TD 8 : 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 27 du cours 8.

> 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

```
1  #ifndef SRC_SOCKET_H_
2  #define SRC_SOCKET_H_
3
4  #include <netinet/ip.h>
5  #include <string>
6  #include <iosfwd>
7
8  namespace pr {
9
10 class Socket {
11       int fd;
12
```

```
13 public :
14       Socket():fd(-1){}
15       Socket(int fd):fd(fd){}
16
17       // tente de se connecter à l'hôte fourni
18       void connect(const std::string & host, int port);
19       void connect(in_addr ipv4, int port);
20
21       bool isOpen() const {return fd != -1;}
22       int getFD() { return fd ;}
23
24       void close();
25 };
26
27 std::ostream & operator<< (std::ostream & os, struct sockaddr_in * addr);
28
29 }
30
31 #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

```
1  #include "Socket.h"
2  #include <unistd.h>
3  #include <sys/types.h>
4  #include <sys/socket.h>
5  #include <netdb.h>
6  #include <cstring>
7  #include <netinet/in.h>
8  #include <arpa/inet.h>
9  #include <iostream>
10
11 namespace pr {
12
13
14 void Socket::close() {
15       if (fd != -1) {
16             shutdown(fd,1);
17             ::close(fd);
18       }
19 }
20
21 void Socket::connect(const std::string & host, int port) {
22       // but : resoudre l'addresse "humains" vers une addresse format machine
23
24       struct addrinfo * addr;
```

```
25        /* Remplir la structure dest */
26        // les choses à null ne servent pas ici
27        if (getaddrinfo(host.c_str(), /* service*/ nullptr, /* hints*/ nullptr, &addr) !=
              0) {
28              perror("getaddrinfo");
29              return;
30        }
31        // champ "data" ai_addr (une addresse en reponse) du premier chainon de addrinfo
32        // dedans on trouve la chose utile : une addresse de socket, qu'il faut encore
33        // fixer vers un pointeur d'adresse de socket du domaine AF_INET
34        // on doit faire un (down)cast pour retyper sockaddr en sockaddr_in
35        in_addr ipv4 = ((struct sockaddr_in *) addr->ai_addr)->sin_addr;
36        // getaddrinfo alloue une liste, on desalloue poliment
37        freeaddrinfo(addr);
38        connect(ipv4, port);
39  }
40
41  void Socket::connect(in_addr ipv4, int port) {
42        // une addresse spécialisée pour internet
43        sockaddr_in dest;
44        dest.sin_family = AF_INET; // discriminant, aide à retrouver le type concret dans
              un switch case
45        dest.sin_addr = ipv4;
46        dest.sin_port = htons(port); // host to network
47
48        // create socket
49        fd = socket(AF_INET,SOCK_STREAM, /* protocole TCP par defaut sur SOCK_STREAM*/ 0);
50        if (fd <0) {
51              perror("socket");
52              return;
53        }
54
55        // ici on doit upcast de sockaddr_in (spécialisé) vers sockaddr plus général
56        // en OO, pas de syntaxe, en C il faut un cast
57        // on passe la taille réelle de l'objet sockaddr spécialisé
58        if (::connect(fd,(struct sockaddr *) &dest, sizeof dest) <0) {
59              perror("connect");
60              ::close(fd);
61              fd=-1;
62              return;
63        }
64  }
65
66  // sockaddr_in est présent dans l'API a plein d'endroits, e.g. expediteur dans un accept
67  // objectif : de l'addresse format machine vers du lisible humain
68  std::ostream & operator<< (std::ostream & os, struct sockaddr_in * addr) {
69        char hname [1024];
70        // obtient à partir de l'addresse machine le nom d'hote
71        if (getnameinfo((struct sockaddr *)addr,sizeof *addr, hname, 1024, nullptr, 0, 0)
              == 0) {
72              os << '"' << hname << '"' << " ";
73        }
74        // inet_ntoa : produit la chain "127.0.0.1" à partir d'une addresse ipv4 sur 4
              octets
75        // ntohs : network to host
76        os << inet_ntoa(addr->sin_addr) << ":" << ntohs(addr->sin_port) << std::endl;
77        return os;
78  }
79
```

```
80
81 }
```

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

ServerSocket.h

