TD 7 : Sockets client Serveur

Objectifs pédagogiques :

- **Sockets**

Introduction

Dans ce TD, nous allons étudier l'API proposée par les sockets pour la communication entre machines.

Les sockets sont un moyen d'établir des communications entre processus distants.

2 Client Serveur TCP

On souhaite construire des classes de librairie C++ pour la gestion des sockets, rendant plus aisée leur utilisation dans une approche orientée objet.

Question 1. Construire un `operator«` permettant d'afficher lisiblement un type `sockaddr_in`. On souhaite afficher l'IP (v4), le nom de l'hôte, et le numéro du port.

On rappelle les chaps d'une addresse de socket du domaine internet :

- `sin_family` : discriminant de type d'addresse, prend la valeur `AF_INET`,
- `sin_port` : le port au format internet,
- `sin_addr` : un pointeur vers l'addresse IPv4;

On utilisera `getnameinfo` pour obtenir le nom d'hôte et `inet_ntoa` pour afficher une IP. Attention à convertir vers le format hôte. Cf. slides 19 à 26 du cours 7.

Question 2. Ecrire une classe Socket représentant une socket TCP. Elle porte en attribut un filedescriptor qui vaut -1 tant que la socket n'est pas connectée.

Socket.h

```
#ifndef SRC_SOCKET_H_
#define SRC_SOCKET_H_

#include <netinet/ip.h>
#include <string>
#include <iosfwd>

namespace pr {

class Socket {
      int fd;

public :
      Socket():fd(-1){}
      Socket(int fd):fd(fd){}

      // tente de se connecter à l'hôte fourni
      void connect(const std::string & host, int port);
      void connect(in_addr ipv4, int port);

      bool isOpen() const {return fd != -1;}
      int getFD() { return fd ;}

      void close();
};

```

```
std::ostream & operator<< (std::ostream & os, struct sockaddr_in * addr);

}

#endif /* SRC_SOCKET_H_ */
```

Question 3. Elaborer à présent une classe représentant une socket serveur.

ServerSocket.h

```
#ifndef SRC_SERVERSOCKET_H_
#define SRC_SERVERSOCKET_H_

#include "Socket.h"

namespace pr {

class ServerSocket {
      int socketfd;

public :
      // Demarre l'ecoute sur le port donne
      ServerSocket(int port);

      int getFD() { return socketfd;}
      bool isOpen() const {return socketfd != -1;}

      Socket accept();

      void close();
};

} // ns pr
#endif /* SRC_SERVERSOCKET_H_ */
```

Dès la construction, on doit se mettre en attente de connexions TCP sur le port indiqué. L'appel à `accept` est bloquant, et rend une socket connectée au client.

On affichera l'addresse du client qui vient de se connecter à chaque connection (dans accept).

Question 4. Ecrivez un code client, qui se connecte à un serveur donné, lui envoie une donnée (un entier), puis lis sa réponse (un entier) avant de quitter.

Question 5. Ecrivez un code serveur, qui attend des connexions, et quand un client se connecte commence par lire un message (un entier) puis renvoie un message (un entier), puis se remet en attente de connexion.

Question 6. On souhaite généraliser le code du serveur, ecrire une classe `TCPServer` qui encapsule le comportement du serveur. La classe porte une opération `bool startServer(int port)` qui démarre l'écoute sur le port ciblé à l'aide d'une ServerSocket. A la construction on lui passe un `ConnectionHandler` : un objet devant gérer une session avec un client donné.

ConnectionHandler.h

```
#ifndef SRC_CONNECTIONHANDLER_H_
#define SRC_CONNECTIONHANDLER_H_

#include "Socket.h"

namespace pr {

// une interface pour gerer la communication
class ConnectionHandler {
```

```
public:
      // gerer une conversation sur une socket
      virtual void handleConnection(Socket s) = 0;
      // une copie identique
      virtual ConnectionHandler * clone() const = 0;
      // pour virtual
      virtual ~ConnectionHandler() {}
};}
#endif /* SRC_CONNECTIONHANDLER_H_ */
```

Il est possible que chaque session avec un client ait un état interne, e.g. le client s'est authentifié, d'où la nécessité de cloner le *handler* à chaque nouvelle connexion d'un client. Cette approche correspond au design pattern *Prototype*.

Question 7. On souhaite que l'invocation à `startServer` ne bloque pas l'appelant, mais crée au contraire un nouveau thread qui s'occupe d'attendre des connexions.

Question 8. Comment gérer l'opération symétrique `stopServer`, qui doit interrompre le thread d'attente et fermer proprement la socket ?

