A SOAP stack for the Qt framework

Peter Hartmann

September 22, 2008

diploma thesis

peter@hartmann.tk

Contents

Abstract

SOAP is a protocol for enabling communication in a distributed system. This document evaluates the SOAP protocol and compares it to other distributed computing techniques like REST, Java RMI and AJAX. Furthermore, it evaluates the Qt framework as well as existing SOAP stacks (Apache Axis2, gSOAP, SOAP::Lite and .NET); after that, the design and implemenetation of a SOAP stack for the Qt framework is proposed. An analysis of this system will then show whether the requirements of a SOAP toolkit could be met.

1 Introduction

1.1 Task definition

The purpose of this document is to propose an architecture and implementation of a SOAP stack for the Qt framework. To achieve a good result, both the requirements of SOAP as well as Qt need to be evaluated.

SOAP evaluation

Before designing an architecture of a SOAP stack, this document will evaluate SOAP, trying to answer the following questions:

- What is the main field of use of the SOAP protocol? How does it overlap with similar techniques?

SOAP is a protocol enabling communication between nodes in a network; this document will compare SOAP to other Web service and distributed programming techniques, and suggest the main use case of SOAP as well as fields where SOAP might not be the right choice and other techniques are better suited.

- What API do other SOAP toolkits offer? What are their strengths and weaknesses?

In order to propose a SOAP framework that fits user needs, other frameworks will be analyzed. The main focus of this evaluation will be the API the toolkits offer; features and architectural paradigms of those toolkits could then be adopted in the SOAP toolkit that is to be designed.

The SOAP evaluation will discuss the SOAP protocol itself and its use cases as well as analyze existing toolkits, and see how SOAP is used in practice.

Qt evaluation

In addition to evaluating SOAP, the Qt framework needs to be analyzed. Here, evaluating means both describing what features the Qt framework already offers in terms of networking and XML support as well as the main use case for programs using Qt. The focus of Qt frameworks and Qt programs might influence the architecture of the Qt SOAP stack; moreover, the SOAP stack should of course offer an API that is similar to the API of other Qt modules.

1.2 Overview

The subsequent chapters of this document are structured as follows:

- chapter 2 gives an introduction into Web services and SOAP from a very general and high-level view.

- chapter 3 describes the Qt framework: The chapters 3.1 and 3.2 describe the framework in general, while chapter 3.3 gives an overview over the tools needed by the SOAP stack (which is described later).

- chapter 4 describes the evaluation of both other Web service techniques as well as other SOAP toolkits: The sections 4.1 and 4.2 try to point out the advantages and disadvantages of SOAP, and outline SOAP fields of application from other techniques. Chapter 4.3 evaluates different SOAP stacks and shows their practical use; as a summary, section 4.4 summarizes the results of the evaluation of the precedent chapters.

- chapter 5 proposes an architecture and evaluation of a SOAP stack for the Qt framework.

- chapter 6 summarizes the results of that documents and tries to answer the questions from chapter 1.1.

2 Basics

2.1 Web services

A Web service is, citing the W3C, "a software system designed to support interoperable machine-to-machine interaction over a network" [11]. That is, a very basic Web service scenario involves at least the following, as can be seen in Figure 1:

1. a service provider offering a specific functionality
2. a service requester demanding that functionality
3. a medium used for communication between the two

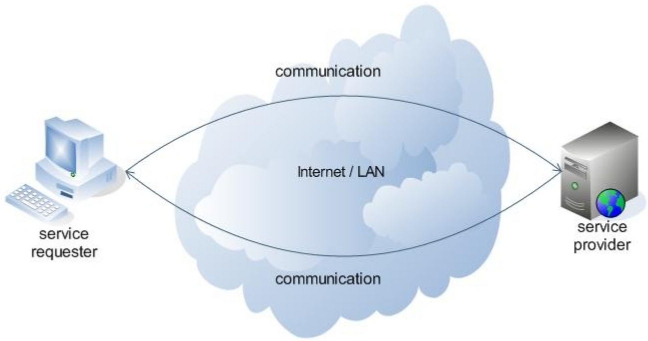


Figure 1: A basic Web service scenario

In the case of a Web service, the medium used for communication is the Internet or a Local Area Network. From this scenario, a lot of questions arise, for instance:

- What underlying protocol is used for communication?

Basically, any application or transport level protocol could be used here; however, in practice the most frequently used protocol is HTTP. Another common use case involves JMS-compliant middleware messaging protocols like ActiveMQ or WebSphereMQ. More rare scenarios use TCP, UDP or XMPP as an underlying protocol.

- How does the service requester know how to invoke the functionality of the service provider?

When a service requester wants to invoke a functionality offered by a provider, the former needs to know where to address its request to and how payload data should be formatted. This means that the provider must publish an interface containing that information. This interface could be encoded in a programming language dependent format, e.g. a Java interface or a C++ header file; in practice, however, a Web service interface is published in an XML-based language called WSDL (Web Services Description Language).

- What message format is used to carry information from the requester to the provider and back?

Since the parties involved in a Web service system want to exchange information, they have to agree on a format to encode the exchanged data. Alternatives for such an encoding are XML, JSON (JavaScript Object Notation), or any self-designed or programming language dependent encoding.

- How is the communication flow organized?

A Web service might require a different number of inbound messages and outbound messages. A standard scenario of organizing message exchange is to have one inbound message to the service provider, which in turn sends one message back to the requester. But several Web services require other Message Exchange Patterns (MEPs), e.g. one-way operation (in-only), solicit-response (first out, then in) or notification (out-only).

More advanced use cases of a Web service include multiple service requesters and providers, service registries, actors that fulfill both the role of a service provider and requester, Web services that are composites of other Web services etc. Some of those advanced scenarios are adressed in later chapters of this document.

2.2 SOAP

SOAP [28] is an XML-based protocol destined for exchanging messages over a network. It is commonly used as a protocol in Web service infrastructures. Going back to the four questions from chapter 2.1, SOAP provides an answer for each of those:

- What underlying protocol is used for communication?

When using SOAP, the usual choice for an underlying protocol is HTTP. The SOAP standard even describes an HTTP binding, but does not mandate its use, i.e. other underlying protocols are possible (with JMS being one of the alternatives).

- How does the service requester know how to invoke the functionality of the service provider?

As already said, to address this issue there must be an interface accessible for the service requester. Using SOAP is tied closely to using WSDL ([3]), an XML-based language for describing network interfaces; so often when talking about SOAP, people mean using SOAP and WSDL in conjunction, although the SOAP standard does not mention WSDL.

- What message format is used to carry information from the requester to the provider and back?

SOAP is using XML as message format.

- How is the communication flow organized?

As SOAP is mostly used with HTTP, the most common message exchange pattern is request-response, since it maps naturally to the HTTP request-response mechanism.

Apart from that, as the purpose of this document is to propose a SOAP implementation for the Qt framework, which is described in chapter 3, using the request-response pattern via HTTP (which can be viewed as issueing a RPC) shall be the common scenario here.
As a basic example of a SOAP communication, consider a scenario where a StockQuote service offers looking up the stock price of a company (this example is similar to the one in [27]). A request would send the symbol of the company that it wants to know the stock price from:

Listing 1: A sample SOAP-over-HTTP request

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "StockQuoteAction"

<soap:Envelope
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Header></soap:Header>
    <soap:Body>
        <ns:GetLastTradePrice xmlns:ns="StockQuoteURI">
            <ns:symbol>TROLL</ns:symbol>
        </ns:GetLastTradePrice>
    </soap:Body>
</soap:Envelope>
```

The stock quote service provider could, upon receiving the message, invoke a procedure to look up the price with symbol “TROLL”, and respond with the following message:

Listing 2: A sample SOAP-over-HTTP response

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

```

```
<soap:Envelope
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Header></soap:Header>
    <soap:Body>
        <ns:GetLastTradePriceResponse
            xmlns:ns="StockQuoteURI">
            <ns:price>16.5</ns:price>
        </ns:GetLastTradePriceResponse>
    </soap:Body>
</soap:Envelope>
```

The response contains the stock price of the symbol requested before, which amounts to 16.5.

A more detailed discussion of SOAP messages structure will be given in chapter 4.1.

3 The Qt framework

This chapter shall provide some introduction to the Qt framework[1], especially explaining the parts that are necessary to create the SOAP stack described in chapter 5.

Qt is a C++ GUI framework for developing cross-platform applications. That is, its primary goal is to enable applications to be written once and compiled anywhere, as the preface of [1] connotes.

There are several well-known applications using Qt, for instance KDE (a popular X11 window manager), Google Earth and Skype.

The Qt toolkit consists of the parts depicted in Figure 2.

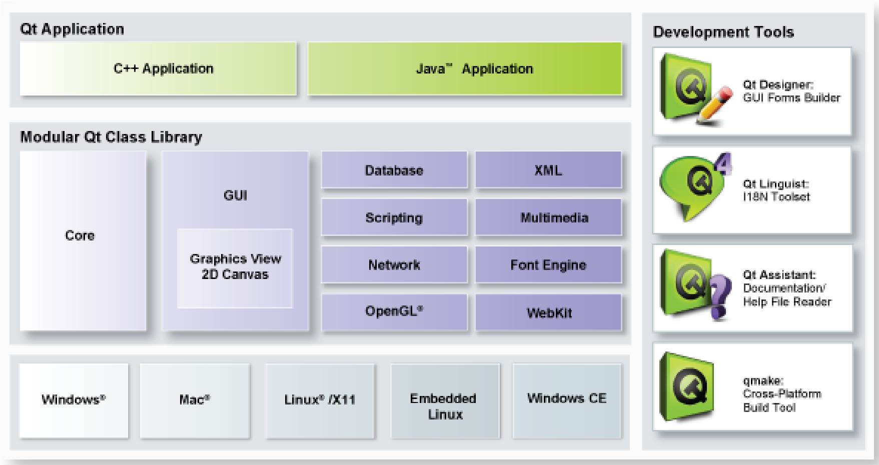


Figure 2: The Qt architecture

The most important parts of the Qt architecture are:

Modular Qt Class Library. The Qt C++ class library consists of a comprehensive list of classes, with the main focus on ready-to-go GUI elements (push buttons, radio buttons, text input fields, icons...). Furthermore, it also provides APIs for database usage, networking and XML, among others. The Qt APIs are designed to be easy and intuitive for end users, with classes and methods having self-documenting names. Figure 3 shows the component diagram of the Qt modules, with the QtCore module being the central component. Since the main focus of the Qt library is set on GUI programming, the QtGui module is also (as the QtCore module) included implicitly in the build process. As an example, Listing 3 shows a simple "Hello World" application (similar to the one from [1]), that displays a "Hello World" button, as depicted in Figure 4.

[1] http://trolltech.com/products/qt/

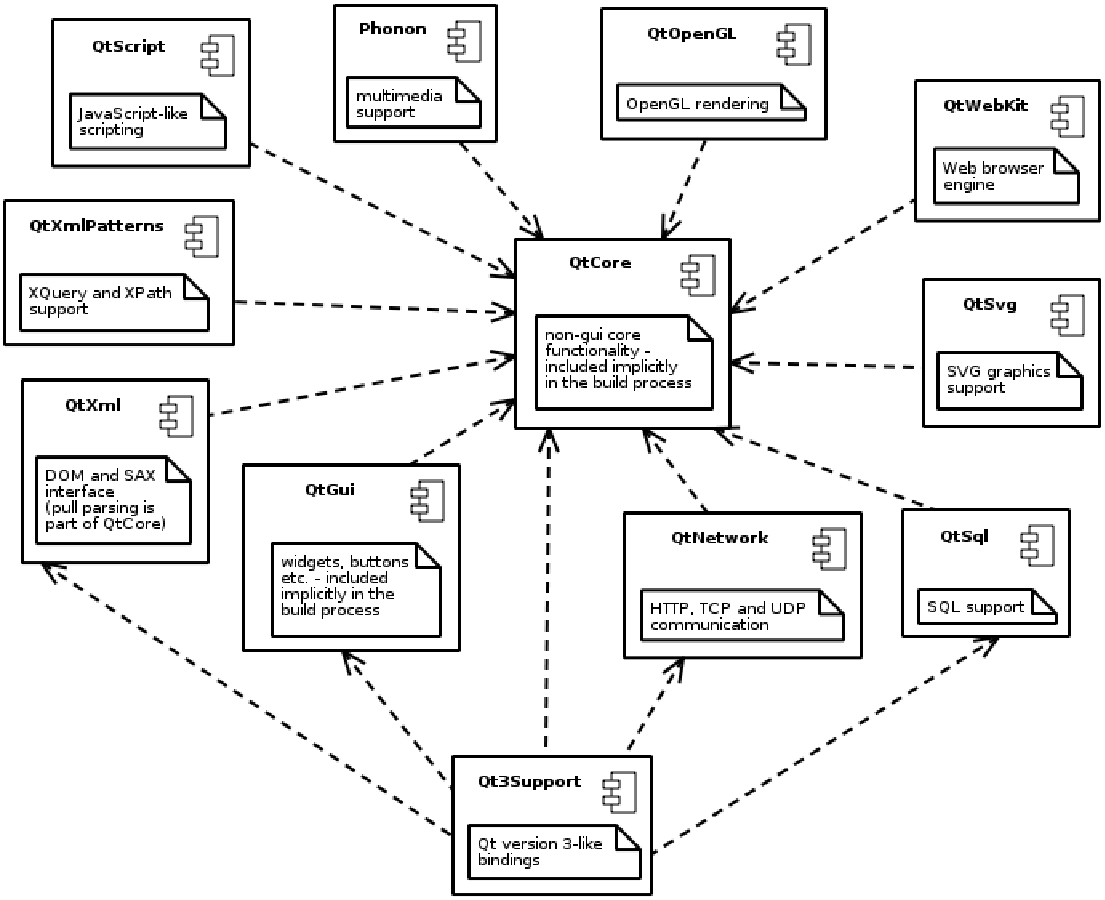

QtScript
JavaScript-like scripting
Phonon
multimedia support
QtOpenGL
OpenGL rendering
QtWebKit
Web browser engine
QtXmlPatterns
XQuery and XPath support
QtCore
non-gui core functionality - included implicitly in the build process
QtSvg
SVG graphics support
QtXml
DOM and SAX interface (pull parsing is part of QtCore)
QtGui
widgets, buttons etc. - included implicitly in the build process
QtNetwork
HTTP, TCP and UDP communication
QtSql
SQL support
Qt3Support
Qt version 3-like bindings

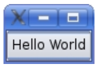
Figure 3: Qt library component diagram

Listing 3: A simple "Hello world" program with Qt

```
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    QPushButton *button = new QPushButton("Hello World");
    button->show();
    return app.exec();
}
```

Figure 4: Hello World on Linux



The details of this example are explained in later chapters.

Another important benefit of Qt is its platform-independency: Applications can be coded once and then run on different operating systems by just rebuilding the application, but without the need to rewrite the code. The most important supported platforms are Windows (XP/Vista/2000 etc.), X11 (including Linux) and Mac OS X (for a full listing of supported platforms and compilers, see [22]). As an example of the platform independency of Qt, consider an application that starts a process. Qt contains a *QProcess* class, which provides functionality to start new processes and communicate with them. A program using that class which is compiled on Windows, will use the "CreateProcessA" or "CreateProcessW" calls from the Windows API ([16]); while the same program compiled on Linux will "fork" (as described in the POSIX standard [13]) to create the process. The *QProcess* class encapsulates this operating system specific functionality to create processes by providing a "start" function.

Similar techniques are applied for the classes *QDir* (directories), *QThread* and *QFile* (for reading from and writing to files), among others. Actually, the most frequently used cross-platform classes in Qt are GUI elements like widgets, toolbars, buttons etc., but since this document focuses more on low-level functionality like networking, the GUI parts of Qt are not discussed here.

qmake: Cross-Platform Build Tool. Qt does not only abstract from operating system functionality as described above, but also offers a platform-agnostic build system. The qmake tool ([23]) maintains a project file, which is the same on every platform, and generates platform-dependent Makefiles from that. Thus, building with the qmake tool consists of generating the Makefile via the command-line tool "qmake" (and the project file as input), and then invoking the platform specific make tool (e.g. "nmake" for the Microsoft build tool or "make" for the GNU make tool on Linux). Consider again the "Hello World" application from Listing 3; a platform-independent project file (called .pro file) looks like in listing 4.

Listing 4: A platform-independent project file

```
TEMPLATE = app
TARGET = helloworld
SOURCES += main.cpp
```

The meaning of the configuration values are the following:

TEMPLATE: Specifies the type of compilation target, e.g. “app” (executable) or “lib” (library).

TARGET: The name of the application (will in the example turn into “helloworld” on Unix and “helloworld.exe” on Windows).

SOURCES: The C++ source files to be compiled.

There are a lot more options for .pro files, listed under [23] (e.g. C++ header files, building in debug and release mode, GUI forms as described below etc.).

Qt Designer. Qt also offers an application for building forms in WYSIWYG mode (see Figure 5). To use a created form in an application, Qt comes with a source code generator called “uic” (user interface compiler) which creates C++ header files from Qt designer form files. The user interface compiler can be either invoked explicitly from the command line or automatically, by including the form file name in a project file. The created header file allows C++ programs to access the GUI elements: If, for instance, the title line edit in Figure 5 below was called “lineEdit”, the generated header file allows access via code like “lineEdit->setText(’my text’)”.

There are several parts of Qt that are not discussed here, for instance internationalization support and scripting; the following chapters rather dig deeper into the Qt library.

3.1 Meta-Object system

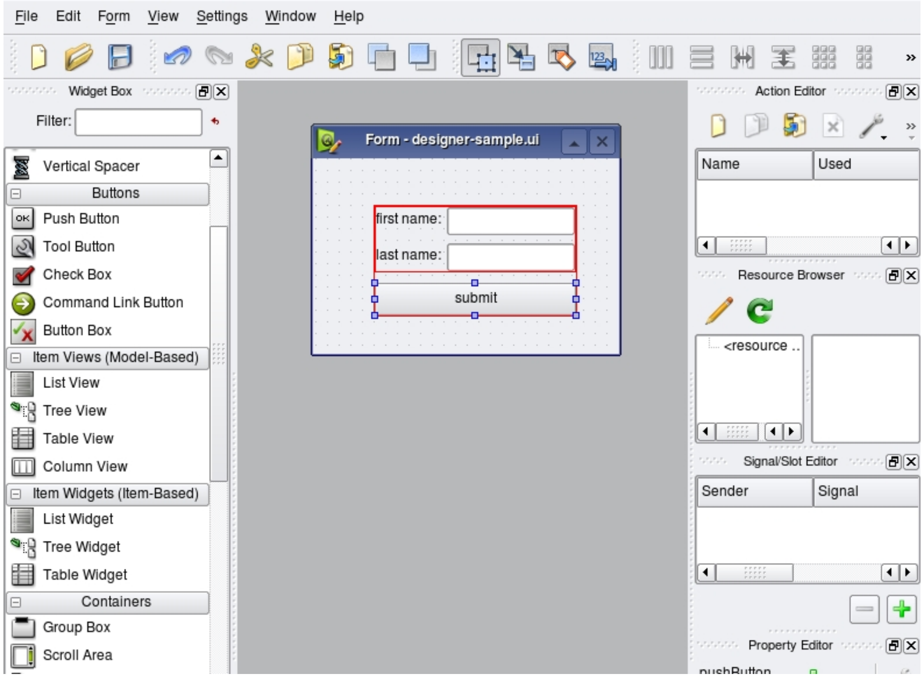
Qt comes with an own meta-object system (described at [21]). This enables some convenient features that are usually not there in C++ programs, the most important of them being *type introspection,* the *signals and slots* mechanism and Qt’s *property system.* To enable these features for a class, it needs to inherit from the class “QObject”, which is the base class of al Qt objects, and additionally include the “Q_OBJECT” macro in the private section of its class declaration, as Listing 5 shows.

Listing 5: A class using Qt’s meta-object system

```
class MyObject : public QObject {

    Q_OBJECT
```

Figure 5: The Qt designer



```
public:
    void doSomething();
private:
    int myMember;
};
```

To use the meta-object features for a class, Qt comes with a command-line tool called "moc" (Meta-Object Compiler); this tool can, similar to the user interface compiler described earlier, be invoked via the command line, but usually is invoked automatically via the qmake build system. The meta-object compiler creates for each class that inherits from "QObject" and includes the "Q_OBJECT" macro a meta class; this meta class can be accessed via the method "metaObject()" of the class "QObject"; this relationship is shown in Figure 6. Note that the meta class is retrieved via the "metaObject()" call and has no name by itself. The meta-object system enables the following features:

introspection. The meta object offers the method "className()", which "returns the class name as a string at run-time, without requiring native run-time type information (RTTI) support through the C++ compiler" ([21]). Additionally, the "QObject" class contains a method "inherits()", which lets class instances determine whether they inherit from a specified class.

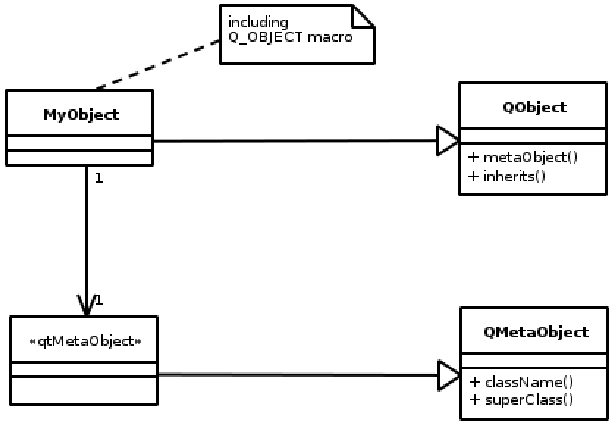


Figure 6: The Qt meta-object system as class diagram

signals and slots. This feature is described in chapter 3.2.

property system. Since C++ does not have native language constructs for properties (unlike e.g. C#), there is a Qt extension to support class properties ([24]): Using the "Q_PROPERTY" macro, a class that inherits from "QObject" can declare its own properties, as Listing 6 shows. Here, the class contains a property called "priority" of type "int", can be read with the member function "priority()" and set via the function "setPriority()". The benefits of using a Qt-style property over using a standard private class member with getter- and setter-methods are twofold: First, a class' properties can be queried by name: Consider for example Listing 6 again: The property "priority" can be set how one would expect via "setPriority(1)", but also via "setProperty('priority', 1)". This lets a programmer set properties of classes he does not know about at compile time. As [24] shows, the property system allows even to discover all properties of a class at runtime. The second benefit of using properties is that they can be added to and removed from an instance of a class at runtime.