```
1  #ifndef SRC_SERVERSOCKET_H_
2  #define SRC_SERVERSOCKET_H_
3
4  #include "Socket.h"
5
6  namespace pr {
7
8  class ServerSocket {
9        int socketfd;
10
11 public :
12       // Demarre l'ecoute sur le port donne
13       ServerSocket(int port);
14
15       int getFD() { return socketfd;}
16       bool isOpen() const {return socketfd != -1;}
17
18       Socket accept();
19
20       void close();
21 };
22
23 } // ns pr
24 #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

```
1 #include "ServerSocket.h"
2 #include <cstring>
3 #include <unistd.h>
4 #include <iostream>
5
6 namespace pr {
7
8 ServerSocket::ServerSocket(int port) :socketfd(-1) {
9
```

```
10        // create socket
11        int fd = socket(AF_INET,SOCK_STREAM,0);
12        if (fd == -1) {
13              perror("create socket");
14              return;
15        }
16
17        // bind
18        struct sockaddr_in sin; /* Nom de la socket de connexion */
19
20        memset(&sin, 0, sizeof(sin)); // utile ?
21        sin.sin_family = AF_INET;
22        sin.sin_addr.s_addr = htonl(INADDR_ANY); // on attend sur n'importe quelle
              interface de la machine
23        sin.sin_port = htons(port); // host to network
24
25        /* nommage, meme probleme de typage sockaddr que dans la Socket */
26        if (bind(fd, (struct sockaddr *) &sin, sizeof(sin)) < 0) {
27              perror("bind");
28              // on veut le close de unistd, pas le close membre de cette classe
29              ::close(fd);
30              return;
31        }
32
33        // listen
34        if (listen(fd, 50) < 0) {
35              perror("listen");
36              ::close(fd);
37              return;
38        }
39
40        // success !
41        socketfd = fd;
42  }
43
44  Socket ServerSocket::accept() {
45        struct sockaddr_in exp;
46        socklen_t len = sizeof(exp);
47        // on qualifie sinon le compilo rale
48        int scom = ::accept(socketfd, (struct sockaddr *) &exp, &len);
49        if (scom < 0) {
50              perror("accept");
51        } else {
52              // en appui sur operator << développé plus haut
53              std::cout << "Accepted connection from " << &exp << std::endl;
54        }
55        return scom;
56  }
57
58  void ServerSocket::close() {
59        if (socketfd != -1) {
60              ::close(socketfd);
61        }
62  }
63
64  }
```

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;
}
```

```
23
24 int main() {
25       pr::ServerSocket ss(1664);
26
27       while (1) {
28             pr::Socket sc = ss.accept();
29
30             int fd = sc.getFD();
31
32             ssize_t msz = sizeof(int);
33             while (1) {
34                   int lu;
35                   auto nblu = read(fd, &lu, msz);
36                   if (nblu == 0) {
37                         std::cout << "Fin connexion par client" << std::endl;
38                         break;
39                   } else if (nblu < msz) {
40                         perror("read");
41                         break;
42                   }
43                   std::cout << "lu =" << lu << std::endl;
44
45                   if (lu == 0) {
46                         break;
47                   }
48                   lu++;
49                   if (write(fd, &lu, msz) < msz) {
50                         perror("write");
51                         break;
52                   }
53                   std::cout << "envoyé =" << lu << std::endl;
54             }
55             sc.close();
56       }
57
58       ss.close();
59       return 0;
60 }
```

3 Un serveur TCP multi-threadé

Question 1. Ecrire une classe `TCPServer` qui encapsule le comportement du serveur. Elle propose :

- une opération `bool startServer(int port)` qui démarre l'écoute sur le port ciblé à l'aide d'une ServerSocket. Cette opération consomme le thread qui l'invoque.
- une opération `void handleClient (Socket scom)` qui isole la partie correspondant à la discussion avec un client.

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.h

```
1 #ifndef SRC_TCPSERVER_H_
```

```
#define SRC_TCPSERVER_H_

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

namespace pr {

// un serveur TCP, la gestion des connections est déléguée
class STCPServer {
      ServerSocket * ss; // la socket d'attente si elle est instanciee

      // ajout dans un second temps.
      std::vector<std::thread> threads;
public :
      STCPServer(): ss(nullptr){}
      void handleClient(Socket scom);
      bool startServer0 (int port);
      bool startServer1 (int port);
      ~STCPServer();
};

} // ns pr

#endif /* SRC_TCPSERVER_H_ */
```

server.cpp

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

namespace pr {

// passage par copie délibéré, pour éviter les problèmes de durée de vie de sc dans le
    contexte thread (si on passait une réf).
void STCPServer::handleClient(Socket sc) {
      // discussion client : read puis write.
      int lu ;
      read(sc.getFD(),&lu,sizeof(lu));
      ++lu;
      write(sc.getFD(),&lu,sizeof(lu));
      sc.close();
}

// Version basique : mobilise le client ( Question 6 )
bool STCPServer::startServer0(int port) {
      ss = new ServerSocket(port);
      if (ss->isOpen()) {
            while (1) {
                  std::cout << "En attente sur accept" << std::endl;
                  Socket sc = ss->accept();
                  if (sc.isOpen()) {
                        handleClient(sc);
                  }
```

```
            }
            // inaccessible
            return true;
      }
      return false;
}

// version multi thread : un par client
bool STCPServer::startServer1(int port) {
      ss = new ServerSocket(port);
      if (ss->isOpen()) {
            while (1) {
                  std::cout << "En attente sur accept" << std::endl;
                  Socket sc = ss->accept();
                  if (sc.isOpen()) {
                        threads.emplace_back(&STCPServer::handleClient,this,sc);
                  }
            }
            // inaccessible
            return true;
      }
      return false;
}

STCPServer::~STCPServer() {
      for (auto & t: threads) {
            t.join();
      }
}

}
```

Question 2. On souhaite à présent que chaque client connecté soit traité par un thread nouvellement créé. Implanter ce comportement.

