Episode 6 - Applications réseaux avancées

Programmation système unix

V3

BOOTSTRAP

Applications réseaux avancée

Lors du dernier épisode, nous avons vu comment créer des applications clients / serveurs simples.

Côté serveur, il y a du boulot :

- Création d'une socket : *socket()*
- Bind sur la socket : *bind()*
- Mise en écoute sur la socket : *listen()*
- Acceptation de la connexion entrante : *accept()*
- Recevoir des données envoyée par le client dans la socket : *recv()*

On aura peut être remarqué lors de nos tests qu'une grande partie des fonctions qu'on utilise avec les sockets, comme *accept()* et *recv()* sont bloquantes. C'est à dire qu'elles suspendent notre processus tant qu'elles ne finissent pas leurs exécutions. Si on lance notre micro-serveur tout seul sans jamais s'y connecter avec un client, il restera indéfiniment bloqué au niveau de son accept() puisque cet appel système attend une demande de connexion qui ne viendra jamais.

Ceci n'est pas une mauvaise chose en soi, mais en pratique, un serveur - le serveur web Apache qui gère https://www.esiee-it.fr par exemple - accepte et répond à des dizaines de clients par seconde. Si on attend de recevoir un message d'un client, cela ne devrait pas nous empêcher d'accepter une nouvelle connexion ou un autre message depuis un autre client.

Ceci est un aspect de la programmation concurrente, où **un programme doit pouvoir gérer plusieurs choses simultanément**.

Heureusement, il y a des méthodes, dites de « **multiplexage** », qui nous permettent de rendre nos sockets non-bloquantes, et pour détecter quand elles sont prêtes à être utilisées.

PREAMBULE

Mettez la musique *Dead Can Dance - Opium* dans vos oreilles. Laissez démarrer.
Posez-vous la question suivante : Si vous regardiez votre vie de l'extérieur, quel conseil vous donneriez-vous ?

Mettez la musique *Lupe Fiasco - Daydreamin' (feat. Jill Scott)* dans vos oreilles.
Posez-vous à nouveau la question suivante : Si vous regardiez votre vie de l'extérieur, quel conseil vous donneriez-vous ?

Mettez la musique *Naâman, Marcus Gad - Soul Plan* dans vos oreilles.
Posez-vous à nouveau la question suivante : Si vous regardiez votre vie de l'extérieur, quel conseil vous donneriez-vous ?

La ressource est en vous. On peut juste changer l'environnement.

Regardez la série Mr Robot, si pas déjà vu. (Pas maintenant, plutôt ce soir.)

Si vous n'avez pas de liste de vos prochaines séries à regarder, il est tant d'en créer une

sur votre téléphone avec n’importe quelle appli de note. Si vous ne savez pas quelle appli prendre : Google Keep.

EXERCICES

DEMO 1 : Rendre les sockets non-bloquantes avec fcntl()

Lorsqu'on invoque l'appel système `socket()` pour récupérer un descripteur de fichier pour notre socket, le noyau du système d'exploitation la crée automatiquement en mode bloquant. Nous pouvons, si on le souhaite, la rendre non-bloquante à l'aide de la fonction de manipulation de fichier **`fcntl()`** de **`<unistd.h>`** et **`<fcntl.h>`**, de cette façon :

```
socket_fd = socket(PF_INET, SOCK_STREAM, 0);
fcntl(socket_fd, F_SETFL, O_NONBLOCK);
```

Quand on rend la socket non-bloquante avec `O_NONBLOCK`, cela empêche les appels systèmes tels que `recv()` de suspendre notre programme le temps de leur exécution. S'il n'y a rien à lire dans une socket, `recv()` renvoie alors immédiatement -1 et indique `EAGAIN` ou `EWOULDBLOCK` dans `errno`. On peut de cette manière boucler sur nos sockets une à une afin de voir si elles ont quelque chose à lire, et sinon on continue. La même chose est possible pour toute fonction bloquante, comme par exemple, `accept()`.

EXERCICE 1 :

Créer un programme serveur **my_tcp_server_nonblock** en C qui reçoit et affiche les messages envoyés par le client **netcat**.