Listing 6: A class using Qt's properties

```
class MyObject : public QObject {

    Q_OBJECT
    Q_PROPERTY(int priority READ priority WRITE
        setPriority)
public:
    void doSomething();
```

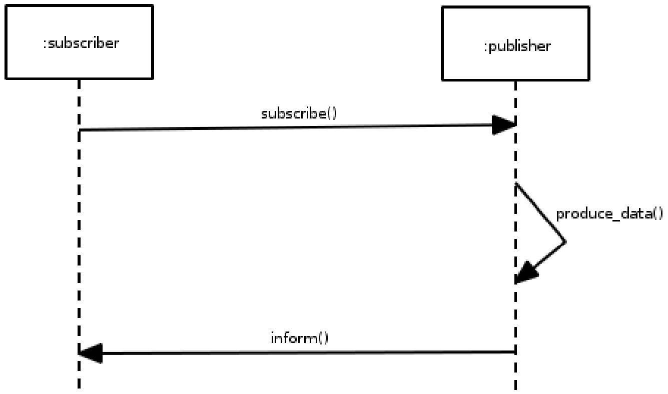


Figure 7: The publish-subscribe pattern as sequence diagram

```
    void setPriority(int priority);
    int priority() const;
private:
    int myMember;
};
```

3.2 Signals and Slots

The term *Signals and slots* describes Qt's way of inter-object communication. Actually, C++ does offer a technique for connecting objects between each other, namely function callbacks: Imagine object B (the *subscriber*) wants to be notified upon a specific event originating in object A (the *publisher*), then object B tells object A to be notified by registering a function pointer with object A; upon the event is emitted, object A calls the registered function at object B (this technique follows the Publish-Subscribe or Observer pattern from [8]). This interaction is depicted as a sequence diagram in Figure 7. However, according to [25] this technique has two major drawbacks: First, it is not type-safe: Whether object A invokes the callback with the right arguments or not can only be determined at runtime. Second, using functions as arguments ties the objects strongly together, and might provide tight coupling between objects that otherwise need not rely on each other at all.

Signals and slots ([25]), on the other side, allow a total decoupling of the sending and the receiving object: Any class dervied from "QObject" can contain signals and slots; a class emitting a signal does not know which class receives that signal (that is, which class' slot has been called). With other words, a *signal* can be viewed as the source of an event, and a *slot* as its sink. Connecting a signal to a slot can be done via the *static* method "connect()" of class "QObject", thus connecting a signal to a slot (or multiple slots) can be done anywhere in code, not necessarily resident in the object containing the signal or the slot.

Figure shows the sequence diagram of the now decoupled sender (publisher) and retriever (subscriber).

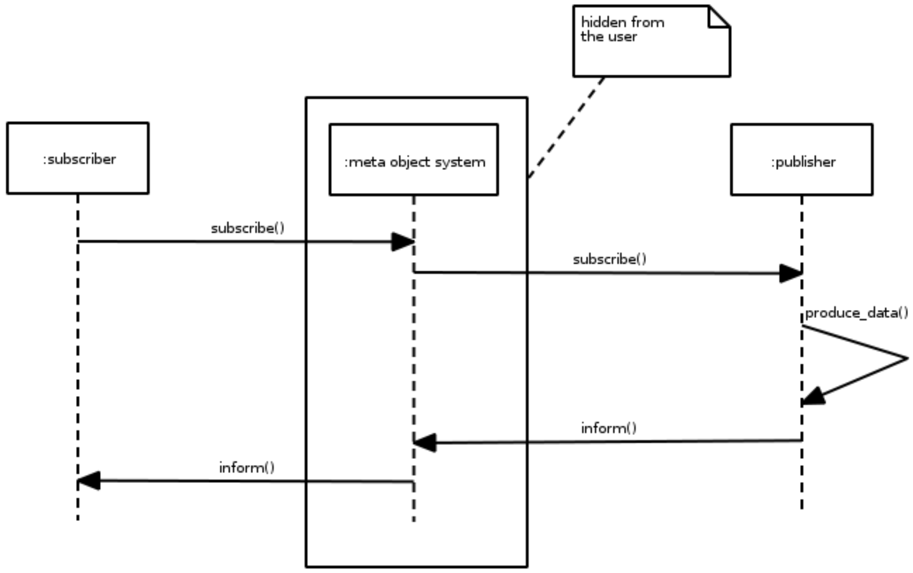


Figure 8: Signals and slots as sequence diagram

In order for code using the "connect()" call to compile, the signature of the signal and the one of the slot need either be identical, or the slot can have a shorter signature than the signal (which means that the slot ignores the extra arguments); this ensures type-safety for the Signals and Slots mechanism.

As a simple example, consider again the "Hello World" example from Listing 3. To quit the application whenever the user clicks on the button, the signal "clicked()" of class "QPushButton" needs to be connected to the slot "quit()" of class "QApplication", as shown in Listing 7.

Listing 7: A simple Signals and Slots example

```
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    QPushButton *button = new QPushButton("Hello World");
    QObject::connect(button, SIGNAL(clicked()), &app,
        SLOT(quit()));
    button->show();
    return app.exec();
}
```

This example connects the button's signal with the application's slot without tying them together. In practice, classes in the Qt library provide a lot of signals and slots suited for GUI programming; some self-describing examples for signals are "QLineEdit::returnPressed()" and "QProgressBar::valueChanged()"; some examples for slots are "QWidget::hide()", "QWidget::showFullScreen()" and "QTextEdit::paste()".

3.3 Basic tools needed for SOAP support

This section shall introduce the modules and classes necessary for SOAP support, i.e. networking and XML. The SOAP specific implementation itself is described in chapter 5.

3.3.1 Networking

The Qt library comes with extensive networking support. As Qt is mostly used as a GUI framework, most network classes concentrate on client side networking, i.e. there are classes for issueing HTTP, TCP and UDP requests, but no HTTP server class (there is, however, a TCP server class). Additionally, the SOAP stack in chapter 5 only supports client-side SOAP over HTTP, so here only the client-side HTTP networking classes shall be considered.

As of version 4.4, Qt contains a fully HTTP 1.1 compatible HTTP stack, called the Network Access API. It consists of the following classes:

QNetworkAccessManager: This class manages sending the request over the network via its methods "get", "post", "put" and "head", which resemble the respective HTTP verbs. Upon receiving the response, it issues the signal "finished()"; this signal can then be connected to a slot in a custom class to parse the response. The "QNetworkAccessManager" class also handles proxy and cookie management as well as SSL errors.

QNetworkRequest: This class resembles an HTTP request. It contains the URL and HTTP header fields, but not the payload data (which is handled separately when e.g. "post"ing or "put"ting via the QNetworkAccess-Manager).

QNetworkReply: This class resembles an HTTP reply, containing the HTTP response headers and the payload data.

Figure 9 shows the common invocation flow of the Network Access API. As an example, Listing 8 shows how the Network Access API can be used: The class "NetworkSample" contains a slot called "replyFinished()" that just prints out the body of the reply to the debug output. The main function instantiates a NetworkAccessManager and a NetworkSample, and connects the manager's signal with the sample's slot (line 25) and "get"s the URL "http://www.trolltech.com" (line 26). Upon receiving the response, the sample prints it out on the debug output.

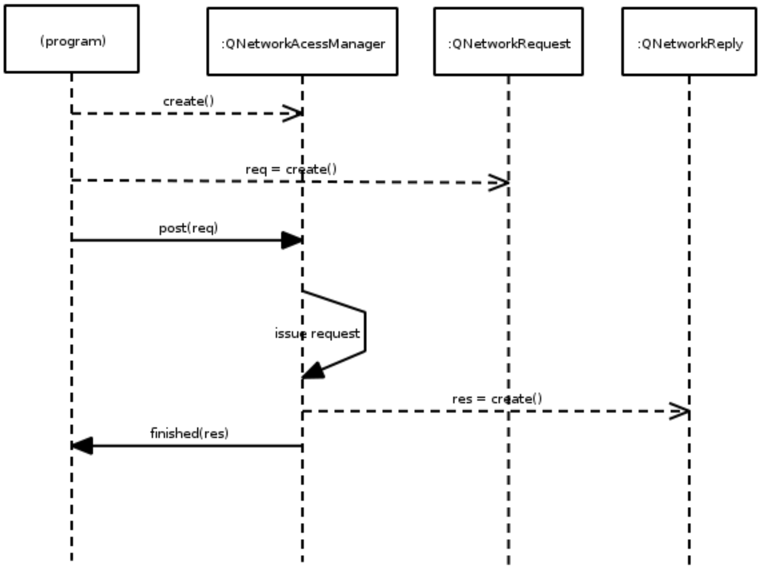


Figure 9: The Network Access API invocation flow

The network access API only supports asynchronous calls via Signals and Slots; retrieving a resource synchronously must be implemented by a user.

Listing 8: example usage of the Network Access API

```
——————————— networksample.h ———————————
#include <QObject>
#include <QNetworkReply>
#include <QDebug>

class NetworkSample : public QObject {
    Q_OBJECT
    public slots:
    void replyFinished(QNetworkReply *reply) {
        qDebug() << reply->readAll();
    }
};

——————————— main.cpp           ———————————
#include <QApplication>
#include <QNetworkAccessManager>
#include <QNetworkRequest>
#include <QNetworkReply>
#include "networksample.h"

```

```
int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    NetworkSample sample;
    QNetworkAccessManager manager;
    QObject::connect(&manager, SIGNAL(finished(
        QNetworkReply*)), &sample, SLOT(replyFinished(
        QNetworkReply*)));
    manager.get(QNetworkRequest(QUrl("http://www.
        trolltech.com")));
    return app.exec();
}
```

3.3.2 XML

Qt provides extensive support for dealing with XML documents and data. It includes classes for accessing XML via the SAX2 interface, the DOM interface and via streaming (see also [1]). The SAX and DOM implementations are not covered here, since they are not used in the SOAP implementation described in chapter 5. The XML classes that are used, however, are the XML stream classes (XML streaming is also known as "pull parsing").

A note on XML pull parsing

XML pull parsing is somehow similar to the SAX API: Events are reported while the XML data is read in; however, unlike with SAX, a pull parser need not provide callbacks, but requests events itself. What this means is that a SAX API provides callbacks that are called from the parser whenever a part of the documents are read; as an example (taken from [1], chapter 15), when parsing the document from Listing 9, the following events are reported to the respective event handlers:

Listing 9: A simple XML document

```
<doc>
    <quote>Ars longa vita brevis</quote>
</doc>
```

```
startDocument()
startElement("doc")
startElement("quote")
characters("Ars longa vita brevis")
endElement("quote")
endElement("doc")
endDocument()
```

When using pull parsing, the user is in control of retrieving events by telling the parser to submit the next token. A pseudo-code example for a pull parser is given in Listing 10. Here, the user can control when events are delivered by telling the parser to read the next token. By controlling the events itself, the user can for instance stop parsing at a certain point, as indicated in the listing (something that would not be possible with SAX). This is especially useful for large XML documents, because then only its necessary parts can be parsed, or parsing can be delayed etc.

Listing 10: A pull parser pseudo code

```
while (not finished) {
    parser.readNext();
    if(too much memory consumed)
        break;
    switch(parser.tokenType()) {
        case StartElement:
            // do something with the start element
        case Text:
            // do something with the text
        case EndElement:
            // do something with the end element
    }
}
```

Qt comes with two classes supporting XML streaming: “QXmlStreamReader” and “QXmlStreamWriter”. The reader class behaves very much like the sample code in Listing 10; how the classes are used in practice is described in chapter 5.

The Qt library is built up of several modules, so that an application only needs to reference the libraries it uses: For instance, the Qt SOAP framework explained later needs the networking and XML classes described above. For networking it needs to reference the QtNetwork module; however, the XML streaming classes are part of the QtCore module, which is always included implicitly.

The Qt library also supports XQuery, which is neither used nor described in this document; there is also work going on on XSLT and XPath, two techniques which indeed are used extensively in the SOAP implementation in chapter 5. But since the Qt XSLT and XPath support is not yet stable, functionality from other libraries had to be used.

4 Web services in use

The theoretical aspects of Web services have been explained in chapter 2.1; this chapter does not only try to explain different Web service standards, but also how they are used in practice and how different standards are typically used together (e.g. SOAP and WSDL).

There are a lot of well-known Websites that offer a Web service API, some of them use SOAP as an interface, and some other techniques like REST; the reasons for choosing either technique are explained in the remainder of this chapter. Some examples for services using a SOAP interface are: eBay API[2], Microsoft MSN Search[3] and Google AdWords[4].

Examples for Web services using a REST interface are: Facebook[5], Flickr[6] and Youtube[7].

4.1 Motivations for using SOAP

This chapter will explain advantages of SOAP over other Web service techniques and common usage scenarios.

The SOAP protocol is specified by the W3C and comes in two versions: Version 1.1 [27] which dates from 2000 and version 1.2 [29], which became a W3C Recommendation in 2007. Although version 1.1 is only a W3C Note and version 1.2 is a W3C Recommendation, version 1.1 is still in frequent use; that is the reason why the implementation part of this document used version 1.1. From a practical point of view there are not many important changes between the two versions; for instance, the namespaces of the SOAP specific XML elements have changed, and SOAP 1.2 is based on XML infoset, while SOAP 1.1 was based on XML 1.0 (which means that version 1.2 could be encoded in a binary XML format instead of the common text encoding). An overview of changes between the two versions can be found at [12].

Some general advantages of SOAP are:

Platform / programming language independence. Since SOAP uses XML to transmit data on the wire, it can be used with any programming language that has XML support, which is virtually every widely-used language. In fact, chapter 4.3 will describe SOAP toolkits written in Java, C#, Perl and C++.

open W3C standard. The SOAP standard is openly available at [28], so new SOAP toolkits can be implemented and tested for correctness by everybody. Furthermore, the SOAP standard was developed by Microsoft, IBM, Lotus Development Corp. and others, and thus can be viewed as a consense of different companies.

[2] http://developer.ebay.com/developercenter/soap/

[3] http://search.msn.com/developer/

[4] http://www.google.com/apis/adwords/

[5] http://developers.facebook.com/documentation.php

[6] http://www.flickr.com/services/api/

[7] http://code.google.com/apis/youtube/overview.html

Human readability of the generated XML. As is common with everything based on XML, this can be considered an advantage and a drawback. As an advantage, XML is humanly readable and therefore easy to debug; this comes with the disadvantage of generating a lot of overhead when transmitting data (especially over a slow connection, e.g. from a mobile phone) and when parsing that data. Here, binary protocols show usually better performance.

attachment support. SOAP supports efficient ways of transmitting attachments ("attachments" here means binary data not inlined in the XML itself) via either MIME or, more efficiently, an own W3C standard called MTOM ([10]).

underlying protocol independency. SOAP is not dependent on an underlying protocol. In fact, the most common case is using SOAP over HTTP, but there are also toolkits supporting e.g. TCP, UDP, JMS or XMPP. This allows not only the nodes that are processing SOAP messages to be independent of each other, but also the underlying network structure to be independent of a specific protocol.

automatic code generation. When transmitting large data sets between different nodes, it is often tedious to build up the XML by hand. As the next chapter explains, SOAP (in addition with WSDL) allows for automatic code generation; this code allows to encapsulate the XML layer of a SOAP call into a programming language dependent class API. By abstracting from the XML layer, a user does not have the burden of dealing with XML peculiarities itself.

4.1.1 common SOAP usage scenarios

As already pointed out, SOAP describes an XML message format; moreover, for the server side, it contains a procedure for handling these messages (i.e. what to do in case of failure, order of processing the different elements etc.). However, "using SOAP" actually means using SOAP in connection with other standards like WSDL, XML Schema and several standards from the Web Services Interoperability Organization[8].

SOAP communication architecture

The by far most common scenario for exchanging messages and interfaces, and the only one treated in this document, is to use an interface specified in WSDL, which is then retrieved by the client to generate code that uses messages encoded in SOAP for transmission on the wire. This scenario is depicted in the Figures 10 and 11.

[8] http://www.ws-i.org/

Figure 10: The common SOAP usage scenario - part 1

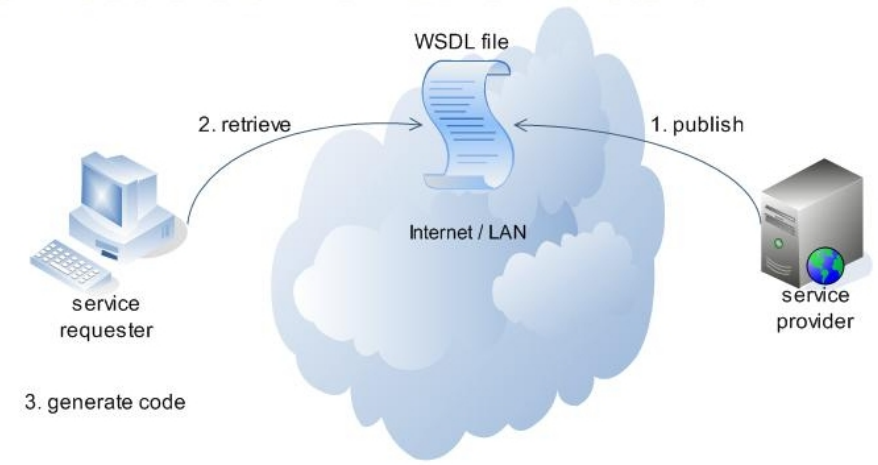
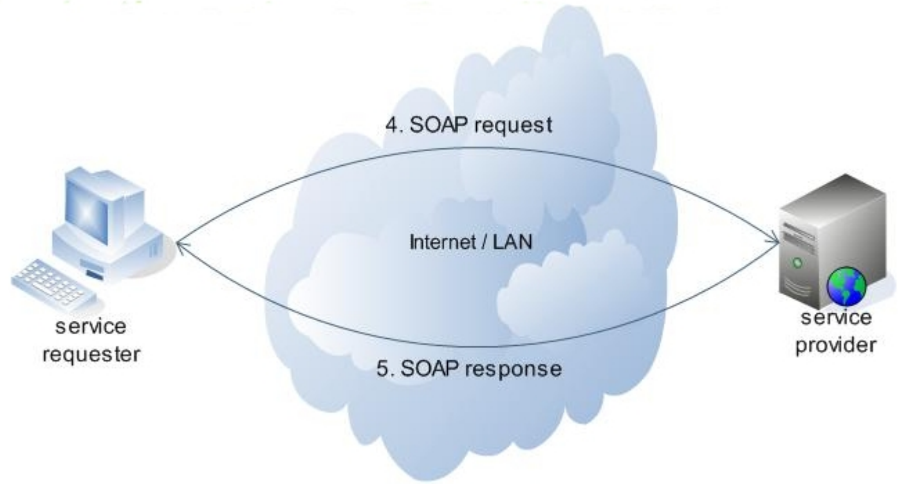


Figure 11: The common SOAP usage scenario - part 2



The reason for splitting the scenario in two diagrams is that the scenario consists of two phases: The first phase, shown in Figure 10, shows the "*compile time*" of the scenario:

1. *publish*: The service provider publishes a WSDL interface. As this step is part of the server side component of SOAP, it is not described in this document. However, a WSDL file is usually automatically generated from a programming language specific interface (e.g. a Java interface or a C++ header file). After creating the WSDL file, it is typically uploaded to a Web server.

2. *retrieve*: The service requester retrieves the WSDL file. Usually, the service provider just tells the requester the URL of the WSDL file. More advanced techniques like Web service registries (e.g. UDDI) are not treated here.

3. *generate code*: The service requester generates program code from the WSDL file. For this purpose, SOAP toolkits usually offer a command-line tool to create programming language code from the WSDL (e.g. "Wsdl2Java" from the Axis framework or "Wsdl2h" from the gSOAP framework).

Figure 11 shows the "*runtime*" phase of the scenario:

4. *SOAP request*: The service requester issues a SOAP request to the service provider. As already explained, this happens by using the code generated in step 3, thus abstracting from the XML layer.

5. *SOAP response*: The service provider sends back a SOAP response. Again, the requester accessess the response via the generated code.

SOAP message structure

When sending a SOAP message, there are different message formats, depending on the role of the message and the used datatypes (explained also in e.g. [30], chapter 4.2.5):

Distinguishing between the programming model, there are two different SOAP message types: *document* and *RPC*. A SOAP message typed *document* is considered an XML document of any form sent to the message receiver; an *RPC* message is considered to contain one procedure name to be invoked as single child of the SOAP body. That procedure name contains as children arguments that need to be passed along with the procedure call. So a *document* SOAP message can contain any XML document, while a *RPC* message typically contains one single XML element (as child of the SOAP body); that single element can contain a list of (in most cases simple) arguments.

Distinguishing between the datatypes used in the message, there are two different data encodings: *SOAP encoding* and *literal encoding*. *SOAP encoding* uses the datatype encoding described in section 5 of the SOAP standard ([27], therefore it is also often called "Section 5 encoding"). A SOAP element with section 5 encoding containing a string could look like '<myString type="SOAP-

ENC:string">hello</myString>', with the "SOAP-ENC" prefix set to a special URI ("http://schemas.xmlsoap.org/soap/encoding/") in the SOAP envelope. The other option, *literal encoding*, does not specify the data type of an element in the SOAP message itself. In that case, the datatype of an element is given in a WSDL file (see chapter 4.1.2) and is usually described with XML Schema (also described in chapter 4.1.2).

