TP 4 - PSE
Programmation réseau et Socket

Le compte-rendu est à déposer dans le puits après les vacances de Noël !!

Objectifs :

L'objectif de ce TP est de se familiariser (et maîtriser) avec la programmation réseau et l'utilisation des sockets (TCP) : ***socket***, ***bind***, ***connect***, ***listen***, ***accept***, ***write et read***, ***sendto et send***, ***recv et recvfrom***, ***close***, ***shutdown***, ***...***

Travail à effectuer :

Étape n°1 : création de la socket (côté client)
Pour créer une socket, on utilisera l'appel système ***socket()***
On commence par consulter la page du manuel associée à cet appel ou consulter les diapositives du cours :
$ man 2 socket
...
#include <sys/types.h> /* See NOTES */
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
DESCRIPTION
socket() crée un point de communication, et renvoie un descripteur.
...
VALEUR RENVOYÉE
Cet appel système renvoie un descripteur référençant la socket créée s'il réussit.
S'il échoue, il renvoie -1 et errno contient le code d'erreur.
...

À l'aide d'un éditeur de texte (gedit, vim, notepad++, ...), tapez le programme suivant dans un fichier que vous nommerez par exemple **"client-TCP-1.c"** :

```c
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <sys/types.h>
#include <sys/socket.h>
#include <errno.h>

int main() {
	int sock;

	// Creation de la socket de communication
	sock = socket(AF_INET, SOCK_STREAM, 0); /* 0 indique que l'on utilisera le protocole associe
par defaut a SOCK_STREAM soit TCP */

	if(sock < 0) {
		perror("socket");
		exit(-1);
	}

	printf("Socket creee avec succes ! (%d)\n", sock);

	// On ferme la socket avant de quitter
	close(sock);
	return(EXIT_SUCCESS);
}
```

Pour le troisième argument (protocol), on a utilisé la valeur 0. On aurait pu écrire aussi ''IPPROTO_TCP''. **Donc testez le programme en remplaçant 0 par IPPROTO_TCP**.

Étape n°2 : connexion au serveur

Maintenant qu'on a créé une socket TCP, on va la connecter au processus serveur distant. Pour cela, on va utiliser l'appel système ***connect().*** On commence par consulter la page du manuel associée à cet appel :

$ man 2 connect

...

#include <sys/types.h> /* See NOTES */

#include <sys/socket.h>

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

DESCRIPTION

L'appel système connect() connecte la socket référencée par le descripteur de fichier sockfd à l'adresse indiquée par serv_addr. ...

VALEUR RENVOYÉE

connect() renvoie 0 s'il réussit, ou -1 s'il échoue, auquel cas errno contient le code d'erreur.

...

On a vu en cours qu'un processus est identifié par :
– une **adresse IPv4** sur 32 bits
– un **numéro de port** sur 16 bits

L'interface socket propose une structure d'adresse générique : struct sockaddr.
Je vous rappelle que cette structure contient les membres suivants :

```
struct sockaddr
{
        unsigned short int sa_family; //au choix, par exemple AF_INET
        unsigned char sa_data[14]; //en fonction de la famille
};

// Remarque : ces structures sont déclarées dans <netinet/in.h>

struct in_addr { unsigned long int s_addr; }; // une adresse Ipv4 (32 bits)
struct sockaddr_in
{
        unsigned short int   sin_family; // <- AF_INET
        unsigned short int   sin_port; // <- numéro de port
        struct in_addr       sin_addr; // <- adresse IPv4
};
```

Il suffit donc de déclarer et d'initialiser une structure **sockaddr_in** avec les informations du serveur (adresse IPv4 et numéro de port).
Pour écrire ces informations dans la structure d'adresse, il nous faudra utiliser :
– inet_aton() pour convertir une **adresse IP** depuis la notation IPv4 décimale pointée vers une forme binaire (dans l'ordre d'octets du réseau)
– htons() pour convertir le **numéro de port** (sur 16 bits) depuis l'ordre des octets de l'hôte vers celui du réseau.
Je vous rappelle que l'ordre des octets du réseau est ***Big Endian***. Il est donc toujours prudent d'appeler des fonctions qui respectent cet ordre pour coder des informations dans les en-têtes des protocoles réseaux.