Terminal 1

```
~/> ./my_tcp_server_nonblock 4242
Starting on port 4242.
Binding OK
Waiting message from client
Nothing to recv in socket, sleeping for 2 seconds...
Nothing to recv in socket, sleeping for 2 seconds...
Nothing to recv in socket, sleeping for 2 seconds...
```

```
Nothing to recv in socket, sleeping for 2 seconds...
Received => Hello guacamole
Nothing to recv in socket, sleeping for 2 seconds...
Nothing to recv in socket, sleeping for 2 seconds...
...
```

Terminal 2

```
~/> ./nc localhost 4242
Hello guacamole
```

Tips :

Dans une boucle, recv en non-bloquant, avec un sleep(2).

DEMO 2 : Surveiller les sockets avec select()

Tout ça c'est bien, mais boucler sur toutes nos sockets clients de cette manière est une opération très intensive pour notre pauvre CPU, particulièrement s'il y en a des centaines ! Il y a de bien meilleurs moyens de gérer ce problème de blocage, et nous allons les découvrir maintenant.

Ce qui serait pratique, ce serait un moyen de surveiller l'ensemble de nos descripteurs de fichiers de sockets pour être avertis lorsque l'un d'entre eux est prêt pour une opération. Après tout, si on sait que la socket est prête, on peut y lire ou y écrire sans crainte de bloquer.

C'est exactement ce que fait la fonction **`select()`**. Pour l'utiliser, il nous faudra importer **`<sys/select.h>`**, **`<sys/time.h>`**, **`<sys/types.h>`**, et **`<unistd.h>`**. Son prototype est le suivant :

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

Ses paramètres semblent un peu compliqués, alors regardons-les de plus près :

- **nfds** : un entier qui indique la valeur du plus grand descripteur de fichier à surveiller, plus un.

- **readfds** : un ensemble de descripteurs de fichiers à surveiller pour la lecture, pour vérifier qu'un appel à `read()` ou `recv()` ne bloquera pas. Peut être `NULL`.
- **writefds** : un ensemble de descripteurs de fichiers à surveiller pour l'écriture, pour vérifier qu'un appel à `write()` ou `send()` ne bloquera pas. Peut être `NULL`.
- **exceptfds** : un ensemble de descripteurs de fichiers à surveiller pour l'occurrence de condition d'exception. Peut être `NULL`.
- **timeout** : un délai après lequel on force `select()` à terminer son exécution si aucun des descripteurs de fichiers ne changent d'état.

En cas de réussite, `select()` modifie chacun des sets pour indiquer quels descripteurs de fichiers sont prêts pour une opération. Elle renvoie aussi le nombre total de descripteurs de fichier qui sont prêts parmi les trois ensembles. Si aucun des descripteurs ne sont prêts avant la fin du timeout indiqué, `select()` peut renvoyer 0.

En cas d'erreur, `select()` renvoie -1 et indique l'erreur dans `errno`. Elle ne modifie dans ce cas aucun des ensembles de descripteurs de fichier.

Manipuler les ensembles de descripteurs pour select()

Pour manipuler les ensembles de descripteurs de fichiers que l'on veut surveiller avec `select()`, on va vouloir faire appel aux macros suivantes :

```
void FD_CLR(int fd, fd_set *set);   // Retire un fd de l'ensemble
int  FD_ISSET(int fd, fd_set *set); // Vérifie si un fd fait partie de
l'ensemble
void FD_SET(int fd, fd_set *set);   // Ajoute un fd à l'ensemble
void FD_ZERO(fd_set *set);          // Met l'ensemble à 0
```

Le timeout de select()

Le paramètre *timeout* représente la limite de temps maximal passé dans la fonction `select()`. Si aucun des descripteurs de fichier dans les ensembles qu'elle surveille ne deviennent prêts à effectuer une opération passé ce délai, `select()` retournera. On pourra alors faire autre chose, comme par exemple imprimer un message pour indiquer qu'on attend toujours.

La structure à utiliser pour la valeur temporelle, `timeval`, se trouve dans `<sys/time.h>`

:

```
struct timeval {
    long    tv_sec;    // secondes
    long    tv_usec;   // microsecondes
};
```

Si cette valeur de temps est à 0, `select()` retournera immédiatement ; si on y met `NULL`, `select()` pourra bloquer indéfiniment si aucun des descripteurs de fichier ne changent d'état.

Sur certains systèmes Linux, `select()` modifie la valeur de *timeval* quand elle termine son exécution pour refléter le temps restant. Ceci est loin d'être universel, donc par souci de portabilité, il ne faudrait pas compter sur cet aspect.

