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 champs d'une adresse de socket du domaine internet :

- `sin_family` : discriminant de type d'adresse, prend la valeur `AF_INET`,
- `sin_port` : le port au format internet,
- `sin_addr` : l'adresse IPv4; ce dernier champ n'est accessible que sur le type `sockaddr_in`, mais pas sur le type `sockaddr` mentionné dans la plupart de l'api, qui supporte d'autres domaines que internet. Cela induit des (cast) un peu maladroits mais nécessaires dans le code.

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.

> Corrigé à la fin de Socket.cpp dans le corrigé de la question suivante.
>
> The getnameinfo function is the ANSI version of a function that provides protocol-independent name resolution. The getnameinfo function is used to translate the contents of a socket address structure to a node name and/or a service name.
>
> For IPv6 and IPv4 protocols, Name resolution can be by the Domain Name System (DNS), a local hosts file, or by other naming mechanisms. This function can be used to determine the host name for an IPv4 or IPv6 address, a reverse DNS lookup, or determine the service name for a port number. The getnameinfo function can also be used to convert an IP address or a port number in a sockaddr structure to an ANSI string. This function can also be used to determine the IP address for a host name.
>
> Les autres c'est des bêtes fonctions de conversions de format/string/host/network. Celle là c'est spécial.

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_ */
```

Donc niveau résolution des noms, on a deux versions, une qui part d'un string humain et qui va vers un format *sockaddr_in* à l'aide *getaddrinfo*, une autre fonction qui utilise le DNS en background.

The getaddrinfo function is the ANSI version of a function that provides protocol-independent translation from host name to address. The getaddrinfo function returns results for the NS_DNS namespace. The getaddrinfo function aggregates all responses if more than one namespace provider returns information. For use with the IPv6 and IPv4 protocol, name resolution can be by the Domain Name System (DNS), a local hosts file, or by other naming mechanisms for the NS_DNS namespace.

Ca rend une chaine de résultats, d'où l'API un peu maladroite en C pour le nettoyer à la fin même si dans l'exemple on ne consulte que la première entrée.

Une fois qu'on a une ipV4, à part les cast pour l'API ça se passe assez directement. On remplit les champs et connect.

Socket.cpp

```
#include "Socket.h"
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <cstring>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <iostream>

namespace pr {


void Socket::close() {
      if (fd != -1) {
            shutdown(fd,1);
            ::close(fd);
      }
}

void Socket::connect(const std::string & host, int port) {
      // but : resoudre l'addresse "humains" vers une addresse format machine
```

```
      struct addrinfo * addr;
      /* Remplir la structure dest */
      // les choses à null ne servent pas ici
      if (getaddrinfo(host.c_str(), /* service*/ nullptr, /* hints*/ nullptr, &addr) != 0) {
            perror("getaddrinfo");
            return;
      }
      // champ "data" ai_addr (une addresse en reponse) du premier chainon de addrinfo
      // dedans on trouve la chose utile : une addresse de socket, qu'il faut encore
      // fixer vers un pointeur d'adresse de socket du domaine AF_INET
      // on doit faire un (down)cast pour retyper sockaddr en sockaddr_in
      in_addr ipv4 = ((struct sockaddr_in *) addr->ai_addr)->sin_addr;
      // getaddrinfo alloue une liste, on desalloue poliment
      freeaddrinfo(addr);
      connect(ipv4, port);
}

void Socket::connect(in_addr ipv4, int port) {
      // une addresse spécialisée pour internet
      sockaddr_in dest;
      dest.sin_family = AF_INET; // discriminant, aide à retrouver le type concret dans un switch case
      dest.sin_addr = ipv4;
      dest.sin_port = htons(port); // host to network

      // create socket
      fd = socket(AF_INET,SOCK_STREAM, /* protocole TCP par defaut sur SOCK_STREAM*/ 0);
      if (fd <0) {
            perror("socket");
            return;
      }

      // ici on doit upcast de sockaddr_in (spécialisé) vers sockaddr plus général
      // en OO, pas de syntaxe, en C il faut un cast
      // on passe la taille réelle de l'objet sockaddr spécialisé
      if (::connect(fd,(struct sockaddr *) &dest, sizeof dest) <0) {
            perror("connect");
            ::close(fd);
            fd=-1;
            return;
      }
}

// sockaddr_in est présent dans l'API a plein d'endroits, e.g. expediteur dans un accept
// objectif : de l'addresse format machine vers du lisible humain
std::ostream & operator<< (std::ostream & os, struct sockaddr_in * addr) {
      char hname [1024];
      // obtient à partir de l'addresse machine le nom d'hote
      if (getnameinfo((struct sockaddr *)addr,sizeof *addr, hname, 1024, nullptr, 0, 0) == 0) {
            os << '"' << hname << '"' << " ";
      }
      // inet_ntoa : produit la chain "127.0.0.1" à partir d'une addresse ipv4 sur 4 octets
      // ntohs : network to host
      os << inet_ntoa(addr->sin_addr) << ":" << ntohs(addr->sin_port) << std::endl;
      return os;
```

```
}


}
```

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'adresse du client qui vient de se connecter à chaque connection (dans accept).

Donc la création d'un serveur TCP nécessite deux étapes de construction supplémentaires de la socket :

1. bind = déclarer la socket sur le réseau/lui donner une interface système avec la carte réseau qui est aussi un périphérique.
2. listen pour mettre en place une file d'attente de connection sur le port indiqué qui sera gérée par le système vu que l'on n'est pas toujours élu sur le CPU.

A ce stade, on à le droit de faire accept pour consommer un des clients dans cette file d'attente, s'il y en a (opération bloquante sinon). Ca rend une socket de connection bidirectionelle où ce que l'on écrit est lu par le client est réciproquement en mode flux.

ServerSocket.h

```
#include "ServerSocket.h"
#include <cstring>
#include <unistd.h>
#include <iostream>

namespace pr {

```

```
ServerSocket::ServerSocket(int port) :socketfd(-1) {

      // create socket
      int fd = socket(AF_INET,SOCK_STREAM,0);
      if (fd == -1) {
            perror("create socket");
            return;
      }

      // bind
      struct sockaddr_in sin; /* Nom de la socket de connexion */

      memset(&sin, 0, sizeof(sin)); // utile ?
      sin.sin_family = AF_INET;
      sin.sin_addr.s_addr = htonl(INADDR_ANY); // on attend sur n'importe quelle
          interface de la machine
      sin.sin_port = htons(port); // host to network

