

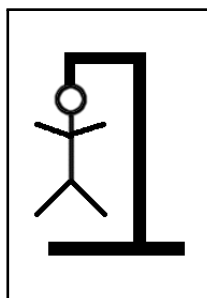
TP Développement Réseau n°3 : Jeu du pendu

© 2012 tv <tvaira@free.fr> - v.1.0

Sommaire

La couche Application	2
Rôle	2
Choix du protocole de transport	2
Principe de base	3
Travail demandé : le jeu du pendu	4
Première partie : élaboration du protocole	4
Principe du jeu	4
Déroulement d'une partie	4
Variantes	4
Identification des rôles	5
Modèle réseau	5
Structure des messages de la couche Application	6
Côté client	6
Côté serveur	6
Diagrammes des échanges	7
Tests de validation	7
Deuxième partie : mise en oeuvre sous Qt	7
Le module QtNetwork	7
Les classes fournies	8
Présentation de la classe QTcpSocket	8
Mise en oeuvre de la classe QTcpSocket	9
Troisième partie : réalisation du programme	17

L'objectif est de programmer un jeu du pendu en réseau à partir des sockets TCP/IP en mode connecté.




La couche Application

Rôle

Cette couche est l'**interface entre l'application utilisateur et le réseau**. Elle va apporter à l'utilisateur les services de base offerts via le réseau, comme par exemple le transfert de fichier, la messagerie ...

Rappel : l'application côté client demande un service et l'application côté serveur rend le service.

 Les protocoles de la couche Application sont souvent orientés caractères pour des raisons de représentation des données à échanger. Les protocoles orientés caractères sont les plus simples à mettre en oeuvre car ils ne nécessitent pas de protocoles supplémentaires pour la représentation des données dans des formats incompatibles (cf. *little endian vs big endian, int vs float, ...*).

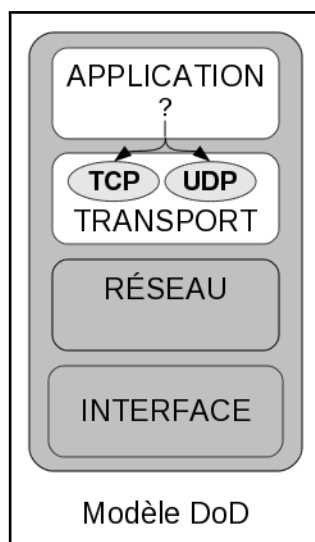
Cette couche contient donc tous les protocoles de haut niveau, comme par exemple Telnet, TFTP (*Trivial File Transfer Protocol*), SMTP (*Simple Mail Transfer Protocol*), HTTP (*HyperText Transfer Protocol*), FTP (*File Transfer Protocol*) ...

Il est évidemment aussi possible d'implémenter ses propres protocoles en fonction des ses besoins et services.

Choix du protocole de transport

Le point important pour la couche Application est le choix du protocole de transport à utiliser.

Rappel : TCP permet une communication en mode connecté et UDP en mode non connecté.



Par exemple, TFTP (surtout utilisé sur réseaux locaux) utilisera UDP, car on part du principe que les liaisons physiques sont suffisamment fiables et les temps de transmission suffisamment courts pour qu'il n'y ait pas d'inversion de paquets à l'arrivée. Ce choix rend TFTP plus rapide que le protocole FTP qui utilise TCP. A l'inverse, SMTP utilise TCP, car pour la remise du courrier électronique, on veut que tous les messages parviennent intégralement et sans erreurs.

Protocole	Avantages	Inconvénients
TCP	fiable (données acquittées et réémises si non reçues), permet de traiter des gros volumes de données en utilisant la segmentation	lent car adapte le débit des données envoyées à la bande passante disponible (contrôle de flux et de congestion), complexe à mettre en oeuvre (notamment le multi-clients côté serveur), utilisation du broadcast et multicast impossible
UDP	simple (pas de connexion, pas d'états entre le client et le serveur), économique en bande passante (en-tête de 8 octets), sans contrôle de congestion donc UDP peut émettre sans attendre	pas d'acquittement donc pas de garantie de bon acheminement, pas de détection de pertes donc pas retransmission possible, pas de contrôle de flux et donc risque de saturation des tampons (<i>buffers</i>), pas de séquençement donc les datagrammes peuvent être traités dans le désordre



En résumé, TCP est considéré comme fiable mais lent. Et inversement, UDP est rapide mais peu fiable.

