2ème année DUT Informatique
IUT Aix-en-Provence

TP2 Programmation répartie
Sockets TCP et UDP en C

Safa YAHI

Exercice 1

Considérons le serveur suivant :

```
#include <arpa/inet.h>
#include <sys/socket.h>
#include <unistd.h>
#include <iostream>
#include <string.h>
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

#define NUM_PORT 50013
#define BACKLOG 50
#define NB_CLIENTS 100
#define TAILLE_BUFFER 1024

using namespace std;

void exitErreur(const char * msg) {
	perror(msg);
	exit( EXIT_FAILURE);

}

int main(int argc, char * argv[]) {
	int port;

	if (argc == 2)
		port = atoi(argv[1]);
	else
		port = NUM_PORT;

	int sock_serveur = socket(AF_INET, SOCK_DGRAM, 0);

	struct sockaddr_in sockaddr_serveur;

	sockaddr_serveur.sin_family = AF_INET;
	sockaddr_serveur.sin_port = htons(port);
	sockaddr_serveur.sin_addr.s_addr = htonl(INADDR_ANY);

	int yes = 1;
	if (setsockopt(sock_serveur, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int))
			== -1)
		exitErreur("setsockopt");

	if (bind(sock_serveur, (struct sockaddr *) &sockaddr_serveur,
			sizeof(sockaddr_in)) == -1)
```

```
            exitErreur("bind");

      sockaddr_in sockaddr_client;
      socklen_t size = sizeof(sockaddr_client);

      char buf[TAILLE_BUFFER];

      char * msg;
      time_t date;

      cout << "Serveur lancé  sur le port " << port << endl;

      for (int i = 1; i <= NB_CLIENTS; i++) {

            recvfrom(sock_serveur, buf, sizeof(buf), 0,
                        (struct sockaddr *) &sockaddr_client, &size);

            date = time(NULL);
            msg = ctime(&date);

            if (sendto(sock_serveur, msg, strlen(msg), 0,
                        (struct sockaddr *) &sockaddr_client,
sizeof(sockaddr_client))
                        == -1)
                  exitErreur("sendto");

      }
      close(sock_serveur);
      return 0;
}
```

Répondez aux questions suivantes :

1) Quel est le protocole transport utilisé ici (TCP ou UDP) ?
2) Comparez le schéma général d'un serveur TCP avec celui d'un serveur UDP.
3) A quel niveau le serveur se bloque-t-il pour attendre les requêtes des clients ?
4) Comment le serveur determine-t-il l'adresse de la socket du client ?
5) A quel moment on utilise cette dernière ?
6) Lancez ce serveur et testez-le en local et en réseau depuis la machine de votre voisin avec l'utilitaire nc (nc pour netcat). Rappelons que netcat fonctionne par défaut en TCP. Pour utiliser le mode UDP, on rajoute l'option -u. Voici un exemple “nc -u 192.168.1.76 50013”.
7) Quelle est l'utilité de la fonction setsockopt( ) ?

Exercice 2

Implémentez un client daytime en UDP. Testez-le avec le serveur précédent aussi bien en local qu'en réseau.

Exercice 3 (Application TCP avec fork())

Considérons l'application client/serveur TCP de transfert de fichiers du TP1. Modifiez-la pour intégrer les éléments suivants :

- Gérer plusieurs clients simultanément avec fork().
- Le nombre de clients à traiter en même temps ne doit pas dépasser une certaine valeur.

Exercice 4 (Multiplexage avec select())

Certaines applications peuvent avoir besoin de surveiller plusieurs descripteurs de fichiers pour voir si c'est possible de faire des E/S dessus. Deux solutions sont possibles :

- utiliser des E/S non bloquantes
- ou utiliser des processus / threads.

Si on veut surveiller plusieurs descripteurs de fichiers avec des E/S non bloquantes, on peut appliquer périodiquement des E/S non bloquantes sur eux. Néanmoins, si la période est longue, celà induit un temps de réponse inacceptable. En revanche, s'il est très court, celà consomme des ressources CPU.