      /* nommage, meme probleme de typage sockaddr que dans la Socket */
      if (bind(fd, (struct sockaddr *) &sin, sizeof(sin)) < 0) {
            perror("bind");
            // on veut le close de unistd, pas le close membre de cette classe
            ::close(fd);
            return;
      }

      // listen
      if (listen(fd, 50) < 0) {
            perror("listen");
            ::close(fd);
            return;
      }

      // success !
      socketfd = fd;
}

Socket ServerSocket::accept() {
      struct sockaddr_in exp;
      socklen_t len = sizeof(exp);
      // on qualifie sinon le compilo rale
      int scom = ::accept(socketfd, (struct sockaddr *) &exp, &len);
      if (scom < 0) {
            perror("accept");
      } else {
            // en appui sur operator << développé plus haut
            std::cout << "Accepted connection from " << &exp << std::endl;
      }
      return scom;
}

void ServerSocket::close() {
      if (socketfd != -1) {
            ::close(socketfd);
      }
}

}
```

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.

Des versions de plus en plus riche dans le corrigé. A la fin on fait une boucle de dix émissions/réponse. Le serveur correspondant sort quand on lui envoie 0.

client.cpp

```cpp
#include "ServerSocket.h"
#include <iostream>
#include <unistd.h>
#include <string>


int main00() {
      pr::Socket sock;
      sock.connect("localhost", 1664);
      int N=42;
      write(sock.getFD(),&N,sizeof(int));
      read(sock.getFD(),&N,sizeof(int));
      std::cout << N << std::endl;
      return 0;
}


// avec controle
int main0() {

      pr::Socket sock;

      sock.connect("localhost", 1664);

      if (sock.isOpen()) {
            int fd = sock.getFD();
            int i = 10;
            ssize_t msz = sizeof(int);
            if (write(fd, &i, msz) < msz) {
                  perror("write");
            }
            std::cout << "envoyé =" << i << std::endl;
            int lu;
            auto nblu = read(fd, &lu, msz);
            if (nblu == 0) {
                  std::cout << "Fin connexion par serveur" << std::endl;
            } else if (nblu < msz) {
                  perror("read");
            }
            std::cout << "lu =" << lu << std::endl;
      }

      return 0;
}


// avec une boucle, on attend un 0
int main() {

      pr::Socket sock;

      sock.connect("localhost", 1664);
```

```
	if (sock.isOpen()) {
		int fd = sock.getFD();

		ssize_t msz = sizeof(int);
		for (int i = 10; i >= 0; i--) {
			if (write(fd, &i, msz) < msz) {
				perror("write");
				break;
			}
			std::cout << "envoyé =" << i << std::endl;

			int lu;
			auto nblu = read(fd, &lu, msz);
			if (nblu == 0) {
				std::cout << "Fin connexion par serveur" << std::endl;
				break;
			} else if (nblu < msz) {
				perror("read");
				break;
			}
			std::cout << "lu =" << lu << std::endl;
		}
	}

	return 0;
}
```

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.

Des versions de plus en plus riche dans le corrigé. A la fin on fait une boucle de dix émissions/réponse. Le serveur correspondant sort quand on lui envoie 0.

server.cpp

```
#include "ServerSocket.h"
#include <iostream>
#include <unistd.h>

int main00() {
	pr::ServerSocket ss(1664);

	while (1) {
		pr::Socket sc = ss.accept();

		int fd = sc.getFD();

		int lu;
		read(fd, &lu, sizeof(int));
		std::cout << "lu =" << lu << std::endl;
		lu++;
		write(fd, &lu, sizeof(int));
		sc.close();
	}
	ss.close();
```

```
	return 0;
}

int main() {
	pr::ServerSocket ss(1664);

	while (1) {
		pr::Socket sc = ss.accept();

		int fd = sc.getFD();

		ssize_t msz = sizeof(int);
		while (1) {
			int lu;
			auto nblu = read(fd, &lu, msz);
			if (nblu == 0) {
				std::cout << "Fin connexion par client" << std::endl;
				break;
			} else if (nblu < msz) {
				perror("read");
				break;
			}
			std::cout << "lu =" << lu << std::endl;

			if (lu == 0) {
				break;
			}
			lu++;
			if (write(fd, &lu, msz) < msz) {
				perror("write");
				break;
			}
			std::cout << "envoyé =" << lu << std::endl;
		}
		sc.close();
	}

	ss.close();
	return 0;
}
```

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*.

TCPServer.h

```
#ifndef SRC_TCPSERVER_H_
#define SRC_TCPSERVER_H_

#include <thread>
#include "ServerSocket.h"
#include "ConnectionHandler.h"

namespace pr {

// un serveur TCP, la gestion des connections est déléguée
class TCPServer {
      std::thread * waitT; // le thread d'attente de connexion s'il est instancie
      ServerSocket * ss; // la socket d'attente si elle est instanciee
      ConnectionHandler * handler; // le gestionnaire de session passe a la cstru
      int killpipe; // introduit a la fin, pour traiter le stopServer
public :
      TCPServer(ConnectionHandler * handler): waitT(nullptr),ss(nullptr),handler(handler
          ),killpipe(-1) {}
      // Tente de creer une socket d'attente sur le port donné
      // engendre un thread d'attente de connections
      // des variantes au fil des questions
      bool startServer0 (int port);
      bool startServer1 (int port);
      bool startServer2 (int port);

      // ferme la socket
      // ce qui interrompt le thread d'attente de connection
      void stopServer () ;
};

} // ns pr

#endif /* SRC_TCPSERVER_H_ */
```

Et la version 0 de startServer.

TCPServer.cpp

```
#include "TCPServer.h"
#include <sys/select.h>
#include <iostream>
#include <unistd.h>
#include <algorithm>
#include "JobHandler.h"
```

```

namespace pr {

// Version basique : mobilise le client ( Question 6 )
bool TCPServer::startServer0(int port) {
	ss = new ServerSocket(port);
	if (ss->isOpen()) {
		while (1) {
			std::cout << "En attente sur accept" << std::endl;
			auto sc = ss->accept();
			if (sc.isOpen()) {
				auto copy = handler->clone();
				copy->handleConnection(sc);
				delete copy;
			}
		}
		// inaccessible
		return true;
	}
	return false;
}