Les applications utiliseront **TCP** car :

- gros volume de données et peu sensible au débit
- besoin d'un service fiable (donc pas de tolérance aux pertes)
- pas besoin du *multicast* ni du *broadcast*

Les applications utiliseront **UDP** car :

- faible volume de données et sensible au débit
- pas besoin d'un service fiable (donc tolérance aux pertes)
- besoin du *multicast* ou du *broadcast*



90% des services d'Internet utilisent TCP!

Principe de base

Rappel : Un protocole rend possible le dialogue entre des processus différents au sein d'un réseau.

De manière générale, un protocole de communication définit l'ensemble des procédures (ou règles) pour réaliser une communication :

- Le dictionnaire : les primitives (demande connexion, acquittement, ...)
- Le scénario du dialogue : enchaînement des primitives (diagramme de l'échange)
- Les modalités : taille et représentation des informations, temps d'attente, etc ...
- Les messages échangés : les différents champs (taille et contenu)

Un protocole est toujours décomposé en deux parties :

- le **PCI** (*Protocol Control Information*) : un en-tête (*header*) décrivant le protocole
- les **DATA** (DONNÉES) : une suite d'octets en provenance de la couche immédiatement supérieure



Un protocole de la couche Application permet l'échange de **messages**.

Travail demandé : le jeu du pendu

L'objectif est de programmer un jeu du pendu en réseau à partir des sockets TCP/IP en mode connecté. Il vous faudra définir un protocole de couche application pour permettre un dialogue entre le client et le serveur.

Vous travaillerez en équipe de 3 pour la définition du protocole et les tests. Par contre, chacun écrira son programme. Si le protocole est bien défini et respecté, les programmes pourront fonctionner correctement au sein même d'une équipe.

Un numéro de port pour le serveur sera attribué par équipe :

- équipe n°1 → 5001,
- équipe n°2 → 5002,
- ...



Une présentation du protocole défini sera faite par l'équipe oralement en classe.

Première partie : élaboration du protocole

Principe du jeu

Le pendu est un jeu consistant à trouver un mot en devinant quelles sont les lettres qui le composent. Le jeu se joue traditionnellement à deux. On a un nombre maximum de tentatives (qui correspondent au dessin du pendu) pour découvrir ce mot mystère.

Déroulement d'une partie

Les deux joueurs dans cet exemple s'appellent A et B.

B choisit un mot et fabrique le mot à découvrir : les lettres composant le mot sont remplacées par des tirets sauf pour la première et la dernière lettre.

A propose une lettre.

La lettre fait-elle partie du mot ?

- Oui : B l'inscrit à sa place autant de fois qu'elle se trouve dans le mot.
- Non : B dessine le premier trait du pendu (un coup joué et perdu).

Le jeu se poursuit jusqu'à ce que :

- A gagne la partie en trouvant toutes les lettres du mot ou en le devinant correctement.
- A perd la partie lorsque le nombre de coups joués et perdus est égal au nombre de coups maximum pour découvrir le mot.
- A abandonne la partie (que fait B? ...)

Variantes

Il existe des variantes :

Le nombre de coups maximum peut varier aussi en fonction du nombre de traits qui composent le dessin. On peut par exemple augmenter leur nombre en dessinant pendant la partie non seulement le pendu, mais aussi la potence, ce qui augmenterait les chances du joueur.

Il serait possible d'envisager que le joueur propose aussi la position de la lettre ce qui augmenterait la difficulté du jeu. Ou qu'il ait un temps limite pour découvrir le mot masqué ...

Dans ces variantes, il apparaît la possibilité d'intégrer un niveau de difficulté pour ce jeu (de débutant à expert par exemple). Le choix des mots à découvrir peut aussi être lié au niveau de la partie. D'autre part, on pourrait instaurer des choix de mots suivant des thèmes.