Combining these options, the most one used in practice is "document/literal". Sometimes the "rpc/encoded" style is applied, but as of this writing (August 2008) it is mostly outdated. The WS-Interoperability Standard ([5], see below) even mandates the usage of "document/literal" SOAP messages, so this one is the only one treated in this document. Other combinations of the styles described above are virtually never used.

4.1.2 standards used in conjunction with SOAP

As the former chapter stated, there are several standards used in conjunction with SOAP:

WSDL

As already said, WSDL ([3]) stands for "Web Service Description Language", and is XML-based. A WSDL file describes all details that are necessary to make a valid SOAP call; consider again Listing 1. What the requester needs to know to make the request is: The URL to post the request to, the HTTP header field "SOAPAction", and the structure of the body. All this information is contained in a WSDL; Listing 11 shows a complete WSDL file (in the currently most widely used version 1.1). It consists usually of the following parts:

- An XML Schema part (Lines 3 - 20). This is described in the next section.
- One or more "message" elements. A WSDL message is considered an entity sent by one SOAP node to another.
- One or more "portType" elements. "A port type is a named set of abstract operations and the abstract messages involved" ([3]).
- One or more "binding" elements. "A binding defines message format and protocol details for operations and messages defined by a particular portType" ([3]). For instance, the most usual used SOAP binding is the HTTP binding.
- One or more "operation" elements inside a "portType" and a "binding". An operation is an action performed on the server side; it can be viewed as resembling a programming language procedure (e.g. C function or Java/C# method).
- One or more "service" elements. "A service groups a set of related ports together" ([3]).

- One or more “port” elements inside a “service” element. A port is an individual endpoint for a binding, i.e. it defines an URL to access a service.

Listing 11: A sample WSDL file

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="StockQuote" targetNamespace="StockQuoteURI" xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="StockQuoteURI">
  <types>
    <xs:schema elementFormDefault="qualified" targetNamespace="StockQuoteURI">
      <xs:element name="GetLastTradePrice">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="symbol" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="GetLastTradePriceResponse">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="price" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:schema>
  </types>
  <message name="PriceRequest">
    <part element="tns:GetLastTradePrice" name="parameters"/>
  </message>
  <message name="PriceResponse">
    <part element="tns:GetLastTradePriceResponse" name="parameters"/>
  </message>
  <portType name="StockQuotePortType">
    <operation name="GetLastTradePrice">
      <input message="tns:PriceRequest"/>
      <output message="tns:PriceResponse"/>
    </operation>
  </portType>
  <binding name="StockQuoteBinding" type="tns:StockQuotePortType">
    <operation name="GetLastTradePrice">
```

```
      <soap:operation soapAction="StockQuoteAction" style
         ="document"/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
  <service name="StockQuoteService">
    <port binding="tns:StockQuoteBinding" name="
        StockQuoteBinding">
      <soap:address location="http://www.stockquoteserver
          .com/StockQuote"/>
    </port>
  </service>
</definitions>
```

XML Schema

XML Schema ([4]) is a language to describe the structure of XML documents. Unlike Document Type Definitions, XML Schema documents are also based on XML. The language is used in WSDL files to describe the structure of the XML sent inside the SOAP body and header data. It will not be described here, but it is necessary to deal with when writing a code generator in chapter 5. As already said, the SOAP standard comes with an own data encoding, but the WS-Interoperability standard ([5], see below) forbids its use.

WS- standards*

To enrich the functionality of SOAP calls, different organizations (W3C, OASIS[9]) have standardized additional protocols supporting the following features:

- WS-Interoperability ([5]): does not really provide new features, but narrows down the SOAP (and WSDL) specification in order to improve interoperability between different SOAP implementations.
- WS-Addressing[10]: supports transport protocol independent routing information; for instance, the “SOAPAction” HTTP header field contains such information, but is tied to HTTP. WS-Addressing includes that information in the SOAP message itself.
- WS-Security[11]: supports signing and encryption of SOAP messages.

[9] http://www.oasis-open.org/

[10] http://www.w3.org/Submission/2004/SUBM-ws-addressing-20040810/

[11] http://www.oasis-open.org/specs/index.php#wssv1.0

- WS-ReliableMessaging[12]: supports sending messages reliably between SOAP nodes. Of course, as HTTP uses TCP, SOAP messages sent over HTTP can be viewed as delivered reliably. However, WS-ReliableMessaging also guarantees reliable delivery in case of spontaneous network or system failures.
- WS-Policy[13]: supports publication of a node's policies like security features, quality of service etc.

There are a lot more of those standards (which are also implemented in SOAP toolkits), but the ones listed above are probably the most important ones.

4.2 Motivations for using other web service techniques

There may be situations where SOAP is not the right choice for accessing services or making services accesible over a network. Among the disadvantages of SOAP are:

- *Huge message overhead / slow performance.* When sending few information via a SOAP call, the amount of SOAP protocol specific information (envelope, header, body etc.) might be even higher than the actual data the SOAP node wants to send. Furthermore, using a binary format (e.g. Java RMI) could result in lower transmission and message parsing time.
- *complex usage.* First, the SOAP standard is often not very specific and leaves room for different options (e.g. HTTP or TCP or something else as underlying transport protocol, text or binary XML representation, SOAP specific encoding or XML Schema). Moreover, for sending only few data between network nodes going through a automatic code generation procedure might be overkill, and just building up the XML "by hand" is the better solution.
- *imperfect interoperability.* In practice, generating WSDL code from programming language code on the server side and the reverse procedure on the client side do not work out-of-the-box and often require some editing by hand. Again, this could result in too complex usage.

In this chapter some techniques are described that could be considered an alternative to SOAP.

4.2.1 Ajax

Ajax stands for "Asynchronous JavaScript and XML". It is used in Web Sites to improve user experience and fasten responsiveness: Using Ajax, the user does not interact in the usual "click and wait" scenario with a Web page, but by

[12] http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-rm/ws-reliablemessaging200502.pdf

[13] http://www.w3.org/Submission/WS-Policy/

dynamically reloading small parts of the page: With the traditional web site browsing scenario, the user controlled at what time an HTTP request was made to the web server (e.g. when clicking a link or the "back" button of the browser). However, when using Ajax, small portions of data are requested from the web server by a layer called the "Ajax engine"; these requests transfer less data than when loading a whole HTML page, but are made more often, which results in better responsiveness and user experience. This scenario is depicted in Figure 12.

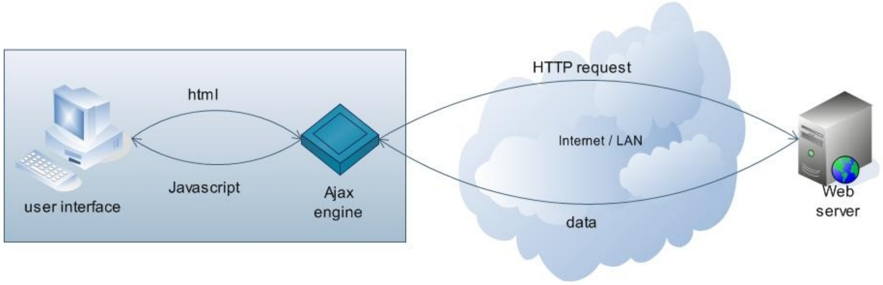


Figure 12: Web site browsing with the Ajax engine

The way the user interface interacts with the Ajax engine (i.e. how the browser displays the data received by the engine) is not explained in detail here; it is sufficient to know that the user interface processes the data to eventually display HTML. The more relevant part is the HTTP request to the server and the corresponding response. As Figure 12 shows, the overall concept of communication between the service requester (Ajax engine) and the service provider (Web server) are similar to the SOAP communication flow: The Ajax engine issues an HTTP request and gets back an HTTP response with the data.

However, the design goals of Ajax and SOAP are different. According to [31], the most important requirement of an Ajax call is to be fast to enhance web site responsiveness. The term "fast" here means both fast in transmitting data to and from the server as well as fast in parsing the server response and update the web pages' contents. Depending on how much and what data is transferred to and from the server, there are different formats for data carried in Ajax calls:

- *HTTP GET / POST*: If possible, carrying no payload data from the Ajax engine to the Web server is ideal; in that case, a simple HTTP GET with information specified in the URL can be used. If the data should or can not be sent in the URL, an HTTP POST with content type "application/x-www-form-urlencoded" can be used, sending simple key/value pairs in the form "key1=value1&key2=value2".
- *XML*: As the "x" in Ajax stands for XML, this is the usual way of transmitting data, especially from the Web server to the receiver. However, one

disadvantage of XML is the high overhead of the transmitted data on the wire; furthermore, when an Ajax engine receives XML, it often needs to be parsed before being displayed. Because of that, there are other solutions saving bandwith and parsing time:

- *JSON*: Being an abbreviation for “JavaScript Object Notation”, JSON is a syntax for storing JavaScript literals, namely objects and arrays. The Listings 12 and 13 show the representation of an array in XML and JSON (adapted from [31]), with the latter one needing less bytes for representation. This difference may seem marginal here, but when transmitting a lot of bytes, it can lead to a performance improvement using the JSON approach. Moreover, when an Ajax engine receives a JSON construct, it need not be parsed by hand to extract information into JavaScript language constructs, but can be included via JavaScript’s “eval()” function for interpreting code. Thus, JSON is a good alternative over XML when making Ajax calls.

Listing 12: sample XML representation

```
<classinfo>
    <students>
        <student>
            <name>Michael Smith</name>
            <average>99.5</average>
            <age>17</age>
            <graduating>true</graduating>
        </student>
    </students>
</classinfo>
```

Listing 13: sample JSON representation

```
{ "classinfo" :
    {
        "students" : [
            {
                "name" : "Michael Smith",
                "average" : 99.5,
                "age" : 17,
                "graduating" : true
            }
        ]
    }
}
```

As a summary, Ajax and SOAP share the same overall communication model, namely requesting and receiving data from a service provider. This data is usually formatted in XML, with JSON being a good alternative. While sharing the same communication model, Ajax and SOAP have different application areas: While SOAP offers a generic model for communication over the network, Ajax is tied closely to displaying HTML in Web pages.

As [31] explains, making SOAP calls with Ajax is possible, yet tedious; when a service provider offers functionality that is to be used within Web pages, an Ajax interface is better than a generic SOAP interface, which was the case why Google shut down its SOAP interface and recommended using its AJAX search API instead [14].

4.2.2 Java RMI

Java RMI[15] (*R*emote *M*ethod *I*nvocation) is a Java API for performing distributed computing over a network. As Java is an object-oriented language, distributed computing in that case means supporting distributed objects, i.e. a program running in a Virtual Machine can interoperate with objects resident in other Virtual Machines on other systems. With RMI, performing remote calls tries to mimic performing local calls, thus hiding the low-level network communication completely from the programmer.

This chapter only describes the parts of RMI that have a functional equivalent with the SOAP technology; advanced distributed object techniques of RMI like e.g. distributed garbage collection are not described here.

As [19] describes, the core concept of RMI is to separate the definition of functionality and its implementation. Thus, if an object's functionality shall be accessible remotely, a Java Interface (definition of the object's functionality) is made public for remote use, while the object's implementation is not exported. Whenever now a client wants to access a remote object, it does so via a proxy object (following the "proxy" design pattern from [8]), which resembles the object in client space. Both objects, the proxy object in client space as well as the "real" object in server space, implement the same interface. Whenever the proxy object's functionality is invoked, it forwards the call over the network to the object in server space. Figure 13, adapted from [19], illustrates this.

In order to make an object accessible, the object resident in the server JVM must register itself at a naming service. Such a naming service could be the Java Naming and Directory Interface (JNDI) or the RMI built-in service called "RMI registry". The RMI regsitry listens on a port for incoming requests; a server program can publish one of its objects to the registry by registering itself at a RMI-style URL (of the form "rmi://hostName:hostPort/myServiceName").

For a client program to invoke functionality of that remote object, it has to know the URL that the remote object registered itself to, and instantiate the object from that URL via RMI's naming technique. That instantiated object is, as Figure 13 displays, a proxy object; when calls are made to this object, they are forwarded via TCP to the object in server space. The serialization of the call parameters is done in background, without the programmer needing to

[14] http://code.google.com/apis/soapsearch/

[15] http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp

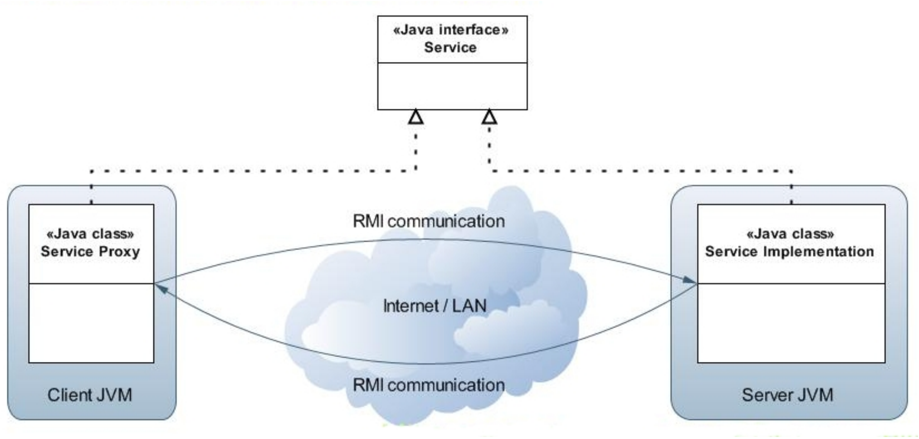


Figure 13: Basic Java RMI architecture

know the details. After instantiating the object from the URL, the object looks like a local object to the programmer.

The serialization of the data from proxy to implementation object and back is done in a binary format; since this binary format uses less network bandwith than XML, RMI is usually more performat than a SOAP-based Web service (see [9]).

Table 1 lists some concepts that are similar in both Java RMI and SOAP.

To summarize this chapter, Java RMI offers a ready-to-go interface for distributed programming. It is not a direct competitor with Web service, since it focuses more on programming distributed objects than on loosely coupled systems as Web services do. That is, RMI offers a lot more functionality needed by distributed objects or middleware systems (e.g. distributed garbage collection and remote object references); by also requiring Java to run on all entities involved, it keeps them coupled together more tightly. As additional benefits, RMI provides lower network bandwith usage and the fact that a lot of the communication and serialization is hidden from the programmer, as Table 1 shows.

Systems interconntected by SOAP Web services, on the other hand, are more loosely coupled than RMI: There are no such concepts like remote object references; the only contract between service provider and consumer is the WSDL file, which describes how the XML sent back and forth should be formed. A service consumer needs to obtain the WSDL to access a service; from that WSDL, he can create code in any language he wants to call the service. That is, using the SOAP approach there is much more freedom to choose (programming languages, code generators etc.), but also more work to do, e.g. the code generation from the WSDL must be invoked by hand.

	RMI	SOAP
programming language	Java	any
interface specification	Java interface	WSDL
service exposure	registering at RMI registry + publishing Java interface at Web server	WSDL generation by code generator + publishing WSDL at Web server
client code creation	proxy object instantiation from service URL (at run time)	code generation from WSDL (at "compile" time)
serialization	binary	XML
underlying protocol	TCP	HTTP
service endpoint specification	RMI URL ("rmi://...") (or JNDI)	HTTP URL (or UDDI)

Table 1: Java RMI and SOAP concepts

4.2.3 REST-based services

REST is an abbreviation for *Re*presentational *S*tate *T*ransfer. It is rather an architectural paradigm than a specified protocol or even a W3C standard. According to [6], REST can be described the following way:

> "Representational State Transfer is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the user progresses through an application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use."

So from the REST point of view, a system is built up as a collection of web pages (or in a broader sense: resources), each of them accessible by "selecting a link" (i.e. each of them is accessible by a URI). Interacting with a REST-based system is done via HTTP.

According to [18], accessing the resources must be considered in two ways:

- *Addressability*: Information to distinguish the resources from each other is kept in the URI.
- *Uniform interface*: Information to distinguish the access method (retrieving, updating, deleting or creating a resource) is kept in the HTTP method: For instance, retrieving a resource is done via an HTTP GET request, while deleting a resource is done via an HTTP DELETE request.

Some examples for REST-based systems are:

- *Web server.* A plain old Web server is a classical example for a RESTful system: Resources are Web pages, which are accessed usually only for retrieving (not for updating); that is, when browsing Web pages, a browser sends only HTTP GET messages to the Web server. However, some Web 2.0 techniques like Ajax described in chapter 4.2.1 violate the REST view of Web sites: Using Ajax, different content can be displayed under the same URL by changing the content dynamically; that is, there can be different resources accessed via one URL.
- *RSS/Atom feeds.* Such feeds are typically used for retrieving updates from a Web site, decoupled from a Web browser context. A newsfeed usually contains an XML document containing the title of the feed as well as a list of feed items (i.e. articles) with their title, URL and publishing date. By checking the publishing date of each article, a feed reader can now retrieve the new feeds via the article URL. Additionally to such a retrieving mechanism, the Atom Publishing Protocol[16] provides means for deleting, creating and updating feed resources.
- *Web site APIs.* A numerous amount of Web sites provide APIs for accessing their resources in a non-browser context; famous examples among them are Facebook[17], Flickr[18], Twitter[19], Last.fm[20] etc., with the resources being contacts, photos, messages and songs, respectively.

Up to now this chapter has only described how resources are addressed; for instance, to upload a photo to Flickr, one would issue a HTTP POST to “http://www.flickr.com/myUsername/myAlbum”. What has to be discussed is the content of such HTTP messages. In fact, REST only describes resource access, but not content formats. Thus, many different content formats are possible:

- XML (either described and built up “by hand” or some XML language, e.g. RSS, Atom or XSPF)
- text formats (e.g. JSON, iCal or url-encoded text)
- binary formats (e.g. photos or videos)

In practice, mostly simple XML is used; i.e. it is no special XML dialect nor specified via an XML schema langauges. Other formats like JSON or text formats are also used.

[16] standardized as RFC 5023

[17] http://developers.facebook.com/documentation.php

[18] http://www.flickr.com/services/api/

[19] http://groups.google.com/group/twitter-development-talk/web/api-documentation

[20] http://www.audioscrobbler.net/data/webservices/

REST compared to SOAP

In the last time, there has been a big debate about REST and SOAP and which one is the better design for a Web Service infrastructure (e.g. [7]). In fact, a lot of the Web site APIs presented earlier in this chapter prefer a REST-based interface to their services over a SOAP interface (e.g. Tim O'Reilly says[21]: "Amazon has both SOAP and REST interfaces to their web services, and 85% of their usage is of the REST interface"). However, REST and SOAP are not direct competitors, but serve rather different fields:

1. *REST is a programming paradigm, SOAP is a W3C standard.* As described earlier in this chapter, the REST paradigm suggests resource-oriented access to a system with four basic access methods (create, read, update, delete) to the resources. It does no mandat the data format used for accessing the system; thus, a lot of heterogeneous systems are considered REST-based. SOAP, on the contrary, is a W3C standard that mandates the usage of XML and describes the structure of SOAP messages.

2. *REST is resource-oriented, SOAP is activity-oriented.* As already explained in detail, the base of a REST-ful system is a collection of resources that are accessed via URIs. SOAP on the other hand, is rather activity-oriented, as its original focus was set on Remote Procedure Calls. However, with a specifically designed WSDL it is possible to mimic a resource-oriented architecture. But in general a REST-based system consists of a lot of resources that are accessed via four easy methods, while a SOAP system consists of a few endpoints (resources) that are accessed via many different (and sometimes complex) methods.

3. *REST is tied closely to HTTP, SOAP is not.* REST is using URIs to identify resources and HTTP verbs to access those resources. That is, REST with e.g. TCP instead of HTTP does not make sense. SOAP, on the other side, can use any underlying protocol for transportation; because of that it can use any underlying protocol only as a transport protocol:

4. *REST uses HTTP as an application protocol, SOAP uses HTTP as a transport protocol (if at all).* REST uses a URI to identify resources and HTTP verbs to access these resources; that is, it uses application-level features of HTTP. There has been a lot of criticism about SOAP using HTTP only as transport-level protocol by tunneling XML data through HTTP POST requests. There is, however, the special HTTP header field "SOAPAction", but it can be avoided by using the WS-Addressing standard. Because a SOAP request always uses HTTP POST, it is not clear from the HTTP semantics what kind of operation is performed. When interacting with a REST-based system, an HTTP POST always resembles

[21] http://www.oreillynet.com/pub/wlg/3005

the change of a resource; for instance, if a resource is only retrieved, an HTTP GET call is used.

5. *REST is easier for programmers to adopt, SOAP is more complex and feature-rich.* Most Web site APIs use a REST-based interface, because it is easier for programmers to get started with. For most of those APIs the queries sent back and forth are easy, so there is no need for SOAP features like code generation, encryption, reliable message transfer etc. If there is no need for those advanced features and an API is to be offered to a broad range of people, REST might be the better solution because of its ease of use.

6. *REST has no interface definition language, SOAP has WSDL.* Most of the APIs of REST-based systems are just described with words[22]; this is enough for easy queries, where an XML document can be built up by hands. In cases, however, where a lot of data is to be exchanged with the service, a machine-processable interface is needed. For this purpose, SOAP systems describe the structure of their calls with the WSDL standard. With a WSDL file a service requester can generate code that hides the low-level XML structure of calls made to the service provider. However, as the new WSDL standard 2.0 ([2]) supports all HTTP verbs, REST can in the future also be used in conjunction with WSDL; but as of this writing (August 2008), WSDL 2.0 has not been adopted widely in practice.

As an example of how the usage of SOAP and REST influence the API of a service, consider again the stockquote example from chapter 2.2, where a Web service offers the following API: looking up the stock quote price of a company and updating it, getting details of a stock price like its course in the past, and getting company details. Figure 14 shows how an API in REST would be based on resources like stock quotes and companies, while a SOAP API rather concentrates on operations and not on addressing entities of a system.

As a general rule of thumb, a system that is supposed to be used by a lot of programmers and does not need advanced features like message encryption etc. might be well suited for the REST approach. As advanced features are needed and a lot of data is sent, SOAP might be the better choice.

4.3 Existing SOAP toolkits

This chapter will analyze different SOAP frameworks written in different programming languages. The focus of the analysis will lie on the following questions:

1. What API does the toolkit provide? How easy to use and how powerful is it?

2. Is code generation from a WSDL file supported?

[22] see for instance http://www.flickr.com/services/api/upload.api.html

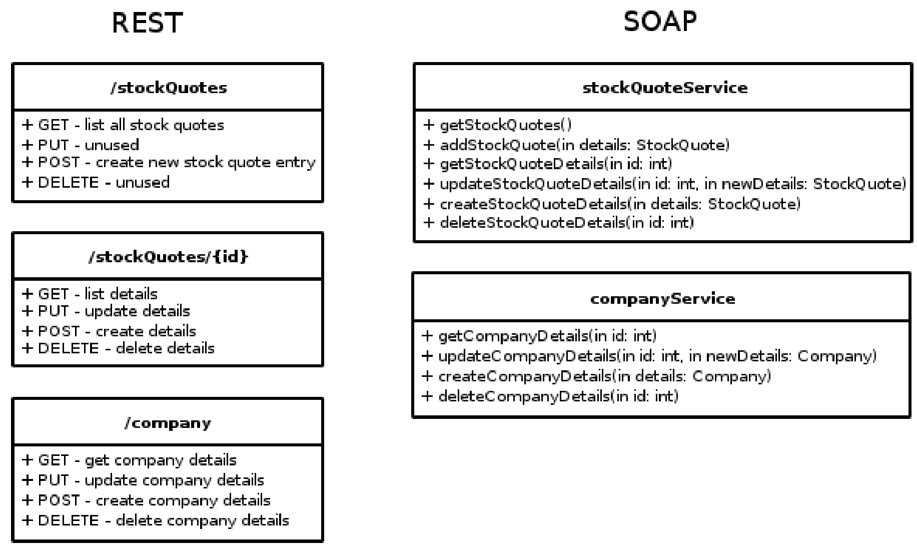

REST
/stockQuotes
+ GET - list all stock quotes
+ PUT - unused
+ POST - create new stock quote entry
+ DELETE - unused
/stockQuotes/{id}
+ GET - list details
+ PUT - update details
+ POST - create details
+ DELETE - delete details
/company
+ GET - get company details
+ PUT - update company details
+ POST - create company details
+ DELETE - delete company details
SOAP
stockQuoteService
+ getStockQuotes()
+ addStockQuote(in details: StockQuote)
+ getStockQuoteDetails(in id: int)
+ updateStockQuoteDetails(in id: int, in newDetails: StockQuote)
+ createStockQuoteDetails(in details: StockQuote)
+ deleteStockQuoteDetails(in id: int)
companyService
+ getCompanyDetails(in id: int)
+ updateCompanyDetails(in id: int, in newDetails: Company)
+ createCompanyDetails(in details: Company)
+ deleteCompanyDetails(in id: int)

Figure 14: REST and SOAP APIs

3. Does the tool comply to Web service standards (SOAP standard and WS-Interoperability) and does it provide support for additional WS-* standards (e.g. WS-Security and WS-Addressing or MTOM for attachments)?

4.3.1 SOAP::Lite (Perl)

SOAP::Lite[23] is a module written in Perl. It supports both SOAP versions 1.1 and 1.2 and a lot of underlying transport protocols[24]: HTTP, HTTPS, TCP, Jabber, MQSeries, FTP and SMTP. The module comes with a command-line tool to generate code from WSDL files.

A disadvantage of SOAP::Lite is its focus on rpc/encoded SOAP calls, which are outdated nowadays, as already explained in chapter 4.1.1.

1. *What API does the toolkit provide?*

SOAP::Lite makes it easy for programmers to begin with. An easy test Web service, that just echoes a string (as shown in the Listings 14 and 15), can be invoked by the perl program described in Listing 16.

Listing 14: A sample SOAP request

