Make your Qt UI available to any browser in 5 steps

Luca Ottaviano, Develer Srl, <lottaviano@develer.com>

Short bio

Technical leader at Develer

Completed 10+ projects on desktop and embedded

Experience with Qt Widgets and Qt Quick

Trainer on C++ and Qt stack @Develer

Is it a marketing move?

Of course it is!

The 5 steps

1. Use Qt Remote Objects on the host process to share all needed objects
2. Use Qt Remote Objects on the client process to receive updates from the host. The rest of the code is left untouched
3. Launch the host process with QtQuick UI on the local screen
4. Launch the client process with “-platform webgl:port=<port>”
5. Connect with the browser to “<your-machine-ip>:<webgl-port>”

Enjoy your application on two screens!

Is it really that simple?

What about non QtQuick UIs?

What are Qt Remote Objects? How much code do I need to write to support them?

What about security?

How many users can connect with this setup?

Which browsers are supported?

Agenda

- Problem statement and use cases
- Remoting options
- Multi user: sharing state between processes
- Final words

Problem statement

You want to interact with a Qt UI from remote without rewriting significant parts of the application

- Example: headless device
- Example: remote maintenance of an appliance with a local UI
- Example: remote training

Need to target the browser, it’s a “no configuration required” option available anywhere!

Remoting options

1. VNC
2. WebGL platform plugin
3. Qt for WebAssembly

Option 1: VNC

- Start your program with the command line option “-platform vnc”
- Connect with a VNC program to the port
- Enjoy your remote application!

Recap

Pros:

- Works with any Qt UI
- Requires no extra coding effort

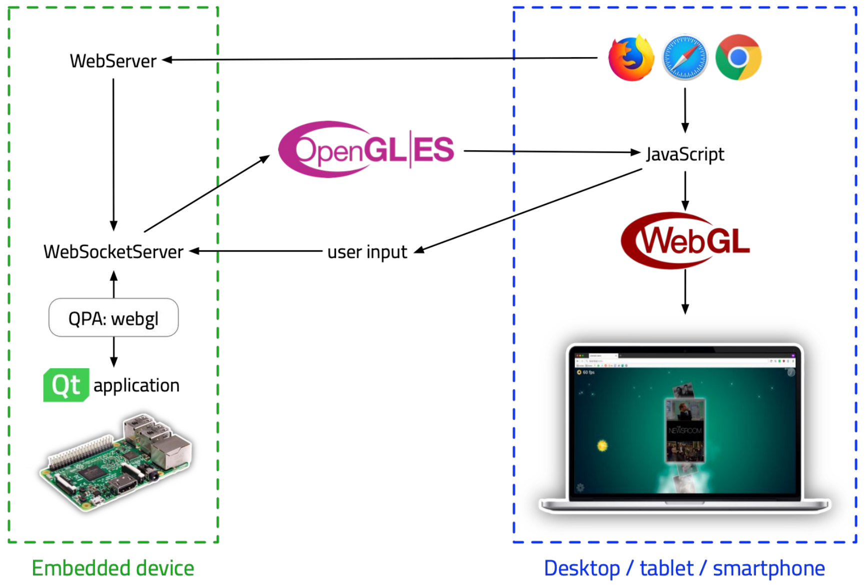
Cons:

- Requires additional software on the user machine
- Very sensitive to network latency
- High bandwidth requirements
- Only one user at the time

Option 2: WebGL platform plugin

- Start your application with suitable command line options: “-platform webgl:port=8080”
- Connect with the browser to “host:port”, eg. 192.168.1.1:8080
- Use the application from the browser!

WebGL platform plugin overview



Recap

Pros:

- Enables remoting with 0 effort
- Works even without an on board GPU

Cons:

- Doesn’t work with raster surfaces (ie. Qt Widgets)
- Performance highly dependent on network latency
- Only one user at the time
- Not all browsers perform well

Option 3: Qt for WebAssembly

WebAssembly is a bytecode format specification, enables JS engines in browsers to execute code nearly at native speed

It’s a companion to JS, rendering is still done with HTML elements

C++ applications can be compiled to WebAssembly and executed as web pages.

Technology preview in Qt 5.13

Recap

Pros:

- Runs on any browser
- Natively multi user

Cons:

- The application is sandboxed (only HTTP requests or web sockets, no access to local file system)
- Need a special compiler to generate WebAssembly code
- Not yet ready for prime time

Multi user support

Qt applications are designed for a single user at once.