Évidemment, tout ceci influencera l'élaboration d'un protocole.

Identification des rôles

Dans une architecture client/serveur, il est important d'identifier les rôles :

- Le client : « celui qui demande le service ».
- Le serveur : « celui qui offre le service ».

Question 1. Quel service est rendu par le serveur ?

Réponse : Ici, le service est de « jouer une partie du jeu du pendu ».

Question 2. Que demande le client ?

Réponse : Le client demande à jouer une partie du jeu du pendu.

Question 3. Que répondra le serveur à la demande du client ?

Réponse : Le serveur lui répondra qu'il accepte (ou non) sa demande en lui proposant un mot mystère à découvrir en un nombre maximum de coups.

Question 4. Qu'est-ce qui est donc à la charge du serveur ?

Réponse : Si le serveur accepte de « jouer », il a la charge de choisir un mot à découvrir et de le proposer au client qui a émis la demande.

Question 5. Qui du serveur ou du client fixe les règles du jeu ?

Réponse : C'est le serveur qui fixe les « règles du jeu » en déterminant et contrôlant par exemple le nombre de coups maximum pour le découvrir.

Modèle réseau

Ici, le choix se porte sur le modèle DoD pour faire fonctionner ce jeu en TCP/IP.

Question 6. Compléter le modèle DoD ci-dessous en choisissant notamment un protocole de transport ?

Couche	Protocole
Application	
Transport	
Réseau	
Interface	

Question 7. Dans ce modèle, que reste-t-il à définir ?

Réponse : Le protocole de la couche Application qui implémente le service « jeu du pendu ».

Structure des messages de la couche Application

Le travail consiste à élaborer la **structure des messages échangés** par le client et le serveur ainsi que leur **séquencement**.

Question 8. Le protocole à mettre en oeuvre sera-t-il orienté caractère ou orienté bit ?

Réponse : Ici, il est préférable d'élaborer un protocole orienté caractère.



On conservera les délimiteurs utilisés par les autres protocoles connus de la couche Application (HTTP, FTP, ...) : l'espace comme séparateur de champ et "`\r\n`" comme délimiteur de fin de message.



Évidemment, on distinguera les messages envoyés par le client de ceux envoyés par le serveur.

On vous fixe les contraintes suivantes dans les messages échangés :

- tous les champs seront échangés sous la forme de caractères
- toutes les informations de protocoles seront échangées en MAJUSCULE
- toutes les lettres et le mot seront échangées en minuscule



Le PCI d'un message pourra avoir une longueur fixe car c'est plus simple à gérer.

Côté client

Dans le message qu'il envoie au serveur, le client doit préciser le type de sa requête (ou si vous préférez de sa demande). Le client peut par exemple :

- demander à jouer une nouvelle partie
- proposer une lettre ou un mot
- abandonner la partie en cours
- ...

Question 9. Définissez précisément la structure et les champs d'un message envoyé par le client.



N'oubliez pas d'indiquer les valeurs attendues et valides des différents champs composant un message.

Côté serveur

Par principe, le serveur reçoit toujours des requêtes (ou si vous préférez des demandes) en provenance du client.

Il est plus prudent de considérer que le serveur peut recevoir des demandes valides ou invalides. Le serveur répondra donc toujours en précisant un **code réponse**.

Les codes réponses seront formatés sur trois caractères (**xxx**) dont les significations sont :

- Le premier chiffre indique le statut de la réponse (1 pour succès ou 2 pour échec)
- Le second chiffre et le troisième chiffre précise un code d'erreur (00 à 99) ou 00 en cas de succès

Si le serveur répond avec succès à une demande du client, il renverra toujours un code réponse **1xx** sinon il renverra un code erreur compris entre 200 à 299.

Le code réponse permettra aussi de gérer l'état de la partie au final. On peut fixer par exemple les code suivants :

- 101 : partie gagnée par le client (le mot a été découvert en un nombre de coups inférieur au maximum fixé)
- 102 : partie perdue par le client (le mot n'a pas été découvert et un nombre maximum de coups a été atteint)
- 103 : partie abandonnée par le client (est-ce que le mot doit être dévoilé?)