Question 3. Si l'opération `handleClient` est déclarée virtuelle pure (ou abstraite), comment définir un serveur TCP concret ?

> en héritant de server et en fournissant un handleClient.
>
> L'extension par héritage est cependant d'assez mauvais goût, on verra en séance suivante une version configurable sans héritage.

TD 9 : Appels de Procédures Distantes

Objectifs pédagogiques :

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

1 Un Serveur TCP générique

Question 1. 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);
                              FD_SET(readpipe, &read);
                              FD_SET(serv->ss->getFD(), &read);
                              // premier argument = plus grand fildescriptor dans un des
                                   set + 1
                              // on peut comprendre que fd_set est un genre de bitset,
                                  on donne sa taille ici.
                              select(std::max(readpipe, serv->ss->getFD()) +1, &read, /*
                                  write*/nullptr, /*except*/nullptr, /*timeout*/nullptr
                                  );

                              // au retour du select, read est modifié pour contenir le
                                  ou les fd qui nous ont reveillé
                              if (FD_ISSET(readpipe,&read)) {
                                    std::cout << "asked to quit" << std::endl;
                                    return;
                              }
                              // sinon ben c'est la socket qui nous a réveillé : un
                                  client s'est pointé
                              auto sc = serv->ss->accept();
                              if (sc.isOpen()) {
                                    auto copy = serv->handler->clone();
                                    copy->handleConnection(sc);
                                    // En version Pool de thread
                                    // pool.submit(new JobHandler(copy,sc));
                                    // et on enlève le delete (fait par JobHandler)
                                    delete copy;
                              }
                        }
                  },this, pipefd[0]);
                  killpipe = pipefd[1];
```

```
                return true;
            }
        }
        return false;
}


void TCPServer::stopServer() {
        if (waitT != nullptr) {
            int n = 1;
            write(killpipe,&n,sizeof(int));
            waitT->join();
            delete waitT;
            delete ss;
            close(killpipe);
            killpipe = -1;
            waitT = nullptr;
        }
}

}
```

Question 2. 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 3. 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 4. 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_ */
```

Question 5. Ajouter un pool de thread, membre de TCPServer, et l'utiliser au lieu de créer un nouveau thread à chaque connection.

Proxy/Stub

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 {

```

```
9  class ChatMessage {
10       std::string author;
11       std::string message;
12 public :
13       ChatMessage (const std::string & author, const std::string & msg):author(author),
             message(msg) {};
14       const std::string & getAuthor() const {return author; }
15       const std::string & getMessage() const {return message; }
16 };
17
18 class IChatter {
19 public :
20       virtual std::string getName() const = 0;
21       virtual void messageReceived (ChatMessage msg) = 0;
22       virtual ~IChatter() {}
23 };
24
25 class IChatRoom {
26 public :
27       virtual std::string getSubject() const = 0;
28       virtual std::vector<ChatMessage> getHistory() const = 0;
29       virtual bool posterMessage(const ChatMessage & msg) = 0;
30       virtual bool joinChatRoom (IChatter * chatter) = 0;
31       virtual bool leaveChatRoom (IChatter * chatter) = 0;
32       virtual size_t nbParticipants() const =0;
33       virtual ~IChatRoom() {}
34 };
35
36 }
37
38 #endif /* SRC_ICHATROOM_H_ */
```

Ainsi que les implantations textuelles triviales suivantes :

TextChatRoom.h

```
1  #ifndef SRC_TEXTCHAT_H_
2  #define SRC_TEXTCHAT_H_
3
4  #include "IChatRoom.h"
5
6  #include <iostream>
7  #include <algorithm>
8
9  namespace pr {
10
11 class TextChatter : public IChatter {
12       std::string name;
13 public :
14       TextChatter (const std::string & name):name(name){}
15       std::string getName() const { return name ; }
16       void messageReceived (ChatMessage msg) { std::cout << "(chez " << name << ") de : " <<
             msg.getAuthor() << " > " << msg.getMessage() << std::endl; }
17 };
18
19 class TextChatRoom : public IChatRoom {
20       std::string subject;
21       std::vector<ChatMessage> history;
22       std::vector<IChatter *> participants;
23 public :
24        TextChatRoom(const std::string & subject) : subject(subject) {}
25        std::string getSubject() const { return subject; }
```