```
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/
    XMLSchema-instance" xmlns:soapenc="http://schemas.
```

[23] http://www.soaplite.com/

[24] http://soaplite.com/features.html

```
    xmlsoap.org/soap/encoding/" xmlns:xsd="http://www.w3.
    org/2001/XMLSchema" soap:encodingStyle="http://schemas
    .xmlsoap.org/soap/encoding/" xmlns:soap="http://
    schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <echo xmlns="http://localhost/Demo">
            <myArgument xsi:type="xsd:string">myValue</
                myArgument>
        </echo>
    </soap:Body>
</soap:Envelope>
```

Listing 15: A sample SOAP response

```
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/
    XMLSchema-instance" xmlns:soapenc="http://schemas.
    xmlsoap.org/soap/encoding/" xmlns:xsd="http://www.w3.
    org/2001/XMLSchema" soap:encodingStyle="http://schemas
    .xmlsoap.org/soap/encoding/" xmlns:soap="http://
    schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <echoResponse xmlns="http://localhost/Demo">
            myValue</echoResponse>
    </soap:Body>
</soap:Envelope>
```

Listing 16: A simple SOAP::Lite client

```
 1 #!/usr/bin/perl
 2
 3 use SOAP::Lite;
 4
 5 print SOAP::Lite
 6  -> uri('http://localhost/Demo')
 7  -> on_action(sub { "mySOAPActionHeader" })
 8  -> proxy('http://localhost/')
 9  -> echo(SOAP::Data->name("myArgument" => "myValue"))
10  -> result;
11 # alternative to above line: XPath
12 #-> valueof('/Envelope/Body/[1]/[1]');
```

In Listing 16, a SOAP call is made and its answer parsed immediately in the lines 5 - 10. Line 6 sets the global namespace of all XML elements that are descendants of the SOAP “body” element (in Listing 14, these are the “echo” and “myArgument” elements).Line 7 sets the SOAPAction HTTP header field; line 8 sets the address to post the HTTP request to; line 9 specifies the actual payload data (as children of the "body" element: the call to "echo" results an "echo" XML element as single child of the SOAP "body" element. Since that call in line 9 looks like a function call, this points out the focus on RPC of SOAP::Lite. Arguments to the "echo" element can be supplied as shown in line 9 via SOAP::Data constructs. After the call is made in line 9, the program blocks until the SOAP response has arrived and prints the result in line 10. The call to the "result" function in line 10 returns the contents of the child of the SOAP body element, in the case of Listing 15, the string "myValue" is returned. This again shows how SOAP::Lite is focused on RPC calls, since the first element of the body item is not returned and only one single argument in the return function is expected. Alternatively, SOAP responses can be parsed via XPath, as shown in line 12: This is a very powerful and elegant way to extract information from complex or deeply nested elements.

The SOAP::Lite API is easy to start with, but has several disadvantages:

- setting the "SOAPAction" HTTP Header field: When not overriden, SOAP::Lite assumes the "SOAPAction" field to be of the form "URI#methodName", in the case of Listing 16 it would be "http://localhost/Demo#echo". In many cases, this needs to be overridden, which can only be done in a non-intuitive way, as Listing 16 shows in line 7.

- building up complex XML elements: When not building up XML elements via SOAP::Data constructs as in the example above, SOAP::Lite tries to guess the data type and encodes it with the SOAP encoding (see also the next point). For instance, when substituting line 9 in Listing 16 above with '-> echo("myArgument")', this will be encoded as '<c-gensym3 xsi:type="xsd:string">myArgument</c-gensym3>'; these additional elements called "c-gensym3" are of course not always wanted. Besides of that, overriding the namespace of XML elements is also not intuitive[25].

- focus on RPC/encoded: By default, SOAP::Lite uses the deprecated SOAP section 5 encoding (as can be seen in Listing 14: the envelope contains the soap:encodingStyle attribute). More importantly, formatting a SOAP call to contain more than one child elements of the "body" is very cumbersome to implement; since by far the most SOAP services use the doc/literal message format, this is a serious drawback of SOAP::Lite (which is also stated by its current maintainer in [15]).

2. Is code generation from a WSDL file supported?

SOAP::Lite comes with a command-line tool to generate code from a WSDL file, called "stubmaker.pl". However, this code generation is only rudimentary: When invoked with a WSDL file, stubmaker generates a Perl package that automatically sets the correct HTTP POST address, the SOAPAction

[25] for how this is done, consult the SOAP::Lite documentation at http://search.cpan.org/~byrne/SOAP-Lite/lib/SOAP/Lite.pm

HTTP header field and the namespace of the XML elements inside the SOAP body. Additionally, it generates the name of the method to call, so that an operation specified in the WSDL can be invoked with that name: For instance, the MSN Search WSDL[26] contains a service element (<wsdl:service name=”MSNSearchService”>) and an operation element (<wsdl:operation

name=”Search”>), which stubmaker.pl uses to generate a package “MSNSearchService” containing a method called “Search”.

However, what stubmaker.pl does not generate is a Perl access layer of the SOAP message structure. For instance, the method “Search” described above requires an XML data structure that is nested in four layers; with stubmaker.pl, this XML data would need to be built up by hand with SOAP::Data elements. Of course, for large XML data this is unfeasible.

So SOAP::Lite does provide code generation from a WSDL file and provides a Perl layer that abstracts from some items of a SOAP call (operation name, SOAPAction header field etc.), but it does not abstract from the XML Schema part of a WSDL file. Since many SOAP calls require a lot and complex XML data, it is not feasible to build up XML data by hand with SOAP::Lite and thus not feasible to use SOAP::Lite at all for code generation from a WSDL file.

3. Does the tool comply to Web service standards and does it provide support for additional WS- standards?*

As already said, SOAP::Lite complies to both SOAP 1.1 and 1.2 and by default sends rpc/encoded SOAP messages. The latter does not comply to the WS-Interoperability standard ([5]), as it states (in section R1005):

> An ENVELOPE MUST NOT contain soap:encodingStyle attributes on any of the elements whose namespace name is "http://schemas.xmlsoap.org/soap/envelope/".

This default usage of rpc/encoded calls can be overriden to use literal calls, but would be quite tedious, since for every XML element used in the call the type would have to be overridden.

Besides of that, SOAP::Lite does not provide support for additional WS-* standards like WS-encryption or WS-Addressing. However, it supports binary attachments via multipart/MIME messages (often called SOAP with Attachments, SwA); additionally, it supports DIME, which is an advanced standard for binary attachments in SOAP[27]; however, this standard was declared obsolete and the usage of MTOM encouraged, which is not supported by SOAP::Lite.

As a summary, SOAP::Lite seems quite outdated nowadays with its focus on rpc/encoded messages, no support for WS-* standards and its poor support of WSDL code generation.

[26] http://soap.search.msn.com/webservices.asmx?wsdl

[27] http://xml.coverpages.org/draft-nielsen-dime-00.txt

4.3.2 Apache Axis2 (Java)

Apache Axis2[28] is a huge comprehensive open source Web Service framework. It supports both SOAP standards 1.1 and 1.2, and its code generator supports both common WSDL verisons 1.1 and 2.0. According to [14], Axis2 offers the following protocols for transportation of SOAP messages: HTTP, HTTPS, TCP, SMTP, JMS and XMPP. Apart from that, it supports attachments via both SOAP with Attachments (SwA) and the more advanced MTOM. Axis2 has also implementations for many WS-* standards (among them the important WS-Security and WS-Addressing), which are explained below.

The Axis2 framework even offers REST-style interfaces for its services; however, there are discussions[29] whether this architecture is really RESTful or just offering a URI-style service access method (for instance, HTTP GET methods can in Axis2 change data resources).

Axis2 has as its two main goals flexibility and extensibility. These goals are achieved by structuring Axis2 in a modular architecture, as shown in Figure 15 (adapted from [14]).

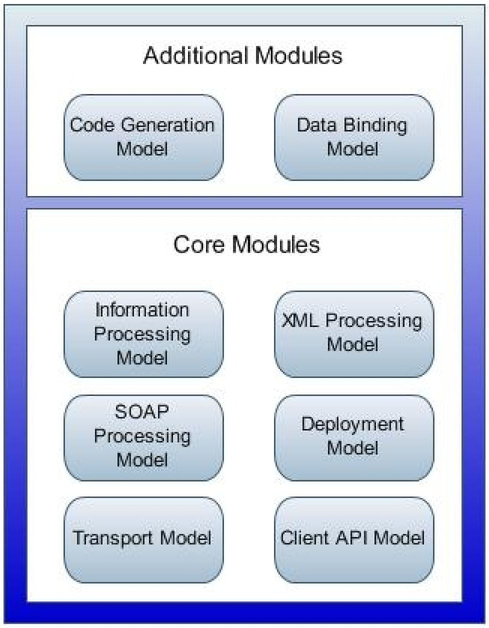


Figure 15: The modular Axis2 architecture

The Axis2 modules have the following functions:

[28] http://ws.apache.org/axis2/index.html

[29] e.g. http://atmanes.blogspot.com/2007/06/how-not-to-do-restful-web-services.html

- *Code Generation Model*: This module is responsible for generating code from a WSDL (on the client and sometimes also on the server side) or generating a WSDL from code (on the server side). When creating a Web service, the former approach (generating server-side code from a WSDL) is called contract-first approach, since the WSDL servers as a contract between the service provider and the service consumers. The other approach is to generate a WSDL from a given piece of code, and is called code-first approach. As these techniques are only relevant on the server side, they are not discussed here. On the client side, the only relevant approach is to generate code from a WSDL.

- *Data Binding Model*: This module is responsible for providing a programming-language access level to SOAP messages, especially to the XML inside the SOAP header and body. It has not been included in the core to leave the flexibility to provide other data bindings in the future. Axis2 comes with four different data bindings; they differ in various items like for instance usage complexity, coverage of supported XML Schema constructs and API offered to the user.

- *Information Processing Model*: This module is responsible for keeping the configuration context of a message. It is split up into a Description Hierarchy, which keeps static data (coming from e.g. system configurations) and a Context Hierarchy, which keeps dynamic data. Both hierarchies provide different layers of configuration contexts (that is: key-value pairs), where the lower layer override the values from the above layer. The layers for both hierarchies are, from down to up: Message layer, Operation layer, Service layer, ServiceGroup layer, Configuration layer.

- *XML Processing Model*: This module is responsible for representing an XML message in Axis2. That representation is called AXIOM (*Axi*s2 *O*bject *M*odel); AXIOM uses a pull-parsing approach (as described earlier) for efficient usage; these details are encapsulated in the implementation and hidden from the user.

- *SOAP Processing Model*: This module is responsible for handling an incoming or outgoing SOAP message. Incoming and outgoing messages walk through different phases and can be changed by different handlers. Such a handler could be responsible for signing or encrypting a message, just logging the arrival time of an incoming message or adding routing headers to the message. Own handlers can also be written and registered, which adds to the extensibility of Axis2.

- *Deployment Model*: This module is responsible for deploying service applications. It provides means for hot updating applications (i.e. without restarting the server) as well as packaging all needed files into one archive file. As it is only needed on the server side, it is not of interest here.

- *Transport Model*: This module is responsible for handling the underlying transport protocols of an incoming or outgoing SOAP message. As already written earlier, it supports the following protocols: HTTP, HTTPS, TCP, SMTP, JMS and XMPP.
- *Client API Model*: This module is responsible for sending and receiving messages; it is described below.

1. What API does the toolkit provide?

Axis2 provides an API that lets the user control every aspect of a SOAP communication (setting/getting message content, controlling the transport protocol, using attachments etc.). In general, there are two sub-APIs that differ in complexity and feature richness:

- *ServiceClient API*: The ServiceClient Class is destined for easy access of basic Web Service features. It allows only to get and set the SOAP body, and to add content to the SOAP header, but it does for instance not allow to retrieve header fields. When sending simple messages without usage of WS-* protocols, the ServiceClient is the right choice.
- *OperationClient API*: The OperationClient Class offers full control over the SOAP message and is a superset of the ServiceClient API. In addition to the features of the ServiceClient, it offers for instance control over the whole SOAP envelope, getting / setting the message exchange pattern (MEP) and getting / setting the SOAP headers.

ServiceClient API

As a usage example of that API, consider again the sample from Listing 14. The Axis2 sequence diagram to issue a client call can be seen in figure 16; the corresponding code to send the message in Listing 14 can be seen in Listing 17.

Listing 17: A simple Axis2 client

```
import [...]; // import statements left out

public class AXIOMClient {

 public static void main(String[] args) {
   try {
     // construct XML
     OMFactory fac = OMAbstractFactory.getOMFactory();
     OMNamespace omNs = fac.createOMNamespace("http://
         localhost/Demo", "tns");
     OMElement payload = fac.createOMElement("echo", omNs)
         ;
     OMElement value = fac.createOMElement("myArgument",
         omNs);
```

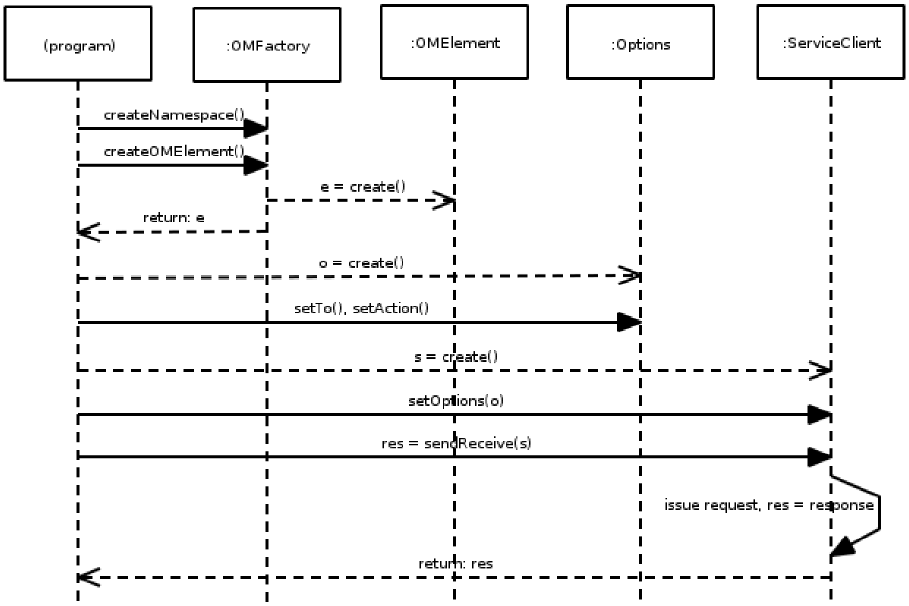


Figure 16: Axis2 invocation flow

```
        value.addChild(fac.createOMText(value, "myValue"));
        payload.addChild(value);

        // handle HTTP request
        Options options = new Options();
        options.setTo(new EndpointReference("http://localhost/Demo"));
        options.setAction("mySOAPActionHeader");
        ServiceClient sender = new ServiceClient();
        sender.setOptions(options);

        // send SOAP message
        OMElement result = sender.sendReceive(payload);

        // parse response
        String response = result.getFirstElement().getText();
        System.err.println("response:␣" + response);
      } catch (Exception e) {
        e.printStackTrace();
    }
  }
}
```

The example above needs a lot more code than the SOAP::Lite one; one reason might be the more compact syntax of Perl compared to Java, but for instance generating the XML requires also more lines of code with Axis2 than with SOAP::Lite. The XML payload generation is done in the lines 8 - 13, and generates an “echo” element with a “myArgument” element as child. Lines 16 - 20 show how the HTTP values are set (endpoint URL and SOAPAction header); line 23 sends the message in a *blocking* manner. Alternatives to a blocking call are discussed below. In the lines 26 and 27, the program parses the response and prints it on the standard output.

The code above does not generate the same code as in Listing 14, there are two differences: First, the SOAP encoding attribute of the envelope is not generated, as Axis2 concentrates on doc/literal SOAP calls. Second, it uses an XML prefix for the children of the SOAP body, which allows for the usage of elements of the same name in different namespaces. Listing 18 shows the SOAP request generated by Axis2.

Listing 18: SOAP request generated by Axis2

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.
    org/soap/envelope/">
    <soapenv:Body>
        <tns:echo xmlns:tns="http://localhost/Demo">
            <tns:myArgument>myValue</tns:myArgument>
        </tns:echo>
    </soapenv:Body>
</soapenv:Envelope>
```

The code in Listing 17 invokes the HTTP call in a blocking manner. That is, the program issues the HTTP request and blocks until the HTTP response is received. This might cause the program to stall, for instance when the Web server of the service provider is down. Hence, in many cases a non-blocking call is the better alternative; however, this is more difficult to program. Axis2 supports non-blocking calls by using callbacks: When sending the message (via the method “sendReceiveNonBlocking”), it just sends out the HTTP request and returns immediately. But when invoking that method, a callback method must be given to handle the HTTP response. When the program receives the response, it calls the callback method that was registered earlier. Thus, while waiting for the response, the program can do other tasks and does not stall. [14] describes how to do asynchronous calls.

The synchronous and asynchronous calls both support sending and immediate receiving of messages (following the Message Exchange Pattern “in-out”). There are also methods for only sending a message (MEP “in-only”):

- fireAndForget: This method sends the SOAP request and continues, no matter whether the call succeeded or not.

- sendRobust: This method also just sends the SOAP request and continues; however, when there is an error, it throws an exception.

OperationClient API
As already sais, that API offers more functionality than the ServiceClient API, but also requires more work. For instance, when creating a SOAP message by hand with the OperationClient API, one has to create a SOAP envelope and add headers and body to it (for how this is done, see [14]).

However, when using enterprise-level web services, the most common way is to generate code from a WSDL file and not build up the SOAP calls by hand. Therefor, the OperationClient API is not described here, but more focus is set on code generation from a WSDL file.

