Un updater avec Qt : vérification du besoin de mise à jour par des fichiers XML

par Thibaut Cuvelier (Site web) (Blog)

Date de publication : 12/07/2009

Dernière mise à jour : 14/10/2009

Dans cet article, nous allons réutiliser les compétences acquises dans **le précédent** (la récupération de fichiers sur Internet) pour vérifier que le client a bien besoin de cette mise à jour. En effet, tous les clients ne disposent pas de l'**ADSL**, et nous voulons écomiser le peu de bande passante qu'ils ont.

Ceci passera par la récupération d'un fichier XML, généré par le serveur, qui contiendra le nom de fichier, sa date de modification, ainsi qu'un hash cryptographique, pour vérifier que le client dispose bien du dernier fichier en date, et qu'il ne l'a pas modifié.

N'hésitez pas à commenter cet article !

I - Le script, côté serveur....3
II - Analyse des données XML....4
II-A - Création de l'arbre DOM....4
II-B - Utilisation de l'arbre....5
III - Récupération des données du client....5
III-A - Dates....5
III-B - Foreach....7
III-C - Hash....8
III-D - Petite optimisation....8

I - Le script, côté serveur

Ce script, écrit en PHP car ce langage est fortement répandu parmi les serveurs, ne sera pas beaucoup détaillé, car il n'est pas l'objet de cet article.

```php
<?php

$dir = opendir('./');
$i;
$list = array();

while($file = readdir($dir))
{
    if ($file != 'index.php' && $file != '.' && $file != '..')
    {
        $list[$i] = array ( 'filename'    =>            $file , 'md5'         =>  md5_file ($file),
                            'sha1'        => sha1_file ($file), 'raw_date'    => filemtime ($file)
  );
        ++$i;
    }
}

closedir($dir);

$result = new XmlWriter ();
$result->openMemory();
$result->startDocument ('1.0', 'utf-8');
$result->startElement ('files');

foreach ($list as $file)
{
    $result->startElement ('file');
    $result->writeElement ('filename', $file['filename']);
    $result->writeElement ('md5', $file['md5']);
    $result->writeElement ('sha1', $file['sha1']);
    $result->writeElement (  'date',  date ('d/m/Y/H/i/s', $file['raw_date'])  );
    $result->endElement();
}

$result->endElement();
$result->endDocument();
echo $result->outputMemory();
?>
```

