Environnement Client/Serveur - TP n°5
Programmation Asynchrone et MVC

Consignes Les exercices ou questions marqués d'un $*$ devront être d'abord rédigés sur papier afin de se préparer aux épreuves écrites de l'examen. Tous les TPs se font sous Linux.

1 Préambule

Les objectifs de ce TP sont les suivants:

1. Mettre à jour l'application de chat développée lors de TP3
2. Appliquer les paradigmes de programmation asynchrone et MVC
3. Consulter la documentation officielle d'une API.

2 Mini-Chat 2.0

Le but de cet exercice consiste à développer une nouvelle version du client Mini-Chat vu durant le TP n°3. Pour cela les mécanismes de boucle évènementielle et de programmation d'interface graphique de la bibliothèque Qt seront utilisés. L'application est développée en suivant les principes du modèle MVC et est composée de 7 fichiers:

main.cpp: Code source de la fonction principale. Celle ci est dédiée à l'instanciation des modules de l'application et au lancement de la boucle évènementielle.

client.cpp/client.hpp: Code source de la classe `Client`. Son rôle est de gérer la partie réseau de l'application.

main_window.cpp/main_window.hpp: Code source de la classe `MainWindow`. Cette dernière permet de construire l'interface graphique de l'application.

controller.cpp/controller.hpp: Code source de la classe `Controller`. Cette classe assure la liaison entre les différents signaux et fonctions de callback des objets formant l'application.

Ce TP consiste à redévelopper la partie client de l'application Mini-Chat. Ce nouveau client devra donc pouvoir communiquer avec la version du serveur qui été développée durant le TP 3. Télécharger le corrigé de ce tp afin de pouvoir utiliser le serveur pour teste le nouveau client une fois le tp terminé.

3 Bibliothèque Qt

Qt est une API complète, développée en C++ et orienté objet. Elle garantie la portabilité des programmes sur différentes plateformes par recompilation dés lors que ceux ci sont développés uniquement avec. Elle intègre des modules permettant de développer des interfaces graphiques, de manipuler des objets multimédia, de faire communiquer des processus, de construire des rendu 3D, d'accéder au réseau,...

3.1 Boucle évènementielle

La boucle évènementielle de l'API Qt est lancée dans la fonction `main(...)` via l'appel à la méthode `exec()` de la classe `QApplication`. Pour pouvoir ajouter des listener et des fonctons de callback à la boucle, on utilise la méthode statique `QObject::connect(...)`. Cette méthode permet de lier des *signals* (listener dans Qt) à des fonctions de callback qui sont appelles *slots*. Ainsi, si un élément graphique tel qu'un bouton venait à émettre un *signal* `clicked()` connecté à une fonction `foo()`, cette fonction sera appelée chaque fois qu'un utilisateur cliquera sur ce bouton.

Exemple: `QObject::connect(&boutonQuitter, SIGNAL(clicked()), qApp, SLOT(quit()))` permet de créer un boutant arrêtant l'application.

3.2 Compilation

La compilation d'un programme utilisant l'API Qt se fait via l'utilisation d'un `Makefile` généré automatiquement. Pour cela, on commence par créer le fichier de configuration du projet via la commande `qmake -project`. Un deuxième appel à `qmake`, sans argument, permet de lire ce fichier et de construire le `Makefile`.

4 Question

4.1 Analyse du code

Ex. 1 — $*$ Identifier le modèle, la vue et le contrôleur de l'application.

Answer (Ex. 1) — Le modèle est la classe `Client`, la vue est la classe `MainWindow` et le contrôleur est la classe `Controller`.

Ex. 2 — $*$ En utilisant la documentation de la bibliothèque, identifier les signaux qui sont susceptibles d'être utilisés pour le développement de cette application. (Justifier)

Answer (Ex. 2) — Nous avons besoin des signaux suivant:

- `clicked()` de la classe `QPushButton` pour l'accès à l'évènement des boutons.
- `error(QAbstractSocket::SocketError)` de la classe `TcpSocket` pour détecter une erreur lors de la phase de connexion au serveur
- `connected()` de la classe `QTcpSocket` pour détecter l'établissement de la connexion avec le serveur.
- `readyRead()` de la classe `QTcpSocket` pour détecter la présence de paquet dans le buffer d'entrée de la socket.
- `newMessage(QString)` de la classe client pour notifier la classe `MainWindow` qu'il y à un message à afficher. Il s'agit d'un signal créé pour l'application.