// version asynchrone permet de demarrer le serveur sans bloquer (Question 7)
bool TCPServer::startServer1(int port) {
	if (waitT == nullptr) {
		ss = new ServerSocket(port);
		if (ss->isOpen()) {
			waitT = new std::thread([](TCPServer * serv){
				while (1) {
					std::cout << "En attente sur accept" << std::endl;
					auto sc = serv->ss->accept();
					std::cout << "Après accept" << std::endl;
					if (sc.isOpen()) {
						auto copy = serv->handler->clone();
						copy->handleConnection(sc);
						delete copy;
					}
				}
			},this);
			return true;
		}
	}
	return false;
}

// version asynchrone, mais ou on peut kill le serveur, fait avec un pipe (Question 8)
bool TCPServer::startServer2 (int port) {
	if (waitT == nullptr) {
		ss = new ServerSocket(port);
		if (ss->isOpen()) {
			int pipefd[2];
			if ( pipe(pipefd) < 0) {
				perror("pipe");
				return false;
			}
			waitT = new std::thread([](TCPServer * serv, int readpipe){
				while (1) {
					// Construire un set pour le select (atente simultanee)
					fd_set read;
```

```
                              FD_ZERO(&read);                                                    66
                              FD_SET(readpipe, &read);                                           67
                              FD_SET(serv->ss->getFD(), &read);                                  68
                              // premier argument = plus grand fildescriptor dans un des         69
                                   set + 1
                              // on peut comprendre que fd_set est un genre de bitset,           70
                                  on donne sa taille ici.
                              select(std::max(readpipe, serv->ss->getFD()) +1, &read, /*      71
                                  write*/nullptr, /*except*/nullptr, /*timeout*/nullptr
                                  );
                                                                                                 72
                              // au retour du select, read est modifié pour contenir le          73
                                  ou les fd qui nous ont reveillé
                              if (FD_ISSET(readpipe,&read)) {                                    74
                                    std::cout << "asked to quit" << std::endl;                   75
                                    return;                                                      76
                              }                                                                  77
                              // sinon ben c'est la socket qui nous a réveillé : un              78
                                  client s'est pointé
                              auto sc = serv->ss->accept();                                      79
                              if (sc.isOpen()) {                                                 80
                                    auto copy = serv->handler->clone();                          81
                                    copy->handleConnection(sc);                                  82
                                    // En version Pool de thread                                 83
                                    // pool.submit(new JobHandler(copy,sc));                     84
                                    // et on enlève le delete (fait par JobHandler)              85
                                    delete copy;                                                 86
                              }                                                                  87
                        }                                                                        88
                  },this, pipefd[0]);                                                            89
                  killpipe = pipefd[1];                                                          90
                  return true;                                                                   91
            }                                                                                    92
      }                                                                                          93
      return false;                                                                              94
}                                                                                                95
                                                                                                 96
                                                                                                 97
                                                                                                 98
void TCPServer::stopServer() {                                                                   99
      if (waitT != nullptr) {                                                                    100
            int n = 1;                                                                           101
            write(killpipe,&n,sizeof(int));                                                      102
            waitT->join();                                                                       103
            delete waitT;                                                                        104
            delete ss;                                                                           105
            close(killpipe);                                                                     106
            killpipe = -1;                                                                       107
            waitT = nullptr;                                                                     108
      }                                                                                          109
}                                                                                                110
                                                                                                 111
}                                                                                                112
```

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.

Cf version 1 de startServer.

En gros on lance un nouveau thread pour chaque connection, on lui passe au minimum : le handler qu'il doit clone, la socket d'attente de connection. Ici je lui passe brutalement le Server entier.

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

Une vraie question plus subtile qu'il n'y parait. Le problème : le thread qu'on souhaite arrêter est bloqué sur le accept.

On n'a pas de variante de accept avec un timeout ou autre.

Option 0 : provoquer un EINTR sur accept, i.e. interrompre le thread. Ca signifie utiliser un signal. MAIS, on est en multi-thread. Du coup pas de garantie (du tout) que le signal soit délivré à un thread en particulier au sein du processus.

Option 1 : utiliser une multi-attente avec une variante de select/poll. select permet d'attendre plusieurs descripteurs avec en plus un timeout. Quand une socket d'attente de connection (état listen) reçoit une demande cela déclenche un état "read".

On peut donc boucler sur un select avec un timeout, et périodiquement consulter une variable membre de Server qui aura été mise à 0 pour signifier qu'on en a marre. Pas génial, attente semi active tout ça.

Option 2 : on ajoute un autre filedescriptor à la liste passée à select, crée pour l'occasion. Par exemple un petit pipe, quand on écrit dedans (n'importe quoi) ca veut dire qu'il faut quitter. C'est cette option qui est fait dans le corrigé.

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_ */
```

Prépare clairement la question sur l'utilisation d'un pool de thread.

JobHandler.h

```
#ifndef SRC_JOBHANDLER_H_
#define SRC_JOBHANDLER_H_

#include "Job.h"
#include "ConnectionHandler.h"

namespace pr {

class JobHandler : public Job {
      ConnectionHandler * ch;
      Socket s;
public :
```

```
      JobHandler(ConnectionHandler * ch, Socket s):ch(ch),s(s) {}
      void run() { ch->handleConnection(s); delete ch ;}
};

}

#endif /* SRC_JOBHANDLER_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
```

TD 8 : Appels de Procédures Distantes

Objectifs pédagogiques :

- **Proxy Distant, Stub serveur et client**
- **Remote Procedure Call**
- **Sérialisation**

Introduction

Dans ce TD, nous allons étudier en appui sur l'API développée en séance 7 (Socket, ServerSocket, TCPServer) la réalisation d'un mécanisme permettant l'invocation de services distants. On s'appuiera sur ProtoBuf plutôt que de développer un protocole ad-hoc.

On considère un exemple de forum de messages avec un historique.

