2ème année DUT Informatique Année 2023-2024
IUT Aix-en-Provence

TP2 Programmation répartie
Sockets en C

Safa YAHI

Exercice 1 : fonction readLine()

Dans plusieurs protocoles applicatifs tels que HTTP, SMTP, IMAP, etc, on est amené à traiter des messages multi-lignes.
Cet exercice s'articule autour d'une fonction **readLine()** (en mode TCP) qui retourne la prochaine ligne (séparée de ce qui suit "\n") disponible en lecture pour une socket. Cette fonction est justement adaptée à des messages multi-lignes.

1. Accédez depuis ametice au code du serveur TCP donné par **serveur_readline().cxx**. Comme vous pouvez le constater, une fois un client se connecte, le serveur fait des appels à la fonction **readLine()** (méthode membre d'une nouvelle classe que nous avons appelée SocketReader) jusqu'à une ceraine condition. Pour tester ce serveur, par exemple avec des messges multi-lignes qui arrivent en même temps, référez vous au **client_readline.cxx** qui lui envoie le message "Hi\nça va?\nbye\n.\n"

=> Executez le serveur en spécifiant, depuis la ligne de commande, uniquement son port d'écoute, et exécutez le client en spécifiant l'adresse et le port du serveur auquel il doit se connecter. Quel est le résultat affiché côté serveur ?

2. Analysez le code et expliquez le fonctionnement de la méthode membre readLine(). Les commentaires devraient vous aider ;) Est-ce que la ligne retournée contient "\n" ?

3. Modifiez le code de l'application précédente (serveur_readline.cxx et client_readline.cxx) pour implémenter une nouvelle application client/serveur TCP pour **le protocole Echo** qui se fait ici dans une boucle : à chaque fois que le serveur reçoit **une ligne** de la part du client, il la lui renvoie, et ce jusqu'à ce qu'il reçoive la ligne contenant ".".

4. Testez ce serveur en lançant plusieurs clients et vérifiez qu'il les traite séquentiellement. Notons qu'il est possible pour un serveur de traiter des clients en parallèle avec l'utilisation des processus fils (appel de fork()) qui sera vu plus tard dans ce cours.

Exercice 2 : Chat à tour de rôle entre client et serveur

Considérons une application **client / serveur TCP** qui s'échangent, à tour de rôle, des messages. Plus précisément :

- le client se connecte au serveur,
- lui envoie un premier message que l'utilisateur aurait saisi.
- le serveur lui répond par un autre message saisi par l'utilisateur

- le client lui envoie un nouveau message et ainsi de suite jusqu'à ce que le client envoie le message “Bye” ou bien il y a déconnexion.

On suppose que le serveur traite les clients séquentiellement et non pas “simultanément”.

Par ailleurs, on sait que pour récupérer le message envoyé par l'autre en TCP, il faut une boucle de read(). Dans cette application, la condition d'arrêt de cette boucle ne peut pas être « read()=0 » car il s'agit d'un dialogue à plusieurs étapes et la déconnexion (et donc read() retourne 0) ne se fait qu'à la fin.

On suppose alors que les messages échangés sont sous forme de lignes se terminant par « \n ».

=> Écrivez d'abord le serveur en C/C++ et testez-le avec un client Telnet. Notez que lorsque l'utilisateur saisit par exemple « Bye » en Telnet, le message envoyé est « Bye\r\n ». Pensez à utiliser getline() au lieu de cin pour lire les messages afin de prendre des « ».

=> Écrivez le client en C/C++

=>Testez cette application en réseau pour discuter avec vos voisin(e)s.

Exercice 3 : Démystifier l’option SO_REUSEADDR.

Déjà dans le TP1, vous avez peut-être remarqué cette portion dans le code du serveur daytime et qui s’est répétée pour les autres serveurs :

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

Kezako **SO_REUSEADDR ?**

=> Reportez vous aux slides 13-19 du cours 2 pour comprendre tout ça et aussi pour faire la manipulation proposée.