'association d'un nom à la socket
- l'ouverture du service

TODO 1 : création de la socket

Pour l'étape de création, on utilise la fonction `socket` dont on récupère le retour dans `fd` :

```
int socket(int domain, int type, int protocol);
```

Cette primitive renvoie soit un descripteur de la socket créée, soit –1 en cas d'erreur. C'est pourquoi nous testerons la valeur de `fd` afin de lancer la commande `FATAL` en cas d'erreur.
De plus, elle prend en argument le domaine, le type et le protocole utilisés. Le choix s'est porté sur `AF_INET` pour le domaine puisque nous souhaitons communiquer par Internet en utilisant des adresses IPv4. Aussi, le type est `SOCK_STREAM`, qui permet une connexion fiable et bidirectionnelle, nécessaire à la réception de requêtes et l'envoi de réponses. Enfin, en mettant 0 pour le protocole, on obtient un protocole par défaut correspondant au type spécifié.

Cela donne pour le TODO 1 :

```
fd = socket(AF_INET, SOCK_STREAM, 0);
if(fd < 0)
{
  FATAL("socket");
}
```

TODO 2 : associer un nom à la socket

Pour associer un nom à la socket, on utilise la primitive `bind` :

```
int bind(int socket, const struct sockaddr *address, socklen_t address_len);
```

Elle prend comme paramètres le descripteur de la socket concernée, ici `fd`, une structure propre aux sockets comportant notamment le numéro du port utilisé, ainsi que la taille de cette structure (sizeof). Une structure `addr_serveur` de type `struct sockaddr_in` a déjà été définie, il s'agit donc la faire correspondre au type `struct addr` à l'aide d'un cast. On indique ensuite `lg_addr_serveur` qui vaut `sizeof(addr_serveur)`, soit la taille de la structure passée en second paramètre.

Comme précédemment, on teste sa valeur de retour qui est négative en cas d'erreur.

Cela donne pour le TODO 2:

```
test = bind(fd, (struct sockaddr *) &addr_serveur, lg_addr_serveur);
if(test < 0)
{
  FATAL("bind");
}
```

TODO 3 : ouverture du service

On ouvre enfin le service pour se placer en attente de potentiels clients. Pour cela, on utilise la primitive `listen` :

```
int listen(int socket, int backlog);
```

Elle prend comme paramètres le descripteur de la socket concernée, ici `fd`, et le nombre de clients simultannés autorisés, appelé *backlog*.

Avec un backlog de 4 comme demandé, cela donne pour le TODO3 :

```
listen(fd, 4);
```

Test du serveur

Pour tester le serveur, il a d'abord fallu compiler les fichiers sources grâce au Makefile fourni, puis le lancer grâce à la commande suivante en étant placé au niveau de l'exécutable dans l'arborescence des fichiers :

```
./serveur-web -p 1050 -d "${PWD}/rep/"
```

Ce qui suit *-p* est le numéro du port utilisé, ici 1050, et ce qui suit *-d* est l'emplacement de la racine des fichiers qui pourront être servis. Un '/' a été ajouté à la fin de la commande pour permettre des requêtes GET manuelles sans mettre de '/' devant l'emplacement voulu.

Ensuite, la visite de l'adresse `http://localhost:1050/rep/index.txt` avec un navigateur web a permis de valider le fonctionnement du serveur. En effet, le contenu du fichier *index.txt* présent dans le répertoire *rep/* était affiché sur une page HTML.

Compléments

Un client HTTP en C

Maintenant que nous avons testé le bon fonctionnement de notre serveur à l'aide d'un navigateur web, il s'agit de réaliser notre propre client HTTP écrit en langage C. Pour cela, nous réalisons un programme *client-http* de fichier source *client-http.c* que nous compilerons à l'aide de la commande suivante :

```
gcc -W -Wall -o client-http client-http.c
```

Cette commande est ajoutée dans le fichier `doIt.bash` pour faciliter la compilation de l'ensemble.

Le principe de fonctionnement du client est de mettre en place une socket connectée au serveur tout d'abord, puis d'envoyer la requête `GET index.txt HTTP/1.1` afin de demander le contenu du fichier *index.txt* au serveur. Enfin, nous affichons le résultat de la requête à l'aide d'une boucle avant de fermer la socket.

Mettre en place la socket du client

Pour mettre en place la socket du client, les étapes importantes sont :

- la récupération de l'hôte
- la création de la socket
- le nommage de la socket
- la connexion au serveur

La récupération de l'hôte se fait à l'aide de la primitive `gethostbyname(const char *name);` qui prend en paramètre l'adresse de l'hôte sous forme de chaîne de caractères et renvoi un `struct hostent *` qui servira au nommage de la socket. Nous stockons le retour de cette fonction dans le pointeur `host`, c'est pourquoi nous vérifions la bonne allocation de mémoire ensuite, en envoyant une commande `FATAL` en cas d'erreur.