On introduit les interfaces suivantes :

IChatRoom.h

```
#ifndef SRC_ICHATROOM_H_
#define SRC_ICHATROOM_H_

#include <string>
#include <vector>

namespace pr {

class ChatMessage {
      std::string author;
      std::string message;
public :
      ChatMessage (const std::string & author, const std::string & msg):author(author),
          message(msg) {};
      const std::string & getAuthor() const {return author; }
      const std::string & getMessage() const {return message; }
};

class IChatter {
public :
      virtual std::string getName() const = 0;
      virtual void messageReceived (ChatMessage msg) = 0;
      virtual ~IChatter() {}
};

class IChatRoom {
public :
      virtual std::string getSubject() const = 0;
      virtual std::vector<ChatMessage> getHistory() const = 0;
      virtual bool posterMessage(const ChatMessage & msg) = 0;
      virtual bool joinChatRoom (IChatter * chatter) = 0;
      virtual bool leaveChatRoom (IChatter * chatter) = 0;
      virtual size_t nbParticipants() const =0;
      virtual ~IChatRoom() {}
};

}

#endif /* SRC_ICHATROOM_H_ */
```

Ainsi que les implantations textuelles triviales suivantes :

TextChatRoom.h

```
#ifndef SRC_TEXTCHAT_H_
#define SRC_TEXTCHAT_H_

#include "IChatRoom.h"

#include <iostream>
#include <algorithm>

namespace pr {

class TextChatter : public IChatter {
      std::string name;
public :
      TextChatter (const std::string & name):name(name){}
      std::string getName() const { return name ; }
      void messageReceived (ChatMessage msg) { std::cout << "(chez " << name << ") de : " <<
           msg.getAuthor() << " > " << msg.getMessage() << std::endl; }
};

class TextChatRoom : public IChatRoom {
      std::string subject;
      std::vector<ChatMessage> history;
      std::vector<IChatter *> participants;
public :
       TextChatRoom(const std::string & subject) : subject(subject) {}
       std::string getSubject() const { return subject; }
       std::vector<ChatMessage> getHistory() const { return history ; }
       bool posterMessage(const ChatMessage & msg) {
             history.push_back(msg);
             for (auto c : participants) {
                   c->messageReceived(msg);
             }
             return true;
       }
       bool joinChatRoom (IChatter * chatter) {
             participants.push_back(chatter);
             return true;
       }
       bool leaveChatRoom (IChatter * chatter) {
             auto it = std::find(participants.begin(),participants.end(),chatter);
             if (it != participants.end()) {
                   participants.erase(it);
                   return true;
             }
             return false;
       }
       size_t nbParticipants() const { return participants.size(); }
};

}

#endif
```

Un exemple de test simple est fourni :

chatbasic.cpp

```
#include "TextChatRoom.h"

using namespace pr;
```

```
using namespace std;


int main () {

      IChatRoom * cr = new TextChatRoom ("C++");
      TextChatter alice("alice");
      TextChatter bob("bob");
      cr->joinChatRoom(&alice);
      cr->joinChatRoom(&bob);

      cr->posterMessage({"bob","salut"});
      cr->posterMessage({"alice","hello"});
      cr->posterMessage({"alice","bye"});

      cr->leaveChatRoom(&alice);
      cr->posterMessage({"bob","reste !"});

      cr->leaveChatRoom(&bob);
      delete cr;
}
```

Son exécution produit l'affichage :

```
(chez alice) de : bob > salut
(chez bob) de : bob > salut
(chez alice) de : alice > hello
(chez bob) de : alice > hello
(chez alice) de : alice > bye
(chez bob) de : alice > bye
(chez bob) de : bob > reste !
```

1 Mise en Place

1.1 Serveur

Notre objectif est de permettre d'accéder aux instances d'une `IChatRoom`. On considère dans cette question la mise en place d'un service dédié à l'accès à une instance de classe, indépendamment des opérations offertes.

Notre architecture actuelle (fin TD7) côté `TCPServer` a abouti à une interface `ConnectionHandler`, munie de `clone` et `handleConnection`.

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_ */
```

L'objectif est de réaliser un `ConnectionHandler` qui permette de jouer le rôle de "squelette" (skeleton) ou stub côté serveur. Le squelette offre les services de la classe au réseau.

- A la construction on fixe le port et on lui passe l'instance d'objet côté serveur (le *sujet*) que l'on souhaite exposer au réseau
- Une fois la connection en place, on se met en attente réseau de *requetes* du client
- Une requête client est constituée d'un identifiant pour le service (représentant le nom de la méthode que l'on souhaite invoquer), suivie par les arguments utilise à la requête, dont on peut déduire le typage d'après le service invoqué
- On ajoutera un code de requête particulier QUIT pour fermer la connection proprement.
- Une fois les arguments et la méthode à invoquer déterminés, on invoque la méthode sur le *sujet*.
- On envoie la réponse obtenue au client, sous une forme homogène à sa requête, d'abord un identifiant de service, puis la valeur de retour.

Question 1. Ecrivez une classe ChatRoomServer; on lui passe à la construction un numéro de port où écouter les demandes de connexion, et un `IChatRoom *` désignant l'instance de ChatRoom que l'on souhaite exposer au réseau. On traite dans un premier temps uniquement les opérations `nbParticipants` et `getSubject`.

Pour faciliter le code, j'ai jouté dans la classe Socket des opérations pour lire et écrire 1. des entiers 2. des String.

Dans la classe Socket.h

```

      int readInt ();
      void writeInt (int n);

      void writeString (const std::string & s);
      std::string readString ();
```

Socket.cpp

```
      }
}

int Socket::readInt () {
      int n=0;
      if ( read(fd,&n,sizeof(n)) < sizeof(n)) {
            perror("readI");
            ::close(fd);
            fd = -1;
      }
      return n;
}

void Socket::writeInt (int n) {
      if ( write(fd,&n,sizeof(n)) < sizeof(n)) {
            perror("writeI");
            ::close(fd);
            fd = -1;
```

```
      }
}

void Socket::writeString (const std::string & s) {
      size_t sz = s.length();
      writeInt(sz);
      if ( write(fd,s.data(),sz) < sz) {
            perror("writeS");
            ::close(fd);
            fd = -1;
      }
}

std::string Socket::readString () {
      size_t sz = readInt();
```

ChatServer.h

```
#ifndef SRC_CHATSERVER_H_
#define SRC_CHATSERVER_H_

#include "IChatRoom.h"
#include "TCPServer.h"

namespace pr {

class ChatServer {
      TCPServer server;
public:
      ChatServer(IChatRoom * cr, int port);
      ~ChatServer();
};

}

#endif /* SRC_CHATSERVER_H_ */
```

ChatServer.cpp

```
#include "ChatServer.h"
#include <unistd.h>
#include <string>
#include <iostream>

namespace pr {


class ChatRoomCH : public ConnectionHandler {
      IChatRoom * cr;
public :
      ChatRoomCH (IChatRoom * cr):cr(cr){}
      void handleConnection(Socket s) {
            while (1) {
                  int req = s.readInt();
                  switch (req) {
                        case 0 :
                        { // get subject
                              // no args
```