```
26        std::vector<ChatMessage> getHistory() const { return history ; }
27        bool posterMessage(const ChatMessage & msg) {
28              history.push_back(msg);
29              for (auto c : participants) {
30                    c->messageReceived(msg);
31              }
32              return true;
33        }
34        bool joinChatRoom (IChatter * chatter) {
35              participants.push_back(chatter);
36              return true;
37        }
38        bool leaveChatRoom (IChatter * chatter) {
39              auto it = std::find(participants.begin(),participants.end(),chatter);
40              if (it != participants.end()) {
41                    participants.erase(it);
42                    return true;
43              }
44              return false;
45        }
46        size_t nbParticipants() const { return participants.size(); }
47  };
48
49  }
50
51  #endif
```

Un exemple de test simple est fourni :

chatbasic.cpp

```
1   #include "TextChatRoom.h"
2
3   using namespace pr;
4   using namespace std;
5
6
7   int main () {
8
9         IChatRoom * cr = new TextChatRoom ("C++");
10        TextChatter alice("alice");
11        TextChatter bob("bob");
12        cr->joinChatRoom(&alice);
13        cr->joinChatRoom(&bob);
14
15        cr->posterMessage({"bob","salut"});
16        cr->posterMessage({"alice","hello"});
17        cr->posterMessage({"alice","bye"});
18
19        cr->leaveChatRoom(&alice);
20        cr->posterMessage({"bob","reste !"});
21
22        cr->leaveChatRoom(&bob);
23        delete cr;
24  }
```

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 !
```

2 Mise en Place

2.1 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

```
1  #ifndef SRC_CONNECTIONHANDLER_H_
2  #define SRC_CONNECTIONHANDLER_H_
3
4  #include "Socket.h"
5
6  namespace pr {
7
8  // une interface pour gerer la communication
9  class ConnectionHandler {
10 public:
11       // gerer une conversation sur une socket
12       virtual void handleConnection(Socket s) = 0;
13       // une copie identique
14       virtual ConnectionHandler * clone() const = 0;
15       // pour virtual
16       virtual ~ConnectionHandler() {}
17 };}
18 #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

```
1
2       int readInt ();
3       void writeInt (int n);
4
5       void writeString (const std::string & s);
6       std::string readString ();
```

Socket.cpp

```
1       }
2   }
3
4   int Socket::readInt () {
5       int n=0;
6       if ( read(fd,&n,sizeof(n)) < sizeof(n)) {
7             perror("readI");
8             ::close(fd);
9             fd = -1;
10      }
11      return n;
12  }
13
14  void Socket::writeInt (int n) {
15      if ( write(fd,&n,sizeof(n)) < sizeof(n)) {
16            perror("writeI");
17            ::close(fd);
18            fd = -1;
19      }
20  }
21
22  void Socket::writeString (const std::string & s) {
23      size_t sz = s.length();
24      writeInt(sz);
25      if ( write(fd,s.data(),sz) < sz) {
26            perror("writeS");
27            ::close(fd);
28            fd = -1;
29      }
30  }
31
32  std::string Socket::readString () {
33      size_t sz = readInt();
```

ChatServer.h

```
1   #ifndef SRC_CHATSERVER_H_
2   #define SRC_CHATSERVER_H_
3
4   #include "IChatRoom.h"
5   #include "TCPServer.h"
6
7   namespace pr {
8
9   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) {
```

```
28              std::unique_lock<std::mutex> l(mut);
29              return deco->joinChatRoom(chatter);
30        }
31        bool leaveChatRoom (IChatter * chatter) {
32              std::unique_lock<std::mutex> l(mut);
33              return deco->leaveChatRoom(chatter);
34        }
35        virtual size_t nbParticipants() const {
36              std::unique_lock<std::mutex> l(mut);
37              return deco->nbParticipants();
38        }
39  };
40
41  }
42
43
44  #endif /* SRC_MTCHATROOM_H_ */
```

Question 3. Proposez un main pour le serveur.

chatserver.cpp

```
1   #include "TextChatRoom.h"
2   #include "ChatServer.h"
3   #include "MTChatRoom.h"
4   #include <iostream>
5   #include <unistd.h>
6   #include <csignal>
7   #include <cstring>
8
9
10  int main() {
11
12        // SIGPIPE on broken connections sucks in multi-thread context.
13        // Le problème : on ne peut pas fiablement savoir quel thread a déclenché le
              sigpipe,
14        // ni le rattrapper correctement avec ce bon thread.
15        // En ignorant SIGPIPE, les read/write sur socket fermée vont rendre -1 + errno =
              EPIPE
16        // mais pas de signaux engendrés
17        // Une alternative serait d'utiliser send (plutot que write) avec un flag
              MSG_NOSIGNAL
18        struct sigaction act;
19        memset(& act,0,sizeof(act));
20        act.sa_handler = SIG_IGN;
21        sigaction(SIGPIPE,&act, nullptr);
22
23        // la version avec signal ne suffit pas de façon durable : le handler est reset à
              SIG_DFL (sur ma machine)
24        // ce comportement varie entre système et versions
25        //signal(SIGPIPE,SIG_IGN);
26
27        pr::TextChatRoom tcr ("C++");
28        pr::MTChatRoom mcr (&tcr);
29        pr::TextChatter schat("Echo Server");
30        mcr.posterMessage({"serveur","début session"});
```