Question 10. Définissez précisément les codes erreurs renvoyés par le serveur.

Question 11. Définissez maintenant la structure et les champs d'un message envoyé par le serveur.



N'oubliez pas d'indiquer les valeurs attendues et valides des différents champs composant un message.

Diagrammes des échanges

Question 12. Définissez précisément les diagrammes qui décrivent : une partie gagnée, perdue, abandonnée par le client.

Tests de validation

Question 13. En utilisant les outils `netcat` et `telnet`, procéder aux tests de validation du protocole défini par l'équipe.



Puis faire valider son protocole par votre enseignant lors d'une réunion d'équipe.

Deuxième partie : mise en oeuvre sous Qt

Le module QtNetwork

Le module `QtNetwork` offre des classes qui vous permettent d'écrire vos propres clients et serveurs TCP/IP.

Pour inclure les déclarations des classes de ce module, il vous faut utiliser la directive suivante :

```
#include <QtNetwork>
```

Pour disposer ce module, il vous faut ajouter cette ligne à votre fichier de projet `.pro` :

```
QT += network
```



N'oubliez pas de refaire un `qmake` lorsque vous modifier un fichier de projet `.pro`.

Les classes fournies

Qt fournit de nombreuses classes pour la programmation réseau (cf. la documentation de Qt).

En résumé, on disposera :

- des classes comme `QFtp` pour les protocoles de la couche Application
- des classes de plus bas niveau comme `QTcpSocket`, `QTcpServer` et `QUdpSocket`
- des classes de plus haut niveau pour une gestion simplifiée du réseau comme `QNetworkConfiguration`, `QNetworkConfigurationManager`, ...

Ici, on aura juste besoin des classes de base pour l'implémentation des *sockets* sous Qt soit :

La classe `QTcpSocket` fournit une interface pour le protocole TCP. On peut donc utiliser `QTcpSocket` pour implémenter des protocoles réseau standard comme POP et SMTP, aussi bien que des protocoles personnalisés.

Présentation de la classe QTcpSocket

Rappel : la programmation Qt est basée sur le mécanisme des signaux et des slots.

La classe `QTcpSocket` est **asynchrone** et émet des signaux pour reporter des changements de statuts et des erreurs. Elle repose sur une boucle d'événements pour détecter des données arrivantes et automatiquement envoyer les données partantes.

Les signaux qu'il faut au minimum gérer par `connect()` côté client :

- `readyRead()` signale que des données ont été reçues et sont prêtes à être lues
- `connected()` signale que la socket est dans l'état connecté
- `disconnected()` signale que la socket est dans l'état déconnecté
- `error()` signale qu'une erreur s'est produite sur la socket

Vous pouvez écrire des données dans le socket avec `QTcpSocket::write()` et en lire avec `QTcpSocket::read()`.



`QTcpSocket` possède deux flux indépendants de données : un pour écrire et l'autre pour lire.

Puisque `QTcpSocket` hérite de `QIODevice`, vous pouvez l'utiliser avec `QTextStream` et `QDataStream` pour gérer vos données.

En lisant depuis un `QTcpSocket`, vous devez être sûr qu'assez de données sont disponibles en appelant `QTcpSocket::bytesAvailable()` avant.

On appellera la méthode non bloquante `QTcpSocket::connectToHost()` pour se connecter à un serveur. Une connexion TCP doit être établie vers un **hôte distant** et un **port** avant que le transfert de données puisse commencer. Une fois la connexion établie, l'adresse IP et le port du point de communication distant sont disponibles avec les fonctions `QTcpSocket::peerAddress()` et `QTcpSocket::peerPort()`. À n'importe quel moment, le point de communication distant peut fermer la connexion et le transfert de données s'arrêtera immédiatement.

Mise en oeuvre de la classe QTcpSocket

Pour la mise en oeuvre de la programmation socket sous Qt, on va partir sur un exemple construit par itérations sans utiliser de composants graphiques.