2. Is code generation from a WSDL file supported?

Axis2 comes with extensive support for WSDL code generation: The Axis2 package contains a tool to generate WSDL files from code (java2wsdl.sh for Unix and java2wsdl.bat for Windows) as well as a tool to generate code from WSDL files (wsdl2java.sh and wsdl2java.bat). The latter tool is used to generate client-side code stubs as well as server-side skeletons (e.g. when a contract-first approach was used). This section here only describes the client-side code generation from a WSDL file.

The wsdl2java.sh tool is to be used from the command line; however, there are also plugins for use in Eclipse or a script for use in Apache Ant[30]. But as all those tools use the command-line tool in the end, the plugins are not described here.

There are a lot of command line parameters for usage with wsdl2java which show the feature-richness of Axis2; the most important among them are:

- options for generating synchronous and/or asynchronous code
- options for generating test cases
- options for choosing a data binding (see below)
- options for configuring code internals (e.g. custom package names or choosing specific ports or services from the WSDL file)

The only mandatory argument the code generation tool takes is the URL or path of the WSDL file. When invoked with that file and no special configuration values (e.g. with the MSN Search WSDL on Unix: "wsdl2java.sh -uri http://soap.search.msn.com/webservices.asmx?wsdl"), it generates the following files:

- Ant build script (build.xml): An Ant script to compile and run the client code.
- Stub code: A file that contains stub classes for usage; it encapsulates all needed HTTP information (POST URL and SOAPAction header field)

[30] http://ant.apache.org/

and provides Java access for items specified in the WSDL file (e.g. operations and XML Schema information).

- Callback handler: By default, Axis2 creates code for synchronous and asynchronous invocation (this can be changed via command line parameters). The callback handler serves for asynchronous invocation and provides special callbacks for the operations defined in the WSDL.

An important aspect of the code generation is the data binding: It controls the programming language layer of the SOAP message that encapsulates the low-level XML handling. Axis2 comes with four different data bindings:

- ADB (Axis Data Binding): This is the default data binding; it is meant to be simple and easy to start with. As a drawback, ADB does not support all of the XML Schema standard (for instance, complex and simple extensions and restrictions are not supported), but in many cases ADB is enough to cover the whole XML Schema part of a WSDL file. In that case, it is the easiest data binding to use.
- XML Beans: This is a more comprehensive data binding; it claims to support the whole XML Schema standard. It is destined for complex XML Schema documents, and has as a drawback a more complex interface. For instance, the code generated for the MSN Search WSDL contains 258 Java files and 470 other files, whereas the code generated for the same WSDL but with ADB contains 2 Java files.
- JAXB-RI: This is a highly customizable data binding, which also supports the whole XML Schema standard. However, according to [20], it shows worse performance than ADB.
- JibX: This data binding is useful for binding own classes to a Web service, for instance when binding a legacy application to a Web service.

As an example, Listing 19 shows the XML Schema part that corresponds to the SOAP messages in the Listings 14 and 15; Listing 20 shows how to send the SOAP message from Listing 14 with the ADB data binding, and the Figures 17 and 18 show the UML representation of the two most important auto-generated classes.

Listing 19: part of a WSDL file

```
<xs:schema xmlns:ns="http://localhost/Demo" attributeFormDefault="qualified" elementFormDefault="qualified" targetNamespace="http://localhost/Demo">
  <xs:element name="echo">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="myArgument" type="xs:string"/>
      </xs:sequence>
```

```
 7          </xs:complexType>
 8      </xs:element>
 9      <xs:element name="echoResponse" type="xs:string"/>
10  </xs:schema>
11  ...
12  <wsdl:portType name="MyServicePortType">
13      <wsdl:operation name="echo">...</wsdl:operation>
14  </wsdl:portType>
```

Listing 20: Using Axis2 with ADB

```
 1  package localhost.demo;
 2
 3  import localhost.demo.MyServiceStub.Echo;
 4  import localhost.demo.MyServiceStub.EchoResponse;
 5
 6  public class MyService {
 7      public static void main(java.lang.String args[]){
 8          try {
 9              MyServiceStub stub = new MyServiceStub();
10              Echo e = new Echo();
11              e.setMyArgument("myValue");
12              EchoResponse er = stub.echo(e);
13              System.out.println("return: ");
14              System.out.println(er.getEchoResponse());
15
16          } catch(Exception e){
17              e.printStackTrace();
18          }
19      }
20  }
```

In the listing above, line 9 creates the stub that was automatically generated from the WSDL. As already explained, that stub encapsulates as much information as possible and contains for each WSDL operation a member method: For instance, as the WSDL in Listing 19 contains an operation called “echo” in line 13, the stub class contains a member method named “echo” (line 12 in Listing 20). Line 11 sets the SOAP body, namely sets the content of the “<myArgument>” element to “myValue”. This shows how the XML is completely encapsulated from the user by offering an interface written in Java. The generated code class takes care of the right namespaces and element names. Line 12 sends out a blocking call and line 14 again extracts the needed information from the SOAP response message.

This chapter has explained the thorough code generation support that Axis2 offers. It is one of the few toolkits that provide full XML Schema support, moreover, it comes with pluggable data binding support and additional features like

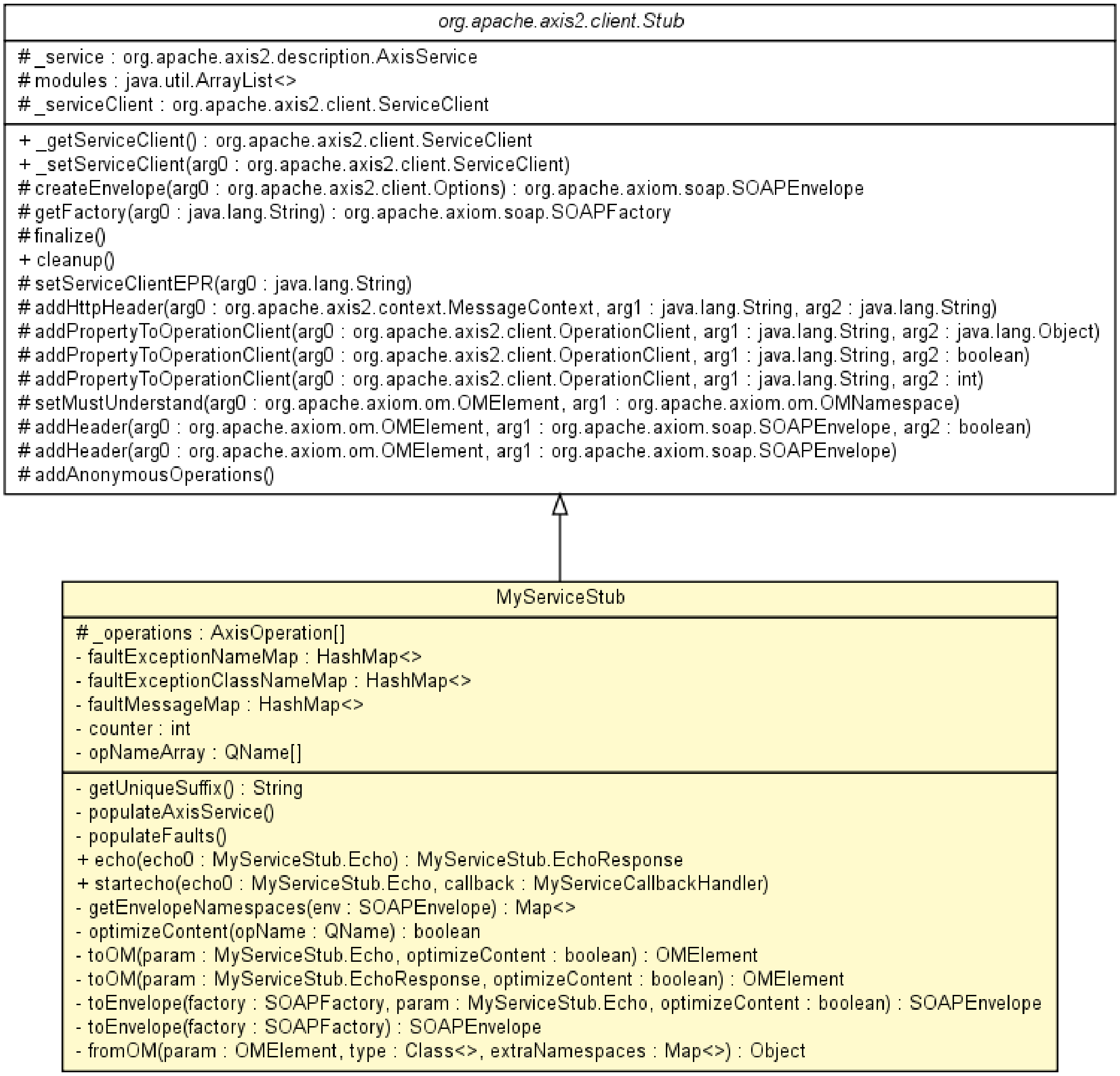

org.apache.axis2.client.Stub
# _service : org.apache.axis2.description.AxisService
# modules : java.util.ArrayList<>
# _serviceClient : org.apache.axis2.client.ServiceClient
+ _getServiceClient() : org.apache.axis2.client.ServiceClient
+ _setServiceClient(arg0 : org.apache.axis2.client.ServiceClient)
# createEnvelope(arg0 : org.apache.axis2.client.Options) : org.apache.axiom.soap.SOAPEnvelope
# getFactory(arg0 : java.lang.String) : org.apache.axiom.soap.SOAPFactory
# finalize()
+ cleanup()
# setServiceClientEPR(arg0 : java.lang.String)
# addHttpHeader(arg0 : org.apache.axis2.context.MessageContext, arg1 : java.lang.String, arg2 : java.lang.String)
# addPropertyToOperationClient(arg0 : org.apache.axis2.client.OperationClient, arg1 : java.lang.String, arg2 : java.lang.Object)
# addPropertyToOperationClient(arg0 : org.apache.axis2.client.OperationClient, arg1 : java.lang.String, arg2 : boolean)
# addPropertyToOperationClient(arg0 : org.apache.axis2.client.OperationClient, arg1 : java.lang.String, arg2 : int)
# setMustUnderstand(arg0 : org.apache.axiom.om.OMElement, arg1 : org.apache.axiom.om.OMNamespace)
# addHeader(arg0 : org.apache.axiom.om.OMElement, arg1 : org.apache.axiom.soap.SOAPEnvelope, arg2 : boolean)
# addHeader(arg0 : org.apache.axiom.om.OMElement, arg1 : org.apache.axiom.soap.SOAPEnvelope)
# addAnonymousOperations()
MyServiceStub
# _operations : AxisOperation[]
- faultExceptionNameMap : HashMap<>
- faultExceptionClassNameMap : HashMap<>
- faultMessageMap : HashMap<>
- counter : int
- opNameArray : QName[]
- getUniqueSuffix() : String
- populateAxisService()
- populateFaults()
+ echo(echo0 : MyServiceStub.Echo) : MyServiceStub.EchoResponse
+ startecho(echo0 : MyServiceStub.Echo, callback : MyServiceCallbackHandler)
- getEnvelopeNamespaces(env : SOAPEnvelope) : Map<>
- optimizeContent(opName : QName) : boolean
- toOM(param : MyServiceStub.Echo, optimizeContent : boolean) : OMElement
- toOM(param : MyServiceStub.EchoResponse, optimizeContent : boolean) : OMElement
- toEnvelope(factory : SOAPFactory, param : MyServiceStub.Echo, optimizeContent : boolean) : SOAPEnvelope
- toEnvelope(factory : SOAPFactory) : SOAPEnvelope
- fromOM(param : OMElement, type : Class<>, extraNamespaces : Map<>) : Object

Figure 17: Axis2 auto-generated class “MyServiceStub”

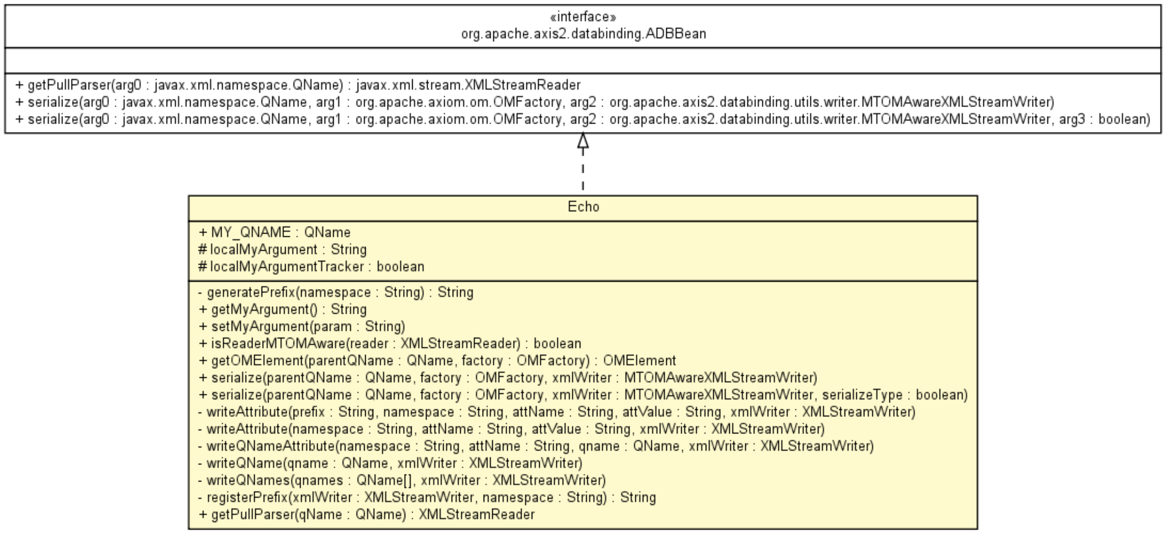

«interface»
org.apache.axis2.databinding.ADBBean
+ getPullParser(arg0 : javax.xml.namespace.QName) : javax.xml.stream.XMLStreamReader
+ serialize(arg0 : javax.xml.namespace.QName, arg1 : org.apache.axiom.om.OMFactory, arg2 : org.apache.axis2.databinding.utils.writer.MTOMAwareXMLStreamWriter)
+ serialize(arg0 : javax.xml.namespace.QName, arg1 : org.apache.axiom.om.OMFactory, arg2 : org.apache.axis2.databinding.utils.writer.MTOMAwareXMLStreamWriter, arg3 : boolean)
Echo
+ MY_QNAME : QName
# localMyArgument : String
# localMyArgumentTracker : boolean
- generatePrefix(namespace : String) : String
+ getMyArgument() : String
+ setMyArgument(param : String)
+ isReaderMTOMAware(reader : XMLStreamReader) : boolean
+ getOMElement(parentQName : QName, factory : OMFactory) : OMElement
+ serialize(parentQName : QName, factory : OMFactory, xmlWriter : MTOMAwareXMLStreamWriter)
+ serialize(parentQName : QName, factory : OMFactory, xmlWriter : MTOMAwareXMLStreamWriter, serializeType : boolean)
- writeAttribute(prefix : String, namespace : String, attName : String, attValue : String, xmlWriter : XMLStreamWriter)
- writeAttribute(namespace : String, attName : String, attValue : String, xmlWriter : XMLStreamWriter)
- writeQNameAttribute(namespace : String, attName : String, qname : QName, xmlWriter : XMLStreamWriter)
- writeQName(qname : QName, xmlWriter : XMLStreamWriter)
- writeQNames(qnames : QName[], xmlWriter : XMLStreamWriter)
- registerPrefix(xmlWriter : XMLStreamWriter, namespace : String) : String
+ getPullParser(qName : QName) : XMLStreamReader

Figure 18: Axis2 auto-generated class "Echo"

e.g. creating automatic test cases.

3. Does the tool comply to Web service standards and does it provide support for additional WS- standards?*

Axis2 supports several WS-* standards. Only one of them is part of the Axis2 distribution, while all other modules are shipped as independent projects. The supported standards are:

- WS-Addressing: contained in the distribution, and serves to address messages and identify Web Services (e.g. to be independent of the HTTP SOAPAction field).
- WS-Security: part of Apache Rampart[31], serves to sign and encrypt SOAP messages.
- WS-Trust: part of Apache Rampart, extends WS-Security and servers to handle trust relationship between Web service nodes (e.g. issuing and validating security tokens or establish such a trust relationship).
- WS-SecureConversation: part of Apache Rampart, built on top of WS-Security and WS-Trust and serves to support secure communication between Web service nodes.
- WS-SecurityPolicy: part of Apache Rampart, serves to define security requirements and assertions of Web service nodes.

[31] http://ws.apache.org/rampart/

- WS-ReliableMessaging: part of Apache Sandesha2[32], serves to send messages between SOAP nodes in a reliable way.
- WS-Coordination: part of Apache Kandula2[33], serves to coordinate behavior of several Web service nodes.
- WS-AtomicTransaction: part of Apache Kandula2, serves to atomically take several short-term actions (in analogy to the ACID principle) via Web services in an "all or nothing" manner.
- WS-BusinessActivity: part of Apache Kandula2, built on top of WS-Coordination, WS-Addressing and WS-Policy and servers to coordinate long-term business transactions.
- WS-Policy: part of Apache Neethi[34], servers to advertise policies of a Web service (e.g. security features or quality of service).
- WS-MetadataExchange: part of Axis2/metadataExchange[35], servers to retrive metadata of a Web service (e.g. a WSDL or XML Schema file).

Moreover, Axis2 supports attachments via several techniques:

- base64 encoding: this might be an option when only few binary data is sent and using one of the techniques below would be overkill.
- SOAP with attachments (SwA): Axis2 automatically detects SOAP messages that contain SwA-style attachments via its content type. Those attachments can be retrieved by accessing the Message Context as described earlier.
- MTOM (Message Transmission Optimization Mechanism): Axis2 also automatically detects MTOM messages without the user needing to take additional actions.

All in all, Axis2 is a very comprehensive Web service framework, if not the most comprehensive at all. With its huge feature support and its flexible architecture it is suited to build enterprise applications.

4.3.3 gSOAP (C++)

gSOAP[36] is a Web service framework written in C++. It supports both SOAP versions 1.1 and 1.2 and WSDL version 1.1; WSDL version 2.0 is not supported as of this writing.

[32] http://ws.apache.org/sandesha/sandesha2/index.html

[33] http://ws.apache.org/kandula/2/index.html

[34] http://ws.apache.org/commons/neethi/index.html

[35] http://wiki.apache.org/ws/Axis2/metadataExchange

[36] http://www.cs.fsu.edu/~engelen/soap.html

Furthermore, it supports several WS-* standards and attachments as well as different transports (HTTP, HTTPS, TCP, UDP) and, on the server side, integration into the Apache Web Server, the Microsoft Internet Information Server and a standalone server version.

1. What API does the toolkit provide?

gSOAP sets its primary focus on code generation, neither the main document about gSOAP ([26]) nor the user documentation[37] contain examples about building up a SOAP query by hand. But as code generation from WSDL is the common use case, this is not a big disadvantage of gSOAP.

2. Is code generation from a WSDL file supported?

gSOAP comes with extensive code generation support. Like Axis2, gSOAP contains a command-line tool ("wsdl2h") to generate code from a WSDL file. The command-line tool supports several arguments, the most important among them being options for generating C source code instead of C++ and options for providing own type mappings to map XML Schema types to C++ types. As a difference to the Axis2 code generator, the gSOAP code generator consists of a two-phase process: First, the wsdl2h command-line tool generates a C/C++ header file containing for each operation in the WSDL file a method signature in C++. For instance, for the operation "echo" in Listing 19, the wsdl2h tool generates the (quite cryptic) signature shown in Listing 21.

Listing 21: header file created by gSOAP's wsdl2h

```
int __ns4__echo(
    _ns2__echo*     ns2__echo,  ///< Request parameter
    std::string    &ns2__echoResponse    ///< Response
        parameter
);
```

The second-phase tool called "soapcpp2", takes as input that header file and generates the following files (via the command "soapcpp2 myservice.h"):

- several files containing sample requests and responses as well as namespace mappings (MyServiceSOAP11Binding.* and MyServiceSOAP12Binding.*)
- XML serializers for data types (soapH.h and soapC.cpp)
- client side stub (soapClient.cpp, soapClientLib.cpp, soapMyServiceSOAP11BindingProxy.h and soapMyServiceSOAP12BindingProxy.h)
- server side skeleton (soapServer.cpp, soapServerLib.cpp, soapMyServiceSOAP11BindingObject.h and soapMyServiceSOAP12BindingObject.h)

[37] http://www.cs.fsu.edu/~engelen/soapdoc2.html

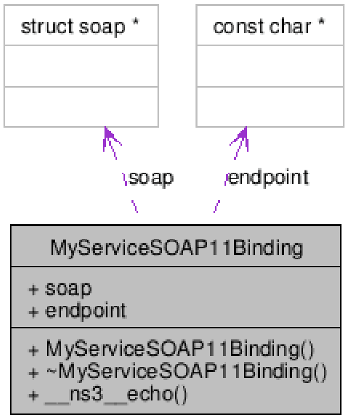


Figure 19: gSOAP auto-generated class “MyServiceSOAP11Binding”

The most important file needed when writing code is the binding proxy (soapMyServiceSOAP11BindingProxy.h), which contains all the necessary information. Listing 22 shows how the generated code can be used to invoke the service (by sending the message from Listing 18). The corresponding collaboration diagrams can be seen in the Figures 19 and 20 .

Listing 22: gSOAP client code

```
#include <iostream>
#include "soapMyServiceSOAP11BindingProxy.h"
#include "MyServiceSOAP11Binding.nsmap"

int main() {
    MyServiceSOAP11Binding binding;
    _ns2__echo *echo = new _ns2__echo();
    echo->myArgument = new std::string("myValue");
    std::string echoResponse;
    if(binding.__ns3__echo(echo, echoResponse) == SOAP_OK
        ) {
        std::cout << "result: " << echoResponse << "\n";
    }
    else {
        soap_print_fault(binding.soap, stderr);
    }
}
```

The code above is compact and self-explanatory. The only drawback here is the cumbersome usage of namespace prefixes in class names (e.g. class “_ns2__echo”). However, this is required to support classes with the same local name resident in different namespaces.

