Travaux Dirigés n°3 programmation réseau avec QT

Côté serveur

1. OBJECTIF

Utilisation des sockets avec la bibliothèque QT

Comprendre le fonctionnement des sockets en mode événementiel. En effet, les logiciels actuels (visual studio, QT…) utilisent le système d'exploitation pour répondre aux sollicitations externes (les événements). Lorsqu'un événement survient, le système appelle une fonction prévue et déclarée à cet effet.

Points techniques abordés :

- Codage de l'information
- Programmation sockets avec QT
- Communication réseau

2. CONDITIONS DE RÉALISATION

Travail sur micro-ordinateurs avec QT Creator

Un client est disponible

3. RESSOURCES

Les classes socket dans la technologie QT, consulter le site http://doc.qt.io/qt-5/qtnetwork-index.html et plus particulièrement la classe, ***QTcpServer***.

4. LE BESOIN

Les techniciens réseau sont souvent appelés pour une défaillance sur un poste informatique. Avant de se déplacer, il serait intéressant de connaître certaines caractéristiques de la machine. Pour cela, un petit programme « Serveur » implanté sur chaque machine que le technicien « Client » pourra interroger est une aide précieuse. Le travail proposé ici permet de définir et de coder ce client.

4.1. Analyse et conception

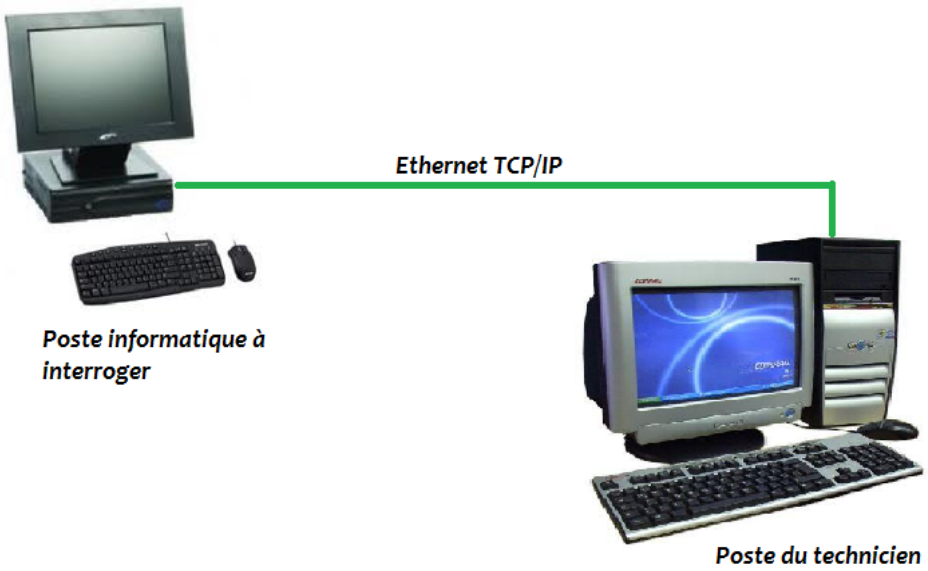
Le serveur est capable de fournir les informations de la machine à l'administrateur client distant.

Selon la demande de celui-ci, le serveur envoie les informations correspondantes :

Demande	*Commande*	*Réponse attendue*
Nom de l'utilisateur	"u"	Le nom de l'utilisateur connecté
Nom de la machine	"c"	Le nom de la machine
Système d'exploitation	"o"	Le type de système d'exploitation
L'architecture du processeur	"a"	Le type de processeur x86 ou amd64 par exemple

D'autres commandes pourront être ajoutées par la suite.