Pour l'utilisation des processus, celà est coûteux. Les threads le sont aussi (bien que moins que les processus) et leur utilisation peut être en plus complexe (gérer l'exclusion mutuelle, utiliser des pools de thread, etc.).

Une des alternatives est d'utiliser le multiplexage des E/S avec les fonctions select() et pool(). Ces fonctions on été largement utilisées et sont portables. Néanmoins, elle ne passent pas très bien à l'échelle pour traiter des centaines de miliers de descripteurs de fichier.

Une autre possibilité est d'utiliser epoll() qui permet de passer à l'échelle mais elle moins portable. Enfin, on a aussi E/S orientées signaux qui passent bien à l'échelle. Cependant, ils sont moins portables et aussi complexes à mettre en oeuvre.

Nous nous intéressons ici uniquement à la fonction select(). Cette fonction se bloque pour scruter plusieurs descripteurs de fichiers, dès que l'un d'eux est prêt pour une lecture ou une écriture, elle retourne. Voici son profil.

```
#include <sys/time.h>
 #include <sys/select.h>


int select(int nfds,
        fd_set *readfds,            // liste DFs à scruter pour lecture
        fd_set *writefds,           // liste DFs à scruter pour écriture
        fd_set *exceptfds,          // liste DFs de données urgentes.
        struct timeval *timeout);   // délai maximum d'attente


retourne le nombre de descripteurs de fichiers prêts, 0 sur timeout, ou -1 en cas d'erreur.
```

La manipulation des listes de DFs (de type fd_set) se fait via les macro suivantes :

```
#include <sys/select.h>

void FD_ZERO(fd_set *fdset);        // initialise l'ensemble pointée par fdset à l'ensemble vide
```

```
void FD_SET(int fd, fd_set *fdset);  // rajoute l'élément fd à la liste fdset
void FD_CLR(int fd, fd_set *fdset); // supprime fd de fdset
int FD_ISSET(int fd, fd_set *fdset); // vérifie si fd est dans fdset.
```

Les arguments readfds, writefds et exceptfds doivent être initialisés avant l'appel de select pour contenir les DFs à scruter. Au retour de select(), ils contiennent les DFs prêts. Si on utilise un select plusieurs fois, il faut penser à réinitialiser les listes de DFs à chaque fois.

Le paramètre nfds doit être initialisé à la plus grande valeurs de DF scruté (dans les trois listes de DFs) à laquelle on rajoute 1.

La structure timeval est définie comme suit :

```
struct timeval {
        time_t tv_sec;                  /* Seconds */
        suseconds_t tv_usec;  /         * Microseconds */
};
```

Si timeout vaut “null”, alors select() se bloque indéfiniment. Si les deux champs sont nuls, alors il revient immédiatement.

Enfin, voici une situation où ce principe de multiplexage a un sens. En fait, certains services réseau sont rarement sollicités et donc les serveurs associés ne font qu'attendre des clients tout en consommant des ressources. Afin d'économiser ces derneières, au lieu de lancer un processus par service, on peut utiliser un seul processus qui attend les demandes de connexion ou les requêtes en UDP et lance le service associé uniquement quand c'est nécessaire. C'est le cas du superserveur “inetd”.

Question :

Implémentez, en utilisant select(), un serveur daytime qui supporte aussi bien le mode TCP que le mode UDP. Plus précisément,

1) pour les deux services, il crée une socket du type appropié et la lie à la bonne adresse locale (appel socket() et appel bind()). S'il s'agit d'une socket TCP, alors elle est mise en mode écoute (appel listen()).
2) Il utilise la fonction select() pour surveiller ces deux sockets.
3) l'appel select() se bloque jusqu'à ce qu'une socket UDP ait un datagramme disponible ou une socket TCP reçoit une demande de connexion. S'il s'agit de TCP, inetd fait en plus un appel accept().
4) Lance le bon traitement
5) revient à l'étape 2°.