Connexions sécurisées avec QtSSLSocket

par Kinji1

Date de publication : 22 février 2009

Dernière mise à jour :

Qt Quarterly est un journal électronique disponible exclusivement aux clients Qt. Chaque trimestre, nous envoyons un e-mail qui, nous l'espérons, ajoutera à votre expérience Qt, avec des articles de qualité écrits par des experts de Qt.

Developpez.com a reçu l'autorisation de Nokia afin de traduire ces articles.

I - Qu'est-ce que le SSL ?

Par Andreas Aardal Hanssen

Les connexions réseaux sécurisées ont deux contraintes. Premièrement, vous devez être sûr que vous communiquez avec le bon correspondant. Deuxièmement, vous devez être certain que les données échangées n'ont pas été altérées ou même lues par une tierce partie ; et dans le cas où elles ont été lues, qu'elles n'ont pas pu être déchiffrées. Cette article expose comment créer un client sécurisé avec la librairie OpenSSL et la solution Qt QtSSLSocket qui prend en compte ces contraintes. (1)

SSL (Secure Sockets Layer) est le protocole standard qui fournit la sécurité au niveau réseau. SSL est une couche logicielle au dessus de la couche TCP, qui utilise des paquets TCP pour envoyer des données chiffrées. (2)

Une transmission SSL comporte trois phases : la phase de négociation (handshake), la phase d'échange des données et la phase de terminaison. Pendant la négociation, l'identité du serveur et de manière optionnelle celle du client sont vérifiées à travers l'échange de clés publiques. Les correspondants s'accordent alors sur un algorithme commun à utiliser pour le chiffrement des données lors de la phase d'échange. Les algorithmes cryptographiques sont initialisés avec des données aléatoires et la phase d'échange commence.

A ce moment vous savez à qui votre client s'est connecté (ou qui s'est connecté à votre serveur). La phase d'échnage de données est celle où les vraies données sont envoyées entre le client et le serveur. Ces données sont chiffrées avec des algorithmes tels que "DES3" ou "Blowfish". Ceux-ci sont considérés comme étant sûrs face à une écoute, à la modification ou au rejeu. Lorsque la phase d'échange de données se termine, la phase de terminaison commence. A ce moment, le client et le serveur s'assurent que chacun a fermé le canal sécurisé et que celui-ci ne transporte plus de données. Le socket de la connection est alors fermé.

Le nombre de services réseaux professionnels sur Internet qui nécessite SSL est élevé et en pleine croissance. Par exemple, vous pourriez vouloir donner des ordres d'achats ou réaliser des transactions de cartes bancaires, et pour cela les banques insisteront pour utiliser SSL. Cependant le support réseau de Qt, bien que bon, ne fournit pas de support de SSL prêt à l'emploi; en effet, il n'y a pas de méthode directe pour utiliser SSL dans des applications Qt.

Une solution évidente est de lier votre application Qt à une librairie SSL. Mais beaucoup d'entre elles sont complexes, avec des API difficiles et une documentation réduite ou pauvre. Et celles-ci nécessitent toutes une connaissance en profondeur du protocole SSL et de son implémentation. Alors que pouvez-vous faire si vous ne savez pas à quoi ressemble un certificat X.509 mais avez besoin de connexions réseau sécurisées ?

La réponse est d'utiliser QtSSLSocket de l'équipe Qt Solutions. QtSSLSocket permet d'ajouter aux applications réseaux basées sur QSocket le support cryptographique SSL. Cette classe a une interface facile à utiliser, et ne requiert aucune connaissance préalable du protocole SSL. Elle fournit un socket pour ses blocs de données, chiffre ce que vous envoyez et déchiffre ce que vous recevez de la part de votre correspondant. En utilisant QtSSLSocket vous pouvez ajouter facilement SSL à vos applications client et serveur, et avoir les coudées franches pour fournir un chiffrement SSL à des protocoles de tout niveau de complexité. Et la faible quantité de connaissance de SSL dont vous avez besoin se trouve dans la documentation.