Voici un exemple de sortie produite par ce script.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<files>
    <file>
        <filename>calques_couleur.png</filename>
        <md5>8f7c5d06a16faa940e6b0d29d18fd8ef</md5>
        <sha1>b7d23a540a7cda882dab7bf2f7c1207365e82ec1</sha1>
        <date>24/08/2009/20/07/00</date>
    </file>
    <file>
        <filename>calques_opacite.png</filename>
        <md5>af0e79070f3c3cb3cae559a8d509af9f</md5>
        <sha1>031775fbbb45c154f7c9911f3b482ecf5a9c280f</sha1>
        <date>24/08/2009/20/07/00</date>
    </file>
</files>
```

Hashage cryptographiques : sont-ils sécurisés ? *Les algorithmes de hashage cryptographique utilisés sont dépassés, et ne permettent en aucun cas une sécurité, même faible, lorsqu'ils sont utilisés seuls. Toute machine actuelle vendue dans le*

commerce peut les compromettre en moins d'une minute : il suffit de trouver deux fichiers qui ont le même hash.
Cela signifie qu'il est possible que deux versions de votre logiciel aient un même hash. C'est pourquoi je l'ai doublé : il est nettement moins probable que les deux hashes correspondent entre les versions.
*Je n'ai pas pu, non plus, utiliser d'autres algorithmes, **MD5** et SHA1 étant les seuls supportés nativement par PHP.*
Cependant, d'autres solutions existent (comme l'extension Mhash ou le module PECL Hash). Ces deux exemples supportent bien d'autres algorithmes de hashage, mais le premier est lent, et les deux ne sont pas automatiquement activés lors de l'installation de PHP.
De plus, pour pouvoir les utiliser du côté Qt, il faut également une autre librairie, dédiée à la cryptographie. En particulier, QCA, qui se base sur Qt.

Pourquoi utiliser le PHP ?
Non, pas par préférence personnelle !
Tout simplement parce que ce langage de script est disponible sur de nombreuses plateformes, est souvent supporté chez les hébergeurs, et ressemble au C++ (tant par la syntaxe que par le modèle objet).
De plus, il dispose de quelques fonctions (basiques) de hash cryptographique et un API simple, inspirée de SAX, pour le XML : pas besoin de livrer quelque librairie supplémentaire à côté du script pour obtenir quelque chose de fonctionnel.
Finalement, pour ceux qui n'ont pas de serveur supportant un langage dynamique, on peut télécharger PHP, lui passer ce script en paramètre et diriger la sortie vers un fichier XML, qu'il suffit de mettre sur le serveur. Cette solution n'est pas simple, elle est juste fonctionnelle.

SAX ? DOM ?
Ici, le modèle SAX a été utilisé pour la lecture. Il existe un autre modèle pour la lecture de fichiers XML : DOM. Celui-ci sera utilisé plus bas, et y sera donc présenté.
SAX propose de lire le fichier petit à petit, en appelant des fonctions à chaque catégorie syntaxique. Pour l'écriture, le principe est semblable : on appelle une fonction pour chaque catégorie syntaxique.

II - Analyse des données XML

II-A - Création de l'arbre DOM

Nous avons vu, dans **la précédente partie**, comment télécharger un fichier sur Internet, et le récupérer dans un périphérique **QIODevice**. Précisément, dans une réponse **QNetworkReply**.

Le fichier que nous allons récupérer est produit par le script montré ci-dessus. Il s'agit donc d'un fichier XML. Nous allons le lire par un arbre DOM.

DOM est un standard pour la représentation de fichiers XML, défini par le **W3C**. Il commence par charger l'entièreté du fichier XML, puis le place dans un arbre, l'arbre DOM. C'est cet arbre qui sera manipulé : lu, écrit, modifié... Si le fichier XML est énorme (plusieurs centaines de mégaoctets), il est strictement impossible d'utiliser DOM pour le lire : on doit se rabattre sur le SAX, qui convient aussi bien dans cette situation.

Le SAX a été utilisé dans le code PHP précédent, mais on aurait aussi pu utiliser un technique DOM. Cependant, un serveur est souvent plus limité quant à ses capacités de calcul (il doit les diviser entre tous les clients) : il vaut mieux alors utiliser SAX. Ce dernier est aussi disponible avec Qt, et les classes **QXmlSimpleReader** et **QXmlStreamWriter**.

Premièrement, il faut créer un arbre DOM vide et lui indiquer un **QIODevice** à partir duquel il doit se charger.

```
QDomDocument doc( "files" );

reply = manager.get (request);

doc.setContent( reply );
```

II-B - Utilisation de l'arbre

Ensuite, il faut trouver l'élément racine (dans notre cas, <files>), et vérifier que son nom correspond à ce que nous attendons.

```
QDomElement root = doc.documentElement();
if( root.tagName() != "files" )
    return -3;
```

Ensuite, on recherche tous les fichiers potentiellement à télécharger (dans leurs balises <file>), ainsi que tous leurs attributs, et on les stocke dans une structure.