Maintenant on va écrire le 2ème programme qu'on nommera ''**client-TCP-2.c**'' :

```
#include <stdio.h>
#include <stdlib.h> /* pour exit */

#include <string.h> /* pour memset */
#include <netinet/in.h> /* pour struct sockaddr_in */
#include <arpa/inet.h> /* pour htons et inet_aton */

#include <sys/types.h> /* pour socket */
#include <sys/socket.h>
#include <errno.h>
```

```c
int main() {
	int sock;
	struct sockaddr_in adresse;
	// socklen_t : an integer type of width of at least 32 bits
	socklen_t longueurAdresse;


	// Crée un socket de communication
	sock = socket(AF_INET, SOCK_STREAM, 0);

	if(sock < 0) {
		perror("socket"); // Affiche le message d'erreur
		exit(-1); // On sort en indiquant un code erreur
	}

	printf("Socket creee avec succes ! (%d)\n", sock);


	// On obtient la longueur en octets de la structure sockaddr_in
	longueurAdresse = sizeof(adresse);

	// On initialise à 0 la structure sockaddr_in
	memset(&adresse, 0x00, longueurAdresse);

	// Maintenant on va renseigner la structure sockaddr_in avec les informations du serveur
distant
	adresse.sin_family = AF_INET;
	// On choisit le numéro de port d'écoute du serveur
	adresse.sin_port = htons(IPPORT_USERRESERVED);
	printf("\n Numero de port choisi est : %d \n", IPPORT_USERRESERVED);
	// On choisit l'adresse IPv4 du serveur ; on suppose que cette adresse est : 172.19.32.22
	inet_aton("172.19.32.22", &adresse.sin_addr); // à modifier selon ses besoins

	// On débute la connexion vers le processus serveur distant
	if( (connect(sock, (struct sockaddr *)&adresse, longueurAdresse) ) == -1) {
		perror("connect");
		close(sock);
		exit(-2);
	}

	printf("Connexion au serveur reussie !\n");
	// On ferme la ressource avant de quitter
	close(sock);
	return EXIT_SUCCESS;
}
```

Lorsqu'on va tester ce client, on risque d'obtenir le message suivant :

```
$ ./client-TCP-2
Socket creee avec succes ! (3)
Numero de port choisi est : 5000
connect: Connection refused
```

Ceci peut s'expliquer tout simplement par le fait qu'il n'y a pas de processus serveur à cette adresse !

Étape n°3 : vérification du bon fonctionnement de la connexion

Pour tester notre client, il nous faut absolument un serveur !
Pour cela, on va utiliser l'outil réseau **netcat** (ou **nc**) en mode serveur (-l) sur le port 5000 (-p 5000) :

```
$ nc -l -p 5000
```

Puis :

```
$ ./client-TCP-2
```

```
Socket creee avec succes ! (3)
Numero de port choisi est : 5000
Connexion au serveur reussie !
```

Dans l'architecture client/serveur, on rappelle que c'est le client qui a l'initiative de l'échange. Il faut donc que le serveur soit en écoute avant que le client fasse sa demande.

Question : Modifier le code du client en lui passant l'adresse du serveur et le numéro de port en arguments. Si le client ne reçoit pas ces 2 arguments il affichera un message comportant l'utilisation de l'exécutable.

Étape n°4 : échange des données

On rappelle qu'une communication TCP est bidirectionnelle ***full duplex*** et orientée flux d'octets. Il nous faut donc des fonctions pour écrire (envoyer) et lire (recevoir) des octets dans la socket.
Les fonctions d'échanges de données sur une socket **TCP** sont :
– ***read()*** et ***write()*** qui permettent la réception et l'envoi d'octets sur une socket.
– ***recv()*** et ***send()*** qui permettent la réception et l'envoi d'octets sur une socket avec des attributs (voir le cours ou taper *man* 2 *send*)
Les appels *recv()* et *send()* sont spécifiques aux sockets en mode connecté.