Each Qt process has a global state in memory.

We need to find a way to share this state to multiple processes.

Inter process communication

IPC to the rescue: share data between processes

IPC is composed of two parts:

- communication protocol
- serialization specification

Communication protocol

Localhost:

- Named pipes
- D-BUS
- ...

Remote:

- Zero MQ
- HTTP
- ...

Serialization specification

Even more options:

- JSON
- protobuf
- Flatbuffers
- CBOR
- ...

Qt Remote Objects

A complete solution to synchronize Qt objects between processes.

Works out of the box for QObjects and QAbstractItemModels

Key concepts:

- Source: the true object that lives in a specific process
- Replica: a lightweight proxy for the Source
- Node: a process on the network

Network topology

Processes that use Qt Remote Objects form a network.

There are two types of nodes:

- Host node: a process that contains (hosts) the Source to be shared
- Client node: a process that acquires a Replice of the Source

The network is peer-to-peer, each client needs a connection to the host node.

Creating a Source object

```
ControlUnit *o = new ControlUnit; // create your object as usual

QRemoteObjectHost srcNode(QUrl("local:ControlUnit")); // set the node address

srcNode.enableRemoting(o, “MyControlUnit”); // set object name on the node
```

Creating a Replica object

```
QRemoteObjectNode repNode;

repNode.connectToNode(QUrl("local:ControlUnit")); // connect to host node

QSharedPointer<QRemoteObjectDynamicReplica> ptr; // hold the replica

ptr.reset(repNode.acquireDynamic("MyControlUnit")); // acquire object by name

// The replica is ready after initialized()

//connect(ptr.data(), SIGNAL(initialized(), this, SLOT(replicaReady()));
```

Creating a Source model

```
MyModel *o = new MyModel; // create your object as usual

QRemoteObjectHost srcNode(QUrl("local:ControlUnit")); // set the node address

// Call the correct overload of enableRemoting()!

srcNode.enableRemoting(o, “MyModel”, o->roleNames().keys().toVector());

// Now the model is visible to the network as MyModel
```

Creating a Replica model

```
QRemoteObjectNode repNode;

repNode.connectToNode(QUrl("local:ControlUnit")); // connect to host node

QSharedPointer<QAbstractItemModelReplica> repPtr;

repPtr.reset(repNode.acquireModel("MyModel"));

// The replica is ready after initialized()

//connect(ptr.data(), SIGNAL(initialized(), this, SLOT(replicaReady()));
```

Connection types

Qt Remote Objects support two connection types in Qt 5.12:

- local: a local-only network type, possibly more efficient than TCP based connections
- tcp: a TCP connection, must be fully specified like this: tcp://192.168.1.1:9876

Extra connection types are supported on any QIODevice, but you must establish the connection yourself (eg. QSslSocket)

```
void addHostSideConnection(QIODevice *ioDevice)

void addClientSideConnection(QIODevice *ioDevice)
```

The 5 steps - again

1. Use Qt Remote Objects on the host process to share all needed objects
2. Use Qt Remote Objects on the client process to receive updates from the host. The rest of the code is left untouched
3. Launch the host process with QtQuick UI on the local screen
4. Launch the client process with “-platform webgl:port=<port>”
5. Connect with the browser to “<your-machine-ip>:<webgl-port>”

Enjoy your application on **two** screens!

You need more infrastructure work if you want more than 2 users, eg. spawning processes on the fly on connection attempts

Conclusions

You can easily access with the browser to any QtQuick UI.

You can support multiple users by sharing state with Qt Remote Objects.

You still need some infrastructure to dynamically adjust the number of allowed users.

No deploy needed (browsers are everywhere)

Performance is not on par with native applications

Questions?

Resources

Links:

- QtQuick WebGL platform plugin: https://doc.qt.io/qt-5/webgl.html
- WebGL or WebAssembly? https://blog.qt.io/blog/2018/06/12/remote-uis-webgl-webassembly/
- WebGL streaming blog post: https://blog.qt.io/blog/2018/11/23/qt-quick-webgl-release-512/
- Qt Remote Objects documentation: https://doc.qt.io/qt-5/qtremoteobjects-index.html