```
31        tcr.joinChatRoom(&schat);
32        {
33              pr::ChatServer server(&mcr,1664);
34
35              // attend entree sur la console
36              std::string s ;
37              std::cin >> s ;
38
39              std::cout << "Début fin du serveur." << std::endl ;
40
41              // on quit = dtor du serveur
42        }
43        tcr.posterMessage({"serveur","fin session"});
44        std::cout << "Ok fin du serveur." << std::endl;
45        tcr.leaveChatRoom(&schat);
46
47        return 0;
48  }
```

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.

2.2 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

```
1   /*
2    * ChatRoomProxy.h
3    *
4    * Created on: Dec 4, 2018
5    * Author: ythierry
6    */
7
8   #ifndef SRC_CHATROOMPROXY_H_
9   #define SRC_CHATROOMPROXY_H_
10
11  #include "IChatRoom.h"
12  #include "ChatterServer.h"
13  #include "Socket.h"
14
15  namespace pr {
16
17  class ChatRoomProxy: public IChatRoom {
18        mutable Socket sock;
19        ChatterServer * serv;
20  public:
21        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

```
#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

```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;
}
```

TD 10 : Future, Promise, Simulation de Tubes

Objectifs pédagogiques :

- **future, promise en C++**
- **simulation de pipe en threads**

1 Future, Promise

L'objectif de cet exercice est de réaliser nos propres versions de `future` et `promise` simplifiées par rapport au standard (pas de gestion des exceptions) en appui sur les primitives de synchronisation classiques des thread (mutex, condition).

On propose d'implanter trois classes template : `shared_result`, `promise`, `future`.

1.1 Shared Result

La classe `shared_result` est le coeur de notre synchronisation ; c'est un conteneur qui *peut* héberger un objet de type T. Son API est la suivante :

- `T get ()` rend la valeur stockée, bloquant si elle n'est pas disponible.
- `void set(T &val)` positionne la valeur, débloque le client bloqué sur `get` le cas échéant.
- `bool is_set () const` (non bloquant) rend vrai si `set` a été invoqué sur cet objet.

Question 1. Proposez, en appui sur les primitives de synchronisation des threads C++11, une implantation pour cette classe.

Pool.h

```
#ifndef SRC_PROMISE_H_
#define SRC_PROMISE_H_

#include <mutex>
#include <condition_variable>
#include <memory>
#include <future>

namespace pr {

template<typename T>
class shared_result {
      // la donnée
      T res;
      // est-on valide ?
      bool isSet;
      // éléments de synchro
      std::mutex mut;
      std::condition_variable cv;
public:
      // init = not set
      shared_result(): res(),isSet(false) {}

      // bloquant si not set
      T get () {
            std::unique_lock<std::mutex> l (mut);
            // NB : cv.wait(l,pred) <=> while (!pred) cv.wait(l);
            // On pourrait utiliser direct l'attribut plutot que l'accesseur
```

```
            cv.wait(l,[this] () { return is_set();});

            // initialement (Q1)
            // en Q1 : version simple (paie la copie)
            // return res;

            // Q5 : on ajoute un move, qui invalide du coup res.
            // du coup mettre is_set à false parait raisonnable
            // mais en vrai on NE DOIT PAS refaire get sur ce future.
            isSet = false;
            return std::move(res);
      }
      void set (T & val) {
            {
                  std::unique_lock<std::mutex> l (mut);
                  // en Q1 : version simple (paie la copie)
                  // res = val;

                  // en Q5 : utilisation du move
                  // du coup on ne paie plus la copie, mais le résultat est invalidé par un get.
                  res = std::move(val);
                  isSet = true;
            }
            cv.notify_one();
      }
      bool is_set () const {
            return isSet;
      }
};

// Question 4 : gestion mémoire
// on utilise un shared_ptr, muni de son compteur de références
// Quand promise et future sont détruits, alors le shared_result peut être libéré
template<typename T>
using Resptr = std::shared_ptr<pr::shared_result<T>>;

// Au début (en question 2 et 3) on utilise juste un "pr::shared_result<T>*"
// et on évite de parler trop des destructeurs...