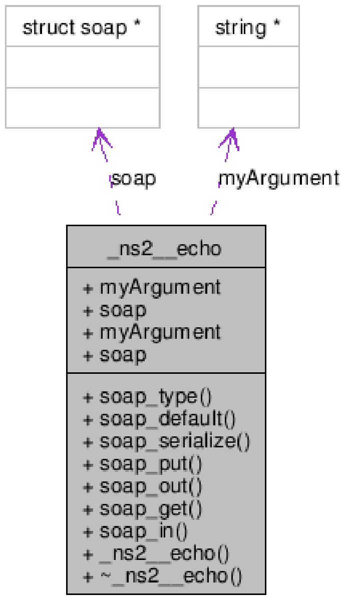


Figure 20: gSOAP auto-generated class "_ns2__echo"

3. Does the tool comply to Web service standards and does it provide support for additional WS- standards?*

As already said, gSOAP supports both SOAP versions 1.1 and 1.2 and WSDL version 1.1. Additionally, it is compliant to the WS-Interoperability Basic Profile 1.0a[38].

gSOAP supports the following WS-* standards:

- WS-Addressing
- WS-Discovery
- WS-Enumeration
- WS-Security

On the attachment side, it supports SOAP with Attachments (SwA) as well as MTOM-style attachments. Moreover, the DIME stadard (which stands for Direct Internet Message Encapsulation) is also supported, but this has been declared obsolete and superseded by MTOM anyway.

As a summary, gSOAP contains all the features needed to build complex Web services (i.e. the most common WS-* standards and attachments). However,

[38] see the fact sheet at http://www.cs.fsu.edu/~engelen/factsheet.pdf

some of its features make it somehow cumbersome to use (e.g. the two-phase construction or the usage of namespace prefixes in class and method names).

4.3.4 .NET (C#)

As Apache Axis2, .Net[39] is a powerful web service framework, including a code generation tool and supporting all SOAP and WSDL versions. It supports several WS-* standards and different kinds of attachments.

1. *What API does the toolkit provide?*

As is the case with gSOAP, .NET does not provide an easy way for building up a SOAP call by hand. Of course there is a way to do that, but this is neither documented nor the main use case of that SOAP toolkit. As with all up to date SOAP toolkits, its main focus is set on code generation.

2. *Is code generation from a WSDL file supported?*

The .NET framework comes with a command-line tool (wsdl.exe) to build code from a WSDL file, similar to the one provided by Axis2. It offers command-line arguments to control the programming language the code should be generated for (C#, Visual Basic or JScript). However, wsdl.exe always generates bindings to issue both synchronous and asynchronous calls. Calling a Web service in a synchronous way requires only a few lines of code and is very easy; how to call the service described in Listing 19 (issueing the call from Listing 14) is shown in Listing 23, with the corresponding collaboration diagram in Figure 21.

Listing 23: .NET client code

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            MyService service = new MyService();
            string result = service.echo("myValue");
            Console.WriteLine(result);
        }
```

[39] http://msdn.microsoft.com/netframework/

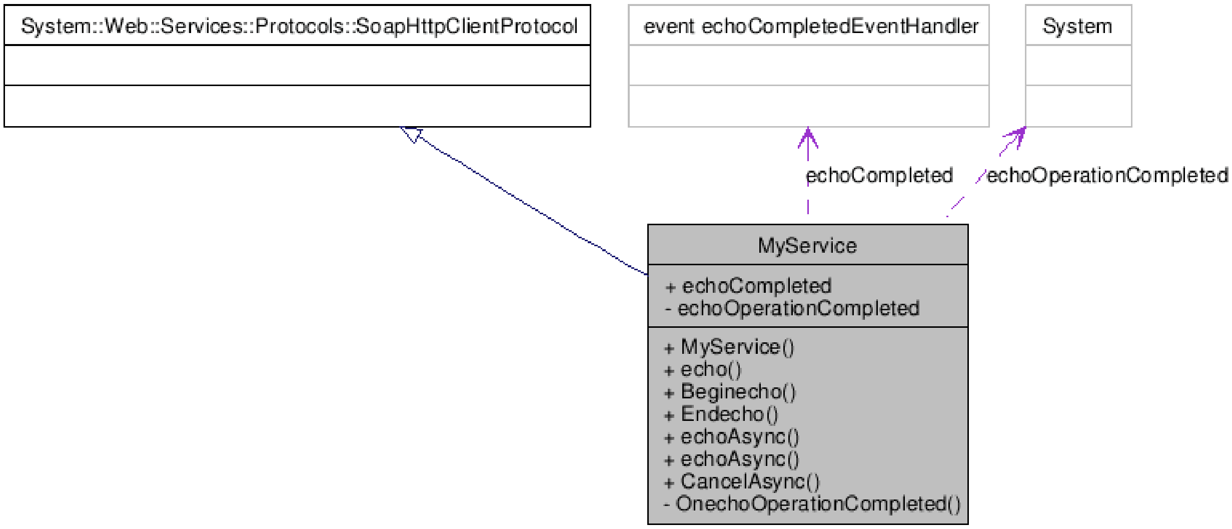


Figure 21: .NET auto-generated class "MyService"

```
    }
}
```

Making asynchronous calls can be done in different ways, as [17] describes:

The common way to make asynchronous calls with .NET generated code is to use callbacks, as already described earlier. Listing 24 shows the part of the generated code containing the synchronous method "echo" as well as the asynchronous method "Beginecho" which takes a callback method as argument, and, after being called, returns immediately. Finally, the "Endecho" call can be called by the callback method when the response has been retrieved.

Listing 24: .NET generated code

```
    public string echo(string myArgument) {
        object[] results = this.Invoke("echo", new object
            [] {
                    myArgument});
        return ((string)(results[0]));
    }

    /// <remarks/>
    public System.IAsyncResult Beginecho(string
       myArgument, System.AsyncCallback callback, object
       asyncState) {
        return this.BeginInvoke("echo", new object[] {
                    myArgument}, callback, asyncState);
    }
    /// <remarks/>
    public string Endecho(System.IAsyncResult asyncResult
        ) {
        object[] results = this.EndInvoke(asyncResult);
        return ((string)(results[0]));
    }
```

Another way of issuing an asynchronous call is to use the WaitHandle technique: Sometimes, when using the Callback technique, it might be undesired that the callback method is called exactly when the Web service is finished; sometimes, it is more convenient to let the user control when the Web service response is to be parsed. This is what the WaitHandle technique does: It calls the same asynchronous method as the Callback technique, but without registering a callback method. Instead, after finishing some other work, after issuing the call and probably doing some other works, at some point it polls for the request to finish. This technique can be seen as a hybrid between the synchronous (polling) technique and the Callback technique. As an advantage, it lets the programmer control when the response is parsed; as a disadvantage, polling for the result to early might lead to wasted time as with synchronous calls.

Those three methods (synchronous calls, Callbacks and WaitHandles) are supported by .NET, as [17] describes.

3. Does the tool comply to Web service standards and does it provide support for additional WS- standards?*

.NET supports a lot of WS-* standards[40] - some of them are part of the Windows Communication Foundation, which is built on top of .NET. The most important of them are:

- WS-ReliableMessaging
- WS-Addressing
- WS-Security
- WS-SecureConversation
- WS-AtomicTransaction
- WS-Discovery

Moreover, it supports the deprecated DIME and MTOM for attachments; a very interesting feature is the support for binary XML, which is not supported in other toolkits as of this writing.

[40] for a complete list, see http://msdn.microsoft.com/en-us/library/aa480728.aspx#wsmsplat_topic32

All in all, the .NET framework comes with excellent support for code generation and WS-* standards and is suited to build enterprise applications. Its internals are not documented as well as e.g. Axis2, but its API is both easy to start with and powerful.

4.4 SOAP toolkit evaluation

Now that different existing SOAP toolkits have been evaluated, this chapter will summarize the requirements for the SOAP toolkit described in section 5. That framework needs to have the following features, which are either results of the SOAP protocol evaluation, the toolkit evaluation and/or the Qt evaluation. Some of the features are indispensable, while others would be nice to have, but are not mandatory.

1. *WSDL code generation.* This is clearly an important part of any SOAP toolkit and the primary use case for building applications using SOAP. The SOAP toolkit proposed in chapter 5 should thus contain a command-line tool that takes as input a URL or a path and produces C++ code stubs that use the Qt library. That code stubs should ideally both offer an easy API and at the same time should be powerful to support most common WSDL and XML Schema constructs.

2. *API to build up SOAP queries by hand.* This is not a primary requirement; the main SOAP use case is to automatically generate code. However, building up queries by hand is useful for testing and debugging; moreover, some legacy Web services might not offer a WSDL interface and building up the query by hand is necessary. In fact, theoretically every toolkit of course supports building up the query by hand, but in practice this can only be done if the API is fairly easy to use and the process of building up the query is well documented.

3. *asynchronous calls.* As explained earlier, asynchronous calls can improve the performance of Web service applications. But since the Qt Network Access API only offers issuing asynchronous calls anyway, this will be the only way to way to issue SOAP calls, which might complicate the API.

4. *pluggable data binding.* This is not an important requirement; it is a feature that is supported by both Axis2 and gSOAP: gSOAP offers different mappings from XML Schema types to programming language data types. The data bindings that Axis2 offers all have their own architecture, which makes the toolkit very flexible. However, for the Qt framework it should always be clear what XML Schema type to map to which Qt data type, so in the first version of the Qt SOAP toolkit it will be acceptable to hard-code the data types.

5. *separate XML Schema parsing entity from WSDL parsing entity.* The WSDL standard is tied closely to the XML Schema standard; that is,

virtually every WSDL file contains an XML Schema part or references an XML Schema file. However, XML Schema is used in different areas of application, with Web services being one of many. So for the SOAP toolkit it would be desirable to separate the XML Schema part from the WSDL part and pack the XML Schema part into a component that is reusable in other contexts.

6. *possibility to add WS-* support and attachments.* The SOAP framework described later will not support WS-* standards or attachments, but it should be designed to support their usage in the future. Different phases and handlers as used by Axis2 might be overkill here, but there should be a possibility to somehow plug in classes supporting WS-* standards.

7. *XML pull parsing.* Again, this is a desirable but not mandatory feature, as it does not provide new functionality; however, it would be nice to have (if not in the first version, then at a later point of time) to improve performance.

Many of the features described above are already on an advanced level (e.g. pull parsing or pluggable data binding); the existing SOAP stacks described above did not contain them when they were released for the first time. Thus the Qt SOAP framework need not contain all of them, but rather leave room for supporting them in the future.

5 SOAP for the Qt framework

This chapter shall propose an architecture for the QtSoap module. The framework consists of two main parts: First, the C++ classes for sending and receiving SOAP messages; second, the WSDL parser for generating code. As already described partially in chapter 3, the C++ classes rely on the following Qt classes:

- *QtCore classes*: classes like QObject for using the signals-and-slots mechanism and the XML streaming classes (QXmlStreamWriter and QXmlStreamReader) reside in QtCore; moreover, basic classes like QString and QLinkedList are also part of QtCore.
- *QtNetwork classes*: the Network access API is used for HTTP transportation, so the QtNetwork module needs to be referenced.

5.1 SOAP module architecture

The C++ classes for the SOAP module can be basically divided into three fields:

- XML data types for representing SOAP message elements (like arrays or nested structures)
- SOAP message representation and transports for sending and receiving
- XML parsing for generating SOAP message elements from XML and back

Figure 22 shows the three components and their dependencies.

The classes described below use the most common SOAP version 1.1, and are designed to support XML Schema types. SOAP section 5 types were not considered while designing the classes.

5.1.1 SOAP / XML data types

The Qt SOAP stack contains several classes for representing SOAP-typical XML elements. The classes were designed to respect the most common XML Schema constructs and do not consider the SOAP section 5 encoding. Figure 23 shows the class diagram containing the data type classes.

The classes in the diagram above serve the following purpose:

QtSoapQName
This class represents a qualified name as defined in the XML Schema standard. That is, it contains a local name and a namespace URI; however, it does not contain a prefix, because the prefix is automatically given to XML elements when using the QXmlStreamWriter class.

QtSoapType
This is an abstract class generalizing the different SOAP data types. Basically speaking, it contains XML information that is common for all elements,

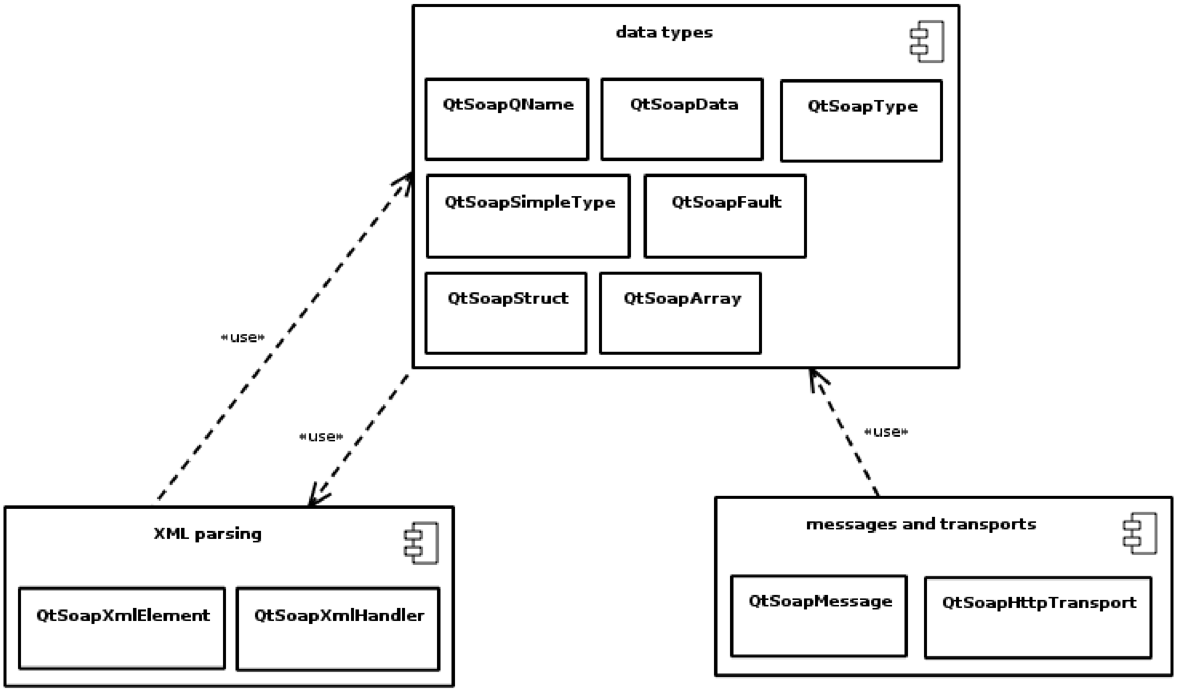


Figure 22: QtSoap component diagram

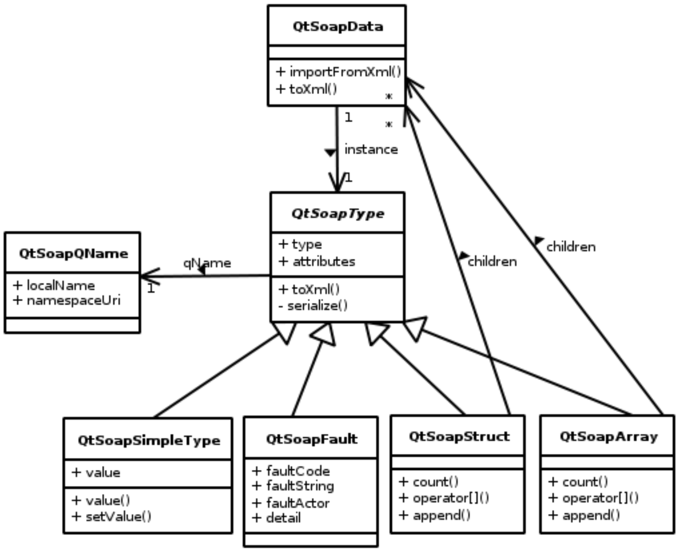


Figure 23: data type class diagram for the QtSOAP framework

e.g. the information from QtSoapQName as well as XML attributes and methods for serialization (the methods “toXml()” and “serialize()” are overriden in the subclasses).

As an API, it contains the union of the functionality of its subclasses (those functions are not shown in the class diagram above). Those functions are implemented as dummy functions as in Listing 25; they are overridden in the subclasses if the classes contain that functionality.

Listing 25: dummy method of abstract class QtSoapType

```
void QtSoapType::append(const QtSoapData &node) {
        qWarning("call␣to␣'append'␣not␣supported␣on␣that␣
            soap␣type");
}
```

QtSoapData

This is a wrapper class for instances of class QtSoapType. It offers more or less the same API as class QtSoapType, but allocates instances on the heap (with the “new” operator). This is necessary to maintain polymorphism; as an example, consider a function that takes as argument any SOAP data type. If its signature would take a QtSoapType as argument (i.e. if it looked like “void function(const QtSoapType &type)”), then data type specific information would be truncated off the argument “type” (e.g. if type was a QtSoapFault, the faultCode would not be available). Since Qt classes as of Qt version 4 do not contain function arguments that are allocated on the heap, the only way to support polymorphism was to allocate the QtSoapType instances internally on the heap in the wrapper class QtSoapData.

QtSoapData was designed to mimic the API of class QtSoapType, because it simply passes calls to its instance, as Listing 26 shows.

Listing 26: QtSoapData offers the same API as QtSoapType

```
void QtSoapData::append(const QtSoapData &item) {
      d->instance->append(item);
}
```

Besides the QtSoapType instance, the class contains a static method to create a QtSoapData instance from a QByteArray; this function is used when receiving a SOAP message over the network.

QtSoapSimpleType

This class represents a specific SOAP data type, namely an XML element that only contains text and no other XML elements as children. Apart from the methods and attributes inherited from QtSoapType, it contains only a “value” attribute (plus its getter- and setter methods) for storing the value of the XML element. Listing 27 shows the XML representation of a QtSoapSimpleType.

Listing 27: A QtSoapSimpleType

```
<myType>myValue</myType>
```

QtSoapFault

This class represents a SOAP fault according to the SOAP standard ([27]). A SOAP fault is a “Fault” element resident in the same namespace URI as the envelope; this element contains the mandatory children “faultcode” and “faultstring” and an optional element “detail”, as in Listing 28.

Listing 28: A QtSoapFault

```
<SOAP–ENV:Fault xmlns:SOAP–ENV='http://schemas.xmlsoap.
    org/soap/envelope/'>
    <faultcode>app.error</faultcode>
    <faultstring>Application Error</faultstring>
    <detail>details go here</detail>
</SOAP–ENV:Fault>
```

QtSoapStruct

This class represents an XML element that contains other elements, as shown in Listing 29. It contains functions (that is, overrides the dummy functions from QtSoapType) for inserting and retrieving children elements; as the children again can be of any SOAP data type, a QtSoapStruct instance contains a list of QtSoapData elements (that is, a member variable “QList<QtSoapData> children”). A SOAP envelope is always of type QtSoapStruct, as it contains at least always a “body” element.

Listing 29: A QtSoapStruct

```
<myType>
    <myChild1>myValue1</myChild1>
    <myChild2>myValue2</myChild2>
    <myChild3>
        <myOtherType1>myOtherValue1</myOtherType1>
        <myOtherType2>myOthervalue2</myOtherType2>
    </myChild3>
</myType>
```

QtSoapArray

This class is a speciality of a QtSoapStruct: A QtSoapArray contains children that are all of the same type, that is, that have the same qualified name, as in Listing 30. As the QtSoapStruct class, it offers accessors for retrieving and manipulating the children elements.

Listing 30: A QtSoapArray

```
<myType>
    <myChild1>myValue1</myChild1>
    <myChild1>myValue2</myChild1>
    <myChild1>myValue3</myChild1>
</myType>
```

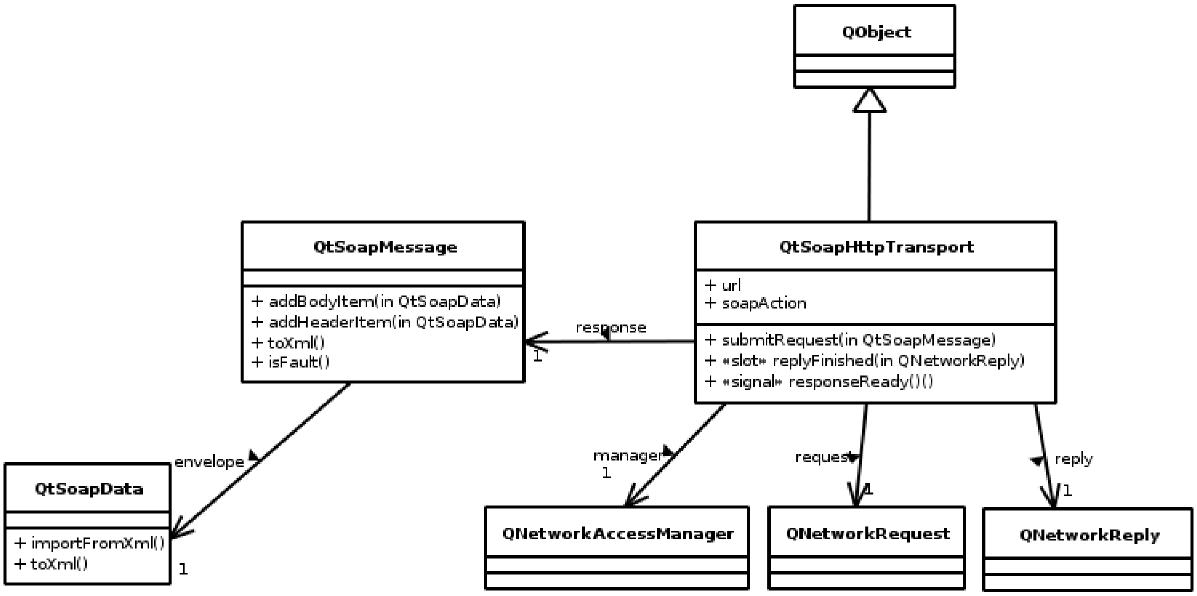


Figure 24: messages and transports class diagram for the QtSOAP framework

Most code in those data type classes are easy getter and setter methods. Exceptions of that are the XML serialization and deserialization, which are described below in section 5.1.3; actually that feature should described here, as the functionality is part of the data type classes, but conceptually they are part of the XML parsing chapter.

5.1.2 SOAP messages and transports

For sending and receiving messages, the QtSoap framework contains the classes depicted in Figure 24.

Those classes contain the following functionality:

QtSoapMessage

This class represents a SOAP 1.1 message; that is, it consists of a SOAP envelope element that contains a body element (which is mandatory according to the SOAP standard) and methods to add QtSoapData elements to the body and the header. To be more precise, QtSoapMessage contains a private variable "envelope" of type QtSoapData which upon class instance construction is initialized with an envelope element; this element contains a child element of name "body". Basically, the QtSoapMessage class is just a wrapper around the QtSoapData class containing a specially designed element (the envelope); the functionality of the class is limited to only modify the "header" and "body" element of the envelope.

Listing 31 shows two public methods of class QtSoapMessage: The method "addBodyItem" takes as argument a QtSoapData element and adds that to the body via QtSoapData's method "append". The "body" method returns a QtSoapData element by accessing the private variable "_envelope" and getting the body via QtSoapData's "operator [](const QtSoapQName &key)" (that is, it selects elements from a QtSoapData instance by the qualified name of the element).

Listing 31: methods of QtSoapMessage

