
Serving QML applications over the network
Jeremy Lainé
Wifirst

Jeremy Lainé

- Using Qt since 2001 (desktop, mobile, embedded)
- Occasional Qt contributor (QDnsLookup)
- Head of software development at Wifirst (ISP)
- Lead developer of a IM / VoIP app for Wifirst customers

Overview

1. Why serve applications over the network?

2. QML network transparency

3. Building your application

4. Deploying your application


1. Why serve applications over the network?

Typical application deployment

- Development
- Packaging
- In-house beta testing
- Push application to all users
- Repeat!

Each iteration requires per-platform installers and an update.

Options for handling updates

- Manual updates
  - Most users cannot be bothered to update
  - You end up with a heterogeneous installed base
  - Your cool new features don't reach your users!
- Automatic updates
  - Not all platforms have an “application store”
  - Each platform requires specific packaging wok
  - Usually requires elevated permissions (Windows UAC..)

Updates are hard!

Not convinced? Look at the Chrome and Firefox codebases!

How about QML apps?

- C++ wrapper code:
  - has usual deployment constraints
- QML / Javascript code and resources:
  - fully cross-platform
  - conveniently split into multiple files
  - can be loaded over the network by QtQuick

Some benefits

- Faster iteration and testing
- Fix bugs after release!
- Progressive update roll-out
- Split testing for UI changes
- Time-limited changes (Christmas specials!)


2. QML network transparency

QDeclarativeView (Qt4) and QQuickView (Qt5) support loading from an HTTP URL

```
int main(int argc, char *argv[])
{
  QApplication app(argc, argv);

  QQuickView view;
  view.setSource(QUrl("http://foo.com/main.qml"));
  view.show();

  return app.exec();
}
```

QML “loader” elements

Built-in QML elements with a “source” property support HTTP URLs

- Image
- Loader
- FontLoader

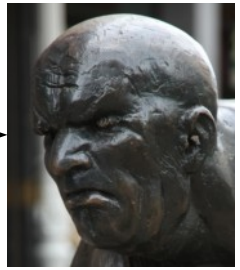
```
Image {
  source: “http://foo.com/bar.img”
}
```

Relative URLs

Relative URLs are resolved relative to the QML document's URL

```
Image {
  source: “head.jpg”
}
```

file:///home/bob/foo.qml



file:///home/bob/head.jpg

```
Image {
  source: “head.jpg”
}
```

http://example.com/foo.qml

http://example.com/head.jpg

You can use the same code locally and remotely!

QML network transparency

- QML type declarations can be served over HTTP, but you need to list your types in a “qmldir” file:

```
Button 1.0 Button.qml
CheckBox 1.0 CheckBox.qml
```

- Javascript code can be served over HTTP

```
import “scripts/utils.js” as Utils
```

Translations

- Loading translations from QML is missing
- You can provide your own TranslationLoader and do

```
TranslationLoader {
   source: “i18n/my_translation.qm”

   onStatusChanged: {
      Console.log(“status is: “ + status);
   }
}
```