A quel point SSL est-il sécurisé ?
Sans un modèle de communication réseau sécurisé vous ne pouvez pas être certain de l'endroit où vous vous connecter sur Internet. Les entrées DNS peuvent être contrefaites, l'hôte Internet peut être remplacé, les données transmises compromises ou écoutées et ainsi de suite. D'un autre côté, le modèle de sécurité SSL fournit une très grande garantie de l'identité de l'hôte auquel vous vous connectez, en vous permettant de vérifier son identité par une tierce partie de confiance appelée Autorité de Certification (CA). En utilisant le certificat du CA, vous pouvez vérifier l'authencité du certificat de votre correspondant, et ainsi son identité. Le mécanisme dépend bien entendu du CA auquel vous faîtes confiance.
Mais comment pouvez-vous vérifier l'identité du certificat du CA ? SSL nécessite que chaque application ait accès à un ensemble fixe de certificats pour tous les CA de

confiance. Si vous n'avez pas le certificat du CA qui a réalisé le certificat du correspondant, vous ne pouvez pas vérifier son identité. Comment un utilisateur peut-il obtenir de manière sûre un tel ensemble de certificats de CA ? Bien évidemment ils ne peuvent pas être téléchargés depuis Internet via une connexion SSL. Donc la plupart des systèmes d'exploitation modernes incluent un paquetage contenant les certificats de CA. Mais si votre système d'exploitation a été installé à partir d'un CD que vous avez acheté dans un magasin ou avez reçu par courrier, vous devez faire confiance à tous les services postaux impliqués, et aux employés du magasin, et bien sûr au fabricant du CD. Et si vous avez téléchargé votre système d'exploitation depuis Internet, êtes-vous certain que votre connexion était sécurisée ?

II - Un client simple

Réalisons un client SSL simple. Nous pouvons écrire un exemple complet en seulement 50 lignes. Commençons par la définition de notre classe *Client*.

```
class Client : public QObject
{
    Q_OBJECT

public:
    Client(const QString &host, int port);

signals:
    void responseReceived();

private slots:
    void waitForGreeting();
    void readResponse();

private:
    QtSSLSocket *socket;
};
```

Comme pour un autre client QSocket, nous créons une sous-classe avec un socket (ici le QtSSLSocket) comme membre privé.

```
Client::Client(const QString &host, int port)
{
    socket = new QtSSLSocket();
    connect(socket, SIGNAL(connected()), SLOT(waitForGreeting()));
    connect(socket, SIGNAL(readyRead()), SLOT(readResponse()));
    connect(socket, SIGNAL(connectionClosed()), qApp, SLOT(quit()));
    connect(socket, SIGNAL(delayedCloseFinished()), qApp, SLOT(quit()));
    socket->connectToHost(host, port);
}
```

Dans le constructeur nous créons notre socket SSL et connectons ses signaux, puis nous nous connectons à l'hôte.

```
void Client::waitForGreeting()
{
    qDebug("Connected; now waiting for the greeting");
}
```

Le slot waitForGreeting() est appelé quand la connection a été établie.

```
void Client::readResponse()
{
    if (socket->canReadLine()) {
        qDebug("Received: %s", socket->readLine().latin1());
        socket->close();
    }
```

```
}
```