```
QVector  <  QMap <QString, QString>  >  files;

QDomNode n = root.firstChild();

int i = 0;

while( ! n.isNull() )
{
    QDomNode e = n.toElement();
    if( ! e.isNull() && e.tagName() == "file" )
    {
        QDomNodeList file = e.childNodes();

        files[i]    = QMap <QString, QString> ();
        files[i]["filename"] = file.item(0).toElement().text();
        files[i]["md5"]      = file.item(1).toElement().text();
        files[i]["sha1"]     = file.item(2).toElement().text();
        files[i]["date"]     = file.item(3).toElement().text();
    }

    n = n.nextSibling();
    ++i;
}
```

III - Récupération des données du client

III-A - Dates

Pour pouvoir récupérer les données du client, qui ne sont que des informations concernant des fichiers, une classe est spécialement prévue dans Qt : **QFileInfo**.

Mais, avant d'éviter du code catastrophique au niveau des performances, et avant même d'utiliser cette classe, je vous propose un petit code, très facilement compréhensible, qui permet de récuperer un tel objet pour chaque fichier d'**un dossier**.

```
QDir dir("./");
foreach(QFileInfo &fileInfo, dir.entryInfoList())
{
    fileInfo->isBundle();
}
```

foreach ?

Il s'agit d'une extension au langage C++ définie par Qt. Il est implémenté à l'aide du préprocesseur. Il est décrit un peu plus loin dans l'article, dans la section éponyme, la suivante.

Premièrement, essayons de récupérer le nom de fichier. Voici la solution qui semble coller :

```
fileInfo->fileName()
```

Cependant, un petit tour dans **la FAQ** nous indique que cette fonction retourne ce genre de résultat :

```
QString ("archive.tar.gz")
```

C'est donc parfaitement ce qui nous convient !

Deuxièmement, nous avons besoin de récupérer la date de dernière modification du fichier.

```
fileInfo->lastModified()
```

Et voilà ! Nous avons ce qu'il nous faut, à l'exception des hash.

Cependant, il y a un petit problème : comment récupérer **la date** et **l'heure** de modification, vu que **l'objet retourné** les mélange ?

```
QDateTime datetime ( /**/ );

QDate date = datetime.date() ;
QTime time = datetime.time() ;

int year  = date.year  ();
int month = date.month ();
int day   = date.day   ();
int hour  = time.hour  ();
int min   = time.minute(); // Certains clients FTP refusent d'envoyer un heure de modification plus
 précise que la minute
int sec   = time.second();
int ms    = time.msec  ();
```

Maintenant, on peut récupérer chaque élément qui nous intéresse séparément. Par le plus grand des hasards, existerait-il un moyen qui permettrait de directement convertir les données dans le format qui nous intéresse (JJ/MM/AAAA/hh/mm/ss) ?

Il y a moyen de le faire pour des **QDate** et pour des **QTime** : pourquoi ne pourrait-on pas le faire pour **QDateTime** ?

Comment le permettent-ils ?
*Ces objets proposent la fonction toString, qui prend en paramètre l'argument format, qui permet de formater ces données. Les valeurs qui suivent peuvent être utilisées pour ce paramètre. La première liste liste les paramètres pour **QDate**, la seconde, pour **QTime**. **QDateTime** les accepte tous.*

Les sigles utilisables pour QDate

- d : le numéro du jour (1 à 31),
- dd : le numéro du jour écrit avec deux chiffres (01 à 31),
- ddd : l'abréviation du jour ('Lun' à 'Dim'), utilise **QDate::shortDayName()**,
- dddd : le jour ('Lundi' à 'Dimanche'), utilise **QDate::longDayName()**,
- M : le numéro du mois (1 à 12),
- MM : le numéro du mois écrit avec deux chiffres (01 à 12),
- MMM : l'abréviation du mois ('Jan' à 'Dec'), utilise **QDate::shortMonthName()**,
- MMMM : le mois ('Janvier' à 'Décembre'), utilise **QDate::longMonthName()**,

- yy : les deux derniers chiffres de l'année (00 à 99),
- yyyy : l'année en quatre chiffres (peut être négatif, - est alors ajouté),

Les sigles utilisables pour QTime

- h : l'heure (0 à 23 ou 1 à 12 si affichage en AM/PM),
- hh : l'heure sur deux chiffres (00 à 23 ou 01 à 12 si affichage en AM/PM),
- H : l'heure (0 à 23, même si affichage en AM/PM),
- HH : l'heure sur deux chiffres (00 à 23, même si affichage en AM/PM),
- m : la minute (0 à 59),
- mm : la minute en deux chiffres (00 à 59),
- s : la seconde (0 à 59),
- ss : la seconde en deux chiffres (00 à 59),
- z : la milliseconde (0 à 999),
- zzz : la milliseconde en trois chiffres (000 à 999),
- AP ou A : affichage en *AM/PM*. Affiche AM ou PM à la place de AP,
- ap ou a : affichage en *am/pm*. Affiche am ou pm à la place de ap.

Au final, le format qui nous intéresse est donc le suivant : dd/MM/yyyy/HH/mm/ss. Le code à utiliser pour récupérer la chaîne est donc :