Question 9. Proposer un adapter pour permettre l'encapsulation de la gestion d'une session de discussion avec un client dans un `Job` muni d'une unique méthode `void run()` que l'on peut soumettre à un Pool de thread.

Job.h

```
#ifndef SRC_JOB_H_
#define SRC_JOB_H_

class Job {
public:
      virtual void run () = 0;
      virtual ~Job() {};
};
#endif /* SRC_JOB_H_ */
```

TME 7 : Sockets Client Serveur

Objectifs pédagogiques :

- **sockets**

0.1 Github pour le TME

Les fichiers fournis sont a récupérer suivant la même procédure qu'au TME6 sous authentification github : https://classroom.github.com/a/ylvDA4uV.

Vous ferez un push de vos résultats sur ce dépôt pour soumettre votre travail.

1 Socket Client/Serveur

Question 1. En reprenant les questions du TD, implantez les classes `Socket` et `ServerSocket`. Testez les à l'aide des main client et serveur fournis.

Question 2. Implantez le `TCPServer` avec une version non bloquante de StartServer.

Question 3. A l'aide du Pool de thread développé en séance 4, et de l'adapteur construit à la fin du TD, modifier le serveur pour qu'il instancie un pool à la construction et délègue les communication avec le client au pool.

2 Exploitation du Serveur

On propose d'exploiter le travail réalisé pour écrire un serveur et un client simulant un service FTP minimal, réduit à 3 opérations :

- "LIST" : obtention de la liste des fichiers disponibles dans le répertoire administré par le serveur ;
- "UPLOAD" : téléchargement de fichier depuis le client vers le serveur ;
- "DOWNLOAD" : téléchargement de fichier depuis le serveur vers le client.

On utilisera bien sûr `opendir` et `readdir` pour parcourir le répertoire concerné.

Question 1. Un mini-serveur FTP

Dans cette question il s'agit d'écrire le serveur, qui reçoit en ligne de commande le numéro de port où l'appeler, et le répertoire où entreposer les fichiers envoyés par les clients. Les connexions se feront en TCP. On s'appuiera sur le TCPServer développé en première partie. Exemple d'appel :

```
$PWD/bin/ftp_server 2000 /tmp &
```

Question 2. Un mini-client FTP

Le client prend sur la ligne de commande l'adresse IP du serveur et son numéro de port. Il s'y connecte immédiatement, et en cas de réussite rentre dans une boucle de lecture ligne par ligne des requêtes de l'utilisateur au clavier.

Pour chaque ligne, il vérifie que la requête demandée est bien l'une des trois indiquée dans l'énoncé, si oui l'envoie au serveur, attend sa réponse et l'affiche dans le flux de sortie.

3 Sockets UDP

3.1 Serveur

On souhaite réaliser un mini-serveur d'environnement qui communique par UDP sur un port dont le numéro est donné sur la ligne de commande du serveur à son démarrage. Ce qu'on appelle ici un

environnement est une liste de couples identificateur, valeur. Les identificateurs et les valeurs sont de type chaîne de caractères.

Le serveur reconnait deux opérations :

- `set(identificateur, valeur)` : pour fixer la valeur d'un identificateur ;
- `get(identificateur)` : pour obtenir la valeur d'un identificateur.

On pourra simplement s'appuyer sur un `unordered_map` pour le stockage.

Exemple d'appel :

```
$PWD/bin/env_serveur 2001 &
```

3.2 Client

Le client du programme précédent prend sur la ligne de commande l'adresse du serveur et son port. Ensuite il lit sur le flux d'entrée une suite de requêtes dont chacune doit avoir l'une des formes suivantes :

- S identificateur valeur, pour un Set ;
- G identificateur, pour un Get.
- Q, pour quitter le client.

Le client construit le message correspondant à la requête et envoie ce message au serveur en utilisant une socket et le protocole UDP. Il attend alors la réponse du serveur et l'envoie sur le flux de sortie. Exemple d'appel :

```
echo "S USER moi;G USER;Q;"|tr ";" "\n"|$PWD/bin/env_client 127.0.0.1 2001
```

4 MultiCast UDP

Réalisez un programme qui permet d'échanger des messages ligne par ligne avec d'autres processus en communiquant par Multicast.

Le programme prend sur la ligne de commande :

- l'adresse IP Multicast où la conversation a lieu ;
- le numéro du port sur lequel la conversation a lieu
- le nom (ou pseudonyme) utilisé dans la conversation

Une fois lancé, le programme affiche tous les messages envoyés par d'autres utilisateurs et permet parallèlement d'envoyer des messages pour participer à la conversation. Il utilisera un Thread pour écrire et un autre pour lire. Exemple d'appel :

```
$PWD/bin/mychat 225.0.0.10 2001 $USER
```