// Notre "future" ne fait quasiment que déléguer au shared_result.
// il manque un "make_shared" entre autres pour que ce soit plus intéressant.
template<typename T>
class future {

      // initialement juste un pointeur ici
      // en Q5 on utilise un shared_ptr plutot
      Resptr<T> result;
public :
      future (const Resptr<T> & res): result(res) {}
      // bloquant si non disponible
      T get () {
            return result->get();
      }
      bool isAvailable() const {
            return result->is_set();
      }
};
```

```
87  // La promise, s'occupe de construire le shared_state, le future
88  template<typename T>
89  class promise {
90        // au début on utilise un simple pointeur
91        Resptr<T> result;
92  public :
93        // ctor : construire un shared result (avec new)
94        // au début ce sera juste un pointeur ici.
95        // Mais ne pas le delete dans le dtor, car future peut aussi le pointer
96        // avec shared_ptr on s'affranchit de ces problèmes de delete
97        promise(): result(new shared_result<T>()) {}
98
99        // construction d'un future
100       future<T> getFuture() {
101             return future<T> (result);
102       }
103       // mise à jour de la valeur + débloquer
104       void set_value(const T & r) {
105             result->set(r);
106       }
107 };
108
109 }
110
111 #endif /* SRC_PROMISE_H_ */
```

1.2 Future

La classe `future` représente le canal de lecture pour un client qui attend un résultat. On construit un objet `future` en lui passant l'adresse d'un `shared_result`. Son API est la suivante :

- `T get ()` : bloquant si pas encore disponible
- `bool isAvailable() const` : vrai si le résultat est prêt

Question 2. Proposez une implantation pour cette classe.

1.3 Promise

La classe `promise` représente le canal en écriture vers la donnée partagée, dans lequel le thread serviteur va poser le résultat de son calcul. Quand on construit un objet `promise`, il crée une instance de `shared_result`. Son API est la suivante :

- `future<T> getFuture()` construit et rend un objet future, attaché au résultat.
- `void set_value(const T & r)` affecte une valeur au résultat

Question 3. Proposez une implantation pour cette classe.

2 Pool de thread et Future

On revient sur la réalisation d'un pool de thread, comme au TD4.

Pool.h

```
1 #ifndef SRC_POOL_H_
2 #define SRC_POOL_H_
```

```
3
4  #include "Queue.h"
5  #include <vector>
6  #include <thread>
7
8  namespace prTD4 {
9
10 class Job {
11 public:
12       virtual void run () = 0;
13       virtual ~Job() {};
14 };
15
16 // fonction passee a ctor de thread
17 inline void poolWorker(Queue<Job> * queue) {
18       while (true) {
19             Job * j = queue->pop();
20
21             // pour la terminaison propre
22             if (j == nullptr) {
23                   // on est non bloquant = il faut sortir
24                   return;
25             }
26
27             j->run();
28             delete j;
29       }
30 }
31
32 class Pool {
33       Queue<Job> queue;
34       std::vector<std::thread> threads;
35 public:
36       Pool(int qsize) : queue(qsize) {}
37       void start (int nbthread) {
38             threads.reserve(nbthread);
39             for (int i=0 ; i < nbthread ; i++) {
40                   threads.emplace_back(poolWorker, &queue);
41             }
42       }
43
44       void stop() {
45             // débloque les threads bloqués sur pop
46             queue.setBlocking(false);
47             // on attend qu'ils aient fini leur job actuel
48             for (auto & t : threads) {
49                   t.join();
50             }
51             threads.clear();
52       }
53       ~Pool() {
54             stop();
55       }
56       void submit (Job * job) {
57             queue.push(job);
58       }
59 };
60
61 } /* namespace pr */
62
63 #endif /* SRC_POOL_H_ */
```

On veut réaliser une version `Pool<T>` auquel on puisse soumettre des `Job<T>` qui définit une seule opération `T run()` (donc au lieu de la signature `void run()` proposée). Pour éviter de tomber trop profondément dans les problèmes d'instantiation de template, tous les `Job<T>` soumis à un `Pool<T>` donné auront donc le même type de retour T.

La nouvelle signature de l'opération `void submitJob (Job *job)` est `future<T> submitJob(Job<T> *job)`.

Question 1. Modifier le Pool de thread et les définitions liées (Job, fonction exécutée par les thread...) pour obtenir ce nouveau comportement.

Pool.h

```
#ifndef SRC_POOL_H_
#define SRC_POOL_H_

#include "promise.h"
#include "Queue.h"
#include <vector>
#include <thread>

namespace pr {

template<typename T>
class Job {
public:
      virtual T run () = 0;
      virtual ~Job() {};
};