Ex. 3 — $*$ À quels slots ces signaux doivent ils être liés dans cette application.

Answer (Ex. 3) — Les liaisons signal/slot à établir dans l'application sont:

- `clicked()` de la classe `QPushButton` se lie aux slot `SendMessage()` du contrôleur et `Disconnect()` de la classe Client.
- `error(QAbstractSocket::SocketError)` de la classe `TcpSocket` se lie au slot `RetryOpenSocket(` de la classe `Client`
- `connected()` de la classe `QTcpSocket` se lie au slot `SendName()` de la classe Client.
- `readyRead()` de la classe `QTcpSocket` se lie au slot `ReadPackets()` de la classe client.
- `newMessage(QString)` de la classe `Client` se lie au slot `UpdateChat(QString)` de la classe MainWindow

4.2 Fenêtre principale

Ex. 4 — Compléter la fonction de callback `UpdateChat(...)` de façon à ce que le contenu de la zone d'affichage du client affiche la chaîne passée en paramètre en plus du texte déjà présent.

4.3 Client

Ex. 5 — Compléter le constructeur de la classe Client.

Ex. 6 — $*$ Compléter la méthode `OpenSocket()`. Où doit elle être appelée ? Ajouter ces appels dans le code de la classe.

Answer (Ex. 6) — Cette méthode doit être appelée dans le constructeur de la classe.

Ex. 7 — Compléter la méthode `SendPacket(...)`. L'utiliser dans les méthodes `SendXYZ(...)`.

Ex. 8 — $*$ Implémenter le protocole de communication dans la méthode `ReadPackets()`. Une boucle doit être utilisée dans ce code. Laquelle et pourquoi ?

Answer (Ex. 8) — Par sécurité la boucle doit itérer sur le buffer d'entrée de la socket. Sans elle, on peut oublier de traiter des paquets qui arriveraient simultanément où qui auraient été signalés par le même signal. Une boucle `while` peut très bien faire l'affaire.

4.4 Contrôleur

Ex. 9 — Compléter le constructeur du contrôleur afin d'établir les liens entre les signaux et slots du modèle, de la vue et du contrôleur.

Ex. 10 — Compléter la méthode `SendMessage()` du contrôleur.

4.5 Fonction principale

Ex. 11 — Le rôle de la fonction principale du programme consiste à instancier les différentes classe du programme et à lancer la boucle évènementielle. Cette dernière est réalisée par l'instruction `app.exec()`. Compléter cette fonction par l'instanciation des modèle, vue et contrôleur.

4.6 Améliorations

Ex. 12 — $*$ Une entorse à été faite par rapport ou modèle MVC vu en cours. Quelle est elle ? Comment peut-elle se justifier ? Quelle mécanisme de la boucle évènementielle de Qt a permis sa réalisation ?

Answer (Ex. 12) — Il n'y a pas de lien entre le modèle et la vue. Nous pouvons l'expliquer par le fait que la vue n'est pas réellement destinées à afficher les données de la classe client. Pour réaliser un "pseudo-lien" entre ces deux classes, nous avons utilisé la possibilité de passer un paramètre à la fonction de callback via le signal émit par le modèle.

Ex. 13 — $*$ Le protocole de communication de l'application présente un défaut affectant la gestion de la liste des utilisateurs connectés au serveur par les clients. Quel est-il ? Proposer une solution pour y remédier.

Answer (Ex. 13) — Lorsqu'un client se connecte, il ne peut pas connaitre la liste des utilisateur qui se sont connectés au serveur avant lui. Il lui est donc impossible de

savoir quel nom afficher lorsqu'un message est reçu. Une solution serait de précéder le contenu des messages par les noms des utilisateurs avant l'envoie par ces derniers.