- A proposal for including it in Qt5

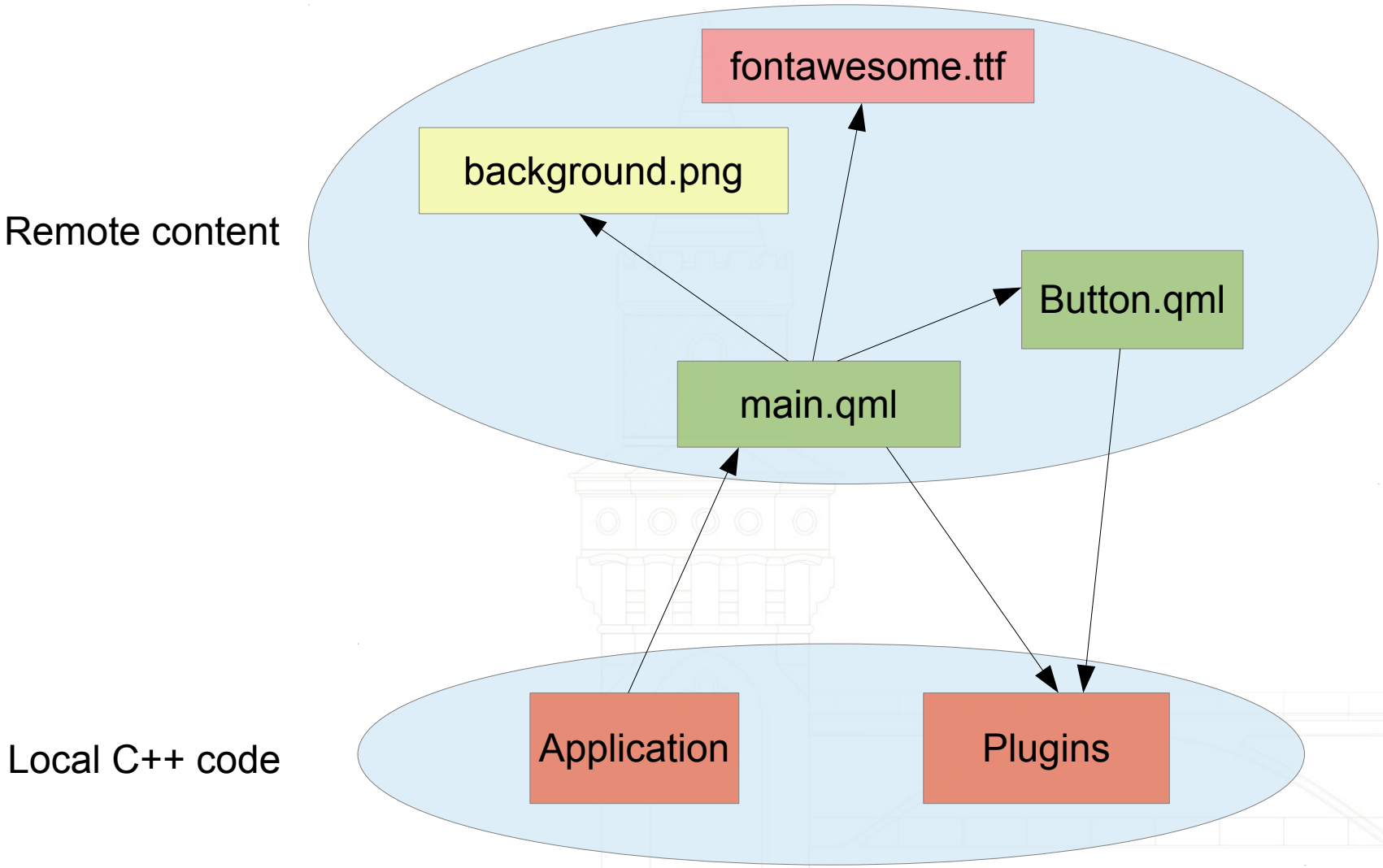
https://codereview.qt-project.org/#change,31864

3. Building your application

General recommendations

- Split models and presentation
- The C++ code still needs traditional updates
  - Make the loader robust
  - Keep the API simple
  - Keep the API stable
- Serve all the rest on the fly
  - QML and Javascript code
  - Resources (images, fonts, translations)

Application architecture



The application

- Creates the QML view
- Sets up the QNetworkAccessManager
- Loads a single “root” QML file over the network
- Can fallback to local files for offline use

Application / setting up QNAM

- Give your application a User-Agent
  - Helps track usage, or serve different content
  - QtWebkit generated request already have a UA
- Specify the preferred language (Accept-Language)
- Set up a persistent network cache
- Configure HTTP pipelining

Application / caching

- QtQuick caches components + pixmaps (memory)
- QNAM supports “If-Modified-Since” but needs a persistent disk cache

```
class MyFactory : public QQmlNetworkAccessManagerFactory
{
public:
    QNetworkAccessManager* create(QObject* parent)
    {
        QNetworkAccessManager* manager = new QNetworkAccessManager(parent);
        QNetworkDiskCache* cache = new QNetworkDiskCache(manager);
        cache->setCacheDirectory(“/some/directory/”);
        manager->setCache(cache);
        return manager;
    }
};

view->engine()->setNetworkAccessManagerFactory(new MyFactory());
```

Application / HTTP pipelining

- Splitting QML into files: good but incurs overhead
- HTTP/1.1 allows sending multiple requests without waiting for replies
  - Particularly useful for high latency links
- Qt5 uses pipelining for all resources
- Qt4 only uses pipelining for pixmaps and fonts
  - subclass QNetworkAccessManager if needed

Application / offline use

- At startup, fall back to a bundled copy of your QML code if loading from network fails

```
void MyView::onStatusChanged(QQuickView::Status status)
{
   if (status == QQuickView::Error && useNetwork) {
      useNetwork = false;
      setSource(QUrl("qrc://main.qml"));
   }
}
```

- Catching errors later is harder..

- Define data models and scriptable objects

- Keep the C++ code simple : if something can be done in QML instead, do it!