// AJOUT : un mécanisme qui stocke la paire Job/promise associée
// dès le submit on doit créer une promise, pour rendre toute suite le future associé
// quand un thread qui pop ce PackagedTask a fini de traiter le job, on set value dans cette promise.
template<typename T>
class PackagedTask {
      Job<T> * job;
      promise<T> result;
public :
      // implicitement : promise<T> est construit, ce qui crée un shared_result<T> (avec un new)
      // et y réfère via un shared_ptr
      PackagedTask (Job<T> * job) : job(job){}
      // API d'une paire
      promise<T> & getPromise() { return result; }
      Job<T> * getJob() { return job; }
      // NB : avec le shared_ptr utilisé pour référer au shared_result, pas de souci ici
      // la promise va mourir aussi, mais pas le résultat shared_result
      ~PackagedTask() { delete job; }
};


// fonction passee a ctor de thread
template<typename T>
inline void poolWorker(Queue<PackagedTask<T>> * queue) {
      while (true) {
            // le pop nous rend maintenant un PackagedTask, une paire Job/Promise
            PackagedTask<T> * task = queue->pop();
            // pour la terminaison propre
            if (task == nullptr) {
```

```
46                  // on est non bloquant = il faut sortir
47                  return;
48            }
49            Job<T> * j = task->getJob();
50            // On invoque run = traitement
51            // dans la foulée, on met à jour la promise avec ce résultat
52            // ca debloque potentiellement le client en attente sur le future
53            task->getPromise().set_value(j->run());
54            // destruction des données du Job, de la promise qui a été satisfaite
55            delete task;
56      }
57  }
58
59  template<typename T>
60  class Pool {
61        // on stocke des paires Job/Promise
62        Queue<PackagedTask<T>> queue;
63        std::vector<std::thread> threads;
64  public:
65        Pool(int qsize) : queue(qsize) {}
66        void start (int nbthread) {
67              threads.reserve(nbthread);
68              for (int i=0 ; i < nbthread ; i++) {
69                    threads.emplace_back(poolWorker<int>, &queue);
70              }
71        }
72
73        void stop() {
74              // débloque les threads bloqués sur pop
75              queue.setBlocking(false);
76              // on attend qu'ils aient fini leur job actuel
77              for (auto & t : threads) {
78                    t.join();
79              }
80              threads.clear();
81        }
82        ~Pool() {
83              stop();
84        }
85        future<T> submit (Job<T> * job) {
86              // on ecapsule dans un PackagedTask
87              // ce qui engendre la création de la promise, donc du shared_result
88              PackagedTask<T> * t = new PackagedTask<T>(job);
89              queue.push(t);
90              // on rend le future
91              return t->getPromise().getFuture();
92        }
93  };
94
95  } /* namespace pr */
96
97  #endif /* SRC_POOL_H_ */
```

Question 2. Expliquer comment utiliser ce nouveau mécanisme dans un main.

main.cpp

```cpp
#include "Pool.h"
#include <iostream>
#include <thread>
#include <vector>

using namespace std;
using namespace pr;


class SleepJob : public Job<int> {
	int foo (int v) {
		// traiter un gros calcul
		this_thread::sleep_for(1s);
		return v % 255;
	}
	int arg;
public :
	SleepJob(int arg) : arg(arg) {}
	int run () {
		std::cout << "Computing for arg =" << arg << std::endl;
		int ret = foo(arg);
		std::cout << "Obtained for arg =" << arg << " result " << ret << std::endl;
		return ret;
	}
	~SleepJob(){}
};

int main () {
	const int NBTHREAD = 5;
	const int NBJOB = 30;

	Pool<int> pool(NBTHREAD);
	pool.start(NBTHREAD);

	vector<pr::future<int> > results;

	for (int i = 0; i < NBJOB ; i++) {
		results.emplace_back(pool.submit(new SleepJob(i)));
	}

	for (auto & i : results) {
		std::cout << i.get() << std::endl;
	}
	// garantie : tout est fini ici
	pool.stop();

	return 0;
}
```

[1] sont censés être accessibles sans assistant de TD.

[1] Les deux premiers exercices sont adaptés d'une annale de 2006, rédigée par L. Arantes, O. Marin et P.Sens.

3 Simulation des Pipes en Thread

Nous voulons offrir un ensemble de fonctions qui permet à des threads de communiquer par un tube. Autrement dit, un mécanisme de communication unidirectionnel où chacune des extrémités permet à un thread soit de lire dans le tube soit d'y écrire des caractères. A cette fin, nous avons défini la structure suivante qui regroupe des informations d'un tube donné :