Exemple de surveillance de sockets avec select()

Tentons donc de créer un petit serveur qui surveille avec `select()` chaque descripteur de fichier de socket connectée pour la lecture. Lorsque l'une d'entre elles est prête à lire, on vérifiera d'abord s'il s'agit de la socket principale de notre serveur, auquel cas il faudra accepter une nouvelle connexion client. En effet, il est très utile de savoir que la socket qui est en train de `listen()` sera marquée prête pour la lecture si elle a une connexion à accepter ! Si, au contraire, le descripteur de fichier prêt à être lu correspond à une socket client, on y lira le message reçu.

```
#include <errno.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/select.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>

#define PORT 4242  // le port de notre serveur

int create_server_socket(void);
void accept_new_connection(int listener_socket, fd_set *all_sockets, int *fd_max);
```

```c
void read_data_from_socket(int socket, fd_set *all_sockets, int fd_max, int
server_socket);

int main(void)
{
      int server_socket;
      int status;
      int i;            // Pour notre bloucle de vérification des sockets
      fd_set all_sockets; // Ensemble de toutes les sockets du serveur
      fd_set read_fds;  // Ensemble temporaire pour select()
      int fd_max;       // Descripteur de la plus grande socket
      struct timeval timer;

      // Création de la socket du serveur
      server_socket = create_server_socket();
      if (server_socket == -1)
      exit(-1);

      // Écoute du port via la socket
      printf("[Server] Listening on port %d\n", PORT);
      status = listen(server_socket, 10);
      if (status != 0)
      {
      printf("[Server] Listen error: %s\n", strerror(errno));
      exit(-1);
      }

      // Préparation des ensembles de sockets pour select()
      FD_ZERO(&all_sockets);
      FD_ZERO(&read_fds);
      FD_SET(server_socket, &all_sockets); // Ajout de la socket principale à
l'ensemble
      fd_max = server_socket; // Le descripteur le plus grand est forcément celui de
notre seule socket
      printf("[Server] Set up select fd sets\n");

      while (1) // Boucle principale
      {
          // Copie l'ensemble des sockets puisque select() modifie l'ensemble surveillé
          read_fds = all_sockets;
          // Timeout de 2 secondes pour select()
          timer.tv_sec = 2;
```

```
        timer.tv_usec = 0;

        // Surveille les sockets prêtes à être lues
        status = select(fd_max + 1, &read_fds, NULL, NULL, &timer);
        if (status == -1)
        {
            printf("[Server] Select error: %s\n", strerror(errno));
            exit(-1);
        }
        else if (status == 0)
        {
            // Aucun descipteur de fichier de socket n'est prêt pour la lecture
            printf("[Server] Waiting...\n");
            continue;
        }

        // Boucle sur nos sockets
        i = 0;
        while (i <= fd_max)
        {
          if (FD_ISSET(i, &read_fds) != 1)
          {
                // Le fd i n'est pas une socket à surveiller
                // on s'arrête là et on continue la boucle
                i++
                continue ;
          }
          printf("[%d] Ready for I/O operation\n", i);
          // La socket est prête à être lue !
          if (i == server_socket)
                // La socket est notre socket serveur qui écoute le port
                accept_new_connection(server_socket, &all_sockets, &fd_max);
          else
                // La socket est une socket client, on va la lire
                read_data_from_socket(i, &all_sockets, fd_max, server_socket);
          i++;
        }
    }
    return (0);
}

// Renvoie la socket du serveur liée à l'adresse et au port qu'on veut écouter
```

```c
int create_server_socket(void)
{
	struct sockaddr_in sa;
	int socket_fd;
	int status;

	// Préparaton de l'adresse et du port pour la socket de notre serveur
	memset(&sa, 0, sizeof sa);
	sa.sin_family = AF_INET; // IPv4
	sa.sin_addr.s_addr = htonl(INADDR_LOOPBACK); // 127.0.0.1, localhost
	sa.sin_port = htons(PORT);

	// Création de la socket
	socket_fd = socket(sa.sin_family, SOCK_STREAM, 0);
	if (socket_fd == -1)
	{
	printf("[Server] Socket error: %s\n", strerror(errno));
	return (-1);
	}
	printf("[Server] Created server socket fd: %d\n", socket_fd);

	// Liaison de la socket à l'adresse et au port
	status = bind(socket_fd, (struct sockaddr *)&sa, sizeof sa);
	if (status != 0)
	{
	printf("[Server] Bind error: %s\n", strerror(errno));
	return (-1);
	}
	printf("[Server] Bound socket to localhost port %d\n", PORT);
	return (socket_fd);
}

// Accepte une nouvelle connexion et ajoute la nouvelle socket à l'ensemble des sockets
void accept_new_connection(int server_socket, fd_set *all_sockets, int *fd_max)
{
	int client_fd;
	char msg_to_send[BUFSIZ];
	int status;

	client_fd = accept(server_socket, NULL, NULL);
	if (client_fd == -1)
	{
```

```
	printf("[Server] Accept error: %s\n", strerror(errno));
	return;
	}
	FD_SET(client_fd, all_sockets); // Ajoute la socket client à l'ensemble
	if (client_fd > *fd_max)
	*fd_max = client_fd; // Met à jour la plus grande socket
	printf("[Server] Accepted new connection on client socket %d.\n", client_fd);
	memset(&msg_to_send, '\0', sizeof msg_to_send);
	sprintf(msg_to_send, "Welcome. You are client fd [%d]\n", client_fd);
	status = send(client_fd, msg_to_send, strlen(msg_to_send), 0);
	if (status == -1)
	printf("[Server] Send error to client %d: %s\n", client_fd, strerror(errno));
}

// Lit le message d'une socket et relaie le message à toutes les autres
void read_data_from_socket(int socket, fd_set *all_sockets, int fd_max, int
server_socket)
{
	char buffer[BUFSIZ];
	char msg_to_send[BUFSIZ];
	int bytes_read;
	int status;

	memset(&buffer, '\0', sizeof buffer);
	bytes_read = recv(socket, buffer, BUFSIZ, 0);
	if (bytes_read <= 0)
	{
	if (bytes_read == 0)
		printf("[%d] Client socket closed connection.\n", socket);
	else
		printf("[Server] Recv error: %s\n", strerror(errno));
	close(socket); // Ferme la socket
	FD_CLR(socket, all_sockets); // Enlève la socket de l'ensemble
	}
	else
	printf("[%d] Got message: %s", socket, buffer);
}
```