```
void QtSoapMessage::addBodyItem(const QtSoapData &newItem
    ) {
     body().append(newItem);
}
QtSoapData &QtSoapMessage::body() {
          return _envelope[QtSoapQName("Body",
              SOAPv11_ENVELOPE)];
}
```

QtSoapHttpTransport

This class provides functionality to send and retrieve SOAP messages over HTTP (which is the only transport protocol supported in the QtSoap framework so far). It encapsulates the network interaction via the Network Access API (i.e. it contains instances of the classes QNetworkAccessManager, QNetworkRequest and QNetworkReply) and provides higher-level signals and slots: When sending a SOAP message, it takes as argument a QtSoapMessage and sends it via a QNetworkRequest as shown in Listing 32.

Listing 32: Sending SOAP requests via QtSoapHttpTransport

```
void QtSoapHttpTransport::submitRequest(const
    QtSoapMessage &request) {
     QString b = request.toXml(false);
     QByteArray a = b.toUtf8().constData();
     d->reply = d->manager.post(d->request, a);
}
```

When receiving a response from the NetworkAccessManager, the class parses the response, then creates a QtSoapMessage, stores it at the "response" member variable and then emits a signal. In order to first parse the message and then emit its own signal, QtSoapHttpTransport connects the QNetworkAccessManager's signal "finished" with its own slot "replyFinished", as shown in Listing 33.

Listing 33: QtSoapHttpTransport constructor

```
QtSoapHttpTransport::QtSoapHttpTransport(QObject *parent)
      : QObject(parent) {
     d = new QtSoapHttpTransportPrivate;
     d->manager.setParent(this);
```

```
    d->request.setHeader(QNetworkRequest::
        ContentTypeHeader, "text/xml;charset=utf-8");
    connect(&d->manager, SIGNAL(finished(QNetworkReply *))
        , this, SLOT(replyFinished(QNetworkReply *)));
    setSoapAction("");
}
```

When the “replyFinished” slot has been called, it parses the response and emits a signal, as shown in Listing 34.

Listing 34: parsing the SOAP response in class QtSoapHttpTransport

```
void QtSoapHttpTransport::replyFinished(QNetworkReply *)
    {

    QByteArray b = d->reply->peek(d->reply->
        bytesAvailable());
    QtSoapData response = QtSoapData::importFromXml(b);
    if(response.type() != QtSoapType::Struct) {
        qWarning("malformed␣envelope␣detected");
    }
    d->response.setEnvelope(response);
    emit responseReady();
}
```

An example of how to use the QtSoapHttpTransport class is given below in chapter 5.1.4.

5.1.3 XML parsing

The XML parsing classes are responsible for creating QtSoap data types from an HTTP message body. Upon inspection of the XML elements, the XML parser determines which of the four data types to create (fault, simple type, struct or array). Figure 25 shows the diagram of classes involved in the XML parsing process.

The classes contain the following functionality:

QtSoapXmlElement

This class can be seen as a “light” version of the QtSoapData class; it only contains the qualified name, attributes and a pointer to its parent (if it has one) and its children (if it has some). The class serves to store XML elements while parsing a document; a QtSoapXmlElement is constructed when it is not yet clear what type of QtSoapType an element will become (this is explained in detail below).

QtSoapXmlHandler

This is the main class involved in XML parsing and also the entry point. When a byte stream is to be parsed by the QtSoapXmlHandler, QtSoapData's

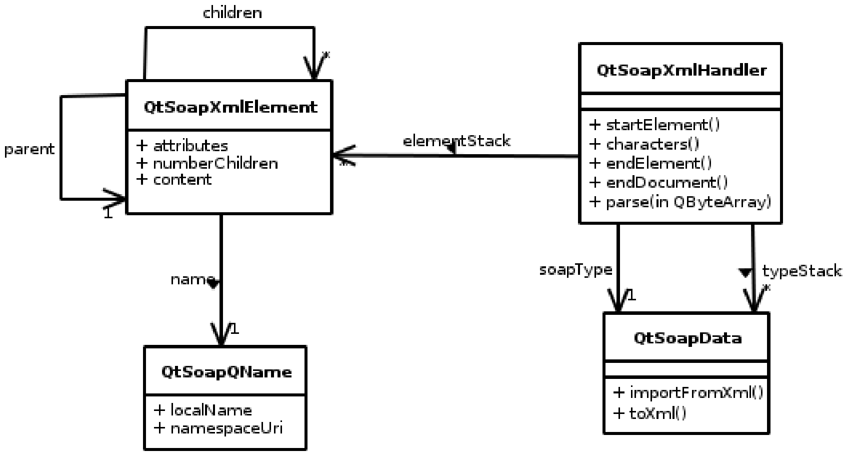


Figure 25: XML parsing class diagram for the QtSoap framework

static function “importFromXml” needs to be called, which calls QtSoapXmlHandler’s “parse” function. When that function is called, the handler creates a QXmlStreamReader object (described in chapter 3), then adds the supplied data to the parser and parses it, as Listing 35 shows.

Listing 35: QtSoap XML parsing

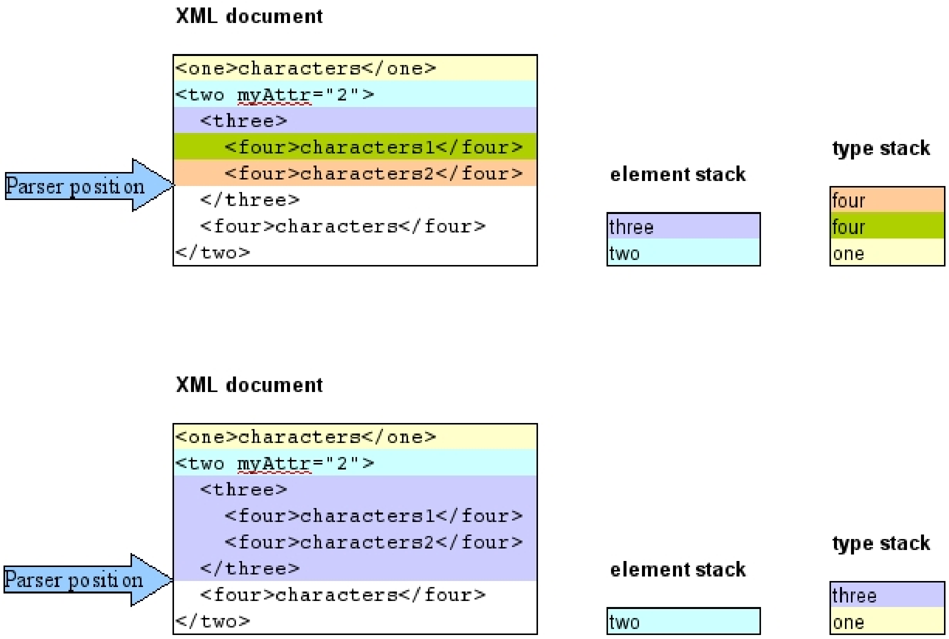
```
QtSoapData &QtSoapXmlHandler::parse(const QByteArray &
    buffer) {
     QTextStream out(stdout);
     QXmlStreamReader xml;
     xml.addData(buffer);
     while (! xml.atEnd()) {
          xml.readNext();
          switch(xml.tokenType()) {
               case QXmlStreamReader::StartElement:
     startElement(xml.namespaceUri().toString(), xml.name
         ().toString(), xml.qualifiedName().toString(), xml
         .attributes());
                    break;
               case QXmlStreamReader::Characters:
                    characters(xml.text().toString());
                    break;
            case QXmlStreamReader::EndElement:
                 endElement(xml.namespaceUri().toString(),
                     xml.name().toString(), xml.qualifiedName
                     ().toString());
                    break;
            case QXmlStreamReader::EndDocument:
                    endDocument();
                    break;
            default:
                    // do nothing
                    break;
            }
        }
        if (xml.hasError()) {
             qWarning("an␣error␣occurred");
            out << xml.errorString() << endl;
        }
          return _soapType;
}
```

As the listing above shows, the parser always processes the whole input at once; there is no “on demand” or delayed parsing. Since this function is also called when a network response is received, the current QtSoap implementation does always parse the whole SOAP response immediately and does not provide a delayed parsing mechanism.

While parsing an XML document, the parser keeps a stack of QtSoapXmlElement objects and another stack of QtSoapData objects. The former keeps elements only from the time their startElement was received by the parser until its corresponding end element was received. When a start element is received, the parser creates a QtSoapXmlElement and sets the correct qualified name and attributes; moreover, a pointer to the parent is set as well as the parent’s pointer to its children. When an end element is received, the parser pops the QtSoapXmlElement off the stack and constructs a corresponding QtSoapData element and pushes it on the QtSoapData stack. The reason for having a QtSoapXmlElement class is that when receiving a startElement event, the parser cannot yet decide which data type is the right one for it; when its corresponding end element is received, the parser can decide whether it should construct a QtSoapStruct, an QtSoapSimpleType etc. from the QtSoapXmlElement on the stack.

As an example, consider Figure 26. In the upper scenario, the parser has already completed parsing of element “one” and both elements with name “four”, thus all three of them are on the type stack. As it has begun, but not ended parsing the elements “two” and “three”, those elements are both on the element stack. In the next step, the parser reads the closing element “</three>”. Then, it pops one item off the element stack and constructs a QtSoapData object from it. Since the “numberChildren” attribute of the “three” element is set to 2, two more elements are popped off the type stack and added as children to the newly constructed QtSoapData object. As the two children are of the same qualified name, the parser knows it must construct an object of type “QtSoapArray”. After that object was constructed, it is pushed onto the type stack, as the lower scenario of Figure 26 shows.

The type stack keeps those elements that have already been parsed; the



```
<one>characters</one>
<two myAttr="2">
  <three>
    <four>characters1</four>
    <four>characters2</four>
  </three>
  <four>characters</four>
</two>
```

Figure 26: QtSoap XML parser states

element stack keeps those elements that have started but not finished parsing. Since an XML element is always in exactly one of the states “already parsed”, “being parsed” and “not yet parsed”, both stacks together keep at most as many elements as there are XML elements in the document.

There are two other parsing events: “characters”, which is called when characters are detected outside angle brackets. In that case, the “content” field of QtSoapXmlElement is set, and, after parsing that element, a QtSoapSimpleType is constructed. The other event is “endDocument”, which is activated when the XML document has been parsed completely. In that function, there must exactly be one element on the type stack (the SOAP envelope) which is then popped and set as the private member variable “_soapType”, as described in Listing 36. That variable is returned as a result by the “parse” function (as already shown in Listing 35).

Listing 36: QtSoapXmlHandler’s endDocument function

```
bool QtSoapXmlHandler::endDocument() {
    if(_typeStack.empty()) {
        return false;
    }
    else {
        _soapType = _typeStack.pop();
        return _typeStack.empty();
    }
}
```

The reverse functionality, XML serialization, usually takes place when sending a message: as Figure 27 shows, when a message is sent, that call is passed on to the SOAP envelope. That "serialize" function takes as argument a QXmlStreamWriter instance, writes its qualified name and attributes, then calls its children's "serialize" function (for structs and arrays) or writes its content (for simple types), and then writes its end attribute, as Listing 37 shows.

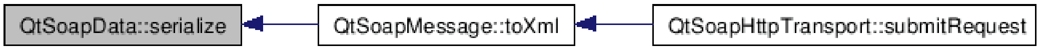


Figure 27: XML serialization caller graph

Listing 37: QtSoapStruct serialization

```
void QtSoapStruct::serialize(QXmlStreamWriter &writer)
    const {
     writer.writeStartElement(name().namespaceUri(), name
         ().localName());
     writer.writeAttributes(attributes());
     for(int a = 0; a < count(); a++) {
          d->children.at(a).serialize(writer);
     }
     writer.writeEndElement();
}
```

When the serialization has finished, the string returned by "QtSoapMessage::toXml" is sent as HTTP body by the QtSoapHttpTransport class.

5.1.4 Sample usage of the QtSoap API

Listing 38 shows a class that uses the QtSoap API to build up a query by hand. It contains a QtSoapHttpTransport as member variable (another option would have been to derive from the class), the "responseReady" signal of the transport class is connected to the "getResponse" slot of the "SoapTest" class. In its constructor, the class builds up a SOAP request via nesting different data types, then adding it to a QtSoapMessage, which is then sent via the QtSoapHttpTransport. When parsing the response, the "operator [](const QString &key)" offers a convenient way to parse the message (it can also be concatenated when parsing deeply nested messages, like e.g. "body[firstLayer][secondLayer][thirdLayer]").

Listing 38: Sample program using the QtSoap API

```
———————————————— soaptest.h ————————————————
#ifndef SOAPTEST_H
```

```
#define SOAPTEST_H
#include "../../src/qtsoap.h"

class SoapTest : public QObject {
    Q_OBJECT
public:
    SoapTest(QObject *parent = 0);
private slots:
    void getResponse();
private:
    QtSoapHttpTransport http;
};
#endif
------------------------------ soaptest.cpp ------------------------------
#include <QtGui/QApplication>
#include <QTextStream>
#include <QNetworkReply>
#include "soaptest.h"

SoapTest::SoapTest(QObject *parent) : QObject(parent),
   http(this) {
    http.setUrl("http://localhost/Demo");
    http.setSoapAction("mySOAPActionHeader");
    connect(&http, SIGNAL(responseReady()), this, SLOT(
        getResponse()));
    QtSoapMessage message;
    QString uri("http://localhost/Demo");
    QtSoapStruct echo("echo", uri);
    QtSoapSimpleType myArgument("myArgument", uri, "
        myValue");
    echo.append(myArgument);
    message.addBodyItem(echo);
    QTextStream out(stdout);
    out << "request:" << endl << message.toXml() << endl;
    http.submitRequest(message);
}

void Msn::getResponse() {
    QTextStream out(stdout);
    QtSoapMessage response = http.soapResponse();
    QtSoapData &body = response.body();
    if(response.isFault()) {
        out << "string:␣" << response.faultString() <<
            endl;
    }
    else {
        out << "selected␣element:␣" << body["echoResponse
            "].value().toString() << endl;
    }
    qApp->quit();
}
```

5.2 WSDL handling

As WSDL parsing is the main use case for dealing with SOAP, the QtSoap framework contains a WSDL parser. That parser is a command-line tool which takes as argument a WSDL file and produces a C++ file using Qt data types. The command-line interface is basically just a wrapper around two XSLT scripts that handle the actual code generation. The first XSLT script, “wsdl2qt.xsl”, parses the WSDL specific part of a WSDL file, while the second script, “xsd-Schema.xsl” parses the XML Schema part of a WSDL file. Thus, the functionality of parsing WSDL and XML Schema are separated, and XML Schema files can also be parsed independently of WSDL files.

The reason for choosing XSLT instead of parsing a WSDL file in C++ are the following:

ease of use. XSLT was destined to transform XML to other formats (here: into code), and is thus a good choice for XML parsing. Writing a WSDL parser in C++ would have required reading in the file, parsing each token and then write to an output file.

performance. XSLT shows a much better performance when parsing a WSDL file. When testing WSDL code generators that were based on “traditional” programming languages like Java or C++, the code generation process took a lot of time and sometimes showed memory problems (e.g. the java generator running out of heap space); this is a problem especially because WSDL files are often auto-generated and several mega bytes of size. Since XSLT relies on a functional programming paradigm, it is easier to parse and output one entity (i.e. XML element) before parsing the next entity, without the need to keep an intermediate state.

As already said, the Qt library does not yet contain an own XSLT parser; thus the parser “xsltproc” from the “libxml” toolkit[41] had to be used. This can be seen as an intermediary solution, since Qt is to support XSLT in the future. Moreover, the currently used “xsltproc” version is using XSLT version 1.0, which has several drawbacks, the most severe being the limitation to output to only one file. It could be argued that this and other drawbacks make XSLT to inflexible for WSDL code generation, but upcoming XSLT 2.0 parsers should provide enough flexibility to handle that task.

[41] http://www.xmlsoft.org/

WSDL	code
<port name=”foo1”>	class “foo1Stub”, derived from QObject
<part name=”foo2”>	class “foo2Element”, derived from QtSoapStruct
<operation name=”foo3”>	member method “foo3()”
<message name=”foo4”>	class “foo4Message”, derived from QtSoapMessage
<types>	(handled by XML Schema stylesheet)

Table 2: WSDL to code transformations

5.2.1 WSDL parsing

As already described, a WSDL file consists of WSDL parts and a usually embedded XML Schema part. The WSDL stylesheet parses the WSDL-specific elements, and includes the XML Schema stylesheet to parse the elements specific to XML Schema. Table 2 lists how the different WSDL entities are transformed to Qt/QtSoap entities; Figure 28 shows the corresponding relationship in UML.

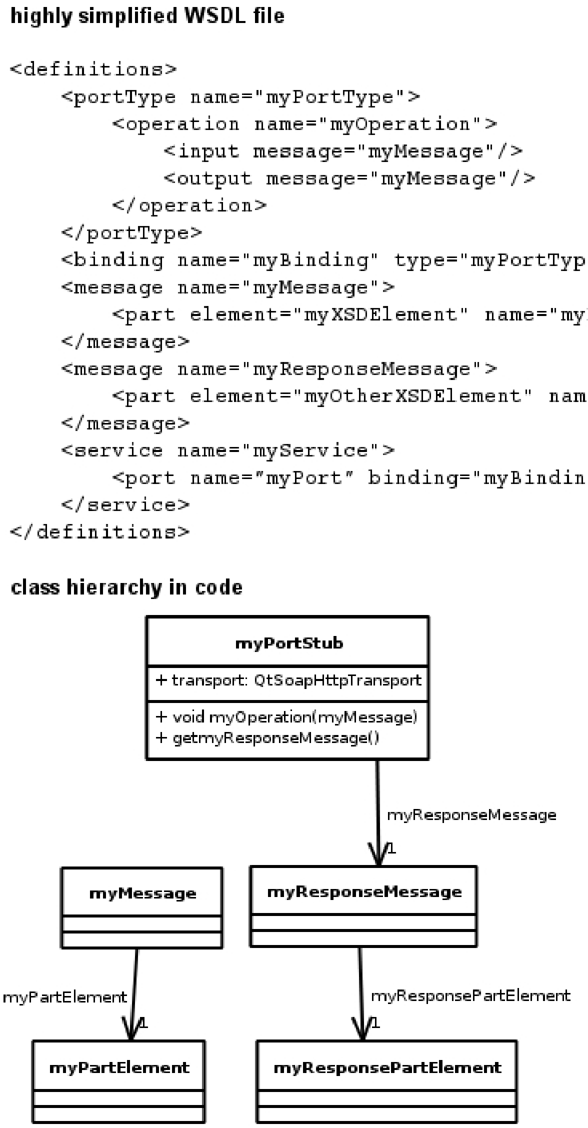
The central entity in the resulting code is the class generated from a “port” element: It is derived from QObject and contains a private QtSoapHttpTransport object, as Listing 39 shows. Furthermore, it contains its own signals and slots, one pair for each “operation” defined in the WSDL (the slot is called by the QtSoapHttpTransport object when it has received its reply, then the message is parsed and the class’ own signal emitted). Besides that, it contains one member method per WSDL “operation”, which takes as argument a QtSoapMessage and sends that message; this is shown in Listing 40.

Listing 39: wsdl2qt.xsl stylesheet generating stub class and constructor

```
    <xsl:template match="wsdl:port">
        <xsl:apply-templates select="wsdl:documentation"/
            >
class <xsl:value-of select="@name"/>Stub : public QObject
    {
    Q_OBJECT
    private:
    QtSoapHttpTransport _transport;
    public:
    <xsl:value-of select="@name"/>Stub(QObject *parent =
        0) : QObject(parent) {
        _transport.setUrl("<xsl:value-of␣select="
            soap:address/@location"/>");
    }
(...)
</xsl:template>
```

Listing 40: wsdl2qt.xsl stylesheet generating methods for WSDL operations

```
<xsl:template match="wsdl:operation" mode="portType">
```



highly simplified WSDL file