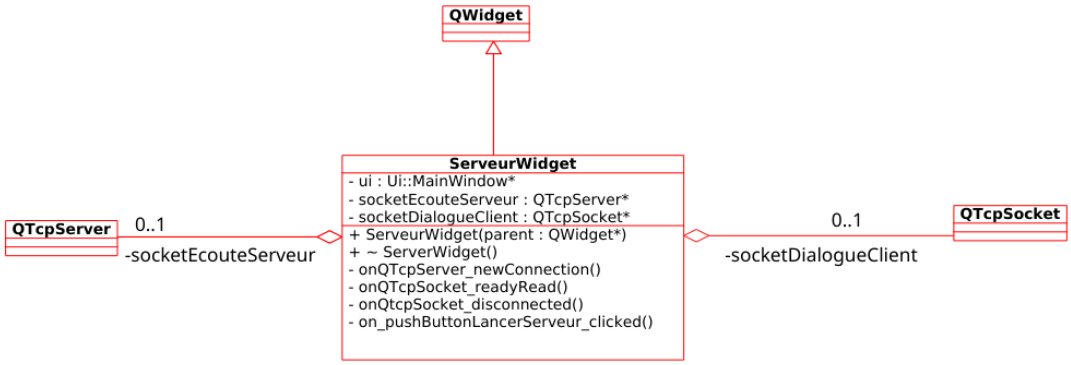
4.2. Mise en situation



5. RÉALISATION DU SERVEUR

5.1. Création du projet

Créez un projet de type Application graphique en C++ sous QT5 avec QT Creator. La classe principale ce nomme ***ServeurWidget*** elle hérite de ***QWidget*** comme le montre le diagramme de classes ci-dessous.



De même, ajoutez l'attribut réalisant la liaison avec la classe ***QTcpServer*** *et* ***QTcpSocket***.

5.2. Création de l'IHM

L'interface du client aura l'aspect suivant :

1. Nommez chaque Widget suivant la convention de nommage habituelle.
2. Associez le bouton de lancement du serveur au slot ***clicked()***. Pour le bouton Quitter, il sera associé au slot ***close()*** de ***QWidget***.
3. Complétez la déclaration de la classe ***ServeurWidget*** de manière à correspondre au diagramme de classe.

5.3. Utilisation de la classe QTcpServer (version mono client)

1. Indiquez le lieu où doit être instancié l'attribut ***socketEcouteServeur*** dans la classe ***ServeurWidget***.
2. Dans quelle méthode de la classe ***ServeurSocket*** sera utilisée la méthode ***listen*** de la classe ***QTcpServer*** ?
3. Quelle méthode de la classe ***ServeurSocket*** sera appelée en cas d'une demande de connexion d'un client ?
4. A quoi correspondant le retour de la méthode ***nextPendingConnection*** de la classe ***QTcpServer*** ?

5. Codez le slot de réception des données **onQTcpSocketReadyRead**.

Vous utiliserez la méthode **readAll** pour lire les données en provenance du client.

Le traitement des demandes "u" et "c" peut se faire à l'aide de la fonction ***getenv*** et de la méthode statique ***localHostName*** de la classe ***QHostInfo*** retournant chacune un ***QString***.

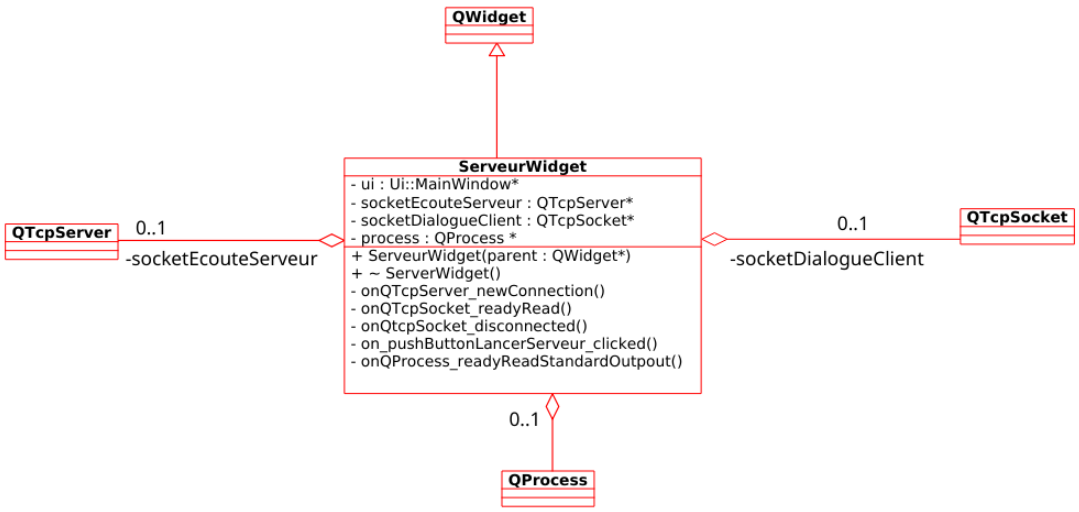
Voici comment les utiliser:

```
QString reponse;
reponse = getenv("USERNAME");
reponse = QHostInfo::localHostName();
```