```
                        std::string subject = cr->getSubject();
                        s.writeString(subject);
                        break;
                    }
                    case 1 :
                    {
                        size_t sz = cr->nbParticipants();
                        s.writeInt(sz);
                        break;
                    }
                    case 2 :
                    {
                        std::cout << "End from client." << std::endl;
                        return;
                    }
                    default :
                        std::cerr << "unknown message " << req << std::endl;
                    }
            }
        }
        // une copie identique
        ConnectionHandler * clone() const {
            return new ChatRoomCH(*this);
        }
};

ChatServer::ChatServer(IChatRoom * cr, int port) :server(new ChatRoomCH(cr)) {
        server.startServer(port);
}

ChatServer::~ChatServer() {
        server.stopServer();
}

}
```

- Rien de très violent, le serveur attend la requête, il it disons un “int“ dans le flux. Ensuite switch sur la valeur lue.
- si c’est nbParticipants on écrit l’int qui va bien
- si c’est subject, on écrit la longueur de string puis le contenu de la string.

Tout ça en dur, i.e. à coup de read/write sur filedescriptor nu. Savoir envoyer une string sera utile dans la suite, on sépare donc “sendString” dans une fonction.

Question 2. Expliquez comment traiter les effets secondaires liés à l’utilisation de la chat room dans ce contexte de serveur multi-thread.

Ben, oui, le serveur TCP est multi-client géré avec des threads. . . donc il faut protéger le sujet.

Le clone qui est fait à chaque connection client aboutit à plusieurs instances de ConnectionHandler concrets qui pointent le même objet serveur.

Solution 1. Ajouter des mutex dans la classe TextChatRoom directement.

Solution 2. Ecrire un décorateur pour IChatRoom muni d’un mutex, qui assure l’exclusion mutuelle. On passe une instance décorée quand on construit le serveur.

Le corrigé fait un decorateur.

MTChatRoom.h

```
#ifndef SRC_MTCHATROOM_H_
#define SRC_MTCHATROOM_H_

#include "IChatRoom.h"
#include <memory>
#include <mutex>

namespace pr {

class MTChatRoom : public IChatRoom {
      IChatRoom * deco;
      mutable std::mutex mut;
public :
      MTChatRoom(IChatRoom * cr) : deco(cr) {};
      std::string getSubject() const {
            std::unique_lock<std::mutex> l(mut);
            return deco->getSubject();
      }
      std::vector<ChatMessage> getHistory() const {
            std::unique_lock<std::mutex> l(mut);
            return deco->getHistory();
      }
      bool posterMessage(const ChatMessage & msg) {
            std::unique_lock<std::mutex> l(mut);
            return deco->posterMessage(msg);
      }
      bool joinChatRoom (IChatter * chatter) {
            std::unique_lock<std::mutex> l(mut);
            return deco->joinChatRoom(chatter);
      }
      bool leaveChatRoom (IChatter * chatter) {
            std::unique_lock<std::mutex> l(mut);
            return deco->leaveChatRoom(chatter);
      }
      virtual size_t nbParticipants() const {
            std::unique_lock<std::mutex> l(mut);
            return deco->nbParticipants();
      }
};

}


#endif /* SRC_MTCHATROOM_H_ */
```

Question 3. Proposez un main pour le serveur.

chatserver.cpp

```
#include "TextChatRoom.h"
#include "ChatServer.h"
#include "MTChatRoom.h"
#include <iostream>
#include <unistd.h>
#include <csignal>
#include <cstring>
```

```cpp


int main() {

      // SIGPIPE on broken connections sucks in multi-thread context.
      // Le problème : on ne peut pas fiablement savoir quel thread a déclenché le sigpipe,
      // ni le rattrapper correctement avec ce bon thread.
      // En ignorant SIGPIPE, les read/write sur socket fermée vont rendre -1 + errno = EPIPE
      // mais pas de signaux engendrés
      // Une alternative serait d'utiliser send (plutot que write) avec un flag MSG_NOSIGNAL
      struct sigaction act;
      memset(& act,0,sizeof(act));
      act.sa_handler = SIG_IGN;
      sigaction(SIGPIPE,&act, nullptr);

      // la version avec signal ne suffit pas de façon durable : le handler est reset à SIG_DFL (sur ma machine)
      // ce comportement varie entre système et versions
      //signal(SIGPIPE,SIG_IGN);

      pr::TextChatRoom tcr ("C++");
      pr::MTChatRoom mcr (&tcr);
      pr::TextChatter schat("Echo Server");
      mcr.posterMessage({"serveur","début session"});
      tcr.joinChatRoom(&schat);
      {
            pr::ChatServer server(&mcr,1664);

            // attend entree sur la console
            std::string s ;
            std::cin >> s ;

            std::cout << "Début fin du serveur." << std::endl ;

            // on quit = dtor du serveur
      }
      tcr.posterMessage({"serveur","fin session"});
      std::cout << "Ok fin du serveur." << std::endl;
      tcr.leaveChatRoom(&schat);

      return 0;
}
```

On note la décoration du tcr nu.

Ici on attends de quit si on appuie sur entrée. Le port et le sujet de conversation pourrait bien évidemment être pris sur la ligne de commande.

1.2 Client

Symétriquement, on souhaite réaliser une classe ChatRoomProxy que l'on peut instancier côté client, et qui cache le réseau. On s'en sert comme d'une `IChatRoom` ordinaire, et elle implémente cette interface, mais le comportement est réalisé à travers le réseau.

Question 4. Ecrivez une classe ChatRoomProxy; on lui passe à la construction un numéro de port et un nom d'hôte (ou une IP) hébergeant le serveur. On traite dans un premier temps uniquement

les opérations `nbParticipants` et `getSubject`.

ChatRoomProxy.h

```h
/*
 * ChatRoomProxy.h
 *
 * Created on: Dec 4, 2018
 * Author: ythierry
 */

#ifndef SRC_CHATROOMPROXY_H_
#define SRC_CHATROOMPROXY_H_

#include "IChatRoom.h"
#include "ChatterServer.h"
#include "Socket.h"

namespace pr {

class ChatRoomProxy: public IChatRoom {
	mutable Socket sock;
	ChatterServer * serv;
public:
	ChatRoomProxy(const std::string & host, int port);
	std::string getSubject() const;
	std::vector<ChatMessage> getHistory() const ;
	bool posterMessage(const ChatMessage & msg) ;
	bool joinChatRoom (IChatter * chatter) ;
	bool leaveChatRoom (IChatter * chatter) ;
	size_t nbParticipants() const;
	~ChatRoomProxy();
};

