Building distributed applications with JOLIE and Qt

How to help your applications enter the service-oriented world

Fabrizio Montesi, Kévin Ottens

italianaSoftware
KDAB

Akademy 2009 – July 6th

Outline

Outline

A distributed world

- Networking technologies have introduced new ways for users to share, store and access data.
- User data is now stored in multiple points, each one possibly implemented in a different way.
- We want to enable users to coordinate their distributed data.
- How do we make UIs interacting with this distributed world?

The objectives

- Reuse existing tools and languages in their own specific area.
- Provide developers with the means for coordinating distributed applications easily.
- Connect UIs to these technologies.
- Pay attention to performance and scalability.

How we do it

- We make use of a service-oriented language (JOLIE) to handle distributed programming.
- We build a layer between C++/Qt and JOLIE so they can interoperate.
- Distributed programming and UI programming remain separated.

Service-Oriented Computing (SOC)

- An emerging paradigm for programming distributed applications.
- Generalizes the concepts of service.
- Offers a model for handling the complexity of service systems.
- Web Services and D-Bus are examples of service-oriented technologies.

What's a service?

- An application exposing functionalities to other applications.
- It communicates through language-independent protocols.
- Offers a description of itself to the other partners.

Service composition and orchestration

- A service can exploit the functionalities of existing ones, by composing them.
- Such composite services are called "orchestrators".
- An orchestrator coordinates other services from its single point of view.
- Writing orchestrators can be a difficult task: we need new appropriate languages and primitives.

Outline

JOLIE

- JOLIE is a new service-oriented language.
- Lightweight and cross-platform.
- Based on a formal theoretical framework.
- It offers new constructs and primitives, focusing on the problems of distributed programming.
- It supports pluggable communication protocols and mediums, without requiring you to change your programs.

Some native features in JOLIE

- Native primitives for performing communications.
- Parallel execution of activities, through the | operator.
- External choice: the orchestrator offers multiple possibilities (like in a switch) and the communicating partner chooses one by sending a message notifying the choice.
- Dynamic fault handling, adaptive w.r.t. the execution flow.

MetaService

- A registry of services, it acts as a service router.
- It allows you to access services without knowing their location, or even the protocol they speak.
- Routes your messages to the registered service, converting the data to the appropriate protocol.
- Can load embedded services on demand.

Outline

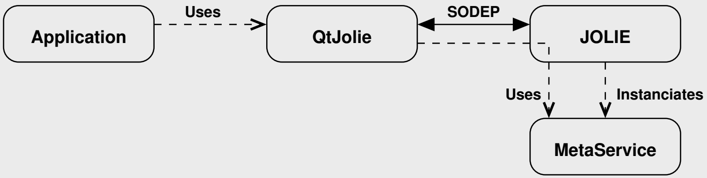
Qt + JOLIE → QtJolie

Exposing C++ applications to the service world since 2009

Goals

- Allow Qt applications to drive a JOLIE instance
- Make it easy to talk to a (Web) Service
- Mid-term, benefit from JOLIE abilities to mix services and protocols
- Provide a Qt-ish API, similar to QtDBus when it makes sense

Architecture



API overview

[WARNING]

Work in progress.
Convenience missing.

API overview

Most important facts

- We’re using the `Jolie` namespace
- You can create a `Client` or a `Server`
- The API allows to manage `Messages`
- `Messages` have data which is a `Value` instance
- `Values` form a tree of key/list pairs and can be int, double or byte array
- In case of problems `Fault` instances shall be issued
- By inheriting `AbstractAdaptor` you can process incoming calls

Making calls

Let's use `Jolie::Client`

```
using namespace Jolie;

Client *client = new Client("localhost", 9000);

Message message("/Printer", "printInput");
message.setData(Value("Patapatapon!"));

Message reply = client->call(message);

Q_ASSERT(reply.toByteArray()=="success");
```

Receiving calls (1/3)

Let’s use `Jolie::Server` and `Jolie::AbstractAdaptor`

```
using namespace Jolie;

class CalculatorAdaptor : public AbstractAdaptor
{
    Q_OBJECT
public:
    CalculatorAdaptor(QObject *parent = 0)
        : AbstractAdaptor(parent) { }
private:
    //==>
```

Receiving calls (2/3)

Let's use `Jolie::Server` and `Jolie::AbstractAdaptor`

```
//<==
Message relay(Server *server,
              const Message &message)
{
  Value data = message.data();
  qint32 x = data.children("x").first().toInt();
  qint32 y = data.children("y").first().toInt();

  Message reply(message.resourcePath(),
                message.operationName(),
                message.id());
  reply.setData(Value(x+y));
  return reply;
}
};
```

Receiving calls (3/3)

Let's use `Jolie::Server` and `Jolie::AbstractAdaptor`

```
using namespace Jolie;

Server *server = new Server(8000);
CalculatorAdaptor *adaptor
    = new CalculatorAdaptor(this);

server->registerAdaptor("/", adaptor);
```

Managing services

Let's use `Jolie::MetaService`

`MetaService` management

- `start()/stop()`
- `loadService("Math", "math.ol")`
- `unloadService("MathService")`
- `listServices()`

Redirections

```
addRedirection("Calculator",
               "socket://localhost:8000",
               Value("sodep"),
               Value("soap"));
```

Outline

Yahoo! Search client

Quick and dirty... but with nice tricks

- Quick demo...
- Show me the code!

Questions ?

Fabrizio Montezi
fmontesi@italianasoftware.com

Kévin Ottens
kevin@kdab.com