Remarque : un développement itératif s'organise en une série de développement très courts de durée fixe nommée itérations. Le résultat de chaque itération est un système partiel exécutable, testé et intégré (mais incomplet).

Étape n°1 : création de la socket et connexion au serveur

On commence par créer un fichier `clientTCP-1.pro` avec le contenu suivant :

```
QT += core network
TEMPLATE = app
TARGET =
DEPENDPATH += .
INCLUDEPATH += .
HEADERS += client-1.h
SOURCES += client-1.cpp clientTCP-1.cpp
```

Le fichier de projet `clientTCP-1.pro`

L'exemple est basé sur la création d'une classe `Client`. Pour bénéficier des fonctionnalités Qt, la classe `Client` héritera de la classe de base `QObject` et intégrera la macro `Q_OBJECT` pour bénéficier du mécanisme *signal/slot*.

```
#ifndef CLIENT_H
#define CLIENT_H

#include <QObject>
#include <QtNetwork>
#include <QDebug>

#define PORT_SERVEUR 5000

class Client : public QObject
{
    Q_OBJECT

public:
    Client( QObject *parent=0 );
    ~Client();
    void demarrer();

private:
    QTcpSocket *tcpSocket; // la socket
};

#endif
```

client-1.h

Dans cette première étape, on réalisera :

1. l'instanciation d'un objet de la classe `QTcpSocket`
2. la connexion au serveur avec la méthode `connectToHost()`

Remarque : Qt fournit une classe `QHostAddress` pour manipuler l'adresse IP d'une machine.

La définition de la classe `Client` sera donc la suivante :

```
#include "client-1.h"

Client::Client( QObject *parent/*=0*/ ) : QObject( parent )
{
    qDebug() << QString::fromUtf8("Instancie un objet QTcpSocket");
    tcpSocket = new QTcpSocket(this); // 1.
    qDebug() << QString::fromUtf8("État de la socket :") << tcpSocket->state();
}

Client::~Client()
{
    qDebug() << QString::fromUtf8("Ferme la socket");
    tcpSocket->close();
}

void Client::demarrer()
{
    qDebug() << QString::fromUtf8("Connexion au serveur");
    //tcpSocket->connectToHost(QHostAddress((QString)"0.0.0.0"), PORT_SERVEUR); // remplacer
    // 0.0.0.0 par l'adresse IP du serveur
    tcpSocket->connectToHost(QHostAddress::LocalHost, PORT_SERVEUR); // 2. (ici le serveur
    // sera localhost)
    qDebug() << QString::fromUtf8("État de la socket :") << tcpSocket->state();
}
```

client-1.cpp

Il ne reste plus qu'à écrire la fonction principale du programme de test :

```
#include <QApplication>

#include "client-1.h"

int main( int argc, char **argv )
{
    QApplication a( argc, argv );

    Client client;

    client.demarrer();

    return 0;
}
```

clientTCP-1.cpp

La procédure de test est assez simple. Il suffit de faire successivement :

```
$ qmake
```

```
$ make
```

```
$ ./clientTCP-1
```

```
"Instancie un objet QTcpSocket"  
"État de la socket :" QAbstractSocket::UnconnectedState  
"Connexion au serveur"  
"État de la socket :" QAbstractSocket::ConnectingState  
"Ferme la socket"
```

Étape n°2 : vérification du bon fonctionnement de la connexion

Dans cette deuxième étape, on va intégrer les fonctionnalités propres à Qt en utilisant le mécanisme *signal/slot*.

On commence par créer un fichier `clientTCP-2.pro` avec le contenu suivant :

```
QT += core network  
TEMPLATE = app  
TARGET =  
DEPENDPATH += .  
INCLUDEPATH += .  
HEADERS += client-2.h  
SOURCES += client-2.cpp clientTCP-2.cpp
```

Le fichier de projet clientTCP-2.pro

*Rappel : La classe QTcpSocket est **asynchrone** et émet des signaux pour reporter des changements de statuts et des erreurs. Elle repose sur une boucle d'événements pour détecter des données arrivantes et automatiquement envoyer les données partantes.*

Les signaux qu'il faudra au minimum gérer côté client sont :