On remarquera ici que nous avons deux ensembles de descripteurs de fichiers : `all_sockets`, qui contient tous les descripteurs de fichiers de toutes les sockets connectées, et `read_fds`, l'ensemble qui sera surveillé pour la lecture. L'ensemble

`all_sockets` est vital pour ne pas perdre de descripteurs en cours de route, vu que `select()` modifie l'ensemble `read_fds` pour indiquer les descripteurs qui sont prêts. C'est donc dans `all_sockets` qu'on voudra ajouter les descripteurs de fichier de nos nouvelles connexions, et enlever les sockets qui ont été fermées. Au début de notre boucle principale, on peut alors copier le contenu de `all_sockets` dans `read_fds` pour s'assurer que `select()` surveille bien toutes nos sockets.

EXERCICE 2 :

Créer un programme serveur **my_tcp_server** qui reçoit et affiche les messages envoyés par de multiples clients **netcat**.

Terminal 1

```
~/> ./my_tcp_server_nonblock 4242
Starting on port 4242.
Binding OK
Waiting...
[socket 4] Accepted new client on socket 4
[socket 5] Accepted new client on socket 5
[socket 5] Got message: HELLO
[socket 5] Got message: Comment ca va ?
[socket 4] Got message: Test
[socket 5] Client socket closed connection on socket 5
[socket 6] Accepted new client on socket 6
[socket 6] Got message: Putain ca marche
[socket 6] Client socket closed connection on socket 6

...
```

Terminal 2

```
~/> ./nc localhost 4242
HELLO
Comment ca va ?
```

Terminal 3

```
~/> ./netcat localhost 4242
Test
^C
```

```
~/>
```

Terminal 4

```
~/> ./netcat localhost 4242
Putain ca marche
^C
~/>
```

CA FONCTIONNE ?

Nous avons donc ici l'ébauche d'un serveur de chat !
Il nous suffira de renvoyer le message reçu à l'ensemble des sockets pour que tous les clients puissent le voir !

Idéalement, on devrait aussi surveiller nos sockets pour l'écriture avant de leur envoyer des données, mais maintenant qu'on comprend le fonctionnement de `select()`, cela ne devrait pas être trop compliqué à implémenter. Il nous faudrait pour cela ajouter une deuxième boucle après notre appel à `select()` pour consulter cette fois l'ensemble des descripteurs de fichiers surveillés pour la lecture.

Vous pouvez commencer le projet my_teams.