```
datetime.toString  ("dd/MM/yyyy/HH/mm/ss")  ;
```

III-B - Foreach

Vous avez pu remarquer dans la section précédent une nouvelle structure : foreach. Non, elle n'existe pas en C++. Oui, Qt la supporte, puisqu'elle permet de réduire de beaucoup l'écriture, sans pour autant complexifier inutilement le reste.

Avec foreach

```
QLinkedList<QString> list;
QString str;
foreach (str, list)
    qDebug() << str;
```

Sans foreach

```
QLinkedList<QString> list;
...
QLinkedListIterator<QString> i(list);
while (i.hasNext())
    qDebug() << i.next();
```

Cela vous semble plus simple, non ? Étudions donc un peu plus ce mot-clé.

Il sert à *itérer* tous les *éléments* d'un *conteneur*. D'où :
foreach (variable, conteneur) { actions }

La variable peut être définie dans le foreach tant que vous n'utilisez pas de conteneur " *à virgule* " (par exemple, QPair<int, int>).

```
QLinkedList<QString> list;
foreach (QString str, list)
    qDebug() << str;
```

À côté de foreach, Qt fournit la structure forever, qui sert pour les boucles infinies.

III-C - Hash

Ensuite, nous avons besoin de hash cryptographiques : cela n'a plus rien à voir avec les métadonnées des fichiers, nous n'utiliserons donc pas **QFileInfo**. Par contre, ils concernent directement le contenu des fichiers : **QFile** nous sera donc utile.

Par contre, cette classe **QFile** ne propose strictement rien de cryptographique. À l'inverse de la classe **QCryptographicHash** !

Cependant, cette méthode n'accepte pas de **QIODevice**, uniquement des **QByteArray**. Cela ne dérangera personne : il est très facile d'obtenir un **QByteArray** à partir d'un **QFile**.

```
QFile file;

QByteArray data = file.readAll();

QByteArray md  = QCryptographicHash::hash (data, QCryptographicHash::Md5 );
QByteArray sha = QCryptographicHash::hash (data, QCryptographicHash::Sha1);
```

III-D - Petite optimisation

La gestion de la seconde partie de ce code va être assez ennuyante pour les performances. Reprenons le code avec lequel j'ai ouvert cette section.

```
QDir dir("./");
foreach  (  QFileInfo &fileInfo, dir.entryInfoList()  )
{
    fileInfo->isBundle();
}
```

Pour récupérer un objet **QFile** d'un **QFileInfo**, il faut récupérer le nom du fichier, puis instancier un **QFile** grâce à ce nom : très lourd, n'est-ce pas ?

```
QDir dir("./");
foreach  (  QFileInfo &fileInfo, dir.entryInfoList()  )
{
    QString name = fileInfo->fileName();

    file = QFile (name);
}
```

Je vais donc vous proposer une autre solution, qui n'est à utiliser qu'en ce cas (si vous n'utilisez pas pour autre chose le nom de fichier, vous obligez le système à charger deux fois les informations concernant chaque fichier, d'où des performances catastrophiques quand le système de fichier peine un peu).

```
foreach  (  QString &filename, dir.entryList()  )
{
    QFile file (filename);
    // Ce que nous voulons du QFile

    QFileInfo info (file);
    // Ce que nous voulons du QFileInfo
}
```