- `readyRead()` signale que des données ont été reçues et sont prêtes à être lues
- `connected()` signale que la socket est dans l'état connecté
- `disconnected()` signale que la socket est dans l'état déconnecté
- `error()` signale qu'une erreur s'est produite sur la socket

On va donc créer les *slots* suivants dans la classe `Client` :

```
#ifndef CLIENT_H  
#define CLIENT_H  
  
#include <QObject>  
#include <QtNetwork>  
#include <QDebug>  
  
#define PORT_SERVEUR 5000  
  
class Client : public QObject  
{  
    Q_OBJECT  
  
public:  
    Client( QObject *parent=0 );  
    ~Client();  
    void demarrer();  
  
private:  
    QTcpSocket *tcpSocket;  
  
private slots:
```

```

    void estConnecte(); // pour le signal connected()
    void estDeconnecte(); // pour le signal disconnected()
    void recevoir(); // pour le signal readyRead()
    void gererErreur(QAbstractSocket::SocketError erreur); // pour le signal error()
};

#endif

```

client-2.h

La définition de la classe Client sera donc la suivante en intégrant la connexion des *slots* avec `connect()` :

```

#include "client-2.h"

Client::Client( QObject *parent/*=0*/ ) : QObject( parent )
{
    qDebug() << QString::fromUtf8("Instancie un objet QTcpSocket");
    tcpSocket = new QTcpSocket(this);

    // La connexion des signaux aux slots
    connect(tcpSocket, SIGNAL(readyRead()), this, SLOT(recevoir()));
    connect(tcpSocket, SIGNAL(connected()), this, SLOT(estConnecte()));
    connect(tcpSocket, SIGNAL(disconnected()), this, SLOT(estDeconnecte()));
    connect(tcpSocket, SIGNAL(error(QAbstractSocket::SocketError)), this, SLOT(gererErreur(
        QAbstractSocket::SocketError)));
}

Client::~Client()
{
    qDebug() << QString::fromUtf8("Ferme la socket");
    tcpSocket->close();
}

void Client::demarrer()
{
    qDebug() << QString::fromUtf8("Connexion au serveur");
    //tcpSocket->connectToHost(QHostAddress((QString)"0.0.0.0"), PORT_SERVEUR);
    tcpSocket->connectToHost(QHostAddress::LocalHost, PORT_SERVEUR);
}

void Client::estConnecte()
{
    qDebug() << QString::fromUtf8("Socket connectée au serveur");
}

void Client::estDeconnecte()
{
    qDebug() << QString::fromUtf8("Socket déconnectée au serveur");
}

void Client::recevoir()
{
    qDebug() << QString::fromUtf8("Des données ont été reçues en provenance du serveur");
}

```

```

void Client::gererErreur(QAbstractSocket::SocketError erreur)
{
    qDebug() << QString::fromUtf8("Une erreur s'est produite sur la socket : ") << tcpSocket
        ->errorString();
    qDebug() << QString::fromUtf8("Erreur : ") << erreur;

    // Il est possible de personnaliser le traitement de l'erreur
    // liste des erreurs : http://doc.qt.digia.com/qt/qabstractsocket.html#SocketError-enum
    /*switch(erreur)
    {
        case QAbstractSocket::HostNotFoundError:
            // ...
            break;
        case QAbstractSocket::ConnectionRefusedError:
            // ...
            break;
        case QAbstractSocket::RemoteHostClosedError:
            // ...
            break;
        default:
            // ...
    }*/
}

```

client-2.cpp

Il ne reste plus qu'à modifier la fonction principale du programme de test pour lui demander de boucler sur la gestions des évènements en appelant `a.exec()` :

```

#include <QApplication>

#include "client-2.h"

int main( int argc, char **argv )
{
    QApplication a( argc, argv );

    Client client;

    client.demarrer();

    return a.exec(); // permettra de traiter les signaux
}

```

clientTCP-2.cpp

On va réaliser trois tests pour valider cette deuxième itération :

```

$ qmake
$ make

```

```

// Test n°1 : il n'y a pas de serveur en écoute sur le port 5000
$ ./clientTCP-2
"Instancie un objet QTcpSocket"
"Connexion au serveur"