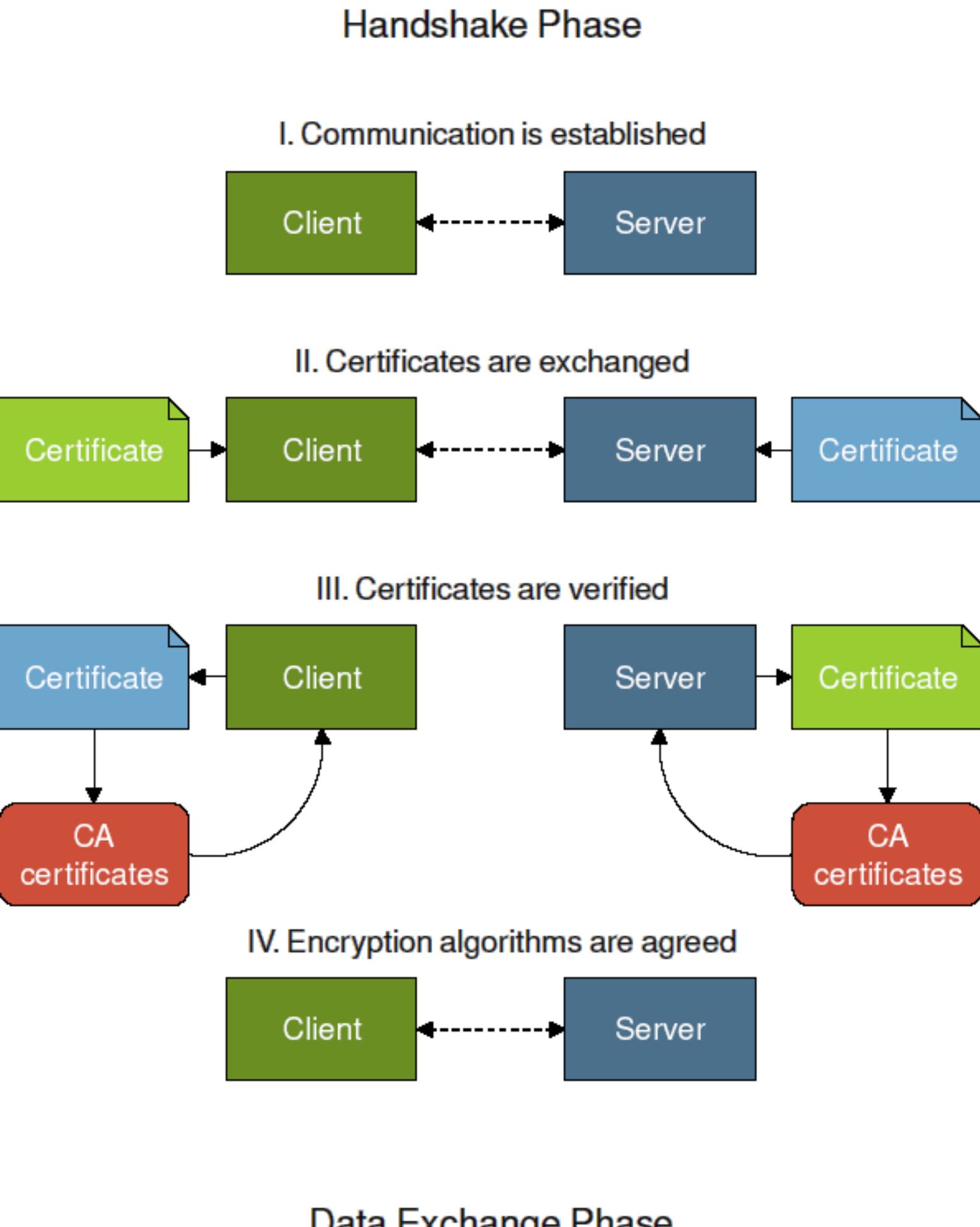
Le slot readResponse() est aussi simple qu'il le serait avec une sous-classe de QSocket normale.

Quand la connexion sera fermée, l'application se terminera. Et c'est tout ! Pour que ce soit complet, voici la fonction main. Nous avons utilisé cet exemple pour nous connecter à notre serveur IMAP, qui écoute sur le port 993 :

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv, false);
    Client client("imap", 993);
    return app.exec();
}
```

L'exécution de cette application donne le résultat suivant sur la console, celui-ci dépend du logiciel IMAP vous utilisez.