} /* namespace pr */

#endif /* SRC_CHATROOMPROXY_H_ */
```

ChatRoomProxy.cpp

```cpp
#include "ChatRoomProxy.h"

namespace pr {

ChatRoomProxy::ChatRoomProxy(const std::string & host, int port) {
	sock.connect(host,port);
}

std::string
ChatRoomProxy::getSubject() const {
	sock.writeInt(0); // 0 = getSubject
	auto s = sock.readString();
	return s;
}

size_t
ChatRoomProxy::nbParticipants() const {
	sock.writeInt(1); // 1 = nbParticipant
	auto s = sock.readInt();
```

```
      return s;
}

ChatRoomProxy::~ChatRoomProxy() {
      sock.writeInt(2); // 2 = QUIT
      sock.close();
}

std::vector<ChatMessage> ChatRoomProxy::getHistory() const { return {} ;}
bool ChatRoomProxy::posterMessage(const ChatMessage & msg) {return true;}
bool ChatRoomProxy::joinChatRoom (IChatter * chatter) { return true; }
bool ChatRoomProxy::leaveChatRoom (IChatter * chatter){ return true; }


} /* namespace pr */
```

On implante donc IChatRoom dans le cadre du DP Proxy.
On fait du code symétrique au serveur, i.e. envoyer le bon code de message, lire les réponses.

Question 5. Proposez un main client simple qui utilise ce qui a été construit.

clientchat.cpp

```
#include "ChatRoomProxy.h"
#include "TextChatRoom.h"
#include <iostream>
#include <unistd.h>
#include <string>
#include <csignal>

using namespace pr;

int main(int argc, char ** argv) {

      std::string myname = "bob";
      if (argc > 1) {
            myname = argv[1];
      }

      // SIGPIPE on broken connections sucks in multi-thread context.
      signal(SIGPIPE,SIG_IGN);


      pr::ChatRoomProxy cr("localhost", 1664);
      std::cout << "Sujet =" << cr.getSubject();
      std::cout << "NbParticipants =" << cr.nbParticipants() << std::endl;
      std::cout << "History =" << std::endl;
      for (auto & h : cr.getHistory()) {
            std::cout << h.getAuthor() << "> " << h.getMessage() << std::endl;
      }
      std::cout << std::endl;
      cr.posterMessage(ChatMessage(myname,"Coucou"));

      TextChatter me (myname);
      cr.joinChatRoom(&me);

      cr.posterMessage(ChatMessage(myname,"Coucou2"));
```

```
	std::cout << "Press q to quit" << std::endl;
	while (1) {
		// attend entree sur la console
		std::string s ;
		std::getline(std::cin, s);
		if (s=="q") {
			std::cout << "Quitting" << std::endl;
			cr.leaveChatRoom(&me);
			break;
		} else {
			cr.posterMessage(ChatMessage(myname,s));
		}
	}

	return 0;
}
```

2 Protocoles

Bien que les données véhiculées soient relativement simples, on voit déjà sur cet exemple que plus les données à échanger sont complexes, plus il va falloir de code dans le client et le serveur, pour décoder et encoder les trames réseau. Il est important que ce code soit synchronisé, ou du moins compatible. Il n'est pas rare que Client et Serveur évoluent à des vitesses de développement différentes, ce qui pose des problèmes de compatibilité au niveau protocole. Par exemple, on peut imaginer qu'on ajoute au bout d'un moment un timestamp sur les messages, un vieux client peut-il encore intéragir avec un serveur récent ?

Ces problèmes peuvent être traités à l'aide d'outillage dédié à la sérialisation comme ProtoBuf.

Question 1. Définir des messages ProtoBuf supportant notre cas d'utilisation :

- ChatRoomRequest : un message muni d'un unique champ obligatoire prenant sa valeur dans une énumération avec une entrée par opération de la classe
- Pour chaque opération "op", un message "opArgs" qui porte les arguments de l'opération, et un message "opResponse" qui porte la réponse.
- On ignore à ce stade les opérations *join et leave* ayant des `IChatter *` dans la signature.

Donc c'est surtout la structure, et la nature des fichiers ProtoBuf.

On pourrait bien sûr factoriser les trames, mais là on fait plutôt un peu systématique en mode ce que génèrerait de façon systématique un outil comme gRPC.

On commente le codage d'une liste de ChatMessage. Le reste est assez trivial.

On rappelle comment s'en servir : protoc sur le fichier généère un .pb.cc/.pb.h, ensuite on include/link là dessus. cf Makefile.

chat.proto

```
# global LDFLAG = Linker flags
AM_LDFLAGS = -pthread -lprotobuf

chat.pb.cc : chat.proto
	protoc chat.proto --cpp_out ./
```

chat.proto

```
syntax ="proto3";

package pr;

message ChatMessagePB {
      string author = 1;
      string msgtext = 2;
}

message ChatRoomRequest {
      enum Request {
            SUBJECT =0;
            NBPARTICIPANTS=1;
            QUIT=2;
            HISTORY=3;
            POST=4;
            JOIN=5;
            LEAVE=6;
      };
      Request req = 1;
}

message GetHistoryResponse {
      repeated ChatMessagePB history = 1;
}

message GetSubjectResponse {
      string msg = 1;
}

message GetNbParticipantsResponse {
      int32 nbpart = 1;
}

message PosterMessageArgs {
      ChatMessagePB msg = 1;
}

message BoolResponse {
      bool response = 1;
}

message JoinArgs {
      string host = 1; // IPV4
      int32 port = 2;
}

message ChatterRequest {
      enum Request {
            GETNAME =0;
            RECVMESSAGE=1;
            QUIT=2;
      };
      Request req = 1;
}

message GetNameResponse {
      string msg = 1;
```

```
}

message RecvMessageArgs {
      ChatMessagePB msg = 1;
}
```

Question 2. Expliquez comment intégrer ces messages dans la solution ébauchée en partie I.

Donc côté serveur le code devient, lire une Request, switch sur la valeur d'enum "op", lire "opArgs". Effectuer la requête, puis construire un opResponse, sérialiser dans une string et envoyer la string.

ChatRoomProxy.cpp