```

```
"Une erreur s'est produite sur la socket : " "Connection refused"
"Erreur : " QAbstractSocket::ConnectionRefusedError
^C

// Test n°2 : il y a un serveur (netcat) en écoute sur le port 5000 et celui envoie des
données
$ ./clientTCP-2
"Instancie un objet QTcpSocket"
"Connexion au serveur"
"Socket connectée au serveur"
"Des données ont été reçues en provenance du serveur"
^C

// Test n°3 : il y a un serveur (netcat) en écoute sur le port 5000 mais celui-ci ferme la
connexion
$ ./clientTCP-2
"Instancie un objet QTcpSocket"
"Connexion au serveur"
"Socket connectée au serveur"
"Une erreur s'est produite sur la socket : " "The remote host closed the connection"
"Erreur : " QAbstractSocket::RemoteHostClosedError
"Socket déconnectée au serveur"
^C
```

Étape n°3 : échange des données

Commencez par créer la version 3 en renommant les fichiers (.pro, .h et .cpp).

On rappelle qu'une communication TCP est bidirectionnelle *full duplex* et orientée flux d'octets. Il nous faut donc des fonctions pour écrire (envoyer) et lire (recevoir) des octets dans la socket.



Normalement les octets envoyés ou reçus respectent un protocole de couche APPLICATION. Ici, pour les tests, notre couche APPLICATION sera vide ! C'est-à-dire que les données envoyées et reçues ne respecteront aucun protocole et ce seront de simples caractères ASCII.

Vous pouvez écrire des données dans le socket avec `QTcpSocket::write()` et en lire avec `QTcpSocket::read()` ou `QTcpSocket::readAll()`.

Faire communiquer deux processus sans aucun protocole de couche APPLICATION est tout de même difficile ! On va simplement fixer les règles d'échange suivantes :

- le client envoie en premier une chaîne de caractères une fois connectée
- et le serveur lui répondra "ok"

Ici, on va simplement utiliser le type `QByteArray` pour l'échange des données. Si vous implémentez un protocole personnalisé pour la couche Application, vous pouvez utiliser aussi `QTextStream` ou `QDataStream` pour gérer vos données échangées.