```
<definitions>
    <portType name="myPortType">
        <operation name="myOperation">
            <input message="myMessage"/>
            <output message="myMessage"/>
        </operation>
    </portType>
    <binding name="myBinding″ type="myPortType"/>
    <message name="myMessage">
        <part element="myXSDElement" name="myPart"/>
    </message>
    <message name="myResponseMessage">
        <part element="myOtherXSDElement" name="myResponsePart"/>
    </message>
    <service name="myService">
        <port name="myPort″ binding="myBinding"/>
    </service>
</definitions>
```

class hierarchy in code

Figure 28: WSDL and code elements relationship

```
(...)
    // member variable
    <xsl:value-of select="$output-name"/>Message _<
        xsl:value-of select="$output-name"/>;

public:
    // member method
    void <xsl:value-of select="@name"/>(<xsl:value-of
        select="$input-name"/>Message &amp;param) {
         set_<xsl:value-of select="@name"/>_soapAction();
         _transport.submitRequest(param.serialize());
    connect(&amp;_transport, SIGNAL(responseReady()),
        this, SLOT(<xsl:value-of select="@name"/>Response
        ()));
    }
(...)
</xsl:template>
```

Those operations take as argument a type that was generated from a WSDL "message" element. That type is derived from QtSoapMessage and contains several member variables; for each WSDL "part" element inside a "message" element the class contains one member variable; WSDL "part" elements are transformed to own classes, which contain variables that were generated from XML Schema types.

The XSL stylesheet shown in excerpts above generates valid C++ code using QtSoap data types; however, it does not (yet) respect namespaces in element names. For instance, a WSDL file could contain two WSDL "message" elements with the same local name but a different namespace URI, which would cause the code generator to create two classes of the same name. This is currently the biggest drawback of the code generator.

5.2.2 XML Schema parsing

The XML Schema stylesheet does not support the whole XML Schema standard ([4]), but only the most frequently used constructs, which are the following: element, simpleType, complexType, restriction, extension, enumeration, sequence, simpleContent, complexContent. Some of the listed types are only supported partially. A mid-term goal of a good WSDL parser would not be to support the whole XML Schema standard, but to support the most frequently used constructs in a robust way.

When parsing the XML Schema part of a WSDL file, it basically constructs two types of classes: QtSoapSimpleType objects from "simpleType" elements and QtSoapStruct objects from "complexType" elements. Figure 29 lists some Schema constructs and its resulting code architecture.

Each generated class contains member variables according to the XML Schema elements, the usual getter- and setter-methods and a serialize and deserialize

XML Schema file

```
<schema>
   <xsd:simpleType name="mySimple">
     <xsd:restriction base="xsd:string">
       <xsd:enumeration value="enum1"/>
       <xsd:enumeration value="enum2"/>
       <xsd:enumeration value="enum3"/>
     </xsd:restriction>
   </xsd:simpleType>
   <xsd:complexType name="myComplex">
     <xsd:sequence>
       <xsd:element maxOccurs="1" minOccurs="1" name="myMultipleElement"
type="xsd:int"/>
       <xsd:element maxOccurs="1" minOccurs="1" name="myMultipleElement"
type="xsd:double"/>
       <xsd:element maxOccurs="1" minOccurs="1" name="myMultipleElement"
type="xsd:string"/>
     </xsd:sequence>
   </xsd:complexType>
   <xsd:complexType name="myComplex2">
     <xsd:sequence>
       <xsd:element maxOccurs="unbounded" minOccurs="0" name="myMultipleElement"
type="xsd:string"/>
     </xsd:sequence>
   </xsd:complexType>
</schema>
```

class hierarchy in code

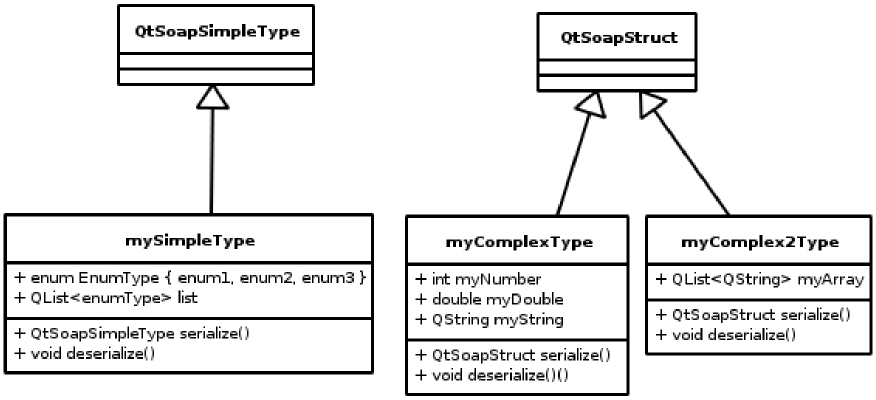


Figure 29: XML Schema and code relationship

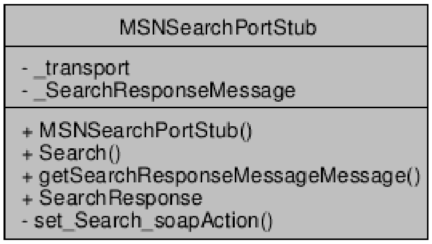


Figure 30: the auto-generated class for using the MSN API

function. When building up a SOAP request, the setter methods of the generated types can be used to build up the request; upon sending, the serialize method is called which returns the correctly formatted QtSoap data type. When receiving a SOAP response, the deserialize method is called to extract information from the XML and set the member variables, which can then be accessed via getter methods.

As already said, by far not the whole XML Schema standard is supported by the XSLT stylesheet; moreover, as with WSDL types, namespaces are not respected. But since the most common constructs are supported, WSDL files that contain easy XML Schema parts are parsed correctly.

5.3 Case study: The MSN Search API

This chapter shall prove the operability of both the QtSoap classes as well as the WSDL code generator. As an example, a search with the MSN Search API[42] is performed, including code generation from the WSDL file[43]. That API has been chosen because the WSDL is relatively easy and small, but nevertheless is a real-world example.

Invoking the code generator with the MSN WSDL produces a header file that contains 29 classes; 5 of them deal with WSDL messages, elements and bindings. The other 24 classes represent XML Schema types. The class responsible for sending and receiving messages is shown in Figure 30. The WSDL file contains only the operation "Search", and thus there is only one method in the MSNSearchPortStub class that takes a QtSoapMessage as argument, namely the method "Search". Besides that, there is one method for getting the response method ("getSearchResponseMessageMessage") and one slot which is called by the QtSoapHttpTransport member "_transport". Listing 41 shows how to create the necessary types to invoke the "Search" method.

Listing 41: invoking the MSN Search with the QtSoap API

```
    MSNSearchPortStub stub;
    SearchMessageMessage message;
```

[42] http://msdn.microsoft.com/en-us/library/bb251794.aspx

[43] http://soap.search.msn.com/webservices.asmx?wsdl

```
    SearchElement searchElement;
    SearchRequestType request;
    request.set_AppID("8
       E56C2042A3D27AC7439A863C38A3D51C396CEC7");
    request.set_Query("Trolltech");
    request.set_CultureInfo("en–US");
    LocationType location;
    location.set_Radius(5.0);
    request.set_Location(location);
    ArrayOfSourceRequestRequestsType requests;
    SourceRequestType type1;
    SourceTypeType source1;
    source1.appendEnum(SourceTypeType::_Web);
    type1.set_Source(source1);
    type1.set_Offset(0);
    type1.set_Count(20);
    SourceRequestType type2;
    SourceTypeType source2;
    source2.appendEnum(SourceTypeType::_News);
    type2.set_Source(source2);
    type2.set_Offset(0);
    type2.set_Count(20);
    requests.append_SourceRequest(type1);
    requests.append_SourceRequest(type2);
    request.set_Requests(requests);
    searchElement.set_Request(request);
    message.set_SearchElement(searchElement);
    stub.Search(message);
```

That code creates a “SearchMessageMessage” object (which is derived from QtSoapMessage), then builds up the necessary request and finally sends out the message. Building up the request requires the confusing task of buliding up a lot of elements and then nesting them repeatedly. However, this can not be avoided because the nested structure of the XML Schema elements and WSDL parts just resembles the nested structure of the required XML; this complicated structure is common to all SOAP frameworks. Figure 31 shows how the classes required for a Search are nested.

When invoking the “Search” method, the class implementation uses the “_transport” instance to send and receive the SOAP message. When the SOAP response arrives, the “_transport” emits its “responseReady” signal, which causes the MSNSearchPortStub instance to parse the message (that is, deserialize the whole SOAP response). After that, it emits its own signal “SearchResponseReady” (see Listing 42), which tells the program it can now access the data via the class’ getter methods, as shown in Listing 43.

Listing 42: receiving the MSN Search response message

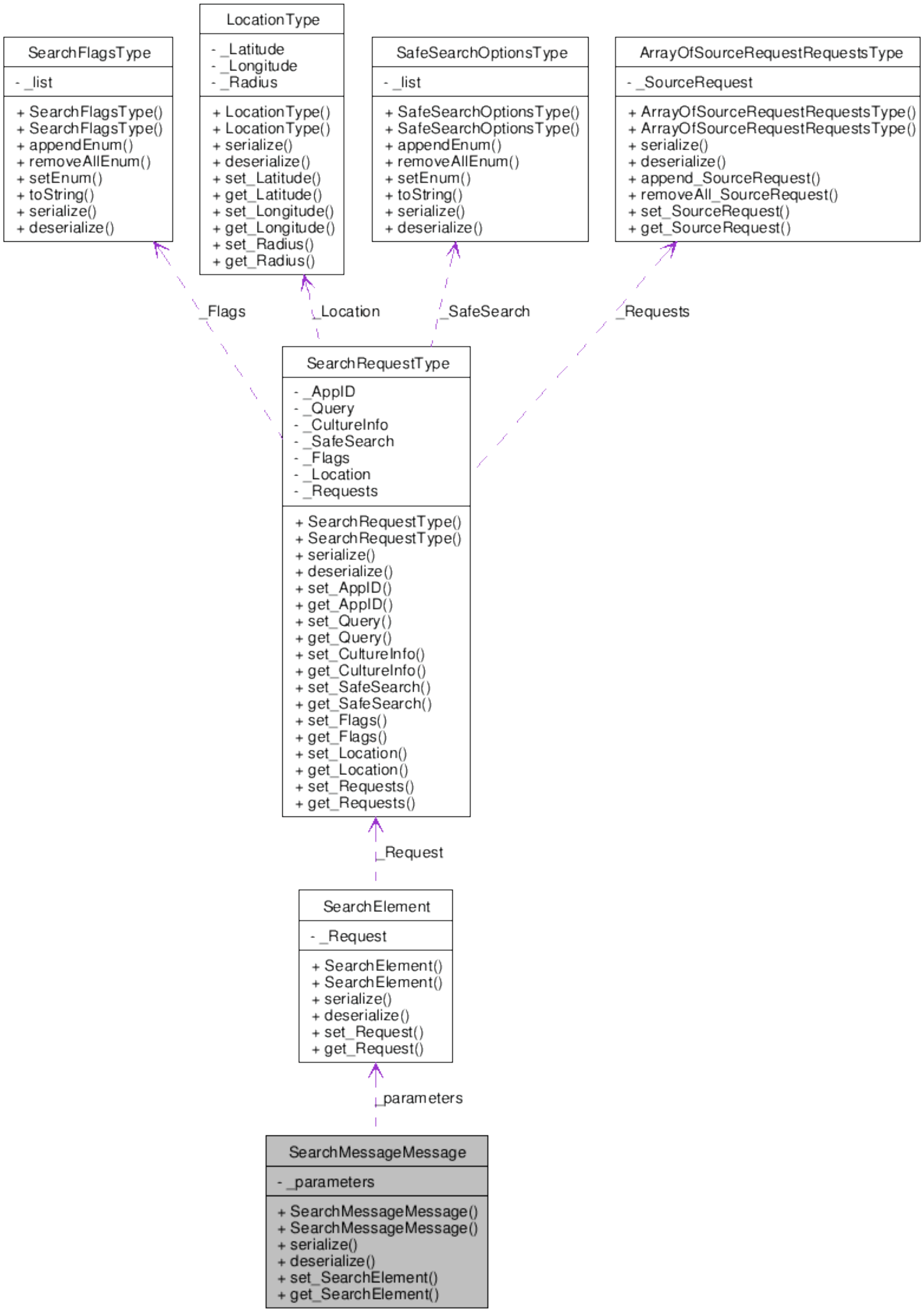


Figure 31: auto-generated class hierarchy for the MSN search

```
public Q_SLOTS:
void SearchResponse() {
     _SearchResponseMessage = SearchResponseMessageMessage
         (_transport.soapResponse());
     _SearchResponseMessage.deserialize();
     emit SearchResponseReady();
}
```

Listing 43: parsing the MSN Search response message

```
SearchResponseMessageMessage message = stub.
    getSearchResponseMessageMessage();
const SearchResponseElement &searchResponseElement =
    message.get_SearchResponseElement();
const SearchResponseType &response =
    searchResponseElement.get_Response();
const ArrayOfSourceResponseResponsesType &responses =
    response.get_Responses();
const QList<SourceResponseType> &sourceResponse =
    responses.get_SourceResponse();
for(int a = 0; a < sourceResponse.count(); a++) {
     out << "------------------------------------------" <<
         endl;
   out << "results␣for:␣" << sourceResponse[a].get_Source
       ().toString() << "␣(" << sourceResponse[a].get_Total
       () << ")" << endl;
     out << "------------------------------------------" <<
         endl << endl;
     const ArrayOfResultResultsType &results =
         sourceResponse[a].get_Results();
     const QList<ResultType> &result = results.get_Result
         ();
     for(int b = 0; b < result.count(); b++) {
          out << "title:␣" << result[b].get_Title() << endl
              ;
          out << "------------------------------------------" <<
               endl;
          out << "description:␣" << result[b].
              get_Description() << endl << endl;
          out << "url:␣" << result[b].get_Url() << endl <<
              endl << endl;
     }
}
```

As with the creation of the SOAP message, the parsing of the response requires accessing the nested class structure in the same way. Again, this cannot be avoided; however, the code listings for creating the SOAP request and parsing the response show that no XML whatsoever is involved, so the XML layer of the SOAP call is encapsulated completely inside the generated code (but it can be accessed e.g. for debugging purposes if needed).

6 Conclusion and outlook

This chapter shall evaluate whether the solutions proposed in the main chapters of this document answers the questions and fulfill the requirements of chapter 1.1. Moreover, the results of the SOAP toolkit evaluation from section 4.4 are compared to the QtSoap features.

The general questions about SOAP from chapter 1.1 were:

- What is the main field of use of the SOAP protocol? How does it overlap with similar techniques?

The main field of use of the SOAP protocol is the field of sophisticated distributed programming systems, which require several quality of service aspects, for instance encryption, reliable messaging or complex message flows. Most of these systems are not realized in the field of publicly available Web APIs, since a SOAP system poses a higher entry barrier onto the programmer than other techniques. Moreover, they require more work to enable the features of a SOAP system. As an alternative to SOAP, REST is a popular choice, especially in the field of Web APIs. Compared to SOAP, REST is easy to adapt, but not as feature-rich as SOAP. A more feature-rich solution than SOAP is for instance Java RMI, which ties the involved systems closer together and is directed towards distributed objects. AJAX is a technique to issue remote procedure calls for Web pages.

All in all, all the distributed programming techniques have their own area of application, where the strengths of SOAP are its feature-richness by simultaneously providing a relatively loose coupling between the distributed systems.

- What API do other SOAP toolkits offer? What are their strengths and weaknesses?

The most important API of a SOAP toolkit is its code generator. Thus, the SOAP class system should be designed to be easily adaptable for the code generator. As a plus, it should also be possible to easily build up a SOAP call by hand with the class system. A more detailed evaluation of the existing SOAP toolkits is described in chapter 4.4.

QtSoap feature evaluation

To evaluate the proposed QtSoap toolkit more closely, the requirements from the SOAP toolkit evaluation need to be compared with the features of the QtSoap toolkit.

1. WSDL code generation. the QtSoap toolkit offers WSDL code generation; however, this is only a prototype and needs to be enhanced to be fully functional.

2. API to build up SOAP queries by hand. QtSoap offers that feature, as was described in chapter 5.1.4.

3. asynchronous calls. As the underlying network infrastructure only supports asynchronous calls, this features was included in QtSoap from the beginning.

4. pluggable data binding. This feature is not supported, since there is always only one candidate of a Qt type to be mapped to a XML Schema type. However, in future versions of QtSoap it should be considered to support both user-defined type mappings as well as different class architectures.

5. separate XML Schema parsing entity from WSDL parsing entity. This feature is included by simply separating the XML Schema stylesheet from the WSDL stylesheet. Thus, by implementing a command-line switch an XML Schema file can be parsed against the XML Schema stylesheet instead of the WSDL stylesheet.

6. possibility to add WS-* support and attachments. This has not been considered thoroughly; enabling these features would require to change the API of existing classes as well as adding new classes. But since the current message and transport classes are not (yet) structured in a complex way, this should not be a big problem.

7. XML pull parsing. This is not supported yet, but could (and should in future versions) be enabled without changing the API of the existing classes.

Outlook

There are several fields were the QtSoap stack could be enhanced. The most important area is clearly the code generator: Here, the WSDL support as well as the XML Schema support should be improved to support a broader range of constructs. After that, attachment support should be added, followed by the most important WS-* standards, which are WS-Addressing and WS-Security. Apart from that, there are a lot of other features left to be implemented (e.g. SOAP version 1.2, WSDL version 2.0 and several other WS-* standards). In general, a SOAP stack is never really complete, so the next features to be implemented should always be considered carefully.

References

[1] Jasmin Blanchette and Mark Summerfield. C++ gui programming with qt 4. *Prentice Hall*, 2006.

[2] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web services description language (wsdl) version 2.0 part 1: Core language. *http://www.w3.org/TR/wsdl20/*, 2007.

[3] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (wsdl) 1.1. *http://www.w3.org/TR/wsdl*, 2001.

[4] David C. Fallside and Priscilla Walmsley. Xml schema part 0: Primer second edition. *http://www.w3.org/TR/xmlschema-0/*, 2004.

[5] Christopher Ferris, Anish Karmarkar, Prasad Yendluri, Keith Ballinger, David Ehnebuske, Martin Gudgin, Canyang Kevin Liu, and Mark Nottingham. Basic profile version 1.2. *http://www.ws-i.org/Profiles/BasicProfile-1.2.html*, 2007.

[6] Roy Thomas Fielding. Architectural styles and the design of network-based software architectures. *http://www.ics.uci.edu/ fielding/pubs/dissertation/top.htm*, 2000.

[7] Torsten Fink. Rest vs. soap. *JavaSpektrum*, 2007.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Elements of reusable object-oriented software. *Addison-Wesley*, 1995.

[9] N.A.B. Gray. Comparison of web services, java-rmi, and corba service implementations. *http://mercury.it.swin.edu.au/ctg/AWSA04/Papers/gray.pdf*, 2004.

[10] Martin Gudgin, Noah Mendesohn, Mark Nottingham, and Herve Ruellan. Soap message transmission optimization mechanism. *http://www.w3.org/TR/soap12-mtom/*, 2005.

[11] Hugo Haas and Allen Brown. Web services glossary. *http://www.w3.org/TR/ws-gloss/*, 2004.

[12] Marc Hadley. What's new in soap 1.2. *http://hadleynet.org/marc/whatsnew.html*, 2002.

[13] The IEEE and The Open Group. fork - create a new process. *The Open Group Base Specifications Issue 6*, 2004.

[14] Deepal Jayasinghe. Quickstart apache axis2. *Packt Publishing*, 2008.

[15] Martin Kutter. State of the soap. *Proceedings of 10th German Perl Workshop*, 2008.

[16] Microsoft. Createprocess function. *http://msdn.microsoft.com/en-us/library/ms682425(VS.85).aspx*, 2008.

[17] Aaron Reed. Asynchronous web services. *.NET Developer's Journal*, 2005.

[18] Leonard Richardson and Sam Ruby. Restful web services. *O'Reilly*, 2007.

[19] Govind Seshadri. Remote method invocation (rmi). *http://java.sun.com/developer/onlineTraining/rmi/RMI.html*, 2000.

[20] Davanum Srinivas, Paul Fremantle, Amila Suriarachchi, and Deepal Jayasinghe. Web services are not slow. *http://wso2.org/node/588/print*, 2007.

[21] Trolltech. Meta-object system. *http://doc.trolltech.com/4.4/metaobjects.html*, 2008.

[22] Trolltech. Platform notes. *http://doc.trolltech.com/4.4/platform-notes-platforms.html*, 2008.

[23] Trolltech. qmake manual. *http://doc.trolltech.com/4.4/qmake-manual.html*, 2008.

[24] Trolltech. Qt's property system. *http://doc.trolltech.com/4.4/properties.html*, 2008.

[25] Trolltech. Signals and slots. *http://doc.trolltech.com/4.4/signalsandslots.html*, 2008.

[26] Robert A. van Engelen and Kyle A. Gallivan. The gsoap toolkit for web services and peer-to-peer computing networks. *proceedings of the 2nd IEEE International Symposium on Cluster Computing and the Grid (CCGrid2002)*, 2002.

[27] W3C. Simple object access protocol (soap) 1.1. *http://www.w3.org/TR/2000/NOTE-SOAP-20000508/*, 2000.

[28] W3C. Soap specifications. *http://www.w3.org/TR/soap/*, 2007.

[29] W3C. Soap version 1.2 part 1: Messaging framework (second edition). *http://www.w3.org/TR/soap12/*, 2007.

[30] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F. Ferguson. Web services platform architecture: Soap, wsdl, ws-policy, ws-addressing, ws-bpel, ws-reliable messaging, and more. *Prentice Hall*, 2005.

[31] Nikolas C. Zakas, Jeremy McPeak, and Joe Fawcett. Professional ajax. *Wiley Publishing*, 2006.