```
Connected! Now waiting for the greeting
Received: * OK imap Cyrus IMAP4 v2.1.12 server ready
```


Handshake Phase
I. Communication is established
Client
Server
II. Certificates are exchanged
Certificate
Client
Server
Certificate
III. Certificates are verified
Certificate
Client
Server
Certificate
CA certificates
CA certificates
IV. Encryption algorithms are agreed
Client
Server
Data Exchange Phase

Alors c'est tout ce qu'il fallait pour communiquer de manière sûre avec SSL ? Nous ne pouvons pas dire si la vérification de l'identité de l'hôte à réussie ou non durant la phase de négociation; il n'y a pas de code dans notre exemple qui prenne en charge cette importante fonctionnalité de sécurité. Ajoutons le code nécessaire pour corriger ce manque.

D'abord, nous spécifions l'emplacement de l'ensemble de certificats des CA. Pour cet exemple, nous avons utilisé le chemin */etc/ssl/certs*, mais cet emplacement est différent selon les systèmes d'exploitation et les distributions. Nous insérons cette ligne dans le constructeur :

```
socket->setPathToCACertDir("/etc/ssl/certs");
```

Nous ajoutons alors un slot pour rattraper les échecs de vérification des certificats.

```
private slots:
    void displayCertError(int, const QString &reason)
```

Puis nous connectons le signal de notre socket à ce slot :

```
connect(socket, SIGNAL(certCheckFailed(int, const QString &)),
                SLOT(displayCertError(int, const QString &)));
```

Maintenant nous implémentons le slot displayCertError().

```
void Client::displayCertError(int, const QString &reason)
{
    qDebug("Certificate check failed: %s", reason.latin1());
    socket->close();
}
```

L'implémentation du slot affiche un message d'erreur et ferme la connexion. Cela signifie que le certificat n'a pas pu être vérifié, la communication est arrêtée plutôt que de risquer une connexion non-sécurisée.

Voici la sortie du programme modifié :

```
Certificate check failed: Self signed certificate in certificate chain
```

C'est tout ce dont nous avons besoin pour ajouter des connexions sécurisées à nos appplications Qt.

Si vous voulez en apprendre plus, la documentation de QtSSLSocket contient un guide pour prendre en charge les clés SSL privées, les certificats et les Autorités de Certification.

Notez que QtSSLSocket dépend de OpenSSL. La bibliothèque OpenSSL est licenciée sous une licence de type Apache, qui signifie principalement que vous êtes libre de l'utiliser à des fins commerciales ou non-commerciales avec quelques conditions simples (cf. **www.OpenSSL.org**).

Tunnels SSL

*Une alternative courante à l'utilisation de SSL depuis une application consiste à utiliser un tunnel SSL comme Stunnel (disponible depuis **www.Stunnel.org**). Stunnel tourne comme un service séparé auquel votre application se connecte. Stunnel se connecte à l'hôte SSL à votre place, chiffre vos données sortantes avant de les envoyer et déchiffre les données entrantes quand elles arrivent.*
Malheureusement, un tunnel SSL n'est pas une solution générale. Il ne résout pas tous les problèmes côté client puisque chaque tunnel ne peut faire suivre les données qu'à un seul serveur alors que votre application nécessite souvent des envois à différents serveurs. De même, bien qu'il fonctionne correctement avec des protocoles client-serveur à connexion unique comme HTTP, il ne peut vous aider à sécuriser FTP, où le port sur lequel vous écoutez les connexions entrantes n'est seulement connu qu'à l'exécution. Mais plus

important, un tunnel SSL nécessite que vous fassiez tourner un serveur séparé à côté de votre application. Ce qui signifie que vous devez distribuer des logiciels supplémentaires, et pour des applications commerciales, cela peut ajouter un coût pour les licences.

1 : Depuis cet article, le support du protocole SSL a été intégré au sein de la bibliothèque Qt (à partir de la version 4.3) au travers de la classe QSslSocket. L'utilisation de QtSSLSocket n'est donc plus nécessaire et le code fourni en exemple dans cet article doit être légèrement adapté si vous voulez l'utiliser avec cette nouvelle classe. Le reste de l'article reste cependant valable et pertinent.
2 : Puisque les datagrammes UDP ne sont pas suffisamment fiables, le protocole SSL ne supporte pas UDP.