La définition de la classe `Client` sera donc la suivante en intégrant une amélioration (affichage des informations sur les points de communication local et distant) et deux ajouts (l'émission et la réception de données) :

```
#include "client-3.h"

Client::Client( QObject *parent/*=0*/ ) : QObject( parent )
{
    qDebug() << QString::fromUtf8("Instancie un objet QTcpSocket");
    tcpSocket = new QTcpSocket(this);

    connect(tcpSocket, SIGNAL(readyRead()), this, SLOT(recevoir()));
    connect(tcpSocket, SIGNAL(connected()), this, SLOT(estConnecte()));
    connect(tcpSocket, SIGNAL(disconnected()), this, SLOT(estDeconnecte()));
    connect(tcpSocket, SIGNAL(error(QAbstractSocket::SocketError)), this, SLOT(gererErreur(
        QAbstractSocket::SocketError)));
}

Client::~Client()
{
    qDebug() << QString::fromUtf8("Ferme la socket");
    tcpSocket->close();
}

void Client::demarrer()
{
    // Par précaution, on désactive les connexions précédentes
    tcpSocket->abort();

    qDebug() << QString::fromUtf8("Connexion au serveur");
    //tcpSocket->connectToHost(QHostAddress((QString)"0.0.0.0"), PORT_SERVEUR);
    tcpSocket->connectToHost(QHostAddress::LocalHost, PORT_SERVEUR);
}

void Client::estConnecte()
{
    qDebug() << QString::fromUtf8("Socket connectée au serveur");

    qDebug() << QString::fromUtf8("Adresse IP du client : ") + tcpSocket->localAddress().
        toString();
    qDebug() << QString::fromUtf8("Numéro de port du client : ") + QString::number(tcpSocket
        ->localPort());
    qDebug() << QString::fromUtf8("Adresse IP du serveur : ") + tcpSocket->peerAddress().
        toString();
    qDebug() << QString::fromUtf8("Numéro de port du serveur : ") + QString::number(tcpSocket
        ->peerPort());

    QByteArray message("Hello world !\n"); // représente le message de la couche Application
    qint64 ecrits = -1;

    // Envoie du message
    ecrits = tcpSocket->write(message);
    switch(ecrits)
```

```
{
    case -1 : /* une erreur ! */
        qDebug() << QString::fromUtf8("Erreur lors de l'envoi !"); break;
    default: /* envoi de n octets */
        qDebug() << QString::fromUtf8("Message envoyé : ") << message;
        qDebug() << QString::fromUtf8("Octets envoyés : ") << ecrits;
        qDebug() << QString::fromUtf8("Message envoyé avec succès !");
    }
}

void Client::estDeconnecte()
{
    qDebug() << QString::fromUtf8("Socket déconnectée au serveur");
}

void Client::recevoir()
{
    QByteArray message;

    if (tcpSocket->bytesAvailable() < 0)
        return;

    message = tcpSocket->readAll();

    qDebug() << QString::fromUtf8("Des données ont été reçues en provenance du serveur");
    qDebug() << QString::fromUtf8("Octets reçus : ") << message.size();
    qDebug() << QString::fromUtf8("Message reçu du serveur : ") << message;
}

void Client::gererErreur(QAbstractSocket::SocketError erreur)
{
    qDebug() << QString::fromUtf8("Une erreur s'est produite sur la socket : ") << tcpSocket
        ->errorString();
    qDebug() << QString::fromUtf8("Erreur : ") << erreur;

    // Il est possible de personnaliser le traitement de l'erreur
    // liste des erreurs : http://doc.qt.digia.com/qt/qabstractsocket.html#SocketError-enum
    /*switch(erreur)
    {
        case QAbstractSocket::HostNotFoundError:
            // ...
            break;
        case QAbstractSocket::ConnectionRefusedError:
            // ...
            break;
        case QAbstractSocket::RemoteHostClosedError:
            // ...
            break;
        default:
            // ...
    }*/
}
```

client-3.cpp

On termine par un test de validation :

```
$ qmake
$ make
```

```
// on utilise en parallèle un serveur netcat sur le port 5000 :
$ ./clientTCP-3
"Instancie un objet QTcpSocket"
"Connexion au serveur"
"Socket connectée au serveur"
"Adresse IP du client : 127.0.0.1"
"Numéro de port du client : 39059"
"Adresse IP du serveur : 127.0.0.1"
"Numéro de port du serveur : 5000"
"Message envoyé : " "Hello world !"
"
"Octets envoyés : " 14
"Message envoyé avec succès !"
"Des données ont été reçues en provenance du serveur"
"Octets reçus : " 3
"Message reçu du serveur : " "ok"
"
^C
```

Pour aller plus loin sur la programmation réseau sous Qt, il vous faudra consulter la documentation officielle :

- Le module *network* (doc.qt.digia.com/qt/qtnetwork.html et (qt.developpez.com/doc/4.7/programmation-reseau/)
- La classe `QTcpSocket` (doc.qt.digia.com/qt/qtcpsocket.html)
- La classe `QAbstractSocket` (doc.qt.digia.com/qt/qabstractsocket.html)
- La classe `QIODevice` (doc.qt.digia.com/qt/qiodevice.html)
- La classe `QByteArray` (doc.qt.digia.com/qt/qbytearray.html)
- La classe `QDataStream` (doc.qt.digia.com/qt/qdatastream.html)
- La classe `QTextStream` (doc.qt.digia.com/qt/qtextstream.html)

On peut conseiller de regarder les exemples fournis par Qt (et principalement pour commencer le client/serveur Fortune) :

- Tous les exemples (doc.qt.digia.com/qt/examples-network.html)
- Le client Fortune (doc.qt.digia.com/qt/network-fortuneclient.html)
- Le serveur Fortune (doc.qt.digia.com/qt/network-fortuneserver.html)

Troisième partie : réalisation du programme

Question 14. Écrire les programmes demandés. Une version graphique du client peut être réalisé.