Rpc Binding

Background

RPC binding is a pretty tried and tested concept, with numerous permutations, combinations and implementations to choose from. Some require the use of IDL's, others provide a container with support for a set of well known types, for example, there are several implementations for different languages of the XML-RPC protocol. JSON also has a remote procedure calling standard for which there are several implementations. Then there's SOAP, CORBA and the RPC libraries that became part of various well known operating systems.

The aim here is not to reinvent the wheel but to provide a Qt friendly way of extending QObject subclasses to allow synchronous RPC calls on methods that have been tagged with Q_INVOKABLE. By "Qt friendly" I mean using concepts and terminology that should be familiar to Qt developers. The idea is to take advantage of Qt's meta object system to glean just enough information about an object for it to reach out to identical objects residing somewhere else. "Somewhere else" means either in a process running on another network node or on a process running on the same machine. Either way, communication between the objects is done using the TCP/IP network stack.

I want to be clear about what I mean by RPC binding, if you haven't already guessed. An example would probably serve this aim better than anything. Imagine you have a user interface, part of a client application which runs on the desktop, written using Qt. Your customer discovers that he wants to add the ability to add some data intensive tasks to the application, but would like these tasks to be done on a server farm with dedicated processing and wants the data to be returned to the client when processing has completed, without stalling the user interface or slowing down the client machine too much.

There are a few ways in which you could achieve this. You might opt for an HTTP/web server approach. You invent some additional fields in the GET request header and let a web server with some extensions forward requests to the data processing process(es) on the server. Alternatively you might choose something like XML-RPC or even write your own protocol using Qt or some other socket wrapper. All these approaches are valid, but might entail using some additional time to implement and they have the overhead associated with text-based protocols, particularly in the case of XML-RPC. You also have the added problem of adapting your Qt client code to use some other framework or library.

As an alternative to all the above, you could link your client project with librpcclient, create a QObject subclass with Q_INVOKABLE methods that encompass the API you'd like to do the data processing on the server side. You create a minimal application for the server which links librpcserver and contains your data processing API object. For the client, you link librpcclient and you run roc on the API object. You rebuild your client application and it's ready to go.

The example in tests/rpctest does just this, albeit in a trivial example.

Limitations

There are a few conditions and limitations in the current implementation. Your API, more specifically, the methods that you wish to remotely invoke must be written using data types that are already listed as supported by QMetaType. This means most Qt types, including QImage, QString etc. There is no support, as yet, for pointer or reference variables used bidirectionally, i.e. if your implementation alters variables that have been passed in as function arguments, they won't be reflected in the caller. The return type must also be a non-pointer/reference type. This is fairly easy to accommodate since you are probably writing the API from scratch and can adhere to the current usage constraints in the process.

Client Side Code Example

Methods that you want to be remotely invokable must be prefixed with the standard Qt Q_INVOKABLE tag, so that the introspection mechanism in Qt is able to recognise them as such. For example, the following class header would be suitable (taken from the autotest example):-

```
class RpcTestClass : public QObject
{
    Q_OBJECT
public:
    RpcTestClass(QObject* parent = 0);
    virtual ~RpcTestClass();
    Q_INVOKABLE void testRpcMethod1(int, bool);
    Q_INVOKABLE int getterTest1(int);
    Q_INVOKABLE bool getterTest2(bool);

};
```

The implementation for this simple class happens to look like this:-

.

.

.

```
void RpcTestClass::testRpcMethod1(int param0, bool param1) {

    qDebug() << "RpcTestClass::testRpcMethod1 called with parameters: "
             << "int param0: "
             << param0
             << "bool param1: "
             << param1;
}
int RpcTestClass::getterTest1(int param0) {

    qDebug() << "RpcTestClass::getterTest1 called with parameter: " << "int para
m0: " << param0;
    return param0;
}
bool RpcTestClass::getterTest2(bool param0) {

    qDebug() << "RpcTestClass::getterTest2 called with paramter: " << "bool para
m0: " << param0;
    return param0;
}
```

When I've passed this class through *roc,* it generates a new implementation which looks like this:-

```
.
.
.
void RpcTestClass::testRpcMethod1( int param_0, bool param_1) {

    methodData_t data;
    bool ok = false;
    data.methodId = 5;
    data.argList.append(QVariant::fromValue(param_0));
    data.argList.append(QVariant::fromValue(param_1));
    data.returnType = "void";
    ok = CallInterfaceFactory::instance(this)->call(&data);
    if (!ok) qWarning() << "RPC Call to testRpcMethod1 failed.";

}

int RpcTestClass::getterTest1( int param_0) {

    methodData_t data;
    bool ok = false;
    int retVal;
    data.methodId = 6;
    data.returnData = QVariant::fromValue(retVal);
    data.argList.append(QVariant::fromValue(param_0));
    data.returnType = "int";
    ok = CallInterfaceFactory::instance(this)->call(&data);
    if (!ok) qWarning() << "RPC Call to getterTest1 failed.";
    return data.returnData.value<int>();
}

bool RpcTestClass::getterTest2( bool param_0) {

    methodData_t data;
    bool ok = false;
    bool retVal;
    data.methodId = 7;
    data.returnData = QVariant::fromValue(retVal);
    data.argList.append(QVariant::fromValue(param_0));
```

```
    data.returnType = "bool";
    ok = CallInterfaceFactory::instance(this)->call(&data);
    if (!ok) qWarning() << "RPC Call to getterTest2 failed.";
    return data.returnData.value<bool>();
}
```

If you rebuild the client project, containing the new implementation, the project will think it is interacting with the real class but method calls will be forwarded to the remote instance.

The last step necessary to enable the client side calls to be forwarded is to create the object in the client application and bind it to the remote "real" object. This is done in the following way:-

```
RpcClient* m_client;
RpcTestClass* m_clientTestObject;
.
.
.
m_clientTestObject = new RpcTestClass(this);
m_clientTestObject->setObjectName("TestObject");
m_client = new RpcClient(this);
.
.
.
m_client->bind(m_clientTestObject,"192.168.0.196",5656);
```

The client binds to an instance of "RpcTestClass" running on the IP address and port given in the bind call.

Server Side Code Example

The server side project, or servlet would link the library librpcserver and it would include the object described above, with the real implementation. The server would then register this object so its interface would be available to clients.

The relevant part of the code (registration of a remote object) is given here and can be found the tests/testrpc:-

```
RpcServer* m_server;
RpcTestClass* m_serverTestObject;
.
.
.
m_serverTestObject = new RpcTestClass(0);
m_serverTestObject->setObjectName("TestObject");
m_server = new RpcServer(this);
.
.
.
m_server->registerReceiver(m_serverTestObject, 5656);
```

The server registers this instance to listen on port 5656 for incoming call requests.