```
1 class tube {
2   char vect [TAILLE_PIPE]; /* vecteur pour sauvegarder le contenu du tube */
3   int lect_off, ecr_off; /* indice de la prochaine case de vect à lire ou libre
        respectivement */
4   int nb_char; /* nombre de caractères dans le tube */
5 public:
6   tube():lect_off(0),ecr_off(0),nb_char(0){}
7 };
```

où :

- `vect [TAILLE_PIPE]` : vecteur de taille `TAILLE_PIPE` qui sauvegarde les caractères qui n'ont pas été encore lus dans le tube. Ce vecteur est géré de façon circulaire ;
- `lect_off et ecr_off` : indice de la prochaine case de vect à lire ou à écrire respectivement ;
- `nb_char` : nombre de caractères dans `vect` ;

On propose de réaliser les méthodes suivantes du tube, conçues pour un contexte multi-thread.

- Lecture de n caractères dans le tube : `int read (char *buf, int n);`.
  - Fonction bloquante qui lit au plus n caractères du tube.
  - Si le tube contient x caractères, la fonction extrait du tube $nb\_lu = min(x, n)$ caractères qui sont alors copiés dans buf;
  - Si le tube est vide, la thread est mise en sommeil jusqu'à ce que le tube ne soit plus vide;
  - La fonction renvoie le nombre de caractères lus dans le tube (nb_lu caractères).
- Ecriture de n caractère dans le tube : `int write (char *buf, int n);`.
  - Fonction qui écrit de façon atomique n caractères dans le tube, s'il y a de la place.
  - S'il y a au moins n emplacements libres dans le tube, une écriture atomique est réalisée et le nombre de caractères écrits est renvoyé. Dans ce cas tous les threads lecteur en attente sur le tube seront réveillées.
  - Sinon la fonction renvoie -1.

Nous considérons que les programmes utilisent correctement les fonctions, c'est-à-dire qu'un thread ne peut que lire ou écrire dans un tube.

Question 1. Ajouter les attributs utiles à la classe `tube` et codez ces deux méthodes. On suggère de procéder caractère par caractère pour plus de facilité, ou de faire deux cas selon qu'on déborde ou non (stockage circulaire) et une à deux copies (avec `memcpy`) selon le cas.

Rien de bien mystérieux, juste attention à bien itérer. Une version avec memcpy ici.

tube.h

```
1 #ifndef SRC_TUBE_H_
2 #define SRC_TUBE_H_
3
4 #include <mutex>
5 #include <cstring>
6 #include <condition_variable>
7 #include <csignal>
8
```

```
namespace pr {

#define TAILLE_PIPE 4096

class tube {
       char vect [TAILLE_PIPE]; /* vecteur pour sauvegarder le contenu du tube */
       int lect_off, ecr_off; /* indice de la prochaine case de vect à lire ou libre
           respectivement */
       int nb_char; /* nombre de caractères dans le tube */
       std::mutex m;
       std::condition_variable cv;

       int nbthread ; // AJOUT
     public:
       tube():lect_off(0),ecr_off(0),nb_char(0){}
       int read (char *buf, int n) {captures all by value
             std::unique_lock<std::mutex> l(m);
             cv.wait(l,[&](){ return nb_char > 0 ;});
             n = n<nb_char?n:nb_char;
             int tot = n;
             if (lect_off + n > TAILLE_PIPE) {
                 auto alire = TAILLE_PIPE - lect_off;
                 ::memcpy(buf,vect + lect_off, alire);
              lect_off = 0;
                 buf += alire;
                 nb_char -=alire;
                 n -= alire;
             }
             ::memcpy(buf,vect + lect_off, n);
             lect_off += n;
             nb_char -=n;
             return tot;
       }
       int write(char *buf, int n) {
             std::unique_lock<std::mutex> l(m);

             // AJOUT en question 3 : signaux
             if (nbthread == 1) {
                   raise(SIGPIPE);
             }
             // FIN AJOUT

             if (n > TAILLE_PIPE - nb_char) {
                   return -1;
             }
             int tot = n;
             if (ecr_off + n > TAILLE_PIPE) {
                   auto aecr = TAILLE_PIPE - ecr_off;
                   ::memcpy(vect + ecr_off, buf, aecr);
                   ecr_off = 0;
                   buf += aecr;
                   nb_char +=aecr;
                   n -= aecr;
             }
             ::memcpy(vect + ecr_off, buf,n);
             ecr_off += n;
             nb_char +=n;
             cv.notify_all();
             return tot;
```

```
67          }
68  };
69
70  }
71
72
73  #endif /* SRC_TUBE_H_ */
```

Question 2. On suppose à présent que l'on ajoute un attribut `nbThread` comptabilisant le nombre de threads au total qui ont une référence vers une le tube. On suppose que cet attribut est correctement mis à jour au fil de la manipulation du pipe. Modifier la fonction `write` pour qu'elle délivre un signal `SIGPIPE` au thread qui invoque write s'il est le seul à pouvoir réaliser une lecture.

> cf corrigé précédent.

Question 3. Quel sera l'effet du signal sur le thread dans write ? Sur les autres threads du programme ?

> Pas terrible, on cible le processus pas une thread, on ne sait pas quelle thread va se le prendre, mais de toutes façons le handler est PROCESSUS spécifique, dont un des threads va traiter SIGPIPE (comportement par défaut = mourir).