Cela donne :

```
host = gethostbyname("localhost");
if(host==NULL)
{
  FATAL("gethostbyname");
}
```

La création de la socket se fait exactement comme pour le serveur et pour les mêmes raisons :

```
fd = socket(AF_INET, SOCK_STREAM, 0);
if(fd < 0)
{
  FATAL("socket");
}
```

Le nommage de la socket se fait à travers le champ `sin_addr.s_addr` de la structure `addr` de type `sockaddr_in`. Nous l'affectons à la valeur `((struct in_addr *) (host->h_addr))->s_addr`, soit l'équivalent de ce champ pour la structure `host`. Ainsi, la socket du client est liée au serveur. Nous précisons 1050 comme numéro de port, il faudra donc lancer le serveur à ce port.

Cela donne :

```
#define NUMPORT 1050
...
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = ((struct in_addr *) (host->h_addr))->s_addr;
addr.sin_port = htons(NUMPORT);
```

La connexion au serveur s'effectue grâce à la primitive suivante qui initie la connexion d'une socket :

```
connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Elle prend comme paramètres le descripteur de notre socket, la structure `addr` qui contient les informations sur le serveur, et la taille de cette structure. Elle renvoi un entier différent de 0 en cas d'erreur.

On obtient donc :

```
test = connect(fd, (struct sockaddr*)&addr, lg_addr);
if(test != 0)
{
  FATAL("Connect");
}
```

Envoyer une requête GET

Une fois la connexion établie avec le serveur, il s'agit de former la requête GET et l'envoyer au serveur.

Pour former la requête, nous avons besoin d'une chaîne de caractères que nous appelons `buffer`, dans laquelle nous écrivons notre requête :

```
GET index.txt HTTP/1.1
```

Cette requête comporte trois parties : le verbe HTTP utilisé (GET), la ressource demandée (index.txt) et le protocole utilisé (HTTP/1.1).

Ensuite, nous envoyons notre requête au serveur à l'aide de la primitive suivante :

```
send(int sockfd, const void *buf, size_t len, int flags);
```

Elle prend comme paramètres le descripteur de notre socket (fd), la chaîne de caractères contenant l'expression de notre requête (buffer), la taille de cette chaîne (strlen(buffer)) ainsi qu'un flag que nous mettons à 0 car nous n'en n'avons pas besoin.
Il peut être intéressant de noter que dans ce cas, la primitive `send` est équivalente à la primitive `write`.

Cela donne :

```
sprintf(buffer, "GET index.txt HTTP/1.1");
send(fd, buffer, strlen(buffer), 0);
```

Récupérer et afficher le résultat

Il faut ensuite afficher le résultat du serveur. En effet, le verbe GET a pour objet de demander une ressource au serveur, il nous renvoie donc cette ressource ! Pour cela, nous utilisons la primitive suivante :

```
recv(int sockfd, void *buf, size_t len, int flags);
```

Elle prend les mêmes paramètres que la primitive `send`. En effet, nous nous servons désormais de la chaîne *buffer* pour stocker le résultat envoyé par le serveur. Cette primitive retourne la taille du retour du serveur, ce qui nous permet de boucler l'affichage tant qu'on obtient quelque chose de ce dernier.

On a ainsi :

```
recu = recv(fd, buffer, sizeof(buffer), 0);
while (recu > 0)
{
  printf("%s", buffer);
  recu = recv(fd, buffer, sizeof(buffer), 0);
}
```

Fermer la socket

La dernière étape de l'implémentation du client consiste à fermer la requête à l'aide de la primitive `close(int fd);` qui prend en paramètre le descripteur de notre socket. L'utilisation de cette primitive nécessite la bibliothèque `<unistd.h>`.

Compilation et test du client

Pour compiler le client, il suffit de lancer le script bash *dolt.bash* qui s'occupé également de lancer le serveur sur le port 1050. Il ne reste plus qu'à lancer l'exécutable *client-http* créé. On obtient le résultat voulu, c'est-à-dire le contenu du fichier *index.txt* mis en forme par le serveur dans un document HTML :

```
HTTP/1.1 200 OK
Server: 0.0.7 pour Unix
Content-Type: text/html; charset=iso-8859-1

<html><head><title>Fichier index.txt</title></head>
<body bgcolor="white"><h1>Fichier /users/elo/kvytheli/enssat/RESEAU-serveurHttp/r
ep/index.txt</h1>
<center><table><tr><td bgcolor="yellow"><listing>
hello
</listing></table></center></body></html>
```

Ceci permet de valider le bon fonctionnement de notre client HTTP pour les requêtes de type GET.

Conclusion

En somme, aucune difficulté n'a été rencontrée pour mettre en place le serveur de base. Cependant, la réalisation du client a fait apparaître des problèmes lors de l'implémentation du verbe GET. En effet, il fallait passer comme requête `GET index.txt HTTP/1.1` et non `GET http://localhost:1050/index.txt`.

En ce qui concerne le degré d'avancement, j'ai complété la mise en place du serveur et j'ai implémenté un client capable d'envoyer une requête de type GET. Je n'ai pas implémenté la reconnaissance de PUT, d'une part par manque de temps et d'autre part par manque de compréhension de comment effectuer une requête pourvue d'un corps.