```cpp
#include "ChatRoomProxy.h"
#include "chat.pb.h"
#include <unistd.h>

namespace pr {

ChatRoomProxy::ChatRoomProxy(const std::string & host, int port):serv(nullptr) {
      sock.connect(host,port);
}

bool writeRequest (Socket & s, ChatRoomRequest::Request req) {
      ChatRoomRequest crr;
      crr.set_req(req);
      s.writeMessage(crr);
      return true;
}

std::string
ChatRoomProxy::getSubject() const {
      writeRequest(sock,ChatRoomRequest::SUBJECT);
      GetSubjectResponse resp;
      sock.readMessage(resp);
      return resp.msg();
}

size_t
ChatRoomProxy::nbParticipants() const {
      writeRequest(sock,ChatRoomRequest::NBPARTICIPANTS);
      GetNbParticipantsResponse resp;
      sock.readMessage(resp);
      return resp.nbpart();
}

ChatRoomProxy::~ChatRoomProxy() {
      writeRequest(sock,ChatRoomRequest::QUIT);
      sock.close();
}

std::vector<ChatMessage> ChatRoomProxy::getHistory() const {
      writeRequest(sock,ChatRoomRequest::HISTORY);
      GetHistoryResponse resp;
      sock.readMessage(resp);
      std::vector<ChatMessage> ret;
      ret.reserve(resp.history_size());
```

```
      for (auto m : resp.history()) {
            ret.push_back(ChatMessage(m.author(),m.msgtext()));
      }
      return ret;
}

bool ChatRoomProxy::posterMessage(const ChatMessage & msg) {
      writeRequest(sock,ChatRoomRequest::POST);
      PosterMessageArgs args;
      args.mutable_msg()->set_author(msg.getAuthor());
      args.mutable_msg()->set_msgtext(msg.getMessage());
      sock.writeMessage(args);
      return true;
}
#include <linux/if_link.h>
#include <ifaddrs.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/types.h>

std::string getHostName() {
      struct ifaddrs *ifaddr;
      if (getifaddrs(&ifaddr) == -1) {
        perror("getifaddrs");
   }
      for (auto ifa = ifaddr; ifa != NULL; ifa = ifa->ifa_next)
         {
            if (ifa->ifa_addr == NULL)
               continue;


            if((strcmp(ifa->ifa_name,"lo")!=0)&&(ifa->ifa_addr->sa_family==AF_INET))
            {
                  char host[NI_MAXHOST];
                  int s=getnameinfo(ifa->ifa_addr,sizeof(struct sockaddr_in),host,
                      NI_MAXHOST, NULL, 0, NI_NUMERICHOST);
                  if (s != 0)
               {
                  perror("getnameinfo");
                  continue;
               }
               printf("Local IP Host : <%s>\n", host);

                  freeifaddrs(ifaddr);
                  return host;
            }
         }

      return "localhost";
}

bool ChatRoomProxy::joinChatRoom (IChatter * chatter) {
      writeRequest(sock,ChatRoomRequest::JOIN);

      // create local server
      if (serv == nullptr) {
            serv = new ChatterServer(chatter,0);
      }
      JoinArgs args;
```

```
      std::string hh = getHostName();
      args.set_host(hh);
      auto addr = serv->getAddress();
      args.set_port(ntohs(addr.sin_port));
      sock.writeMessage(args);
      std::cout << "Creating server for chatter " << args.host() <<":" << args.port() <<
           std::endl;
      return true;
}
bool ChatRoomProxy::leaveChatRoom (IChatter * chatter){ return true; }


} /* namespace pr */
```

ChatServer.cpp

```
#include "ChatServer.h"
#include "ChatterProxy.h"
#include "chat.pb.h"
#include <unistd.h>
#include <string>
#include <iostream>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>


namespace pr {



class ChatRoomCH : public ConnectionHandler {
      IChatRoom * cr;
      IChatter * proxy;
public :
      ChatRoomCH (IChatRoom * cr):cr(cr),proxy(nullptr){}
      void handleConnection(Socket s) {
            while (s.isOpen()) {
                  ChatRoomRequest crr;
                  s.readMessage(crr);
                  if (!s.isOpen()) {
                        break;
                  }
                  //std::cout << "read request type :" << crr.req()<<std::endl;
                  switch (crr.req()) {
                        case ChatRoomRequest::SUBJECT :
                        { // get subject
                              // no args
                              GetSubjectResponse resp ;
                              resp.set_msg(cr->getSubject());
                              s.writeMessage(resp);
                              break;
                        }
                        case ChatRoomRequest::NBPARTICIPANTS :
                        {
                              GetNbParticipantsResponse resp ;
                              resp.set_nbpart(cr->nbParticipants());
                              s.writeMessage(resp);
                              break;
```

```
                        }
                        case ChatRoomRequest::QUIT :
                        {
                              std::cout << "End from client." << std::endl;
                              return;
                        }
                        case ChatRoomRequest::HISTORY :
                        {
                              GetHistoryResponse resp;
                              for (auto & m : cr->getHistory()) {
                                    ChatMessagePB * mpb = resp.add_history();
                                    mpb->set_author(m.getAuthor());
                                    mpb->set_msgtext(m.getMessage());
                              }
                              s.writeMessage(resp);
                              break;
                        }
                        case ChatRoomRequest::POST :
                        {
                              PosterMessageArgs args;
                              s.readMessage(args);
                              cr->posterMessage(ChatMessage(args.msg().author(), args.
                                  msg().msgtext()));
                              break;
                        }
                        case ChatRoomRequest::JOIN :
                        {
                              JoinArgs args;
                              s.readMessage(args);
                              // create Proxy
                              if (proxy == nullptr) {
                                    std::cout << "Creating proxy on server for chatter "
                                         << args.host() <<":" << args.port() << std::
                                        endl;
                                    proxy = new ChatterProxy(args.host(),args.port());
                                    cr->joinChatRoom(proxy);
                              }
                              break;
                        }
                        case ChatRoomRequest::LEAVE :
                        {
                              if (proxy != nullptr) {
                                    cr->leaveChatRoom(proxy);
                                    delete proxy;
                                    proxy = nullptr;
                              }
                              break;
                        }
                        default :
                              std::cerr << "unknown message " << crr.req() << std::endl;
                        }
                  }
                  std::cout << "Client disconnected "<<std::endl;
            }
            // une copie identique
            ConnectionHandler * clone() const {
                  return new ChatRoomCH(*this);
            }
            ~ChatRoomCH() {
```

```
            if (proxy != nullptr) {
                  cr->leaveChatRoom(proxy);
                  delete proxy;
            }
      }
};