Pour les demandes "o" et "a", il faut passer par l'appel d'un processus externe.

Pour cela, nous allons utiliser la classe ***QProcess*** (un peu l'équivalent de la fonction popen en C), qui permet de lancer un processus externe au programme et de récupérer le résultat sur la sortie standard.

Pour cela nous allons modifier notre application comme suit:



Le principe de fonctionnement est le que pour ***QTcpSocket***, lorsque des données seront disponibles, un signal ***readyReadStandardOutput*** est émis.

La liaison entre l'objet process et le slot associé pour ce signal se fait lorsqu'un nouveau client se connecte, donc dans le slot ***onQTcpServer_newConnection***.

Voici le code du slot appelé:

```
void ServeurWidget::onQProcess_readyReadStandardOutput()
{
    QString reponse = process->readAllStandardOutput();
    if(!reponse.isEmpty())
    {
     QString message = "Réponse envoyée à " + socketDialogueClient->peerAddress().toString()+" : " + reponse;
        ui->textEditLogs->append(message);
        socketDialogueClient->write(reponse.toLatin1());
    }
}
```

Pour lancer le process, il faut utiliser la méthode start de la classe QProcess.

Cela se fait évidemment lorsque l'on sait ce que le client a envoyé.

Voici les comment lancer les process pour "o" et pour "a":

```
process->start("uname",QStringList("-p"));
process->start("uname");
```

5.4. Version multi client

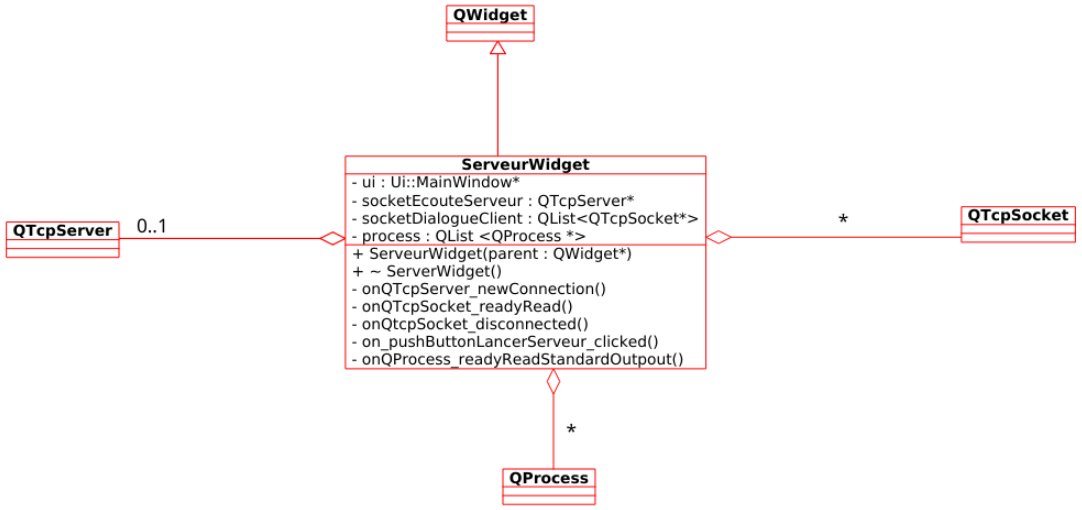
Pour pouvoir traiter un nombre de clients simultanément de façon presque illimitée, ce n'est pas un seul attribut ***QTcpSocket** *** dont nous aurons besoin, mais d'une quantité dynamique s'adaptant au nombre réel de clients connectés.