- Keep the API stable : if you change it, you will probably need different QML files

- Welcome to an asynchronous world!

```
var component = Qt.createComponent(source);
var object = component.createObject(parent);
```

BAD

```
var component = Qt.createComponent(source);
if (component.status == Component.Loading)
   component.statusChanged.connect(finishCreation);
else
   finishCreation();

function finishCreation() {
   if (component.status == Component.Ready) {
      var object = component.createObject(parent);
   }
}
```

GOOD

- Review your timing assumptions!
  - Do not depend on objects loading in a set order
  - Use Component.onCompleted with care

- Having lots of icons can be a problem, consider using web fonts like FontAwesome

4. Deploying your application

Hosting the QML code

- You have all the usual web hosting options
  - Run your own servers (nginx, apache, ..)
  - Use cloud services
  - Do load-balancing, fail-over, etc..
- QML files can be 100% static files, or even generated on the fly

Version your root URL

- Plan for multiple versions of your C++ app, as the API will probably change, e.g. :

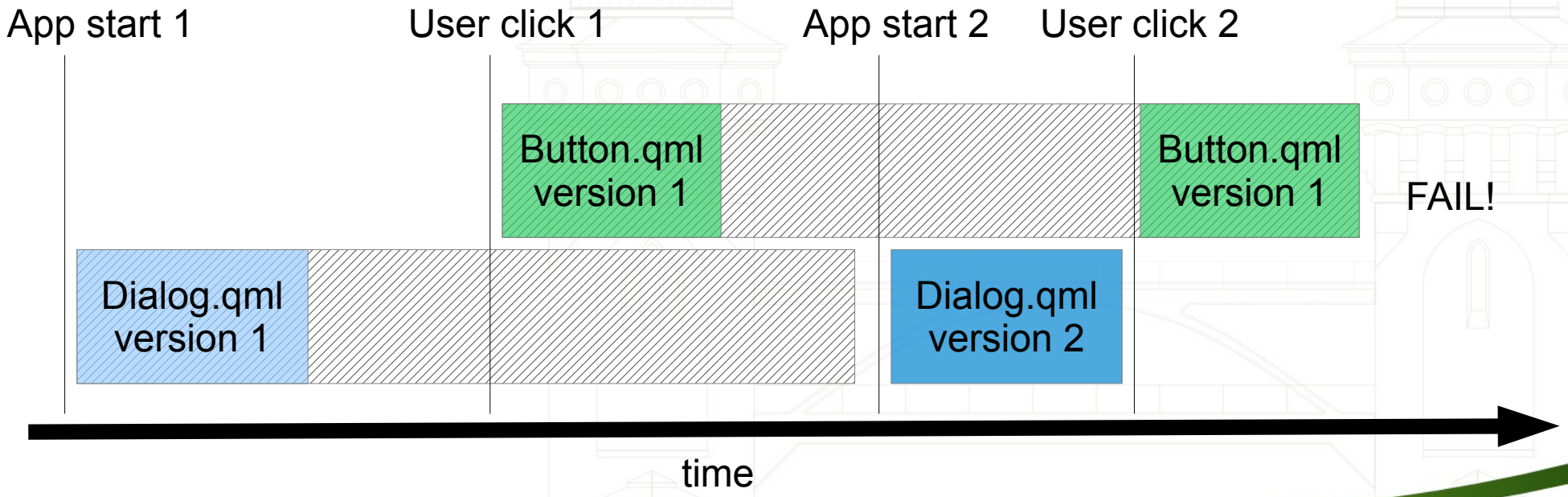
http://example.com/myapp/1.0/main.qml

http://example.com/myapp/1.1/main.qml

- Alternatively, switch on User-Agent

Cache “consistency”

- Consider two related files Dialog.qml and Button.qml, which must be in the same version to work
- Caching can cause inconsistency



Cache “consistency”

- Your main.qml can load all subsequent contents from a subdirectory to “version” the QML code

- Layout:
  - main.qml
  - RELEASE_ID/Button.qml
  - RELEASE_ID/Dialog.qml

Security

- Make use of HTTPS to avoid your application loading malicious code (DNS hijacking)
- Make sure your certificates are valid!
- Some platforms or custom certificates will require adding your CA certificates

```
QSslSocket::addDefaultCaCertificates("./myca.pem");
```

Performance considerations

- Enable gzip compression for QML and JS files
- Set an Expires header to avoid QNAM re-checking all files on startups
- Serve from a cookie-less domain


5. Questions

Get the code

- Source code for the Wifirst IM / VoIP client

```
git clone git://git.wifirst.net/wilink.git
```