Maintenant on va écrire un client qui envoie en premier une chaîne de caractères et le serveur lui répondra "ok"

```c
#include <stdio.h>
#include <stdlib.h> /* pour exit */

#include <sys/types.h> /* pour socket */
#include <sys/socket.h>

#include <string.h> /* pour memset */
#include <netinet/in.h> /* pour struct sockaddr_in */
#include <arpa/inet.h> /* pour htons et inet_aton */
```

```c
#define LG_MESSAGE 256


int main() {
	int sock;
	struct sockaddr_in adresse; /* adresse et numero de port du serveur */
	socklen_t longueurAdresse;
	char messageEnvoi[LG_MESSAGE]; /* le message envoye */
	char messageRecu[LG_MESSAGE]; /* le message recu */
	int nbo_ecrits, nbo_lus, retour;

	sock = socket(AF_INET, SOCK_STREAM, 0);

	if(sock < 0) {
		perror("socket");
		exit(-1);
	}
	printf("Socket creee avec succes ! (%d)\n", sock);

	longueurAdresse = sizeof(adresse);
	memset(&adresse, 0x00, longueurAdresse);
	adresse.sin_family = AF_INET;
	adresse.sin_port = htons(IPPORT_USERRESERVED); // = 5000
	inet_aton("172.19.32.22", &adresse.sin_addr);

	// On se connecte au serveur
	if((connect(sock, (struct sockaddr *)&adresse, longueurAdresse)) == -1) {
		perror("connect");
		close(sock);
		exit(-2);
	}

	printf("Connexion au serveur reussie avec succes !\n");

	// On initialise à 0 les messages
	memset(messageEnvoi, 0x00, LG_MESSAGE*sizeof(char));
	memset(messageRecu, 0x00, LG_MESSAGE*sizeof(char));

	// Envoie un message au serveur
	sprintf(messageEnvoi, "HELLO MR VS-UP5IUT !\n");
	nbo_ecrits = write(sock, messageEnvoi, strlen(messageEnvoi));
	switch(nbo_ecrits) {
		case -1 : /* en cas d'une erreur */
			perror("write");
			close(sock);
			exit(-3);
		case 0 : /* la socket est fermee */
			fprintf(stderr, "La socket a ete fermee par le serveur !\n\n");
			close(sock);
			return 0;
		default: /* envoi de n octets */
			printf("Message %s a ete envoye avec succes (%d octets)\n\n", messageEnvoi,
nbo_ecrits);
	}
```

```
	/* On recoit des données du serveur */
	nbo_lus = read(sock, messageRecu, LG_MESSAGE*sizeof(char)); /* on attend un message de TAILLE
fixe */
	switch(nbo_lus) {
		case -1 :
			perror("read");
			close(sock);
			exit(-4);
		case 0 : /* la socket est fermee */
			fprintf(stderr, "La socket a ete fermee par le serveur !\n\n");
			close(sock);
			return 0;
		default: /* reception de nbo_lus octets */
			printf("Message recu du serveur : %s (%d octets)\n\n", messageRecu, nbo_lus);
	}


	// On ferme toujours la socket avant de quitter
	close(sock);
	return EXIT_SUCCESS;
}
```

On utilise la même procédure de test que précédemment en démarrant un ***serveur netcat*** sur le port 5000 : $ *nc -l -p 5000*
Puis, on exécute notre client :
$ ./client-TCP-3
Socket creee avec succes ! (3)
Connexion au serveur reussie avec succes !
Message HELLO MR VS-UP5IUT !
a ete envoye avec succes (21 octets)
Message recu du serveur : Hello dear !
(13 octets)
Pour que le client puisse recevoir le message du serveur, dans la console où on a exécuté nc il suffit de taper le message ''Hello dear !'' et après valider avec la touche **Entrée** !
Dans la console où on a exécuté le serveur netcat, cela donne :
$ nc -l -p 5000
HELLO MR VS-UP5IUT !
Hello dear !

Question : que se passe-t-il si le serveur s'arrête (en tapant Ctrl-C par exemple !) au lieu d'envoyer "ok" ?

Dans les codes sources ci-dessus, nous avons utilisés l'appel ***close()*** pour fermer la socket et donc la communication. En TCP, la communication étant *full duplex*, il est possible de fermer plus finement l'échange en utilisant l'appel ***shutdown()*** :
$ man 2 shutdown
...
shutdown - Terminer une communication en full-duplex
...
#include <sys/socket.h>

int shutdown(int s, int how);
DESCRIPTION
La fonction shutdown() termine tout ou partie d'une connexion full-duplex sur la socket s.
Si how vaut SHUT_RD, la réception est désactivée.
Si how vaut SHUT_WR, l'émission est désactivée.
Si how vaut SHUT_RDWR, l'émission et la réception sont désactivées.
VALEUR RENVOYÉE
Cet appel système renvoie 0 s'il réussit, ou -1 s'il échoue, auquel cas errno contient le code d'erreur.
...