Pour cela, nous utiliserons une liste, et plus précisément une ***QList*** de ***QTcpSocket ***.

Il faudra également autant de process lançables que de client.

Nous aurons donc une ***QList*** de ***QProcess ** .

Voici notre diagramme de classe final:



Nous utiliserons la méthode append de la classe QList pour ajouter un objet à la liste.

Voici un extrait de la méthode ***onQTcpServer_newConnection*** avec les listes:

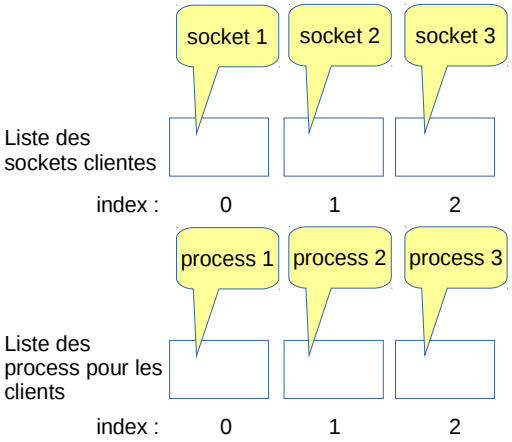
```
void ServeurWidget::onQTcpServer_newConnection()
{
    QTcpSocket *client;
    client = socketEcouteServeur->nextPendingConnection();
     connect(client,&QTcpSocket::readyRead,this,&ServeurWidget::onQTcpSocket
_readyRead);
    socketDialogueClient.append(client);
```

1. Complétez cette méthode afin de créer un process à associer au slot ***onQProcess_readyReadStandardOutput*** et que vous ajouterez à la liste des process.

Ainsi, pour chaque client se connectant, on aura un process.

Il est possible de connaitre a position d'un objet dans une QList à l'aide de la méthode **indexOf**.

Les sockets cliente et les process étant créés en même temps, les index dans chacune des listes sont les mêmes.



Lorsqu'un client envois des données, c'est toujours le slot ***OnQTcpSocket_readRead*** qui est appelé, il faut donc déterminer, quel client est à l'origine du signal.

Comme dans le TD calculatrice, on va utiliser la méthode ***sender***. Voici comment récupérer le client à l'origine du signal:

```
QTcpSocket *client=qobject_cast<QTcpSocket*>(sender());
```

Il faut, si le client a envoyé la commande "o" ou "a", lancer le bon process. Pour cela, on doit connaître l'index de ce dernier par rapport à la socket cliente. La méthode ***at*** de la classe ***QList*** nous permet de nous placer sur l'objet de trouvant à un index donné.

Voici un extrait de code illustrant notre problématique pour la commande "a".

```
QTcpSocket *client=qobject_cast<QTcpSocket*>(sender());
int indexProcess=socketDialogueClient.indexOf(client);
process.at(indexProcess)->start("uname");
```

2. Complétez la méthode ***OnQTcpSocket_readRead***.
3. Complétez la méthode ***OnQProcess_readReadStandardOutput***.

6. APPROFONDISSEMENT

Complétez votre code afin que:

- Lorsqu'un client se déconnecte, il soit retiré de la liste des sockets et que le process associé soit également retiré de la liste des process.
    - Les méthodes ***removeOne*** et ***removeAt*** de la classe ***QList*** devraient vous être d'un grand secours.