TD 8 : Appels de Procédures Distantes

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 !
```

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

```
#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`.

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

Question 3. Proposez un main pour le serveur.

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`.

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

2.3 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 6. 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.

Question 7. Expliquez comment intégrer ces messages dans la solution ébauchée en partie I.

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.

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

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.