ChatServer::ChatServer(IChatRoom * cr, int port) :server(new ChatRoomCH(cr)) {
      server.startServer(port);
}

ChatServer::~ChatServer() {
      server.stopServer();
}



}
```

Le client est symétrique.

Pas trouvé mieux que passer par une string pour envoyer un message, on ajoute des primitives tempalte à Socket :

Dans la classe Socket.h

```

      template <typename T>
      void writeMessage (const T & msg) {
            std::string str;
            msg.SerializeToString(& str);
            writeString(str);
      }

      template <typename T>
      void readMessage (T & msg) {
            std::string str = readString();
            msg.ParseFromString(str);
      }
```

3 Abonnement et Notification

Question 1. Quel problème fondamental se pose pour les signatures de "join" et "leave" ? Proposez une approche permettant de contourner le problème, en restant en appui sur l'idée de Proxy distant.

Ce sont des adresses, clairement non valables dans l'espace d'adressage du serveur.

Il faut monter en abstraction, ce que je te donne, c'est quelque chose de nécessaire et suffisant pour que tu retrouves l'objet et puisse invoquer des traitements dessus.

On pense à un système de nommage, par exemple la paire "IP:port" peut permettre d'identifier de façon unique des objets.

Donc le Client qui souhaite s'abonner au ChatRoom, commence par créer un TCPServer qui fait squelette pour le Chatter. Il envoie le couple IP:port de ce serveur dans le réseau.

Côté serveur (i.e. là où la ChatRoom habite), le squelette de ChatRoom doit reconstruire un objet qui habite sur le serveur, et qui fait proxy pour le IChatter distant. C'est cet objet qu'on va abonner à la

ChatRoom.

L'ensemble de ces mécanismes devient beaucoup plus fluide avec un service de nommage distribué, i.e. à partir de l'identifiant unique de l'instance, on demande au service de nommage de sa propre machine de fournir l'objet sous forme utilisable.

Soit on me rend l'objet directement s'il est sur cette machine, soit s'il est distant mais qu'on a déjà un proxy instancié pour on rend le proxy, sinon on crée un proxy qu'on connecte et on rend ça.

Question 2. Réalisez un client pour le chat et un serveur pour le chat complets.

On réalise ce qui est décrit dans le pragraphe précédent.

Cf. les corrigés précedents et les nouvelles classes Proxy/serveur pour Chatter.

ChatterProxy.cpp

```
#include "ChatterProxy.h"
#include "chat.pb.h"
#include <unistd.h>

namespace pr {

ChatterProxy::ChatterProxy(const std::string & host, int port) {
      sock.connect(host,port);
}

bool writeRequest (Socket & s, ChatterRequest::Request req) {
      ChatterRequest crr;
      crr.set_req(req);
      s.writeMessage(crr);
      return true;
}

std::string
ChatterProxy::getName() const {
      writeRequest(sock,ChatterRequest::GETNAME);
      GetSubjectResponse resp;
      sock.readMessage(resp);
      return resp.msg();
}


void ChatterProxy::messageReceived (ChatMessage msg) {
      writeRequest(sock,ChatterRequest::RECVMESSAGE);
      RecvMessageArgs args;
      args.mutable_msg()->set_author(msg.getAuthor());
      args.mutable_msg()->set_msgtext(msg.getMessage());
      sock.writeMessage(args);
}

ChatterProxy::~ChatterProxy() {
      writeRequest(sock,ChatterRequest::QUIT);
      sock.close();
}

} /* namespace pr */
```

ChatterServer.cpp

```cpp
#include "ChatterServer.h"
#include "chat.pb.h"
#include <unistd.h>
#include <string>
#include <iostream>

namespace pr {


class ChatterCH : public ConnectionHandler {
	IChatter * cr;
public :
	ChatterCH (IChatter * cr):cr(cr){}
	void handleConnection(Socket s) {
		while (s.isOpen()) {
			ChatterRequest crr;
			s.readMessage(crr);
			if (! s.isOpen()) {
				break;
			}
			//std::cout << "read request type :" << crr.req()<<std::endl;
			switch (crr.req()) {
				case ChatterRequest::GETNAME :
				{ // get subject
					// no args
					GetNameResponse resp ;
					resp.set_msg(cr->getName());
					s.writeMessage(resp);
					break;
				}
				case ChatterRequest::RECVMESSAGE :
				{
					RecvMessageArgs args ;
					s.readMessage(args);
					ChatMessage m (args.msg().author(), args.msg().msgtext());
					cr->messageReceived(m);
					break;
				}
				case ChatterRequest::QUIT :
				{
					std::cout << "End from client." << std::endl;
					return;
				}
				default :
					std::cerr << "unknown message " << crr.req() << std::endl;
				}
		}
		std::cout << "Client disconnected" << std::endl;
	}
	// une copie identique
	ConnectionHandler * clone() const {
		return new ChatterCH(*this);
	}
};

ChatterServer::ChatterServer(IChatter * cr, int port) :server(new ChatterCH(cr)) {
	server.startServer(port);
}
```

```
ChatterServer::~ChatterServer() {
	server.stopServer();
}

}
```

ChatterProxy.h

```
/*
 * ChatRoomProxy.h
 *
 * Created on: Dec 4, 2018
 * Author: ythierry
 */

#ifndef SRC_CHATTERPROXY_H_
#define SRC_CHATTERPROXY_H_

#include "IChatRoom.h"
#include "Socket.h"

namespace pr {

class ChatterProxy: public IChatter {
	mutable Socket sock;
public:
	ChatterProxy(const std::string & host, int port);
	std::string getName() const ;
	void messageReceived (ChatMessage msg) ;
	~ChatterProxy();
};

} /* namespace pr */

#endif /* SRC_CHATROOMPROXY_H_ */
```

ChatterServer.h

```
#ifndef SRC_CHATTERSERVER_H_
#define SRC_CHATTERSERVER_H_

#include "IChatRoom.h"
#include "TCPServer.h"

namespace pr {

class ChatterServer {
	TCPServer server;
public:
	ChatterServer(IChatter * cr, int port);
	~ChatterServer();
	struct sockaddr_in getAddress() { return server.getAddress() ; }
};

}

#endif /* SRC_CHATSERVER_H_ */
```

TME 8 : Proxy Distant

Objectifs pédagogiques :

- **proxy distant, RPC**
- **protobuf**

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/sMZeI78W`.
Vous ferez un push de vos résultats sur ce dépôt pour soumettre votre travail.

1 Client/Serveur de Chat

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