Étape n°5 : réalisation d'un serveur TCP

Évidemment, un serveur TCP a lui aussi besoin de créer une socket SOCK_STREAM dans le domaine AF_INET.
Le code source d'un serveur TCP basique sera différent d'un client TCP dans le principe.
On rappelle qu'un serveur TCP attend des demandes de connexion en provenance de ***processus clients.*** Ce dernier doit connaître au moment de la connexion le numéro de port d'écoute du serveur. Pour mettre en œuvre cela, le serveur va utiliser l'appel système ***bind()*** qui va lui permettre de lier sa socket d'écoute à une interface et à un numéro de port local à sa machine.
Une fois que le serveur a créé et attaché une socket d'écoute, il doit la placer en attente passive, c'est-à-dire être capable d'accepter les demandes de connexion des processus clients. Pour cela, on va utiliser l'appel système ***listen()***.

Donc, éditer le programme suivant dans un fichier que vous nommerez ''serveur-TCP-1.c''.

```
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <unistd.h> /* pour sleep */
#include <string.h> /* pour memset */

#include <sys/types.h> /* pour socket */
#include <sys/socket.h>


#include <netinet/in.h> /* pour struct sockaddr_in */
#include <arpa/inet.h> /* pour htons et inet_aton */

#define PORT IPPORT_USERRESERVED // = 5000


int main() {
	int socketEcoute;
	struct sockaddr_in adresse; /* adresse d'attachement locale */
	socklen_t longueurAdresse;

	socketEcoute = socket(AF_INET, SOCK_STREAM, 0);

	if(socketEcoute < 0) {
		perror("socket");
		exit(-1);
```

```
        }

        printf("Socket creee avec succes ! (%d)\n", socketEcoute);

        // On prépare l'adresse d'attachement locale
        longueurAdresse = sizeof(struct sockaddr_in);
        memset(&adresse, 0x00, longueurAdresse);
        adresse.sin_family = AF_INET;
        adresse.sin_addr.s_addr = htonl(INADDR_ANY);
        adresse.sin_port = htons(PORT); // = 5000

        // On demande l'attachement local de la socket
        if(( bind(socketEcoute, (struct sockaddr *)&adresse, longueurAdresse) ) < 0) {
                perror("bind");
                exit(-2);
        }
        printf("Socket attachee avec succes !\n");


        // La taille de la file d'attente est fixee a 5
        if(listen(socketEcoute, 5) < 0) {
                perror("listen");
                exit(-3);
        }

        printf("Socket placee en ecoute passive ...\n");

        // On ferme la ressource avant de quitter
        close(socketEcoute);
        return EXIT_SUCCESS;
}
```

Le test de ce programme nous donne :
Socket creee avec succes ! (3)
Socket attachee avec succes !
…

Étape n°6 : réalisation d'un serveur TCP avec acceptation des demandes de connexions !

Cette étape est cruciale pour le serveur. Il lui faut maintenant accepter les demandes de connexion en provenance des processus clients. Pour cela, il va utiliser l'appel système ***accept()*** :

```
$ man 2 accept
…
accept - Accepter une connexion sur une socket
…
#include <sys/types.h> /* See NOTES */
#include <sys/socket.h>
int accept (int sockfd, struct sockaddr *adresse, socklen_t *longueur);
```

DESCRIPTION
L'appel système accept() est employé avec les sockets utilisant un protocole en mode connecté SOCK_STREAM. Il extrait la première connexion de la file des connexions en attente de la socket sockfd à l'écoute, crée une nouvelle socket connectée, et renvoie un nouveau descripteur qui fait référence à cette socket. La nouvelle socket n'est pas en état d'écoute. La socket originale sockfd n'est pas modifiée par l'appel système.
...
VALEUR RENVOYÉE
S'il réussit, accept() renvoie un entier non négatif, constituant un descripteur pour la nouvelle socket. S'il échoue, l'appel renvoie -1 et errno contient le code d'erreur.

Questions :

1) **Modifier le programme précédent (serveur-TCP-1.c) pour permettre au serveur d'accepter une demande de connexion, envoyer et recevoir un message d'un client (n'hésitez pas à aller voir l'exemple expliqué en cours).**

2) **Compléter le programme précédent pour permettre au serveur d'accepter et de traiter plusieurs demandes de connexions de différents clients (listen, accept, fork, etc.)**